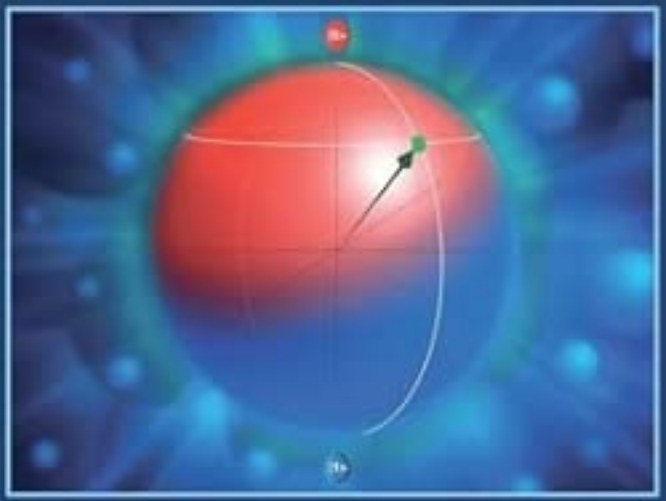
Robert Hundt

Quantum Computing for Programmers



SECOND EDITION

Quantum Computing for Programmers

Second Edition

This introduction to quantum computing from a classical programmer's perspective is meant for students and practitioners alike. About 50 fundamental algorithms are explained with full mathematical derivations and classical code for simulation, using an open-source code base developed from the ground up in Python and C++. New material throughout this fully revised and expanded second edition includes new chapters on Quantum Machine Learning, State Preparation, and Similarity Tests.

After presenting the basics of quantum computing and the software infrastructure used for simulation, a section on modeling classical logic with quantum gates prepares for the quantum supremacy experiment. With this background, the following sections discuss, derive, and implement with working code algorithms exploiting entanglement, blackbox algorithms, algorithms for state preparation and state similarity tests, algorithms based on amplitude amplification, the quantum Fourier transform and phase estimation, several quantum optimization algorithms, quantum walks, and a short section on foundational quantum machine learning algorithms. The list of algorithms includes Shor's algorithm, Grover's algorithm, SAT3, graph coloring, the Solovay–Kitaev algorithm, Möttönen's algorithm, quantum mean, median, and minimum finding, Deutsch's algorithm, Bernstein–Vazirani, quantum teleportation and superdense coding, and the CHSH game. From the field of quantum machine learning, the book discusses Euclidean distance, principal component analysis, and the HHL algorithm. The book also addresses issues around programmer productivity, including quantum noise, error correction, quantum programming languages, compilers, and techniques for transpilation.

Robert Hundt is a distinguished engineer at Google. He has led many compiler and performance projects, including an open-source CUDA compiler and the high-level synthesis toolchain XLS. He is the senior tech lead for Google's low-level machine learning software infrastructure, which includes the OpenXLA compiler for CPU, GPU, and TPU. He has more than 25 scientific publications, holds more than 35 patents, and is a senior member of the Institute of Electrical and Electronics Engineers.

"There is a great deal of interest in quantum computing today. What many would like is a book that explains quantum computing to people who already know how to program conventional computers. This book successfully fills that need."

– David Patterson, 2017 ACM A.M. Turing Award Laureate

"There is a critical need for quantum software engineers in the emerging quantum computing industry. Robert Hundt is a classical software engineer who presents quantum computing as simply as possible to others with a similar background. This book could be the perfect vehicle for many interested in this emerging area."

– Fred Chong, Seymour Goodman Professor, University of Chicago

"Quantum mechanics, the century-old theory underlying modern physics and chemistry, has a reputation for being incomprehensible. Professional physicists have a standard approach to this conundrum: 'Shut up and calculate!' This book provides an alternative much better suited to the programmers of the twenty-first century interested in quantum computing: 'Shut up and program!'"

– Sergio Boixo, Google

"This book strikes just the right balance between theory and practice. Exploring quantum computing from the perspective of a classical programmer, using software and simulators to explain all concepts and algorithms, leads to an intuitive, accessible, yet deep learning experience. I highly recommend this book!"

– Kunle Olukotun, Cadence Design Professor, Stanford University

"This book takes a unique approach of introducing quantum computing with a combination of precise but manageable mathematics, open-source code, and detailed derivations of many core quantum algorithms, which makes it an ideal learning resource for the community of software programmers, including both students and professionals, to explore the fascinating land of quantum computing."

– Jason Cong, Volgenau Chair for Engineering Excellence, UCLA

Quantum Computing for Programmers

Second Edition

ROBERT HUNDT

CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre,
New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment,
a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of
education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781009548533

DOI: 10.1017/9781009548519

When citing this work, please include a reference to the DOI 10.1017/9781009548519

First published 2022

Second edition 2026

A catalogue record for this publication is available from the British Library

A Cataloging-in-Publication data record for this book is available from the Library of Congress

ISBN 978-1-009-54853-3 Hardback

To Mary, Thalia, and Johannes

Contents

Acknowledgments

A book like this would not be possible without the help of many. I am grateful to the numerous people who helped with the first edition of the book. Vincent Russo, Timofey Golubev, Gabriel Hannon, Fedor Kostritsa, and Ton Kalker provided technical advice and reviews. Sergio Boixo, Benjamin Villalonga, and Craig Gidney corrected many of my misunderstandings. Wes Cowley and Sarah Schedler provided line editing, Eleanor Bolton provided outstanding copyediting services, and Sue Klefstad produced the impressive index. My colleagues Dave Bennet, Michael Dorner, Mark Heffernan, Chris Leary, Rob Springer, and Mirko Rossini gave additional feedback. I also thank Aamer Mahmood for his support.

Several people helped with the online source code repository: Kevin Crook from the Universty of California, Berkeley, Moez A. AbdelGawad from Alexandria University, Egypt, Stefanie Scherzinger, Abdolhamid Pourghazi and Stefan Klessinger, from the University of Passau, as well as Michael Broughton, Mikhail Remnev, Colin Zhu, and Andrea Novellini.

Several people helped with the first and this second edition. Dave Bennet and Craig Gidney maintain the elegant and invaluable online simulator Quirk. I am grateful to Tiago Leao for pointing me to Beauregard (2003), which was the key for my implementation of Shor's refactoring. Together with Rui Maia, Tiago also provides the community with a much appreciated reference implementation (Leao, 2021). H. Morrell Jr., H. Y. Wong, and A. Zaman from San Jose State University maintain the step-by-step HHL algorithm walk-through (Morell et al., 2023). Without it, my implementation of the algorithm would not have been possible. Several of my questions were answered on the Quantum Computing Stack Exchange, a very helpful resource and community.

For this edition, I want to thank Adam Zaman, Afham Afham, Jon Pilarte, Pijus Petkevičius, and Shahla Novruzova, for their deep and precise technical reviews. Avione Lee provided the line editing, Fiona Cole did the outstanding copy-editing, and Sue Klefstad did the impressive index again.

Without exception, my contacts at Cambridge University Press were outstanding. First and foremost, I must thank my editor, Lauren Cowles, who did a tremendous job guiding me through the whole process.

Finally, and most importantly, I am incredibly thankful to my family, including my dogs Charlie and Archie, for their love, patience, and support.

Introduction

I think I can safely say that nobody understands quantum mechanics.

Feynman (1965)

I have been impressed by numerous instances of mathematical theories that are really about particular algorithms; these theories are typically formulated in mathematical terms that are much more cumbersome and less natural than the equivalent formulation today's computer scientists would use.

Knuth (1974)

This book is an introduction to quantum computing from the perspective of a classical programmer. All major concepts and algorithms are explained with code, based on the insight that much of the complicated-looking math typically found in quantum computing books may look quite simple in code. For many programmers, reading code is faster than reading complex mathematical derivations. Coding also allows experimentation, which helps build intuition and understanding of the fundamental mechanisms of quantum computing. I believe that this approach will make it efficient and fun to get started.

Contrary to other learning resources, we will not use available software frameworks in this book, such as the well-developed Qiskit toolkit from IBM or Google's Cirq. The goal is to learn about quantum computing without being burdened by the complexities of these frameworks. Instead, we build our own infrastructure from the ground up, based initially on Python's `numpy` library. It turns out that, to learn the fundamentals, only a few hundred lines of code are required. This initial code is slow but easy to debug and experiment with, making it an excellent learning vehicle.

We also improve this infrastructure, accelerate it with C++, and detail an elegant sparse representation. We introduce basic compiler concepts that allow for the transpilation of our circuits to platforms like Qiskit, Cirq, and others. This enables the use of these systems' advanced features, such as scale-out performance and advanced error models.

Typically, an introduction to quantum computing is preceded by a sizable reintroduction of complex linear algebra. We will not follow this pattern here. Many programmers have a solid foundation in linear algebra, but others lack the background or interest in this topic. It is my goal to produce an attractive learning resource for both groups without getting too deep into linear algebra. Hence, I only assume basic

familiarity with complex numbers, vectors, and matrices. Some core concepts are reviewed in Chapter 1. These basics will be sufficient for most of the book.

Nevertheless, this second edition of the book goes deeper into the mathematical foundation and adds a dedicated section on more advanced topics. Although those will only be needed for a few algorithms, the book would not be complete without them. I hope that this format will be helpful to the linear algebra-challenged and not be too shallow for the cognoscenti.

The book is organized into 16 chapters plus an appendix. To get started, it is recommended to read the first three chapters on the fundamentals of quantum computing and the code base that we use throughout the book. Chapters 4 and 5 are the bridge to the remaining chapters on various classes of algorithms, which can be read in almost any order. The book closes with a discussion of quantum programming languages, compilation techniques, and other aspects of productivity in Chapter 16. Some of the later chapters may reuse insights from earlier chapters, but the references and extensive index will allow you to find any missing information.

Chapter 1 The Mathematical Minimum This brief chapter discusses the minimum mathematical background required to fully understand the derivations in this text. Basic familiarity with matrices and vectors is assumed. The chapter reviews key properties of complex numbers, the Dirac notation with inner and outer products, the Kronecker product, unitary and Hermitian matrices, eigenvalues and eigenvectors, the matrix trace, and how to construct the Hermitian adjoint of matrix-vector expressions.

Chapter 2 Quantum Computing Fundamentals This chapter introduces the fundamental concepts and rules of quantum computing. In parallel, it develops an initial, easy-to-understand Python code base for building and simulating small-scale quantum circuits and algorithms. The chapter details single qubits, superposition, quantum states with many qubits and operators, including a sizable set of important single-qubit gates and controlled gates. The Bloch sphere and the quantum circuit notation are introduced. Entanglement follows, that fascinating "spooky action at a distance," as Einstein called it. The chapter then discusses maximally entangled Bell states, the no-cloning and no-deleting theorems, local and global phases, and uncomputation. The quantum postulates are discussed briefly in preparation for the discussion on measurements.

Chapter 3 Simulation Infrastructure This chapter builds a more complete software framework, including a high-performance simulator. It discusses transpilation, a powerful compiler-based technique that allows seamless porting of circuits to other frameworks. The methodology further enables implementing of key features found in quantum programming languages, such as automatic uncomputation or conditional blocks. The chapter also introduces an elegant sparse representation.

Chapter 4 Quantum Tools and Techniques This chapter details the mathematical tools and techniques required by some of the advanced algorithms. Beginners may choose to skip this section and refer back to it as needed. The chapter discusses the spectral theorem, density matrices, the partial trace, Schmidt decomposition, state purification, and various operator decompositions.

Chapter 5 Beyond Classical This chapter serves as a bridge from the introductory material to the sections on quantum algorithms. We start by implementing a classical circuit using quantum gates and show that quantum computers are at least as capable as classical computers. Then we discuss the term "beyond classical," which is now the preferred term to describe a computation that can be run efficiently on a quantum computer but would be intractable to run on a classical computer. For this, we discuss Google's seminal quantum supremacy paper in detail.

Chapter 6 Algorithms Exploiting Entanglement This chapter presents the first real algorithm – a quantum "Hello World" program, a simple random number generator. The chapter then details quantum teleportation, superdense coding, entanglement swapping, and the CHSH game. This game is a simplified version of the Bell inequalities, which established that classical theories assuming hidden states cannot explain quantum entanglement.

Chapter 7 State Similarity Tests This chapter discusses the terms overlap and similarity between quantum states and introduces the important swap test, as well as the Hadamard test, and the inversion test. The mathematical derivations in this chapter are still very detailed.

Chapter 8 Black-Box Algorithms The algorithms presented in this chapter were the first to establish a query complexity advantage for quantum algorithms. The list includes the Bernstein–Vazirani algorithm, Deutsch's algorithm, and Deutsch–Jozsa algorithm. Quantum oracles and their construction are introduced.

Chapter 9 State Preparation Quantum algorithms operate on inputs encoded as quantum states. Preparing these input states can be quite complicated. The chapter discusses the trivial basis and amplitude encoding schemes, as well as Hamiltonian encoding. It also discusses smaller circuits for two- and three-qubit states. Then, this chapter presents two of the most complex algorithms in this book, the general state preparation algorithms from Möttönen, and the Solovay–Kitaev algorithm for gate approximation. Beginners may decide to skip these two algorithms on a first read.

Chapter 10 Algorithms Using Amplitude Amplification This chapter discusses the fundamental Grover's algorithm, which enables searching over a domain of N elements with $\mathcal{O}\left(\sqrt{N}\right)$ complexity. Several derivative algorithms and applications are being discussed, including amplitude amplification, amplitude estimation, quantum counting, Boolean satisfiability, graph coloring, and quantum mean, median, and minimum finding.

Chapter 11 Algorithms Using Quantum Fourier Transform The quantum Fourier transform is another fundamental quantum algorithm. The chapter begins with a simple phase-kick circuit and expands to quantum phase estimation before detailing the quantum Fourier transform itself. A short section on arithmetic in the quantum domain introduces techniques that are used in a final elaborate section on Shor's famous algorithm for number factorization.

Chapter 12 Quantum Walk Algorithms A quantum walk algorithm is the quantum analog to a classical random walk with potential applications in search problems, graph problems, quantum simulation, and even machine learning. In

this section, we describe the basic principles of this class of algorithms on a simple one-dimensional topology.

Chapter 13 Optimization Algorithms This chapter details several optimization algorithms. The variational quantum eigensolver is presented, which allows finding a minimum eigenvalue for a given Hamiltonian. This chapter also includes extensive notes on performing measurements in arbitrary bases. After a brief introduction to the quantum approximate optimization algorithm, the chapter further discusses the quantum maximum cut algorithm and the quantum subset sum algorithm in great detail.

Chapter 14 Quantum Machine Learning Quantum machine learning is an exciting field that explores the intersection of quantum computing and machine learning. It aims to leverage the principles of quantum computing to enhance machine learning algorithms and potentially revolutionize how we analyze data and solve complex problems. This chapter begins with a simple algorithm for computing the Euclidean distance between vectors. We discuss the quantum principal component analysis and, finally, detail the complex but beautiful HHL algorithm for solving systems of linear equations.

Chapter 15 Quantum Error Correction This chapter discusses quantum noise and techniques for quantum error correction, which is necessary for quantum computing. It discusses bit-flip errors, phase-flip errors, and their combinations. The formalism of quantum operations is introduced, along with the operator-sum representation and the Kraus operators. With this in mind, the chapter discusses the depolarization channel and imprecise gates, as well as (briefly) amplitude and phase damping. For error correction, repetition codes are introduced to motivate Shor's 9-qubit error correction technique.

Chapter 16 Quantum Languages, Compilers, and Tools We have introduced a compact infrastructure for exploration and experimentation, at the level of individual gates. Higher levels of abstraction are needed to scale to larger programs. The chapter discusses several quantum programming languages, including their specific tooling, such as hierarchical program representations or entanglement analysis. General challenges for compilation are discussed as well as compiler optimization techniques.

Appendix The appendix contains a detailed description of the sparse simulation infrastructure.

Notes on the 2nd Edition

This second edition is a substantial rewrite and edit of the first edition. No page has been left untouched.

- The book is now organized into 16 chapters, compared to the 8 chapters before. Much of the material from the first edition had to be compacted to make space for the new material.
- The didactic flow has been substantially improved. Several sections have been rearranged to provide a better learning experience.

- Many graphical elements have been modified to be more clear and visually appealing. Pointers to the code are now clearly marked (with hyperlinks in the online editions).
- More attention is now given to the mathematical foundation. This is based on reader feedback. For some, even the limited math of the first edition was already too much, while for many others the math was too shallow. The additional focus on the math may not help the first group, but will make the second group much happier.

A lot of new content has been added:

- Chapter 2 adds content on the W state, the U_3 gate, and how it can be used to make other gates, a section on the No-Deleting Theorem, and a short exercise section on the practice of tensor expressions.
- Chapter 3 condenses the previous sections on various pieces of software infrastructure and acceleration into a single section.
- Chapter 4 is a new chapter on mathematical tools and techniques. It includes discussions of the spectral theorem, density matrices, the Schmidt decomposition and state purification, maximal entanglement, and several operator decompositions, such as the Pauli, *ZYZ*, and *XYX* decompositions. The previous section on the partial trace has been moved here as well.
- Chapter 6 on algorithms using entanglement adds the entanglement swapping algorithm and a discussion of the CHSH game.
- Chapter 7 is a new chapter on similarity tests and adds the Hadamard test, the inversion test, and a new multi-qubit swap test, to the previous swap test.
- Chapter 9 is a new chapter on state preparation. It discusses the basis, amplitude, and Hamiltonian encoding, adds material on effective initialization of 2-qubit and 3-qubit states, and an elaborate section on Möttönen's algorithm for general state preparation. The material on Solovay–Kitaev's algorithm has been moved here as well.
- Chapter 10 is an extended chapter on algorithms using quantum amplitude amplification. It has previous sections on Grover's algorithm, amplitude amplification, and quantum counting, and it adds new sections on amplitude estimation, Boolean satisfiability, graph coloring, quantum mean, median, and minimum finding.
- Chapter 11 on the quantum Fourier transform and Shor's algorithm is a rewrite and restructuring of prior material with more emphasis on detailed mathematical derivations. The section on arithmetic now adds multiplication in the Fourier domain.
- Chapter 14 on quantum machine learning algorithms is new and includes a discussion of the Euclidean distance, the principal component analysis, and a very detailed discussion of the HHL algorithm for solving systems of linear equations.

A significant number of problems and inaccuracies in the first edition were corrected during the writing of this second edition. It is my sincere hope that I fixed more problems from the first edition than I introduced in this second edition. Naturally, the

inevitable remaining errors are solely my own responsibility, and I apologize for all of them.

Source Code

Much of the content of this book is explained with both math and code. However, to avoid turning this book into a giant code listing, we abbreviate less interesting or repetitive code with constructs such as `[...]`. Scaffolding code, such as Python `import` statements or `#include` directives for C++, as well as many redundant comments, are typically omitted.

To run the code, as a minimal setup, a working Python interpreter is required with the Python packages `absl`, `numpy`, and `scipy`. Without the C++ acceleration described later in the book, some of the algorithms will continue to work but run rather slowly. The complete sources are hosted under a permissive Apache license on GitHub, along with instructions on how to download, build, and run:

`www.github.com/qcc4cp/qcc`

I will maintain the errata on this site as well. Contributions, comments, and suggestions are always welcome. The code typesetting may have introduced errors, but the source of truth is the working code in the online repository. The code may also have evolved beyond what is published here.

1 The Mathematical Minimum

In this first chapter, we briefly discuss the minimum mathematical background required to follow this text. This section is quite compact, as it is mainly meant as a reference. Readers who are familiar with the concepts may skip this section. Readers easily discouraged by even basic math may proceed to the next chapters and refer back to here as needed.

1.1 Complex Numbers

Quantum computing runs on complex numbers (Penrose R., 2021). Let us start by briefly recalling the most important properties of complex numbers. A complex number z is of the form

$$z = x + iy.$$

The x is called the *real* part of z, and y is the *imaginary* part. The imaginary number i is defined as $i = \sqrt{-1}$, the solution to the equation

$$x^2 + 1 = 0.$$

The *conjugate* of a complex number is created by replacing i with $-i$. The conjugate is often denoted by $\bar{z}$ or z^*. For example, for $z = 5 + 2i$, the conjugate is simply $z^* = 5 - 2i$. The conjugate of a product of complex numbers is equal to the product of the conjugates of the complex numbers. This tongue twister translates to the simple rule

$$(ab)^* = a^* b^*.$$

The *norm* of a complex number z, denoted by $|z|$, is calculated by multiplying z with its conjugate z^* and taking the root, the result being a real number. The norm is commonly referred to as the *modulus* or *absolute value*:

$$|z| = \sqrt{z^* z}, \quad \text{or, equivalently,} \quad |z|^2 = z^* z.$$

A complex number $z = x + iy$ can be drawn in the 2D plane[1] where the x and y give the coordinates. If you think of a complex number as a vector from the origin to the coordinate (x,y), the norm of a complex number is the length of this vector. It is a real number and can be computed using Pythagoras' theorem as

[1] Also called the *complex plane*.

$$|z| = |x + iy| = \sqrt{(x - iy)(x + iy)} = \sqrt{x^2 + y^2}.$$

Note the difference between the square of a complex number and its squared norm. The square is computed as

$$z^2 = (x + iy)^2 = (x + iy)(x + iy) = x^2 + 2ixy - y^2.$$

Complex exponentiation is defined by Euler's famous formula:

$$re^{i\phi} = r\big(\cos\phi + i\sin\phi\big).$$

Complex numbers with norm $|z| = r = 1.0$ are on a unit circle:

$$z = e^{i\phi} = \cos\phi + i\sin\phi.$$

In Python, complex numbers are conveniently a part of the language. However, note that the imaginary i is written as `j` in Python, which is commonly used in electrical engineering. For the example of $x = 1 + \frac{i}{2}$, we write in Python:

```
x = 1.0 + 0.5j
x.real  # returns 1.0
x.imag  # returns 0.5
```

To conjugate, you can use the built-in `conjugate()` function for complex data types or use `numpy`'s `conj()` function. For example:

```
x_conj = x.conjugate()   # Python builtin, or
x_conj = np.conj(x)      # via numpy
```

1.2 Dirac Notation, Bras, and Kets

In quantum computing, we think of qubits and states as column vectors of n complex numbers, where n is typically a power of 2. We will soon learn why this is the case. A vector with n elements is also called an *n-dimensional* vector. In the so-called *Dirac* notation, or *bra–ket* notation, a column vector is called a *ket* and written as $|x\rangle$ with

$$|x\rangle = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}, \quad \text{with } x_i \in \mathbb{C} \text{ and } |x\rangle \in \mathbb{C}^n.$$

Recall that to *transpose* a matrix A, we take the column i of A and make it row i of the transpose A^T, or $A^T_{ij} = A_{ji}$. The *Hermitian conjugate* of a column vector $|x\rangle$, denoted by a dagger $|x\rangle^\dagger$, is the transpose of the vector with each element conjugated. This is also called the *adjoint* of the vector. We write it as $\langle x|$, changing the direction of the angle bracket and indicating that now we have a row vector:

$$|x\rangle^\dagger = \big(|x\rangle^*\big)^T = \langle x| = \begin{pmatrix} x_0^* & x_1^* & \dots & x_{n-1}^* \end{pmatrix}.$$

In Dirac notation, this row vector $\langle x|$ is called a *bra* or the *dual vector* of ket $|x\rangle$. Transposition and conjugation go both ways – applying the transformation twice results in the original ket. In other words, the dagger operation is its own inverse, a property called *involutivity*:

$$\begin{aligned} |x\rangle^\dagger &= \langle x|, \\ \langle x|^\dagger &= |x\rangle, \\ (|x\rangle^\dagger)^\dagger &= |x\rangle. \end{aligned}$$

There is the potential for confusion around conjugates. Conjugates in a bra are *not* explicitly marked with x_i^* or $x_i^\dagger$, as in $\langle x_0^* \;\; x_1^* \;\; \dots \;\; x_{n-1}^*|$. Converting a ket to a bra already *implies* conjugation of all vector elements.

1.3 Inner Product

The *inner product* of two vectors, which is also called the *scalar product* or the *dot product*, is computed as a matrix product of a bra and a ket, which simplifies to the product between a row vector and a column vector – an element-wise vector multiplication and summation, which produces a single number. It is written in the following forms, with the dot $(\cdot)$ denoting a scalar product:

$$\langle x| \cdot |y\rangle = \langle x||y\rangle = \langle x|y\rangle.$$

For a ket $|x\rangle$ and its dual bra vector $\langle x|$, the inner product with another ket $|y\rangle$ is defined as

$$|x\rangle = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}, \quad \langle x| = \begin{pmatrix} x_0^* & x_1^* & \dots & x_{n-1}^* \end{pmatrix}, \quad \text{and} \quad |y\rangle = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix},$$

$$\langle x|y\rangle = x_0^* y_0 + x_1^* y_1 + \cdots + x_{n-1}^* y_{n-1}.$$

The inner product is how the vectors in this notation get their names because they form the product of a bra and a ket, a *bra(c)ket*. Naming is difficult in general, and quantum computing is no exception. The inner product of complex vectors can result in a complex value. Note that $\langle x|y\rangle$ generally does not equal $\langle y|x\rangle$. For example, consider two kets $|x\rangle$ and $|y\rangle$:

$$|x\rangle = \begin{pmatrix} -1 \\ 2i \\ 1 \end{pmatrix}, \quad |y\rangle = \begin{pmatrix} 1 \\ 0 \\ i \end{pmatrix}. \tag{1.1}$$

We construct the corresponding bras as Hermitian conjugates (transposition, and negation of the imaginary parts):

$$\langle x| = \begin{pmatrix} -1 & -2i & 1 \end{pmatrix}, \quad \langle y| = \begin{pmatrix} 1 & 0 & -i \end{pmatrix}.$$

We then compute the two inner products as

$$\langle x|y\rangle = (-1)1 - (2i)0 + 1i = -1 + i,$$
$$\langle y|x\rangle = 1(-1) + 0(2i) - 1i = -1 - i.$$

The two inner products are different. The second result is the conjugate of the first, which points to the general rule:

$$\langle x|y\rangle^* = \langle y|x\rangle.$$

Two vectors are *orthogonal* if and only if their scalar product is zero. For 2D or 3D vectors, we can visualize orthogonal vectors as being perpendicular to each other.

$$\langle x|y\rangle = 0 \quad \Leftrightarrow \quad x,y \text{ are orthogonal.}$$

A set of vectors is called *linear independent* if no vector in the set can be expressed as a linear combination of other vectors in the set. A set is called *orthogonal* if the scalar product of any pair of distinct vectors is 0.

A set of vectors forms a *basis* for the (vector) space of all vectors that can be constructed from linear combinations of the vectors.[2] The basis for a given vector space is not unique, as we shall see later in this book. Typically, only orthogonal basis vectors are considered.

Related to the way we compute the norm of a complex number, the norm of a complex vector[3] is the root of the scalar product of the vector with its dual vector. A vector is *normalized* if its norm (or just its inner product) is 1:

$$\big||x\rangle\big| = \sqrt{\langle x|x\rangle} = 1 \quad \Rightarrow \ |x\rangle \text{ is normalized.}$$

State vectors in quantum computing represent probability distributions that must add up to 1 by definition. Hence, as we will see shortly, normalized vectors play an important role in quantum computing.

1.4 Outer Product

Where there is an inner product, there should also be an *outer* product. We can construct the outer product between a ket $|x\rangle$ and a bra $\langle y|$ by changing the order of the operands of the inner product. Instead of $\langle y|x\rangle$, we write

$$|x\rangle\langle y| = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \begin{pmatrix} y_0^* & y_1^* & \cdots & y_{n-1}^* \end{pmatrix} = \begin{pmatrix} x_0y_0^* & x_0y_1^* & \cdots & x_0y_{n-1}^* \\ x_1y_0^* & x_1y_1^* & \cdots & x_1y_{n-1}^* \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1}y_0^* & x_{n-1}y_1^* & \cdots & x_{n-1}y_{n-1}^* \end{pmatrix}.$$

[2] Vector spaces and their properties are an important topic in linear algebra. However, in this book, we will not go into great depth on this topic.

[3] This norm is also called the Euclidean norm or L^2 norm. Vector norms are often written with two bars on each side, such as $||x||$ or $||\,|x\rangle\,||$. However, for ease of readability, we will only use single bars in this book.

In the example given by Equation (1.1), $|x\rangle$ is a 3×1 vector, $|y\rangle$ is a 1×3 vector, and $\langle y|$ is a 3×1 vector. According to the rules of matrix multiplication, their outer product will be a 3×3 matrix. Again, if the vector elements are complex, we conjugate the vector elements when converting from bra to ket and vice versa.

1.5 Tensor Product

To denote the *tensor product*[4] of two vectors, which can be either bras or kets, we use the $\otimes$ operator symbol and may use one of the following shorthand notations for kets

$$|x\rangle \otimes |y\rangle = |x\rangle\,|y\rangle = |x,y\rangle = |xy\rangle\,, \tag{1.2}$$

and bras

$$\langle x| \otimes \langle y| = \langle x|\langle y| = \langle x,y| = \langle xy|.$$

In a tensor product, each element of the first constituent is multiplied by the whole of the second constituent. Therefore, an $n \times m$ matrix tensored with a $k \times l$ matrix will result in an $nk \times ml$ matrix. For example, to compute the tensor products of the following two kets:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

$$|0\rangle \otimes |1\rangle = |01\rangle = \begin{pmatrix} 1 & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 0 & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

You can see that the tensor product of two kets is a ket, the tensor product of two bras is a bra, and the tensor product of two diagonal matrices is a diagonal matrix.

Of course, tensor products are also defined for general matrices. Here we show an example of two 2×2 matrices being tensored together:

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \otimes \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}\begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} & a_{01}\begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \\ a_{10}\begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} & a_{11}\begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} a_{00}b_{00} & a_{00}b_{01} & a_{01}b_{00} & a_{01}b_{01} \\ a_{00}b_{10} & a_{00}b_{11} & a_{01}b_{10} & a_{01}b_{11} \\ a_{10}b_{00} & a_{10}b_{01} & a_{11}b_{00} & a_{11}b_{01} \\ a_{10}b_{10} & a_{10}b_{11} & a_{11}b_{10} & a_{11}b_{11} \end{pmatrix}.$$

For multiplication of scalars α and β with a tensor product, these rules apply:

$$\alpha(|x\rangle \otimes |y\rangle) = \alpha\,|x\rangle \otimes |y\rangle = |x\rangle \otimes \alpha\,|y\rangle\,,$$

[4] I am ignoring the differences between the tensor product and the Kronecker product and will use these terms interchangeably.

$$(\alpha+\beta)(|x\rangle\otimes|y\rangle)=\alpha\,|x\rangle\otimes|y\rangle+\beta\,|x\rangle\otimes|y\rangle\,.$$

Assume that we have a tensor product of two matrices A and B, and another tensor product of two vectors $|a\rangle$ and $|b\rangle$. If the two products are multiplied with standard matrix multiplication, the very important *mixed-product* rule applies, which is used in many places in this text:

$$(A\otimes B)(|a\rangle\otimes|b\rangle)=A\,|a\rangle\otimes B\,|b\rangle\,. \tag{1.3}$$

Transposition and conjugation distribute over the tensor product:

$$(A\otimes B)^T=A^T\otimes B^T,$$
$$(A\otimes B)^*=A^*\otimes B^*.$$

Because the adjoint consists of the transpose and the complex conjugate, the adjoint is distributed as well:

$$(A\otimes B)^\dagger=A^\dagger\otimes B^\dagger,\quad\text{and similarly} \tag{1.4}$$
$$\big(|\phi\rangle\otimes|\chi\rangle\big)^\dagger=\langle\phi|\otimes\langle\chi|.$$

With this, we find that for two *composite* kets

$$|\psi_1\rangle=|\phi_1\rangle\otimes|\chi_1\rangle\quad\text{and}\quad|\psi_2\rangle=|\phi_2\rangle\otimes|\chi_2\rangle\,,$$

and with Equation (1.3), the inner product between $|\psi_1\rangle$ and $|\psi_2\rangle$ is

$$\begin{aligned}\langle\psi_1|\psi_2\rangle&=\big(|\phi_1\rangle\otimes|\chi_1\rangle\big)^\dagger\big(|\phi_2\rangle\otimes|\chi_2\rangle\big)\\&=\big(\langle\phi_1|\otimes\langle\chi_1|\big)\big(|\phi_2\rangle\otimes|\chi_2\rangle\big)\\&=\langle\phi_1|\phi_2\rangle\langle\chi_1|\chi_2\rangle.\end{aligned} \tag{1.5}$$

It follows that the tensor product of two unit vectors (with norm 1) also has the unit norm.[5]

1.6 Eigenvalues and Eigenvectors

There is a special case of matrix-vector multiplication where the following equation holds, where A is a square matrix, $|\psi\rangle$ a ket, and λ a complex scalar:

$$A|\psi\rangle=\lambda|\psi\rangle.$$

Applying A to the special vector $|\psi\rangle$ only *scales* the vector with a complex number, it does not change its orientation. We call λ an *eigenvalue* of A. There can be multiple eigenvalues for a given matrix. The corresponding vectors for which this equation holds are called *eigenvectors*. In quantum mechanics, the synonym *eigenstates* is also used. Zero vectors are generally excluded from the exclusive club of eigenvectors.

For a diagonal matrix, finding the eigenvalues is trivial. Given a diagonal matrix of the form

[5] Discussed in http://quantumcomputing.stackexchange.com/a/32146.

$$\begin{pmatrix} \lambda_0 & & & \\ & \lambda_1 & & \\ & & \ddots & \\ & & & \lambda_{n-1} \end{pmatrix},$$

we can pick the eigenvalues right off the diagonal. The corresponding eigenvectors are $(1,0,0,\ldots)^T, (0,1,0,\ldots)^T, \ldots, (0,0,\ldots,1)^T$, a set that is also called *computational basis*. Note that any multiple of an eigenvector is also an eigenvector for a given eigenvalue.

Generally, eigenvalues for a given (smaller) matrix U can be found by solving[6] the *characteristic equation* $\det(U-\lambda I) = 0$. In this text, we keep it simple and use `numpy` to find the eigenvalues of a given matrix:

```
import numpy as np
[...]
umat = ... # some matrix
eigvals, eigvecs = np.linalg.eig(umat)
```

1.7 Hermitian Matrices

A square matrix A is a *Hermitian matrix* if it is equal to its transposed complex conjugate $A^\dagger$. As such, the diagonal elements must be real numbers, and the elements mirrored along the main diagonal are complex conjugates of each other. For example, the following matrix A is Hermitian:

$$A = A^\dagger = \begin{pmatrix} 1 & 3+i\sqrt{2} \\ 3-i\sqrt{2} & 0 \end{pmatrix}.$$

Similarly to the way we compute Hermitian conjugates for vectors in Section 1.2, to construct the Hermitian conjugate of a square matrix, you have to transpose the matrix and conjugate its elements. A Hermitian conjugate is also called *Hermitian adjoint*, or just *adjoint* for short. The terms adjoint and Hermitian conjugate are synonymous and can be used interchangeably.

The eigenvalues of Hermitian matrices are always real. A perhaps surprising property of Hermitian matrices is that their eigenvectors are orthogonal for distinct eigenvalues. A Hermitian matrix M is *positive semidefinite* if all its eigenvalues λ_i are positive, denoted by $M \geq 0$. Similarly, M is positive semidefinite if $\langle v|M|v\rangle \geq 0$ for all vectors $|v\rangle$. This will become clear later in Section 4.1 on the spectral decomposition.

1.8 Unitary Matrices

A square matrix A is *normal* if $AA^\dagger = A^\dagger A$. It is *unitary* if its conjugate transpose is equal to its inverse, with $A^\dagger A = AA^\dagger = I$. Both Hermitian and unitary matrices are

[6] With det being the determinant of a matrix. See, for example: http://en.wikipedia.org/wiki/Determinant. The matrix I is the identity matrix which has 1s on the diagonal and 0s everywhere else.

normal. Unitary matrices are *norm-preserving*. Multiplying a unitary matrix with a vector might change the orientation of the vector, but it will not change its norm. The columns of a unitary matrix form an orthonormal basis in $\mathbb{C}^n$. In general, a matrix is unitary if it transforms any orthonormal basis in $\mathbb{C}^n$ into another orthonormal basis.

The eigenvectors of a unitary (square and complex) matrix are orthonormal. We can prove[7] that the eigenvalues of a unitary matrix are *unimodular*; they have a modulus of 1.

Proof We know that eigenvalues are defined as

$$U|u\rangle = \lambda|u\rangle.$$

Assume a normalized eigenvector $|u\rangle$ with an inner product of 1. By computing the norm on both sides, we have

$$\langle uU^\dagger|Uu\rangle = \langle u\lambda^*|\lambda u\rangle.$$

We know that $U^\dagger U = I$ because U is unitary. We pull the factor $(\lambda^*|\lambda) = |\lambda|^2$ in front of the inner product:

$$\begin{aligned} \langle uU^\dagger|Uu\rangle &= (\lambda^*\lambda)\langle u|u\rangle, \\ \langle u|u\rangle &= |\lambda|^2\langle u|u\rangle, \\ \Rightarrow |\lambda|^2 &= 1. \end{aligned}$$

□

Since $|\lambda^2| = 1$, we can write the complex eigenvalues as $\lambda = e^{i\phi}$. In the following example, the matrix Y is both unitary and Hermitian. The matrix S is unitary, but *not* Hermitian:

$$Y = Y^\dagger = \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix} \quad \text{and} \quad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \neq \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} = S^\dagger.$$

1.9 Hermitian Adjoint of Expressions

Here are the rules for conjugating expressions of matrices and vectors. We use these rules extensively throughout this book. Previously, we learned how to convert between bras and kets as $|\psi\rangle^\dagger = \langle\psi|$ and $\langle\psi|^\dagger = |\psi\rangle$. To compute the adjoint of a matrix scaled by a complex factor α, we use

$$(\alpha A)^\dagger = \alpha^* A^\dagger = A^\dagger \alpha^*.$$

For matrix–matrix products, the order reverses (this is an important rule used often in this book):

$$(AB)^\dagger = B^\dagger A^\dagger. \tag{1.6}$$

Note that this differs from the rule for the tensor product shown in Equation (1.4). To compute the adjoint for products of matrices and vectors, the order reverses as well:

[7] This simple proof showcases a few of the tricks used later in this book.

$$(A|\psi\rangle)^\dagger = \langle\psi|A^\dagger,$$
$$(AB|\psi\rangle)^\dagger = \langle\psi|B^\dagger A^\dagger.$$

For matrices in outer product notation, this rule follows from Equation (1.6) (by taking $|\psi\rangle$ as the A in Equation (1.6) and $\langle\phi|$ as the B):

$$A = |\psi\rangle\langle\phi| \quad \Rightarrow \quad A^\dagger = |\phi\rangle\langle\psi|.$$

And finally, the adjoint of a sum of two operators is

$$(A+B)^\dagger = A^\dagger + B^\dagger.$$

1.10 Trace of a Matrix

The *trace* of an $n \times n$ matrix A is defined as the sum of its diagonal elements:

$$\mathrm{tr}(A) = \sum_{i=0}^{n-1} a_{ii} = a_{00} + a_{11} + \cdots + a_{n-1n-1}\,.$$

The following are basic properties of the trace, where c is a scalar, and A and B are square matrices:

$$\mathrm{tr}(A+B) = \mathrm{tr}(A) + \mathrm{tr}(B),$$
$$\mathrm{tr}(cA) = c\,\mathrm{tr}(A),$$
$$\mathrm{tr}(AB) = \mathrm{tr}(BA).$$

In general, the trace operation is *cyclic* (as long as the dimensions of the matrices allow it). For example, for a product of three matrices,

$$\mathrm{tr}(ABC) = \mathrm{tr}(BCA) = \mathrm{tr}(CAB).$$

For the trace of tensor products, this important relation holds:

$$\mathrm{tr}(A \otimes B) = \mathrm{tr}(A)\,\mathrm{tr}(B).$$

The next relation is important for quantum measurements, as we will discover soon. Suppose we have two kets $|x\rangle$ and $|y\rangle$:

$$|x\rangle = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \quad \text{and} \quad |y\rangle = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

The trace of the outer product $|x\rangle\langle y|$ is equal to the inner product of the operands in reverse order:

$$\mathrm{tr}(|x\rangle\langle y|) = \langle y|x\rangle. \tag{1.7}$$

We can see this directly from writing the outer product as

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \begin{pmatrix} y_0^* & y_1^* & \cdots & y_{n-1}^* \end{pmatrix} = \begin{pmatrix} x_0 y_0^* & x_0 y_1^* & \cdots & x_0 y_{n-1}^* \\ x_1 y_0^* & x_1 y_1^* & \cdots & x_1 y_{n-1}^* \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1} y_0^* & x_{n-1} y_1^* & \cdots & x_{n-1} y_{n-1}^* \end{pmatrix}$$

$$\implies \operatorname{tr}(|x\rangle\langle y|) = \sum_{i=0}^{n-1} x_i y_i^* = \langle y|x\rangle.$$

We will use Equation (1.7) often in this book for expressions like the following:

$$\operatorname{tr}\big(\langle x|A|x\rangle\big) = \operatorname{tr}\big(A|x\rangle\langle x|\big) = \operatorname{tr}\big(|x\rangle\langle x|A\big).$$

Finally, the trace of a matrix A is the sum of its n eigenvalues λ_i, counted with *multiplicity*:[8]

$$\operatorname{tr}(A) = \sum_{i=0}^{n-1} \lambda_i. \tag{1.8}$$

We will use this property in several places in this book as well, but for a proof we have to wait until Section 4.1 on the spectral decomposition.

[8] Which means that identical eigenvalues are counted multiple times.

2 Quantum Computing Fundamentals

This chapter outlines the basic principles and rules of quantum computing. In parallel, we develop an initial, easy-to-understand, easy-to-debug code base for building and simulating smaller-scale algorithms.

The chapter is structured as follows. First, we introduce our basic underlying data type, the Python `Tensor` type, which is derived from `numpy`'s `ndarray` data type. Using this type, we construct single qubits and quantum states composed of many qubits. We define operators that allow us to modify states and describe a range of important single-qubit gates. Controlled gates, which play a similar role to control flow in classical computing, come next. We detail how to describe quantum circuits via the Bloch sphere and in quantum circuit notation. A discussion of entanglement follows, that fascinating "spooky action at a distance," as Einstein called it. In quantum physics, measurement might be even more problematic than entanglement (Norsen, 2017). In this text, we avoid philosophy and conclude the chapter by describing a simple way to simulate measurements.

2.1 Tensors

Quantum computing is expressed in the language of linear algebra, with vectors, matrices, and operations such as the inner product. Quantum algorithms are, to a large degree, algorithms based on linear algebra. Because of that, some of the mathematics is unavoidable. A very compressed summary of necessary mathematical concepts was already presented in the previous chapter. However, since this book has "programmer" in the title, we balance the mathematical development of the algorithms with working code for experimentation.

Let us start by describing a Python data structure that will serve as the basis for all the code in this book. Python may be slow to execute but is fast to develop.[1] It also has the vectorized and accelerated `numpy` numerical library for scientific computing. We make great use of this library and avoid implementing standard numerical linear algebra operations ourselves. In general, we follow Google's coding style guides for Python (Google, 2021b) and C++ (Google, 2021a).

One of the insights here is that it only takes a little bit of code to implement and simulate the algorithms. Of course, there are many existing frameworks and

[1] To be fair, only for programs up to moderate size.

libraries available. The advantage of quickly developing our own framework is that you can focus on learning quantum computing and not be distracted by having to learn another complex framework. Learning quantum computing is already hard enough.

The core data types, such as states and operators, are all vectors and matrices of complex numbers. It is good practice to base the types on one common array abstraction and hide the underlying implementation. The typical benefits are an improved development speed and a smaller jump from experimentation on your laptop to running on a distributed supercomputer. The data type for all subsequent work will be our Python `Tensor` class.

We derive `Tensor` from the `ndarray` array data structure in `numpy`. It will behave just like a `numpy` array, but we can augment it with additional convenience functionality. For example, for ease of debugging, we allow a tensor to have a descriptive name. There are several complex ways to instantiate an `ndarray`. The proper way to derive a class from this data type is complicated but well documented.[2]

Find the code
In file `src/lib/tensor.py`

```
import numpy as np

class Tensor(np.ndarray):
  def __new__(cls, input_array, op_name=None) -> Tensor:
    cls.name = op_name
    return np.asarray(input_array, dtype=tensor_type()).view(cls)

  def __array_finalize__(self, obj) -> None:
    if obj is None:
      return
    # If new attributes are needed, add them like this:
    #   self.info = getattr(obj, 'info', None)
```

Note the use of `tensor_type()` in this code snippet: It abstracts the floating-point representation of complex numbers. The choice of which complex data type to use is an interesting question. Should we use complex numbers based on 64-bit doubles, 32-bit floats, or perhaps something else, for example, a TPU 16-bit bfloat[3] format? Smaller data types may be faster to simulate due to lower memory bandwidth requirements, but they come at the cost of reduced numerical precision. The `numpy` package supports `np.complex128` (consisting of two 64-bit doubles) and `np.complex64` (with two 32-bit floats). We define a command line flag that holds the width of the type and functions to return the corresponding `numpy` data type and the number of bits:

[2] Refer to http://numpy.org/doc/stable/user/basics.subclassing.html.

[3] TPU stands for Google's "Tensor Processing Unit," a hardware accelerator for machine learning algorithms. It also introduced the *bfloat* data type, which is a standard fp32 data type but without the lower 16 bits.

```
from absl import flags
flags.DEFINE_integer('tensor_width', 64, 'Width of complex (64, 128)')

def tensor_width():
  return flags.FLAGS.tensor_width

def tensor_type():
  assert tensor_width() == 64 or tensor_width() == 128
  return np.complex64 if tensor_width() == 64 else np.complex128
```

As we shall see in our discussion of quantum states in Section 2.4, the Kronecker product of tensors is an important operation. As mentioned in Section 1.5, this product is commonly referred to as the tensor product, which is also the term we will use.[4] We implement it by adding the member function `kron` to the `Tensor` class. This function delegates to the function of the same name in `numpy`.

We will use this operation in many places, so we additionally overload the Python multiplication operator $*$ for convenience. There is the potential to confuse this $*$ operator with simple matrix multiplication. However, in Python and in `numpy`, matrix multiplication is done with the *at* operator `@`. We conveniently inherit this multiplication operator from `numpy` and do not have to implement it ourselves:

```
def kron(self, arg: Tensor) -> Tensor:
  return self.__class__(np.kron(self, arg))

def __mul__(self, arg: Tensor) -> Tensor:
  return self.kron(arg)
```

We will often construct larger matrices by tensoring together many *identical* matrices, which corresponds to calling the `kron` function multiple times. To tensor together n matrices A, we will use a notation similar to raising the matrix to the power of n, but add the $\otimes$ operator in the notation:

$$\underbrace{A \otimes A \otimes \cdots \otimes A}_{n} = A^{\otimes n} \quad \neq \quad A^n = \underbrace{AA \cdots A}_{n}.$$

It looks like a power function, but instead of matrix multiplication, it uses Kronecker products. Naming is hard, but this function names itself: We should call it the Kronecker power function, or `kpow` (pronounced "Kah-Pow"). We handle cases where the exponent is 0 as a special case with $x^0 = 1$. As expected, `numpy` correctly computes tensor products with scalars.

```
def kpow(self, n: int) -> Tensor:
  if n == 0:
    return self.__class__(1.0)
  t = self
  for _ in range(n - 1):
```

[4] *Tensoring states* rolls off the tongue much more easily than *Kroneckering states*.

```
    t = np.kron(t, self)
  return self.__class__(t)  # Return a Tensor type
```

Often, especially during testing, we want to compare a `Tensor` with another tensor. We are working with complex numbers based on floating-point data types. Direct comparison of values of these types is considered bad practice due to issues with floating-point precision. Instead, for equality, we have to check that the difference between two numerical values is less than a given ε.

Fortunately, `numpy` comes to the rescue and offers the function `allclose()`, which compares full tensors, so we do not have to iterate over dimensions and compare real and imaginary parts. Here, and in almost all other places, we use a tolerance of 10^{-6} and add the `is_close` method to our `Tensor` type.[5] Python's `math` module has an `isclose()` function. However, we follow Google's coding style, which requires us to name functions with a trailing underscore after `is`, as in `is_close()`:

```
def is_close(self, arg) -> bool:
  return np.allclose(self, arg, atol=1e-6)
```

In Section 1.8, we learned about Hermitian and unitary matrices. The two helper functions below check for these properties:

```
def is_hermitian(self) -> bool:
  if len(self.shape) != 2 or self.shape[0] != self.shape[1]:
    return False
  return self.is_close(np.conj(self.transpose()))

def is_unitary(self) -> bool:
  return Tensor(np.conj(self.transpose()) @ self).is_close(
               Tensor(np.eye(self.shape[0])))
```

Another interesting matrix type is a *permutation matrix*, which has a single 1 in each row and column. Multiplying a column vector by such a matrix allows us to permute the vector elements. The `Tensor` class offers the member function `is_permutation()` to verify this matrix property:

```
def is_permutation(self) -> bool:
  x = self
  return (x.ndim == 2 and x.shape[0] == x.shape[1] and
          (x.sum(axis=0) == 1).all() and
          (x.sum(axis=1) == 1).all() and
          ((x == 1) or (x == 0)).all())
```

[5] Note that for scalars, `math.isclose` is significantly faster than `np.allclose`. We will use it in performance-critical code.

2.2 Qubits

In classical computing, a bit can have the values 0 or 1. It is off or on, like a switch. You could say that a bit is in the off state (0 state) or in the on state (1 state). Quantum bits, which we call *qubits*, can also be in a 0 or a 1 state. What makes them quantum is that they can be in *superposition* of these states: They can be in the 0-state and the 1-state at the same time. What exactly does this mean?

First, we must distinguish between a qubit and *state of a qubit*. Physical qubits, developed for real quantum computers, are real physical entities, such as ions captured in an electric field or Josephson junctions in an ASIC. The state of a qubit describes some measurable property of that qubit, such as the energy level of an electron.

In quantum computing, at the level of programming abstractions, the physical implementation does not matter; we are only concerned with the measurable state. This is analogous to classical computing, where very few people care about the quantum effects that enable transistors at the level of logic gates. In this text, we will use the terms qubit and state of the qubit interchangeably.

The state of one or more qubits is often denoted by the Greek symbol $|\psi\rangle$ ("psi"). The standard notation for the 0-state of a qubit is $|0\rangle$ in the Dirac notation and $|1\rangle$ for the 1-state. You can think of these as physically distinguishable states, such as the energy levels of electrons. *Superposition* now means that the state of a qubit is a linear combination of the orthonormal basis[6] states, for example, the $|0\rangle$ and $|1\rangle$ states, as

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

where α and β are complex numbers, called the *probability amplitudes*. We further require that

$$|\alpha|^2 + |\beta|^2 = 1, \tag{2.1}$$

for reasons explained below. Using the basis vectors $(1,\ 0)^T$ and $(0,\ 1)^T$, we define the state of a qubit elegantly as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

The choice of $(1,\ 0)^T$ and $(0,\ 1)^T$ as orthonormal basis vectors, which is also called the *computational basis*, is intuitive and simplifies many of the calculations. The basis vectors are orthogonal with a scalar product of $\langle 0|1\rangle = 0$ and normalized with scalar products of $\langle 0|0\rangle = \langle 1|1\rangle = 1$.

Other bases are possible, especially those resulting from rotations, which are commonplace in quantum computing. For example, the *Hadamard basis* consists of the two orthonormal vectors $|+\rangle$ and $|-\rangle$, which are defined as

[6] To be rigorous, one would say superposition can be the linear combination of *any* two distinct, not necessarily orthogonal states.

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix},$$

$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}.$$

For the superposition $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, we required $|\alpha|^2 + |\beta|^2 = 1$. As will become clear later, this follows from one of the fundamental postulates of quantum mechanics, which states that on measurement, the state collapses to $|0\rangle$ with probability $|\alpha|^2$, or to $|1\rangle$ with probability $|\beta|^2$. The state has to collapse to one of the two. The probabilities must add up to a real value of 1. Since amplitudes are complex numbers in general, we use the absolute value of the inner product to calculate the real physical probabilities.

Let us look at a standard example. Suppose we have a qubit in the state

$$|\phi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{i}{2}|1\rangle.$$

For a single complex number c, the conjugate c^* is the same[7] as the Hermitian adjoint $c^\dagger$. The probability $p_{|0\rangle}$ of measuring $|0\rangle$ is the norm squared:[8]

$$p_{|0\rangle} = \left|\frac{\sqrt{3}}{2}\right|^2 = \left(\frac{\sqrt{3}}{2}\right)^\dagger \left(\frac{\sqrt{3}}{2}\right) = \left(\frac{\sqrt{3}}{2}\right)\left(\frac{\sqrt{3}}{2}\right) = \frac{3}{4}.$$

To compute the probability $p_{|1\rangle}$ of measuring $|1\rangle$, we take the norm squared of the amplitude $i/2$ as

$$p_{|1\rangle} = \left|\frac{i}{2}\right|^2 = \left(\frac{i}{2}\right)^\dagger \left(\frac{i}{2}\right) = \left(\frac{-i}{2}\right)\left(\frac{i}{2}\right) = \frac{1}{4}.$$

You can see that the two probabilities of $3/4$ and $1/4$ add up to 1.

The following code will translate these concepts into a straightforward implementation. As a forward reference, we use the type `State`, which we will discuss in Section 2.4. In simple terms, `State` is a vector of complex numbers implemented using `Tensor`.

To construct a qubit, we need α or β, or both. If only one is provided, we can easily compute a candidate[9] for the other one since their squared norms must add up to 1. To compute the squared norms of the complex numbers α and β, we multiply each by its complex conjugate (using `np.conj`). The result will be a real number.[10] To avoid generating a type error from `numpy`, we have to explicitly convert the result to `np.real()`. We compare the results to 1.0, and if it is within tolerance, we construct and return the qubit as a `State`.

[7] We will often use the dagger for simplicity.

[8] We are really computing a projection $|\langle 0|\psi\rangle|^2$, but for more details we will have to wait for Section 2.13.2 on measurements.

[9] Here, we ignore a possible *local phase*, which we will learn about in Section 2.3.

[10] We could also use `np.abs(alpha)**2`, but I prefer it this way; it is more explicit. You will find this construction in many places in this book.

PY **Find the code**
In file src/lib/state.py

```
def qubit(alpha: complex = None, beta: complex = None) -> State:
  if alpha is None and beta is None:
    raise ValueError('alpha, beta, or both, need to be specified')
  if beta is None:
    beta = np.sqrt(1.0 - np.real(np.conj(alpha) * alpha))
  if alpha is None:
    alpha = np.sqrt(1.0 - np.real(np.conj(beta) * beta))

  norm2 = np.real(np.conj(alpha) * alpha) + np.real(np.conj(beta) * beta)
  assert math.isclose(norm2, 1.0), 'Qubit probabilities not equal to 1.'
  return State([alpha, beta])
```

2.3 Bloch Sphere

We now introduce the *Bloch sphere*, a 3D visualization of the state of a qubit, named after the famous physicist Felix Bloch, even though it was first introduced by Feynman (1957). It may be especially useful for visual learners. We will use it in the following sections to visualize the effect of operators on qubits. To begin, let us introduce some basic trigonometry and an angle θ. Using

$$\alpha = \cos\frac{\theta}{2} \quad \text{and} \quad \beta = \sin\frac{\theta}{2},$$

we meet the requirement from Equation (2.1) that

$$|\alpha|^2 + |\beta|^2 = \cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2} = 1.$$

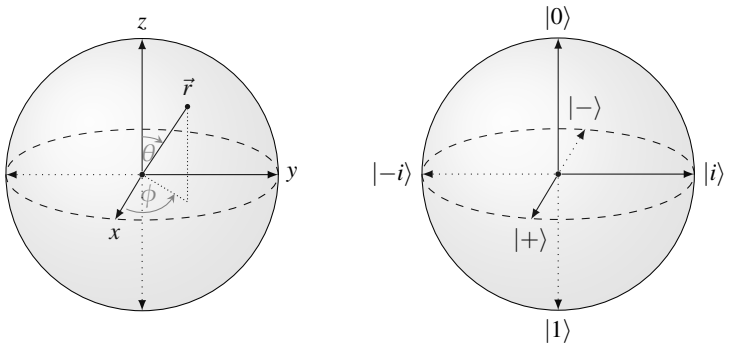
Now we introduce a second angle ϕ as a phase $e^{i\phi}$ between $|0\rangle$ and $|1\rangle$. This phase is called a *local phase* and it plays an important role in many algorithms. We must not ignore it, and, more importantly, Equation (2.1) still holds with it. With this, we can write a qubit in the alternative form

$$|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle\right). \tag{2.2}$$

The parameters γ and ϕ are real numbers in $[0,2\pi)$ and θ in $[0,\pi)$. The first term $e^{i\gamma}$ in Equation (2.2) is called a *global phase*. Multiplying a state by such a complex coefficient does not have an actual physical meaning because the *expectation value* of the state with or without the coefficient does not change. This is also related to what physicists call *phase invariance*.

The expectation value for an operator A on state $|\psi\rangle$ (which we will develop in Section 2.13 on measurement) is $\langle\psi|A|\psi\rangle$. The Hermitian adjoint of $(c\,|\psi\rangle)^\dagger = \langle\psi|c^*$. We can see that the expectation value with and without a global phase remains unchanged. States with or without a global phase cannot be distinguished:

$$\langle\psi|e^{-i\phi}Ae^{i\phi}|\psi\rangle = \langle\psi|e^{-i\phi}e^{i\phi}A|\psi\rangle = \langle\psi|A|\psi\rangle.$$



(a) Sphere with axes x, y, and z.

(b) Basis states on the sphere.

Figure 2.1 The Bloch sphere representation.

The two parameters θ and ϕ are sufficient to specify a qubit. This leads to a representation as a point on a three-dimensional sphere with unit radius as shown in Figure 2.1(a), where a qubit $|\psi\rangle$ lies on the surface of the sphere. With some trigonometry, we see that this point is specified by a vector $\vec{r} = (\cos\phi \sin\theta, \sin\phi \sin\theta, \cos\theta)$, the so-called *Bloch vector*.

Let us explore where we can find specific states on the sphere, as shown in Figure 2.1(b). The position at the sphere's north pole has $\theta = 0$ and $\phi = 0$ (other values for ϕ will still land the qubit on the north pole, but let us ignore this case for now). With this, the state becomes

$$\cos\frac{0}{2}|0\rangle + e^{i\phi}\sin\frac{0}{2}|1\rangle = 1|0\rangle + 0|1\rangle = |0\rangle\,.$$

The state $|0\rangle$ sits at the top. Similarly, for $\theta = \pi$, state $|1\rangle$ sits at the south pole:

$$\cos\frac{\pi}{2}|0\rangle + e^{i\phi}\sin\frac{\pi}{2}|1\rangle = 0|0\rangle + 1|1\rangle = |1\rangle\,.$$

The points where the positive and negative x-axes intersect with the sphere have angles $\theta = \pi/2$ with $\phi = 0$ and $\phi = \pi$. This is where we find the *Hadamard bases* $|+\rangle$ and $|-\rangle$ with

$$\cos\frac{\pi/2}{2}|0\rangle + e^{i\phi}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = |+\rangle\,,$$

$$\cos\frac{\pi/2}{2}|0\rangle + e^{i\pi}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle = |-\rangle\,.$$

Finally, the points that intersect the positive and negative y-axis have angles $\theta = \pi/2$ with $\phi = \pi/2$ and $\phi = 3\pi/2$. These states form another basis which is affectionately, and sometimes confusingly, denoted as $|i\rangle$ and $|-i\rangle$:

$$\cos\frac{\pi/2}{2}|0\rangle + e^{\pi/2}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle + i\frac{1}{\sqrt{2}}|1\rangle = |\,i\,\rangle \;\; = |{+}y\rangle\,,$$

$$\cos\frac{\pi/2}{2}|0\rangle + e^{3\pi/2}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - i\frac{1}{\sqrt{2}}|1\rangle = |{-}i\rangle = |{-}y\rangle\,.$$

The term $|i\rangle$ can be confusing because it is often used to denote arbitrary computational basis states. Instead of $|i\rangle$ and $|-i\rangle$, we will also use the terms $|+y\rangle$ and $|-y\rangle$ for these basis states. To add to the confusion, note that states with anti-parallel Bloch vectors are orthogonal. Orthogonal states do not have orthogonal Bloch vectors.

Another interesting question is how to compute the x,y,z coordinates for a given state $|\psi\rangle$ on a Bloch sphere. We will have to make more progress before we can answer this question in Section 2.7.3. Bloch spheres are only defined for single-qubit states. You can visualize the Bloch sphere of an individual qubit in a multi-qubit system by *tracing out* all the other qubits in the state. This is done with the *partial trace* procedure, a useful tool we introduce in Section 4.3.

2.4 States

As we saw in Section 2.2, the possible *quantum state* of a qubit is a vector of complex numbers that represent probability amplitudes. We should use our trusty `Tensor` class to represent states in code and inherit the `State` class from `Tensor`. In this way, we also conveniently inherit the Python `__repr__` and `__str__` functions from the base class.

Find the code
In file `src/lib/state.py`

```
class State(tensor.Tensor):
  """class State represents single- and multi-qubit states."""
```

So far, we have learned how to construct a single-qubit state. But what about a state that consists of multiple qubits? The state of two or more qubits is defined as their tensor product. To compute it, we added the `*` operator to the underlying `Tensor` type in Section 2.1 (implemented as the corresponding Python `__mul__` member function). Given this definition, the quantum state of n qubits is a `Tensor` of 2^n complex probability amplitudes. And we already know, from Equation (1.2), that for two qubits $|\phi\rangle$ and $|\chi\rangle$ we can write the combined state as

$$|\psi\rangle = |\phi\rangle \otimes |\chi\rangle = |\phi\rangle|\chi\rangle = |\phi,\chi\rangle = |\phi\chi\rangle.$$

For two qubits, there are four basis states, and we can write the state $|\psi\rangle$ as[11]

[11] By convention, computational basis states are often denoted as $|e_i\rangle$. Basis states for a general state $|\psi\rangle$ may also be written as $|\psi_0\rangle, \ldots, |\psi_{n-1}\rangle$, which is a convention we will use often.

$$
\begin{aligned}
|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} &= c_0 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + c_1 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + c_2 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + c_3 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \\
&= c_0 |e_0\rangle + c_1 |e_1\rangle + c_2 |e_2\rangle + c_3 |e_3\rangle \\
&= c_0 |\psi_0\rangle + c_1 |\psi_1\rangle + c_2 |\psi_2\rangle + c_3 |\psi_3\rangle \\
&= \sum_{i=0}^{3} c_i |\psi_i\rangle.
\end{aligned}
$$

We already learned in Section 1.5 that the norm of a tensor product of vectors with unit norm is also 1, which is exactly what we need for state vectors to represent probabilities. Probability amplitudes are complex numbers. To compute the inner product, we multiply by the complex conjugates and exploit the fact that the basis states are normalized with an inner product of 1:

$$
\begin{aligned}
\langle\psi|\psi\rangle &= c_0^* \langle\psi_0| c_0 |\psi_0\rangle + c_1^* \langle\psi_1| c_1 |\psi_1\rangle + \cdots + c_n^* \langle\psi_{n-1}| c_n |\psi_{n-1}\rangle \\
&= c_0^* c_0 \langle\psi_0|\psi_0\rangle + c_1^* c_1 \langle\psi_1|\psi_1\rangle + \cdots + c_n^* c_n \langle\psi_{n-1}|\psi_{n-1}\rangle \\
&= c_0^* c_0 + c_1^* c_1 + \cdots + c_{n-1}^* c_{n-1} \\
&= 1\,.
\end{aligned}
$$

We can extract an individual value c_i by computing the inner product of the state with the corresponding computational basis vector $|e_i\rangle$. For example, to extract c_2 (you can get c_2^* by reversing the order of the inner product):

$$
\langle e_2|\psi\rangle = \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \end{pmatrix}^T = c_2. \tag{2.3}
$$

2.4.1 Tensoring States

To build systems of multiple qubits, the individual states of the participating qubits are tensored together. Given our definition of the tensor product in Section 1.3, this was easy to understand when the states were expressed as vectors:

$$
\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a \begin{pmatrix} c \\ d \end{pmatrix} \\ b \begin{pmatrix} c \\ d \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix}. \tag{2.4}
$$

But what if a state $|\psi\rangle$ is written as an expression, such as

$$
|\psi\rangle = \big(a|0\rangle + b|1\rangle\big) \otimes \big(c|0\rangle + d|1\rangle\big).
$$

Similarly, and sometimes confusingly, the product is often written *without* the operator $\otimes$, as

$$
\big(a|0\rangle + b|1\rangle\big) \otimes \big(c|0\rangle + d|1\rangle\big) \equiv \big(a|0\rangle + b|1\rangle\big)\big(c|0\rangle + d|1\rangle\big).
$$

This can be confusing because it may look like a matrix product or dot product. It is important to be aware of the context. We can "multiply out" the expression, just like a normal product of two terms. As we multiply the two bracketed terms, the scalar factors turn into simple products, and the qubit states are tensored together. For scalar products, the order of their operands does not matter. For qubit states, the ordering must be maintained:

$$\begin{aligned}|\psi\rangle &= \big(a\,|0\rangle + b\,|1\rangle\big)\big(c\,|0\rangle + d\,|1\rangle\big)\\ &= a\,|0\rangle\,\big(c\,|0\rangle + d\,|1\rangle\big) + b\,|1\rangle\,\big(c\,|0\rangle + d\,|1\rangle\big)\\ &= ac\,|00\rangle + ad\,|01\rangle + bc\,|10\rangle + bd\,|11\rangle\,.\end{aligned}$$

Writing this state as a vector results in the same state vector $\begin{pmatrix} ac & ad & bc & bd \end{pmatrix}^T$ as in Equation (2.4). To make the required ordering clear, individual qubits sometimes get a subscript, indicating who they belong to, such as Alice or Bob:[12]

$$\begin{aligned}|\psi\rangle &= \big(a\,|0_A\rangle + b\,|1_A\rangle\big)\big(c\,|0_B\rangle + d\,|1_B\rangle\big)\\ &= ac\,|0_A0_B\rangle + ad\,|0_A1_B\rangle + bc\,|1_A0_B\rangle + bd\,|1_A1_B\rangle\,.\end{aligned}$$

The multiplication procedure can be reversed; we can *factor out* individual qubits. This should not be surprising, but it may be helpful to see it at least once:

$$\begin{aligned}|\psi\rangle &= ac\,|00\rangle + ad\,|01\rangle + bc\,|10\rangle + bd\,|11\rangle\\ &= |0\rangle\,\big(ac\,|0\rangle + ad\,|1\rangle\big) + |1\rangle\,\big(bc\,|0\rangle + bd\,|1\rangle\big)\\ &= a\,|0\rangle\,\big(c\,|0\rangle + d\,|1\rangle\big) + b\,|1\rangle\,\big(c\,|0\rangle + d\,|1\rangle\big)\\ &= \big(a\,|0\rangle + b\,|1\rangle\big)\big(c\,|0\rangle + d\,|1\rangle\big).\end{aligned}$$

You will find these types of state manipulations in several places in this book.

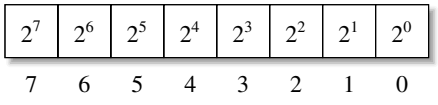
2.4.2 Qubit Ordering

As we compose states of multiple qubits, we must consider the issue of *endianness*. Consider how the bits in a typical byte are numbered, with the least significant bit 0 on the right, as shown in Figure 2.2(a). However, when we think of arrays, we typically have element 0 at address 0 at the top of the array, as shown in Figure 2.2(b).

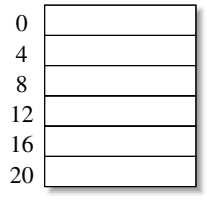
Classical binary numbers are combinations 0s or 1s, which we interpret as an n-ary number. In quantum computing, we can also combine multiple qubits in the basis states $|0\rangle$ and $|1\rangle$ with the tensor product and interpret the resulting state as a classical binary number. There are two distinct conventions:

1. In the *little-endian* convention, the least significant part of a data structure is placed at the lowest address. The Intel x86 family of CPUs follows this convention. The hexadecimal value `0x1234` is stored in a 16-bit memory space as `0x3412`, with the least significant byte `0x34` at the lower byte address.

[12] Alice and Bob are widely used as stand-ins to denote two distinct systems A and B.



(a) In a byte, bits are numbered from right to left.



(b) Array addresses go from top to bottom or left to right.

Figure 2.2 Ordering conventions for most/least significant parts.

2. On the other hand, in the *big-endian* convention, the most significant part of a data structure comes first. The IBM Power CPU family follows this convention, representing the hex value `0x1234` as the 16-bit integer `0x1234`, with the higher-order byte `0x12` at the lower byte address.

As you can see, it depends on convention, how entries are numbered in an underlying array-like structure, and what endianness convention is used. In this book, we primarily use the big-endian convention. We number the qubits from 0 to $n-1$ and place the most significant qubit first at index 0. In this way, when we see a tensor product such as $|0\rangle \otimes |1\rangle \otimes |1\rangle$, we know to interpret it as binary `0b011` or decimal 3.

There is no standard convention, and major software frameworks use different conventions. Some of the algorithms in this book are based on reference implementations. In such cases, we may switch to the little-endian convention. The key points to internalize are:

- As qubits are added to a circuit, they are added from left to right (in a binary string), from the high-order qubit to the low-order qubit.
- In Dirac notation, a two-qubit state is written as $|x,y\rangle$, for example, as $|0,1\rangle$ or $|01\rangle$. The most significant qubit is the first to appear in big-endian notation:

$$\underbrace{|0\rangle}_{\text{High-order}} \otimes \cdots \otimes \underbrace{|0\rangle}_{\text{Low-order}} .$$

- We will see in Section 2.9 that the circuits are drawn as a vertical stack of qubits, and the top qubit is considered the most significant in big-endian notation.
- We will soon learn about simple functions to construct composite states from $|0\rangle$ and $|1\rangle$ states. In these functions, the first qubit to appear will be the most significant qubit, similar to the circuit notation. For example, we call `state.bitstring(1, 1, 0)` to generate the state $|\psi\rangle = |1\rangle \otimes |1\rangle \otimes |0\rangle$.
- When we print a state, the most significant bit will also be on the left.

2.4.3 Binary Interpretation

We can write tensor products of the basis states $|0\rangle$ and $|1\rangle$ as in this three-qubit example:

$$|0\rangle \otimes |1\rangle \otimes |1\rangle = |011\rangle.$$

For brevity, when interpreting the bit strings as binary numbers, we can simplify the notation and write out the binary numbers as decimals, as in this example:

$$|011\rangle = |3\rangle.$$

Be aware of the potential for confusion between the state $|000\rangle$, the corresponding decimal state $|0\rangle$, and the state $|0\rangle$ of a single qubit. How does the decimal interpretation of the sequence of basis states relate to the state vector?

- State $|00\rangle$ is computed as $\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & 0\end{pmatrix}^T$, also called $|0\rangle$.
- State $|01\rangle$ is computed as $\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0 & 1 & 0 & 0\end{pmatrix}^T$, also called $|1\rangle$.
- State $|10\rangle$ is computed as $\begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}0 & 0 & 1 & 0\end{pmatrix}^T$, also called $|2\rangle$.
- State $|11\rangle$ is computed as $\begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0 & 0 & 0 & 1\end{pmatrix}^T$, also called $|3\rangle$.

To find the probability amplitude for a given state, we can use binary addressing. The state vector for the three-qubit state $|011\rangle$ is the following, where we indicate the index into the state vector as i_n underneath each vector element:

$$\big(\underset{i_0}{0}, \underset{i_1}{0}, \underset{i_2}{0}, \underset{i_3}{1}, \underset{i_4}{0}, \underset{i_5}{0}, \underset{i_6}{0}, \underset{i_7}{0}\big)^T.$$

Interpreting the rightmost qubit in $|011\rangle$ as the least significant bit with a classical bit value of $1 \cdot 2^0$, the middle qubit with a classical bit value of $1 \cdot 2^1$, and the leftmost qubit with a classical value of $0 \cdot 2^2$, the state $|011\rangle$ corresponds to the decimal value 3 or state $|3\rangle$.

When we store and manipulate state vectors in Python, we index the vector as an array from left to right, from 0 to $n - 1$. For state $|011\rangle$, the element at index i_3 will be set to 1. Using this simple binary addressing scheme, the amplitudes for each basis state in the state vector can be quickly found. Note that the tensor product representation of this 3-qubit state contains the amplitudes for all eight possible states, but seven states have an amplitude of 0. This already hints at a potentially more efficient sparse representation, which we explore in Section 3.9.

We add a few helper functions to support the conversion back and forth between numbers and their bit representations. Note that you can use the array slice operator `[::-1]` to reverse the order of elements in a Python list.

Find the code
In file src/lib/helper.py

```
def bits2val(bits: List[int]) -> int:
  return sum(v * (1 << (len(bits)-i-1)) for i, v in enumerate(bits))

def val2bits(val: int, nbits: int):
  return [int(c) for c in format(val, '0{}b'.format(nbits))]
```

To iterate over all bit strings of a given length `nbits`, we use the following function (note the use of Python's `yield` construct, which allows usage of this function in Python `for` loops):

```
def bitprod(nbits: int) -> Iterable[int]:
  for bits in itertools.product([0, 1], repeat=nbits):
    yield bits
```

Binary bits can also be interpreted as *binary fractions*. For example, using big-endian convention, for individual qubits x_i in state

$$|\psi\rangle = |x_0\, x_1\, \cdots x_{n-2}\, x_{n-1}\rangle,$$

we introduce this big-endian notation, where the most significant fractional part comes first:

$$\begin{aligned}
\frac{x_0}{2^1} &= x_0\frac{1}{2^1} = 0.x_0, \\
\frac{x_0}{2^1} + \frac{x_1}{2^2} &= x_0\frac{1}{2^1} + x_1\frac{1}{2^2} = 0.x_0x_1, \\
\frac{x_0}{2^1} + \frac{x_1}{2^2} + \frac{x_2}{2^3} &= x_0\frac{1}{2^1} + x_1\frac{1}{2^2} + x_2\frac{1}{2^3} = 0.x_0x_1x_2, \\
&\vdots
\end{aligned}$$

In little-endian notation, x_0 would be the least significant fractional part of the binary fraction, as in $0.x_{n-1}\cdots x_1x_0$. The function `bits2frac` computes the fraction for a given big-endian string:

```
def bits2frac(bits: Iterable) -> float:
  return sum(bit * 2 ** (-idx - 1) for idx, bit in enumerate(bits))
```

Here are a few examples and results from printing `val`. You can see how bits 0 and 1 are interpreted as $2^{-1} = 0.5$ and $2^{-2} = 0.25$, respectively:

```
    val = helper.bits2frac((0,))
>> 0
    val = helper.bits2frac((1,))
>> 0.5
    val = helper.bits2frac((0, 1))
>> 0.25
    val = helper.bits2frac((1, 0))
```

```
>>0.5
    val = helper.bits2frac((1, 1))
>>0.75
```

To approximate a given floating point number $x < 1.0$ with binary fractions, we add the routine `frac2bits`:

```
def frac2bits(val: float, nbits: int):
  assert val < 1.0, 'frac2bits: value must be strictly < 1.0'
  res = []
  while nbits:
    nbits -= 1
    val *= 2
    res.append(int(val))
    val -= int(val)
  return res
```

2.4.4 State Member Functions

Now that we understand the order of qubits and state vector indexing, we can add functions to `State` to return the amplitude and probability of a given state. The probability is a real number, but we still have to convert it to an actual real number with `np.real()` to avoid a type conflict and a warning message.

Find the code
In file src/lib/state.py

```
def ampl(self, *bits) -> np.complexfloating:
  return self[helper.bits2val(bits)]

def prob(self, *bits) -> float:
  amplitude = self.ampl(*bits)
  return np.real(amplitude.conj() * amplitude)
```

We use Python parameters (such as `bits` above) that are decorated with the asterisk `*`. This means a variable number of arguments is allowed. In Python parlance, the parameters are *packed* into a *tuple*. To unpack the tuple, you have to prefix any accesses with a `*` again, as shown in the function `prob` above. As an example, for a four-qubit state, you can get the amplitude and probability for the state $|1011\rangle$ in the following way:

```
  psi.ampl(1, 0, 1, 1)
  psi.prob(1, 0, 1, 1)
```

The following snippet iterates over all possible states and prints the probabilities for each state:

```
  for bits in helper.bitprod(4):
    print(psi.prob(*bits))
```

Given a state, we often want to know the number of qubits it consists of. We could maintain this length as an extra member variable to `State`, but it is easy to compute from the length of the state vector (which is already maintained by `numpy`). Because this property is required for all classes derived from `Tensor` (e.g., `States` and `Operators`), we add the `nbits` property to the `Tensor` base class:

```
  @property
  def nbits(self) -> int:
    return int(math.log2(self.shape[0]))
```

When developing an algorithm, we often want to find the state with the highest probability. For this, to find the largest (absolute) element in an array and its index, we add the convenience function `maxprob`, which uses the clever `numpy` function `argmax()`. Then we use a helper function to convert the found index into a binary bit string:

```
  def maxprob(self) -> (List[float], float):
    idx = np.abs(self).argmax()
    maxprob = np.real(self[idx].conj() * self[idx])
    maxbits = helper.val2bits(idx, self.nbits)
    return maxbits, maxprob
```

It can become necessary to renormalize a state vector. This is done with the `normalize` member function. This function asserts that the dot product is not equal to 0 to avoid a division by zero exception:

```
  def normalize(self):
    dprod = np.conj(self) @ self
    assert not dprod.is_close(0.0), 'Normalizing to 0-probability state'
    self /= np.sqrt(np.real(dprod))  # modify object in place.
    return self
```

The *phase* of a basis state is the angle obtained by converting the state's complex amplitude to polar coordinates. We only use this during printouts and convert the phase to degrees here:

```
def phase(self, *bits) -> float:
  amplitude = self.ampl(*bits)
  return math.degrees(cmath.phase(amplitude))
```

Finally, to assist in debugging, it is always helpful to have a function `dump()` that lists all relevant state information. By default, this function only prints the basis states

that have a nonzero probability (set parameter `prob_only` to `False` to see all basis states). An optional description string can also be passed in:

```
def dump(self, desc: str = None, prob_only: bool = True) -> None:
[...]
```

The output from the dumper may look like the following, showing all basis vectors with nonzero probability:

```
|001> (|1>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|011> (|3>):  ampl: +0.35+0.35j prob: 0.25 Phase:  45.0
|101> (|5>):  ampl: +0.00+0.50j prob: 0.25 Phase:  90.0
|111> (|7>):  ampl: -0.35+0.35j prob: 0.25 Phase: 135.0
```

2.4.5 State Constructors

Using the methods described so far, let us define standard constructors to create composite states. The first two functions are for states consisting of only $|0\rangle$ and $|1\rangle$. The state vector for these vectors is all zeros, except for a 1 at index 0 for a state of all $|0\rangle$, or a 1 at the last index for a state consisting of all $|1\rangle$:

```
def zeros_or_ones(d: int = 1, idx: int = 0) -> State:
  assert d > 0, 'Need to specify at least 1 qubit'
  t = np.zeros(2**d, dtype=tensor.tensor_type())
  t[idx] = 1
  return State(t)

def zeros(d: int = 1) -> State:
  return zeros_or_ones(d, 0)

def ones(d: int = 1) -> State:
  return zeros_or_ones(d, 2**d - 1)
```

The function `bitstring` allows the construction of states from a defined sequence of $|0\rangle$ and $|1\rangle$ states. As noted above, the most significant bit comes first:

```
def bitstring(*bits) -> State:
  arr = np.asarray(bits)
  assert len(arr) > 0, 'Need to specify at least 1 qubit'
  assert ((arr == 1) | (arr == 0)).all(), 'Bits must be 0 or 1'

  t = np.zeros(1 << len(bits), dtype=tensor.tensor_type())
  t[helper.bits2val(bits)] = 1
  return State(t)
```

Sometimes, especially for testing or benchmarking, we want to generate a tensor product of n random $|0\rangle$ and $|1\rangle$ states:

```
def rand_bits(n: int) -> State:
  bits = [random.randint(0, 1) for _ in range(n)]
  return bitstring(*bits)
```

While the canonical single-qubit states $|0\rangle$ and $|1\rangle$ are used often, we should *not* define global variables for them.[13] Global variables are bad style. Must. Resist. Temptation.

Can we initialize a state with a given normalized vector? Yes, we can. We will see this pattern later in Section 11.2.1 on phase estimation, where we directly initialize a state as the eigenvector of a unitary matrix:

```
  umat = scipy.stats.unitary_group.rvs(2**nbits)
  eigvals, eigvecs = np.linalg.eig(umat)
  psi = state.State(eigvecs[:, 0])
```

2.5 Representing States as Matrices

In Section 4.2 we will learn that there are questions in quantum computing that cannot be answered by representing states as simple vectors. To address this, we will introduce the so-called *density matrix formalism* to represent states as matrices. We briefly mention these matrices here, as we will use one of their properties later in this chapter.

For a given state $|\psi\rangle$, we can construct its density matrix by computing the outer product of a state with itself. For convenience, we add the function `density()` to our `State` class. Typically, the Greek letter ρ ("rho") is used to denote a density matrix as $\rho = |\psi\rangle\langle\psi|$:

```
def density(self) -> tensor.Tensor:
  return tensor.Tensor(np.outer(self, self.conj()))
```

Given how this matrix is being constructed, the diagonal elements are the probabilities of measuring one of the basis states for $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$:

$$|\psi\rangle\langle\psi| = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}\begin{pmatrix}\alpha^* & \beta^*\end{pmatrix} = \begin{pmatrix}\boxed{\alpha\alpha^*} & \alpha\beta^*\\ \beta\alpha^* & \boxed{\beta\beta^*}\end{pmatrix}.$$

For a pure state, the vector and matrix representations contain the same information. This density matrix has a rank[14] of 1. Its trace must also be 1 because it represents the sum of the probabilities. We will discuss density matrices in more detail in Section 4.2.

[13] As some commercial systems do. Tsk tsk tsk.

[14] See also http://en.wikipedia.org/wiki/Rank_(linear_algebra). The rank of a matrix is the maximal number of its linearly independent columns.

2.6 Operators

Now that we have learned about qubits and states, how are these states *modified* in quantum computing? Classical bits are manipulated through logic gates such as AND, OR, XOR, and NAND. In quantum computing, qubits and states are manipulated with unitary matrices that we call *operators*. It seems appropriate to think of operators as the Instruction Set Architecture (ISA) of a quantum computer. It is a different ISA from that of a typical classical computer, but it is nonetheless an ISA that enables computation. This section discusses operators, their structure, properties, and how to apply them to states.

2.6.1 Unitary Operators

Any unitary matrix of dimension 2^n can be considered a quantum operator. Operators are also called *gates*, in analogy to classical logic gates. Unitary matrices are *norm preserving*; that is, when multiplied with a state vector, they do not change the modulus of the vector. A state vector represents probabilities as probability amplitudes. Applying an operator to this state might change the amplitudes of individual states but must not change the fact that all probabilities must still add up to 1. This is important enough to warrant a brief proof.

Proof To show that a unitary U is norm-preserving, we need to show that $\langle Uv|Uw\rangle = \langle v|w\rangle$. This is to show that if U preserves the structure of the inner product, it must also preserve the norm:

$$\langle Uv|Uw\rangle = (v^\dagger U^\dagger)(Uw) = v^\dagger(U^\dagger U)w.$$

Now, $v^\dagger(U^\dagger U)w = v^\dagger w = \langle v,w\rangle$ implies that $(U^\dagger U) = I$. Any operator that preserves the norm must be unitary. □

An example of a single-qubit unitary gate is the Pauli X gate which we describe in Section 2.7.2. It swaps the probability amplitudes of a qubit:

$$X|\psi\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}.$$

We detail many standard gates later in this section. Note that because $UU^\dagger = I$, unitary matrices are necessarily invertible simply by using the conjugate transpose. Hermitian matrices, on the other hand, are not necessarily unitary. In Section 2.13, we will see that the Hermitian operators used for measurements are neither unitary nor reversible.

2.6.2 Base Class

Since operators are matrices, we derive them from the `Tensor` base class and inherit the `__repr__` and `__str__` functions from the underlying `numpy` array data structure. We also add a convenience function to compute the adjoint. A simple `dump` function allows us to print the state with a given number of digits using the underlying functionality of `numpy`.

Find the code
In file src/lib/ops.py

```
class Operator(tensor.Tensor):
  """Operators are represented by square, unitary matrices."""

  def __new__(subtype, input_array, name=None):
    obj = super().__new__(subtype, input_array)
    obj.name = name
    return obj

  def adjoint(self) -> Operator:
    return self.__class__(np.conj(self.transpose()))

  def dump(self, desc = None, digits = 3) -> None:
    np.set_printoptions(precision=digits)
    if desc:
      print(f'{desc} ({self.nbits}-qubit(s) operator)')
    print(self)
```

2.6.3 Operator Application

To apply an operator to a state vector, we have to compute the matrix–vector product between the operator matrix and the state vector. In Python, we define the function call operator `()` for this purpose. For example, to apply a gate `X` to a state `psi`, we simply call `ops.X(psi)`. The `__call__` function wraps the `apply` function, which we define next. For convenience, we will make the call operator accept a state or an operator as its argument.

```
def __call__(self,
             arg: Union[state.State, ops.Operator],
             idx: int = 0) -> state.State:
  return self.apply(arg, idx)
```

In the following, we gradually build up the `apply` function. The initial versions will be fairly incomplete. We apply an operator to a state vector using `numpy`'s matrix multiplication function `np.matmul`:

```
def apply(self,
          arg: Union[state.State, ops.Operator],
          idx: int) -> state.State:
  [...]
  assert isinstance(arg, state.State), 'Error, expected State.')
  [...]
  return state.State(np.matmul(self, arg))
```

We can also apply an operator to another operator. In this case, the application results in a matrix–matrix multiplication. This raises the question of the order of the matrices when multiple operators are applied in sequence. Assume that we have an X gate and a Y gate (to be explained later) that will be applied in sequence. We can write this in Python the following way, where gates are applied to a state and return the updated state:

```
psi_1 = X(psi_0)
psi_2 = Y(psi_1)
```

These are Python assignments, not to be confused with a mathematical notation like $x = y$, which expresses an equivalence. In Python, variables are mutable. We could omit the indices and overwrite a single state variable `psi`.

In function call notation, we write symbols from left to right, but function parameters are evaluated first before the actual invocation of a function. This means that the parameters are applied first:

```
# A function call means that X is applied before Y.
Y(X)
```

If we express the combined operator as a product of matrices, we must *reverse* their order (recall that the operator `@` is the matrix multiply operator in Python):

```
# In a combined operator matrix, X is applied first:
(Y @ X)(psi)
```

This leads to the following (still incomplete) implementation of `apply`, assuming that the sizes of the operator and the state vector match:

```
def apply(self,
          arg: Union[state.State, ops.Operator],
          idx: int) -> Union[state.State, ops.Operator]:
  if isinstance(arg, Operator):
    assert self.nbits == arg.nbits, 'Mismatched dimensions.'
    return arg @ self

  assert isinstance(arg, state.State), 'Error, expected State.'
  # Note the reversed parameters.
  return state.State(np.matmul(self, arg))
```

2.6.4 Multiple Qubits

The code above makes it possible to apply a gate to a single qubit. How does this work if we have a state of two or more qubits and want to apply a 2×2 gate to just

one qubit in the tensor product? The key property of the tensor product that enables handling this case is Equation (1.3), replicated here:

$$(A \otimes B)(|a\rangle \otimes |b\rangle) = A\,|a\rangle \otimes B\,|b\rangle\,.$$

We can utilize this equation by using the identity gate I (see also Section 2.7.1), since applying I to a qubit leaves the qubit intact:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \text{with} \quad I\,|\psi\rangle = I \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = |\psi\rangle\,.$$

As an example, for a given three-qubit state, Equation (1.3) allows us to apply the X gate (discussed earlier) to the second qubit *only* by tensoring together the identity I with the X gate and another identity I to obtain an 8×8 operator matrix:

```
psi = state.bitstring(0, 0, 0)
op = ops.Identity() * ops.PauliX() * ops.Identity()
psi = op(psi)
psi.dump()
```

When interpreting the state $|\psi_0\rangle = |0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$ as the binary number 0 (recall that by our big-endian convention, the least significant bit is to the right), element 0 of the state vector of 8 elements should contain the value 1.0, which we can confirm by dumping the state:

```
[1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0]
```

Applying the X gate to qubit 1 in this way

$$|\psi_1\rangle = (I \otimes X \otimes I)\,|\psi_0\rangle$$

becomes, again according to Equation (1.3),

$$|\psi_1\rangle = I|0\rangle \otimes X|0\rangle \otimes I|0\rangle.$$

Since we are in the computational basis,[15] the X gate flips the probability amplitudes. Another way to say this colloquially is that it flips a state from $|0\rangle$ to $|1\rangle$ (or from $|1\rangle$ to $|0\rangle$). As a result, we managed to apply the gate X to qubit 1 only and the modified state $|\psi_1\rangle$ becomes

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

Interpreting $|010\rangle$ as the binary number 2, we should find the value 1.0 in the state vector at index 2, and indeed, there it is:

```
[0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0]
```

[15] Recall that in the computational basis, we represent $|0\rangle$ as $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle$ as $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

To apply multiple operators in sequence, their individual expanded operators can be multiplied to build a single combined operator. For example, to apply the X gate to qubit 1 and the Y gate to qubit 2, we write the following:

```
psi = state.bitstring(0, 0, 0)
opx = ops.Identity() * ops.PauliX() * ops.Identity()
opy = ops.Identity() * ops.Identity() * ops.PauliY()
big_op = opy(opx)
psi = big_op(psi)
```

We can use a shorthand notation for this. To indicate that gate A should be applied to a qubit at a specific index i, we write A_i. This notation means that this operator is padded on both sides with identity matrices. For the example above, to apply the X gate to qubit 1 and the Y gate to qubit 2, we write $X_1 Y_2$.

Of course, in terms of performance, building the full combined operator up front for n qubits can be the worst possible case, as we have to perform full matrix multiplication with matrices of size $(2^n)^2$. Matrix multiplication is of cubic[16] complexity $\mathcal{O}(n^3)$. Since a matrix–vector product is of complexity $\mathcal{O}(n^2)$, it can be faster to apply the gates individually, depending on the number of gates. In this particular example, instead of applying the gates one by one:

```
psi = state.bitstring(0, 0, 0)
opx = ops.Identity() * ops.PauliX() * ops.Identity()
psi = opx(psi)
opy = ops.Identity() * ops.Identity() * ops.PauliY()
psi = opy(psi)
```

We could have simply combined the gates in one step:

```
psi = state.bitstring(0, 0, 0)
opxy = ops.Identity() * ops.PauliX() * ops.PauliY()
psi = opxy(psi)
```

2.6.5 Operator Padding

Having to "pad" operators with identity matrices on the left and right is annoying and error-prone. It is more convenient to apply an operator to a qubit at index `idx` and let the infrastructure do the rest for us. This is what *operator padding* does, which we will implement next. To apply a given gate, say the X gate, to a state `psi` at a given qubit index `idx`, we write:

```
X = ops.PauliX()
psi = X(psi, idx)
```

[16] This is an approximation to make a point, which we will use in several places. More efficient algorithms are known, such as the Coppersmith–Winograd algorithm with complexity $\mathcal{O}(2^{2.3752477})$.

To achieve this, we augment the function call operator for `Operator`. If an index is provided as a parameter, we pad the operator up to this index with identity matrices. Then, we compute the size of the given operator, which can be larger than 2×2, and if the resulting matrix dimension is still smaller than the state to which it is applied, we pad it further with identity matrices. In the above example, instead of writing:

```
psi = state.bitstring(0, 0, 0)
opx = ops.Identity() * ops.PauliX() * ops.Identity()
psi = opx(psi)
```

We can now write the more compact code:

```
psi = state.bitstring(0, 0, 0)
psi = ops.PauliX()(psi, 1)
```

This syntax may need to be clarified. The first pair of parentheses to `PauliX()` returns a 2×2 `Operator` object. The parentheses `(psi, 1)` are parameters passed to the operator's function call operator `__call__`, which delegates to the `apply` function. This is where the automatic padding magic happens.

We can now finalize the implementation of `apply`:

```
def apply(self,
          arg: Union[state.State, ops.Operator],
          idx: int) -> Union[state.State, ops.Operator]:
  if isinstance(arg, Operator):
    arg_bits = arg.nbits
    if idx > 0:
      arg = Identity().kpow(idx) * arg
    if self.nbits > arg.nbits:
      arg = arg * Identity().kpow(self.nbits - idx - arg_bits)
    assert self.nbits == arg.nbits, 'Mismatched dimensions.'
    return arg @ self

  assert isinstance(arg, state.State), 'Error, expected State.'
  op = self
  if idx > 0:
    op = Identity().kpow(idx)  * op
  if arg.nbits - idx - self.nbits > 0:
    op = op * Identity().kpow(arg.nbits - idx - self.nbits)

  return state.State(np.matmul(op, arg))
```

2.7 Single-Qubit Gates

In this section, we list single-qubit gates that are commonly used in quantum computing. These gates are analogous to logic gates seen in classical computing in that

a full understanding of the basic gates is needed to construct more sophisticated circuits. However, while quantum gates share similarities with their classical computing counterparts, their functions are quite different.

We start with simple gates and then discuss the more complicated roots and rotations, including the important Hadamard gate, which maps computational basis states to superpositions of basis states. For each gate, we define a constructor function and allow passing a *dimension parameter* `d`, which allows the construction of a multi-qubit operator from the same underlying single-qubit gate. As an example, to compute the tensor product of two identity matrices and a Y gate, we write

$$Y_2 = I \otimes I \otimes Y = I^{\otimes 2} \otimes Y.$$

Note again the subscript in Y_2, which indicates that the Y gate should only be applied to qubit 2. With this, we have two ways to apply the identity gates:

```
# Explicit way:
y2 = ops.Identity() * ops.Identity() * ops.PauliY()
# Compact way:
y2 = ops.Identity(2) * ops.PauliY()
```

2.7.1 Identity Gate

The general identity matrix is a square matrix with 1s on its diagonal and 0s everywhere else. As a single-qubit operator, the operator I is the matrix

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Applying this gate to a state leaves the state intact:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

It is easy to construct. We will use the following code pattern for most gate constructor functions:

```
def Identity(d: int = 1) -> Operator:
  return Operator([[1.0, 0.0], [0.0, 1.0]], 'Id').kpow(d)
```

2.7.2 Pauli Matrices

The three Pauli matrices play an essential role in quantum computing and have many uses, some of which we will discover as we go along. Pauli matrices are usually denoted with the greek σ ("sigma") as $\sigma_x, \sigma_y, \sigma_z$, alternatively as $\sigma_1, \sigma_2, \sigma_3$, or simply

as X, Y, and Z, which is the notation we will use most often. Sometimes, the identity matrix I is added as a first Pauli matrix σ_0:

$$\sigma_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

The Pauli gate σ_x is also known as the X gate, the quantum Not gate, or just X for short. It is called a Not gate because it seemingly "flips" basis states in the following way:[17]

$$X|0\rangle = |1\rangle \quad \text{and} \quad X|1\rangle = |0\rangle.$$

This can be confusing, especially for beginners. To clarify, the computational basis states remain unmodified as they represent physical states. The X gate only swaps the probability amplitudes:

$$X|\psi\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}.$$

It changes $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to $|\psi'\rangle = \beta|0\rangle + \alpha|1\rangle$, for all possible values of α and β, including the cases where α or β is 0 and the other is 1. In code:

```
def PauliX(d: int = 1) -> Operator:
  return Operator([[0.0, 1.0], [1.0, 0.0]], 'X').kpow(d)
```

The Z gate is also known as the phase-flip gate. It inverts the sign of the qubit's β factor:

$$Z|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ -\beta \end{pmatrix}.$$

This gate changes the state from $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to $|\psi'\rangle = \alpha|0\rangle - \beta|1\rangle$. Again, the basis states remain unchanged; only the *sign* of the coefficient β changes. The choice of basis matters because, for example, in the Hadamard basis, the Z gate acts as a bit-flip gate, with $Z|+\rangle = |-\rangle$ and $Z|-\rangle = |+\rangle$. In code:

```
def PauliZ(d: int = 1) -> Operator:
  return Operator([[1.0, 0.0], [0.0, -1.0]], 'Z').kpow(d)
```

The action of the Pauli Y gate on a state $|\psi\rangle$ is

$$Y|\psi\rangle = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} -i\beta \\ i\alpha \end{pmatrix}.$$

```
def PauliY(d: int = 1) -> Operator:
  return Operator([[0.0, -1.0j], [1.0j, 0.0]], 'Y').kpow(d)
```

[17] Again, in the computational basis.

Applying the Y gate to the standard basis states leads to both a bit flip and a phase flip:

$$Y|0\rangle = iXZ|0\rangle = iX|0\rangle = i|1\rangle,$$
$$Y|1\rangle = iXZ|1\rangle = -iX|1\rangle = -i|0\rangle.$$

Some useful mathematical properties of the Paul matrices are as follows. Pauli matrices are Hermitian with eigenvalues $+1$ and -1. Their trace is $\operatorname{tr}\sigma_i = 0$ and the determinants (for X, Y, and Z) are $\det\sigma_i = -1$. On the Bloch sphere, the X gate rotates the state about the x-axis, the Y gate rotates the state about the y-axis, and the Z gate rotates the state about the z-axis, all by 180° (π in radians, or half a circle).

Together with the identity matrix, the Pauli matrices form a basis for the vector space of 2×2 Hermitian matrices.[18] *Any* 2×2 Hermitian matrix M can be constructed using a linear combination of Pauli matrices as

$$M = \frac{I + xX + yY + zZ}{2}. \tag{2.5}$$

Pauli matrices are *involutory*:

$$II = XX = YY = ZZ = I.$$

We can use the basis states to construct the Pauli matrices as follows:

$$X = |+\rangle\langle +| - |-\rangle\langle -|,$$
$$Y = |+y\rangle\langle -y| - |-y\rangle\langle +y|,$$
$$Z = |0\rangle\langle 0| - |1\rangle\langle 1|.$$

The effects of the Pauli matrices on the computational and Hadamard basis states can be summarized as follows:

$$X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle, \quad X|+\rangle = |+\rangle, \quad X|-\rangle = -|-\rangle,$$
$$Z|0\rangle = |0\rangle, \quad Z|1\rangle = -|1\rangle, \quad Z|+\rangle = |-\rangle, \quad Z|-\rangle = |+\rangle,$$
$$Y|0\rangle = i|1\rangle, \quad Y|1\rangle = -i|0\rangle, \quad Y|+\rangle = -i|-\rangle, \quad Y|-\rangle = i|+\rangle.$$

2.7.3 Bloch Sphere Coordinates

With Equation (2.5) we can compute the Cartesian coordinates for a given state $|\psi\rangle$ on the Bloch sphere.[19] The Pauli matrices form a basis for the space of 2×2 Hermitian matrices. The outer product $\rho = |\psi\rangle\langle\psi|$ is such a Hermitian matrix. Hence we can write

$$\rho = \frac{I + xX + yY + zZ}{2} = \frac{1}{2}\begin{pmatrix} 1+z & x-iy \\ x+iy & 1-z \end{pmatrix}. \tag{2.6}$$

[18] To be precise, *any* 2×2 matrix can be constructed from the Pauli matrices. However, if the matrix is Hermitian, the coefficients to the Pauli matrices are necessarily real.

[19] Found in http://quantumcomputing.stackexchange.com/a/17180.

If we think of ρ as a matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, then

$$2a = 1 + z \quad \text{and} \quad 2c = x + iy.$$

Consequently, $x = 2\,\mathrm{Re}(c)$, $y = 2\,\mathrm{Im}(c)$, and $z = 2a - 1$. In code, we can do this in two simple steps:

1. Compute the outer product of the state $|\psi\rangle$ with itself as $\rho = |\psi\rangle\langle\psi|$. This matrix ρ is a special case of a *density matrix*, as discussed in Section 2.5.
2. Apply the helper function `density_to_cartesian(rho)`, shown below, which returns the corresponding coordinates x, y, and z.

PY

Find the code
In file src/lib/helper.py

```
def density_to_cartesian(rho: np.ndarray) -> Tuple[float, float, float]:
  a = rho[0, 0]
  c = rho[1, 0]
  x = 2.0 * c.real
  y = 2.0 * c.imag
  z = 2.0 * a - 1.0
  return np.real(x), np.real(y), np.real(z)
```

2.7.4 Rotations

When we say that we apply a *rotation* we mean applying a unitary operator on a quantum state or, equivalently, a rotation operator on a Bloch vector for a single-qubit state. We define rotations about the orthogonal axes x, y, and z, with help of the Pauli matrices as

$$R_x(\theta) = e^{-i\frac{\theta}{2}X},$$
$$R_y(\theta) = e^{-i\frac{\theta}{2}Y},$$
$$R_z(\theta) = e^{-i\frac{\theta}{2}Z}.$$

While seeing an exponential function with a matrix in the exponent may seem unfamiliar, the process is not overly complex. Here, we provide a proof for the following statement, which explains the mechanics of matrix exponentiation through a simple power series expansion.

THEOREM: *If an operator A is involutory (which means it is its inverse), then*

$$e^{i\theta A} = \cos(\theta)I + i\sin(\theta)A.$$

Proof The exponential function e^A has the power series expansion

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \frac{A^4}{4!} + \cdots .$$

For the specific function $e^{i\theta A}$, this becomes

$$e^{i\theta A} = I + i\theta A - \frac{(\theta A)^2}{2!} - i\frac{(\theta A)^3}{3!} + \frac{(\theta A)^4}{4!} + \cdots .$$

If the operator is involutory and satisfies $A^2 = I$, then we can reorder the terms into the Taylor series for $\sin(\cdot)$ and $\cos(\cdot)$:

$$\begin{aligned} e^{i\theta A} &= I + i\theta A - \frac{\theta^2 I}{2!} - i\frac{\theta^3 A}{3!} + \frac{\theta^4 I}{4!} + \cdots \\ &= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \cdots\right) I + i\left(\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \cdots\right) A \\ &= \cos\theta I + i \sin\theta A. \end{aligned} \tag{2.7}$$

□

Since the Pauli matrices are involutory, with $II = XX = YY = ZZ = I$, we can write:

$$\begin{aligned} R_x(\theta) = e^{-i\frac{\theta}{2}X} &= \cos\left(\frac{\theta}{2}\right) I - i \sin\left(\frac{\theta}{2}\right) X \\ &= \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}, \end{aligned} \tag{2.8}$$

$$\begin{aligned} R_y(\theta) = e^{-i\frac{\theta}{2}Y} &= \cos\left(\frac{\theta}{2}\right) I - i \sin\left(\frac{\theta}{2}\right) Y \\ &= \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}, \end{aligned} \tag{2.9}$$

$$\begin{aligned} R_z(\theta) = e^{-i\frac{\theta}{2}Z} &= \cos\left(\frac{\theta}{2}\right) I - i \sin\left(\frac{\theta}{2}\right) Z \\ &= \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}. \end{aligned} \tag{2.10}$$

We will learn more about rotations and how to compute the axis, coordinates, and rotation angles in Section 9.4.3. For now, we implement rotations about the standard Cartesian x, y, and z axes with these functions:

```
def Rotation(vparm: List[float], theta: float) -> Operator:
  v = np.asarray(vparm)
  [...] # error handling
  return Operator(np.cos(theta / 2) * Identity() - 1j * np.sin(theta / 2)
      * (v[0] * PauliX() + v[1] * PauliY() + v[2] * PauliZ()))

def RotationX(theta: float) -> Operator:
  return Rotation([1.0, 0.0, 0.0], theta, 'Rx')

def RotationY(theta: float) -> Operator:
  return Rotation([0.0, 1.0, 0.0], theta, 'Ry')
```

```
def RotationZ(theta: float) -> Operator:
  return Rotation([0.0, 0.0, 1.0], theta, 'Rz')
```

In general, rotations are defined about *any* arbitrary axis $\hat{n} = (n_1, n_2, n_3)$ as

$$R_{\hat{n}} = exp\left(-i\theta\hat{n}\frac{1}{2}\hat{\sigma}\right).$$

We will see an advanced example of this in Section 9.4 on the Solovay–Kitaev algorithm for gate approximation.

2.7.5 Hadamard Gate

As the final rotation gate, we now discuss the all-important Hadamard gate, which is defined as

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

If you look at this definition carefully, you can see that it is made up of the X and Z gates, representing a rotation about the axis $(\vec{x}+\vec{z})/\sqrt{2}$. Let us apply this gate to $|0\rangle$ and $|1\rangle$ respectively to obtain

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ 1\end{pmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}},$$

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}0\\ 1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ -1\end{pmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Both results can be stated as the sum or difference of the $|0\rangle$ and $|1\rangle$ bases, scaled by $1/\sqrt{2}$. As we have seen earlier, these basis states are so common they get the symbolic names $|+\rangle$ and $|-\rangle$:

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

The Hadamard gate is important because it maps the computational basis states to an equal superposition. For a general state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the Hadamard operator yields:

$$\begin{aligned} H|\psi\rangle &= H\big(\alpha|0\rangle + \beta|1\rangle\big)\\ &= \alpha H|0\rangle + \beta H|1\rangle\\ &= \alpha\frac{|0\rangle + |1\rangle}{\sqrt{2}} + \beta\frac{|0\rangle - |1\rangle}{\sqrt{2}}\\ &= \alpha\,|+\rangle + \beta\,|-\rangle\\ &= \frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle. \end{aligned}$$

In code, the operator is defined using our standard template:

```
def Hadamard(d: int = 1) -> Operator:
  return Operator(1 / np.sqrt(2) * np.array([[1.0, 1.0], [1.0, -1.0]]),
                  'H').kpow(d)
```

A Hadamard gate is its own inverse with $H = H^{-1}$ and $HH = I$. It is Hermitian and also involutory, just like the Pauli matrices:

$$HH = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\frac{1}{2}\begin{pmatrix}2 & 0\\ 0 & 2\end{pmatrix} = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = I.$$

A common operation is the application of Hadamard gates to several adjacent qubits. If those qubits were all in the $|0\rangle$ state, the resulting state becomes an equal superposition with amplitudes $\frac{1}{\sqrt{2^n}}$:

$$H^{\otimes n}\,|0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle\,.$$

This construction is used in many of our algorithms and examples. Let us spell it out explicitly for two and three qubits, which also illustrates some of the notations we will be using:

$$(H\otimes H)\big(\,|0\rangle\otimes|0\rangle\,\big) = \frac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big),$$

$$\begin{aligned}&(H\otimes H\otimes H)\big(\,|0\rangle\otimes|0\rangle\otimes|0\rangle\,\big)\\ &= \frac{1}{\sqrt{2^3}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle\big)\\ &= \frac{1}{\sqrt{2^3}}\big(\,|0\rangle + |1\rangle + |2\rangle + |3\rangle + |4\rangle + |5\rangle + |6\rangle + |7\rangle\big)\\ &= \frac{1}{\sqrt{2^3}}\sum_{x=0}^{7}|x\rangle \quad = \frac{1}{\sqrt{2^3}}\sum_{x\in\{0,1\}^3}|x\rangle.\end{aligned}$$

2.7.6 Phase Gate

The *phase gate*, also called the *S* gate, *P* gate, or *Z90* gate, represents a phase of 90° around the *z*-axis for the $|1\rangle$ part of a qubit. Because this rotation is quite common, it gets its own name:

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

The phase gate can be derived using Euler's formula for the angle $\phi = \pi/2$:

$$\begin{aligned}e^{i\phi} &= \cos(\phi) + i\sin(\phi),\\ e^{i\pi/2} &= \cos(\pi/2) + i\sin(\pi/2) = i.\end{aligned}$$

In code, we construct this gate using our standard recipe:

```
def Phase(d: int = 1) -> Operator:
  return Operator([[1.0, 0.0], [0.0, 1.0j]], 'S').kpow(d)

def Sgate(d: int = 1) -> Operator:
  return Phase(d)
```

Note the effect this gate has on the basis states $|0\rangle$ and $|1\rangle$, the gate only affects the $|1\rangle$ part of the qubit state:

$$S|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle \quad \text{and} \quad S|1\rangle = \begin{pmatrix} 0 \\ i \end{pmatrix} = i|1\rangle .$$

You can spot a potential source of errors: the direction of rotations, especially when porting code from other infrastructures that might interpret angle directions differently. For much of this text, we are shielded from this problem. However, it may be one of the first things to look for when the results do not meet expectations.

Finally, remember the Z gate and how similar it is to the phase gate? The relationship is easy to see — applying two phase gates, each affecting a rotation of $\pi/2$, yields a rotation of π, which we get from applying the Z gate:

$$S^2 = SS = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = Z.$$

2.7.7 Flexible Phase Gates

There are other, more flexible versions of phase gates. The general $U_1(\lambda)$ gate is also known as the *phase shift* or *phase kick* gate:

$$U_1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}.$$

It has no restrictions on the phase to use and its implementation is straightforward:

```
def U1(lam: float, d: int = 1) -> Operator:
  return Operator([(1.0, 0.0),
                   (0.0, cmath.exp(1j * lam))], 'U1').kpow(d)
```

The derived *discrete phase gate* (also known as the R_k gate) performs rotations about the z-axis by fractional powers of $2\pi/2^k$ for $k > 0$, such as π, $\pi/2$, $\pi/4$, and so on:

$$R_k(k) = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}.$$

```
def Rk(k: int, d: int = 1) -> Operator:
  return U1(2 * math.pi / (2**k)).kpow(d)
```

For integer powers of 2, the relationship between R_k and U_1 is

$$R_k(n) = U_1\left(2\pi/2^n\right).$$

Some of the named gates are just special cases of R_k (and therefore of U_1). In particular, the identity gate I, the Z gate, the S gate, and the T gate (which we define in Section 2.7.9 below) can be constructed with these flexible phase gates. This test code may help to clarify:

```
def test_rk(self):
  self.assertTrue(ops.Rk(0).is_close(ops.Identity()))
  self.assertTrue(ops.Rk(1).is_close(ops.PauliZ()))
  self.assertTrue(ops.Rk(2).is_close(ops.Sgate()))
  self.assertTrue(ops.Rk(3).is_close(ops.Tgate()))
```

2.7.8 U_3 gate

Physical quantum computers may implement other types of gates. IBM machines specify the general U_3 gate with real angles θ, ϕ, and λ:

$$U_3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{pmatrix}.$$

This gate is quite versatile, as it can be used to construct several other standard gates.[20] The likely simplest form generates the identity gate as

$$U_3(0,0,0) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I.$$

To construct the flexible phase gate U_1 (with which we can generate Z, S, and T gates), we set

$$U_3(0,0,\lambda) = \begin{pmatrix} \cos(0) & -e^{-i\lambda}\sin(0) \\ e^{0i}\sin(0) & e^{i(\lambda+0)}\cos(0) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix} = U_1.$$

To make an X gate, we set $\theta = \pi$, resulting in $\cos(\theta/2) = 0$ and $\sin(\theta/2) = 1$:

$$U_3(\pi,\phi,\lambda) = \begin{pmatrix} 0 & -e^{i\lambda} \\ e^{i\phi} & 0 \end{pmatrix}. \tag{2.11}$$

The lower left term must be 1, which we get with $\phi = 0$. The upper right term must be 1 as well, which we get with $\lambda = \pi$:

$$U_3(\pi,0,\pi) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = X.$$

Using Equation (2.11) we can derive the Pauli Y gate as

$$U_3(\pi,\frac{\pi}{2},\frac{\pi}{2}) = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = Y.$$

[20] Found in http://qiskit.org/textbook/ch-states/single-qubit-gates.html, section 7.

We can create more complex gates. For example, to construct a Hadamard gate, we set

$$U_3(\theta = \frac{\pi}{2}, \phi = 0, \lambda = \pi) = \begin{pmatrix} \cos(\pi/4) & -(-1)\sin(\pi/4) \\ 1\sin(\pi/4) & -1\cos(\pi/4) \end{pmatrix}.$$

With $\cos\frac{\pi}{4} = \sin\frac{\pi}{4} = \frac{1}{\sqrt{2}}$, the result is a Hadamard gate:

$$U_3(\frac{\pi}{2}, 0, \pi) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = H.$$

We can even construct rotation gates:

$$U_3(\theta, -\frac{\pi}{2}, \frac{\pi}{2}) = R_x(\theta),$$
$$U_3(\theta, 0, 0) = R_y(\theta).$$

Similarly to the other standard gates, we add this constructor to `src/lib/ops.py`:

```
def U3(theta: float, phi: float, lam: float, d: int = 1) -> Operator:
  return Operator(
      [(np.cos(theta / 2),
        -cmath.exp(1j * lam)*np.sin(theta / 2)),
       (cmath.exp(1j * phi)*np.sin(theta / 2),
        cmath.exp(1j * (phi + lam))*np.cos(theta / 2))], 'U3').kpow(d)
```

2.7.9 Square Roots of Gates

What is the square root of a classical NOT gate? There is no such thing. There is no classical gate that, when applied twice, flips a bit. However, it is possible to find a matrix $V = \sqrt{X}$ in quantum computing.

When asked what the root is of 4, the usual answer is 2. However, there are actually two roots, namely 2 and -2. Similarly, matrices can have multiple roots; in particular, if $X = V^2$, then also $X = (-V)^2$. As we shall see later in Section 2.10.2, roots play an important role in the construction of efficient two-qubit gates.

The root of the X gate is the V gate. V is unitary,[21] with $VV^\dagger = I$, but also $V^2 = X$. It can be defined in the following ways[22] (we only implement the first option):

$$V = \sqrt{X} = \frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix} = \frac{1}{\sqrt{2}}e^{i\pi/4}(I - iX),$$
$$= \frac{1}{2}\begin{pmatrix} -1-i & -1+i \\ -1+i & -1-i \end{pmatrix} = -\frac{1}{\sqrt{2}}e^{i\pi/4}(I - iX).$$

```
def Vgate(d: int = 1) -> Operator:
  return Operator(0.5 * np.array([(1 + 1j, 1 - 1j),
                                  (1 - 1j, 1 + 1j)]), 'V').kpow(d)
```

[21] Any root of a unitary gate is also unitary. We omit the proof here.
[22] Found in http://quantumcomputing.stackexchange.com/a/30216.

The root of a rotation by an angle ϕ is a rotation about the same axis, in the same direction, by the angle $\phi/2$. Two half-rotations result in one full rotation. This is obvious from the exponential form

$$\sqrt{e^{i\phi}} = (e^{i\phi})^{1/2} = e^{i\phi/2}.$$

The root of the S gate is called the T gate. The S gate represents a phase of 90°. Consequently, the T gate is equivalent to a 45° phase:

$$T = \sqrt{S} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}.$$

```
def Tgate(d: int = 1) -> Operator:
  return Operator([[1.0, 0.0],
                   [0.0, cmath.exp(cmath.pi * 1j / 4)]]).kpow(d)
```

The T gate is sometimes called the $\pi/8$ gate, which seems counterintuitive since the gate has a factor of $\pi/4$ in it! The name may come from the fact that pulling out a factor makes the gate appear symmetric:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} = e^{i\pi/8} \begin{pmatrix} e^{-i\pi/8} & 0 \\ 0 & e^{i\pi/8} \end{pmatrix}.$$

The root of the Y gate has no special name (which we know of), but is required later in the text, so we introduce it here as `Yroot`. It is defined as

$$Y_{\text{root}} = \sqrt{Y} = \frac{1}{2} \begin{pmatrix} 1+i & -1-i \\ 1+i & 1+i \end{pmatrix},$$

which translates to this code:

```
def Yroot(d: int = 1) -> Operator:
  return Operator(0.5 * np.array([(1 + 1j, -1 - 1j),
                                  (1 + 1j, 1 + 1j)]), 'YRoot').kpow(d)
```

There are other interesting roots, but these are the main ones we will encounter in this text. We can test for correct implementations with code like this:

```
def test_gates_roots(self):
  t = ops.Tgate()
  self.assertTrue(t(t).is_close(ops.Phase()))
  v = ops.Vgate()
  self.assertTrue(v(v).is_close(ops.PauliX()))
  yr = ops.Yroot()
  self.assertTrue(yr(yr).is_close(ops.PauliY()))
```

Finding a root in closed form can be cumbersome. In case of analytical problems, you can use the `scipy` function `sqrtm()` to compute the root of a gate. For this to work, `scipy` must be installed:

```
from scipy.linalg import sqrtm
[...]
computed_yroot = sqrtm(ops.PauliY())
self.assertTrue(ops.Yroot().is_close(computed_yroot))
```

2.7.10 Projection Operators

A projection operator for a given *basis* state, or *projector* for short, is the outer product of a basis state with itself.[23] The term projector comes from the fact that applying a basis state's projector to a given state extracts the amplitude of the basis state. The state is *projected* on the basis state, similar to how the cosine function is a projection of a two-dimensional vector on the x-axis. The projectors for the states $|0\rangle$ and $|1\rangle$ are[24]

$$P_{|0\rangle} = \Pi_{|0\rangle} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},$$

$$P_{|1\rangle} = \Pi_{|1\rangle} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Applying the projector for the $|0\rangle$ state to a random qubit yields the probability amplitude of the qubit being found in the $|0\rangle$ state (and similar for the projector to the $|1\rangle$ state):

$$P_{|0\rangle}|\psi\rangle = |0\rangle\langle 0|(\alpha\,|0\rangle + \beta|1\rangle) = \alpha\,|0\rangle.$$

Projection operators are Hermitian, hence $P = P^\dagger$, but note that the projection operators are *not* unitary or reversible. Their two eigenvalues are 0 and 1. If the basis states of a projection operator are normalized, the projection operator is equal to its square $P = P^2$; it is *idempotent*. We will use this result below in Section 2.13 on measurement. Similar to basis states, two projection operators are *orthogonal* if and only if their product is 0, which means that for each state

$$P_{|0\rangle}P_{|1\rangle}|\psi\rangle = \vec{0}.$$

The sum of all projection operators for any given orthonormal basis $\{\vec{i}\}$ adds up to the identity operator. This is also known as the *completeness relation*. You can try this out, for example, with the Hadamard basis:

$$\sum_{i=0}^{n-1} P_{|i\rangle} = I,$$

$$|+\rangle\langle +| + |-\rangle\langle -| = I.$$

In general, when writing an outer product as $|r\rangle\langle c|$, you can think of this as a two-dimensional index [row,col] into a matrix. This is also called the *outer product representation* of an operator:

[23] Strictly speaking, it does not have to be a basis state, but this is what we will typically use.
[24] Often the symbol Π ("Pi") is used to denote projectors. In this book, we will use a slant P.

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = a|0\rangle\langle 0| + b|0\rangle\langle 1| + c|1\rangle\langle 0| + d|1\rangle\langle 1|.$$

This also works for larger operators. For example, for this two-qubit operator U with just one nonzero element α,

$$U = \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 \end{pmatrix} \end{array},$$

the outer product representation for the single nonzero element α in this operator would be $\alpha|11\rangle\langle 01|$, an index pattern of $|\text{row}\rangle\langle\text{col}|$. For derivations, this representation can be more convenient than having to deal with full matrices. For example, to express the application of the X gate to a qubit, we would write

$$\begin{aligned} X &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|, \\ X&\big(\alpha\,|0\rangle + \beta\,|1\rangle\big) \\ &= \big(|0\rangle\langle 1| + |1\rangle\langle 0|\big)(\alpha\,|0\rangle + \beta\,|1\rangle) \\ &= |0\rangle\langle 1|\alpha\,|0\rangle + |0\rangle\langle 1|\beta\,|1\rangle + |1\rangle\langle 0|\alpha\,|0\rangle + |1\rangle\langle 0|\beta\,|1\rangle \\ &= \alpha|0\rangle\underbrace{\langle 1|0\rangle}_{=0} + \beta|0\rangle\underbrace{\langle 1|1\rangle}_{=1} + \alpha|1\rangle\underbrace{\langle 0|0\rangle}_{=1} + \beta|1\rangle\underbrace{\langle 0|1\rangle}_{=0} \\ &= \beta\,|0\rangle + \alpha|1\rangle. \end{aligned}$$

We add the following helper functions to construct the common n-bits projectors for the basis states $|00\ldots0\rangle$ and $|11\ldots1\rangle$, where we exploit the fact that the resulting matrices have a singular 1 in either the top left or bottom right corner of the matrix:

```
def ZeroProjector(nbits: int) -> Operator:
  zero_projector = np.zeros((2**nbits, 2**nbits))
  zero_projector[0, 0] = 1
  return Operator(zero_projector)

def OneProjector(nbits: int) -> Operator:
  dim = 2**nbits
  zero_projector = np.zeros((dim, dim))
  zero_projector[dim - 1, dim - 1] = 1
  return Operator(zero_projector)
```

At this point, we have made good progress in learning about single-qubit gates and how to construct multi-qubit states. Yet, a key ingredient to computing is still missing: What are the control-flow constructs that are ubiquitous in classical computing? The

quantum equivalents of these constructs are called *controlled gates*, which we will discuss next.

2.8 Controlled Gates

Quantum computing does not have classic control flow with branches around conditionally executed parts of the code. As described earlier, in a quantum circuit, all qubits are active at all times. The quantum analog of control-dependent execution is achieved with *controlled* gates. Controlled gates are always applied but show an effect only under certain conditions. At least two qubits are involved: a controller qubit and a controlled qubit. Note that 2-qubit gates in this form cannot be decomposed into single-qubit gates.

REMARK: *Before we continue, we have to agree on naming (which is hard). Shall we call a controlled not gate a, well, controlled not gate, a controlled-not gate, or Controlled-Not gate, or even a Controlled-Not-gate? Should it be X-gate or X gate?*

We will follow the convention of using uppercase gate names and no hyphens, such as controlled Not gate, X gate, or Hadamard gate. In mathematical notation, gates are referred to by their symbolic names, such as X, Y, and Z.

Let us explain the function of controlled gates by example. Assume that we have two qubits, numbered qubit 0 and qubit 1, and we somehow want qubit 0 to influence the effect of qubit 1. Consider how the following two-qubit controlled Not matrix (abbreviated as *CNOT*, or *CX*), spanning both qubits, operates on combinations of the $|0\rangle$ and $|1\rangle$ basis states:

$$CX_{0,1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Eagle-eyed readers will find the *X* gate in the lower right quadrant of this matrix and the identity matrix in the upper left. This can be misleading. The important thing to note is that a controlled Not gate is a permutation matrix. Applying this matrix to states $|00\rangle$ and $|01\rangle$ leaves the states intact:

$$CX_{0,1}\,|00\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |00\rangle,$$

$$CX_{0,1}\,|01\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = |01\rangle.$$

Application to $|10\rangle$ however *flips* the second qubit to a resulting state of $|11\rangle$:

$$CX_{0,1}\,|10\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |11\rangle.$$

Similarly, application to $|11\rangle$ *flips* the second qubit to a resulting state of $|10\rangle$:

$$CX_{0,1}\,|11\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = |10\rangle.$$

The CX matrix flips the second qubit from $|0\rangle$ to $|1\rangle$, or from $|1\rangle$ to $|0\rangle$, but only if the first qubit is in state $|1\rangle$. In this example, the X gate on the second qubit is *controlled* by the first qubit, but any 2×2 quantum gate can be controlled this way. We can control Z gates, rotations, or any other 2×2 gate.

The CX gate is usually introduced, as we did here, by its effects on the $|0\rangle$ and $|1\rangle$ states of the second qubit where both qubits are *adjacent*. What if the controller and the controlled qubit are farther apart or in inverted order? The following shows a general way to construct a controlled unitary operator U with the help of projectors. In the tensor products below, the projectors $P_{|0\rangle}$ and $P_{|1\rangle}$ are at the position of the controlling qubit and U is at the position of the controlled qubit:

$$CU_{0,1} = P_{|0\rangle} \otimes I + P_{|1\rangle} \otimes U. \tag{2.12}$$

With this recipe, we can construct a controlled Not gate $CX_{1,0}$ from 1 to 0. Note that in this gate, you won't find the original X gate or the identity matrix in the operator, but it is still a permutation matrix:

$$CX_{1,0} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

If there are n qubits between the controlling and controlled qubits, n identity matrices must also be tensored between them. If the index of the controlling qubit is higher than the index of the controlled qubit, the positions of the gates and the projectors change in the tensor product in Equation (2.12). Here is an example of qubit 2 controlling gate U on qubit 0:

$$CU_{2,0} = I \otimes I \otimes P_{|0\rangle} + U \otimes I \otimes P_{|1\rangle}.$$

The corresponding code is straightforward. We have to make sure that the right number of identity matrices are being added to pad the operator:

```
def ControlledU(idx0: int, idx1: int, u: Operator) -> Operator:
  assert idx0 != idx1, 'Controller / Controlled must not be equal.'
  p0 = ZeroProjector(1)
```

```
    p1 = OneProjector(1)

    # space between qubits
    ifill = Identity(abs(idx1 - idx0) - 1)

    # 'width' of U in terms of Identity matrices
    ufill = Identity().kpow(u.nbits)

    if idx1 > idx0:
      op = p0 * ifill * ufill + p1 * ifill * u
    else:
      op = ufill * ifill * p0 + u * ifill * p1
    return op
```

With this code, we can control operators larger than 2×2 matrices. We can also construct controlled–controlled gates, controlled–controlled–controlled gates, and even longer sequences of controlled-... gates, which are required for many interesting algorithms.

The code creates one large operator matrix. This can be a problem for larger circuits, e.g., for a circuit with 20 qubits, with qubit 0 controlling qubit 19 (or any other padded operator). The operator would be a matrix of size $(2^{20})^2$, multiplied by the size of a Python `complex` data type, which could amount to a total of 8 or 16 terabytes[25] of memory. Building such a large matrix in memory and applying it via matrix–vector multiplication can become intractable. Since this is how we express operators at this point, we are limited by the number of qubits we can experiment with. Fortunately, there are techniques to significantly improve scalability, which we will discuss in Chapter 3.

Also, note that we allow the controller and controlled qubits to be at arbitrary distances from each other in our simulations. In a real quantum computer, there are topological limitations to the possible interactions between qubits. Mapping an algorithm on a concrete physical topology introduces another set of interesting problems. IBM (2021b) shows several examples, which we will touch on in Section 16.4.

2.8.1 Controlled Not Gate

The controlled Not gate (*CNOT*) is a key ingredient in introducing entanglement into a circuit, as we shall see shortly. It deserves its own constructor function. We already discussed this gate at the beginning of Section 2.8:

```
def Cnot(idx0: int = 0, idx1: int = 1) -> Operator:
  return ControlledU(idx0, idx1, PauliX())
```

The Controlled-by-0 Not gate (*CNOT0*) is similar to the *CNOT* gate, except that it is controlled by the $|0\rangle$ part of the controlling qubit. This is accomplished by inserting an X gate before and after the controlling qubit:

[25] tebibytes, to be precise.

```
def Cnot0(idx0: int = 0, idx1: int = 1) -> Operator:
  if idx1 > idx0:
    x2 = PauliX() * Identity(idx1 - idx0)
  else:
    x2 = Identity(idx0 - idx1) * PauliX()
  return x2 @ ControlledU(idx0, idx1, PauliX()) @ x2
```

Of course, this construction to control a gate by $|0\rangle$ works for *any* target gate. We will see several examples of this in later sections.

2.8.2 Swap Gate

The *Swap gate* is another important gate. Just as its name suggests, it swaps two qubits. Concretely, for two qubits $q_0 = (a \;\; b)^T$ and $q_1 = (c \;\; d)^T$, their product state is $q_0 \otimes q_1 = (ac \;\; ad \;\; bc \;\; bd)^T$. The swap gate will swap the elements at indices 1 and 2 in the state vector and turn it into $(ac \;\; bc \;\; ad \;\; bd)^T = q_1 \otimes q_0$. As a matrix, the gate is a permutation matrix:

$$SWAP_{0,1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Using Equation (2.12) to construct controlled gates cannot produce this gate. However, it turns out that a sequence of three *CNOT* gates swaps the probability amplitudes of the basis states, which compounds to a Swap gate. To swap qubits 0 and 1, you apply the three gates CX_{10}, CX_{01}, and CX_{10}. This is analogous to classical computing, where a sequence of three XOR operations can be used to swap classical bit values. These techniques do not require additional temporary storage, such as a temporary variable or a helper qubit.

There are many other ways to construct Swap gates, specifically for cases where the participating qubits are not adjacent. Several interesting alternatives are given in Gidney (2021b). Here is a standard implementation using the three *CNOT* gates:

```
def Swap(idx0: int = 0, idx1: int = 1) -> Operator:
  return Cnot(idx1, idx0) @ Cnot(idx0, idx1) @ Cnot(idx1, idx0)
```

2.8.3 Controlled Swap Gate

Like any other unitary operator, Swap gates can also be controlled. A controlled Swap gate is also known as the *Fredkin gate*. Similarly to the Toffoli gate, the Fredkin gate is a universal gate in classical computing, but not in quantum computing.[26] As a black

[26] There is no single universal gate in quantum computing, only sets of gates. We will not expand on this further. See, for example, www.scottaaronson.com/qclec/16.pdf for more details.

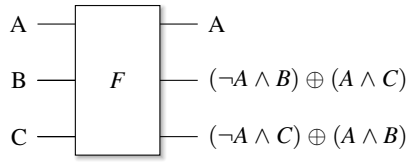


Figure 2.3 Block diagram for the Fredkin gate, which is a controlled Swap gate.

box, it represents the logic shown in Figure 2.3, which may be difficult to reason about in isolation (with $\wedge$ as logical AND and $\neg$ as logical NOT). However, the logic is simple. If qubit A is $|1\rangle$, qubits B and C are swapped in the tensor product of the three qubits:

$$CSWAP\ |A,B,C\rangle = |A,C,B\rangle, \quad \text{if } A = |1\rangle\,.$$

The first physical quantum Fredkin gate was built relatively recently (Patel et al., 2016) and used to construct GHZ states, which we describe in Section 2.11.4.

2.8.4 Controlled Phase Gate

Phase gates can also be controlled. They are especially interesting because they are symmetric: the controller and controlled qubits for a controlled phase gate can be *swapped* without changing the resulting operator matrix. With Equation (2.12) in Section 2.8 on controlled gates we saw how to construct controlled unitary gates as

$$CU_{0,1} = P_{|0\rangle} \otimes I + P_{|1\rangle} \otimes U,$$
$$CU_{1,0} = I \otimes P_{|0\rangle} + U \otimes P_{|1\rangle}.$$

Let's use the controlled Z gate as an example and compute the operator matrices. This will work for all phase gates derived from the U_1 gate, but we use the Z gate here; it is the easiest to read:

$$CZ_{0,1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

Swapping the indices from 0,1 to 1,0 results in

$$CZ_{1,0} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

In both cases, the identical result is

$$CZ_{0,1} = CZ_{1,0} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

```
def test_controlled_z(self):
  z0 = ops.ControlledU(0, 1, ops.PauliZ())
  z1 = ops.ControlledU(1, 0, ops.PauliZ())
  self.assertTrue(z0.is_close(z1))
```

2.9 Quantum Circuit Notation

Circuits will soon become quite sophisticated; we need an effective graphical circuit visualization, which we detail in this section. Qubits are drawn from top to bottom. In our big-endian convention, qubits are depicted from the most significant qubit to the least significant qubit. In equations, the top qubit will be on the left of a state, such as the 1 in $|\psi\rangle = |1000\rangle$. Another way to visualize this order may be to imagine $|\psi\rangle$ as a vector. Transposing this vector will move the leftmost qubit to the top spot in the transposed vector.

Graphically, the initial states of the qubits are drawn to the left of horizontal lines that go to the right, as shown in Figure 2.4. Lines indicate how the state changes over time as operators are applied. Again, note the absence of any classical control flow. All qubits are active at all times in the combined state. By convention, qubits are always initialized in state $|0\rangle$. However, because it is trivial to insert X or Hadamard gates, we sometimes take shortcuts and draw circuits as if they were present.

The application of a Hadamard gate (or any other gate) to a qubit is drawn with the gate symbol (H in this case) on the line corresponding to the qubit. To describe the state at a given point during the execution of the circuit, we add dotted vertical lines and mark the states at that point with a subscript, like $|\psi_0\rangle$ and $|\psi_1\rangle$ in Figure 2.5a.

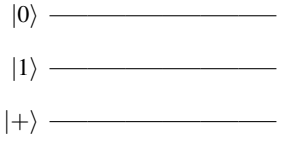
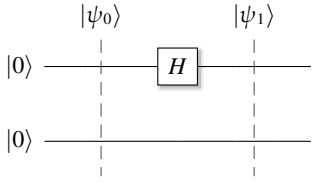
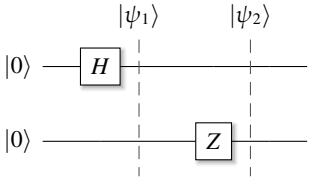


Figure 2.4 The structure of a quantum circuit. Qubits are initialized, and computation flows from left to right. This circuit has no gates yet.



(a) A Hadamard gate on the first qubit



(b) A Z gate to the right of the H gate.

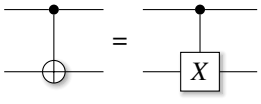
Figure 2.5 Applying a Hadamard gate and a Z gate in sequence.

Recall that the state of the circuit is the combined state of all the qubits in the state. For example, the initial state before the Hadamard gate is $|\psi_0\rangle = |0\rangle \otimes |0\rangle = |00\rangle$, indicated by a dotted vertical line marked with $|\psi_0\rangle$. The Hadamard gate puts the top qubit in superposition $(|0\rangle + |1\rangle)/\sqrt{2} = |+\rangle$. As a result, the state $|\psi_1\rangle$ in Figure 2.5a is the tensor product of the top qubit with the bottom qubit $|0\rangle$:

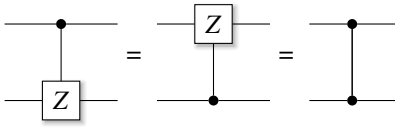
$$|\psi_1\rangle = |+\rangle \otimes |0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |0\rangle = \frac{1}{\sqrt{2}}\big(|00\rangle + |10\rangle\big).$$

Applying a Z gate to qubit 1 after the Hadamard gate results in the circuit shown in Figure 2.5b. The fact that the Z gate is to the right of the Hadamard gate indicates that this operator should be applied after the Hadamard gate (although, in this case, their order would not matter).

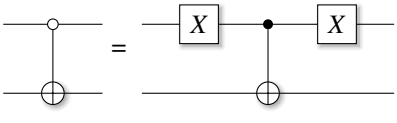
Controlled X gates are indicated with a solid circle for the controller qubit and the addition-modulo-2 symbol $\oplus$ for the controlled qubit (though not to be confused with the symbol for the tensor product $\otimes$). In some instances, we may still want to denote an X gate, but these two are identical:



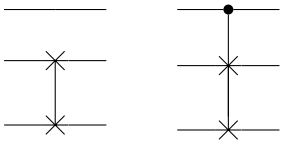
Any single-qubit gate can be controlled in this way. The controlled Z gate is symmetric (as are all other phase gates; see Section 2.8.4). It is quite common and gets its own graphical representation:



The Controlled-by-0 Not gate can be built by applying an X gate before and after the controller. It is drawn with an empty circle on the controlling qubit:



Swap gates are marked with two connected $\times$ symbols, as in the circuit diagrams below. Like any other gate, swap gates can also be controlled:



If a gate is controlled by more than one qubit, it is drawn with multiple black or empty circles, depending on whether the gates are controlled by $|1\rangle$ or $|0\rangle$. In the example in Figure 2.7, qubits 0 and 2 must be $|1\rangle$ (have an amplitude for this base state), and qubit 1 must be $|0\rangle$ to activate the X gate on qubit 3.

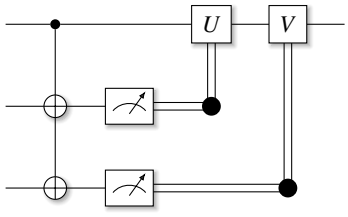


Figure 2.6 Measurement and flow of classical data after measurement, which is indicated with double lines.

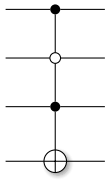


Figure 2.7 An example multi-controlled Not gate. The first and third qubit are controlled by $|1\rangle$, and the second qubit is controlled by $|0\rangle$.

We will learn more about measurements in Section 2.13. Measurement gates produce real, classical values and are indicated with a meter symbol. Classical information flow is drawn with double lines. In the example in Figure 2.6, measurements are being made, and the real, classical measurement data may then be used to build or control other unitary gates, U and V in the example.[27]

2.10 Multi-controlled Gates

Now that we know how to create controlled gates, the logical next step is to devise mechanisms to control gates with multiple controllers. In this section, we will first show how to create a controlled–controlled Not operator. Next, we introduce the Sleator–Weinfurter construction, which uses only 2-qubit gates and enables efficient simulation. We will conclude with a mechanism for creating gates that are controlled by an arbitrary number of qubits.

2.10.1 Controlled–Controlled Not Gate

The full matrix construction for the controlled gates works in a nested fashion, extending the control to already controlled gates. A double-controlled X gate is also called the Toffoli gate or, for short, the *CCX* gate. This gate is interesting in classical computing because it is a universal gate – every classical logic function can be constructed using just this gate. As mentioned before, this universality attribute does *not* hold in quantum

[27] All circuit diagrams in this book were created using the excellent LaTeX package `quantikz`, with a few custom settings.

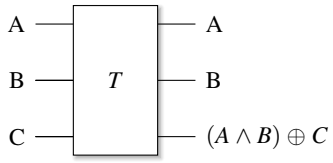


Figure 2.8 Block diagram for the Toffoli gate.

computing. In quantum computing, there are only *sets* of universal gates (see also Section 9.4).

This is how the Toffoli gate works: If the first two inputs are $|1\rangle$ it flips the third qubit. This is often shown as a logic block diagram (with $\wedge$ as the logical AND), as in Figure 2.8. In matrix form, we can describe it using block matrices, with 0_n as an $n \times n$ null matrix. Note that changing the indices of the controller and controlled qubits may destroy these patterns, but the resulting 8×8 matrix will still be a permutation matrix:

$$\begin{pmatrix} I_4 & 0_4 \\ 0_4 & CX \end{pmatrix} = \begin{pmatrix} I_2 & 0_2 & 0_2 & 0_2 \\ 0_2 & I_2 & 0_2 & 0_2 \\ 0_2 & 0_2 & I_2 & 0_2 \\ 0_2 & 0_2 & 0_2 & X \end{pmatrix}.$$

The constructor code is fairly straightforward and is a good example of how to construct a double-controlled gate:

```
def Toffoli(idx0: int, idx1: int, idx2: int) -> Operator:
  cnot = Cnot(idx1, idx2)
  toffoli = ControlledU(idx0, idx1, cnot)
  return toffoli
```

We observe that because we are able to construct quantum Toffoli gates and because Toffoli gates are classical universal gates, it follows that quantum computers are at least as capable as classical computers.

2.10.2 Sleator–Weinfurter Construction

For a given unitary matrix U and one of its square roots $R = \sqrt{U}$, we can construct a double-controlled U gate using only two-qubit gates with the pattern shown in Figure 2.9. This is important for simulation performance because two-qubit gates can be simulated very efficiently, as we will show in Chapter 3. Furthermore, building gates consisting of more than two qubits for physical machines can be a major challenge, if not impossible.

An example of a double-controlled gate is the Toffoli gate from Section 2.10.1, which is a double-controlled X gate, as shown in the quantum circuit notation on the left side of Figure 2.9. The Toffoli gate can be built with the *Sleator–Weinfurter* construction (Barenco et al., 1995), which is illustrated on the right-hand side of Figure 2.9.

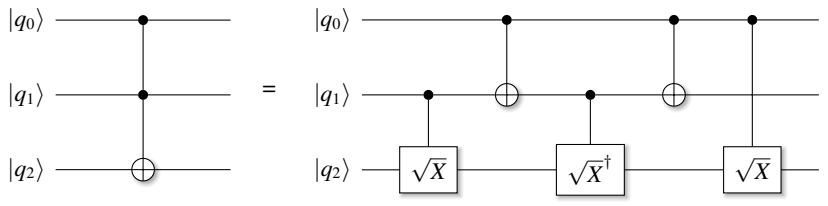


Figure 2.9 The Sleator–Weinfurter construction for a double-controlled X gate.

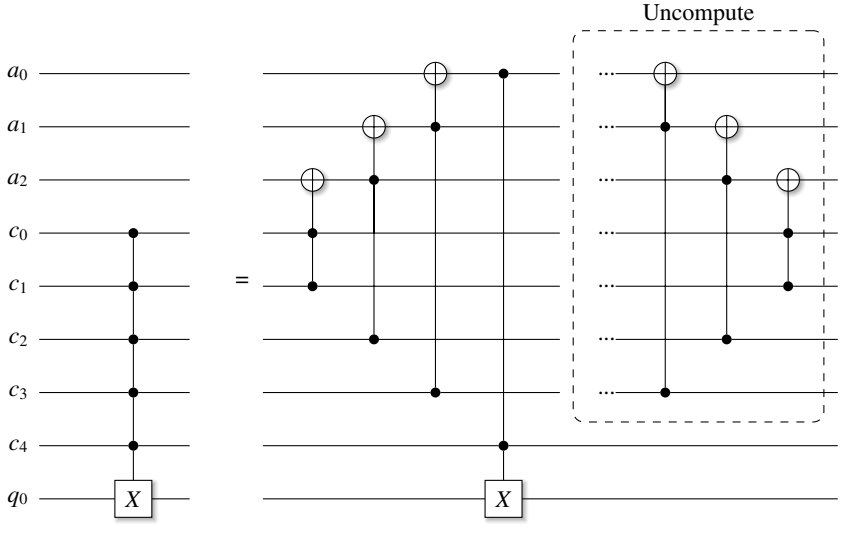


Figure 2.10 A multi-controlled X gate.

We know that the square root of the X gate is the V gate, and we also have the `adjoint()` function to compute the adjoint of a tensor. This construction works for any single-qubit gate and its root, so we can construct double-controlled X, Y, Z, T, and any other controlled 2×2 gates. We show the implementation in Section 3.3.5.

2.10.3 Multi-controlled Gates

Building on the efficient two-qubit construction introduced in Section 2.10.2, an elegant construction for a gate controlled by n controlling qubits requires $n - 2$ ancillary qubits. To see how this works, let us examine the first half of the circuit in Figure 2.10. This circuit has three ancilla qubits a_0, a_1, and a_2. It has the five controller qubits c_0 to c_4 and the controlled qubit q_0. The circuit builds a cascade of Toffoli gates to ultimately control the qubit q_0. First, the controllers c_0 and c_1 control the ancilla a_2. The next control qubit c_2 is then connected to a_2 with a Toffoli gate, controlling the ancilla a_1. Finally, the control qubit c_3 connects a Toffoli gate to ancilla a_1 to control the top ancilla a_0. This is also the top of the cascade. From here, we use a final Toffoli gate to control the X gate at the bottom, on qubit q_0, with both c_4 and a_0. This X gate

will have an effect only if all the controlling qubits and the topmost ancilla qubit were in state $|1\rangle$.

This construction can be used to control other single-qubit gates. A potential problem is that the system's state is now entangled with the ancillary qubits. A solution to this problem, which we detail in Section 2.12, is to *uncompute* the cascade of Toffoli gates by computing the adjoints of the gates and applying them in reverse order. By doing this, as shown in the right-hand half of Figure 2.10, the ancillary qubits return to their initial state. The state can then again be expressed as a product state, as all entanglement with the ancilla qubits has been eliminated. We detail an implementation of multi-controlled gates that may have 0, 1, or more controllers and that can be controlled by $|0\rangle$ or $|1\rangle$ in Section 3.3.6.

Other constructions are possible. Mermin (2007) proposes multi-controlled gates that trade additional gates for lower numbers of ancillae, as well as circuits that do not require the ancillae to be in $|0\rangle$ states (which may save a few uncomputation gates).

2.11 Entanglement

Entanglement is one of the most fascinating aspects of quantum physics. When two qubits (or systems) are entangled, measurement results are strongly correlated, even if the states are physically separated, be it by a millimeter or across the universe! This is the effect that Albert Einstein famously called "spooky action at a distance." If we entangle two qubits in a specific way (described below), and qubit 0 is measured to be in state $|0\rangle$, qubit 1 will always be in state $|0\rangle$ as well.

Why is this truly remarkable? Assume that we took two coins, placed them heads up in two boxes, and shipped one of the boxes to Mars. Regardless of how we shipped the boxes or in which order we opened the boxes, when we opened them, both coins showed heads. So, what is so special about the quantum case? In this example, the coins have a *hidden state*. We have placed them in the boxes *before* shipment, knowing which side to place on top in an initial, defined, non-probabilistic state. We also know that this state will not change during shipment.

If there were some form of a hidden state in quantum mechanics, then the whole theory would be incomplete. The quantum mechanical wave functions would be insufficient to describe a physical state fully. This was the point that Einstein, Podolsky, and Rosen attempted to make in their famous *EPR paper* (Einstein et al., 1935).

However, a few decades later, it was shown that there *cannot* be a hidden state in an entangled quantum system. A famous thought experiment, the Bell inequalities (Bell, 1964), proved this and it was later experimentally confirmed. We will detail a variant of the inequalities in Section 6.5 about the CHSH game.

Qubits collapse *probabilistically* during measurement to either $|0\rangle$ or $|1\rangle$.[28] This is equivalent to placing the coins in the boxes while they spin on their edges and shipping one of them to Mars. Let's assume that the long and likely bumpy journey by a rocket does not disturb their twirling. Only when we open the boxes will the coins fall to

[28] This is true as long as we measure in this computational basis. We talk about measurements in different bases in Section 13.1.3.

one of their sides. Perfect coins would fall on each side 50% of the time. Similarly, if we prepare a qubit in the $|0\rangle$ state and apply a Hadamard gate to it, this qubit will measure either $|0\rangle$ or $|1\rangle$, with a probability of 50% for each outcome. The magic of quantum entanglement is that both qubits of an entangled pair will measure the same value, either $|0\rangle$ or $|1\rangle$, 100% of the time. This is equivalent to the coins falling to the same side 100% of the time on Earth and Mars!

There are profound philosophical arguments about entanglement, measurement, and what they tell us about the very nature of reality. Many of the greatest physicists of the last century have argued over this for decades, including Einstein, Schrödinger, Heisenberg, and Bohr. These discussions have not been resolved to this day; there is no agreement. Many books and articles have been written solely on this topic and go into much more detailed and nuanced explanations than we are able to do here. We are not even going to try it. Instead, we take the laws of nature (as postulated) and use them as rules that we can use for computation.

This sentiment might put us in the camp of the *Copenhagen interpretation* of quantum mechanics (Faye, 2019). Ontology is a fancy term for questions like "What is?" or "What is the nature of reality?" The Copenhagen interpretation refuses to answer all ontological questions. To quote David Mermin (Mermin, 1989, p. 2): *If I were forced to sum up in one sentence what the Copenhagen interpretation says to me, it would be "Shut up and calculate!"*

The key here is that progress can be made, even if ontological questions remain unanswered.[29]

2.11.1 Product States

Consider a two-qubit system. Constructing the tensor product between the pure states of two qubits leads to a state where each qubit can still be described without reference to the other. There is no correlation between the two states.

There is an intuitive (though not general) way to visualize this. The state can be expressed as the result of a tensor product with the result $(a,b,c,d)^T$. If two states are *not* entangled, they are said to be in a *product state*. This is the case if $ad = bc$. If the states *are* entangled, this identity will *not* hold.

Proof As a quick proof, assume two qubits $q_0 = (i,k)^T$ and $q_1 = (m,n)^T$. Their Kronecker product is $q_0 \otimes q_1 = (im,in,km,kn)^T$. Multiplying the outer elements and the inner elements, corresponding to the $ad = bc$ form above, we see that

$$\underbrace{im}_{a}\underbrace{kn}_{d} = imkn = inkm = \underbrace{in}_{b}\underbrace{km}_{c}. \qquad \square$$

2.11.2 Entangler Circuit

The circuit in Figure 2.11 is the quintessential quantum entangler circuit. We will see many uses of it in this text. Let us discuss in detail how the state changes as the gates are applied.

[29] My colleague Sergio Boixo modified this quote to "Shut up and program" for this book.

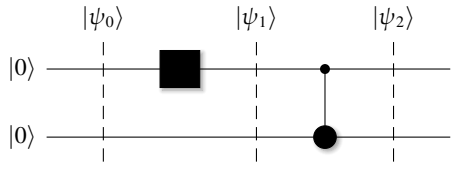


Figure 2.11 A simple circuit to entangle two qubits.

The initial state $|\psi_0\rangle$ before the Hadamard gate is the tensor product of the two $|0\rangle$ states, which is $|00\rangle$ with a state vector of $(1,\ 0,\ 0,\ 0)^T$. The Hadamard gate puts the first qubit in superposition of the $|0\rangle$ and $|1\rangle$ basis states and the state $|\psi_1\rangle$ becomes the tensor product of the superpositioned first qubit with the second qubit:

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}|0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle).$$

In code, we compute this with the snippet below. The resulting state has nonzero entries at indices 0 and 2, corresponding to the states $|00\rangle$ and $|10\rangle$:

```
   psi = state.zeros(2)
   op = ops.Hadamard() * ops.Identity()
   psi = op(psi)
   print(psi)
>>
[0.70710677+0.j 0.        +0.j 0.70710677+0.j 0.        +0.j]
```

Now we apply the controlled Not gate. The $|0\rangle$ part of the first qubit in superposition does not affect the second qubit, and the $|00\rangle$ part remains unchanged. However, the $|1\rangle$ part of the superpositioned first qubit controls the second qubit. It will flip the qubit to $|1\rangle$ and change the $|10\rangle$ part of the state to $|11\rangle$. The resulting state $|\psi_2\rangle$ after the controlled Not gate thus becomes

$$|\psi_2\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}},$$

which corresponds to the state vector

$$|\psi_2\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix} = \begin{pmatrix}a\\b\\c\\d\end{pmatrix}.$$

This state is now entangled because the $ad = bc$ identity in the above rule does not hold. The product of elements 0 and 3 is $1/2$, but the product of elements 1 and 2 is 0. The state can no longer be expressed as a product state.

In code, we take the state `psi` we computed above and apply the controlled Not gate. As we print the entangled 2-qubit state vector, we can see that vector elements 0 and 3, which correspond to the basis states $|00\rangle$ and $|11\rangle$, hold the value $1/\sqrt{2}$:

```
   psi = ops.Cnot(0, 1)(psi)
   print(psi)
>>
[0.70710677+0.j 0.         +0.j 0.         +0.j 0.70710677+0.j]
```

Only states $|00\rangle$ and $|11\rangle$ can now be measured. The other two basis states have a probability of 0. If qubit 0 is measured as $|0\rangle$, the other qubit will also be measured as $|0\rangle$, since the only nonzero probability state with a $|0\rangle$ as the first qubit is $|00\rangle$.

This explains the correlations (that spooky action at a distance), at least mathematically. The measurement results of the two qubits are 100% correlated. We don't know why, what physical mechanism facilitates this effect, or what reality is. Perhaps another famous Einstein quote applies: "Reality is just an illusion, albeit a very persistent one." At least for simple circuits and their respective matrices, we now have a means to express this unreal feeling of reality.

2.11.3 Bell States

Bell states are named after the great physicist John Bell (Burke et al., 1999), whose thought experiment using standard probability theory proved that entangled qubits could not have a hidden state or hidden information (Bell, 1964). This discovery was one of the defining moments for quantum mechanics, in particular, because a few years later, a physical experiment was devised that confirmed the theory.

We saw the first of four possible Bell states above, constructed with the entangler circuit and $|00\rangle$ as input. There are a total of four Bell states, resulting from the four inputs $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. We denote $|\beta_{xy}\rangle$ as the state resulting from the inputs[30] $|x\rangle$ and $|y\rangle$. In the literature, you will also find the symbols $|\Phi\rangle$ and $|\Psi\rangle$ to denote these states:

$$|\beta_{00}\rangle = \left|\Phi^+\right\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix},$$

$$|\beta_{10}\rangle = \left|\Phi^-\right\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\-1\end{pmatrix},$$

$$|\beta_{01}\rangle = \left|\Psi^+\right\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}0\\1\\1\\0\end{pmatrix},$$

$$|\beta_{11}\rangle = \left|\Psi^-\right\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}0\\1\\-1\\0\end{pmatrix}.$$

[30] We should interpret these indices themselves as little-endian.

PY

Find the code
In file src/lib/bell.py

```
def bell_state(a: int, b: int) ->state.State:
  assert a in [0, 1] and b in [0, 1], 'Bell bits must be 0 or 1.'
  psi = state.bitstring(a, b)
  psi = ops.Hadamard()(psi)
  return ops.Cnot()(psi)
```

The four Bell states form an orthonormal basis for two-qubit states, which are complex vectors in $\mathbb{C}^4$. Any two-qubit state $|\psi\rangle$ can be expressed as a linear combination of these four basis states:

$$|\psi\rangle = c_0\,|\beta_{00}\rangle + c_1\,|\beta_{01}\rangle + c_2\,|\beta_{10}\rangle + c_3\,|\beta_{11}\rangle\,.$$

Similarly to Equation (2.3), we can derive the individual factors c_i simply by computing the inner product of $|\psi\rangle$ with the four Bell states $c_i = \langle\beta_{xy}|\psi\rangle$. A simple example of this can be found in file bell_basis.py in the open-source repository.

PY

Find the code
In file src/bell_basis.py

Bell states are the simplest forms of entangled states. We will encounter them in Chapter 6 on entanglement-based algorithms, such as quantum teleportation or superdense coding. There are other entangled states with very interesting properties, namely the *GHZ state* and the *W state*, which we discuss next.

2.11.4 GHZ States

A generalization of Bell states is the GHZ state of three or more qubits, named after Greenberger, Horne, and Zeilinger (Greenberger et al., 2008). It can be constructed with a circuit as shown in Figure 2.12, which propagates the superposition from the top qubit to all other qubits via cascading controlled Not gates. Note that, as an alternative way to construct the circuit, instead of a cascade of controlled Not gates, we could instead connect the top qubit 0 with each of the lower qubits with a controlled Not gate.

This construction can be extended to more than three qubits, generalizing the GHZ states to $(|00\ldots0\rangle + |11\ldots1\rangle)/\sqrt{2}$. Only one of two possible states can be measured in the computational basis, each with a probability of $1/2$.

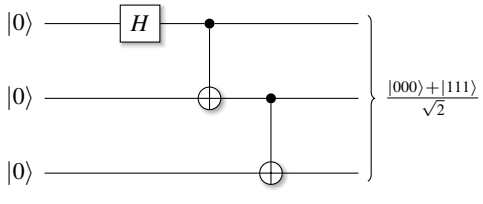


Figure 2.12 Circuit to construct a GHZ state.

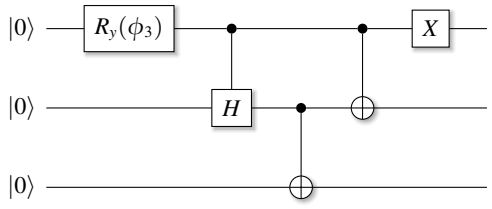


Figure 2.13 Circuit to construct a W state with $\phi_3 = 2 \arccos \frac{1}{\sqrt{3}}$.

```
def ghz_state(nbits: int) -> state.State:
  psi = state.zeros(nbits)
  psi = ops.Hadamard()(psi)
  for offset in range(nbits-1):
    psi = ops.Cnot(0, 1)(psi, offset)
  return psi
```

GHZ states are considered maximally entangled (see Section 4.4 for a definition of this term), even though no standard metric seems to exist for multipartite systems. The GHZ state is interesting because it is no longer entangled after "losing" a single qubit. We can simulate this with the partial trace routine we introduce in Section 4.3. For example, by tracing out the third qubit (at index 2) from the state density matrix, we obtain

$$\mathrm{tr}_2 \left[\left(\frac{|000\rangle + |111\rangle}{\sqrt{2}} \right) \left(\frac{\langle 000| + \langle 111|}{\sqrt{2}} \right) \right] = \frac{|00\rangle \langle 00| + |11\rangle \langle 11|}{2}.$$

The result is an unentangled mixed state, a statistical ensemble of pure states. We will learn more about these terms in Section 4.2.

2.11.5 W State

There are two typical ways to entangle three qubits. The above GHZ state is one way, and the W state, named after Wolfgang Dür (Dür et al., 2000), is the other. The GHZ and W states are *inequivalent*, meaning they cannot be transformed into each other by standard unitary transformations. The W state has the form

$$|W\rangle = \frac{1}{\sqrt{3}} \big(|001\rangle + |010\rangle + |100\rangle \big).$$

This state is interesting because it is more robust against the loss of a qubit. After losing one, the remaining state is still entangled. We will experiment with this in Section 4.3 on the partial trace. Following the circuit diagram in Figure 2.13, the W state can be generated with this corresponding code:

```
def w_state() -> state.State:
  psi = state.zeros(3)
  phi3 = 2 * np.arccos(1 / np.sqrt(3))
```

```
psi = ops.RotationY(phi3)(psi, 0)
psi = ops.ControlledU(0, 1, ops.Hadamard())(psi, 0)
psi = ops.Cnot(1, 2)(psi, 1)
psi = ops.Cnot(0, 1)(psi, 0)
psi = ops.PauliX()(psi, 0)
return psi
```

2.11.6 No-Cloning Theorem

There is another profound difference between classical and quantum computing. In classical computing, it is always possible to copy a bit, a byte, or any memory many times. This is verboten in quantum computing. It is generally impossible to clone the state of an unknown arbitrary qubit. This inability to copy is expressed with the so-called *No-Cloning Theorem* (Wootters and Zurek, 1982). This restriction is related to the topic of measurements and the fact that it is impossible to create a measurement device that does not impact (entangle with) a state.

THEOREM: *Given an arbitrary unknown quantum state* $|\psi\rangle = |\phi\rangle|0\rangle$, *there cannot exist a unitary operator* U *such that* $U|\psi\rangle = |\phi\rangle|\phi\rangle$.

Proof Assume that such an operator U exists. This means that U would take an arbitrary state $|\phi\rangle\,|0\rangle$ and transform it into

$$U\,|\phi\rangle\,|0\rangle = |\phi\rangle\,|\phi\rangle\,.$$

The state $|\phi\rangle$ is an unknown arbitrary state, which means that the operator U should also work for another such unknown state $|\psi\rangle$:

$$U\,|\psi\rangle\,|0\rangle = |\psi\rangle\,|\psi\rangle\,.$$

We know that U must be unitary, and unitary matrices preserve the inner product. Let us take the inner product of these multi-qubit states before and after cloning. Before cloning, we calculate the inner product as

$$\big(\langle\phi|\,\langle 0|\big)\cdot\big(|\psi\rangle\,|0\rangle\big),$$

which, according to Equation (1.5), is

$$\langle\phi|\psi\rangle\,\underbrace{\langle 0|0\rangle}_{=1} = \langle\phi|\psi\rangle.$$

However, after cloning, the inner product becomes

$$\big(\langle\phi|\,\langle\phi|\big)\cdot\big(|\psi\rangle\,|\psi\rangle\big) = \langle\phi|\psi\rangle\langle\phi|\psi\rangle = |\langle\phi|\psi\rangle|^2.$$

Since we expect U to preserve the inner product, it must be true that $\langle\phi|\psi\rangle = |\langle\phi|\psi\rangle|^2$. However, this only holds if $\langle\phi|\psi\rangle = 0$ or $\langle\phi|\psi\rangle = 1$. This is not true in the general case and for arbitrary states. It follows that an unknown arbitrary state cannot be cloned. □

Arbitrary states can be *moved* but not *cloned*. Obviously, this leads to interesting challenges in quantum algorithm design and the design of quantum programming languages.

Note the special cases of the basis states $|0\rangle$ and $|1\rangle$. These states *can* be cloned as long as they are not in superposition. This is easy to see for a state of the form $|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$. With one of α or β being 1 and the other being 0, only one term can remain after applying U:

$$\begin{aligned} U|\phi\rangle|0\rangle &= \alpha^2|00\rangle + \beta\alpha|10\rangle + \alpha\beta|01\rangle + \beta^2|11\rangle \\ &= \alpha|00\rangle + \beta|11\rangle. \end{aligned}$$

2.11.7 No-Deleting Theorem

Similarly to the just described No-Cloning Theorem, which states that in general an unknown quantum state cannot be cloned, the *No-Deleting Theorem* (Pati and Braunstein, 2000) proves that for two qubits in an unknown but identical state $|\psi\rangle$, there cannot be a unitary operator to *delete* or *reset* only one of the two qubits back to state $|0\rangle$.

THEOREM: *Given a general quantum state $|\psi\rangle|\psi\rangle|A\rangle$, with two qubits in an identical state $|\psi\rangle$ and an ancilla $|A\rangle$, there cannot be a unitary operator U, such that $U\,|\psi\rangle\,|\psi\rangle\,|A\rangle = |\psi\rangle\,|0\rangle\,|A'\rangle$, where A' is the ancilla's state after the application of U.*

Proof Assume we had an operator U that is capable of performing the deletion operation:

$$\begin{aligned} U\,|0\rangle\,|0\rangle\,|A\rangle &= |0\rangle\,|0\rangle\,|A'\rangle\,, \\ U\,|1\rangle\,|1\rangle\,|A\rangle &= |1\rangle\,|0\rangle\,|A'\rangle\,. \end{aligned}$$

As before, we calculate the application of U to state $|\psi\rangle|\psi\rangle|A\rangle$ in two different ways. First, for an individual qubit in state $\alpha\,|0\rangle + \beta\,|1\rangle$ with $|\alpha|^2 + |\beta|^2 = 1$ and with the operator U as defined above, we get

$$\begin{aligned} U\,|\psi\rangle\,|\psi\rangle\,|A\rangle &= |\psi\rangle\,|0\rangle\,|A'\rangle \\ &= \big(\alpha\,|0\rangle\,|0\rangle + \beta\,|1\rangle\,|0\rangle\big)\,|A'\rangle\,. \end{aligned} \tag{2.13}$$

Now let's compute the state as the tensor product of the qubits and apply the hypothetical operator U:

$$\begin{aligned} U\,|\psi\rangle\,|\psi\rangle\,|A\rangle &= U\big((\alpha\,|0\rangle + \beta\,|1\rangle)(\alpha\,|0\rangle + \beta\,|1\rangle)\,|A\rangle\big) \\ &= U\big(\alpha^2\,|00\rangle + \alpha\beta\,|01\rangle + \beta\alpha\,|10\rangle + \beta^2\,|11\rangle\big)\,|A\rangle \\ &= \alpha^2 U\,|00\rangle\,|A\rangle + \beta^2 U\,|11\rangle\,|A\rangle + \alpha\beta U\,|01\rangle\,|A\rangle + \beta\alpha U\,|10\rangle\,|A\rangle \\ &= \alpha^2\,|00\rangle\,|A'\rangle + \beta^2\,|10\rangle\,|A'\rangle + \alpha\beta U\big(|01\rangle + |10\rangle\big)\,|A\rangle\,. \end{aligned}$$

This form is different from Equation (2.13) because it has an additional (entangled) component $\left(|01\rangle + |10\rangle\right)$, which we can abbreviate as $|\Phi\rangle$. The final form becomes

$$\left(\alpha^2 |00\rangle + \beta^2 |10\rangle\right) |A'\rangle + \alpha\beta U |\Phi\rangle |A\rangle . \tag{2.14}$$

In general, Equation (2.13) is different from Equation (2.14), a contradiction that proves that no such operator U can exist. □

Note that if $\alpha = 0$ or $\beta = 0$, we again deal with the equivalent of classical bits. For these cases, the final term in Equation (2.14) disappears, and thus, an operator U for these classical states is feasible.

2.12 Uncomputation

In the last sections, we learned about the No-Cloning Theorem and the No-Deleting Theorem. We have also seen how qubits entangle with ancilla qubits and themselves during the construction of complex circuits. How are we supposed to extract clean, high-probability results if resulting states are just hairballs of all-entangled qubits? This is where the technique of *uncomputation* comes to the rescue, which we discuss in this section.

The question of logical reversibility of computation was raised by Bennett (1973). That paper was an answer to Landauer, who is also known for Landauer's principle (Landauer, 1973). That principle states that the *erasure* of information during computing must result in heat dissipation.[31] Truly *reversible* computing would use almost no energy (in theory), but reversing a computation would also undo any obtained result. Therefore, the question was whether it was feasible to construct a reversible circuit from which any meaningful result could be obtained. Given that quantum computing is reversible by definition, it would be utterly useless if we did not answer this question. Fortunately, Bennet found an elegant construction to resolve this issue.

Bennet's paper is formal and based on a three-tape Turing machine.[32] The proposed mechanism would compute a result, then *fan out* the result to a new tape, before uncomputing the result via reverse computation of one of the Turing machine's tapes. The goal at the time was to mitigate heat dissipation. In quantum computation, our goal is to break undesirable entanglement with ancillary qubits. Bennet's approach works for both.

We mentioned ancillary qubits before. Let us quickly define the relevant terms:

- For constructions like the multi-controlled gate from Section 2.10.3, we need *additional* qubits to perform the computation correctly. You may think of these qubits as temporary qubits or helper qubits, which play no essential role for the algorithm. They are equivalent to the stack space allocated by a classical compiler to mitigate register pressure. These qubits are called *ancilla qubits*, or the plural *ancillae*. You will also see the term *ancillary qubits*.

[31] Landauer's principle does contribute to modern CPU's heat dissipation, but the effect is very small when compared to leakage current and other more dominant effects (Bérut, A. et al., 2015).

[32] See also en.wikipedia.org/wiki/Multitape_Turing_machine.

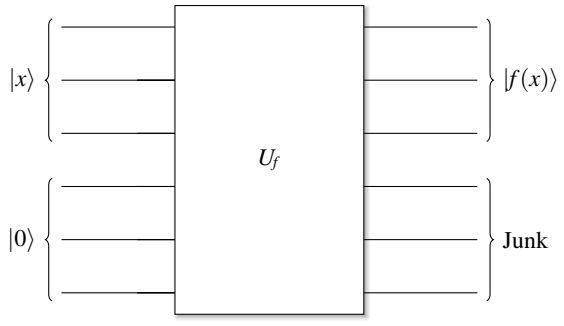


Figure 2.14 Typical structure of a quantum computation.

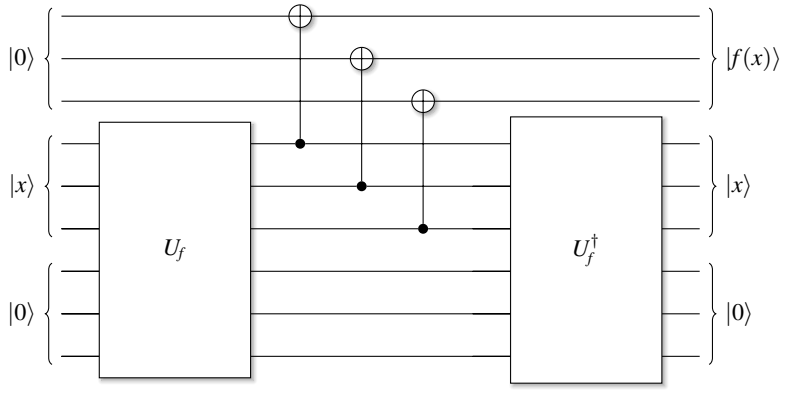


Figure 2.15 Computation of a result with U_f, followed by the transfer of the results to ancilla qubits, followed by *uncomputation* with $U_f^\dagger$.

- Ancilla qubits may start in state $|0\rangle$ and also end up in state $|0\rangle$ after a construction such as the multi-controlled gate. In other scenarios, however, the ancillae may remain entangled with a state, potentially destroying a desired result. In this case, we call these ancillae *junk qubits*, or simply *junk*.

The typical structure of a quantum computation looks like the one shown in Figure 2.14. All quantum gates are unitary, so we can pretend that we have packed them all up into one giant unitary operator U_f. There are the input state $|x\rangle$ and some ancillary qubits, all initialized to $|0\rangle$. The result of the computation will be $|f(x)\rangle$ and some leftover ancillae, which are now *junk*; they serve no purpose; they just hang around, intent on messing up our results. The problem is that the junk qubits may still be entangled with the result, nullifying the intended effects of quantum interference upon which quantum algorithms are based.

Here is the uncomputation procedure, as shown in Figure 2.15. After computing a solution with operator U_f, we have to apply the inverse unitary operation $U_f^\dagger$ to completely undo the computation. We say we "uncomputed" the effect of U_f. We can either build giant combined unitary operators as shown in the figure or, if we have constructed U_f as a circuit using individual gates, we apply the inverses of the gates in

reverse order to undo the computation. This works because the operators are unitary and $U^\dagger U = I$.

The problem is that in this way, we will lose the result $|f(x)\rangle$ that we were trying to compute. Here is the "trick" to work around this problem (which is similar to Bennet's recipe). After the computation, but before the uncomputation with $U_f^\dagger$, we fan out the result qubits to a set of ancillary qubits with the help of controlled Not gates, as shown in the middle part of Figure 2.15. With this circuit, the result of $|f(x)\rangle$ will be in the upper ancillae, and the uncomputation restores the other registers to their original state, eliminating all unwanted entanglement!

How does this work? We start in the state composed of an input state $|x\rangle$ and a working register initialized with $|0\rangle$. The first U_f transforms the initial state $|x\rangle|0\rangle$ into $|f(x)\rangle|g(0)\rangle$, with some algorithm-dependent, likely entangled $|g(0)\rangle$. To extract the result, we add an ancillary register at the top to manufacture the product state $|0\rangle|f(x)\rangle|g(0)\rangle$.

Suppose that, when expressed as basis states, $|f(x)\rangle = \sum_i c_i\,|i\rangle$. As we now connect the result register to the ancilla register with controlled Not gates, we obtain the state $\sum_i c_i\,|i\rangle\,|i\rangle\,|g(0)\rangle$. Fortunately, this does *not* violate the no-cloning theorem because the result is not a product state.[33] The two registers cannot be measured independently and give the same result.

We apply $U_f^\dagger$ to the two lower registers to uncompute U_f and obtain $|f(0)\rangle|x\rangle|0\rangle$. The final result is now in the top register, the bottom registers have been successfully restored, and we have succeeded in extracting the result. This is one of the fundamental techniques of quantum computing and we make use of it in many places in this book.

2.13 Measurement

We have almost reached the end of this introductory chapter. The remaining task is to discuss measurements. This is a complex subject with many subtleties and layered theories. We will keep it simple and stick to projective measurements only.

2.13.1 Postulates of Quantum Mechanics

The rules of quantum mechanics are different from the typical observed physical laws of nature in that they are *postulates*. Depending on the author and context, you may find between four and six of them, presented in a different order and with different focus and rigor. In keeping with the spirit of our text, we present them here in a more informal way that conveys just enough information to understand the essence of the postulates:

1. The state of a system is represented by a ket, which is a unit vector of complex numbers representing probability amplitudes.

[33] In fact, writing the state incorrectly as $|f(x)\rangle|f(x)\rangle|g(0)\rangle$ would have violated the cloning theorem.

2. A state evolves as the result of unitary operators operating on the state (in a closed system) as $|\psi'\rangle = U|\psi\rangle$. This is derived from the time-independent Schrödinger equation. To describe the evolution of a system in continuous time, this postulate is expressed with the time-dependent Schrödinger equation (which we mostly ignore in this text).[34]
3. Quantum measurements are described by *measurement operators*. Measurement means obtaining a singular measurable value, which is a real eigenvalue of a Hermitian observable. The probability amplitudes and the corresponding probabilities determine the likelihood of a specific measurement result. This may sound more scary than it actually is and will be the focus of this section.
4. After measurement, the state *collapses* to the measurement result. This is also called the *Born rule*. We will explain the implications of this postulate and the need for renormalization.
5. The state space of a composite physical system is the tensor product of the individual state spaces of components of the system. We already used this postulate in Section 2.4, where we discussed multi-qubit states.

The postulates are postulates, not standard physical laws. As noted above, they also have been the subject of almost a century of scientific disputes and philosophical interpretation. See, for example, Einstein et al. (1935), Bell (1964), Norsen (2017), Faye (2019), and Ghirardi and Bassi (2020), and many more. As we have stated before, we will avoid philosophy and focus on how the postulates enable interesting forms of computation.

2.13.2 Projective Measurements

The class of projective measurements is easy to understand and is the only method we will use in this text. Given a system in a superposition of two states, the idea behind making a projective measurement is simply to determine the probability that the system is in one state or the other. If we measure along the z-axis, we may wonder if a qubit was more likely to be in the $|0\rangle$ state or in the $|1\rangle$ state. The measurement returns only one of the two with a given probability. After measurement, according to Born's rule, the state *collapses* to the measured state (postulate 5). The qubit will now be in basis state $|0\rangle$ or $|1\rangle$ and will remain in this state for all future measurements.

To obtain the probabilities of a state $|\psi\rangle$ in the computational basis, we compute the norm of the inner product of the state with the computational basis vectors. We *project* the state onto the basis vectors to obtain the probabilities of a measurement outcome as

$$p_{|0\rangle} = |\langle 0|\psi\rangle|^2 = \left| \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \right|^2 = |\alpha|^2,$$

$$p_{|1\rangle} = |\langle 1|\psi\rangle|^2 = \left| \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \right|^2 = |\beta|^2.$$

[34] This is expressed as postulate 2' in Nielsen and Chuang (2011).

If we apply a Hadamard gate to state $|0\rangle = (1 \quad 0)^T$, the resulting state $|\psi\rangle = |+\rangle$ is located on the x-axis of the Bloch sphere. If we measure this state along the perpendicular z-axis, there should be a 50/50 chance that the result will be the $|0\rangle$ or $|1\rangle$ state. We will see that this is the basic idea of the random number generator we will discuss in Section 6.1. Using the same expressions from above for $|+\rangle$, measured in the computational basis, we get the expected probabilities as

$$p_{|0\rangle} = |\langle 0|\psi\rangle|^2 = \left|\begin{pmatrix}1 & 0\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}\right|^2 = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2},$$
$$p_{|1\rangle} = |\langle 1|\psi\rangle|^2 = \left|\begin{pmatrix}0 & 1\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}\right|^2 = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}.$$

How do we measure in a different basis? For example, can we measure the state along the x-axis, in the Hadamard basis $\{|+\rangle, |-\rangle\}$? In this case, we would expect a probability of 1 for state $|\psi\rangle = |+\rangle$. We can follow the same projection recipe as above, but this time using the Hadamard basis vectors:

$$p_{|+\rangle} = |\langle +|\psi\rangle|^2 = \left|\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}\right|^2 = 1,$$
$$p_{|-\rangle} = |\langle -|\psi\rangle|^2 = \left|\frac{1}{\sqrt{2}}\begin{pmatrix}1 & -1\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}\right|^2 = 0.$$

We will detail general measurements in other bases in Section 13.1.3. But let us return to the computational basis. We can perform a similar computation as above to extract probabilities using the projection operators $P_{|0\rangle}$ and $P_{|1\rangle}$:

$$P_{|0\rangle} = |0\rangle\langle 0| = \begin{pmatrix}1\\0\end{pmatrix}\begin{pmatrix}1 & 0\end{pmatrix} = \begin{pmatrix}1 & 0\\0 & 0\end{pmatrix},$$
$$P_{|1\rangle} = |1\rangle\langle 1| = \begin{pmatrix}0\\1\end{pmatrix}\begin{pmatrix}0 & 1\end{pmatrix} = \begin{pmatrix}0 & 0\\0 & 1\end{pmatrix}.$$

Applying a projector to a qubit "extracts" a subspace and reveals its probability amplitude. For example, for $P_{|0\rangle}$:

$$\begin{aligned} P_{|0\rangle}|\psi\rangle &= |0\rangle\langle 0|(\alpha|0\rangle + \beta|1\rangle) \\ &= \begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}\begin{pmatrix}\alpha\\\beta\end{pmatrix} = \begin{pmatrix}\alpha\\0\end{pmatrix} \\ &= \alpha\begin{pmatrix}1\\0\end{pmatrix} = \alpha|0\rangle. \end{aligned}$$

To compute the probability $p_{|i\rangle}$ of finding the ith basis state, we square the norm of the probability amplitude, as stated in the fourth postulate:

$$\begin{aligned} p_{|i\rangle} &= \left|P_{|i\rangle}|\psi\rangle\right|^2 \\ &= \left(P_{|i\rangle}|\psi\rangle\right)^\dagger\left(P_{|i\rangle}|\psi\rangle\right) \\ &= \langle\psi|P_{|i\rangle}^\dagger P_{|i\rangle}|\psi\rangle. \end{aligned}$$

The probabilities must add to 1 in

$$\sum_i p_{|i\rangle} = \sum_i \langle\psi|P_{|i\rangle}^\dagger P_{|i\rangle}|\psi\rangle = 1,$$

which leads us directly to the *completeness relation* for projection operators:[35]

$$\sum_i P_{|i\rangle}^\dagger P_{|i\rangle} = I.$$

Projectors are Hermitian and hence equal to their adjoint:

$$\begin{aligned} p_{|i\rangle} &= \langle\psi|P_{|i\rangle}P_{|i\rangle}|\psi\rangle \\ &= \langle\psi|P_{|i\rangle}^2|\psi\rangle. \end{aligned}$$

For projectors of normalized basis vectors, we also know that

$$P_{|i\rangle}^2 = P_{|i\rangle}.$$

This leads us to the final form for the probabilities as

$$p_{|i\rangle} = \langle\psi|P_{|i\rangle}|\psi\rangle.$$

The term $\langle\psi|P_{|i\rangle}|\psi\rangle$ is also called the *expectation value* of the operator $P_{|i\rangle}$, which is the quantum equivalent of the *average* of $P_{|i\rangle}$. It is often denoted with square brackets as $[P_{|i\rangle}]$. Now, from Equation (1.7) we know that

$$\mathrm{tr}\left(|x\rangle\langle y|\right) = \sum_{i=0}^{n-1} x_i y_i^* = \langle y|x\rangle. \tag{2.15}$$

By rearranging terms and, using Equation (2.15) and interpreting $\langle\psi|$ as $\langle y|$ and $P_{|i\rangle}|\psi\rangle$ as $|x\rangle$, we arrive at the form we will use in our code:

$$p_{|i\rangle} = \langle\psi|P_{|i\rangle}|\psi\rangle = \mathrm{tr}\left(P_{|i\rangle}|\psi\rangle\langle\psi|\right). \tag{2.16}$$

You can intuitively understand this form. The density matrix of the state $|\psi\rangle\langle\psi|$ has the probabilities p_i for each basis state $|x_i\rangle$ on the diagonal, as shown in Section 2.5. The projector zeros out all diagonal elements that are not covered by the projector's basis state. What remains on the diagonal are the probabilities of states that match the projector. The trace then sums up all these remaining probabilities.

After measurement, the state collapses to the measured result. Basis states that disagree with the measured qubit values get a resulting probability of 0. As a result, the remaining states' probabilities no longer add up to 1 and need to be renormalized, which we achieve with the following complicated-looking expression (no worries, in code, this will look quite simple):

[35] From here, it is only a small step to *generalized measurements* in the POVM (positive operator-valued measure) formalism, which we will not pursue in this text. In this formalism, our projectors are special cases of general measurement operators M_i, which also obey the completeness equation. The positive operator $E_m = M_i^\dagger M$ is a POVM element and the complete set of operators $\{E_m\}$ is called POVM. For each of the M_i, the *Kraus operator representation* is a set of matrices such that $M_i = A_i^\dagger A_i$. The A_i are called the Kraus operators. Since for our projectors $P_{|i\rangle}^\dagger = P_{|i\rangle}$ and $P_{|i\rangle}^\dagger P_{|i\rangle} = P_{|i\rangle}$, the projectors are also Kraus operators.

$$|\psi'\rangle = \frac{P_{|i\rangle}|\psi\rangle}{\sqrt{\langle\psi|P_{|i\rangle}|\psi\rangle}}. \tag{2.17}$$

As an example, let us assume we have the state

$$|\psi\rangle = \frac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big).$$

Each of the four basis states has the same probability $(1/2)^2 = 1/4$ of being measured. In addition, assume that qubit 0 is measured as $|0\rangle$. This means that the only choices for measuring the final full state are $|00\rangle$ or $|01\rangle$. The first qubit is "fixed" at $|0\rangle$ after measurement. This means that the states where qubit 0 is $|1\rangle$ now have a zero probability of ever being measured. The state collapses to the unnormalized state

$$|\psi\rangle_{(\neq|1|)} = \frac{1}{2}\big(|00\rangle + |01\rangle\big) + 0\,\big(|10\rangle + |11\rangle\big).$$

In this form, the norms squared of the probability amplitudes no longer add up to 1. We must renormalize the state following Equation (2.17) and divide by the square root of the expectation value (which was $1/2$):

$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|00\rangle + |01\rangle\big).$$

This normalization step might be surprising. How does Mother Nature know when and if to normalize? Given that we adhere to the Copenhagen interpretation and have decided to "Shut up and program," a possible answer is that the need for renormalization is simply a remnant of the mathematical framework, nothing more and nothing less.

2.13.3 Implementation

The function to measure a specific qubit has four parameters. The state to be measured is passed as parameter `psi`. The qubit to measure, indexed from the top/left, is passed as parameter `idx`. Whether to measure the probability that the state collapses to $|0\rangle$ or $|1\rangle$ is controlled by parameter `tostate`. Finally, whether the state should collapse after measurement is controlled by parameter `collapse`. In the physical world, measurement destroys superposition, but in our simulation we can just take a peek-a-boo at the probabilities without affecting the superposition of states.

The way this function is written, if we measure and collapse to state $|0\rangle$, the state is made to collapse to this state *independently* of the actual probabilities. There are other ways to implement this, for example, by selecting the measurement result based purely on probabilities. At this early point in our exploration, the ability to force a result works quite well; it makes debugging easier. Care must be taken not to force the state to collapse to a result with probability 0. This would lead to a division by 0 with likely very confusing subsequent measurement results.

The function returns two values: the probability of measuring the desired qubit state and a state. The returned state is the collapsed post-measurement state if `collapse`

was set to `True`, or the unmodified state otherwise. In the implementation, the function first computes the density matrix and the padded operator around the projection operator. The probability is computed from a trace over the matrix resulting from the multiplication of the padded projection operator with the density matrix, as in Equation (2.16):

```
def Measure(psi: state.State, idx: int,
            tostate:int=0, collapse:bool=True) -> (float, state.State):
  rho = psi.density()
  op = ZeroProjector(1) if tostate == 0 else OneProjector(1)

  if idx > 0:
    op = Identity().kpow(idx) * op
  if idx < psi.nbits - 1:
    op = op * Identity().kpow(psi.nbits - idx -1)

   # Probability is the trace.
   prob = np.trace(np.matmul(op, rho))
```

If state collapse is required, we update the state and renormalize it before returning the updated (or unmodified) probability and state.

```
  if collapse:
    mvmul = np.dot(op, psi)
    divisor = np.real(np.linalg.norm(mvmul))

    assert divisor > 1e-10, 'Measurement collapses to p 0'
    normed = mvmul / divisor
    return np.real(prob), state.State(normed)

  # Return original state, enable chaining.
  return np.real(prob), psi
```

To clarify one more time, the measurement operators are projectors. They are Hermitian and positive semidefinite with eigenvalues 0 and 1, and an eigenvector $|1\rangle$. A measurement will produce $|0\rangle$ or $|1\rangle$, corresponding to the probabilities of the basis states. Measurement will not measure, for example, a value of 0.75. It will measure one of the two basis states with a probability of 0.75. This can be a source of confusion for novices. In the real world, we have to measure several times to estimate the probabilities with statistical significance.

2.13.4 Examples

Let us look at a handful of examples to see measurements in action. In the first example, let us create a 4-qubit state and look at the probabilities:

```
    psi = state.bitstring(1, 0, 1, 0)
    psi.dump()
>>
    |1010> (|10>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

There is only one state with nonzero probabilities. If we just measure the second qubit, the probability of finding basis state $|0\rangle$ is 1:

```
    p0, _ = ops.Measure(psi, 1)
    print(p0)
>>
    1.0
```

But if we tried to measure this second qubit to be in the basis state $|1\rangle$, which is a state that it cannot be in, we would expect an error:

```
    p1, _ = ops.Measure(psi, 1, tostate=1)
    print(p1)
>>
    AssertionError: Measurement collapses to 0.0.
```

Here is an example with a collapsing measurement. Let us create a Bell state:

```
    psi = bell.bell_state(0, 0)
    psi.dump()
>>
    |00> (|0>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
    |11> (|3>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
```

This state has only two possible measurement results, $|00\rangle$ and $|11\rangle$. Let us measure the first qubit to be $|0\rangle$ without collapsing the state:

```
    psi = bell.bell_state(0, 0)
    p0, _ = ops.Measure(psi, 0, 0, collapse=False)
    print('Probability: ', p0)
    psi.dump()
>>
    Probability:  0.49999997
    |00> (|0>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
    |11> (|3>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
```

This shows the correct probability of 0.5 of measuring $|0\rangle$, but the state is still unmodified. Now let's change this and collapse the state after the measurement, which is more reflective of making an actual, physical measurement:

```
    psi = bell.bell_state(0, 0)
    p0, psi = ops.Measure(psi, 0, 0, collapse=True)
    print('Probability: ', p0)
    psi.dump()
>>
    Probability:  0.49999997
    |00> (|0>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

Now only one possible measurement outcome remains, the state $|00\rangle$, which from now on will be measured with 100% probability.

3 Simulation Infrastructure

The basic infrastructure we have implemented so far, with its tensors, states, and operators implemented as large matrices, is sufficient to explore many small-scale quantum algorithms. It is great for learning and experimentation. However, more complex algorithms typically consist of much larger circuits with many more qubits. For these, the matrix-based infrastructure becomes unwieldy, error-prone, and does not scale. In this chapter, we develop an improved infrastructure that easily scales to larger problems. If you are not interested in infrastructure, you may only skim this content for now. Most of the remainder of the book is understandable without the low-level details presented here. However, we are building an initial high-performance quantum simulator. You don't want to miss it!

First, we give an overview of various levels of infrastructure with the corresponding computational complexities and levels of performance. We introduce *quantum registers*, which are named groups of qubits. We describe a *quantum circuit model*, where most of the complexity of the base infrastructure is hidden elegantly. To handle larger circuits, we need faster simulation speeds. We detail an approach to applying an operator with linear complexity rather than the methods with quadratic or even cubic complexities that we started with. We further accelerate this method with C++, attaining a performance improvement of up to 100 times over the Python version. For a specific class of algorithms, we can do even better with a sparse state representation, which we detail at the end of this chapter.

3.1 Simulation Complexity

This book focuses on algorithms and how to simulate them efficiently on a classical computer. The key attributes of the various implementation strategies are computational complexity, resulting performance, and the maximum number of qubits that can be simulated in *reasonable* time with *reasonable* resource requirements. To some extent, this whole endeavor seems doomed from the start. You will need a quantum computer to effectively run quantum algorithms, as a classical computer can only go so far. However, luckily for us, our techniques will take us far enough to learn the principles.

The size of the state vector grows exponentially with the number of qubits. For a single qubit, we only need to store two complex numbers, which amount to 8 bytes when using `float` or 16 bytes when using `double` as the underlying data type.

Two qubits require four complex numbers, and n qubits require 2^n complex numbers. Simulation speed and the ability to fit a state into memory are typically measured by the number of qubits at which a given methodology is still tractable. By tractable, we mean a result that can be obtained in less than roughly an hour. At the time of writing, the world record for storing and simulating a full wave function was 48 qubits (De Raedt et al., 2019).

Due to the exponential nature of the problem, improving the performance by a factor of eight means that we can handle only three more qubits ($2^3 = 8$). If we see a speed-up of 100 times, this means that we can handle six or seven additional qubits. In the following, we use n to count qubits and $N = 2^n$ for the size of the corresponding state vector. These are the five different approaches we describe in this book:

- **Worst**. Implementing gates as potentially huge matrices and constructing operators using matrix–matrix products is of complexity $\mathcal{O}(N^3)$. This is how we started in the previous chapter, and it is the worst case; avoid it if possible. It becomes intractable even with a relatively small number n of qubits, around $n \sim 8$.
- **Bad**. We can apply a gate to a state one at a time as a matrix$\times$vector product with complexity $\mathcal{O}(N^2)$, which is already a substantial improvement. We can simulate roughly $n \sim 12$ qubits.
- **Good**. In Section 3.5, we will learn that one- and two-qubit gates can be applied by linearly iterating over the state vector, which is a massive improvement. We should be able to simulate roughly $n \sim 18$ qubits with this technique.
- **Better**. We started our journey with Python but can accelerate it using C++. In Section 3.6 we will implement the previous `apply` functions in C++, extending Python with its foreign function interface (FFI). The performance gain of C++ over Python is about $100\times$ for these types of problems, and we may be able to simulate $n \sim 25$ qubits, depending on the problem.
- **Best**. In Section 3.9, we will change the underlying representation to a sparse one. This approach is still $\mathcal{O}(N)$ in the worst case, but it can and does win over other implementations by a significant factor. Improvements are possible because, for many circuits, the number of nonzero probability states is less than 3% or even lower. With this, we may reach $n \sim 30$ qubits or more for some algorithms.

We could further improve our techniques (which are also called *Schröedinger full-state simulations*) with well-known techniques from the field of high-performance computing (HPC), such as vectorization (which adds one or two qubits) or parallelization (64 cores could add $\log_2(64) = 6$ additional qubits). We could employ machine clusters with 128 or more machines and the corresponding additional qubits to reach a simulation capability of around 45 qubits using 512 TB of memory. Today's largest supercomputers would add another handful of qubits (if they were fully dedicated to a simulation job, including all their secondary storage).

These techniques do not add much to our material, and we will not discuss them further. We list a range of open-source solutions in Section 16.4.9, several of which support distributed simulation. The transpilation techniques detailed in Section 3.4.7 allow the targeting of several of these simulators. What these numbers demonstrate is how quickly the simulation hits the limits. Improving performance or scalability

by 10^3 only gains about 10 qubits. Adding 20 qubits results in 10^6 higher resource requirements.

There are other important simulation techniques. For example, there is the so-called Schrödinger–Feynman simulation technique, which is based on path history (Rudiak-Gould, 2006; Frank et al., 2009). This technique trades performance for reduced memory requirements. Other simulators work efficiently on restricted gate sets, such as the Clifford gates (Aaronson and Gottesman, 2004; Anders and Briegel, 2006). Furthermore, there is ongoing research on improving the simulation of specific circuit types (Markov et al., 2018; Pan and Zhang, 2021).) as well as *circuit cutting*, which breaks a large circuit into smaller subcircuits and combines their simulation results classically (Piveteau and Sutter, 2024).

3.2 Quantum Registers

For larger and more complex circuits, we want to make algorithms more readable by addressing qubits in named groups. For example, the circuit in Figure 3.1 has a total of eight qubits. We want to group the first four as `data`, the next three as `ancilla`, and the bottom one as a single `control` qubit. On the right side, the figure shows the global qubit number as g_i and the local offset into the named groups. For example, the global qubit index 5 for qubit g_5 corresponds to the local offset 1 for qubit $ancilla_1$. These named groups of adjacent qubits are called *quantum registers*.

The full state of the system is the tensor product of eight qubits, numbered g_0 to g_7. We want to address `data` with indices ranging from 0 to 3, which should produce the same global qubit indices 0 to 3 in the combined state. We want to index the `ancilla` qubits from 0 to 2, resulting in global qubit indices 4 to 6. Finally, we want to address `control` at index 0, resulting in global qubit index 7. In code, a simple list of indices will do the trick.

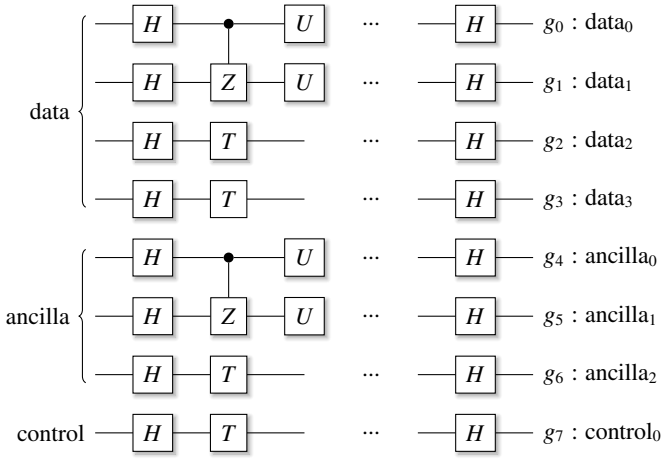


Figure 3.1 Quantum registers *data*, *ancilla*, and *control*.

The initial implementation is a bit rough. No worries, we will wrap this code up nicely in Section 3.3. We introduce a Python class `Reg` (for "Register") and initialize it by passing the size of the register file that we want to create *and* the current global offset, which must be manually maintained for this interface. In Figure 3.1, the first global offset is 0, for the second register it is 4, and for the last register it is 7.

We derive this class from the Python built-in `list` type, which means that we get functions like `__len__` or `__getitem__` for free. This enables the use of the Python `len` function and standard array indexing. In addition to the indices, we also want to maintain the potential initial values for each qubit, which we hold in the member array `val`.

By default, all qubits are assumed to be $|0\rangle$, but an initializer, `init`, can be provided. If `init` is an integer, it is converted to a string with the binary number representation. If `init` is a string (including after the previous step), tuple, or list, the `val` array is initialized with 0s and 1s according to the binary numbers passed.

```
class Reg(list):
  def __init__(self, size: int, init=None, global_reg=0) -> None:
    super().__init__([global_reg + idx for idx in range(size)])
    self.val = [0] * size
    if init:
      if isinstance(init, int):
        init = format(init, '0{}b'.format(size))
      if isinstance(init, (str, tuple, list)):
        for idx, val in enumerate(init):
          if val == '1' or val == 1:
            self.val[idx] = 1
```

For example, to create and initialize `data` with $|1011\rangle$ and `ancilla` with $|111\rangle$ (decimal 7), and to access global qubit 5, we can write:

```
data = state.Reg(4, (1, 0, 1, 1), 0)   # 0b1011
ancilla = state.Reg(3, 7, 4)   # 0b111
# Access global qubit[5] == ancilla[1] as:
... = ancilla[1]
```

Only two additional functions are needed. To give a textual representation of the register with the initial state, we write a short string conversion function `__str__` to print the register in state notation. To produce a quantum state from this register, the member function `psi` may be called once after the initialization is complete. All other typically required functionality to manage a list of indices, including slice management, is conveniently handled by the underlying `list` class.

```
  def __str__(self) -> str:
    return '|' + ''.join([f'{val}' for val in self.val]) + '>'

  def psi(self) -> State:
    return bitstring(*self.val)
```

Now, let's move on to discuss a more convenient abstraction and interface for quantum circuits that will utilize our quantum register implementation.

3.3 Circuits

So far, we have used full-state vectors and operator matrices to learn about the fundamentals and to implement small circuits. This infrastructure is easy to understand and works quite well for algorithms with a limited number of qubits. It is helpful for learning, but the representation is very verbose and explicit. It also exposes the underlying data structures, and that can cause problems like the following:

- Describing states and operators explicitly at a very low level of abstraction requires a lot of typing, which is inconvenient and error-prone.
- The representation exposes the implementation details. Changing aspects of the implementation would be challenging – all users of the base infrastructure would have to be updated.
- A minor point to make is that this style of representation differs from that commonly found in existing frameworks such as Qiskit (Gambetta et al., 2019) or Cirq (Google, 2021c).

The second problem is especially important in our context, as we want to develop faster ways to apply gates in Python and C++-accelerated Python. We might want to change the representation of states themselves from storing a full state vector to a sparse representation. The current level of abstraction does not allow that without changing all dependent client code.

To remedy these problems, we create a data structure called a quantum circuit `qc`. It nicely wraps up all the functions that we have discussed so far. The naming convention is to use all lower case to distinguish from the explicit representation discussed in Chapter 2. Importantly, this data structure enables quite sophisticated functionality:

- Gates can be executed as the circuit is being constructed, similar to the initial infrastructure above. This is also called *eager* execution.
- It also enables *non-eager* circuit construction, where gate sequences are simply recorded in an internal data structure that we call *intermediate representation* (IR), following the typical compiler phraseology.
- Using this IR, we can do several advanced things. We can execute a circuit multiple times, execute it in reverse for uncomputation using adjoint gates, and even control a whole circuit using another qubit.
- We can also *transpile* the IR to target other simulators or commercial frameworks. In this book, we used this technique in a few places to produce circuit diagrams in LaTeX.

The first thing we need is a circuit constructor that accepts a string argument to assign a name to the circuit. This name is used in printing and debugging. The internally stored quantum state `psi` is initialized to 1.0, indicating that there are no qubits in this

circuit immediately after creation. The parameter `eager` controls whether the circuit is executed eagerly during construction or whether all gates should be *stored* in the IR for later execution. For convenience, the IR object itself is always constructed, but the IR construction is controlled by the Boolean `build_ir`. We only enable IR construction in non-eager execution mode.

```
class qc:
  """Wrapper class to maintain state + operators."""

  def __init__(self, name=None, eager: bool = True):
    self.name = name
    self.psi = state.State(1.0)
    self.ir = ir.Ir()
    self.eager = eager
    self.build_ir = not eager
    self.global_reg = 0
```

3.3.1 Qubits

The circuit class supports quantum registers. As they are constructed with the functions detailed below, they are immediately tensored to the circuit's internal full state, using the helper function `_tprod`. This function maintains the global register count, hiding the rough interface of the underlying `Reg` class. Together, adding support for the qubit constructors discussed above is straightforward. We add the functions `rand_bits()` to create a random bit string of n qubits and `arange()` to build a non-quantum vector of values 0 to $n-1$ (used for debugging only).

```
  def _tprod(self, new_state, nqubits: int):
    self.psi = self.psi * new_state
    self.global_reg = self.global_reg + nqubits

  def reg(self, size: int, it=0, *, name: str = None) -> state.Reg:
    ret = state.Reg(size, it, self.global_reg)
    self._tprod(ret.psi(), size)
    return ret

  def qubit(self, alpha: np.complexfloating = None,
            beta: np.complexfloating = None) -> None:
    self._tprod(state.qubit(alpha, beta), 1)

  # and similar for these functions.
  def zeros(self, n: int) -> None
  def ones(self, n: int) -> None
  def bitstring(self, *bits) -> None
  def arange(self, n: int) -> None
  def rand_bits(self, n: int) -> None
```

3.3.2 Gate Application

To apply gates to qubits, assume that there are functions `apply1` for single-qubit gates and `applyc` for controlled two-qubit gates. We will develop their implementations in the following sections. Let us pretend that these functions will apply gates at index `idx`, with the control qubit at index `ctl` for controlled gates. In Python, optional function parameters can be specified after a single parameter `*`. Gates that need a parameter, such as rotations, get their optional value as `val`.

```
  def apply1(self, gate: ops.Operator, idx: int,
             name: str  = None, *, val: float = None):
    [...]

  def applyc(self, gate: ops.Operator, ctl: int, idx: int,
             name: str = None, *, val: float = None):
    [...]
```

3.3.3 Standard Gates

With these two apply functions in place, we can now wrap all standard gates and make them member functions of the circuit. But how should we apply adjoint gates? There are a variety of implementation strategies in Python, but we keep it simple in this book. If a gate is invoked with `gate_name`, we will add a corresponding function to apply the adjoint as `gate_name_dag`.

For non-parameterized single-qubit gates, we add the following code to the circuit constructor to add member functions to apply the gate (for example, as `qc.s` for the *S* gate, the adjoint gate (as `qc.s_dag`), the controlled gate (`qx.cs`), and the controlled adjoint gate (`qc.cs_dag`). These functions are added as object attributes that hold a lambda function. This way, we can add an optional parameter `cond` to the lambda for conditional gate application.

```
    self.simple_gates = [
        ['h', ops.Hadamard()], ['s', ops.Sgate()],  ['t', ops.Tgate()],
        ['v', ops.Vgate()],    ['x', ops.PauliX()], ['y', ops.PauliY()],
        ['z', ops.PauliZ()],   ['yroot', ops.Yroot()],
    ]
    for gate in self.simple_gates:
      self.add_single(gate[0], gate[1])
      self.add_single(gate[0] + 'dag', gate[1].adjoint())
      self.add_ctl('c' + gate[0], gate[1])
      self.add_ctl('c' + gate[0] + 'dag', gate[1].adjoint())
[...]
  def add_single(self, name: str, gate: ops.Operator):
    setattr(self, name, lambda idx, cond = True:
            self.apply1(gate, idx, name) if cond else None)

  def add_ctl(self, name: str, gate: ops.Operator):
```

```
  setattr(self, name, lambda idx0, idx1, cond = True:
          self.applyc(gate, idx0, idx1, name) if cond else None)
```

3.3.4 Parameterized Gates

For parameterized gates requiring a value, such as an angle, we add simple wrapper functions. Again, you can see how we call the `apply1` function for standard gates and the `applyc` function for controlled gates. The value is passed as parameter `val` to these functions.

```
def u1(self, idx: int, val):
  self.apply1(ops.U1(val), idx, 'u1', val=val)

def cu1(self, idx0: int, idx1: int, value):
  self.applyc(ops.U1(value), idx0, idx1, 'cu1', val=value)

def rx(self, idx: int, theta: float):
  self.apply1(ops.RotationX(theta), idx, 'rx', val=theta)

def crx(self, ctl: int, idx: int, theta: float):
  self.applyc(ops.RotationX(theta), ctl, idx, 'crx', val=theta)

# ... and similar for ry, cry, rz, crz
```

3.3.5 Controlled–Controlled Gates

For general single-qubit gates, we add functions for controlled and controlled–controlled gate applications, using the Sleator–Weinfurter construction outlined in Section 2.10.2. We add the alias `ccx` for the double-controlled Pauli *X* gate, a common abbreviation. The helper function `_ctl_by_0(ctl)` checks whether the index of the controlling qubit is passed as an integer or a single-element list (as in `[idx]`), indicating that the control qubit should be used as a Controlled-by-0 qubit.

```
def _ctl_by_0(self, ctl):
  if isinstance(ctl, int):
    return ctl, False
  return ctl[0], True

def cu(self, idx0: int, idx1: int, op: ops.Operator, desc: str = None):
  assert op.shape[0] == 2, 'cu only supports 2x2 operators'
  self.applyc(op, idx0, idx1, desc)

def ccu(self, idx0: int, idx1: int, idx2: int,
        op: ops.Operator, desc=''):
  """Sleator-Weinfurter Construction for general operators."""
```

```
    # Enable Control-By-0 (if idx is being passed as [idx])
    i0, c0_by_0 = self._ctl_by_0(idx0)
    i1, c1_by_0 = self._ctl_by_0(idx1)

    with self.scope(self.ir, f'cc{desc}({idx0}, {idx1}, {idx2})'):
      self.x(i0, c0_by_0)
      self.x(i1, c1_by_0)

      v = ops.Operator(sqrtm(op))
      self.cu(i0, idx2, v, desc + '^1/2')
      self.cx(i0, i1)
      self.cu(i1, idx2, v.adjoint(), desc + '^t')
      self.cx(i0, i1)
      self.cu(i1, idx2, v, desc + '^1/2')

      self.x(i1, c1_by_0)
      self.x(i0, c0_by_0)

  def ccx(self, idx0: int, idx1: int, idx2: int):
    self.ccu(idx0, idx1, idx2, ops.PauliX(), 'ccx')
```

3.3.6 Multi-Controlled Gates

To build multi-controlled gates as outlined in Section 2.10.3, we use the approach outlined here and make it quite fancy:

- For the controlling gates, we distinguish the special cases of 0, 1, 2, and more controllers.
- We allow for Controlled-by-1 gates and Controlled-by-0 gates. To mark a gate as Controlled-by-0, the index `idx` of the controller is passed as a single element list item `[idx]`.

Let us use the example in Figure 3.2, which has a controlled X gate on qubit q_4. This gate is controlled by By-1 and By-0 control qubits, marked as solid and hollow circles. To produce this controlled gate, we make the following function call, where the By-1 gates are passed as indices `0` and `3` and the By-0 gates as single-list elements `[1]` and `[2]`:

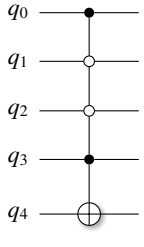


Figure 3.2 A multi-controlled X gate.

```
qc.multi_control([0, [1], [2], 3], 4, aux, ops.PauliX(), 'multi-X'))
```

Of course, we have to make sure that we have reserved enough space for the ancillae in the `aux` register. For our implementation, n control qubits require $n - 1$ ancilla qubits. For the full implementation, we also modified the function `applyc` to enable Controlled-by-0 gates by adding an X gate before and after the controller qubit (not shown here).

```
def multi_control(self, ctl, idx1, aux, gate, desc: str):
  """Multi-Controlled gate, using aux as ancilla."""

  with self.scope(self.ir, f'multi({ctl}, {idx1}) # {desc}'):
    if len(ctl) == 0:
      self.apply1(gate, idx1, desc)
      return
    if len(ctl) == 1:
      self.applyc(gate, ctl[0], idx1, desc)
      return

    # Compute the predicate.
    self.ccx(ctl[0], ctl[1], aux[0])
    aux_idx = 0
    for i in range(2, len(ctl)):
      self.ccx(ctl[i], aux[aux_idx], aux[aux_idx+1])
      aux_idx = aux_idx + 1

    # Use predicate to single-control qubit at idx1.
    self.applyc(gate, aux[aux_idx], idx1, desc)

    # Uncompute predicate.
    aux_idx = aux_idx - 1
    for i in range(len(ctl)-1, 1, -1):
      self.ccx(ctl[i], aux[aux_idx], aux[aux_idx+1])
      aux_idx = aux_idx - 1
    self.ccx(ctl[0], ctl[1], aux[0])
```

3.3.7 Swap Operations

We also add implementations of the Swap gate (`swap`) and the controlled Swap gate (`cswap`), as described earlier in Sections 2.8.2 and 2.8.3. The `cswap` gate will be used later in Section 11.7 on quantum order finding, which is part of Shor's algorithm. It is easy to implement by simply changing the `cx` gates in a Swap gate to double-controlled `ccx` gates.

```
def swap(self, idx0: int, idx1: int) -> None:
  self.cx(idx1, idx0)
  self.cx(idx0, idx1)
  self.cx(idx1, idx0)

def cswap(self, ctl: int, idx0: int, idx1: int) -> None:
  self.ccx(ctl, idx1, idx0)
  self.ccx(ctl, idx0, idx1)
  self.ccx(ctl, idx1, idx0)
```

3.3.8 Measurement

As the final construct, we have to wrap the measurement operator, which is done in this straightforward way:

```
def measure_bit(self, idx: int, tostate: int = 0,
                collapse: bool = True) -> (float, state.State):
  return ops.Measure(self.psi, idx, tostate, collapse)
```

Note that here, we construct a full-matrix measurement operator, meaning that this method of measuring will not scale. Fortunately, in many cases, we don't have to perform an actual measurement to determine a most likely measurement outcome. We can just look at the state vector and find the one state with the highest probability – we say we perform *measurement by peek-a-boo*.

3.4 Intermediate Representation (IR)

As mentioned above, adding the ability to represent a circuit in an internal data structure enables useful capabilities, which we will study in this brief section. We start by outlining the IR data structure and show how it can be used in various flexible ways to deal with subcircuits. Then, we show how to use it to transpile a circuit to other quantum frameworks in Section 3.4.7.

3.4.1 IR Nodes

An IR *node* holds all the information that defines an individual operation, such as the gate type, control qubit, target qubit, and values such as rotation angles. Nodes are represented by a Python class that holds all these values. A single class type is sufficient to represent all possible node types – we keep it simple. To make working with this basic implementation more comfortable, we add functions to check for specific properties. We also add a notion of *sections* to aid with debugging (but we will not elaborate on this feature here).

Find the code
In file src/lib/ir.py

```
class Op(enum.Enum):
  SINGLE = 1
  CTL = 2

class Node:
  def __init__(self, opcode, name, idx0=0, idx1=None, val=None):
    self._opcode = opcode
    self._name = name
    self._idx0 = idx0
    self._idx1 = idx1
    self._val = val

  def is_single(self):
    return self._opcode == Op.SINGLE

  def is_ctl(self):
    return self._opcode == Op.CTL
```

This simple node implementation contains all the information needed to specify single-qubit and controlled gates, as well as gates with an optional parameter. The trivial function `is_single` checks whether a node represents a single-qubit gate, and `is_ctl` checks for controlled gates.

3.4.2 IR Base Class

The `Ir` class maintains a single list of nodes and has member functions to add single-qubit gates and controlled gates. It also offers the function `reg` to create a quantum register.

```
class Ir:
  def __init__(self):
    self._ngates = 0  # gates in this IR
    self.gates = []   # [] of gates
    self.regs = []    # [] of tuples (global reg index, name, reg index)
    self.nregs = 0    # number of registers
    self.regset = []  # [] of tuples (name, size, reg) for registers

  def reg(self, size, name, register):
    self.regset.append((name, size, register))
    for i in range(size):
      self.regs.append((self.nregs + i, name, i))
    self.nregs += size

  def single(self, name, idx0, val=None):
    self.gates.append(Node(Op.SINGLE, name, idx0, None, val))
    self._ngates += 1
```

```
def controlled(self, name, idx0, idx1, val=None):
  self.gates.append(Node(Op.CTL, name, idx0, idx1, val))
  self._ngates += 1

@property
def ngates(self):
  return self._ngates
```

3.4.3 Circuits of Circuits

Using this simple IR we can now introduce the notion of a *subcircuit*, which can be stored, combined, and executed later in flexible ways. The member function `sub` creates a completely independent new circuit. The member `sub_circuits` is only used for printing and debugging.

```
def sub(self, name: str = ''):
  sub = qc(f'inner_{self.sub_circuits}{name}', eager=False)
  self.sub_circuits += 1
  return sub
```

To execute a circuit in the context of another circuit, the member function `qc` is used. For a main circuit `main` and a subcircuit `sub`, the subcircuit can be invoked with `main.qc(sub)`. The `qc` function simply replays all the gates in the subcircuit in the parent circuit at an optional offset. For example, we can create a main circuit and a subcircuit and replay this subcircuit three times with this code:

```
main = circuit.qc('main circuit, eager execution')
[... add gates to main, eager]

sub1 = circuit.sub('subcircuit')
[... add gates to sub1, non-eager]

# Now add three copies of sub1 to main (eager),
# all at different offsets 0, 1, and 2:
main.qc(sub1, 0)
main.qc(sub1, 1)
main.qc(sub1, 2)
```

To reiterate, in non-eager mode the subcircuit is only being constructed, not executed. The gates and their order are only recorded for replay later. This behavior is controlled by setting parameter `eager` to `False` in the `sub` function.

3.4.4 Uncomputation with Inverse Circuit

A second useful capability of the IR makes the uncomputation from Section 2.12 easy and error-free. To obtain the inverse of a gate sequence, we reverse the stored list of

gates in the IR and replace each gate with its adjoint. To aid debugging, we decorate the textual gate names with a `'*'`. The implementation is simple and elegant:

```
def inverse(self):
  """Return, but don't apply, the inverse circuit."""

  newqc = qc(self.name, eager=False)
  for gate in self.ir.gates[::-1]:
    val = -gate.val if gate.val else None
    if gate.is_single():
      newqc.apply1(gate.gate.adjoint(), gate.idx0, gate.name + '*',
                   val=val)
    if gate.is_ctl():
      newqc.applyc(
          gate.gate.adjoint(), gate.ctl, gate.idx1, gate.name + '*',
          val=val
      )
  return newqc
```

In the example from the prior section, to reverse the application of the three subcircuits in the code example, we can now use the following code:

```
# Create an inverse copy of sub1 (which is still non-eager)
sub1_inv = sub1.inverse()

# Now add three copies of sub1_inv to main (eager),
# at the reverted list of offsets:
main.qc(sub1_inv, 2)
main.qc(sub1_inv, 1)
main.qc(sub1_inv, 0)
```

3.4.5 Controlling Subcircuits

Another useful feature of the IR is that it allows controlling a whole subcircuit by another qubit with function `control_by`. This function iterates over all gates in the circuit. Individual single-qubit gates are converted to a controlled gate, and controlled gates are converted to multi-controlled gates, as shown in this implementation:

```
def control_by(self, ctl: int):
  res = ir.Ir()
  for _, gate in enumerate(self.ir.gates):
    if gate.is_single():
      gate.to_ctl(ctl)
      res.add_node(gate)
      continue
    if gate.is_ctl():
      sub = qc('multi', eager=False)
      sub.multi_control(
```

```
            [ctl, gate.ctl], gate.idx1, None, gate.gate, gate.desc
        )
        for gate in sub.ir.gates:
          res.add_node(gate)
    self.ir = res
```

3.4.6 Inverting a Register

The individual qubits in a register can be *inverted*, or flipped vertically, by inverting the indices of the gates applied to the register. For example, assume that a register has four qubits with local indices $(0,1,2,3)$ and global indices $(2,3,4,5)$. Further, assume a single gate is applied to local qubit 0 in the register (global qubit 2) and a controlled gate from the register's local qubit 1 to qubit 3. The `invert` procedure will change the ordering and apply the single gate to local qubit 3 in the register (global qubit 5) and change the controlled gate to go from local qubit 0 to qubit 2.

```
def invert(self, reg):
  def swap_bits(reg, idx):
    d = int(len(reg) - idx - 1)
    tmp = reg[idx]
    reg[idx] = reg[d]
    reg[d] = tmp

  for gate in self.ir.gates:
    swap_bits(reg, gate.idx0, reg.size)
    if gate.is_ctl():
      swap_bits(reg, gate.idx1, reg.size)
```

3.4.7 Transpilation

Finally, the IR can be used to output a circuit in a format that can serve as input to other frameworks. For example, to produce a simple QASM format (Svore et al., 2006), the code traverses the list of nodes and outputs the nodes with their names as found. Fortunately, the names chosen for the operators already match the QASM specification.[1]

Typically, one needs a few helper functions to make the output more readable. For example, the code below uses `helper.pi_fractions` to convert values into fractions of π. There are several other transpilers in the file `src/lib/dumpers.py`, including transpilers to IBM's Qiskit, Google's Cirq, a rudimentary LaTeX converter,[2] a transpiler to our own `libq` implementation, which is detailed in Section 3.9, and a minimal text generator.

[1] This is not a coincidence.
[2] We used this transpiler quite often in this book to typeset larger circuits.

Find the code
In file src/lib/dumpers.py

```
def qasm(ir) -> str:
  res = 'OPENQASM 2.0;\n'
  for regs in ir.regset:
    res += f'qreg {regs[0]}[{regs[1]}];\n'
  res += '\n'

  for op in ir.gates:
    if op.is_gate():
      res += op.name
      if op.val is not None:
        res += '({})'.format(helper.pi_fractions(op.val))
      if op.is_single():
        res += f' {reg2str(ir, op.idx0)};\n'
      if op.is_ctl():
        res += f' {reg2str(ir, op.ctl)},{reg2str(ir, op.idx1)};\n'
  return res
```

That's it! It is really that simple. It is just an iteration over all gates, where each gate is printed according to its type. Here is an output example. It shows a few quantum registers at the top, a few Hadamard gates on register `q1`, a couple of controlled U_1 gates followed by measurement operators:

```
OPENQASM 2.0;
qreg q2[4];
qreg q1[8];
qreg q0[6];
creg c0[8];
h q1[0];
h q1[1];
h q1[2];
[...]
cu1(-pi/64) q1[7],q1[1];
cu1(-pi/128) q1[7],q1[0];
h q1[7];
measure q1[0] -> c0[0];
measure q1[1] -> c0[1];
[...]
```

QASM is fairly simple and supported by several other infrastructures. It is very useful for debugging complex algorithms, as it allows for direct comparisons with results obtained by other infrastructures.[3]

[3] I used it extensively during the development of the algorithms.

3.5 Fast Gate Application

Up to this point, we have applied a gate by first tensoring it with identity matrices and then applying the resulting large matrix to a full state vector. As described in Section 3.1, this does not scale well beyond a small number of qubits. For ten qubits, the operator matrix is a 1024×1024 matrix, which requires 1024^2 multiplications and additions for a matrix–vector multiplication (1024^3 if you want to multiply it with another operator of this size). Can we devise a more efficient way to apply gates? Yes, we can.

Let us analyze what happens during gate application. To start the analysis, we create a pseudo state vector containing a sequence of numbers. We do *not* normalize the vector because this makes it easier to visualize what happens to the vector elements as the gates are applied to the individual qubits.

```
    qc = circuit.qc('test')
    qc.arange(4)
    print(qc.psi)
>>
[ 0.+0.j  1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+0.j  7.+0.j
  8.+0.j  9.+0.j 10.+0.j 11.+0.j 12.+0.j 13.+0.j 14.+0.j 15.+0.j]
```

Now we apply the *X* gate to qubits 0 to 3, one by one, always starting with a freshly created vector. The *X* gate is interesting because it multiplies the entries in the state vector by 0 and 1, causing the appearance of values being swapped. This is similar to the application of the *X* gate to a regular qubit, which seems to "flip" $|0\rangle$ and $|1\rangle$:

```
    # Let's try this for qubits 0 to 3.
    for idx in range(4):
      qc = circuit.qc('test')
      qc.arange(4)
      qc.x(idx)
      print('Applied X to qubit {}:\n {}'.format(idx, qc.psi))
```

We start by applying the *X* gate to qubit 0 and get:

```
Applied X to qubit 0:
 [ 8.+0.j  9.+0.j 10.+0.j 11.+0.j 12.+0.j 13.+0.j 14.+0.j 15.+0.j
   0.+0.j  1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+0.j  7.+0.j]
```

The right half of the vector was swapped with the left half. Let us try the next qubit index. Applying the *X* gate to qubit 1 results in the following:

```
Applied X to qubit 1:
 [ 4.+0.j  5.+0.j  6.+0.j  7.+0.j  0.+0.j  1.+0.j  2.+0.j  3.+0.j
  12.+0.j 13.+0.j 14.+0.j 15.+0.j  8.+0.j  9.+0.j 10.+0.j 11.+0.j]
```

Now chunks of four vector elements are being swapped. The elements 4–7 swap position with elements 0–3, and the elements 12–15 swap position with elements 8–11. A pattern is emerging. Let us apply the X gate to qubit 2:

```
Applied X to qubit 2:
 [ 2.+0.j  3.+0.j  0.+0.j  1.+0.j  6.+0.j  7.+0.j  4.+0.j  5.+0.j
  10.+0.j 11.+0.j  8.+0.j  9.+0.j 14.+0.j 15.+0.j 12.+0.j 13.+0.j]
```

The pattern continues; now groups of two elements are swapped. And finally, for qubit 3, we see that individual qubits are being swapped:

```
Applied X to qubit 3:
 [ 1.+0.j  0.+0.j  3.+0.j  2.+0.j  5.+0.j  4.+0.j  7.+0.j  6.+0.j
   9.+0.j  8.+0.j 11.+0.j 10.+0.j 13.+0.j 12.+0.j 15.+0.j 14.+0.j]
```

We recognize a clear "power-of-2" pattern. The state vector for four qubits and $2^4 = 16$ elements. To express the numbers from 0–15, we need four classical bits: $b_3 b_2 b_1 b_0$. Recall that we enumerate qubits from left to right and classical bits from right to left. Also, remember that we are using the X gate, which is a permutation matrix and multiplies with values of 0 and 1, leaving the impression of swapping elements. The mechanism works for all single-qubit gates; we just use the X gate for effective visualization.

- **Qubit 0**. Applying the X gate to qubit 0 swaps the first half of the state vector with the second half. If we interpret vector indices as binary numbers, the state elements with indices that had bit 3 set (most significant bit) switched position with the indices that did not have bit 3 set. Positions 8–15 had bit 3 set and switched with positions 0–7, which did not have bit 3 set. Two blocks of eight elements were switched.
- **Qubit 1**. Applying the X gate to qubit 1 swaps the second quarter of the state vector with the first quarter and the fourth quarter with the third. Consequently, the vector elements with indices that had bit 2 set switched with those that had bit 2 not set, "bracketed" by the bit pattern in bit 3. What does it mean that an index is bracketed by a higher-order bit? It simply means that the higher-order bit did not change, it remained 0 or 1. Only the lower-order bits switch between 0 and 1. Here, four-element blocks were switched. There are four such blocks for qubit 1 – two blocks where binary bit 3 was 0 and another two blocks where bit 3 was 1.[4]

[4] This is admittedly confusing. It doesn't help that qubits are numbers from 0 to 3 and the binary bits from 3 to 0.

- **Qubit 2**. Applying the X gate to qubit 2 swaps the second eighth of the state vector with the first, the fourth with the third, the sixth with the fifth, and the eighth with the seventh. As above, the vector elements with indices with bit 1 set switched places with those that did not have bit 1 set. This swapping is bracketed by the bit pattern in bit 2 and further bracketed by the bit patterns of bit 3.
- **Qubit 3**. Finally, applying the X gate to qubit 3 now swaps single elements: element 0 with element 1, element 2 with element 3, and so on.

We can put this pattern in a closed form by looking at the binary bit pattern for the state vector indices (Smelyanskiy et al., 2016). Let us introduce this *bit index* notation for a state with a classical binary bit representation (where we omit the state kets $|\cdot\rangle$ for ease of notation):

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots\beta_0}.$$

If we expect a specific 0 or 1 at a given bit position k, we specify this bit value with this notation:

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0},$$
$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0}.$$

Applying a single-qubit gate to qubit k in an n-qubit state (qubits 0 to $n-1$) applies the gate to a pair of amplitudes whose indices differ in bit $(n-1-k)$ in binary representation. In our first example, we have four qubits. Qubit 0 translates to classical bit 3 in this notation, and qubit 3 corresponds to classical bit 0. We apply the X gate to the probability amplitudes that correspond to the states where the bit index switches between 0 and 1, thus swapping chunks of the state vector.

Suppose we want to apply a single-qubit gate G to a qubit of a system in state $|\psi\rangle$, where G is a 2×2 matrix. Let us name the four elements of the matrix G_{00}, G_{01}, G_{10}, and G_{11}, corresponding to the top left, top right, bottom left, and bottom right.

Applying a gate G to the kth qubit corresponds to the following recipe. This notation indicates looping over the full state vector. All vector elements whose indices match the specified bit patterns are multiplied by the gate elements G_{00}, G_{01}, G_{10}, and G_{11}, as specified in this recipe:

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0} = G_{00}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0} + G_{01}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0},$$
$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0} = G_{10}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0} + G_{11}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0}.$$

For controlled gates, the pattern can be extended. We have to ensure that the control bit c is set to 1 and only apply the gates to states for which this is the case:

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 0_k\ldots\beta_0} = G_{00}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 0_k\ldots\beta_0} + G_{01}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 1_k\ldots\beta_0},$$
$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 1_k\ldots\beta_0} = G_{10}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 0_k\ldots\beta_0} + G_{11}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 1_k\ldots\beta_0}.$$

In the implementation, we have to be mindful of the bit orderings. Qubit 0 is the topmost qubit, but for the classical bits, as is common, bit 0 is the least significant bit. This means that in the implementation we have to reverse the bit indices.

To apply a single gate, we add this function to our implementation of states in file `src/lib/state.py` (with `1<<n` as an optimization for `2**n`):

Find the code
In file `src/lib/state.py`

```
def apply(self, gate: ops.Operator, index: int) -> None:
  # To maintain qubit ordering in this infrastructure,
  # index needs to be reversed.
  index = self.nbits - index - 1
  pow_2_index = 1 << index

  for g in range(0, 1 << self.nbits, 1 << (index + 1)):
    for i in range(g, g + pow_2_index):
      t1 = gate[0, 0] * self[i] + gate[0, 1] * self[i + pow_2_index]
      t2 = gate[1, 0] * self[i] + gate[1, 1] * self[i + pow_2_index]
      self[i] = t1
      self[i + pow_2_index] = t2
```

The implementation for controlled gates is very similar, but note the additional `if` statement in the code, which checks for the control bit.

```
def applyc(self, gate: ops.Operator, ctrl: int, target: int) -> None:

  index = self.nbits - target - 1
  ctrl = self.nbits - ctrl - 1
  pow_2_index = 1 << index

  for g in range(0, 1 << self.nbits, 1 << (qbit+1)):
    idx_base = g * (1 << self.nbits)
    for i in range(g, g + pow_2_index):
      if (idx_base + i) & (1 << ctrl):
        t1 = gate[0, 0] * self[i] + gate[0, 1] * self[i + pow_2_index]
        t2 = gate[1, 0] * self[i] + gate[1, 1] * self[i + pow_2_index]
        self[i] = t1
        self[i + pow_2_index] = t2
```

We could now go ahead and add these routines to the quantum circuit class, but wait – we can do even better and accelerate these routines with C++! This will be the topic of Section 3.6.

3.6 Accelerated Gate Application

We now understand how to apply gates to a state vector with linear complexity, but the code is written in Python, which is known to run slower than C++. In order to add a few more qubits to our simulation capabilities, we need to accelerate the gate application further. To achieve this, we implement the gate application functions in

C++ and import them into Python using standard extension techniques. The overhead of calling these C++ functions from Python is minimal, which means we can continue to program in Python but with the execution speed of C++.

This section contains a lot of C++ code. For these routines, the C++ code executes about 100× faster than the Python code, giving us the ability to simulate six or seven additional qubits. The basic principles are shown in Section 3.5. We detail this code because it might be useful to readers without experience extending Python with C++. The implementations of `apply1` and `applyc` are quite similar, so we only show the code for the former.

In the code, `<path>` to `numpy` must be set correctly to point to a local setup. The open-source repository will have the latest instructions on how to compile and use this Python extension. We also want to support both `float` and `double` complex numbers and use C++ templates to specialize the code for these two types. Since building C++ extensions can be difficult on some platforms, we provide Python fallback implementations. This enables all our open-source algorithms to run correctly, if at slower execution speed.

Find the code
In file `src/lib/xgates.cc`

```
// Make sure this header can be found:
#include <Python.h>

#include <stdio.h>
#include <stdlib.h>
#include <complex>

// Configure the path, likely found in the BUILD file:
#include "<path>/numpy/core/include/numpy/ndarraytypes.h"
#include "<path>/numpy/core/include/numpy/ufuncobject.h"
#include "<path>/numpy/core/include/numpy/npy_3kcompat.h"

typedef std::complex<double> cmplxd;
typedef std::complex<float> cmplxf;

// apply1 applies a single gate to a state.
// 2x2 gates are flattened to a 1x4 array:
//     a  b
//     c  d   ->  a b c d
template <typename cmplx_type>
void apply1(cmplx_type *psi, cmplx_type gate[4],
            int nbits, int tgt) {
  tgt = nbits - tgt - 1;
  int q2 = 1 << tgt;
  for (int g = 0; g < 1 << nbits; g += (1 << (tgt+1))) {
    for (int i = g; i < g + q2; ++i) {
      cmplx_type t1 = gate[0] * psi[i] + gate[1] * psi[i + q2];
      cmplx_type t2 = gate[2] * psi[i] + gate[3] * psi[i + q2];
      psi[i] = t1;
```

```
      psi[i + q2] = t2;
    }
  }
}
```

To extend Python and make this extension loadable as a shared module, we add standard Python bindings for single-qubit gates. Function `apply1_python` obtains C++ pointers to the arguments and calls the C++ `apply1` function. The function `apply1_call` verifies the type of state parameter `param_psi`, which is identified by parameter `bit_width`, and calls the correctly typed flavors of the templatized C++ function:

```
template <typename cmplx_type, int npy_type>
void apply1_python(PyObject *param_psi, PyObject *param_gate,
                   int nbits, int tgt) {
  PyArrayObject *psi_arr =
    PyArrayObject*) PyArray_FROM_OTF(param_psi, npy_type, NPY_IN_ARRAY);
  cmplx_type *psi = ((cmplx_type *)PyArray_GETPTR1(psi_arr, 0));

  PyArrayObject *gate_arr =
    PyArrayObject*) PyArray_FROM_OTF(param_gate, npy_type, NPY_IN_ARRAY);
  cmplx_type *gate = ((cmplx_type *)PyArray_GETPTR1(gate_arr, 0));

  apply1<cmplx_type>(psi, gate, nbits, tgt);
  Py_DECREF(psi_arr);
  Py_DECREF(gate_arr);
}

static PyObject *apply1_call(PyObject *dummy, PyObject *args) {
  PyObject *param_psi = NULL;
  PyObject *param_gate = NULL;
  int nbits, tgt, bit_width;

  if (!PyArg_ParseTuple(args, "OOiii", &param_psi, &param_gate,
                        &nbits, &tgt, &bit_width))
    return NULL;
  if (bit_width == 128) {
    apply1_python<cmplxd, NPY_CDOUBLE>(param_psi,
                                       param_gate, nbits, tgt);
  } else {
    apply1_python<cmplxf, NPY_CFLOAT>(param_psi,
                                      param_gate, nbits, tgt);
  }
  Py_RETURN_NONE;
}
```

This is followed by the standard functions that the Python interpreter will call when importing a module. We register the Python wrappers in a module named `xgates` with standard boilerplate code:

```
// Python boilerplate to expose above wrappers to programs.
static PyMethodDef xgates_methods[] = {
    {"apply1", apply1_call, METH_VARARGS,
     "Apply single-qubit gate, complex double"},
    {NULL, NULL, 0, NULL}};

// Give a name to the module (xgates) and register above array.
static struct PyModuleDef xgates_definition = {
  PyModuleDef_HEAD_INIT,
  "xgates",
  "Python extension to accelerate quantum simulation math",
  -1,
  xgates_methods
};

// Standard registering function, identified by Python by name (xgates).
PyMODINIT_FUNC PyInit_xgates(void) {
  Py_Initialize();
  import_array();
  return PyModule_Create(&xgates_definition);
}
```

Python typically finds extensions with the help of an environment variable. For example, on Linux:

```
export PYTHONPATH=path_to_xgates.so
```

Alternatively, you can extend Python's module search path programmatically with code like this (of course, this will make code changes necessary to adjust the path):

```
import sys
sys.path.append('/path/to/search')
```

3.7 Circuits Finally Finalized

With our accelerated implementation, we can finally finish the gate application functions in the quantum circuit `qc` class. Single-qubit gates can be applied to an individual qubit, a whole register, or a list of qubits. For each of these sets of indices, gates are added to the IR and, in eager mode, directly applied using the accelerated routines:[5]

[5] In the code in the repository, you will find another indirection to ensure that everything runs successfully, even if `xgates` could not be built or found. In that case, a message will warn about potentially degraded execution speed.

```
def apply1(self, gate: ops.Operator, idx_set, name: str = None, *,
           val: float = None):
  indices = []
  if isinstance(idx_set, int):
    indices.append(idx_set)
  if isinstance(idx_set, state.Reg):
    indices += idx_set.reg
  if isinstance(idx_set, list):
    indices += idx_set

  for idx in indices:
    if self.build_ir:
      self.ir.single(name, idx, gate, val)
    if self.eager:
      xgates.apply1(self.psi, gate.reshape(4), self.psi.nbits, idx,
                    tensor.tensor_width())
```

Controlled qubits can be applied to an individual qubit or single-qubit register. When the controlling qubit is specified as a single-element list (as above), the gate will be a Controlled-By-0 gate. Similarly to the above, the IR is constructed, and the gates are applied in eager mode only.

```
def applyc(self, gate: ops.Operator, ctl: int, idx: int,
           name: str = None, *, val: float = None):
  if isinstance(idx, state.Reg):
    assert idx.size == 1, 'Controlled n-qbit register not supported'
    idx = idx[0]

  ctl_qubit, by_0 = self._ctl_by_0(ctl)
  self.x(ctl_qubit, by_0)
  if self.build_ir:
    self.ir.controlled(name, ctl_qubit, idx, gate, val)
  if self.eager:
    xgates.applyc(self.psi, gate.reshape(4), self.psi.nbits, ctl_qubit,
                  idx, tensor.tensor_width())
  self.x(ctl_qubit, by_0)
```

You can refer back to Figure 3.2 and the corresponding code to see an example of how this code can be invoked from Python.

3.8 Premature Optimization, First Act

Looking at the standard gates, we find many 0s and 1s, which means that several gate applications may run faster if we optimize for these special cases. Emphasis on *should*. Let us run an experiment to verify this assumption.

We start by constructing a benchmark to compare the general and fast gate application routines from Section 3.5 with a specialized function for the X gate, which

Table 3.1. Benchmark results (program output), comparing hand-optimized and nonoptimized gate application routines.

Benchmark	Time(ns)	CPU(ns)	Iterations
BM_apply_single	116403527	116413785	24
BM_apply_single_opt	132820169	132829412	21
BM_apply_controlled	81595871	81600200	34
BM_apply_controlled_opt	89064964	89072559	31

has two 0s and two 1s. Multiplications by 0 can be replaced by just 0, additions of 0 and multiplications by 1 can also be removed. For fast gate application routines specialized in this way for the *X* gate, a total of four multiplications, two additions, and some memory accesses per single qubit should be saved. This is the original inner loop:

```
for (int i = g; i < g + q2; ++i) {
  cmplx t1 = gate[0][0] * psi[i] + gate[0][1] * psi[i + q2];
  cmplx t2 = gate[1][0] * psi[i] + gate[1][1] * psi[i + q2];
  psi[i] = t1;
  psi[i + q2] = t2;
}
```

And this is the optimized version of the loop:[6]

```
for (int i = g; i < g + q2; ++i) {
  cmplx t1 = psi[i + q2];
  cmplx t2 = psi[i];
  psi[i] = t1;
  psi[i + q2] = t2;
}
```

The results of comparing the two implementations are in Table 3.1. Recall our hypothesis that the optimized version would be faster because it executes fewer multiplications and additions. The column `Iterations` shows iterations per second; higher is better. Surprisingly, you can see that the specialized version runs about 10% *slower*! For the given x86 platform, the compiler was able to vectorize the unspecialized version, leading to a slightly higher overall throughput.[7]

In summary, we found a way to apply gates with linear complexity over the size of the state vector and accelerated it by a significant factor with C++. This infrastructure is sufficient for all the algorithms in this book.

[6] I wonder whether classical compilers can be made smart enough to perform this transformation "automagically."

[7] To riff on a quote that is (potentially incorrectly) ascribed to Lenin: Intuition is good, but verification is better.

There are other ways to simulate quantum algorithms (Altman et al., 2021), as we discussed at the end of Section 3.1. A specifically interesting methodology represents states *sparsely*. Indeed, for many circuits, this is the most efficient representation. We give a brief overview of it in Section 3.9 and a full implementation will be provided in the Appendix.

3.9 Sparse Representation

So far, our data structure for representing quantum states is a dense array that holds all the probability amplitudes of the superimposed basis states, where the amplitude for a specific basis state can be found via binary indexing. However, for many circuits and algorithms, a high percentage of states can have a probability equal to or very close to zero. Storing these 0-states and applying gates to them will have no effect and is wasteful. This fact can be exploited with a sparse representation. An excellent reference implementation of this principle can be found in the venerable open-source library `libquantum` (Butscher and Weimer, 2013).

Find the code
In file `src/libq`

We reimplement the core ideas of that library as they relate to this book; `libquantum` addresses other aspects of quantum information, which we do not cover. Therefore, we name our implementation `libq` to distinguish it from the original. The original library is in plain C, but our implementation was moderately updated with C++ for improved readability and performance. We maintain some of the C naming conventions for key variables and functions to help with direct code comparisons.

Here is the core idea: Assume that we have a state of n qubits, all initialized to be in the state $|0\rangle$. The dense representation stores 2^n complex numbers in the state vector, where only the very first entry is a 1 and all other values are 0, corresponding to state $|00\ldots0\rangle$.

The `libq` library turns this on its head. Basis states are encoded as simple binary bits in an integer (currently up to 64 qubits, but this can be extended), where the binary digits 0s and 1s correspond to states $|0\rangle$ and $|1\rangle$. Each of these bit combinations stored in the integer variable is paired with a probability amplitude. Only states with nonzero amplitudes are being stored. In the above example, `libq` would store the basis state $|00\ldots0\rangle$ with amplitude 1 as the only Python tuple `(0b00...0, 1.0)`, indicating that the only state with nonzero probability is $|00\ldots0\rangle$. For 53 qubits, the full-state representation would require 72 petabytes of memory. In contrast, the sparse representation only requires a total of 16 bytes if the amplitude is stored as a double precision value (only 12 bytes are needed if we use 4-byte floating point values).[8]

Applying a Hadamard gate to a qubit will put the state in superposition. In `libq`, there will now be two states with nonzero amplitudes. For example, applying the Hadamard gate to the right-most qubit will lead to the two basis states with equal amplitudes:

[8] 15 orders of magnitude — *Is that all you got?*

$$|00\ldots00\rangle \quad \text{with probability 50\%, and}$$
$$|00\ldots01\rangle \quad \text{with probability 50\%.}$$

Consequently, `libq` now stores two tuples, each with a probability amplitude of $1/\sqrt{2}$, using 32 bytes (or 24 bytes with 4-byte floats).

During the execution of a circuit, superposition is generated and destroyed. Individual states become probable and no longer probable. A key aspect of `libquantum` is that gates are recognized as producing or destroying superposition and handled accordingly. Furthermore, it filters out all states with amplitudes close to 0 after the application of superposition generating gates. This reduces the number of stored tuples and accelerates future gate applications.

The gate application itself becomes very fast. For example, assume that we need to apply the *X* gate to the least significant qubit. In the dense case, the entire state vector needs to be traversed and modified, as described in Section 3.6 on accelerated gate application.

In `libq`, only a bit flip is needed. In the example above, assuming an initial state of all $|0\rangle$, applying the *X* gate to the least significant qubit means that we only have to flip the least significant bit in the bit mask; the tuple `(0x00...00, 1.0)` becomes `(0x00...01, 1.0)`. This is dramatically faster than having to traverse and modify a potentially very large state vector, especially if the number of nonzero probability states is low. To maintain the state tuples, we need to support two main operations:

- Iterate over all available state tuples.
- Find or create a specific state tuple.

The original `libquantum` implements a hash table to manage the tuples, and, as we will see, despite the favorable performance characteristics of hash tables, it ultimately remains the performance bottleneck in the implementation. Our `libq` moderately improves this core data structure.

The implementation of `libq` consists of just about 500 lines of C++ code. A detailed, annotated description, which also includes optimization wins and fails, can be found in the Appendix.

This design also has downsides, which may prevent it from scaling to very large numbers of qubits or circuits with a high percentage of nonzero probabilities. Individual states are efficiently encoded as tuples of a bit mask to encode a state and a probability amplitude. However, there are additional data structures, such as the hash table, to maintain existing states. The memory requirement per individual basis state is higher than that in a full-state representation. This means there is a crossover point where the sparse representation becomes *less* efficient than the full state representation. In particular, it appears to perform poorly for the quantum random algorithms that we discuss in Section 5.3.2.

Another downside might arise from the way the hash table is used to store the states. At some size threshold, the hash table's random memory accesses will be outperformed by linear memory accesses, which benefit from caches and can be prefetched effectively. Furthermore, hash table entries might be distributed unpredictably across machines in a distributed computing environment. The gate application might thus incur prohibitively high communication costs.

Benchmarking

Here, we provide anecdotal evidence for the efficiency of the sparse representation. A full performance evaluation is ill-advised in a book like this – the results will be out of date and no longer relevant by the time you read this.

The most complex algorithm in this book is Shor's integer factorization algorithm (Section 11.6). The quantum part of the algorithm is called *order finding*. Factoring the number 15 requires 18 qubits and 10,533 gates; factoring 21 requires 22 qubits and 20,671 gates, and factoring 35 requires 26 qubits and 36,373 gates. We run this circuit in two different ways:

- Run it as is, using the accelerated quantum circuit implementation.
- Construct the circuit non-eagerly and transpile it to `libq`. We described the transpilation in Section 3.4.7. The output is a C++ source file, which is compiled and linked with the `libq` library to produce an executable.

Both versions will compute the same result; the textual output differs and shows the maximum number of nonzero states reached during execution. Factoring the number 21 with 22 qubits, we get the following output. A maximum of only 1.6% of all possible states ever obtained a nonzero probability at one point or the other during execution.

```
# of qubits        : 22
# of hash computes : 2736
Maximum # of states: 65536, theoretical: 4194304, 1.562%
States with nonzero probability:
 0.499966 +0.000000i|4> (2.499658e-01) (|00 0000 0000 0000 0000 0100>)
 0.000001 -0.000000i|32772> (6.14857e-13) (|00 0000 1000 0000 0000 0100>)
-0.499970 +0.000000i|65536> (2.4997e-01) (|00 0001 0000 0000 0000 0000>)
 0.499966 +0.000000i|65540> (2.49966e-01) (|00 0001 0000 0000 0000 0100>)
 0.000001 -0.000000i|98308> (6.14856e-13) (|00 0001 1000 0000 0000 0100>)
 0.499970 -0.000000i|0> (2.4997e-01) (|00 0000 0000 0000 0000 0000>)
```

The `libq` version runs in less than five seconds on a modern workstation, while the circuit version takes about 2.5 minutes, a speed-up of roughly 25 times. Factoring the number 35 with 26 qubits, the `libq` version runs for about 3 minutes, while the full-state simulation takes about an hour. Again, another solid acceleration of about 20 times. We ignore the compilation times for the generated C++ code, which we would have to include in an actual scientific evaluation.[9]

[9] Which this is not.

4 Quantum Tools and Techniques

In quantum computing, there are several standard techniques for working with states and operators and to assist with debugging of computational processes. In this section, we detail several of these mathematical tools. Some of the material may be confusing for novices. If you fall into this group, do not be discouraged. Many of the algorithms presented in this book can be understood without this material. However, if you seek a deeper understanding of the relevant linear algebra, this material is for you.

4.1 Spectral Theorem for Normal Matrices

This section discusses the important *spectral theorem* in an informal and code-based manner. We will use it in several algorithms in this book. Recall from Section 1.8 that the Hermitian and unitary matrices are special cases of normal matrices for which $AA^\dagger = A^\dagger A$. The complex spectral theorem states that any normal matrix is diagonalizable by some unitary matrix. Since we are not going too deeply into linear algebra, we will use a more targeted version of this theorem for Hermitian matrices:

THEOREM: (Spectral Theorem) *For any Hermitian matrix H,*

- *All eigenvalues of H are real.*
- *The eigenvectors corresponding to distinct eigenvalues are orthogonal.*[1]
- *The eigenvectors form a basis for the vector space of H (we mostly ignore the linear algebra of vector spaces; we also ignore this part of the theorem).*

Furthermore, for any Hermitian[2] matrix H there exists a unitary matrix U with

$$HU = U\Lambda,$$

where the matrix Λ is a diagonal matrix with the real eigenvalues λ_i of H on the diagonal. The columns of U are the eigenvectors $|v_i\rangle$ of H (not of U). This means that we can write H in the spectral decomposition as

$$H = \sum_i \lambda_i |v_i\rangle\langle v_i|. \tag{4.1}$$

[1] Which we can normalize to get an orthonormal basis.
[2] The *general* spectral theorem states that this holds for any normal matrix.

If H is a unitary matrix, then all eigenvalues have the absolute value of 1. We already proved this in Section 1.8 on unitary matrices. The proofs for the other parts of the theorem can be found in existing material on linear algebra, for example, in Nielsen and Chuang (2011), so we will not repeat them here. Instead, we play with code to convince ourselves of these results. We create a unitary[3] operator U in the Python variable `umat` using `scipy` and make it Hermitian by computing $H = (U + U^\dagger)/2$ and storing the result in variable `hmat`:

Find the code
In file src/spectral_decomp.py

```
def spectral_decomp(ndim: int):
  u = scipy.stats.unitary_group.rvs(ndim)
  umat = ops.Operator(u)

  hmat = 0.5 * (umat + umat.adjoint())
  assert np.allclose(hmat, hmat.adjoint()), 'Something is wrong'
```

We compute eigenvalues and eigenvectors using `numpy` and check that the eigenvalues are real and that the eigenvectors are orthonormal:

```
  w, v = np.linalg.eig(hmat)
  for i in range(ndim):
    assert np.allclose(w[i].imag, 0.0), 'Non-real eigenvalue!'

  for i in range(ndim):
    for j in range(i + 1, ndim):
      dot = np.dot(v[:, i], v[:, j].conj())
      assert np.allclose(dot, 0.0, atol=1e-5), 'Not orthogonal'

  for i in range(ndim):
    dot = np.dot(v[:, i], v[:, i].conj())
    assert np.allclose(dot, 1.0, atol=1e-5) 'Not orthonormal'
```

Now we can write the matrix in the form of Equation (4.1) and verify that the decomposition is correct:

```
  x = np.matrix(np.zeros((ndim, ndim)))
  for i in range(ndim):
    x = x + w[i] * np.outer(v[:, i], v[:, i].conj())
  assert np.allclose(hmat, x, atol=1e-5), 'Spectral decomp failed.'
```

Spectral decomposition is powerful for many reasons. In particular, if we look at Equation (4.1) and squint our eyes, we can see that the matrix trace is independent of a chosen basis. It depends only on the eigenvalues. In fact, as we have already stated in Equation (1.8), the trace is the sum of the eigenvalues. We can also compute

[3] Note that this works for any square matrix.

the inverse of an invertible matrix (with nonzero eigenvalues) simply by using the reciprocals of the eigenvalues:

$$H = \sum_i \lambda_i |v_i\rangle\langle v_i| \quad \Leftrightarrow \quad H^{-1} = \sum_i \lambda_i^{-1} |v_i\rangle\langle v_i|.$$

```
x = np.matrix(np.zeros((ndim, ndim)))
for i in range(ndim):
  x = x + 1 / w[i] * np.outer(v[:, i], v[:, i].conj())
assert np.allclose(np.linalg.inv(hmat), x, atol=1e-5), 'Inverse Error.'
```

4.2 Density Matrices

So far, we have explored *pure* states which represent a single well-defined state of a quantum system. This formalism will carry us through most of the remainder of this book. However, the formalism is insufficient to answer the following questions.

- First, assume that we have an apparatus that does not just produce a single state but a statistical *mixture* of different states. Having a single mathematical formalism to describe such a system would be nice.
- Secondly, assume that we have an EPR pair β_{AB} of qubits as described in Section 2.11.3, where the qubit A may be physically separated from B. We should have a way to describe the individual qubits without having access to the other.

We need a better methodology for these cases, and the trick will be to describe states not just as vectors but as matrices. As hinted in Section 2.5, for a state vector $|\psi\rangle$, we construct a *density matrix* ρ by computing the outer product of the state $|\psi\rangle$ with itself as

$$\rho = |\psi\rangle\langle\psi|.$$

The density matrix for a pure state has a rank of 1 and also a trace of 1. If we measure a state $|\psi\rangle$ in a basis with a basis vector $|b\rangle$, we already know that the probability of measuring $|b\rangle$ is

$$\begin{aligned} |\langle b|\psi\rangle|^2 &= \langle b|\psi\rangle\langle\psi|b\rangle \\ &= \langle b|\rho|b\rangle. \end{aligned}$$

Let us assume that we have an apparatus that produces a state $|\psi_0\rangle$ with probability p_0 and another state $|\psi_1\rangle$ with probability p_1. The machine produces a *statistical mixture* of states. To represent the full system, we add the individual density matrices of the states weighted by their system-level probabilities as

$$\rho = p_0|\psi_0\rangle\langle\psi_0| + p_1|\psi_1\rangle\langle\psi_1|.$$

We can generalize this to any mixture of states as

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|.$$

This may look similar to the spectral decomposition from Section 4.1, but it is different, as the $|\psi_i\rangle$ are not necessarily basis states here. Any matrix with the following two properties can be considered a density matrix:

1. The matrix must be positive semidefinite with eigenvalues $\lambda_i \geq 0$.
2. The trace of the matrix must be 1.

A set of density matrices with their probabilities $\{(p_i, \rho_i)\}$ is called an *ensemble* of states. Density matrices are not unique and can result from different ensembles. For example, the matrix $I/2$ (with I being the identity matrix) is a valid density matrix. Suppose an apparatus generates states $|0\rangle$ and $|1\rangle$ with equal probability and a second apparatus generates states $|+\rangle$ and $|-\rangle$ with equal probability. In that case, their density matrices are the same and physically indistinguishable:

$$\begin{aligned}\frac{I}{2} &= \frac{1}{2}\big(|+\rangle\langle+| + |-\rangle\langle-|\big) \\ &= \frac{1}{2}\big(|0\rangle\langle 0| + |1\rangle\langle 1|\big).\end{aligned}$$

As mentioned above, if the rank of matrix ρ is 1, then ρ is a *pure* density matrix. Otherwise, it is a mixed density matrix. Furthermore, the trace operation gives us another mathematical definition of mixed and pure states with

$$\begin{aligned}\mathrm{tr}\big(\rho^2\big) &= 1 : \text{Pure state}, \\ \mathrm{tr}\big(\rho^2\big) &< 1 : \text{Mixed state}.\end{aligned}$$

This term $\mathrm{tr}(\rho^2)$ as a metric is also known as the *purity* of a state. For a single qubit, it can be visualized on the Bloch sphere. The Bloch vector for a pure state is located on the surface of the Bloch sphere, while for a mixed state, the Bloch vector is somewhere in the sphere's interior.

We apply an operator U with $|\psi'\rangle = U|\psi\rangle$ to modify a state vector. To apply an operator U to a state expressed as a density matrix, we must apply U from the left and $U^\dagger$ from the right, as $\rho' = U\rho\, U^\dagger$.

4.3 Reduced Density Matrix and Partial Trace

The density matrix formalism allows us to reason about physically separated qubits and their subspaces. For this, we will use what is called a *reduced density operator*, which we can derive with the help of a procedure called a *partial trace*. Recall how we defined the trace of a matrix in Section 1.10 as the sum of the diagonal elements as

$$\mathrm{tr}(A) = \sum_{i=0}^{n-1} a_{ii} = a_{00} + a_{11} + \cdots + a_{n-1n-1}. \tag{4.2}$$

The trace of a matrix is independent of the basis used to represent the matrix, as we hinted in Section 4.1 on the spectral decomposition. We also know from the above that the trace of a density matrix is 1. This means that we can write the trace of an operator as

$$\mathrm{tr}(A) = \sum_{i=0}^{n-1} \langle i|A|i\rangle,$$

where $|i\rangle$ are the basis vectors of an orthonormal basis. For a product state of two qubits expressed as a density matrix, we take the trace over the subsystem B as follows. The states $|i\rangle$ are the basis states of an orthonormal basis for ρ_B. From Equation (4.2), we know that the sum over the terms $\langle i|\rho_B|i\rangle$ is 1. Hence:

$$\mathrm{tr}_B(\rho_{AB}) = \sum_i \rho_A \otimes \mathrm{tr}(\rho_B) = \sum_i \rho_A \otimes \underbrace{\langle i|\rho_B|i\rangle}_{\sum=1} = \rho_A. \tag{4.3}$$

We say that subsystem B is being "traced out". In general, the state does not need to be a product state for this procedure to work. We can also generalize it and use identity matrices to leave subspaces untouched. This is quite similar to what we do during general operator applications.

To see how this works, let us first consider a two-qubit state and trace out qubit 0 as subsystem A. We construct the special operator matrices $|0_A\rangle$ and $|1_A\rangle$ below by tensoring the basis states $|0\rangle$ and $|1\rangle$ with an identity matrix I, resulting in matrices of size 4×2. The order of basis state and identity matrix depends on which specific qubit we intend to trace out. Here we want to trace out qubit 0, so the $|0\rangle$ and $|1\rangle$ states come first.

$$|0_A\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} = \begin{pmatrix}1 & 0\\0 & 1\\0 & 0\\0 & 0\end{pmatrix}, \quad |1_A\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} = \begin{pmatrix}0 & 0\\0 & 0\\1 & 0\\0 & 1\end{pmatrix}.$$

Then we use $|0_A\rangle$ and $|1_A\rangle$ and compute the partial traces as shown in Equation (4.3) with

$$\rho_A = \mathrm{tr}_B(\rho_{AB}) = \langle 0_B|\rho_{AB}|0_B\rangle + \langle 1_B|\rho_{AB}|1_B\rangle,$$
$$\rho_B = \mathrm{tr}_A(\rho_{AB}) = \langle 0_A|\rho_{AB}|0_A\rangle + \langle 1_A|\rho_{AB}|1_A\rangle.$$

The partial trace is a *dimension-reducing* operation. For example, let us assume that we have a system of two qubits A and B (at indices 0 and 1) with a 4×4 density matrix, and we want to trace out qubit 0. Multiplying the 4×4 density matrix from the right with a 4×2 matrix results in a 4×2 matrix. Multiplying this matrix from the left with a (now transposed) 2×4 matrix results in a 2×2 matrix.

Find the code
In file src/lib/ops.py

```
def TraceOutSingle(rho: Operator, index: int) -> Operator:
  nbits = int(math.log2(rho.shape[0]))
  assert index <= 0 < nbits, 'TraceOutSingle: Invalid index.'

  eye = Identity()
  zero = Operator([1.0, 0.0])
  one = Operator([0.0, 1.0])
  p0 = p1 = tensor.Tensor(1.0)
  for idx in range(nbits):
    if idx == index:
      p0 = p0 * zero
      p1 = p1 * one
    else:
      p0 = p0 * eye
      p1 = p1 * eye

  rho0 = p0 @ rho @ p0.transpose()
  rho1 = p1 @ rho @ p1.transpose()
  return rho0 + rho1
```

If we have a state of n qubits and are interested in the state of just one of the qubits, we must trace out *all other* qubits. For this, we add a convenience function to file src/lib/ops.py:

```
def TraceOut(rho: Operator, index_set: List[int]) -> Operator:
  for idx, val in enumerate(index_set):
    nbits = int(math.log2(rho.shape[0]))
    rho = TraceOutSingle(rho, val)
    for i in range(idx+1, len(index_set)):
      index_set[i] = index_set[i] - 1
  return rho
```

Experiments

Let us see this procedure in action. We start by producing a state from two well-defined qubits (assuming a local phase of 0, which means the factors are real), with

$$q_0 = \frac{1}{2}|0\rangle + \frac{\sqrt{3}}{2}|1\rangle \quad \text{and} \quad q_1 = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle .$$

```
    q0 = state.qubit(alpha=0.5)         # sqrt(0.25)
    q1 = state.qubit(alpha=0.8660254)   # sqrt(0.75)
    psi = q0 * q1
>>> psi
State([0.433+0.j, 0.25 +0.j, 0.75 +0.j, 0.433+0.j], dtype=complex64)
```

```
>>> psi.density()
Tensor([[0.188+0.j, 0.108+0.j, 0.325+0.j, 0.188+0.j],
        [0.108+0.j, 0.062+0.j, 0.188+0.j, 0.108+0.j],
        [0.325+0.j, 0.188+0.j, 0.562+0.j, 0.325+0.j],
        [0.188+0.j, 0.108+0.j, 0.325+0.j, 0.188+0.j]], dtype=complex64)
```

Tracing out one qubit should leave the other in the resulting density matrix, with the top left matrix element having the value $|\alpha|^2$ and the bottom right matrix element having the value $|\beta|^2$ for the remaining qubit. For the example, tracing out qubit q_1 should result in a value of $0.5^2 = 0.25$ in the top left matrix element, which is the norm squared of $\alpha = 0.5$ for qubit q_0.

```
    reduced = ops.TraceOut(psi.density(), [1])
    self.assertTrue(math.isclose(np.real(np.trace(reduced)), 1.0))
>>> reduced
Tensor([[0.25 +0.j, 0.433+0.j],
        [0.433+0.j, 0.75 +0.j]], dtype=complex64)
```

Tracing out qubit q_0 should leave $0.8660254^2 = 0.75$ at the top left:

```
    reduced = ops.TraceOut(psi.density(), [0])
    self.assertTrue(math.isclose(np.real(np.trace(reduced)), 1.0))
>>> reduced
Tensor([[0.75 +0.j, 0.433+0.j],
        [0.433+0.j, 0.25 +0.j]], dtype=complex64)
```

As an example of an entangled state, let us take the first Bell state β_{00}. For this state, the square of the trace of the density matrix is 1. After tracing out qubit 0, the square of the trace is just $0.5^2 + 0.5^2 = 0.5$.

```
    psi = bell.bell_state(0, 0)
    reduced = ops.TraceOut(psi.density(), [0])
    self.assertTrue(math.isclose(np.real(np.trace(reduced)),
                                 1.0, abs_tol=1e-6))
    self.assertTrue(math.isclose(np.real(reduced[0, 0]),
                                 0.5, abs_tol=1e-6))
    self.assertTrue(math.isclose(np.real(reduced[1, 1]),
                                 0.5, abs_tol=1e-6))
```

This already hints at the methodology to distinguish pure and mixed states, which we elaborate upon further in Section 4.4.

4.4 Maximal Entanglement

A *maximally mixed* state is a state that exhibits maximum uncertainty, or randomness, in its outcome when measured. When represented by a density matrix, a maximally mixed state is proportional to the identity matrix.

For two-qubit states, we define *maximally entangled* as follows. The partial trace allows us to reason about a subspace of a state. Tracing out a subspace leaves a reduced density matrix. We call a two-qubit state *maximally entangled* if the remaining reduced density matrices are maximally mixed after tracing out individual qubits.

For example, in Section 4.3, we saw that the density matrix of a Bell state was $I/2$ after tracing out a single qubit. The diagonal elements are all identical, and the off-diagonal elements are 0, meaning it is a maximally mixed state. This also means that Bell states[4] are maximally entangled states. The trace of the reduced density matrix is 1, as required for a density matrix. However, the trace squared of the reduced and maximally mixed state is 0.5:

$$\begin{aligned} \mathrm{tr}(I/2) &= 1, \\ \mathrm{tr}\ (I/2)^2 &= 0.5 < 1. \end{aligned}$$

This result is as expected for an entangled state. The joint state of the two qubits is a pure state, which means that we know everything there is to know about the state. However, looking at the individual qubits of the entangled Bell state with the help of density matrices, we find that those are in a mixed state.

In general, maximal entanglement is defined as maximizing a specific entanglement measure. Our bipartite two-qubit case above was easy to reason about. For multipartite states, things become considerably more complicated and are beyond the scope of this book. A good discussion can be found in Plenio and Virmany (2006).

4.5 Schmidt Decomposition

For a given state, determining whether the state is separable or entangled can be of great interest. For a two-qubit state, we derived a simple entanglement test in Section 2.11.2, but this test was not general. In this section, we introduce the *Schmidt decomposition*, a well-known linear algebra technique that is useful in quantum computing, as it provides a general test for entanglement. Furthermore, it even points to a measure of the entanglement strength.

Assume we have a pure state $|\psi\rangle$ in the composite bipartite quantum system AB. For simplicity, assume that A and B have the same dimensionality n. The *Schmidt decomposition* states that there exists an orthonormal basis $\{u_0, u_1, \ldots, u_{n-1}\}$ for system A and a basis $\{v_0, v_1, \ldots, v_{n-1}\}$ for system B, such that

$$|\psi\rangle = \sum_{i=0}^{n-1} \lambda_i \, |u_i\rangle_A \otimes |v_i\rangle_B \,. \tag{4.4}$$

[4] As well as GHZ states.

The λ_i are called the *Schmidt coefficients*. They are real positive numbers with $\sum_i \lambda_i^2 = 1$. We can use these coefficients to test for separability: A state $|\psi\rangle$ is separable if and only if the number of distinct coefficients is 1 exactly.

This description may seem somewhat abstract; it will be helpful to look at two examples. First, consider a separable state of two qubits in equal superposition, where the first qubit belongs to system A and the second qubit to system B:

$$|\psi\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle .$$

We know from Section 2.4.1 that we can factor this state into

$$\begin{aligned} |\psi\rangle &= \frac{1}{\sqrt{2}}\big(|0\rangle_A + |1\rangle_A\big) \otimes \frac{1}{\sqrt{2}}\left(|0\rangle_B + |1\rangle_B\right) \\ &= 1 \cdot \big(|+\rangle_A \otimes |+\rangle_B\big) + 0 \cdot \big(|-\rangle_A \otimes |-\rangle_B\big). \end{aligned}$$

Since there is only a single nonzero Schmidt coefficient, the state $|\psi\rangle$ is separable. For an example of an entangled state, let us look at the $|W\rangle$ state from Section 2.11.5 and separate it into two systems, one containing the first two qubits and the other having the third qubit:

$$\begin{aligned} |W\rangle &= \frac{1}{\sqrt{3}}\big(|001\rangle + |010\rangle + |100\rangle\big). \\ |W_{AB}\rangle &= \frac{1}{\sqrt{3}}\big(|00\rangle_A |1\rangle_B + |01\rangle_A |0\rangle_B + |10\rangle_A |0\rangle_B\big). \end{aligned}$$

We set the basis for system A as $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ and the basis for system B as $\{|0\rangle, |1\rangle\}$. With this and a little algebra to separate A and B, we manually decompose the state into

$$|W_{AB}\rangle = \underbrace{\frac{1}{\sqrt{3}}}_{\lambda_0}\big(|00\rangle_A \otimes |1\rangle_B\big) + \underbrace{\sqrt{\frac{2}{3}}}_{\lambda_1}\left(\frac{1}{\sqrt{2}}|01\rangle_A + \frac{1}{\sqrt{2}}|10\rangle_A\right) \otimes |0\rangle_B$$

to find the two Schmidt coefficients $\lambda_0 = \sqrt{1/3}$ and $\lambda_1 = \sqrt{2/3}$. The basis vectors for A are $|u_0\rangle = |00\rangle$ and the more complex $|u_1\rangle = \frac{1}{\sqrt{2}}\big(|01\rangle_A + |10\rangle_A\big)$. In general, any bipartite state can be written as

$$|\psi_{AB}\rangle = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} c_{ij}\ |\chi_i\rangle_A \otimes |\phi_j\rangle_B .$$

The Schmidt decomposition reduces this by introducing new bases such that

$$|\psi_{AB}\rangle = \sum_{i=0}^{n-1} \lambda_i\ |u_i\rangle_A \otimes |v_i\rangle_B .$$

The question is how to derive the λ_i from the c_{ij}. There are two ways to approach this. The first method is based on the *singular value decomposition (SVD)* (Strang (2016), page 364). Let us again use the example of the $|W\rangle$ state from above, with the

basis states for system A of $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ and $\{|0\rangle, |1\rangle\}$ for B. We can write any state in the combined basis as

$$\begin{aligned}|\psi_{AB}\rangle = c_{00}\,|00\rangle\,|0\rangle + c_{01}\,|01\rangle\,|0\rangle + c_{02}\,|10\rangle\,|0\rangle + c_{03}\,|11\rangle\,|0\rangle \\ + c_{10}\,|00\rangle\,|1\rangle + c_{11}\,|01\rangle\,|1\rangle + c_{12}\,|10\rangle\,|1\rangle + c_{13}\,|11\rangle\,|1\rangle\,.\end{aligned}$$

In general, we can arrange the coefficients as a matrix C_{AB}, where we index the rows with the basis states $|\psi\rangle$ from system A and the columns with the basis states $|\phi\rangle$ from system B:

$$C_{AB} = \begin{array}{c} \\ |\psi_0\rangle \\ |\psi_1\rangle \\ \vdots \\ |\psi_{n-1}\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |\phi_0\rangle & |\phi_1\rangle & \dots & |\phi_{n-1}\rangle \end{array} \\ \begin{pmatrix} c_{00} & c_{01} & \dots & c_{0,n-1} \\ c_{10} & c_{11} & \dots & c_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n-1,0} & c_{n-1,1} & \dots & c_{n-1,n-1} \end{pmatrix} \end{array}.$$

Now we make use of the SVD, which is a standard technique in linear algebra with many great resources discussing it. We will not discuss it any further here. What the SVD does is decompose a square matrix C_{AB} into

$$C_{AB} = U\Sigma V^\dagger,$$

where both U and $V^\dagger$ are $n \times n$ unitary matrices.[5] Recall that a unitary matrix is a matrix with orthonormal columns. This is exactly what we need for the Schmidt decomposition. After the SVD, the columns of U will be the orthonormal basis u_i. The rows of $V^\dagger$ will have the adjoints of the orthonormal basis v_i. The diagonal matrix Σ will have the Schmidt coefficients λ_i on its diagonal.

We can also look at the Schmidt decomposition from a different angle using the partial trace. For a bipartite pure state $|\psi_{AB}\rangle$ in Schmidt form and its density matrix ρ_{AB}, we get the reduced density matrix ρ_A for subsystem A as

$$\begin{aligned}|\psi_{AB}\rangle &= \sum_{i=0}^{n-1} \lambda_i\ |u_i\rangle_A \otimes |v_i\rangle_B \quad \text{and} \quad \rho_{AB} = |\psi_{AB}\rangle\langle\psi_{AB}|. \\ \rho_A &= \mathrm{tr}_B(\rho_{AB}) \\ &= \mathrm{tr}_B\left(\sum_{i=0}^{n-1} \lambda_i\ |u_i\rangle_A \otimes |v_i\rangle_B \sum_{j=0}^{n-1} \lambda_j\ \langle u_j|_A \otimes \langle v_j|_B\right).\end{aligned}$$

We can reorder this and use the definition of the trace to pull it to the right. Also, recall from Equation (1.7) that the trace over an outer product is equal to its reverse inner product, which leads to

$$\rho_A = \mathrm{tr}_B\left(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} \lambda_i\lambda_j\ |u_i\rangle\langle u_j|_A \otimes |v_i\rangle\langle v_j|_B\right)$$

[5] In general, things get more complex if C_{AB} is an $n \times m$ rectangular matrix.

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \lambda_i \lambda_j \, |u_i\rangle\langle u_j|_A \otimes \underbrace{\mathrm{tr}\big(|v_i\rangle\langle v_j|_B\big)}_{=\langle v_j|v_i\rangle}.$$

The v_i and v_j are orthonormal basis states. Their inner product is 0, except for $i = j$, where it is 1. This allows us to write the final reduced density matrix as

$$\rho_A = \sum_{i=0}^{n-1} \lambda_i^2 \, |u_i\rangle\langle u_i|,$$

which is the spectral decomposition of ρ_A with eigenvectors $|u_i\rangle$ and eigenvalues λ_i^2. We can do the same thing for the partial density matrix ρ_B to arrive at

$$\rho_B = \sum_{i=0}^{n-1} \lambda_i^2 \, |v_i\rangle\langle v_i|.$$

Now we have the Schmidt coefficients, which are identical for both ρ_A and ρ_B, and the bases for both A and B, which is what we needed for the Schmidt decomposition in Equation (4.4). This is also what we implemented in the code below.

Find the code
In file src/schmidt_decomposition.py

```
def compute_eigvals(psi: state.State, expected: int, tolerance: float):
  rho = psi.density()

  rho0 = ops.TraceOut(rho, [1])
  eigvals0 = np.linalg.eigvalsh(rho0)
  rho1 = ops.TraceOut(rho, [0])
  eigvals1 = np.linalg.eigvalsh(rho1)

  assert np.allclose(eigvals0, eigvals1, atol=1e-6), 'Whaa'
  assert np.allclose(np.sum(eigvals0), 1.0), 'Whaa'

  # Count the number of nonzero eigenvalues and match against expected.
  nonzero = np.sum(eigvals0 > tolerance)
  if nonzero != expected:
    print(f'  Unstable math: {eigvals0[0]:.4f}, {eigvals0[1]:.4f}')

  # Construct the state from the eigenvalues and the new bases.
  a0, d0, _ = np.linalg.svd(rho0)
  a1, _, _ = np.linalg.svd(rho1)
  newpsi = (np.sqrt(d0[0]) * np.kron(a0[:, 0], a1[0, :]) +
            np.sqrt(d0[1]) * np.kron(a0[:, 1], a1[1, :]))
  assert np.allclose(psi, newpsi, atol=1e-3), 'Incorrect Schmidt basis'
  return eigvals0
```

Now we can run a few examples with entangled and non-entangled states to convince ourselves that things work as expected!

```
def main(argv):
  iterations = 1000
  print('\tSchmidt Decomposition for seperable states.')
  for _ in range(iterations):
    psi = state.qubit(random.random()) * state.qubit(random.random())
    compute_eigvals(psi, 1, 1e-3)

  print('\tSchmidt Decomposition for entangled states.')
  for _ in range(iterations):
    psi = state.bitstring(0, 0)
    psi = ops.Hadamard()(psi)
    angle = random.random() * np.pi
    psi = ops.ControlledU(0, 1, ops.RotationY(angle))(psi)
    compute_eigvals(psi, 2, 1e-9)

  print('\tSchmidt Decomposition for max-entangled state.')
  psi = state.bitstring(0, 0)
  psi = ops.Hadamard()(psi)
  psi = ops.Cnot()(psi)
  eigv = compute_eigvals(psi, 2, 1e-9)
  if abs(eigv[0] - eigv[1]) > 0.001:
    raise AssertionError('Wrong computation for max-entangled state.')
```

4.6 State Purification

When a pure state interacts with its environment, it can become a mixed state because of noise and decoherence. *State purification* attempts to create a pure state from a mixed state. We can think of it as a dual to the partial trace and ask the inverse question: Given a mixed state ρ_A for a quantum system A, is it possible to introduce another system B such that the state $|AB\rangle$ is a pure state with partial trace $\rho_A = \mathrm{tr}_B(|AB\rangle\langle AB|)$?

This is indeed possible with *state purification*, a mathematical procedure that is generally considered to have no real physical relevance.[6] Here is how it works. Let us assume that we have state A in its spectral decomposition form with the eigenvalues λ_i and basis states $|i_A\rangle$ as

$$\rho_A = \sum_i \lambda_i |i_A\rangle\langle i_A|.$$

We introduce another system B with the same state space as A and the corresponding orthonormal basis states $|i_B\rangle$. Then state

$$|AB\rangle = \sum_i \sqrt{\lambda_i}|i_A\rangle|i_B\rangle$$

is a pure state. We can check that taking the partial trace of system B will result in exactly ρ_A.

[6] Which may not be a correct statement (Kleinmann et al., 2006).

Find the code
In file src/purification.py

Let us quickly verify this in code. We use the same eigenvectors for the subsystems A and B:

```
def purify(rho: ops.Operator, nbits: int):
  rho_eig_val, rho_eig_vec = np.linalg.eig(rho)

  # Construct combined system, using same basis vectors.
  psi1 = np.zeros((2**(nbits * 2)), dtype=np.complex128)
  for i in range(len(rho_eig_val)):
    psi1 += (np.sqrt(rho_eig_val[i]) *
             np.kron(rho_eig_vec[:, i], rho_eig_vec[:, i]))

  # Make sure it is a pure state.
  mat = psi1.reshape((2**nbits, 2**nbits))
  assert np.allclose(np.trace(mat@mat), 1.0, atol = 1e-5)

  # Another way to compute the reduced density matrix:
  reduced = ops.TraceOut(state.State(psi1).density(),
                         [x for x in range(int(nbits),
                                           int(nbits*2))])
  assert np.allclose(rho, reduced), 'Wrong reduced density'
```

We test a variety of density matrices to ensure that this procedure works for entangled and unentangled states:

```
def main(argv):
  print('  Single qubit.')
  purify(ops.Operator([(0.22704306, 0.34178495),
                       (0.34178495, 0.77295694)]), 1)

  print('  Bell states.')
  purify(bell.bell_state(0, 0).density(), 2)
  purify(bell.bell_state(0, 1).density(), 2)

  print('  GHZ state.')
  purify(bell.ghz_state(4).density(), 4)

  print('  Random 2 qubit states.')
  for _ in range(1000):
    psi = state.State(np.random.rand(4)).normalize()
    purify(psi.density(), 2)
```

4.7 Pauli Representation of Operators

In Section 2.7.2 on the Pauli operators X, Y, Z, and the identity operator I, we stated that Pauli matrices form a basis for 2×2 matrices. This means that a density matrix can be written as follows, which is also called the *Pauli representation* of an operator:[7]

$$\rho = \frac{I + xX + yY + zZ}{2}. \tag{4.5}$$

Let us derive this result. First, note that since we claim that the Pauli matrices form an orthonormal basis for *any* 2×2 matrix, we should be able to write any such matrix as

$$A = cI + xX + yY + zZ. \tag{4.6}$$

If A is Hermitian, all four factors c,x,y,z will be real. By simply adding up the four matrix terms, we get

$$A = \begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix}.$$

Comparing Equation (4.5) and Equation (4.6) leads to three questions:

1. Why is there no factor c in front of I in Equation (4.5)?
2. Where does the factor $1/2$ come from?
3. Given a matrix, what are the factors x, y, and z, and maybe c?

To answer the second and third questions first, for a given state $|\psi\rangle$ and its density matrix $\rho = |\psi\rangle\langle\psi|$, we extract the individual factors by multiplying the density matrix with the corresponding Pauli matrix and taking the trace.[8] Let's see how this works. To extract the factor x, we compute

$$\begin{aligned} X\rho &= X \begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix} \\ &= \begin{pmatrix} x+iy & c-z \\ c+z & x-iy \end{pmatrix}. \end{aligned}$$

Taking the trace of this matrix as

$$\mathrm{tr}(X\rho) = x + iy + x - iy = 2x,$$

we are able to extract the factor x, but with a factor of 2. This is the reason why, in Equation (4.5), we compensate with a factor of $1/2$. Let's derive this for the other factors, starting with Y:

[7] Here we implicitly assume that the trace of the operator is 1.

[8] We are using the density matrix of a pure state here, but the same mechanism will work for *any* 2×2 matrix.

$$
\begin{aligned}
Y\rho &= Y\begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix} \\
&= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}\begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix} \\
&= \begin{pmatrix} -ix+y & -i(c-z) \\ i(c+z) & ix+y \end{pmatrix}, \\
\Rightarrow \operatorname{tr}(Y\rho) &= -ix+y+ix+y = 2y.
\end{aligned}
$$

And similarly, for Z:

$$
\begin{aligned}
Z\rho &= Z\begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix} \\
&= \begin{pmatrix} c+z & x-iy \\ -x-iy & z-c \end{pmatrix}, \\
\Rightarrow \operatorname{tr}(Z\rho) &= c+z+z-c = 2z.
\end{aligned}
$$

Finally, for the identity I, the right side remains unchanged:

$$
\begin{aligned}
I\rho &= I\begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix} \\
&= \begin{pmatrix} c+z & x-iy \\ x+iy & c-z \end{pmatrix}, \\
\Rightarrow \operatorname{tr}(I\rho) &= c+z+c-z = 2c.
\end{aligned}
$$

Now we use the fact that the trace of a density matrix must be 1. Since we already applied a factor $1/2$ in Equation (4.5), the factor c in Equation (4.6) must be 1. This is why we were able to omit a scalar factor to I in Equation (4.5).

PY

Find the code
In file `src/pauli_rep.py`

This is easy to verify in code. We construct a random qubit and extract the factors as described above. With these factors, we can verify that we calculated the correct results simply by inserting them into Equation (4.5). We compute the factors c,x,y,z in Equation (4.6) as the Python variables `c`, `x`, `y`, and `z`.

```
qc = circuit.qc('random qubit')
qc.random()
rho = qc.psi.density()

c = np.trace(ops.Identity() @ rho)  # not strictly needed.
x = np.trace(ops.PauliX() @ rho)
y = np.trace(ops.PauliY() @ rho)
z = np.trace(ops.PauliZ() @ rho)
```

```
new_rho = 0.5 * (c * ops.Identity() + x * ops.PauliX() +
                 y * ops.PauliY() + z * ops.PauliZ())
assert np.allclose(rho, new_rho), 'Invalid Pauli Representation'
```

Decomposition with Projectors

With the factors x, y, and z from Equation (4.6), there is an interesting alternative representation with application in the circuit-cutting technique (Tang et al., 2021). We compute the projectors as usual:

$$P_{|0\rangle} = |0\rangle\langle 0|,\ P_{|1\rangle} = |1\rangle\langle 1|,\ P_{|+\rangle} = |+\rangle\langle +|,\ P_{|+y\rangle} = |+y\rangle\langle +y|.$$

With the following four matrices A_i and the factors c,x,y,z calculated above, we can decompose a density matrix ρ as[9]

$$\begin{aligned} A_1 &= (c+z)P_{|0\rangle}, & A_2 &= (c-z)P_{|1\rangle}, \\ A_3 &= x(2P_{|+\rangle} - P_{|0\rangle} - P_{|1\rangle}), & A_4 &= y(2P_{|+y\rangle} - P_{|0\rangle} - P_{|1\rangle}), \\ \Rightarrow \rho &= \frac{A_1 + A_2 + A_3 + A_4}{2}. \end{aligned}$$

Two Qubits

So far, we have computed the density matrix of a single qubit with the Pauli matrices σ_i as

$$\rho = \frac{1}{2}\sum_{i=0}^{3} c_i\sigma_i.$$

This technique can be extended to two qubits by applying the same principles and multiplying the density matrix by *all* tensor products of two Pauli matrices. Similar to Equation (4.5), the density matrix can be constructed from the two-qubit bases in the following way (note that the factor is now $1/4$, or $1/2^n$ in the general case for n qubits):

$$\rho = \frac{1}{4}\sum_{i,j=0}^{3} c_{i,j}\,(\sigma_i \otimes \sigma_j).$$

To generalize to any number of qubits: Generate n-dimensional tensors holding the factors $(c_{i_1,\ldots,i_n})$, tensor together n Pauli matrices of each kind, sum up all the terms, and normalize:

$$\rho = \frac{1}{2^n}\sum_{i_1,\ldots,i_n=0}^{3} c_{i_1,\ldots,i_n}\,(\sigma_{i_1} \otimes \cdots \otimes \sigma_{i_n}).$$

[9] The corresponding code and test are in the open-source repository.

But back to the two-qubit case. We compute the factors (c_{ij}) in the same way as in the single-qubit case. In the following code, we first create a state and corresponding density matrix of two potentially entangled qubits:

```
qc = circuit.qc('random qubit')
qc.random(2)
qc.h(0)
qc.cx(0, 1)
rho = qc.psi.density()
```

Now we multiply by all the Pauli matrix tensor products and compute the factors from the trace. Since two qubits are involved, instead of a vector of factors (c_i), we will now get a matrix of factors (c_{ij}):

```
paulis = [ops.Identity(), ops.PauliX(), ops.PauliY(), ops.PauliZ()]
c = np.zeros((4, 4), dtype=np.complex64)
for i in range(4):
  for j in range(4):
    tprod = paulis[i] * paulis[j]
    c[i][j] = np.trace(rho @ tprod)
```

Note that in the computation of the trace above, we switched the order of `rho` and `tprod` compared to the single-qubit case. We can do this because for two matrices A and B, $\mathrm{tr}(AB) = \mathrm{tr}(BA)$. Similarly to the above, we can now construct a new state and verify that the computed factors are correct:

```
new_rho = np.zeros((4, 4), dtype=np.complex64)
for i in range(4):
  for j in range(4):
    tprod = paulis[i] * paulis[j]
    new_rho = new_rho + c[i][j] * tprod
assert np.allclose(rho, new_rho / 4, atol=1e-5),  'Invalid result'
```

4.8 *ZYZ* Decomposition

In this section, we will show and derive that any single-qubit unitary gate can be decomposed (Nielsen and Chuang (2011), Theorem 4.1) into the form

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta), \tag{4.7}$$

where R_y and R_z are the rotations about the y-axis and z-axis as described in Section 2.7.4:

$$\begin{aligned} R_y(\theta) = e^{-i\frac{\theta}{2}Y} &= \cos\left(\frac{\theta}{2}\right) I - i \sin\left(\frac{\theta}{2}\right) Y \\ &= \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}, \end{aligned}$$

$$R_z(\theta) = e^{-i\frac{\theta}{2}Z} = \cos\left(\frac{\theta}{2}\right) I - i \sin\left(\frac{\theta}{2}\right) Z$$
$$= \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}.$$

Let's find the values for these parameters and verify in code that we have calculated the right results. First, we should think of a general single-qubit gate U as having the form

$$U = e^{i\alpha} \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

To compute α, we consider the new unitary $V = e^{-i\alpha} U$ with the special property $\det V = 1$. The determinant of a unitary[10] is the product of the eigenvalues. For n-qubit operators, we have $k = 2^n$ eigenvalues. If we multiply U by a constant c, then each eigenvalue is multiplied by this constant, and $\det(cU) = c^k \det(U)$. Hence, for this 2×2 matrix U, the determinant is $\det U = a^{i2\alpha}$. Similarly to how we find the angle of a phase of a complex number, we can find the angle α as the arctangent between the imaginary part of the determinant and its real part:

$$\alpha = \frac{1}{2} \arctan\left(\frac{\mathrm{Im}\big(\det(U)\big)}{\mathrm{Re}\big(\det(U)\big)}\right).$$

To compute γ, we multiply out Equation (4.7) and get

$$U = e^{i\alpha} \begin{pmatrix} e^{-i\frac{1}{2}\beta - i\frac{1}{2}\delta} \cos(\frac{\gamma}{2}) & -e^{-i\frac{1}{2}\beta + i\frac{1}{2}\delta} \sin(\frac{\gamma}{2}) \\ e^{+i\frac{1}{2}\beta - i\frac{1}{2}\delta} \sin(\frac{\gamma}{2}) & e^{+i\frac{1}{2}\beta + i\frac{1}{2}\delta} \cos(\frac{\gamma}{2}) \end{pmatrix}. \tag{4.8}$$

From this we can compute γ. We compute it from the upper left with $2 \arccos |a|$ or from the upper right with $2 \arcsin |b|$. For numerical stability, we choose the largest of the two matrix elements a and b. For α, we use Python's `arctan2` to account for all quadrants.

Find the code
In file `src/zy_decomp.py`

```
def zy_decompose(umat):
  a = umat[0][0]
  b = umat[0][1]
  c = umat[1][0]

  det = np.linalg.det(umat)
  alpha = 0.5 * np.arctan2(det.imag, det.real)
  if a >= b:
    gamma = 2 * np.arccos(abs(a))
  else:
    gamma = 2 * np.arcsin(abs(b))
```

[10] To be precise, the determinant of *any* diagonalizable matrix.

When we look at the matrix elements a and c in Equation (4.8) (top left and bottom left elements), we can see that the sign differs for the $\beta/2$ terms. Taking the whole exponentiated term (before cos) as a rotation means that the whole β is the phase difference between these matrix elements. Similarly, the angle δ is the phase difference between $-b$ and c (though there are special cases for $\gamma = 0$ or $\gamma = \pi$, which we omit here):[11]

```
beta = cmath.phase(c) - cmath.phase(a)
delta = cmath.phase(-b) - cmath.phase(a)
return alpha, beta, gamma, delta
```

The code in file `zy_decomp.py` contains routines to construct the matrices from these angles and to ensure correct calculations.

4.9 *XYX* Decomposition

Given the insights from Section 4.8, we can quickly derive an *XYX* decomposition as well. Going from one coordinate system to another can be done with a unitary transformation. In this case, the Hadamard gate allows us to change our frame of reference in the following way:

$$\begin{aligned} HZH &= X \quad \text{and} \quad HYH = -Y, \\ HR_z(\theta)H &= R_x(\theta) \quad \text{and} \quad HR_y(\theta)H = R_y(-\theta). \end{aligned}$$

If we assign $U' = HUH^{-1} = HUH$ (the Hadamard gate is its own inverse), we can apply the *ZYZ* decomposition as above on $U' = e^{i\alpha}R_z(\beta)R_y(\gamma)R_z(\delta)$. The computed angles will be the *XYX* decomposition we were looking for.

$$\begin{aligned} U &= HU'H \\ &= e^{i\alpha}\left(HR_z(\beta)H\right)\left(HR_y(\gamma)H\right)\left(HR_z(\delta)H\right) \\ &= e^{i\alpha}R_x(\beta)R_y(-\gamma)R_x(\delta). \end{aligned}$$

PY

Find the code
In file `src/zy_decomp.py`

```
def make_u_xy(alpha, beta, gamma, delta):
  return (
      ops.RotationX(beta) @ ops.RotationY(gamma) @ ops.RotationX(delta)
  ) * cmath.exp(1.0j * alpha)

[...]
udash = ops.Hadamard() @ umat @ ops.Hadamard()
alpha, beta, gamma, delta = zy_decompose(udash)
unew = make_u_xy(alpha, beta, -gamma, delta)
if not np.allclose(umat, unew, atol=1e-4):
  raise AssertionError('X-Y decomposition failed')
```

[11] Found in http://quantumcomputing.stackexchange.com/a/16263.

5 Beyond Classical

Quantum computing is of great interest because of its promise of being able to execute certain tasks much faster than is possible with classical computers. We will soon learn about the quadratic speed-up of search with Grover's algorithm and even exponential speed-up for integer factorization with Shor's algorithm. One natural question to ask is whether quantum computers are limited to a small set of specific tasks only.

In this chapter, we demonstrate how any classical digital circuit can be implemented with a quantum circuit. This proves that quantum computers are at least as *capable* as classical computers. Then we detail and discuss the seminal quantum supremacy experiment, which, for the first time, seemed to demonstrate a true quantum advantage for a specific type of algorithm. This result, however, did not come without controversy, so buckle up, this will be interesting.

5.1 Classical Arithmetic

Let's begin by implementing a standard classical logic circuit, the full adder, with quantum gates instead of classical gates. The quantum circuit is very basic in that it does not utilize any specific features of quantum computing (we detail arithmetic in the quantum Fourier domain in Section 11.4).

A 1-bit full adder block is usually drawn as shown in Figure 5.1. The input bits are A and B; their sum comes out as bit *Sum*. We only have 1 bit to represent the result, so if both A and B are 1, their binary sum overflows back to 0, and we set a carry-out bit C_{out}. You can see all bit combinations in Table 5.1.

Multiple instances of the full adder can be chained together in sequence to facilitate the addition of multibit binary numbers. In this scenario, the potential carry-out bit of a one full adder is chained to the next full adder as a carry-in bit C_{in}.

Figure 5.1 The 1-bit full adder block diagram.

Table 5.1. Truth table for the full adder logic circuit.

A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

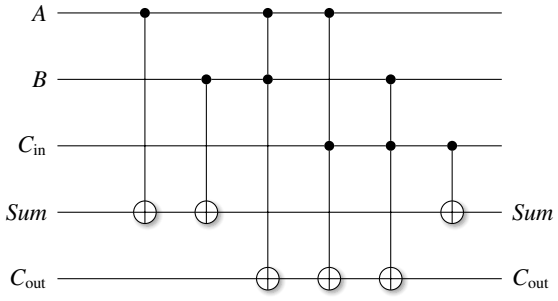


Figure 5.2 Classical full adder, implemented with quantum gates. All input qubits are in one of the basis states $|0\rangle$ or $|1\rangle$.

Classical circuits use, unsurprisingly, classical gates such as AND, OR, NAND, and others. The task at hand is to construct a quantum circuit that produces the same truth table as the classical circuit but only uses quantum gates. Classical 0s and 1s are represented by the basis states $|0\rangle$ and $|1\rangle$. With some thought (and experimentation), we arrive at the circuit in Figure 5.2. Let's walk through the circuit to convince ourselves that it is working properly:

- If A is $|1\rangle$, *Sum* will toggle to $|1\rangle$ (with the controlled Not from A to *Sum*).
- If B is $|1\rangle$, *Sum* will flip to $|1\rangle$ or back to $|0\rangle$ if it was already set to $|1\rangle$.
- If C_{in} is $|1\rangle$, *Sum* will flip again with the controlled Not on the right.
- C_{out} will toggle if both A and B are set, or both A and C_{in} are set, or both B and C_{in} are set.
- What happens if all A, B, and C_{in} are set? *Sum* will start as $|0\rangle$ and go through these states: $|0\rangle, |1\rangle, |0\rangle, |1\rangle$. C_{out} will also start as $|1\rangle$ and go through these states: $|0\rangle, |1\rangle, |0\rangle, |1\rangle$. The final results are $|1\rangle$ and $|1\rangle$ for both *Sum* and C_{out}, as expected.

The implementation of the circuit is straightforward using controlled and double-controlled Not gates. Measurements are probabilistic, but in this case, the probability of the correct result is 100% in the computational basis. There will only be a single

resulting state with nonzero probability. We apply each gate to the state in the order shown in Figure 5.2.

Find the code
In file src/arith_classic.py

```
def fulladder_matrix(psi: state.State):
  psi = ops.Cnot(0, 3)(psi, 0)
  psi = ops.Cnot(1, 3)(psi, 1)
  psi = ops.ControlledU(0, 1, ops.Cnot(1, 4))(psi, 0)
  psi = ops.ControlledU(0, 2, ops.Cnot(2, 4))(psi, 0)
  psi = ops.ControlledU(1, 2, ops.Cnot(2, 4))(psi, 1)
  psi = ops.Cnot(2, 3)(psi, 2)
  return psi
```

Next, we conduct experiments as follows. First, we construct the state from the inputs (A, B, C_{in}) and augment it with two $|0\rangle$ states for the expected outputs `sum` and `cout`. Then, we apply the circuit we just constructed. We measure the probabilities of the outputs being 1, which means we will get a probability of 0 if the state was $|0\rangle$ and a probability of 1 if the state was $|1\rangle$:

```
def experiment_matrix(a: int, b: int, cin: int,
                      expected_sum: int, expected_cout: int):
  psi = state.bitstring(a, b, cin, 0, 0)
  psi = fulladder_matrix(psi)

  bsum, _ = ops.Measure(psi, 3, tostate=1, collapse=False)
  bout, _ = ops.Measure(psi, 4, tostate=1, collapse=False)
  print(f'a: {a} b: {b} cin: {cin} sum: {bsum} cout: {bout}')
  if bsum != expected_sum or bout != expected_cout:
    raise AssertionError('invalid results')
```

Lastly, we check the circuit for all inputs and expected results:

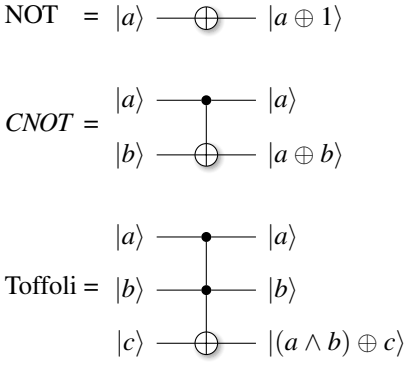
```
def add_classic():
  for exp_function in [experiment_matrix]:
    exp_function(0, 0, 0, 0, 0)
    exp_function(0, 1, 0, 1, 0)
    exp_function(1, 0, 0, 1, 0)
    exp_function(1, 1, 0, 0, 1)
    [...]

def main(argv):
  add_classic()
>>
a: 0 b: 0 cin: 0 sum: 0.0 cout: 0.0
a: 0 b: 1 cin: 0 sum: 1.0 cout: 0.0
a: 1 b: 0 cin: 0 sum: 1.0 cout: 0.0
a: 1 b: 1 cin: 0 sum: 0.0 cout: 1.0
[...]
```

Other classical circuits can be implemented and combined to build more powerful circuits. We show a general construction below, but it is important to note that all these circuits point to a general statement about quantum computers: Since classical *universal* logic gates can be implemented on quantum computers, a quantum computer is at least as capable as a classical computer.

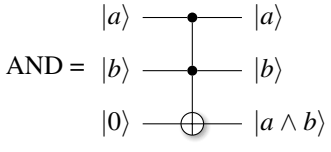
However, this does not mean that a quantum computer performs *better* in the general case. The circuit presented in Figure 5.2 may just be a very inefficient way to implement a simple 1-bit adder. However, we will soon learn about algorithms that perform significantly better on quantum computers than on classical computers by some measure of complexity.

5.2 General Construction of Logic Circuits

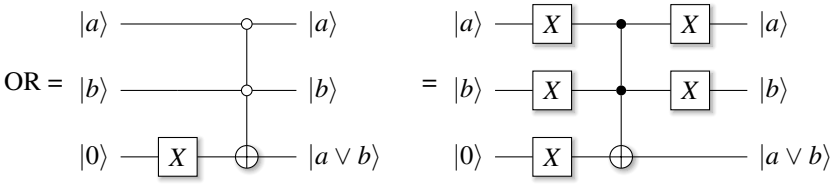
This section briefly discussed how to construct general classical logic circuits with quantum gates (Williams, 2011). This method uses only three quantum gates:



These gates are sufficient to construct quantum analogs to the classical gates AND ($\wedge$), OR ($\vee$), and, of course, the NOT gate. The AND gate is a Toffoli gate with a $|0\rangle$ as its third input:



The OR gate is slightly more involved but still based on a Toffoli gate:



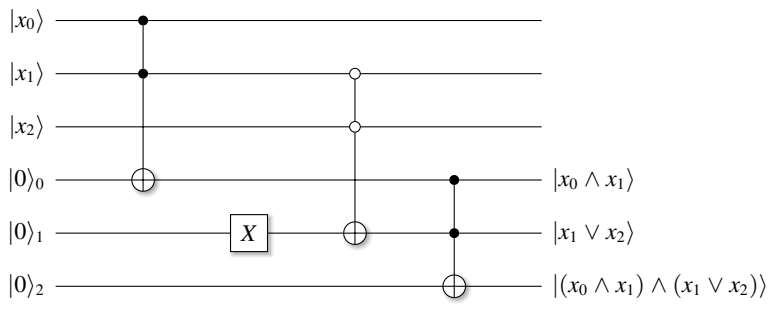
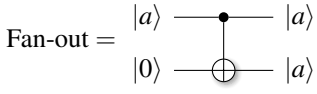


Figure 5.3 A Boolean formula, expressed with quantum gates.

We know that with NOT and AND, we can build the classically universal NAND gate, which means we can construct the quantum analog of any classical logic circuit with quantum gates.

We might need fan-out to connect single wires to multiple gates for complex logic circuits. Therefore, we need a *fan-out* circuit. Is this possible, or does fan-out violate the no-cloning theorem? The answer is no, it does not because, in this scenario, logical 0 and 1 are strictly represented by qubits in the basis states $|0\rangle$ or $|1\rangle$. For these states, cloning and fan-out are possible, as shown in Section 2.12.



With these elements, and knowing that any Boolean formula can be expressed as a product of sums[1] we can build any logic circuit with quantum gates. Of course, making this construction efficient would require additional techniques, such as ancilla management, uncomputation, logic optimizations, and general minimization of gates.

An example of a quantum circuit for the Boolean formula $(x_0 \wedge x_1) \wedge (x_1 \vee x_2)$ is shown in Figure 5.3. The top three qubits are the inputs $|x_i\rangle$, and the bottom three qubits are ancilla qubits initialized to state $|0\rangle$. In the circuit diagram, we do not show the uncomputation following the final gate that would be required to disentangle the ancillae from the state. The ability to uncompute ancillae in a large chain of logic expressions can reduce the number of required ancillae. In this example, we could uncompute $|0\rangle_0$ and $|0\rangle_1$ and make them available again for future temporary results.

5.3 The Quantum Supremacy Experiment

The term *Beyond Classical* is now the preferred term over *Quantum Advantage*, which in turn was the preferred term over the unfortunate term *Quantum Supremacy*. Prof. John Preskill originally coined that term to describe a computation that can be run

[1] http://en.wikipedia.org/wiki/Canonical_normal_form

efficiently on a quantum computer but would be intractable to run on a classical computer (Preskill, 2012; Harrow and Montanaro, 2017).

Computational complexity theory is a pillar of computer science. A good introduction and extensive literature references can be found in Dean (2016). A large set of complexity classes exists. The best-known big categories may be the following:

Class P, the class of decision problems (with a yes or no answer) with problem size n that run in polynomial time $\mathcal{O}(n^x)$, with x being of complexity $\mathcal{O}(1)$.

Class NP, which contains decision problems of size n that run in exponential time $\mathcal{O}(x^n)$ and can be *verified* in polynomial time.

Class NP-complete, which is a somewhat technical construction. It is a class of NP problems that other NP-complete problems can be mapped to in polynomial time. Finding a single example of this class that falls into P would mean that all members of this class are also in P.

Class NP-hard, the class of problems that are at least as hard as the hardest problems in NP. To simplify, this is the class of NP problems that may not be a decision problem, such as integer factorization, or for which there is no known polynomial-time algorithm for verification, such as the traveling salesperson problem (Applegate et al., 2006).

There are dozens of complexity classes with various properties and interrelationships. The famous question of whether P = NP remains one of the great challenges in computer science today.[2]

Interest in quantum computing arises from the belief that quantum algorithms fall into the BQP class, the class of algorithms that can be solved by a quantum Turing machine in polynomial time with an error probability of less than $1/3$ (which is a somewhat arbitrary bound). This group is believed to be more powerful than the BPP class, the class of algorithms that can be solved in polynomial time by a probabilistic Turing machine with a similar error rate. Put simply, there may be a class of algorithms that can run exponentially faster on quantum machines than on classical machines.

From a complexity-theoretical point of view, since BQP *contains* BPP, this would mean that quantum computers can efficiently simulate classical computers. However, would we run a word processor or a video game on a quantum computer? Theoretically, we could, but today it appears that classical and quantum computing complement each other. The term *beyond* seems to have been well chosen to indicate that there is a complexity class for algorithms that run tractably only on quantum computers.

To establish the quantum advantage, we will not take a complexity-theoretic approach in this book. Instead, we will try to estimate and validate the results of the quantum supremacy paper by Arute et al. (2019) to convince ourselves that quantum computers do indeed reach capabilities beyond those of classical machines.

[2] It can be answered jokingly with *yes*, if $N = 1$ or $P = 0$.

5.3.1 10,000 Years, 2 Days, or 200 Seconds

In 2019, Google published a seminal paper claiming to finally have reached a quantum advantage on their 53-qubit Sycamore chip (Arute et al., 2019). In their work, the researchers used a quantum *random algorithm*. This type of algorithm assembles a small set of gates randomly into a circuit, following only a small set of rules. The resulting circuit is a valid circuit and computes *something*. However, the result itself has no specific meaning. The researchers then *sampled* the result by performing a large number of measurements. Since a random circuit introduces random superpositions, it will produce probabilistic results. However, if the circuit is measured often enough, the results will be correlated and not purely random, proving that an actual computational process has occurred. In their paper, the researchers computed and sampled a random circuit 1,000,000 times in just 200 seconds, producing a result that would take the world's fastest supercomputer 10,000 years to produce and which can only be obtained by classically simulating the random circuit.

Shortly after that, IBM, a competitor in the field of quantum computing, followed up with the estimate that a similar result could be achieved in just a few days, with higher accuracy, on a classical supercomputer (Pednault et al., 2019). A few days versus 200 seconds is a factor of about 1,000. A few days versus 10,000 years is another factor of 1,000. Disagreements of this magnitude are exciting. How is it possible that these two great companies disagree to the tune of a combined factor of a million?

5.3.2 Quantum Random Circuit Algorithm

In order to make performance claims, you first need a proper benchmark. Typical benchmark suites are SPEC (`www.spec.org`) for CPU performance and recent MLPerf benchmarks (`http://mlcommons.org`) for machine learning systems. It is also known that as soon as benchmarks are published, large groups embark on efforts to optimize and tune their various infrastructures towards the benchmarks. When these efforts cross into an area where optimizations *only* work for specific benchmarks, these efforts are called *benchmark gaming*.

The challenge in setting benchmarks for quantum computing is, therefore, to build a benchmark that is meaningful, general, and yet difficult to game. Google suggested the methodology of using quantum random circuits (QRC) and cross-entropy benchmarking (XEB) (Boixo et al., 2018). QRC observes that the measurement probabilities of a random circuit follow certain patterns, which would be destroyed if there were errors or chaotic randomness in the system. XEB samples the resulting bit strings and uses statistical modeling to confirm that the chip performed a non-chaotic computation. The math used here is beyond the scope of this text, and we refer to Boixo et al. (2018) for further details.

How do you construct a random circuit? Initially, Google used a set of 2×2 operators and controlled Z gates. The choice of this particular set of gates and connectivity restrictions was influenced by the capabilities of the Sycamore chip (Google, 2019).

The problem size with a 53-qubit random circuit is huge. Assuming complex numbers of size 2^3 bytes, a traditional Schrödinger full-state simulation of the random circuit would require 2^{56} bytes or 72 PB of storage; twice that for 16-byte complex numbers. Assuming that a full-state simulation would not be realistic, the Google team used a hybrid simulation technique that combined full-state simulation with a simulation technique based on Schrödinger–Feynman path history (Rudiak-Gould, 2006). This method trades exponential space requirements for exponential runtime. The hybrid technique breaks the circuit into two (or more) chunks. It simulates each half using the Schrödinger full-state method, and for gates spanning the divided hemispheres, it uses path history techniques. The performance overhead of these hemisphere-spanning gates is very high, but their numbers are comparatively small. Based on benchmarking of the hybrid technique, as well as evaluation of full-state simulation on a supercomputer (Häner and Steiger, 2017), it was estimated that a full simulation for 53 qubits would take thousands of years, even when run on a million server class machines.

Soon after publication, methods were indeed found to game the benchmark with targeted simulation techniques for this specific circuit type, exploiting some unfortunate patterns in how the circuits were constructed. The benchmark needed to be refined. Fortunately, relatively simple changes, such as introducing new gate types, counter these techniques. Details can be found in Arute et al. (2020).

There are concerns that this choice of benchmark is a somewhat artificial proposition – an algorithm of no practical use for which no other classically equivalent algorithm exists other than quantum simulation. To play the devil's advocate, let us take a pendulum with a magnetic weight and have it swing right over an opposite magnetic pole. The movement will be highly chaotic. Simulating this behavior from some assumed starting conditions can theoretically be done in polynomial time, but enormous compute resources are required to model the motion accurately over a prolonged period of time. Even then, it is impossible to model *all* starting conditions – the proverbial flap of a butterfly wing on the other side of earth will eventually influence the motion. If we ran the simulations n times and sampled the final positions, the results would be chaotically random and differ from equivalent physical experiments. On the other hand, simply letting the pendulum swing as a physical system "performs" (and does not compute) the problem in real-time, using practically no computational resources and resulting in an equally chaotic random outcome. Have we really proven the pendulum-swing computer advantage?

This is an intriguing argument but flawed. The pendulum-swing computer is a chaotic, physical, analog, and, most importantly, non-repeatable process. The most insignificant changes in the initial conditions will lead to different, unpredictable, and unrepeatable outcomes. As such, it does not perform a computation (which is why we used the term *perform* above).

A random quantum circuit, on the other hand, is a computation. A significant change in the setup, such as modified sequences of different gates or starting from a different initial state, will change the outcome in random ways. However, small changes to parameterized gates, different noise levels, or modest exposure to errors will not cause the resulting probabilities to change meaningfully; the deviations are

bounded. In future machines, quantum error correction will make the results even more robust and repeatable.

The key argument is now the following. A random but non-chaotic calculation was computed efficiently on a quantum computer (a million runs in just 200 seconds). Computing the same result on a classical machine runs dramatically less efficiently, to the tune of thousands of years, thus proving a quantum advantage.

In all cases, it is just a matter of time until we can run something big *and* meaningful on a quantum computer, perhaps Shor's algorithm utilizing millions of qubits with error correction. In the meantime, let us take a closer look at Google's quantum circuit and estimate how long it would take us to simulate it using *our* infrastructure.

5.3.3 Circuit Construction

There are specific constraints for the gates on the Google chip, as they cannot be placed at random. We follow the original construction rules from Boixo et al. (2018). The supremacy experiment uses three types of gates, each a rotation by $\pi/2$ around an axis on a Bloch sphere. Note that the following definitions of the gates are slightly different from those we presented earlier:

$$X^{1/2} \equiv R_x(\pi/2) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix},$$

$$Y^{1/2} \equiv R_y(\pi/2) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix},$$

$$W^{1/2} \equiv R_{x+y}(\pi/2) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -\sqrt{i} \\ \sqrt{-i} & 1 \end{pmatrix}.$$

There is also a list of specific constraints for circuits:

- For each qubit, the very first and last gates must be Hadamard gates. This is reflected in a notation for circuit depth as *1-n-1*, indicating that n steps, or gate levels, must be sandwiched between Hadamard gates.

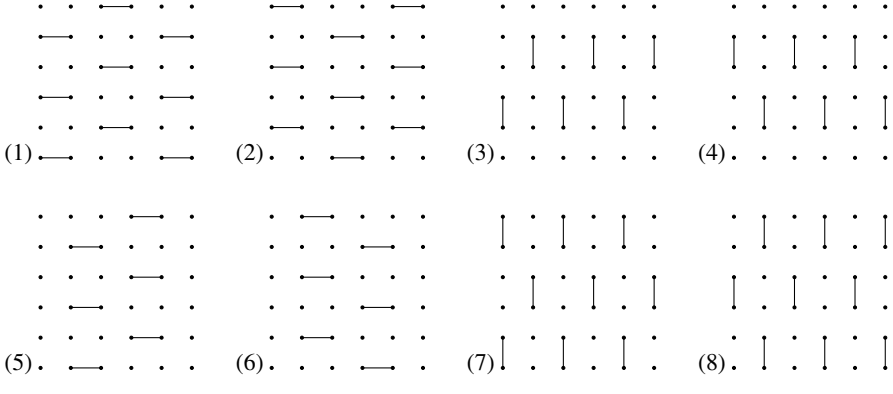


Figure 5.4 Patterns for applying controlled gates on the Sycamore chip.

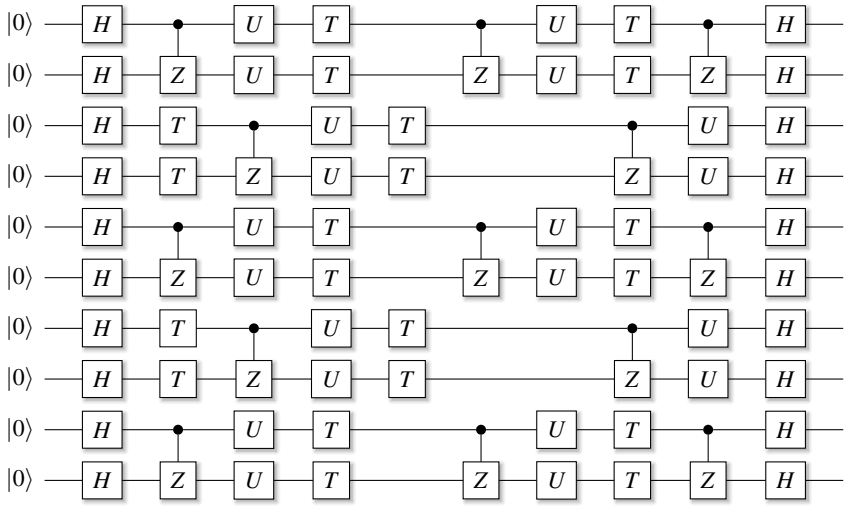


Figure 5.5 A smaller-scale, semi-random supremacy circuit.

- Apply *CZ* gates in the patterns shown in Figure 5.4, alternating between horizontal and vertical layouts.
- Apply single qubit operators $X^{1/2}$, $Y^{1/2}$, and T (or $W^{1/2}$) to qubits that are not affected by the *CZ* gates, using the criteria below. For our simulation infrastructure, which does not specialize for specific gates, the choice of gates does *not* matter with respect to computational complexity: They are all 2×2 gates. For more sophisticated methodologies, such as tensor networks, the choice of gates can make a difference.
- If the previous cycle had a *CZ* gate, apply any of the three single-qubit unitary gates.
- If the previous cycle had a non-diagonal unitary gate, apply the T gate.
- If the previous cycle had no unitary gate (except Hadamard), apply the T gate.
- Otherwise, do not apply a gate.
- Repeat the above steps for a given number of steps (which we call *depth* in our implementation).
- Measure after the final Hadamard gates.

This interpretation of the rules produces a circuit similar to the one shown in Figure 5.5. Note that there have been refinements since the initial publication; Arute et al. (2020) has the details. The main motivation for making changes was to make it harder for the new circuits to be simulated by tensor networks, the most efficient simulation technique for this type of network (Pan and Zhang, 2021). In our case, we are looking for orders of magnitude differences, so we stick to the original definition[3] and apply the corresponding fudge factors in the final estimation.

[3] As we understand it.

5.3.4 Estimation

In the implementation, it does not matter which gates are used specifically; the simulation time is the same for each gate in our infrastructure. Our estimation should be reasonably accurate as long as the gate types and density are roughly aligned with the Google circuit. Note that other simulation infrastructures, including Google's `qsimh`, do apply an additional range of optimizations to improve the simulation performance.

For example, we can construct a sample circuit with 12 qubits and a depth of *1-10-1* (a single Hadamard gate at the beginning and end, 10 random steps in the middle) and print the circuit.

Find the code
In file `src/supremacy.py`

```
def print_state(states, nbits, depth):
[...]
>>
    0  1  2  3  4  5  6  7  8  9 10 11 12
 0: h  cz cz u  t                      h
 1: h  t  cz u  t                  cz cz h
 2: h  cz u  t              cz u  t  cz h
 3: h  t     cz u  t  cz cz cz cz cz u  h
 4: h  cz cz cz u  t  cz cz u  cz u  t  h
 5: h  t  cz u  t                  cz u  h
 6: h  cz u  t              cz u  t     h
 7: h  t     cz u  t  cz cz cz cz cz u  h
 8: h  cz cz cz u  t  cz cz u  cz u  t  h
 9: h  t  cz u  t                  cz cz h
10: h  cz u  t              cz u  t  cz h
11: h  t                    cz u  cz u  h
```

The simulation is done with a function that iterates over the depth of the circuit and simulates each gate one by one. To estimate the time it would take to execute this circuit at 53 qubits, we make more assumptions:

- We assume that the single- and two-qubit gate application times are linear over the size of the state vector.
- Performance is strictly memory-bound, which means that providing additional compute units or ALUs would not help. The bandwidth with which we can bring in data limits performance.
- For a very large circuit, we know that we would have to distribute the computation over multiple machines, but we ignore the communication cost.
- We assume a number of machines and a number of cores on those machines. A relatively small number of cores on a high-core machine can saturate the available memory bandwidth, so we take a guess on what the number of reasonably utilized cores would be (we assume 255, which is very likely too high and can be adjusted).

With these assumptions, we will use the metric *Time per gate per byte in the state vector* to extrapolate the results. It is remarkably stable across qubits and circuit depths, and thus, we estimate the approximate performance of larger circuits by simulating and measuring smaller circuits. To estimate how many gates there would be in a larger circuit, we calculate a gate density, which is the number of gates in a circuit divided by `(nbits * depth)`. We present the key results in code as the following:

```
print('\nEstimate simulation time on larger circuit:\n')
gate_ratio = ngates / nbits / depth
print('Simulated circuit:')
print('  Qubits                  : {:d}'.format(nbits))
print('  Circuit Depth           : {:d}'.format(depth))
print('  Gates                   : {:.2f}'.format(ngates))
print('  State Memory            : {:.4f} MB'.format(
    2 ** (nbits-1) * 16 / (1024 ** 2)))
print('Estimated Circuit Qubits  : {}'.format(target_nbits))
print('Estimated Circuit Depth   : {}'.format(target_depth))
print('Estimated State Memory    : {:.5f} TB'.format(
    2 ** (target_nbits-1) * 16 / (1024 ** 4)))
print('Machines used             : {}'.format(flags.FLAGS.machines))
print('Estimated cores per server: {}'.format(flags.FLAGS.cores))
print('Estimated gate density    : {:.2f}'.format(gate_ratio))

estimated_sim_time_secs = (
    # time per gate per byte
    (duration / ngates / (2**(nbits-1) * 16))
    # gates
    * target_nbits
    # gate ratio scaling factor to circuit size
    * gate_density
    # depth
    * target_depth
    # memory
    * 2**(target_nbits-1) * 16
    # number of machines
    / flags.FLAGS.machines
    # Active core per machine
    / flags.FLAGS.cores)
print('Estimated for {} qbits: {:.2f} y or {:.2f} d or ({:.0f} sec)'
      .format(target_nbits,
              estimated_sim_time_secs / 3600 / 24 / 365,
              estimated_sim_time_secs / 3600 / 24,
              estimated_sim_time_secs))
```

For the specific result, we assume that the target circuit has 53 qubits and is run on 100 machines, each with 255 fully available cores. The number of gates in our simulation seems to roughly align with the number of gates published by Google, though not exactly.[4] For the example parameters, the estimation results are the following:

[4] There is a bit of ambiguity in the description of the algorithm to construct the circuit.

```
Estimate simulation time for larger circuit:
Simulated smaller circuit:
  Qubits                  : 20
  Circuit Depth           : 20
  Gates                   : 320.00
  State Memory            : 8.0000 MB
Estimated Circuit Qubits  : 53
Estimated Circuit Depth   : 20
Estimated State Memory    : 65536.00000 TB
Machines used             : 100
Estimated cores per server: 255
Estimated gate density    : 0.80
Estimated for 53 qbits: 0.01 y or 4.81 d or (415780 sec)
Estimated sim for FULL experiment, 53 qbits: 13184.29 years
```

With all of our simplifying assumptions, we arrive at a simulation time of 4.81 days for a single simulation of the 53-qubit circuit. Of course, these parameters could be made more realistic. For example, how would we provision 72 PB of memory on just 100 machines? Assuming that we can provision 1 TB per server, we would need at least 72K hosts. At this scale, we cannot ignore communication costs. At the same time, we are using our non-optimized infrastructure. You may want to experiment with more realistic settings.

The supremacy experiment performed and sampled this circuit 1,000,000 times in about 200 seconds. Given a duration of 4.81 days for a single run, we would need about 13,184.3 years to simulate the circuit an equal number of times. For comparison, let us look at the massive Summit supercomputer (Oak Ridge National Laboratory, 2021). It can theoretically perform up to 10^{17} single-precision floating-point operations per second. Calculating 2^{53} equivalents of 2×2 matrix multiplications requires 2^{56} floating-point operations. At 100 percent utilization, it would take Summit just a few seconds to simulate one of the iterations or perhaps a few months to take all of the 1,000,000 samples!

To store a full state of 53 qubits, we need 72 PB bytes of storage. Summit has an estimated 2.5 PB of RAM on all sockets and 250 PB of secondary storage. This means we should expect the simulation to encounter high communication overhead when moving data from permanent storage to RAM. Much of the permanent storage would also have to be reserved for this experiment. The IBM researchers found an impressive way to minimize data transfers, a major contribution by Pednault et al. (2019). With this technique, a slowdown of about $500\times$ was anticipated, leading to the estimate that the full simulation could run in about two days.

Now let us answer the question that started this section: Where does the discrepancy of 10,000 years versus days come from? This is a factor of about $1,000,000$, after all.

The Google Quantum X team based their estimations on a different simulator architecture (Markov et al., 2018), assuming that a full-state simulation is not realistic. The simulation techniques were benchmarked on a smaller scale. The results of the full-state simulation were evaluated on a supercomputer. From these data points, the

computational costs were extrapolated to 1,000,000 machines, arriving at an estimate of 10,000 years of simulation time for 53 qubits and a circuit depth of 20.

The IBM researchers, on the other hand, found an elegant way to squeeze the problem into one of the biggest supercomputers in the world. The results are only estimated; an experiment was not performed. It is difficult to determine how realistic the estimates are in practice because, at petabyte scale, other factors have to be taken into account, for example, disk error rates. This also assumes that most of the machine's secondary storage was committed to the experiment.

Is there a right or wrong? The answer is *no* because we compare apples to oranges. The evaluated simulation techniques are different based on different assumptions of what can realistically run on a supercomputer. The supremacy experiment was physically run, while the Summit paper was only estimated. Even if the physical simulation took just a day on Summit, adding a handful of additional qubits will exhaust its storage capacity. The simulation technique would have to change and trade storage requirements for simulation time, similar to the Schrödinger–Feynman path history technique (Rudiak-Gould, 2006). At that point, and only then would we be able to make a more fair apples-to-apples comparison.

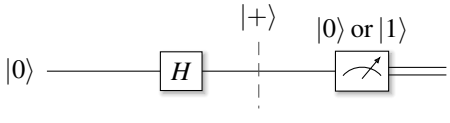
It is safe to anticipate that other clever simulation techniques will emerge. However, as long as $\mathrm{BPP} \subset \mathrm{BQP}$ is true, it is also safe to assume that additional qubits or moderately modified benchmarks will again defeat attempts to simulate these circuits classically.

6 Algorithms Exploiting Entanglement

In this chapter, we further familiarize ourselves with the basics of gates and states and study several entanglement-based algorithms. The calculations are explicit and detailed since this is early in the book and we have yet to get used to the code and mathematical formalism. We start with the "Hello World" of quantum computing, a quantum random number generator. We quickly follow this with three algorithms exploiting entanglement: quantum teleportation, superdense coding, and entanglement swapping. We conclude with a discussion of the CHSH game, a variant of Bell's inequalities. The CHSH game may be the most complex of the algorithms presented in this chapter. It will also lead us to philosophical aspects of quantum mechanics and reality itself.

6.1 Quantum Hello World

Every programming system introduces itself with the equivalent of a "Hello World" program. In quantum computing, this may be a *random number generator*. We are ready to discuss it now using the material presented so far. It is the simplest possible quantum circuit that does something meaningful, and it does so with just one qubit and one gate:



The Hadamard gate puts the state in an equal superposition of the basis states $|0\rangle$ and $|1\rangle$, namely

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

On measurement,[1] the state will collapse to either $|0\rangle$ or $|1\rangle$ with 50% probability for each case. You can validate this with these two lines of code to make the state, apply the Hadamard gate, and print the probabilities:

[1] In the computational basis.

```
  psi = ops.Hadamard()(state.zeros(1))
  psi.dump()
>>
0.70710678+0.00000000i  |0>  50.0%
0.70710678+0.00000000i  |1>  50.0%
```

Since we can construct a random number generator with just a single qubit, which we interpret as a classical bit after measurement, bundling multiple qubits in parallel or sequence allows the generation of random numbers of any bit width. By *random*, we mean true, intrinsically quantum randomness, not classical pseudo-randomness.

This circuit can barely be called a circuit, not to mention an *algorithm* (even though we call it that in Section 10.2 on amplitude amplification). It only has one gate, so it is the simplest of all possible circuits. Nevertheless, it exploits crucial quantum computing properties, namely superposition and probabilistic collapse of the wave function on measurement. It is trivial, and it is not. Both at the same time. A true quantum circuit.

6.2 Quantum Teleportation

We now describe the quantum algorithm with one of the most intriguing algorithm names of all time – quantum teleportation (Bennett et al., 1993). This algorithm is a small example of the fascinating field of quantum information, which includes encryption and error correction. This type of algorithm exploits entanglement to send a quantum state between spatially separate locations without transmitting any physical qubits, only information!

As is typical in quantum computing, the algorithmic story begins with our protagonists, Alice and Bob, the placeholders for the distinct systems A and B. At the beginning of the story, they are together in a lab on Earth and create an entangled pair of qubits, the Bell state $|\Phi^+\rangle = \beta_{00}$. Let us mark the first qubit as Alice's and the second one as belonging to Bob in obvious notation:

$$|\Phi^+\rangle = \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}}.$$

As we tell the story, we will weave in code snippets to make the mathematical concepts concrete and allow experimentation. After creating the state, they each take one of the qubits and physically separate them. Alice goes to the Moon, and Bob ships off to Mars. We should not worry about how they are getting their supercooled quantum qubits across the solar system. No one said that teleportation was easy. In code, we start with a call to create the Bell state `psi`:

Find the code
In file `src/teleportation.py`

```
def main(argv):
  # Step 1: Alice and Bob share an entangled pair and separate.
  psi = bell.bell_state(0, 0)
```

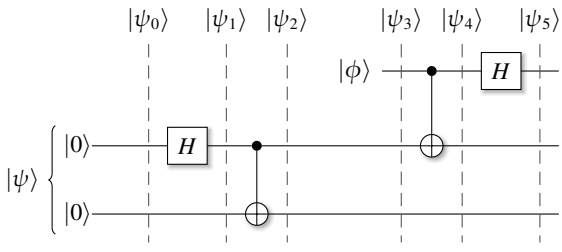


Figure 6.1 Quantum teleportation in circuit notation.

Sitting there on the Moon, Alice happens to be in possession of this other qubit $|\phi\rangle$, which is in a specific state with probability amplitudes α and β:

$$|\phi\rangle = \alpha|0\rangle + \beta|1\rangle.$$

Alice does not know what the values of α and β are and cannot measure them because measuring the qubit would destroy their superposition.[2] Alice wants to communicate α and β to Bob so that he will be in possession of the state of $|\phi\rangle$. On measurement, he will obtain a basis state of $|\phi\rangle$ with the corresponding probabilities. How can Alice "send" or "teleport" the state of $|\phi\rangle$ to Bob? She can do this by exploiting the entangled qubit she already has from the time before her Moon travel.

In code, we create the qubit $|\phi\rangle$ with defined values for α (0.6) and β to check later whether Alice has teleported the state to Bob correctly. The combined state of the new qubit $|\phi\rangle$ with the qubit $|\psi\rangle$ she brought with her from Earth, the one that is entangled with Bob's qubit, is their tensor product that we store in `combo`:

```
# Step 2: Alice wants to teleport a qubit |phi> to Bob
#         with |phi> = a|0> + b|1>, a^2 + b^2 == 1:
a = 0.6
b = math.sqrt(1.0 - a * a)
phi = state.qubit(a, b)
print('Quantum Teleportation')
print(f'Start with EPR Pair a={a:.2f}, b={b:.2f}')

# Produce combined state 'combo'.
combo = phi * psi
```

The state is now $|\phi, \Phi^+\rangle$. She continues and connects $|\phi\rangle$ and $|\Phi^+\rangle$ with a controlled Not gate, followed by a final Hadamard gate on qubit $|\phi\rangle$. This reverse entangler circuitry (see Section 2.11.2), with a first controlled Not gate followed by a Hadamard gate, is also called making a *Bell measurement* since the circuit is the adjoint of the circuit used to create Bell states. The whole procedure is shown in Figure 6.1 as a circuit.

[2] For this algorithm, it doesn't matter whether either Alice or Bob knows these parameters.

```
alice = ops.Cnot(0, 1)(combo)
alice = ops.Hadamard()(alice, idx=0)
```

Let's analyze how the state progresses from left to right and explain the math in detail. Starting in the lab, before the first Hadamard gate, the state is just the tensor product of the two qubits:

$$|\psi_0\rangle = |0_A\rangle \otimes |0_B\rangle = |0_A 0_B\rangle.$$

The first Hadamard gate creates a superposition of qubit $|0_A\rangle$:

$$|\psi_1\rangle = \frac{|0_A\rangle + |1_A\rangle}{\sqrt{2}} \otimes |0_B\rangle.$$

The controlled Not gate entangles the two qubits and generates a Bell state, as we have seen before in Section 2.11.2 on entanglement. Note that up to this point, Bob and Alice are still at the same location in the lab on Earth:

$$|\psi_2\rangle = \Phi^+ = \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}}.$$

Alice has now traveled to the Moon, where she has the other qubit $|\phi\rangle$. The combined state is the tensor product of her new qubit $|\phi\rangle$ with the qubit she brought with her from Earth, resulting in state

$$\begin{aligned}|\psi_3\rangle &= \big(\alpha|0\rangle + \beta|1\rangle\big) \otimes \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}} \\ &= \frac{\alpha|0\rangle\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big) + \beta|1\rangle\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big)}{\sqrt{2}}.\end{aligned}$$

Now she applies the controlled Not from $|\phi\rangle$ to her part of the entangled qubit (now the middle qubit 1 in the circuit). The $|1\rangle$ component of $|\phi\rangle$ will flip the controlled qubit. As a result, qubits $|0_A\rangle$ and $|1_A\rangle$ flip in the right-hand side of the expression $|\psi_3\rangle$ to

$$|\psi_4\rangle = \frac{\alpha|0\rangle\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big) + \beta|1\rangle\big(|1_A 0_B\rangle + |0_A 1_B\rangle\big)}{\sqrt{2}}.$$

Finally, we apply the Hadamard gate to $|\phi\rangle$, resulting in

$$|\psi_5\rangle = \frac{\alpha\big(|0\rangle + |1\rangle\big)\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big) + \beta\big(|0\rangle - |1\rangle\big)\big(|1_A 0_B\rangle + |0_A 1_B\rangle\big)}{2}.$$

We multiply out $|\psi_5\rangle$ to get

$$\begin{aligned}|\psi_5\rangle = \frac{1}{2}\Big(&\alpha\big(|0\,0_A 0_B\rangle + |0\,1_A 1_B\rangle + |1\,0_A 0_B\rangle + |1\,1_A 1_B\rangle\big) \\ &+ \beta\big(|0\,1_A 0_B\rangle + |0\,0_A 1_B\rangle - |1\,1_A 0_B\rangle - |1\,0_A 1_B\rangle\big)\Big).\end{aligned}$$

We are almost there. Notice how all the last qubits in $|\psi_5\rangle$ are Bob's. Alice has the first two qubits in her possession. If we regroup the above expression and isolate the

first two qubits, we arrive at our target expression. We can omit the subscripts as the first two qubits are Alice's, and the last is Bob's.

$$
\begin{aligned}
|\psi_5\rangle = \frac{1}{2}\Big(& |00\rangle\big(\alpha|0\rangle + \beta|1\rangle\big) \\
& +|01\rangle\big(\beta|0\rangle + \alpha|1\rangle\big) \\
& +|10\rangle\big(\alpha|0\rangle - \beta|1\rangle\big) \\
& +|11\rangle\big(-\beta|0\rangle + \alpha|1\rangle\big)\Big).
\end{aligned}
$$

Alice can measure her first two qubits while leaving the superposition of Bob's third qubit intact. On her measurement, the state collapses and leaves Bob's qubit in a state with a combination of parameters α and β that depends on Alice's measurement outcome. As the final step, Alice *tells* Bob about her measurement result over a classic communication channel — she may be able to teleport a state, but she will not be able to do it faster than the speed of light. If she measured:

$|00\rangle$ - Bob's qubit is now in state $\alpha|0\rangle + \beta|1\rangle$.
$|01\rangle$ - Bob's qubit is now in state $\beta|0\rangle + \alpha|1\rangle$.
$|10\rangle$ - Bob's qubit is now in state $\alpha|0\rangle - \beta|1\rangle$.
$|11\rangle$ - Bob's qubit is now in state $-\beta|0\rangle + \alpha|1\rangle$.

At this point, Alice has teleported the state $|\phi\rangle$ to Bob successfully. For Bob to know how to reconstruct the state of $|\phi\rangle$ for his qubit, she still had to classically communicate her measurement results. However, the spooky action at a distance "modified" Bob's entangled qubit on Mars to obtain the probability amplitudes from Alice's qubit $|\phi\rangle$, which she created on the Moon. The spooky action is truly spooky.

The final step, depending on Alice's classical communication, is to apply gates to Bob's qubit to put it in the desired teleported state of $\alpha|0\rangle + \beta|1\rangle$:

- If she sends 00, nothing needs to be done.
- If she sends 01, Bob must flip the amplitudes by applying the X gate.
- If she sends 10, Bob flips the phase by applying the Z gate.
- Similarly, for 11, Bob applies a Z gate and an X gate.

After this, Bob's qubit on Mars will be in the state of Alice's original qubit $|\phi\rangle$ on the Moon. Teleportation completed. Minds blown.

We perform four experiments corresponding to the four possible measurement results. For each experiment, we pretend that Alice measured a specific result and apply the corresponding decoder gates to Bob's qubit. Then we measure Bob's qubit and confirm that it matches expectations.

```
def alice_measures(alice: state.State,
            expect0: np.complexfloating, expect1: np.complexfloating,
            qubit0: np.complexfloating, qubit1: np.complexfloating):
  _, alice0 = ops.Measure(alice, 0, tostate=qubit0)
  _, alice1 = ops.Measure(alice0, 1, tostate=qubit1)
  if qubit0 == 0 and qubit1 == 0:
```

```
    pass
  if qubit0 == 0 and qubit1 == 1:
    alice1 = ops.PauliX()(alice1, idx=2)
  if qubit0 == 1 and qubit1 == 0:
    alice1 = ops.PauliZ()(alice1, idx=2)
  if qubit0 == 1 and qubit1 == 1:
    alice1 = ops.PauliX()(ops.PauliZ()(alice1, idx=2), idx=2)

  p0, _ = ops.Measure(alice1, 2, tostate=0, collapse=False)
  p1, _ = ops.Measure(alice1, 2, tostate=1, collapse=False)

  # We sqrt() the probability to get the (original) amplitude.
  bob_a = math.sqrt(p0.real)
  bob_b = math.sqrt(p1.real)
  print('Teleported (|{:d}{:d}>)   a={:.2f}, b={:.2f}'.format(
      qubit0, qubit1, bob_a, bob_b))

  if (not math.isclose(expect0, bob_a, abs_tol=1e-6) or
      not math.isclose(expect1, bob_b, abs_tol=1e-6)):
    raise AssertionError('Invalid result.')
```

As a final step, we run the four experiments and inspect the output:

```
  # Alice measures and communicates the result to Bob.
  alice_measures(alice, a, b, 0, 0)
  alice_measures(alice, a, b, 0, 1)
  alice_measures(alice, a, b, 1, 0)
  alice_measures(alice, a, b, 1, 1)
>>
Quantum Teleportation
Start with EPR Pair a=0.60, b=0.80
Teleported (|00>)   a=0.60, b=0.80
Teleported (|01>)   a=0.60, b=0.80
Teleported (|10>)   a=0.60, b=0.80
Teleported (|11>)   a=0.60, b=0.80
```

6.3 Superdense Coding

Superdense coding, yet another algorithm with a super cool name, takes the core idea from quantum teleportation and turns it on its head. It uses entanglement to communicate classical bits with the help of a smaller number of qubits. This protocol was suggested by Charles H. Bennett and Stephen Wiesner in 1970 and was later published in Bennett et al. (1992).

We start with a familiar story. Alice and Bob again share an entangled pair of qubits. Alice takes her qubit to the Moon, while Bob takes his qubit to Mars. Sitting on the Moon, Alice wants to communicate two classical bits to Bob. Superdense coding encodes two classical bits and transmits their values to Bob by physically

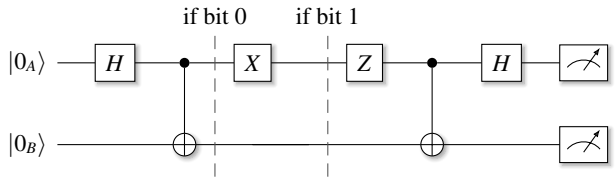


Figure 6.2 Superdense coding in circuit notation.

shipping just a single qubit. Again, we don't care how this is done in the real world (nobody said superdense coding was easy). Two qubits are still needed in total, but the communication itself is done with just a single qubit.

There exists no other classical compression scheme that would allow for the compression of two classical bits into one. Of course, here we are dealing with entangled qubits, which have two degrees of freedom (two angles define the position on a Bloch sphere). The challenge is to exploit this fact to encode information. To understand how this works, we begin with an entangled pair of qubits again.

PY

Find the code
In file `src/superdense.py`

```
# Step 1: Alice and Bob share an entangled pair and separate.
psi = bell.bell_state(0, 0)
```

Alice manipulates her qubit on the Moon according to the rules of how to encode two classical bits into a single qubit, as shown below. In a twist of events, she will classically ship her qubit to Bob's Mars station. There, Bob will disentangle and measure. Based on the measurement results, he can derive Alice's original two classical bits. Alice sent just one qubit to allow Bob to restore two classical bits.

To start the process, Alice manipulates her entangled qubit in the following way. She wants to communicate the two classical bits b_0 and b_1.

- If classical bit b_0 is 1, she applies the X gate.
- If classical bit b_1 is 1, she applies the Z gate.
- Of course, if both bits b_0 and b_1 are 1, she applies both the X and Z gate.
- And, for completeness, if both bits b_0 and b_1 are 0, nothing needs to be done.

The whole procedure is shown in circuit notation in Figure 6.2. In the code, the two classical bits encode four possible cases 00, 01, 10, and 11. For experimentation, we iterate over these four combinations in `main` below:

```
def alice_manipulates(psi: state.State,
                      bit0: int, bit1: int) -> state.State:
  ret = ops.Identity(2)(psi)
  if bit0:
    ret = ops.PauliX()(ret)
```

```
  if bit1:
    ret = ops.PauliZ()(ret)
  return ret

def main(argv):
  for bit0 in range(2):
    for bit1 in range(2):
      psi_alice = alice_manipulates(psi, bit0, bit1)
      bob_measures(psi_alice, bit0, bit1)
```

Let's work through the math. The entangled pair is initially in the Bell state

$$\left|\Phi^{+}\right\rangle = \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}}.$$

If the classical bit b_0 is 1, Alice applies an X gate to her qubit, which turns the state into the different Bell state $\left|\Psi^{+}\right\rangle$:

$$(X \otimes I)\left|\Phi^{+}\right\rangle = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \left|\Psi^{+}\right\rangle.$$

If the classical bit b_1 is 1, Alice applies a Z gate, which changes the state and flips the first subscript of the Bell state:

$$(Z \otimes I)\left|\Phi^{+}\right\rangle = \frac{|0_A 0_B\rangle - |1_A 1_B\rangle}{\sqrt{2}} = \left|\Phi^{-}\right\rangle.$$

Finally, if both bits b_0 and b_1 are 1, Alice uses an X gate and a Z gate and changes the state to $\left|\Psi^{-}\right\rangle$:

$$(Z \otimes I)(X \otimes I)\left|\Phi^{+}\right\rangle = i(Y \otimes I)\left|\Phi^{+}\right\rangle = \frac{|0_A 1_B\rangle - |1_A 0_B\rangle}{\sqrt{2}} = \left|\Psi^{-}\right\rangle.$$

After receiving Alice's qubit, Bob applies a reverse entangler circuit between his entangled qubit and the qubit he just received, with a controlled Not gate followed by the Hadamard gate. This is shown as the final two gates before the measurement operator in Figure 6.2.

Going through the entangler circuit in reverse uncomputes the entanglement and changes the state to one of the computational basis states $|00\rangle, |01\rangle, |10\rangle$, or $|11\rangle$, depending on the value of the original classical bits. The probability of each result will be 100%. With a measurement, Bob can reliably determine the classical bit values Alice wanted to communicate.

```
def bob_measures(psi: state.State, expect0: int, expect1: int) -> None:
  psi = ops.Cnot(0, 1)(psi)
  psi = ops.Hadamard()(psi)

  p0, _ = ops.Measure(psi, 0, tostate=expect1)
  p1, _ = ops.Measure(psi, 1, tostate=expect0)
```

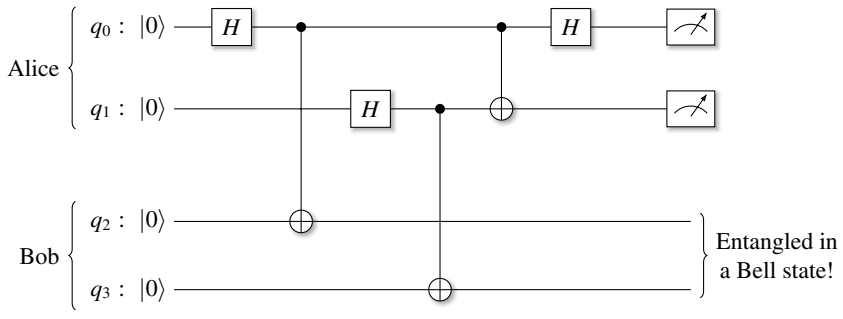


Figure 6.3 Entanglement teleportation circuit.

```
if (not math.isclose(p0, 1.0, abs_tol=1e-6) or
    not math.isclose(p1, 1.0, abs_tol=1e-6)):
  raise AssertionError(f'Invalid Result p0 {p0} p1 {p1}')
print(f'Expected/matched: {expect0}{expect1}.')
```

Based on how Alice manipulated the qubit, we should get the expected results:

```
Expected/matched: 00
Expected/matched: 01
Expected/matched: 10
Expected/matched: 11
```

6.4 Entanglement Swapping

Another algorithm of this type, which we only briefly mention here, is *entanglement swapping*, also called *entanglement teleportation* (Berry and Sanders, 2002). In this story, Alice and Bob each have a pair of entangled qubits. Unlike before, each keeps their pair of qubits to themselves. This is illustrated in Figure 6.3. Alice has qubits q_0 and q_1, Bob has q_2 and q_3. However, before physically separating, they entangle q_0 with q_2 and q_1 with q_3. Note that at this time, there is no entanglement between Bob's qubits q_2 and q_3.

Now Alice performs a Bell measurement on her two qubits. Here is where the magic happens: After measurement, Bob's qubits q_2 and q_3 will be in an entangled Bell state! We simulate this miraculous effect in the open-source repository with just a few lines of code similar to these:

PY

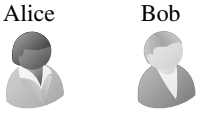
Find the code
In file src/entanglement_swap.py

```
def main(argv):
  qc = circuit.qc('Entanglement swap')
  qc.reg(4, 0)

  # Alice has qubits 0, 1, Bob has qubits 2, 3. Entangle 0, 2 and 1, 3:
  qc.h(0)
  qc.cx(0, 2)
  qc.h(1)
  qc.cx(1, 3)

  # Alice performs a Bell measurement between her qubits 0 and 1,
  qc.cx(0, 1)
  qc.h(0)

  # Measure and check results (all combinations of 0 and 1).
  qc.measure_bit(0, 0, collapse=True)
  qc.measure_bit(1, 0, collapse=True)
  [...]
```

6.5 The CHSH Game

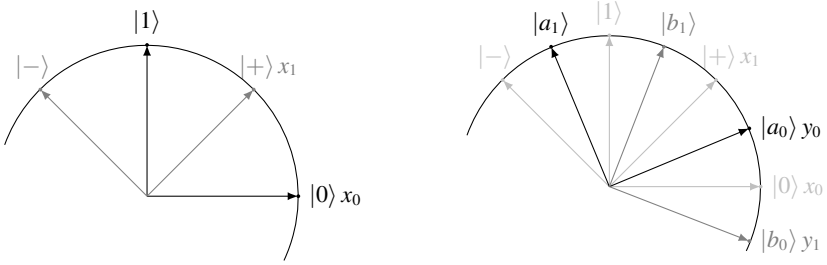
The CHSH game, named after its authors (Clauser, Horne, Shimony, Holt, 1969), is an implementation of their CHSH equality, a simplified form of the Bell inequalities (Bell, 1964). The CHSH game is a powerful demonstration of the use of entanglement as a resource to obtain results that go beyond what can be achieved classically.

In the game, Alice and Bob each receive a random classical bit. Alice receives bit x, and Bob receives bit y, as indicated in Figure 6.4. Both x and y are drawn from a random distribution. Alice and Bob cannot communicate during the game, but they can agree on a strategy before the game starts. Based on the bit values they each receive, Alice and Bob will respond with bit values a and b. The goal of the game is to produce matching bits a and b, except when both $x = y = 1$. In this case, a and b



Input	x	y
Output	a	b
Goal	$xy = a \oplus b$	

Figure 6.4 The CHSH Game.



(a) Alice measures in bases $\{|0\rangle, |1\rangle\}$ or $\{|+\rangle, |-\rangle\}$.

(b) Bob measures in bases $\{|a_0\rangle, |a_1\rangle\}$ or $\{|b_0\rangle, |b_1\rangle\}$, rotated by $\pm\pi/8$.

Figure 6.5 Alice and Bob use different bases for measurement.

must differ. In closed form, the winning condition can be written as $xy = a \oplus b$, with $\oplus$ being the usual addition modulo 2 (XOR).

A plausible classical strategy for Alice and Bob is to always respond with $a = b = 0$, a strategy with a success probability of $3/4$. With some thought, you can convince yourself that this is indeed the best possible outcome for the classical case.

Let us now explore how quantum entanglement can help improve their chances of winning. Before the game starts, Alice and Bob create an entangled Bell state $|\beta_{00}\rangle$ and each takes their qubit with them before physically separating:

$$|\beta_{00}\rangle = |\Psi^+\rangle = \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}}.$$

Depending on the classical bits x and y they receive, Alice and Bob will measure in specific bases. When Alice receives a classical bit $x = 0$, she measures in the computational basis $\{|0\rangle, |1\rangle\}$. If she gets bit $x = 1$, she measures in the Hadamard basis $\{|+\rangle, |-\rangle\}$, as shown in Figure 6.5(a).

If Bob receives his classical bit $y = 0$, he measures in the basis $\{|a_0\rangle, |a_1\rangle\}$, which is the computational basis rotated by $\pi/8$:

$$|a_0\rangle = \cos\frac{\pi}{8}|0\rangle + \sin\frac{\pi}{8}|1\rangle,$$
$$|a_1\rangle = -\sin\frac{\pi}{8}|0\rangle + \cos\frac{\pi}{8}|1\rangle.$$

If Bob receives $y = 1$, he measures in the basis $\{|b_0\rangle, |b_1\rangle\}$, which is the computation basis rotated by $-\pi/8$, as shown in Figure 6.5(b):

$$|b_0\rangle = \cos\frac{\pi}{8}|0\rangle - \sin\frac{\pi}{8}|1\rangle$$
$$|b_1\rangle = \sin\frac{\pi}{8}|0\rangle + \cos\frac{\pi}{8}|1\rangle.$$

When measuring in these bases, the results for Alice and Bob will be correlated because the qubits were entangled. In the four cases of the classical bits x and y, and with a little bit of trigonometry, we find:

x=0, y=0: The bases $|0\rangle$ and $|a_0\rangle$ are separated by an angle of $\pi/8$. If Alice measures $|0\rangle$, the probability that Bob will measure $|a_0\rangle$ is $\cos^2 \pi/8$.

x=0, y=1: The bases $|0\rangle$ and $|b_0\rangle$ are separated by an angle of $-\pi/8$. If Alice measures $|0\rangle$, the probability that Bob will measure $|b_1\rangle$ is also $\cos^2 \pi/8$.

x=1, y=0: The bases $|+\rangle$ and $|a_0\rangle$ are separated by an angle of $-\pi/8$. If Alice measures $|1\rangle$, the probability that Bob will measure $|a_0\rangle$ is $\cos^2 \pi/8$ again.

x=1, y=1: The bases $|+\rangle$ and $|b_0\rangle$ are separated by an angle of $3\pi/8$. If Alice measures $|1\rangle$, the probability that Bob will measure $|b_0\rangle$ is $\cos^2(3\pi/8)$. However, in this case, we want the results to be different according to the rules of the game. The probability that the results differ is $1 - \cos^2(3\pi/8) = \sin^2(3\pi/8) = \cos^2 \pi/8$.

In all four cases, entanglement improved the chances of winning to $\cos^2 \pi/8 \approx 0.85$, or 85%. Recall that the best classical strategy had a maximum 75% chance of winning. The entanglement led to a better outcome. It can also be shown that rotating the bases by $\pi/8$ is the optimal choice (Tsirelson, 1980).

The key argument here is that since the classical bits and, hence, the measurement bases are chosen at random, there cannot be any unexplained variables hidden in the Bell state to predetermine the measurement result. This seems to confirm that quantum physics is *non-local*, where entangled qubits can affect each other, even when separated by vast distances.

Let us put philosophy aside and try this in code. We simulate a random measurement in function `measure` and carry out the actual experiment in function `run_experiment`.

Find the code
In file `src/chsh.py`

```
def measure(psi: state.State):
  """Simulated, probabilistic measurement."""
  r = random.random() - 0.001
  total = 0
  for i in range(len(psi)):
    total += psi[i] * psi[i].conj()
    if r < total:
      psi = helper.val2bits(i, 2)
      return psi[0], psi[1]

def run_experiments(experiments: int, alpha: float) -> float:
  wins = 0
  for _ in range(experiments):
    x = random.randint(0, 1)
    y = random.randint(0, 1)
    psi = bell.bell_state(0, 0)

    if x == 0:
      pass
    if x == 1:
      psi = ops.RotationY(2.0 * alpha)(psi, 0)
```

```
    if y == 0:
      psi = ops.RotationY(alpha)(psi, 1)
    if y == 1:
      psi = ops.RotationY(-alpha)(psi, 1)

    a, b = measure(psi)
    if x * y == (a + b) % 2:
      wins += 1
  return wins / experiments * 100.0
```

With these two functions in place, we can run two types of experiments. First, we check the experimental winning percentage, which should be around 86%. Second, we iterate over multiple angles and plot horizontal bars scaled to the winning percentages. You can see this in the output below.

```
def main(argv):
  print('Quantum CHSH evaluation.')
  percent = run_experiments(1000, 2.0 * np.pi / 8)
  print(f'Optimal Angle 2 pi / 8, winning: {percent:.1f}%')
  assert percent > 80.0, 'Incorrect result, should reach above 80%'

  # Run a few incremental experiments.
  steps = 32
  inc_angle = (2.0 * np.pi / 8) / (steps / 2)
  for i in range(0, 66, 2):
    percent = run_experiments(500, inc_angle * i)
    s = '(opt)' if i == 16 else ''
    print(
        f'{i:2d} * Pi/64 ={inc_angle * i:.2f}: winning: {percent:5.2f}%'
        f'{"#" * int(percent/3)}{s}'
    )
>>
Quantum CHSH evaluation.
Optimal Angle 2 pi / 8, winning: 86.2%
[...]
10 * Pi/64 = 0.49: winning: 81.80% ############################
12 * Pi/64 = 0.59: winning: 85.20% #############################
14 * Pi/64 = 0.69: winning: 85.40% #############################
16 * Pi/64 = 0.79: winning: 86.40% ############################(opt)
18 * Pi/64 = 0.88: winning: 83.60% ############################
20 * Pi/64 = 0.98: winning: 84.40% #############################
[...]
36 * Pi/64 = 1.77: winning: 33.60% ###########
38 * Pi/64 = 1.87: winning: 29.40% #########
40 * Pi/64 = 1.96: winning: 24.40% ########
[...]
```

7 State Similarity Tests

This chapter discusses a few techniques that allow us to determine how close two states are to each other. This is important in various algorithms, as we shall see later. Sometimes we use the term *overlap* to make a statement about how close states are to each other, and sometimes we use the term *similarity*. There are subtle differences between the two:

- The overlap between two quantum states $|\psi\rangle$ and $|\phi\rangle$ is defined as the absolute value of the inner product $|\langle\psi|\phi\rangle|$. As we already know, this product is 0 if the states are orthogonal and 1 if there is complete overlap.
- The similarity between two states is a more general notion than overlap. It can be expressed in different ways, such as by the trace distance between the density operators of two states. We can find an example of this in Section 9.4.4.

We will use the terms overlap and similarity interchangeably, but keeping this distinction in mind is helpful. Also, since we are still early in the book, the math here is still very detailed to help us get used to some of the typical algebraic transformations in quantum computing.

7.1 Swap Test

The *quantum swap test* measures the similarity between two quantum states without directly measuring the two states (Buhrman et al., 2001). Instead, the trick is to introduce an *ancilla* qubit and a controlled Swap gate and *only* measure the ancilla. The two states were very different if the resulting measurement probability for the basis state $|0\rangle$ is close to 0.5. A measurement probability closer to 1.0 means the two states were very similar. In the physical world, we must run the experiment multiple times to measure the probabilities. Our implementation will only look at the probabilities encoded in the state vector.

The swap test is an example of a quantum algorithm that allows the derivation of an *indirect* measure. It will not tell us what the two states are, which would constitute a measurement. It also does not tell us which state has the larger amplitude in a given basis. However, it does tell us how similar two unknown states are without measuring them. The circuit to measure the proximity of the qubit states $|\psi\rangle$ and $|\phi\rangle$ is shown in Figure 7.1.

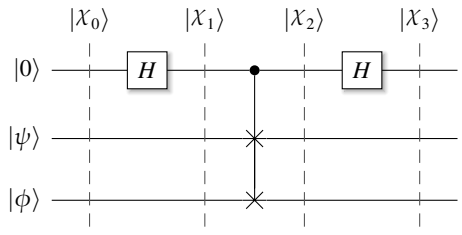


Figure 7.1 The basic swap test circuit.

Let us denote the state of the 3-qubit system by $|\chi\rangle$ and see how it changes going from left to right. At the start of the circuit, the state $|\chi_0\rangle$ is the tensor product of the three qubits:

$$|\chi_0\rangle = |0,\psi,\phi\rangle.$$

The Hadamard gate on qubit 0 superimposes the system to state

$$|\chi_1\rangle = \frac{1}{\sqrt{2}}\big(\underbrace{|0,\psi,\phi\rangle}_{a} + \underbrace{|1,\psi,\phi\rangle}_{b}\big).$$

The controlled Swap gate modifies the second half of this expression due to the controlling $|1\rangle$ state of the ancilla qubit. In the part marked b above, $|\phi\rangle$ and $|\psi\rangle$ are swapped and state $|\chi_2\rangle$ becomes

$$|\chi_2\rangle = \frac{1}{\sqrt{2}}\big(|0,\psi,\phi\rangle + |1,\phi,\psi\rangle\big).$$

The second Hadamard gate now superimposes further. The first part of state $|\chi_2\rangle$ turns into

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

where the Hadamard superposition of the $|0\rangle$ state introduces a plus sign. On the other hand, the second term in $|\chi_2\rangle$ becomes

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

because the Hadamard superposition of $|1\rangle$ introduces a minus sign. Combined, this results in state $|\chi_3\rangle$ in Figure 7.1:

$$|\chi_3\rangle = \frac{1}{2}\big(|0,\psi,\phi\rangle + |1,\psi,\phi\rangle + |0,\phi,\psi\rangle - |1,\phi,\psi\rangle\big).$$

We can factor out the first qubit and simplify to

$$|\chi_3\rangle = \frac{1}{2}\,|0\rangle\,\underbrace{\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big)}_{\chi_+} + \frac{1}{2}\,|1\rangle\,\underbrace{\big(|\psi,\phi\rangle - |\phi,\psi\rangle\big)}_{\chi_-},$$

where we name the terms next to $|0\rangle$ and $|1\rangle$ as χ_+ and χ_-. Now we measure the first ancillary qubit. We *only* consider measurements that result in state $|0\rangle$ for the ancilla

and ignore all others. To compute the probability of measuring $|0\rangle$, we usually take the probability amplitude and compute the squared norm. However, we don't have just an amplitude; we have the more complex states χ_+ and χ_-.

To obtain the probabilities for $|0\rangle$ or $|1\rangle$, we first calculate the inner product of $|\chi_3\rangle$ with itself and then analyze the terms. We have

$$\begin{aligned}
1 &= \langle \chi_3 | \chi_3 \rangle \\
&= \frac{1}{4}\big(\langle 0| \langle \chi_+| + \langle 1| \langle \chi_-|\big) \cdot \big(|0\rangle |\chi_+\rangle + |1\rangle |\chi_-\rangle\big) \\
&= \frac{1}{4}\big(\langle 0|0\rangle\langle \chi_+|\chi_+\rangle + \langle 0|1\rangle\langle \chi_+|\chi_-\rangle + \langle 1|0\rangle\langle \chi_-|\chi_+\rangle + \langle 1|1\rangle\langle \chi_-|\chi_-\rangle\big).
\end{aligned}$$

The second and third terms are zero as $\langle 0|1\rangle = \langle 1|0\rangle = 0$. The surviving terms are

$$1 = \langle \chi_3 | \chi_3 \rangle = \frac{1}{4}\big(\langle 0|0\rangle\langle \chi_+|\chi_+\rangle + \langle 1|1\rangle\langle \chi_-|\chi_-\rangle\big).$$

The first term is the probability that the outcome of the measurement is $|0\rangle$, and the second term is the probability of measuring $|1\rangle$. If we substitute that back in the expression for $|\chi_+\rangle$, we get the probability $p_{|0\rangle}$ as

$$\begin{aligned}
p_{|0\rangle} &= \frac{1}{2}\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big)^\dagger \ \frac{1}{2}\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big) \\
&= \frac{1}{2}\big(\langle\psi,\phi| + \langle\phi,\psi|\big) \ \frac{1}{2}\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big) \\
&= \frac{1}{4}\underbrace{\langle\psi,\phi|\psi,\phi\rangle}_{=1} + \frac{1}{4}\langle\psi,\phi|\phi,\psi\rangle + \frac{1}{4}\langle\phi,\psi|\psi,\phi\rangle + \frac{1}{4}\underbrace{\langle\phi,\psi|\phi,\psi\rangle}_{=1}.
\end{aligned}$$

The inner product of a normalized state with itself is 1, which means that the first and fourth terms each become $1/4$, and the expression simplifies to

$$p_{|0\rangle} = \frac{1}{2} + \frac{1}{4}\langle\psi,\phi|\phi,\psi\rangle + \frac{1}{4}\langle\phi,\psi|\psi,\phi\rangle. \tag{7.1}$$

Now recall how to compute the inner product of two compound tensors from Equation (1.5) as

$$\begin{aligned}
|\psi_1\rangle &= |\phi_1\rangle \otimes |\chi_1\rangle, \\
|\psi_2\rangle &= |\phi_2\rangle \otimes |\chi_2\rangle, \\
\Rightarrow \langle\psi_1|\psi_2\rangle &= \langle\phi_1|\phi_2\rangle\langle\chi_1|\chi_2\rangle.
\end{aligned}$$

This means we can rewrite Equation (7.1) (changing the order of the inner products; they are just complex numbers) as

$$\begin{aligned}
p_{|0\rangle} &= \frac{1}{2} + \frac{1}{4}\langle\psi,\phi|\phi,\psi\rangle + \frac{1}{4}\langle\phi,\psi|\psi,\phi\rangle \\
&= \frac{1}{2} + \frac{1}{4}\langle\psi|\phi\rangle\langle\phi|\psi\rangle + \frac{1}{4}\langle\phi|\psi\rangle\langle\psi|\phi\rangle \\
&= \frac{1}{2} + \frac{1}{4}\langle\psi|\phi\rangle\langle\phi|\psi\rangle + \frac{1}{4}\langle\psi|\phi\rangle\langle\phi|\psi\rangle
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2} + \frac{1}{2}\langle\psi|\phi\rangle\langle\phi|\psi\rangle \\
&= \frac{1}{2} + \frac{1}{2}\langle\phi|\psi\rangle^*\langle\phi|\psi\rangle \\
&= \frac{1}{2} + \frac{1}{2}|\langle\psi|\phi\rangle|^2.
\end{aligned}
$$

The scalar product of the two states is the key to measuring similarity. The probability for basis state $|0\rangle$ will be close to $1/2$ if the dot product of $|\psi\rangle$ and $|\phi\rangle$ is close to 0, which means that these two states were orthogonal and maximally different. The probability will be close to 1 if the dot product is close to 1, meaning the states were almost identical.

Find the code
In file src/swap_test.py

In code, this looks quite simple. In each experiment, we construct the circuit shown in Figure 7.1:

```
def run_experiment(a1: np.complexfloating, a2: np.complexfloating,
                   target: float) -> None:
  psi = state.bitstring(0) * state.qubit(a1) * state.qubit(a2)
  psi = ops.Hadamard()(psi, 0)
  psi = ops.ControlledU(0, 1, ops.Swap(1, 2))(psi)
  psi = ops.Hadamard()(psi, 0)
```

Then we measure and calculate the probability that the ancilla qubit is in state $|0\rangle$. That is all there is to it. The variable `p0` will be the probability that qubit 0 will be found in the $|0\rangle$ state. What is left to do now is to compare this probability with a target and check that the results are valid. We allow for a 5% error margin (0.05).

```
  p0, _ = ops.Measure(psi, 0)
  if abs(p0 - target) > 0.05:
    raise AssertionError(
        'Probability {:.2f} off more than 5 pct from target {:.2f}'
        .format(p0, target))
  print('Similarity of a1: {:.2f}, a2: {:.2f} ==>  \%: {:.2f}'
        .format(a1, a2, 100.0 * p0))
```

Lastly, we run experiments and verify that the results match our expectations:

```
def main(argv):
  print('Swap test. 0.5 means different, 1.0 means similar')
  run_experiment(1.0, 0.0, 0.5)
  run_experiment(0.0, 1.0, 0.5)
  run_experiment(1.0, 1.0, 1.0)
  run_experiment(0.0, 0.0, 1.0)
  run_experiment(0.1, 0.9, 0.65)
```

```
  [...]
>>
Swap test to compare state. 0.5 means different, 1.0 means similar
Similarity of a1: 1.00, a2: 0.00 ==>  %: 50.00
Similarity of a1: 0.00, a2: 1.00 ==>  %: 50.00
Similarity of a1: 1.00, a2: 1.00 ==>  %: 100.00
Similarity of a1: 0.00, a2: 0.00 ==>  %: 100.00
Similarity of a1: 0.10, a2: 0.90 ==>  %: 63.71
[...]
```

7.2 Swap Test for Multi-qubit States

We have learned how to use the swap test to compute the overlap of two single-qubit states, but how could we make this work for multi-qubit states? The answer is surprisingly simple. We just have to compose multiple swap tests, one for each pair of qubits, all connected to the same ancillary, as shown in Figure 7.2.

We show a simple code snippet for two-qubit states. The code constructs two random, entangled two-qubit states `psi_a` and `psi_b` and makes a final state by tensoring them together with an ancillary qubit initialized as $|0\rangle$. This is followed by two controlled swap gates, similar to what is shown in Figure 7.2. Finally, we measure and ensure correct results:

PY **Find the code**
In file src/swap_test.py

```
def run_experiment_double(a0: np.complexfloating,
                          a1: np.complexfloating,
                          b0: np.complexfloating,
                          b1: np.complexfloating,
                          target: float) -> None:
  psi_a = state.qubit(a0) * state.qubit(a1)
```

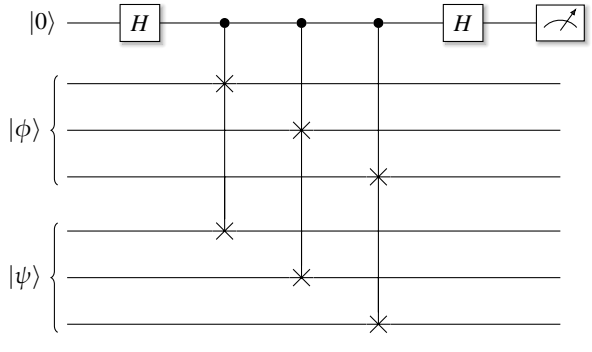


Figure 7.2 The swap tests for two three-qubit states $|\phi\rangle$ and $|\psi\rangle$.

```
psi_a = ops.Cnot(0, 1)(psi_a)
psi_b = state.qubit(b0) * state.qubit(b1)
psi_b = ops.Cnot(0, 1)(psi_b)

psi = state.bitstring(0) * psi_a * psi_b

psi = ops.Hadamard()(psi, 0)
psi = ops.ControlledU(0, 1, ops.Swap(1, 3))(psi)
psi = ops.ControlledU(0, 2, ops.Swap(2, 4))(psi)
psi = ops.Hadamard()(psi, 0)

# Measure once.
p0, _ = ops.Measure(psi, 0)
if abs(p0 - target) > 0.05:
  raise AssertionError(
      'Probability {:.2f} off more than 5% from target {:.2f}'
      .format(p0, target))
```

To run the experiments, we drive this implementation with a simple loop:

```
probs = [0.5, 0.5, 0.5, 0.52, 0.55, 0.59, 0.65, 0.72, 0.80, 0.90]
for i in range(10):
  run_experiment_double(1.0, 0.0, 0.0 + i * 0.1, 1.0 - i * 0.1,
                        probs[i])
```

7.3 Hadamard Test

In Section 7.1, we discussed the swap test to measure the similarity between two unknown states $|\psi\rangle$ and $|\phi\rangle$ without having to measure the states directly. This section presents another test of this nature, the *Hadamard Test*.

The swap and Hadamard tests can be visualized using an analogy with real-valued vectors. The numbers show differently, but the principle is the same. Think about how we compute the inner product of the sum $(\vec{a}+\vec{b})$ of two *normalized*, real-valued vectors $\vec{a}$ and $\vec{b}$ (they have to be normalized, else the math doesn't work out):

$$\begin{aligned}(\vec{a}+\vec{b})^T(\vec{a}+\vec{b}) &= \sum_i (a_i+b_i)^2 \\ &= \underbrace{\sum_i a_i^2}_{=1} + \underbrace{\sum_i b_i^2}_{=1} + 2\sum_i a_i b_i \\ &= 2 + 2\vec{a}^T\vec{b}. \end{aligned} \tag{7.2}$$

For the three extreme cases where $\vec{a}$ and $\vec{b}$ point in the same direction, are orthogonal, or are antiparallel, Equation (7.2) yields:

$$\text{parallel:}\quad \vec{a} = \begin{pmatrix}1\\0\end{pmatrix} \quad \text{and} \quad \vec{b} = \begin{pmatrix}1\\0\end{pmatrix} \quad \text{then} \quad \vec{a}^T\vec{b} = 1.$$

$$\text{orthogonal:} \quad \vec{a} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \text{and} \quad \vec{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{then} \quad \vec{a}^T \vec{b} = 0.$$

$$\text{anit-parallel:} \quad \vec{a} = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \text{and} \quad \vec{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{then} \quad \vec{a}^T \vec{b} = -1.$$

Now let us apply this principle to the Hadamard test. Recall how the swap test used two quantum registers to hold the states $|\psi\rangle$ and $|\phi\rangle$. The Hadamard test is different. It uses only one quantum register that holds the superposition of the two states $|a\rangle$ and $|b\rangle$ for which we want to determine the overlap. To simplify, let us focus on single-qubit states. As a precondition, we need to prepare the initial state $|\psi\rangle$ as

$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|b\rangle\big). \tag{7.3}$$

How can we generate such a state? First, let's see how the partial expressions look as state vectors:

$$|0\rangle|a\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ 0 \\ 0 \end{pmatrix} \quad \text{and} \quad |1\rangle|b\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ b_0 \\ b_1 \end{pmatrix}.$$

As a vector, state $|\psi\rangle$ in Equation (7.3) would be

$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|b\rangle\big) = \frac{1}{\sqrt{2}}\begin{pmatrix} a_0 & a_1 & b_0 & b_1 \end{pmatrix}^T.$$

We define the operators A and B to produce the states $|a\rangle$ and $|b\rangle$ when applied to the state $|0\rangle$. Note how we arrange the matrix elements for A and B to produce the desired output vectors:

$$A|0\rangle = A\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_0 & a_2 \\ a_1 & a_3 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = |a\rangle,$$

$$B|0\rangle = B\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} b_0 & b_2 \\ b_1 & b_3 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = |b\rangle.$$

To construct the circuit, we use the top qubit as an ancilla and an initial Hadamard gate to produce an equal superposition of $|0\rangle$ and $|1\rangle$ with which we control the gates A and B on the bottom qubit, as shown in the circuit in Figure 7.3. In the literature, the two controlled operators A and B are often referred to as a single combined operator U.

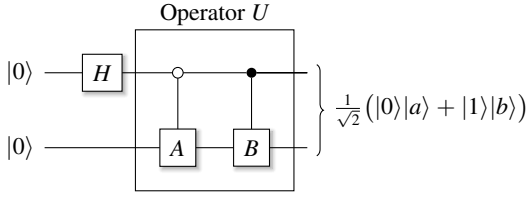


Figure 7.3 The base circuit for the Hadamard similarity test.

PY

Find the code
In file src/hadamard_test.py

Let's verify this in code! We create random unitary operators A and B and apply them to state $|0\rangle$ to extract the relevant state components a_0, a_1, b_0, and b_1:

```
def make_rand_operator():
  U = ops.Operator(unitary_group.rvs(2))
  psi = U(state.bitstring(0))
  return (U, psi[0], psi[1])

def hadamard_test():
  A, a0, a1 = make_rand_operator()
  B, b0, b1 = make_rand_operator()
```

With these parameters, we can construct the state in two different ways. First, we compute it explicitly, following Equation (7.3):

```
  psi = (1 / cmath.sqrt(2) *
         (state.bitstring(0) * state.State([a0, a1]) +
          state.bitstring(1) * state.State([b0, b1])))
```

To compare, we construct the state with a circuit and confirm that the result matches the closed form above:

```
  qc = circuit.qc('Hadamard test - initial state construction.')
  qc.reg(2, 0)
  qc.h(0)
  qc.applyc(A, [0], 1)  # Controlled-by-0
  qc.applyc(B, 0, 1)    # Controlled-by-1

  # The two states should be identical!
  assert np.allclose(qc.psi, psi), 'Incorrect result'
```

Now let's add another Hadamard gate to the top ancilla qubit, as shown in Figure 7.4. This changes the state to

$$\begin{aligned}\frac{1}{\sqrt{2}}H\big(|0\rangle|a\rangle + |1\rangle|b\rangle\big) &= \frac{1}{\sqrt{2}}\big(H|0\rangle|a\rangle + H|1\rangle|b\rangle\big)\\ &= \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|a\rangle + |0\rangle|b\rangle - |1\rangle|b\rangle\big)\\ &= \frac{1}{2}|0\rangle\big(|a\rangle + |b\rangle\big) + \frac{1}{2}|1\rangle\big(|a\rangle - |b\rangle\big).\end{aligned}$$

We calculate the probability $p_{|0\rangle}$ of measuring state $|0\rangle$ in the same way as above for the swap test:

$$p_{|0\rangle} = \frac{1}{2}\big(\langle a| + \langle b|\big)\frac{1}{2}\big(|a\rangle + |b\rangle\big)$$

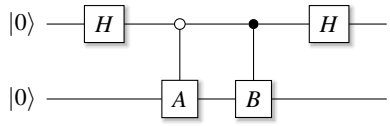


Figure 7.4 The Hadamard similarity test circuit for the real part.

$$= \frac{1}{4}\big(\underbrace{\langle a|a\rangle}_{=1} + \langle a|b\rangle + \langle b|a\rangle + \underbrace{\langle b|b\rangle}_{=1}\big)$$
$$= \frac{1}{4}\big(2 + \langle a|b\rangle + \langle a|b\rangle^*\big). \tag{7.4}$$

The two inner products are complex numbers. For a given complex number z, $z + z^* = a + ib + a - ib = 2a$. We use this in Equation (7.4) and get the probability of measuring $|0\rangle$ on the top qubit as

$$p_{|0\rangle} = \frac{1}{2} + \frac{1}{2}\mathrm{Re}\big(\langle a|b\rangle\big), \quad \text{and also}$$
$$2p_{|0\rangle} - 1 = \mathrm{Re}\big(\langle a|b\rangle\big)$$

We can quickly verify this in code as well:

```
[...]
qc.h(0)
dot = np.dot(np.array([a0, a1]).conj(), np.array([b0, b1]))
p0 = qc.psi.prob(0, 0) + qc.psi.prob(0, 1)
if not np.allclose(2 * p0 - 1, dot.real, atol = 1e-6):
  raise AssertionError('Incorrect inner product estimation')
```

Can we also obtain an estimate for the imaginary part of the inner product? Yes, we can. For this, we start with a slightly modified initial state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle - i|1\rangle|b\rangle\big).$$

The construction is similar to the one above, but we have to apply a factor of $-i$ to the $|1\rangle$ part of the state by adding an $S^\dagger$ gate right after the initial Hadamard gate, as shown in Figure 7.5. The final Hadamard gate changes the state to

$$\frac{1}{\sqrt{2}}H\big(|0\rangle|a\rangle - i|1\rangle|b\rangle\big) = \frac{1}{\sqrt{2}}\big(H|0\rangle|a\rangle - Hi|1\rangle|b\rangle\big)$$
$$= \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|a\rangle - i|0\rangle|b\rangle + i|1\rangle|b\rangle\big)$$
$$= \frac{1}{2}|0\rangle\big(|a\rangle - i\,|b\rangle\big) + \frac{1}{2}|1\rangle\big(|a\rangle + i\,|b\rangle\big).$$

The probability $p_{|0\rangle}$ of measuring state $|0\rangle$ for the ancilla is then

$$p_{|0\rangle} = \frac{1}{2}\big(\langle a| + i\,\langle b|\big)\frac{1}{2}\big(|a\rangle - i\,|b\rangle\big)$$

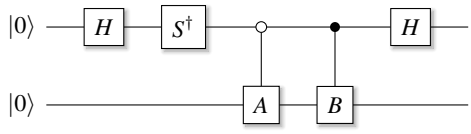


Figure 7.5 The Hadamard similarity test circuit for measuring the imaginary part.

$$= \frac{1}{4}\big(\underbrace{\langle a|a\rangle}_{=1} - i\langle a|b\rangle + i\langle b|a\rangle + \underbrace{\langle b|b\rangle}_{=1}\big)$$
$$= \frac{1}{4}\big(2 - i\langle a|b\rangle + i\langle a|b\rangle^*\big). \tag{7.5}$$

The inner products are complex numbers. For any complex number $z = a + ib$, the conjugate is $z^* = a - ib$ and these relations hold:

$$\begin{aligned} -iz &= -i(a+ib) = -ia + b, \\ iz^* &= i(a-ib) = ia + b, \\ \Rightarrow -iz + iz^* &= -ia + b + ia + b \\ &= 2b \\ &= 2\,\mathrm{Im}(z). \end{aligned}$$

Substituting this into Equation (7.5), we obtain the final result as

$$p_{|0\rangle} = \frac{1}{2} + \frac{1}{2}\,\mathrm{Im}\big(\langle a|b\rangle\big), \quad \text{and also}$$
$$2p_{|0\rangle} - 1 = \mathrm{Im}\big(\langle a|b\rangle\big).$$

This is also what we use in the code:

```
psi = (1 / cmath.sqrt(2) *
       (state.bitstring(0) * state.State([a0, a1]) -
        1.0j * state.bitstring(1) * state.State([b0, b1])))
qc = circuit.qc('Hadamard test - initial state construction.')
qc.reg(2, 0)
qc.h(0)
qc.sdag(0)            # <- this gate is new.
qc.applyc(A, [0], 1)  # Controlled-by-0
qc.applyc(B, 0, 1)    # Controlled-by-1
# The two states should be identical!
assert np.allclose(qc.psi, psi), 'Incorrect result'

# Now let us apply a final Hadamard to ancilla.
qc.h(0)

# And compute the dot product and p0.
dot = np.dot(np.array([a0, a1]).conj(), np.array([b0, b1]))
p0 = qc.psi.prob(0, 0) + qc.psi.prob(0, 1)

# Compare and verify results.
```

```
if not np.allclose(2 * p0 - 1, dot.imag, atol = 1e-6):
  raise AssertionError('Incorrect inner product estimation')
```

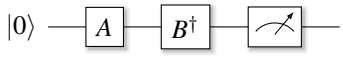
7.4 Inversion Test

So far, we have learned about the swap test, which utilized a register for each input $|a\rangle$ and $|b\rangle$ combined with an ancilla. We also learned about the Hadamard test, which uses an ancilla but only one register, assuming that there are operators A and B to construct the states $|a\rangle$ and $|b\rangle$. A third way to calculate the similarity between two states is the *inversion test*, which estimates the scalar product of the states.

The inversion test takes this one step further. It no longer needs an ancilla, just one quantum register, but it needs the ability to construct $B^\dagger$. We again assume operators A and B produce states $|a\rangle$ and $|b\rangle$, with

$$A|0\rangle = A\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}a_0 & a_2\\ a_1 & a_3\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}a_0\\a_1\end{pmatrix} = |a\rangle\,,$$
$$B|0\rangle = B\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}b_0 & b_2\\ b_1 & b_3\text{‘}\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}b_0\\b_1\end{pmatrix} = |b\rangle\,,$$

and construct this simple circuit:



The expectation value of the projective measurement $M = |0\rangle\langle 0|$ is given by

$$\begin{aligned}\big(\langle 0|A^\dagger B|0\rangle\big)\big(\langle 0|B^\dagger A|0\rangle\big) &= \langle 0|A^\dagger B|0\rangle\langle 0|B^\dagger A|0\rangle\\ &= |\langle 0|B^\dagger A|0\rangle|^2\\ &= |\underbrace{\langle 0|B^\dagger}_{=\langle b|}\underbrace{A|0\rangle}_{=|a\rangle}|^2\\ &= |\langle b|a\rangle|^2\\ &= |\langle a|b\rangle|^2.\end{aligned}$$

Note that the norm of the inner product is symmetric.[1] In code, we reuse the mechanism introduced in Section 7.3 on the Hadamard test to construct random unitaries A and B with function `make_rand_operator()`. The inversion test itself is easy to implement:

```
def inversion_test():
  A, a0, a1 = make_rand_operator()
  B, b0, b1 = make_rand_operator()
  Bdag = B.adjoint()
```

[1] Found in http://quantumcomputing.stackexchange.com/q/26135.

```
# Compute the dot product <a|b>:
dot = np.dot(np.array([a0, a1]).conj(), np.array([b0, b1]))

qc = circuit.qc('Hadamard test - initial state construction.')
qc.reg(1, 0)
qc.apply1(A, 0)
qc.apply1(Bdag, 0)

p0, _ = qc.measure_bit(0, 0)
assert np.allclose(dot.conj() * dot, p0), 'Error'
```

8 Black-Box Algorithms

In this chapter, we discuss several algorithms that fall into the class of the so-called oracle algorithms, where a large black-box unitary operator performs a critical task. As we discuss the algorithms, it is initially not clear *how* the oracles are implemented. However, what they intend to achieve can be described, and this will be sufficient to demonstrate a quantum advantage.

You get the impression that there is some "trick" to construct the oracle, a magical quantum way of doing this, which allows the oracle to answer specific algorithmic questions. This can be confusing for novices. We will learn that to construct the oracle, we need to consider all possible input states and build the oracle in a way that gives the correct answers for all inputs. This means that, in order to construct the oracle, we need to know the solution. However, a third party querying the oracle does not. This will become clearer in the description of the algorithms in the following sections.

The oracle can be a circuit or a permutation matrix. What makes the oracle a quantum oracle is that we can feed it states in superposition. This leads to *quantum parallelism*, where all the answers are computed in parallel. Unfortunately, the state will collapse during measurement, and only one result can be obtained. The challenge for quantum algorithms is hence to amplify the probabilities of the states representing solutions such that they can be reliably measured.[1]

A handful of oracle algorithms exist in the literature. We will visit $2\frac{1}{2}$ of them. Although the Deutsch algorithm (Deutsch, 1985) historically came earlier, the Bernstein–Vazirani algorithm (Bernstein, Vazirani, 1997) seems easier to understand. We discuss it first. Then we discuss Deutsch's algorithm and its extension to more than two input qubits. We add another $1/2$ algorithm by showing how to formulate the previously discussed Bernstein–Vazirani algorithm in oracle form using the general oracle constructor developed in these chapters.

These algorithms were the first to demonstrate a quantum advantage. Their *query complexity*[2] is lower than that for their equivalent classical algorithms. For example, for n bits, a single query is sufficient to find the answer in the Bernstein–Vazirani algorithm, whereas n queries are required in the classical case. Let us dive right in!

[1] Drawing an analogy to classical wave interference, you may also see the term *quantum interference* being used.

[2] *Query complexity* refers to the number of queries needed to solve a computational problem where an input or internal state can only be accessed through queries.

8.1 Bernstein–Vazirani Algorithm

Assume we have an input string b consisting of n bits. Further, assume that there is another secret bit string s of the same length with the property that the scalar product of the input and output bits modulo 2 equals 1. In other words, if the input bits are b_i and the secret string has bits s_i, then this product should hold:

$$b \cdot s = b_0 s_0 \oplus b_1 s_1 \oplus \cdots \oplus b_{n-1} s_{n-1} = 1. \tag{8.1}$$

The goal is to find the secret string s. For example, assume an input string $b = [1,1,1,0,0]$ and an example string $s = [1,0,1,0,0]$. The result of the product would be

$$\begin{aligned} b \cdot s &= 1 \cdot 1 \oplus 0 \cdot 1 \oplus 1 \cdot 1 \oplus 0 \cdot 0 \oplus 0 \cdot 0 \\ &= \underbrace{1 \oplus 0}_{=1} \oplus \underbrace{1 \oplus 0}_{=1} \oplus 0 \\ &= 1 \oplus 1 \oplus 0 \\ &= 0. \end{aligned}$$

We would have to try n times on a classical computer to find the secret string. Each experiment would have an input string of all 0s, except for a single 1. Each iteration for which Equation (8.1) holds identifies a single 1-bit in s at position t, one for each trial $t \in [0, n-1]$. For example, with the secret string from above, we would start with an input string of $b = [1,0,0,0,0]$. With this input, Equation (8.1) becomes

$$\begin{aligned} b \cdot s &= 1 \cdot 1 \oplus 0 \cdot 0 \oplus 1 \cdot 0 \oplus 0 \cdot 0 \oplus 0 \cdot 0 \\ &= 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \\ &= 1, \end{aligned}$$

as it should, as there was a 1 at the first position in the secret string.

In the quantum formulation, we will construct a smart circuit and perform a single query to an oracle. After running the circuit, the output qubits will be in states $|0\rangle$ and $|1\rangle$, corresponding to the bits of the secret string. In the example in Figure 8.1, the secret string is `1010`.

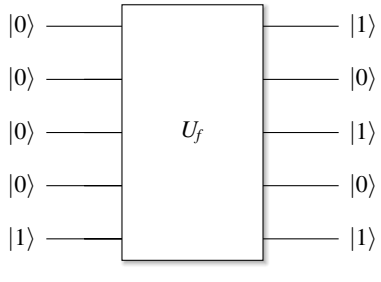


Figure 8.1 The oracle U_f for the secret bit string `1010`. The bottom qubit is an ancilla, initialized to $|1\rangle$.

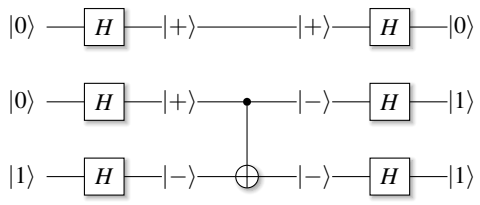


Figure 8.2 A controlled Not gate from a state $|+\rangle$ to a state $|-\rangle$ changes the controller's state to $|-\rangle$.

To see how this works, we need to understand the mechanics of basis changes. Recall how the $|0\rangle$ and $|1\rangle$ states are put in superposition with Hadamard gates:

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

As a first step, we create an input state of length n, initialized with all $|0\rangle$, with an additional ancilla qubit in state $|1\rangle$. We apply a Hadamard gate to all qubits, resulting in an equal superposition of $|+\rangle$ for the $|0\rangle$ input qubits and $|-\rangle$ for the $|1\rangle$ ancilla.

If we apply a controlled Not from a controlling qubit in the $|+\rangle$ state to a qubit in the $|-\rangle$ state, the effect is that the controlling qubit flips into the $|-\rangle$ state! In closed form, we can write this as $CNOT\,|+\rangle\,|-\rangle = |-\rangle\,|-\rangle$. This is the crucial trick because applying another Hadamard gate to each qubit at the end of the circuit will rotate the bases from $|+\rangle$ back to $|0\rangle$ and from $|-\rangle$ back to $|1\rangle$. In other words, qubits that were in state $|-\rangle$, which are the qubits corresponding to 1s in the secret string, will now be in the resulting state $|1\rangle$.

We can visualize this effect with the circuit in Figure 8.2. In this figure, we abuse the circuit notation a little and mark the states of the individual qubits on the horizontal lines.

Let us write this in code. First, we create the secret string as a tuple of length `nbits` of 0s and 1s:

PY

Find the code
In file `src/bernstein.py`

```
def make_c(nbits: int) -> Tuple[int]:
  constant_c = [0] * nbits
  for idx in range(nbits-1):
    constant_c[idx] = int(np.random.random() < 0.5)
  return tuple(constant_c)
```

Next, we construct the oracle, which in this case will be a simple circuit. As described in the introduction to this chapter, an important aspect of all oracle algorithms is that to construct the oracle, we have to know the solution. However, once the oracle is constructed, a third party without knowledge of the solution only needs a single query to the quantum oracle to find the solution.

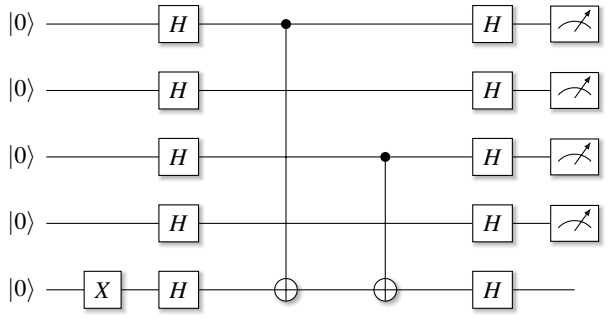


Figure 8.3 The circuit for the Bernstein–Vazirani algorithm with secret string `1010`.

This is the difference from classical computing. In classical computing, we would also have to know the secret string and encode a function that matches an input against this secret string. However, to find the entire secret string (which users of the function cannot see), one has to call this function multiple times. The difference lies in the *query complexity*.

The construction is simple: We apply a controlled Not for each qubit corresponding to a 1 in the secret string. For example, for the secret string `1010`, we build the circuit in Figure 8.3. We construct the corresponding circuit as one big unitary matrix operator U. This limits the maximal number of qubits we can still simulate but is sufficient to explore the algorithm.

```
def make_u(nbits: int, constant_c: Tuple[int]) -> ops.Operator:
  op = ops.Identity(nbits)
  for idx in range(nbits-1):
    if constant_c[idx]:
      op = ops.Identity(idx) * ops.Cnot(idx, nbits-1) @ op
  assert op.is_unitary(), 'Constructed non-unitary operator.'
  return op
```

For experimentation, we perform the following steps. First, we create a secret string of length `nbits-1` and construct the corresponding large unitary with function `make_u`. Then we build a state consisting of `nbits-1` states initialized as $|0\rangle$ and tensor it with an ancilla qubit initialized as $|1\rangle$. After this, we sandwich the big unitary between Hadamard gates and measure and compare the results as a final step:

```
def run_experiment(nbits: int) -> None:
  c = make_c(nbits-1)
  u = make_u(nbits, c)
  psi = state.zeros(nbits-1) * state.ones(1)

  psi = ops.Hadamard(nbits)(psi)
  psi = u(psi)
  psi = ops.Hadamard(nbits)(psi)
  check_result(nbits, c, psi)
```

To verify the results, we find all states with probability $p > 0.1$. There should only be a single state with a higher probability, and that state should represent the secret string. In the code below, we iterate over all basis states and only print the states with high enough probability.

```
def check_result(nbits: int, c: Tuple[int], psi: state.State) -> None
  print(f'Expect:', c)
  for bits in helper.bitprod(nbits):
    if psi.prob(*bits) > 0.1:
      print(f'Found : {bits[:-1]}, with prob: {psi.prob(*bits):.1f}')
      assert bits[:-1] == c, 'Invalid result'
```

Running this program should produce output like the following, showing the secret bit strings and the resulting probabilities (which should be very close to 1):

```
Expect: (1, 0, 1, 0, 0)
Found : (1, 0, 1, 0, 0), with prob: 1.0
```

8.2 Deutsch's Algorithm

Deutsch's algorithm is another, somewhat contrived, algorithm with no apparent practical use (Deutsch, 1985). However, it was one of the first to showcase the potential power of quantum computers, and therefore, it is always one of the first algorithms to be discussed in textbooks. Never fight the trend; let us discuss it right away.

8.2.1 Problem: Distinguish Two Types of Functions

Assume we have a function f that accepts a single bit as input and produces a single bit as output, mapping an input of 0 or 1 to an output of 0 or 1:

$$f\colon \{0,1\} \to \{0,1\}.$$

The four possible cases for this function fall into two categories, which we call *constant* or *balanced*:

$$\begin{aligned} f(0) = 0,\ \ f(1) = 0 \quad &\Rightarrow \text{constant},\\ f(0) = 0,\ \ f(1) = 1 \quad &\Rightarrow \text{balanced},\\ f(0) = 1,\ \ f(1) = 0 \quad &\Rightarrow \text{balanced},\\ f(0) = 1,\ \ f(1) = 1 \quad &\Rightarrow \text{constant}. \end{aligned}$$

This function essentially performs a test for bit parity, which is why it is sometimes called a parity detector. Deutsch's algorithm answers the following question: *Given one of these four functions f, which type of function is it: balanced or constant?*

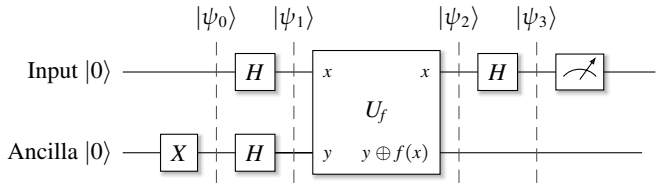


Figure 8.4 The circuit representation of Deutsch's algorithm.

To answer this question with a classical computer, you must evaluate the function for all possible inputs. We have to feed both a 0 and a 1 to the function and evaluate the results to determine the type of the function. In the quantum model, we assume that we have an oracle that, given an input qubit $|x\rangle$ and an ancilla $|y\rangle$, changes the state to Equation (8.2) (again, with $\oplus$ as addition modulo 2).

$$|x,y\rangle \rightarrow |x,y \oplus f(x)\rangle. \tag{8.2}$$

The input $|x\rangle$ remains unmodified and $|y\rangle$ is being XOR'ed with $f(|x\rangle)$. This is a formulation that we will see in other oracle algorithms as well – there is always an ancilla $|y\rangle$, and the result of the evaluated function is XOR'ed with that ancilla. Recall that quantum operators must be reversible; this is one way to achieve this.

Assuming that we have an oracle U_f representing and applying the unknown function $f(x)$, the Deutsch algorithm can be drawn as the circuit shown in Figure 8.4. It is a convention to start every circuit with all qubits in state $|0\rangle$. The algorithm requires the ancilla qubit to be in state $|1\rangle$, which can be easily achieved by applying an X gate to the lower qubit.

Let us go through the math in detail. Initially, after the X gate on qubit 1, the state is

$$|\psi_0\rangle = |01\rangle.$$

After the first Hadamard gates, the state is in superposition:

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |+\rangle \otimes |-\rangle\,.$$

We still don't know how to construct U_f, but we know from Equation (8.2) that applying U_f to the second qubit (let's not be confused by the use of $\oplus$ and $\otimes$) yields the state

$$|\psi_2\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}}.$$

If $f(x) = 0$, then $|\psi_2\rangle = |\psi_1\rangle$:

$$\begin{aligned} |\psi_2\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0 \oplus 0\rangle - |1 \oplus 0\rangle}{\sqrt{2}} \\ &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |+\rangle \otimes |-\rangle\,. \end{aligned}$$

But if $f(x) = 1$, then (note the minus sign at the end)

$$|\psi_2\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0 \oplus 1\rangle - |1 \oplus 1\rangle}{\sqrt{2}}$$
$$= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|1\rangle - |0\rangle}{\sqrt{2}} = |+\rangle \otimes -|-\rangle\,.$$

We can combine the two results into a single expression:

$$|\psi_2\rangle = (-1)^{f(x)} \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} \right) \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Now we multiply the constant factor $(-1)^{f(x)}$ into the first term, with x values of 0 and 1 corresponding to the basis states $|0\rangle$ and $|1\rangle$:

$$|\psi_2\rangle = \frac{(-1)^{f(x)}|0\rangle + (-1)^{f(x)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

We substitute the corresponding value for x as

$$|\psi_2\rangle = \frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{8.3}$$

Finally, applying the final Hadamard to the top qubit takes the state from the Hadamard basis back to the computational basis. To see how this works, let us quickly remind ourselves that the Hadamard operator is its own inverse:

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

If we look at Equation (8.3) and take $f(0) = f(1) = 0$, we get

$$|\psi_2\rangle = \frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$
$$= \frac{(-1)^0|0\rangle + (-1)^0|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$
$$= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Applying the final Hadamard gate to $|\psi_2\rangle$ yields the state

$$|\psi_3\rangle = H\frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |0\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

For $f(0) = f(1) = 1$ we get the same expression, but with a minus sign in front of the first qubit:

$$|\psi_3\rangle = -|0\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Back to $|\psi_2\rangle$, for a balanced function, $f(0) = 0$ and $f(1) = 1$, we get

$$\begin{aligned}|\psi_2\rangle &= \frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} \\ &= \frac{(-1)^0|0\rangle + (-1)^1|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} \\ &= \frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.\end{aligned}$$

Applying the final Hadamard gate to $|\psi_2\rangle$ now produces the state

$$|\psi_3\rangle = H\frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |1\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Similarly, for $f(0) = 1$ and $f(1) = 0$, we get a similar expression, just with a minus sign in front:[3]

$$|\psi_3\rangle = -|1\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

You can see that for a constant function f, we always have a $|0\rangle$ in the first qubit, and in the balanced case, the first qubit will always be in the state $|1\rangle$. This means that after a single run of the circuit, we can determine the type of f simply by measuring the first qubit (it helps that $|0\rangle$ and $|1\rangle$ are orthogonal).

The superposition allows the computation of the results for both basis states $|0\rangle$ and $|1\rangle$ simultaneously. As mentioned earlier, this is an example of quantum parallelism. The XOR'ing to the ancilla qubit allows the math to add up in a smart way such that a result can be obtained with high probability. The result does not tell us which specific function it is out of the four possible cases, but it does tell us which of the two classes it belongs to. In the classical case, two queries are required to determine the type of function. Because the algorithm can exploit superposition to compute the results in parallel, a single query is sufficient.

The mere saving of a single query may not look that impressive, but we will soon learn about the Deutsch–Jozska algorithm in Section 8.3. This algorithm extends to functions of the form $f\colon \{0,1\}^N \to \{0,1\}$, with $N = 2^n$ for n qubits. Classically, this algorithm requires $2^{n-1} + 1$ queries,[4] but in the quantum case, still only a single query is required. This represents an exponential speed-up. We can see that the algorithm has a true query complexity advantage over its classical equivalent.

8.2.2 Construct U_f

The math in Section 8.2.1 may seem quite abstract, but things become clear when considering how to construct U_f. To reiterate, for a combined state of two qubits, the four basis states are

$$|00\rangle = \begin{pmatrix}1 & 0 & 0 & 0\end{pmatrix}^T,$$

[3] You may want to verify this yourself.

[4] If the results were all the same after checking half of all possible inputs, the next query will reveal whether the function was constant or balanced.

$$|01\rangle = \begin{pmatrix}0 & 1 & 0 & 0\end{pmatrix}^T,$$
$$|10\rangle = \begin{pmatrix}0 & 0 & 1 & 0\end{pmatrix}^T,$$
$$|11\rangle = \begin{pmatrix}0 & 0 & 0 & 1\end{pmatrix}^T.$$

We want to construct an operator that takes any linear combination of these input states such that

$$|x,y\rangle \rightarrow |x,y \oplus f(x)\rangle.$$

Constant Functions

The function f only modifies the second qubit as a function of the first. For the case where $f(0) = f(1) = 0$, the truth table is shown in the left half of Table 8.1. The columns x and y represent the input qubits. y is the ancilla and always 1, but we still need to consider it to build a full permutation matrix. $f(x)$ produces a constant 0 in this first case.

The next column shows the result of XOR'ing the function's return value with y, which is $y \oplus f(x)$. The last column finally shows the resulting new state, which leaves the first qubit (x) unmodified and changes the second qubit (y) to the result of the previous XOR. Similarly, for the case $f(0) = f(1) = 1$, the truth table is in the right-hand half of Table 8.1.

We can express these cases with a 4×4 permutation matrix, where rows and columns are marked with the four basis states. We use the combination of x and y as a row index and the new state as column index. A permutation matrix is reversible, which is what we need. In the case of $f(0) = f(1) = 0$, the old and new states are identical, and the resulting $U_{0,0}$ matrix is simply the identity matrix I. The matrix $U_{1,1}$ for the case of $f(0) = f(1) = 1$ is more interesting:

$$U_{0,0} = \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \quad \text{and} \quad U_{1,1} = \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}.$$

Table 8.1. Truth table for constant functions.

		$f(0) = 0, f(1) = 0$			$f(0) = 1, f(1) = 1$		
x	y	$f(x) = 0$	$y \oplus f(x)$	new	$f(x) = 1$	$y \oplus f(x)$	new
0	0	0	0	0, 0	1	1	0, 1
0	1	0	1	0, 1	1	0	0, 0
1	0	0	0	1, 0	1	1	1, 1
1	1	0	1	1, 1	1	1	1, 0

Table 8.2. Truth table for balanced functions.

		$f(0)=0, f(1)=1$			$f(0)=1, f(1)=0$		
x	y	$f(x)$	$y \oplus f(x)$	new	$f(x)$	$y \oplus f(x)$	new
0	0	0	0	0, 0	1	1	0, 1
0	1	0	1	0, 1	1	0	0, 0
1	0	1	1	1, 1	0	0	1, 0
1	1	1	0	1, 0	0	1	1, 1

Balanced Functions

The construction for the two balanced functions follows the same pattern as above, with the truth tables shown in Table 8.2. The table translates to operators $U_{0,1}$ and $U_{1,0}$:

$$U_{0,1} = \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \quad \text{and} \quad U_{1,0} = \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}.$$

8.2.3 General Oracle Operator

We can see that the operator depends only on the function f. A combination of basis states is taken to another combination of basis states through a permutation matrix (which has a single 1 per row and column), and the process is solely controlled by the function and an XOR operation. So far, we have only considered one input qubit and one ancilla qubit, but this can be easily generalized and extended to any number of input qubits (as we find below in Section 8.3). Since the oracle can be used for other algorithms, we add this constructor function to the list of operator constructors.

Find the code
In file src/lib/ops.py

```
def OracleUf(nbits: int, f: Callable[[List[int]], int]):
  dim = 2**nbits
  u = np.zeros(dim**2).reshape(dim, dim)
  for row in range(dim):
    bits = helper.val2bits(row, nbits)
    fx = f(bits[0:-1])      # f(x) without the y.
    xor = bits[-1] ^ fx     # xor with ancilla (the last qubit)

    new_bits = bits[0:-1]
    new_bits.append(int(xor))
```

```
    # Construct new column.
    new_col = helper.bits2val(new_bits)
    u[row][new_col] = 1.0

  op = Operator(u)
  assert op.is_unitary(), 'Constructed non-unitary operator.'
  return op
```

8.2.4 Experiments

For experimentation, we construct the circuit and measure the first qubit. If it collapses to $|0\rangle$, f is a constant function according to the above calculations. If it collapses to $|1\rangle$, f is a balanced function.

First, we define a function `make_f` that returns a function closure object,[5] according to one of the four possible function flavors. We can call the returned function object as $f(0)$ or $f(1)$ where the integer parameter indexes into one of the subarrays in `flavors`, returning a single 0 or 1:

Find the code
In file src/deutsch.py

```
def make_f(flavor: int) -> Callable[[int], int]:
  flavors = [[0, 0], [0, 1], [1, 0], [1, 1]]
  def f(bit: int) -> int:
    return flavors[flavor][bit]
  return f
```

The full experiment first constructs this function object, followed by the oracle. Hadamard gates are applied to each qubit in an initial state $|0\rangle \otimes |1\rangle$, followed by the oracle operator and a final Hadamard gate on the top qubit.

```
def run_experiment(flavor: int) -> None:
  f = make_f(flavor)
  u = make_uf(f)
  h = ops.Hadamard()

  psi = h(state.zeros(1)) * h(state.ones(1))
  psi = u(psi)
  psi = h(psi)
  p0, _ = ops.Measure(psi, 0, tostate=0, collapse=False)

  print(f'f(0) = {f(0)}, f(1) = {f(1)} -> ')
  if math.isclose(p0, 0.0):
    print('balanced')
```

[5] For readers not familiar with closures, in this context, it means that the returned function object `f` still has access to the local array `flavors` even though `f` escapes the scope of the surrounding function `make_f`.

```
    assert flavor in [1, 2], 'Invalid result, expected balanced.'
  else:
    print('constant')
    assert flavor in [0, 3], 'Invalid result, expected constant.'
```

Finally, we verify that we have the right answers for all four functions. To make this more clear, we specify the inputs as binary numbers. The output should look like the one printed below. The fact that we did not hit an assert means that the code produces a valid result:

```
def main(argv):
  run_experiment(0b00)
  run_experiment(0b01)
  run_experiment(0b10)
  run_experiment(0b11)
>>
f(0) = 0 f(1) = 0  constant
f(0) = 0 f(1) = 1  balanced
f(0) = 1 f(1) = 0  balanced
f(0) = 1 f(1) = 1  constant
```

8.2.5 Bernstein–Vazirani in Oracle Form

As promised, we present the Bernstein–Vazirani algorithm in oracle form. Much of the implementation remains the same, but instead of explicitly constructing a circuit with controlled Not gates to represent the secret number, we write an oracle function and call the `OracleUf` constructor from above. This also demonstrates how a multi-qubit input can be used to build the oracle.

First, we construct the function to compute the dot product between the state and the secret string:

```
def make_oracle_f(c: Tuple[bool]) -> ops.Operator:
  const_c = c
  def f(bit_string: Tuple[int]) -> int:
    val = 0
    for idx in range(len(bit_string)):
      val += const_c[idx] * bit_string[idx]
    return val % 2
  return f
```

Then we repeat the original algorithm, but this time using the oracle:

```
def run_oracle_experiment(nbits: int) -> None:
  c = make_c(nbits-1)
  f = make_oracle_f(c)
  u = ops.OracleUf(nbits, f)
```

```
psi = state.zeros(nbits-1) * state.ones(1)
psi = ops.Hadamard(nbits)(psi)
psi = u(psi)
psi = ops.Hadamard(nbits)(psi)
check_result(nbits, c, psi)
```

Lastly, we run the code and check that we implemented all this correctly:

```
Expected: (0, 1, 0, 1, 0, 0)
Found   : (0, 1, 0, 1, 0, 0), with prob: 1.0
```

8.3 Deutsch–Jozsa Algorithm

The Deutsch–Jozsa algorithm is a generalization of the Deutsch algorithm to multiple input qubits (Deutsch and Jozsa, 1992). The function to evaluate is still balanced or constant, but over an expanded domain with multiple input bits:

$$f\colon \{0,1\}^n \to \{0,1\}.$$

The mathematical treatment of this case parallels the two-qubit Deutsch algorithm. The key result is that we measure the state of n qubits. If we find qubits in the state $|0\rangle$ only, the function is constant. If we find anything else, the function is balanced. We only need a single query in the quantum case, whereas classically, this would again require $2^{n-1}+1$ queries. The circuit, shown in Figure 8.5, looks similar to the two-qubit case, except that multiple qubits are used for both input and output. The single ancilla qubit at the bottom will still be the key to the answer.

Implementation

The mathematical derivation of this result is sizable but does not provide much additional value. Let's focus on the code, which is quite compact with our U_f operator. First, we create the many-qubit function as either a constant function (all 0s or all 1s with equal probability) or a balanced function (the same number of 0s and 1s, randomly distributed over the length of the input bit string). We create an array of bits and fill it with 0s and 1s accordingly. Finally, we return a function object that

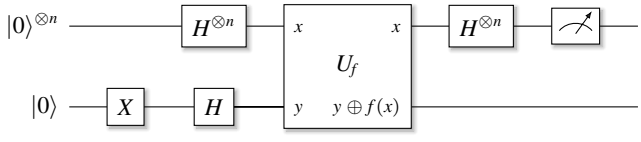


Figure 8.5 The Deutsch–Jozsa algorithm as a circuit diagram.

returns one of the values from this prepopulated array, thus representing one of the two function types.

Find the code
In file src/deutsch_jozsa.py

```
def make_f(dim: int = 1,
           flavor: int = exp_constant) -> Callable[[List[int]], int]:
  power2 = 2**dim
  bits = np.zeros(power2, dtype=np.uint8)
  if flavor == exp_constant:
    bits[:] = int(np.random.random() < 0.5)
  else:
    bits[np.random.choice(power2, size=power2//2, replace=False)] = 1

  def f(bit_string:List[int]) -> int:
    idx = helper.bits2val(bit_string)
    return bits[idx]
  return f
```

To carry out an experiment, we construct the circuit shown in Figure 8.5 and measure. If the measurement finds that only the state $|00\ldots0\rangle$ has a nonzero probability amplitude, then we have a constant function. If we measure anything else, then we have a balanced function.

```
def run_experiment(nbits: int, flavor: int):
  f = make_f(nbits-1, flavor)
  u = ops.OracleUf(nbits, f)

  psi = (ops.Hadamard(nbits-1)(state.zeros(nbits-1)) *
         ops.Hadamard()(state.ones(1)))
  psi = u(psi)
  psi = (ops.Hadamard(nbits-1) * ops.Identity(1))(psi)

  # Measure all of |0>. If allclose to 1.0, f() is constant.
  for idx in range(nbits - 1):
    p0, _ = ops.Measure(psi, idx, tostate=0, collapse=False)
    if not math.isclose(p0, 1.0, abs_tol=1e-5):
      return exp_balanced
  return exp_constant
```

Finally, we run the experiments on numbers of qubits ranging from 2 to 7 to run reasonably fast and ensure that the results match the expectations. Note that we still generate operators and oracles as full matrices, which limits the number of qubits we can handle:

```
def main(argv):
  for qubits in range(2, 8):
    result = run_experiment(qubits, exp_constant)
```

```
    assert result == exp_constant, f'Want: {exp_constant}'
    result = run_experiment(qubits, exp_balanced)
    assert result == exp_balanced, f'Want: {exp_balanced}'
>>
Found: constant (2 qubits) (expected: constant)
Found: balanced (2 qubits) (expected: balanced)
Found: constant (3 qubits) (expected: constant)
Found: balanced (3 qubits) (expected: balanced)
[...]
Found: constant (7 qubits) (expected: constant)
Found: balanced (7 qubits) (expected: balanced)
```

Other algorithms of this nature are *Simon's algorithm* and *Simon's generalized algorithm* (Simon, 1994). We will not discuss them here, but implementations can be found in the open-source repository in files `simon.py` and `simon_general.py`.

9 State Preparation

The question of how to encode and store data in a quantum machine is a complex one. Algorithms often assume a specific initial state. For example, the optimization algorithms that we will discover in Chapter 13, or the quantum machine learning algorithms in Chapter 14, may all require a specific initial state for the algorithm to work correctly. However, this state may be difficult to prepare. The preparation overhead could potentially reduce the quantum advantage of an algorithm, or the preparation may require gate types that are not available on a given machine. These challenges are collectively referred to as the problem of *state preparation*, which is the topic of this chapter.

We first describe typical ways to encode data in a quantum state. Then we explore a few ways to prepare a quantum state. There are trivial techniques for states that are initialized with just the basis states $|0\rangle$ and $|1\rangle$ and slightly more sophisticated techniques for arbitrary two- and three-qubit states. To prepare an arbitrary multi-qubit state, we implement Möttönen's algorithm. However, this elegant algorithm requires gates that may not be physically available on a given machine. The Solovay–Kitaev algorithm addresses this problem. It is a seminal result in quantum computing as it shows how to approximate any gate with sequences of tolerable lengths of standard universal gate sets.

These last two algorithms are some of the *most advanced* algorithms discussed in this book. They have the potential to completely frustrate novices and even advanced readers.[1] Depending on your skill level, you may want to revisit these algorithms at a later time.

9.1 Data Encoding

This section discusses a few typical ways of representing data in a quantum state. We will discuss the trivial but qubit-intensive basis encoding, the potentially more complex but qubit-efficient amplitude encoding, and methods that use Hamiltonian operators to encode data.

9.1.1 Basis Encoding

Integers can be quantum-encoded with a scheme called *basis encoding*. In this scheme, the binary bits of an integer are directly encoded as states $|0\rangle$ or $|1\rangle$, with a binary bit 0

[1] Including this author.

mapping to the basis state $|0\rangle$ and a binary bit 1 mapping to the basis state $|1\rangle$. We have already seen this in Section 2.4.3, where we also discussed bit ordering.

Similarly, floating point numbers can be encoded as binary fractions, with qubits representing fractional powers of 2. The achievable accuracy depends on the number of qubits that represent a value. With n qubits, the precision of the approximation is $1/2^n$. For example, if we reserve 5 qubits to represent a binary fraction (1 sign bit, 4 bits to represent the value), the vector $\begin{pmatrix}0.1 & -0.7 & 1.0\end{pmatrix}$ can be approximated with a maximum error of $1/16 = 0.0625$ as

$$\begin{aligned}
0.082 \simeq |00001\rangle &= + \left(0\frac{1}{2^1} + 0\frac{1}{2^2} + 0\frac{1}{2^3} + 1\frac{1}{2^4}\right) = +0.0625 \quad (\Delta = 0.0195),\\
-0.7 \simeq |11011\rangle &= - \left(1\frac{1}{2^1} + 0\frac{1}{2^2} + 1\frac{1}{2^3} + 1\frac{1}{2^4}\right) = -0.6875 \quad (\Delta = 0.0125),\\
1.0 \simeq |01111\rangle &= + \left(1\frac{1}{2^1} + 1\frac{1}{2^2} + 1\frac{1}{2^3} + 1\frac{1}{2^4}\right) = +0.9375 \quad (\Delta = 0.0625).
\end{aligned}$$

To represent vectors of values, we create a state that concatenates the states using basis encoding. In the example, the encoded state would be the concatenation of the three 5-qubit basis states:

$$|\psi\rangle = |00001\,11011\,01111\rangle\,.$$

With this type of encoding, the vector does not necessarily need to be normalized. The largest fractional value that can be approximated asymptotically is 1.0, which means that individual vector elements must be strictly scaled with $|x_i| < 1.0$.

We will use this type of encoding in many places in this book. The main advantage of this scheme is that state preparation is exceedingly trivial (at least in simulation). Starting from a state of all $|0\rangle$, one only has to apply an X gate to the qubits that have a corresponding 1 bit in the binary representation. The disadvantages are that a potentially large number of qubits is required to represent data and that, similarly to classical computing, the floating-point values are only approximated.

9.1.2 Amplitude Encoding

To represent an arbitrary vector $\vec{v}$ in *amplitude encoding*, we encode the individual vector elements $\vec{v}_i$ as probability amplitudes of basis states. Since the probabilities of a state vector must sum up to 1, the vector $\vec{v}$ may require normalization. Given that state vectors have lengths that are powers of two, $\vec{v}$ is also padded to lengths that are powers of two. Putting it all together, a vector $\vec{v}$ is encoded as a state $|\psi\rangle$ with the computational basis vectors $|e_i\rangle$ as

$$\vec{v}_{\text{normed}} = \frac{\vec{v}}{|\vec{v}|} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} \quad \rightarrow \quad |\psi\rangle = v_0\,|e_0\rangle + v_1\,|e_1\rangle + \cdots + v_{n-1}\,|e_{n-1}\rangle\,.$$

As an example, we take the vector $\begin{pmatrix}0.1 & -0.7 & 1.0\end{pmatrix}$ from above, extend it to four elements, and normalize it to $\begin{pmatrix}0.08 & -0.57 & 0.82 & 0.0\end{pmatrix}$:

```
def amplitude_encoding():
  psi = state.State([0.1, -0.7, 1.0, 0.0])
  psi.normalize()
  psi.dump()  # only prints non-0 amplitudes.
>>
|00> (|0>):  ampl: +0.08+0.00j prob: 0.01 Phase:   0.0
|01> (|1>):  ampl: -0.57+0.00j prob: 0.33 Phase: 180.0
|10> (|2>):  ampl: +0.82+0.00j prob: 0.67 Phase:   0.0
```

The advantage of this encoding is that only a small number of qubits are required to encode a vector (only two qubits for this example, compared to 15 for basis encoding). In addition, floating point values are stored with infinite precision in the system.[2] However, the main disadvantage is that the physical preparation of the state can be very difficult. State preparation is trivial in our code, as we simply assign amplitudes. In an actual physical circuit, potentially very complex sequences of gates must be used to prepare a state properly.

For the special case of a state with *equal* nonzero probabilities for a small number of basis states, we discuss an algorithm based on amplitude amplification in Section 10.2.1.

9.1.3 Encoding with Rotations

To encode a *real* value $|\alpha| \leq 1.0$ with a single qubit, we ignore a potential local phase and write the state in the form

$$|\psi\rangle = \sqrt{1-\alpha^2}\,|0\rangle + \alpha\,|1\rangle\,.$$

This is valid because adding up the norms of the probability amplitudes sums up to 1. Because we compute the square root $\sqrt{1-\alpha^2}$ (and ignore complex values here), α must be smaller than 1, which may require normalization of α. From Equation (2.9) we know that we can prepare this state with an R_y operator performing a rotation about the y-axis:

$$R_y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}.$$

$$R_y(\theta)\,|0\rangle = R_y(\theta)\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}\cos\frac{\theta}{2}\\ \sin\frac{\theta}{2}\end{pmatrix}$$

$$= \begin{pmatrix}\sqrt{1-\sin^2\frac{\theta}{2}}\\ \sin\frac{\theta}{2}\end{pmatrix}.$$

[2] On a computer, analog values are still discretized using floating-point formats.

With $\theta = 2\ \arcsin(\alpha)$, we get the result we were looking for as

$$R_y(\theta)\,|0\rangle = \sqrt{1-\alpha^2}\,|0\rangle + \alpha\,|1\rangle\,.$$

We can easily verify this in code. For a given Python value `alpha`, we compute the result in two different ways and make sure the results match:

```
factor_0 = np.sqrt(1 - 1.0 / (alpha * alpha))
factor_1 = 1.0 / alpha
theta = 2.0 * np.arcsin(1.0 / alpha)

psi = state.zeros(1)
psi = ops.RotationY(theta)(psi)

assert np.isclose(factor_0, psi[0], atol=0.001), 'Incorrect factor 0'
assert np.isclose(factor_1, psi[1], atol=0.001), 'Incorrect factor 1'
```

In the literature, this technique is sometimes called *time-evolution encoding*, for reasons that will become clear in Section 9.1.4.

9.1.4 Hamiltonian Encoding

In Section 2.7.4 we write the R_y operator as $R_y(\theta) = e^{-i\frac{\theta}{2}Y}$ with an exponentiated Pauli Y matrix. We can generalize this to other Hermitian matrices. In quantum mechanics, a *Hamiltonian* $\hat{\mathcal{H}}$ is an operator that corresponds to the energy of a system. It is a Hermitian operator with $\hat{\mathcal{H}} = \hat{\mathcal{H}}^\dagger$. Unitaries and Hamiltonians are connected with the time-dependent Schrödinger equation, which states that

$$i\hbar\frac{\partial}{\partial t}\,|\Psi\rangle = \hat{\mathcal{H}}\,|\Psi\rangle\,.$$

It has a solution $|\Psi(t)\rangle = U(t)\,|\Psi(0)\rangle$, with $U(t)$ being a time-dependent unitary operator. The key to an encoding technique is that for a time-dependent Hamiltonian, a unitary operator can be defined as

$$U(t) = e^{-it\hat{\mathcal{H}}},$$

with the Hermitian $\hat{\mathcal{H}}$ and setting $\hbar = 1$. Let us prove this identity. The proof uses the important concept of an *operator function*, which we will also use later in the book.

Proof Since U is a normal matrix, the spectral theorem applies (as discussed in Section 4.1). We can write U as

$$U = \sum_i \lambda_i |x_i\rangle\langle x_i|,$$

where λ_i are the eigenvalues of U and $|x_i\rangle$ its eigenvectors. Since U is unitary with $UU^\dagger = I$, it follows that $|\lambda_i|^2 = 1$ and hence $\lambda_i = e^{i\theta_i}$ for some angle θ_i. Now we can define $\hat{\mathcal{H}}$ as

$$\hat{\mathcal{H}} = \sum_i \theta_i |x_i\rangle\langle x_i|.$$

To apply an operator function $f(\cdot)$ to a normal operator A, we spectrally decompose A and apply $f(\cdot)$ to the eigenvalues of A:

$$f(A) = \sum_i f(\lambda_i)|x_i\rangle\langle x_i|.$$

With f as the exponential function $f(\cdot) = \exp(\cdot)$ and $A = i\hat{\mathcal{H}}$, we find that

$$e^{i\hat{\mathcal{H}}} = \sum_i e^{i\theta_i}|x_i\rangle\langle x_i| = \sum_i \lambda_i |x_i\rangle\langle x_i| = U. \qquad \square$$

Pauli matrices can be used to represent Hamiltonians. This is why the rotational encoding scheme from Section 9.1.3, which uses the R_y gate, is sometimes called *time-evolution encoding*.

We will learn more details in Section 11.2.1 on phase estimation and find a concrete use case in the important HHL algorithm for matrix inversion in Section 14.3. We will encounter a way to construct Hamiltonians inspired by the Ising model of ferromagnetism in Section 13.3, with use cases in Section 13.4 on the max-cut problem and in Section 13.5 on the subset-sum problem.

9.2 State Preparation for Two- and Three-Qubit States

State preparation for a single qubit is trivial, at least in code. We have already seen in Section 2.3 that any location on the surface of a Bloch sphere can be reached with just two rotations, for example, a rotation around the y-axis and another around the z-axis.

Things get considerably more complicated for states of more than just a single qubit. Specialized and optimized preparation mechanisms have been found for two- and three-qubit circuits. For example, for 2-qubit states, Perdomo (2022) and the corresponding video on YouTube (Perdomo, 2 qubits, YT, 2022), discuss the circuit shown in Figure 9.1. The same authors also present a circuit for preparing a 3-qubit state (Perdomo, 3 qubits, YT, 2022). This work presents improvements over the previous work by (Znidaric, 2008) and (Acin, 2000).

For the 2-qubit case, to compute the unitary operators w_1, w_2, and w_3 in Figure 9.1, we follow the instructions on YouTube. The code transforms a given general state into $|0\rangle$. Hence, to prepare a state, you must reverse the circuit.

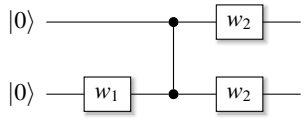


Figure 9.1 Compact circuit to prepare a 2-qubit state.

PY

Find the code
In file src/state_prep.py

```
def run_experiment_2qubit() -> None:
  def norm(x):
    return np.linalg.norm(x)
  def u(x, y):
    return (1 / np.sqrt(norm(x)**2 + norm(y)**2) *
            ops.Operator([[x, y], [-np.conj(y), np.conj(x)]]))

  psi = np.random.random([4])  + 1j * np.random.random(4)
  psi = psi / norm(psi)
  print('Random input:', psi, ' -> |0>')

  a1 = np.array([psi[0], psi[1]])
  a2 = np.array([psi[2], psi[3]])
  a12 = np.inner(a1.conj(), a2)
  if a12 == 0:
    k = norm(a2) / norm(a1)
  else:
    k = -norm(a2) / norm(a1) * a12 / norm(a12)

  w1 = u(psi[3] - k * psi[1], (psi[2] - k * psi[0]).conj()).transpose()
  psi1 = (ops.Identity() * w1) @ psi
  psi1 = ops.ControlledU(0, 1, ops.PauliZ()) @ psi1

  w2 = u(psi1[1].conj(), psi1[3].conj())
  psi2 = (w2 * ops.Identity()) @ psi1

  w3 = u(psi2[0].conj(), (-psi2[1]).conj()).transpose()
  psi3 = (ops.Identity() * w3) @ psi2

  assert np.allclose(psi3[0], 1.0, 1e-6), 'Yikes'
```

9.3 Möttönen's Algorithm

Now that we know how to prepare states with up to three qubits, what should we do for larger, general circuits? An elegant algorithm for preparing general states was given by Möttönen (2004,1). In this chapter, we review and implement these results. For detailed derivations, refer to Möttönen (2004, 2) by the same authors.

This is one of the most complicated algorithms presented in this book. Novice readers may have difficulty with this material. If you are in this group, we recommend not reading this section linearly but returning to it later. At the high level, the algorithm uses controlled rotations to explicitly "set" the amplitudes for all individual basis states. However, since this would require a large number of gates and ancillas, the algorithm improves by demonstrating an elegant way to reduce the required number for both. Let's see how this works.

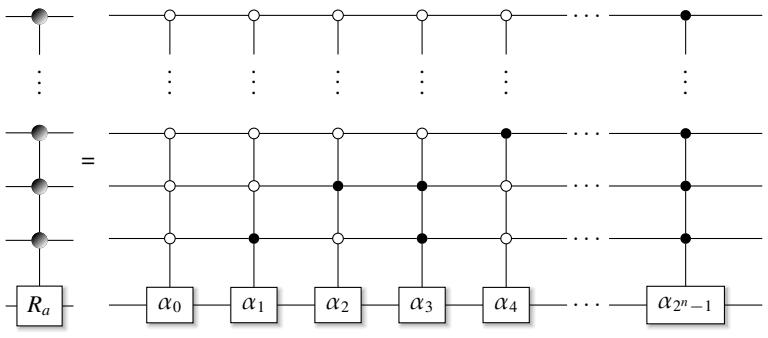


Figure 9.2 The uniformly controlled rotation gate $F_m^k(a,\alpha)$ with k controlling qubits and angles α_i.

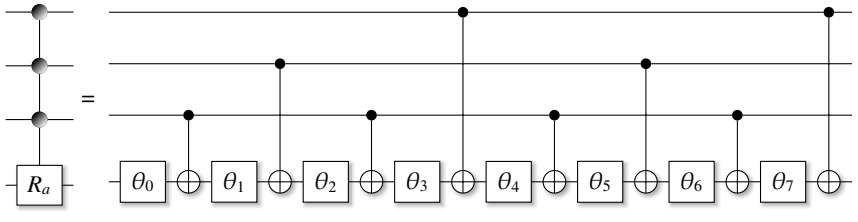


Figure 9.3 A recursive implementation of the uniformly controlled rotation gate, shown for three controlling qubits ($k = 3$).

First, let us introduce the concept of a *uniformly controlled rotation* $F_m^k(a,\vec{\alpha})$, which has k qubits that control rotations by angles $\{\alpha_i\}$ about the axis a on the target qubit m. Figure 9.2 shows the construction, which covers the 2^k binary combinations of control qubits controlling rotations by angles α_i. In circuit diagrams, we indicate these controlled gates with black-white shaded dots at the controlling qubits, as seen on the left side of the figure.

This construction can prepare any desired state but requires many gates, especially with our implementation of multi-controlled gates. However, previous work in Möttönen (2004, 2) showed that this gate F_m^k can be implemented with the recursive construction shown in Figure 9.3 (for the example of $k = 3$). This method represents significant savings in the amount of required gates and ancillary qubits. The challenge is to derive the angles $\{\theta_i\}$ from the angles $\{\alpha_i\}$, which we show next.

The construction is recursive. To add another qubit k, take the construction for $k-1$ qubits, add a controlled Not to the new controlling qubit, and repeat the full sequence twice. For the case of $k = 0$, there is no controlled gate, only the rotation gate. We find the control qubit indices recursively in the following elegant way.[3]

[3] It took me a while to figure this out. The secret to success is to "kill" the last token in the recursive call.

PY **Find the code**
In file src/state_prep_mottonen.py

```
def compute_ctl(idx: int):
  if idx == 0:
    return []
  side = compute_ctl(idx - 1)[:-1]
  return side + [idx - 1] + side + [idx - 1]
```

The angles θ_i in Figure 9.3 can be calculated from the angles α_i with Equation (9.1). To stay close to the reference in Möttönen (2004,1), we use 1-based indexing in the mathematical formulation (this is Equation (3) in the reference):

$$\begin{pmatrix} \theta_1 \\ \vdots \\ \theta_{2^k} \end{pmatrix} = M \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_{2^k} \end{pmatrix}, \quad \text{with } M_{ij} = 2^{-k}(-1)^{b_{j-1} \cdot g_{i-1}}. \tag{9.1}$$

Here, b_j stands for the binary representation of integer j and g_i is the binary reflected Gray code[4] of integer i. We know how to get to a binary representation of an integer. The Gray code uses a combination of XOR and shift operations to ensure that subsequent numbers differ only by a single bit in their binary representation. We compute the Gray code with a classic routine[5] and construct the matrix M as follows:

```
def gray_code(i: int) -> int:
  return i ^ (i >> 1)

def compute_m(k: int):
  n = 2**k
  m = np.zeros([n, n])
  for i in range(n):
    for j in range(n):
      m[i, j] = (-1) ** bin(j & gray_code(i)).count('1') * 2 ** (-k)
  return m
```

With these preliminaries in place, we can now discuss the algorithm for the preparation of the state. The reference paper takes an arbitrary state and constructs the gate sequence required to reduce it to the first computational basis state $|e_1\rangle = |00\ldots0\rangle$. To borrow the nomenclature, it assumes a state in the form

$$|a\rangle = \left(|a_1|e^{i\omega_1}, |a_2|e^{i\omega_2}, \ldots, |a_N|e^{i\omega_N}\right)^T. \tag{9.2}$$

The transformation to $|e_1\rangle$ then happens in two steps:

1. First, a cascade of uniformly controlled z-rotations to equalize the phases ω_i make the vector real up to a global phase.

[4] See http://en.wikipedia.org/wiki/Gray_code.
[5] "Classic" as in "classic car."

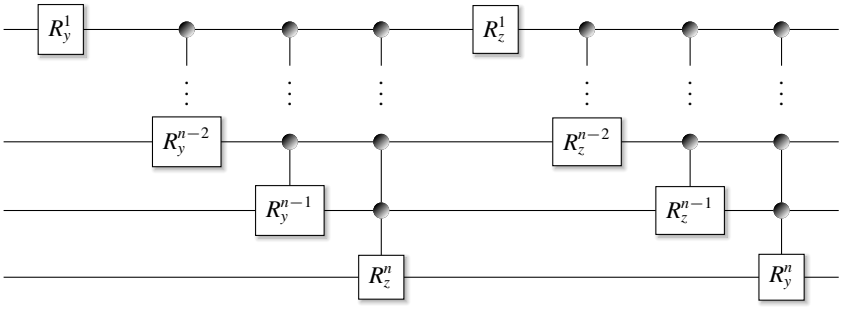


Figure 9.4 Cascades of R_y and R_z rotations to transform an initial state $|00\ldots0\rangle$ into any desired state.

2. Rotate the resulting real vector to $|e_1\rangle$ with a similar cascade of rotations around the y-axis.

However, we want to achieve the opposite. We want to start from a ket $|00\ldots0\rangle$ and transform it into the desired state. Hence, we perform this algorithm in reverse. We start with a cascade of R_y rotations, followed by a cascade of R_z rotations, as shown in Figure 9.4.

Mathematically, cascades (using R_y or R_z rotations) can be written as the product of cascading gates, where we use the definition of $F_m^k(a,\alpha)$:

$$\prod_{j=1}^{n} F_j^{j-1}\left(z,\alpha_{n-j+1}^z\right) \otimes I_{2^n-j}.$$

To implement the cascades, we need to calculate the various angles α_i from the phase angles ω_i in Equation (9.2). These are the phases that we want to eliminate. For the R_z rotations, the angles are (following Equation (5) in the reference):

$$\alpha_{j,k}^z = \sum_{l=1}^{2^{k-1}} \left(\omega_{(2j-1)2^{k-1}+l} - \omega_{(2j-2)2^{k-1}+l}\right)/2^{k-1},$$

with $j = 1,2,\ldots,2^{n-k}$ and $k = 1,2,\ldots,n$.

```
def compute_alpha_z(omega, k: int, j: int):
  # Since the mathematical notation is 1-based but the
  # Python code is 0-based, we have to add a correction
  # term to the code: j becomes j+1:
  m = 2 ** (k - 1)
  ind1 = [(2 * (j + 1) - 1) * m + l for l in range(m)]
  ind2 = [(2 * (j + 1) - 2) * m + l for l in range(m)]
  diff = (omega[ind1] - omega[ind2]) / m
  return sum(diff)
```

The expression to calculate the rotation angles around the y-axis is more complicated, with j,k as above (this is Equation (8) in the reference).

$$\alpha_{j,k}^{y} = 2\arcsin\left(\sqrt{\sum_{l=1}^{2^{k-1}} |a_{(2j-1)2^{k-1}+l}|^2} \Big/ \sqrt{\sum_{l=1}^{2^{k}} |a_{(j-1)2^{k}+l}|^2}\right).$$

As mentioned above, we also have to apply the correction term (`j` becomes `j+1`) here. The corresponding code is:

```
def compute_alpha_y(vec, k: int, j: int):
  m = 2 ** (k - 1)
  enumerator = sum(vec[(2 * (j + 1) - 1) * m + l] ** 2 for l in range(m))
  m = 2**k
  divisor = sum(vec[j * m + l] ** 2 for l in range(m))
  if divisor != 0:
    return 2 * np.arcsin(np.sqrt(enumerator / divisor))
  return 0.0
```

With all these building blocks in place, we can now compose the routine to perform a uniformly controlled rotation. The procedure is the same for the R_y and R_z rotations, which allows us to pass the actual rotation gate as a parameter. The following code uses the functions `compute_m` and `compute_ctl` that were introduced above:

```
def controlled_rotation(qc, alpha_k, control, target, rotgate):
  k = len(control)
  thetas = compute_m(k) @ alpha_k
  ctl = compute_ctl(k)
  for i in range(2**k):
    rotgate(target, thetas[i])
    if k > 0:
      qc.cx(control[k - 1 - ctl[i]], target)
```

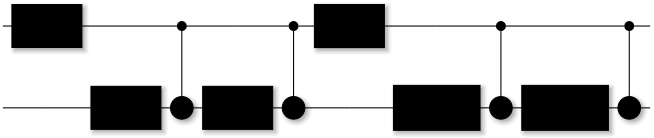
Now we can implement the cascades of rotations as shown in Figure 9.3. Note that the procedure still leaves a global phase in place. We will have to account for it later in our experiments.

```
def prepare_state_mottonen(qc, qb, vector, nbits: int = 3):
  """Construct the Mottonen circuit based on input vector."""

  # Ry gates for the absolute amplitudes.
  avec = abs(vector)
  for k in range(nbits):
    alpha_k = [compute_alpha_y(avec, nbits - k, j) for j in range(2**k)]
    controlled_rotation(qc, alpha_k, qb[:k], qb[k], qc.ry)

  # Rz gates to normalize up to a global phase. This is only
  # needed for complex values.
  omega = np.angle(vector)
```

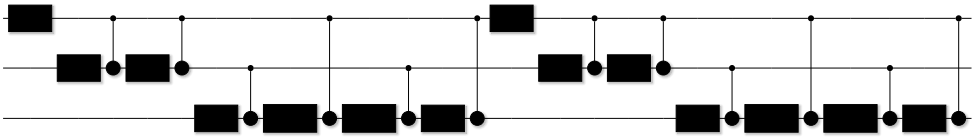
```
    if np.allclose(omega, 0.0):
      return

    for k in range(0, nbits):
      alpha_z = [compute_alpha_z(omega, nbits - k, j) for j in range(2**k)]
      controlled_rotation(qc, alpha_z, qb[:k], qb[k], qc.rz)
```

It is interesting to inspect the circuit structure for states with one, two, or three qubits. For the circuits shown here, all values are taken from random inputs. It is safe to ignore the actual values; we only want to show the structure of the generated circuits. In the circuit diagrams, the gates Y_s and Z_s denote the rotations R_y and R_z by s in radians. For a single-qubit state, we only need a single R_y and R_z gate:

For states with two qubits, the circuit looks like this:



For states of three qubits, the number of gates grows quite considerably. They may be too small to decipher here, but you should be able to recognize the recursive structure:



To convince ourselves that the whole procedure is correct, we run a set of experiments. For each experiment, we consider an arbitrary state vector, apply the algorithm, account for a possible global phase, and check for correctness. We then run this procedure on circuits ranging from a single qubit to up to 10 qubits.

```
def run_experiment(nbits: int = 3):
  """Prepare a random state with nbits qubits."""

  vector = np.random.random([2**nbits]) +
           np.random.random([2**nbits]) * 1j
  vector = vector / np.linalg.norm(vector)
  print(f'  Qubits: {nbits:2d}, vector: {vector[:6]}...')

  qc = circuit.qc()
  qb = qc.reg(nbits)
  prepare_state_mottonen(qc, qb, vector, nbits)
```

```
    # For complex numbers, this algorithm introduces a global phase
    # which we can account for (and ignore) here:
    phase = vector[0] / qc.psi[0]
    if not np.allclose(vector, qc.psi * phase, atol=1e-5):
      raise AssertionError('Invalid State initialization.')

def main(argv):
  print("State Preparation with Moettoenen's Algorithm...")
  for nbits in range(1, 11):
    run_experiment(nbits)
```

9.4 Solovay–Kitaev Theorem and Algorithm

State preparation may require flexible quantum gates, such as specific rotation gates, which may not exist on physical hardware. A given architecture may implement only a smaller universal set of gates, such as the set of Hadamard and T gates. We know that we can synthesize any gate from this universal set of gates.[6] However, we do not want to destroy any quantum advantage. For example, if a quantum algorithm has a theoretical complexity of $\mathcal{O}(\sqrt{N})$ but requires $\mathcal{O}(N^2)$ gates for its physical implementation, the advantage would be nullified.

The Solovay–Kitaev (SK) theorem and corresponding algorithm (Kitaev et al., 2002) are important results in quantum computing, as they address this problem. The theorem shows that not only can any unitary gate be approximated from a finite universal set of gates, but it can also be approximated *quickly*. A version of the theorem that seems appropriate in our context is the following (even though it uses terminology that we have not explained yet, such as $\mathrm{SU}(2)$ or $\langle G \rangle$).

THEOREM: (Solovay–Kitaev theorem) *Let G be a finite set of elements in $SU(2)$ containing its own inverses, such that $\langle G \rangle$ is dense in SU(2). Let $\varepsilon > 0$ be given. Then there is a constant c such that for any U in $SU(2)$ there is a sequence S of gates of length $\mathcal{O}(\log^c(1/\varepsilon))$ such that $||S - U|| < \varepsilon$.*

In English, this theorem states that for a given unitary gate U, a finite sequence of universal gates will approximate U up to any precision ε. The important part: The complexity scales only *polylogarithmically*, as a power of $log(1/\varepsilon)$. This algorithm is seminal and supremely elegant. Since its development in 1995, there have been many improvements and also variations (Kliuchnikov et al., 2015; Ross and Selinger, 2016, 2021).

We will study it following the pedagogical review from Dawson and Nielsen (2006) as a guide. We start with a few important concepts and functions. Then, we outline the high-level structure of the algorithm before diving deeper into the complex parts and implementation. We will omit a small number of mathematical derivations that go well beyond the scope of this book.[7]

[6] Otherwise, the set would not be called universal.
[7] And the comfort level of this author.

9.4.1 Universal Gates

In quantum computing, unlike classical computing, there is no *single universal* gate from which all other gates can be derived. Only *sets of gates* have this property. For single qubits, one of such sets consists of the Hadamard gate H and the T gate. Any point on a Bloch sphere can be reached by a sequence of only these two gates. We prove this by showing that the SK algorithm, based on (minor adjustments of) just these two gates, can approximate any 2×2 unitary matrix up to arbitrary precision (hence the term *dense* in the theorem above).

9.4.2 SU(2)

One of the requirements of the SK algorithm is that the universal gates involved are part of the $\mathrm{SU}(2)$ group, which is the group of all 2×2 unitary matrices with determinant 1. The determinants of the Hadamard and T gates are not equal to 1 (you may want to convince yourself of this). Since their determinants are not 0, we can divide by the determinant and apply this simple transformation to make the gates become members of $\mathrm{SU}(2)$:

$$U' = \sqrt{\frac{1}{\det U}} U.$$

Using this simple adjustment, we compute the set of universal gates H' and T' with this routine:

Find the code
In file src/solovay_kitaev.py

```
def to_su2(U):
  return np.sqrt(1 / np.linalg.det(U)) * U
```

We will not go deeper into $\mathrm{SU}(2)$ and the related mathematics. For our purposes, we should think of $\mathrm{SU}(2)$ in terms of rotations. For a given rotation V, the inverse rotation is $V^\dagger$, with $VV^\dagger = I$. For two rotations U and V, the inverse of UV is $V^\dagger U^\dagger$, with $UVV^\dagger U^\dagger = I$. However, similar to how two perpendicular sides on a Rubik's cube rotate against each other, if we change the order of rotations, then $UVU^\dagger V^\dagger \neq I$. The rotations do not commute; their order matters. This also means that the two rotations can gradually move a state about the Bloch sphere, which is exactly what the SK algorithm does.

9.4.3 Bloch Sphere Angle and Axis

Any 2×2 unitary matrix represents a quantum gate that can move a state about the Bloch sphere. It rotates a state by an angle θ around an axis $\vec{n}$ in a three-dimensional coordinate system. To calculate this angle and axis, let us think of the gate U as being of the form

$$U = \begin{pmatrix} a & b \\ c & d \end{pmatrix}. \tag{9.3}$$

We can also write the operator in the following way, where $\hat{n}$ refers to 3-dimensional orthogonal axes and $\vec{\sigma}$ refers to the Pauli matrices, using the Taylor expansion from Equation (2.7):

$$U = e^{i\theta\hat{n}\frac{1}{2}\vec{\sigma}} = I\cos(\theta/2) + i\hat{n}\cdot\vec{\sigma}\sin(\theta/2).$$

We already know that any unitary matrix can be constructed from a linear combination of Pauli matrices. Applying the Pauli matrices one by one compounds to a single rotation about an axis $\hat{n}$ by an angle θ. Elements on a rotation axis remain unmoved by the rotation. With this insight, we can compute the angle and axis using the following derivations.

$$\begin{aligned}
U &= e^{i\theta\hat{n}\cdot\frac{1}{2}\vec{\sigma}} = e^{i\theta/2\hat{n}\cdot\vec{\sigma}} \\
&= I\cos(\theta/2) + \hat{n}\cdot i\vec{\sigma}\ \sin(\theta/2) \\
&= I\cos(\theta/2) + n_1 i\sigma_1\sin(\theta/2) + n_2 i\sigma_2\sin(\theta/2) + n_3 i\sigma_3\sin(\theta/2) \\
&= \begin{pmatrix} \cos(\theta/2) & 0 \\ 0 & \cos(\theta/2) \end{pmatrix} + \begin{pmatrix} 0 & n_1 i\sin(\theta/2) \\ n_1 i\sin(\theta/2) & 0 \end{pmatrix} \\
&\quad + \begin{pmatrix} 0 & n_2\sin(\theta/2) \\ -n_2\sin(\theta/2) & 0 \end{pmatrix} + \begin{pmatrix} n_3 i\sin(\theta/2) & 0 \\ 0 & -n_3 i\sin(\theta/2) \end{pmatrix} \\
&= \begin{pmatrix} \cos(\theta/2) + n_3 i\sin(\theta/2) & n_2\sin(\theta/2) + n_1 i\sin(\theta/2) \\ -n_2\sin(\theta/2) + n_1 i\sin(\theta/2) & \cos(\theta/2) - n_3 i\sin(\theta/2) \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.
\end{aligned}$$

We compute the relevant parameters with these algebraic transformations:

$$\begin{aligned}
\theta &= 2\arccos\frac{a+d}{2}, \\
n_1 &= \frac{b+c}{2i\sin(\theta/2)}, \\
n_2 &= \frac{b-c}{2\sin(\theta/2)}, \\
n_3 &= \frac{a-d}{2i\sin(\theta/2)},
\end{aligned}$$

and directly translate them into this code:

```
def u_to_bloch(U):
  angle = np.real(np.arccos((U[0, 0] + U[1, 1])/2))
  sin = np.sin(angle)
  if sin < 1e-10:
    axis = [0, 0, 1]
  else:
    nx = (U[0, 1] + U[1, 0]) / (2j * sin)
    ny = (U[0, 1] - U[1, 0]) / (2 * sin)
    nz = (U[0, 0] - U[1, 1]) / (2j * sin)
```

```
    axis = [nx, ny, nz]
  return axis, 2 * angle
```

9.4.4 Similarity Metrics

The *trace distance*[8] tells us how similar two states are. Typically, this concept is applied for states expressed as density matrices, but we may as well adopt it here to measure the similarity between operators. For two density operators ρ and σ, the trace distance is defined as

$$T(\rho,\sigma) = \frac{1}{2}\,\mathrm{tr}\left[\sqrt{(\rho-\sigma)^\dagger(\rho-\sigma)}\right].$$

In code, we use this definition and pass two parameters U and V to the routine `trace_dist`. Notice that we do not use `np.sqrt`, which computes the root of individual elements of the matrix, not the root of the matrix. Instead, we must use the slower but correct `scipy.linalg.sqrtm`:

```
def trace_dist(U, V):
  return np.real(0.5 *
                 np.trace(sp.linalg.sqrtm((U - V).adjoint() @ (U - V))))
```

9.4.5 Pre-computing Gates

The SK algorithm is recursive. At the innermost step, it maps a given unitary operator U against a library of precomputed gate sequences, selecting the gate *closest* to U, as measured by the trace distance.

To precompute gate sequences, we provide a trivial implementation that is slow but has the advantage of being easy to understand. There are only two base gates H' and T', as shown in Section 9.4.2, which we hold in the simple two-element Python list `basegates`. We generate all strings of bits up to a certain length, such as 0 and 1 for length 1, the bit strings 00, 01, 10, 11, for length 2, and so on. We initialize a temporary gate as the identity gate I and iterate through the bits of each bit string, multiplying the temporary gate by one of the two basis gates H' or T', depending on whether a bit in the bit string was set to 0 or 1 respectively. The function then returns the list of all precomputed gates.

```
def create_unitaries(basegates, limit):
  gate_list = []
  for width in range(limit):   # length of bit string
    for bits in helper.bitprod(width):
      U = ops.Identity()
      for bit in bits:
```

[8] See also http://en.wikipedia.org/wiki/Trace_distance.

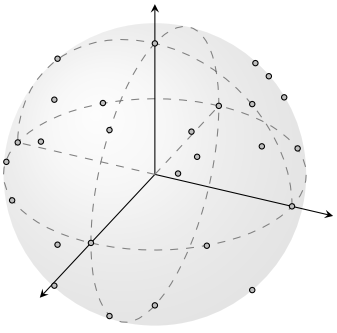


Figure 9.5 Distribution of 256 generated gate sequences applied to state $|0\rangle$. The trivial method used to generate these gates leads to many duplicates.

```
      U = U @ basegates[bit]
    gate_list.append(U)
return gate_list
```

To find the closest gate to a given gate U, we iterate over the list, compute the trace distance of each gate in the list to U, and return the gate with the minimum distance. There are ways to significantly accelerate this search, for example, with KD-trees (Wikipedia, 2021a).

```
def find_closest_u(gate_list, u):
  min_dist, min_u = 1e6, ops.Identity()
  for gate in gate_list:
    tr_dist = trace_dist(gate, u)
    if tr_dist < min_dist:
      min_dist, min_u = tr_dist, gate
  return min_u
```

Note that our method of generating gate sequences results in duplicate gates. For example, when plotting the effects of the generated gates on state $|0\rangle$ in Figure 9.5, we see that the resulting distinct states are quite sparse on the Bloch sphere. Of course, this is easy to optimize.

9.4.6 Algorithm

Now we are ready to discuss the algorithm, which we write in code and explain line by line. The inputs are the unitary operator U, which we seek to approximate, the list of precomputed gates, and the recursion depth n.

```
def sk_algo(U, gates, n):
  if n == 0:
    return find_closest_u(gates, U)
  else:
```

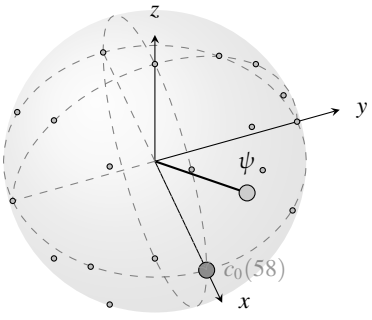


Figure 9.6 A random gate U moves state $|0\rangle$ to state $|\psi\rangle$ (light gray dot). The closest precomputed gate (gate 58) is on the x-axis (dark gray dot).

```
  U_next = sk_algo(U, gates, n-1)
  V, W   = gc_decomp(U @ U_next.adjoint())
  V_next = sk_algo(V, gates, n-1)
  W_next = sk_algo(W, gates, n-1)
  return (V_next @ W_next @ V_next.adjoint() @ W_next.adjoint() @
          U_next)
```

The recursion counts down from an initial value of `n` and stops when it reaches the termination case with `n==0`. At this point, the algorithm looks for the closest precomputed gate. If we specify a maximum recursion depth of 0, this gate will be the result, as shown in Figure 9.6.

```
if n == 0:
  return find_closest_u(gates, U)
```

Starting with this basic approximation, the following steps further improve the approximation by applying sequences of other inaccurate gates. The first recursive step tries to find an approximation `U_next` of U as U_{n-1}. For example, if `n==1`, the recursion with `n-1` reaches the termination clause and returns the closest precomputed gate as `U_next`.

```
  U_next = sk_algo(U, gates, n-1)
```

Assume that U_{n-1} is an approximation of U with error $|U - U_{n-1}| = \varepsilon_{n-1}$. We define $\Delta = UU_{n-1}^{\dagger}$ (note the dagger) and try to find an approximation of Δ with error $\varepsilon_n < \varepsilon_{n-1}$. Then we concatenate the sequence $UU_{n-1}^{\dagger}$ with the previous approximation U_{n-1} to get an approximation U_n with error $|U - U_n| \leq \varepsilon_n$.

To approximate Δ, we decompose it as a *group commutator*, defined as $\Delta = VWV^{\dagger}W^{\dagger}$ for some unitary gates V, W. There are an infinite number of such decompositions. In the following, we apply an accuracy criterion to get a *balanced group commutator*. The underlying mathematics motivating this decomposition is beyond the

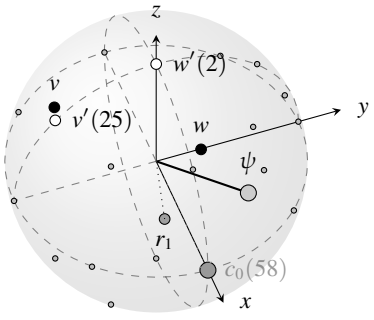


Figure 9.7 The decomposed gates V and W (black dots) and their closest precomputed gates v' (gate 25) and w' (gate 2) as white dots.

scope of this book. We refer to Dawson and Nielsen (2006) and Kitaev et al. (2002) for more details. Here, we accept the result and show how to implement `gc_decomp()`, which we call in the algorithm to get the operators V and W:

```
V, W   = gc_decomp(U @ U_next.adjoint())
```

The next recursive steps are to calculate *improved* approximations for V and W using the same algorithm. Once we have those, the algorithm returns a new and improved approximation:

$$\begin{aligned} U_n &= \Delta U_{n-1} \\ &= U U_{n-1}^{\dagger} U_{n-1} \\ &= V_{n-1} W_{n-1} V_{n-1}^{\dagger} W_{n-1}^{\dagger} U_{n-1}. \end{aligned}$$

```
V_next = sk_algo(V, gates, n-1)
W_next = sk_algo(W, gates, n-1)
return (V_next @ W_next @ V_next.adjoint() @ W_next.adjoint() @
        U_next)
```

For our example, we visualize the gates V and W and their closest precomputed gates in Figure 9.7. The dot marked r_1 is the approximated gate (when applied to $|0\rangle$) after one level of recursion.

9.4.7 Balanced Group Commutator

Now let us explore the definition of the balanced group commutator in more detail. For a unitary operator U, an infinite number of group commutator decompositions exists. We are looking for one for which $VWV^{\dagger}W^{\dagger} = U$, but for which the distance between the identity I and both V and W is less than a specific error bound. The idea is to continuously reduce the error in subsequent recursions. We apply more and more

inaccurate gates to increase the accuracy of the final gate, which is quite miraculous. Mathematically, the propagation of the error goes beyond the scope of this book. We focus mainly on the implementation.

For our balanced group commutator, we consider V as a rotation by angle ϕ about the x-axis of a Bloch sphere and W as a similar rotation about the y-axis. The group commutator $VWV^\dagger W^\dagger$ is a rotation about the Bloch sphere around the axis $\hat{n}$ by an angle θ, satisfying Equation (9.4). This equation admittedly seems to come out of the blue, but in the following paragraphs, we will derive this equation and then solve for ϕ:

$$\sin(\theta/2) = 2\sin^2(\phi/2)\sqrt{1-\sin^4(\phi/2)}. \tag{9.4}$$

Both V and W were defined as rotations about the x-axis and y-axis:

$$\begin{aligned} V &= R_x(\phi), \\ V^\dagger &= R_x(\phi)^\dagger = R_x(-\phi), \\ U &= VWV^\dagger W^\dagger = R_x(\phi)R_y(\phi)R_x(-\phi)R_y(-\phi). \end{aligned}$$

From Equation (2.8) we know we can write rotations as

$$\begin{aligned} R_x(\phi) &= \cos(\phi/2)\, I + i\sin(\phi/2)\, X, \\ R_y(\phi) &= \cos(\phi/2)\, I + i\sin(\phi/2)\, Y. \end{aligned}$$

We can multiply out U and again think of the resulting matrix as being in the form of Equation (9.3) with a, b, c, and d as stand-ins for the four matrix elements. We evaluate the diagonal elements as above with $\cos\left(\theta/2\right) = (a+d)/2$ and arrive at:

$$\cos(\theta/2) = \cos^4(\phi/2) + 2\cos^2(\phi/2)\sin^2(\phi/2) - \sin^4(\phi/2).$$

We factor out $\cos^2(\phi/2) + \sin^2(\phi/2)$:

$$\begin{aligned} \cos(\theta/2) &= \cos^4(\phi/2) + 2\cos^2(\phi/2)\sin^2(\phi/2) - \sin^4(\phi/2) \\ &= \left(\cos^2(\phi/2) + \sin^2(\phi/2)\right)^2 - 2\sin^4(\phi/2) \\ &= 1 - 2\sin^4(\phi/2). \end{aligned}$$

Using the Pythagorean theorem, we get the form we are looking for:

$$\begin{aligned} \sin^2(\theta/2) &= 1 - \cos^2(\theta/2) \\ &= 1 - \left(1 - 2\sin^4(\phi/2)\right)^2 \\ &= 4\sin^4(\phi/2) - 4\sin^8(\phi/2) \\ &= 4\sin^4(\phi/2)\left(1 - \sin^4(\phi/2)\right), \\ \Rightarrow \sin(\theta/2) &= 2\sin^2(\phi/2)\sqrt{1-\sin^4(\phi/2)}. \end{aligned}$$

Now we solve for ϕ. From what we have done so far, we know how to compute θ for an operator. We eliminate the square root in Equation (9.4) by squaring the whole

equation. For ease of notation, we substitute x for the left side as

$$\begin{aligned} x = \left(\frac{\sin(\theta/2)}{2}\right)^2 &= \left(\sin^2(\phi/2)\sqrt{1-\sin^4(\phi/2)}\right)^2 \\ &= \sin^4(\phi/2)\left(1-\sin^4(\phi/2)\right) \\ &= \sin^4(\phi/2) - \sin^8(\phi/2), \\ \Rightarrow 0 &= \sin^4(\phi/2) - \sin^8(\phi/2) - x \\ &= \sin^8(\phi/2) - \sin^4(\phi/2) + x. \end{aligned}$$

This is a quadratic equation that we can solve:

$$\begin{aligned} y^2 - y + x &= 0, \\ \Rightarrow \sin^4(\phi/2) = y &= \frac{1 \pm \sqrt{1-4x}}{2}, \\ \sin(\phi/2) &= \sqrt{\sqrt{y}}, \\ \Rightarrow \phi &= 2\arcsin\left(y^{1/4}\right). \end{aligned} \tag{9.5}$$

Expand y (and remember that $\cos^2(\phi) + \sin^2(\phi) = 1$):

$$\begin{aligned} y &= \frac{1 \pm \sqrt{1-4x}}{2} \\ &= \frac{1 \pm \sqrt{1 - 4\sin^2(\theta/2)/4}}{2} \\ &= \frac{1 \pm \cos(\theta/2)}{2}. \end{aligned}$$

Substituting this into Equation (9.5) leads to the final result for ϕ. We ignore the $+$ case from the quadratic equation, as the goal was to arrive at Equation (9.4):[9]

$$\phi = 2\arcsin\left(\frac{1-\cos(\theta/2)}{2}\right)^{1/4}.$$

Let us write this in code. First, we define the function `gc_decomp`, adding a helper function to diagonalize a unitary matrix. We compute θ and ϕ as described above:

```
def gc_decomp(U):

  def diagonalize(U):
    _, V = np.linalg.eig(U)
    return ops.Operator(V)

  axis, theta = u_to_bloch(U)
  phi = 2.0 * np.arcsin(np.sqrt(
      np.sqrt((0.5 - 0.5 * np.cos(theta) / 2))))
```

[9] We recommend that rigor-sensitive readers please hold their noses here.

After that, we compute the rotation angle and axis on the Bloch sphere as shown above and construct the rotation operators V and W:

```
V = ops.RotationX(phi)
if axis[2] > 0:
  W = ops.RotationY(2 * np.pi - phi)
else:
  W = ops.RotationY(phi)
```

Construction continues as follows. We calculated that U is a rotation by angle θ about some axis $\hat{n}$. We defined V and W as rotations by an angle ϕ around a different compound axis $\hat{p}$. We align the axis $\hat{p}$ to axis $\hat{n}$ with the similarity transformation $U = S(VWV^\dagger W^\dagger)S^\dagger$ for some unitary matrix S, which we compute in the code below as a change of basis matrix. We define $\hat{V} = SVS^\dagger$ and $\hat{W} = SWS^\dagger$ and obtain

$$U = \hat{V}\hat{W}\hat{V}^\dagger\hat{W}^\dagger.$$

In code, this may be a bit easier to read:

```
VWVdWd = diagonalize(V @ W @ V.adjoint() @ W.adjoint())
S = diagonalize(U) @ VWVdWd.adjoint()

V_hat = S @ V @ S.adjoint()
W_hat = S @ W @ S.adjoint()
return V_hat, W_hat
```

In Figure 9.8, we show how the approximation improves in our example as we increase the recursion depth. Interestingly, the results at recursion levels 1 and 2 are almost identical, but the accuracy improves further at deeper levels of recursion.

9.4.8 Evaluation

For a brief evaluation, we define key parameters and run a few experiments. The number of experiments is given by `num_experiments`. The variable `depth` is the

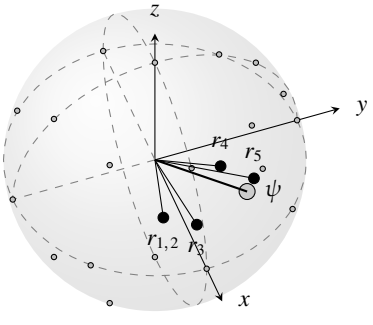


Figure 9.8 Increasing the depth of recursion (r_x) quickly leads to better accuracy.

maximum length of the bit strings we use to precompute gates. For a depth value x, $2^x - 1$ gates are precomputed. The variable `recursion` is the depth of recursion of the SK algorithm. It is instructive to experiment with these values and explore the accuracy and performance you can achieve:

```
def main(argv):
  num_experiments = 10
  depth = 8
  recursion = 4
  print('SK algorithm - depth: {}, recursion: {}, experiments: {}'.
        format(depth, recursion, num_experiments))
```

Next, we compute the $SU(2)$ base gates from the Hadamard and T gates and generate the list of precomputed gates:

```
  base = [to_su2(ops.Hadamard()), to_su2(ops.Tgate())]
  basegates = create_unitaries(base, depth)
  sum_dist = 0.0
```

Finally, we run the experiments. In each experiment, we create a unitary gate U from a randomly chosen combination of rotations. We apply the algorithm and compute distance metrics for the results. We also compare the impact of the original and approximate unitary gates on a state $|0\rangle$ and show how much the results differ, measured in percent. This can give an intuitive measure of the remaining approximation errors.

```
  for i in range(num_experiments):
    U = (ops.RotationX(2.0 * np.pi * random.random()) @
         ops.RotationY(2.0 * np.pi * random.random()) @
         ops.RotationZ(2.0 * np.pi * random.random()))

    U_approx = sk_algo(U, basegates, recursion)
    dist = trace_dist(U, U_approx)
    sum_dist += dist

    phi1 = U(state.zero)
    phi2 = U_approx(state.zero)
    print('[{:2d}]: Trace Dist: {:.4f} State: {:6.4f}%'.
          format(i, dist,
                 100.0 * (1.0 - np.abs(np.dot(phi1, phi2.conj())))))
  print('Gates: {}, Mean Trace Dist:: {:.4f}'.
        format(len(basegates), sum_dist / num_experiments))
```

This should result in output as shown below. With just 255 precomputed gates (including duplicates) and a recursion depth of 4, the approximation error consistently falls below 1%.

```
SK algorithm, depth: 8, recursion: 4, experiments: 10
[ 0]: Trace Dist: 0.0063 State: 0.0048%
[ 1]: Trace Dist: 0.0834 State: 0.3510%
[ 2]: Trace Dist: 0.0550 State: 0.1557%
[...]
[ 8]: Trace Dist: 0.1114 State: 0.6242%
[ 9]: Trace Dist: 0.1149 State: 0.6631%
Gates: 255, Mean Trace Dist:: 0.0698
```

10 Algorithms Using Amplitude Amplification

In this chapter, we explore algorithms associated with quantum amplitude amplification. We introduce Grover's algorithm, a fundamental technique that enables searching over N elements in a domain with complexity of only $\mathcal{O}(\sqrt{N})$. In the following algorithms, we represent the domain by the $N = 2^n$ computational basis states of n qubits and consider one or more of these as special elements, or "solutions," representing the elements we were searching for. Grover's algorithm operates on states in equal superposition. Its extension to unequal superposition is covered in the section on quantum amplitude amplification. Quantum counting determines the total number of solutions in states in equal superposition, whereas quantum amplitude estimation extends this to states in unequal superposition.

After covering the necessary preliminaries, we explore how these methods apply to algorithms such as graph coloring and Boolean satisfiability. The final three techniques presented here, namely quantum mean finding, quantum median finding, and quantum minimum finding, are frequently referenced but may rely on assumptions that may not be physically realizable. We have much ground to cover!

10.1 Grover's Algorithm

Grover's algorithm is one of the fundamental algorithms of quantum computing (Grover, 1996). It allows searching for a special element in a domain of N elements in $\mathcal{O}(\sqrt{N})$ time. We will represent the domain by the $N = 2^n$ basis states of n qubits, so the space complexity is $\mathcal{O}(\log N)$. The special element is also called a "solution." In general, special elements form a set of solutions S. By "searching" we mean that there is a function $f(x)$ and one (or more) special element x' for which

$$f(x) = \begin{cases} 1, & x \in S \quad (\text{or } x = x'), \\ 0, & x \notin S \quad (\text{or } x \neq x'). \end{cases}$$

The classical algorithm to find x' has complexity $\mathcal{O}(N)$ in the worst case since it has to evaluate all possible inputs to f. Being able to do this with complexity $\mathcal{O}(\sqrt{N})$ is, of course, an exciting prospect and one of the main reasons for the interest in this quantum algorithm.

To understand the algorithm, we first describe it at a high level in fairly abstract terms. We need to learn two new concepts: *phase inversion* and *inversion about the*

mean. Once these concepts are explained, we detail several variants of their implementation. Then we assemble all the pieces into the complete Grover's algorithm and run a few experiments.

10.1.1 High-Level Overview

At a high level, the algorithm performs the following steps given a domain encoded with n qubits and a special element $|x'\rangle$:

1. Create an equal superposition state $|++\cdots+\rangle = |+\rangle^{\otimes n}$ by applying Hadamard gates to an initial state $|00\ldots0\rangle$ of n qubits.
2. Construct a *phase inversion* operator U_f around the basis state representing the special input $|x'\rangle$, which is defined mathematically as
$$U_f = I^{\otimes n} - 2|x'\rangle\langle x'|.$$
3. Construct an *inversion about the mean* operator $U_\perp$, defined as
$$U_\perp = 2(|+\rangle\langle+|)^{\otimes n} - I^{\otimes n}.$$
4. Combine $U_\perp$ and U_f into the Grover operator G:
$$G = U_\perp U_f.$$
5. Repeat steps 2 to 4 a total of k times, applying G to the state in each iteration. We derive the iteration count k below. The resulting state will be close to the special state $|x'\rangle$:
$$G^k |+\rangle^{\otimes n} \sim |x'\rangle.$$

This basically explains the whole procedure. Some of you may look at this, shrug mildly, and understand it right away. For the rest of us, the following sections explain this procedure in great detail and in multiple different ways. Grover's algorithm is foundational; we want to make sure we understand it completely.

10.1.2 Phase Inversion

The first new concept we need to learn is *phase inversion*. Assume a given state $|\psi\rangle$ with probability amplitudes c_x and the basis states $|x\rangle$ representing the elements in a domain on N elements, with $N = 2^n$ for n qubits:

$$|\psi\rangle = \sum_x c_x|x\rangle \quad \text{with } x \in \{0,1,\ldots,2^n-1\}.$$

For simplicity, assume equal $c_i = 1/\sqrt{N}$. Figure 10.1 shows a bar graph where the x-axis enumerates the states $|x_i\rangle$, and the y-axis plots the height of the corresponding probability amplitudes c_i. It is safe to ignore the *actual* values; we are just trying to make a point.

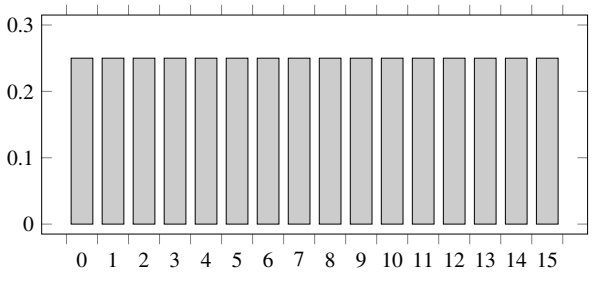


Figure 10.1 Equally distributed probability amplitudes.

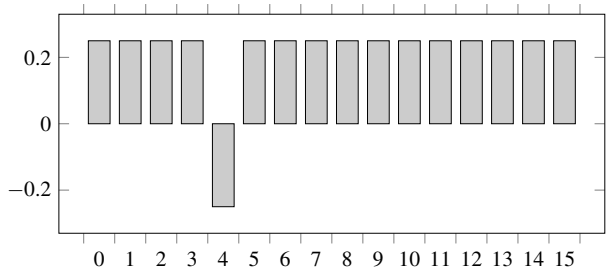


Figure 10.2 Probability amplitudes after phase inversion.

We also assume that one of these input states $|x_i\rangle$ is the special element $|x'\rangle$ mentioned above. Phase inversion converts the original state into a state where the phase for the special element $|x'\rangle$ picks up a factor of $e^{i\pi} = -1$, its phase is being "negated."[1]

$$|\psi\rangle = \sum_x c_x|x\rangle \quad \rightarrow \quad |\psi\rangle = \sum_{x\neq x'} c_x|x\rangle - c_{x'}|x'\rangle.$$

In the graph in Figure 10.2, we negated the phase of state $|4\rangle$, which is our special state $|x'\rangle$. To relate this back to the function $f(x)$ that we are trying to analyze, we use phase inversion to negate the phase for the special elements only, which we can express in closed form as

$$|\psi\rangle = \sum_x c_x|x\rangle \quad \rightarrow_{\text{inv}} \quad |\psi\rangle = \sum_x c_x(-1)^{f(x)}|x\rangle. \tag{10.1}$$

Similarly to the black-box algorithms of Chapter 8, a key aspect of this procedure is that the function f must be known. Otherwise, we would not be able to build the operators and circuits required by this algorithm. This is an important distinction: Although an implementation must know the function, observers who try to construct and measure the function still have to go through N steps in the classical case but only $\sqrt{N}$ in the quantum case. This will become clearer in Sections 10.5 and 10.6, where we provide examples of applications of this algorithm.

[1] Note that in general, the c_i can be complex.

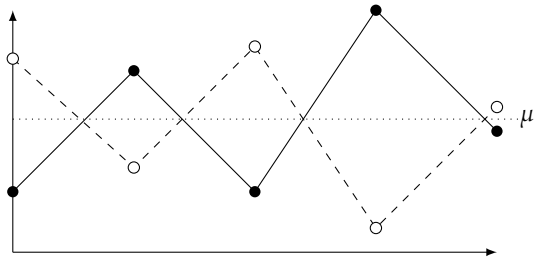


Figure 10.3 An example of random (real) data (black circles) and its inversion about the mean (white circles). The lines connecting the dots have no meaning but help to visualize the mean inversion.

10.1.3 Inversion about the Mean

The second new concept is *inversion about the mean*. In general, the probability amplitudes c_x are complex. However, for simplicity in the following paragraphs, we assume only real probability amplitudes. We calculate the mean μ ("mu") of the probability amplitudes c_x of the original state as

$$\mu = \Big(\sum_x c_x\Big)/N.$$

Inversion about the mean is the process of mirroring each c_x about the mean. To achieve this, we take the distance of each value from the mean, which is $\mu - c_x$, and add it to the mean. For values above the mean, $\mu - c_x$ is negative, and the value is reflected below the mean. Conversely, for values below the mean, $\mu - c_x$ is positive, and the values are reflected up. Figure 10.3 shows an example with a random set of values plotted as black dots and the reflected values as white dots. Again, note our simplification, we only consider real coefficients. In closed form, we compute

$$\begin{aligned} c_i &\to \mu + (\mu - c_i) = (2\mu - c_i), \\ \sum_x c_x|x\rangle &\to \sum_x (2\mu - c_x)|x\rangle. \end{aligned} \tag{10.2}$$

10.1.4 Simple Numerical Example

With these new concepts, we can now describe a single step in Grover's algorithm using the simple example with 4 qubits and 16 states, as shown in Figure 10.1. Here is how it works:

1. **Initialization.** As seen in Section 10.1.1, we put states in superposition and start with all the states being equally likely with an amplitude of $1/\sqrt{N}$.
2. **Phase inversion.** Apply phase inversion as shown in Equation (10.1). The amplitude of the special element becomes negative, thus pushing the mean of all amplitudes down. In our example with 16 states and amplitude $1/\sqrt{16} = 0.25$, the overall mean is roughly pushed down to $(0.25 * 15 - 0.25)/16 = 0.22$.

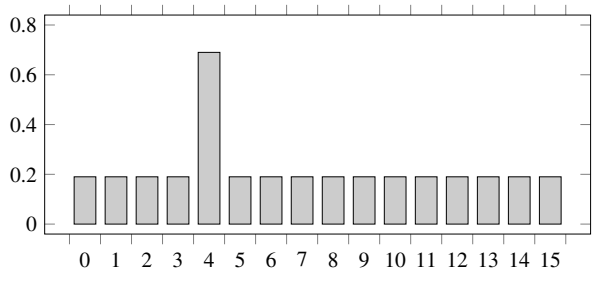


Figure 10.4 Distribution of (real) amplitudes after phase and mean inversion. The amplitude for the special element $|x'\rangle = |4\rangle$ has been amplified, and all other amplitudes have been lowered.

3. **Inversion around the mean**. This will reduce the amplitudes of 0.25 to $0.22 + (0.22 - 0.25) = 0.19$ but will amplify the special element to a value of $0.22 + (0.22 + 0.25) = 0.69$.

For the general case, rinse and repeat steps 2 and 3. For our artificial amplitude example above, a single step transforms the initial state into the state shown in Figure 10.4.

10.1.5 Two-Qubit Example

Let us make this even more concrete and visualize the procedure using an example with two qubits, inspecting the operator matrices and state vectors. In a two-qubit system, our special element shall be $|x'\rangle = |11\rangle$ with its corresponding outer product:

$$|x'\rangle = |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad \text{and} \quad |x'\rangle\langle x'| = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The solution $|x'\rangle$ corresponds to the solution space $|\beta\rangle$ in Figure 10.5. The phase inversion operator U_f from step 2 in Section 10.1.1 then becomes the following (note that in the implementation below, we use a different methodology to get this operator):

$$U_f = I - 2|x'\rangle\langle x'| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

We know how to create an equal superposition state $|s\rangle = |{+}{+}\rangle$. The state $|x^{\perp}\rangle$ is the difference between $|s\rangle$ and $|x'\rangle$ and corresponds to the axis $|\alpha\rangle$ in Figure 10.5, which is the subspace of all non-solutions:

$$|s\rangle = H^{\otimes 2}|00\rangle = |{+}{+}\rangle = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad \text{and} \quad |x^{\perp}\rangle = \frac{1}{\sqrt{3}}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

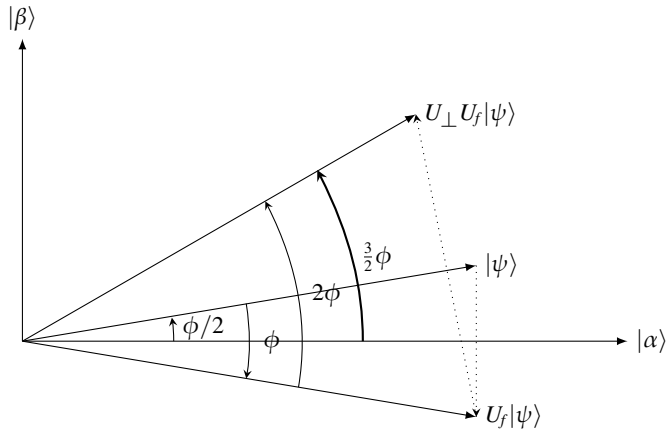


Figure 10.5 Geometric interpretation of a Grover rotation.

The state $|x^\perp\rangle$ is *orthogonal* to $|x'\rangle = |\beta\rangle$, the subspace of solutions in Figure 10.5. We can confirm this by computing the inner product $\langle x^\perp|x'\rangle = 0$. The state $|x^\perp\rangle$ is also "close" to $|s\rangle$ when we think of closeness as how many 1s and 0s are common in the state vectors (this is also called *Hamming distance*).[2]

The state $|\psi\rangle$ in the figure corresponds to the initial $|s\rangle$. It is easy to see how applying the operator U_f inverts the phase of the $|x'\rangle$ component in $|s\rangle$ to

$$U_f\,|\psi\rangle = U_f\,|s\rangle = \frac{1}{2}\begin{pmatrix}1\\1\\1\\-1\end{pmatrix}. \tag{10.3}$$

In Figure 10.5, this corresponds to a reflection of the state $|\psi\rangle$ (which is our $|s\rangle$) about the α-axis, drawn as the bottom vector marked as $U_f\,|\psi\rangle$. The inversion about the mean operator $U_\perp$, as defined in step 3 above, is

$$\begin{aligned} U_\perp &= 2\big(|+\rangle\,\langle+|\big)^{\otimes 2} - I^{\otimes 2} \\ &= 2|s\rangle\langle s| - I^{\otimes 2} \\ &= \frac{1}{2}\begin{pmatrix}-1 & 1 & 1 & 1\\ 1 & -1 & 1 & 1\\ 1 & 1 & -1 & 1\\ 1 & 1 & 1 & -1\end{pmatrix}. \end{aligned}$$

This operator $U_\perp$ reflects $U_f|\psi\rangle$ from Equation (10.3) about the original state $|s\rangle$ into the new state $U_\perp U_f|\psi\rangle = |11\rangle$:

$$\frac{1}{2}\begin{pmatrix}-1 & 1 & 1 & 1\\ 1 & -1 & 1 & 1\\ 1 & 1 & -1 & 1\\ 1 & 1 & 1 & -1\end{pmatrix}\frac{1}{2}\begin{pmatrix}1\\1\\1\\-1\end{pmatrix} = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} = |11\rangle\,.$$

[2] See http://en.wikipedia.org/wiki/Hamming_distance.

For this example with only two qubits, a single iteration is sufficient to move the state $|s\rangle$ to the special state $|x'\rangle = |11\rangle$ that we were looking for. All of these steps look very compact in code:

```
    x = state.bitstring(1, 1)
    s = ops.Hadamard(2)(state.bitstring(0, 0))

    Uf = ops.Operator(ops.Identity(2) - 2 * x.density())
    Ub = ops.Operator(2 * s.density() - ops.Identity(2))
    (Ub @ Uf)(s).dump()
>>
|11> (|3>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

The iteration count of 1 for this example agrees with the general iteration count in Equation (10.7) below, which we will derive next.

10.1.6 Iteration Count

How many iterations k should we perform? How do we know when to stop? It turns out we need exactly k iterations (with k rounded down to an integer), where

$$k = \frac{\pi}{4}\sqrt{N}.$$

Let's derive this result. First, we define two subspaces as indicated above: The space $|\alpha\rangle$ of all states that do not contain a special element and the space $|\beta\rangle$ of only special states. Note that in Grover's algorithm, we search for only one special element $|x'\rangle$, but here we generalize this derivation to search for M solutions in a population of N elements. As we are using Hadamard gates, all states are in equal superposition. The two subspaces are

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}} \sum_{x \notin S} |x\rangle \quad \text{and} \quad |\beta\rangle = \frac{1}{\sqrt{M}} \sum_{x' \in S} |x'\rangle.$$

With this, we can define the whole state $|\psi\rangle$ as the composite of the two subspaces:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle. \tag{10.4}$$

We can visualize this space in two dimensions, where the x-axis corresponds to state space $|\alpha\rangle$ and the y-axis to solution space $|\beta\rangle$, as shown in Figure 10.5.

Application of phase inversion (with the corresponding operator U_f, as before) reflects the state about the axis or subspace $|\alpha\rangle$. This, in essence, negates the second part of the superposition, similar to the effect of a Z gate on a single qubit, where a and b are the probability amplitudes for the subspaces α and β:

$$U_f\big(a|\alpha\rangle + b|\beta\rangle\big) = a|\alpha\rangle - b|\beta\rangle.$$

In the figure, this is shown as the vector marked with $|\psi\rangle$ being reflected about the axis marked as $|\alpha\rangle$ to the final vector marked as $U_f\,|\psi\rangle$.

The inversion around the mean (with operator $U_\perp$) then performs another reflection about the vector $|\psi\rangle$. The two reflections compound to a rotation, which means that the state remains in the space spanned by $|\alpha\rangle$ and $|\beta\rangle$. Furthermore, the state rotates incrementally towards the solution space $|\beta\rangle$. We have seen in Equation (10.4) that

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle.$$

We can geometrically position the state vector with simple trigonometry. We define the initial angle between $|\psi\rangle$ and $|\alpha\rangle$ as $\phi/2$. Moving forward, Equation (10.5) will be important; we use it in Section 10.3 on quantum counting:

$$\begin{aligned} \cos\left(\frac{\phi}{2}\right) &= \sqrt{\frac{N-M}{N}}, \\ \sin\left(\frac{\phi}{2}\right) &= \sqrt{\frac{M}{N}}, \\ |\psi\rangle &= \cos\left(\frac{\phi}{2}\right)|\alpha\rangle + \sin\left(\frac{\phi}{2}\right)|\beta\rangle. \end{aligned} \tag{10.5}$$

From Figure 10.7, we can see that after phase inversion and inversion about the mean, the state has rotated by ϕ towards $|\beta\rangle$. The angle between $|\alpha\rangle$ and $|\psi\rangle$ is now $\frac{3}{2}\phi$. We call the combined operator the Grover operator $G = U_\perp U_f$, which, after one iteration, produces the state

$$G|\psi\rangle = \cos\left(\frac{3\phi}{2}\right)|\alpha\rangle + \sin\left(\frac{3\phi}{2}\right)|\beta\rangle.$$

We can see that repeated application of the Grover operator G will take the state to

$$G^k|\psi\rangle = \cos\left(\frac{2k+1}{2}\phi\right)|\alpha\rangle + \sin\left(\frac{2k+1}{2}\phi\right)|\beta\rangle.$$

Now, to maximize the probability of measuring $|\beta\rangle$, the term $\sin\left(\frac{2k+1}{2}\phi\right)$ should be as close to 1 as possible. Taking the $\arcsin$ of the expression yields

$$\begin{aligned} \sin\left(\frac{2k+1}{2}\phi\right) &= 1 \\ \frac{2k+1}{2}\phi &= \pi/2 \\ k = \frac{\pi}{2\phi} - \frac{1}{2} &= \frac{\pi}{4\frac{\phi}{2}} - \frac{1}{2}. \end{aligned} \tag{10.6}$$

Note that an iteration count must be an integer, so the question we face now is what to do with the term $-1/2$. In our implementation, we simply ignore it. For our examples below, the probabilities of finding solutions are around 40% or higher, and this term seems to have no impact. Now let us solve for k. From Equation (10.5), we know that

$$\sin\left(\frac{\phi}{2}\right) = \sqrt{\frac{M}{N}}.$$

Since we can assume that $N \gg M$, we use the approximation that, for small angles, $\sin(x) \approx x$. Substituting in $\frac{\phi}{2} = \sqrt{M/N}$ and $M = 1$ into Equation (10.6), we reach the final result for the number of iterations k as the rounded down integer of

$$k = \frac{\pi}{4}\sqrt{\frac{N}{M}} = \frac{\pi}{4}\sqrt{N}. \tag{10.7}$$

10.1.7 Phase Inversion Oracle Operator

We have already seen the mathematical way to compute the matrix operator $U_f = I - 2|x'\rangle\langle x'|$. As a second strategy, we will use an oracle operator, which, at this point, we *suspect* can be implemented as a circuit (we also want to demonstrate the utility of the oracle operator one more time).

The oracle structure, shown in Figure 10.6, is similar to the Deutsch–Jozsa oracle – the input x is a whole register of qubits initialized as $|0\rangle$ and put in equal superposition with Hadamard gates. The lower qubit y is an ancilla initialized as $|1\rangle$. The Hadamard gate puts it in state $|-\rangle$.

Recall from Equation (10.1) that for U_f, the goal is to transform the input state as

$$|\psi\rangle = \sum_x c_x|x_i\rangle \quad \rightarrow_{\text{inv}} \quad |\psi\rangle = \sum_x c_x(-1)^{f(x)}|x\rangle.$$

How does this work? State $|-\rangle$ is

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

Since we use an oracle, all input values are computed in parallel. If $f(x) = 0$, the bottom qubit in state $|-\rangle$ is XOR'ed with $|0\rangle$, which means that its state remains unmodified. However, if $f(x) = 1$, the ancilla qubit in state $|-\rangle$ is XOR'ed with $|1\rangle$, which means the state gets negated as

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

For the ancilla, the result in closed form is $(-1)^{f(x)}|-\rangle$. The tensor product of input bits and the ancilla is

$$\sum_x c_x|x\rangle(-1)^{f(x)}|-\rangle.$$

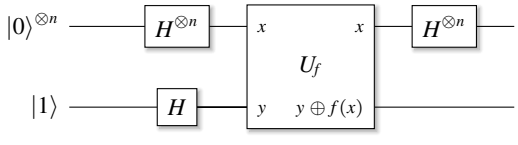


Figure 10.6 The Grover oracle is similar to the Deutsch–Jozsa oracle detailed in Section 8.3.

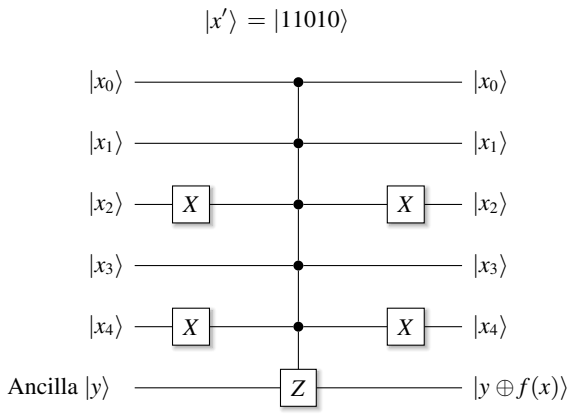


Figure 10.7 Phase inversion circuit for the special element $|x'\rangle = |11010\rangle$ in the state space of $n = 5$ qubits. The controlled Z gate acts on the ancilla non-trivially *only* when the first five control qubits are in state $|x'\rangle$.

We slightly rearrange the terms, ignore the ancilla,[3] and arrive at the closed form we are looking for:

$$|\psi\rangle = \sum_x c_x(-1)^{f(x)}|x\rangle.$$

10.1.8 Phase Inversion Circuit

So far, we have constructed the phase inversion operator as a giant matrix, which is inefficient for larger numbers of qubits. Here is a more efficient construction with a multi-controlled Z gate. It will show better performance, even though $n-1$ ancilla qubits are required with our implementation, as outlined in Section 2.10.3. We are trying to compute a unitary operator U_f such that

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle, \text{ where } \begin{cases} f(x) = 0, & x \neq x', \\ f(x) = 1, & x = x'. \end{cases}$$

The way to read this is that we only want to apply phase inversion for the special state $|x'\rangle$ for which $f(x') = 1$. This means we must multi-control the final qubit as shown in Figure 10.7, ensuring that the control bits are all $|1\rangle$ *only* for the special state. We will find more instances of this type of construction in other Grover-based algorithms later in this book.

10.1.9 Inversion about the Mean Operator

To reiterate, inversion about the mean μ is this procedure:

$$\sum_x c_x|x\rangle \rightarrow \sum_x (2\mu - c_x)|x\rangle.$$

[3] We can ignore the ancilla here, but in general, we should uncompute it.

In matrix form, we can accomplish this by multiplying the state vector with a matrix with values $2/N$ everywhere, except for the diagonal elements, which are $2/N - 1$. We derive this matrix in the following paragraphs. The matrix can also be written as

$$U_\perp = 2(|+\rangle \langle +|)^{\otimes n} - I^{\otimes n}.$$

This matrix is also called *diffusion operator* for a variety of reasons. The main reason is that Lov Grover himself called it the diffusion operator because of how probabilities seem to spread out with bias towards the special element(s). There are similarities to diffusion in that values closer to the mean decrease, while values farther away from the mean increase. This is the operator we aim to construct:

$$U_\perp = \begin{pmatrix} 2/N-1 & 2/N & \dots & 2/N \\ 2/N & 2/N-1 & \dots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \dots & 2/N-1 \end{pmatrix}. \tag{10.8}$$

Why do we look for this specific operator? Recall from Equation (10.2) that we want to construct an operator that performs the transformation

$$\sum_x c_x|x\rangle \to \sum_x (2\mu - c_x)|x\rangle.$$

How does the operator $U_\perp$ work? Each row multiplies and adds each state vector element by $2/N$ before subtracting the one element corresponding to the diagonal:

$$\begin{pmatrix} 2/N-1 & 2/N & \dots & 2/N \\ 2/N & 2/N-1 & \dots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \dots & 2/N-1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} \tag{10.9}$$

$$= \begin{pmatrix} (2c_0/N + 2c_1/N + \cdots + 2c_{n-1}/N) - c_0 \\ (2c_0/N + 2c_1/N + \cdots + 2c_{n-1}/N) - c_1 \\ \vdots \\ (2c_0/N + 2c_1/N + \cdots + 2c_{n-1}/N) - c_{n-1} \end{pmatrix}$$

$$= \begin{pmatrix} 2\mu - c_0 \\ 2\mu - c_1 \\ \vdots \\ 2\mu - c_{n-1} \end{pmatrix}.$$

We derived the matrix from Equation (10.8) mathematically. But what procedure and what operators should we use in practice to get this matrix as an operator? We have seen the geometrical interpretation above. We can think of inversion about the mean as a reflection around a subspace. Hence, a possible derivation consists of three steps:

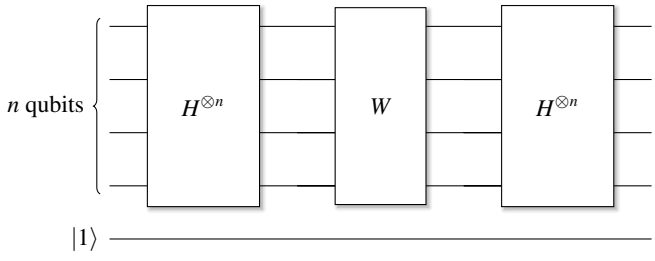


Figure 10.8 Inversion about the mean circuit for the operator $U_{\perp}$. The diagonal matrix $W = 2(|0\rangle\langle 0|)^{\otimes n} - I^{\otimes n}$ has a 1 in the top left element, and all remaining diagonal elements are -1.

1. Ideally, we would like to rotate the space in equal superposition $|+ + \cdots +\rangle$. But it is hard to construct an operator to do this reflection in this basis. Therefore, we use Hadamard gates to get into the computational basis and construct the reflection there.
2. Leaving the Hadamard basis, the state $|+ + \cdots +\rangle$ becomes the state $|00\ldots 0\rangle$, which seems like an obvious choice to reflect about. We could pick another state for reflection, as long as that state is still almost orthogonal to the subspace α, but for state $|00\ldots 0\rangle$, the inversion operator has an elegant construction (which we show in Section 10.1.10).
3. Transform the state back to the Hadamard basis with Hadamard gates.

These three steps define the circuit shown in Figure 10.8. For Steps 1 and 3, we apply Hadamard gates to get in and out of the computational basis, as we are in the Hadamard basis from the phase inversion before. For Step 2, we will want to leave the state $|00\ldots 0\rangle$ alone but reflect all other states. If we think about how states are represented in binary and how matrix–vector multiplication works, we can achieve this by constructing the matrix W, which is easy to derive as

$$\begin{aligned} W &= 2(|0\rangle\langle 0|)^{\otimes n} - I^{\otimes n} \\ &= \begin{pmatrix} 2 & & & \\ & 0 & & \\ & & \ddots & \\ & & & 0 \end{pmatrix} - \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \end{pmatrix}. \end{aligned} \tag{10.10}$$

Again, we could pick any state as the axis to reflect about, but the math is elegant and simple when picking the state $|00\ldots 0\rangle$. This will become clearer with the derivation immediately below.

Only the first bit in the state vector remains unmodified, and that first bit corresponds to the state $|00\ldots0\rangle$, as indicated in Equation (10.10). Recall that the state vector for this state is all 0s, except the very first element, which is a 1. The projector $P_{|0\rangle} = |0\rangle\langle 0|$ has a single 1 at the top left corner and 0s everywhere else. Therefore, using the matrix W, which has a -1 on all remaining diagonal elements, all other states are negated. In combination, we want to compute

$$\begin{aligned} H^{\otimes n}WH^{\otimes n} &= H^{\otimes n}\begin{pmatrix} 1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \end{pmatrix} H^{\otimes n} \\ &= H^{\otimes n}\left[\begin{pmatrix} 2 & & & \\ & 0 & & \\ & & \ddots & \\ & & & 0 \end{pmatrix} - I\right] H^{\otimes n} \\ &= H^{\otimes n}\begin{pmatrix} 2 & & & \\ & 0 & & \\ & & \ddots & \\ & & & 0 \end{pmatrix} H^{\otimes n} - H^{\otimes n}IH^{\otimes n}. \end{aligned}$$

Since the Hadamard operator is its own inverse, the second term reduces to the identity matrix I. Multiplying in the left and right Hadamard gates as

$$\begin{aligned} &= \begin{pmatrix} 2/\sqrt{N} & 0 & \ldots & 0 \\ 2/\sqrt{N} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 2/\sqrt{N} & 0 & \ldots & 0 \end{pmatrix} H^{\otimes n} - I \\ &= \begin{pmatrix} 2/N & 2/N & \ldots & 2/N \\ 2/N & 2/N & \ldots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \ldots & 2/N \end{pmatrix} - I. \end{aligned}$$

Finally, subtracting the identity I produces a matrix where all elements are $2/N$, except the diagonal elements, which are $2/N - 1$:

$$U_{\perp} = \begin{pmatrix} 2/N-1 & 2/N & \ldots & 2/N \\ 2/N & 2/N-1 & \ldots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \ldots & 2/N-1 \end{pmatrix}.$$

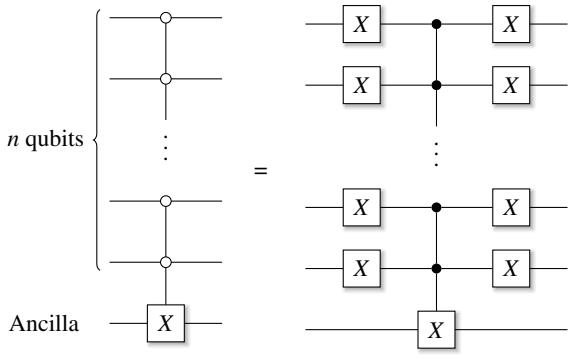


Figure 10.9 Inversion about the mean circuit (omitting leading and trailing Hadamard gates applied to the top n qubits). We want to apply the X gate to state $|00\ldots0\rangle$ only. Hence all control bits must be $|0\rangle$.

This is the matrix $U_\perp$ we were looking for. Applying this matrix to a state transforms each element c_x into $2\mu - c_x$, as shown in Equation (10.9).

10.1.10 Inversion about the Mean Circuit

As a third implementation strategy, we can construct a quantum circuit for the inversion about the mean using reasoning similar to that for the phase inversion operator (Mermin, 2007).

The main "trick" for building an operator for mean inversion is to realize that the direction of the rotation for amplitude amplification does not matter; it can be negative or positive. This means that instead of constructing $W = 2(P_{|0\rangle})^{\otimes n} - I^{\otimes n}$ as before, we construct

$$W' = I^{\otimes n} - 2(P_{|0\rangle})^{\otimes n} = I^{\otimes n} - 2|00\ldots0\rangle\langle00\ldots0|.$$

We want to build a gate that leaves all states untouched, except $|00\ldots0\rangle$, for which we want to flip the amplitudes. An X gate will do this for us. Because the X gate must be controlled to apply only to $|00\ldots0\rangle$, we expect all inputs to be $|0\rangle$. Hence, to control the X gate, we sandwich the controller qubits between the X gates, omitting the left and right Hadamard gates from the construction in Equation (10.8), as shown in Figure 10.9.

As a result, for the big inversion operator $U_\perp$ from Equation (10.8), the circuit in Figure 10.9 corresponds to the closed form below (with $(CX)^{n+1}$ indicating a multi-controlled X gate with n control qubits controlling the ancilla qubit at index $n+1$), which yields the operator

$$U_\perp = H^{\otimes n}X^{\otimes n}(CX)^{n+1}X^{\otimes n}H^{\otimes n}.$$

In the implementation shown below, we can verify that the order of rotation does not matter by modifying this line in file `grover.py`:

```
<<
  reflection = op_zero * 2.0 - ops.Identity(nbits)
>>
  reflection = ops.Identity(nbits) - op_zero * 2.0
```

10.1.11 Oracle Implementation of Grover's Algorithm

Now let's put all the pieces together. The complete Grover iteration circuit is shown in Figure 10.10. In the code, we first define the function f we intend to analyze. The `make_f` function creates an array of all 0s, except for one or more special elements randomly set to 1, corresponding to $|x'\rangle$. The function returns a lambda function object that converts its parameter, a sequence of address bits, to a decimal index and returns the value of the array at that index.

PY

Find the code
In file `src/grover.py`

```
def make_f(d: int = 3, solutions: int = 1):
  answers = np.zeros(1 << d, dtype=np.int8)
  solutions = random.sample(range(1 << d), nsolutions)
  answers[solutions] = 1
  return lambda bits: answers[helper.bits2val(bits)]
```

The initial state of the circuit is a register of $|0\rangle$ qubits with an additional ancilla qubit in state $|1\rangle$. Applying the Hadamard gate to all of the qubits puts the ancilla into the state $|-\rangle$.

```
  # State initialization:
  psi = state.zeros(nbits) * state.ones(1)
  for i in range(nbits + 1):
    psi.apply(ops.Hadamard(), i)
```

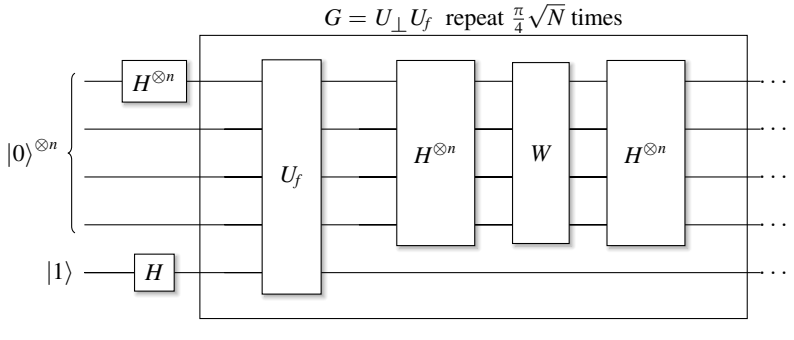


Figure 10.10 Full circuit for Grover algorithm with the U_f and $U_\perp$ operators. As drawn, the circuit would only perform a single iteration.

In order to implement phase inversion, we generate an oracle with the function object we created above. To create the oracle itself, we use our trusty `OracleUf` operator and give it the function object as a parameter. Using an oracle this way is slow as it utilizes the full matrix implementation. Of course, any given operator can be implemented with quantum gates, but this can be quite cumbersome.[4] Fortunately for us, this is not the case here, as shown for the elegant phase inversion operator in Figure 10.9.

```
f = make_f(nbits)
uf = ops.OracleUf(nbits+1, f)
```

Now we move on to mean inversion. We first construct an all-0 matrix with a single 1 in the top left element. This is equivalent to building an `nbits`-dimensional $|0\rangle\langle 0|$ projector (using the helper function `ZeroProjector`). With this, we construct the $2|00\ldots0\rangle\langle 00\ldots0| - I^{\otimes n}$ reflection matrix:

```
op_zero = ops.ZeroProjector(nbits)
reflection = op_zero * 2.0 - ops.Identity(nbits)
```

The full inversion operator $U_{\perp}$ consists of the Hadamard gates surrounding the reflection matrix W. We add an identity gate to account for the ancilla we added earlier for the phase inversion oracle. We build the complete Grover operator `grover` as the product of the mean inversion operator `inversion` with the phase inversion operator `uf`. Finally, we iterate the desired number of times based on the size of the state, as calculated with Equation (10.7):

```
hn = ops.Hadamard(nbits)
inversion = hn(reflection(hn)) * ops.Identity()
grover = inversion(uf)

iterations = int(math.pi / 4 * math.sqrt(2**nbits))
for _ in range(iterations):
  psi = grover(psi)
```

To check whether we have computed the right result, we perform measurement by peek-a-boo and compare the state with the highest probability to the intended result:

```
maxbits, maxprob = psi.maxprob()
result = f(maxbits[:-1])
print('Got f({}) = {}, want: 1, #: {:2d}, p: {:6.4f}'
      .format(maxbits[:-1], result, solutions, maxprob))
assert result == 1, 'Something went wrong, invalid state'
```

[4] Perhaps *more* cumbersome than what we have shown so far.

Experimenting with a few bit widths should produce a result similar to this:

```
def main(argv):
  for nbits in range(3, 8):
    run_experiment(nbits)
>>
Got f((1, 0, 1)) = 1, want: 1, #:  1, p: 0.3906
Got f((1, 0, 1, 1)) = 1, want: 1, #:  1, p: 0.4542
Got f((1, 0, 1, 0, 0)) = 1, want: 1, #:  1, p: 0.4485
Got f((1, 0, 0, 1, 1, 1)) = 1, want: 1, #:  1, p: 0.4818
Got f((0, 1, 0, 1, 0, 0, 0)) = 1, want: 1, #:  1, p: 0.4710
```

So far, we have operated with big matrices and projectors, which are easy to construct mathematically. In order to implement Grover's algorithm on a physical machine, we need to implement the operators with gates. This is the topic of Section 10.1.12.

10.1.12 Circuit Implementation of Grover's Algorithm

Now, let us explore the implementation of Grover's algorithm using gates instead of big matrices. We will find similar constructions in many variants of the algorithm. To start, we modify the function `make_f` from Section 10.1.11 to mark only a single special element and additionally return the binary bit pattern for the special element:

```
def make_f1(d: int = 3):
  answers = np.zeros(1 << d, dtype=np.int8)
  answer_true = np.random.randint(0, 1 << d)
  answers[answer_true] = 1
  return (lambda bits: answers[helper.bits2val(bits)],
          helper.val2bits(answer_true, d))
```

We introduce a helper function `multi-masked` that applies a gate only if it matches a specific mask and masking value:

```
def run_experiment_circuit(nbits: int) -> None:
  def multi_masked(qc: circuit.qc, gate: ops.Operator, idx: List[int],
                   mask, allow: int):
    for i in idx:
      if mask[i] == allow:
        qc.apply1(gate, i, 'multi-mask')
```

To construct the state, we add the input register, another ancilla initialized with $|1\rangle$, and an additional `aux` register for the multi-controlled gates used later in the circuit:

```
qc = circuit.qc('Grover')
reg = qc.reg(nbits, 0)  # n bits for functions
qc.reg(1, 1)  # ancilla.
aux = qc.reg(nbits - 1, 0)  # auxiliary bits for multi_control
```

We create the function object, which gives us the bit pattern of the special elements in variable `bits`. We also compute the number of iterations as in the oracle-based implementation.

```
f, bits = make_f1(nbits)
iterations = int(math.pi / 4 * math.sqrt(2**nbits))
```

In addition, we create a range of indices in the list variable `idx`. These are the indices for the qubits to which we want to apply the Hadamard and X gates in the diffusion circuit. At the start of the algorithm, we also have to apply Hadamard gates to all qubits, including the ancillary qubit. We use a similar Python list comprehension for this (similar to other single-qubit gates, the Hadamard function `qc.h` accepts a single qubit index as well as a list of indices as input):

```
idx = [i for i in range(nbits)]
qc.h([i for i in range(nbits + 1)])
```

With these pieces in place, we can now create the loop and construct phase inversion and mean inversion circuits, as outlined above. For phase inversion, we apply Z gates to the qubit indices that represent a 0 in the binary representation of the special element. This is encompassed by passing `bits` with a mask of `0` to `multi_masked`. For the mean inversion, we use the convenience of being able to pass lists of indices to the single-gate functions and apply the controlled X gate to the ancilla, which resides at index `nbits`:

```
for _ in range(iterations):
  # Phase Inversion
  multi_masked(qc, ops.PauliX(), idx, bits, 0)
  qc.multi_control(reg, nbits, aux, ops.PauliZ(), 'Phase Inversion')
  multi_masked(qc, ops.PauliX(), idx, bits, 0)

  # Mean Inversion
  qc.h(idx)
  qc.x(idx)
  qc.multi_control(reg, nbits, aux, ops.PauliX(), 'Mean Inversion')
  qc.x(idx)
  qc.h(idx)
[...] # check results
```

All that is left now is to obtain the state with the highest probability and to ensure that everything goes as planned. This code is almost identical to the equivalent code

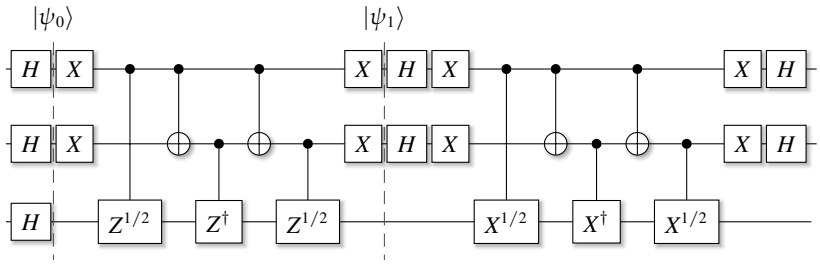


Figure 10.11 A 2-qubit Grover algorithm implementation with a bottom ancilla qubit. The phase inversion subcircuit starts at $|\psi_0\rangle$, and the mean inversion subcircuit starts at $|\psi_1\rangle$. Measurement gates have been omitted.

in the oracle-based implementation, so we do not replicate it here. It is interesting to compare the run-time behavior and performance of the matrix-based and circuit-based implementations; you may want to run a few experiments. In Figure 10.11, we show the full circuit for a 2-qubit Grover circuit.

10.2 Quantum Amplitude Amplification (QAA)

How should we modify Grover's algorithm to account for a set of solutions S with more than one special element? Naively, this is relatively easy to achieve: We have to adjust the phase inversion, the inversion about the mean, and the iteration count. The function `make_f` shown above already accepts parameter `solutions` to specify how many elements to mark.

We derived the proper iteration count in the derivation for Grover's algorithm in Equation (10.7) as

$$k = \frac{\pi}{4}\sqrt{\frac{N}{M}}, \quad \text{with } M \ll N.$$

In Section 10.1.6 we assumed $M = 1$. To account for multiple special elements, we have to adjust the computation of the iteration count and divide by a larger M, which is the parameter `solutions` in the code.

PY **Find the code**
In file src/grover.py

```
iterations = int(math.pi / 4 * math.sqrt(2**nbits / solutions))
```

We add a test sequence to our main driver code to check whether any solution can be found and with what probability. For good performance, we hold the number of qubits at eight and gradually increase the number of solutions from 1 to 8:

```
for solutions in range(1, 9):
  run_experiment(8, solutions)
```

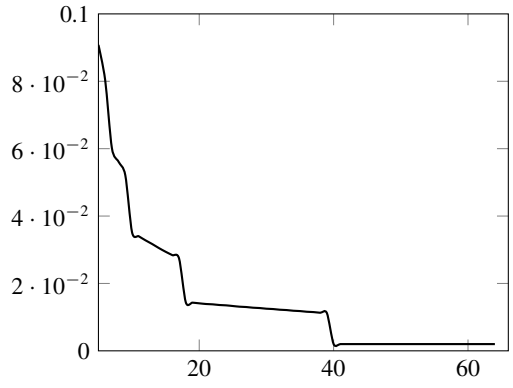


Figure 10.12 Probability of finding a special element when the total number of such elements ranges from 5 up to 64 in a state space of 128 elements. The *y*-axis shows the probability; the *x*-axis shows the number of special elements.

If we were to print the number of states with nonzero probability, we would find that all of their probabilities are identical but that there are twice as many states with nonzero probability as there are solutions! This is an artifact of our oracle construction and the entanglement with the ancilla qubit.

As we run the experiment, and if all goes well, we should get output like the following. Note how we increase the number of solutions (`sols`):

```
Got f((1, 1, 0, 0, 1, 0, 0, 0)) = 1, want: 1, sols: 1, found 1, p: 0.491
Got f((1, 0, 1, 1, 1, 1, 0, 1)) = 1, want: 1, sols: 2, found 1, p: 0.235
Got f((1, 1, 1, 0, 0, 1, 1, 0)) = 1, want: 1, sols: 3, found 1, p: 0.162
Got f((0, 0, 1, 0, 0, 0, 1, 0)) = 1, want: 1, sols: 4, found 1, p: 0.120
```

In this experiment, the algorithm is able to find a single solution each time, but the probability of finding a solution (`p`) decreases as the total number of solutions increases. This is a limitation of Grover's algorithm when dealing with multiple solutions, and the results here are consistent with that theoretical understanding. As M grows, the small-angle approximation used in Grover's algorithm no longer holds.

Let us visualize the probabilities in the graph in Figure 10.12. On the *x*-axis, we have the number of solutions ranging from 5 to 64. On the *y*-axis, we ignore the first few cases with high probability and set a maximum of 0.1. We can see how the probabilities decrease rapidly and drop to 0 after the total number of solutions exceeds 40.

What if there are many more solutions, perhaps even a majority of the state space,[5] or what if the solution probabilities are not all equal? To answer these questions, Grover's algorithm has been generalized by Brassard et al. (2002) as *Quantum Amplitude Amplification* (QAA), which we discuss in this section.

[5] Let us ignore that in this scenario, a random choice would give a correct solution with high probability.

Grover expected only one special element and initialized the search with an equal superposition of all inputs by applying the *algorithm* $A = H^{\otimes n}$ to the input. Note the unusual use of the term algorithm. In the context of this section, an algorithm can mean just a single gate, as in Grover, with equal superposition states. But it can also mean other, more complex algorithms with sequences of gates that may produce unequal state probabilities. QAA supports any algorithm A to initialize the input and changes the Grover iteration to the more general form:[6]

$$Q = AU_{\perp}A^{\dagger}U_f. \tag{10.11}$$

The operator U_f is the phase inversion operator for multiple solutions, and $U_{\perp}$ is the inversion about the mean operator that we saw in Grover's algorithm. What changes is the derivation of the iteration count k, which has been shown to be proportional to the probability p_{good} of finding a solution (see Kaye et al., 2007, section 8.2). For QAA, the iteration count shall be

$$k = \sqrt{\frac{1}{p_{\text{good}}}}.$$

The square root in the formula for k comes from Grover's algorithm, where the number of iterations needed to find a solution scales as the square root of the ratio of the search space to the number of solutions. In Quantum Amplitude Amplification (QAA), this idea is generalized: The number of iterations is proportional to the inverse square root of the probability p_{good} of finding a solution. This reflects how the algorithm amplifies the probability of success over time.

Let us see how the probabilities improve with this new and improved iteration count. As an experiment, we keep $A = H^{\otimes n}$ and compute the new iteration count as the following, where we now divide by `solutions` to reflect the probability of finding a solution:

```
iterations = int(math.sqrt(2**nbits / solutions))
```

Figure 10.13 shows the probabilities for the two iteration counts, where the thick line represents the probabilities obtained with the new iteration count. We see that the situation improves markedly, but the probabilities still drop to 0 for more than 64 solutions. When the ancilla qubit is used in Grover's algorithm, it becomes entangled with the other qubits. This entanglement effectively doubles the number of basis states because the ancilla qubit can be $|0\rangle$ or $|1\rangle$. As a result, even though the number of solutions (marked by the oracle) remains the same, there are now twice as many states overall with nonzero probabilities due to the extra entanglement introduced by the ancilla qubit.

When the number of solution states becomes half of the total state space, Grover's algorithm becomes inefficient and starts to fail because the amplitude amplification

[6] You will also see this written as $Q = AU_{\perp}A^{-1}U_f$. However, the algorithm A is unitary and invertible; this is why we can use the dagger.

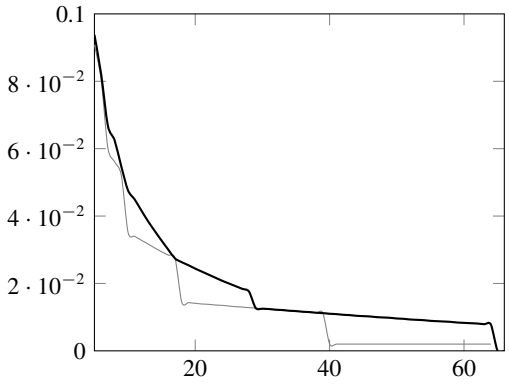


Figure 10.13 Probabilities for amplitude amplification finding 1 out of up to 64 solutions in a state space with 128 elements: (thick black line) amplitude amplification, (light gray line) and Grover's search. The y-axis shows the probability; the x-axis shows the number of special elements

starts to overshoot, reducing the probabilities of measuring a solution state. To avoid this, adding another qubit effectively doubles the size of the state space, providing more room for Grover's algorithm to work efficiently again without crashing. The additional qubit increases the total number of basis states, ensuring that the number of solutions is no longer half the state space, thereby solving the problem.

The technique of amplitude amplification requires knowledge of the number of good solutions and their probability distribution. A general technique called *amplitude estimation* can help with this (see Kaye et al., 2007, section 8.2). We detail this technique in Section 10.4. However, before that, in Section 10.3, we detail a special case of amplitude estimation, called *quantum counting*, which assumes an equal superposition of the search space with the algorithm $A = H^{\otimes n}$, similar to Grover.

10.2.1 State Preparation with QAA

In Section 9.1.2 on amplitude encoding, we hinted at a way to initialize a state such that specific states $|x_i\rangle$ have a high probability of equal magnitude, while all other states have a probability close to 0. With QAA in our arsenal of techniques, this is now straightforward to implement. You create a function that marks all the elements that should have high probability as special elements and run Grover's algorithm. After the correct number of iterations, the result will be exactly as expected. A short implementation of this technique can be found in file `state_prep.py` in the open-source repository.

Find the code
In file `src/state_prep.py`

10.3 Quantum Counting

Quantum Counting is an interesting extension of the search problems that we solved with Grover's algorithm and amplitude amplification. Combining these search algorithms with phase estimation in an interesting way solves the problem of not knowing how many solutions M exist in a population of N elements. Recall that amplitude amplification requires knowledge of M to determine the proper iteration count. Quantum counting is a special case of *amplitude estimation* that seeks to estimate this number M. Because it expects an equal superposition of the search space, similar to Grover with algorithm $A = H^{\otimes n}$, we can reuse much of the Grover implementation from Section 10.2 above.

As in Grover's algorithm, we partition the state space into a space $|\alpha\rangle$ with no solutions and the space $|\beta\rangle$ with only solutions as

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle.$$

Applying the Grover operator amounts to a rotation by an angle ϕ towards the solution space $|\beta\rangle$. You may refer again to Figure 10.5 for a graphical illustration of this process. Since this is a counterclockwise rotation, we can express the Grover operator as a standard rotation matrix:

$$G(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}.$$

Rotation matrices are unitary matrices with eigenvalues $\lambda_{0,1} = e^{\pm i\phi}$. In the analysis of Grover's algorithm, we found Equation (10.5), replicated here, with N being the number of elements and M being the number of solutions:

$$\sin\left(\frac{\phi}{2}\right) = \sqrt{\frac{M}{N}}.$$

If we had a way to find ϕ, we would be able to estimate M because we already know N. Fortunately, we will learn in Section 11.2.1 about phase estimation that will allow us to find ϕ with a circuit as shown in Figure 10.14. Don't worry about its function at this point, it will become clear in Chapter 11.

PY

Find the code
In file `src/counting.py`

Let us translate this circuit into code. We reuse the function `make_f` from Section 10.1.11. It returns 1 for a solution and 0 otherwise. Next, we build the Grover operator, just as we did in Section 10.1 on Grover's algorithm. The parameter `nbits_phase` specifies how many qubits to use for phase estimation, and parameter `nbits_grover` indicates how many qubits to use for the Grover operator itself. Since this code utilizes the full matrix implementation, we can use only a limited number of qubits. Nevertheless, the more qubits we use for phase estimation, the more numerically accurate the results will become.

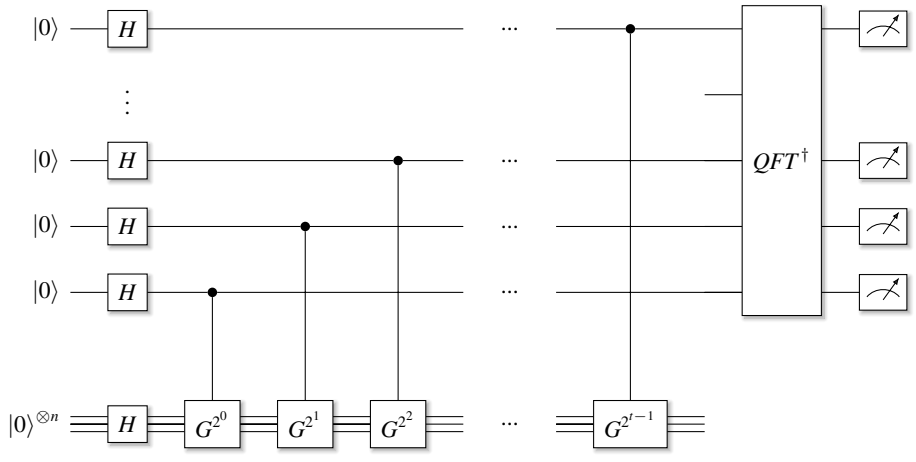


Figure 10.14 Phase estimation for the Grover operator G.

In the code, we use Hadamard gates before and after the reflection operator. Later, in Section 10.4, we will generalize and allow for other operators (algorithms) that may result in unequal probability distributions.

```
def run_experiment(nbits_phase: int, nbits_grover: int,
                   solutions: int) -> None:

  op_zero = ops.ZeroProjector(nbits_grover)
  reflection = op_zero * 2.0 - ops.Identity(nbits_grover)

  hn = ops.Hadamard(nbits_grover)
  inversion = hn(reflection(hn)) * ops.Identity()
  grover = inversion(u)

  f = make_f(nbits_grover, solutions)
  u = ops.OracleUf(nbits_grover + 1, f)
```

Now we build the circuit in Figure 10.14. The Grover operator needs an ancilla in state $|1\rangle$, which we also add to the state (not shown in Figure 10.14). We apply a Hadamard gate to the inputs and the ancilla and then apply the Grover operator `iterations` times:

```
  psi = state.zeros(nbits_phase + nbits_grover) * state.ones(1)
  for i in range(nbits_phase + nbits_grover + 1):
    psi.apply1(ops.Hadamard(), i)

  iterations = int(math.pi / 4 * math.sqrt(2**nbits / solutions))
  for _ in range(iterations):
    psi = grover(psi)
```

Finally, we follow this with a sequence of exponentiated gates for phase estimation and a final inverse QFT (again, the details are being explained in Chapter 11).

For convenience, we also wrap this code for phase estimation in a helper function `PhaseEstimation()`, which we will use later:

```
  cu = grover
  for inv in reversed(range(nbits_phase)):
    psi = ops.ControlledU(inv, nbits_phase, cu)(psi, inv)
    cu = cu(cu)
  psi = ops.Qft(nbits_phase).adjoint()(psi)
```

This completes the circuit. We measure and find the state with the highest probability. Then we reconstruct the phase from the binary fractions and use Equation (10.5) to estimate M, the number of solutions:

```
  maxbits, maxprob = psi.maxprob()
  phi_estimate = helper.bits2frac(maxbits)

  m = round(2**nbits_grover * math.sin(phi_estimate * math.pi) ** 2, 2)
  print(
      f'Estimate: {phi_estimate:.4f} prob: {maxprob * 100.0:5.2f}% '
      f'--> m: {m:5.2f}, want: {solutions:2d}'
  )
  assert np.allclose(np.round(m), solutions), 'Incorrect result.'
```

Let us run some experiments with seven qubits for phase estimation and four qubits for the Grover operator. For $N = 64$, we let M range from 1 to 5:

```
def main(argv):
  for solutions in range(1, 6):
    run_experiment(7, 4, solutions)
```

Running this code should produce output like the following:

```
Estimate: 0.9250 prob:  9.12% --> m:  0.87, want:  1
Estimate: 0.8843 prob:  4.73% --> m:  2.02, want:  2
Estimate: 0.1460 prob:  0.80% --> m:  3.14, want:  3
Estimate: 0.1714 prob:  0.73% --> m:  4.21, want:  4
Estimate: 0.8125 prob:  1.04% --> m:  4.94, want:  5
```

We can see that our estimates will round to the correct number of solutions. Also note that the probability of a solution decreases significantly with higher values of M.

10.4 Amplitude Estimation

In Section 10.3 on quantum counting, we constructed the Grover operator using Hadamard gates as the "algorithm" to put the state in an equal superposition of basis states.

Then we used phase estimation (details in Chapter 11) to calculate the operator's eigenvalue. From this value, given that there was an *equal* superposition, we could count the number of solutions. However, equal superposition is really a special case.

Quantum Amplitude Estimation (QAE) is the generalization of this procedure to unequal superpositions. In QAE, an algorithm A will put the state in a potentially unequal superposition of special elements (x_i') and regular elements (x_i). QAE then estimates the probability of *finding a solution*.

To recap, we have seen in Section 10.1 that for a population of N elements with M solutions, we can write the state as the following, defining the initial angle between $|\psi\rangle$ and $|\alpha\rangle$ as $\phi/2$:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle, \quad \text{and}$$
$$|\psi\rangle = \cos\left(\frac{\phi}{2}\right)|\alpha\rangle + \sin\left(\frac{\phi}{2}\right)|\beta\rangle. \tag{10.12}$$

Taken together, QAE will estimate the probability of finding a solution as

$$\sin^2\left(\frac{\phi}{2}\right) = \frac{M}{N}. \tag{10.13}$$

In QAE, we can use *any* unitary matrix representing an algorithm for state preparation. Consequently, we change the code that constructs the Grover operator in the same way as Equation (10.11) in Section 10.2 on amplitude amplification with an algorithm A:

$$Q = AU_{\perp}A^{\dagger}U_f.$$

Similarly to quantum counting, we apply phase estimation on Q. For a given eigenvector $|u\rangle$ with eigenvalue $e^{2\pi i\phi}$, phase estimation gives us the value ϕ. From Equation (10.13) we know that we are looking for $\phi/2$. Since ϕ is an angle and we work within a trigonometric context, phase estimation typically returns a phase in units ranging from 0 to 1, rather than 0 to 2π. To convert this phase into the correct trigonometric angle, the phase is multiplied by 2π. Since Equation (10.13) involves $\phi/2$, multiplying by π adjusts the phase correctly and eliminates the need to divide ϕ by 2 again. Once we have that, we know the probability of finding at least one solution.

The implementation is quite similar to quantum counting. In the code snippet below, we only highlight the few code changes required for this generalization. First, we modify `make_f` again and add a parameter specifying the list of solutions we want to mark as special elements.

Find the code
In file src/amplitude_estimation.py

```
def make_f(nbits: int, solutions: List[int]):
  answers = np.zeros(1 << nbits, dtype=np.int32)
  answers[solutions] = 1
  return lambda bits: answers[helper.bits2val(bits)]
```

We pass in the unitary representing the algorithm as parameter `algo` and construct the Grover operator with it. We also pass in the specific solutions that should be marked:

```
def run_experiment(nbits_phase: int, nbits_grover: int,
                   algo: ops.Operator,
                   solutions: List[int]) -> None:
  [...]
  inversion = algo.adjoint()(reflection(algo)) * ops.Identity()
  grover = inversion(u)
  [...]
```

As in quantum counting in Section 10.3, we perform phase estimation and apply the inverse QFT. Using Equation (10.12), we calculate the amplitude `ampl`, print, and return it.

```
  psi = ops.PhaseEstimation(grover, psi, nbits_phase, nbits_phase)
  psi = ops.Qft(nbits_phase).adjoint()(psi)

  maxbits, _ = psi.maxprob()
  ampl = np.sin(np.pi * helper.bits2frac(maxbits[:nbits_phase]))

  print('  AE: ampl: {:.2f} prob: {:5.1f} % {}/{} solutions ({})'
        .format(ampl, ampl * ampl * 100, len(solutions),
                1 << nbits_grover, solutions))
  return ampl
```

The code in the open-source repository explores a few examples of equal and unequal state superposition with a varying number of solutions. In the first experiment, we create an equal superposition state of $n = 3$ qubits with Hadamard gates. We range the number of solutions from 0 to 2^n, selected at random. We compute the resulting amplitude and ensure it is close to the expected amplitude. This is easy to calculate because it must adhere to the equal superposition. For eight qubits, the probability of finding a solution with zero marked elements must be zero. If all eight solutions are marked, the probability of finding a solution must be 100%. For everything in between, the probability should be a multiple of $1/8$.

```
  algorithm = ops.Hadamard(3)
  for nsolutions in range(9):
    ampl = run_experiment(7, 3, algorithm,
                          random.sample(range(2**3), nsolutions))
    if not math.isclose(ampl, np.sqrt(nsolutions / 2**3), abs_tol=0.03):
      raise AssertionError('Incorrect AE.')
>>>
Algorithm: Hadamard (equal superposition)
  AE: ampl: 0.00 prob:   0.0% 0/8 solutions ([])
  AE: ampl: 0.34 prob:  12.1% 1/8 solutions ([4])
  AE: ampl: 0.47 prob:  22.7% 2/8 solutions ([0, 1])
```

```
  [...]
  AE: ampl: 0.93 prob:  87.0% 7/8 solutions ([3, 6, 4, 5, 1, 2, 7])
  AE: ampl: 1.00 prob: 100.0% 8/8 solutions ([6, 0, 4, 5, 7, 3, 2, 1])
```

To experiment with other algorithms, we create a random operator that will produce an unequal superposition of basis states. In the next experiment, we individually mark a single basis state as a special element and ensure that the estimated probability matches the amplitude of that state.

In the code snippet below, we dump the state as a reference to display the probability amplitudes for each basis state. Then we run the experiment, iterating over the basis states and marking a single basis state by passing a single-element list `[i]`. As we print the estimated probabilities, you can see that they match within the rounding accuracy:

```
  i1 = ops.Identity(1)
  algorithm = (ops.Hadamard(3) @
               (ops.RotationY(random.random()/2) * i1 * i1) @
               (i1 * ops.RotationY(random.random()/2) * i1) @
               (i1 * i1 * ops.RotationY(random.random()/2)))
  psi = algorithm(state.zeros(3))
  psi.dump()
  for i in range(len(psi)):
    ampl = run_experiment(7, 3, algorithm, [i])
>>>
Algorithm: Random (unequal superposition), single solution
|000> (|0>):  ampl: +0.53+0.00j prob: 0.28 Phase:   0.0
|001> (|1>):  ampl: +0.39+0.00j prob: 0.15 Phase:   0.0
[...]
|111> (|7>):  ampl: +0.20+0.00j prob: 0.04 Phase:   0.0

  AE: ampl: 0.53 prob:  28.6% 1 out of 8 solutions ([0])
  AE: ampl: 0.38 prob:  14.3% 1 out of 8 solutions ([1])
  [...]
  AE: ampl: 0.20 prob:   3.8% 1 out of 8 solutions ([7])
```

In a final experiment, we take the same unequal superposition state and vary the number of marked solutions from 0 to 2^n. Again, for zero marked solutions, the probability of finding a solution should be zero. If all states are marked, the probability should be 100%. The probabilities for k marked solutions with $k \in [1, 2^n - 1]$ should accumulate as the sum of the individual probabilities of the marked states:[7]

```
  print('Algorithm: Random (unequal superposition), multiple solutions')
  for i in range(len(psi)+1):
    ampl = run_experiment(7, 3, algorithm, [i for i in range(i)])
>>>
Algorithm: Random (unequal superposition), multiple solutions
  AE: ampl: 0.00 prob:   0.0% 0/8 solutions ([])
```

[7] I'm pushing the limits with the list comprehension `[i for i in range(i)]`. For clarity, the rightmost `i` is from the outer loop.

```
AE: ampl: 0.53 prob:  28.6% 1/8 solutions ([0])
AE: ampl: 0.65 prob:  43.2% 2/8 solutions ([0, 1])
[...]
AE: ampl: 1.00 prob: 100.0% 8/8 solutions ([0, 1, 2, 3, 4, 5, 6, 7])
```

With all these pieces in place, let's explore a few practical applications of Grover's algorithm.

10.5 Boolean Satisfiability

The question of *Boolean Satisfiability* is the following: Given a Boolean formula in *Conjunctive Normal Form (CNF)*, does this formula have an assignment of values to variables such that the formula is True? A CNF is defined as follows.

- A *literal* is a Boolean variable or its negation, written as x or $\neg x$.
- A *clause* is a disjunction (logical OR, $\vee$) of literals, for example

$$(x_0 \vee \neg x_1 \vee \neg x_2).$$

- A *formula* is a conjunction (logical AND, $\wedge$) of clauses, for example

$$(x_0 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_0 \vee \neg x_2).$$

The goal is to find an assignment of the Boolean values True (T) and False (F) to variables x_i such that the formula becomes true. For example, for the clause $(x_0 \vee \neg x_1 \vee \neg x_2)$, an assignment of $x_0 = T$ and any random Boolean value for x_1 and x_2 will make the clause yield True, as the values are OR'ed together. Only one element of the clause needs to yield True for the clause to become True.

Classically, this problem has an exponential run time and belongs to the class of NP-complete algorithms. In fact, this problem was the first problem that Cook (1971) found to belong to this complexity class. The good news for us quantum programmers is that we can use Grover's algorithm for this problem. Let us see how to do that. In the implementation, we provide an oracle-based solution and a circuit-based solution (we only describe the latter here; it is the more interesting one).

A 3SAT problem is a CNF where each clause has exactly 3 literals, each involving a different variable. For example

$$(x_0 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_0 \vee \neg x_4 \vee x_7).$$

The restriction to 3SAT standardizes the problem somewhat. To keep things simple in our code, we further require that every clause contains all literals. We won't allow a clause like $(x_0 \vee x_1 \vee x_3)$, as it misses the literal x_2. We further simplify and only deal with a single clause, but in exchange, we allow it to have more than three literals. This is meant to simplify the implementation only; it does not restrict the generality of the approach (as we can convert *any* CNF into a 3CNF).

We represent the clause with a simple list of 0s and 1s, where a 1 means that the literal is to be taken, well, literally, while a value of 0 means that the literal should

be negated. In other words, the list `[1, 0, 1]` corresponds to the clause $x_0 \vee \neg x_1 \vee x_2$. The `0` at index 1 indicates that x_1 is to be negated. A formula is then just a list of clauses. In code, to produce random clauses and formulas, we use the function `make_clause`.

PY

Find the code
In file src/sat3.py

```
def make_clause(variables: int):
  return [random.randint(0, 1) for _ in range(variables)]

def make_formula(variables: int, clauses: int):
  return [make_clause(variables) for _ in range(clauses)]
```

To evaluate whether a given list of bits satisfies a given clause, we check for every bit whether it matches the bit in the list representing the clause. This makes it easy to evaluate a complete formula for a given bit string:

```
def eval_formula(bits, clauses: List[List[int]]):
  for clause in clauses:
    res = [bit == clause[idx] for idx, bit in enumerate(bits)]
    if not True in res:
      return False
  return True
```

Another simplification in our approach is that we need to find a negative solution. We want to find a string of bits for which the formula is false. A clause is false if all assigned literals evaluate to false, which, in effect, inverts the clause. Since we only deal with a single clause, only one bit assignment will yield False. There is only one solution and that makes things work nicely with Grover's algorithm, as we shall see shortly. To find negative solutions classically, we use this function:

```
def find_negative_solutions(variables: int, formula):
  for bits in itertools.product([0, 1], repeat=variables):
    res = eval_formula(bits, formula)
    if not res:
      return bits
```

This is all we need for the classical scaffolding code. Now we can move to the quantum part and the implementation of Grover's algorithm for this problem. The circuit is shown in Figure 10.15. We add a helper function for the inversion about the mean operation, as discussed in Section 10.1.9. This is represented by the gate $U_\perp$ and a controlled X gate to the right of state $|\psi_3\rangle$ in the figure. The corresponding code is:

```
def diffuser(qc: circuit.qc, reg, checker, aux):
  qc.h(reg)
  qc.x(reg)
```

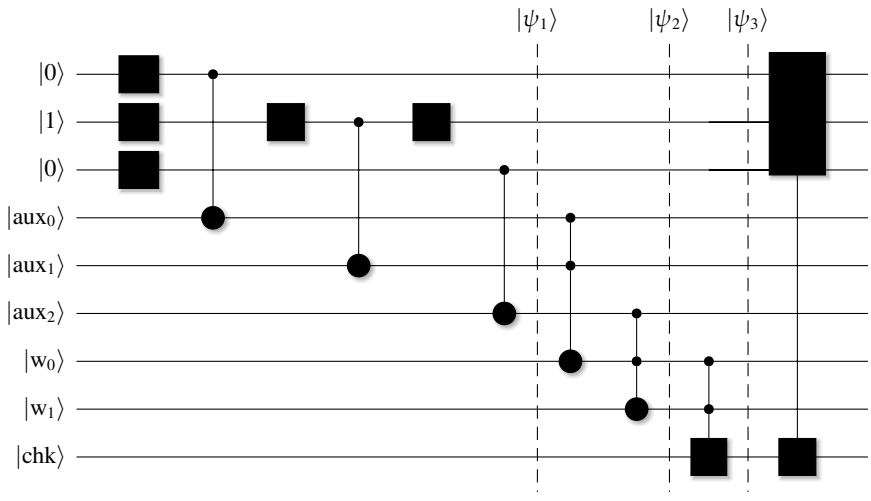


Figure 10.15 Initialization and a single step of the Grover algorithm to compute Boolean satisfiability for the clause $(x_0 \vee \neg x_1 \vee x_2)$. The gate between states $|\psi_2\rangle$ and $|\psi_3\rangle$ represents the phase inversion operator. The operator on the right represents the inversion about the mean operator, which covers only the top three qubits. Gates required for uncomputation are not shown in this figure.

```
qc.multi_control(reg, checker, aux, ops.PauliX(), 'Diffuser Gate')
qc.x(reg)
qc.h(reg)
```

To implement Grover's algorithm for this problem, we first compute the single clause we want to experiment with, find a solution classically, and compute the number of required iterations in the default way by using the number of literals (`variables`):

```
def grover_with_circuit(variables: int = 3):
  formula = make_formula(variables, 1)
  clause = formula[0]
  solution = find_solutions(variables, formula)
  iterations = int(math.pi / 4 * math.sqrt(2**variables))
```

To construct the circuit, we create a register `reg` to hold the initial values of the clause. We create another register `aux` of the same size and copy the original or negated values to it with controlled X gates. To verify the equality of two or more values, we create another register `w` and use a cascade of Toffoli gates to accumulate the final result in a single qubit register `chk`.

```
  qc = circuit.qc('Outer')
  reg = qc.reg(variables, 0)
  aux = qc.reg(variables, 0)  # can be optimized away.
  w = qc.reg(variables - 1, 0)
  chk = qc.reg(1, 0)[0]
```

In code, following Grover's algorithm, we apply Hadamard gates to the initial register `reg`. Then we iterate `iterations` times and construct a subcircuit `cc` in each iteration. We want to find an assignment to the clause that returns false. De Morgan's law tells us that to negate a clause, we have to negate each literal individually and change the logical OR to a logical AND:

$$\neg(x \vee y \vee z) = \neg x \wedge \neg y \wedge \neg z.$$

If a literal is already negated, we leave it alone and copy the corresponding qubit from `reg` to `aux` with a controlled X gate. If it still needs to be negated, we negate it explicitly by bracketing the controlled X gates with single X gates before and after. This corresponds to the circuit to the left of state $|\psi_1\rangle$ in Figure 10.15. Using the subcircuit (`cc`) makes the uncomputation below quite convenient:

```
qc.h(reg)
for _ in range(iterations):
  cc = circuit.qc('Gates', eager=False)

  # First we negate each literal if it was not already negated.
  for idx in range(variables):
    if clause[idx] == 1:
      cc.x(reg[idx])
      cc.cx(reg[idx], aux[idx])
      cc.x(reg[idx])
    else:
      cc.cx(reg[idx], aux[idx])
```

To compute the logical AND between qubits, we compute a cascade of Toffoli gates, using the `w` register to store temporary intermediate comparison results in the w_i ancilla qubits. This cascade is shown in Figure 10.15 as the gates between the states $|\psi_1\rangle$ and $|\psi_2\rangle$.

```
  cc.toffoli(aux[0], aux[1], w[0])
  for idx in range(2, variables):
    cc.toffoli(aux[idx], w[idx - 2], w[idx - 1])
```

Finally, we link this subcircuit (`cc`) to the main circuit. Then we use a controlled Z gate to export the final result to the `chk` register. The gate between states $|\psi_2\rangle$ and $|\psi_3\rangle$ corresponds to the phase-inversion operator outlined in Section 10.1.8.

We have to uncompute the subcircuit, which we can do quite conveniently with the `inverse` function of the subcircuit (the uncomputation gates are not shown in the figure). As a final step, we add the mean-inversion circuit from above. Note that this diffuser operator only connects the three input qubits and the final result qubit.

```
  # Add and execute the subcircuit.
  qc.qc(cc)
  # Phase inversion - connect the result to the chk qubit.
```

```
    qc.cz(w[idx - 1], chk)
    # Uncompute the subcircuit.
    qc.qc(cc.inverse())
    # Mean inversion.
    diffuser(qc, reg, chk, aux)
```

This completes the implementation of the Grover algorithm for this problem. All that is left to do is to find the state with the highest probability and compare it with the expected results.

```
  maxbits, maxprob = qc.psi.maxprob()
  print(f'Circuit: Want: {list(solution[0])}, ', end='')
  print(f'Got: {list(maxbits[:variables])}, p: {maxprob:.2f}')
  assert solution[0]) == maxbits[:variables], 'Incorrect Result'
```

Lastly, we perform experiments with clauses of varying length and verify that this works as expected:

```
def main(argv):
  for variables in range(4, 7):
      grover_with_circuit(variables)
>>
Circuit: Want: [1, 1, 1], Got: [1, 1, 1], p: 0.44
Circuit: Want: [0, 1, 1], Got: [0, 1, 1], p: 0.44
Circuit: Want: [0, 1, 0], Got: [0, 1, 0], p: 0.44
Circuit: Want: [1, 1, 0, 0], Got: [1, 1, 0, 0], p: 0.38
Circuit: Want: [1, 1, 1, 1, 0], Got: [1, 1, 1, 1, 0], p: 0.35
Circuit: Want: [0, 0, 0, 1, 0, 1], Got: [0, 0, 0, 1, 0, 1], p: 0.32
```

10.6 Graph Coloring

Graph coloring is the problem of assigning labels to vertices in a graph such that no pair of vertices connected by an edge have the same label. Traditionally, those labels are called "colors." In our case, we use integers to represent colors. For example, in Figure 10.16, you can see that only two colors are needed to color a line graph with only one edge or a rectangular graph. For a triangle, on the other hand, you need three colors.

If a maximum of k colors is required to color a graph, we call that *k-coloring*. Graph coloring has many applications, such as map coloring or register allocation in a compiler (Briggs, 1992). In the general case, the complexity of this problem appears to be $\mathcal{O}\left(2.4423^n\right)$ in time and space for a graph with n vertices (Lawler, 1976). Better-performing solutions have been found for special types of graphs, such as constraint graphs (Wikipedia, 2021e).

Let us see how we can use Grover's algorithm for this problem. To simplify the code, we turn the problem on its head and find solutions where all vertices have the *same* color. This makes the code less complex without limiting generality.

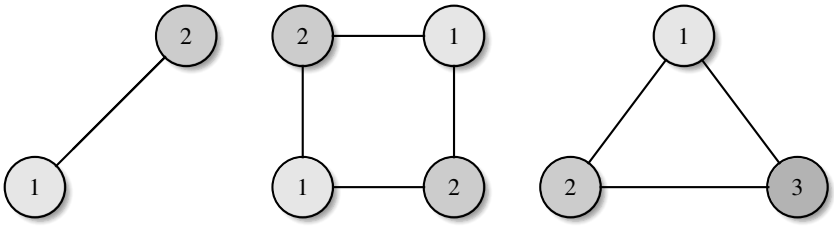


Figure 10.16 Coloring of graphs. A line and a rectangle only need two colors, while a triangle needs three.

We use basis encoding to represent vertices and their colors. For example, to represent up to four colors, we will need two qubits to represent the coloring of a single vertex. In general, if we have a graph with n vertices that require m colors, we will need $n\lceil\log_2(m)\rceil$ qubits (rounded up to the next integer). For example, for a graph with four nodes and four colors, we can represent its state with eight qubits as follows, where each vertex is represented by the two qubits specifying one of four possible colors. The first two qubits, representing vertex v_0, have color c_0 or binary `0b00`. Vertex v_1 has color c_2 or binary `0b10`. Vertex v_2 has color c_3 (`0b11`), and vertex v_3 has color c_1 (`0b01`):

$$|\underbrace{0\ 0}_{v_0:c_0}\ \underbrace{1\ 0}_{v_1:c_2}\ \underbrace{1\ 1}_{v_2:c_3}\ \underbrace{0\ 1}_{v_3:c_1}\rangle.$$

To define a graph in code, we build a simple `Graph` data structure. The edges of the graph are tuples of two integers representing the *from* and *to* vertices.[8] We add a member function to verify whether or not all colors in the state representing the graph are equal. We restrict ourselves to four colors, represented by two qubits per color.

PY

Find the code
In file src/graph_coloring.py

```
class Graph:
  def __init__(
      self, num_vertices: int, desc: str, edges: List[Tuple[int, int]]
  ):
    self.num = num_vertices
    self.edges = edges
    self.desc = desc

  def verify(self, bits, n: int = 2):
    for edge in self.edges:
      if (bits[edge[0] * n: edge[0] * n + n] !=
          bits[edge[1] * n: edge[1] * n + n]):
        return True  # different colors!
    return False  # all colors are the same.
```

[8] We use these terms even though the edges are not directed.

Table 10.1. Truth table for the controlled X gate. $|\phi\rangle$ is $|0\rangle$ for identical inputs.

$\|\psi\rangle$	$\|\phi\rangle$	$\rightarrow \|\phi\rangle$
$\|0\rangle$	$\|0\rangle$	$\|0\rangle$
$\|0\rangle$	$\|1\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|1\rangle$	$\|0\rangle$

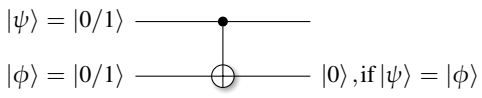


Figure 10.17 A single controlled X gate with inputs of $|0\rangle$ or $|1\rangle$. If $|\psi\rangle$ and $|\phi\rangle$ are both $|0\rangle$ or both $|1\rangle$, we will measure $|0\rangle$ on qubit $|\phi\rangle$.

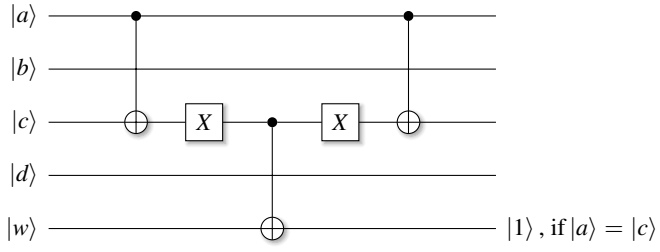


Figure 10.18 Check two qubits a and c for equality with a controlled Not gate. If they are equal, qubit c will be in state $|0\rangle$. We use a Controlled-by-0 X gate from c to w to record this result as a state $|1\rangle$ in the ancilla w. The final controlled X gate is for uncomputation.

To compare qubits, let's look at the truth table of the controlled X gate in Figure 10.17. As you can see in Table 10.1, the second qubit ϕ will be in state $|0\rangle$ if the input qubits both are in the same state $|0\rangle$ or $|1\rangle$. In larger circuits, it is always good practice to avoid entangling ancillary qubits. In order to compare two qubits, we construct the circuit in Figure 10.18, which includes the uncomputation, and record the result in an ancilla qubit. This scheme can easily be extended to compare arbitrary tuples of qubits for equality, or even inequality, with just minor changes. Since we restrict ourselves to only four vertices and colors, we must compare pairs of qubits. For this, we build the circuit in Figure 10.19.

In code, we add the identical `diffuser` helper function for the standard mean inversion operator from Section 10.5. To implement the circuit shown in Figure 10.19 to compare pairs of qubits, we add this function:

```
def compare_pairs(qc, a, b, c, d, w0, w1, chk):
  qc.cx(a, c)
  qc.cx0(c, w0)
  qc.cx(b, d)
```

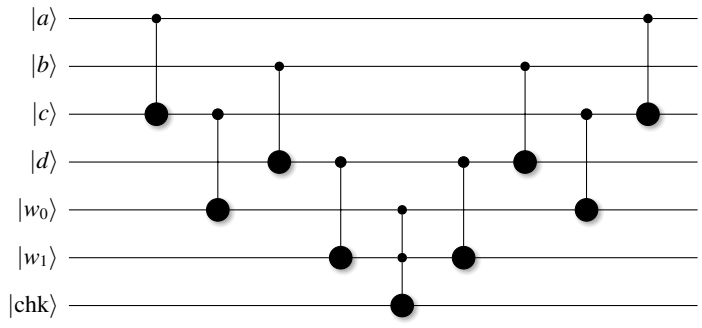


Figure 10.19 Compare pairs of qubits $(a,b) = (c,d)$ for equality and record the result in the ancillary qubit $|\text{chk}\rangle$. The first two gates compare $|a\rangle$ and $|c\rangle$ and record the result in $|w_0\rangle$. Gates three and four compare $|b\rangle$ and $|d\rangle$ and record the result in $|w_1\rangle$. The double-controlled Not gate then compares $|w_0\rangle$ and $|w_1\rangle$ and records the result in $|\text{chk}\rangle$. The remaining gates are for uncomputation.

```
qc.cx0(d, w1)

qc.ccx(w0, w1, chk)

qc.cx0(d, w1)
qc.cx(b, d)
qc.cx0(c, w0)
qc.cx(a, c)
```

With these preliminaries in place, we can now implement the whole circuit. To represent all vertices, we create the register `reg` with two qubits per vertice. We use the register `chk` to keep all the intermediate comparison results. A multi-controlled gate over this complete register will place the final result in the register `res`. We will have to connect `chk` with `res` in the phase inversion and `reg` with `res` in the mean inversion. For proper uncomputation, we utilize a subcircuit in the construction below.

```
def build_circuit(g: Graph):
  qc = circuit.qc('Graph Circuit')
  reg = qc.reg(g.num * 2)
  chk = qc.reg(len(g.edges))
  res = qc.reg(1)[0]
  tmp = qc.reg(g.num * 2 - 1)

  iterations = int(math.pi / 4 * math.sqrt(2**(g.num * 2)))
  qc.h(reg)
  for _ in range(iterations):
    sc = qc.sub()
    for idx, edge in enumerate(g.edges):
      fr = edge[0] * 2
      to = edge[1] * 2
      compare_pairs(sc, fr, fr + 1, to, to + 1, tmp[0], tmp[1], chk[idx])
```

```
    # Phase inversion.
    qc.qc(sc)
    qc.multi_control(chk, res, tmp, ops.PauliZ(), 'multi')  # (!)
    qc.qc(sc.inverse())

    # Mean inversion.
    diffuser(qc, reg, res, tmp)
```

To check the results, we test each basis state with nonzero probability. Since we want all colors to be the same, we require that all pairs of qubits be the same. We should find exactly four solutions for each input problem, one for each of the color combinations $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$.

```
  # results, show all states with nonzero probability.
  _, maxprob = qc.psi.maxprob()
  for idx, val in enumerate(qc.psi):
    if np.real(val.conj() * val) > (maxprob - 0.005):
      bits = helper.val2bits(idx, qc.nbits)
      print('  Color:', bits[0 : g.num * 2])
      assert not g.verify(bits), 'Incorrect color assignment found.'
```

After that, we construct a few simple graph shapes and verify that the implementation works as expected.

```
def main(argv):
  print("Graph coloring via Grover's Search. ", end='')
  print('Find identical colors (2 qubits each).')
  build_circuit(Graph(2, 'simple line', [(0, 1)]))
  build_circuit(Graph(3, 'simple triangle', [(0, 1), (1, 2), (2, 0)]))
  build_circuit(Graph(4, 'star formation', [(0, 1), (0, 2), (0, 3)]))
  build_circuit(Graph(4, 'rectangle', [(0, 1), (1, 2), (2, 3)]))

>>
Solving [simple line]: 2 vertices, 1 edges -> 9 qubits
  Color: [0, 0, 0, 0]
  Color: [0, 1, 0, 1]
  Color: [1, 0, 1, 0]
  Color: [1, 1, 1, 1]
Solving [simple triangle]: 3 vertices, 3 edges -> 15 qubits
  Color: [0, 0, 0, 0, 0, 0]
  Color: [0, 1, 0, 1, 0, 1]
  Color: [1, 0, 1, 0, 1, 0]
  Color: [1, 1, 1, 1, 1, 1]
[...]
```

10.7 Quantum Mean Estimation

Grover's algorithm can find the mean of a set of values, as proposed in Terhal (1999). A more straightforward way was described by Mosca (2008), which is the approach we will discuss here. We assume that we have $N = 2^n$ values (represented by n qubits), for which we want to calculate the mean. If the values do not sum up to 1.0, we must normalize them. For an example vector of $\begin{pmatrix}1 & 3 & 6 & 7\end{pmatrix}$, we use the L_2 norm and divide the vector by its norm to get normalized vector x_n and mean $\overline{x_n}$ as

$$x_n = \frac{\begin{pmatrix}1 & 3 & 6 & 7\end{pmatrix}}{|x|} = \begin{pmatrix}0.103 & 0.308 & 0.616 & 0.718\end{pmatrix},$$
$$\overline{x_n} = 0.436.$$

To calculate the original mean, we multiply the normalized mean by the norm. We define the functions $f(i)$ to return the ith original value and $F(i)$ to return the normalized value at index i. Recall from Section 9.1.3 that with a rotation about the y-axis by an angle $\theta = 2\arcsin(\alpha)$, we can transform the basis state $|0\rangle$ into

$$R_y(\theta)|0\rangle = \sqrt{1-\alpha^2}\,|0\rangle + \alpha\,|1\rangle.$$

For more than one qubit, the "trick" for quantum mean estimation is to construct a circuit that applies specific controlled rotations to each basis state $|i\rangle$ in equal superposition. We construct a circuit implementing a unitary U_a such that

$$U_a : |0\rangle^{\otimes n}|0\rangle \mapsto \frac{1}{2^{n/2}}\sum_i |i\rangle\left(\sqrt{1-F(i)^2}|0\rangle + F(i)|1\rangle\right),$$

with $|i\rangle$ as the basis state representing the index of the ith value in binary. For example, $F(|011\rangle)$ would return the normalized value at index 3 (`0b011`).

For two qubits, the circuit looks like the one in Figure 10.20. First, we put the state in an equal superposition with Hadamard gates. We use multi-controlled rotation gates to rotate each basis state $|i\rangle$ in binary form by an angle $\theta_i = 2\arcsin(F(i))$. For a 0 in the binary representation of $|i\rangle$, we use a Controlled-by-0 rotation gate. For a 1, we use a Controlled-by-1 gate. For two qubits, there are $2^2 = 4$ bit patterns or four multi-controlled rotation gates. After the rotations, we get out of the superposition with another set of Hadamard gates. The resulting amplitude of measuring a $|1\rangle$ on

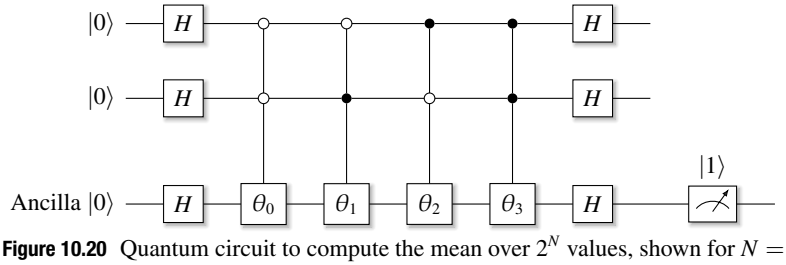


Figure 10.20 Quantum circuit to compute the mean over 2^N values, shown for $N = 2$.

the ancilla qubit will then accumulate to the mean μ with

$$\sum_i \frac{F(i)}{2^{n/2}} = \mu.$$

The reason for the normalization by $2^{n/2}$ is related to the distribution of the amplitudes across the states in superposition. Since the initial Hadamard gates create 2^n states with the same amplitude $\frac{1}{2^{n/2}}$, this factor appears in the calculation of the final mean.

Let us explore this in code. In each experiment, we create a random array of `nbits` positive and negative values and store the normalized vector in variable `xn`. We construct a state consisting of input qubits, auxiliary ancilla qubits for the multi-controlled rotations, and the extra ancilla for the rotations.

Find the code
In file src/quantum_mean.py

```
def run_experiment(nbits: int):
  x = np.array([random.randint(0, 10) - 5 for _ in range(2**nbits)])
  xn = x / np.linalg.norm(x)

  qc = circuit.qc('mean calculator')
  inp = qc.reg(nbits, 0)       # State input.
  aux = qc.reg(nbits - 1, 0)   # Aux qubits for multi-contrl. gates.
  ext = qc.reg(1, 0)           # Target 'extra' qubit.
```

Next, we apply the rotation gates using the bit patterns of the binary representations of the basis states $|i\rangle$ to determine whether to use a Controlled-by-0 or Controlled-by-1 gate. We use these bits to multi-control a y-rotation by the values stored in `xn`, as explained above. We bracket the whole circuit with Hadamard gates:

```
  qc.h(inp)
  for bits in itertools.product([0, 1], repeat=nbits):
    idx = helper.bits2val(bits)
    # Control-by-zero is indicated with a single-element list.
    ctl = [i if bit == 1 else [i] for i, bit in enumerate(bits)]
    qc.multi_control(ctl, ext, aux,
                     ops.RotationY(2 * np.arcsin(xn[idx])))
  qc.h(inp)
```

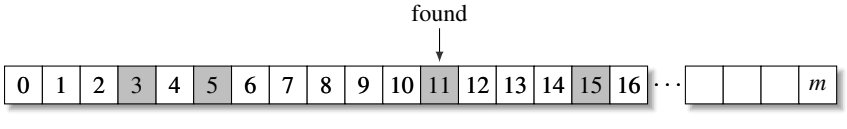
Now, recall that the state vector for $|00\ldots1\rangle$ holds a single 1 at index 1 and 0s everywhere else. In other words, we can obtain the probability amplitude for this state by looking at index 1 in the state vector. To obtain the original classical mean `qclas`, we multiply this value by the vector norm, undoing the above normalization. In the code below, we calculate these values, assert that they are equal, and run a few experiments over a range of numbers of qubits:

```
  qmean = np.real(qc.psi[1])
  qclas = np.real(qc.psi[1] * np.linalg.norm(x))

  # Check results.
  assert np.allclose(np.mean(xn), qmean, atol=0.001), 'Whaaa'
  assert np.allclose(np.mean(x), qclas, atol=0.001), 'Whaaa'
  print(f'Mean ({nbits} qb): classic: {np.mean(x):.3f}, q: {qclas:.3f}')

def main(argv):
  for nbits in range(2, 8):
    run_experiment(nbits)
>>
Mean (2 qb): classic: 0.500, q: 0.500
Mean (3 qb): classic: -1.125, q: -1.125
Mean (4 qb): classic: -1.688, q: -1.688
Mean (5 qb): classic: -0.500, q: -0.500
Mean (6 qb): classic: 0.172, q: 0.172
Mean (7 qb): classic: 0.227, q: 0.227
```

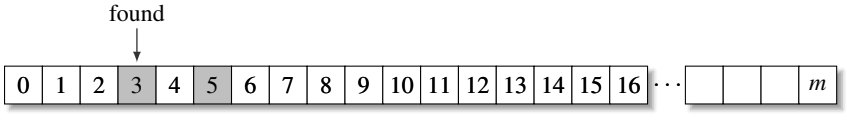
10.8 Quantum Minimum Finding

We discussed black-box algorithms in Chapter 8. Sometimes, these algorithms appear to push the reasoning about their quantum advantage a bit too far. A good example may be the often-cited *quantum minimum finding* algorithm published by Durr and Hoyer (1999). Since it utilizes Grover's algorithm at its core, we discuss it here with focus on a few important aspects of this algorithm.

To establish a quantum advantage, the algorithm makes several assumptions and simplifications that may be impossible to implement on a real machine. However, as long as we just *pretend* that some of the assumptions can be implemented at zero cost, the algorithm does have a quantum advantage. The basic algorithm goes as follows, with a short example shown in Figure 10.21.



(a) In the range of $0 \ldots m$, we want to find the minimum of the four elements marked in gray. In the example, QAA found element 11.



(b) In a second step, only elements smaller than 11 are marked. In the example, QAA found element 3, which is the minimum.

Figure 10.21 An example of the quantum minimum finding algorithm.

1. First, we need to define the set of numbers for which we want to find the minimum. In code, we create a set of `num_vals` random numbers in a given range with function `get_distro`.

Find the code
In file src/minimum_finding.py

```
def get_distro(min_value: int, max_value: int, num_vals: int):
  return sorted(np.random.choice(
                    np.arange(min_value, max_value), num_vals))
```

We should think of the whole range of numbers as the available state space and the generated list of numbers as the special elements or the set of solutions S.

2. The goal is to find the smallest number in the set of special elements. We will use Grover's algorithm to achieve this goal. The number of elements in S should be small compared to the whole range of numbers to ensure that we find a solution with reasonably high probability. In the example in the figure, the range of numbers goes from 0 to a maximum number m, all represented by basis states. We only mark the four elements 3, 4, 11, and 15.

   Similarly to the regular Grover algorithm, we construct a function in `make_f` that returns a 1 for every marked number in the set. However, there is a twist: We mark a value only if it is *smaller* than a given value M. This is easy to do in code[9] and perhaps in a thought experiment. However, this may be impossible to implement efficiently on a quantum computer. Again, here we pretend it would be possible at zero cost. Note that we already know the minimum classically as we construct the oracle and mark the values. Note that as we construct the oracle and mark the values, we already know the minimum classically. In the example in Figure 10.21(a), since we know that 15 is the largest value M, we would start by picking $M + 1$ as the upper bound.

```
def make_f(d: int, numbers: List[int], max_value: int):
  num_inputs = 2**d
  answers = np.zeros(num_inputs, dtype=np.int8)
  answers[[i for i in numbers if i < max_value]] = 1
  return lambda bits: answers[helper.bits2val(bits)]
```

3. The third step is now identical to what is found in Grover's algorithm. After performing this algorithm, the marked special elements will have a higher amplitude than the remaining elements in the list. We use this fact to measure and find one of the marked values, which are all smaller than a given M.
4. In terms of measurement, we simulate measurement by simply picking one of the found elements. All marked elements have a higher probability than the other

[9] Note the clever loop comprehension used as index expression.

states, so this approach is reasonably legitimate. In the example in the figure, element 11 was found in the first iteration.

```
new_max = np.random.choice(results)
print(' -> New Max:', new_max)
result = f(helper.val2bits(new_max, nbits))
assert result == 1, 'something went wrong, measured invalid state'
return new_max
```

5. Now go back to step 2 and use the newly found upper bound 11 as the new maximum M to mark the remaining special elements. In the example, we find element 3 in the next step. Just by chance, this value is also the minimum.
6. As soon as we hit the smallest value, we are done. We have found the smallest special element.

There are concerns with this approach and its claims of a quantum advantage:

- The oracle has to be constructed somehow. However, the paper assumes that this construction is free and has zero cost. This point is moot if we only consider query complexity.
- We terminate after finding the smallest number, but this requires knowing the smallest number. Alternatively, we run Grover's algorithm multiple times. If, at a given step, Grover fails to find elements smaller than the current limit M, we may have found the smallest element.
- We must know how many numbers remain below a current maximum to adjust the Grover iteration number (even though there may be ways around this).

The last two points could be solved via quantum counting or perhaps other mechanisms to find when there is no more solution to be marked. However, taken together, this algorithm makes several critical assumptions. If all assumptions hold, the algorithm would indeed demonstrate a quantum speed-up because of the use of Grover's algorithm. However, this author happens to believe that this algorithm, as stated, is not implementable in a practical or efficient way.[10] Depending on the use case, combining classical and quantum computations may solve this particular problem in a more assumption-free way.

10.9 Quantum Median Estimation

We can combine the estimation of the quantum mean and quantum minimum finding to determine the *median* of an ensemble of data points (Brassard et al., 2011). However, let us be cautious; besides the assumptions from the quantum algorithm for minimum finding, this algorithm makes further assumptions that may not be feasible to implement efficiently on a quantum computer.

[10] Stated with the caveat that, as usual, I might be wrong.

The median z is the one value in an ensemble of n values for which the distance to all other points in the set is the minimum. With a distance function $\text{dist}(x,x_i)$ between an individual value x and the values x_i at index i, we can write z in closed form as

$$z = \min_{x \in \{0,\ldots,x_{n-1}\}} \sum_{i=0}^{n-1} \text{dist}(x,x_i).$$

To implement a feasibility study of this algorithm, we first create an array of random, normalized values in the range of $\{0,\ldots,2^n-1\}$ for n qubits.

Find the code
In file src/quantum_median.py

```
def run_experiment(nbits: int):
  x = np.array([random.randint(0, 2**nbits) for _ in range(2**nbits)])
  xn = x / np.linalg.norm(x)
```

We then iterate over all the n individual values x_i in the `xn` array and compute the n difference vectors $\vec{d}_i$ to x_i, where the element d_{ij} at index j will be the difference of element d_{ij} and x_i:

$$\vec{d}_i = \big(|x_0 - x_i|,\ |x_1 - x_i|,\ \ldots\ ,|x_{n-1} - x_i|\big).$$

In programming terms, you can think of this as a two-deep nested loop:

```
  for idx, z in enumerate(xn):
    diff = [abs(xval - z) for xval in xn]
```

It is not clear how to compute the difference vector efficiently in the quantum domain. It may be one of those cases where we assume it would be easily doable and don't really care or because we are only interested in the query complexity.

Assuming that we can use a quantum algorithm to compute the mean, as described above, we compute the mean of this vector and store the result for each value x_i. In a quantum algorithm, we would store all the mean values in a new vector and use the quantum minimum finding algorithm to find the smallest value.

In the code, we do not implement full quantum techniques to find the mean and minimum. For the minimum specifically, we just maintain the smallest value as we iterate over the elements in `xn`:

```
    # Normalization (required for quantum, also improves
    # accuracy by an order of magnitude).
    diffnorm = np.linalg.norm(diff)

    # Compute the mean (which we know how to do quantumly)
    # for the element xn[idx].
    mean = np.mean(diffnorm)
```

```
    # Find the minimum (which we would also know how to do quantumly).
    # Here, classically, maintain the index of the smallest element.
    if mean < min_mean:
      min_mean = mean
      median = idx
```

Finally, we compare the results and check for correctness, which should produce an output like the one shown below. The quantum results are integer indices:

```
  print(
      f' Median ({nbits} qb): Classic: {np.mean(x):.3f},'
      f' Quantum: {x[median]:.0f}'
  )
  if max(np.mean(x), x[median]) / min(np.mean(x), x[median]) > 1.02:
    raise AssertionError('Incorrect median computation.')
>>
Classic Sim of Quantum Median Computation.
 Median (10 qb): Classic: 525.596, Quantum: 525
 Median (10 qb): Classic: 487.644, Quantum: 487
 Median (10 qb): Classic: 517.676, Quantum: 519
[...]
```

11 Algorithms Using Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is a cornerstone of several quantum algorithms because it leverages the principles of superposition and entanglement to perform a Fourier transform on quantum data exponentially faster than classical algorithms. This speed-up is crucial for algorithms like Shor's algorithm, which uses the QFT to factor large numbers efficiently, potentially breaking widely used encryption schemes. Essentially, the QFT allows quantum computers to analyze the periodic patterns in quantum states, revealing hidden information that would be intractable to find classically. This ability to efficiently extract information from quantum systems makes the QFT a fundamental tool for unlocking the power of quantum computation.

We start by describing the almost trivial *phase kick* mechanism that transfers a phase from a controlled qubit to its controller. This mechanism is the basis for *quantum phase estimation* (QPE), another fundamental quantum algorithm with broad applications. It efficiently estimates the eigenvalues for known eigenvectors of a unitary operator. This ability is used in numerous quantum algorithms, including Shor's algorithm, quantum simulations for material science, and quantum chemistry.

The QPE and QFT are inextricably linked, and one cannot be discussed without the other. We first describe the QPE and make forward references to the QFT.

We follow these fundamental algorithms with a few applications. First, we explain a cute way to estimate π. Then we describe the arithmetic operations of addition and multiplication in the Fourier domain. With these tools in place, we finally describe Shor's famous algorithm for number factorization. It is a beautiful but complex algorithm and one of the main reasons for the excitement in quantum computing.

11.1 Phase-Kick Circuit

First, we shall explore the *phase-kick* mechanism, which is the basis for algorithms such as quantum phase estimation and the quantum Fourier transform, as we will see shortly. Let us examine the simple phase-kick circuit in Figure 11.1. Why is this circuit called a phase *kick* circuit? The state $|\psi_1\rangle$ after the Hadamard gate is

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

After the controlled S operation, where only the $|1\rangle$ part of the state of the first qubit triggers the S gate, the state $|\psi_2\rangle$ at the end of the phase-kick circuit is

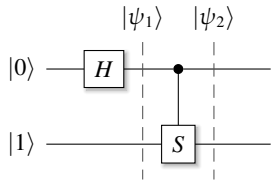


Figure 11.1 A simple *phase-kick* circuit.

$$\begin{aligned}|\psi_2\rangle &= \frac{1}{\sqrt{2}}\big(|0\rangle|1\rangle + |1\rangle S|1\rangle\big) \\ &= \frac{1}{\sqrt{2}}\big(|0\rangle|1\rangle + |1\rangle e^{i\pi/2}|1\rangle\big) \\ &= \frac{1}{\sqrt{2}}\big(|0\rangle + e^{i\pi/2}|1\rangle\big)\ |1\rangle\,.\end{aligned}$$

You can see that the second qubit remains *unmodified* as $|1\rangle$. The trick is that $|1\rangle$ is an eigenstate of the S gate, which allows us to *kick* the phase from one qubit to another via controlled rotations. We will further elaborate on this in Section 11.2.1 on phase estimation.

This mechanism also enabled the Bernstein–Vazirani algorithm, covered in Section 8.1. We did not use rotation gates in that implementation, but rather controlled Not gates on states in the Hadamard basis. But in this basis, a controlled Not gate corresponds to a simple Z gate (see also Section 16.4.5), which manifests as a 180° rotation about the z-axis.

Controlled rotation gates have the nice property that they can be used in an additive fashion. An example of this basic principle is shown in Figure 11.2. Here, the two top qubits are initialized as $|0\rangle$ and placed in superposition with Hadamard gates. A bottom ancilla qubit starts in state $|1\rangle$. We apply the controlled S gate from the top qubit to the ancilla, controlling a rotation by 90°. The T gate on the ancilla is controlled by the middle qubit, which controls a rotation by 45°. Recall that these gates only add a phase to the $|1\rangle$ component of a state:

$$S\big(\alpha\,|0\rangle + \beta\,|1\rangle\big) = \begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}\begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \begin{pmatrix}\alpha\\ i\beta\end{pmatrix} = \alpha\,|0\rangle + i\beta\,|1\rangle\,.$$

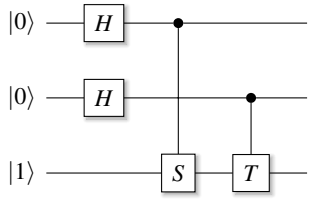


Figure 11.2 A two-qubit phase-kick circuit. The individual phases add up.

PY

Find the code
In file src/phase_kick.py

This is easy to implement with just a few lines of code.

```
psi = state.bitstring(0, 0, 1)
psi = ops.Hadamard(2)(psi)
psi = ops.ControlledU(0, 2, ops.Sgate())(psi)
psi = ops.ControlledU(1, 2, ops.Tgate())(psi, 1)
psi.dump()
```

Due to superposition, the $|1\rangle$ part of each of the top two qubits triggers the rotation of a controlled gate and adds a local phase to the controlling qubit. Having the top qubit as $|1\rangle$ adds 90°, and the $|1\rangle$ part of the middle qubit adds another 45°. Here we restrict ourselves to S and T gates, but we could use arbitrary rotations about the z-axis. We can use this type of circuit to express additions or subtractions in terms of phases as long as we normalize to 2π and avoid overflows. For the example:

- The rotation of 180° as a fraction of 2π is $e^{2\pi i/2^1} = e^{\pi i}$. Expressed as a phase angle, this is -1.
- A rotation of 90° as a fraction of 2π is $e^{2\pi i/2^2} = e^{\pi i/2}$, a phase of i.
- The rotation of 45° as a fraction of 2π is $e^{2\pi i/2^3} = e^{\pi i/4}$.
- Finally, a rotation by $135° = 90° + 45°$ as a fraction of 2π is $e^{2\pi i(1/2^2+1/2^3)}$.

The resulting probability amplitudes and phases will be as follows:

```
|001> (|1>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|011> (|3>):  ampl: +0.35+0.35j prob: 0.25 Phase:  45.0
|101> (|5>):  ampl: +0.00+0.50j prob: 0.25 Phase:  90.0
|111> (|7>):  ampl: -0.35+0.35j prob: 0.25 Phase: 135.0
```

This ability to add phases in a controlled fashion is a powerful mechanism and the foundation of the quantum Fourier transformation, which we will explore shortly. The detailed math for two or more qubits is more challenging, we detail it in Section 11.2.2 below.

11.2 Quantum Phase Estimation (QPE)

Quantum phase estimation (QPE) (Kitaev, 1995) is a key building block for several advanced algorithms. It allows finding the unknown eigenvalues for known eigenvectors of a given operator U. After going through the QPE circuit, the state will be in a form closely related to the output of the quantum Fourier transform (QFT), which we discuss in Section 11.4. In other words, QPE and QFT are inextricably linked, and one cannot be discussed without the other. We start by discussing the QPE, but we must

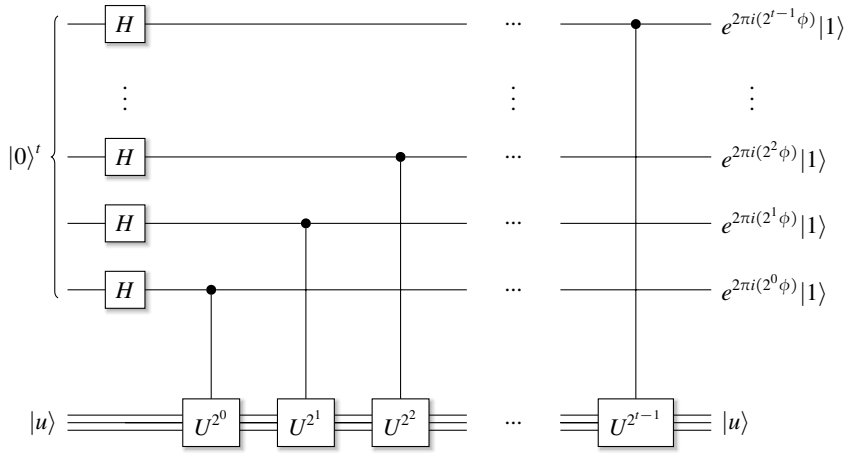


Figure 11.3 The first part of the phase estimation circuit.

make a few forward references to QFT. Once you are through with the section on the QFT, you may want to refer back to this section and fill in the blanks.

11.2.1 Phase Estimation

For a given unitary matrix U with a known eigenvector $|u\rangle$ and corresponding unknown eigenvalue $e^{2\pi i\phi}$, the quantum phase estimation (QPE) aims to estimate the value of ϕ using a two-step procedure:[1]

1. Basis encode the unknown phase using a circuit that produces a result identical to the result of a QFT, which we will discuss in Section 11.4 below.
2. Apply the adjoint $QFT^\dagger$ operator to estimate the phase ϕ as a binary fraction.

In this section, we focus on the first task. To start, we define a result register with t qubits representing the bits in a binary fraction, where t is determined by the precision we want to achieve. The more qubits, the more fine-grained fractions of powers of 2 will approximate the final result. We initialize the t qubits in the first register with $|0\rangle$ and put them in equal superposition with Hadamard gates, as shown on the left of Figure 11.3.

We add a second register representing the known eigenvector $|u\rangle$. We connect the top result register with controlled gates to a sequence of t instances of U, each taken to increasing powers of 2 $(U, U^2, \ldots, U^{t-1})$, which we apply to the second register.

To achieve the powers of 2, we multiply U by itself and accumulate the results. For example, we start with U itself, as $U_0 = U_0^{2^0} = U_0$. Next, we compute $U_1 = U_0U_0 = U_0^{2^1}$, the U_1U_1 to get $U_2 = U_1U_1 = U_0^{2^2}$, and so on. The goal is to accumulate

[1] The literature distinguishes between *quantum phase estimation* and *quantum eigenvalue estimation*. The eigenvalues of a unitary operator have unit modulus and are characterized by their phase. Hence, for unitary operators, the algorithm can be used for both. Here, we interpret it as the estimation of an eigenvalue, as that seems to be the majority opinion.

phase information by repeatedly applying these unitaries to the eigenvector $|u\rangle$. The entire procedure is shown in Figure 11.3 in circuit notation, and the math is detailed in Section 11.2.2 below.

The first question to ask may be why $|u\rangle$ has to be initialized as an eigenvector. Wouldn't this procedure work for any normalized state vector $|x\rangle$? The answer is no. For one or more applications of a unitary U, the eigenvalue equation only holds for eigenvectors:

$$U^n|u\rangle = \lambda^n|u\rangle.$$

This means we can apply U and any power of U to $|u\rangle$ as often as we want. Since $|u\rangle$ is an eigenvector, it will only be scaled by a number, the corresponding power of the complex eigenvalue λ, which has a modulus of 1. Moreover, since we are using controlled gates, "some" information will be transferred to the controllers. We develop all the details in Section 11.2.2. If you are not interested in the math, jump to Section 11.2.3 on the implementation.

11.2.2 Detailed Derivation

First, recall from Section 1.8 that the eigenvalues of a unitary matrix have a modulus of 1. Since $|\lambda| = 1$, we can write an eigenvalue as

$$\lambda_i = e^{2\pi i\phi},$$

with ϕ being a factor between 0 and 1. In Section 2.4.3, we used the following notation for binary fractions with t bits of resolution and the binary bits ϕ_i having values 0 or 1:

$$\begin{aligned}\phi &= 0.\phi_0\phi_1\ldots\phi_{t-1}\\ &= \phi_0\frac{1}{2^1} + \phi_1\frac{1}{2^2} + \cdots + \phi_{t-1}\frac{1}{2^t}.\end{aligned}$$

For example, a binary number written as 0.1101 has the decimal value of

$$0.1101 = 1\frac{1}{2} + 1\frac{1}{4} + 0\frac{1}{8} + 1\frac{1}{16} = 0.8125\ .$$

With these preliminaries, let us see what happens to the state in Figure 11.4, which is a first small part of the phase estimation circuit. The state $|u\rangle$ may consist of multiple qubits and must be an eigenstate of U. You may notice the similarity to the phase kick circuit we discussed in Section 11.1. This circuit has only one qubit at the top, which limits the precision of the approximated eigenvalue of U to a single fractional

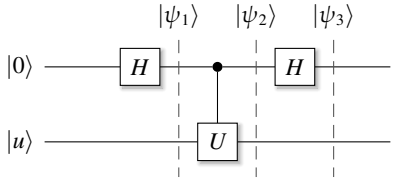


Figure 11.4 An initial phase estimation circuit with a single $U = U^{2^0}$ operator.

bit. However, once we understand how this works for a single fractional bit, we can expand to two bits and generalize.

Let us start and assume U has the eigenvalue $\lambda = e^{2\pi i 0.\phi_0}$, with the term $0.\phi_0$ representing a single binary fractional bit with the value of $1/2 = 2^{-1}$. With this, the estimated phase can only be calculated as having a value of 0 or 0.5. The state $|\psi_1\rangle$ after the first Hadamard gate is:

$$|\psi_1\rangle = |+\rangle \otimes |u\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle\,|u\rangle + |1\rangle\,|u\rangle\big).$$

After the controlled U gate, the state $|\psi_2\rangle$ will be the following, as only the $|1\rangle$ part of the state triggers the U gate:

$$\begin{aligned}|\psi_2\rangle &= \frac{1}{\sqrt{2}}\big(|0\rangle\,|u\rangle + |1\rangle U\,|u\rangle\big)\\ &= \frac{1}{\sqrt{2}}\big(|0\rangle\,|u\rangle + e^{2\pi i 0.\phi_0}|1\rangle\,|u\rangle\big)\\ &= \frac{1}{\sqrt{2}}\big(|0\rangle + e^{2\pi i 0.\phi_0}|1\rangle\big)\,|u\rangle\,.\end{aligned}$$

The state $|u\rangle$ remains unchanged, as it should be since it is an eigenstate of U. We can apply U as often as we want, $|u\rangle$ will remain the same. However, the eigenvalue has become a local phase of the $|1\rangle$ part of the first qubit. We *kicked* the phase *back* to qubit 0, as described in Section 11.1.

This concludes the first part of the single-qubit phase estimation algorithm. There is still the problem that, when measuring the first qubit, the state could still collapse to $|0\rangle$ or $|1\rangle$ with the same probability of $1/2$, regardless of whether the phase value is 0 or 0.5. Kicking the phase up does not change the probabilities.

To resolve this, we apply another Hadamard gate to the top qubit. In the next section, we will learn that this is equivalent to a single-qubit inverse quantum Fourier transform. After applying the final Hadamard gate, we obtain state $|\psi_3\rangle$ (omitting the trailing qubit $|u\rangle$ for ease of notation), as shown in Figure 11.4:

$$\begin{aligned}|\psi_3\rangle &= \frac{1}{\sqrt{2}}H\left(|0\rangle + e^{2\pi i 0.\phi_0}|1\rangle\right)\\ &= \frac{1}{2}\left(1 + e^{2\pi i 0.\phi_0}\right)|0\rangle + \frac{1}{2}\left(1 - e^{2\pi i 0.\phi_0}\right)|1\rangle.\end{aligned}$$

The term ϕ_0 is a binary digit and can only be 0 or 1. If it is 0, we have that the factor $e^{2\pi i 0.\phi_0} = e^0$ becomes 1, and $|\psi_3\rangle$ becomes

$$|\psi_3\rangle = \frac{1}{2}|0\rangle + \frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle - \frac{1}{2}|1\rangle = |0\rangle.$$

On the other hand, if the digit $\phi_0 = 1$, then $1(2^{-1}) = 1/2$, the factor $e^{2\pi i 0.\phi_0}$ becomes $e^{2\pi i/2} = -1$ and $|\psi_3\rangle$ becomes

$$|\psi_3\rangle = \frac{1}{2}|0\rangle - \frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle + \frac{1}{2}|1\rangle = |1\rangle.$$

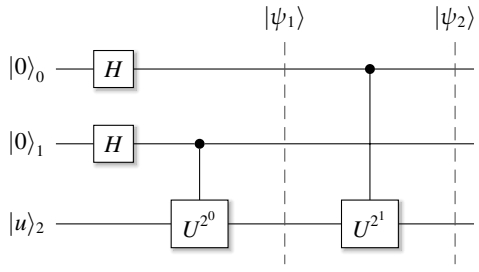


Figure 11.5 Phase estimation circuit with an accuracy of two binary digits (corresponding to $\frac{1}{2}$ and $\frac{1}{4}$) and the corresponding unitaries U^1 and U^2. Here, the top two qubits with subscripts 0 and 1 are in little-endian.

Since $|0\rangle$ and $|1\rangle$ are orthogonal, we will now measure $|0\rangle$ or $|1\rangle$ with certainty, depending on whether the bit ϕ_0 was 0 or 1.

Now that we see how to compute a single digit let us move on and consider two fractional parts for a phase with two decimal binary digits $\phi = 0.\phi_0\phi_1$. With two digits, we can approximate the phase with fractional values 0.0,0.25,0.5, and 0.75, which is already better than a single digit. The corresponding quantum circuit uses two exponentiated gates U^1 and U^2 as shown in Figure 11.5. From above, we know that $|\psi_1\rangle$ will have the form

$$|\psi_1\rangle = \frac{1}{\sqrt{2^2}} \underbrace{\left(|0\rangle + |1\rangle\right)}_{\text{qubit 0}} \otimes \underbrace{\left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1}\,|1\rangle\right)}_{\text{qubit 1}} \otimes \underbrace{|u\rangle}_{\text{qubit 2}} .$$

Let us study the effect of the controlled U^{2^1} on qubit 0. We know that squaring a rotation means doubling the rotation angle as in

$$U^2|\psi\rangle = e^{2\pi i(2\phi)}|\psi\rangle.$$

Looking at the fractional representation $0.\phi_0\phi_1$ and the effect of U^2, we can see that the binary point shifts by one digit to the left:

$$\begin{aligned} 2\phi &= 2(0.\phi_0\phi_1) \\ &= 2\left(\phi_0 2^{-1} + \phi_1 2^{-2}\right) \\ &= \phi_0 + \phi_1 2^{-1} \\ &= \phi_0.\phi_1 \ . \end{aligned}$$

After shifting $0.\phi_0\phi_1$ to $\phi_0.\phi_1$, we can split the exponent:

$$\begin{aligned} e^{2\pi i(2\phi)} &= e^{2\pi i(\phi_0.\phi_1)} \\ &= e^{2\pi i(\phi_0 + 0.\phi_1)} \\ &= e^{2\pi i(\phi_0)} e^{2\pi i(0.\phi_1)}. \end{aligned}$$

Similarly to the single-digit case above, the term ϕ_0 corresponds to a binary digit and can only be 0 or 1. This means that the first factor corresponds to a rotation of 0 or 2π, which has no effect. The final result is

$$e^{2\pi i(2\phi)} = e^{2\pi i(0.\phi_1)},$$

where we shifted ϕ_1 to the left and eliminated ϕ_0. We can generalize this for a phase with t fractional parts $\phi = 0.\phi_0\phi_1 \ldots \phi_{t-1}$ as

$$e^{2\pi i(2^k\phi)} = e^{2\pi i 0.\phi_k\phi_{k+1}\cdots\phi_{t-1}}.$$

For our three-qubits circuit above, after the two controlled gates U^1 and U^2, the state ψ_2 becomes

$$\begin{aligned} |\psi\rangle &= \frac{1}{\sqrt{2^2}} \underbrace{\left(|0\rangle + e^{2\pi i 0.\phi_1}|1\rangle\right)}_{\text{qubit 0}} \otimes \underbrace{\left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1}|1\rangle\right)}_{\text{qubit 1}} \otimes \underbrace{|u\rangle}_{\text{qubit 2}} \\ &= \frac{1}{\sqrt{2^2}} \left(|0\rangle + e^{2\pi i 2^1\phi}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 2^0\phi}|1\rangle\right) \otimes |u\rangle\,. \end{aligned} \tag{11.1}$$

In Section 11.4 on the quantum Fourier transform, we will see that this form is identical to the result from applying the *QFT* operator to a state of two qubits $|\phi_0\rangle$ and $|\phi_1\rangle$ (ignoring the ancilla). This means that we will be able to apply the two-qubit *adjoint* operator $QFT^\dagger$ to retrieve the binary bits of $\phi = 0.\phi_0\phi_1$ as a state with the qubits $|\phi_0\rangle$ and $|\phi_1\rangle$ in states $|0\rangle$ or $|1\rangle$, depending on how the digits ϕ_0 and ϕ_1 were set:

$$QFT^\dagger_{0,1}|\psi_2\rangle = |\phi_0\rangle \otimes |\phi_1\rangle \otimes |u\rangle\,.$$

Let us derive a closed form for the phase estimation. For two qubits, we ignore qubit 2 in Equation (11.1) and multiply out the remaining terms:

$$\begin{aligned} |\psi_2\rangle &= \frac{1}{\sqrt{2^2}} \left(|0\rangle + e^{2\pi i 2^1\phi}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 2^0\phi}|1\rangle\right) \\ &= \frac{1}{\sqrt{2^2}} \left(|00\rangle + e^{2\pi i 2^0\phi}\,|01\rangle + e^{2\pi i 2^1\phi}\,|10\rangle + e^{2\pi i (2^1+2^0)\phi}\,|11\rangle\right) \\ &= \frac{1}{\sqrt{2^2}} \left(|00\rangle + e^{2\pi i 1\phi}\,|01\rangle + e^{2\pi i 2\phi}\,|10\rangle + e^{2\pi i 3\phi)}\,|11\rangle\right), \end{aligned}$$

To generalize, we connect the 0th power of 2 to the last qubit in the t register and the $(t-1)$'s power of 2 to the first qubit,[2] as shown in Figure 11.3. The resulting state becomes

$$\frac{1}{2^{t/2}} \left(|0\rangle + e^{2\pi i 2^{t-1}\phi}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 2^{t-2}\phi}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i 2^0\phi}|1\rangle\right).$$

Multiplying this out results in the important general form for a state $|\psi\rangle$ with basis states $\{|k\rangle\}$. Note that we reintroduce the final qubit $|u\rangle$ here, as the final state is the superposition of the phase register and the unchanged $|u\rangle$:

[2] You can also do this the other way around, depending on your bit ordering convention.

$$QPE\,|\psi\rangle = \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i k\phi}\,|k\rangle \otimes |u\rangle\,. \tag{11.2}$$

We write ϕ with t binary bits in the fractional notation (we will use the same form in Section 11.4 on QFT) as

$$\phi = 0.\phi_0\phi_1\cdots\phi_{t-1}.$$

Multiplying this angle with the powers of two shifts the digits of the binary representation to the left, and the state after the circuit in Figure 11.3 is

$$\begin{aligned} &\frac{1}{2^{t/2}}\left(|0\rangle + e^{2\pi i 0.\phi_{t-1}}\,|1\rangle\right) \\ &\quad\otimes\ \left(|0\rangle + e^{2\pi i 0.\phi_{t-2}\phi_{t-1}}|1\rangle\right) \\ &\qquad\vdots \\ &\quad\otimes\ \left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1\ldots\phi_{t-1}}|1\rangle\right). \end{aligned} \tag{11.3}$$

Again, we will see shortly in Section 11.4 that this is also the result of a QFT applied to a specific computational basis state. The final step of phase estimation is, therefore, to *reverse* the QFT by applying the inverse $QFT^\dagger$ operator to reconstruct the input, a representation of ϕ as a binary fraction in basis encoding. The almost complete circuit is shown in Figure 11.6, where we have yet to measure the qubits. You will find this step in the implementation.

A cautious word on endianness is in order. We can think of the qubits in the figure as addresses from top to bottom, with the lowest address 0 at the top. The first controlled U gate contributes the most to the binary fraction. It is the most significant qubit. In the figure, the most significant qubit appears at the highest address, which means

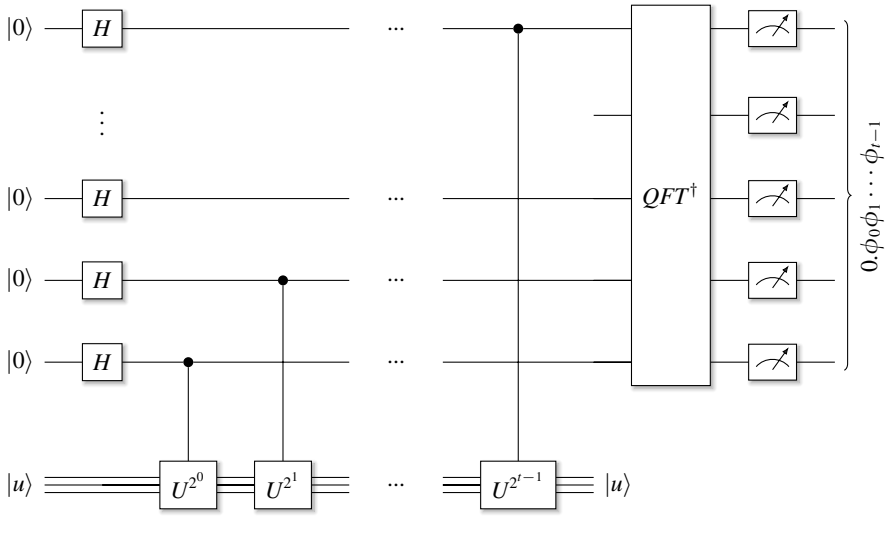


Figure 11.6 Full quantum phase estimation circuit.

we are using the little-endian convention, where the least significant contributor to a value resides at the lowest address. We will often switch between little-endian and big-endian notations in this book;[3] this is something to be aware of.

11.2.3 Implementation

We drive the algorithm from `main()`, where we reserve seven qubits for t (this is arbitrary and chosen for performance reasons only) and two qubits for the unitary operator. You can experiment with these numbers to see how the probabilities change with the size of the unitary operator.

Find the code
In file src/phase_estimation.py

```
def main(argv):
  nbits = 2
  t = 7
  print('Estimating {} qubits random unitary eigenvalue '
        .format(nbits) + 'with {} bits of accuracy.'.format(t))
  for i in range(10):
    run_experiment(nbits, t)
```

In each experiment, we create a random operator and obtain its eigenvalues and eigenvectors to verify the computed estimates:

```
def run_experiment(nbits: int, t: int = 8):
  umat = scipy.stats.unitary_group.rvs(2**nbits)
  eigvals, eigvecs = np.linalg.eig(umat)
  u = ops.Operator(umat)
```

We choose the eigenvector at index 0 in the example, but the procedure works for all other pairs of eigenvectors and eigenvalues. To verify the algorithm, we calculate the angle `phi` to be estimated in advance. Since we assume that the eigenvalue is $e^{2\pi i\phi}$, as discussed in Section 11.2.1, we divide by `2j*np.pi` and correct for negative values. Again, this angle does not participate in the algorithm. We only compute it upfront to compare it against the approximated phase value later:

```
  eigen_index = 0
  phi = np.real(np.log(eigvals[eigen_index]) / (2j*np.pi))
  if phi < 0:
    phi += 1
```

In the construction of the circuit, we initialize the state `psi` with `t` qubits in state $|0\rangle$ tensored to another state that is directly initialized with an eigenvector. Then we

[3] Homework assignment: Count how often I am confusing the endianness.

perform the phase estimation (the code follows shortly) and the inverse QFT on the resulting state:

```
  psi = state.zeros(t) * state.State(eigvecs[:, eigen_index])
  psi = phase_estimation(psi, u, t)
  psi = ops.Qft(t).adjoint()(psi)
```

The heart of this circuit is the controlled connection of the operators taken to powers of two, which is implemented in function `phase_estimation` (we also add a version of it to the library file `src/lib/ops.py`, as it is used in several other algorithms). You can see that after the Hadamard gates have been applied to the `t` register, the code iterates, and the `u2` operator is repeatedly multiplied by itself to produce the powers of two required by the phase estimation algorithm:

```
def phase_estimation(psi: state.State, u: ops.Operator, t: int):
  psi = ops.Hadamard(t)(psi)
  u2 = u
  for inv in reversed(range(t)):
    psi = ops.ControlledU(inv, t, u2)(psi, inv)
    u2 = u2(u2)
  return psi
```

All that is left to do is to simulate a measurement by picking the state with the highest probability, computing the representation of the state as a binary fraction, and comparing the result against the target value. Since we have limited bits to represent the result, we allow an error margin of 2%. More bits for *t* will make the circuit run slower but also improve the error margins.

```
  # Find state with highest measurement probability and show results.
  maxbits, maxprob = psi.maxprob()
  phi_estimate = helper.bits2frac(maxbits[:t])

  delta = abs(phi - phi_estimate)
  print('Phase    : {:.4f}'.format(phi))
  print('Estimate: {:.4f} delta: {:.4f} probability: {:5.2f}%'
        .format(phi_estimate, delta, maxprob * 100.0))
  if delta > 0.02 and phi_estimate < 0.98:
    print('*** Warning: Delta is large')
```

There is the potential for `delta` to be greater than the hard-coded 2% when an insufficient number of bits was reserved for `t`. Another error case is when the eigenvalue rounds to 1.0. In this case, all digits after the dot are 0, and the estimated binary fraction will also be 0 instead of the correct value of 1.0. The code warns about this case. The results should look similar to this:

```
Estimating 2 qubits random unitary eigenvalue with 7 bits of accuracy
Phase   : 0.5180
Estimate: 0.5156 delta: 0.0024 probability: 31.65%
Phase   : 0.3203
Estimate: 0.3125 delta: 0.0078 probability:  7.30%
[...]
Phase   : 0.6688
Estimate: 0.6719 delta: 0.0030 probability: 20.73%
```

11.2.4 Estimating Multiple Eigenvalues

So far, we have explored the case of a single eigenvalue and eigenvector pair. What if the initial state $|u\rangle$ was in a superposition of multiple eigenvectors? Can we derive multiple eigenvalues with this procedure? Yes, we can, with the caveat that measurements are still probabilistic and obtaining all eigenvalues is subject to the usual probability laws. We use this feature later in Section 14.3 on the HHL algorithm and provide another detailed derivation there.

For experimentation, we mirror the code above very closely. To generate a superposition of multiple eigenvectors, we again generate a random unitary and find its multiple pairs of eigenvalues and eigenvectors. We calculate the eigenvalues as fractions of π in the `phi` array and add 1 to any negative values. As a simplification, we calculate the superposition of the eigenvectors as a simple addition of scaled vectors in variable `ini` and use it to directly initialize the state.[4] All that is left to do is run the phase estimation circuitry and apply the inverse QFT:

```
umat = scipy.stats.unitary_group.rvs(2**nbits)
eigvals, eigvecs = np.linalg.eig(umat)

phi = np.array([np.real(np.log(v) / (2j*np.pi)) for v in eigvals])
phi[phi < 0] += 1

fac = np.sqrt(1 / 2**nbits)
ini = np.sum(fac * eigvecs, axis=1)

# Make state and circuit to estimate phi (similar to above).
psi = state.zeros(t) * state.State(ini)
psi = phase_estimation(psi, u, t)
psi = ops.Qft(t).adjoint()(psi)
```

We collect the states with the highest probability ($\geq 1\%$), as the eigenvalues encoded in these states will have the highest amplitudes. The final step is to find whether these states represent the eigenvalues. If an estimate has been found correctly, we mark it as `-> Found` and as `-> ***` otherwise. Since we have set more or less arbitrary limits for the required probability and precision, a small number of eigenvalues is expected to be marked as not found (`***`), as shown in the output below.

[4] We already saw in Chapter 9 that initializing a state with an actual circuit can be quite challenging.

```
  estimates = [helper.bits2frac(bits)
               for bits in helper.bitprod(psi.nbits)
               if psi.prob(*bits) >= 0.03]
  for p in phi:
    print(f'Phase : {p:.4f} ', end='')
    est = [x for x in estimates if abs(p - x) < 0.01]
    print('-> Found' if len(est) else '-> ***')
>>
Phase : 0.2342 -> Found
Phase : 0.9560 -> Found
Phase : 0.4984 -> Found
Phase : 0.6832 -> Found

Phase : 0.9660 -> ***
Phase : 0.6609 -> Found
Phase : 0.2599 -> Found
Phase : 0.4468 -> Found
[...]
```

11.3 Approximating π

Phase estimation can be used in an interesting manner to approximate the value of π. As discussed in Section 11.2.1, when a unitary operator U is applied to one of its eigenvectors such as $|\psi\rangle$, its eigenvalues have unit norm:

$$U|\psi\rangle = e^{2\pi i\phi}|\psi\rangle. \tag{11.4}$$

We already know the $U_1(\theta)$ operator from Section 2.7.7:

$$U_1(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

This operator is a diagonal matrix and has its eigenvalues 1 and $e^{i\theta}$ on the diagonal. If we apply $U_1(\theta)$ to its eigenstate $|\psi\rangle = |1\rangle$, as in

$$U_1(\theta)|1\rangle = e^{i\theta}|1\rangle, \tag{11.5}$$

and furthermore equate Equations (11.4) and (11.5), we get

$$e^{2\pi i\phi} = e^{i\theta}.$$

We smartly set $\theta = 1$, which leads to an approximation of π as

$$2\pi\phi = 1 \quad \Rightarrow \quad \pi = \frac{1}{2\phi}.$$

Now we can estimate ϕ with the QPE. We define a function to run experiments with a given number of qubits to represent the phase values – as usual, the more qubits we use, the more accurate the estimation of π will be, but the slower the experiment will run.

First, we build a circuit with the given number of qubits to hold the results of the phase estimation. We also have to add an ancilla for the phase estimation and apply an X gate to turn it into eigenstate $|1\rangle$:

Find the code
In file src/estimate-pi.py

```
def run_experiment(nbits_phase):
  qc = circuit.qc('pi estimator')
  qclock = qc.reg(nbits_phase)
  qbit = qc.reg(1)
  qc.x(qbit)  # make it |1>

  # Perform phase estimation.
  qc.h(qclock)
  for inv in range(nbits_phase):
    qc.cu1(qclock[inv], qbit[0], 2**(nbits_phase - inv - 1))
  qc.inverse_qft(qclock)

  # Compute pi (and the delta from it).
  bits, _ = qc.psi.maxprob()
  theta = helper.bits2frac(bits[:nbits_phase][::-1])
  pi = 1 / (2 * theta)
  delta = np.abs(pi - np.pi)

  print(f'Pi Est: {pi:.5f} (qb: {nbits_phase:2d}) Delta: {delta:.6f}')
  assert delta < 0.06, 'Incorrect Estimation of Pi.'
```

We run over an increasing number of qubits (you may have to adjust the upper limit for performance reasons) to get increasingly more accurate approximations for π:

```
def main(argv):
  print('Estimate Pi via phase estimation...')
  for nbits in range(7, 20):
    run_experiment(nbits)
>>
Estimate Pi via phase estimation...
Pi Est: 3.20000 (qb:  7) Delta: 0.058407
Pi Est: 3.12195 (qb:  8) Delta: 0.019641
Pi Est: 3.16049 (qb:  9) Delta: 0.018901
[...]
Pi Est: 3.14159 (qb: 19) Delta: 0.000001
```

How scalable is this methodology for finding π? At the time of writing, the world record for classically estimating π was 105 trillion digits. Observing the progression of the results in our experiments, about three qubits are required to improve the estimation accuracy by a factor of 10, gaining a single additional digit of precision. The biggest classical chip at the time of writing was a full-wafer chip with about 4 trillion

classical bits. In other words, let us not wait for a quantum computer to try to break that world record.

However, there is also a related helpful result. A similar technique was presented by Bochkin et al. (2020) to estimate the precision of individual qubits by estimating π and propagating the remaining error back to the qubits.

11.4 Quantum Fourier Transform (QFT)

In classical engineering, mathematics, and physics, the discrete Fourier transform (DFT) is an analysis technique that, for a complex-valued function $f(\cdot)$, finds a series of underlying periodic frequencies and amplitudes that combine to the original function $f(\cdot)$. It is written as

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i j k/N}, \tag{11.6}$$

where a sequence of N complex values x_j is transformed into the sequence y_k. The computational complexity of DFT is $\mathcal{O}(N^2)$, which makes it impractical for large problems or problems with fast latency requirements. The development of the fast Fourier transform (FFT) reduced the complexity to $\mathcal{O}(N \log N)$, a major breakthrough in classical computing.

The discovery of the quantum Fourier transform (QFT), which promises a further reduction of a theoretical complexity of just $\mathcal{O}(\log^2 N)$, $N = 2^n$ for n qubits, was an exciting moment for quantum computing.[5] The QFT is one of the fundamental algorithms of quantum computing. It enables important algorithms, such as phase estimation, which we learned about in Section 11.2.1. As we shall see shortly, phase estimation is also a key ingredient in Shor's factoring algorithm.[6]

The QFT has a similar form as the DFT but is expressed as a unitary state evolution, where for n qubits, an operator QFT of size $2^n \times 2^n$ is applied to transform a state in the computational basis $|x\rangle = \sum_{i=0}^{N-1} x_i |i\rangle$ into a state $|y\rangle = \sum_{i=0}^{N-1} y_i |i\rangle$:

$$QFT\,|x\rangle = |y\rangle\,. \tag{11.7}$$

We define $\omega_N = e^{2\pi i/N}$. The elements of the set $\{\omega^n = 2^{2\pi i n/N}\}$ are called the Nth roots of unity (the nth root of unity[7] is a complex number z such that $z^n = 1$). The individual components of the state vector are transformed as

$$y_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \omega_N^{nk}, \quad k = 0,1,\ldots,N-1.$$

If the state $|x\rangle$ in Equation (11.7) is a basis state (which in our cases it always will be), then we can write the closed form for the QFT operator as

[5] The complexity of QFT was later refined to $\mathcal{O}(N \log^2 N)$ to account for measurements (Musk, 2020).

[6] Note that for historical reasons, the inverse DFT is often considered analogous to the QFT, but let's not get hung up on those details here.

[7] See also http://en.wikipedia.org/wiki/Root_of_unity.

$$QFT\,|x\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \omega_N^{jk}\,|j\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi ijk/N}\,|j\rangle\,. \tag{11.8}$$

Note how similar this form is to the classical DFT in Equation (11.6). Sometimes, authors put a minus sign in front of the exponent. This is a matter of convention. As an example, to apply this formula to the two-qubit state $|10\rangle = |2\rangle$, with $N = 4$ and $k = 2$, we get

$$\begin{aligned} QFT\,|10\rangle &= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi ijk/N}\,|j\rangle \\ &= \frac{1}{2}\left(e^{2\pi i 0\cdot 2/4)}\,|0\rangle + e^{2\pi i 1\cdot 2/4)}\,|1\rangle + e^{2\pi i 2\cdot 2/4)}\,|2\rangle + e^{2\pi i 3\cdot 2/4)}\,|3\rangle\right) \\ &= \frac{1}{2}\big(|0\rangle - |1\rangle + |2\rangle - |3\rangle\big). \end{aligned}$$

To write the QFT as an operator for n qubits with $N = 2^n$ basis states, we define the general and quite elegant form of the operator with $\omega = \omega_N$ as

$$QFT_N = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{pmatrix}.$$

In fact, a simple mnemonic for the entry at row i and column j is

$$QFT_{i,j} = \frac{\omega^{i\cdot j}}{\sqrt{N}} |i\rangle\langle j|.$$

For two qubits, this matrix is the following. Note that the factors in column 2, $(+1,-1,+1,-1)$, match the resulting signs in the above example, where we calculated $QFT\,|10\rangle = \frac{1}{2}\,(+\,|0\rangle - |1\rangle + |2\rangle - |3\rangle)$. We would get similar results for the other basis states and their corresponding columns (specifically column 0, because applying QFT to state $|00\rangle$ will result in all-zero exponents):

$$QFT_4 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}.$$

To follow this line of reasoning to the smallest operator, for a single qubit, the QFT_2 operator is the same as the Hadamard gate:

$$QFT_2 = H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

As we stated in Section 11.2.1, applying the QFT to a state will match the output of a phase estimation circuit. Our mission in this chapter is to derive this form and the circuit to produce it.

11.4.1 QFT Circuit

Let us again consider a state in the binary interpretation $\phi = |\phi_0\phi_1\cdots\phi_{n-1}\rangle$ representing the decimal value

$$0.\phi_0\phi_1\ldots\phi_{n-1} = \phi_0\frac{1}{2^1} + \phi_1\frac{1}{2^2} + \ldots + \phi_{n-1}\frac{1}{2^{n-1}},$$

with the ϕ_i representing the binary bits 0 or 1. Recall from Section 11.2.1 that the result of phase estimation on $|\phi\rangle$ was Equation (11.3), reproduced here as Equation (11.9). We also claimed that this state was identical to the result of a QFT applied to state $|\phi\rangle$:

$$\frac{\left(|0\rangle + e^{2\pi i 0.\phi_{t-1}}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.\phi_{t-2}\phi_{t-1}}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1\ldots\phi_{t-1}}|1\rangle\right)}{2^{t/2}} \tag{11.9}$$

Let us derive this result. The arithmetic looks quite daunting, you may choose to focus on the final result only.

$$\begin{aligned}
|\phi\rangle \rightarrow & \frac{1}{2^{n/2}}\sum_{k=0}^{N-1} e^{2\pi i\phi k/N}|k\rangle \\
= & \frac{1}{2^{n/2}}\sum_{k_0=0}^{1}\cdots\sum_{k_{n-1}=0}^{1} e^{2\pi i\phi\left(\sum_{l=1}^{n}2^{-l}\right)}|k_0\ldots k_{n-1}\rangle \\
= & \frac{1}{2^{n/2}}\sum_{k_0=0}^{1}\cdots\sum_{k_{n-1}=0}^{1}\bigotimes_{l=0}^{n-1} e^{2\pi i\phi k_l 2^{-l}}|k_l\rangle \\
= & \frac{1}{2^{n/2}}\bigotimes_{l=0}^{n-1}\left[\sum_{k_l=0}^{1} e^{2\pi i\phi k_l 2^{-l}}|k_l\rangle\right] \\
= & \frac{1}{2^{n/2}}\bigotimes_{l=0}^{n-1}\left[|0\rangle + e^{2\pi i\phi 2^{-l}}|1\rangle\right] \\
= & \frac{\left(|0\rangle + e^{2\pi i 0.\phi_{n-1}}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.\phi_{n-2}\phi_{n-1}}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1\cdots\phi_{n-1}}|1\rangle\right)}{2^{n/2}}.
\end{aligned}$$

Depending on the endianness of the circuit, the terms may be in reverse order. Not to worry, we can augment a QFT circuit with final Swap gates to reverse the order of the terms. Alternatively, instead of hooking up the unitaries with control qubits going from the bottom to the top, we can also build a circuit where the control qubits go from top to bottom. In this book, we will find examples of both ways.

How would we build the circuit to produce this state? In the following, we will use the R_k gate from Section 2.7.7:

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}.$$

The following process closely mirrors what we have already learned from phase estimation. For notation, we will write H_x to indicate that a Hadamard gate is applied

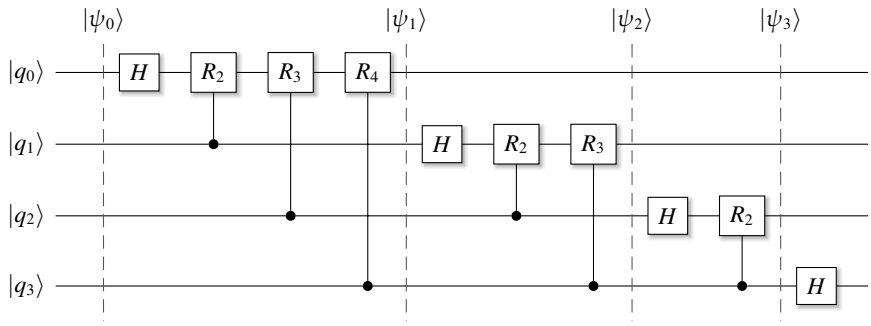


Figure 11.7 A QFT circuit for four qubits. This circuit uses little-endian convention (the least significant qubit is at the lowest index, which is the top qubit). To produce a circuit in big-endian convention, you can mirror the qubit indices about the middle with qubits $0 \leftrightarrow 3$ and $1 \leftrightarrow 2$.

to the qubit at index x. For the controlled R_k gates, we will write $R_{k(x)}$ to indicate that the R_k gate is applied to the qubit at index x. We follow the sequence of gates shown in Figure 11.7. Note that this whole mechanism is an extended phase-kick circuit.

For the first qubit, we get a first digit ϕ_0 by applying a Hadamard gate H_0. Applying this gate results in the state

$$H_0 |\phi\rangle = \frac{1}{2^{1/2}} \left(|0\rangle + e^{2\pi i 0.\phi_0}|1\rangle\right) |\phi_1 \cdots \phi_{n-1}\rangle .$$

If bit $\phi_0 = 1$, then $e^{2\phi i 0.\phi_0} = -1$, if $\phi_0 = 0$, then $e^{2\phi i 0.\phi_0} = 1$ and we get the expected result from the Hadamard gate. We can append a second digit by applying a controlled R_2 gate:

$$R_{2(0)} H_0 |\phi\rangle = \frac{1}{2^{1/2}} \left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1}|1\rangle\right) |\phi_2 \cdots \phi_{n-1}\rangle .$$

For the remaining digits, we follow this with a sequence of controlled R_k gates, $k = 3, \ldots, n$, resulting in

$$|\psi_1\rangle = \frac{1}{2^{1/2}} \left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1\cdots\phi_{n-1}}\right) .$$

For the second qubit q_1 (at index 1), we repeat an almost identical sequence of gates, but it will be shorter by one rotation gate. We start again with a single Hadamard gate to the second qubit

$$\begin{aligned} H |\psi_1\rangle &= H_1 R_{n-1(0)} \cdots R_{2(0)} H_0 |\phi\rangle \\ &= \frac{1}{2^{2/2}} \left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1\cdots\phi_{n-1}}\right) \left(|0\rangle + e^{2\pi i 0.\phi_1}|1\rangle\right) |\phi_2 \cdots \phi_{n-1}\rangle , \end{aligned}$$

and apply a controlled R_2 gate as above to get a second digit for the second qubit

$$\begin{aligned} &R_{2(1)} H_1 R_{n-1(0)} \cdots R_{2(0)} H_0 |\phi\rangle \\ &= \frac{1}{2^{2/2}} \left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1\cdots\phi_{n-1}}\right) \left(|0\rangle + e^{2\pi i 0.\phi_1\phi_2}|1\rangle\right) |\phi_3 \cdots \phi_{n-1}\rangle . \end{aligned}$$

We continue this process for the remaining digits. We then use this methodology for the remaining terms in Equation (11.9) and the corresponding qubits, with ever-shorter gate sequences, until only a final Hadamard gate remains after $|\psi_3\rangle$ (you may recall the final Hadamard gate in Figure 11.4 in the section on QPE; this gate indeed corresponded to a single-qubit QFT).

Note that the terms are in reverse order compared to Equation (11.9). We would only have to invert the qubit indices to obtain a matching result, for example, by starting the very first Hadamard gate on qubit 3 instead of qubit 0. We chose the current order because it is easier to write. You will find that both the big-endian and little-endian conventions are being used in this book and the literature.

Recall that controlled phase gates are symmetrical, as shown in Section 2.9, so sometimes you may see a functionally identical circuit but with switched controlled and controlling qubits.

As we have already stated in Section 11.2.1, the QFT operator is a unitary operator, as it should be because it is made up of other unitary operators. Since it is unitary, it has an inverse, the Hermitian conjugate. To repeat what we have learned in Section 11.2.1 on QPE, this is how we get from the output of phase estimation to a state representing binary bits:

$$\begin{aligned} QFT^\dagger \frac{1}{2^{n/2}} \otimes & \left(|0\rangle + e^{2\pi i 0.\phi_{n-1}}|1\rangle\right) \\ & \otimes \left(|0\rangle + e^{2\pi i 0.\phi_{n-2}\phi_{n-1}}|1\rangle\right) \\ & \vdots \\ & \otimes \left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1\cdots\phi_{n-1}}|1\rangle\right) \\ & = |\phi_0\ \phi_1\ \cdots\ \phi_{n-1}\rangle. \end{aligned} \tag{11.10}$$

In many algorithms, we will apply the inverse QFT to remove the superposition and obtain a result, as shown in Equation (11.10). An important aspect of QFT is that, while it enables the encoding of (binary) states in superposition with phases, on measurement in the computational bases, the state collapses to just one basis state. All other information will be lost. The challenge for QFT-based algorithms is to use tricks and transformations so that we can find a solution to a given problem with high probability.

How many fractions do we need to achieve a reliable result for a specific algorithm? This is an interesting metric to play with. Early work on the approximate quantum Fourier transform indicates that, for Shor's algorithm, you can stop adding rotation gates as the rotation angles become smaller than π/n^2 (Coppersmith, 2002). This reduces the complexity of the QFT circuit without affecting the accuracy of the results, making this a viable optimization for practical implementations of quantum algorithms.

11.4.2 QFT Operator

In code, we implement the QFT operator as a full matrix. We put the input qubits in superposition with Hadamard gates and apply controlled rotations for each fractional

part. Be careful to put the indices in the right order.[8] We also provide an optional facility to reverse the order of qubits with Swap gates:

Find the code
In file src/lib/ops.py

```
def Qft(nbits: int, swap: bool = True) -> Operator:
  op = Identity(nbits)
  h = Hadamard()
  for idx in range(nbits):
    op = op(h, idx)
    for rk in range(2, nbits - idx + 1):
      controlled_from = idx + rk - 1
      op = op(ControlledU(controlled_from, idx, Rk(rk)), idx)
  if swap:
    for idx in range(nbits // 2):
      op = op(Swap(idx, nbits - idx - 1), idx)
  assert op.is_unitary(), 'Constructed non-unitary operator.'
  return op
```

Calculating the inverse of the QFT operator is trivial. QFT is a unitary operator, so the inverse is simply the adjoint:

```
    Qft = ops.Qft(nbits)
    [...]
    InvQft = Qft.adjoint()
```

Suppose QFT is computed via explicit gate applications in a circuit. In that case, the inverse has to be implemented as the application of the inverse gates in reverse order, as outlined in Section 2.12 on reversible computing (we should also recall that for a product of matrices, $(AB)^{-1} = B^{-1}A^{-1}$). We add implementations of `qft` and `inverse_qft` to the `circuit` class and will see examples of their use shortly.

Find the code
In file src/lib/circuit.py

```
  def qft(self, reg, with_swaps: bool = False) -> None:
    for i in reversed(range(len(reg))):
      self.h(reg[i])
      for j in reversed(range(i)):
        self.cu1(reg[i], reg[j], np.pi/2**(i - j))
    if with_swaps:
      self.flip(reg)

  def inverse_qft(self, reg, with_swaps: bool = False) -> None:
    if with_swaps:
      self.flip(reg)
```

[8] This is very easy to get wrong.

```
    for idx, r in enumerate(reg):
      self.h(r)
      if idx != len(reg) - 1:
        for y in range(idx, -1, -1):
          self.cu1(reg[idx + 1], reg[y], -np.pi / 2 ** (idx + 1 - y))
```

Note that qubit ordering can be an issue, as discussed. Depending on whether the big-endian or little-endian convention is used, the order in which gates are applied to qubits may have to be inverted. Alternatively, the binary interpretation of qubits can be reverted.

11.4.3 Online Simulation

It can be helpful to use one of the available online simulators to verify the results. Be aware that the simulators might not agree on the qubit ordering. For experiments, we can always add Swap gates at the end of a circuit to follow online simulators' qubit ordering. Alternatively, we can also add the Swap gates to the circuits in the online simulators.

A widely used online simulator is Quirk (Gidney, 2021a). Let us construct a simple two-qubit QFT circuit in Quirk, as shown in Figure 11.8. To the right of this graphical representation, we can reconstruct the phases from the gray circles (blue on the website). We see that the state $|00\rangle$ (top left) has a phase of 0 (the direction of the x-axis), the state $|01\rangle$ (top right) has a phase of 180°, the state $|10\rangle$ (bottom left) has a phase of $-90°$, and the state $|11\rangle$ has a phase of 90°.

In our infrastructure, we would construct the same circuit:

```
qc = circuit.qc()
reg = qc.reg(2, [1, 1])
qc.qft(reg, True)    # True, for the final swap gates.
qc.psi.dump()
>>
|00> (|0>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|01> (|1>):  ampl: -0.50+0.00j prob: 0.25 Phase: 180.0
|10> (|2>):  ampl: -0.00-0.50j prob: 0.25 Phase: -90.0
|11> (|3>):  ampl: +0.00+0.50j prob: 0.25 Phase:  90.0
```

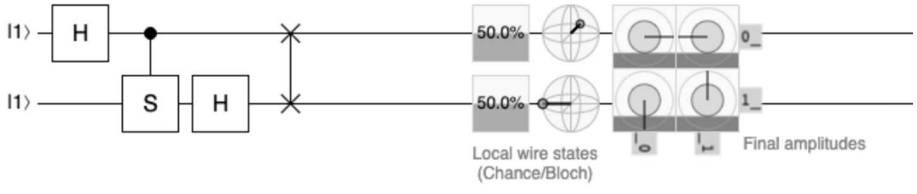


Figure 11.8 A partial screenshot from http://algassert.com/quirk, showing a two-qubit QFT operation with swap gates.

Quirk agrees with our qubit ordering (or we agree with Quirk). Quirk also shows the state of individual qubits on a Bloch sphere. How does this work, as we are dealing with a two-qubit tensored state, and Bloch spheres only represent single qubits? We discussed the partial trace in Section 4.3, which allows *tracing out* qubits from a state. The result is a reduced density matrix. In Section 2.3, we showed how to compute the Bloch sphere coordinates from a density matrix. For systems of more than two qubits, all qubits that are not of interest must be traced out so that only a 2×2 density matrix remains.

Let's try this out. From the state shown in Figure 11.8, we trace out qubit 0 and qubit 1 individually and compute the Bloch sphere coordinates:

```
    psi = state.bitstring(1, 1)
    psi = ops.Qft(2)(psi)
    rho0 = ops.TraceOut(psi.density(), [1])
    rho1 = ops.TraceOut(psi.density(), [0])
    x0, y0, z0 = helper.density_to_cartesian(rho0)
    x1, y1, z1 = helper.density_to_cartesian(rho1)
    print('x0: {:.1f} y0: {:.1f} z0: {:.1f}'.format(x0, y0, z0))
    print('x1: {:.1f} y1: {:.1f} z1: {:.1f}'.format(x1, y1, z1))
>>
x0: -1.0 y0: 0.0 z0: -0.0
x1: -0.0 y1:-1.0 z1: -0.0
```

This result agrees with Quirk as well. The first qubit is located at -1 on the x-axis of the Bloch sphere (the x-axis goes from the back of a page to the front), and the second qubit is located at -1 on the y-axis (going from left to right).

11.5 Quantum Arithmetic

We saw in Section 5.1 how a quantum circuit can emulate a classical full adder, using quantum gates without exploiting any of the unique features of quantum computing, such as superposition or entanglement. All qubits were $|0\rangle$ or $|1\rangle$, which was equivalent to classical computing. This was a nice exercise demonstrating the universality of quantum computing but an inefficient way to construct a full adder. There was no demonstrable quantum advantage.

In this section, we discuss another algorithm that performs addition and subtraction. Here, the math is being developed in the Fourier domain with a technique first described by Draper (2000). Updated techniques can be found in Cuccaro et al. (2004), Gidnay (2018), and Wang et al. (2023).

To perform addition, we apply a QFT, a sprinkle of magic, and a final inverse QFT to obtain the desired numerical result. We explain this algorithm with just a hint of math and a lot of code. This implementation uses a different direction from the controller to the controlled qubit as in our early QFT operator. This is not difficult to

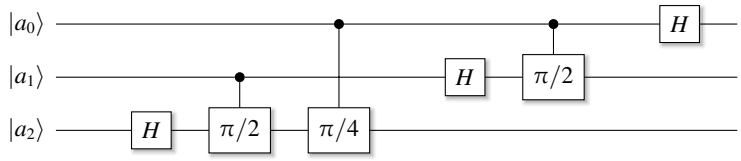


Figure 11.9 A three-qubit QFT in big-endian convention (the least-significant qubit is at the highest address, index 2).

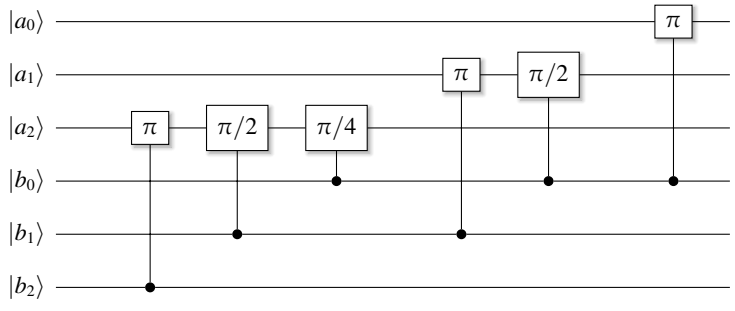


Figure 11.10 The evolve step for a three-qubit quantum addition in the Fourier domain.

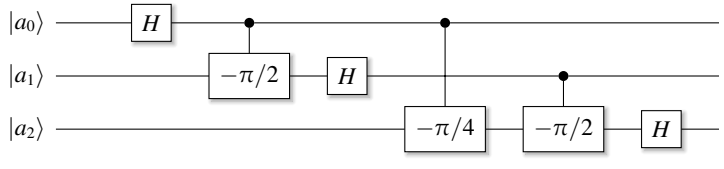


Figure 11.11 The inverse QFT for three qubits, also in big-endian convention.

follow; simply inverting the qubits in a register leads to identical implementations. We use explicit angles and the controlled U_1 gate.

The algorithm performs three basic operations to compute $a + b$, for which we show some numerical examples below:

- Apply QFT to the qubits that represent an input value a. This encodes the bits as phases on states. The corresponding circuit for three qubits is shown in Figure 11.9.
- Evolve a by the value b to compute their addition. This cryptic-sounding step performs another set of QFT-like rotations on a using the same controlled rotation mechanism as regular QFT. However, it is not a full QFT. There are also no initial Hadamard gates, as the states are already in superposition. We detail the steps in the following; an example for three qubits is shown in Figure 11.10
- Perform an inverse QFT to decode phases back to bits. The three-qubit example is in Figure 11.11

In code, these key functions are implemented as the following:

Find the code
In file src/arith_quantum.py

```
def qft(qc: circuit.qc, reg: state.Reg, n: int) -> None:
  qc.h(reg[n])
  for i in range(n):
    qc.cu1(reg[n - (i + 1)], reg[n], math.pi / float(2 ** (i + 1)))

def evolve(qc: circuit.qc, reg_a: state.Reg, reg_b: state.Reg,
           n: int, factor: float) -> None:
  for i in range(n + 1):
    qc.cu1(reg_b[n - i], reg_a[n], factor * math.pi / float(2**i))

def inverse_qft(qc: circuit.qc, reg: state.Reg, n: int) -> None:
  for i in range(n):
    qc.cu1(reg[i], reg[n], -1 * math.pi / float(2 ** (n - i)))
  qc.h(reg[n])
```

To drive the algorithm, we first need to specify the bit width of the inputs a and b. For n-bit arithmetic, we need $(n+1)$ bits to account for overflow.

The signature of our entry point will get the bit width as `n` and the two initial integer values `init_a` and `init_b`, which must fit the available bits. The parameter `factor` will be 1.0 for addition and -1.0 for subtraction. We will see shortly how this factor applies. We instantiate the two registers with bit width $n+1$. Because we interpret the bits in reverse order, we have to invert the bits when initializing the registers:

```
def arith_quantum(n: int, init_a: int, init_b: int, factor: float = 1.0):
  a = qc.reg(n+1, helper.val2bits(init_a, n)[::-1], name='a')
  b = qc.reg(n+1, helper.val2bits(init_b, n)[::-1], name='b')

  for i in range(n+1):
    qft(qc, a, n-i)
  for i in range(n+1):
    evolve(qc, a, b, n-i, factor)
  for i in range(n+1):
    inverse_qft(qc, a, i)
```

Let us look at these three steps in detail, using the example of a two-qubit addition using three qubits for both the a and b registers. The initial QFT is a standard three-qubit QFT circuit. We can enumerate the qubits from 0 to 2 or 2 to 0; it does not make a real difference as long as we remain consistent. After the first loop, we constructed a standard QFT circuit. The middle loop in Figure 11.10 is where the magic happens – we explain how this works in the following. The construction of the inverse QFT circuit takes place in the third loop. All of the first QFT gates are inverted and applied in reverse order.

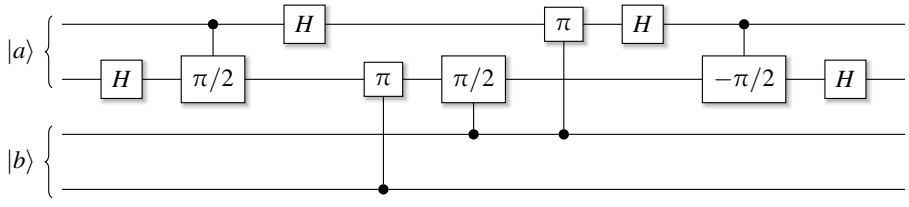


Figure 11.12 The addition circuit for single-qubit inputs, each with an additional overflow qubit.

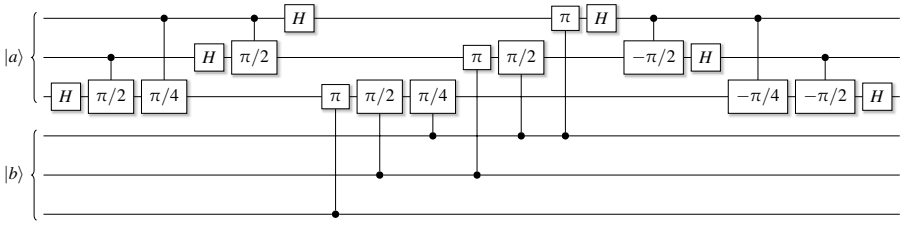


Figure 11.13 The addition circuit for two-qubit inputs, each with an additional overflow qubit.

Recall that the inverse of the Hadamard gate is another Hadamard gate, and the inverse of a rotation is a rotation by the same angle but in the opposite direction. For single-qubit registers a and b, the addition circuit, including the overflow qubit, is shown in Figure 11.12. A corresponding circuit for two-qubit addition with an overflow qubit is shown in Figure 11.13.

To elaborate further on the numerical results for the example in Figure 11.13, we construct two two-qubit states holding the value 1 using the little-endian convention (we have to revert the bits with `[::-1]`). With the extra qubit for overflow, the state after initialization is the following:

```
  qc = circuit.qc('qadd')
  a = qc.reg(n + 1, helper.val2bits(init_a, n)[::-1], name='a')
  b = qc.reg(n + 1, helper.val2bits(init_b, n)[::-1], name='b')
>>
|100100> (|36>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

After QFT on the first qubit, represented by the first three digits in the textual representation, the state becomes the following, where the bottom three qubits representing the input b are still in state $|100\rangle$ (marked in bold in this snippet):

```
|000100> (| 4>):  ampl: +0.35+0.00j prob: 0.12 Phase:   0.0
|001100> (|12>):  ampl: +0.25+0.25j prob: 0.12 Phase:  45.0
|010100> (|20>):  ampl: +0.00+0.35j prob: 0.12 Phase:  90.0
|011100> (|28>):  ampl: -0.25+0.25j prob: 0.12 Phase: 135.0
|100100> (|36>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|101100> (|44>):  ampl: -0.25-0.25j prob: 0.12 Phase: -135.0
|110100> (|52>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
|111100> (|60>):  ampl: +0.25-0.25j prob: 0.12 Phase: -45.0
```

The state after the evolution step becomes a little harder to interpret:

```
|000100> (| 4>):  ampl: +0.35+0.00j prob: 0.12 Phase:   0.0
|001100> (|12>):  ampl: +0.00+0.35j prob: 0.12 Phase:  90.0
|010100> (|20>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|011100> (|28>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
|100100> (|36>):  ampl: +0.35-0.00j prob: 0.12 Phase:  -0.0
|101100> (|44>):  ampl: +0.00+0.35j prob: 0.12 Phase:  90.0
|110100> (|52>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|111100> (|60>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
```

The final state after the inverse QFT will have the correct result of binary `010` or decimal 2 in the top three qubits (marked in bold):

```
|010100> (|20>):  ampl: +1.00-0.00j prob: 1.00 Phase:  -0.0
```

How does this work? Let us first try to explain it mathematically. First, remember that QFT in little-endian takes this input state:

$$|\psi\rangle = |a_{n-1}\ a_{n-2}\ \cdots\ a_1\ a_0\rangle,$$

and transforms it to

$$\begin{aligned} QFT|\psi\rangle = \frac{1}{2^{n/2}} &\left(|0\rangle + e^{2\pi i 0.a_{n-1}a_{n-2}\cdots a_0}|1\rangle\right) \\ &\vdots \\ &\otimes \left(|0\rangle + e^{2\pi i 0.a_{n-1}a_{n-2}}|1\rangle\right) \\ &\otimes \left(|0\rangle + e^{2\pi i 0.a_{n-1}}|1\rangle\right). \end{aligned}$$

Applying the rotations of the `evolve` step adds the binary fractions of b to a. For example, let us look at the first term and apply the various controlled gates CR_k. Recall from Section 2.7.7 that $R_k(n) = U_1(2\pi/2^n)$. Doing this for all the fractional parts in the evolve step, the final state becomes

$$\begin{aligned} |\psi(a+b)\rangle &= \left(|0\rangle + e^{2\pi i 0.a_{n-1}a_{n-2}\cdots a_0}|1\rangle\right) \\ &\rightarrow \left(|0\rangle + e^{2\pi i 0.a_{n-1}a_{n-2}\cdots a_0 + 0.b_{n-1}}|1\rangle\right) && CR_2 \text{ gate on } b_{n-1}, \\ &\rightarrow \left(|0\rangle + e^{2\pi i 0.a_{n-1}a_{n-2}\cdots a_0 + 0.b_{n-1}b_{n-2}}|1\rangle\right) && CR_3 \text{ gate on } b_{n-2}, \\ &\quad\vdots \\ &\rightarrow \left(|0\rangle + e^{2\pi i 0.a_{n-1}a_{n-2}\cdots a_0 + 0.b_{n-1}b_{n-2}\cdots b_0}|1\rangle\right) && CR_{n-1} \text{ gate on } b_0. \end{aligned}$$

Given the insight that rotations in the Fourier domain facilitate addition, it is almost too easy to implement subtraction – we add a factor of -1 to b to evolve the state in the opposite direction. This is already implemented in the `evolve` function above.

With the same line of reasoning, we can easily express multiplications of the form $a+cb$, by just applying that factor to the rotation gates. We have to be careful with numerical overflow and make sure to reserve enough qubits to hold the result.

The algorithm does not implement an actual multiplication, as recently proposed by Gidney (2019), where the factor c is held in another quantum register as input to the algorithm. However, performing multiplication in this way has an important use case. In Section 11.6 on Shor's algorithm, we will multiply by constant integers for which we can classically compute the rotation angles required for addition.

To test our code, we check the results with a routine that performs a measurement. As usual, we do not actually measure but look up the state with the highest probability. After the rotations and coming out of the superposition, the basis state representing the sum $a + b$ will have a probability close to 1.

```
def check_result(psi: state.State, a, b,
                 nbits: int, factor: float = 1.0) -> None:
  maxbits, _ = psi.maxprob()
  result = helper.bits2val(maxbits[0:nbits][::-1])
  assert result == a + factor * b, 'Incorrect addition.'
```

11.5.1 Adding a Constant

Adding a *known constant* value to a quantum state representing a binary number does not require a second quantum register, as in the general case of addition. We can precompute the rotation angles and apply them directly as if they were controlled by a second register that holds that constant. To precompute the required angles, we use this function:

```
def precompute_angles(a: int, n: int) -> List[float]:
  angles = [0.0] * n
  for i in range(n):
    for j in range(i, n):
      if (a & (1 << n - j - 1)):
        angles[n - i - 1] += 2 ** (-(j - i))
    angles[n - i - 1] *= math.pi
  return angles
```

We modify the `evolve` step and add the precomputed rotation gates directly with U_1 gates instead of using controlled gates, as shown in the following code snippet. We will use this method later in Shor's algorithm as well.

```
  for i in range(n+1):
    qft(qc, a, n-i)
  for idx, angle in enumerate(precompute_angles(c, n)):
    qc.u1(a[idx], angle)
  for i in range(n+1):
    inverse_qft(qc, a, i)
```

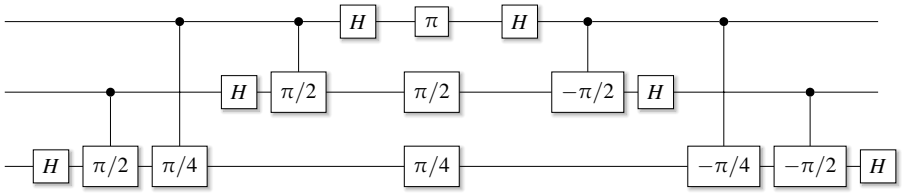


Figure 11.14 The circuit for a three-qubit addition of a constant value 1 to a quantum register. The individual rotations are precomputed; no further controlled rotations are needed. The left and right sides of the circuit are the QFT and inverse QFT operation, and the three rotations in the middle by π, $\pi/2$, and $\pi/4$ represent the evolve step.

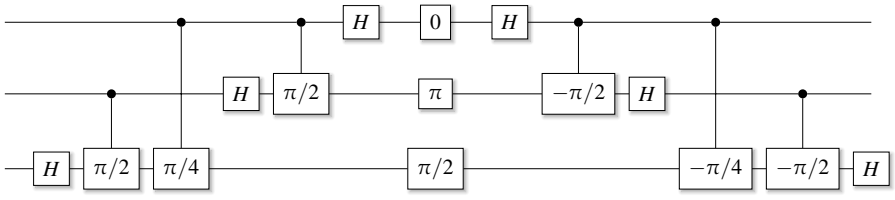


Figure 11.15 The circuit for a three-qubit addition of a constant value 2 to a quantum register. The individual rotations are precomputed and differ from the case shown in Figure 11.14; they are now 0, π, and $\pi/2$.

For example, for the three-qubit addition of a constant 1, the circuit without the b register is shown in Figure 11.14. To compare, the corresponding circuit for the addition of a constant 2 is shown in Figure 11.15. Notice the modified rotation angles in the evolution step.

11.5.2 Multiplication

So far, our methodology allows us to compute $a + cb$, for a constant c. Using this as a building block, we can implement full multiplication of two quantum registers a and b. Observe that if we write a in its binary form $a = a_{n-1}a_{n-2}\cdots a_0$, we can decompose the multiplication ab into

$$a_{n-1}2^{n-1}b + a_{n-2}2^{n-2}b + \ldots + a_1 2^1 b + a_0 2^0 b.$$

This is helpful because we know how to implement each of the subterms as an addition circuit with the terms $2^{n-1}, 2^{n-2}, \ldots, 2^1, 2^0$ as the constant factor c in the partial term cb. The complete circuit is shown in Figure 11.16, starting with term a_0. In code, we define a nested function `add_src_to_target()` that replicates the code above to calculate $a + cb$.

```
def arith_quantum_mult(nbits_a: int, init_a: int,
                       nbits_b: int, init_b: int) -> None:

  def add_src_to_targ(qc, nbits: int, src, targ, factor: float = 1.0):
    for i in range(nbits):
      qft(qc, targ, nbits - i)
```

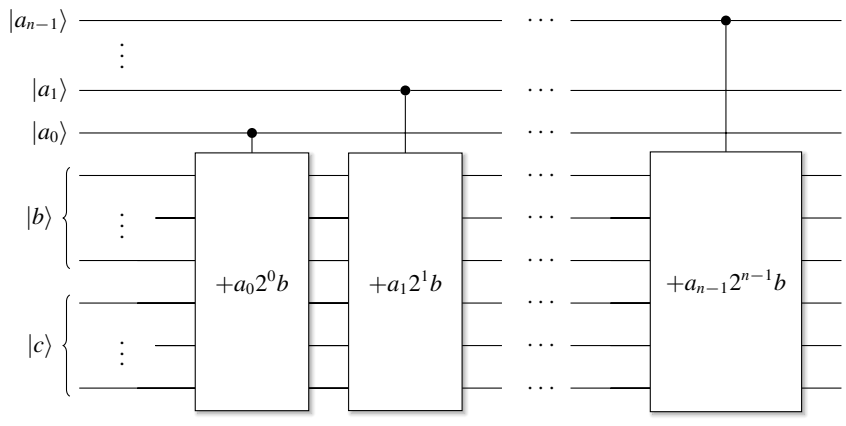


Figure 11.16 A multiplication circuit to calculate $c = ab$. The individual qubits of $|a\rangle$, interpreted as powers of 2, control additions of powers of 2 of the bit values of binary $|b\rangle$. The effects are accumulative, resulting in a multiplication.

```
  for i in range(nbits):
    evolve(qc, targ, src, nbits - i, factor)
  for i in range(nbits):
    inverse_qft(qc, targ, i)
```

The complete circuit instantiates three registers for a, b, and c, reserving enough qubits for a full multiplication result. We build the circuit as a non-eager circuit. After the registers are built, we iterate over all bits of a, generating a subcircuit to compute a subterm. We use a subcircuit because we want to control the addition circuit with a qubit in the a register. The convenience function `control_by()`, described in Section 3.4.5, makes this easy. Finally, in each iteration, we multiply `factor` by 2. Once everything is constructed, a final `qc.run()` runs the entire circuit. We follow this with our usual code to ensure that everything went according to plan.

```
qc = circuit.qc('qmult', eager=False)
a = qc.reg(nbits_a, helper.val2bits(init_a, nbits_a)[::-1])
b = qc.reg(nbits_b * 2 + 1, helper.val2bits(init_b, nbits_b)[::-1])
c = qc.reg(nbits_b * 2 + 1, 0)

factor = 1.0
for idx in range(nbits_a):
  sc = qc.sub()
  add_src_to_targ(sc, nbits_b * 2, b, c, factor)
  sc.control_by(a[idx])
  qc.qc(sc)
  factor *= 2
qc.run()

maxbits, _ = qc.psi.maxprob()
result = helper.bits2val(maxbits[c[0] : c[0 + nbits_b * 2]][::-1])
assert result == init_a * init_b, 'Incorrect multiplication'
```

11.6 Shor's Algorithm

Shor's algorithm for number factorization sparked tremendous interest in quantum computing (Shor, 1994). The Internet's RSA (Rivest, Shamir, Adleman) encryption algorithm (Rivest et al., 1978) is based on the assumption that number factoring is an intractable problem. If quantum computers could crack this code, it would obviously have severe implications.

Shor's algorithm is complex to implement. Factoring small numbers like 15 or 21 already requires a large number of qubits and many gates, on the order of many thousands. Despite this, there is still a quantum advantage. The best-known complexity for classical order finding uses a general number field sieve with subexponential complexity $\mathcal{O}\big(\exp(1.9(\log N)^{1/3}(\log\log N)^{2/3})\big)$. The best known theoretical quantum complexity is $\mathcal{O}\big((\log N)^2(\log\log N)\big)$, which puts the algorithm in the BQP complexity class (Wikipedia, 2024f). The algorithm has three main steps:

1. It has a classical part, grounded in number theory, which relies on modular arithmetic, followed by a process called order finding.
2. Order finding is classically intractable, but an efficient probabilistic quantum algorithm was discovered. This quantum part is at the heart of Shor's algorithm.
3. Once the order has been found, the prime factors can be derived.

We split the description of the algorithm into two parts. The classical part is discussed in this section. The quantum parts, including factoring, will be discussed in Section 11.7 on order finding.

11.6.1 Modular Arithmetic

Modular arithmetic is a complete arithmetic on integers that *wrap around* a given number, called the *modulus*, and considers the remainder.[9] One definition is

$$a \equiv b \bmod N \quad \Rightarrow \quad b \equiv qN + a, \text{ for some } q.$$

Equivalently,

$$a \equiv b \bmod N \quad \Rightarrow \quad a \bmod N \equiv b \bmod N.$$

Simple algebraic rules hold:

$$(x + y) \bmod N \equiv x \bmod N + y \bmod N, \tag{11.11}$$

$$(xy) \bmod N \equiv \big((x \bmod N)(y \bmod N)\big) \bmod N. \tag{11.12}$$

We can use Equations (11.11) and (11.12) to simplify computation with large numbers, for example:

$$(121 + 241) \bmod 12 \equiv 1 + 1 = 2,$$
$$(121 \cdot 241) \bmod 12 \equiv (1 \cdot 1) \bmod 12 = 1.$$

[9] The modulus *mirrors* the C++ or Python percent operators.

11.6.2 Greatest Common Divisor

We will need to calculate the *greatest common divisor* (GCD) of two integers. Recall that we get that by decomposing the numbers into their prime factors and finding the largest common factor. For example, the GCD of the integers 15 and 21 is 3, as

$$15 = 3 \cdot 5,$$
$$21 = 3 \cdot 7.$$

We compute the GCD with the famous Euclidean algorithm (Wikipedia, 2024f):

```
def gcd(a: int, b: int) -> int:
    while b != 0:
        t = b
        b = a % b
        a = t
    return a
```

11.6.3 Factorization

Now, let us see how to use modular arithmetic and the GCD to factor a large number. We are only considering numbers that have exactly two prime factors. In general, a number can potentially be factored into several prime factors p_i, as

$$N = p_0^{e_0} p_1^{e_1} \cdots p_{n-1}^{e_{n-1}},$$

but factoring is computationally most difficult if N has only two prime factors of roughly equal length.[10] This is the property upon which the RSA encryption mechanism is based. With p and q prime, we assume that

$$N = pq.$$

We can restate the factorization problem in the following interesting and perhaps surprising way. The problem of factoring the large number N into two primes is equivalent to finding nontrivial solutions to the equation

$$x^2 \equiv 1 \bmod N. \tag{11.13}$$

Below we will find that x will be of the form $x = a^{r/2}$ for a seed value a and an even "order" r, but let us not get ahead of ourselves. The two trivial solutions for Equation (11.13) are $x = 1$ and $x = -1$. But what about other solutions? For the example of $N = 21$, we can find another solution by iterating over the values $[1, \ldots, N-1]$ and searching for a value x for which Equation (11.13) holds:

[10] The typical reasons given for this are a) the absence of obvious weakness from the imbalance of factors, b) the fact that the search space cannot be reduced, and c) the complexity of the known factorization algorithms, which is measured in terms of the size of the factors.

$$
\begin{aligned}
1 \cdot 1 &= \quad 1 = \quad 1 \bmod N, \\
2 \cdot 2 &= \quad 4 = \quad 4 \bmod N, \\
3 \cdot 3 &= \quad 9 = \quad 9 \bmod N, \\
&\ldots \\
8 \cdot 8 &= 64 = \quad 1 \bmod N \quad \text{(Bingo!)}.
\end{aligned}
$$

We find that Equation (11.13) holds for $x = 8$. Now, since

$$x^2 \equiv 1 \bmod N \quad \Rightarrow \quad x^2 - 1 \equiv 0 \bmod N,$$

we can factor the left side using the quadratic formula as

$$(x+1)(x-1) \equiv 0 \bmod N.$$

The remainder of 0 means N divides the product on the left. This means we can find the prime factors $\{p,q\}$ with

$$
\begin{aligned}
p &= \gcd(N, x+1), \quad \text{and} \\
q &= \gcd(N, x-1).
\end{aligned}
$$

For the example of $N = 21$ and $x = 8$, we find $p = 3$ and $q = 7$.

This seems easy but suffers from the "little technical problem" of having to find that number x. In the classical case, our only options are to either iterate over all numbers or pick random values, square them, and check whether we find a number that, when taken modulo N, produces a 1. The choice of random values means that the birthday paradox may apply.[11] We would have a 50% chance of finding a positive result after about $\sqrt{2N}$ searches. This is intractable for the large numbers used in internet encryption with lengths of 2048 bits, 4096 bits, and higher.

11.6.4 Period Finding

We saw above that we want to find the important even order r for a seed value a with $x = a^{r/2}$, such that $x^2 = 1 \bmod N$. To find these values, we perform the following three somewhat unexpected steps to find r. Later, we will find an efficient quantum algorithm for the order finding part in Step 2:

Step 1 – Select Seed Number

We pick a random number $a < N$ that does not have a nontrivial factor[12] in common with N. We also say that a and N are *coprime*. For example, the numbers 5 and 21 are coprime, even though 21 by itself is not a prime number. This can be tested with the help of the GCD. If two numbers are coprime they do not[13] have a common factor and their GCD is 1. If our initial selection of a divides N without a remainder, we were lucky and found a factor already.

[11] See also http://en.wikipedia.org/wiki/Birthday_problem.
[12] Double negative. Perhaps it is better to say that a and N have only trivial factors in common.
[13] On the other hand, if the numbers are not coprime, their GCD will find a nontrivial factor of N.

Step 2 – Find Order

Use the function $f_{a,N}(x) = a^x \bmod N$ and iterate over x in the sequence

$$\begin{aligned} a^{x=0} \bmod N &= 1, \\ a^{x=1} \bmod N &= \ldots, \\ a^{x=2} \bmod N &= \ldots, \\ &\vdots \end{aligned}$$

Leonhard Euler showed (Euler, 1763) that for any coprime a of N (which again means that a and N have no common factors), this sequence will result in 1 for some nontrivial $x < N$. Since the sequence started with $a^{x=0} = 1$, once it hits 1 for $x > 0$, the sequence will repeat itself. For example, for $a = 7$ and $N = 15$, the sequence is

$$\begin{aligned} a^0 \bmod N &= 7^0 \bmod 15 = 1, \\ a^1 \bmod N &= 7^1 \bmod 15 = 7, \\ a^2 \bmod N &= 7^2 \bmod 15 = 4, \\ a^3 \bmod N &= 7^3 \bmod 15 = 13, \\ a^4 \bmod N &= 7^4 \bmod 15 = 1. \end{aligned}$$

The length of the sequence r (4 in the example) is called the *order*, or period, of the function. We can write this mathematically as

$$f_{a,N}(s + r) = f_{a,N}(s).$$

This is the problematic classical step. We do not know of a polynomial-time classical algorithm for it. In Section 11.7, we will learn about a quantum algorithm for this task. For now, let us just pretend that we have an efficient way to compute the order and learn how we can use it to factor a number.

Step 3 – Factor

If we find an order r that is an odd number, we give up, throw the result away, and try again with a different initial value of a in Step 1. If, on the other hand, we find an order r that is an even number, we can use what we discovered earlier. Namely, we can get the factors if we can find the x in the equation

$$x^2 \equiv 1 \bmod N.$$

We just found in step 2 above that

$$a^r \equiv 1 \bmod N.$$

If r is even, we can rewrite this as

$$\left(a^{r/2}\right)^2 \equiv 1 \bmod N.$$

In this form, we can compute the prime factors of N with the GCD as

$$p = \gcd\left(N, a^{r/2} + 1\right),$$
$$q = \gcd\left(N, a^{r/2} - 1\right).$$

There is another small (as in, actually small) caveat. We do not know whether a given initial value of a will result in an even or odd order. We cannot use odd orders because they would not lead to useful factors. It can be shown that the probability of selecting an a that produces an even order r is $1/2$. This means we might have to run the algorithm multiple times until we find an even order.

The three steps of selecting a seed number, finding the order, and factoring are the core of Shor's algorithm. As mentioned, we develop a quantum algorithm for order finding in Section 11.7. But before we get there, let us write some code and explore the concepts developed so far.

11.6.5 Playground

Let's use some random examples to experiment with what we have learned so far. We classically compute the order and derive the prime factors from it. Since the numbers are small, the problems are still tractable. Let us first develop some helper functions. When choosing a random number `num` to play with, we must make sure that it is not prime and can be factored. To check for primality,[14] we iterate over all the odd numbers starting at 3 and confirm that none of them divides the candidate number `num`.

Find the code
In file src/shor_classic.py

```
def is_prime(num: int) -> bool:
  for i in range(3, num // 2, 2):
    if num % i == 0:
      return False
  return True
```

The algorithm requires picking a random number seed, which must be coprime to the larger number (no common factors).

```
def is_coprime(num: int, larger_num: int) -> bool:
  return math.gcd(num, larger_num) == 1
```

We find a random, odd, and nonprime number in the range of numbers from `fr` to `to`. We also add a routine to find a coprime:

[14] It's a word. I checked.

```
def get_odd_non_prime(fr: int, to: int) -> int:
  while True:
    n = random.randint(fr, to)
    if n % 2 == 0:
      continue
    if not is_prime(n):
      return n

def get_coprime(larger_num: int) -> int:
  while True:
    val = random.randint(3, larger_num - 1)
    if is_coprime(val, larger_num):
      return val
```

Finally, we will need a routine to compute the order of a given modulus. The code below iterates until it finds a result of 1 (which is guaranteed to exist).

```
def classic_order(num: int, modulus: int) -> int:
  order = 1
  while 1 != (num ** order) % modulus
    order += 1
  return order
```

For the main experiments, we first select a random N and a coprime a, as described above. N is the number to factorize; it must not be prime or divisible by 2. Once we have the numbers, we classically compute the order. Once the order is found, we compute the factors from it and check the results:

```
def run_experiment(fr: int, to: int) -> (int, int):
  n = get_odd_non_prime(fr, to)
  a = get_coprime(n)
  order = classic_order(a, n)

  factor1 = math.gcd(a ** (order // 2) + 1, n)
  factor2 = math.gcd(a ** (order // 2) - 1, n)
  if factor1 == 1 or factor2 == 1:
    return None

  print('Found Factors: N = {:4d} = {:4d} * {:4d} (r={:4})'.
        format(factor1 * factor2, factor1, factor2, order))
  assert factor1 * factor2 == n, 'Invalid factoring'
  return factor1, factor2
```

We run a number of tests and should find results as follows. Even for small random numbers of up to 9,999, the order can already be quite large.

```
def main(argv):
  print('Classic Part of Shor\'s Algorithm.')
  for i in range(25):
    run_experiment(21, 9999)
>>
Classical Part of Shor's Algorithm.
Found Factors: N = 3629 =  191 *   19 (r=1710)
Found Factors: N = 4295 =    5 *  859 (r=1716)
Found Factors: N = 9023 = 1289 *    7 (r=3864)
[...]
```

In summary, we have examined how to factorize a number N into two prime factors using order finding and modular arithmetic. However, we also find that classical order finding for large numbers is intractable. Thankfully, an efficient quantum algorithm has been discovered for this purpose, which we will discuss next.

11.7 Order Finding

So far, we have learned how finding the order of a specific function classically lets us find two prime factors. In this section, we discuss an effective quantum algorithm to improve on the classical task. We start by restating the objective in a slightly different way: We want to find the phase of one particular operator. It may not be immediately clear how this pertains to determining the order, but no worries, we will elaborate on all the details in the subsequent sections.

Quantum order finding boils down to estimating the phase for the operator U_x, which is defined as

$$U_x|y\rangle = |xy \bmod N\rangle. \tag{11.14}$$

Here, x plays the role of the value a from Section 11.6.4, where we exponentiated a with increasing integer exponents and computed the modulus until we found a remainder of 1. We can also see that there is a modulus operation in the operator. We will have to find a quantum way to implement this operation.

To begin, let us first find the form of the eigenvalues of this operator. We use a process similar to the power iteration process from Section 11.6.4. We know that the eigenvalues must have norm 1, otherwise the state probabilities would not sum up to 1. We also know that the eigenvalues are defined as $U|v\rangle = \lambda|v\rangle$. Hence, we can state:

$$U^k|v\rangle = \lambda^k|v\rangle,$$

and, substituting this into the operator of Equation (11.14), we get

$$U^k|y\rangle = \left|x^k y \bmod N\right\rangle.$$

If r is the order of $x \bmod N$, then $x^r = 1 \bmod N$, and

$$U^r|y\rangle = \lambda^r|y\rangle = \left|x^r y \bmod N\right\rangle = |y\rangle.$$

In turn, this allows us to derive that $\lambda^r = 1$. This means that the eigenvalues of U are rth *roots of unity*, which are complex numbers that yield 1 when raised to some integer power n. They are defined as

$$\lambda_s = e^{2\pi i s/r}, \quad \text{for } s = 0, \ldots, r-1.$$

Using phase estimation, we will find these eigenvalues. The final trick will be to get to the order r from the fraction s/r.

There is, of course, an initialization problem. For the phase estimation circuit to work, we need to know an eigenvector. In the following, we will show that the eigenvectors of the operator U with order r, a value s with $0 \leq s < r$, and a seed value a, are

$$|v_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i k s/r} \left|a^k \bmod N\right\rangle.$$

However, because we do not yet know the order r, we do not know any of the eigenvectors. Here comes another smart trick. We can see that the operator U from Equation (11.14) is a permutation operator. How does this work? Following the pattern of modular arithmetic, given states are uniquely mapped to different states with order r. Let us interpret the states as integers, with state $|1\rangle$ representing decimal 1, and state $|1001\rangle$ representing decimal 9. For all values less than r, this mapping is a 1:1 mapping – each input state maps to a unique output state, which ensures that no two input states map to the same output state. For the operator

$$U|y\rangle = |xy \bmod N\rangle,$$

we see that state $|y\rangle$ is multiplied by $x \bmod N$. As we iterate over exponents, this becomes

$$U^n|y\rangle = |x^n y \bmod N\rangle.$$

For example, with $x = 2$ and $N = 21$, each application multiplies the state of the input register by $2 \bmod N$. We started with $2^0 = 1 = 1 \bmod N$, which corresponds to state $|1\rangle$. Then

$$\begin{aligned}
U^1|1\rangle &= |2\rangle, \\
U^2|1\rangle &= UU|1\rangle = U|2\rangle = |4\rangle, \\
U^3|1\rangle &= |8\rangle, \\
U^4|1\rangle &= |16\rangle, \\
U^5|1\rangle &= |11\rangle, \\
U^6|1\rangle &= U^r|1\rangle = |1\rangle.
\end{aligned}$$

We still need to find an eigenvector but discover that the first eigenvector of this operator is in *superposition of all states*. This may be surprising but is easy to understand from another simple example.[15] Let us take the unitary X gate, which only permutes between the two states $|0\rangle$ and $|1\rangle$, with

[15] http://quantumcomputing.stackexchange.com/a/15590.

$$X|0\rangle = |1\rangle \quad \text{and} \quad X|1\rangle = |0\rangle.$$

Applying X to the *superposition* of these basis states leads to the following result with an eigenvalue of 1:

$$\begin{aligned} X\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) &= \frac{X|0\rangle + X|1\rangle}{\sqrt{2}} \\ &= \frac{|1\rangle + |0\rangle}{\sqrt{2}} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \\ &= 1\ \frac{|0\rangle + |1\rangle}{\sqrt{2}}. \end{aligned}$$

Now that we have found at least one plausible eigenvector, we can generalize the operator in Equation (11.14) to multiple basis states. As shown, the superposition of the basis states is an eigenvector of U with eigenvalue 1:

$$|u_1\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \left|a^k \bmod N\right\rangle.$$

We also found that the eigenvalues are

$$\lambda_s = e^{2\pi i s/r}, \quad \text{for } s = 0, \ldots, r-1.$$

We can introduce a factor k to define eigenstates where the phase of the kth basis state is proportional to k as

$$|u_1\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i k/r} \left|a^k \bmod N\right\rangle. \tag{11.15}$$

For our example, applying the operator to this eigenvector follows the permutation rules of the operator U ($|1\rangle \to |2\rangle, |2\rangle \to |4\rangle, \ldots$):

$$|u_1\rangle = \frac{1}{\sqrt{6}} \left(|1\rangle + e^{2\pi i/6}|2\rangle + e^{4\pi i/6}|4\rangle + e^{6\pi i/6}|8\rangle + e^{8\pi i/6}|16\rangle + e^{10\pi i/6}|11\rangle\right),$$

$$U|u_1\rangle = \frac{1}{\sqrt{6}} \left(|2\rangle + e^{2\pi i/6}|4\rangle + e^{4\pi i/6}|8\rangle + e^{6\pi i/6}|16\rangle + e^{8\pi i/6}|11\rangle + e^{10\pi i/6}|1\rangle\right).$$

We can pull out the factor $e^{-2\pi i/6}$ to arrive at:

$$\begin{aligned} U|u_1\rangle &= \frac{1}{\sqrt{6}} e^{-2\pi i/6} \left(e^{\frac{2\pi i}{6}}|2\rangle + e^{\frac{4\pi i}{6}}|4\rangle + e^{\frac{6\pi i}{6}}|8\rangle + e^{\frac{8\pi i}{6}}|16\rangle + e^{\frac{10\pi i}{6}}|11\rangle + \underbrace{e^{\frac{12\pi i}{6}}}_{=1}|1\rangle\right) \\ &= e^{-2\pi i/6}|u_1\rangle. \end{aligned}$$

Note how the order $r = 6$ now appears in the denominator. To make this general for all eigenvectors, we multiply in a factor s in the exponent, thereby obtaining:

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i k s/r} \left|a^k \bmod N\right\rangle.$$

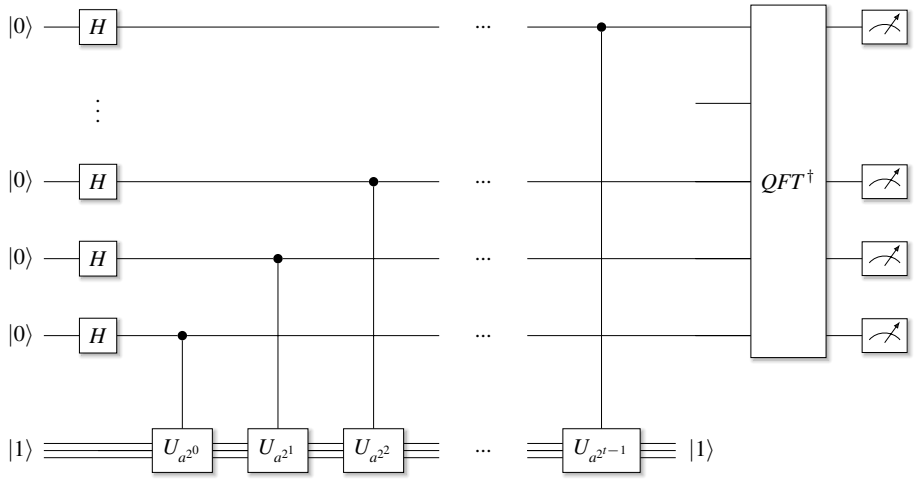


Figure 11.17 Order finding circuit, which consists of phase estimation for the operator U followed by an inverse QFT operation.

As a result, for our operator, we now get a unique eigenvector for each integer $s = 0, \ldots, r-1$, with the following eigenvalues (note that if we added the minus sign to Equation (11.15), the minus sign here would disappear; we can ignore it):

$$e^{-2\pi i s/r}|u_s\rangle. \tag{11.16}$$

There is another important result here: If we add all these eigenvectors, the phases cancel out except for $|1\rangle$ (not shown here; it is voluminous but not challenging). This helps us because now we can use $|1\rangle$ as the eigenvector input to the phase estimation circuit. There, phase estimation for any of the eigenvalues in Equation (11.16) will give us the following result:

$$\phi = \frac{s}{r}.$$

But why can we use $|1\rangle$ to initialize the phase estimation? Here is an answer:[16] Phase estimation should work for one eigenvector/eigenvalue pair. But in this case, we initialize the circuit with the sum of all eigenvectors, which we can consider as the *superposition* of all eigenstates. On measurement, the state will collapse to one of them. We do not know which one, but we know from the above that it will have a phase $\phi = s/r$. This is all we need to find the order with the method of continued fractions.

With all these preliminaries, we can now construct a phase estimation circuit as shown in Figure 11.17. The big challenge is how to implement the large unitary operator U. Our solution is based on a paper by Stephane Beauregard (Beauregard, 2003) and a reference implementation by Tiago Leao and Rui Maia (Leao and Maia 2021). The implementation is quite complicated. Fortunately, we are already familiar with many of the building blocks.

[16] http://quantumcomputing.stackexchange.com/q/15589.

In our implementation, which is not highly optimized and far from theoretical limits, we need 22 qubits and more than 20,000 gates to factor the number 21. Because there are many QFTs and uncomputations, the number of gates increases rapidly. With our accelerated implementation, we can still simulate this circuit tractably for smaller numbers requiring about five classical binary bits. The overall implementation consists of about 250 lines of Python code.

As with all oracles or high-level unitary operators, you might expect some quantum trick, a specially crafted matrix that just happens to compute the modulo exponentiation. Unfortunately, that magical matrix does not exist. Instead, we have to compute the exponentiation explicitly with quantum gates by implementing addition and multiplication (by a constant) in the Fourier domain. We also have to implement the modulo operation, which we have not seen before.

We describe the implementation as follows. First, we outline the main routine driving the whole process. Then, we describe the helper routines, e.g., for addition. We have seen most of these before in other sections. Finally, we describe the code that implements the unitary operators, which we connect in a phase estimation circuit. We finally get experimental results from the estimated phase with the help of continued fractions.

11.7.1 Main Program

In the implementation, we get the number N to factor and the seed value a as command-line parameters. From these values, we compute the required bit width and construct three registers.

- `aux` for ancillae.
- `up` is the top register in the circuit shown in Figure 11.17. We will compute the inverse QFT on this register to get the phase estimation.
- `down` is the register that will hold the unitary operators. We initialize it to state $|1\rangle$, which we can think of as the superposition of all eigenvectors, as explained above.

PY

Find the code
In file `src/order_finding.py`

```
def main(argv):
  number = flags.FLAGS.N
  a = flags.FLAGS.a
  nbits = number.bit_length()
  print('Shor: N = {}, a = {}, n = {} -> qubits: {}'
        .format(number,         a, nbits, nbits*4 + 2))

  qc   = circuit.qc('order_finding')
  aux  = qc.reg(nbits+2)
  up   = qc.reg(nbits*2)
  down = qc.reg(nbits)
```

We follow this with a one-to-one implementation of the circuit diagram in Figure 11.17. We apply Hadamard gates to the `up` register and a single X gate to the top qubit of the `down` register to initialize it as $|1\rangle$. Note that to stay close to the reference implementation (Leao, 2021), we interpret `down` in reverse order. Then, we iterate over the number of `up` bits `(nbits * 2)` and create and connect the unitary gates with the controlled Multiply-Modulo routine `cmultmodn`. All of this is then followed by a final inverse QFT:

```
qc.h(up)
qc.x(down[0])
for i in range(nbits*2):
  cmultmodn(qc, up[i], down, aux, int(a**(2**i)), number, nbits)
inverse_qft(qc, up, 2*nbits, with_swaps=1)
```

Finally, we check the results. For the example numbers given ($N = 15$, $a = 4$), we expect the highest probability end states to correspond to a result of 128 or 0 in the `up` register, corresponding to interpretations as binary fractions of 0.5 and 0.0. We will detail the steps necessary to get to the factors from these fractions at the end of this section. Note again that we again inverted the order of the qubits with `[::-1]`.

```
for bits in helper.bitprod(nbits * 4 + 2):
  prob = qc.psi.prob(*bits)
  if prob > 0.01:
    bitslice = bits[nbits + 2 : nbits + 2 + nbits * 2][::-1]
    intval = helper.bits2val(bitslice)
    phase = helper.bits2frac(bitslice)

    [... compute fractions here]

  total_prob += prob
  if total_prob > 0.999:
    break
```

As we measure, we will find the correct factors with a probability of 50% (or less). The algorithm is probabilistic. On a real machine, we might find only factors 1 and N and have to run the algorithm multiple times until we find at least one of the other prime factors. In our infrastructure, of course, we can just peek at the resulting probabilities without the need to run multiple times.

```
[...]
Swap...
Uncompute...
Measurement...
Final x-value. Got:   0 Want: 128, probability: 0.250
Final x-value. Got:   0 Want: 128, probability: 0.250
Final x-value. Got: 128 Want: 128, probability: 0.250
Final x-value. Got: 128 Want: 128, probability: 0.250
```

11.7.2 Support Routines

We use the variable a to compute a modulo number. Since we have to perform uncomputation, we need the *modulo inverse* of this number. The modulo inverse of $x \bmod N$ is the number x_{inv}, such that $xx_{\text{inv}} = 1 \bmod N$. We can compute this number with the help of the extended Euclidean algorithm (Wikipedia, 2021c):

```
def modular_inverse(a: int, m: int) -> int:
  """Compute Modular Inverse."""

  def egcd(a: int, b: int) -> (int, int, int):
    if a == 0:
      return (b, 0, 1)
    else:
      g, y, x = egcd(b % a, a)
    return (g, x - (b // a) * y, y)

  # Modular inverse of x mod m is the number x^-1 such that
  #   x * x^-1 = 1 mod m
  g, x, _ = egcd(a, m)
  assert g == 1, f'Modular inverse ({a}, {m}) does not exist.'
  return x % m
```

Our implementation of the algorithm requires a large number of QFTs and inverse QFTs. Many of these operations are part of adding a constant to a quantum register. We saw in Section 11.5.1 on quantum arithmetic how to precompute the angles with this routine:

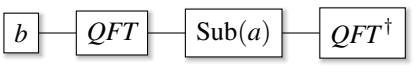
```
def precompute_angles(a: int, n: int) -> List[float]:
  angles = [0.0] * n
  for i in range(n):
    for j in range(i, n):
      if (a & (1 << n - j - 1)):
        angles[n - i - 1] += 2 ** (-(j - i))
    angles[n - i - 1] *= math.pi
  return angles
```

We will need circuitry to compute addition, controlled addition, and double-controlled addition. We implement constant addition in `add` using the `u1` gate. The controlled addition in `cadd` uses the controlled gate `cu1`, and the double-controlled addition in `ccadd` uses the double-controlled gate `ccu1`.

```
def add(qc, q, a: int, n: int, factor: float) -> None:
  for idx, angle in enumerate(precompute_angles(a, n)):
    qc.u1(q[idx], factor * angle)

def cadd(qc, q, ctl, a: int, n: int, factor: float) -> None:
  for idx, angle in enumerate(precompute_angles(a, n)):
    qc.cu1(ctl, q[idx], factor * angle)
```

```
def ccadd(qc, q, ctl1: int, ctl2: int, a: int, n: int,
          factor: float) -> None:
  for idx, angle in enumerate(precompute_angles(a, n)):
    qc.ccu1(ctl1, ctl2, q[idx], factor * angle)
```

Using the fact that a subtraction circuit is the adjoint of an addition circuit, $\text{Sub}(a) = \text{Add}^{\dagger}(a)$, we get $(b - a)$ if $b \geq a$, and $(2^{n-1} - (a - b))$ if $b < a$. So we can use this to subtract and compare numbers. If $b < a$, then the most significant qubit will be $|1\rangle$. We will use this qubit to control other gates later.



$$= \begin{cases} |b-a\rangle & \text{if } b \geq a, \\ |2^{n-1} - (a-b)\rangle & \text{if } b < a. \end{cases}$$

For QFT and inverse QFT operations, we reuse the `qft` and `inverse_qft` functions that we implemented in Section 11.4.2 in the `circuit` class. We will perform QFT on partial registers, so we provide wrappers to these functions that allow us to specify how many qubits in a register to operate on. We use the Python slice operator since our registers are conveniently just Python lists of indices.

```
def qft(qc, up_reg, n: int, with_swaps: bool = False) -> None:
  qc.qft(up_reg[:n], with_swaps)

def inverse_qft(qc, up_reg, n: int, with_swaps: bool = False) -> None:
  qc.inverse_qft(up_reg[:n], with_swaps)
```

11.7.3 Modular Addition

At this point, we know how to add numbers and check whether a value has turned negative by checking the sign qubit. This means that we should have all the necessary ingredients for *modular* addition: We compute $a + b$ and subtract N if $a + b > N$.

We achieve this by adding an ancilla in the initial state $|0\rangle$. We start by adding a and b as before. We also reserve an overflow bit. Then, we use the adjoint of the adder to subtract N (a fancy way of saying that we apply a negative factor in the addition routines above).

To get to the most significant qubit and determine if this result was negative, we have to perform the inverse QFT. We connect the most significant qubit and the ancilla with a controlled Not gate. It will only be set to $|1\rangle$ if $a + b - N$ is negative. After this, we go back to the Fourier domain with another QFT. If $a + b - N$ is negative, we use the ancilla qubit to control the addition of N to make the result positive again. The circuit is shown in Figure 11.18.

There is a resulting problem that is not easy to solve – the ancilla qubit is still entangled. It has turned into a junk qubit. We have to find a way to return it to its original state of $|0\rangle$, otherwise it will mess up our results (as junk qubits have a habit of doing).

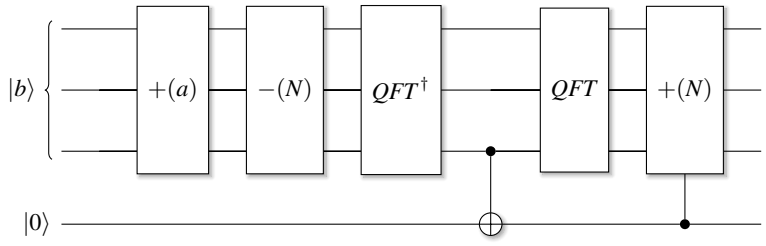


Figure 11.18 First half of the modular addition circuit.

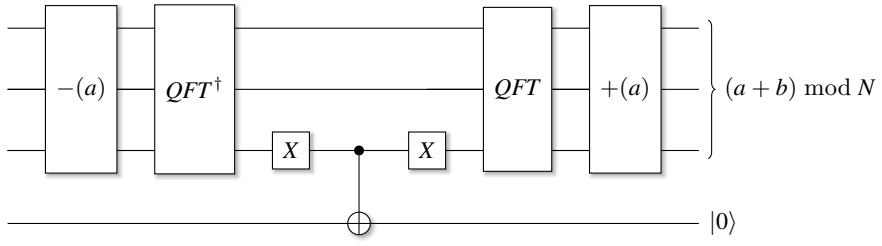


Figure 11.19 Second half of the modulo addition circuit, disentangling the ancilla. Note that this implementation uses the little-endian convention with the most significant qubit being at the bottom.

To resolve this, we use an almost identical circuit again, but with a twist. We observe that, after the modulo operation, if the remainder is larger than a, then

$$(a+b) \bmod N \geq a \quad \Rightarrow \quad a+b < N.$$

We change the circuit and, this time, run an inverse addition to subtract a from the above result and compute $(a+b) \bmod N - a$. The most significant bit will be $|0\rangle$ if $(a+b) \bmod N \geq a$. We apply a NOT gate and use it as the controller for a controlled Not to the ancilla. With this, the ancilla has been restored.

Now we have to uncompute what we just did. To do that, we apply another NOT gate to the most significant qubit, followed by a QFT and an addition of a to reverse the initial subtraction. The end result is a clean computation of $(a+b) \bmod N$. In circuit notation, the second half of the circuit is shown in Figure 11.19. In code:

```
def cc_add_mod_n(qc, q, ctl1, ctl2, aux, a, number, n):
    """Circuit that implements doubly controlled modular addition by a."""

    ccadd(qc, q, ctl1, ctl2, a, n, factor=1.0)
    add(qc, q, number, n, factor=-1.0)
    inverse_qft(qc, q, n, with_swaps=0)
    qc.cx(q[n-1], aux)
    qft(qc, q, n, with_swaps=0)
    cadd(qc, q, aux, number, n, factor=1.0)

    ccadd(qc, q, ctl1, ctl2, a, n, factor=-1.0)
```

```
  inverse_qft(qc, q, n, with_swaps=0)
  qc.x(q[n-1])
  qc.cx(q[n-1], aux)
  qc.x(q[n-1])
  qft(qc, q, n, with_swaps=0)
  ccadd(qc, q, ctl1, ctl2, a, n, factor=1.0)
```

For uncomputation, we will also need the inverse of this procedure. As explained in Section 2.12, we apply the inverse gates in reverse order:

```
def cc_add_mod_n_inverse(qc, q, ctl1, ctl2, aux, a, number, n):
  """Inverse of the double-controlled modular addition."""

  ccadd(qc, q, ctl1, ctl2, a, n, factor=-1.0)
  inverse_qft(qc, q, n, with_swaps=0)
  qc.x(q[n-1])
  qc.cx(q[n-1], aux)
  qc.x(q[n-1])
  qft(qc, q, n, with_swaps=0)
  ccadd(qc, q, ctl1, ctl2, a, n, factor=1.0)

  cadd(qc, q, aux, number, n, factor=-1.0)
  inverse_qft(qc, q, n, with_swaps=0)
  qc.cx(q[n-1], aux)
  qft(qc, q, n, with_swaps=0)
  add(qc, q, number, n, factor=1.0)
  ccadd(qc, q, ctl1, ctl2, a, n, factor=-1.0)
```

Uncomputing circuits like this is tedious. In Section 3.4.3 we showed how to automate uncomputation in an elegant way. However, that relatively simple method will not work here because we use a coprime number in the computation step. In order to uncompute, we must use its modular inverse. This value is not apparent and hence not available in the automated uncomputation infrastructure.

11.7.4 Controlled Modular Multiplication

The next step is to build a controlled modular multiplier from the modular adders we just constructed. Our circuit will be controlled by a qubit $|c\rangle$ and take the state $|c,x,b\rangle$ to the state $|c,x,b + (ax) \bmod N\rangle$, if $|c\rangle = |1\rangle$. Otherwise, it will leave the original state intact.

We perform successive applications of the controlled modular addition gate, controlled by the individual bits x_i of x, as shown in Figure 11.20. The bit positions correspond to powers of 2 in the identity

$$(ax) \bmod N =$$
$$\left(\ldots \left((2^0 a x_0) \bmod N + 2^1 a x_1\right) \bmod N + \cdots + 2^{n-1} a x_{n-1}\right) \bmod N.$$

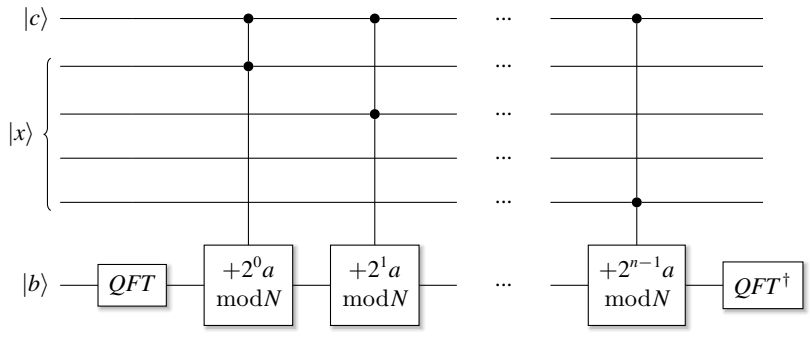


Figure 11.20 Circuit for controlled modular multiplication.

As described in Section 2.12 on uncomputation, to eliminate the entanglement with $|b\rangle$, we swap out $|x\rangle$ and uncompute the circuit after the swap. In the code, we see three sections. In the first section, it computes the multiplication modulo N. In the second block, it connects the results with controlled gates to swap out $|x\rangle$ to the `aux` register (the `cswap` was introduced in Section 3.3 on quantum circuits). Finally, it uncomputes the results. This means we must implement the inverse computation of the first block using the modular inverse.

```
def cmultmodn(qc, ctl, q, aux, a, number, n):
  """Controlled Multiply of q by number, with n bits."""

  print('Compute...')
  qft(qc, aux, n+1, with_swaps=0)
  for i in range(n):
    cc_add_mod_n(qc, aux, q[i], ctl, aux[n+1],
                 ((2**i)*a) % number, number, n+1)
  inverse_qft(qc, aux, n+1, with_swaps=0)

  print('Swap...')
  for i in range(n):
    qc.cswap(ctl, q[i], aux[i])
  a_inv = modular_inverse(a, number)

  print('Uncompute...')
  qft(qc, aux, n+1, with_swaps=0)
  for i in range(n-1, -1, -1):
    cc_add_mod_n_inverse(qc, aux, q[i], ctl, aux[n+1],
                         ((2**i)*a_inv) % number, number, n+1)
  inverse_qft(qc, aux, n+1, with_swaps=0)
```

In summary, the modular multiplication circuit performs:

$$|x\rangle|0\rangle \rightarrow |ax \bmod N\rangle|0\rangle.$$

We shall name this circuit CU_a. There is still a problem – the phase estimation algorithm requires powers of 2 of this circuit. Does this mean we have to multiply

this circuit n times by itself to get to $(CU_a)^n$ for each power of 2 as required by phase estimation? Fortunately, we do not. We simply compute a^n classically and use

$$(CU_a)^n = CU_{a^n}.$$

This can be seen at the top level in the code, where we iterate over the calls to the modular arithmetic circuit (the expressions containing `2**i`).

11.7.5 Continued Fractions

We are close to the finish line. We mentioned in Section 11.7.1 that an expected result for the `up` register was 128. This was an interpretation of this register as an integer. However, we performed phase estimation, so we have to interpret the bits of the register as binary fractions. A value of 0 corresponds to a phase of 0.0, and a value of 128 corresponds to a phase of 0.5. We also know that phase estimation will give a phase of the following form with order r:

$$\phi = \frac{s}{r}.$$

This means that if we find a fraction of integers that approximate this phase, we would have an initial guess of the order r. To approximate a fractional value to an arbitrary degree of precision, we can use the technique of *continued fractions*.[17] Fortunately, an implementation of it already exists in the `fractions` Python library. We first decode the x-register as a binary fraction using the helper function `bits2frac`. Then we obtain the lowest denominator from the continued fractions algorithm. We need to limit the accuracy via `limit_denominator` to ensure that we get reasonably sized denominators:

```
import fractions
[... loop over high-probability states from above]
    bitslice = bits[nbits + 2 : nbits + 2 + nbits * 2][::-1]
    intval = helper.bits2val(bitslice)
    phase = helper.bits2frac(bitslice)

    r = fractions.Fraction(phase).limit_denominator(8).denominator
    guesses = [math.gcd(a ** (r // 2) - 1, number),
               math.gcd(a ** (r // 2) + 1, number)]
    print('Final x: {:3d} phase: {:3f} prob: {:.3f} factors: {}'.
          format(intval, phase, prob.real, guesses))
```

With this `r`, we can then follow the explanations on the non-quantum part of Shor's algorithm and compute the factors. We might just get 1s or Ns, which do not help us. However, with a little luck (and the relevant probabilities), we might find one or both of the real factors.

[17] http://en.wikipedia.org/wiki/Continued_fraction.

11.7.6 Experiments

Let's run just a few examples to demonstrate the working machinery. To factorize 15 with a seed value $a = 4$, we run a circuit with 10,553 gates and obtain two sets of factors, the trivial ones with 1 and 15, but Eureka! also the real factors of 3 and 5:

```
.../order_finding -- --a=4 --N=15
Final x-value int:   0 phase: 0.000000 prob: 0.250 factors: [15, 1]
Final x-value int: 128 phase: 0.500000 prob: 0.250 factors: [3, 5]
Circuit Statistics
  Qubits: 18
  Gates : 10553
```

To factor 21 with $a = 5$, the required number of qubits increases from 18 to 22, increasing the number of gates to over 20,000. The run time increases roughly by a factor of 8. In addition to trivial factors, at least the routine finds the value 3 as one of the real factors:

```
Final x-value int:   0 phase: 0.000000 prob: 0.028 factors: [21, 1]
Final x-value int: 512 phase: 0.500000 prob: 0.028 factors: [1, 3]
Final x-value int: 853 phase: 0.833008 prob: 0.019 factors: [1, 21]
Final x-value int: 171 phase: 0.166992 prob: 0.019 factors: [1, 21]
Final x-value int: 683 phase: 0.666992 prob: 0.019 factors: [1, 3]
Circuit Statistics
  Qubits: 22
  Gates : 20671
```

Finally, factoring 35 with a seed $a = 4$ uses over 36,000 gates and requires a runtime of approximately 60 minutes on a standard-issue laptop:

```
Final x-value int:    0 phase: 0.000000 prob: 0.028 factors: [35, 1]
Final x-value int: 2048 phase: 0.500000 prob: 0.028 factors: [1, 5]
Final x-value int: 1365 phase: 0.333252 prob: 0.019 factors: [1, 5]
Final x-value int: 3413 phase: 0.833252 prob: 0.019 factors: [7, 5]
Final x-value int:  683 phase: 0.166748 prob: 0.019 factors: [7, 5]
Final x-value int: 2731 phase: 0.666748 prob: 0.019 factors: [1, 5]
Circuit Statistics
  Qubits: 26
  Gates : 36373
```

You may want to experiment and perhaps transpile the code to `libq` with the transpilation facilities described in Section 3.4.7. The code runs significantly faster in `libq`, which allows experimentation with a much larger number of qubits. As a rough and unscientific estimate, factorization with 22 qubits runs for about two minutes on a standard workstation. After compilation to `libq`, it runs much faster due to the sparse representation and takes less than 5 seconds to complete. This is a speed-up factor of more than 25 times! Factoring 35 with 26 qubits accelerates from about an hour to about three minutes with `libq`, still a significant speed-up of about 20 times.

To summarize, the algorithm as a whole – from the classical parts to the quantum parts to finding the order with continued fractions – is truly magical. No wonder it has received so much attention and stands out as one of the key contributors to today's interest in quantum computing.

What may be even more exciting is that progress has not stood still. Shor's algorithm is usually estimated to require at least $\mathcal{O}\big(\log(n)^2 \log\log n\big)$ gates. The algorithm we outlined above requires $2n + 3$ qubits and $\mathcal{O}\big(n^3 \log_2 n\big)$ gates. Recently, Regev (2024) showed that Shor's algorithm could be computed by running a circuit with just $\mathcal{O}\big(n^{3/2}\big)$ gates approximately $\sqrt{n} + 4$ times, representing a significant improvement.

Should we be concerned that quantum computers will crack RSA-2048 soon? As of this writing, quantum computers operate with only tens or hundreds of qubits and without full error correction, making them incapable of handling such complex computations. Fully error-corrected qubits, which are required to reduce noise and decoherence, would vastly increase the number of physical qubits and gates needed, pushing "cracking the code" even further beyond the capabilities of quantum hardware today.[18]

[18] This paragraph may be a candidate for the category of "Famous Last Words."

12 Quantum Walk Algorithms

In this chapter, we briefly introduce quantum random walks, which are quite different from classical random walks. There is a large class of problems that can be solved with this technique, but here we focus on basic principles only.

12.1 Quantum Random Walk

A *classical random walk* describes a process of random movement in a given topology, such as moving randomly left or right on a number line, left/right and up/down on a 2D grid, or along the edges of a graph. Random walks accurately model an extensive range of real-world phenomena in disciplines as diverse as physics, chemistry, economics, and sociology. In computer science, random walks are effectively used in randomized algorithms. Some of these algorithms have a lower computational complexity than previously known deterministic algorithms.

Random walks have fascinating properties. For example, assume that two random walkers start their journey at the same location on a 2D grid. Will the walkers meet again in the future, and if so, how often? The answer is yes, they will meet again, and furthermore, they will meet again infinitely many times.

A *quantum random walk* is the quantum analog of a classical random walk (Kempe, 2003), but of course, adding quantum mechanics makes things more interesting. In a quantum walk on a grid, the walker, due to superposition, exists in multiple states and moves in all directions simultaneously, taking all possible paths. Furthermore, the paths can interfere with each other.

To achieve this, the quantum walker needs an extra degree of freedom, often called a "coin," which determines the direction of the movement. The coin is a quantum system, existing in a superposition of states, allowing the walker to move in all directions simultaneously. This leads to some unique properties:

- A quantum walker spreads across the grid much faster than a classical walker.
- The probability of finding the quantum walker at a specific location after a certain number of steps creates very different patterns from a classical walk.

Specific problems, such as the glued tree algorithm developed by Childs et al. (2003, 2009), cannot be computed in a tractable way on a classical machine. Herein lies the great interest in quantum random walks: Some of these intractable problems become

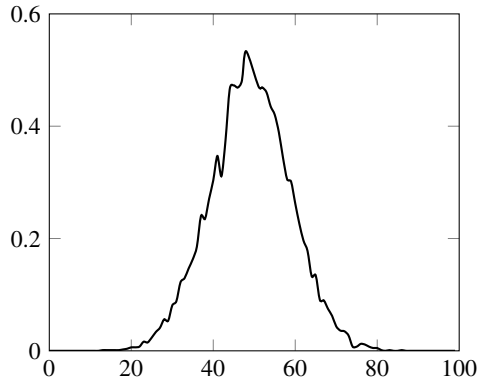


Figure 12.1 Results from a limited number of simulated classical random walks, plotting the likelihood of final position after starting in the middle of the range.

tractable on a quantum machine. In this section, we further explore this phenomenon and touch on basic principles, such as how probabilities propagate through a topology.

12.1.1 1D Walk

Let us start in the simplest of scenarios by considering a classical 1-dimensional walk on a number line. For each step, the toss of a fair coin determines whether to move left or right. After many moves, the probability distribution of the final location will be shaped like a classic bell curve, with the highest probability clustering around the origin of the journey.[1] Figure 12.1 shows the result of a simple experiment.[2]

PY

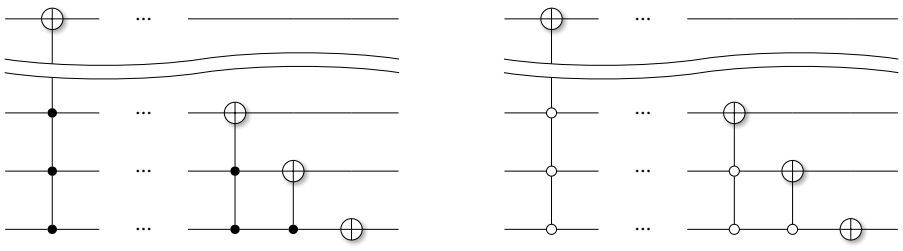
Find the code
In file `src/tools/random_walk.py`

The equivalent quantum walk operates in an analogous fashion with coin tosses and movements. Because this is quantum, we exploit the superposition and move in both directions at the same time. In short, a quantum random walk is the repeated application of an operator $U = MC$, with C being the coin toss followed by the move operator M. What are these unitary operators C and M?

The most straightforward coin-toss operator may just be a single Hadamard gate. In this context, the coin is called a *Hadamard coin*. The $|0\rangle$ part of the resulting superposition will control a movement to the left, and the $|1\rangle$ part controls a movement to the right.

The movement circuits can be constructed as shown in Douglas and Wang (2009). A number line has infinite length, which cannot be adequately represented in a quantum state. We simplify and assume that the underlying topology for the walk is a circle with N stops on it. Each stop represents one of the N computational basis states.

[1] A biased coin would lead to a skewed distribution.
[2] In fairness, the curve simply reflects the random number distribution chosen for the experiment.



(a) An n-qubit increment operator. (b) An n-qubit decrement operator.

Figure 12.2 Increment and decrement operators for quantum walks.

Simple up and down counters with overflow and underflow between N and 0 will work as movement operators by changing a current high-probability basis state $|B\rangle$ to $|B+1\rangle$ and $|B-1\rangle$ in superposition. We can construct an n-qubit increment circuit as shown in Figure 12.2a, with the corresponding Python code below.

PY

Find the code
In file src/quantum_walk.py

```
def incr(qc, idx: int, nbits: int, aux, controller=[]):
  for i in range(nbits):
    ctl=controller.copy()
    for j in range(nbits-1, i, -1):
      ctl.append(j+idx)
    qc.multi_control(ctl, i+idx, aux, ops.PauliX(), 'multi-1-X')
```

The analogous n-qubit-decrement circuit is also easy to construct with this code, following Figure 12.2b.

```
def decr(qc, idx: int, nbits: int, aux, controller=[]):
  for i in range(nbits):
    ctl=controller.copy()
    for j in range(nbits-1, i, -1):
      ctl.append([j+idx])
    qc.multi_control(ctl, i+idx, aux, ops.PauliX(), 'multi-0-X')
```

With these tools, we can construct an initial n-qubit quantum circuit *step*, as shown in Figure 12.3. A step has to be applied multiple times to simulate a walk (consisting of more than just a single step).

For both increment and decrement, N is a power of 2. We can construct other types of counters, for example, counters with step size larger than 1 or counters that increment modulo another number. For example, to construct a counter modulo 9, we add gates that match the binary representation of 9 and force the counter to reset to 0 once it reaches that limit value, as shown in Figure 12.4.

We can see how to generalize this pattern to other topologies. For example, for a 2D walk across a grid, we can use two Hadamard coins: one for the left or right movement

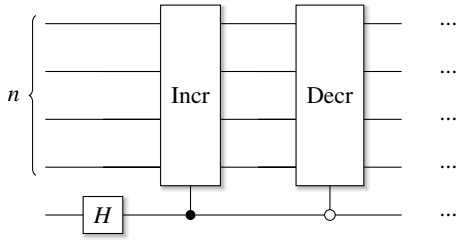


Figure 12.3 A single step for a quantum walk. The increment and decrement unitary operators change the current high-probability computational basis state $|B\rangle$ to $|B+1\rangle$ and $|B-1\rangle$ in superposition.

and one for movements up or down. For graph traversals, we would encode a graph's connectivity as a unitary operator. Several other examples of this can be found in Douglas and Wang (2009).

12.1.2 Walk the Walk

To simulate a given number of steps, we use the following driver code. The current position in the walk is encoded as a basis state, and we will go left and right by adding and subtracting 1. If we start at 0 and subtract 1, we have an immediate underflow to deal with. We have a similar problem with overflow at the high end of the number range. To make our lives a little easier, we initialize the `x` register *in the middle* of the state number range for n qubits. The middle of the binary number range for a given number of bits is the binary number that has a single 1 as the most significant bit, for example, the binary `0b100...0`. Starting there, we avoid the immediate underflow below zero, and the visualizations appear centered.

Note how the increment operator is controlled by `coin[0]`, while the decrement operator is controlled by the single-element list `[coin[0]]`. The former is a standard Controlled-by-1 gate, while the latter is a Controlled-by-0 gate, as described in Section 3.3.6.

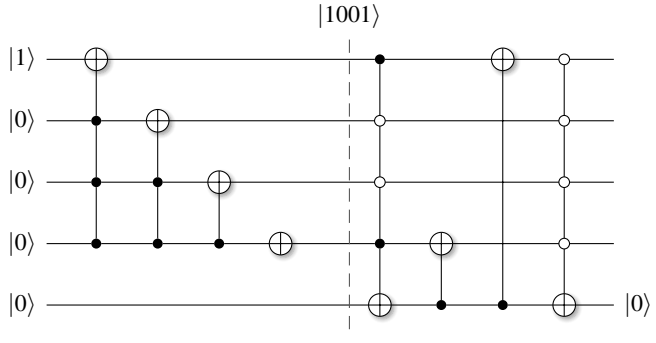


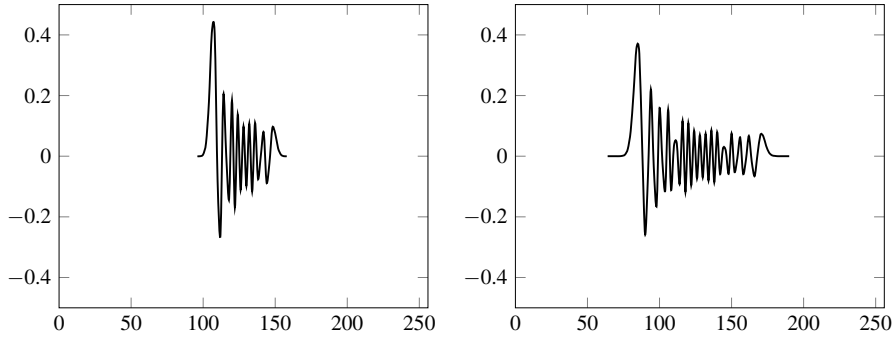
Figure 12.4 An increment modulo 9 operator.

```
def simple_walk():
  """Simple quantum walk."""

  nbits = 8
  qc = circuit.qc('simple_walk')
  x = qc.reg(nbits, 0x80)
  aux = qc.reg(nbits, 0)
  coin = qc.reg(1, 0)  # Add single coin qubit

  for _ in range(64):
    qc.h(coin[0])
    incr(qc, 0, nbits, aux, [coin[0]])     # ctrl-by-1
    decr(qc, 0, nbits, aux, [[coin[0]]])  # ctrl-by-0
```

What is happening here? With n qubits, we can represent 2^n states in superposition with the corresponding number of probability amplitudes. As we perform step after step, nonzero amplitudes will *propagate out* over the state space. Looking at the examples in Figure 12.5b and 12.6b, we see that, in contrast to a classical walk, the amplitude distribution spreads out faster and with a different shape. A series of 32 steps produces a nonzero amplitude in 64 states. The walk progresses in both directions at the same time. The farther away from the origin, the larger the amplitudes. These are the key properties that quantum walk algorithms exploit to solve classically intractable problems.

To visualize how fast the amplitudes spread out, we print and graph the amplitudes after a number of steps.

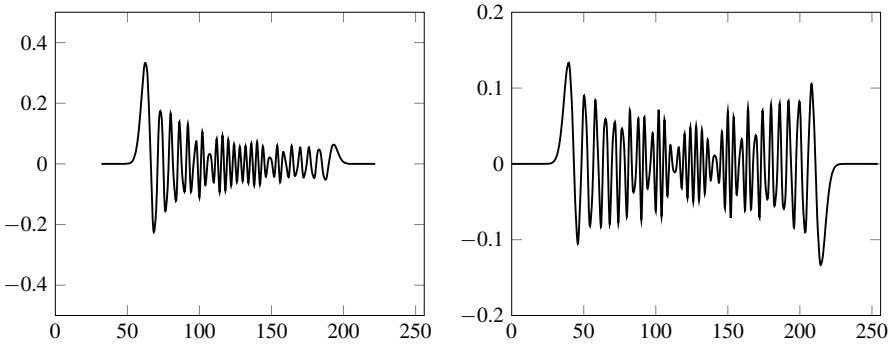
```
  for bits in helper.bitprod(nbits):
    idx_bits = bits
    for i in range(nbits+1):
        idx_bits = idx_bits + (0,)
    if qc.psi.ampl(*idx_bits) != 0.0:
      print('{:5.3f}'.format(qc.psi.ampl(*idx_bits0).real))
```



(a) 8 qubits, 32 steps, starting at basis state $|100\ldots0\rangle$.

(b) 8 qubits, 64 steps, starting at basis state $|100\ldots0\rangle$.

Figure 12.5 Propagating amplitudes after 32 and 64 steps.



(a) 8 qubits, 96 steps, starting at basis state $|100\ldots0\rangle$, coin in initial state $|0\rangle$.

(b) 8 qubits, 96 steps, starting at basis state $|100\ldots0\rangle$, coin in initial state $|1\rangle$.

Figure 12.6 Propagating amplitudes with different initial states.

Let us experiment with eight qubits. The starting position for the walk should be in the middle of the range of basis states, which we encode as a binary basis state with a single $|1\rangle$ in the most significant qubit and $|0\rangle$ in the other qubits. With eight qubits, there are 256 possible basis states. We initialize the eight qubits as $|100\ldots0\rangle$, which is binary `0x80`, the middle of the range. The amplitudes after 32, 64, and 96 steps are shown in Figure 12.5a, Figure 12.5b, and Figure 12.6a. The x-axis shows the state space (256 unique basis states for eight qubits). The y-axis shows the amplitude of each basis state.

Notice how, in the figures, the amplitudes progress in a biased fashion. It is possible to create coin operators that are biased to the other side or even balanced coin operators. Alternatively, we can start in a state different from $|0\rangle$. In the example in Figure 12.6b, we simply initialize the coin state as $|1\rangle$.

There are countless more experiments that you can perform with different coin operators, starting points, initial states, number of qubits, iteration counts, and more complex topologies beyond simple 1D and 2D walks.

The 2010 IARPA program announcement set a challenge of eight complex algorithms to drive the development of scalable quantum software and infrastructure (IARPA, 2010). Three of these algorithms used quantum walks: The triangle finding algorithm (Buhrman et al., 2005; Magniez et al., 2005), the Boolean formula algorithm (Childs et al., 2009), and the welded tree algorithm (Childs et al., 2003).

It is exciting to know that if we can express a particular algorithmic reachability problem as a quantum walk circuit, the fast speed of quantum walks and the dense storage of states can lead to quantum algorithms with lower complexity than their corresponding classical algorithms.

13 Optimization Algorithms

At this point, we have convinced ourselves that several quantum algorithms have an advantage in computational complexity over their classical analogs. We have seen algorithms that use quantum search, algorithms based on the quantum Fourier transform, and algorithms utilizing quantum random walks. In this section, we discuss a small number of quantum optimization problems and touch on the topic of simulating quantum systems.

We begin with the variational quantum eigensolver algorithm (VQE), which allows finding the minimum eigenvalue of a Hamiltonian. As an application, we develop a quantum version of the graph maximum cut algorithm by framing the problem as a Hamiltonian. This algorithm was introduced as part of the quantum approximate optimization algorithm (QAOA), which we present briefly. We conclude with a discussion of the Subset Sum algorithm.

13.1 The Variational Quantum Eigensolver (VQE)

Welcome to a brief foray into the area of *quantum simulation*. In general, the goal of quantum simulation is to use a controllable quantum system to study another quantum system that is difficult to simulate classically. Classical simulation must deal with the exponentially growing number of basis states in superposition and the computationally complex equations that govern the evolution of a system. The original idea of using a quantum computer to simulate a quantum system was presented by Richard Feynman in his talk "Simulating physics with computers" (Feynman, 1982), which many regard as the origin of quantum computing.

We will start with the *variational quantum eigensolver* (VQE), which is primarily an optimization algorithm. The VQE is a hybrid classical/quantum algorithm, as it leverages a quantum computer to prepare and measure quantum states and a classical computer to optimize a set of parameters for finding the ground-state energy (lowest eigenvalue) of a given Hamiltonian.

It is possible to use quantum phase estimation (QPE) for this purpose. For realistic Hamiltonians, however, the number of required gates can reach millions, even billions, making it challenging to keep a physical quantum machine coherent long enough to run the computation. For VQE, on the other hand, the quantum part requires fewer gates and much shorter coherence times than QPE (Zhang et al., 2022). This is why it created such great interest in today's era of Noisy Intermediate Scale Quantum

Computers (NISQ), which have limited resources and short coherence times (Preskill, 2018).

Any self-respecting book on quantum computing must mention the Schrödinger equation at least once. This is that section in this book. We begin by marveling at the beauty of the equation, although we will not solve it here. The purpose of showing it is to derive the spectral decomposition of Hamiltonians from eigenvectors (see also Section 4.1) and to show how the variational principle enables the approximation of a minimum eigenvalue. This is followed by a discussion of measurements in different bases and the hybrid classical/quantum algorithm itself.

13.1.1 System Evolution

In Section 2.13, we describe the evolution of a closed quantum system in postulate 2 as $|\psi'\rangle = U|\psi\rangle$. This is what we have used in this text so far. To change a state, we applied a unitary operator that did not depend on a time parameter; we used U and not $U(t)$. This *discrete time* evolution of a system is sufficient for all the algorithms discussed in previous sections. However, it is a simplification, as time does not move in discrete steps (as far as we know, or perhaps *suspect*).

The following paragraphs derive a specific form of the time-independent Schrödinger equation. The details are not overly important in the context of this text. We focus primarily on the final form because that is where the VQE will come into play.

The *time-dependent* evolution of the state $|\Psi\rangle$ of a system is described by the beautiful Schrödinger equation (which typically does not use the bracket notation). Here, we discuss the one-dimensional[1] version only. Again, let us marvel at this differential equation. We don't have to solve it here:

$$i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + V\Psi.$$

This equation can be transformed into the *time-independent* form[2]

$$-\frac{\hbar^2}{2m}\frac{d^2|\psi\rangle}{dx^2} + V|\psi\rangle = E|\psi\rangle. \tag{13.1}$$

In classical mechanics, the total energy of a system, which is the kinetic energy plus the potential V, is called a *Hamiltonian*, denoted as $\mathcal{H}$, not to be confused with our Hadamard operator H:

$$\begin{aligned}\mathcal{H}(x,p) &= \frac{mv^2}{2} + V(x)\\ &= \frac{(mv)^2}{2m} + V(x)\\ &= \frac{p^2}{2m} + V(x).\end{aligned}$$

[1] This dimension of the physical system is different from the dimension of the Hilbert space of the quantum state.

[2] We assume a time-independent potential.

As a side note, the factor $\hbar$ (the reduced Planck constant)[3] is the same factor used in the famous Heisenberg uncertainty principle for the position x and the momentum p_x, with $\Delta x \Delta p_x \geq \hbar/2$. A *Hamiltonian operator* is obtained by the standard substitution of the momentum operator p:

$$p \to -i\hbar \frac{\partial}{\partial x},$$
$$\hat{\mathcal{H}} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x).$$

We use this form to rewrite Equation (13.1) as the following, with $\hat{\mathcal{H}}$ being the operator, E being an energy eigenvalue and $|\psi\rangle$ being an eigenstate. Note the parallel to the definition of eigenvectors as $A\vec{x} = \lambda\vec{x}$:

$$\hat{\mathcal{H}}\,|\psi\rangle = E\,|\psi\rangle\,.$$

The expectation value for the total energy is then

$$\langle\hat{\mathcal{H}}\rangle = E.$$

The Hamiltonian operator is Hermitian. On measurement, we obtain real values, which means that the eigenvalues must be real. The operator has a complete set of orthonormal eigenvectors $|E_0\rangle, |E_1\rangle, \ldots, |E_{n-1}\rangle$, with the corresponding real eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$. Hence, we can describe a state as a linear combination of the eigenvectors as

$$|\psi\rangle = c_0|E_0\rangle + c_1|E_1\rangle + \cdots + c_{n-1}|E_{n-1}\rangle, \tag{13.2}$$

with complex coefficients c_i and basis vectors E_i, such that $\sum_i |c_i|^2 = 1$. This is the result that we were looking for. For a detailed derivation, see, for example, Fleisch (2020).

13.1.2 The Variational Principle

Assume that we are looking for the *ground state energy* E_0 of a system described by a given Hamiltonian. Knowing the ground state energy is important in many fields. For example, in thermodynamics, it describes behavior at temperatures close to absolute zero. In chemistry, this enables us to draw conclusions about electron energy levels.

Let us now assume that we cannot solve the time-independent Schrödinger Equation (13.1). We know that the measurement will project the state onto an eigenvector, and the measurement result will be the corresponding eigenvalue. The *variational principle* gives an *upper bound* for E_0 with an expectation value for $\hat{\mathcal{H}}$ as

$$E_0 \leq \langle\psi|\hat{\mathcal{H}}|\psi\rangle \equiv \langle\hat{\mathcal{H}}\rangle.$$

However, what is this state $|\psi\rangle$? The answer is *potentially any* state. The actual chosen state will determine the remaining error for estimating E_0. We have to be smart

[3] See also https://en.wikipedia.org/wiki/Planck_constant. For simplicity, it is common to normalize $\hbar = h/(2\pi)$ to 1.

about how to construct it. This is the key idea of the VQE algorithm. To see how this principle works, let us again take the state

$$|\psi\rangle = c_0|E_0\rangle + c_1|E_1\rangle + \cdots + c_{n-1}|E_{n-1}\rangle.$$

Let's assume that λ_0 is the minimal eigenvalue. Calculating $\langle\psi|\hat{\mathcal{H}}|\psi\rangle$ as follows demonstrates that *any* computed expectation value will be greater than or equal to λ_0 (recall that the $\{|c_i|\}_i^{n-1}$ form a vector of probabilities, that's why this inequality holds):

$$\begin{aligned}(c_0^*\langle E_0| + c_1^*\langle E_1| + \cdots + c_{n-1}^*\langle E_{n-1}|)\ \hat{\mathcal{H}}\ (c_0|E_0\rangle + c_1|E_1\rangle + \cdots + c_{n-1}|E_{n-1}\rangle) \\ = |c_0|^2\lambda_0 + |c_1|^2\lambda_1 + \cdots + |c_{n-1}|^2\lambda_{n-1} \\ \geq \lambda_0.\end{aligned}$$

The structure of real Hamiltonians is another complication. The VQE algorithm works with Hamiltonians that can be written as a sum of a polynomial number of terms of Pauli operators and their tensor products (Peruzzo et al., 2014). This type of Hamiltonian is used in quantum chemistry, the Heisenberg model, the quantum Ising model, and many other fields. For example, for a helium hydride ion (He-H^+) with bond distance 90 pm, the Hamiltonian (with $\sigma_z\sigma_x$ as a shorthand notation for $\sigma_z \otimes \sigma_x$) is

$$\begin{aligned}\hat{\mathcal{H}} = &-3.851II - 0.229I\sigma_x - 1.047I\sigma_z - 0.229\sigma_xI + 0.261\sigma_x\sigma_x \\ &+ 0.229\sigma_x\sigma_z - 1.0467\sigma_zI + 0.229\sigma_z\sigma_x + 0.236\sigma_z\sigma_z.\end{aligned}$$

To measure states affected by such Hamiltonians, we need to be able to measure in an arbitrary Pauli basis. This will be the topic of Section 13.1.3.

13.1.3 Measurement in Pauli Bases

So far in this book, we have mainly described measurement as projecting a state onto basis states, such as the computational basis states $|0\rangle$ and $|1\rangle$. If we recall the Bloch sphere representation as shown in Figure 13.1, this type of standard measurement projects the state to either the north or south pole of the Bloch sphere, corresponding

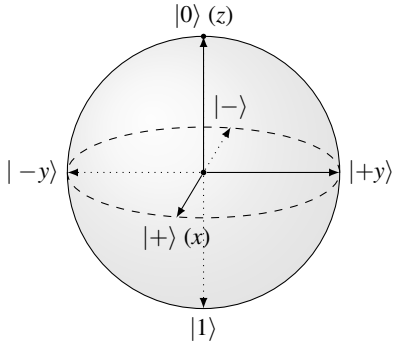


Figure 13.1 Bloch sphere representation with axes x, y, z.

to a measurement along the z-axis. However, what if the current state was aligned with a different axis, such as the x-axis or the y-axis? In both cases, a measurement along the z-axis would result in a random toss between $|0\rangle$ and $|1\rangle$ and not produce the result we were looking for.

To measure in a different basis, we should *rotate* the state to the standard basis on the z-axis and perform a standard measurement there. The results can be interpreted as if they were along the original bases, and we get the added benefit of only needing a measurement apparatus in one direction.

For example, to measure along the x-axis, we can apply a Hadamard gate or rotate about the y-axis. Correspondingly, to get a measurement along the y-axis, we may rotate about the x-axis. To compute expectation values for states composed of Pauli matrices, recall the X, Y, and Z bases states:

$$\begin{aligned}
X: \quad & |+\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}, \qquad |-\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}, \\
Y: \quad & |+y\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\i\end{pmatrix}, \qquad |-y\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-i\end{pmatrix}, \\
Z: \quad & |0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \qquad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}.
\end{aligned}$$

Pauli operators have eigenvalues of -1 and $+1$. Applying these operators to basis states with eigenvalues $+1$ yields

$$X|+\rangle = |+\rangle, \quad Z|0\rangle = |0\rangle, \quad Y|+y\rangle = |+y\rangle.$$

The same operators applied to basis states with eigenvalues -1 yields

$$X|-\rangle = -|-\rangle, \quad Z|1\rangle = -|1\rangle, \quad Y|-y\rangle = -|-y\rangle.$$

Let us now talk about expectation values. For a state in the Z basis with amplitudes c_0^z and c_1^z we write the state as

$$|\psi\rangle = c_0^z|0\rangle + c_1^z|1\rangle.$$

Calculating the expectation value for the Z gate, measured in the Z basis, yields the following. You can compute this for the X and Y bases in a similar fashion:

$$\begin{aligned}
\langle\psi|Z|\psi\rangle &= \left(c_0^{z*}\langle 0| + c_1^{z*}\langle 1|\right) Z \left(c_0^z|0\rangle + c_1^z|1\rangle\right) \\
&= \begin{pmatrix}c_0^{z*} & c_1^{z*}\end{pmatrix}\begin{pmatrix}1 & 0\\0 & -1\end{pmatrix}\begin{pmatrix}c_0^z\\c_1^z\end{pmatrix} \\
&= \begin{pmatrix}c_0^{z*} & -c_1^{z*}\end{pmatrix}\begin{pmatrix}c_0^z\\c_1^z\end{pmatrix} \\
&= |c_0^z|^2 - |c_1^z|^2.
\end{aligned}$$

The values $|c_0^z|^2$ and $|c_1^z|^2$ are the measurement probabilities for $|0\rangle$ and $|1\rangle$. If we run N experiments and measure state $|0\rangle$ n_0 times and state $|1\rangle$ n_1 times, then

$$|c_0^z|^2 = \frac{n_0}{N}, \quad |c_1^z|^2 = \frac{n_1}{N}.$$

This means that the empirical expectation value for Z is

$$\langle Z \rangle = |c_0^z|^2 - |c_1^z|^2 = \frac{n_0 - n_1}{N}.$$

For example, let's assume we have the random number generator from Section 6.1, which consists of a single-qubit state initialized as $|0\rangle$ and a Hadamard gate. The state after this gate is $|+\rangle$, which you can find in Figure 13.1 on the positive x-axis. If we now measure N times in the Z basis, about 50% of the measurements will return $|0\rangle$, and 50% will return $|1\rangle$. The $|0\rangle$ corresponds to eigenvalue $\lambda_0 = +1$, and the $|1\rangle$ corresponds to eigenvalue $\lambda_1 = -1$. Hence, the expectation value is

$$\frac{\lambda_0 N/2 + \lambda_1 N/2}{N} = \frac{(+1)N/2 + (-1)N/2}{N} = 0.$$

If we rotate the state into the Z basis with another Hadamard gate, the expectation value of $|0\rangle$ in the Z basis would now be 1, which corresponds to the expectation value of the state $|+\rangle$ originally in the X basis.

In our infrastructure, we do not have to make measurements to compute probabilities because we can directly look at the amplitudes of a state vector. To compute the expectation values for measurements made on Pauli operators with eigenvalues $+1$ and -1 corresponding to measuring $|0\rangle$ or $|1\rangle$, we add this function to our quantum circuit implementation.

PY

Find the code
In file src/lib/circuit.py

```
def pauli_expectation(self, idx: int):
  # Pauli eigenvalues are -1 and +1, hence we can calculate the
  # expectation value as:
  p0, _ = self.measure_bit(idx, 0, False)
  return p0 - (1 - p0)
```

Let us run a few experiments to familiarize ourselves with these concepts. What happens to the eigenvectors and eigenvalues for a Hamiltonian constructed from a single Pauli matrix multiplied by a factor? Is the result still unitary or Hermitian?

```
factor = 0.6
H = factor * ops.PauliY()
eigvals = np.linalg.eigvalsh(H)

print(f'Eigenvalues of {factor} X = ', eigvals)
print(f'is_unitary: {H.is_unitary()}')
print(f'is_hermitian: {H.is_hermitian()}')
>>
```

```
 Eigenvalues of 0.6 X =  [-0.6  0.6]
 is_unitary: False
 is_hermitian: True
```

We see that the eigenvalues scale with the factor. Hamiltonians are Hermitian but not necessarily unitary. Let us create a $|0\rangle$ state, show its Bloch sphere coordinates, and compute its expectation value in the Z basis.

```
  qc = circuit.qc('test')
  qc.reg(1, 0)
  qubit_dump_bloch(qc.psi)

  print(f'Expectation value for 0 State: {qc.pauli_expectation(0)}')
>>
x: 0.00, y: 0.00, z: 1.00
Expectation value for 0 State: 1.0
```

As expected, the current position is on top of the north pole, corresponding to the state $|0\rangle$. The expectation value is 1, and the state $|1\rangle$ cannot be measured. Now, if we add just a single Hadamard gate, we will get:

```
x: 1.00, y: 0.00, z: -0.00
Expectation value for |0>: -0.00
```

The position on the Bloch sphere is now on the x-axis, and the corresponding expectation value in the Z basis is 0. This is because the probabilities of measuring $|0\rangle$ or $|1\rangle$ are equal, leading to an average of 0.

13.1.4 VQE Algorithm

With these preliminaries, let us take a look at the VQE algorithm itself, which takes the following three steps:

1. **Ansatz**. Prepare a parameterized initial state $|\psi\rangle$, which is called the *ansatz*.
2. **Measurement**. Measure the expectation value $\langle\psi|\hat{\mathcal{H}}|\psi\rangle$.
3. **Minimize**. Tune the parameters of the ansatz to minimize the expectation value. The smallest value will be the best approximation of the minimum eigenvalue achievable with the given ansatz.

This is best explained by an example. Let us first focus on the single-qubit case here. We know from Section 2.3 that we can reach any point on the Bloch sphere with rotations about the x-axis and the y-axis. Let us use the simple parameterized circuit in Figure 13.2 as the ansatz to create a state $|\psi\rangle$. With this circuit, we can set any angles θ and ϕ as an initial guess to calculate the expectation value of a Hamiltonian $\hat{\mathcal{H}}$.

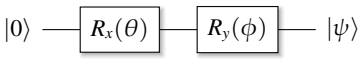


Figure 13.2 A flexible circuit to construct a single-qubit ansatz.

We will construct multiple instances of ansatzes[4] to find improved values for the angles. Let us write a function to make these ansatzes:

PY

Find the code
In file src/vqe_simple.py

```
def single_qubit_ansatz(theta: float, phi: float) -> circuit.qc:
  qc = circuit.qc('single-qubit ansatz Y')
  qc.qubit(1.0)
  qc.rx(0, theta)
  qc.ry(0, phi)
  return qc
```

Let us further assume a Hamiltonian $\hat{\mathcal{H}}$ of the form

$$\hat{\mathcal{H}} = \hat{\mathcal{H}}_0 + \hat{\mathcal{H}}_1 + \hat{\mathcal{H}}_2 = 0.2X + 0.5Y + 0.6Z.$$

We can compute the minimum eigenvalue of -0.8062 with `numpy`:

```
  H = 0.2 * ops.PauliX() + 0.5 * ops.PauliY() + 0.6 * ops.PauliZ()
  eigvals = np.linalg.eigvalsh(H)
  print(eigvals)
>>
[-0.8062258  0.8062258]
```

To compute the expectation value, let's create a state $|\psi\rangle$ and compute the expectation value $\langle\psi|\hat{\mathcal{H}}|\psi\rangle$ with the two given angles `theta` and `phi`:[5]

```
def run_single_qubit_experiment2(theta: float, phi: float):
  # Construct Hamiltonian.
  H = 0.2 * ops.PauliX() + 0.5 * ops.PauliY() + 0.6 * ops.PauliZ()

  # Compute known minimum eigenvalue.
  eigvals = np.linalg.eigvalsh(H)

  # Build the ansatz with two rotation gates.
  ansatz = single_qubit_ansatz(theta, phi)

  # Compute <psi | H | psi>. Find smallest one, which will be
  # the best approximation to the minimum eigenvalue from above.
```

[4] Which have a fun rhyme to them and is the proper English plural form. The correct German plural Ansätze does not sound quite as melodic.

[5] This code is modified from the open-source version and for illustration purposes only.

```
  val = np.dot(ansatz.psi.adjoint(), H(ansatz.psi))

  # Result from computed approach:
  print('Minimum: {:.4f}, Estimated: {:.4f}, Delta: {:.4f}'.format(
      eigvals[0], np.real(val), np.real(val - eigvals[0])))
```

We can experiment with a few different values for `theta` and `phi`:

```
  run_single_qubit_experiment2(0.1, -0.4)
  run_single_qubit_experiment2(0.8, -0.1)
  run_single_qubit_experiment2(0.9, -0.8)
>>
Minimum: -0.8062, Estimated:  0.4225, Delta: 1.2287
Minimum: -0.8062, Estimated:  0.0433, Delta: 0.8496
Minimum: -0.8062, Estimated: -0.2210, Delta: 0.5852
```

We are moving in the right direction as the delta is getting smaller and smaller. We are approaching the lowest eigenvalue, but we are still pretty far away. Since this particular ansatz is simple, we can incrementally iterate over both angles and approximate the minimum eigenvalue with good precision. We could also pick random numbers, which, for a simple case like this, may work quite well. In general, we should use techniques such as gradient descent to find the best possible arguments more quickly (Wikipedia, 2021d).

For experimentation, we perform ten experiments with random single-qubit Hamiltonians and iterate over the angles θ and ϕ in increments of 10 degrees:

```
  for i in range(0, 180, 10):
    for j in range(0, 180, 10):
      theta = np.pi * i / 180.0
      phi = np.pi * j / 180.0
  [...]
  # run 10 experiments with random H's.
>>
Minimum: -0.6898, Estimated: -0.6889, Delta: 0.0009
Minimum: -0.7378, Estimated: -0.7357, Delta: 0.0020
[...]
Minimum: -1.1555, Estimated: -1.1552, Delta: 0.0004
Minimum: -0.7750, Estimated: -0.7736, Delta: 0.0014
```

In the above, we explicitly computed the expectation value with dot products. However, in the physical world, we have to measure. The key to success here is that the ansatz must be able to find the minimum eigenvalue and its eigenvector. We need a circuit that is general enough.

There are many ways to prepare arbitrary two-qubit states; see, for example, Shende et al. (2004) or Section 9.2. However, for much larger Hamiltonians, the number of gates required for the ansatz may grow significantly. The challenge is to minimize the number and type of gates, especially on today's limited machines. The construc-

tion of suitable ansatzes is a research challenge. The specific learning technique for converging on an approximation is another topic of ongoing interest in the field, although standard techniques from the field of machine learning seem to work well.

13.1.5 Measuring Eigenvalues

In a physical setting, we cannot simply multiply a state by a Hamiltonian, as we have done here in the code. We have to measure along the Pauli bases and derive the eigenvalues from the expectation values, as explained above. As before, we assume that we can only measure in one direction. Let the Hamiltonian $\hat{\mathcal{H}}$ again be of the following form. We again choose the three random factors 0.2, 0.5, and 0.6 for the individual Pauli matrices. They are the key to success, and we must remember them:

$$\hat{\mathcal{H}} = 0.2X + 0.5Y + 0.6Z.$$

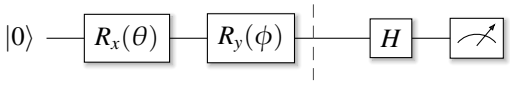
We calculate the expectation values in the Z basis with the help of gate equivalences (and the S gate from Section 2.7.6). Note how we isolate Z in the last line, representing the measurement in the Z basis:

$$\begin{aligned}\langle\psi|\hat{\mathcal{H}}|\psi\rangle &= \langle\psi|0.2X + 0.5Y + 0.6Z|\psi\rangle \\ &= 0.2\langle\psi|X|\psi\rangle + 0.5\langle\psi|Y|\psi\rangle + 0.6\langle\psi|Z|\psi\rangle \\ &= 0.2\langle\psi|HZH|\psi\rangle + 0.5\langle\psi|S^\dagger HZHS|\psi\rangle + 0.6\langle\psi|Z|\psi\rangle \\ &= 0.2\langle\psi H|Z|H\psi\rangle + 0.5\langle\psi S^\dagger H|Z|HS\psi\rangle + 0.6\langle\psi|Z|\psi\rangle.\end{aligned}$$

In our experimental code, we do not use the remembered factors but construct random Hamiltonians:

```
a = random.random()
b = random.random()
c = random.random()
H = (a * ops.PauliX() + b * ops.PauliY() + c * ops.PauliZ())
```

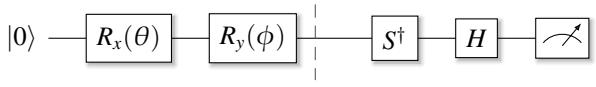
We have to build three circuits. The first is for the term $\langle\psi|X|\psi\rangle$, which requires a Hadamard gate.



We compute the expectation value and multiply it by the factor `a` from above to compute `val_a`:

```
    # X basis
    qc = single_qubit_ansatz(theta, phi)
    qc.h(0)
    val_a = a * qc.pauli_expectation(0)
```

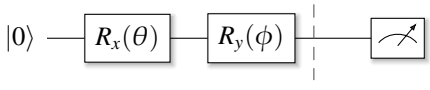
Then we build a circuit for $\langle\psi|Y|\psi\rangle$, which requires a Hadamard and an $S^\dagger$ gate.



We multiply the calculated expectation value with the factor `b` from above:

```
    # Y basis
    qc = single_qubit_ansatz(theta, phi)
    qc.sdag(0)
    qc.h(0)
    val_b = b * qc.pauli_expectation(0)
```

Finally, we build a circuit for the measurement in the Z basis $\langle\psi|Z|\psi\rangle$. In this basis, we can measure as is, there is no need for additional gates, but we still multiply the expectation value by the factor `c` from above.




```
    # Z basis
    qc = single_qubit_ansatz(theta, phi)
    val_c = c * qc.pauli_expectation(0)
```

As before, we iterate over the angles ϕ and θ and use increments of 5 degrees this time.[6] For each iteration, we take the scaled expectation values `val_a`, `val_b`, and `val_c`, add them, and find the smallest sum:

```
  expectation = val_a + val_b + val_c
  if expectation < min_val:
    min_val = expectation
  [...]
print('Minimum eigenvalue: {:.3f}, Delta: {:.3f}'
     .format(eigvals[0], min_val - eigvals[0]))
```

That value `min_val` should be our estimate, and below we can see that the results are numerically very accurate:

```
Minimum eigenvalue: -0.793, Delta: 0.000
Minimum eigenvalue: -0.986, Delta: 0.000
Minimum eigenvalue: -1.278, Delta: 0.000
Minimum eigenvalue: -0.937, Delta: 0.000
[...]
```

[6] This is fairly coarse; you may want to experiment with different values.

13.1.6 Multiple Qubits

How do we extend measurements to more than just one qubit? We begin with the simplest two-qubit Hamiltonians we can think of and extrapolate from there. Let us look at the tensor product $Z \otimes I$ and the corresponding operator matrix:

$$Z \otimes I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

We know that for diagonal matrices, the diagonal elements are the eigenvalues, which are $+1$ and -1 in this case.[7] This matrix has two subspaces that correspond to these eigenvalues. On measurement, we will get the result of $+1$ or -1.

Any unitary two-qubit transformation U of this matrix will map to a space with the same eigenvalues of $+1$ and -1, which we explained in Section 4.1. This means we can apply a trick similar to the one-qubit case and apply the following transformations. We are dealing with matrices and have to multiply from both sides:

$$U^{\dagger}(Z \otimes I)\, U.$$

We can change any Pauli measurement's basis into $Z \otimes I$. For example, to change the basis for $X \otimes I$ to $Z \otimes I$, we apply a Hadamard gate, just as above, with the operator $U = H \otimes I$. Let us verify this in code:

```
U = ops.Hadamard() * ops.Identity()
(ops.PauliZ() * I).dump('Z x I')
(U.adjoint() @ (ops.PauliX() * I) @ U).dump('Udag(X x I)U')
>>
Z x I (2-qubits operator)
 1.0       -         -         -
 -         1.0       -         -
 -         -        -1.0       -
 -         -         -        -1.0
Udag(X x I)U (2-qubits operator)
 1.0       -         -         -
 -         1.0       -         -
 -         -        -1.0       -
 -         -         -        -1.0
```

From this, it is straightforward to construct the operators for a first set of Pauli measurements that contain at least one identity operator, as shown in Table 13.1. But now it gets complicated. The operator we need to transform $Z \otimes Z$ is the controlled Not gate $CX_{1,0}$! How does this happen? The matrix for $Z \otimes Z$ is

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

[7] Non-unique eigenvalues are also referred to as degenerate eigenvalues, but this doesn't matter here.

Table 13.1. Operators for measurements containing an identity.

Pauli Measurement	Operator U
$Z \otimes I$	$I \otimes I$
$X \otimes I$	$H \otimes I$
$Y \otimes I$	$HS^\dagger \otimes I$
$I \otimes Z$	$(I \otimes I)\ SWAP$
$I \otimes X$	$(H \otimes I)\ SWAP$
$I \otimes Y$	$(HS^\dagger \otimes I)\ SWAP$

Table 13.2. Operators for measurements with no identity.

Pauli Measurement	Operator U
$Z \otimes Z$	$CX_{1,0}$
$X \otimes Z$	$CX_{1,0}\ (H \otimes I)$
$Y \otimes Z$	$CX_{1,0}\ (HS^\dagger \otimes I)$
$Z \otimes X$	$CX_{1,0}\ (I \otimes H)$
$X \otimes X$	$CX_{1,0}\ (H \otimes H)$
$Y \otimes X$	$CX_{1,0}\ (HS^\dagger \otimes H)$
$Z \otimes Y$	$CX_{1,0}\ (I \otimes HS^\dagger)$
$X \otimes Y$	$CX_{1,0}\ (H \otimes HS^\dagger)$
$Y \otimes Y$	$CX_{1,0}\ (HS^\dagger \otimes HS^\dagger)$

To turn this matrix into $Z \otimes I$, we need a specific permutation. Applying the controlled Not from both the left and the right (note that $CX_{1,0}^\dagger = CX_{1,0}$), as

$$CX_{1,0}^\dagger\ (Z \otimes Z)\ CX_{1,0} = (Z \otimes I),$$

yields the result we were looking for, as shown in this code snippet:

```
(ops.Cnot(1, 0).adjoint() @ (ops.PauliZ() * ops.PauliZ()) @
             ops.Cnot(1, 0)).dump()
  >>
 1.0        -          -          -
 -          1.0        -          -
 -          -         -1.0        -
 -          -          -         -1.0
```

The operator matrices for $CX_{1,0}$ perform the required permutation. You may think of this gate as having the potential to generate entanglement, like for a simple Bell state, or removing entanglement, as in this case. With this background, we can define the remaining 4×4 Pauli measurement operators as shown in Table 13.2.

We can generalize the construction for $Z \otimes Z$ to more than two qubits (see also Whitfield et al. (2011) on Hamiltonian simulation). All we have to do is surround

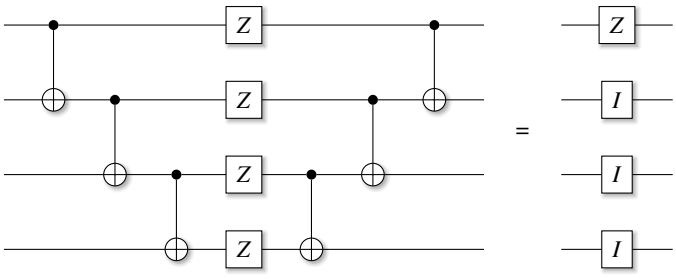


Figure 13.3 Measuring in the *ZZZZ* basis.

the multi-*Z* Hamiltonian with *cascading* controlled Not gates. For example, for the three-qubit *ZZZ*, we write this code:

```
ZII = ops.PauliZ() * ops.Identity()* ops.Identity()
C10 = ops.Cnot(1, 0) * ops.Identity()
C21 = ops.Identity() * ops.Cnot(2, 1)
C10adj = C10.adjoint()
C21adj = C21.adjoint()
ZZZ = ops.PauliZ() * ops.PauliZ() * ops.PauliZ()

res = C10adj @ C21adj @ ZZZ @ C21 @ C10
self.assertTrue(res.is_close(ZII))
```

Note that the adjoint of the *X* gate is identical to the *X* gate, and the adjoint of a controlled Not is also a controlled Not. To go even further, for *ZZZZ* or longer sequences of *Z* gates, we build cascading gate sequences, as shown in Figure 13.3. Moreover, just to be sure, you can verify the construction for *ZZZZ* with a short code sequence like this:

```
op1 = ops.Cnot(1, 0) * ops.Identity() * ops.Identity()
op2 = ops.Identity() * ops.Cnot(2, 1) * ops.Identity()
op3 = ops.Identity() * ops.Identity() * ops.Cnot(3, 2)

bigop = op1 @ op2 @ op3 @ ops.PauliZ(4) @ op3 @ op2 @ op1
op = ops.PauliZ() * ops.Identity(3)
self.assertTrue(bigop.is_close(op))
```

13.2 Quantum Approximate Optimization Algorithm

In this section, we briefly introduce the *Quantum Approximate Optimization Algorithm*, or QAOA (pronounced "Quah-Wah"). It was first introduced in the seminal paper by Farhi et al. (2014), which also details the use of QAOA for the implementation of the Max-Cut algorithm. We explore Max-Cut in Section 13.4.

The QAOA technique is related to VQE, but we only provide a brief overview. There are two operators in QAOA, U_C and U_B. The first operator U_C applies a phase to pairs of qubits with a problem-specific cost function C, which is similar to the Ising formulation below in Section 13.3, with Z_i being the Pauli Z gate applied to qubit i and l being the number of qubits or vertices involved:

$$C = \sum_{j,k}^{l} w_{jk} Z_j Z_k.$$

The operator U_C itself depends on a phase angle γ:

$$U_C(\gamma) = e^{-i\gamma C} = \prod_{j,k} e^{-i\gamma w_{jk} Z_j Z_k}.$$

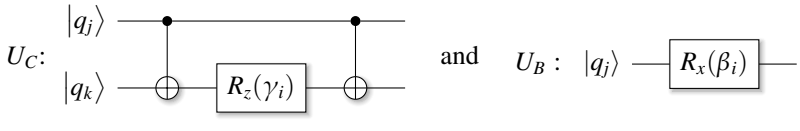
This operator acts on two qubits and thus can be used for problems that can be expressed as weighted graphs. The second operator U_B depends on parameter β. It is problem-independent and applies the following rotations to each qubit, where each X_j is a Pauli X gate:

$$U_B(\beta) = e^{-i\beta B} = \prod_j e^{-i\beta X_j}, \quad \text{where } B = \sum_j X_j.$$

For problems with higher depth, these two operators U_C and U_B are applied repeatedly, each with their own set of hyper-parameters γ_i and β_i, on an initial state of $|+\rangle^{\otimes n}$ as

$$U_B(\beta_{n-1}) U_C(\gamma_{n-1}) \ldots U_B(\beta_0) U_C(\gamma_0) \ |+\rangle^{\otimes n}.$$

The task at hand is similar to that of VQE: Find the best possible set of hyper-parameters to minimize the expectation value of the cost function $\langle \gamma, \beta | C | \gamma, \beta \rangle$, using well-known optimization techniques. The operators U_C and U_B can be approximated with these circuits:



We already know from Section 13.1 on VQE how to implement this type of search, so we will not expand on it further.

The original QAOA paper showed that for 3-regular graphs, which are cubic graphs with each vertex having exactly three edges, the algorithm produces a cut that is at least 70% of the maximum cut, a number that we can roughly confirm in our experiments below. Together with VQE, QAOA is an attractive algorithm for today's NISQ machines with limited resources since the corresponding circuits have a shallow depth (Preskill, 2018). At the same time, the utility of QAOA for industrial problems is still under debate (Harrigan et al., 2021).

13.3 Ising Formulations of NP Problems

The *Ising model* of ferromagnetism is a powerful statistical model of magnetism. Magnetic dipole moments of atomic spins are modeled as having values of $+1$ or -1. In the model, individual atoms are arranged on a grid. The *interaction* between a pair (i,j) of neighboring spins on the grid is J_{ij}. It takes the values $J_{ij} = 0$ for no interaction, $J_{ij} > 0$ for ferromagnetism, and $J_{ij} < 0$ for anti-ferromagnetism. There may also be an external magnetic field h_i, which interacts with individual atoms on the grid. With a magnetic moment μ (which is often omitted) and with $x_i \in \{+1, -1\}$ representing the spin of an atom at grid location i, the Hamiltonian for the system is then defined as

$$\hat{\mathcal{H}} = -\mu \sum_{i}^{N} h_i x_i - \sum_{i,j} J_{ij} x_i x_j.$$

For our quantum algorithms, we use this model to construct the Hamiltonian for a system using Pauli σ_z operators, as demonstrated in (Lucas, 2014):

$$\hat{\mathcal{H}}(x_0, x_1, \ldots, x_n) = -\mu \sum_{i}^{N} h_i \sigma_i^z - \sum_{i,j} J_{ij} \sigma_i^z \sigma_j^z. \tag{13.3}$$

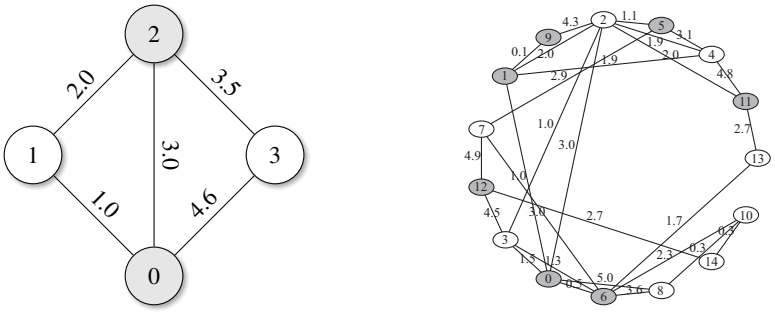
The term σ_i^z is the Pauli Z gate living in the Hilbert space of the ith qubit. The term μ will be 0 in our examples. There is no equivalent of an external field. The minus signs indicate that we look for a minimum eigenvalue. For problems such as Max-Cut, we use σ^z because we want an operator[8] with eigenvalues $+1$ and -1.

With this background, Lucas (2014) details several NP-complete or NP-hard problems for which this approach may work and lead to a quantum algorithm. The list of algorithms includes partitioning problems, graph coloring problems, covering and packing problems, Hamiltonian cycles (including the traveling salesperson problem), and tree problems. We will develop the related graph Max-Cut problem in Section 13.4, and a slightly modified formulation of the Subset Sum problem in Section 13.5.

13.4 Maximum Cut Algorithm

Previously, in Section 13.1, we saw how the VQE approach finds the minimum eigenvalue and its eigenvector for a Hamiltonian. This is exciting for quantum computing because if we can successfully frame an optimization problem as a Hamiltonian, we can use VQE to find an optimal solution. This section briefly describes how to construct a class of such Hamiltonians using the Ising spin-glass model we described above. The treatment here is admittedly shallow, but it is sufficient to implement impressive examples, such as the Max-Cut and Min-Cut algorithms in this section and the Subset Sum problem in Section 13.5.

[8] For other algorithms, we may need other eigenvalues. For example, an operator with eigenvalues 0 and 1, such as $(I - Z)/2$.



(a) A graph with 4 vertices

(b) A graph with 15 vertices

Figure 13.4 Example graphs with marked Max-Cut sets.

13.4.1 Max-Cut/Min-Cut

For a graph, a *cut* is a partition of the graph's vertices into two non-overlapping sets L and R. A *maximum cut* is the cut that maximizes the number of edges between L and R. The assignment of weights to the edges of the graph transforms the problem into the more general *weighted maximum cut*, which aims to find the cut that maximizes the weights of the edges between sets L and R. This is the *Max-Cut* problem we are trying to solve in this section.

The weights can be positive or negative. The Max-Cut problem turns into a Min-Cut problem simply by changing the sign of each weight. As an example, for the graph with four nodes shown in Figure 13.4a, the maximum cut is between the sets $L = \{0,2\}$ and $R = \{1,3\}$. You can manually verify that separating the nodes in this way maximizes the edge weights between the sets L and R. In the figure, the nodes are colored white or gray, depending on which set they belong to.

For small graphs, you can still examine all possible set partitions to find the maximum cut. However, this process becomes impractical very rapidly for larger graphs, as illustrated in Figure 13.4b for a graph with merely 15 vertices. The general Max-Cut problem is NP-complete (Karp et al., 1972); no polynomial-time algorithm is known to provide an optimal solution. This looks like a formidable challenge for a quantum algorithm!

13.4.2 Construct Graphs

We begin our exploration with code to construct a random graph with n vertices. As usual, the code is designed for simplicity.[9]

PY

Find the code
In file `src/max_cut.py`

We number the vertices from 0 to $n - 1$ and represent them with simple Python tuples `[from_node, to_node, weight]`. A graph is then just a list of these tuples.

[9] Making excuses for somewhat clumsy code.

The code starts with a triangle of three nodes and then randomly adds new nodes up to the limit `num`. We set 5 as the maximal edge weight, chosen at random. The way the loop is written, no double edges can be generated.

```
def build_graph(num: int = 0) -> Tuple[int, List[Tuple[int, int,
↪  float]]]:
  assert  num >= 3, 'Must request graph of at least 3 nodes.'

  # Nodes are tuples: (from: int, to: int, weight: float).
  weight = 5.0
  nodes = [(0, 1, 1.0), (1, 2, 2.0), (0, 2, 3.0)]
  for i in range(num - 3):
    rand_nodes = random.sample(range(0, 3 + i - 1), 2)
    nodes.append((3 + i, rand_nodes[0], weight * np.random.random()))
    nodes.append((3 + i, rand_nodes[1], weight * np.random.random()))
  return num, nodes
```

For debugging and for building intuition, it helps to visualize the graph. We add a helper function to print the graph in the dot file format for the Graphviz (graphviz.org, 2021) tool. The graph in Figure 13.4b was produced in this way.

```
def graph_to_dot(n: int, nodes: List[int], max_cut) -> None:
  print('graph {')
  print('  {\n    node [  style=filled ]')
  pattern = bin(max_cut)[2:].zfill(n)
  for idx, val in enumerate(pattern):
     if val == '0':
        print(f'    "{idx}" [fillcolor=lightgray]')
  print('  }')
  for node in nodes:
    print('  "{}" -- "{}" [label="{:.1f}",weight="{:.2f}"];'
          .format(node[0], node[1], node[2], node[2]))
  print('}')
```

13.4.3 Compute Max-Cut

We will use a binary representation to encode a cut. The graph nodes are numbered from 0 to $n-1$. The nodes in set L are marked with 1, nodes in set R with 0. For the example in Figure 13.4a, nodes 0 and 2 are in set L, and nodes 1 and 3 are in set R. We associate node 0 with index 0 (not bit 0) in a binary bit string (counting the indices from left to right) and represent the cut as a binary string 1010. We then apply this scheme to a quantum state by associating qubit q_i with graph node n_i:

$$\big|\underbrace{1}_{n_0}\underbrace{0}_{n_1}\underbrace{1}_{n_2}\underbrace{0}_{n_3}\big\rangle$$

In code, we can compute the Max-Cut exhaustively (and quite inefficiently, given our choice of data structures). For n nodes, we generate all binary bit strings from 0

to n. For each bit string, we iterate over the individual bits and build two index sets: indices with a 0 in the bit string and indices with a 1 in the bit string. For example, the bit string `11001` would create sets $L = \{0,1,4\}$ and $R = \{2,3\}$. Note the symmetry – if the Max-Cut is $L = \{0,1,4\}$ and $R = \{2,3\}$, then $L = \{2,3\}$ and $R = \{0,1,4\}$ is a Max-Cut as well.

The code then iterates over all edges in the graph. For each edge, if one vertex is in L and the other in R, there is an edge between sets. We add the edge weight to the currently computed maximum cut and maintain the absolute maximum cut.

Finally, we return the corresponding bit pattern as a simple decimal number. For example, if the maximum cut was binary `11001`, the routine will return 25, which means that this routine will only work with up to 64 bits or vertices (which is already beyond our simulation capabilities).

```
def compute_max_cut(n: int,
                    nodes: List[Tuple[int, int, float]]) -> int:
  max_cut = -1000.0
  for bits in helper.bitprod(n):
    # Collect in/out sets.
    iset = []
    oset = []
    for idx, val in enumerate(bits):
      if val == 0:
        iset.append(idx)
      else:
        oset.append(idx)

    # Compute costs for this cut, record maximum.
    cut = 0.0
    for node in nodes:
      if node[0] in iset and node[1] in oset:
        cut += node[2]
      if node[1] in iset and node[0] in oset:
        cut += node[2]
    if cut > max_cut:
      max_cut_in, max_cut_out = iset.copy(), oset.copy()
      max_cut = cut
      max_bits = bits

  state = bin(helper.bits2val(max_bits))[2:].zfill(n)
  print('Max Cut. N: {}, Max: {:.1f}, {}-{}, |{}>'
        .format(n, np.real(max_cut), max_cut_in, max_cut_out,
                state))
  return helper.bits2val(max_bits)
```

The performance of this code is, of course, quite horrible, but perhaps indicative of the combinatorial character of the problem. On a standard workstation, computing the Max-Cut for 20 nodes takes about 10 seconds; for 23 nodes, it takes about 110 seconds. Even considering the performance differences between Python and C++ and

the relatively poor choice of data structure, it is evident that the run time will quickly become intractable for larger graphs.

13.4.4 Construct Hamiltonian

Classically, the Max-Cut problem can be expressed as the optimization problem:

$$\max_s \frac{1}{2}\sum_{i,j}(1 - s_i s_j), \quad s_i \in \{-1, +1\}.$$

where the s_i are positive or negative weights of magnitude 1 on the vertices, depending on which set they belong to. If an edge is cut between two vertices, one vertice will have a positive 1 while the other has a negative 1 and the term $\frac{1}{2}(1 - s_i s_j) = 1$. If both vertices have the same sign, then $\frac{1}{2}(1 - s_i s_j) = 0$, which does not contribute to the objective. We can also introduce weighted edges by adding factors to the edges $s_i s_j$.

For the corresponding quantum formulation, we will use a variation of Equation (13.3) to construct the Hamiltonian using the Pauli σ_z operators, which, conveniently, also have eigenvalues -1 and $+1$.

Concretely, we iterate over the edges of the graph. We build the tensor product with identity matrices for nodes not part of the edge and Pauli σ_z matrices for the vertices connected by the edge. This follows the methodology briefly outlined above in Section 13.3. We may also use the intuition that Pauli σ_z are "easy" to measure, as we have outlined in Section 13.1 on measuring in the Pauli bases. Since the Pauli matrix σ_z has eigenvalues $+1$ and -1, an edge can *increase* or *decrease* the "energy" of the Hamiltonian, depending on the set in which the vertices fall. This construction decreases the energy for the vertices in the same set.

As an example, for the graph in Figure 13.4a we build the tensor products for the edges $e_{\text{from,to}}$ as[10]

$$\begin{aligned}
e_{0,1} &= 1.0\,(Z \otimes Z \otimes I \otimes I),\\
e_{0,2} &= 3.0\,(Z \otimes I \otimes Z \otimes I),\\
e_{0,3} &= 4.6\,(Z \otimes I \otimes I \otimes Z),\\
e_{1,2} &= 2.0\,(I \otimes Z \otimes Z \otimes I),\\
e_{2,3} &= 3.5\,(I \otimes I \otimes Z \otimes Z),
\end{aligned}$$

and add up these partial operators to the final Hamiltonian $\hat{\mathcal{H}}$, mirroring Equation (13.2):

$$\hat{\mathcal{H}} = e_{0,1} + e_{0,2} + e_{0,3} + e_{1,2} + e_{2,3}.$$

Here is the code to construct the Hamiltonian in full matrix form. It iterates over the edges and constructs the full tensor products as shown above:

[10] Note that even though we use *from* and *to*, the edges are not directed.

```
def graph_to_hamiltonian(n: int, nodes) -> ops.Operator:
  hamil = np.zeros((2**n, 2**n))
  for node in nodes:
    idx1 = max(node[0], node[1])
    idx2 = min(node[0], node[1])

    op = ops.Identity(idx1) * (node[2] * ops.PauliZ())
    op = op * ops.Identity(idx2 - idx1 + 1)
    op = op * (node[2] * ops.PauliZ())
    op = op * ops.Identity(n - idx2 + 1)

    hamil = hamil + op
  return ops.Operator(hamil)
```

As described so far, for a graph with n nodes, we have to build operator matrices of size $2^n \times 2^n$, which does not scale well. However, note that the identity matrix and σ_z are diagonal matrices. The tensor product of diagonal matrices is also a diagonal matrix. For example:

$$
\begin{aligned}
I \otimes I \otimes Z &= \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{pmatrix} \\
&= \mathrm{diag}(1, -1, 1, -1, 1, -1, 1, -1).
\end{aligned}
$$

If we apply a factor to any individual operator, that factor multiplies across the whole diagonal. Let us look at what happens to the signs of the diagonal values if we use σ_z at indices $0, 1, 2, \ldots, n-1$ (from right to left) in the tensor products

$$
\begin{aligned}
I \otimes I \otimes Z &= \mathrm{diag}(+1, -1, +1, -1, +1, -1, \underbrace{+1}_{2^0}, \underbrace{-1}_{2^0}), \\
I \otimes Z \otimes I &= \mathrm{diag}(+1, +1, -1, -1, \underbrace{+1, +1}_{2^1}, \underbrace{-1, -1}_{2^1}), \\
Z \otimes I \otimes I &= \mathrm{diag}(\underbrace{+1, +1, +1, +1}_{2^2}, \underbrace{-1, -1, -1, -1}_{2^2}).
\end{aligned}
$$

There are groups of values with the same sign in power-of-2 patterns, similar to those we have seen in the fast gate apply routines. This means that we can optimize the construction of the diagonal Hamiltonian and only construct a diagonal tensor product. The full matrix code is very slow and can barely handle 12 graph nodes. The diagonal version below can easily handle twice as many. C++ acceleration might

help to further improve scalability, especially because calls to `tensor_diag` can be parallelized.[11]

```
def tensor_diag(n: int, fr: int, to: int, w: float):
  def tensor_product(w1: float, w2: float, diag):
    return [j for i in zip([x * w1 for x in diag],
                           [x * w2 for x in diag]) for j in i]

  diag = [w, -w] if (0 == fr or 0 == to) else [1, 1]
  for i in range(1, n):
    if i == fr or i == to:
      diag = tensor_product(w, -w, diag)
    else:
      diag = tensor_product(1, 1, diag)
  return diag

def graph_to_diagonal_h(n: int, nodes) -> List[float]:
  h = [0.0] * 2**n
  for node in nodes:
    diag = tensor_diag(n, node[0], node[1], node[2])
    for idx, val in enumerate(diag):
      h[idx] += val
  return h
```

13.4.5 VQE by Peek-A-Boo

After constructing the Hamiltonian, we would typically run the variational quantum eigensolver (VQE) to find the minimum eigenvalue. The corresponding eigenstate encodes the Max-Cut in binary form, where a 0 at the index i indicates that the vertex i belongs to a first set and otherwise to a second set. In our simulated environment, we don't have to run the computationally expensive VQE, we can just take a peek at the matrix representation of the Hamiltonian. It is diagonal, meaning that the eigenvalues are on the diagonal. The corresponding eigenstate is a state vector with the same binary encoding as the index of the minimum eigenvalue. For example, for the graph in Figure 13.5, the Hamiltonian is

$$\begin{aligned}\mathcal{H} = \mathrm{diag}(&49.91, -21.91, -18.67, -5.32, 10.67, -2.67, -41.91, 29.91,\\ &29.91, -41.91, -2.67, 10.67, -5.32, -18.67, -21.91, 49.91).\end{aligned}$$

The minimum value is -41.91 and appears in two places: At index 6, which is binary `0110`, and at the complementary index 9, which is binary `1001`. This corresponds to the state $|0110\rangle$ and the complementary state $|1001\rangle$. We interpret this as nodes 0 and 3 belonging to one set and nodes 1 and 2 belonging to a second set. You can see that this is precisely the Max-Cut pattern of Figure 13.5. We have found the Max-Cut by applying *VQE by peek-a-boo* on a properly prepared Hamiltonian!

[11] The `tensor_product` routine is admittedly difficult to read. Python linters warn about its complexity. I still use it here because it *is* a thing of beauty.

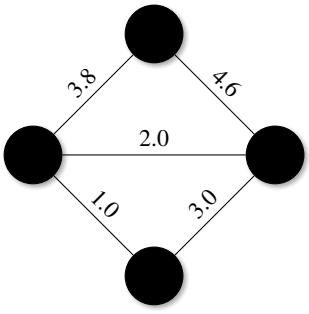


Figure 13.5 Graph with 4 nodes; Max-Cut is $\{0,3\},\{1,2\}$, or `0110` in binary set encoding.

Here is the code to run the experiments. It constructs the graph and computes the Max-Cut exhaustively. Then it computes the Hamiltonian and obtains the minimum value and its index off the diagonal.

```
def run_experiment(num_nodes: int):
  n, nodes = build_graph(num_nodes)
  max_cut  = compute_max_cut(n, nodes)

  # These two lines are the basic implementation, where
  # a full matrix is being constructed.
  #   H       = graph_to_hamiltonian(n, nodes)
  #   diag    = H.diagonal()
  # This code is much faster:
  diag     = graph_to_diagonal_h(n, nodes)
  min_idx  = np.argmin(diag)

  if min_idx == max_cut:
    print('SUCCESS: {:+10.2f} |{}>'.format(np.real(diag[min_idx]),
                  bin(min_idx)[2:].zfill(n)))
  else:
    print('FAIL   : {:+10.2f} |{}>  '.format(np.real(diag[min_idx]),
                  bin(min_idx)[2:].zfill(n)), end='')
    print('Max-Cut: {:+10.2f} |{}>'.format(np.real(diag[max_cut]),
                  bin(max_cut)[2:].zfill(n)))
```

Running this code, we find that it does *not always* work; it fails in about 20%–30% of the invocations. Some of it may be because our criteria are stringent: To mark a run as successful, we classically check whether the *optimal* cut was found. Anything else is considered a failure. However, even if the optimal cut was not found, the results are still within 30% of the optimal classical cut and typically significantly closer than 20% (this seems to agree with the analysis in the QAOA paper.)

As a larger example, running over graphs with 12 nodes may produce output like the following results, which show the number of nodes N and the maximal cut, computed classically, including the two sets for the cut. Then we find the index of the

smallest diagonal element of the Hamiltonian. A test passes when the binary basis state corresponding to this index matches the classically computed maximum. For example, in the first line of the following output, the set $\{2,3,5,6,8,11\}$ should match a binary state that has 1s at only these bit indices, as in $|001101101001\rangle$, which matches the result found (shown in **bold** below):

```
Max Cut. N: 12, Max: 38.9, [0, 1, 4, 7, 9, 10]-[2, 3, 5, 6, 8, 11],
↪  |001101101001>
SUCCESS :    -129.39 |001101101001>
Max Cut. N: 12, Max: 39.5, [0, 1, 5, 6, 7, 9]-[2, 3, 4, 8, 10, 11],
↪  |001110001011>
SUCCESS :    -117.64 |001110001011>
Max Cut. N: 12, Max: 46.0, [0, 3, 5, 8, 11]-[1, 2, 4, 6, 7, 9, 10],
↪  |011010110110>
FAIL    :    -146.79 |001010110110>  Max-Cut:    -145.05 |011010110110>
[...]
Max Cut. N: 12, Max: 43.7, [0, 1, 3, 4, 7, 8, 9, 10]-[2, 5, 6, 11],
↪  |001001100001>
SUCCESS :    -124.69 |001001100001>
```

Exploring the maximum degree[12] of the graph is instructive, as it seems to be one of the factors that affect the failure rate of this algorithm. Finally, note that a quantum advantage for this algorithm has not been fully established, except for a special class of graphs (Carlson et at., 2023).

13.5 Subset Sum Algorithm

Now that we know how to solve an optimization problem let us explore another algorithm of this type, namely the so-called *Subset Sum* problem. This problem is known to be NP-hard (Garey et al., 1990) and can be stated as follows: Given a set S of integers, can S be partitioned into two sets, with L and $R = S - L$, such that the sum of the elements in L equals the sum of the elements in R:

$$\sum_{i}^{|L|} l_i = \sum_{j}^{|R|} r_j.$$

We will approach this problem with a Hamiltonian constructed similarly to the one in Max-Cut. There, we used two weighted Z gates to represent an edge in a graph. Here, we will introduce only a single weighted Z gate to represent a single number in S. In Max-Cut, we were looking for a minimal energy state. For the problem of balancing partial sums, we will look for a zero-energy state, as such a state represents the "energy equilibrium" of balanced partial sums.

[12] See also http://en.wikipedia.org/wiki/Degree_(graph_theory).

13.5.1 Implementation

Our implementation only decides whether or not a solution exists. It does not identify a specific solution. To start, since this algorithm is begging to be experimented with, we define relevant parameters as command line options.

Find the code
In file src/subset_sum.py

The maximum integer in S is specified by parameter `nmax`. Similarly to Max-Cut, we encode integers as positions in a bit string corresponding to a state. This means that for integers up to `nmax`, we will need `nmax` qubits. The size $|S|$ of the set S is specified by parameter `nnum`. Finally, the number of experiments to run is specified by parameter `iterations`.

```
flags.DEFINE_integer('nmax', 15, 'Maximum number')
flags.DEFINE_integer('nnum',  6,
                     'Maximum number of set elements [1-nmax]')
flags.DEFINE_integer('iterations', 20, 'Number of experiments')
```

The next step is to produce `nnum` random and unique integers ranging from 1 to `nmax` (exclusive). Other ranges are possible, specifically ranges including negative numbers, but given that we use integers as bit positions, we have to map such arbitrary ranges to the positive range of 1 to `nmax`. In the code, we check that the sum of all selected random numbers is even because otherwise, the partitioning into equal sums is not possible:

```
def select_numbers(nmax: int, nnum: int) -> List[int]:
  while True:
    sample = random.sample(range(1, nmax), nnum)
    if sum(sample) % 2 == 0:
      return sample
```

To compute the diagonal tensor product, we only have to check for a single number (compared to the Max-Cut algorithm, where we had to check for two numbers) and apply a correspondingly weighted (by index i) Z gate.

```
def tensor_product(w1: float, w2: float, diag):
  return [j for i in zip([x * w1 for x in diag],
                         [x * w2 for x in diag]) for j in i]

def tensor_diag(n: int, num: int):
  diag = [1, 1]
  for i in range(1, n):
    if i == num:
      diag = tensor_product(i, -i, diag)
    else:
```

```
    diag = tensor_product(1, 1, diag)
  return diag
```

The final step in building the Hamiltonian is to add all the diagonal tensor products. This function is similar to the function `graph_to_diagonal_h` in the Max-Cut algorithm, except for the invocation of `tensor_diag` to compute the diagonal tensor product. If we implemented more algorithms of this type, we should spend more effort to generallize this construction.

```
def set_to_diagonal_h(num_list: List[int], nmax: int) -> -> List[float]:
  h = [0.0] * 2**nmax
  for num in num_list:
    diag = tensor_diag(nmax, num)
    for idx, val in enumerate(diag):
      h[idx] += val
  return h
```

13.5.2 Experiments

Now we move on to experiments. In each experiment, we create a list of random numbers and exhaustively find potential partitions. Then, similarly to Max-Cut, we divide the set of numbers into two sets with the help of binary bit patterns. For each division, we will compute the two sums for the two sets. If the two sums are equal, we will add the corresponding bit pattern to the list of positive results. The routine then returns this list, which can be empty if no solution is found for a given set of numbers. The partitioning for an example set is shown in Figure 13.6.

```
def compute_partition(num_list: List[int]):
  solutions = []
  for bits in helper.bitprod(len(num_list)):
    iset = []
    oset = []
    for idx, val in enumerate(bits):
```

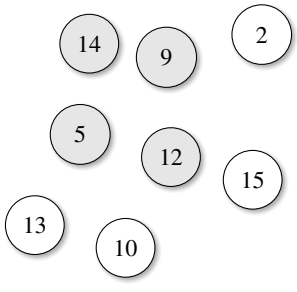


Figure 13.6 A subset partition for a set of eight integers. The partial sums of all elements in the white and gray sets are equal.

```
      if val == 0:
        iset.append(num_list[idx])
      else:
        oset.append(num_list[idx])
    if sum(iset) == sum(oset):
      solutions.append(bits)
  return solutions
```

Next, we add a short helper function to print the solutions found:

```
def dump_solution(bits: List[int], num_list: List[int]):
  iset = []
  oset = []
  for idx, val in enumerate(bits):
    if val == 0:
      iset.append(f'{num_list[idx]:d}')
    else:
      oset.append(f'{num_list[idx]:d}')
  return '+'.join(iset) + ' == ' + '+'.join(oset)
```

Finally, we run the experiments. For each experiment, we create a set of numbers, compute the solutions exhaustively, and compute the Hamiltonian.

```
def run_experiment(num_list: List[int]) -> bool:
  nmax = flags.FLAGS.nmax
  if not num_list:
    num_list = select_numbers(nmax, flags.FLAGS.nnum)
  solutions = compute_partition(num_list)
  diag = set_to_diagonal_h(num_list, nmax)
```

Now we can again perform VQE by peek-a-boo. For Max-Cut, we took an index and value from the diagonal. But what is the correct value to look for here? We are looking for a zero energy state that indicates a balance between the sets L and R. Hence, we look for zeros on the diagonal of the Hamiltonian. There can be multiple zeros, but as long as one single zero can be found, we know there should be a solution.

If no solution was found exhaustively, but we still find a zero on the diagonal, we know that we have encountered a false positive. Conversely, if no zero was found on the diagonal, but the exhaustive search found a solution, we encountered a false negative. The code below checks both of these conditions:

```
  non_zero = np.count_nonzero(diag)
  if non_zero != 2**nmax:
    print('  Solution should exist...', end='')
    if solutions:
      print(' Found: ', dump_solution(solutions[0], num_list))
      return True
    assert False, 'False positive found.'
  print('  No Solution Found.', sorted(num_list))
```

```
  assert not solutions, 'False negative found.'
  return False
```

As we run the code, we should see a success rate of 100%.

```
def main(argv):
  for _ in range(flags.FLAGS.iterations):
    run_experiment(None)
>>
Solution should exist... Found Solution: 13+1+5+3 == 14+8
Solution should exist... Found Solution: 10+1+14 == 4+12+9
Solution should exist... Found Solution: 4+9+14 == 12+5+10
[...]
Solution should exist... Found Solution: 1+3+11+2 == 5+12
```

We also test for negative cases which are sets of numbers that cannot be divided into two equal sums:

```
  sets = [
      [1, 2, 3, 7],
      [1, 3, 5, 10],
      [2, 7, 8, 10, 12, 13],
      [...]
  ]
  for s in sets:
    if run_experiment(s):
      raise AssertionError('Incorrect Classification')
```

14 Quantum Machine Learning

At the time of this writing, machine learning (ML), artificial intelligence (AI), and even artificial general intelligence (AGI), are the hottest topics in academia and industry, with billions of dollars in funding and several companies evaluated at multi-trillion dollar stock values. The computational needs for large language models (LLMs) based on the transformer architecture (Vaswani et al., 2017) are gigantic and continue to grow exponentially, with models having 10 trillion parameters and more.

Quantum machine learning (QML) merges machine learning algorithms with quantum computing to potentially unlock new efficiencies. Since QML promises to address the scalability limitations of ML, it is a rapidly evolving area of exciting research and promising applications.

This section focuses on just three representatives from this field: Euclidean distance, principal component analysis, and the HHL algorithm. We will start with a quantum way to calculate the Euclidean distance between two vectors, a fundamental operation in several classical ML algorithms. Next, we explore principal component analysis (PCA), which we can view as another efficient way to calculate eigenvalues. Both the PCA and the Euclidean distance calculation utilize specific forms of the swap test.

Finally, we will work through the HHL algorithm for matrix inversion. This algorithm is one of the most complex algorithms in this book but also one of the most beautiful. It was at the center of the interest in quantum machine learning and we explore it in great detail.

14.1 Euclidean Distance

This section explores how to compute the Euclidean distance between two state vectors. This computation can be the dominant factor in several classical algorithms, particularly machine learning algorithms. We mention a few examples of such algorithms in Section 14.1. Being able to perform this calculation quantum mechanically enables these classical algorithms to (potentially) run faster on quantum machines. As we shall see shortly, the core mechanism for calculating the distance between two arbitrary real vectors $\vec{A}$ and $\vec{B}$ is closely related to the swap test discussed in Section 7.1.

Classically, the Euclidean distance[1] between two vectors of real numbers is computed as the norm of the vector difference

$$D = |\vec{A} - \vec{B}| = \sqrt{(a_0 - b_0)^2 + (a_1 - b_1)^2 + \cdots + (a_{n-1} - b_{n-1})^2}.$$

The quantum technique makes an initial assumption that the encoding of the input vectors as quantum states is feasible. Furthermore, when comparing the complexity of the classical algorithms with analogous quantum algorithms that utilize the Euclidean distance, it is often assumed that state initialization is a zero-cost procedure. This may be difficult to achieve in practice, as shown in Chapter 9.

With these caveats, we assume that, without loss of generality, our vectors have dimensions that are powers of 2. In the first step, we use amplitude encoding to represent the vectors as quantum states, where each basis state $|i\rangle$ gets an amplitude corresponding to the vector element A_i. As discussed in Section 9.1.2, this representation is efficient in the number of qubits; only $\log_2 N$ qubits are required to represent vectors of size N.

We calculate the norm of the classical vectors $|\vec{A}|$ and $|\vec{B}|$ and normalize the vectors by dividing each vector element by their respective vector norm. This allows us to represent the vectors as the quantum states

$$\vec{A} \to |A\rangle = \frac{1}{|\vec{A}|} \sum_i A_i |i\rangle ,$$

$$\vec{B} \to |B\rangle = \frac{1}{|\vec{B}|} \sum_i B_i |i\rangle .$$

We can get the original vector $\vec{A}$ back with $\vec{A} = |\vec{A}|\,|A\rangle$. In code, normalizing the vectors is straightforward with `numpy`. We also compute the value $Z = |\vec{A}|^2 + |\vec{B}|^2$ that we will need later.

PY

Find the code
In file src/euclidian_distance.py

```
def run_experiment(a, b):
  norm_a = np.linalg.norm(a)
  norm_b = np.linalg.norm(b)
  assert norm_a != 0 and norm_b != 0, 'Invalid zero-vectors.'
  normed_a = a / norm_a
  normed_b = b / norm_b
  z = (norm_a**2) + (norm_b**2)
```

Next, we cleverly construct two states $|\phi\rangle$ and $|\psi\rangle$, with $|\phi\rangle$ as a single-qubit state and $|\psi\rangle$ encoding both vectors $\vec{A}$ and $\vec{B}$. It will become clear soon why we are creating these specific states:

[1] Also known as the L_2 norm of a vector v, often written as $||v||$. In this book, we mostly use single vertical bars $|v|$.

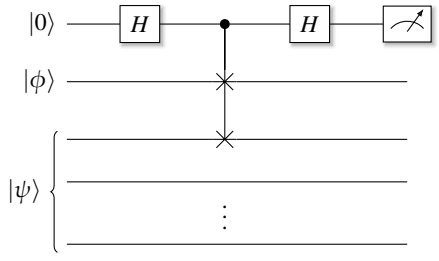


Figure 14.1 Circuit using a swap test to calculate the Euclidean distance between two vectors of length $N = 2^n$.

$$|\phi\rangle = \frac{1}{\sqrt{Z}}\big(|\vec{A}|\,|0\rangle - |\vec{B}|\,|1\rangle\big),$$
$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle \otimes |A\rangle + |1\rangle \otimes |B\rangle\big).$$

We can borrow ideas from the Hadamard test to construct these states with a circuit. However, in our code, we simply initialize the state vectors for $|\phi\rangle$ and the larger state $|\psi\rangle$ directly. Then, similarly to the swap test from Section 7.1, we create a state `combo` consisting of an ancilla initialized as $|0\rangle$ and tensored to states $|\phi\rangle$ and $|\psi\rangle$:

```
#  |phi> = 1 / sqrt(Z) (|a| |0> - |b| |1>)
phi = state.State(1 / np.sqrt(Z) * np.array([norm_a, -norm_b]))

#  |psi> = 1 / sqrt(2) |0>|a> + |1>|b>)
psi = (state.bitstring(0) * state.State(normed_a) +
       state.bitstring(1) * state.State(normed_b)) / np.sqrt(2)

combo = state.bitstring(0) * phi * psi
```

Now we use the swap test circuit to estimate the probability $p_{|0\rangle}$ of the ancilla being measured in state $|0\rangle$. The state $|\psi\rangle$ has multiple qubits, but we only connect the controlled Swap gate with $|\phi\rangle$ and the top qubit of $|\psi\rangle$, as shown in Figure 14.1 and with this equivalent code:

```
# Construct a swap test and find the ancilla probability
combo = ops.Hadamard()(combo, 0)
combo = ops.ControlledU(0, 1, ops.Swap(1, 2))(combo)
combo = ops.Hadamard()(combo, 0)

p0, _ = ops.Measure(combo, 0)
```

To see how this works, let us compute the inner product $\langle\phi|\psi\rangle$ of the two states and use the fact that, for a tensor product of a scalar and vector, $k \otimes \vec{A} = k\vec{A}$. The left and right sides of the dot product have different dimensions. However, the inner

product is performed only on the matching subsystems. You can think[2] of $\langle\phi|\psi\rangle$ as $\langle\phi \otimes I^{\otimes m}|\psi\rangle$:

$$
\begin{aligned}
&\langle\phi|\psi\rangle \\
&= \Big\langle \frac{1}{\sqrt{Z}}\big(|\vec{A}|\,|0\rangle - |\vec{B}|\,|1\rangle\big) \;\Big|\; \frac{1}{\sqrt{2}}\big(|0\rangle \otimes |A\rangle + |1\rangle \otimes |B\rangle\big)\Big\rangle \\
&= \frac{1}{\sqrt{Z}}\big(|\vec{A}|\,\langle 0| - |\vec{B}|\,\langle 1|\big)\frac{1}{\sqrt{2}}\big(|0\rangle \otimes |A\rangle + |1\rangle \otimes |B\rangle\big) \\
&= \frac{1}{\sqrt{2Z}}\big(|\vec{A}|\underbrace{\langle 0|0\rangle}_{=1} \otimes |A\rangle + |\vec{A}|\underbrace{\langle 0|1\rangle}_{=0} \otimes |B\rangle - |\vec{B}|\underbrace{\langle 1|0\rangle}_{=0} \otimes |A\rangle - |\vec{B}|\underbrace{\langle 1|1\rangle}_{=1} \otimes |B\rangle\big) \\
&= \frac{1}{\sqrt{2Z}}\big(|\vec{A}| \otimes |A\rangle - |\vec{B}| \otimes |B\rangle\big) \\
&= \frac{1}{\sqrt{2Z}}\big(|\vec{A}|\,|A\rangle - |\vec{B}|\,|B\rangle\big). \qquad (14.1)
\end{aligned}
$$

From the swap test, we know the probability of measuring the ancilla as $|0\rangle$ is

$$
\begin{aligned}
p_{|0\rangle} &= \frac{1}{2} + \frac{1}{2}\big|\langle\phi|\psi\rangle\big|^2, \quad \text{and hence} \\
2\Big(p_{|0\rangle} - \frac{1}{2}\Big) &= \big|\langle\phi|\psi\rangle\big|^2.
\end{aligned}
$$

Substituting in Equation (14.1), we compute the Euclidean distance D as

$$
\begin{aligned}
2\Big(p_{|0\rangle} - \frac{1}{2}\Big) &= \big|\langle\phi|\psi\rangle\big|^2 \\
&= \frac{1}{2Z}\big||\vec{A}|\,|A\rangle - |\vec{B}|\,|B\rangle\big|^2 \\
&= \frac{1}{2Z}|\vec{A} - \vec{B}|^2 \\
&= \frac{1}{2Z}D^2. \\
\Rightarrow D &= \sqrt{4Z\Big(p_{|0\rangle} - \frac{1}{2}\Big)}.
\end{aligned}
$$

In code, we calculate the distance D both classically and quantum mechanically. We run a range of experiments where we use random vectors containing `k=4` integers ranging from 0 to 10 and compare the classical and quantum results:

```
  # Compute the classic and quantum distances after swap test:
  eucl_dist_q = (4 * Z * (p0 - 0.5)) ** 0.5
  eucl_dist_c = np.linalg.norm(a - b)
  assert np.allclose(eucl_dist_q, eucl_dist_c, atol=1e-4), 'Whaaa'

def main(argv):
  for iter in range(10):
    a = np.array(random.choices(range(10), k=4))
```

[2] See also http://quantumcomputing.stackexchange.com/a/40084.

```
    b = np.array(random.choices(range(10), k=4))
    run_experiment(a, b)
>>
Compute Quantum Euclidean Distance.
Quantum Euclidean Distance between a=[3 6 4 4] b=[0 3 8 6]
  Classic: 6.16, quantum: 6.16, Correct
Quantum Euclidean Distance between a=[9 0 0 7] b=[1 5 5 7]
  Classic: 10.68, quantum: 10.68, Correct
[...]
```

14.1.1 Quantum Algorithms Using Euclidean Distance

Now that we have a quantum way to compute the Euclidean distance between vectors, we can find a range of classical algorithms that can potentially be accelerated using a quantum computer. The two typical examples given in the area of machine learning are the k-nearest neighbor algorithm (KNN) (Basheer et al., 2021) and the minimum spanning tree algorithm (Soltan et al., 2008). Both classical algorithms spend a lot of time computing vector distances, and assuming a zero-cost approach to translate these computations into a quantum algorithm is promising.

At time of writing, the most important topics in machine learning were transformer models such as ChatGPT and other large-language models (LLMs), as well as the so-called large-embedding models (LEMs). Embeddings are floating-point vectors, and large LEM and LLM models make heavy use of embeddings and vector databases. Hence, using quantum encoding and algorithms like the Euclidean distance calculation is an exciting optimization opportunity for quantum algorithms to accelerate those important workloads.

14.2 Principal Component Analysis

Machine learning deals with huge amounts of high-dimensional data and suffers from what is often called the *curse of dimensionality*. Adding features to data sets increases their size exponentially, inevitably introducing sparsity and inflicting enormous computational costs. There are adverse effects for models relying on distance measures as distances in high-dimensional spaces become less meaningful. It can be challenging to identify which specific features are important and which can be removed with limited impact on the overall quality of a model.

Principal component analysis (PCA) (Greenacre et al., 2022) is a well-known statistical technique that allows reducing the dimensionality of a data set while preserving its most important features, its *principal components*, which contribute maximally to the variance of the data set. This is a lossy data compression technique that attempts to keep the features that contribute the most to the variance of the data (and hence to the quality of a model) and remove correlated or insignificant data. The technique itself consists of the following high-level steps:

- The PCA aims to explain the variances in the data set. In order to avoid excessive influence of individual variables X_i, variables are standardized by centering them around their mean and normalizing them to values of similar magnitude. It is important to bring variables to the same scale, especially when they have different units.
- Compute the covariance matrix Σ consisting of all dot products $X_i X_j$. Since we center variables around their mean, the expectation value for each variable is $\mathbb{E}(X_i) = 0$. Note that Σ will be a symmetric matrix.
- Perform an eigenvalue decomposition (EVD) of Σ. This will produce the eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ of Σ as well as the eigenvectors corresponding to each eigenvalue. Note that a more efficient way to calculate the eigenvalues is using singular value decomposition (SVD).
- The sum of the eigenvalues equals the total variance. The components with the smallest eigenvalues contribute the least to the variance of the data set.[3] Those are the prime candidates for removal.

For a classical example, we take the data from (Abhijith et al., 2020) showing the correlation between apartment rooms and prices. In code, we can store the data as a two-dimensional Python array `x`:

```
x = [[4, 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 5, 4, 3, 4],
     [3028, 1365, 2726, 2538, 1318, 1693, 1412, 1632, 2875,
      3564, 4412, 4444, 4278, 3064, 3857]]
```

As a first step, we center each dimension around the mean and normalize all values to range from 0 to 1 as

$$X_0 \leftarrow \frac{X_0 - \mathbb{E}[X_0]}{|X_0|}, \quad \text{and} \quad X_1 \leftarrow \frac{X_1 - \mathbb{E}[X_1]}{|X_1|},$$

where $\mathbb{E}[X_0]$ is the expectation value of X_0 and $\mathbb{E}[X_0] = \mathbb{E}[X_1] = 0$ after we center the variables around the mean. Then we compute the covariance matrix Σ between these centered and normalized vectors as[4]

$$\Sigma = \begin{pmatrix} \mathbb{E}[X_0 X_0] & \mathbb{E}[X_0 X_1] \\ \mathbb{E}[X_1 X_0] & \mathbb{E}[X_1 X_1] \end{pmatrix} = \frac{1}{15-1} \begin{pmatrix} X_0^T X_0 & X_0^T X_1 \\ X_1^T X_0 & X_1^T X_1 \end{pmatrix}.$$

The algorithm proceeds to diagonalize Σ with the eigenvalues $e_0, e_1, \ldots, e_{n-1}$ on the diagonal in decreasing order. For PCA, the lowest right eigenvalues and their corresponding features will add the least to the data variance and could potentially be eliminated. In our example of a two-feature case, the matrix becomes

$$\Sigma = \begin{pmatrix} e_0 & 0 \\ 0 & e_1 \end{pmatrix}.$$

Our code strictly follows this recipe. We add a factor of 2 to house prices for numerical stability:

[3] Covariance matrices are symmetric and positive semi-definite with eigenvalues ≥ 0.

[4] You may wonder why we divide by 14, even though the data has 15 entries. This is related to the Bessel correction. See http://en.wikipedia.org/wiki/Bessel_correction for details.

```
def pca(x):
  x[0] = x[0] - np.average(x[0])
  x[0] = x[0] / np.linalg.norm(x[0])
  x[1] = (x[1] - np.average(x[1]))
  x[1] = 2 * x[1] / np.linalg.norm(x[1])

  m = (np.array([[np.dot(x[0], x[0]), np.dot(x[0], x[1])],
                 [np.dot(x[1], x[0]), np.dot(x[1], x[1])]]) /
      (len(x[0])-1))
```

The quantum version of this algorithm was first introduced in Lloyd et al. (2014), which uses phase estimation to find the eigenvalues. Here, we follow the refinement offered in Abhijith et al. (2020) on single-qubit states.

In the classical pre-processing step, we computed the covariance matrix Σ in the Python variable `m`. The first key idea is to interpret this matrix as a density matrix. Density matrices must have a trace of 1, so we simply divide the matrix by its current trace:

$$\rho = \frac{\Sigma}{\mathrm{tr}(\Sigma)}.$$

Next, we compute *purity* P as the trace of the squared density matrix, as described in Section 4.2:

$$P = \mathrm{Tr}(\rho^2).$$

For a pure state, the purity is $P = 1$, which for a single-qubit state means that the state lies on the surface of the Bloch sphere. Here is the second trick: From P we can determine the two eigenvalues λ_0 and λ_1 because we know that the trace of a density matrix is the sum of its eigenvalues and that for pure states, the trace must be 1. We know that $P = \lambda_0^2 + \lambda_1^2$ and $\lambda_0 + \lambda_1 = 1$. Hence, with a little algebra,

$$\begin{aligned} \underbrace{(\lambda_0 + \lambda_1)^2}_{=1} &= \underbrace{\lambda_0^2 + \lambda_1^2}_{=P} + 2\lambda_0\lambda_1, \\ \lambda_0\lambda_1 &= (1-P)/2. \end{aligned} \tag{14.2}$$

Now, with

$$\begin{aligned} \lambda_1 &= 1 - \lambda_0, \\ \lambda_0\lambda_1 &= \lambda_0(1-\lambda_0) \\ &= \lambda_0 - \lambda_0^2 \quad \text{and with Equation (14.2)} \\ &= (1-P)/2. \\ \Rightarrow \lambda_0^2 - \lambda_0 + (1-P)/2 &= 0. \end{aligned}$$

We can do the same computation for λ_1 and use the quadratic formula to get the eigenvalues as

$$\lambda_{0,1} = \frac{1 \pm \sqrt{2P-1}}{2}. \tag{14.3}$$

So far, so good, but how do we find the purity in a quantum way? In essence, we will use a cleverly constructed swap test. Here are the steps. We know the density matrix has a spectral decomposition of the following form, where we define the purity P as the trace of the squared density matrix:

$$\rho = \sum_i p_i \left|a_i\right\rangle \left\langle a_i\right|,$$
$$P = \mathrm{tr}(\rho^2) = \sum_i p_i^2.$$

A density matrix ρ may represent a mixed state, a statistical mixture of pure states. We can use the process of *purification* to transform a mixed state into a larger pure state, as we learned in Section 4.6. After purification, we can write the purified state $|\psi\rangle$ as

$$\left|\psi\right\rangle = \sum_i \sqrt{p_i} \left|a_i\right\rangle \left\langle b_i\right|,$$

where we use the same trick as in Section 4.6 on purification and reuse the bases $\{|a_i\rangle\}$ for the bases $\{|b_i\rangle\}$ on the right-hand side of the outer product. We can interpret this as making copies of the density matrix. We then apply a swap test between the purified copies. The expectation value of the Swap gate under state $\langle\psi|$ is $|\psi\rangle$

$$\begin{aligned}\left\langle\psi\right| \left\langle\psi\right| \mathrm{SWAP} \left|\psi\right\rangle \left|\psi\right\rangle &= \sum_{ij} \left\langle b_j\right| \left\langle a_j\right| \left\langle b_i\right| \left\langle a_i\right| \sqrt{p_i p_j} \sqrt{p_i p_j} \left|a_j\right\rangle \left|b_i\right\rangle \left|a_i\right\rangle \left|b_j\right\rangle \\ &= \sum_i p_i^2.\end{aligned}$$

The purity P equals the expectation value of the Swap gate. We will use this to find the purity P first and then the eigenvalues from it:

$$P = \mathrm{tr}\left(\rho^2\right) = \left\langle\psi\right| \left\langle\psi\right| \mathrm{SWAP} \left|\psi\right\rangle \left|\psi\right\rangle.$$

Our implementation builds the circuit shown in Figure 14.2. First, we construct the purified state $|\psi\rangle$. This is a bit like cheating since in order to construct this state, we have to compute the eigenvalues, the computation of which is the whole point of this algorithm. On a real quantum machine, this would have to be done by state preparation, as shown in Chapter 9.

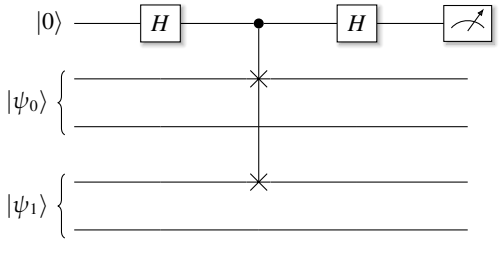


Figure 14.2 A simple PCA circuit. A swap test is performed between two instances of a purified input state $|\psi\rangle$, which we label as $|\psi_0\rangle$ and $|\psi_1\rangle$.

Find the code
In file src/quantum_pca.py

```
rho_eig_val, rho_eig_vec = np.linalg.eig(rho)
p_vec = np.concatenate((np.sqrt(rho_eig_val), np.sqrt(rho_eig_val)))
u_vec = rho_eig_vec.reshape((4))
psi = state.State(p_vec * u_vec)
```

Next, we construct the swap test. The expectation value of the swap gate under the purified state allows us to re-construct the eigenvalues. In the code, we initialize qubits 1 and 2 as well as qubits 3 and 4 with this state vector `psi`. Again, on a real quantum computer, we would have to add circuitry to prepare the state.

```
qc = circuit.qc('pca')
qc.reg(1, 0)
qc.state(psi)  # qubits 1, 2
qc.state(psi)  # qubits 3, 4

# Swap Test.
qc.h(0)
qc.cswap(0, 1, 3)
qc.h(0)
```

Now we can compute the purity from its expectation value and the two eigenvalues as shown in Equation (14.3). We also compare against the classically computed eigenvalues we get with a call to `numpy.linalg.eig`:

```
purity = qc.pauli_expectation(idx=0)
m_0 = (1 - np.sqrt(2 * purity - 1)) / 2 * np.trace(m)
m_1 = (1 + np.sqrt(2 * purity - 1)) / 2 * np.trace(m)
print(f'Eigenvalues Quantum PCA: {m_0:.6f}, {m_1:.6f}')

# Compare to classically derived values, which must match.
m, _ = np.linalg.eig(m)
if (not np.isclose(m_0, m[0], atol=1e-5) or
    not np.isclose(m_1, m[1], atol=1e-5)):
  raise AssertionError('Incorrect Computation.')
print(f'Eigenvalues Classically: {m[0]:.6f}, {m[1]:.6f}. Correct')
```

Running a few experiments will convince us that our calculations are correct:

```
for _ in range(10):
   for idx, _ in enumerate(x[0]):
     x[1][idx] = random.random() * 10000
   pca(x)
>>
Quantum Principal Component Analysis (PCA).
Eigenvalues Quantum PCA: 0.018904, 0.314429
```

```
Eigenvalues Classically: 0.018904, 0.314429. Correct
Eigenvalues Quantum PCA: 0.066605, 0.266728
Eigenvalues Classically: 0.066605, 0.266728. Correct
[...]
```

14.3 HHL Algorithm

The Harrow–Hassidim–Lloyd (HHL) algorithm (Lloyd et al., 2009)) is exciting and was central to the promises of *quantum machine learning*.[5] It uses several techniques we have already studied so far – and adds a few more – to solve for $\vec{x}$ in a system of linear equations

$$A\vec{x} = \vec{b}.$$

To set expectations right from the start, the algorithm does not fully "solve" for $\vec{x}$. Instead, it calculates specific properties of $\vec{x}$, such as how the vector elements relate to each other, which will only allow estimating $\vec{x}$ in the end.

This algorithm is one of the more complex algorithms in this book. The description follows the didactic flow of the excellent step-by-step guide by Morell et al. (2023). In the open-source repository, we faithfully implemented this reference in file `src/hhl_2x2.py`, where we added numerical verification checks for each main step. We present a slightly more general implementation, which can be found in the file `src/hhl.py` in the open-source repository.

A system of linear equations is also called a *linear system problem* (LSP). For example, for 3 equations with 3 variables x, y, and z, this system of equations

$$\begin{aligned} 2x + 2y - 1z &= 3, \\ x - 3y + 4z &= 4, \\ -x + 1y - 2z &= -5, \end{aligned}$$

can be written in matrix form as

$$\begin{pmatrix} 2 & 2 & -1 \\ 1 & -3 & 4 \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ -5 \end{pmatrix}.$$

The solution to this linear system would be $x = 1$, $y = 2$, and $z = 3$. In general, we can write the problem of finding $\vec{x}$ in $A\vec{x} = \vec{b}$ as a system of linear equations:

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} &= b_0, \\ a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} &= b_1, \\ &\vdots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} &= b_{n-1}. \end{aligned}$$

[5] At time of writing, machine learning is the hottest field in computer science and industry, so why would we combine it with another hot buzzword – quantum? Living in Silicon Valley, this is how venture capital is being unlocked!

Needless to say, this is a very important classical problem with innumerable applications. It is also a computationally very demanding problem. Let's discuss complexity and the case for a quantum algorithm next.

14.3.1 Complexity

Classically, this problem can be solved using Gaussian elimination,[6] which is generally of complexity $\mathcal{O}(n^3)$. Depending on the type of matrix, faster techniques may be applicable. Let us introduce two matrix properties:

- A matrix is *s-sparse* if it has at most s nonzero entries in each row and column. Typically, *s-sparse vectors* are defined as having only s nonzero elements, so s-sparse matrices have s-sparse vectors as rows and columns. Note that in the literature, the definition of sparsity may only pertain to either rows or columns, but we require s-sparsity for both.
- The matrix *condition number* κ ("kappa") is a metric on how *invertible* a given matrix is and is usually calculated as $\kappa = ||A|| \cdot ||A^{-1}||$ using the operator norm.[7] It hints at the accuracy that can be obtained when solving an LSP. A large condition number indicates high sensitivity; small changes in $\vec{b}$ can result in large effects and errors in $\vec{x}$. For Hermitian matrices, κ can be calculated as the ratio of largest to smallest eigenvalue after taking the absolute value (which is undefined if the denominator is 0).

For the special case of sparse matrices and a given approximation accuracy of ε, solving the LSP of size N classically with the *conjugate gradient method*[8] has a complexity of $\mathcal{O}(Ns\kappa \log(1/\varepsilon))$. The quantum HHL algorithm, on the other hand, has a complexity of $\mathcal{O}(\log(N)s^2\kappa^2/\varepsilon)$. This presents an exponential speed-up over the size N of the system but a polynomial slowdown in s and κ.

There are questions about this algorithm. The vector $\vec{b}$ has to be encoded as a state, and we have seen in Chapter 9 that state preparation can be a costly operation. Reading out a full result may result in an additional complexity of $\mathcal{O}(N)$. This means that the algorithm may only be applicable in cases where sampled results suffice. In other words, if an application only requires approximate information or statistics derived from the data, quantum algorithms may still offer a speed-up since fewer measurements are needed. Aaronson (2015) enumerates more reasons why we should be cautious about the promises made by this algorithm, including the fact that it only estimates relationships between vector elements, as mentioned above.

Nevertheless, despite these concerns, there is general recognition that the algorithm is, first of all, marvelous and that, at least for certain classes of input, it will indeed have a quantum advantage.

[6] See, for example, http://en.wikipedia.org/wiki/Gaussian_elimination.
[7] See also http://en.wikipedia.org/wiki/Operator_norm.
[8] See also http://en.wikipedia.org/wiki/Conjugate_gradient_method.

14.3.2 Algorithm

In the quantum algorithm, the matrix A must be an $N \times N$ Hermitian matrix with $N = 2^n$ for some n. Vectors $\vec{x}$ and $\vec{b}$ are N-dimensional vectors. A and $\vec{b}$ are known, $\vec{x}$ is the unknown we are trying to solve for. This means we can write the problem as

$$\vec{x} = A^{-1}\vec{b}. \tag{14.4}$$

Let's recall the required linear algebra. A complex square matrix A is *normal* if it commutes with its conjugate transpose $A^\dagger A = AA^\dagger$. It can be shown that A is normal *if and only if* it is unitarily diagonalizable. This means there exists a unitary matrix U such that

$$A = UDU^\dagger, \tag{14.5}$$

where D is a diagonal matrix. The diagonal elements of D will hold the eigenvalues of A, which must be real if A is Hermitian. The columns of U will be the orthonormal eigenvectors of A.

Since all unitary and Hermitian matrices are normal, the *finite-dimensional spectral theorem* applies, as shown in Section 4.1. Normal matrices can be written as the sum of the products of the eigenvalues λ_i with the outer product of the eigenvectors u_i. If you think about it, this is just a different way of writing Equation (14.5):

$$A = \sum_{i=0}^{N-1} \lambda_i |u_i\rangle\langle u_i|. \tag{14.6}$$

In this form, it is trivial to compute the inverse of A as

$$A^{-1} = \sum_{i=0}^{N-1} \lambda_i^{-1} |u_i\rangle\langle u_i|. \tag{14.7}$$

From Section 4.1, we know that a complex $N \times N$ Hermitian matrix has N linearly independent orthogonal basis vectors with real eigenvalues. This means that *any* vector in $\mathbb{C}^N$ can be constructed from such an orthogonal basis. Hence we can write $\vec{b}$ as a linear combination of A's basis vectors $|u_i\rangle$ as

$$|b\rangle = \sum_{i=0}^{N-1} b_i \, |u_i\rangle \, . \tag{14.8}$$

We combine this with Equations (14.4) and (14.7) as

$$\begin{aligned} |x\rangle &= A^{-1} \, |b\rangle \\ &= \sum_{i=0}^{N-1} \lambda_i^{-1} |u_i\rangle\langle u_i| b_i \, |u_i\rangle \\ &= \sum_{i=0}^{N-1} \lambda_i^{-1} b_i \, |u_i\rangle \underbrace{\langle u_i|u_i\rangle}_{=1}, \end{aligned}$$

$$\Rightarrow |x\rangle = \sum_{i=0}^{N-1} \lambda_i^{-1} b_i \, |u_i\rangle \, . \tag{14.9}$$

Since we encode the vectors as state vectors, Equations (14.8) and (14.9) require that

$$\sum_{i=0}^{2^{n_b}-1} |b_i|^2 = 1 \quad \text{and} \quad \sum_{i=0}^{2^{n_b}-1} |\lambda_i^{-1} b_i|^2 = 1.$$

As an example, let us define the 4×4 Hermitian Python matrix `a` as:

```
a = ops.Operator([[15, 9, 5, -3],
                  [9, 15, 3, -5],
                  [5, 3, 15, -9],
                  [-3, -5, -9, 15]]) / 4
print(a)
>>
[[ 3.75+0.j  2.25+0.j  1.25+0.j -0.75+0.j]
 [ 2.25+0.j  3.75+0.j  0.75+0.j -1.25+0.j]
 [ 1.25+0.j  0.75+0.j  3.75+0.j -2.25+0.j]
 [-0.75+0.j -1.25+0.j -2.25+0.j  3.75+0.j]]
```

Let's set a vector $\vec{b} = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}^T$ and solve the system with `numpy`:

```
b = ops.Operator([0, 0, 0, 1])
x = np.linalg.solve(a, b)
>>> x
array([-0.094+0.j,  0.156+0.j,  0.281+0.j,  0.469+0.j])
```

The quantum algorithm will *not* give us these concrete numbers. Instead, it will provide information on how these values relate to each other, which we classically compute as the ratios of all norms over the norm of the smallest element. The goal of our quantum HHL implementation is to produce similar ratios. We write this simple function to compute the ratios classically:

Find the code
In file `src/hhl.py`

```
def check_classic_solution(a, b):
  x = np.linalg.solve(a, b)
  ratio = []
  for i in range(1, len(x)):
    ratio.append(np.real((x[i] * x[i].conj()) / (x[0] * x[0].conj())))
    print(f'Classic ratio: {ratio[-1]:6.3f}')
  return ratio
```

For the example, the output will be:

```
Classic ratio:  2.778
Classic ratio:  9.000
Classic ratio: 25.000
```

We need a routine to classically compute the eigenvalues and eigenvectors, sorted by magnitude, which we get with the following code using `numpy`. Note that the function `eig` returns the eigenvalues as columns – you find the eigenvector for the eigenvalue `w[i]` in column `v[:, i]`. To make our lives easier, we take the eigenvectors out of the columns and "decolumn"[9] the vectors on return. Since the eigenvalues of a Hermitian matrix are real, we explicitly convert the eigenvalues with `np.real(w)` before returning the results to avoid Python type conflicts:

```
def compute_sorted_eigenvalues(a):
  w, v = np.linalg.eig(a)

  # Return sorted (real) eigenvalues and eigenvectors.
  idx = w.argsort()
  return np.real(w[idx]), v[:, idx]
```

Let's use this function to compute the eigenvalues and eigenvectors for the matrix `a` defined above. Note how we have chosen the matrix to have eigenvalues that are powers of 2:

```
w, v = compute_sorted_eigenvalues(a)
print('Eigenvalues:', w)
print('Eigenvectors:\n', v)
>>
Eigenvalues: [1. 2. 4. 8.]
Eigenvectors:
[[ 0.5+0.j  0.5+0.j -0.5+0.j  0.5+0.j]
 [-0.5+0.j -0.5+0.j -0.5+0.j  0.5+0.j]
 [-0.5+0.j  0.5+0.j  0.5+0.j  0.5+0.j]
 [-0.5+0.j  0.5+0.j -0.5+0.j -0.5+0.j]]
```

We can use the eigenvalues `w` and decolumned eigenvectors `v` to reconstruct the matrix `a` and its inverse `inv` with the following code, following Equations (14.6) and (14.7). We can check for correctness by comparing the reconstructed matrix `x` with the original `a` and by comparing the matrix product of `x` and its inverse `inv` with the identity matrix:

```
ndim = a.shape[0]
x = np.matrix(np.zeros((ndim, ndim)))
for i in range(ndim):
  x = x + w[i] * np.outer(v, v.adjoint())
assert np.allclose(a, x)

inv = np.matrix(np.zeros((ndim, ndim)))
for i in range(ndim):
  inv = inv + (w[i]**-1) * np.outer(v, v.adjoint())
assert np.allclose(inv @ x, ops.Identity(a.nbits))
```

[9] I am not sure this is a word. We take the vectors out of the columns and make them an array of vectors. With this, you get an eigenvector with `v[i]` instead of `v[:, i]`.

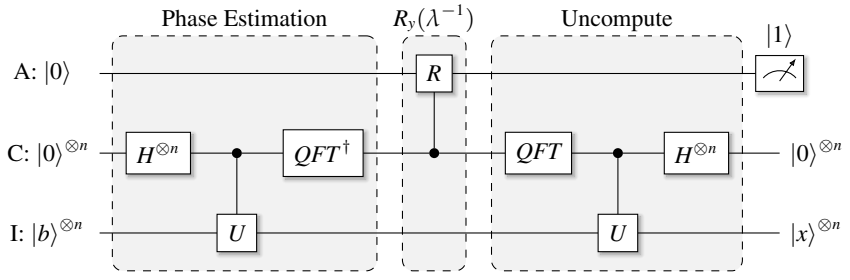


Figure 14.3 The HHL algorithm as a circuit diagram.

Equation (14.9) is the form we will use in the HHL algorithm to solve for $|x\rangle$. The algorithm has the following major steps, which are also shown in Figure 14.3:

1. State Preparation. The vector $\vec{b}$ needs to be encoded as a quantum state via state preparation. As we have learned in Chapter 9, this can be quite difficult in practice. Here, we just assume that vector $\vec{b}$ is readily available as a quantum state.
2. The next step is to perform quantum phase estimation to encode the eigenvalues of an operator U that represents the matrix A. For experimentation, in order to deal with known eigenvalues, we also have to construct the U operator.
3. This is followed by the so-called ancilla rotation. This step is new and complicated in its general form. Here, we simplify slightly and require that the eigenvalues of U be powers of 2.
4. We are almost done at this point. We uncompute the QPE from Step 2 above and measure and interpret the results.

To construct the circuit, let us assume that the vector $\vec{b}$ has $N_b = 2^{n_b}$ components for n_b qubits. We use amplitude encoding (Section 9.1.2) to encode this vector as a quantum state. The circuit consists of three register files:

1. A single *ancilla* qubit A. When written as a substate, we will denote this qubit by $|\,\rangle_a$.
2. A *clock register* C of width 2^t. This register will contain the encoded eigenvalues coming out of the quantum phase estimation. A larger number t of qubits will result in higher accuracy. In state notation, we refer to this register as $|\,\rangle_c$.
3. An *input register* I of width 2^{n_b}, which will contain the results of the state preparation to represent the vector $\vec{b}$. Similarly to the above, we denote the state of this register as $|\,\rangle_b$.

14.3.3 State Preparation

We start by using amplitude encoding as described in Section 9.1 to encode the vector $\vec{b}$ as a quantum state $|b\rangle$. We assume that this step is possible and efficient in a physical system. For simplicity, in our examples, we will only use states $|0\rangle^n$ or $|1\rangle^n$, which are easy to prepare. With the notation shown above, the system after state preparation will be in state

$$|\psi\rangle = |0\rangle_a |0\cdots 00\rangle_c |b\rangle_b ,$$

with $|b\rangle_b$ indicating a register of size n_b. We will see below that the algorithm produces a result in basis encoding, as described in Section 9.1.1. But let's not get ahead of ourselves.

14.3.4 Hamiltonian Encoding

To encode the matrix A, we use Hamiltonian encoding from Section 9.1.4. We interpret the Hermitian A as a time-dependent Hamiltonian and define the unitary operator

$$U(t) = e^{iAt}.$$

The factor t is the evolution time of the Hamiltonian and we will calculate a specific value for it shortly. The practical way to build U is to use the spectral decomposition of A and compute its eigenvalues λ_i as above (in the Python variable `w`). Then construct U using the computational basis vectors $|e_i\rangle$ as

$$U(t) = \sum_{i=0}^{N-1} e^{i\lambda_i t} |e_i\rangle\langle e_i|. \tag{14.10}$$

In this way, U will be a diagonal matrix. To keep the eigenvalues but change the basis back to the original basis of A, we use the matrix similarity transformation:[10]

$$U = VUV^\dagger,$$

where V is a matrix with the original eigenvectors of A as its rows.[11] We computed this V in the variable `v` above. V is also called a *change of basis* matrix.[12]

Note that this explicit construction is *only* for experimentation. The whole purpose of the HHL algorithm is to approximate the eigenvalues for a *given* unknown Hamiltonian. In code, we follow the exact steps as outlined above. The function `compute_u_matrix` gets the original matrix A as parameter `a`, the eigenvalues and decolumned eigenvectors as `w` and `v`, and the `t` time evolution parameter:

```
def compute_u_matrix(a, w, v, t):
  u = ops.Operator(np.zeros((a.shape[0], a.shape[1]),
                   dtype=np.complex64))
  for i in range(a.shape[0]):
    u[i][i] = np.exp(1j * w[i] * t)
  u = v @ u @ v.adjoint()
  return u
```

[10] http://en.wikipedia.org/wiki/Matrix_similarity.

[11] You will often find the similarity transformation written as $U = V^\dagger UV$ with V having the eigenvectors as columns. However, we already decolumned the eigenvectors of V into rows. Therefore, the adjoint $V^\dagger$ comes last.

[12] http://en.wikipedia.org/wiki/Change_of_basis.

14.3.5 Phase Estimation

Now that we know how to compute the matrix U for our experiments, the next step is to discuss the quantum phase estimation at a slightly more mathematical level. We know from Section 11.2.1 and Equation (11.2) that we can write the QPE in closed form as

$$QPE\,|\psi\rangle = \frac{1}{2^{n/2}} \sum_{k=0}^{N-1} e^{2\pi i k\phi}\,|k\rangle\,|\phi\rangle\,.$$

We also learned in Section 11.4 that we can write the quantum Fourier transform QFT in the closed form of Equation (11.8) as

$$QFT\,|x\rangle = \frac{1}{2^{n/2}} \sum_{j=0}^{N-1} e^{2\pi i j k/N}\,|j\rangle\,.$$

We apply the inverse QFT to the output of the QPE for the resulting state

$$\begin{aligned}
|\psi\rangle &= |b\rangle\, QFT^{\dagger} \left(\frac{1}{2^{n/2}} \sum_{k=0}^{N-1} e^{2\pi i k\phi}\,|k\rangle \right) |0\rangle_a \\
&= |b\rangle\, \frac{1}{2^{n/2}} \sum_{k=0}^{N-1} e^{2\pi i k\phi} \left(QFT^{\dagger}\,|k\rangle \right) |0\rangle_a \\
&= |b\rangle\, \frac{1}{2^{n}} \sum_{k=0}^{N-1} e^{2\pi i k\phi} \left(\sum_{y=0}^{N-1} e^{-2\pi i y k/N}\,|y\rangle \right) |0\rangle_a \\
&= |b\rangle\, \frac{1}{2^{n}} \sum_{k=0}^{N-1} \sum_{y=0}^{N-1} e^{2\pi i k(\phi - y/N)}\,|y\rangle\,|0\rangle_a\,.
\end{aligned}$$

If for some y the term $\phi - y/N = 0$, then the exponent will be 0, the exponential term will be 1, which leads to

$$\frac{1}{2^n} \sum_{y=0}^{N-1} e^{2\pi i k(\phi - y/N)} = \frac{1}{2^n} \sum_{y=0}^{N-1} e^{2\pi i k 0} = \frac{2^n}{2^n} = 1.$$

Since the resulting amplitude for this y_0 is 1, all other amplitudes must[13] be 0. This is also called destructive interference, which arises from the periodic nature of the complex exponentials. Since $\phi - y_0/N = 0$, we can solve for y_0:

$$\begin{aligned}
\phi - y_0/N &= 0, \\
\phi &= y_0/N, \\
\rightarrow y_0 &= \phi N.
\end{aligned}$$

With $|y_0\rangle = |\phi N\rangle$ we can write the state after QPE as the following, with $|b\rangle$ still being the unmodified input register, $|N\phi\rangle$ the phase estimate stored in the clock register, and $|0\rangle_a$ the unmodified ancilla:

[13] See http://quantumcomputing.stackexchange.com/a/39933 for a rigorous proof.

$$|b\rangle\,|N\phi\rangle\,|0\rangle_a\,. \tag{14.11}$$

This form is for a single y_0. We still have to consider that there are multiple eigenvalues in superposition and we have to decide what to do with that mysterious t parameter. This will be the focus of Section 14.3.6.

In code, let's write down what we have so far. To construct the circuit, we use the function `construct_circuit` and give it the following parameters: The vector $\vec{b}$, the eigenvalues in parameter `w`, the unitary operator U we constructed above, a constant `c` that we will derive below, and the number of qubits `clock_bits` we want to assign to the clock register. With these parameters, we construct the circuit `qc` and create the ancilla, clock, and $\vec{b}$ quantum registers, assigning the vector $\vec{b}$ to the `breg` register directly:

```
def construct_circuit(b, w, u, c, clock_bits):
  qc = circuit.qc('hhl')

  # State preparation - just initialize the b register.
  anc = qc.reg(1, 0)
  clock = qc.reg(clock_bits, 0)
  breg = qc.state(b)
```

Then we build the phase estimation circuit. We know we must uncompute this later, so as we iterate over the clock bits, we also store the inverse U^{-1} gates in variable `u_inv_gates`. The rest of the loop is similar to the QPE implementation in Section 11.2.3:

```
  # Phase estimation to bring the eigenvalues into the clock register.
  qc.h(clock)
  u_inv_gates = []
  for idx in range(clock_bits):
    op = ops.ControlledU(clock[idx], breg[0], u)
    qc.unitary(op, clock[idx])
    u_inv_gates.append(np.linalg.inv(u))
    u = u @ u
  qc.inverse_qft(clock, True)
```

This is only the first half of the function. We will finalize the implementation in just a few pages.

14.3.6 Time Evolution Parameter t

Hamiltonian encoding has that time evolution parameter t in the exponent. We will see that, as long as the eigenvalues are integer multiples of each other, we can use t to map the eigenvalues to actual integers. Let us derive a way to calculate t.

We know that U in the Hamiltonian encoding in Equation (14.10) is a normal matrix and that the spectral theorem applies (as discussed in Section 4.1):

$$U = \sum_i \lambda_i |x_i\rangle\langle x_i|.$$

Since U is unitary with $UU^\dagger = I$, it follows that $|\lambda_i|^2 = 1$ and $\lambda_i = e^{2\pi\phi_i}$ for some angle ϕ_i. If we set $|b\rangle$ as one of the eigenvectors $|u_i\rangle$ of U, then

$$U|b\rangle = e^{2\pi i\phi_i}|u_i\rangle.$$

We also know from Equation (14.10) that

$$U|b\rangle = e^{i\lambda_i t}|u_i\rangle.$$

We can equate the two and find ϕ_i with

$$\begin{aligned} i\lambda_i t &= 2\pi i\phi_i, \\ \Rightarrow \phi_i &= \frac{\lambda_i t}{2\pi}. \end{aligned}$$

We substitute this into Equation (14.11) and get

$$|\psi\rangle = |b\rangle \left|N\frac{\lambda_j t}{2\pi}\right\rangle |0\rangle_a.$$

In general, $|b\rangle$ will be in superposition as $|b\rangle = \sum_{j=0}^{2^{n_b}-1} b_j |u_j\rangle$ and we can generalize $|\psi\rangle$ to our almost final form:

$$|\psi\rangle = \sum_{j=0}^{2^{n_b}-1} b_j |u_j\rangle |N\lambda_j t/(2\pi)\rangle |0\rangle_a.$$

With all this in place, we can finally calculate a proper value for t. The eigenvalues λ_i are usually not integers, but as long as they are related to each other by integer factors, we can scale them to integers simply by dividing them by their smallest value λ_0. In practice, this λ_0 must be carefully chosen.

We want the eigenvalues to be integers for two reasons. Firstly, it makes it easy to map the binary eigenvalue values onto the clock register. More importantly, though, in order to invert the eigenvalues, we must apply controlled rotations by specific angles. These angles are much easier to find from integer eigenvalues, especially integers that are powers of two, as we will see shortly.

We still have to apply a factor t when we construct the matrix U. To obtain integer eigenvalues, we define t as

$$t = \frac{2\pi}{\lambda_0 N},$$

and define *scaled* eigenvalues with a tilde over the symbol λ as

$$\widetilde{\lambda_j} = N\lambda_j t/(2\pi) = \frac{\lambda_j}{\lambda_0}.$$

As long as the eigenvalues are integer multiples of each other, the scaled eigenvalues will be integers. With this, we arrive at the final form we were looking for:

$$|\psi\rangle = \sum_{j=0}^{2^{n_b}-1} b_j |u_j\rangle \left|\widetilde{\lambda}_j\right\rangle |0\rangle_a . \tag{14.12}$$

After QPE and inverse QFT with a time factor t, we will have the integer eigenvalues in superposition in the clock register. Section 14.3.7 describes how to rotate the eigenvalues to obtain their inverse values.

14.3.7 Inversion of Eigenvalues

In this section, we will take the state in Equation (14.12) and apply rotations by specific factors to change the state into a form that may seem confusing at first. We will measure the ancilla and only keep the results for which the ancilla measured $|1\rangle$. The remaining state will allow us to derive the rotation angles. At this point, it will also become apparent why we had to scale the eigenvalues to integer values. Let's dive right in.

We have seen in Section 9.1.3 that we can encode a value α with an R_y rotation by angle $\theta = 2\ \arcsin(\alpha)$ such that

$$R_y(\theta)|0\rangle = \sqrt{1-\alpha^2}\,|0\rangle + \alpha\,|1\rangle .$$

We choose a constant C and apply controlled $R_y(\theta_k)$ rotations between the ancilla $|0\rangle_a$ and the clock register with $\alpha = C/\widetilde{\lambda}_j$. In just a few paragraphs, it will become clear what the value of C should be and how to compute the angles θ_k. The rotations change the state in Equation (14.12) to

$$|\psi\rangle = \sum_{j=0}^{2^{n_b}-1} b_j |u_j\rangle \left|\widetilde{\lambda}_j\right\rangle \left(\sqrt{1-\frac{C^2}{\widetilde{\lambda}_j^{\,2}}}\,|0\rangle_a + \frac{C}{\widetilde{\lambda}_j}|1\rangle_a\right). \tag{14.13}$$

Now we measure the ancilla and discard the measurements that result in $|0\rangle$ for the ancilla. For the other cases, the state will hold the inverse eigenvalues and become

$$|\psi\rangle = c_m \sum_{j=0}^{2^{n_b}-1} b_j |u_j\rangle \frac{C}{\widetilde{\lambda}_j}|1\rangle_a, \quad \text{with} \quad c_m = \frac{1}{\sqrt{\sum_{j=0}^{2^{n_b}-1}\left|\frac{b_j C}{\widetilde{\lambda}_j}\right|^2}}.$$

The complicated factor c_m is just a normalization factor after measurement to ensure that the total probabilities in the state vector still add up to 1. With this, we can now derive a good value for C. The probability of measuring $|1\rangle_a$ in Equation (14.13) is $p_{|1\rangle} = |C/\widetilde{\lambda}_j|^2$. Probabilities must be less than or equal to 1. We mapped the eigenvalues to integers above, and the smallest mapped value is $\lambda_0 = 1$. Hence, to maximize the probability $p_{|1\rangle}$ of measuring $|1\rangle$ we want to use the largest possible value for C such that $p_{|1\rangle} \leq 1$. For this, we set C to 1.

We can now complete the function `construct_circuit`. We left off right after we built the QPE circuitry. The next steps are to compute the angles by calling

`compute_angles`, which we develop below, and apply the controlled R_y rotations between the clock register and the ancilla using `qc.cry`. To uncompute, we apply the inverse QPE using the gates we stored in `u_inv_gates`. The final step in this function is to measure and force the ancilla to state $|1\rangle$ and return this ingeniously constructed circuit `qc`:

```
angles = compute_angles(w, c)
for idx, angle in enumerate(angles):
  qc.cry(clock[idx], anc, angle)

# Uncompute.
qc.qft(clock, True)
for idx in reversed(range(clock_bits)):
  op = ops.ControlledU(clock[idx], breg[0],
                       u_inv_gates[idx])
  qc.unitary(op, clock[idx])
qc.h(clock)

# Measure (and force) ancilla to state |1>.
qc.measure_bit(anc[0], 1, collapse=True)
return qc
```

So far so good, but how do we go about computing the rotation angles? We know that in order to transform the ancilla from $|0\rangle_a$ to the right side of Equation (14.13), we have to apply an R_y rotation by an angle $\theta = 2 \arcsin \frac{1}{\widetilde{\lambda}_j}$ with:

$$R_y(\theta)\,|0\rangle = \sqrt{1 - \frac{1}{\widetilde{\lambda}_j^2}}\,|0\rangle_a + \frac{1}{\widetilde{\lambda}_j}\,|1\rangle_a\,.$$

We can calculate the angles θ_j for each integer eigenvalue $\widetilde{\lambda}_j = \{1,2,\ldots\}$ as

$$\theta_1 = 2 \arcsin \frac{1}{\mathbf{1}} = \pi,$$
$$\theta_2 = 2 \arcsin \frac{1}{\mathbf{2}} = \frac{\pi}{3},$$
$$\ldots$$

Now it becomes clear why we chose the eigenvalues to be powers of two. We can decompose a value c in the clock register before rotations in the binary form $c = c_{n-1}c_{n-2}\cdots c_{c_0}$ and apply the rotations for each of the individual distinct binary bits. In this way, the rotations do not overlap, as they would for eigenvalues with overlapping binary bits, such as 1 (`0b01`) and 3 (`0b11`). There are complex ways to solve cases where the binary representations of the scaled eigenvalues do overlap, but with this simplified power-of-2 scheme, we can compute the angles simply with:

```
def compute_angles(w, c):
  unis = np.unique(w)
  return [2 * np.arcsin(c / eigen) for eigen in unis]
```

14.3.8 Putting It All Together

We now have all the pieces in place to run this algorithm on a variety of inputs. We define a function `run_experiment` with parameters `a` for the matrix A and `b` for the vector $\vec{b}$. The function computes the sorted eigenvalues, as shown above, and produces the scaled integer eigenvalues in `lam`. It computes parameter `t`, sets the constant `c` (to 1), constructs the circuit with the code shown above, and compares the results. Without test and print statements, the code looks rather compact:

```
def run_experiment(a, b):
  clock_bits = len(b)
  n = 2 ** clock_bits

  w, v = compute_sorted_eigenvalues(a)
  lam = [w[i] / w[0] for i in range(a.shape[0])]
  t =  2 * np.pi / (w[0] * n)
  c = np.min(np.abs(lam))

  qc = construct_circuit(b, lam, u, c, clock_bits)
  check_results(qc, a, b)
```

To compare the results, we compute the ratios of the classical and quantum solutions. Then we compare the ratios one by one and ensure that they are within an acceptable accuracy range.

```
def check_results(qc, a, b):
  ratio_classical = check_classic_solution(a, b)
  res = (np.abs(qc.psi) > 0.07).nonzero()[0]
  ratio_quantum = [np.real(qc.psi[res[j]] ** 2 / qc.psi[res[0]] ** 2)
                   for j in range(1, len(res))]

  for idx, ratio in enumerate(ratio_quantum):
    delta = ratio - ratio_classical[idx]
    print(f'Quantum ratio: {ratio:6.3f}, delta: {delta:+5.3f}')
    if abs(delta) > 0.2:
      raise AssertionError('Incorrect result.')
```

In the Python `main` function, we construct several examples and run the experiments, resulting in output like this one:

```
def main(argv):
  a = ops.Operator([[15, 9, 5, -3],
                    [9, 15, 3, -5],
                    [5, 3, 15, -9],
                    [-3, -5, -9, 15]]) / 4
  b = ops.Operator([0, 0, 0, 1]).transpose()
  run_experiment(a, b, clock_bits=4)
  [...]
>>
```

```
Clock bits   : 4
Dimensions A : 4x4
  lambda[0]  : 1.0
  lambda[1]  : 2.0
  lambda[2]  : 2.0
  lambda[3]  : 8.0
Set C to min : 1.0
Classic ratio:  1.044
Classic ratio:  0.738
Classic ratio:  3.545
Quantum ratio:  1.044, delta: +0.000
Quantum ratio:  0.738, delta: -0.000
Quantum ratio:  3.545, delta: +0.000
```

15 Quantum Error Correction

Quantum computing operates at a (smaller than) microscopic scale, with a high likelihood of noise, errors, and decoherence[1] in larger circuits. Because of this, it has been believed for the longest time that practical quantum computing will not be feasible. This all changed with the discovery of quantum error correction techniques,[2] which we touch upon in this chapter. We have ignored this topic so far and assumed an ideal, error-free execution environment. However, for real machines, this assumption does not hold. Quantum error correction is a fascinating and wide-ranging topic. This section is primarily an introduction with focus on core principles.

15.1 Quantum Noise

Building a real, physical quantum computer that has a large enough number of qubits to perform useful computations presents enormous challenges. A quantum system must be isolated from the environment as much as possible to avoid entanglement with the environment and other perturbations, all of which could induce errors. For example, molecules may bump into qubits and change their relative phase, even at temperatures close to absolute zero. However, a quantum system cannot be entirely isolated because we want to program the machine, change its internal state, and make measurements.

A summary of the available technologies presented in Nielsen and Chuang (2011) is shown in Table 15.1. The table lists the underlying technology, the time τ_Q the system can remain coherent before losing coherence, the time τ_{op} it takes to apply a typical unitary gate, and the number n_{op} of operations that can be executed while still in a coherent state. For several technologies, the number of coherently executable instructions is rather small and will likely not suffice to execute larger algorithms, especially those with many qubits and millions or potentially billions of gates. However, encouraging improvements have been reported recently in Anferov et al. (2024).

Errors are inevitable at the atomic scale, and the environment is very likely to perturb the system. Let us compare the expected quantum and classical error rates.

[1] See also: http://en.wikipedia.org/wiki/Quantum_decoherence.
[2] Other techniques for decoherence reduction exist today. See, for example, http://en.wikipedia.org/wiki/Dynamical_decoupling.

Table 15.1. Estimates for decoherence times τ_Q (secs), gate application latency τ_{op}(secs), and number of gates n_{op} that can be applied while coherent. Data from Nielsen and Chuang (2011).

System	τ_Q	τ_{op}	n_{op}
Nuclear spin	10^{-2} to 10^{-8}	10^{-3} to 10^{-6}	10^5 to 10^{14}
Electron spin	10^{-3}	10^{-7}	10^4
Ion trap	10^{-1}	10^{-14}	10^{13}
Electron – Au	10^{-8}	10^{-14}	10^6
Electron – GaAs	10^{-10}	10^{-13}	10^3
Quantum dot	10^{-6}	10^{-9}	10^3
Optical cavity	10^{-5}	10^{-14}	10^9
Microwave cavity	10^0	10^{-4}	10^4

For a modern CPU, a typical error rate is about one per year, or one error for every 10^{17} operations. The actual error rate might be higher, but mitigation strategies and redundancies are in place.

In contrast, data from 2020 from IBM (Maldonado, 2022) show an average single-qubit gate error rate of about one per 10^{-3} seconds. Based on frequency, this could reach up to one error for every 200 operations. This is a difference of almost ten orders of magnitude! Next, let us explore possible quantum error conditions and model their likelihood of occurrence.

Bit-Flip Error

A *bit-flip error* causes the probability amplitudes of a qubit to flip, similar to the effect of an X gate or even a classical bit-flip error:

$$\alpha|0\rangle + \beta|1\rangle \rightarrow \beta|0\rangle + \alpha|1\rangle.$$

This is also called a *dissipation-induced* bit-flip error. Dissipation is the process of losing energy to the environment. If we think of a qubit in the state $|1\rangle$ as an electron in an excited state, as it loses energy, it may fall to the lower energy $|0\rangle$ state and emit a photon. Consequently, by absorbing a photon, it can jump from $|0\rangle$ to $|1\rangle$, in which case it should probably be called an *excitation-induced* error.

Phase-Flip Error

The *phase-flip error* causes the relative phase to flip from $+1$ to -1, similar to the effect of a Z gate:

$$\alpha|0\rangle + \beta|1\rangle \rightarrow \alpha|0\rangle - \beta|1\rangle.$$

This is also called a *decoherence-induced* phase-shift error. In the example, the phase on β was shifted by π, but for decoherence, we should also consider more minor phase changes and their insidious tendency to compound over time.

Combined Phase/Bit-Flip Error

The combination of the two error conditions above

$$\alpha|0\rangle + \beta|1\rangle \rightarrow \beta|0\rangle - \alpha|1\rangle,$$

is equivalent to applying the Y gate and ignoring the global phase:

$$Y\big(\alpha|0\rangle + \beta|1\rangle\big) = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = -i\beta|0\rangle + i\alpha|1\rangle = -i\big(\beta|0\rangle - \alpha|1\rangle\big).$$

No Error

We only mention this case for completeness. It is the effect of applying an identity gate to a qubit or, equivalently, doing nothing.[3]

Errors will occur with a certain probability. To model this properly, we will introduce the concept of quantum operations next. Using those, we can describe statistical error distributions in an elegant way.

15.1.1 Quantum Operations

So far, we have primarily focused on describing quantum states as vectors of probability amplitudes. We explained how density operators can also describe states, allowing us to describe mixtures of states. In the following, we adopt the formalism presented in Nielsen and Chuang (2011).

Similarly to how a state evolves as $|\psi'\rangle = U|\psi\rangle$ with a unitary U, a state's density operator $\rho = |\psi\rangle\langle\psi|$ evolves as

$$\rho' = \mathcal{E}(\rho),$$

where the $\mathcal{E}$ is called a *quantum operation*. The two types of operations we have encountered so far are unitary transformations and measurements with a projection operator M. Note again the matrix multiplication from both sides:

$$\mathcal{E}(\rho) = U\rho\, U^\dagger \quad \text{and} \quad \mathcal{E}_M(\rho) = M\rho M^\dagger. \tag{15.1}$$

In a *closed* quantum system, which has no interaction with the environment, the system evolves as

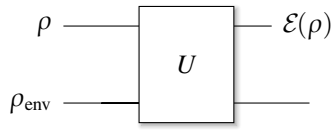
$$\rho \longrightarrow \boxed{U} \longrightarrow U\rho\, U^\dagger.$$

In an *open* system, we model the system as the tensor product of state and environment as $\rho \otimes \rho_{\text{env}}$. The system evolves as described in Equation (15.1), with U being a unitary in the Hilbert space of both the system and the environment, expressed with density matrices:

$$U(\rho \otimes \rho_{\text{env}})\, U^\dagger.$$

[3] It is impossible to resist mentioning the famous error message "No Error" in a certain operating system that shall remain unnamed.

We can visualize this with this conceptual circuit:



To describe the system without the environment, we can trace out the environment using the methodology from Section 4.3:

$$\mathcal{E}(\rho) = \mathrm{tr}_{\mathrm{env}} \left[U(\rho \otimes \rho_{\mathrm{env}}) \, U^{\dagger} \right]. \tag{15.2}$$

Now, quantum operators can be expressed in the *operator-sum representation*, which only describes the behavior of the principal system in Equation (15.2). Let $|e_k\rangle$ be the orthonormal basis of the environment, and let $|e_{\mathrm{env}}\rangle = |e_0\rangle\langle e_0|$ be the initial state of the environment, which we assume to be pure. It can be shown (see Nielsen and Chuang, 2011, section 8.2.3) that

$$\mathcal{E}(\rho) = \sum_k E_k \rho E_k^{\dagger}, \tag{15.3}$$

where $E_k = \langle e_k | U | e_0 \rangle$. The E_k are the *operation elements* for the quantum operation $\mathcal{E}$. They are also called *Kraus operators* and operate on the quantum system of interest only. Now, let us see how we can use this formalism to describe the various error modes.

15.1.2 Quantum Channels

The term *channel* is an abstraction in information theory to model how information is transmitted in the presence of noise, errors, and potential attackers intent on stealing transmitted information. In this book, we are not discussing these aspects of quantum cryptography. Viddick (2023) provides an excellent introduction to the topic. However, we can use the formalism developed therein to model quantum error modes and probabilities.

A classical channel could be as simple as the vibrations of a string in a string telephone or as complex as light traveling in a fiber optic line. In the quantum case, we have to consider that in addition to unitary operations and measurements, as discussed above, there are two other types of quantum operations that eavesdroppers can use to steal information, for example, with cloning attacks (Scarani, 2005):[4]

- We can add an ancilla qubit to a given state $|\psi\rangle$ and increase the dimension of the state by a factor of 2 as $|\psi\rangle \rightarrow |\psi\rangle\,|0\rangle$.
- We can trace out a qubit and reduce the dimension of a state by a factor of 2, as described in Section 4.3.

With these, we define a general *quantum channel* $\mathcal{C}$ as an operation from $(\mathbb{C}^2)^n \rightarrow (\mathbb{C}^2)^m$ consisting of a sequence of unitary operations, the addition of ancillas, and the tracing out of qubits.

[4] Quantum cryptography is a very large field. We will not go into great depth in this book.

15.1.3 Bit Flip and Phase Flip Channels

The *bit-flip channel* $\mathcal{C}_{\text{bit-flip}}$ flips the states from $|0\rangle$ to $|1\rangle$ with probability $1-p$ (probability p of not introducing an error). It has the operation elements

$$E_0 = \sqrt{p}\,I = \sqrt{p}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \quad \text{and} \quad E_1 = \sqrt{1-p}\,X = \sqrt{1-p}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}.$$

Expressed in the form of Equation (15.3), the bit-flip channel for a density matrix ρ is

$$\mathcal{C}_{\text{bit-flip}}(\rho) = (1-p)\rho + pX\rho X.$$

The *phase-flip channel* flips the phase as described above with probability $1-p$. It has the operation elements

$$E_0 = \sqrt{p}\,I = \sqrt{p}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \quad \text{and} \quad E_1 = \sqrt{1-p}\,Z = \sqrt{1-p}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}.$$

Finally, the *bit-flip phase-flip channel* has the operation elements

$$E_0 = \sqrt{p}\,I = \sqrt{p}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \quad \text{and} \quad E_1 = \sqrt{1-p}\,Y = \sqrt{1-p}\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}.$$

15.1.4 Depolarization Channel

The *depolarization channel* is another standard way to describe quantum noise. Depolarization means an original state is transformed into a completely mixed state $I/2$. In Section 4.3, we briefly talked about pure and mixed states and derived that a maximally mixed bipartite state is proportional to I, which means that the two subsystems are maximally entangled with each other. For the scenario we are considering, one of the systems is the environment.

Quantum noise means that if some disturbances have a probability p of changing a state, the state remains *unmodified* with probability $1-p$. The state, expressed as the density matrix ρ, can be written as

$$\rho' = p\frac{I}{2} + (1-p)\rho.$$

For an arbitrary (single-qubit) ρ, the following holds in operator-sum notation, which we verify in the test `test_rho` in file `src/lib/ops_test.py` (this equation is related to Equation (2.6)):

$$I = \frac{\rho + X\rho X + Y\rho Y + Z\rho Z}{2}.$$

Find the code
In file `src/lib/ops_test.py`

```
def test_rho(self):
  for _ in range(100):
    q = state.qubit(alpha=random.random())
    rho = q.density()
```

```
ident, x, y, z = ops.Pauli()
u = (rho + x @ rho @ x + y @ rho @ y + z @ rho @ z) / 2
self.assertTrue(np.allclose(u, ident))
```

Suppose that we assign a probability of $(1 - p)$ for a state to remain unmodified by noise and assign a probability of $1/3$ for each of the operators X, Y, and Z to introduce noise (other probability distributions are possible). In that case, the operator sum expression above becomes

$$\mathcal{E}(\rho) = (1 - p)\rho + \frac{p}{3}(X\rho X + Y\rho Y + Z\rho Z).$$

This is the result that we were looking for. It allows us to model quantum noise by injecting Pauli gates with a given probability. Assume a gate E, which may be one of the Pauli matrices with a probability as follows:

$$E = \begin{cases} X & \text{with } p_x, \\ Y & \text{with } p_y, \\ Z & \text{with } p_z, \\ I & \text{with } 1 - (p_x + p_y + p_z). \end{cases}$$

To model noise, we introduce error gates E with a given probability, injecting bit-flip and phase-flip errors. Example circuits before and after error injection are shown in Figures 15.1 and 15.2. It is educational to experiment with injecting these error gates and evaluating their impact on various algorithms. We will do just that in Section 15.2.

15.1.5 Amplitude and Phase Damping

We mention amplitude and phase damping only for completeness, but we will not elaborate further. *Amplitude damping* seeks to model *energy dissipation*, the energy loss in a quantum system. It is described by two operator elements, with γ (gamma) being

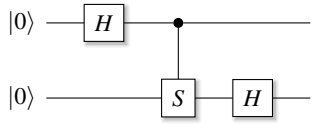


Figure 15.1 Circuit before noise injection.

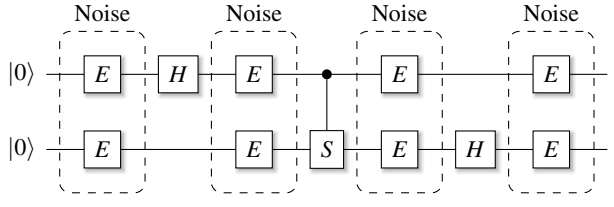


Figure 15.2 Circuit with injected noise.

the likelihood of energy loss, such as the emission of a photon in a physical system:

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix} \quad \text{and} \quad E_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix}.$$

Phase damping describes the process of a system losing relative phase between qubits, thus introducing errors in algorithms that rely on successful quantum interference. The operator elements are

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix} \quad \text{and} \quad E_1 = \begin{pmatrix} 0 & 0 \\ 0 & \sqrt{\gamma} \end{pmatrix}.$$

The factor γ might be expressed as an exponential function in more realistic modeling environments.

15.1.6 Effect of Imprecise Gates

Gates themselves may be imperfect. There could be issues with manufacturing, external influences, temperature, and other conditions, which all influence gate accuracy. Additionally, it is unlikely that all software gates we use in this text will be available on physical machines. The software gates may have to be decomposed into a series of hardware gates or be approximated. Approximations have residual errors, as we detailed in Section 9.4 on the Solovay–Kitaev algorithm.

The impact of gate imprecision varies by algorithm. We have the source code and can simulate the algorithms, which allows us to run experiments and inject various error conditions and distributions. In the following brief example, we modify the final inverse QFT in the phase estimation circuit from Section 11.2.1 by introducing errors in the R_k phase gates. To achieve this, we compute a normally distributed random number in the range of 0 to 1 and scale a noise factor n_f with it. For example, a factor $n_f = 0.1$ means that a *maximum* error of 10% can be introduced. The following code is a simple model, and you are encouraged to experiment with other error distributions.

```
def Rk(k):
  return Operator(np.array([(1.0, 0.0),
         (0.0, cmath.exp((1 + (random.random() * flags.FLAGS.noise)) *
         (2.0 * cmath.pi * 1j / 2**k)))]))
```

Then, for values of n_f ranging from 0.0 to 2.0, we perform 50 experiments and count the number of experiments that result in a phase estimation error greater than 2%. In other words, we test the robustness of phase estimation against small to large errors in the inverse QFT rotation gates. Figure 15.3 shows the distribution.

You can see that the inverse QFT is surprisingly robust against sizable maximum errors in the rotation gates. The exact result would depend on the statistical distribution of the actual errors. We should also expect that each algorithm has different tolerances and sensitivities. For comparison, introducing depolarization with only 0.1% proba-

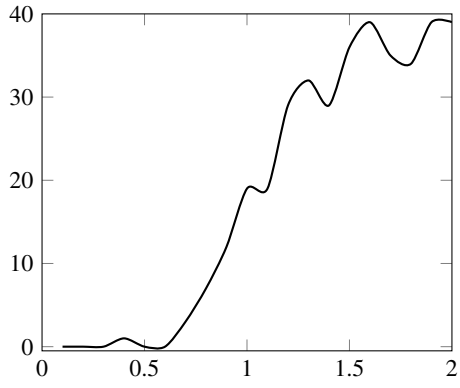


Figure 15.3 Phase estimation errors exceeding a threshold of 2% from increasing noise levels, for N = 50 experiments per setting. The *x*-axis represents noise ranging from 0% to 200%, and the *y*-axis represents the number of experiments exceeding the threshold. The number 40 corresponds to 80% of the 50 experiments conducted.

bility leads to significantly different outcomes in the order finding algorithm, which is very sensitive to this particular type of error.[5]

15.2 Quantum Error Correction

Moving forward, we will need some form of error correction techniques to control the impact of noise. In classical computing, there is a large body of known error-correcting techniques. Error correction code memory, or ECC (Wikipedia, 2021b), may be one of the best known. There are many more techniques to prevent invalid data, missing data, or spurious data. NASA, in particular, has developed impressive techniques to communicate with its ever-more-distant exploratory vehicles.

A simple classical error correction technique uses repetition codes and majority voting. For example, we could triple each binary digit:

$$0 \rightarrow 000,$$
$$1 \rightarrow 111.$$

As we receive data over a noisy channel, we measure it and perform majority voting with the scheme shown in Table 15.2. This simple scheme does not account for missing or erroneous bits but is a good start to explain basic principles. In quantum computing, the situation is generally more difficult:

- Physical quantum computers operate at the quantum level of atomic spins, photons, and electrons, which are all very sensitive systems. There is a high probability of encountering errors or decoherence, especially for longer-running computations.
- Errors can be more subtle than simple bit flips. There are multiple error modes.

[5] This behavior could be because an angle error will still lead to a pure state, while depolarizing errors cause the state to become mixed, leading to a loss in coherence.

Table 15.2. Majority voting for a simple repetition code.

Measured	Voted	Measured	Voted
000	0	111	1
001	0	110	1
010	0	101	1
100	0	011	1

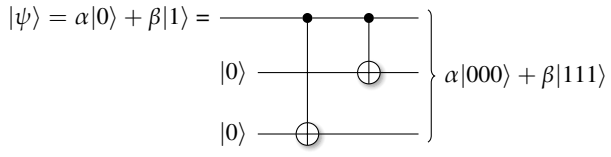


Figure 15.4 A circuit to produce a quantum repetition code. In essence, it makes a GHZ state.

- Errors, such as relative phase errors, compound during execution.
- Simple repetition codes will not work because of the no-cloning theorem.
- Most problematically, you cannot directly observe errors, as that would constitute a measurement that destroys the superposition and entanglement that an algorithm relies on.

Because of these difficulties and especially because of the inability to read a corrupted state, early speculation was that error correction codes could not exist. Hence, it would be nearly impossible to produce a viable quantum computer (Rolf, 1995; Haroche and Raimond, 1996). Fortunately, this changed when Shor presented a viable nine-qubit error correction code (Shor, 1995). The principles of this approach are the foundation of many quantum error correction techniques in use today.

15.2.1 Quantum Repetition Code

We can make a quantum repetition code with the circuit shown in Figure 15.4. Note its similarity to the GHZ circuit in Section 2.11.4. To illustrate how it works, we can use the following code snippet to produce the state and dump the state vector:

```
    qbit = state.qubit(random.random())
    psi = qbit * state.zeros(2)
    psi = ops.Cnot(0, 2)(psi)
    psi = ops.Cnot(0, 1)(psi)
    psi.dump()
>>
|000> (|0>):  ampl: +0.78+0.00j prob: 0.61 Phase:   0.0
|111> (|7>):  ampl: +0.62+0.00j prob: 0.39 Phase:   0.0
```

15.2.2 Correct Bit Flip Errors

Here is the main *trick* to error correction, which is related to quantum teleportation. First, we introduce redundancy and triple each single-qubit state into a GHZ state, as shown in Figure 15.4. We entangle this three-qubit state with two ancillae and measure *only* the ancillae, leaving the original state intact. Based on the measurement outcome, we apply gates to the original three-qubit state to correct it.

Figure 15.5 shows this procedure in circuit notation, assuming a single qubit-flip error for qubit 0, which we indicate on the left side of the circuit. The state $|\psi_1\rangle$ right before a measurement, where the bottom two qubits have been flipped to $|10\rangle$, is

$$|\psi_1\rangle = \alpha|10010\rangle + \beta|01110\rangle.$$

Right after measurement, this turns into

$$|\psi_2\rangle = \big(\alpha|100\rangle + \beta|011\rangle\big) \otimes |10\rangle.$$

This measurement result is also called an *error syndrome*. Based on the syndrome, we decide what to do next and which qubit to flip back with another X gate:

- For a measurement result of $|00\rangle$, do nothing.
- For a measurement result of $|01\rangle$, apply X gate to qubit 2.
- For a measurement result of $|10\rangle$, apply X gate to qubit 0.
- For a measurement result of $|11\rangle$, apply X gate to qubit 1.

The way Figure 15.5 is drawn is somewhat sloppy because the R gate is different for each measurement result. Making physical measurements and reacting to the outcome is not a realistic scenario; it would be hard to achieve in practice, and even if it worked, it would likely destroy a quantum computer's performance advantage because of Amdahl's law.[6] In larger circuits, we should also disentangle the ancillae.

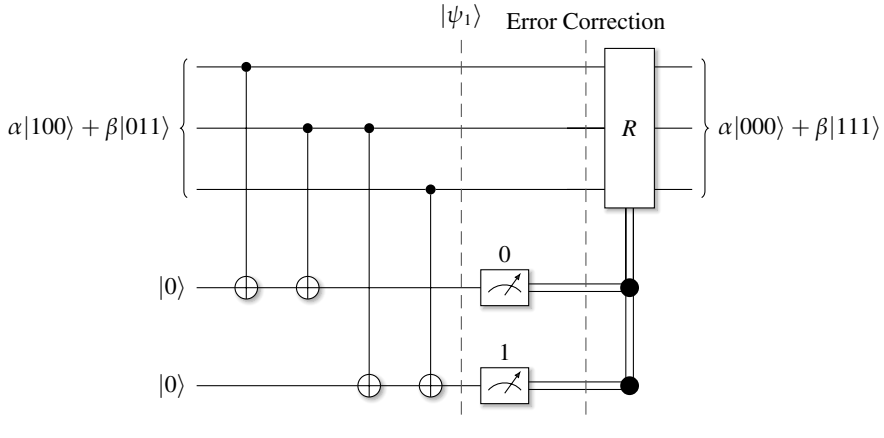


Figure 15.5 Bit-flip error correcting circuit. After turning the state into a GHZ state, we entangle it with two ancillae and only measure the ancillae. Correction gates are applied based on the measurement outcome.

[6] http://en.wikipedia.org/wiki/Amdahl%27s_law.

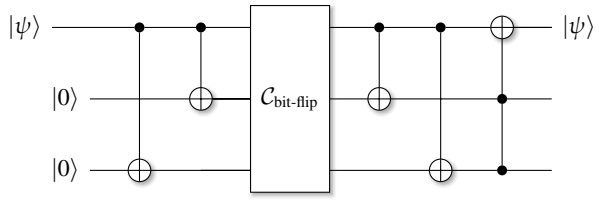


Figure 15.6 Error correction for bit-flip error.

Hence, a common practical construction to correct for bit-flip errors can be found in the circuit in Figure 15.6. We know that the noisy channel $\mathcal{C}$ introduces bit-flip errors according to Equation (15.4) as

$$\mathcal{C}_{\text{bit-flip}}(\rho) = (1-p)\rho + p(X\rho X). \tag{15.4}$$

In code, we can simulate this and inject an error by introducing an X gate.

PY

Find the code
In file src/lib/circuit_test.py

```
def test_x_error(self):
  qc = circuit.qc('x-flip / correction')
  qc.qubit(0.6)

  # Replication code setup.
  qc.reg(2, 0)
  qc.cx(0, 2)
  qc.cx(0, 1)
  qc.psi.dump('after setup')

  # Error insertion.
  qc.x(0)

  # Fix.
  qc.cx(0, 1)
  qc.cx(0, 2)
  qc.ccx(1, 2, 0)
  qc.psi.dump('after correction')
```

If no error has been injected, we will see this output:

```
|210> 'after setup'
|000> (|0>):  ampl: +0.60+0.00j prob: 0.36 Phase:   0.0
|111> (|7>):  ampl: +0.80+0.00j prob: 0.64 Phase:   0.0
|210> 'after correction'
|000> (|0>):  ampl: +0.60+0.00j prob: 0.36 Phase:   0.0
|100> (|4>):  ampl: +0.80+0.00j prob: 0.64 Phase:   0.0
```

If an error has indeed been injected, the state becomes:

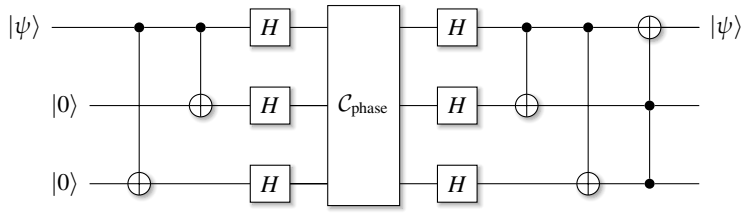


Figure 15.7 Error correction for phase-flip error.

```
|210> 'after setup'
|000> (|0>):  ampl: +0.60+0.00j prob: 0.36 Phase:   0.0
|111> (|7>):  ampl: +0.80+0.00j prob: 0.64 Phase:   0.0
|210> 'after correction'
|011> (|3>):  ampl: +0.60+0.00j prob: 0.36 Phase:   0.0
|111> (|7>):  ampl: +0.80+0.00j prob: 0.64 Phase:   0.0
```

Note the slight difference in the final states with nonzero probabilities. Without an injected error, the state becomes $|000\rangle$ and $|100\rangle$, the original input state. With an injected error, the ancilla qubits become $|11\rangle$ for both resulting states (which may require uncomputation if we wanted to reuse the ancillae).

15.2.3 Correct Phase-Flip Errors

We can use the same idea to correct phase-flip errors. Recall how the application of Hadamard gates puts a state from the computational basis into the Hadamard basis.[7] A phase-flip error in the computational basis is the same as a bit-flip error in the Hadamard basis. Correspondingly, we can use the circuit in Figure 15.7 to create a quantum repetition and error correction circuit, similar to Figure 15.6, but with surrounding Hadamard gates. We use a similar code sequence for this as above, but we change it to the following for error injection:

```
    [...]
    qc.h([0, 1, 2])
    qc.z(0)
    qc.h([0, 1, 2])
    [...]
```

The probability distribution of the resulting nonzero probability states is the same, but we get a few states with different phases. For example, without error injection:

```
|210> 'after setup'
|000> (|0>):  ampl: +0.60+0.00j prob: 0.36 Phase:   0.0
|111> (|7>):  ampl: +0.80+0.00j prob: 0.64 Phase:   0.0
```

[7] You may want to convince yourself of this mathematically.

```
|210> 'after correction'
|000> (|0>):  ampl: +0.60+0.00j prob: 0.36 Phase:   0.0
|001> (|1>):  ampl: +0.00+0.00j prob: 0.00 Phase:   0.0
|010> (|2>):  ampl: -0.00+0.00j prob: 0.00 Phase: 180.0
|011> (|3>):  ampl: -0.00+0.00j prob: 0.00 Phase: 180.0
|100> (|4>):  ampl: +0.80+0.00j prob: 0.64 Phase:   0.0
|101> (|5>):  ampl: +0.00+0.00j prob: 0.00 Phase:   0.0
|110> (|6>):  ampl: +0.00+0.00j prob: 0.00 Phase:   0.0
```

15.3 Nine-Qubit Shor Code

All of what we have done so far leads to the final nine-qubit Shor code (Shor, 1995). It combines the circuits to find bit-flip, phase-flip, and combined errors into one large circuit, as shown in Figure 15.8.

PY

Find the code
In file `src/lib/circuit_test.py`

The Shor nine-qubit circuit can identify and correct one bit-flip error, one phase-flip error, or one of each on a single qubit! Let us verify this in code and apply all Pauli gates to each of the qubits of this circuit. For this experiment, we construct a qubit with the factor $\alpha = 0.60$ to the $|0\rangle$ basis state:

```
def test_shor_9_qubit_correction(self):
  for i in range(9):
```

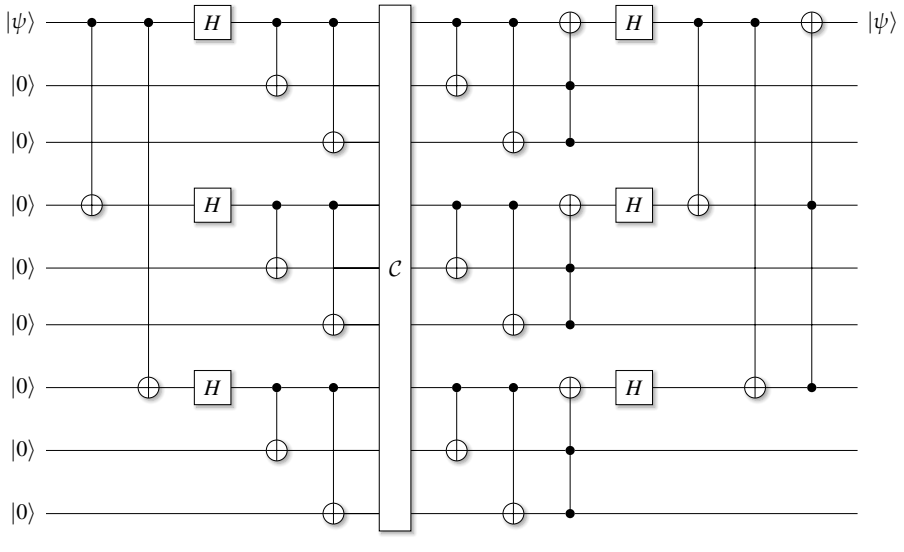


Figure 15.8 Shor's nine-qubit error correction circuit.

```
qc = circuit.qc('shor-9')
print(f'Init qubit as 0.6|0> + 0.8|1>, error on qubit {i}')
qc.qubit(0.6)
qc.reg(8, 0)

# Left Side.
qc.cx(0, 3)
qc.cx(0, 6)
qc.h(0); qc.h(3); qc.h(6);
qc.cx(0, 1); qc.cx(0, 2)
qc.cx(3, 4); qc.cx(3, 5)
qc.cx(6, 7); qc.cx(6, 8)

# Error insertion, use x(i), y(i), or z(i)
qc.x(i)

# Fix.
qc.cx(0, 1); qc.cx(0, 2); qc.ccx(1, 2, 0)
qc.h(0)
qc.cx(3, 4); qc.cx(3, 5); qc.ccx(4, 5, 3)
qc.h(3)
qc.cx(6, 7); qc.cx(6, 8); qc.ccx(7, 8, 6)
qc.h(6)

qc.cx(0, 3); qc.cx(0, 6)
qc.ccx(6, 3, 0)

prob0, s = qc.measure_bit(0, 0)
prob1, s = qc.measure_bit(0, 1)
print('          Measured: {:.2f}|0> + {:.2f}|1>'.format(
    math.sqrt(prob0), math.sqrt(prob1)))
```

We should see the following results with the corresponding output:

```
Initialize qubit as 0.60|0> + 0.80|1>, error on qubit 0
          Measured: 0.60|0> + 0.80|1>
[...]
Initialize qubit as 0.60|0> + 0.80|1>, error on qubit 8
          Measured: 0.60|0> + 0.80|1>
```

This is all we will cover in this section. Quantum error correction is a vibrant field. Numerous other techniques and formalisms exist for quantum information and quantum error correction. As a next step, it may be of interest to read the comprehensive overview in Devitt et al. (2013) or refer to standard textbooks (Nielsen and Chuang, 2011).

16 Quantum Languages, Compilers, and Tools

At this point, we understand the principles of quantum computing, the important foundational algorithms, and the basics of quantum error correction. We have developed a compact and reasonably fast classical infrastructure for simulation and experimentation. The infrastructure is working as intended but may be a long way from enabling high productivity, as composing algorithms at this level of abstraction is both labor-intensive and error-prone. We can build circuits with maybe 10^6 gates, but some more realistic solutions may require trillions of gates with orders of magnitude more qubits.

In classical computing, programs are constructed at higher levels of abstraction using programming *languages*, which allows the targeting of several general-purpose architectures in a portable way. On a high-performance CPU, programs execute billions of instructions per second on just a single core. Building quantum programs on that scale with a "flat" programming model such as QASM, which stitches together individual gates, does not scale to large programs. We discuss QASM below in Section 16.3.1. This approach is the equivalent of programming today's classical machines in assembly language and, to make it even more interesting, without control-flow constructs.

There are parallels to the 1950s, where assembly language[1] was the trade of the day to program early computers. That is when FORTRAN[2] emerged as one of the first compiled programming languages, unlocking major productivity gains. In quantum computing, there are similar attempts today to develop quantum programming languages that try to raise the abstraction level and make programming quantum computers easier, safer, and more productive.

This chapter discusses a representative cross-section of quantum programming languages and briefly touches on productivity tooling, such as simulators or entanglement analysis. There is also a discussion of quantum compiler optimizations, a fascinating topic with unique challenges. We write this chapter with the understanding that comparisons between toolchains are necessarily incomplete but educational nonetheless.

16.1 Challenges for Quantum Compilation

The design of compilers for quantum computing presents distinct challenges. This section outlines some of the main difficulties. The subsequent sections will provide further details and suggest possible solutions.

[1] And even cruder, switches!

[2] See also http://en.wikipedia.org/wiki/Fortran.

Quantum computing needs a programming model – what will run, how, when, and where? Today, the most common classical coprocessor is a graphics processing unit (GPU). GPUs provide massive parallelism, but computation itself is still expressed in terms of programming a CPU core. There are just a boatload of those cores with additional abilities and constraints to support parallelism, paired with a dynamic runtime to manage the various compute kernels on device.

Quantum computers are unlikely to offer general-purpose functionality similar to that of a CPU. Instead, we should expect a classical machine to entirely control the quantum computer, moving programs and data in and out of the machine. A model called *QRAM* was proposed early in the history of quantum computing (Knill, 1996). We will discuss this model in Section 16.2. Note that today's use of QRAM is different and typically refers to "Quantum Random Access Memory," a broad set of techniques to store and retrieve values as quantum states.

A key question is how realistic this idealized model can be. Quantum circuits operate at micro-Kelvin temperatures. It will be a challenge for standard CPU manufacturing processes to operate at this temperature, although progress has been made (Patra et al., 2020). The CPU could alternatively operate physically distant from the quantum circuit, but then the bandwidth between classical and quantum circuits may be severely limited. Some recent research can be found in Xue et al. (2021).

Constructing quantum circuits gate-by-gate is tedious and error-prone. There are challenges such as the no-cloning theorem and the need for automatic error correction. Programming languages offer a higher level of abstraction and will be essential for programmer productivity. But what is the "right" level of abstraction? We sample several existing approaches to quantum programming languages in Section 16.3. Compiler construction and intermediate representation (IR) design are challenges in themselves. It seems apparent that a flat, QASM-like, linked-list IR will not scale to programs with trillions of gates.

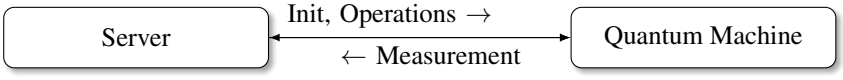
The required precision of the gates is an important design parameter. We will have to approximate certain unitaries by sequences of existing physical gates, which introduces inaccuracies and noise. Some algorithms are robust against noise, others not at all. The toolchain also plays an essential role in this area.

Aspects of dynamic code generation may become necessary, for example, to approximate specific rotations dynamically or to reduce noise (Wilson et al., 2020). There are challenges in fast gate approximation, compile time, accuracy, and optimality of approximated gate sequences. To give a taste of these problems, we have already detailed the Solovay–Kitaev algorithm in Section 9.4.

Compiler optimization has a novel set of transformations to consider in an exponentially growing search space. We are currently in the era of physical machines with up to 1000 physical qubits, the noisy intermediate-scale quantum computers (Preskill, 2018). Future systems will have more qubits and qubits with likely different characteristics than today's qubits. Compiler optimizations and code generation techniques will have to evolve accordingly as well. We discuss several optimization techniques in Section 16.4.

16.2 Quantum Programming Model

As our standard model of computation, we assume the old quantum random access model (QRAM) proposed by Knill (1996) (again, this name means something entirely different today). The model proposes connecting a general-purpose machine with a quantum computer to use it as an accelerator. Registers are explicitly quantum or classical. There are functions to initialize quantum registers, to program gate sequences into the quantum device, and to get results back via measurements.



On the surface, this model is not very different from today's programming models for PCIe-connected accelerators,[3] such as GPUs or storage devices, which are ubiquitous today. The elegant CUDA programming model for GPUs provides clear abstractions for code that is supposed to run on either device or server (Buck et al., 2004; Nickolls et al., 2008). The program source code for the accelerator and the host can be mixed in the same source file to enhance programmer productivity.

QRAM is an idealization. Communication between the classical and quantum parts of a program may be severely limited. There may be a significant lack of computing power close to the quantum circuit, which operates at micro-Kelvin temperature, or bandwidth-limited communication to a CPU further away.

It is important to keep the separation between classical and quantum in mind. In QRAM (as in our simulation infrastructure), the separation of classical and quantum is muddled because it runs classical loops over applications of quantum gates interspersed with print statements. This might be a good approach to learning, but it may not be realistic for a real machine. The approach is more akin to an infrastructure such as the machine learning framework TensorFlow, which first builds up computation as a graph before executing the graph in a distributed fashion on CPU, GPU, or TPU.

Another aspect of the QRAM model is the expectation of available *universal* gates on the target quantum machine. Several universal sets of gates have been described in the literature (see Nielsen and Chuang, 2011, section 4.5.3). We showed how any unitary gate can be approximated by universal gates in Section 9.4. With this in mind, we assume that any gate may be used freely in our idealized infrastructure.

With these preliminaries and simplifications, let us now explore a range of approaches that have been taken to make the programming of quantum machines more productive and less error-prone.

16.3 Quantum Programming Languages

This section discusses a representative cross-section of quantum programming languages with corresponding compilers and tooling. The descriptions are brief and, therefore, necessarily incomplete. Most importantly, the selection does not judge the

[3] http://en.wikipedia.org/wiki/PCI_Express.

quality of the non-selected languages. A more complete collection of references to quantum programming languages and systems can be found in Quantum Programming (2024).

If we were to build a hierarchy of programming abstractions, we should consider these levels:

- The high abstraction level of programming languages. This level may provide automatic ancilla management, support correct program construction with advanced typing rules, offer libraries for standard operations (such as QFT), and perhaps offer meta-programming techniques (such as C++ templates).
- Programming at the gate level. This is the level of abstraction that we mostly used in this text. At this level, we directly construct and manipulate individual qubits and gates. We also manage ancillae and uncomputation manually and explicitly.
- Direct machine control with pulses and waveforms operating on the physical device. We will not discuss related infrastructure, such as OpenPulse (Gokhale et al., 2020).

For each of these levels, several alternative implementations and approaches are available online, many with material for learning and experimentation. In the following sections, we highlight some selected, perhaps seminal examples. Many of the features that we will explore should inspire you to think about how to make them available in simpler frameworks, such as the one presented in this book.[4]

16.3.1 QASM

The quantum assembly language (QASM) was an early attempt to standardize a textual specification of quantum circuits (Svore et al., 2006). The structure of a QASM program is very simple. Qubits and registers are declared upfront, and gate applications follow one by one. There are no looping constructs, function calls, or other constructs that would help to structure and densify the code. As an example, a simple entangler circuit would be written as follows:

```
qubit x,y;
gate h;
gate cx;
h x;
cx x,y;
```

More capable variants emerged that augment QASM in a variety of ways. OpenQASM adds the ability to define new gates, control flow constructs, and barriers (Cross et al., 2017). It also offers looping constructs. cQASM is one attempt to unify QASM dialects into a single form[5] (Khammassi et al., 2018). It offers additional language features, such as explicit parallelization, register mapping/renaming, and a variety of

[4] As a matter of fact, we did some of this already, for example, for the automatic control of a subcircuit.

[5] See also `xkcd` cartoon #927 on attempts to replace N existing standards with a single new standard, the result, of course, being that now we have $N + 1$ standards.

measurement types. An example implementation of a three-qubit Grover algorithm takes about 50 lines of code (Quantum Inspire, 2024).

16.3.2 QCL

The *quantum computing language* (QCL) was an early attempt to use classical programming constructs to express quantum computation (Ömer, 2000, 2005; QCL Online). Algorithms run on a classical machine controlling a quantum computer and might have to run multiple times until a solution is found. The quantum and classical codes are intermixed. Qubits are defined as registers of a given length, and gates are applied directly to the registers. For example:

```
qureg q[l];          // Define quantum register q of length l
qureg f[l];          // Define quantum register q of length l
H(q);                // Hadamard gates on register q
Not(f);              // X gates on register f
const n=#q;          // classical length of q register
for i=1 to n {       // classical loop
   Phase(pi/2^(i));  // quantum phase gate at implicit index i
}
```

QCL defines several quantum register types: A `qureg` is an unrestricted qubit, `quconst` defines an invariant qubit, and `quvoid` specifies a register to be *empty* (it is guaranteed to be initialized in state $|0\rangle$). The register type `quscratch` denotes ancillae. Gates have specific names, such as `H`, `Not`, or `Phase`.

The code is organized into *quantum functions*. Operators and functions of operators are reversible by definition, making the uncomputation of an operator easy. Prefixing a function with an exclamation point produces the inverse, as in this example from the Grover algorithm[6] (calling `!diffuse(reg)` would call the inverse operator):

```
operator diffuse(qureg q) {
  H(q);                // Hadamard transform
  Not(q);              // Invert q
  CPhase(pi,q);        // Rotate if q=1111...
  !Not(q);             // Undo inversion
  !H(q);               // Undo Hadamard transform
}
```

QCL defines several types of functions, such as the non-reversible `procedure`, which may contain classical code and allow side effects. Functions marked as `operator` and `qufunct` are guaranteed to be free from side effects and reversible. To facilitate uncomputation, QCL supports a `fanout` operation. It restores scratch and auxiliary registers while preserving the results, as described in Section 2.12 on uncomputation. Let's take a closer look at the fanout operation.

[6] Of course, both the Hadamard and the NOT gates are their own inverses. This might not be the most convincing example.

Assume that $f(x)$ is the function we want to compute and $g(x)$ is a byproduct of the computation, some random state that ended up in the ancillary qubits. As described in Section 2.12, the first step performs the computation, with the desired result ending up in the bottom register (in the example below) and some ancillary state in the middle register. The second fanout step connects the bottom register holding the results with the target register, the first register in the example. The third step performs the uncomputation. The result $f(x)$ is now available in the top register, and the other registers are properly uncomputed:[7]

$$\begin{aligned} |x,0,0,0\rangle &\rightarrow |x,0,g(x),f(x)\rangle \\ &\rightarrow |x,f(x),g(x),f(x)\rangle \\ &\rightarrow |x,f(x),0,0\rangle. \end{aligned}$$

The implementation of `fanout` is quite elegant. Assume a function $F(x,y,s)$ with x being the input, y being the output, and s being junk qubits. Allocate the ancilla t and transform F into the following, adding t to its signature:

$$F'(x,y,s,t) = F^{\dagger}(x,t,s)\ \texttt{fanout}(t,y)\ F(x,t,s).$$

What makes this elegant is the fact that `fanout` is written in QCL itself:

```
cond qufunct fanout(quconst ancilla, quvoid b) {
    int i;
    for i=0 to #ancilla-1 {
        CNot(b[i], ancilla[i]);
    }
}
```

QCL supports conditionals in interesting ways. Standard controlled gates are supported as described in Section 2.8. Suppose a function signature is marked with the keyword `cond` and has as a parameter the `quconst` condition qubit. In that case, QCL automatically transforms the operators of the function into controlled operators. Here is an example of such a function signature for a controlled function `inc`:

```
  cond qufunct cinc(qureg x, quconst e) {...}
```

Additionally, QCL supports an `if` statement, where `if e {inc(x);}` is equivalent to a new function `cinc(x, e)` as shown above, with the if-then-else statement translating into:

```
  if e {
      inc(x);
  } else {
      !inc(x);
  }
=>
  cinc(x, e);
```

[7] This notation is a little bit sloppy. See section 2.12 on uncomputation for details.

```
  Not(e);
  !cinc(x, e);
  Not(e)
```

Here is an example implementation of a QFT in QCL (Ömer, 2000):

```
cond qufunct flip(qureg q) {
  int i;                // declare loop counter
  for i=0 to #q/2-1 {   // swap 2 symmetric bits
    Swap(q[i],q[#q-i-1]);
  }
}

operator qft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=1 to n {
    for j=1 to i-1 {    // apply conditional phase gates
      V(pi/2^(i-j),q[n-i] & q[n-j]);
    }
    H(q[n-i]);          // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

16.3.3 Scaffold

Scaffold takes a different approach (Javadi-Abhari et al., 2014). It *extends* the open-source LLVM compiler (Lattner and Adve, 2004) and its Clang-based front end for C/C++. Scaffold introduces the data types *qbit* and *cbit* to distinguish quantum data from classical data. Quantum gates, such as the *X* or Hadamard gates, are implemented as *built-ins*, the equivalent of opaque function calls. The compiler recognizes them as such and can reason internally about them in transformation passes.

Scaffold supports a hierarchical code structure through *modules*, which are specially marked functions. Quantum circuits do not support calls and returns, so modules representing subcircuits need to be *instantiated*, similar to how Verilog modules are instantiated in a hardware design. Modules must be reversible, either by design or via automatic compiler transformations.

Scaffold offers convenient functionality for converting classical circuits to quantum gates via the Classical-To-Quantum-Circuit (CTQC) tool. This tool is of great utility for quantum algorithms that perform classical computation in the quantum domain. CQTC emits QASM assembly. To enable whole-program optimization, Scaffold has a QASM to LLVM IR transpiler, which can be used to import QASM modules, enabling further cross-module optimization.

Modules are parameterized. This means that the compiler has to manage module instantiation, for example, with help of IR duplication. This can lead to sizable code

bloat and corresponding extreme compile times. The example given is the following code snippet, where the module `Oracle` would have to be instantiated $N = 3000$ times via the outer loop, with an additional factor of 3 from the inner loop. Clearly, a parameterized IR would alleviate this problem considerably.

```
#define N 3000 // iteration count
module Oracle (qbit a[1], qbit b[1], int j) {
  double theta = (-1)*pow(2.0, j)/100;
  X(a[0]);
  Rz(b[0], theta);
}

module main () {
  qbit a[1], b[1];
  int i, j;
  for (i=1; i<=N; i++) {
    for (j=0; j<=3; j++) {
       Oracle(a, b, j);
    }
  }
}
```

As a result, Javadi-Abhari et al. (2014) report compile times ranging from 24 hours to several days for a large triangle-finding problem with size $n = 15$ (see also Magniez et al., 2005).

Hierarchical QASM

Scaffold intends to scale to very large circuits. The existing QASM model, as shown above, is *flat*, which is not suitable for large circuits. One of the contributions of Scaffold is the introduction of *hierarchical* QASM. Additionally, the compiler employs heuristics to decide which code sequences to flatten or keep in a hierarchical structure. For example, the compiler distinguishes between *forall* loops to apply a gate to all qubits in a register and *repeat* loops, such as those required for Grover iterations.

Entanglement Analysis

Scaffold includes tooling for *entanglement analysis*. In the development of Shor's algorithm, we observed a certain ancilla qubit that was still entangled after modular addition. How does an automatic tool reason about this?

Scaffold tracks entanglement-generating gates, such as controlled Not gates, on a stack. As inverse gates are executed during uncomputation in reverse order, items are popped off the stack. If, for a given qubit, no more entangling gates are found on the stack, the qubit is marked as likely *unentangled*. As a result of the analysis, the generated output can be decorated to show the estimated remaining entangled qubits:

```
module EQxMark_1_1 ( qbit* b , qbit* t ) {
...
Toffoli ( x[0] , b[1] , b[0] );
// x0, b1, b0
Toffoli ( x[1] , x[0] , b[2] );
// x1, x0, b2, b1, b0
...
}
// Final entanglements:
// (t0, b4, b3, b2, b1, b0);
```

16.3.4 Q language

We can contrast the compiler-based work in Section 16.3.3 with a pure C++-embedded approach, as presented in Bettelli et al. (2003). This approach consists of a library of C++ classes modeling quantum registers, operators, operator application, and other functions, such as the reordering of quantum registers. During program construction, the class library builds an internal data structure to represent the computation, similar in nature to the infrastructure we developed in this book. It is interesting to think about the question of which approach makes more engineering sense:

- Extension of the C/C++ compiler with specific quantum types and operators, as in Scaffold, or
- A C++ class library as in the *Q language*.

Both approaches appear equally powerful in principle. The compiler-based approach benefits from a large set of established compiler passes, such as inlining, loop transformations, redundancy elimination, and many other scalar, loop, and inter-procedural optimizations. The C++ class library has the advantage that the management of the IR, all optimizations, and final code generation schemes are maintained *outside* of the compiler. Since compilers can prove impenetrable for non-compiler experts, this approach might have a maintenance advantage, but at the cost of potentially having to reimplement many standard optimization passes.

16.3.5 Quipper

Haskell is a popular choice for programming language theorists and enthusiasts. A major reason for this is Haskell's powerful type system. An example of a Haskell-embedded implementation of a quantum programming system can be found with the Quantum IO Monad (Altenkirch and Green, 2013). Another even more rigid example is van Tonder's proposal for a λ-calculus to express quantum computation (van Tonder, 2004).

What these approaches have in common is the attempt to guarantee correctness by construction with support of the type system. This is also one of Quipper's core design ideas (Green et al., 2013; Quipper Online, 2021). Quipper is an embedded DSL in Haskell. At the time of Quipper's publication, Haskell lacked linear types, which

could have guaranteed that objects were only referenced once, as well as dependent types, which are types tied to a value. Dependent types, for example, allow you to distinguish a QFT operator over n qubits from one over m qubits.

Quipper is designed to scale and handle large programs with up to 10^{12} operators. Quipper has a notion of the *scope* of an ancilla, with the ability to reason about live ranges. With this, allocating ancilla qubits turns into a register allocation problem. Unfortunately, the programmer still has to mark ancilla live ranges explicitly.

At the language level, qubits are held in variables and gates are applied to these variables. For example, this is the code to generate a Bell state:

```
bell :: Qubit -> Qubit -> Circ (Qubit, Qubit)
bell a b = do
   a <- hadamard a
   (a, b) <- controlled_not a b
   return (a, b)
```

To control an entire block of gates, Quipper offers a `with_controls` construct, similar to QCL's `if` blocks. Another block-level construct allows for the explicit management of ancillae via the `with_ancilla` construct. Circuits defined this way can be reversed with a `reverse_simple` construct. Quipper's type system distinguishes different types of quantum data, such as simple qubits, or fixed-point interpretations of multiple qubits.

Automatic Oracles

Quipper offers tooling for the automatic construction of oracles. Typically, oracles are constructed with the following four manual steps:[8]

1. Build a classical oracle, such as a permutation matrix.
2. Translate the classic oracle into classical circuits.
3. Compile classical circuits into quantum circuits, potentially using additional ancillae. We saw examples of this in Section 5.2.
4. Finally, make the oracle reversible, typically with an XOR construction to another ancilla.

Quipper utilizes Template Haskell to automate steps 2 and 3. The approach has high utility and has been used to synthesize millions of gates in a set of benchmarks. In direct comparison to QCL on the Binary Welded Tree algorithm, it appears that QCL generates significantly more gates and qubits than Quipper. On the other hand, Quipper appears to generate more ancillary qubits.

Despite tooling, type checks, oracle automation, and utilization of the Haskell environment, it still took 55 person-months to implement the 11 algorithms in a given benchmark set (IARPA, 2010). This is undoubtedly a productivity improvement over manually constructing all the benchmarks at the gate level, but it still compares unfavorably against programmer productivity on classical infrastructure.

[8] Open-source implementations are available for these techniques, for example (Soeken et al., 2019).

Quipper led to interesting follow-up work, such as Proto-Quipper-M (Rios and Selinger, 2018), Proto-Quipper-S (Ross, 2017), leading to Proto-Quipper-D (Fu et al., 2020). These attempts are steeped in type theory and improve on program correctness by a variety of techniques, for example, using linear types to enforce the no-cloning theorem and linear dependent types to support the construction of type-safe families of circuits.

16.3.6 Silq

Based on a fork of the PSI probabilistic programming language (PSI Online, 2021), *Silq* is another step in the evolution of quantum programming languages, supporting safe and *automatic* uncomputation (Bichsel et al., 2020). It explicitly distinguishes between the classical and quantum domains with syntactical constructs. Giving the compiler the responsibility for safe uncomputation leads to two major benefits. First, the code becomes more compact. Direct comparisons with Quipper and Q# show significant code size savings for Silq in the range of 30% to over 40%. Second, the compiler may choose an optimal strategy for uncomputation, minimizing the required ancillae. As an added benefit, the compiler may choose to skip uncomputation for simulation altogether and just renormalize states and unentangle ancillae.

Many of the Haskell-embedded DSLs bemoan either the absence of linear types or the difficulties in handling constants. Silq resolves this by using linear types for non-constant values and a standard type system for constants. This leads to safe semantics, even across function calls, and the no-cloning theorem falls out naturally. Function type annotations are used to aid the type checker:

- The annotation `qfree` indicates that a function can be classically computed. For example, the quantum *X* gate is considered `qfree`, while the superposition-inducing Hadamard gate is not.
- Function parameters marked as `const` are preserved and not *consumed* by a function. They continue to be accessible after a function call. Parameters not marked as `const` are no longer available after the function call. Functions with only `const` parameters are called `lifted`.
- Functions marked as `mfree` promise not to perform measurements and are reversible.

Silq supports other quantum language features, such as function calls, measurements, explicit reversal of an operator via `reverse`, and an `if-then-else` construct that can be classical or quantum, similar to other quantum languages. Looping constructs must be classical. As an improvement over previous approaches, Silq supports Oracle construction with quantum gates.

With the annotations and the corresponding operational semantics, Silq can safely deduce which operations are safe to reverse and uncompute, even across function calls. There are many examples of potentially hazardous corner cases that are being handled correctly (Bichsel et al., 2020).

As a program example, this code snippet solves one of the challenges in Microsoft's Q# Summer 2018 coding contest:[9]

Given classical binary string $b \in \{0,1\}^n$ *with* $b[0] = 1$*, return the state* $\frac{1}{\sqrt{2}}(|b\rangle + |0\rangle)$*, where* $|0\rangle$ *is represented using n qubits.*

The code itself demonstrates several of Silq's features, for example, using the exclamation point `!` to denote classical values and types.

```
def solve[n:|N|](bits:|!B|^n){
  // prepare superposition between 0 and 1
  x:=H(0:|!B|);
  // prepare superposition between bits and 0
  qs := if x then bits else (0:int[n]) as |!B|^n;
  // uncompute x
  forget(x=qs[0]); // valid because bits[0]==1
  return qs;
}

def main(){
  // example usage for bits=1, n=2
  x := 1:|!|int[2];
  y := x as |!B|^2;
  return solve(y);
}
```

16.3.7 Commercial Systems

Commercial systems are open-source infrastructures maintained by commercial entities. The most important systems appear to be IBM's Qiskit (Gambetta et al., 2019), Microsoft's Q# (Microsoft Q#, 2021), Google's Cirq (Google, 2021c), and ProjectQ (Steiger et al., 2018). Microsoft's Q# is a functional standalone language and a part of the Quantum Developer Kit (QDK). Qiskit, Cirq, and ProjectQ all provide Python embeddings. By the time you read this, others may have become more popular.

These ecosystems are vast, fast-evolving, and provide excellent learning materials we do not have to cover here. For further reading, we recommend (Garhwal et al., 2021), which details Q#, Cirq, ProjectQ, and Qiskit, or Chong et al. (2017), which describes some of the major challenges for quantum tool flows in general.

16.4 Compiler Optimization

Compiler optimization is a fascinating topic in classical compilers. For quantum compilers, it becomes even more interesting, given the exponential complexity and novelty of transformations. Compiler optimizations play an important role in several areas:

[9] See also http://codeforces.com/blog/entry/60209.

- **Ancilla management**. As we use higher-level abstractions and programming languages, ancilla qubits should be managed automatically in a manner similar to register allocation for classical compilers. The compiler can trade off circuit depth against the number of ancilla bits, supporting the goal of squeezing a circuit into limited resources. Minimizing ancillae in the general case appears to be an open problem.
- **Noise reduction**. The application of quantum gates is subject to noise. Some gates introduce more noise than others. Hence, the role of the optimizer is to minimize gates as a whole and emit gate sequences to actively contain noise.
- **Gate mapping to physical machines**. Current quantum computers only support a small number of different gates. The compiler must decompose logical gates and map them to available physical gates. Furthermore, at least in the short term, the number of available qubits is extremely limited. One of the compiler's main roles is mapping circuits onto those limited resources.
- **Logical to physical register mapping**. Quantum computers have topological constraints on how qubits can interact with each other. For example, only next-neighbor interactions may be possible in some cases. Multi-qubit gates spanning non-neighboring qubits thus must be decomposed into two-qubit gates between neighboring qubits.
- **Accuracy tuning**. Individual gates may not be accurate enough for a given algorithm; multiple gates may be necessary to achieve the desired result. The compiler plays a central role in determining the required accuracy and the corresponding generation of approximating circuits.
- **Error correction**. The automatic insertion of minimal error-correcting circuitry is an important task for the compiler.
- **Tooling**. The compiler sees the whole circuit and can apply whole program analyses, such as the entanglement analysis we saw in Section 16.3.3.
- **Performance**. Optimization should also target circuit depth and complexity. Given the short coherence times of current machines, the shorter a circuit has to run, and the fewer gates it needs to execute, the higher the chances of reliable outcomes.

The space is large and complex, and we cannot cover it exhaustively. Instead, we again provide representative examples of key principles and techniques in order to give a taste of the challenges.

16.4.1 Classic(al) Compiler Optimizations

In our infrastructure and many of the other platforms we described in Section 16.3, classical code is freely intermixed with quantum code. This means that classical optimizations, such as loop unrolling, function inlining, redundancy elimination, constant propagation, and many other scalar, loop, and inter-procedural optimizations still apply. This is necessary because all classical constructs must be eliminated before sending a circuit to the quantum accelerator. Besides, classical techniques such as dead code elimination and constant folding equally apply to quantum circuitry.

Scaffold is a great example of the mix of the classical and quantum worlds and the impact of classical optimizations on the performance of a quantum circuit (Javadi-Abhari et al., 2014). Scaffold represents quantum operations in the intermediate representation (IR) of a classical compiler and directly benefits from the rich library of available optimization passes in LLVM (Lattner and Adve, 2004).

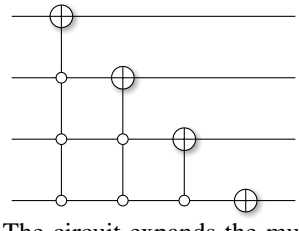
Other known classical techniques also apply. Analysis of communication overhead and routing strategies developed for distributed systems work with modifications for quantum computing (Ding et al., 2018). Register allocation can lead to optimal allocation and reuse of quantum registers (Ding et al., 2020).

16.4.2 Simple Gate Transformations

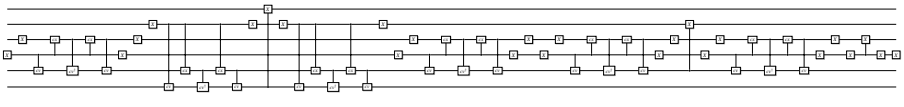
The most basic optimization is to eliminate gates that will have no effect. For example, two X gates in a row, or two other involutory matrices in sequence acting on the same qubit, or two rotations adding up to 0; all of these can be eliminated:

$$Z_i \underbrace{X_i X_i}_{\text{redundant}} Y_i = Z_i Y_i.$$

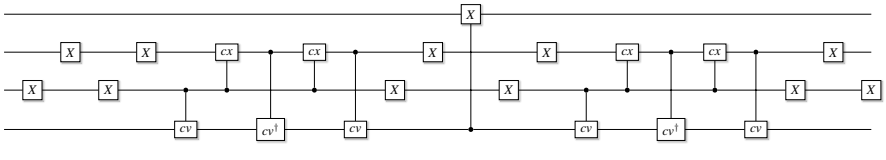
Sequences like this can be found as the result of higher-level transformations that link independent circuit fragments. For example, take the four-qubit decrement circuit, which we detailed in Section 12.1 on quantum random walks:



The circuit expands the multi-controlled gates into this much longer sequence of gates (don't worry, you are not expected to be able to decipher this):



Zooming in at the right, you can see the opportunity to eliminate redundant X gates:

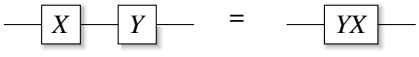


In general, for a single-qubit operator U, if the compiler can prove that the input state is an eigenstate of U with an eigenvalue of 1 (which means $U|\psi\rangle = |\psi\rangle$), it can simply remove the gate. For example, if the qubit is in the $|+\rangle$ state, the X gate has no effect, as $X|+\rangle = |+\rangle$.

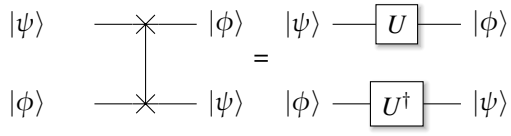
Depending on the numerical conditioning of an algorithm, the compiler may also decide to remove gates with only minor effects. As an example, we have seen the effectiveness of this technique in the approximate QFT (Coppersmith, 2002).

16.4.3 Gate Fusion

For simulation and perhaps for physical machines with a suitable gate set, we can *fuse* consecutive gates by means of simple matrix multiplication. Some high-performance simulators use this technique. Fusion can happen at several levels and across a varying number of qubits. The resulting gates may not be available on a physical machine, in which case the compiler will have to approximate the fused gates. This can nullify the benefits of fusion, but in cases where two gates X and Y both have to be approximated, it may be beneficial to approximate the combined gate YX:



The compiler can also exploit the fact that qubits may be unentangled. For example, assume that qubits $|\psi\rangle$ and $|\phi\rangle$ are known to be unentangled and must be swapped, potentially by a Swap gate spanning multiple qubits. Since the gates are unentangled and in a pure state, we may be able to classically find a unitary operator U such that $U|\psi\rangle = |\phi\rangle$ and $U^\dagger|\phi\rangle = |\psi\rangle$. The operator U is specific to $|\psi\rangle$ and $|\phi\rangle$ and input dependent. In circuit notation:

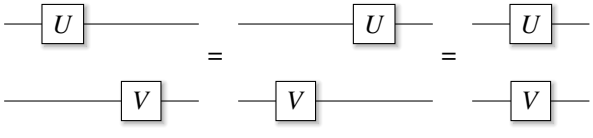


16.4.4 Gate Scheduling

We have described many gate equivalences, and many more are available in the literature. The specific gate sequence to use will depend on topological constraints, on what a specific quantum computer can support, and also on the relative cost of specific gates. For example, T gates might be an order of magnitude slower than other gates and may have to be avoided.

In order to find the best equivalences, pattern matching can be used. To maximize the number of possible matches, you may have to reorder and reschedule gates. Therefore, valid and efficient recipes for reordering are a rich area of research. As a simple example, single-qubit gates applied to different qubits can be reordered and parallelized as

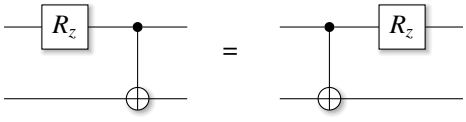
$$(U \otimes I)(I \otimes V) = (I \otimes V)(U \otimes I) = (U \otimes V)$$



There are many other opportunities to reorder. For example, if a gate is followed by a controlled gate of the same type, the two gates can be re-ordered:

$$Y_i CY_{ji} = CY_{ji} Y_i.$$

Rotations are a popular target for reordering. For example, the S gate, T gate, and phase gate represent rotations, which can be applied in any order (as long as they rotate around the same axis). Nam et al. (2018) provide many recipes, rules for rewriting, and examples, such as this one:



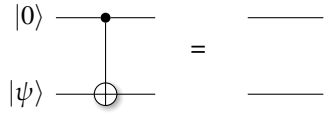
In simulation, it may not help to parallelize gates, at least in our implementation. However, on a physical quantum computer, it is safe to assume that multiple gates will be able to operate in parallel. Mapping gates to parallel running qubits will improve device utilization and have the potential to reduce circuit depth. Shorter depth means shorter runtime and a higher probability of finishing execution before decoherence.

Measurements typically occur at the end of a circuit execution. Qubits have a limited lifetime, so it is a good strategy to initialize qubits as late as possible. This is achieved with a policy to schedule gates *as late as possible* (ALAP), working backward from the measurement. This is also the default policy in IBM's Qiskit compiler. Ding and Chong (2020) detail other scheduling policies and additional techniques to minimize communication costs.

16.4.5 Peephole Optimization

Peephole optimization gets its name from the fact that this type of optimization looks at only a small sliding window over code or circuitry, hoping to find exploitable patterns in this window. This is a standard technique in classical compilers but applies to quantum computing as well (McKeeman, 1965). Limited window pattern matching approaches have in common that the underlying unitary operator must not change for a given gate replacement. This guarantees the correctness of a transformation.

With *relaxed peephole optimization*, this constraint can be, well, *relaxed* (Liu et al., 2021). For example, if a controlling qubit is known to be in state $|0\rangle$, as shown above, we can eliminate the controlled gate. The circuit is still logically equivalent, but the underlying operator has changed. We can exploit this insight in the following ways. A controlled U operation with a controlling $|0\rangle$ qubit has no effect and can be eliminated (the compiler has to *ascertain* that the controller will be $|0\rangle$):



We can also "squeeze" the Swap gate and remove one of the controlled gates if one of the inputs is known to be $|0\rangle$:

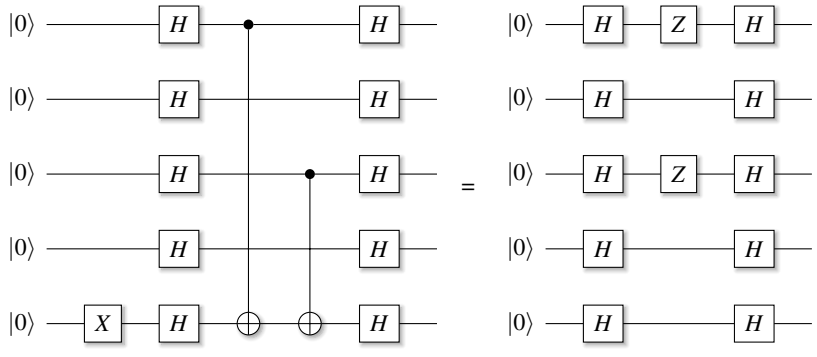
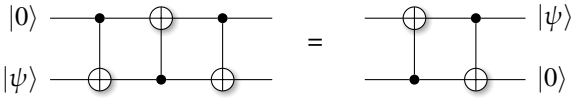


Figure 16.1 Optimized Bernstein–Vazirani circuit.



The controlled Not gates in the Bernstein–Vazirani oracle circuit can be replaced by simple Z gates because the leading Hadamard gates put the qubits in the $|+\rangle$ basis. This is shown in Figure 16.1. The techniques can be generalized to multi-controlled gates as well. More examples of this technique, along with a full evaluation, can be found in (Liu et al., 2021).

16.4.6 High-Performance Pattern Libraries

Efficient matching of patterns to gate sequences is a challenge. A possible approach is to precompute a library of high-performance subcircuits and then transpile nonoptimal and permuted subcircuits into known high-performance circuits. This approach is similar to the end-game library in a computer playing chess.[10]

16.4.7 Logical to Physical Mapping

We have already seen many gate equivalences in this text. Choosing which ones to apply will depend on the physical constraints of an underlying architecture. In this context, logical to physical qubit mapping presents an optimization challenge.

For example, Swap gates may only be applied to neighboring physical qubits. If there is a swap between logical qubits 0 and (very large) n, it might be better to place the physical qubit n right next to qubit 0. Otherwise, *communication overhead* will be very high. For example, a construction like the one in Figure 16.2 is needed to swap qubits 0 and 2 in a three-qubit circuit. The circuit presented is not very efficient; it simply stitches together a series of two-qubit Swap gates. To bridge swaps across

[10] In the olden times, *before* AlphaZero: http://en.wikipedia.org/wiki/AlphaZero.

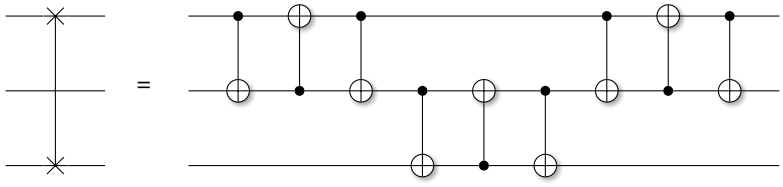


Figure 16.2 Decomposition of a Swap gate spanning three qubits into next-neighbor controlled gates.

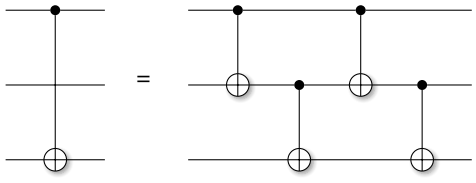


Figure 16.3 A controlled Not from qubit 0 to qubit 2 is decomposed into next-neighbor controlled Not gates.

longer distances, this ladder must be extended to more qubits, all the way down and back up again.

If the physical qubit assignment has been decided, gates may have to be further deconstructed to fit the topological constraints. In the example shown in Figure 16.3, a controlled Not from a qubit 0 to qubit 2 is being decomposed into next-neighbor controlled gates. Several other controlled Not deconstructions are presented in Garcia-Escartin and Chamorro-Posada (2011).

A related proposed technique is *wire optimization* (Paler et al., 2016). It uses a qubit lifetime analysis to *recycle* wires and qubits, the insight being that not all qubits are needed during the execution of a full circuit. Under the assumption that we can measure and reuse qubits, this work shows drastic reductions in the number of qubits required for an algorithm of up to 90%. This mirrors the results we find with our sparse implementation. However, at the time of this writing, it does not appear that intermittent measurement and re-initialization of qubits can be performed efficiently.

16.4.8 Physical Gate Decomposition

Finally, an important step for compiler and optimizer is to decompose higher-level gates into physically available gates while respecting connectivity constraints. For example, IBMQX5 has five qubits and the gates U_1, U_2, U_3, as well as a *CNOT* gate (IBM, 2021a), which can only be applied to neighboring gates:

$$U_1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix},$$

$$U_2(\phi,\lambda) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix}$$
$$= R_z(\phi+\pi/2)R_x(\pi/2)R_z(\lambda-\pi/2),$$
$$U_3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{pmatrix}.$$

Other architectures offer different gates available in different topologies. Mapping idealized gates to physical gates is challenging, especially if the physical gates have an unusual structure. A broader analysis and taxonomy can be found in Murali et al. (2019).

We also discussed earlier in Section 16.2 that in an idealized programming model, we may use any gate, knowing that gates can be approximated. In the context of software/hardware codesign, key questions may be as follows.

1. What is the *best* set of gates to realize in hardware?
2. What is the impact of this choice on the gate approximation or other design parameters and circuit depth?
3. What is the accuracy impact of gate approximations on an algorithm?
4. If approximation would require an exponentially growing set of gates, would this not nullify the complexity advantage of quantum algorithms?

Some abstract gates will be easier to approximate than others on a given physical instruction set, such as the IBM machine above. Each target and algorithm will hence require targeted methodology and compilation techniques.

16.4.9 Open-Source Simulators

We discussed the basic principles of how to construct an efficient but still bare-bones simulator. With the help of our transcoding facilities, we can target other available simulators, for example, to utilize simulators that support distributed computation or advanced noise models. This section provides a cross-section of the most cited and well-developed simulators. A more exhaustive list of simulators can be found in Quantiki (2021).

The qHipster full-state simulator implements threading, vectorization, and distributed computation through MPI and OpenMP (Smelyanskiy et al., 2016). It uses highly optimized libraries on Intel platforms. At the time of writing, the simulator was rebranded as the Intel Quantum Simulator (Guerreschi et al., 2020), which is available on Github (Intel, 2021). This simulator also allows the modeling of quantum noise processes, which enables the simulation of quantum hardware subject to the noise.

The only sparse implementation we are aware of is libquantum (Butscher and Weimer, 2013). We used it as the foundation for our `libq`. The library is no longer actively maintained (the last release was in 2013). Even though it offers excellent single-thread performance for circuits where the maximum number of states with

nonzero amplitudes is only a small fraction of all possible states. It also makes provisions for quantum error correction and allows modeling of decoherence effects.

QX is an open-source implementation of a high-performance simulator (Khammassi et al., 2017). It accepts as input *quantum code*, a variation of QASM that supports explicit parallelism between gates, debug print statements, and looping constructs. It performs aggressive optimizations but still appears to store the full state vector. QX also supports noisy execution using a variety of error models. It is part of a larger quantum development environment from the University of Delft.

ProjectQ is a Python-embedded compiler-supported framework for quantum computing (Steiger et al., 2018). It allows targeting of both real hardware and the simulator included in the distribution. The simulator allows "shortcuts" to set the expected result of an expensive computation without simulating it. ProjectQ's distribution contains transpilers to several other available frameworks. It can call into RevKit (Soeken et al., 2012) to automatically construct reversible oracles from classical gates, a function of great utility.

QuEST, the Quantum Exact Simulation Toolkit, is a full-state, multithreaded, distributed, and GPU-accelerated simulator (Jones et al., 2019). It hybridizes MPI and OpenMP and has demonstrated impressive scaling on large supercomputers. It supports state-vector and density matrix simulation, general decoherence channels of any size, general unitaries with any number of control and target qubits, and other advanced facilities like Pauli gadgets and higher-order Trotterization. The related QuESTlink (Jones and Benjamin, 2020) system allows use of the QuEST features within the Mathematica package.

Recently, Cirq published two high-performance simulators, qsim and qsimh (Google, 2021d). The former, qsim, targets single machines, whereas qsimh allows distributed computation via OpenMP. The implementations are vectorized and perform several optimizations, such as gate fusion. The qsim simulator is a full state Schrödinger simulator. The qsimh simulator (note the character *h*) is a Schrödinger–Feynman simulator (Markov et al., 2018), which trades performance for reduced memory requirements.

Microsoft's quantum development kit offers several simulators, including a full-state simulator, several resource estimators, and an accelerated simulator for Clifford gates, which can handle millions of gates (Microsoft QDK Simulators, 2021).

The Qiskit ecosystem offers a range of simulators, including full-state simulators, resource estimation tools, noisy simulations, and QASM simulators (Qiskit, 2021).

Appendix: Sparse Implementation

This appendix details the implementation of `libq`, including some optimization successes and failures. The full source code can be found online in the directory `src/libq` of the open-source repository. It is about 500 lines of C++ code. Correspondingly, this section is very code-heavy.

A.1 Register File and Program State

The entire program state, including the basis states and their amplitudes, is maintained in the structure type `qureg_t` defined in file `libq.h`. The important parts of this `struct` are:

Find the code
In file `src/libq/libq.h`

```
typedef uint64 state_t;
struct qureg_t {
  cmplx* amplitude;
  state_t* state;
  int width; /* number of qubits in the qureg */
  int size;  /* number of nonzero vectors */
  int hashw; /* width of the hash array */
  int* hash;
  bool bit_is_set(int index, int target) __attribute__ ((pure)) {
    return state[index] & (static_cast<state_t>(1) << target);
  }
  void bit_xor(int index, int target) {
    state[index] ^= (static_cast<state_t>(1) << target);
  }
};
typedef struct qureg_t qureg;
qureg *new_qureg(state_t initval, int width);
void delete_qureg(qureg *reg);
void print_qureg(qureg *reg);
void print_qureg_stats(qureg *reg);
void flush(qureg* reg);
```

Again, we use similar names in `libq` as found in libquantum to enable line-by-line comparisons. As described in Section 3.9, individual basis states are stored

as an array of bit masks in `state`, paired with their complex amplitude in array `amplitude`.

Here is an interesting tidbit: In an earlier version of this library included in the SPEC 2006 benchmarks, those two arrays were written as an array of C++ structs, where each individual `struct` element had a single amplitude and state. This was not good for performance, as iterations over the array during state modification had to iterate over more memory than necessary, as the state bit masks were interleaved with the amplitudes.[1]

The member `width`, which probably deserves a better name, represents the number of qubits available in the program state. The member `size` has the number of nonzero probabilities, and `hash` is a pointer to the hash table, with `hashw` being the size of the hash table.

Operations to check whether a bit is set and to XOR a specific bit with a value are very common and done with the two inline member functions `bit_is_set` and `bit_xor`. In the header file, there are a handful of functions to manipulate the program state as follows.

The function `new_qureg` creates a new program state with a quantum register of a given size `width` and initializes an initial single state with a given bit mask `initval` with probability 1 (at least one state must be defined). The function's main job is to `calloc()` the various arrays and make sure that there are no out-of-memory errors.

To free all allocated data structures and set relevant pointers to `nullptr`, we use the function `delete_qureg(qureg *reg)`. To print a textual representation of the current state by listing all states with nonzero probability, we use the function `print_qureg(qureg *reg)`. Function `print_qureg_stats(qureg *reg)` can be used to display statistics such as how many qubits were stored, how often the hash table was recomputed, and the maximum number of nonzero probability states reached during the execution of an algorithm.

For certain experiments, parts of the internal state are cached. The function `flush(qureg* reg)` ensures that all remaining states are flushed. This could mean that a computation is completed or that some pending prints are flushed to `stdout`.

A.2 Superposition-Preserving Gates

These are gates that neither create nor destroy superposition. They represent the "easy" case in this sparse representation. Let us look at some representative gates. To apply the *X* gate to a specific qubit, the bit corresponding to the qubit index must be flipped. Recall that the gate's function is determined by

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}.$$

[1] This author implemented a rather involved automatic data layout transformation in the HP compilers for Itanium that would transform the array of `structs` into a `struct` of arrays (Hundt et al., 2006). A later version of the library then modified the source code itself, a two-line change that completely obliterated the need and benefit of the complex compiler transformation.

Let us assume that we have a basis state encoded as the binary `0b000101` and apply the X gate to qubit 2 in bit order (from the right). To achieve this, we XOR bit 2 with a 1. If the bit was 0, it would become 1, and if it was already 1, it would flip to 0. In the example, the bit mask changes to `0b000001`.

If there are n states with nonzero amplitudes in the system, there are n pairs of states and amplitudes. To flip one qubit's amplitude according to the X gate, we have to flip the bit corresponding to that qubit in each of those tuples since that represents the operation of this gate on all the states. Mechanically, the probability amplitudes for that qubit are flipped by just flipping the bit in the state bit masks. There is no other data movement, and the code is remarkably simple:

Find the code
In file src/libq/gates.cc

```
  void x(int target, qureg *reg) {
    for (int i = 0; i < reg->size; ++i)
      reg->bit_xor(i, target);
}
```

For another class of operators, we must check whether a bit is set before applying a transformation. For example, applying the Z gate to a state acts like this:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ -\beta \end{pmatrix}.$$

The gate only has an effect if β is nonzero. In the sparse representation, this means that there must be a tuple representing a nonzero probability that has a 1 at the intended qubit location. We iterate over all state tuples, check for the condition, and only negate the amplitude if that bit was set:

```
void z(int target, qureg *reg) {
  for (int i = 0; i < reg->size; ++i)
    if (reg->bit_is_set(i, target))
      reg->amplitude[i] *= -1;
}
```

Recall that if the qubit is in superposition, there will be two tuples: one with the corresponding bit set to 0 and the amplitude set to α, and the other with the bit set to 1 and the amplitude set to β. For the Z gate, we only need to change the second tuple. This is similar for the T gate and other phase gates:

```
void t(int target, qureg *reg) {
  static cmplx z = cexp(M_PI / 4.0);
  for (int i = 0; i < reg->size; ++i)
    if (reg->bit_is_set(i, target))
        reg->amplitude[i] *= z;
}
```

The Y gate is moderately more complex and combines the methods shown above. The operation of the gate is:

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} -i\beta \\ i\alpha \end{pmatrix}.$$

In code, we first flip the bit with the X gate and then multiply by i or $-i$, depending on whether the bit is set after it was flipped:

```
void y(int target, qureg *reg) {
  for (int i = 0; i < reg->size; ++i) {
    reg->bit_xor(i, target);
    if (reg->bit_is_set(i, target))
      reg->amplitude[i] *= cmplx(0, 1.0);
    else
      reg->amplitude[i] *= cmplx(0, -1.0);
  }
}
```

A.3 Controlled Gates

Controlled gates are a logical extension of the above. In order to control a gate, we have to check whether the corresponding control bit is set to 1. For example, for the controlled X gate:

```
void cx(int control, int target, qureg *reg) {
  for (int i = 0; i < reg->size; ++i)
    if (reg->bit_is_set(i, control))
      reg->bit_xor(i, target);
}
```

Similarly, for the controlled Z gate:

```
void cz(int control, int target, qureg *reg) {
  for (int i = 0; i < reg->size; ++i)
    if (reg->bit_is_set(i, control))
      if (reg->bit_is_set(i, target))
        reg->amplitude[i] *= -1;
}
```

This even works for double-controlled gates, where we only have to check that both control bits are set. Here is the implementation of a double-controlled X gate:

```
void ccx(int control0, int control1, int target, qureg *reg) {
  for (int i = 0; i < reg->size; ++i)
    if (reg->bit_is_set(i, control0))
```

```
      if (reg->bit_is_set(i, control1))
        reg->bit_xor(i, target);
}
```

A.4 Superpositioning Gates

The difficult case is for gates that create or destroy superposition. We provide an implementation in function `libq_gate1`, which we detail in Section A.6. The function expects the 2×2 gate as a parameter. For example, for the Hadamard gate:

Find the code
In file src/libq/apply.cc

```
void h(int target, qureg *reg) {
  static cmplx mh[4] = {sqrt(1.0/2), sqrt(1.0/2), sqrt(1.0/2),
                        -sqrt(1.0/2)};
  libq_gate1(target, mh, reg);
}
```

The implementation applies the same technique we saw earlier in Section 3.6 on accelerated gates: a linear traversal over the states, except that it is adapted to the sparse representation. Additionally, it manages memory by filtering out close-to-zero states.

A.5 Hash Table

First, as indicated above, the states are maintained in a hash table with this hash function:

```
static inline unsigned int hash64(state_t key, int width) {
  unsigned int k32 = (key & 0xFFFFFFFF) ^ (key >> 32);
  k32 *= 0x9e370001UL;
  k32 = k32 >> (32 - width);
  return k32;
}
```

The hash lookup function `get_state` checks whether a given state exists with nonzero amplitude. It calculates the hash index for state `a` and iterates over the dense array, hoping to find that state. If a 0 state was found (the marker for an unpopulated entry) or if the search wraps around, no state was found and -1 is returned. Otherwise, the position in the hash table is returned:

```
state_t get_state(state_t a, qureg *reg) {
  unsigned int i = hash64(a, reg->hashw);
  while (reg->hash[i]) {
    if (reg->state[reg->hash[i] - 1] == a)
      return reg->hash[i] - 1;
    i++;
    if (i == (1 << reg->hashw))
      break;
  }
  return -1;
}
```

Of course, there is a function to add a state to the hash table:

```
void libq_add_hash(state_t a, int pos, qureg *reg) {
  int mark = 0;

  int i = hash64(a, reg->hashw);
  while (reg->hash[i]) {
    i++;
    if (i == (1 << reg->hashw)) {
      if (!mark) {
        i = 0;
        mark = 1;
      }
    }
  }
  reg->hash[i] = pos + 1;
  // -- Optimization will happen here (later).
}
```

The most interesting function from a performance perspective is the one that reconstructs the hash table. Since the function to apply a gate will filter out states with probabilities close to 0 after gate application, we have to reconstruct the hash table to ensure it contains only valid entries. This is the most expensive operation of the entire `libq` implementation. We show some optimizations below, where the first loop is being replaced with a `memset()`, and more tricks in Section A.8.

```
void libq_reconstruct_hash(qureg *reg) {
  reg->hash_computes += 1;   // count invocations.

  for (int i = 0; i < (1 << reg->hashw); ++i)
    reg->hash[i] = 0;
  for (int i = 0; i < reg->size; ++i)
    libq_add_hash(reg->state[i], i, reg);
}
```

The first thing to note is the first loop, which resets the hash array to all zeros:

```
  for (int i = 0; i < (1 << reg->hashw); ++i)
    reg->hash[i] = 0;
```

You might expect the compiler to transform this loop into a vectorized `memset` operation. However, it does not. The loop-bound `reg->hashw` aliases with the loop body, which means that the compiler cannot infer whether the loop body would modify the loop bound. Manually changing this to `memset` speeds up the entire simulation by approximately 20%.

```
    memset(reg->hash, 0, (1 << reg->hashw) * sizeof(int));
```

This `memset` is still the slowest part of the implementation. We will show how to optimize it further below.

A.6 Gate Application

Now we describe the routine for applying a gate. It starts by assuming that something might have changed since the last invocation, so its first task is to reconstruct the hash table:

```
void libq_gate1(int target, cmplx m[4], qureg *reg) {
  int addsize = 0;
  libq_reconstruct_hash(reg);
  [...]
}
```

The superposition of a given qubit means that states with both a 0 and a 1 at a given bit position must exist. So, the function iterates and counts how many of those states are missing and need to be added:

```
/* calculate the number of basis states to be added */
for (int i = 0; i < reg->size; ++i) {
  /* determine whether XOR'ed basis state already exists */
  if (get_state(reg->state[i] ^
      (static_cast<state_t>(1) << target), reg) == -1)
       addsize++;
 }
```

If new states need to be added, the function reallocates the arrays. It also does some bookkeeping and remembers the largest number of states with a nonzero probability:

```
/* allocate memory for the new basis states */
if (addsize) {
  reg->state = static_cast<state_t *>(
    realloc(reg->state, (reg->size + addsize) * sizeof(state_t)));
  reg->amplitude = static_cast<cmplx *>(
    realloc(reg->amplitude, (reg->size + addsize) * sizeof(cmplx)));

  memset(&reg->state[reg->size], 0, addsize * sizeof(int));
  memset(&reg->amplitude[reg->size], 0, addsize * sizeof(cmplx));
  if (reg->size + addsize > reg->maxsize)
    reg->maxsize = reg->size + addsize;
}
```

This is all for state and memory management. Now we move on to applying the gates. We allocate an array `done` to remember which states we have already handled. The variable `limit` will be used at the end of the function to remove states with a probability close to zero.

```
    char *done =
       static_cast<char *>(calloc(reg->size + addsize, sizeof(char)));
    int next_state = reg->size;
    float limit = (1.0 / (static_cast<state_t>(1) << reg->width))
                  * 1e-6;
```

We then iterate over all states and check if a state has not yet been handled. We check whether a target bit has been set and obtain the index of the other base state in the variable `xor_index`. The amplitudes for the basis states $|0\rangle$ and $|1\rangle$ are stored in `tnot` and `t`.

```
/* perform the actual matrix multiplication */
for (int i = 0; i < reg->size; ++i) {
  if (!done[i]) {
    /* determine if the target of the basis state is set */
    int is_set = reg->state[i] & (static_cast<state_t>(1) << target);
    int xor_index =
         get_state(reg->state[i] ^
         (static_cast<state_t>(1) << target), reg);
    cmplx tnot = xor_index >= 0 ? reg->amplitude[xor_index] : 0;
    cmplx t = reg->amplitude[i];
  }
  [...]
}
```

The matrix multiplication follows the patterns we have seen for the fast gate application in Section 3.6. If states are found, we apply the gate. If the XOR'ed state was not found, this means that we have to add a new state and perform the multiplication:

```
    if (is_set) {
      reg->amplitude[i] = m[2] * tnot + m[3] * t;
    } else {
      reg->amplitude[i] = m[0] * t + m[1] * tnot;
    }

    if (xor_index >= 0) {
      if (is_set) {
        reg->amplitude[xor_index] = m[0] * tnot + m[1] * t;
      } else {
        reg->amplitude[xor_index] = m[2] * t + m[3] * tnot;
      }
    } else { /* new basis state will be created */
      if (abs(m[1]) == 0.0 && is_set) break;
      if (abs(m[2]) == 0.0 && !is_set) break;

      reg->state[next_state] =
          reg->state[i] ^ (static_cast<state_t>(1) << target);
      reg->amplitude[next_state] = is_set ? m[1] * t : m[2] * t;
      next_state += 1;
    }
    if (xor_index >= 0)
      done[xor_index] = 1;
```

As a final step, we filter out the states with an amplitude close to 0. The code below densifies the array by moving up all nonzero elements before finally reallocating the amplitude and state arrays to a smaller size (which is a redundant operation):

```
reg->size += addsize;
free(done);

/* remove basis states with extremely small amplitude */
if (reg->hashw) {
  int decsize = 0;
  for (int i = 0, j = 0; i < reg->size; ++i) {
    if (probability(reg->amplitude[i]) < limit) {
      j++;
      decsize++;
    } else if (j) {
      reg->state[i - j] = reg->state[i];
      reg->amplitude[i - j] = reg->amplitude[i];
    }
  }
  if (decsize) {
    reg->size -= decsize;
  }
}
```

A.7 Premature Optimization, Second Act

Here is an anecdote that might serve as a lesson to over-eager code optimizers.[2] After implementing the code and running initial benchmarks, it appeared obvious that repeated iterations over the memory just had to be a bottleneck. Some form of mini-JIT (Just-In-Time compilation) should be helpful, which first collects all the operations and then fuses gate applications into the same loop iteration. The goal would be to significantly reduce repeated iterations over the states to avoid memory traffic, which was assumed to be the problem. The code is available online. It might become valuable in the future,[3] as other performance bottlenecks are being resolved.

The goal of the main routine was to execute something like the following, with just one outer loop and a switch statement over all superposition-preserving gates:

```
[...]
 void Execute(qureg *reg) {
   for (int i = 0; i < reg->size; ++i) {
     for (auto op : op_list_) {
       switch (op.op()) {
         case op_t::X:
           reg->bit_xor(i, op.target());
           break;

         case op_t::Y:
           reg->bit_xor(i, op.target());
           if (reg->bit_is_set(i, op.target()))
             reg->amplitude[i] *= cmplx(0, 1.0);
           else
             reg->amplitude[i] *= cmplx(0, -1.0);
           break;

         case op_t::Z:
           if (reg->bit_is_set(i, op.target())) {
             reg->amplitude[i] *= -1;
           }
           Break;
[...] }}}}
```

As a complete surprise, running the JIT'ed version produced a performance improvement of roughly 0%. Simple profiling then revealed that about 96% of the execution time was spent on reconstructing the hash table. Gate application was not a performance bottleneck at all. Lesson learned again – intuition is good, but verification is better.

A.8 Actual Performance Optimization

As noted above, reconstructing the hash table is the most expensive operation in this library. The hash table is sized to hold all potential states, given the number of qubits.

[2] Such as myself.
[3] Or serve as a warning to future readers.

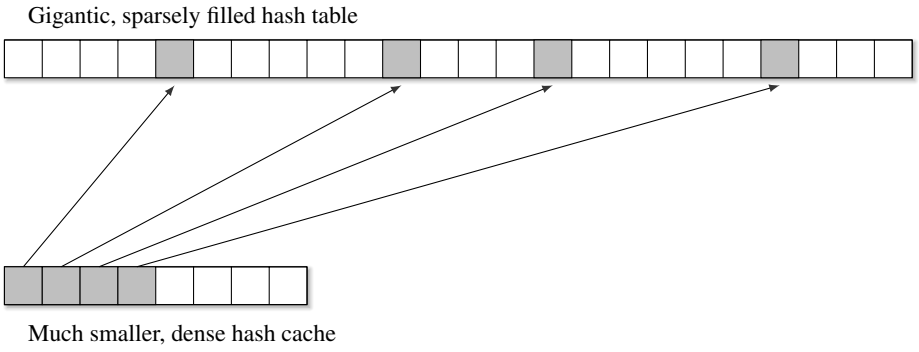


Figure A.1 A caching scheme to accelerate hash table zeroing.

However, even for complex algorithms, the actual maximal number of states with nonzero probability can be relatively small. For example, for two benchmarks that we extract from quantum arithmetic (`Arith`) and order finding (`Order`), we show the maximum number of nonzero states reached (8,192) and, given the number of qubits involved, the theoretical maximal number of states. The percentage is 3.125% for `Order`, and only 0.012% for `Arith`. It has a lot more qubits and hence a very large potential number of states:

```
Arith:  Maximum of states: 8192, theoretical: 67108864, 0.012%
Order:  Maximum of states: 8192, theoretical: 262144, 3.125%
```

During execution, the number of states changes dynamically in powers of two as `libq` removes states very close to 0. Therefore, there is an opportunity to augment the hash table and track, or cache, the addresses of elements that have been set, up to a given threshold, for example, up to 64K elements.

To reset the hash table, we iterate over the entries in *hash cache* and zero out the marked elements in the hash table, as shown in Figure A.1. There will be a crossover point. For some size of the hash cache, just linearly sweeping the hash table will be faster than the random memory access patterns from the cache because of hardware prefetching dynamics. We chose 64K as the cache size, which significantly improves the runtime for our examples. This is an interesting space to experiment with to find better heuristics and data structures.

In function `libq_reconstruct_hash`, we additionally maintain an array called `hash_hits`, which holds the addresses of states in the main hash table, along with a counter `reg->hits` of those. Then, we selectively zero out only those memory addresses in the hash table that we cached. If the hash cache was not big enough, we have to resort to zeroing out the full hash table:

```
void libq_reconstruct_hash(qureg *reg) {
 reg->hash_computes += 1;

 if (reg->hash_caching && reg->hits < HASH_CACHE_SIZE) {
   for (int i = 0; i < reg->hits; ++i) {
```

```
      reg->hash[reg->hash_hits[i]] = 0;
      reg->hash_hits[i] = 0;
    }
    reg->hits = 0;
  } else {
    memset(reg->hash, 0, (1 << reg->hashw) * sizeof(int));
    memset(reg->hash_hits, 0, reg->hits * sizeof(int));
    reg->hits = 0;
  }
  for (int i = 0; i < reg->size; ++i)
    libq_add_hash(reg->state[i], i, reg);
}
```

All that's left to do now is to fill in this array `hash_hits` whenever we add a new element in `libq_add_hash` using the following code at the very bottom:

```
[...]
reg->hash[i] = pos + 1;
 if (reg->hash_caching && reg->hits < HASH_CACHE_SIZE) {
   reg->hash_hits[reg->hits] = i;
   reg->hits += 1;
 }
```

Performance gains from this optimization can be substantial, depending on the characteristics of the algorithm. Anecdotal evidence points to improvements in the range of 20–30% for `Arith` and `Order`, as long as the nonzero states fit in the hash cache.

References

Aaronson, S. Read the fine print. *Nature Physics*, 11, 291–293, 2015. http://doi.org/10.1038/nphys3272.

Aaronson, S. and D. Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5), 2004. doi: 10.1103/physreva.70.052328.

Abhijith, J., A. Adetokunbo, J. Ambrosiano, et al. Quantum algorithm implementations for beginners, 2020. arXiv:1804.03719v2 [cs.ET].

Acín, A., A. Andrianov, L. Costa, E. Jané, J. I. Latorre, and R. Tarrach Generalized Schmidt Decomposition and Classification of Three-Quantum-Bit States. *Physical Review Letters*, 85(7) 1560–1563, 2000. doi: 10.1103/PhysRevLett.85.1560.

Altenkirch, T. and A. Green. The quantum IO monad. *Semantic Techniques in Quantum Computation*, 2013. doi: 10.1017/CBO9781139193313.006.

Altman, E., K. R. Brown, G. Carleo, et al. Quantum simulators: Architectures and opportunities. *PRX Quantum*, 2(1), 2021. doi: 10.1103/prxquantum.2.017003.

Anders, S. and H. J. Briegel. Fast simulation of stabilizer circuits using a graph-state representation. *Physical Review A*, 73(2), 2006. doi: 10.1103/physreva.73.022334.

Anferov, A., S. P. Harvey, F. Wan, J. Simon, and D. Schuster. Superconducting qubits above 20 GHz operating over 200 mK. arxiv.org/abs/2402.03031.

Applegate, D. L., R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2006.

Arute, F., K. Arya, R. Babbush, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019. doi: 10.1038/s41586-019-1666-5.

Arute, F., K. Arya, R, Babbush, et al. Supplementary information: Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 2019. http://arxiv.org/pdf/1910.11333.

Barenco, A., C. H. Bennett, R. Cleve, et al. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995. doi: 10.1103/physreva.52.3457.

Basheer, A., A. Afham, and S. K. Goyal. Quantum k-nearest neighbors algorithm. *Quantum Information Processing*, Springer, 21(18), 2022. arxiv.org/abs/2003.09187.

Beauregard, S. Circuit for Shor's algorithm using 2n+3 qubits. *Quantum Information and Compututation*, 3(2):175–185, 2003.

Bell, J. S. On the Einstein Podolsky Rosen paradox. *Physics Physique Fizika*, 1:195–200, 1964. doi: 10.1103/PhysicsPhysiqueFizika.1.195.

Bennett, C. H. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973. doi: 10.1147/rd.176.0525.

Bennett, C. H., G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein–Podolsky–Rosen channels. *Physical Review Letters*, 70:1895–1899, 1993. doi: 10.1103/PhysRevLett.70.1895.

Bennett, C. H. and S. Wiesner Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Physical Review Letters*, 69(20): 2881–2884, 1992. doi: :10.1103/PhysRevLett.69.2881.

Bernstein, E. and U. Vazirani Quantum complexity theory. *SIAM Journal on Computing*, 26(5): 1411–1473, 1993. doi: 10.1137/S0097539796300921.

Berry, D. W. and B. C. Sanders. Quantum teleportation and entanglement swapping for systems of arbitrary spin. Published under licence by IOP Publishing Ltd. *New Journal of Physics*, 4, 2002. doi: 10.1088/1367-2630/4/1/308.

Bérut, A., A. Petrosyan, and S. Ciliberto. Information and thermodynamics: Experimental verification of Landauer's Erasure principle. *Journal of Statistical Mechanics: Theory and Experiment*, 6:1742–5468, 2015. doi: 10.1088/1742-5468/2015/06/p06015.

Bettelli, S., T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal D – Atomic, Molecular and Optical Physics*, 25(2):181–200, 2003. doi: 10.1140/epjd/e2003-00242-2.

Bichsel, B., M. Baader, T. Gehr, and M. T. Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In A. F. Donaldson and E. Torlak, eds., *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15–20, 2020*, pp. 286–300. ACM, 2020. doi: 10.1145/3385412.3386007.

Bochkin, G. A., S. I. Doronin, E. B. Feldman, and A. I. Zenchuk. Calculation of π on the IBM quantum computer and the accuracy of one-qubit operations. *Quantum Information Processing* 19(8):257, 2020. doi: 10.1007/s11128-020-02759-6.

Boixo, S., S. V. Isakov, V. N. Smelyanskiy, et al. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595–600, 2018. doi: 10.1038/s41567-018-0124-x.

Brassard, G., F. Dupuis, S. Gambs, and A. Tapp. An optimal quantum algorithm to approximate the mean and its application for approximating the median of a set of points over an arbitrary distance. *arXiv*, 2011. doi: http://doi.org/10.48550/arXiv.1106.4267.

Brassard, G., P. Høyer, M. Mosca, and A. Tapp. Quantum amplitude amplification and estimation. *Quantum Computation and Information*, pp. 53–74, 2002. doi: 10.1090/conm/305/05215.

Briggs, P. Register via Graph Coloring. PhD thesis, 1992. www.cs.utexas.edu/~mckinley/380C/lecs/briggs-thesis-1992.pdf.

Buck, I., T. Foley, D. Horn, et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23:777–786, 2004. doi: 10.1145/1186562.1015800.

Buhrman, H., R. Cleve, J. Watrous, and R. de Wolf. Quantum fingerprinting. *Physical Review Letters*, 87(16):167902–167905, 2001. doi: 10.1103/physrevlett.87.167902.

Buhrman, H., C. Dürr, M. Heiligman, et al. Quantum algorithms for element distinctness. *SIAM Journal on Computing*, 34(6):1324–1330, 2005. doi: 10.1137/s0097539702402780.

Burke, P. G. and I. C. Percival John Stewart Bell. 28 July 1928–1 October 1990: Elected F.R.S. 1972, *Biographical Memoirs of Fellows of the Royal Society*: Vol. 45, pp. 1–17, Nov. 1999. doi: http://doi.org/10.1098/rsbm.1999.0001.

Butscher, B. and H. Weimer. libquantum. www.libquantum.de/, 2013. Accessed: February 10, 2021.

Carlson, C., Z. Jorquera, A. Kolla, and S. Kordonowy. A quantum advantage over classical for local max cut. doi: arxiv.org/abs/2304.08420.

Childs, A. M., R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman. Exponential algorithmic speedup by a quantum walk. *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing – STOC '03*, 2003. doi: 10.1145/780542.780552.

Childs, A. M., R. Cleve, S. P. Jordan, and D. Yonge-Mallo. Discrete-query quantum algorithm for nand trees. *Theory of Computing*, 5(1):119–123, 2009. doi: 10.4086/toc.2009.v005a005.

Chong, F. T., D. Franklin, and M. Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187, 2017. doi: 10.1038/nature23459.

Clauser, John F., Michael A. Horne, Abner Shimony, and Richard A. Holt. Proposed experiment to test local hidden-variable theories. *Physical Review Letters*, 23(15):880–884, 1969. doi: 10.1103/PhysRevLett.23.880.

Coook, A. The complexity of theorem-proving procedures ACM, STOC '21 (151–158):8. doi: 10.1145/800157.805047.

Coppersmith, D. An approximate Fourier transform useful in quantum factoring. *arXiv e-prints*, art. quant-ph/0201067, Jan. 2002.

Cory, D. G., M. D. Price, W. Maas, et al. Experimental quantum error correction. *Physical Review Letters*, 81(10):2152–2155, 1998. doi: 10.1103/physrevlett.81.2152.

Cross, A. W., L. S. Bishop, J. A. Smolin, and J. M. Gambetta. Open quantum assembly language, 2017. arXiv:1707.03429.

Cuccaro, S., T. Draper, S. Kutin, and D. Moulton A new quantum ripple-carry addition circuit, http://arxiv.org/abs/quant-ph/0410184.

Dawson, C. M. and M. A. Nielsen. The Solovay–Kitaev algorithm. *Quantum Information and Computation*, 6(1):81–95, 2006.

De Raedt, H., F. Jin, D. Willsch, et al. Massively parallel quantum computer simulator, eleven years later. *Computer Physics Communications*, 237:47–61, 2019. doi: 10.1016/j.cpc.2018.11.005.

Dean, W. Computational complexity theory. In E. N. Zalta, ed., *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2016.

Deutsch, D. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Series A*, 400(1818):97–117, 1985. doi: 10.1098/rspa.1985.0070.

Deutsch, D. and R. Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A*, 439(1907):553–558, 1992. doi: 10.1098/rspa.1992.0167.

Devitt, S. J., W. J. Munro, and K. Nemoto. Quantum error correction for beginners. *Reports on Progress in Physics*, 76(7):076001, 2013. doi: 10.1088/0034-4885/76/7/076001.

Ding, Y. and F. T. Chong. Quantum computer systems: Research for noisy intermediate-scale quantum computers. *Synthesis Lectures on Computer Architecture*, 15(2):1–227, 2020. doi: 10.2200/S01014ED1V01Y202005CAC051.

Ding, Y., A. Holmes, A. Javadi-Abhari, D. Franklin, M. Martonosi, and F. Chong. Magic-state functional units: Mapping and scheduling multi-level distillation circuits for fault-tolerant quantum architectures. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018. doi: 10.1109/micro.2018.00072.

Ding, Y., X.-C. Wu, A. Holmes, A. Wiseth, D. Franklin, M. Martonosi, and F. T. Chong. Square: Strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020. doi: 10.1109/isca45697.2020.00054.

Douglas, B. L. and J. B. Wang. Efficient quantum circuit implementation of quantum walks. *Physical Review A*, 79:1050–2947, 2009. doi: 10.1103/PHYSREVA.79.052335.

Draper, T. G. Addition on a quantum computer. *arXiv e-prints*, art. quant-ph/0008033, 2000.

Durr, C. and P. Hoyer. A quantum algorithm for finding the minimum *quant-ph/9607014*, 1999. arXiv:quant-ph/9607014.

Dür, W., G. Vidal, and J. I. Cirac. Three qubits can be entangled in two inequivalent ways. *Physical Review A*, 6, 2000. doi: 10.1103/physreva.62.062314.

Einstein, A., B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47:777–780, 1935. doi: 10.1103/PhysRev.47.777. http://link.aps.org/doi/10.1103/PhysRev.47.777.

Euler, L. Novi commentarii Academiae Scientiarum Imperialis Petropolitanae. *Typis Academiae Scientarum*, 8, 1763.

Farhi, E., J. Goldstone, and S. Gutmann. A quantum approximate optimization algorithm, 2014. http://arxiv.org/abs/1411.4028.

Faye, J. Copenhagen interpretation of quantum mechanics. In E. N. Zalta, ed., *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2019.

Feynman, R. *The Character of Physical Law*. MIT Press, 1965.

Feynman, R. Simulating physics with computers. *International Journal of Theoretical Physics* 21, 467–488 (1982). doi: 10.1007/BF02650179.

Feynman, R., F. Vernon, L. Frank, and R. Hellwarth Geometrical representation of the Schrödinger equation for solving maser problems. *Journal of Applied Physics*, 28, 1, *49–52*, doi: 10.1063/1.1722572.

Fleisch, D.A. *A Student's Guide to the Schroedinger Equation*. Cambridge University Press, 2020.

Frank, M. P., U. H. Meyer-Baese, I. Chiorescu, L. Oniciuc, and R. A. van Engelen. Space-efficient simulation of quantum computers. *Proceedings of the 47th Annual Southeast Regional Conference on – ACM-SE 47*, 2009. doi: 10.1145/1566445.1566554.

Fu, P., K. Kishida, N. J. Ross, and P. Selinger. A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper, 2020. http://arxiv.org/abs/2005.08396.

Gambetta, J., D. M. Rodríguez, A. Javadi-Abhari, et al. Qiskit/qiskit-terra: Qiskit Terra 0.7.2, 2019. http://doi.org/10.5281/zenodo.2656592.

Garcia-Escartin, J. C. and P. Chamorro-Posada. Equivalent quantum circuits, 2011. http://arxiv.org/abs/1110.2998.

Garey, M. and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Bell Telephone Laboratories, 1979. ISBN: 0716710455.

Garhwal, S., M. Ghorani, and A. Ahmad. Quantum programming language: A systematic review of research topic and top cited languages. *Archives of Computational Methods in Engineering*, 28(2):289–310, 2021. doi: 10.1007/s11831-019-09372-6.

Ghirardi, G. and A. Bassi. Collapse theories. In E. N. Zalta, ed., *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2020.

Gidney, C. Halving the cost of quantum addition, *Quantum*, 2:74, 2018. doi: 10.22331/q-2018-06-18-74.

Gidney, C. Asymptotically Efficient Quantum Karatsuba Multiplication, 2019. arxiv.org/abs/1904.07356.

Gidney, C. Quirk online quantum simulator. http://algassert.com/quirk, 2021a. Accessed: February 10, 2021.

Gidney, C. Breaking down the quantum swap. http://algassert.com/post/1717, 2021b. Accessed: February 10, 2021.

Gokhale, P., A. Javadi-Abhari, N. Earnest, Y. Shi, and F. T. Chong. Optimized quantum compilation for near-term algorithms with OpenPulse. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 186–200, 2020. doi: 10.1109/MICRO50266.2020.00027.

Google. Quantum supremacy using a programmable superconducting processor. http://ai.googleblog.com/2019/10/quantum-supremacy-using-programmable.html, 2019. Accessed: February 10, 2021.

Google. C++ style guide. http://google.github.io/styleguide/cppguide.html, 2021a. Accessed: February 10, 2021.

Google. Python style guide. http://google.github.io/styleguide/pyguide.html, 2021b. Accessed: February 10, 2021.

Google. Cirq. http://cirq.readthedocs.io/en/stable/, 2021c. Accessed: February 10, 2021.

Google. qsim and qsimh. http://quantumai.google/qsim, 2021d. Accessed: February 10, 2021.

Graphviz.org. Graphviz, 2021. Accessed: February 10, 2021.

Green, A. S., P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 333–342, Seattle, Washington, USA, 2013. Association for Computing Machinery. doi: 10.1145/2491956.2462177.

Greenacre, M., P. J. F. Groenen, T. Hastie, et al. Principal component analysis. *Nature Reviews Methods Primer* 2:100, 2022. doi: 10.1038/s43586-022-00184-w.

Greenberger, D. M., M. A. Horne, and A. Zeilinger. Going beyond Bell's theorem, In: Kafatos, M. (ed.) *Bell's Theorem, Quantum Theory and Conceptions of the Universe. Fundamental Theories of Physics, vol 37*. Springer, Dordrecht, 2008. doi: 10.1007/978-94-017-0849-4_10.

Grover, L. K. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pp. 212–219, New York, NY, 1996. Association for Computing Machinery. doi: 10.1145/237814.237866.

Guerreschi, G. G., J. Hogaboam, F. Baruffa, and N. P. D. Sawaya. Intel quantum simulator: A cloud-ready high-performance simulator of quantum circuits. *Quantum Science and Technology*, 5(3):034007, 2020. doi: 10.1088/2058-9565/ab8505.

Häner, T. and D. S. Steiger. 0.5 petabyte simulation of a 45-qubit quantum circuit. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2017. doi: 10.1145/3126908.3126947.

Haroche, S. and J.-M. Raimond. Quantum computing: Dream or nightmare? *Physics Today*, 49:51–52, 1996.

Harrigan M. P., K. J. Sung, M. Neeley, et al. Quantum approximate optimization of non-planar graph problems on a planar superconducting processor. *Nature Physics*, 17(3):332–336, 2021. doi: 10.1038/s41567-020-01105-y.

Harrow, A. W., A. Hassidim, and S. Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15):150502, 2009.

Harrow, A. W. and A. Montanaro. Quantum computational supremacy. *Nature*, 549(7671): 203–209, 2017. doi: 10.1038/nature23458.

Hundt, R., S. Mannarswamy, and D. Chakrabarti. Practical structure layout optimization and advice. In *International Symposium on Code Generation and Optimization, CGO 2006*, 2006. doi: 10.1109/CGO.2006.29.

IARPA. Quantum Computer Science (QCS) Program Broad Agency Announcement (BAA). http://beta.sam.gov/opp/637e87ac1274d030ce2ab69339ccf93c/view, 2010. Accessed: February 10, 2021.

IBM. IBM Q 16 Rueschlikon V1.x.x. http://github.com/Qiskit/ibmq-device-information/tree/master/backends/rueschlikon/V1, 2021a. Accessed: February 10, 2021.

IBM. Quantum Computation Center. www.ibm.com/blogs/research/2019/09/quantum-computation-center/, 2021b. Accessed: February 10, 2021.

Intel. Intel quantum simulator. http://github.com/iqusoft/intel-qs, 2021. Accessed: February 10, 2021.

Javadi-Abhari, A., S. Patil, D. Kudrow, et al. ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, New York, NY, 2014. Association for Computing Machinery. doi: 10.1145/2597917.2597939.

Jones, T. and S. Benjamin. QuESTlink—Mathematica embiggened by a hardware-optimised quantum emulator. *Quantum Science and Technology*, 5(3):034012, 2020. doi: 10.1088/2058-9565/ab8506.

Jones, T., A. Brown, I. Bush, and S. C. Benjamin. Quest and high performance simulation of quantum computers. *Scientific Reports*, 9(1):10736, 2019. doi: 10.1038/s41598-019-47174-9.

Karp, R. M. Reducibility among combinatorial problems. In: Miller, R. E., Thatcher, J. W., and Bohlinger, J. D. (eds.) *Complexity of Computer Computations. The IBM Research Symposia Series*. Springer, Boston, MA, 1972. doi: 10.1007/978-1-4684-2001-2_9

Kaye, P., R. Laflamme, and M. Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.

Kempe, J. Quantum random walks: An introductory overview. *Contemporary Physics*, 44(4):307–327, 2003. doi: 10.1080/0010715103100011077 6.

Khammassi, N., I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels. QX: A high-performance quantum computer simulation platform. In *Design, Automation Test in Europe Conference Exhibition, 2017*, pp. 464–469, 2017. doi: 10.23919/DATE.2017.7927034.

Khammassi, N., G. G. Guerreschi, I. Ashraf, et al. cQASM v1.0: Towards a common quantum assembly language. 2018. https://doi.org/10.48550/arXiv.1805.09607.

Kitaev, A. Quantum measurements and the Abelian Stabilizer Problem. doi: arXiv:quant-ph/9511026.

Kitaev, A. Y., A. H. Shen, and M. N. Vyalyi. *Classical and Quantum Computation*. American Mathematical Society, 2002.

Kleinmann, M., H. Kampermann, T. Meyer, and D. Bruß. Physical purification of quantum states. *Physical Review*, A 73, 062309, 8 June 2006.

Kliuchnikov, V., A. Bocharov, M. Roetteler, and J. Yard. A framework for approximating qubit unitaries, 2015. arXiv:1510.03888v1 [quant-ph].

Knill, E. Conventions for quantum pseudocode. Technical Report from Los Alamos National Laboratory, 1996. doi: 10.2172/366453.

Knuth, D. E. Computer science and its relation to mathematics. *The American Mathematical Monthly*, 81(4):323–343, 1974. doi: 10.1080/00029890.1974.11993556.

Landauer, D. Wikipedia: Landauer's principle, 1973. http://en.wikipedia.org/wiki/Landauer's%27s%95principle. Accessed: January 9, 2021.

Lattner, C. and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, p. 75, 2004. IEEE Computer Society.

Lawler, E. L. A note on the complexity of the chromatic number problem. *Information Processing Letters*, 5 (3):66–67, 1976, doi: 10.1016/0020-0190(76)90065-X.

Leao, T. Shor's algorithm in Qiskit. http://github.com/ttlion/ShorAlgQiskit, 2021. Accessed: February 10, 2021.

Liu, J, L. Bello, and H. Zhou. Relaxed peephole optimization: A novel compiler optimization for quantum circuits. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, 301–314, doi: 10.1109/CGO51591.2021.9370310.

Lloyd, S., M. Mohseni, and P. Rebentrost. Quantum principal component analysis. *Nature Physics*, 10, 631–633, 2014.

Lucas, A. Ising formulations of many NP problems. *Frontiers in Physics*, 2, 2014. doi: 10.3389/fphy.2014.00005.

Magniez, F., M. Santha, and M. Szegedy. Quantum algorithms for the triangle problem. In *Proceedings of SODA'05*, pp. 1109–1117, 2005.

Maldonado, T. J., J. Flick, J. Krastanov, et al. Error rate reduction of single-qubit gates via noise-aware decomposition into native gates. *Scientific Republic* 12, 6379 (2022). doi: 10.1038/s41598-022-10339-0.

Markov, I. L., A. Fatima, S. V. Isakov, and S. Boixo. Quantum supremacy is both closer and farther than it appears, 2018. arXiv:1807.10749v3 [quant-ph].

McKeeman, W. M. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965. doi: 10.1145/364995.365000.

Mermin, N. D. What's wrong with this pillow? *Physics Today*, 42(4):9, 1989. doi: 10.1063/1.2810963.

Mermin, M. D. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007. doi: 10.1017/CBO9780511813870.

Microsoft Q#. http://docs.microsoft.com/en-us/quantum/, 2021. Accessed: February 10, 2021.

Microsoft QDK Simulators. http://docs.microsoft.com/en-us/azure/quantum/user-guide/machines/, 2021. Accessed: February 10, 2021.

Mosca, M. Quantum algorithms, 2008. arXiv:0808.0369v1 [quant-ph].

Möttönen, M., J. J. Vartiainen, V. Bergholm, and M. Salomaa. Transformation of quantum states using uniformly controlled rotations, 2004. arXiv:0407010 [quant-ph], arxiv.org/abs/quant-ph/0407010.

Möttönen, M., J. J. Vartiainen, V. Bergholm, and M. Salomaa. Quantum Circuits for General Multiqubit Gates. *Physical Review Letters*, 93:13, 2004 doi: 10.1103/physrevlett.93.130502.

Murali, P., N. M. Linke, M. Martonosi, et al. Full-stack, real-system quantum computer studies: Architectural comparisons and design insights, Association for Computing Machinery, New York, NY, USA 2019. doi: 10.1145/3307650.3322273.

Musk, D. R. A comparison of quantum and traditional Fourier transform computations, *Computing in Science & Engineering*, 22(6):103–110, 1 Nov.–Dec. 2020, doi: 10.1109/MCSE.2020.3023979.

Nam, Y., N. J. Ross, Y. Su, A. M. Childs, and D. Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1), 2018. doi: 10.1038/s41534-018-0072-4.

Nickolls, J., I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008. doi: 10.1145/1365490.1365500.

Nielsen, M. A. and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 10th ed. 2011.

Norsen, T. *Foundations of Quantum Mechanics*. Springer International Publishing, 2017.

Oak Ridge National Laboratory. Summit Supercomputer. www.olcf.ornl.gov/summit/, 2021. Accessed: February 2, 2021.

Ömer, B. QCL – A programming language for quantum computers, Unpublished Master's thesis, Technical University of Vienna, 2000. http://tph.tuwien.ac.at/~oemer/doc/quprog.pdf.

Ömer, B. Classical concepts in quantum programming. *International Journal of Theoretical Physics*, 44(7):943–955, 2005. doi: 10.1007/s10773-005-7071-x.

Paler, A., R. Wille, and S. J. Devitt. Wire recycling for quantum circuit optimization. *Physical Review A*, 94(4), 2016. doi: 10.1103/physreva.94.042337.

Pan, F. and P. Zhang. Simulating the Sycamore quantum supremacy circuits, 2021. arXiv:2103.03074v1 [quant-ph].

Patel, R. B., J. Ho, F. Ferreyrol, T. C. Ralph, and G. J. Pryde. A quantum Fredkin gate. *Science Advances*, 2(3), 2016. doi: 10.1126/sciadv.1501531.

Pati, A. K. and S. L. Braunstein. Impossibility of deleting an unknown quantum state. *Nature*, 404(6774):164–165, 2000 doi: 10.1038/404130b0.

Patra, B., J. P. G. van Dijk, S. Subramanian, et al. A scalable cryo-CMOS 2-to-20GHz digitally intensive controller for 4x32 frequency multiplexed spin qubits/transmons in 22nm FinFET technology for quantum computers. In *2020 IEEE International Solid-State Circuits Conference – (ISSCC)*, pp. 304–306, 2020. doi: 10.1109/ISSCC19947.2020.9063109.

Pednault, E., J. A. Gunnels, G. Nannicini, L. Horesh, and R. Wisnieff. Leveraging secondary storage to simulate deep 54-qubit Sycamore circuits, 2019. arXiv:1910.09534.

Penrose, R. *The Road to Reality: A Complete Guide to the Laws of the Universe*. London: Vintage, 2005. ISBN: 9780593315309 http://books.google.com/books?id=SkwiEAAAQBAJ.

Perdomo, O. N. Castaneda, and R. Vogeler. Preparation of 3-qubit states. arXiv:2201.03724 [quant-ph].

Perdomo, O. Preparation of a 2-qubit state. http://youtu.be/LIdYSs-rE-o. Accessed: March 26, 2024.

Perdomo, O. Preparation of a 3-qubit state. http://youtu.be/Kne0Vq7gyzQ?si=-kXNUpNYSI8OORxD. Accessed: March 26, 2024.

Peruzzo, A., J. McClean, P. Shadbolt, et al. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1):4213, 2014. doi: 10.1038/ncomms5213.

Piveteau, Christoph and Sutter, David, Circuit knitting with classical communication. *IEEE Transactions on Information Theory*, (70) 2734–2745.

Plenio, M., S. Virmani An introduction to entanglement measures. arXiv:quant-ph/0504163, 2006.

Preskill, J. Quantum computing and the entanglement frontier, 2012. arXiv:1203.5813v3 [quant-ph].

Preskill, J. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018. doi: 10.22331/q-2018-08-06-79.

PSI Online. PSI. http://psilang.org/, 2021. Accessed: February 10, 2021.

www.quantum-inspire.com/kbase/grover-algorithm/ Accessed: September 1, 2024.

QCL Online. QCL. http://tph.tuwien.ac.at/∼oemer/qcl.html, 2021.

Qiskit. IBM qiskit simulators. http://qiskit.org/documentation/tutorials/simulators/1_aer_provider.html, 2021. Accessed: February 10, 2021.

Quantiki. List of simulators. http://quantiki.org/wiki/list-qc-simulators, 2021. Accessed: February 10, 2021.

Quantum Programming http://en.wikipedia.org/wiki/Quantum_programming Accessed: July 27, 2024.

Quipper Online. Quipper. www.mathstat.dal.ca/~selinger/quipper/, 2021. Accessed: February 10, 2021.

Regev, O. An efficient quantum factoring algorithm, arXiv, quant-ph, 2308.06572, 2024.

Rios, F. and P. Selinger. A categorical model for a quantum circuit description language (extended abstract). *Electronic Proceedings in Theoretical Computer Science*, 266:164–178, 2018. doi: 10.4204/eptcs.266.11.

Rivest, R. L., A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

Rolf, L. Is quantum mechanics useful? *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 353:367–376, 1995. doi: 10.1098/rsta.1995.0106.

Ross, N. J. Algebraic and logical methods in quantum computation, 2017. http://arxiv.org/abs/1510.02198.

Ross, N. J. and P. Selinger. Optimal ancilla-free Clifford+T approximation of z-rotations. *Quantum Information and Computation*, 11–12:901–953, 2016.

Ross, N. J. and P. Selinger. Exact and approximate synthesis of quantum circuits. www.mathstat.dal.ca/ selinger/newsynth/, 2021. Accessed: February 10, 2021.

Rudiak-Gould, B. The sum-over-histories formulation of quantum computing. *arXiv e-prints*, art. quant-ph/0607151, 2006.

Scarini, V. et al. Quantum cloning. *Reviews of Modern Physics*. doi: 10.1103/RevModPhys.77.1225.

Shende, V. V., I. L. Markov, and S. S. Bullock. Minimal universal two-qubit controlled-NOT-based circuits. *Physical Review A*, 69(6):062321, 2004. doi: 10.1103/physreva.69.062321.

Shor, P. W. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

Shor, P. W. Scheme for reducing decoherence in quantum computer memory. *Physics Review A*, 52:R2493–R2496, 1995. doi: 10.1103/PhysRevA.52.R2493.

Simon, D. On the power of quantum computation. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 116–123, 1994. doi: 10.1109/SFCS.1994.365701.

Smelyanskiy, M., N. P. D. Sawaya, and A. Aspuru-Guzik. qHiPSTER: The quantum high performance software testing environment, 2016. arXiv:1601.07195v2 [quant-ph].

Soeken, M., S. Frehse, R. Wille, and R. Drechsler. RevKit: An open source toolkit for the design of reversible circuits. In *Reversible Computation 2011*, vol. 7165 of *Lecture Notes in Computer Science*, pp. 64–76, 2012. RevKit is available at www.revkit.org.

Soeken, M., H. Riener, W. Haaswijk, et al. The EPFL logic synthesis libraries, 2019. doi: arXiv:1805.05121v2.

Soltan, A., Z. Ahmad, A. Mamat, and Z. Hishamuddin, A quantum algorithm for minimal spanning tree, Proceedings of the International Symposium on Information Technology, 2008. doi: 10.1109/ITSIM.2008.4632038.

Steane, A. Multiple particle interference and quantum error correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 452(1954):2551–2577, 1996. doi: 10.1098/rspa.1996.0136.

Steiger, D. S., T. Häner, and M. Troyer. ProjectQ: An open source software framework for quantum computing. *Quantum*, 2:49, 2018. doi: 10.22331/q-2018-01-31-49.

Strang, G. *Introduction to Linear Algebra*. MIT Press, 2016.

Svore, K. M., A. V. Aho, A. W. Cross, I. Chuang, and I. L. Markov. A layered software architecture for quantum computing design tools. *Computer*, 39(1):74–83, 2006. doi: 10.1109/MC.2006.4.

Tang, W., T. Tomesh, M. Suchara, J. Larson, and M. Martonosi. CutQC: Using small Quantum computers for large Quantum circuit evaluations. 2021. doi: 10.1145/3445814.3446758.

Terhal, B. DS-1999-04: Quantum Algorithms and Quantum Entanglement *Thesis, University of Amsterdam*:17, 1999. URI: http://eprints.illc.uva.nl/id/eprint/2012.

van Tonder, A. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004. doi: 10.1137/s0097539703432165.

Tsirelson (Cirel'son), B. S. (1980). Quantum generalizations of Bell's inequality. *Letters in Mathematical Physics*, 4(2), 93–100. doi: 10.1007/BF00417500.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention Is All You Need*. ISBN: 9781510860964.

Vidick, W. *Introduction to Quantum Cryptography*. Cambridge University Press, 2023.

Wang, S., A. Baksi, and A. Chattopadhyay. A Higher radix architecture for quantum carry-lookahead adder. *Nature*, http://arxiv.org/abs/2304.02921.

Whitfield, J. D., J. Biamonte, and A. Aspuru-Guzik. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics*, 109(5):735–750, 2011. doi: 10.1080/00268976.2011.552441.

Wikipedia. Constraint graphs. http://en.wikipedia.org/wiki/Constraint_graph, 2024a Accessed: September 3, 2024.

Wikipedia. ECC, Error correction code memory. http://en.wikipedia.org/wiki/ECC_memory, 2021b. Accessed: February 10, 2021.

Wikipedia. Extended Euclidean algorithm. http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm, 2021c. Accessed: February 10, 2021.

Wikipedia. Gradient descent. http://en.wikipedia.org/wiki/Gradient_descent, 2021d. Accessed: February 10, 2021.

Wikipedia. KD-Trees. http://en.wikipedia.org/wiki/K-d_tree, 2021a. Accessed: February 10, 2021.

Wikipedia. Shor's algorithm. http://en.wikipedia.org/wiki/Shor%27s_algorithm, 2024f. Accessed: September 7, 2024.

Wikipedia. Euclidean algorithm. http://en.wikipedia.org/wiki/Euclidean_algorithm, 2024g. Accessed: September 7, 2024.

Williams, C. P. *Explorations in Quantum Computing*. Springer-Verlag, London, 2011. doi: 10.1007/978-1-84628-887-6.

Wilson, E., S. Singh, and F. Mueller. Just-in-time quantum circuit transpilation reduces noise. IEEE International Conference on Quantum Computing and Engineering. 2020. doi: 10.1109/QCE49297.2020.00050.

Wootters, W. K. and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982. doi: 10.1038/299802a0.

Xue, X., B. Patra, J. P. G. van Dijk, et al. Cmos-based cryogenic control of silicon quantum circuits. *Nature*, 593(7858):205–210, 2021. doi: 10.1038/s41586-021-03469-4.

Morrell, H. Jr., A. Zaman, and H. Y. Wong. A step-by-step HHL algorithm walkthrough to enhance understanding of critical quantum computing concepts. IEEE Access. 2023. doi: 10.1109/access.2023.3297658.

Zhang, Y., L. Cincio, C. Negre, et al. Variational quantum eigensolver with reduced circuit complexity. *Nature*. 2022. doi: 10.1038/s41534-022-00599-z.

Znidaric, M., O. Giraud, and B. Georgeot. Optimal number of controlled-NOT gates to generate a three-qubit state. *American Physical Society*. 77(3):032320, 2008. doi: 10.1103/PhysRevA.77.032320.

Index