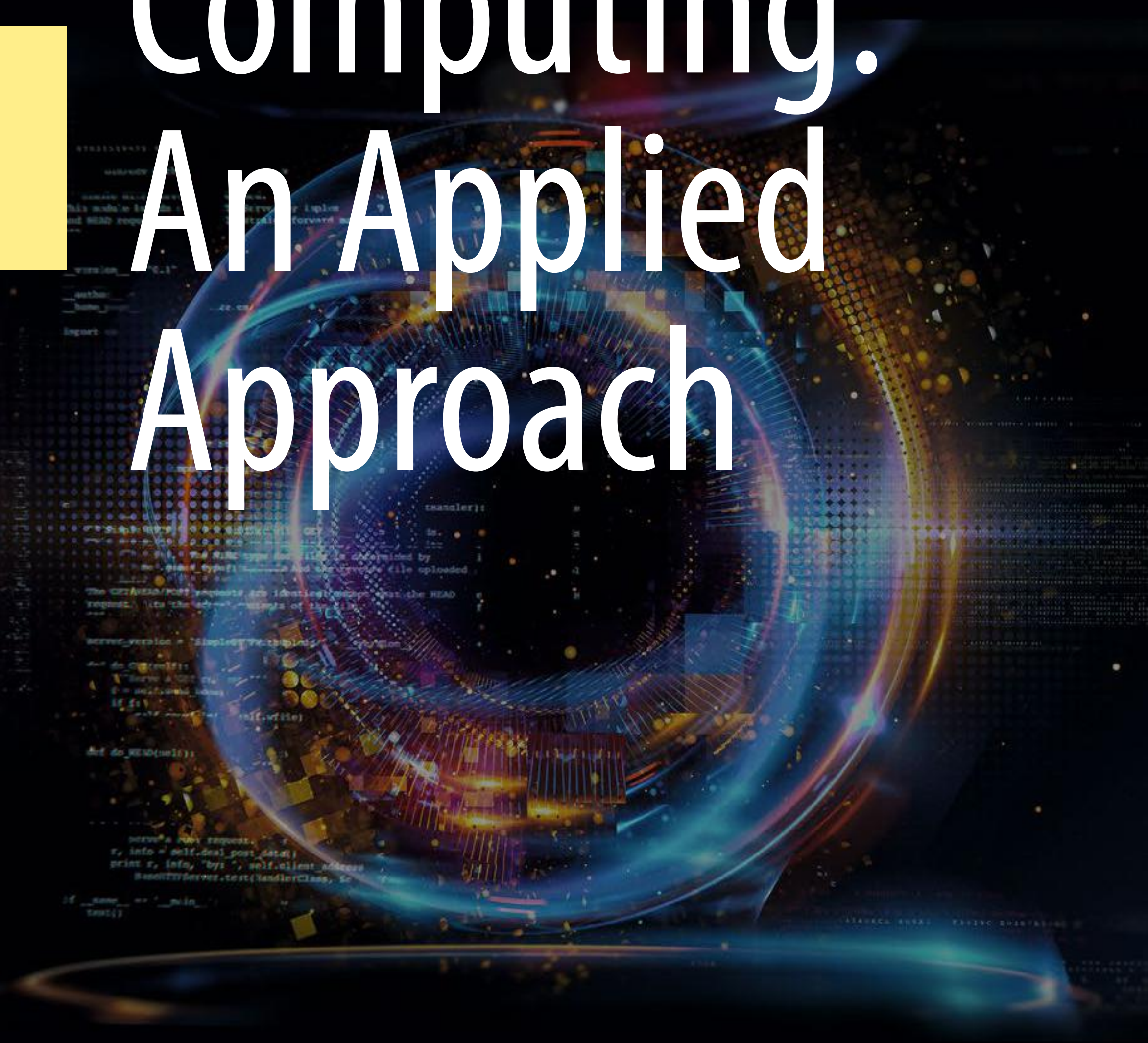
Jack D. Hidary

Quantum Computing: An Applied Approach



Springer

Quantum Computing: An Applied Approach

Jack D. Hidary

Quantum Computing: An Applied Approach

Springer

Jack D. Hidary
Alphabet X
Mountain View, CA
USA

ISBN 978-3-030-23921-3 ISBN 978-3-030-23922-0 (eBook)
https://doi.org/10.1007/978-3-030-23922-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

I Foundations

II Hardware and Applications

III Toolkit

Preface

We are entering a new era of computation that will catalyze discoveries in science and technology. Novel computing platforms will probe the fundamental laws of our universe and aid in solving hard problems that affect all of us. Machine learning programs powered by specialized chips are already yielding breakthrough after breakthrough.

In this book we will explore quantum computing – an emerging platform that is fundamentally different than the way we compute with current digital platforms. To be sure, we are years away from scaled quantum computers. Yet, we now know that such systems are possible; with advances in engineering we are likely to see real impact.

Quantum computing is part of the larger field of quantum information sciences (QIS). All three branches of QIS – computation, communication and sensing – are advancing at rapid rates and a discovery in one area can spur progress in another. Quantum communication leverages the unusual properties of quantum systems to transmit information in a manner that no eavesdropper can read. This field is becoming increasingly critical as quantum computing drives us to a post-quantum cryptography regime. We will cover quantum teleportation and superdense coding, which are both quantum-specific protocols, in chapter 7.

Quantum sensing is a robust field of research which uses quantum devices to move beyond classical limits in sensing magnetic and other fields. For example, there is an emerging class of sensors for detecting position, navigation and timing (PNT) at the atomic scale. These micro-PNT devices can provide highly accurate positioning data when GPS is jammed or unavailable.

In this book we will focus on quantum computation. One of the critical differences between quantum and classical computation is that in quantum computation *we are manipulating quantum states themselves*; this gives us a much larger computing space to work in than in classical computers. In classical computers, if we wish to model a real-world quantum physical system, we can only do so with representations of such a system and we cannot implement the physics itself.

This key difference leads to exciting possibilities for the future of computing and science. All this starts with fundamental truths about our world that were developed during the quantum mechanics revolution in the first half of the 20th century. We will review a number of these core concepts in the first chapter.

I had the fortunate circumstance to have studied quantum mechanics before learning classical physics and therefore relate to quantum physics as the norm – it is my intellectual home. Until we change our educational system, most students will learn the classical before the quantum and so the quantum will seem doubly strange — both from their own human experience as well as from the inculcation of classical ideas before quantum ideas can be introduced.

What is ironic about this state of affairs is that the primary mathematical tool in quantum mechanics is linear algebra, a powerful but very accessible branch of mathematics. Most students, however, only take linear algebra after two or three semesters of calculus, if they take it at all. Yet, no calculus is needed to introduce linear algebra! In any case, we will leave the remedying of mathematics education to another day while we embark here on a journey into a new form of computing.

In this book we will explore how to build a computer of a very different kind than humans have ever built before. What is distinct about this book is that we will go beyond the theoretical into the practical work of how we can build such computers and *how we can write applications for these systems*. There are now several development libraries which we can use to program cloud-based quantum systems. We will walk through code examples and show the reader how to build a quantum circuit comprised of a set of operators to address a particular challenge. We will mainly use Python in this book.

We are currently in the regime of **n**oisy **i**ntermediate-**s**cale **q**uantum (NISQ) computers, a term coined by John Preskill of CalTech [176]. This refers to systems that do not yet have full error-correction (thus noisy) and have dozens to thousands of qubits – well short of the 10^6+ necessary for scaled fault-tolerant computing. Despite the limitations of these initial systems, the theory, algorithms and coding techniques we cover in this book will serve readers as they transition to larger systems that are to come in the future.

This work is three books in one: the first part covers the necessary framework that drives the design of quantum computers and circuits. We will also explore what kinds of problems may be amenable to quantum computation in our treatment of complexity classes.

The second part of the book is for those readers who wish to delve into the programming that makes these new machines tick. If you already have a background in quantum mechanics, quantum information theory and theoretical computer science (you know who you are!), you can jump right to the second part and dig into the code. Please refer to the navigation guide in the following pages to chart a course through this material.

In the third part we provide a set of critical tools to use in the journey to master quantum computing (QC). We build up the core concepts of linear algebra and tie them specifically to their use in QC. The table of operators and circuit elements in chapter 14 is a handy reference as you design your own quantum computing protocols.

The book is also a portal to the growing body of literature on the subject. We recommend that the reader use the bibliography to explore both foundational and recent papers in the field.

We will provide further online examples and code tutorials on a continual basis. This a living text that will develop as QC technology matures. We are all travelers together on this new adventure; join us online at this book's GitHub site.[1] We are excited to see what you will develop with these new platforms and tools. Contact us via the site — we look forward to hearing from you.

Jack D. Hidary
June 2019
35,000 ft up

[1] http://www.github.com/jackhidary/quantumcomputingbook

Acknowledgements

Let me start by thanking my publisher, Elizabeth Loew, who has been so supportive throughout the process, and the entire SpringerNature team for their excellence.

A book like this is a significant undertaking and it took a team of people to help in so many ways. The greatest of appreciation to Stefan Leichenauer who did a great job as book editor and formatter-in-chief. Stefan reviewed large parts of the book and I thank him for his commitment to the project. The entire book is written in TeX and we often pushed the boundaries of what TeX is capable of implementing.

Thank you to Sheldon Axler, author of *Linear Algebra Done Right* (also by Springer) who generously shared his TeX template so that we could format the book correctly for the Springer standards.

Thanks to the many experts who gave of their time to review key sections of the book and provide technical advice. These include (in alphabetical order): Scott Aaronson, Ryan Babbush, Sergio Boixo, Ruffin Evans, Eddie Farhi, Patrick Hayden, Gerry Gilbert, Matt Reagor and Lenny Susskind. Each improved the work materially with their input.

I would also like to recognize the significant achievement of Michael Nielsen and Isaac Chuang in developing their textbook [161]. Recognition as well to John Preskill for his in-depth lecture notes [174]. Nielsen, Chuang and Preskill as well as Mermin [151] and Rieffel [186] have helped many individuals enter the field.

Thanks to my many colleagues at Alphabet X and Google who have encouraged this effort including: Sergey Brin, Astro Teller and Hartmut Neven and his excellent team.

Thank you to the many participants in the workshops I taught on quantum computing as well as my courses on linear algebra and other mathematical topics. Your feedback has been invaluable.

Hearty thanks to James Myer who worked with me intensively on the math sections; James' methodical approach assured us of success. We reviewed these sections countless times, continually reworking them. James is not only passionate about mathematics; he also cares deeply about pedagogy and we had productive discussions on the best way to present the core concepts. Thank you as well to Tai-Danae Bradley who also reviewed the math sections and made very helpful suggestions.

Thank you to Ryan LaRose who worked with me on the code sections. In this emerging field where the code frameworks have only been developed within the last few years, solid information and examples can be hard to come by. Ryan combined great skill in research as well as compiling the information in succinct forms. Ryan also did a great job on the book's GitHub site.

Thank you to Ellen Cassidy who did such a professional job proofreading the text for grammar and consistency of format. Ellen has an eagle eye and I commend her work to any author. Joe Tricot also did a wonderful job coordinating the overall process.

Naturally, even with all this help, there will remain items to fix for the next edition. All remaining errors are mine and I will be posting updates on the GitHub site and then include fixes in upcoming versions.

Thank you to my parents, David and Aimee Hidary, and my entire wonderful family for their support through this process. It is a great feeling to share this accomplishment with you.

Navigating this Book

Here are our suggestions to make the best use of this book:

1. *University instructors:* You can build several different courses with the material in this book. All code from the book is on the book's website. The math chapters have exercises embedded throughout; for other chapters please consult the online site for coding exercises and other problem sets.
   (a) Course in Quantum Computing for STEM majors:
      i. For this course we recommend assigning chapters 1 and 2 as pre-reading for the course and then proceeding chapter by chapter with the exercises provided on the GitHub site. Solutions are also available on the site.
      ii. If the students do not have sufficient depth in formal linear algebra and related mathematical tools, Part III forms a strong basis for a multi-week treatment with exercises.
   (b) Course in Quantum Computing for physics graduate students:
      i. For this course, we recommend using this book in conjunction with Mike and Ike (which is the way many of us refer to Nielsen and Chuang's excellent textbook [161]) or another suitable text which covers the theoretical concepts in depth. We all owe a huge debt of gratitude to Michael Nielsen, Isaac Chuang and authors of other textbooks over the last twenty years. We also recommend referring to John Preskill's lecture notes as you build your course for advanced physics students [174]. Our work is meant to be complementary to Mike and Ike in several respects:
         A. This work is more focused on coding. For obvious reasons, books written prior to the past few years could not have covered the dev tools and Python-based approaches to quantum computing that now exist.

B. This book does not go into the depth that Mike and Ike does on information theoretic concepts.

C. This book's mathematical tools section has a more detailed ramp-up for those students who may not have taken a rigorous linear algebra course. The short summaries of linear algebra and other requisite math tools in other textbooks on quantum mechanics are often insufficient in our experience.

ii. We recommend first assigning chapters 1 and 2 as pre-reading.

iii. Next, we suggest covering the chapters on unitary operators, measurement and quantum circuits with exercises on the Github site to check knowledge.

iv. We then recommend spending the bulk of the course in Part II to provide the students with hands-on experience with the code.

(c) Course in Quantum Computing for CS graduate students:

i. We suggest assigning the first two chapters as pre-reading and then a review of mathematical tools in Part III. Prior exposure to only undergraduate linear algebra is typically insufficient as it was most likely taught without the full formalism.

ii. We then recommend chapters 3 and 4 to build up familiarity with unitary operators, measurement and complexity classes in the quantum regime. The instructor can make use of the review questions and answers on the GitHub site.

iii. The course can then cover the approaches to building a quantum computer followed by all the coding chapters.

Please check the book's GitHub site to find additional resources including: code from the book, problem sets, solutions, links to videos and other pedagogical resources.

2. *Physicists:* For physicists who specialize in fields outside of quantum computing and wish to ramp up quickly in this area, we recommend reading the brief history of QC as we provide more detail than typical treatments, then the survey of quantum hardware followed by the applications in the second part of the book.

3. *Software engineers:* We recommend starting with the opening two chapters, then reviewing the toolkits in Part III. We then suggest returning to the treatment of qubits and unitary operators in Part I and proceeding from there.

4. *Engineering and business leaders:* For readers who will not be doing hands-on coding, we recommend focusing on chapters 1-4. The more adventurous may want to work through some of the code examples to get a tangible feel for the algorithms.
5. *Independent study:* This book can easily be used as a text for independent study. We recommend combining it with online resources. Please consult the GitHub site for an updated list of resources:

   `http://www.github.com/jackhidary/quantumcomputingbook`

   We recommend first assessing your current fluency on the core tools in Part III; there are numerous self-tests throughout the section that can be used for this purpose. The reader can then proceed to Part I.

   For those with a strong background in quantum mechanics and/or information theory we recommend looking up the papers referenced in chapters 2-4 to gain a deeper understanding of the state of the field before proceeding to Part II: Hardware and Applications.

Please consult the book's GitHub site to find a range of resources including: code from the book, problem sets, solutions, links to videos and other pedagogical resources.

Part I

Foundations

CHAPTER 1

Superposition, Entanglement and Reversibility

What is a quantum computer? The answer to this question encompasses quantum mechanics (QM), quantum information theory (QIT) and computer science (CS).

For our purposes, we will focus on the core of what makes a quantum computer distinct from classical computers.

1.1 Quantum Computer Definition

A quantum computer is a device that leverages specific properties described by quantum mechanics to perform computation.

Every classical (that is, non-quantum) computer can be described by quantum mechanics since quantum mechanics is the basis of the physical universe. However, a classical computer does not take advantage of the specific properties and states that quantum mechanics affords us in doing its calculations.

To delve into the specific properties we use in quantum computers, let us first discuss a few key concepts of quantum mechanics:

- How do we represent the superposition of states in a quantum system?
- What is entanglement?
- What is the connection between reversibility, computation and physical systems?

We will be using Dirac notation, linear algebra and other tools extensively in this text; readers are encouraged to refer to the math chapters later in this work to review as needed.

According to the principles of quantum mechanics, systems are set to a definite state only once they are measured. Before a measurement, systems

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_1

are in an indeterminate state; after we measure them, they are in a definite state. If we have a system that can take on one of two discrete states when measured, we can represent the two states in Dirac notation as $|0\rangle$ and $|1\rangle$. We can then represent a *superposition of states* as a linear combination of these states, such as

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

1.2 The Superposition Principle

The linear combination of two or more state vectors is another state vector in the same Hilbert space[a] and describes another state of the system.

[a] See Part III for a treatment of Hilbert spaces

As an example, let us consider a property of light that illustrates a superposition of states. Light has an intrinsic property called *polarization* which we can use to illustrate a superposition of states. In almost all of the light we see in everyday life — from the sun, for example — there is no preferred direction for the polarization. Polarization states can be selected by means of a *polarizing filter*, a thin film with an axis that only allows light with polarization parallel to that axis to pass through.

With a single polarizing filter, we can select one polarization of light, for example *vertical polarization*, which we can denote as $|\uparrow\rangle$. *Horizontal polarization*, which we can denote as $|\rightarrow\rangle$, is an orthogonal state to vertical polarization[1]. Together, these states form a basis for any polarization of light. That is, any polarization state $|\psi\rangle$ can be written as linear combination of these states. We use the Greek letter ψ to denote the state of the system

$$|\psi\rangle = \alpha|\uparrow\rangle + \beta|\rightarrow\rangle$$

The coefficients α and β are complex numbers known as *amplitudes*. The coefficient α is associated with vertical polarization and the coefficient β is associated with horizontal polarization. These have an important interpretation in quantum mechanics which we will see shortly.

After selecting vertical polarization with a polarizing filter, we can then introduce a second polarizing filter after the first. Imagine we oriented the

[1] We could have equally used $|0\rangle$ and $|1\rangle$ to denote the two polarization states; the labels used in kets are arbitrary.

axis of the second filter perpendicular to the axis of the first. Would we see any light get through the second filter?

If you answered no to this question, you would be correct. The horizontal state $|{\rightarrow}\rangle$ is orthogonal to the first, so there is no amount of horizontal polarization after the first vertical filter.

Suppose now we oriented the axis of the second polarizing filter at 45° (i.e., along the diagonal $\nearrow$ between vertical $\uparrow$ and horizontal $\rightarrow$) to the first instead of horizontally. Now we ask the same question — would we see any light get through the second filter?

If you answered no to this question, you may be surprised to find the answer is *yes*. We would, in fact, see some amount of light get through the second filter. How could this be if all light after the first filter has vertical polarization? The reason is that we can express vertical polarization as a *superposition* of diagonal components. That is, letting $|{\nearrow}\rangle$ denote 45° polarization and $|{\nwarrow}\rangle$ denote $-45°$ polarization, we may write

$$|{\uparrow}\rangle = \frac{1}{\sqrt{2}}|{\nearrow}\rangle + \frac{1}{\sqrt{2}}|{\nwarrow}\rangle$$

As you may expect from geometric intuition, the vertical state consists of equal parts $|{\nearrow}\rangle$ and $|{\nwarrow}\rangle$.

It is for this reason that we see some amount of light get past the second filter. Namely, the vertical polarization can be written as a *superposition* of states, one of which is precisely the 45° diagonal state $|{\nearrow}\rangle$ we are allowing through the second filter. Since the $|{\nearrow}\rangle$ state is only one term in the superposition, not all of the light gets through the filter, but some does. The amount that gets transmitted is precisely $1/2$ in this case. (More formally, the intensity of the transmitted light is $1/2$ that of the incident light.) This value is determined from the amplitudes of the superposition state by a law known as Born's rule, which we now discuss.

Max Born demonstrated in his 1926 paper that **the modulus squared of the amplitude of a state is the probability of that state resulting after measurement** [38]. In this case, since the amplitude is $\frac{1}{\sqrt{2}}$ the probability of obtaining that state is $\left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$, so the probability of measuring the light in either the vertical or horizontal polarization state is 50%. Note that we chose an amplitude of $\frac{1}{\sqrt{2}}$ in order to *normalize* the states so that the sum of the modulus squared of the amplitudes will equal one; this enables us to connect the amplitudes to probabilities of measurement with the Born rule.

1.3 The Born rule

In a superposition of states, the modulus squared of the amplitude of a state is the probability of that state resulting after measurement. Furthermore, the sum of the squares of the amplitudes of all possible states in the superposition is equal to 1. So, for the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, we have

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

While in the polarization example above we have a 50/50 split in probability for each of two states, if we examined some other physical system it may have a 75/25 split or some other probability distribution. One critical difference between classical and quantum mechanics is that amplitudes (*not probabilities*) can be complex numbers.

In other words, the coefficients α and β which appear in the statement of the Born rule can be complex numbers, such as $i := \sqrt{-1}$ or $(1 + i)/\sqrt{2}$. It is only after we take the square of the modulus of these amplitudes that we get real numbers, hence actual probabilities. Refer to chapter 11 to review complex numbers and how to determine the square of the modulus of a complex number.

As if quantum superposition were not odd enough, QM describes a specific kind of superposition which stretches our imagination even further: *entanglement*. In 1935, when **E**instein worked with **P**odolsky and **R**osen to publish their paper on quantum entanglement [75], their aim was to attack the edifice of QM (this paper is now known as EPR). Even though Einstein earned the Nobel Prize for his 1905 work on the quantum nature of the photoelectric effect, he nevertheless railed against the implications of QM until his later years.

Einstein wrote in 1952 that quantum mechanics appears to him to be “a system of delusion of an exceedingly intelligent paranoiac concocted of incoherent elements of thought” [74]. He hoped that the EPR paper would demonstrate what he perceived to be the deficiencies of QM.

EPR showed that if you take two particles that are entangled with each other then and then measure one of them, this automatically triggers a correlated state of the second — even if the two are at a great distance from each other; this was the seemingly illogical result that EPR hoped to use to show that QM itself must have a flaw. Ironically, we now consider entanglement to be a cornerstone of QM. Entanglement occurs when we have a superposition of states that is not separable. We will put this into a more formal context later on in this text.

This "spooky action at a distance" seems at odds with our intuition and with previous physics. Podolsky, the youngest of the co-authors, reportedly leaked the paper to the New York Times to highlight this assault on the tower of QM to the public. The Times ran the story on the front page of the May 4th, 1935 edition with the headline “Einstein Attacks Quantum Theory."

Not only is entanglement accepted as part of standard quantum mechanics, we shall see later in this work that we can leverage entanglement to perform novel types of computation and communication. From an information theoretic point of view, entanglement is a different way of encoding information. If we have two particles that are entangled, the information about them is not encoded locally in each particle, but rather in the correlation of the two.

John Preskill likes to give the analogy of two kinds of books: non-entangled and entangled [176]. In the regular, non-entangled book we can read the information on each page as we normally do. In the entangled book, however, each page contains what appears to be gibberish. The information is encoded in the correlation of the pages, not in each page alone. This captures what Schrödinger expressed when he coined the term entanglement:

> Another way of expressing the peculiar situation is: the best possible knowledge of a whole does not necessarily include the best possible knowledge of all its parts. [196]

Schrödinger further noted that in his opinion entanglement was not just *one* of the phenomena described by quantum mechanics,“but rather *the* characteristic trait of quantum mechanics, the one that enforces its entire departure from classical lines of thought” [196].

1.4 Entanglement

Two systems are in a special case of quantum mechanical superposition called *entanglement* if the measurement of one system is correlated with the state of the other system in a way that is stronger than correlations in the classical world. In other words, the states of the two systems are not *separable*. We will explore the precise mathematical definitions of separability and entanglement later in this book.

Now that we have covered two core ideas of quantum mechanics – superposition and entanglement – let us turn to another fundamental concept that is not treated as often – the physicality of information. Rolf Landauer opened a new line of inquiry when he asked the following question:

> The search for faster and more compact computing circuits leads directly to the question: What are the ultimate physical limitations on the progress in this direction? ...we can show, or at the very least strongly suggest, that information processing is inevitably accompanied by a certain minimum amount of heat generation. [128]

In other words, is there a lower bound to the energy dissipated in the process of a basic unit of computation? Due to Landauer and others we now believe that there is such a limit; this is called Landauer's bound (LB). More specifically, the energy cost of erasure of n bits is $nkT \ln 2$ where k is the Boltzmann constant, T is the temperature in Kelvin of the heat sink surrounding the computing device and $\ln 2$ is, of course, the natural log of 2 (~ 0.69315). This limit is the minimum amount of energy dissipated for an irreversible computation.

Landauer acknowledged that this minimum is not necessarily the constraining factor on the energy draw of the system:

> It is, of course, apparent that both the thermal noise and the requirements for energy dissipation are on a scale which is entirely negligible in present-day computer components. The dissipation as calculated, however, is an absolute minimum. [128]

Landauer defined logical irreversibility as a condition in which "the output of a device does not uniquely define the inputs." He then claimed that "logical irreversibility...in turn implies physical irreversibility, and the latter is accompanied by dissipative effects." This follows from the second law of thermodynamics which states that the total entropy of a system cannot decrease and, more specifically, must increase with an irreversible process. For further background on reversibility, thermodynamics and computation see Feynman's *Lectures on Computation* [84].

In classical computing we make use of irreversible computations. For example, the Boolean inclusive *OR* (denoted $\vee$) gate has the following truth table, where 0 denotes "false" and 1 denotes "true":

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

Note that an output of value 1 cannot be traced uniquely to a set of inputs. We can arrive at that output through combinations of inputs; the state of the inputs is lost once we move to the output. This does not violate the conservation of information because the information was converted into dissipative heat.

The exclusive *OR* is also irreversible as is the *NAND* gate, which is universal for classical computing. *NAND* stands for "*NOT AND*" and is the inverse of the Boolean *AND* operator. Verify for yourself that *NAND* is irreversible by examining its truth table:

X	Y	X ↑ Y
0	0	1
0	1	1
1	0	1
1	1	0

In quantum computing, we limit ourselves to *reversible* logical operations [161, p. 29]. Later in this book we will consider which combinations of quantum operators are universal. For now, let's focus on the requirement in quantum computation that we limit our set of operators to reversible gates.

This requirement derives from the nature of irreversible operations: if we perform an irreversible operation, we have lost information and therefore a measurement. Our computation cycle then will be done and we can no longer continue with our program. Instead, by limiting all gates to reversible operators, we may continue to apply operators to our set of qubits as long as we can maintain coherence in the system. When we say reversible, we are assuming a theoretical noiseless quantum computer. In a noisy QC that decoheres, we cannot, of course, reverse the operation.

1.5 Reversibility of Quantum Computation

All operators used in quantum computation other than for measurement must be reversible.

In this chapter, we have examined four essential principles of quantum mechanical systems: superposition, the Born rule, entanglement and reversible computation. All four are essential to understanding the difference between classical and quantum computing as we shall see further in the book. We provide references on this book's website to a number of resources for those who wish to deepen their understanding of quantum mechanics.

Our generous universe comes equipped with the ability to compute.
—*Dave Bacon* [19]

CHAPTER 2

A Brief History of Quantum Computing

The possibility that we can leverage quantum mechanics to do computation in new and interesting ways has been hiding in plain sight since the field's early days; the principles of superposition and entanglement can form the basis of a very powerful form of computation. The trick is to build such a system that we can easily manipulate and measure.

While Richard Feynman is often credited with the conception of quantum computers, there were several researchers who anticipated this idea. In 1979, Paul Benioff, a young physicist at Argonne National Labs, submitted a paper entitled "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines" [23].[1] In this paper, Benioff demonstrated the theoretical basis for quantum computing and then suggested that such a computer could be built:

> That is, the whole computation process is described by a pure state evolving under the action of a given Hamiltonian. Thus all the component parts of the Turing machine are described by states which have a definite phase relation to one another as the calculation progresses...The existence of such models at least suggests that the possibility of actually constructing such coherent machines should be examined.

Yuri Manin also laid out the core idea of quantum computing in his 1980 book *Computable and Non-Computable* [140]. The book was written in Russian, however, and only translated years later.

[1] Note: Benioff completed and submitted the paper in 1979. It was published in the following year, 1980.

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_2

In 1981, Feynman gave a lecture entitled "Simulating Physics with Computers" [83].[2] In this talk, he argued that a classical system could not adequately represent a quantum mechanical system:

> ...nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy...

He then set out the features that a quantum computer should have to be useful. At the time of this lecture, however, it was unclear to Feynman and the physics community how one could build such a machine.

Once Benioff, Manin and Feynman opened the doors, researchers began to investigate the nature of the algorithms that could be run on QCs. David Deutsch, a physicist at Oxford, suggested a more comprehensive framework for quantum computing in his 1985 paper [65]. In this work, he describes in detail what a quantum algorithm would look like and anticipates that "one day it will become technologically possible to build quantum computers."

Deutsch then went on to develop an example of an algorithm that would run faster on a quantum computer. He then further generalized this algorithm in collaboration with Richard Jozsa [67]. We will cover these and the other algorithms in more detail with code examples later on in this text.

In computer science and quantum computing, it is often important to evaluate how efficient an algorithm is — that is, how many steps would it take to run such an algorithm. We use big-O notation to represent the upper bound of the worst case of running an algorithm. The O in big-O notation comes from the "order" of the algorithm. We use big-Ω (Omega) notation to indicate the lower bound of the worst-case scenario. So, while Deutsch's problem takes at *worst* $O(n)$ steps to solve on a classical computer, Deutsch-Jozsa's algorithm solves the problem in one step on a quantum computer. Big-O notation will be helpful throughout this book in illuminating the difference between the classical and quantum algorithms.

Umesh Vazirani and his student Ethan Bernstein picked up where Deutsch and Jozsa left off. In 1993, Bernstein and Vazirani (BV) published a paper which described an algorithm that showed clear quantum-classical separation even when small errors are allowed [29]. Why is this significant? While Deutsch-Jozsa demonstrated a *deterministic* quantum advantage, if small errors are allowed in the computation, both classical and quantum versions

[2] Note: Feynman gave his lecture in 1981 and submitted the lecture for publication in May of 1981. The lecture was published by IJTP in 1982.

can be run at worst in $O(1)$ steps, showing no separation. By contrast, the Bernstein-Vazirani (BV) algorithm demonstrates separation even when small errors are allowed, thus showing non-deterministic quantum advantage. The problem posed in BV can be solved in $O(n)$ time on a classical computer and in $O(1)$ using the BV circuit on a quantum computer.

BV made a further contribution in their 1993 paper. They described a quantum version of the Fourier transform. This quantum Fourier transform (QFT) would serve as a critical component for Peter Shor when he developed his algorithm to factor large numbers.

The work of BV was quickly followed by Daniel Simon, then a postdoc at the University of Montreal, in 1994. Simon outlined a problem that a quantum computer would clearly solve exponentially faster than a classical one [203]. To be more specific, Simon's algorithm has an upper bound of $O(n)$ on a quantum computer, but a higher $\Omega(2^{n/2})$ on a classical computer. Since the lower bound on the classical computer is of higher order than the upper bound on the quantum computer, there is a clear demonstration of quantum advantage.

Just prior to Daniel Simon's work on algorithms, Seth Lloyd, working at Los Alamos, published a paper in *Science* which described a method of building a working quantum computer [136]. He proposed that a system sending pulses into a unit can represent a quantum state:

> Arrays of pulsed, weakly coupled quantum systems provide a potentially realizable basis for quantum computation. The basic unit in the array could be a quantum dot, a nuclear spin, a localized electronic state in a polymer, or any multistate quantum system that interacts locally with its neighbors and can be compelled to switch between states with resonant pulses of light.

Lloyd realized that:

> The proposed device is capable of purely quantum-mechanical information-processing capacities above and beyond the conventional digital capacities already presented. One of the most important of these capacities is that bits can be placed in superpositions of 0 and 1 by the simple application of pulses at the proper resonant frequencies but at a length different from that required to fully switch the bit. Such bits have a number of uses, including the generation of random numbers.

This was the first practical approach to a working QC. It is interesting that Lloyd noted the possible use case of generating random numbers from a quantum system; this has been a topic of recent research in the quantum computing community. See, for example. [107] and [1].

Enter Peter Shor. In 1994, Shor was a researcher in the mathematical division of Bell Labs in New Jersey. Shor studied the work of Deutsch, BV and Simon and realized he could construct an algorithm for factoring large numbers into two prime factors; factoring large numbers is believed to be intractable on a classical computer, but Shor's factoring algorithm runs quickly on a QC. Factoring large numbers is, of course, the intentionally hard problem at the core of public key cryptography (PKC) as implemented in the RSA algorithm[188], the kind of cryptography that is the basis of almost all communications today over the internet. This includes securely sending credit card numbers, bank payments and ensuring the security of online messaging systems.

RSA-based cryptography depends on the one-way hardness of the factoring of large numbers into two prime factors. Producing the large number is easy; we just multiply the two factors. Given an arbitrarily large number, however, it is exponentially difficult to find its two prime factors.

Inspired by Simon, Shor realized that we can use a QC to solve another problem that is equivalent to the factoring problem; the factoring problem is in fact equivalent to the period-finding problem which Simon had tackled in his paper [200]. He also realized that the QFT described by BV was exactly what he needed to set up the amplitudes of each qubit prior to measurement so that the measurement would yield the answer needed from the quantum computation with high probability.

Shor's breakthrough led more researchers to work on quantum algorithms since it then became clear that QCs, if built, would be quite powerful. In fact, Shor's algorithm is one of the first to have been demonstrated on early QC physical systems. In 2001, Isaac Chuang et al. implemented Shor's algorithm on a nuclear magnetic resonance (NMR) system to factor the number 15 as a demonstration [219].

After Shor, Lov Grover contributed to the quantum algorithm arsenal by demonstrating that one can achieve some speedup in a search algorithm on a QC [98]. Grover's algorithm only achieves quadratic speedup, not exponential speedup (as Shor's does), but this is still significant. Quadratic speedup means that if an algorithm would take $O(N)$ steps on a classical computer, we can achieve the same goal in $O(\sqrt{N})$ steps on a QC. A few months after Grover's paper in May of 1996, Farhi and Gutmann laid out the framework for a continuous time Hamiltonian version of Grover's algorithm [80]. This introduced the concept of Hamiltonian oracles and the idea of implementing continuous time models of quantum computation which are different than gate-based approaches.

As one set of researchers were making progress in identifying algorithms that would run on a quantum computer with speedup over classical computers, others were making progress on the physical implementation of a QC. In 1999-2001, Yasunobu Nakamura built and demonstrated a functioning, controllable superconducting qubit [157, 158]. Nakamura used Josephson junctions to create a two-level system that the user could manipulate between its two states. We will discuss processors based on superconducting qubits in chapter 5.

Another approach to implementing a quantum computer is to trap and manipulate ions. In 1995, Cirac and Zoller proposed an ion trap as the physical system to perform quantum computation [59]. In this setup, lasers are used to ionize atoms which are then trapped in electric potentials. We will cover ion trap quantum computers in chapter 5 as well.

As the activity in the quantum computing field began to rise, researchers in the field formalized what constituted a quantum computer and computation. In 1996, David DiVincenzo outlined the key criteria of a quantum computer [69] in this manner:[3]

1. A scalable physical system with qubits that are distinct from one another and the ability to count exactly how many qubits there are in the system (no fudging allowed). The system can be accurately represented by a Hilbert space.
2. The ability to initialize the state of any qubit to a definite state in the computational basis. For example, setting all qubits to state $|0\rangle$ if the computational basis vectors are $|0\rangle$ and $|1\rangle$.
3. The system's qubits must be able to hold their state. This means that the system must be isolated from the outside world, otherwise the qubits will decohere. Some decay of state is allowed (ϵ, where ϵ is a small quantity). In practice, the system's qubits must hold their state long enough to apply the next operator with assurance that the qubits have not changed state due to outside influences between operations.
4. The system must be able to apply a sequence of unitary operators to the qubit states. The system must also be able to apply a unitary operator to two qubits at once. This entails entanglement between those qubits. As DiVincenzo states in his paper, "...entanglement between different

[3] Note that we are summarizing DiVincenzo's criteria from his original 1996 paper. See [70] for another version of his criteria.

parts of the quantum computer is good; entanglement between the quantum computer and its environment is bad, since it corresponds to decoherence" [69, p. 4].

5. The system is capable of making "strong" measurements of each qubit. By strong measurement, DiVincenzo means that the measurement says "which orthogonal eigenstate of some particular Hermitian operator the quantum state belongs to, while at the same time projecting the wavefunction of the system irreversibly into the corresponding eigenfunction." This means that the measuring technique in the system actually does measure the state of the qubit for the property being measured and leaves the qubit in that state. DiVincenzo wants to prevent systems that have weak measurement, in other words, measuring techniques that do not couple with the qubit sufficiently to render it in that newly measured state. At the time he wrote the paper, many systems did not have sufficient coupling to guarantee projection into the new state.

In upcoming chapters, we will explore in further detail the various methods to physically construct a quantum computer and how to program them once built. Let's now turn our attention to qubits and the operators we use in quantum computation.

CHAPTER 3

Qubits, Operators and Measurement

In this chapter we will cover qubits and the core set of operators we use to manipulate the state of qubits.

A *qubit* is a quantum bit. A qubit is similar to a classical bit in that it can take on 0 or 1 as states, but it differs from a bit in that it can also take on a continuous range of values representing a superposition of states. In this text we will use qubit to refer to quantum bits and the word bit to refer to classical bits.

While in general we use two-level qubit systems to build quantum computers we can also choose other types of computing architectures. For example, we could build a QC with *qutrits* which are three-level systems. We can think of these as having states of 0, 1 or 2 or a superposition of these states.

The more general term for such a unit is *qudit*; qubits and qutrits are specific instances of qudits which can be computing units of any number of states. The Siddiqi Lab at UC Berkeley, for example, has designed a qutrit-based QC [34]. In a qutrit system we can represent more states than a qubit system with the same number of computational units.

A qubit system of say 100 qubits can handle 2^{100} states (1.26765E+30), while a qutrit system can handle 3^{100} states (5.15378E+47), a number which is 17 orders of magnitude larger. Put another way, to represent the same number space as a 100 qubit system, we only need ~ 63 qutrits ($\log_3(2^{100})$). Since it is more difficult to build qutrit systems, the mainstream QCs are currently based on qubits. Whether we choose qubits, qutrits or some other

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_3

qudit number, each of these systems can run any algorithm that the others can, i.e., they can simulate each other.[1]

In QM we represent states as vectors, operators as matrices and we use Dirac notation instead of traditional linear algebra symbols to represent vectors and other abstractions. Chapter 11 contains a review of linear algebra, Dirac notation and other mathematical tools that are crucial for our inquiry in this book. In this chapter we will assume knowledge of these mathematical tools; we encourage the reader to use the math chapters to review these concepts in the context of quantum computing.

Let us begin with the definition of a qubit:

3.1 What is a Qubit?

A physical qubit is a two-level quantum mechanical system. As we will see in the chapter on building quantum computers, there are many ways to construct a physical qubit. We can *represent* a qubit as a two-dimensional complex Hilbert space, $\mathbb{C}^2$. The *state* of the qubit at any given time can be represented by a vector in this complex Hilbert space.

The Hilbert space is equipped with the inner product which allows us to determine the relative position of two vectors representing qubit states. We denote the inner product of vectors $|u\rangle$, $|v\rangle$ as $\langle u|v\rangle$; this will equal 0 if $|u\rangle$ and $|v\rangle$ are orthogonal and 1 if $|u\rangle = |v\rangle$. To represent two or more qubits we can tensor product Hilbert spaces together to represent the combined states of the qubits. As we shall see, we have methods to represent separable states, where the qubits are independent of one another, and entangled states such as a Bell state, where we cannot separate the two qubit states.

We can represent the states $|0\rangle$ and $|1\rangle$ with vectors as shown below. We call these two the computational basis of a two-level system. We can then apply operators in the form of matrices to the vectors in the state space.

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

[1] Note that we could consider the same question in classical systems, i.e., we could have used a 3-state “trit” instead of the bit, but we choose to use bits as there are distinct advantages to the binary system.

3.2 Quantum Operators

In gate-based quantum computers, the operators which we use to evolve the state of the qubits are unitary and therefore reversible. Some of the operators are unitary, reversible *and* involutive (i.e., they are their own inverses); others are not involutive. A measurable quantity, or observable, is a Hermitian operator; thus the measurement in a quantum computer outputs real values from the system. We use the terms operators and gates interchangeably.

In addition to an inner product of two vectors, linear algebra gives us the outer product. This is when we take two vectors and form a matrix (whereas an inner product gives us a scalar). If we take the outer product $|0\rangle\langle 0|$, for example, we produce the following operator

$$|0\rangle\langle 0| = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

Similarly, we can take the outer product of the other three combinations to produce these matrices

$$|0\rangle\langle 1| = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

$$|1\rangle\langle 0| = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

$$|1\rangle\langle 1| = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

We can take the sum of two of these matrices to form a unitary matrix, like so

$$|0\rangle\langle 1| + |1\rangle\langle 0| = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{3.3}$$

This, in fact, is the X or NOT operator which we will encounter shortly in this chapter.

We have established that a qubit can be in one of the computational basis states of 0 or 1 or in a superposition of these two states. How can we represent the superposition of multiple states? We can do so as a linear combination of the computational bases of the state space.

3.4 Representing Superposition of States

We represent a superposition of states as the linear combination of computational bases of the state space. Each term in the superposition has a complex coefficient or *amplitude*.

Using the two computational basis vectors in the case of a single qubit, two examples of superpositions of states are

$$|+\rangle := \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$$

and

$$|-\rangle := \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right)$$

These two states differ by a minus sign on the $|1\rangle$ state. More formally, we call this difference a *relative phase*. The term phase has numerous meanings in physics — in this context, it refers to an angle. The minus sign is related to the angle π (180°) by Euler's identity[2]

$$e^{i\pi} = -1$$

Relative phases are of fundamental importance for quantum algorithms in that they allow for *constructive interference* and *destructive interference*. For example, if we evaluate the sum of the above states, we obtain

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

Here, we say that the amplitudes of the $|1\rangle$ state interfere *destructively* — the differing relative phases cause them to sum to zero. On the other hand, the amplitudes of the $|0\rangle$ state interfere *constructively* — they have the same sign (relative phase), so they do not sum to zero, and thus we are left with the state $|0\rangle$ as the result.

We can also consider subtracting the two superposition states. We leave it to the reader to verify that

$$\frac{1}{\sqrt{2}}(|-\rangle - |+\rangle) = -|1\rangle$$

[2] For more on Euler's identity, refer to chapter 13.

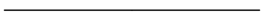
Here, the amplitudes of the $|0\rangle$ state interfere destructively while the amplitudes of the $|1\rangle$ state interfere constructively. In this example, we do not end up with the $|1\rangle$ state exactly — it is multiplied by a minus sign. As we saw above, we can interpret this minus sign as an angle (or phase) $e^{i\pi}$. Here, it is applied to the entire state, not just one term in the superposition. We refer to this type of phase as a *global phase*.

While it is true that the $-|1\rangle$ state is not exactly the $|1\rangle$ state, we will see in future chapters that a global phase change has no impact on quantum measurements. That is, the measurement statistics obtained by measuring the $-|1\rangle$ state and the $|1\rangle$ state are exactly identical. In this case, we often say that the two states are *equal, up to global phase*.

Quantum Circuit Diagrams

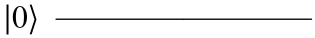
We use circuit diagrams to depict quantum circuits. We construct and read these diagrams from left to right; we can think of circuit diagrams like a staff of music which we read in the same direction. Barenco et al. set forth a number of the foundational operators that we use today in QC [22]. Fredkin and Toffoli [217, 89] added to this set with two ternary operators.

We begin the construction of a quantum circuit diagram with the circuit wire which we represent as a line

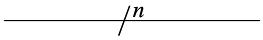


A line with no operator on it implies that the qubit remains in the state in which it was previously prepared. This means that we are relying on the quantum computer to maintain the state of the qubit.

We denote the initial prepared state with a ket and label on the left of the wire

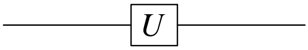


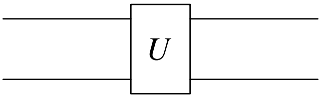
We denote n number of qubits prepared in that state with a slash n symbol across the wire.



3.1 *Quantum Operators*

Let us now turn to the set of commonly used quantum operators. We denote a single-qubit operator with a box containing the letter representing that operator straddling the line. We denote a binary gate with an operator box spanning two quantum wires and spanning three wires for a ternary operator, etc. Note that we could have chosen a different set of operators to accomplish universal quantum computation; the set of operators chosen is arbitrary and is sufficient as long as it meets the test of universality which we will cover later in this chapter. Here are representations for unary and binary operators





Unary Operators

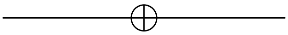
Let us now cover the set of one-qubit, or unary, quantum operators. The first three operators we will examine are the Pauli matrices. These three matrices along with the identity matrix and all of their ± 1 and $\pm i$ multiples constitute what is known as the *Pauli group*. First, we have X, which is the *NOT* operator (also known as the bit flip operator and can be referred to as σ_x)

$$X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

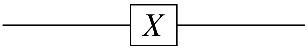
If we apply X to $|0\rangle$ then we have

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0+0 \\ 1+0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

We can represent the initial state of a qubit and the operators we apply to them with a circuit diagram. We use the following symbol to represent the X operator in circuit diagrams



This is different from the convention of the operator name in a box, which one may also encounter in circuit diagrams



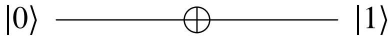
As we have seen, we can represent the X operator in ket notation as

$$X := |0\rangle\langle 1| + |1\rangle\langle 0|$$

and the application of the X operator like so:

$$X\,|j\rangle = |j \oplus 1\rangle$$

where $j \in \{0, 1\}$. Here the $\oplus$ operation denotes addition modulo-2, and $j \oplus 1$ is equivalent to the *NOT* operation. So if we start with the qubit in state $|0\rangle$ and apply *NOT* then we have



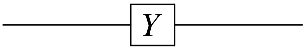
Next we have the Y operator, also denoted σ_y, which rotates the state vector about the y axis[3].

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

So that if we apply it to the $|1\rangle$ state we have

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 - i \\ 0 + 0 \end{pmatrix} = \begin{pmatrix} -i \\ 0 \end{pmatrix} = -i\,|0\rangle$$

The circuit diagram for the Y operator is



And the Z operator, also denoted σ_z, which rotates the state vector about the z axis (also called the phase flip operator since it flips it by π radians or 180 degrees)

$$Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

If we apply Z to the computational basis state we have

$$Z\,|j\rangle = (-1)^j\,|j\rangle$$

or to show this in matrix form for the special case $j = 0$

[3] The x, y and z axes in this section refer to representation of the qubit's state on a Bloch sphere, which we will cover later in this chapter.

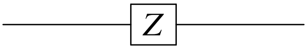
$$\begin{pmatrix}1 & 0\\0 & -1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1+0\\0+0\end{pmatrix} = \begin{pmatrix}1\\0\end{pmatrix} = (-1)^0\,|0\rangle = |0\rangle$$

For the case where $j = 1$ we have

$$\begin{pmatrix}1 & 0\\0 & -1\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0+0\\0-1\end{pmatrix} = \begin{pmatrix}0\\-1\end{pmatrix} = (-1)^1\,|1\rangle = -\,|1\rangle$$

Note that we can multiply the bit-flip operator X by the phase-flip operator Z to yield the Y operator with a global phase shift of i. That is, $Y = iXZ$.

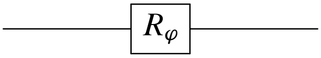
The circuit diagram for the Z operator is



Next we turn to the more general phase shift operator. When we apply this operator we leave the state $|0\rangle$ as is and we take the state $|1\rangle$ and rotate it by the angle (or phase) denoted by φ, as specified in the matrix

$$R_\varphi := \begin{pmatrix}1 & 0\\0 & e^{i\varphi}\end{pmatrix}$$

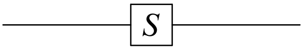
So the Pauli Z operator is just a special case of R_φ where $\varphi = \pi$. Let's recall that $e^{i\pi} = -1$ by Euler's identity (see chapter 13) so we can replace $e^{i\pi}$ with -1 in the Z matrix. The circuit diagram for the R operator is



Let's discuss two additional phase shift operators that are special cases of the R_φ matrix. First, the S operator, where $\varphi = \pi/2$

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

The S operator thus rotates the state about the z-axis by 90°. The circuit diagram for the S operator is

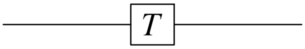


Next let's turn to the T operator which rotates the state about the z-axis by 45°. If we give φ the value of $\pi/4$ then[4]

[4] Note that the T gate is also known as the $\pi/8$ gate, since if we factor out $e^{i\pi/8}$, the diagonal components each have $|\varphi| = \pi/8$, but this is of course the same operator.

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

Note that $S = T^2$. In other words, if we apply the T matrix to the vector representing the state and then apply T again to the resulting vector from the first operation we have accomplished the same result as applying S once (45° + 45° = 90°). The circuit diagram for the T operator is



Now let's turn to the Hadamard operator. **This operator is crucial in quantum computing since it enables us to take a qubit from a definite computational basis state into a superposition of two states.** The Hadamard matrix is

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

It was actually the mathematician John Sylvester who developed this matrix, but we name it after Jacques Hadamard (see Stigler's law of eponymy which, of course, was probably conceived by Merton and others). The circuit diagram for the H operator is

If we apply the Hadamard to state $|0\rangle$ we obtain

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

And to state $|1\rangle$ we have

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

So we can see that the H operator takes a computational basis state and projects it into a superposition of states $(|0\rangle + |1\rangle)/\sqrt{2}$ or $(|0\rangle - |1\rangle)/\sqrt{2}$, depending on the initial state.

What is the $\sqrt{2}$ doing in this state? Let us recall the Born rule that the square of the modulus of the amplitudes of a quantum state is the probability of that state. Furthermore, for all amplitudes α, β, etc. of a state

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

That is, the probabilities must sum to one since one of the states will emerge from the measurement.

Before moving on to the binary operators, let us define the identity operator and then determine which operators can be expressed as sequences of other operators. The identity operator is simply the matrix which maintains the current state of the qubit. So for one qubit we can use

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

Having covered the set of unary operators, we can show the following identities:

$$\begin{aligned} HXH &= Z \\ HZH &= X \\ HYH &= -Y \\ H^{\dagger} &= H \\ H^2 &= I \end{aligned}$$

Please see chapter 14 for a list of additional identities.

Binary Operators

Let us now consider two qubit, or *binary*, operators. In a two-qubit system, by convention, we use the following computational basis states:

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

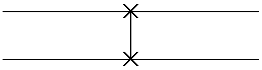
Let us first discuss the *SWAP* operator. The *SWAP* takes the state $|01\rangle$ to $|10\rangle$ and, of course, $|10\rangle$ to $|01\rangle$. We can represent this operator with the following matrix

$$SWAP := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

And apply it to a 4-d vector representing the state $|01\rangle$ as follows

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0+0+0+0 \\ 0+0+0+0 \\ 0+1+0+0 \\ 0+0+0+0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = |10\rangle$$

Satisfy yourself that this operator applied to one of the two-qubit computational basis vectors will have the desired result. For the circuit diagram of the *SWAP* operator we use



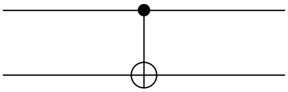
Now we come to a critical operator for quantum computing — controlled-*NOT* (*CNOT*). In this binary operator, we identify the first qubit as the *control qubit* and the second as the *target qubit*. If the control qubit is in state $|0\rangle$ then we do nothing to the target qubit. If, however, the control qubit is in state $|1\rangle$ then we apply the *NOT* operator (X) to the target qubit. **We use the *CNOT* gate to entangle two qubits in the QC**. We can represent *CNOT* with the following matrix

$$CNOT := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

So, for example, we compute the action of *CNOT* on the state $|10\rangle$ as follows

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0+0+0+0 \\ 0+0+0+0 \\ 0+0+0+0 \\ 0+0+1+0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |11\rangle$$

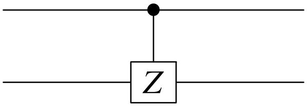
And for the circuit diagram, we depict the *CNOT* in this way



Here is an identity connecting the *SWAP* and *CNOT* operators:

$$SWAP_{ij} = CNOT_{ij}CNOT_{ji}CNOT_{ij}$$

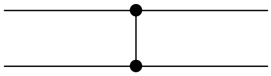
Now let's turn to another control operator: *CZ*. Here we have a control qubit and a target qubit just as with *CNOT*; however, in this operation if the control qubit is in state $|1\rangle$ then we will apply the Z operator to the target qubit. We can represent the *CZ* operator in a circuit diagram as



and as a matrix

$$CZ := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

We can represent the *CZ* operator in circuit diagrams as



Note that unlike the *CNOT* gate, the *CZ* gate is symmetric: we can choose either qubit as the control or the target and we will end up with the same result. This is why we can represent the *CZ* gate with a dot on both circuit wires.

Ternary Operators

We have discussed both unary and binary operators. Now let's consider the ternary or 3-qubit operators. First, we have the Toffoli operator, also known as the *CCNOT* gate [217]. Just as in the *CNOT* operator, we have control and target qubits. In this case, the first two qubits are control and the third is the target qubit. Both control qubits have to be in state $|1\rangle$ for us to modify the target qubit. Another way of thinking about this is that the first two qubits (x and y) have to satisfy the Boolean *AND* function — if that equals TRUE then we apply *NOT* to the target qubit, z. We can represent this action as

$$(x, y, z) \mapsto (x, y, (z \oplus xy))$$

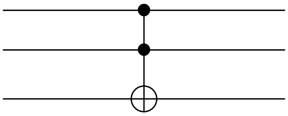
or, as a matrix,

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

As an example, we apply this gate to the state $|110\rangle$

$$\begin{pmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{pmatrix}\begin{pmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{pmatrix}=\begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{pmatrix}=|111\rangle$$

In circuit diagrams, we use the following to denote the Toffoli



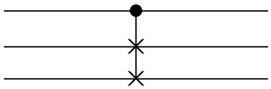
Next, let's consider the Fredkin gate, also known as the *CSWAP* gate [89]. When we apply this operator, the first qubit is the control and the other two are the target qubits. If the first qubit is in state $|0\rangle$ we do nothing and if it is in state $|1\rangle$ then we *SWAP* the other two qubits with each other. The matrix representing this operations is

$$\begin{pmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\end{pmatrix}$$

For example, the Fredkin gate applied to $|110\rangle$ gives

$$\begin{pmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\end{pmatrix}\begin{pmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{pmatrix}=\begin{pmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{pmatrix}=|101\rangle$$

In circuit diagrams we use this symbol for the Fredkin operator



3.2 Comparison with Classical Gates

In classical computing we have a set of commonly used gates: *AND*, *NOT*, *OR*, *NAND*, *XOR*, *FANOUT*, etc. We can use combinations of these gates to perform any computation in classical computing. A classical computer that can run these gates is Turing-complete or universal. In fact, we can prove that the *NAND* gate alone is sufficient to construct all other classical operators [198]. We can construct classical circuits with these basic building blocks such as a circuit for the half-adder

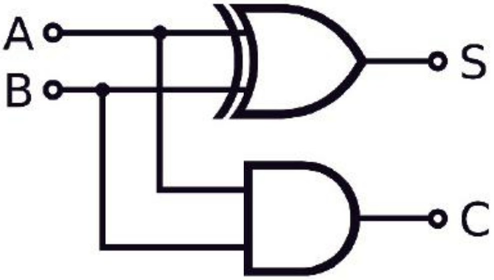


Figure 3.1: Half-adder in classical computing Source: Wikimedia

We can then build a full-adder from those elements:

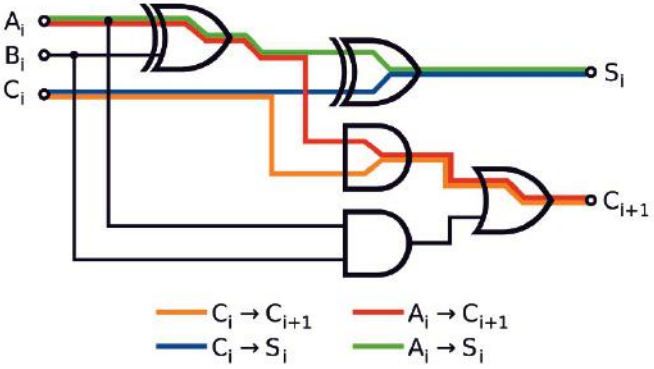


Figure 3.2: Full-adder in classical computing Source: Wikimedia

Neither *AND*, *OR*, *XOR*, *NAND* or *FANOUT* can be used in quantum computing. The *AND*, *OR*, *XOR* and *NAND* gates are not reversible. The

FANOUT gate would not be allowed in quantum computing since it involves the duplication, or cloning, of a state; this would violate the no-cloning theorem. Of the primary classical gates, only the *NOT* operator can be used in the quantum computing regime as it is reversible and does not involve cloning.

3.3 *Universality of Quantum Operators*

If *NAND* is universal for classical computing, is there such a gate or set of gates that are universal for quantum computing? In fact, there are several combinations of unary and binary operators that lead to universality. No set of unary gates on their own can achieve universal QC. Two of the gate sets that yield universality are:

1. The Toffoli gate is universal for QC when paired with a basis-changing unary operator with real coefficients (such as H) [199].
2. Another set of gates which is universal is $\{CNOT, T, H\}$ [44, 161].

3.4 *Gottesman-Knill and Solovay-Kitaev*

The Gottesman-Knill theorem states that circuits built with only Clifford gates can be simulated efficiently on classical computers assuming the following conditions:

- state preparation in the computational basis
- measurements in the standard basis
- any classical control conditioned on the measurement outcomes

The Clifford group of operators is generated by the set $C = \{CNOT, S, H\}$ [96] [168].

A further theorem that is worth considering at this junction is that of Solovay-Kitaev. This theorem states that if a set of single-qubit quantum gates generates a dense subset of $SU(2)$, which is the special unitary group of unitary matrices which are 2 x 2, then that set is guaranteed to fill $SU(2)$ quickly, i.e., it is possible to obtain good approximations to any desired gate using surprisingly short sequences of gates from the given generating set [62]. The theorem generalizes to multi-qubit gates and for operators from SU(d) [62].

A simplified version of this statement is that all finite universal gate sets can simulate a given gate set to a degree δ of precision. More precisely, if L is the size of the circuit (i.e., the number of gates) then the approximation L' of

L has a bounded number of gates; this can be specified in big-O notation by

$$L' = O\left(L \log^4\left(\frac{L}{\delta}\right)\right)$$

If D denotes the depth of the circuit, i.e., the number of computational steps, then the approximation D' of D has a bounded depth specified in big-O notation by

$$D' = O\left(L \log^4\left(\frac{D}{\delta}\right)\right)$$

So, these expressions demonstrate that the simulation is quite efficient and better than polynomial time.

3.5 The Bloch Sphere

There are several ways to represent the state of a qubit:

1. We can write out the state in Dirac notation. For example, if we have a qubit that is prepared in state $|0\rangle$ and then apply the X operator, we will then find the qubit in state $|1\rangle$ (assuming no outside noise)

$$X\,|0\rangle \rightarrow |1\rangle$$

2. We can use the Bloch sphere to represent the state of a single qubit. Any state in a quantum computation can be represented as a vector that begins at the origin and terminates on the surface of the unit Bloch sphere. By applying unitary operators to the state vectors, we can move the state around the sphere. We take as convention that the two antipodes of the sphere are $|0\rangle$ on the top of the sphere and $|1\rangle$ on the bottom.

As we can see in Figure 3.3, one of the advantages of visualization with the Bloch sphere is that we can represent superposition states such as

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

as we see at the X axis. We can also differentiate between states that contain different phases as is shown in the states along the X and Y axes.

Let us return to computational universality which we treated above. Now that we have introduced the Bloch sphere, another way to think about a set of gates that satisfies universal computation is one which enables us to reach any point on the Bloch sphere.

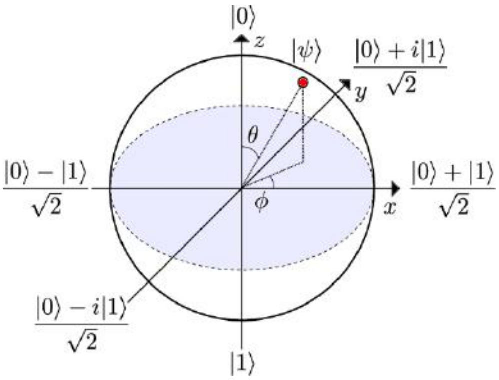

$|0\rangle$
z
$|\psi\rangle$
$\frac{|0\rangle + i|1\rangle}{\sqrt{2}}$
y
θ
$\frac{|0\rangle - |1\rangle}{\sqrt{2}}$
ϕ
x
$\frac{|0\rangle + |1\rangle}{\sqrt{2}}$
$\frac{|0\rangle - i|1\rangle}{\sqrt{2}}$
$|1\rangle$

Figure 3.3: The Bloch sphere Source: [93]

For interactive visualizations of qubits on the Bloch sphere, see the book's online website. Now that we have covered the main unitary operators which we use in QC, let's turn to the measurement of the QC's state.

3.6 *The Measurement Postulate*

Measurement in classical physics is a seemingly straightforward process. The act of measurement is assumed to have no effect on the item that we are measuring. Furthermore, we have the ability to measure one property of a system, get a reading, then measure another property and be confident that the first property measured still retains its observed value. Not so in quantum mechanics; in this regime, the act of measurement has a profound effect on the observation.

Building on the principles of quantum mechanics, we can state the measurement postulate as:

3.5 Measurement Postulate

Every measurable physical quantity, o, is described by a corresponding Hermitian operator, O, acting on the state Ψ.

According to this postulate, there exists a Hermitian operator, which we call an observable, associated with each property. So, for example, the observable $\hat{x}$ is associated with the position of a particle.

We recall that a Hermitian operator is equal to its adjoint (which is its complex conjugate transpose). If O is Hermitian then we can state that $O = O^\dagger$ (see chapter 12 for more discussion on Hermitians).

Hermitian operators have the desirable property that their eigenvalues are guaranteed to be real numbers. When measuring a physical system for properties such as momentum or position we need to specify a real number.

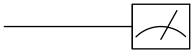
Each possible outcome of the measurement is an eigenvalue, λ, of the observable and is characterized by $|P\,|\psi\rangle\,|^2$ where P is the projector onto the eigenspace of the observable. Since we normalize the vectors, the state after measurement is represented by a unit eigenvector of the observable with eigenvalue λ_i.

We discussed earlier how a system can be in a superposition of two or more states. We can represent this as a linear combination of the orthonormal basis vectors. For example,

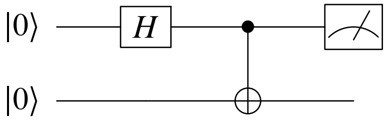
$$|\Psi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$$

This leads to the question of how we can set up the measurement in such a way as to obtain the output we require. This in turn is dependent on the amplitudes of each state since, as discussed previously, the square of the modulus of the amplitude is the probability of that state appearing as the output upon measurement (Born's rule).

We can represent a measurement in a quantum circuit like this:



Now that we have our unitary and measurement operators, we will construct some basic quantum circuits. Let's figure out what this one does:



We start with two qubits, let's call them $q0$ and $q1$, each prepared in state $|0\rangle$. We then apply the Hadamard operator to $q0$ which puts it into the superposition of states

$$|q0\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$$

We then apply a *CNOT* across $q0$ and $q1$. This entangles the two qubits so we now have the combined, non-separable state of the two qubits of

$$\frac{1}{\sqrt{2}}\,|00\rangle + \frac{1}{\sqrt{2}}\,|11\rangle$$

We have thus created a Bell state, or an EPR pair. We then measure $q0$ with a 50/50 chance of finding a 0 or a 1 value for the real-valued output.

3.7 *Computation-in-Place*

We can depict the circuit in a 3-dimensional manner as in Figure 3.4. Here you can see the qubits beginning in a prepared state followed by a Hadamard step to put them all into a superposition; then they undergo a series of one and two-qubit operations such as X, Y, T and CZ. Measurement is then applied in the final step.

In this diagram, we see a crucial difference between classical and quantum computing.

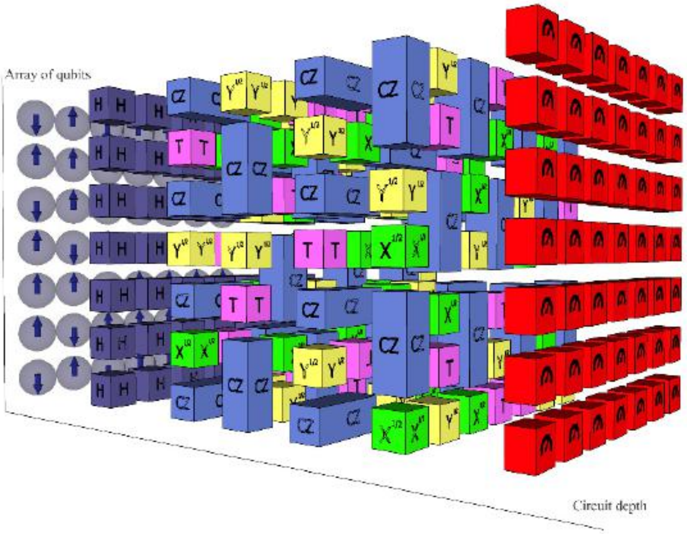


Figure 3.4: 3-D Quantum circuit diagram Source: Google

3.6 Computation-in-Place

In most forms of gate-based quantum computing, the information is represented in the states of the qubits as they evolve over time with the successive application of unitary operators.

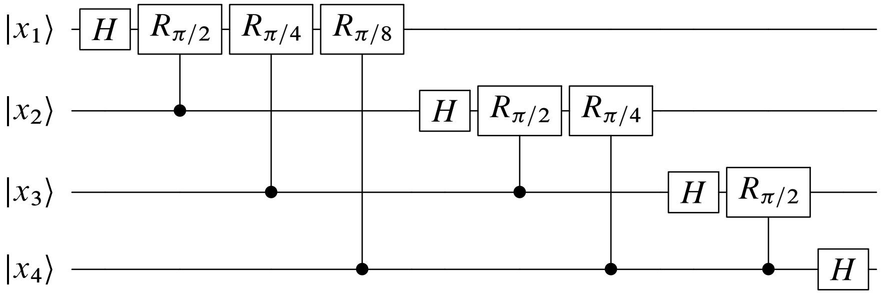
Computation-in-place is in stark contrast to classical computing, where we shuttle data around the processor to various memory and calculation registers. In most forms of quantum computers, all processing takes place on the qubits themselves. After measurement in a QC we output real-valued bits which we can share with the CPU that is controlling the quantum processor and, if necessary, incorporate in further processing on classical machines.

We now come to one of the key questions in quantum computing: if measurement is based on the modulus squared of the amplitude of each qubit state, then how can we pre-determine which of the qubits will be the output? Deutsch, Jozsa, Bernstein, Vazirani, Shor and others realized that we can influence the output by setting up the amplitudes prior to measurement to favor the output we need for that computational task.

One method for accomplishing this goal is the quantum Fourier transform (QFT – not to be confused with quantum field theory!). By applying a QFT across all qubits prior to measurement, we can obtain phase information upon measurement instead of amplitude information.

The QFT is an efficient process on a quantum computer: the discrete Fourier transform on 2^n amplitudes only takes $O(n^2)$ applications of Hadamard and phase-shift operators (where n is the number of qubits). We will cover the QFT in greater detail later in this text.

Here we display the circuit for QFT for $n = 4$:



Before we turn our attention to quantum hardware, let's delve into computational complexity in the next chapter. This will give us the foundation to understand which sorts of problems are appropriate for a quantum computer.

> *If you take just one piece of information from this blog: Quantum computers would not solve hard search problems instantaneously by simply trying all the possible solutions at once.*
>
> —Scott Aaronson

CHAPTER 4

Complexity Theory

Since quantum computing offers an alternative approach to computation, it is logical to consider which classes of problems are now tractable in this new regime that were not thought to be tractable in a classical framework. To do so, let's consider a range of problem classes.

4.1 *Problems vs. Algorithms*

Let us first clarify the difference between computational problems (or tasks), algorithms and programs. Here is an example of a *computational problem*:

> Given a data set of n numbers, sort the numbers in increasing numerical order.

We can then analyze several different *algorithms* to solve this problem: quicksort, merge sort, insertion sort and others. An algorithm is a hardware-independent *method* of solving a computational problem. We generally try to find algorithms that can solve a problem efficiently. A program is a particular implementation of an algorithm in a given coding language.

Analysis of algorithms is the study of the resources that an algorithm needs to run.[1] We can bucket algorithms into different computational orders both in time (number of steps) and space (amount of memory). Computational complexity theory, by contrast, is the study of classes of problems; we will define a number of important problem classes below.

[1] Note: It is common to refer to the analysis of the time and memory resource requirements of an algorithm as the time and space complexity of the algorithm. Although the strict definition of "complexity" would only apply to *computational problems*, the usage is sufficiently widespread that we will occasionally use complexity in the context of algorithms as well.

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_4

When defining complexity classes we focus on decision problems; these are mathematical problems that can be answered with a binary yes/no response given an input. Examples of decision problems include:

Given x, is x a prime number?
Given x and y, does y divide x evenly?

Complexity analysis for a decision problem determines the computational resources needed to deliver an answer. We generally look at the characteristics of the worst-case scenario for this determination.

4.2 *Time Complexity*

As we discussed in an earlier chapter, we can use big-O notation to denote the upper bound of the worst case of a problem. For example, if we have a series of items that we want to sort, the big-O time complexity will depend on which algorithm we choose to sort the items. Here are some complexity orders of common sorting algorithms:

- Insertion sort: $O(n^2)$
- Mergesort: $O(n\log(n))$
- Timsort: $O(n\log(n))$

These differences can be significant if n is large. When analyzing algorithms for the computational resources needed to run them, we are performing *asymptotic analysis*; in other words, what resources will be needed as the input, n, gets very large. Refer to Figure 4.1 for a plot of common big-O time computational orders.

In addition to big-O notation which designates the upper bound on the worst case computational order for that algorithm, we can consider big-Ω (Omega) notation which designates the lower bound of the worst case computational order. Adding now to the list above we can compare lower and upper bounds for the computational resources needed for these sorting algorithms:

- Insertion sort: $\Omega(n)$, $O(n^2)$
- Mergesort: $\Omega(n\log(n))$, $O(n\log(n))$
- Timsort: $\Omega(n)$, $O(n\log(n))$

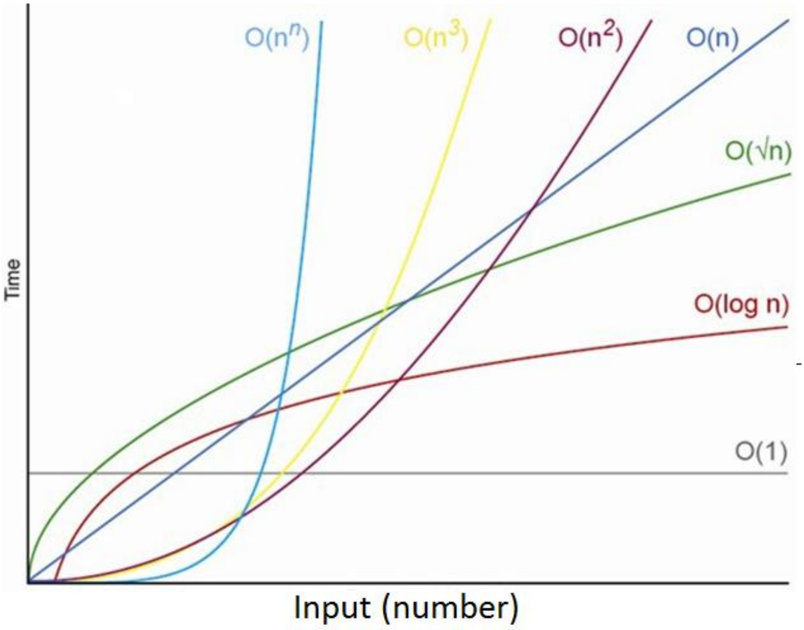


Figure 4.1: Big-O time complexity chart Source: [15]

We see that the lower bound for insertion sort and timsort are significantly better than their upper bound. However, since we have to plan for the upper bound, we generally focus on the big-O complexity.

There is a third metric we use for cases in which the upper bound (big-O) and the lower bound (big-Ω) of the worst-case match. For these algorithms we can describe their big-Θ computational order, which is both their big-O and their big-Ω since they match.

To use more formal notation, we say that a function $f(n)$ (which could represent the time necessary to run an algorithm for an input of size n) is *on the order of* a function $g(n)$ (which could represent the time necessary to run a different algorithm given n inputs) if and only if the limit superior of the absolute value of the quotient of $f(n)$ and $g(n)$ is finite as n tends to infinity.

In symbols

$$f(n) = O(g(n)) \text{ if and only if } \limsup_{n\to\infty} \left| \frac{f(n)}{g(n)} \right| < \infty$$

In this expression, *lim sup* is the limit superior or supremum limit. Following Hardy and Littlewood [102] and as described by Knuth [121], we state that

$$f(n) = \Omega(g(n)) \text{ iff } g(n) = O(f(n))$$

and

$$f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

There are two more notations called little-o and little-ω. Little-o provides a strict upper bound (equality condition is removed from big-O) and little-ω provides a strict lower bound (equality condition removed from big-Ω).

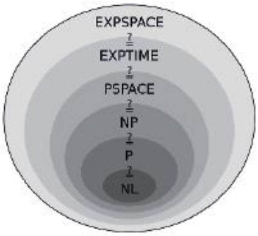
Let's also introduce the concept of *completeness*. A problem, G, is said to be H-complete if it is a part of class H and we can prove that all problems in class H can be reduced to G. In other words, if we have a subroutine, S, which when run can solve for problem G, and this subroutine can also solve for any problem in H, then we can say that all problems in H can be reduced to G and that G is H-complete.

4.3 *Complexity Classes*

Let us first define a number of common complexity classes from classical computing:

P – Polynomial time: problems that can be solved in polynomial time. In other words, the problem can be solved in a reasonable amount of time on a classical computer. Problems of $O(1)$, $O(\log(n))$, $O(n)$, $O(n\log(n))$, and $O(n^2)$, as examples, are in class P. Although $O(\log(n))$ doesn't look polynomial since $\log(n)$ is not a polynomial in n, $O(\log(n))$ is in P since is it upper-bounded by $O(n^2)$, which is a polynomial.

NP – Non-deterministic polynomial time: a problem is in NP if whenever the answer is "yes," there's a polynomial-size witness or proof for the yes-answer, which a polynomial-time algorithm can verify. We can imagine a Turing machine which can switch to a state that is not determined by its previous state. If through such a move it happens upon a correct solution to a problem, this solution can be verified in polynomial time.



PSPACE and related classes
Source: Wikimedia

A problem, A, is said to be **NP-Complete** iff[2]: (1) A is in NP, and (2) all NP problems are polynomial-time reducible to A. If we only know (2), and not necessarily (1), we say A is **NP-hard**.

[2] Note: we often abbreviate *if and only if* as iff throughout the book.

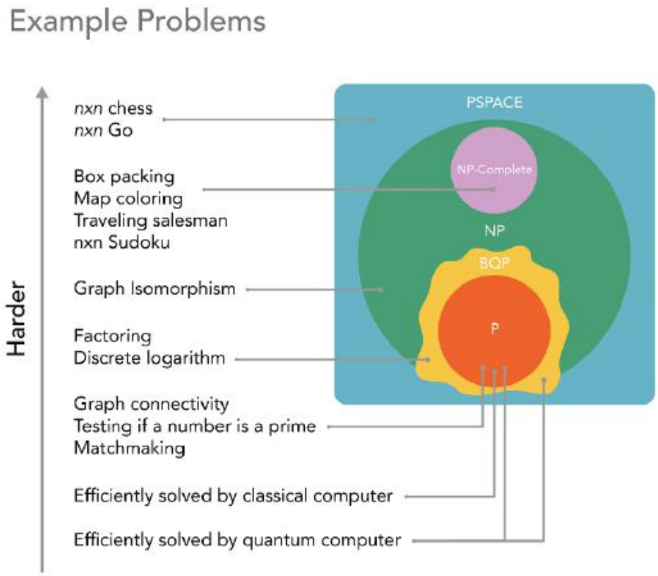


Figure 4.2: Complexity classes - note: it is not yet proven whether graph isomorphism is in P or not Source: [2]

One of the seven Clay Mathematics Institute Millennium Problems is the question of whether the complexity class P is equivalent to the class NP. This $1 million challenge is the most recently developed of all the Clay prizes and is most closely tied to computer science; this question is still the subject of active research.

PSPACE – Polynomial space: This class focuses on memory resources, not time. PSPACE is the class of decision problems that are solvable by some algorithm whose total space usage, in all instances, can be upper-bounded by a polynomial in the instance size. See Figure 4.2 for the position of PSPACE in the context of other classes.

BPP – Bounded-error probabilistic polynomial time: BPP is a class that contains P; many believe that BPP = P, but we have not been able to prove that yet [3, Lecture 4]. BPP is the class of decision problems for which there exists a polynomial-time randomized algorithm that solves every instance with a success probability of at least $2/3$ (over the choice of random bits). The core idea of BPP is that sometimes randomized algorithms give us faster time results than a deterministic algorithm trying to achieve the same goal. Problems that are in BPP either have a deterministic algorithm that can run in polynomial time or have a probabilistic algorithm which will give the wrong answer to a decision problem no worse than $1/3$ of the time.

The bound of $1/3$ is not fixed. We can choose any lower bound between 0 and below $1/2$ and BPP will not change. The reason is that when run many times the probability of producing a wrong error each time is low. This is described by the central limit theorem.

Now let's consider a range of complexity classes that arise with quantum computing:

BQP – Bounded-error quantum polynomial time: BQP is the quantum analogue of the class BPP for classical computation. A decision problem is in BQP if it can run in polynomial time and yields a correct result with a high probability. BQP is the primary complexity class we focus on for QC.

As you can see from Figure 4.2, BQP contains problems which are thought to be intractable in the classical regime but are thought to be tractable in bounded-error polynomial time for a quantum computer. We say "thought to be" as it is still not yet proven whether the computational problem of factoring large numbers, for example, is in P or NP. Although we have no classical algorithm at the moment that can factor large numbers, this does not mean such an algorithm does not exist. Thus, complexity theorists are not sure of the exact position of BQP with respect to P, NP and PSPACE [161].

EQP – Exact quantum polynomial time: this is the set of decision problems solvable by a quantum circuit that yields the correct answer with a probability of 1. In other words, this class is the same as BQP except that it must give the correct answer with probability of 1 instead of having some bounded error margin [6].

Note that a QC does not render all NP problems tractable; only problems that have some structure we can exploit can be handled efficiently by a QC. For example, Shor's algorithm takes advantage of the periodicity of the function which then enables us to solve the equivalent problem of factoring a large number.

QMA – Quantum Merlin-Arthur: QMA is the quantum analogue to the non-probabilistic class MA. In a Merlin-Arthur (MA) problem, a prover (Merlin) sends a message to a verifier (Arthur). In the classical complexity class MA, Arthur can verify the message in polynomial time. Problems that live in QMA are ones with the following characteristic [37]:

- If the answer is Yes, then the verifier can verify with probability greater than $2/3$ using a proof that can run in polynomial time on a quantum computer.
- If the answer is No, then the verifier can reject the proof with a wrong outcome in no more than $1/3$ of the cases.

Table of Classical and Quantum Complexity Classes[3]

Classical	Quantum
P	EQP
BPP	BQP
NP	QMA

4.4 *Quantum Computing and the Church-Turing Thesis*

Let us now turn to the relationship between quantum computing and the Church-Turing thesis. Alonzo Church and Alan Turing developed the initial conjecture that:

4.1 Church-Turing Thesis (CTT)

If an algorithm can be performed on any piece of hardware (say, a modern personal computer), then there is an equivalent algorithm for a Universal Turing Machine (UTM) which performs exactly the same algorithm [161, p. 5].

This conjecture was then updated take into account the *efficiency* of the algorithm. Algorithmic efficiency refers to the quantification of a particular resource that is used in running the algorithm. It is important in this analysis to maintain consistency when comparing the efficiency of a set of algorithms; if we are analyzing the number of steps of one algorithm, we should do so for all others in the comparison and not switch to the analysis of memory resources, for example, midway through the analysis. The additional requirement of running an algorithm *efficiently* gives us the Strong Church-Turing Thesis (SCTT):

[3] Readers interested in this subject should consult the complexity zoo [6].

4.2 Strong Church-Turing Thesis (SCTT)

Any algorithmic process can be simulated efficiently using a Universal Turing Machine (UTM) [161, p. 5].

Researchers then realized that there were probably counterexamples to the SCTT, namely, algorithms that made use of randomness. For example, Solovay and Strassen demonstrated that an algorithm using randomness could test for primality of a number [208]. By repeating the algorithm a finite number of times, one could obtain a correct answer with almost near certainty.

It was later shown that there is an efficient polynomial-time algorithm for testing primality [8], but historically, it was the initial insight that an algorithm using randomness could get the task done that led to the refinement of the SCTT. The SCTT was thus updated to give us the Extended CTT (ECTT):

4.3 Extended Church-Turing Thesis (ECTT)

Any algorithmic process can be simulated efficiently using a Probabilistic Turing Machine (PTM) [161, p. 6].

Enter quantum computation. QC challenges the ECTT by demonstrating that it can solve certain problems exponentially faster than a classical computer. This undermines the ECTT assertion that any UTM or even PTM can simulate any other computing device in running any given algorithm [10]. We now arrive at the quantum version of the ECTT, which still stands:

4.4 Quantum Extended Church-Turing Thesis (QECTT)

Any realistic physical computing device can be efficiently simulated by a fault-tolerant quantum computer.

Now that we have a foundation of quantum operators, circuits and complexity classes, let us explore how to build a physical quantum computer.

Part II

Hardware and Applications

CHAPTER 5

Building a Quantum Computer

Now that we have covered the essential workings of a quantum computer, let us discuss how we can physically realize these devices. There are many different architectures and designs of gate-based quantum computers each with its own pros and cons. In this chapter we will cover the leading paradigms of quantum computational hardware. Check the book's online site for updates as the technology is changing rapidly.[1]

Each of the architectures below require a set of classical computers for control of the system. As you can see in Figure 5.1, a superconducting qubit QC, for example, is controlled by a traditional computer. We develop our quantum circuit protocols in high-level languages on the classical computer which can then manipulate the quantum system to apply the operators in the circuit. Upon measurement, the output of a QC is *classical* information which is fed back to the classical computer for readout or further processing.

We can think of the QC portion of a computation as a subroutine in a much larger computation, much of which may take place in the classical regime. In the application of Shor's algorithm, for example, many tasks are performed by the classical computer and then the hard part – the implementation of the period-finding algorithm (which is equivalent to prime number factoring) – is sent out as a subroutine to the QC. This output is then integrated back into the classical platform.

5.1 Assessing a Quantum Computer

There are many advances occurring in quantum hardware. Here is a useful checklist to analyze the potential impact of engineering progress in this field:

[1] Here is the online site: https://github.com/jackhidary/quantumcomputingbook

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_5

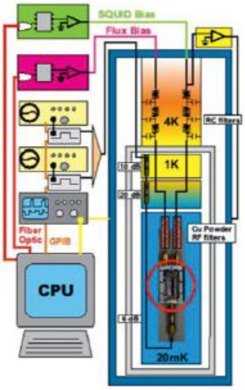


Figure 5.1: Classical CPU controlling a superconducting quantum computer Source: [169]

1. **Universality**: The first question to ask about a hardware platform that is presented as a quantum computer is whether it is Turing-complete, or universal. The device may be a non-universal annealer, for instance. We can turn to the DiVincenzo criteria we described in chapter 2 to aid us in this test. For example, DiVincenzo calls for the qubits to be individually addressable in a quantum computer. If we have, for example, a two-level system comprised of an ensemble of atoms, but where the qubits are not individually addressable, this system would fail the QC test.
2. **Fidelity**: While it is tempting to focus on the horse race of the number of qubits of each platform, we recommend first examining the claims made on the fidelity of those qubits. Fidelity is a measure of the ability for a qubit to remain in coherence through a computation; specifically it is calculated as: 1 - the error rate. When presented with a system, it is useful to examine the fidelity of the qubits under one and two-qubit operations. Fidelity is harder to maintain when spanning two qubits with a *CNOT*, for example, than when applying single-qubit operators such as X or Y.
3. **Scalability**: Is the architecture scalable to 10^6 qubits and beyond? While it is of some benefit to create NISQ-regime QCs across a multitude of hardware frameworks at this stage, it is also important to examine the ability to achieve a fault-tolerant platform in that architecture.
4. **Qubits**: Once the above questions are considered, we can turn to numbers of qubits. Maintaining the simultaneous coherence of larger and

larger numbers of qubits is a technical challenge. Also, it is important to look at architecture-specific limitations of the qubits, such as nearest-neighbor connections. In some platforms, for example, if we operate on two adjacent qubits, we cannot make use of other adjacent qubits at the same time due to potential crosstalk.

5. **Circuit depth**: This refers to how many operations we can implement before coherence breaks down. A 100,000 qubit computer would be wonderful, but if it cannot implement more than a few operations before losing coherence it is of limited value.
6. **Logical connectivity**: Can we implement two-qubit gates on any pair of qubits, or only for certain pairs? Limited logical connectivity requires that logical *SWAP* operations be inserted into our algorithms to effectively simulate greater connectivity. More operators means more potential for noise and errors.
7. **Cloud access**: Is the hardware easily made available over the cloud? It is unlikely that most organizations will purchase or build their own QCs. Instead, they will rely on a range of academic and commercial providers who will provide cloud access. Criteria to consider in this category include reset time between computations and labor required to keep the platform available to meet its service level agreement (SLA).

With that introduction, let's consider the leading QC architectures in alphabetical order. Please consult this book's online site for links to additional papers and updates as they are posted. For additional background on the approaches outlined in this chapter, please see [126].

5.2 *Neutral Atom*

Neutral atoms present an intriguing approach to quantum computing. While trapped-ion research has been going on for some time, several labs have more recently ramped up their ability to control ensembles of neutral atoms.

To implement a neutral atom system, engineers can set up four laser beams around the atom ensemble to form a magneto-optical trap (MOT). Labs typically use either cesium (Cs) or rubidium (Rb) atoms for this work. By confining the atoms with this quadruple laser system, we can cool the atoms down to mK temperatures. Now that we have hundreds of millions of neutral atoms in a reservoir, we can transfer a small number of them into an addressable array (see Figure 5.2).

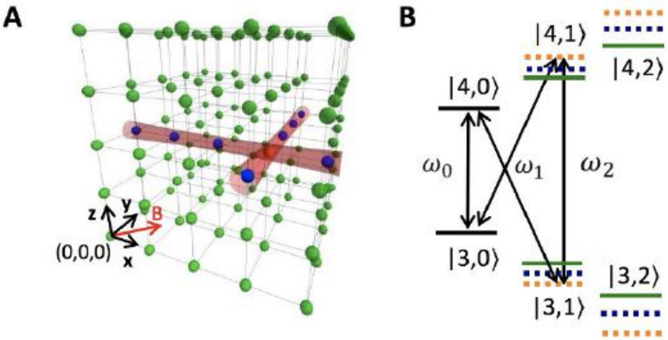


Figure 5.2: A. Diagram of addressing a $5 \times 5 \times 5$ *array of neutral atoms. Each addressing beam can be parallel-translated within* $5\mu s$ *to any line of atoms, so that any site can be put at their intersection. The addressing beams are circularly polarized, and the* $140mG$ *magnetic field is in the same plane. B.) The relevant part of the ground state energy level structure for addressing (not to scale) a target atom experiences twice the AC Stark shift of any other atom (its shift is illustrated by the orange dashed lines), so that, starting in the storage basis,* $|3,0\rangle$ *and* $|4,0\rangle$*, it alone is resonant with* ω_1*. After it is transferred to the computation basis,* $|3,1\rangle$ *and* $|4,1\rangle$*, it alone is resonant with* ω_2*. Image and caption from [228]*

The Lukin Lab at Harvard has made good progress on neutral atom systems [172, 133]. David Weiss and his group at Penn State have also focused on neutral atom platforms and have demonstrated a Stern-Gerlach-inspired neutral atom experiment [228, 238].

Neutral atom systems meet the DiVincenzo criteria for a quantum computer: qubits are individually separable and addressable, they can hold their state and we can perform a measurement on them. Saffman et al. offers a good review of implementing gates on a neutral atom system [193]. The challenge now is to scale such systems. See [232] for a helpful review.

5.3 *NMR*

Some of the first demonstrations of quantum computation made use of spin qubits using nuclear magnetic resonance (NMR) devices. Chuang and colleagues demonstrated the factoring of the number 15 using Shor's algorithm in a liquid-state NMR setup (LSNMR) [219]. While the number 15 is trivial to factor, it was one of the first demonstrations that quantum principles could be used in a physically realizable system for computation. Later, researchers experimented with solid state NMR (SSNMR) using nitrogen-vacancy centered

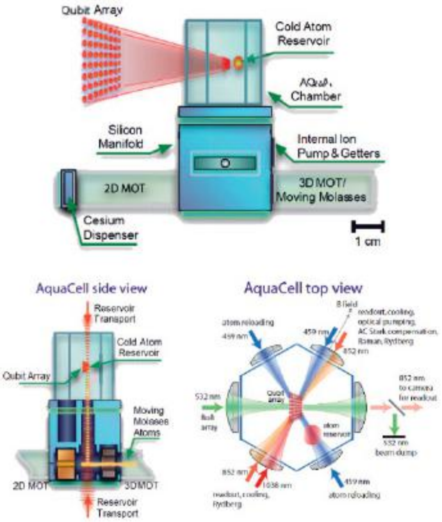


Figure 5.3: Neutral atom system setup Source: [14]

diamonds, which relates to another approach covered in this chapter. While a number of researchers use NMR platforms to test various QC phenomena, this platform is unlikely to scale for fault-tolerant quantum computation. We will need millions of physical qubits (translating to thousands of logical qubits) for such a device and NMR as currently implemented becomes impractical at those scales.

5.4 *NV Center-in-Diamond*

In the nitrogen-vacancy (NV) center-in-diamond approach to QC, two carbon atoms in the diamond lattice are missing, and one of them is replaced with a nitrogen ion. The resulting system forms a paramagnetic defect that can act as a qubit. The qubit state can be manipulated with microwave fields and read out optically. A number of labs have demonstrated basic NV systems [195, 218].

Instead of doping with nitrogen, several labs have doped with silicon [110, 76]. These and other materials each have their own unique advantages and disadvantages [53, 122]. For helpful reviews of NV approaches see [55, 71].

Researchers are currently investigating the successful application of a two-qubit operator on this platform (See [135, 28, 171, 109]).

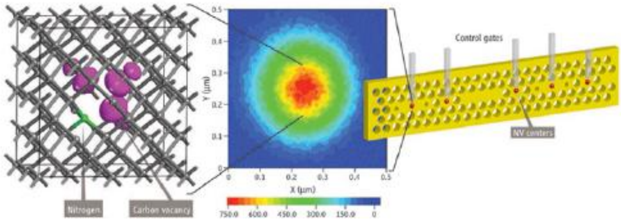


Figure 5.4: NV diamond schematic   Source: [173]

5.5 *Photonics*

Photonics can also be used to construct a gate-based quantum computer. Linear Optics Quantum Computing (LOQC) uses linear optical elements (such as mirrors, beam splitters and phase shifters) to process quantum information [7, 120, 123]. These optical elements preserve the coherence of input light and hence equivalently apply a unitary transformation on a finite number of qubits. However, photons do not interact with each other in a vacuum. They can only interact indirectly via another medium.

In 2001, Knill, Laflamme and Milburn (KLM) [120] presented a method of implementing scalable quantum computing using single photons, linear optics, photodetection and post-processing. These elements enable the application of nonlinear operations solely with linear optical elements [123]. Another paradigm is the so-called one-way, measurement-based or cluster-state quantum computer (MBQC) [179, 180, 181, 182] which first prepares a highly entangled multi-particle cluster state. Then, in order to implement a quantum circuit, one has to perform a sequence of single-qubit projective measurements. Every subsequent measurement choice is driven by the outcome of the previous measurement.

Arbitrary two-qubit processing requires the equivalent of three consecutive entangling gates in the circuit model of quantum computing [101] which is beyond the level of complexity that can be practically constructed and maintained with free-space quantum optics [143, 52]. Photonic chips can take advantage of the entire silicon-based infrastructure to miniaturize LOQC and bring down the cost [201]. Researchers from the University of Bristol, together with researchers from China's National University of Defense Technology, have demonstrated a photonic quantum processor on a silicon photonics chip. The processor generates two photonic qubits on which it performs arbitrary two-qubit unitary operations, including arbitrary entangling operations [178]. The quantum processor was fabricated with mature Complementary Metal

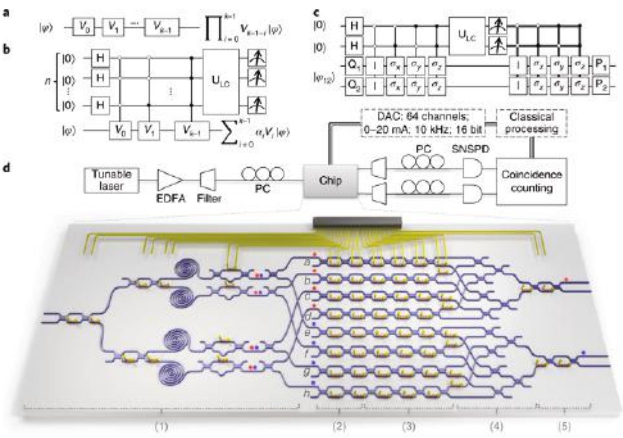


Figure 5.5: Quantum information processing circuits and schematic of the experimental set-up Source: [178]

Oxide Semiconductor (CMOS) compatible processing and comprises more than 200 photonic components; the processor was programmed to implement 98 different two-qubit unitary operations with an average quantum process fidelity of 93.2 ± 4.5%.

Semiconductor quantum transistor

The lack of deterministic photon-photon interactions is a challenge for quantum computation in this approach. To deterministically control an optical signal with a single photon requires strong interactions with quantum memory. Nanophotonic structures coupled to quantum emitters may offer an attractive approach to realize single-photon nonlinearities in a compact solid-state device. Recently, there has been great progress in controlling photons with solid-state qubits [16, 205, 30], as well as controlling a solid-state qubit with a photon [212].

Researchers at the University of Maryland and the Joint Quantum Institute have realized the first single-photon switch and transistor enabled by solid-state quantum memory using a semiconductor chip [211]. The device allows one photon to switch other photons, and hence to produce strong and controlled photon-photon interactions. It consists of a spin qubit strongly coupled to a nanophotonic cavity (an idea first proposed by Duan and Kimble [72]). They combined a semiconductor membrane with quantum dots; together they sit in the middle of the array. The array forms a photonic crystal, which uses the Bragg reflection mechanism where light bounces around a trap. That allows the quantum dot to store the information about the photon with a single electron, which has spin properties. By using this transistor, they should be able to apply quantum gates to photons. A scalable device suitable for quantum information processing will require higher efficiencies, as photon loss constitutes a dominant error source for photonic qubits.

Topological photonic chip

Topological insulators are exotic materials which insulate in their bulk but conduct on their surface [105, 177]. These states exhibit remarkable properties such as unidirectional propagation and robustness to noise. Since the discovery of these phases of matter, several topological effects have been observed using integrated photonics [138, 167, 240, 229, 99, 183, 100, 32, 153, 239, 54, 162, 134, 125, 221, 220, 244, 33, 154].

Topological photonics is a promising option for scalable quantum computers since they have the advantage of not requiring strong magnetic fields and feature intrinsically high-coherence, room-temperature operation and easy manipulation. Recently, scientists from RMIT University, Australia, in collaboration with researchers from the Politecnico di Milano and ETH Zurich, have developed a topological photonic chip that encodes, processes and transfers quantum information at a distance [214]. They have used photonic chips to demonstrate that topological states can undergo quantum interference by replicating the well known Hong-Ou-Mandel (HOM) experiment with 93.1 ± 2.8% visibility [112].

5.6 *Spin Qubits*

Silicon-based spin-qubit technology represents another approach to quantum computation. If we could construct qubits from common semiconductor materials, we could scale such a system by leveraging the decades of know-how in the integrated chip industry. It is difficult, however, to develop a stable, addressable qubit on the standard CMOS platform. Initial prototypes have been demonstrated, but it has been challenging to stabilize these qubits for scale-up. In these early attempts, a pair of quantum dots is placed between the source and sink in a traditional CMOS setup on a silicon semiconductor substrate. The entire device is placed in a dilution fridge to bring it down to about 1K; this is much warmer than the temperature needed for superconducting qubits. Microwave pulses are then used to apply unitary operators to the qubits [230].

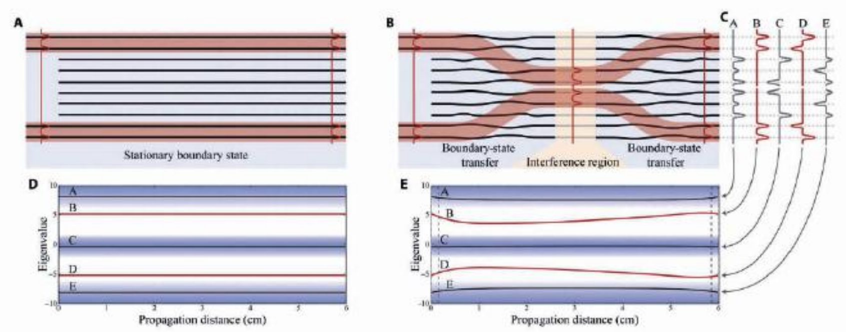


Figure 5.6: Photonic boundary-state beamsplitter. (A) Illustrative representation of a waveguide array implementing stationary topologically boundary states (red shaded regions) that propagate at the edges of the device. This device is used to confirm that the boundary state is preserved during the propagation inside the array. (B) Illustrative representation of a waveguide array implementing a TBS that interferes two topologically boundary states. (C) Photonic supermodes (eigenvectors) of the arrays at the start and end of the both devices. (D and E) Band structure (eigenenergies) along the length of the arrays (A and B). The topological bands (B and D) are highlighted in red, and the bulk bands (A, C, and E) are shaded in blue. Image and caption from [214]

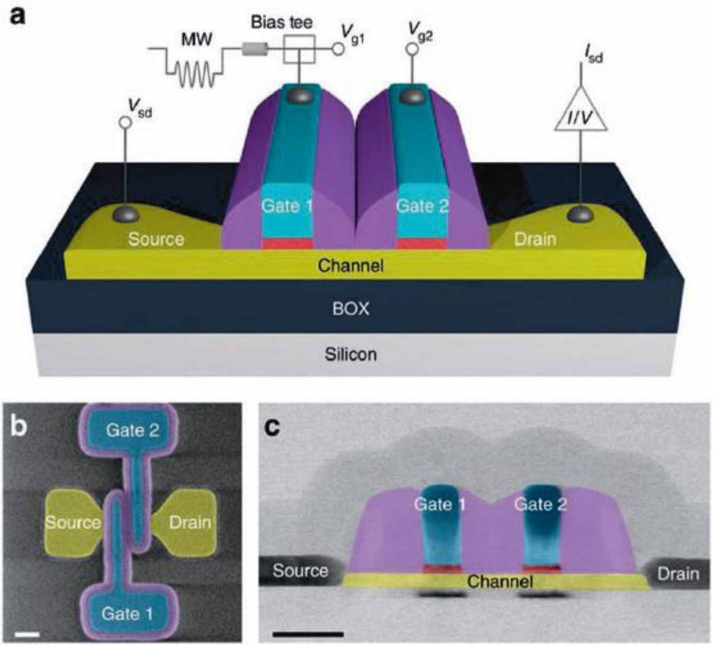


Figure 5.7: Spin qubit experimental system Source: [145]

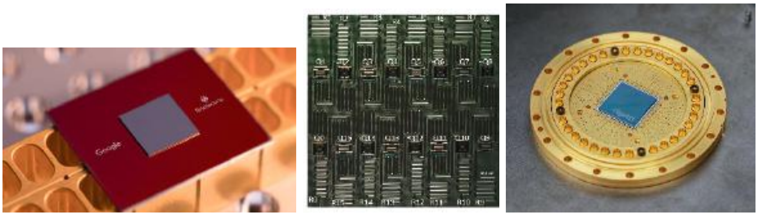


Figure 5.8: Superconducting processors (l to r): Google, IBM, Rigetti Source (l to r): Google, [113], [241]

Intel, HRL Laboratories, as well as labs at the University of Cambridge, Delft University of Technology, Harvard, and the University of New South Wales (UNSW) are working on silicon-based spin qubit approaches.

5.7 *Superconducting Qubits*

Several groups are building quantum computers with superconducting qubits. The core design is based on a qubit made from a Cooper pair with a Josephson junction [42]. Microwave leads are attached to the qubit to control it. By sending specific pulses of microwave frequencies for controlled amounts of time into the physical qubit, the user can apply the range of unitary operators. The entire apparatus must be cooled below 10mK to operate. The system is also shielded from magnetic fields and other factors that could cause decoherence.

The NAS report summarizes the various types of superconducting qubits [159, page C-1]:

> *Fixed-frequency versus tunable qubits*: Frequency-tunable qubits can be calibrated and corrected for qubit frequency variations that arise from variations in the fabrication process or as a result of device aging. An advantage is that one microwave tone can control multiple qubits, a savings in hardware. Gaining this advantage requires an additional control signal to adjust the frequency and adds an additional path for noise to enter the qubit. The two most common qubits in use today for digital superconducting quantum computing are the "transmon qubit", which comes in single-junction nontunable and two-junction tunable forms, and the "flux

> qubit."...Both transmon designs are being used in leading edge efforts.
>
> *Static versus tunable coupling*: Static coupling between qubits — for example, by using a capacitor or an inductor to mediate interaction — is an "always-on" coupling that is fixed by design. The coupling is turned "on" by bringing two qubits into resonance, and it is turned off by detuning the qubits. Yet even in the off state, there still is a small residual coupling. This tuning can be further reduced by adding a third object — either another coupler qubit or a resonator — between the two qubits. The two qubits are then coupled by adjusting the qubits and the resonator to the proper frequency.

A number of groups are working on superconducting qubit quantum computers including: Google, IBM, Rigetti, QCI and others (see Figure 5.8) and [165, 190]. Check this book's GitHub site for updated information in this area. Krantz, et al., offer a helpful review of this approach [124].

5.8 *Topological Quantum Computation*

In addition to the platforms we covered above, there are many other efforts to build quantum computing devices. The topological approach takes advantage of the unique properties of anyons. An anyon is a 2-dimensional quasiparticle which is neither a boson (such as a photon) nor a fermion (such as an electron). By braiding the pathways of an anyon in 4D spacetime, one can theoretically create a system for quantum computation which is robust to decohering effects (see Figure 5.9). This is due to the braided nature of the system — even if there are a series of small perturbations to the system, the topology of the system does not change, and therefore it stays coherent and quantum computation can continue. See Roy and DiVincenzo for more background on this approach [192].

Alexei Kitaev, then of the Landau Institute of Theoretical Physics, first proposed the idea of topological quantum computing [119]. Freedman, Kitaev, et al. then demonstrated that such a topological computer would be Turing complete [90]. Freedman and others then launched a program at Microsoft to investigate the physical realization of a topological quantum computer. For a useful survey, see [127].

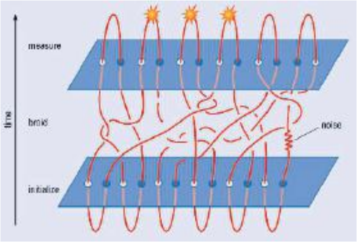
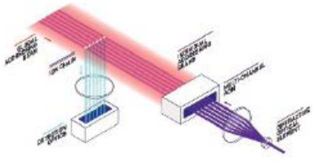


Figure 5.9: Braided anyons for topological computing *Source: [204]*

5.9 *Trapped Ion*



Source: [237]

In the trapped-ion approach, ytterbium atoms (or another element) are ionized with lasers and trapped in electric potentials to form a line of qubits. An additional laser is then used to measure the state of the qubits. Proponents of trapped-ion systems point to the ability to run their systems without having to cool it to mK ranges which is a requirement for a number of other approaches. Cirac and Zoller did early work in this field [59]. Chris Monroe of the University of Maryland, College Park and Jungsang Kim of Duke University have been active in this space and reported on several advances [12, 21]. Other groups include those at NIST [35], Oxford [139], Innsbruck [92], MIT [131] and ETH [86].

Figure 5.10 details two types of qubits that can be realized in a trapped ion system: optical qubits and hyperfine qubits. Optical qubits leverage the difference in energy levels between ground metastable states; hyperfine qubits distinguish between two different ground states. See [50] and [159] for helpful reviews of the trapped-ion approach.

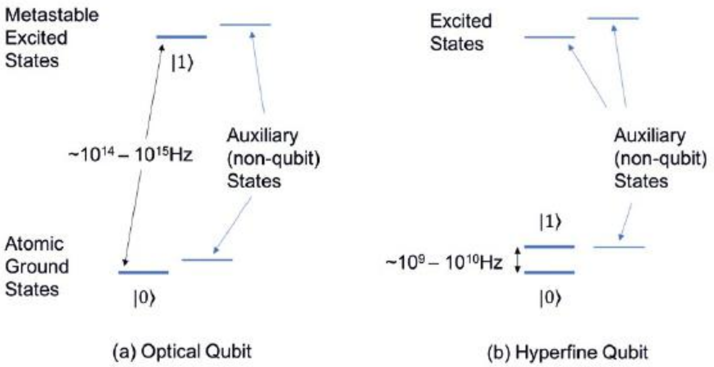


Figure 5.10: Qubits in an atomic ion. (a) An optical qubit consists of one of the atomic ground states and one of the metastable excited states, separated by approx. 10^{14} *to* $10^{15}\,Hz$*. (b) A hyperfine qubit consists of two of the ground states, separated by approx.* 10^{9} *to* $10^{10}\,Hz$*. Usually some excited states are used to support qubit manipulation operations. In both cases, there are other (auxiliary) states in the ground, excited and metastable excited states than those chosen to represent the qubit. Source: [159, Appendix B]*

5.10 *Summary*

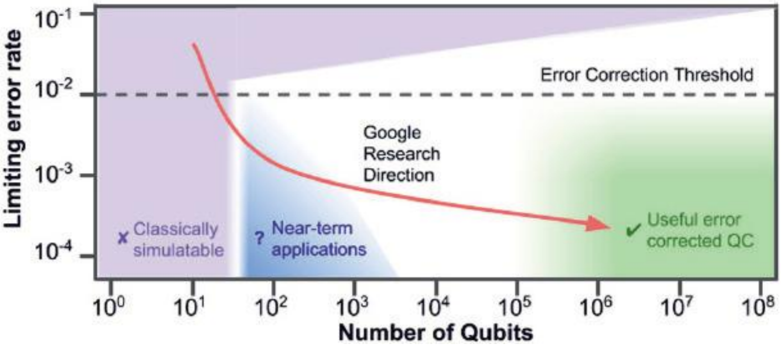


Figure 5.11: Quantum computing roadmap Source: Google

Researchers are pursuing a range of architectures in the quest for a fault-tolerant, scaled universal quantum computer. This period in quantum computing hardware is probably most akin to the 1940s and 1950s in classical computing. Research groups today are still figuring out which architecture will scale and to understand the kinds of problems which can be addressed with these platforms. Quantum hardware development is likely to move at a faster pace as we have the benefit of a global community of academic and commercial researchers forging ahead.

Figure 5.11 indicates that we are in the NISQ era of 10^2 to 10^3 qubits. We aim to reach $>10^6$ qubits in order to build a fully error-corrected quantum computer. Current quantum error correction techniques require about 1,000 physical qubits for every logical qubit. Since at least a few thousand logical qubits are needed for Shor's and other important algorithms, we will have to be in the 10^6 to 10^7 regime to realize this goal.

Now that we have surveyed the various approaches to quantum computing hardware, let us turn to the QC development platforms and software.

CHAPTER 6

Development Libraries for Quantum Computer Programming

With the growing interest in quantum computing, there are an increasing number of development libraries and tools for the field. There are development environments and simulators in all the major languages including Python, C/C++, Java and others. A comprehensive list can be found on this book's website.

Many of the leading QC research centers have focused on Python as the language of choice for building quantum circuits. One of the reasons for choosing Python is that it is a flexible, high-level language that allows programmers to focus on the problem being solved without worrying about too many formal details. For example, Python is dynamically typed (meaning variable types do not have to be declared by the programmer) and is an interpreted language (meaning it does not have to be pre-compiled into a binary executable). For these reasons and others, Python has a relatively easy learning curve for new users and has already seen strong support from the QC community.

In this chapter, we will provide an overview of how these libraries work and provide code examples for each framework. In the upcoming chapters, we will go into further detail with examples of algorithm implementations using these libraries. These quantum development libraries have methods for all the major unary and binary operators that we cover in this book. A few offer built-in modules for ternary operators, but these can be constructed if not offered in the library.

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_6

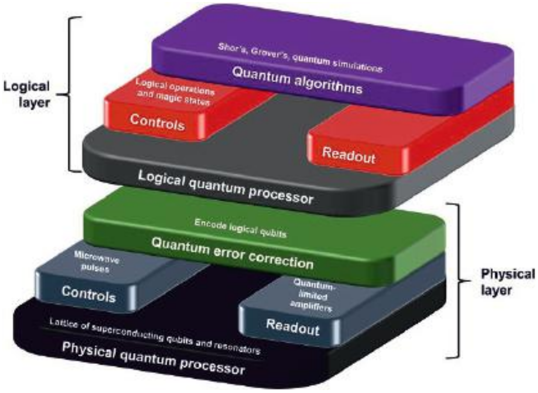

Logical layer
Shor's, Grover's, quantum simulations
Quantum algorithms
Logical operations and magic states
Controls
Readout
Logical quantum processor
Encode logical qubits
Quantum error correction
Microwave pulses
Controls
Quantum-limited amplifiers
Readout
Physical layer
Lattice of superconducting qubits and resonators
Physical quantum processor

Figure 6.1: Quantum computing stack Source: [94]

Figure 6.1 shows a schematic diagram of the quantum computing stack, which ranges from quantum algorithms and applications at the highest level to physical realizations of quantum computers at the lowest level. Numerous components sit between these layers such as control, readout, and — in the future — quantum error correction modules. Quantum programming languages (QPL) are used to interface with the top of the stack [129]. A quantum language consists of low-level instructions indicating which gates to perform on which qubits. Since it would be very tedious to program this by hand, when we program quantum computers, we interface with the higher-level quantum programming language in a development library.

Many QPLs are now available including both functional and imperative languages. The functional set includes: Qiskit, LIQUi|⟩, Q# and Quipper. Imperative languages for QC programming include: Cirq, Scaffold and ProjectQ. See [85] for a helpful review of open source frameworks for quantum computing.

6.1 *Quantum Computers and QC Simulators*

Cloud quantum computers make it possible to run algorithms on real quantum hardware, and development libraries enable this capability. In order to test code prior to running it on a quantum computer, most QC frameworks provide a QC simulator which runs on a classical computer. This simulator can run locally or in the cloud. Since it is running on a classical computer, it cannot, of course, process actual quantum states, but it is helpful to test the code syntax and flow.

There are numerous techniques to classically simulate quantum circuits, all of which suffer from the "exponential explosion" of classical memory: to store the most general state of an n qubit system, all 2^n complex numbers of the system's wavefunction must be stored. How much memory does this require? For simplicity, suppose that each complex number is stored using one byte. Then, for $n = 30$ qubits, 2^{30} bytes, or a gigabyte, of memory is required. For $n = 40$, a terabyte of memory is required, and for just $n = 50$, a petabyte is required. These memory requirements are already reaching the limits of today's best supercomputers, even for a modest number of qubits. Larger systems cannot hope to be simulated, since we wouldn't even have enough memory to write the wavefunction down!

The most basic method for simulating a quantum computer is to note that a quantum circuit simply expresses a unitary transformation, U, on a wavefunction, $|\psi\rangle$. The QC simulator algorithm then just performs matrix multiplication to get the resulting state $|\psi'\rangle$ via

$$|\psi'\rangle = U\,|\psi\rangle$$

Note that this method requires storing the entire unitary of the circuit, which is a $2^n \times 2^n$ matrix, in memory (in addition to storing the wavefunction).

A method that improves on these memory requirements, only having to store the wavefunction itself, works by applying one- and two-qubit gates to the wavefunction individually. To apply a single-qubit gate

$$G = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix}$$

to the ith qubit, one applies a matrix product to each amplitude α whose index differs in the ith bit [91, 206]

$$\begin{aligned} \alpha_{*\cdots*0_i*\cdots*} &= G_{11}\alpha_{*\cdots*0_i*\cdots*} + G_{12}\alpha_{*\cdots*1_i*\cdots*} \\ \alpha_{*\cdots*1_i*\cdots*} &= G_{21}\alpha_{*\cdots*0_i*\cdots*} + G_{22}\alpha_{*\cdots*1_i*\cdots*} \end{aligned}$$

Here, the subscript i on either 0 or 1 denotes that this index is in the ith position, and asterisks denote indices that are the same on either side of the equation. A similar update equation exists for two-qubit gates. The algorithm for this "state vector" or "wavefunction" simulator then consists of iterating through all single-qubit and two-qubit gates in the circuit and applying the appropriate update equation.

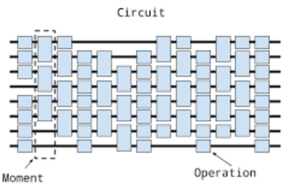
Other types of QC simulators exist, such as Clifford circuit simulators, which can efficiently simulate several hundreds or thousands of qubits. However, as discussed in chapter 3, these circuits are not universal. In this chapter, we focus on programming universal QCs and QC simulators. When a program is sent to a quantum backend, it is first compiled into gates the computer can actually implement. This compilation is expressed in a lower level language — quantum assembly or instruction language — which is sent to the computer. At the lowest level, gates are implemented by physical operations acting on the qubits. These physical operations can include microwave pulses, laser pulses or other interactions acting on a qubit, depending on which physical realization of a qubit is used.

6.2 Cirq

Cirq Overview

Institution	Google
First Release	v0.1 on April 17, 2018
Open Source?	Yes
License	Apache-2.0
Github	https://github.com/quantumlib/Cirq
Documentation	https://cirq.readthedocs.io/en/stable/
OS	Mac, Windows, Linux
Classical Language	Python

Figure 6.2: Overview of the Cirq dev library. Modified with permission from [129].



Source:[60]

Cirq is Google's quantum computing development library. Cirq enables the developer to build and execute quantum circuits comprised of all the usual unary, binary and ternary operators we have covered in this book. We will use Cirq for the majority of code examples throughout this book.

To get acquainted with the language, a sample program in Cirq is provided below [60]. This program creates a quantum circuit with one qubit and performs the *NOT* operator on it followed by a measurement. This circuit is simulated several times and its measurement outcomes are displayed to the console. The full program is shown below.[1]

```
"""Simple program in Cirq."""

# Import the Cirq package
import cirq

# Pick a qubit
qubit = cirq.GridQubit(0, 0)

# Create a circuit
circuit = cirq.Circuit.from_ops([
   cirq.X(qubit), # NOT.
   cirq.measure(qubit, key=’m’) # Measurement
   ]
)

# Display the circuit
print("Circuit:")
print(circuit)

# Get a simulator to execute the circuit
simulator = cirq.Simulator()

# Simulate the circuit several times
result = simulator.run(circuit, repetitions=10)

# Print the results
print("Results:")
print(result)
```

In this circuit, we first set up a qubit within a grid pattern. Cirq also allows qubits to be set up in a linear fashion if needed, since both linear and two-dimensional qubit arrays are the most popular for near-term quantum computer architectures. We then apply the *NOT* operator to the qubit. If the qubit was in a state of $|0\rangle$ before, it will now be in the state $|1\rangle$ and vice-versa. We then measure the qubit to output the classical bit of the measurement result. Note that the keyword argument `key=’m’` in the measure operation makes it easy to access measurement results using the `histogram` method of a Cirq TrialResult. In this simple program, it is not necessary, but in more complex programs it can be a useful tool.

The next lines of code get a QC simulator and execute the circuit ten times. Then, the results of executing the circuit are printed to the screen. A sample output from this program is shown below.

[1] To find versions of each program shown in this chapter for the latest releases of each development library, see the book’s GitHub page.

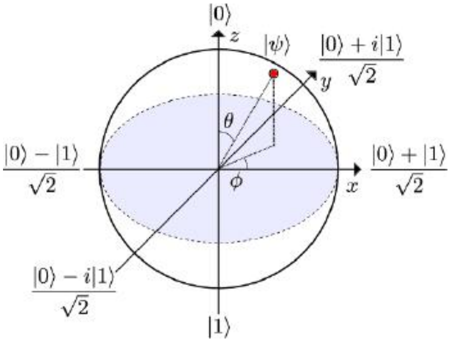

$|0\rangle$
z
$|\psi\rangle$
$\frac{|0\rangle + i|1\rangle}{\sqrt{2}}$
y
θ
$\frac{|0\rangle - |1\rangle}{\sqrt{2}}$
$\frac{|0\rangle + |1\rangle}{\sqrt{2}}$
ϕ
x
$\frac{|0\rangle - i|1\rangle}{\sqrt{2}}$
$|1\rangle$

Figure 6.3: The Bloch sphere Source: [93]

```
Circuit:
(0, 0): ---X---M('m')---
Results:
m=1111111111
```

The circuit is printed out in ASCII text, as is standard for drawing circuits in Cirq, and the results are indicated by a sequence of binary digits. As we anticipate, on a noiseless QC simulator all measurement outcomes are equal to 1. Finally, note that the string of measurement outcomes is labeled by the measurement key `m` provided as a keyword argument to the measure operation.

We can represent the application of the X operator in this circuit on a Bloch sphere (see Figure 6.3). We recall that the Bloch sphere represents the computational basis states of $|0\rangle$ and $|1\rangle$ at the poles. By applying the X gate to the prepared state of $|0\rangle$ we move the qubit to state $|1\rangle$. If we then apply a Hadamard gate, we can represent the new qubit state with a horizontal vector on the Bloch sphere corresponding to:

$$\frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

For circuits of many operators we can use dynamic Bloch sphere simulation software from Q-Ctrl (and other sites listed on the companion website) to visualize the unitary transformations.

6.3 *Qiskit*

The Quantum Information Science Kit, or Qiskit for short, is a quantum computing dev library created by IBM. The Qiskit library is a flexible frame-

Qiskit Overview

Institution	IBM
First Release	0.1 on March 7, 2017
Open Source?	Yes
License	Apache-2.0
Homepage	https://qiskit.org/
Github	https://github.com/Qiskit
Documentation	https://qiskit.org/documentation/
OS	Mac, Windows, Linux
Classical Host Language	Python
Quantum Language	OpenQASM

Figure 6.4: Overview of the Qiskit development library. Modified with permission from [129].

work for programming quantum computers. The Qiskit development library consists of four core modules distributed across the quantum computing stack:

- *Qiskit Terra*: Terra provides core elements for composing quantum programs at the level of circuits and pulses, and optimizing them for the constraints of a particular physical quantum processor.
- *Qiskit Aer*: Aer provides a C++ simulator framework and tools for constructing noise models for performing realistic noisy simulations of the errors that occur during execution on real devices.
- *Qiskit Ignis*: Ignis is a framework for understanding and mitigating noise in quantum circuits and devices.
- *Qiskit Aqua*: Aqua contains a library of cross-domain quantum algorithms upon which applications for near-term quantum computing can be built.

Except for Aqua, at the time of writing, each of these components is installed automatically with Qiskit. The Aqua module can be installed separately and requires a working installation of the core Qiskit library. To demonstrate the coding syntax in Qiskit, we include below the same example program that was shown previously in Cirq.

```
"""Simple program in Qiskit."""

# Import the Qiskit package
import qiskit

# Create a quantum register with one qubit
qreg = qiskit.QuantumRegister(1, name='qreg')

# Create a classical register with one qubit
creg = qiskit.ClassicalRegister(1, name='creg')
```

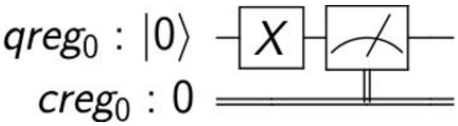


Figure 6.5: Circuit diagram drawn in Qiskit for the above program

```
# Create a quantum circuit with the above registers
circ = qiskit.QuantumCircuit(qreg, creg)

# Add a NOT operation on the qubit
circ.x(qreg[0])

# Add a measurement on the qubit
circ.measure(qreg, creg)

# Print the circuit
print(circ.draw())

# Get a backend to run on
backend = qiskit.BasicAer.get_backend("qasm_simulator")

# Execute the circuit on the backend and get the measurement results
job = qiskit.execute(circ, backend, shots=10)
result = job.result()

# Print the measurement results
print(result.get_counts())
```

This program in Qiskit closely mirrors that in Cirq, with minor differences due to language design, syntax and notation. After importing the Qiskit development library, the program declares a quantum and classical register with one qubit, which it then uses to create a circuit. Note how this is different from Cirq, in which (1) a classical register is never explicitly created and (2) qubits are only referred to in a circuit when operations are added. Continuing through the Qiskit program, the next lines add the appropriate operations (*NOT* and measure) to the circuit, which is subsequently printed out by drawing the circuit.

Qiskit has the ability to draw and save circuits as files in addition to printing out text representations. Figure 6.5 shows the circuit for this program drawn with the code shown below:

```
    circ.draw(filename="qiskit-circuit", output="latex")
```

After printing the circuit, a backend is declared for executing the quantum circuit. In the final lines, the circuit is executed, the results are retrieved, and finally the measurement statistics (counts) are printed to the screen. An example output of this program is shown below. Note that the circuit will also be printed to the console in the above program — we omit the text representation here.

```
{'1': 10}
```

In Qiskit, measurement outcomes are stored as dictionaries (a Python data type consisting of key-value pairs) where keys are bit strings and values are the number of times each bit string was measured. Here, this output says that the only bit string present in the measurement is 1. Similar to the Cirq output, since we are running this program on a QC simulator without activating a noise model, all measurement outcomes are 1 as expected.

6.4 *Forest*

Forest Overview

Institution	Rigetti
First Release	v0.0.2 on Jan 15, 2017
Open Source?	Yes
License	Apache-2.0
Homepage	https://www.rigetti.com/forest
GitHub	https://github.com/rigetti/pyquil
Documentation	pyquil.readthedocs.io/en/latest/
OS	Mac, Windows, Linux
Classical Language	Python
Quantum Programming Library	pyQuil
Quantum Language	Quil

Figure 6.6: Overview of the Forest development library. Modified with permission from [129].

Forest is a development library by Rigetti. Similar to the previous two libraries, Forest is Python-based and features a collection of tools for effective quantum programming. Users type quantum programs in pyQuil, and low-level instructions are sent to the quantum computer as Quil, short for Quantum Instruction Language [207]. The name Forest refers to the collection of all programming tools in this toolchain including Quil, pyQuil and other components such as Grove, a collection of quantum algorithms written in pyQuil.

To get a sense for the language we implement the "*NOT* and measure" program in pyQuil below.

```
"""Simple program in pyQuil."""

# Import the pyQuil library
import pyquil

# Create a quantum program
prog = pyquil.Program()

# Declare a classical register
creg = prog.declare("ro", memory_type="BIT", memory_size=1)

# Add a NOT operation and measurement on a qubit
prog += [
   pyquil.gates.X(0),
   pyquil.gates.MEASURE(0, creg[0])
   ]

# Print the program
print("Program:")
print(prog)

# Get a quantum computer to run on
computer = pyquil.get_qc("1q-qvm")

# Simulate the program many times
prog.wrap_in_numshots_loop(10)

# Execute the program on the computer. NOTE: This requires the QVM
    to be running
result = computer.run(prog)

# Print the results
print(result)
```

Here, we first import the pyQuil library and then create a program, the equivalent of a circuit in Cirq or Qiskit. After creating the program, we declare a classical register of one bit, then add the operations for this circuit — *NOT* and measure. Note that qubits can be indexed dynamically in pyQuil; there is no explicit reference to a qubit register, but rather we provide an index (0) to the gate operations. In contrast, classical memory does have to be explicitly declared, and so we measure the qubit into a classical register.

After printing out the program to visualize its instructions, we send the instructions to a QC for execution. Here, the string key designates that we want a one qubit (`1q`) quantum virtual machine (`qvm`), which is Rigetti's terminology for their quantum computer simulator. To simulate the program many times, we call the method `wrap_in_numshots_loop` on the program, and provide as input a number of repetitions (shots). Finally, we run the program on the specified computer and print the result. To execute this program on the quantum virtual machine, we can initiate the QVM with the following command in a terminal:

```
qvm -S
```

Once executed in the QVM, the program will produce and the following results:

```
Program:
DECLARE ro BIT[1]
X 0
MEASURE 0 ro[0]

Result:
[[1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]]
```

Unlike Cirq and Qiskit, pyQuil does not produce a circuit diagram of the program, but rather displays the Quil instructions line by line. In this particular Quil program, a readout register called `ro` is declared, a *NOT* operation is applied to qubit 0, and the measurement outcome of qubit 0 is stored into the readout register. The results of simulating the circuit appears as a list of lists. Each inner list is the measurement outcome for a particular execution of the circuit. The number of inner lists is equal to the total number of times the circuit was executed. All of these are wrapped in an outer list containing the results of all simulations. As with the previous programs, all measurement outcomes are 1 on a noiseless quantum computer simulator, as we expect.

6.5 *Quantum Development Kit*

The Quantum Development Kit (QDK) is a quantum computing development library by Microsoft. Unlike the previous languages, which were based in Python, the QDK contains its own language, called Q# (pronounced “Q sharp”), for writing quantum programs.

QDK Overview

Institution	Microsoft
First Release	0.1.1712.901 on Jan 4, 2018
Open Source?	Yes
License	MIT
Homepage	microsoft.com/en-us/quantum/development-kit
Github	https://github.com/Microsoft/Quantum
Documentation	docs.microsoft.com/en-us/quantum/?view=qsharp-preview
OS	Mac, Windows, Linux
Quantum Prog. Lang.	Q#

Figure 6.7: Overview of the QDK development library. Modified with permission from [129].

The Q# language is different in several respects compared with Python-based libraries such as Cirq, Qiskit and pyQuil. In Q# we need to explicitly declare types, as well as use curly braces instead of indents as in Python. Additionally, three separate files are required to execute programs using the QDK:

1. A file ending in .qs where quantum operations (analogues of functions in Python) are stored
2. A driver file ending in .cs where quantum operations are executed in the main program
3. A file ending in .csproj which defines the project and contains metadata about computer architecture and package references

An example program in Q# that executes the same "*NOT* and measure" circuit we have seen in other languages is included below. We first show the .qs file which defines the quantum operations we will use in the driver file.

```
namespace Quantum.Simple
{
   // Importing the libraries
   open Microsoft.Quantum.Primitive;
   open Microsoft.Quantum.Canon;

   // Sets a qubit in a desired state
   operation Set(desired_state: Result, qubit: Qubit) : Unit {
      let current = M(qubit);
      if (current != desired_state) {
         X(qubit);
      }
   }

   // Executes the NOTandMeasure circuit for an input number
   // of repetitions and returns the number of ones measured
   operation NotAndMeasure(repetitions: Int) : Int {
      // Variable to store the number of measured ones
      mutable num_ones = 0;

      // Get a qubit to use
      using (qubit = Qubit()) {
         // Loop over the desired number of repetitions
         for (test in 1..repetitions) {
            // Get a qubit in the zero state
            Set(Zero, qubit);

            // Perform a NOT operation
            X(qubit);

            // Measure the qubit
            let res = M (qubit);

            // Keep track of the number of ones we measured
            if (res == One) {
               set num_ones = num_ones + 1;
            }
         }
         // "Released qubits" must be in the zero state to avoid a
             System.AggregateException
         Set(Zero, qubit);
      }
      // Return the number of ones measured
      return num_ones;
   }
}
```

In the first line we define a *namespace* for the operations. This namespace is used in the driver file to access these operations. The next two lines are the equivalent of import packages in Python — they make operations defined in the QDK available for us to use in the program (e.g., the X gate). Next, we declare an operation called Set which inputs a desired computational basis state and an arbitrary qubit state; it then changes the state of the qubit to match the desired state. This is accomplished by measuring the qubit in the computational basis and then performing a *NOT* operation if necessary.

We then define the `NotAndMeasure` operation which sets up the quantum circuit and measures the qubit for a user-specified number of times. Explicitly, the operation sets a qubit to be in the 0 state using the previous operation we defined, performs a *NOT* operation, then measures, keeping track of the number of 1s measured. After the total number of repetitions, the number of 1s measured is returned.

In order to use this file, we must define a separate driver file ending in .cs which executes these operations. The driver file used to execute this program is shown below.

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

namespace Quantum.Simple {
   class Driver {
      static void Main() {
         // Get a quantum computer simulator
         using (var qsim = new QuantumSimulator()) {
            // Run the operation NotAndMeasure and get the result
            var num_ones = NotAndMeasure.Run(qsim, 10).Result;

            // Print the measurement outcome to the console
            System.Console.WriteLine(
               $"Number of ones measured: {num_ones, 0}.");
         }
      }
   }
}
```

Here, we import System, which allows us to print to the console, and the necessary tools to use the QC simulator in the QDK. Next, we invoke the namespace declared in the .qs file above and define a Main function inside the driver class. This Main function uses a QC simulator to run the `NotAndMeasure` operation, then prints the results to the console. The output of this program is shown below.

```
Number of ones measured: 10.
```

As in other languages, we obtain the correct output of measuring all ones. The final file .csproj needed to execute this code is included on this book's GitHub site. We omit the program here since it is nearly identical for each project in the QDK.

6.6 *Dev Libraries Summary*

In the sample programs shown for these libraries, we have seen that most frameworks are fairly similar in terms of constructing quantum circuits. The general recipe for each of them is:

1. Build a quantum circuit consisting of quantum and/or classical registers
2. Add operations to the circuit
3. Simulate the circuit

However, there are some differences across the various libraries. For example, qubits are defined separately from a quantum circuit in Cirq, whereas qubits are required as input to a quantum circuit in Qiskit. Similarly, all qubits must be allocated in a quantum register before performing operations on them in Qiskit, but qubits can be allocated dynamically when programming in pyQuil.

While these differences may seem small, there are bigger differences in the features each development library contains. For example, some libraries include the ability to simulate noise, compile algorithms to arbitrary architectures, access the wavefunction for debugging and so on. These differences lead to certain algorithms being easier to implement in particular libraries. While one development library is usually sufficient for most tasks, having experience in multiple libraries is helpful to choose the the appropriate library for certain cases.

Using the Libraries

In the remainder of the book, we will use these development libraries to explore the core algorithms that make up the canon of quantum computing and more recent QC methods. While Cirq is the main library used, some programs are shown in other libraries for comparison. Before continuing on to this core part of the text, we briefly mention other development libraries that we have not yet covered.

Other Development Libraries

As mentioned, before the emergence of cloud quantum computing, many quantum computer simulators were developed in languages ranging from C++ (e.g., Quantum++) to Java (e.g., jSQ) to Rust (e.g., QCGPU). A full list of quantum computer simulators is available at https://quantiki.org/wiki/list-qc-simulators. While many of these simulators are outdated or deprecated, a number of them are still actively developed and being improved upon. Indeed, while classical algorithms for simulating quantum circuits are widely believed to be intractable, finding new ways to simulate more qubits with greater speed is still an active area of research.

In addition to the libraries above, here are other libraries of interest: ProjectQ, a Python library with a high-performance C++ quantum computer simulator; Strawberry Fields, a Python library built around continuous variable quantum computing; and Ocean, a Python library for quantum annealing on D-Wave quantum computers. Additionally, a hardware independent quantum programming framework called XACC is also actively being developed.

For a full list of open-source quantum software projects, see the companion website to this book.

6.7 *Additional Quantum Programs*

In principle, knowing how to program circuits, simulate them and access measurement results are the core skills needed for coding quantum algorithms in future chapters. While the simple "*NOT* and measure" program contains these three key features, it is nonetheless simple. To help bridge this gap to the more complex programs to come, we include two additional example quantum programs in this section.

Bell States

One common circuit pattern is the set of operators we use to prepare one of the four Bell states

$$|\Phi^{+}\rangle = \frac{1}{\sqrt{2}}[|00\rangle + |11\rangle]$$

from the ground state. It can be easily verified that such a circuit consists of a Hadamard on the first qubit and then a *CNOT* between the two qubits, where the first qubit is the control qubit. This structure appears, for example, in the quantum teleportation circuit for creating entangled pairs of qubits between two parties.

A circuit for preparing a Bell state, written in Cirq, is shown below. For pedagogical purposes, we add measurements after the Bell state preparation circuit to see the possible measurement outcomes.

```
"""Script for preparing the Bell state |\Phi^{+}> in Cirq."""

# Import the Cirq library
import cirq

# Get qubits and circuit
qreg = [cirq.LineQubit(x) for x in range(2)]
circ = cirq.Circuit()

# Add the Bell state preparation circuit
circ.append([cirq.H(qreg[0]),
             cirq.CNOT(qreg[0], qreg[1])])

# Display the circuit
print("Circuit")
print(circ)
```

```
# Add measurements
circ.append(cirq.measure(*qreg, key="z"))

# Simulate the circuit
sim = cirq.Simulator()
res = sim.run(circ, repetitions=100)

# Display the outcomes
print("\nMeasurements:")
print(res.histogram(key="z"))
```

Note that in this Cirq program, we utilize the measurement key to access measurement outcomes using the `histogram` method. An example output of this program is shown below:

```
Circuit
0: ---H---@---
          |
1: -------X---

Measurements:
Counter({3: 53, 0: 47})
```

Here, the circuit drawing shows the Bell state preparation circuit consisting of a Hadamard and *CNOT* gate. After, the measurement outcomes enumerate how many 0 and 3 states were measured. Note that this is a binary representation of bitstrings — i.e., 0 stands for 00 and 3 stands for 11. As expected, these are the only two measurement outcomes in the program, meaning that the measurements are perfectly correlated: when one qubit is measured 0(1), the other qubit is always measured 0(1).

Gates with Parameters

Several quantum gates are defined in terms of angles, for example the standard rotation gates $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$. In many quantum algorithms, known as variational quantum algorithms, these angles, or parameters, are iteratively adjusted in order to minimize cost. For example, in the variational quantum eigensolver, gate parameters are adjusted in order to minimize the expectation of a Hamiltonian $\langle\psi(\theta)|H|\psi(\theta)\rangle$. Regardless of the application, such variational quantum algorithms depend critically on the ability to update and change gate parameters.

Because of this, most quantum computing development libraries contain built-in features and methods for working with parameterized gates. In the following program, we demonstrate this functionality in Cirq. In particular, we set up a simple quantum circuit with one parameterized gate, execute the algorithm for a `sweep` of parameters and plot the measurement results.

```python
"""Working with parameterized gates in Cirq."""

# Imports
import matplotlib.pyplot as plt
import sympy

import cirq

# Get a qubit and a circuit
qbit = cirq.LineQubit(0)
circ = cirq.Circuit()

# Get a symbol
symbol = sympy.Symbol("t")

# Add a parameterized gate
circ.append(cirq.XPowGate(exponent=symbol)(qbit))

# Measure
circ.append(cirq.measure(qbit, key="z"))

# Display the circuit
print("Circuit:")
print(circ)

# Get a sweep over parameter values
sweep = cirq.Linspace(key=symbol.name, start=0.0, stop=2.0,
    length=100)

# Execute the circuit for all values in the sweep
sim = cirq.Simulator()
res = sim.run_sweep(circ, sweep, repetitions=1000)

# Plot the measurement outcomes at each value in the sweep
angles = [x[0][1] for x in sweep.param_tuples()]
zeroes = [res[i].histogram(key="z")[0] / 1000 for i in
    range(len(res))]
plt.plot(angles, zeroes, "--", linewidth=3)

# Plot options and formatting
plt.ylabel("Frequency of 0 Measurements")
plt.xlabel("Exponent of X gate")
plt.grid()

plt.savefig("param-sweep-cirq.pdf", format="pdf")
```

We highlight several key components in this program. First, a `symbol` in Cirq is used to represent a numerical value of a parameter that will be determined later, before executing the circuit. When the circuit is printed (see below), the name of the symbol appears in the circuit.

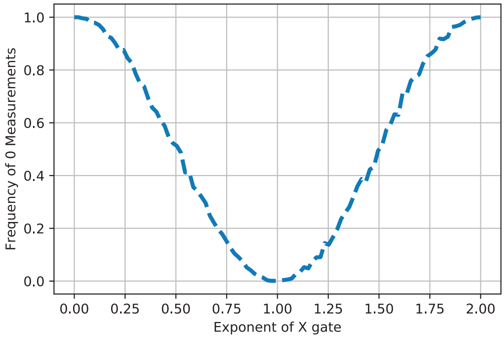


Figure 6.8: Measurement outcomes at each value of the exponent $t \in [0, 2]$ *in the circuit* $X^t |0\rangle$

```
Circuit:
0: ---X^t---M('z')---
```

Next, a `sweep` is a set of values that the symbol will take on. Simulators in Cirq have a method, called `run_sweep`, for executing circuits at all values in a sweep. At each value, the circuit is simulated a number of times set by the repetitions keyword argument. The remaining code in the program is for plotting the outcome of the `run_sweep`, which is shown in Figure 6.8.

Now that we have explored the various quantum computing development libraries, let us look at three protocols — quantum teleportation, superdense coding, and Bell's inequality test — and then turn to the canon of quantum algorithms that helped establish the field.

CHAPTER 7

Teleportation, Superdense Coding and Bell's Inequality

Two of the most fascinating quantum circuits enable us to transmit information in ways that are not possible in the classical regime. In this chapter, we will learn how to build these two circuits. We will then examine a foundational advance in quantum mechanics, the Bell Inequality.

7.1 *Quantum Teleportation*

Quantum teleportation, despite its name, does not teleport any physical object. It does transmit the state of a qubit over any given distance in a way that is completely secure. It is remarkable that it took more than seventy years from the formulation of QM to realize that this framework gave us a new form of secure communication. The protocol was developed in 1993 [26] by Bennett and Brassard et al.; it was experimentally verified in 1997 [41]. Bennett and Brassard had also developed quantum key distribution in 1984, known as BB84 [25].

One of the key insights of quantum teleportation is that *we can treat entangled states as a resource*. We can use entangled states (known as EPR pairs or Bell states) to perform a range of tasks that cannot be accomplished using classical means.

In quantum teleportation, Alice is the sender and she wishes to transmit the state of a qubit, Q, to a receiver, Bob. The algorithm requires three qubits in total. Let's walk through the protocol in detail:

1. We set up the system with three qubits:

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_7

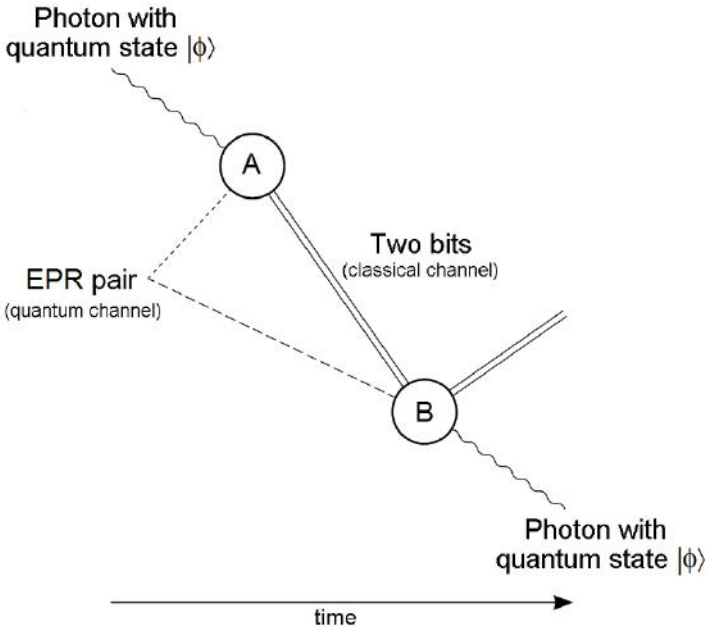


Figure 7.1: Quantum teleportation diagram. Relative to the notation in the main text, $|\Phi\rangle$ is the state of qubit Q, $A = R$ and $B = S$. *Source: Wikimedia*

   (a) Alice has a qubit, Q, with state $|\Phi\rangle$. Alice wishes to transmit the state $|\Phi\rangle$ to Bob in a secure manner.

   (b) To accomplish this goal, Alice also starts with two additional qubits, which we will label R and S. One of these qubits, say S, will be sent to Bob, and the other will stay with Alice. In practice, S can be sent over a quantum channel such as an optical fiber if the qubits are photons.

2. Alice prepares a Bell state with qubits R and S. This is done by applying a Hadamard to qubit R and then a *CNOT* between R and S, controlling on R. At this point, Alice sends qubit S to Bob.
3. Alice now performs a Bell measurement on her original qubit Q and her half of the EPR pair, R. This is done by performing a *CNOT* between the qubits, controlling on Q, then performing a Hadamard gate on Q, and finally measuring both qubits in the computational basis.
4. After measuring, Alice now has two bits of classical information, one from each measured qubit. Alice now transmits these bits to Bob over a classical communication channel. Note that there are four possible outcomes from her measurements: 00, 01, 10, and 11.

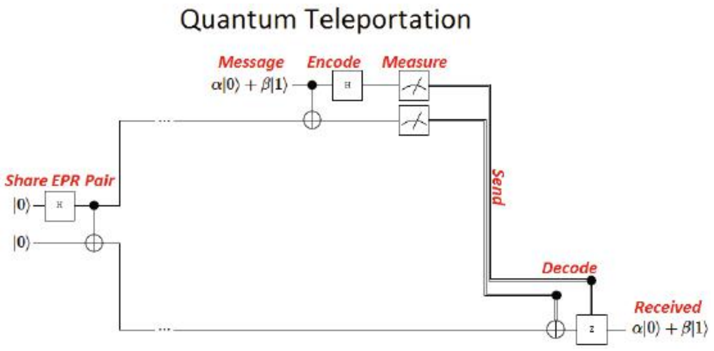
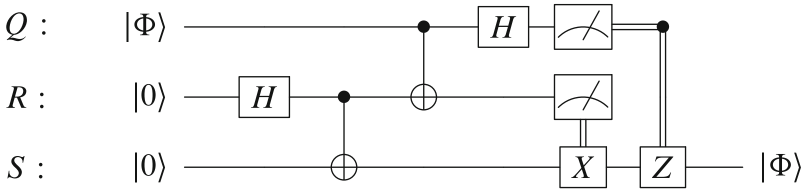


Figure 7.2: Circuit diagram for quantum teleportation Source: Wikimedia

5. Depending on which bit string Bob receives from Alice, he performs a set of operations on his qubit, S. The dictionary of operations for each measurement result is listed below. Performing the appropriate operation guarantees Bob's qubit, S, will be in the same state as Alice's original qubit Q — even though neither Alice nor Bob know what this state is!

If Alice transmits	then Bob applies this operator
00	None — Bob's qubit is in the right state
01	Z
10	X
11	XZ (apply Z first then X)

Two ways to represent quantum teleportation in a circuit diagram can be seen in figure 7.2 and below; they are equivalent and it is useful to be familiar with different ways of presenting circuits:



Let us highlight the following points on teleportation:

1. Note how Alice prepares a Bell state with an H on R and then a *CNOT* across the pair of qubits, R and S.
2. Note how Alice transmits two bits of classical information to Bob — these are denoted with the sets of double wires in the quantum circuit.

3. In this circuit Alice successfully transmits the state of $|\Phi\rangle$ to Bob via an EPR pair and the use of two classical bits. We can also represent this transmission as

$$[qq] + [cc] \geq [q]$$

where $[qq]$ represents an EPR pair, $[cc]$ represents a pair of classical bits and $[q]$ is the state of a qubit we wish to transmit.

While quantum teleportation is a tool we can use in quantum communications, it also has applications in quantum computing [58]. We can potentially use quantum teleportation to create a modular architecture for quantum computing by sending a quantum state from one module to another in a scaled quantum computer (see [58]).

7.2 *Superdense Coding*

Superdense coding is a method to transmit classical bits by sending only one qubit from sender to receiver. If Alice wishes to transmit two classical bits to Bob using a classical channel, she would have to use two bits. With superdense coding, however, she can communicate the two bits with the transmission of *just one qubit*.

This protocol was initially developed by Bennett and Wiesner [27] and then further specified as a secure communications protocol [226]. Anton Zeilinger experimentally demonstrated superdense coding transmission in 1995 [144].

To achieve superdense coding, Alice first prepares an EPR pair. She then performs one of four operations on her half of the pair. Let's say that these are a pair of photons. To create the Bell pair, Alice first applies a Hadamard to her photon and then a *CNOT* across the two photons as she did in the case of preparing an EPR pair for quantum teleportation protocol. The pair of photons is now entangled.

Now Alice chooses which of four classical states she wishes to transmit to Bob as the intended message. Depending on the message she chooses to send, Alice applies a specific quantum operator to her photon.

If Alice wants to send	Alice applies
00	I (identity operator)
01	X
10	Z
11	ZX (first apply X, then apply Z)

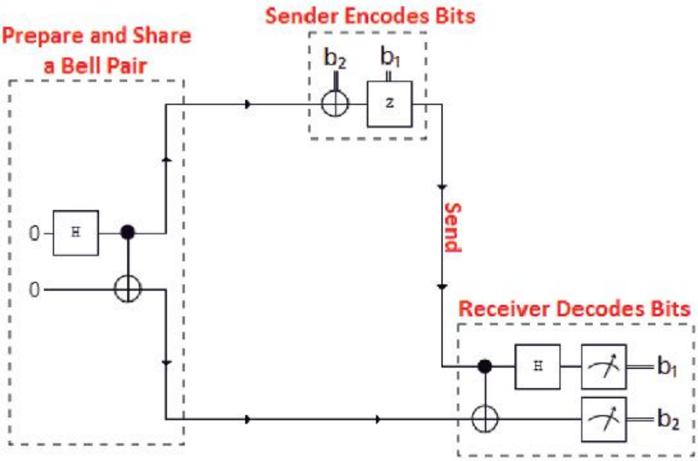


Figure 7.3: Circuit diagram for superdense coding Source: Wikimedia

Next she sends her photon to Bob via a quantum communications channel that preserves entanglement. Upon receipt of the photon, Bob applies the Hadamard to her photon and then a *CNOT* across the photon pair. He then performs a measurement. The result will be two classical bits of information. Let us recall that the output of a measurement is classical information.

We can represent superdense coding in shorthand as follows:

$$[q] + [qq] \geq [cc]$$

7.3 *Code for Quantum Teleportation and Superdense Communication*

A program for quantum teleportation is provided in Cirq below. This program encodes a random quantum state in Alice's qubit and prints out its Bloch sphere (x, y, z) components. It then executes the quantum teleportation circuit and prints out the Bloch sphere (x, y, z) components of Bob's qubit.

```
"""Quantum teleportation in Cirq. Modified from
    quantum_teleportation.py example at:

https://github.com/quantumlib/Cirq/tree/master/examples
"""

# Imports
import random
import cirq


def make_quantum_teleportation_circuit(ranX, ranY):
   """Returns a quantum teleportation circuit."""
   circuit = cirq.Circuit()
   msg, alice, bob = cirq.LineQubit.range(3)

   # Creates Bell state to be shared between Alice and Bob
   circuit.append([cirq.H(alice), cirq.CNOT(alice, bob)])

   # Creates a random state for the Message
   circuit.append([cirq.X(msg)**ranX, cirq.Y(msg)**ranY])

   # Bell measurement of the Message and Alice’s entangled qubit
   circuit.append([cirq.CNOT(msg, alice), cirq.H(msg)])
   circuit.append(cirq.measure(msg, alice))

   # Uses the two classical bits from the Bell measurement to
       recover the
   # original quantum Message on Bob’s entangled qubit
   circuit.append([cirq.CNOT(alice, bob), cirq.CZ(msg, bob)])

   return msg, circuit


def main():
   # Encode a random state to teleport
   ranX = random.random()
   ranY = random.random()
   msg, circuit = make_quantum_teleportation_circuit(ranX, ranY)

   # Simulate the circuit
   sim = cirq.Simulator()
   message = sim.simulate(cirq.Circuit.from_ops(
      [cirq.X(msg)**ranX, cirq.Y(msg)**ranY]))

   # Print the Bloch Sphere of Alice’s qubit
   print("Bloch Sphere of Alice’s qubit:")
   b0X, b0Y, b0Z = cirq.bloch_vector_from_state_vector(
      message.final_state, 0)
   print("x: ", round(b0X, 4),
        "y: ", round(b0Y, 4),
        "z: ", round(b0Z, 4))

   # Display the teleportation circuit
   print("\nCircuit:")
   print(circuit)

   # Record the final state of the simulation
   final_results = sim.simulate(circuit)

   # Print the Bloch sphere of Bob’s qubit
   print("\nBloch Sphere of Bob’s qubit:")
   b2X, b2Y, b2Z = cirq.bloch_vector_from_state_vector(
      final_results.final_state, 2)
   print("x: ", round(b2X, 4),
        "y: ", round(b2Y, 4),
        "z: ", round(b2Z, 4))


if __name__ == ’__main__’:
   main()
```

An example output of this program is shown below.

```
Bloch sphere of Alice’s qubit:
x: 0.654 y: -0.6177 z: -0.4367

Bloch sphere of Bob’s qubit:
x: 0.654 y: -0.6177 z: -0.4367
```

As can be seen, the Bloch sphere components of Alice and Bob’s qubit are identical – in other words, the qubit has been “teleported” from Alice to Bob.

A program for superdense coding written in Cirq is provided below.

```
"""Superdense coding in Cirq."""

# Imports
import cirq

# Helper function for visualizing output
def bitstring(bits):
   return ’’.join(’1’ if e else ’0’ for e in bits)

# Create two quantum and classical registers
qreg = [cirq.LineQubit(x) for x in range(2)]
circ = cirq.Circuit()

# Dictionary of operations for each message
message = {"00": [],
        "01": [cirq.X(qreg[0])],
        "10": [cirq.Z(qreg[0])],
        "11": [cirq.X(qreg[0]), cirq.Z(qreg[0])]}

# Alice creates a Bell pair
circ.append(cirq.H(qreg[0]))
circ.append(cirq.CNOT(qreg[0], qreg[1]))

# Alice picks a message to send
m = "01"
print("Alice’s sent message =", m)

# Alice encodes her message with the appropriate quantum operations
circ.append(message[m])

# Bob measures in the Bell basis
circ.append(cirq.CNOT(qreg[0], qreg[1]))
circ.append(cirq.H(qreg[0]))
circ.append([cirq.measure(qreg[0]), cirq.measure(qreg[1])])
```

```
# Print out the circuit
print("\nCircuit:")
print(circ)

# Run the quantum circuit on a simulator backend
sim = cirq.Simulator()
res = sim.run(circ, repetitions=1)

# Print out Bob's received message: the outcome of the circuit
print("\nBob's received message =",
    bitstring(res.measurements.values()))
```

An example output of this program for the message 01 is shown below.

```
Alice's sent message = 01

Circuit:
0: ---H---@---X---@---H---M---
          |       |
1: -------X-------X-------M---

Bob's received message = 01
```

As can be seen, Bob's received message is exactly Alice's sent message via superdense coding.

7.4 *Bell Inequality Test*

Now let's turn to another code walk-through: the Bell inequality test. After briefly describing the experiment, we will go through a complete program in Cirq to simulate it.

The Bell inequality test is best understood through a cooperative game involving two players, Alice and Bob, that make decisions based on input from a referee. Alice and Bob are separated (sitting in different rooms, say) and cannot communicate during the game. At each round of the game, the referee sends one bit to Alice, call it x, and one bit to Bob, call it y. Depending on the value of the bit, Alice sends a bit of her own, $a(x)$, back to the referee. Similarly, Bob sends a bit of his own, $b(y)$, back to the referee. The referee looks at both bits and decides if Alice and Bob win or lose that round. The condition for winning the round is

$$a(x) \oplus b(y) = xy$$

where $\oplus$ denotes addition modulo-2 (or, equivalently, XOR).

Alice and Bob's goal is to win as many rounds as possible. Although they cannot communicate *during* the game, they are allowed to meet *before* the game and set up a strategy. An example strategy might be "Alice always sends back $a(x) = x$, and Bob always sends back $a(y) = 0$." Since each of $a(x)$ and $b(y)$ can have two possible values, there are four possible deterministic strategies that Alice and Bob can implement. Additionally, since there are only four bits involved in the entire game, it's not difficult to enumerate all possible outcomes and see which strategy allows Alice and Bob to win the most rounds (or, equivalently, win each round with the highest probability).

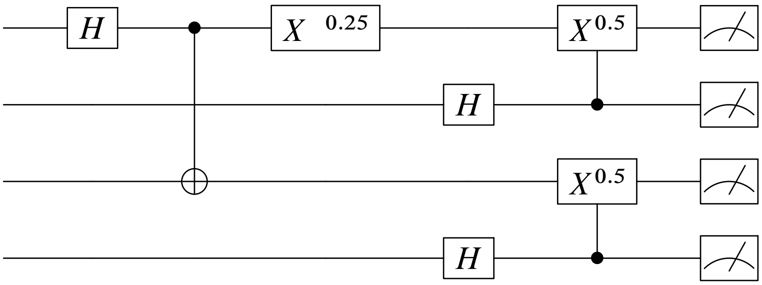
x	y	$a(x)$	$b(y)$	$a(x) \oplus b(y)$	xy	Win?	Strategy #
0	0	0	0	0	0	Yes	#1
0	0	0	1	1	0	No	#2
0	0	1	0	1	0	No	#3
0	0	1	1	0	0	Yes	#4
0	1	0	0	0	0	Yes	#1
0	1	0	1	1	0	No	#2
0	1	1	0	1	0	No	#3
0	1	1	1	0	0	Yes	#4
1	0	0	0	0	0	Yes	#1
1	0	0	1	1	0	No	#2
1	0	1	0	1	0	No	#3
1	0	1	1	0	0	Yes	#4
1	1	0	0	0	1	No	#1
1	1	0	1	1	1	Yes	#2
1	1	1	0	1	1	Yes	#3
1	1	1	1	0	1	No	#4

Table 7.1: All possible outcomes of the Bell inequality test game: the first two columns show bits the referee sends to Alice (x) and Bob (y). The next two columns show Alice's response ($a(x)$) and Bob's response ($b(y)$). The next column computes $a(x) \oplus b(y)$, and the next column shows the product xy. If these are equal, Alice and Bob win. The final column shows which strategy Alice and Bob are using in each row, numbered #1 - #4.

Table 7.1 enumerates all possible outcomes of the game. By analyzing each strategy, one can see that Alice and Bob win with *at most* 75% of the time. The strategies that achieve this win percentage are #1, in which $a(x) = b(y) = 0$, and #4 in which $a(x) = b(y) = 1$. Thus, the best possible classical strategy wins at most 75% of the time.

An interesting phenomena happens when we allow a *quantum strategy* between Alice and Bob. By a quantum strategy, we mean Alice and Bob are allowed to use entanglement as a resource in their strategy. As we have seen in this book, entanglement allows for stronger than classical correlations in physical systems. If Alice and Bob are allowed to share entangled qubits, they can remarkably win the Bell inequality test game with a higher probability! The best quantum strategy achieves a winning probability of $\cos^2(\pi/8)$, or about 85%.

The quantum strategy for this game is shown in the circuit diagram below. Here, the top (first) qubit belongs to Alice, and the third qubit from the top belongs to Bob. The first part of the circuit creates entanglement between Alice and Bob's qubits. Then, the referee "sends" in a random bit to Alice and Bob. In the circuit, this is done by performing a Hadamard operation on a "fresh qubit" (one in the $|0\rangle$ state) to produce equal superposition. Alice and Bob then perform a controlled-$\sqrt{X}$ operation on their qubits and measure to record the results.



Below, we show the complete program in Cirq for setting up this circuit and simulating the quantum strategy for the Bell inequality test.

```
"""Creates and simulates a circuit equivalent to a Bell inequality
    test."""

# Imports
import numpy as np

import cirq

def main():
   # Create circuit
   circuit = make_bell_test_circuit()
   print('Circuit:')
   print(circuit)

   # Run simulations
   print()
   repetitions = 1000
   print('Simulating {} repetitions...'.format(repetitions))
   result = cirq.Simulator().run(program=circuit,
                                  repetitions=repetitions)
   # Collect results
   a = np.array(result.measurements[’a’][:, 0])
   b = np.array(result.measurements[’b’][:, 0])
   x = np.array(result.measurements[’x’][:, 0])
   y = np.array(result.measurements[’y’][:, 0])

   # Compute the winning percentage
   outcomes = a ^ b == x & y
   win_percent = len([e for e in outcomes if e]) * 100 / repetitions

   # Print data
   print()
   print(’Results’)
   print(’a:’, bitstring(a))
   print(’b:’, bitstring(b))
   print(’x:’, bitstring(x))
   print(’y:’, bitstring(y))
   print(’(a XOR b) == (x AND y):\n ’, bitstring(outcomes))
   print(’Win rate: {}%’.format(win_percent))


def make_bell_test_circuit():
   # Qubits for Alice, Bob, and referees
   alice = cirq.GridQubit(0, 0)
   bob = cirq.GridQubit(1, 0)
   alice_referee = cirq.GridQubit(0, 1)
   bob_referee = cirq.GridQubit(1, 1)

   circuit = cirq.Circuit()

   # Prepare shared entangled state between Alice and Bob
   circuit.append([
      cirq.H(alice),
      cirq.CNOT(alice, bob),
      cirq.X(alice)**-0.25,
   ])

   # Referees flip coins
   circuit.append([
      cirq.H(alice_referee),
      cirq.H(bob_referee),
   ])

   # Players do a sqrt(X) based on their referee’s coin
   circuit.append([
      cirq.CNOT(alice_referee, alice)**0.5,
      cirq.CNOT(bob_referee, bob)**0.5,
   ])

   # Then results are recorded
   circuit.append([
      cirq.measure(alice, key=’a’),
      cirq.measure(bob, key=’b’),
      cirq.measure(alice_referee, key=’x’),
      cirq.measure(bob_referee, key=’y’),
   ])
   return circuit


def bitstring(bits):
   return ''.join('1' if e else '_' for e in bits)


if __name__ == '__main__':
   main()
```

An example output of this program is shown below, where we simulate 75 repetitions only to visualize the output more easily. The output of this program is interpreted as follows. The bitstrings for Alice and Bob are shown in the first line. These are the bits each player sends back to the referee (the underscore indicates the 0 bit). The next lines show the bitstrings sent to Alice (x) and Bob (y). Finally, the winning condition $a(x) \oplus b(y) = xy$ is shown as a bitstring for each round. The win rate is computed from the number of 1s in this bitstring, which indicate a win.

```
Simulating 75 repetitions...

Results
a: 1_1111_1_1111_11_1111_1_1______1_1__
11_1__111_1_1_111_1111__11_1111_1_1____

b: 1_1__11_1_1_1_11__11__1_1_1_____1___
_11__11_1_1___11__1__1111_1_1___1__11_1

x: 11_11_1_1111_____1_11____11_1__111_1
1__1_11_1___11_111111111__11__1_111__11

y: _1_11111111__11_11__1__1111___1111__
1__111_1__111___1__11__1_111_11___1___1

(a XOR b) == (x AND y):
  1_11111_11__1111111111111_11111111111
  1_1111_11111111111111_11_111_11111__11
Win rate: 84.0%
```

As can be seen, we achieve a win rate of 84% in this example, which is greater than possible with purely classical strategies. Note that using a larger number of repetitions (i.e., simulating more rounds) is recommended to see asymptotic convergence to the optimal win rate of $\cos^2(\pi/8) \approx 85\%$.

Summary

Quantum teleportation, superdense coding and Bell's inequality test are some of the most intriguing circuits for quantum processors. Now let us turn to the canon of quantum algorithms that demonstrated the potential of quantum advantage over classical computing.

CHAPTER

8

The Canon: Code Walkthroughs

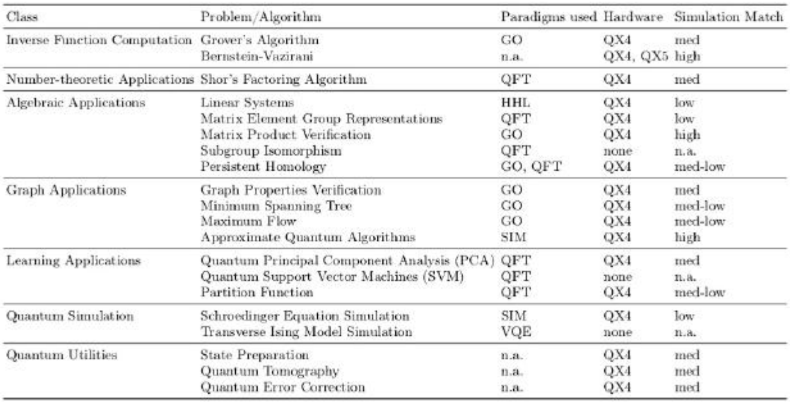
In this chapter, we will walk through a number of fundamental quantum algorithms. We call these algorithms *the canon* as they were all developed in the early years of quantum computing and were the first to establish provable computational speedups with quantum computers. We discussed most of these algorithms at a high level in chapter 2; we will now walk through them in a more detailed manner. A number of these algorithms require a quantum computer that is still in the future, but by analyzing them now we can deepen our understanding of what will be possible. Additionally, variants of these algorithms can be used to prove advantages with near-term quantum computers in the noiseless [47] and even noisy [48] regimes.

Several of the algorithms considered in this chapter are known as "black box" or "query model" quantum algorithms. In these cases, there is an underlying function which is unknown to us. However, we are able to construct another function, called an oracle, which we can query to determine the relationship of specific inputs with specific outputs. More specifically, we can query the oracle function with specific inputs in the quantum register and reversibly write the output of the oracle function into that register. That is, we have access to an oracle O_f such that

$$O_f|x\rangle = |x \oplus f(x)\rangle \tag{8.1}$$

where $\oplus$ denotes addition modulo-2. It's easy to see that O_f is unitary (reversible) because it is self-inverse. This can seem like "cheating" at first — how could we construct a circuit to perform O_f? And how could we know it's an efficient circuit? One reason to think about quantum algorithms in the query model is because it provides a lower bound on the number of steps (gates). Each query is *at least* one step in the algorithm, so if it cannot be done efficiently with queries, it can certainly not be done efficiently with gates. Thus, the query model can be useful for ruling out fast quantum algorithms.

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_8



Class	Problem/Algorithm	Paradigms used	Hardware	Simulation Match
Inverse Function Computation	Grover's Algorithm	GO	QX4	med
	Bernstein-Vazirani	n.a.	QX4, QX5	high
Number-theoretic Applications	Shor's Factoring Algorithm	QFT	QX4	med
Algebraic Applications	Linear Systems	HHL	QX4	low
	Matrix Element Group Representations	QFT	QX4	low
	Matrix Product Verification	GO	QX4	high
	Subgroup Isomorphism	QFT	none	n.a.
	Persistent Homology	GO, QFT	QX4	med-low
Graph Applications	Graph Properties Verification	GO	QX4	med
	Minimum Spanning Tree	GO	QX4	med-low
	Maximum Flow	GO	QX4	med-low
	Approximate Quantum Algorithms	SIM	QX4	high
Learning Applications	Quantum Principal Component Analysis (PCA)	QFT	QX4	med
	Quantum Support Vector Machines (SVM)	QFT	none	n.a.
	Partition Function	QFT	QX4	med-low
Quantum Simulation	Schroedinger Equation Simulation	SIM	QX4	low
	Transverse Ising Model Simulation	VQE	none	n.a.
Quantum Utilities	State Preparation	n.a.	QX4	med
	Quantum Tomography	n.a.	QX4	med
	Quantum Error Correction	n.a.	QX4	med

Figure 8.1: Overview of studied quantum algorithms; paradigms include: Grover Operator (GO), Quantum Fourier Transform (QFT), Harrow/Hassidim/Lloyd (HHL), Variational Quantum Eigensolver (VQE), and direct Hamiltonian simulation (SIM). The simulation match column indicates how well the hardware quantum results matched the simulator results. Table and Caption Source: [61]

However, the query model can also be used to prove fast quantum algorithms *relative* to the oracle. We can give both a quantum computer and a classical computer access to the same oracle and see which performs better. It's possible to prove lower bounds or exact expressions for the number of queries in the classical and quantum cases, thereby making it possible to prove computational advantages relative to oracles. Examples of quantum algorithms with provable relativized speedups include Deutsch's algorithm and the Berstein-Vazirani algorithm.

Finally, if one can find a way to *instantiate* the oracle in a number of gates that scales polynomially in the size of the input register, one can find "true" (i.e., not relativized) quantum speedups. This is the case with Shor's algorithm for quantum factoring.[1] Shor's algorithm built off previous work in query model algorithms for artificial problems. Shor was able to modify this work and instantiate the oracle to construct an explicit (i.e., not involving oracle access) algorithm for factoring. Factoring is markedly *not* an artificial problem — it is exceedingly important as we base most of our public-key cryptography on the belief that factoring is hard to do! Section 8.5 discusses this further.

Before this, we discuss the historical quantum algorithms leading up to Shor's algorithm. We note that these black box/oracle algorithms are one

[1] The classical complexity of factoring has not been proven to be intractable, but it is widely believed to be so since no one has yet discovered an efficient algorithm.

particular class of quantum algorithms. Other algorithm classes such as quantum simulation are shown in Figure 8.1. In upcoming chapters, we will cover additional applications and code for algorithms that are focused on the NISQ-regime processors. For now, though, we cover the canon.

8.1 *The Deutsch-Jozsa Algorithm*

Deutsch's algorithm was the first to demonstrate a clear advantage of quantum over classical computing. In Deutsch's problem we are given a black box which computes a one-bit boolean function. That is, a function which takes in one bit and outputs one bit. We can represent the function f as

$$f : \{0, 1\} \to \{0, 1\} \tag{8.2}$$

We can imagine, for example, as David Deutsch has pointed out, that the black box function is computing some complicated function such as a routing algorithm and the output (0 or 1) indicates which route is chosen [66].

There are exactly four one-input, one-output Boolean functions:[2]

x	f_0	f_1	f_x	$f_{\bar{x}}$
0	0	1	0	1
1	0	1	1	0

The first two of these are *constant* functions, f_0 and f_1. That is, they always output a *constant* value. We call the other two, f_x and $f_{\bar{x}}$, *balanced*. Over their inputs 0 and 1, they have an equal number of 0s and 1s in their truth table.

We can now state Deutsch's problem:

> Given access to a *one bit* input and *one bit* output Boolean function, determine, by querying the function as few times as possible, whether the function is *balanced* or *constant*.

If you were to approach this with classical tools, you would have to query the function at least twice: first to see the output when the input is 0 and then the output when the input is 1. The remarkable discovery by David Deutsch is that you only need one query on a quantum computer! The original Deutsch algorithm handles this case of a one-bit Boolean oracle [65], and the

[2] Code and exposition of the DJ algorithm adapted from [60].

Deutsch-Jozsa (DJ) algorithm generalizes the approach to handle Boolean functions of n inputs [67]. It's not difficult to see that, with classical tools, one must query an n bit Boolean function at least n times. The DJ algorithm solves this problem in only one query.

8.3 Quantum Advantage Demonstrated by Deutsch-Jozsa

Classically, one must query the one-bit Boolean function twice to distinguish the constant function from the balanced function. For an n-bit Boolean function, one must query n times. On a quantum computer using DJ one only has to query once.

We now turn to the quantum approach to this problem. Above, we have described a classical function on bits that is not reversible, e.g., the constant functions f_0 and f_1. That is, knowing the value of the output does not allow us to determine uniquely the value of the input. In order to run this on a quantum computer, however we need to make this computation reversible. A trick for taking a classical non-reversible function and making it reversible is to compute the value in an extra register (or, equivalently, store inputs to the function in an extra register). Suppose we have an n bit input x and we are computing a (potentially non-reversible) Boolean function $f(x)$. Then we can implement this via a unitary U_f that acts on $n + 1$ qubits

$$U_f\left(|x\rangle|y\rangle\right) := |x\rangle|y \oplus f(x)\rangle$$

Here the symbol $\oplus$ denotes addition modulo-2 (a.k.a. *XOR*); in the expression above, we have identified how U_f acts on all computational basis states $|x\rangle$ and on the single output qubit $|y\rangle$. Since we have defined U_f on the computational basis, we extend its definition to all vectors in the state space by linearity. To see that this is reversible, one can note that applying the transformation twice returns the state to its original form. Even further, U_f is unitary since it maps the orthonormal computational basis to itself and so preserves the length of the vectors that it acts on.

One core idea in this algorithm is that we will measure in a different basis than the computational basis. If we measure in the computational basis (e.g., the z-basis) then we will gain no quantum advantage as we have two basis states, $|0\rangle$ and $|1\rangle$, and those correspond to the classical bits, 0 and 1. One of the tricks that makes this algorithm work is that we will measure in the Hadamard basis state which is a superposition of $|0\rangle$ and $|1\rangle$ [137]. Figure 8.2 shows a circuit diagram of DJ.

Let's see how to implement these functions in Cirq. f_0 enacts the transform

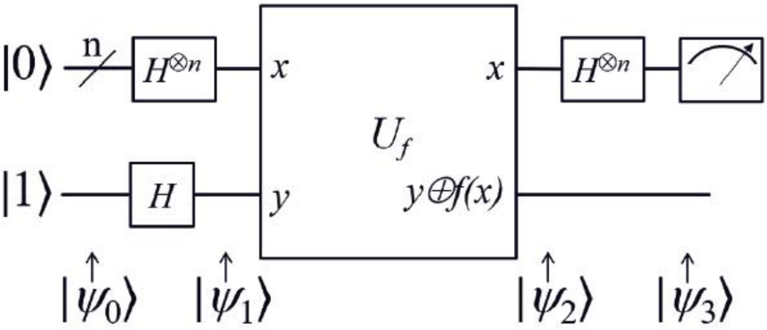


Figure 8.2: The DJ circuit   Source: Wikimedia

$$\begin{gathered} |00\rangle \rightarrow |00\rangle \\ |01\rangle \rightarrow |01\rangle \\ |10\rangle \rightarrow |10\rangle \\ |11\rangle \rightarrow |11\rangle \end{gathered}$$

This is just the identity transform, i.e., an empty circuit. f_1 enacts the transform

$$\begin{gathered} |00\rangle \rightarrow |01\rangle \\ |01\rangle \rightarrow |00\rangle \\ |10\rangle \rightarrow |11\rangle \\ |11\rangle \rightarrow |10\rangle \end{gathered}$$

This is a bit flip gate on the second qubit.

To gain an understanding of how this newly defined reversible operator works, we will compute $U_{f_x}\left(|0\rangle|0\rangle\right)$ as an example to demonstrate. Recall that $|00\rangle$ is shorthand for $|0\rangle|0\rangle$. Then, by definition,

$$U_{f_x}\left(|00\rangle\right) = U_{f_x}\left(|0\rangle|0\rangle\right) := |0\rangle|0 \oplus f_x(0)\rangle = |0\rangle|0 \oplus 1\rangle = |0\rangle|1\rangle$$

It is worthwhile to compute U_{f_x} for each of $|0\rangle|1\rangle$, $|1\rangle|0\rangle$ and $|1\rangle|1\rangle$; then let us check that f_x enacts the transform

$$\begin{gathered} |00\rangle \rightarrow |00\rangle \\ |01\rangle \rightarrow |01\rangle \\ |10\rangle \rightarrow |11\rangle \\ |11\rangle \rightarrow |10\rangle \end{gathered}$$

This is nothing more than a *CNOT* from the first qubit to the second qubit. Finally $f_{\bar{x}}$ enacts the transform

$$\begin{aligned} |00\rangle &\rightarrow |01\rangle \\ |01\rangle &\rightarrow |00\rangle \\ |10\rangle &\rightarrow |10\rangle \\ |11\rangle &\rightarrow |11\rangle \end{aligned}$$

which is a *CNOT* from the first qubit to the second qubit followed by a bit flip on the second qubit. We can encapsulate these functions into a dictionary which maps oracle names to the operations in the circuit needed to enact this function:

```
# Import the Cirq Library
import cirq

# Get two qubits, a data qubit and target qubit, respectively
q0, q1 = cirq.LineQubit.range(2)

# Dictionary of oracles
oracles = {’0’: [], ’1’: [cirq.X(q1)], ’x’: [cirq.CNOT(q0, q1)],
        ’notx’: [cirq.CNOT(q0, q1), cirq.X(q1)]}
```

Let us turn now to Deutsch’s algorithm. Suppose we are given access to the reversible oracle functions we have defined before. By a similar argument for our irreversible classical functions, you can show that you cannot distinguish the balanced from the constant functions by using this oracle only once. But now we can ask the question: what if we are allowed to query this function in superposition, i.e., what if we can use the power of quantum computing?

Deutsch was able to show that you could solve this problem with quantum computers using only a single query of the function. To see how this works, we need two simple insights. Suppose we prepare the second qubit in the superposition state

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

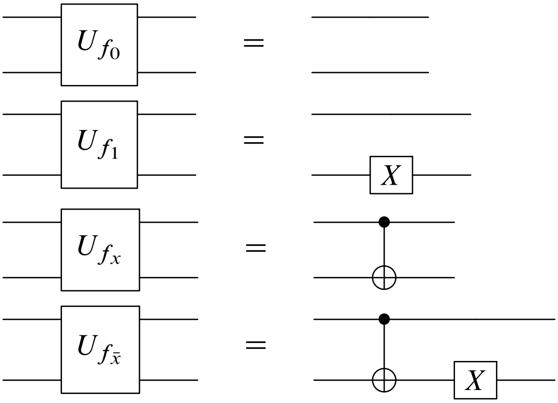
and apply the oracle. Using the linearity of the operator U_f to acquire the second equation and an observation for the third, we can check that

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

This is the *phase kickback trick.* By applying U_f onto a target which is in superposition, the value of the function ends up in the global phase, thus *kicking back* the information we need on whether the function is constant or balanced; *the information is encoded in the phase.*

How can we leverage this to distinguish between constant and balanced functions? Note that for constant functions the phase that is applied is the same for all inputs $|x\rangle$, whereas for balanced functions the phase is different for each value of x. To use the phase kickback trick for each of the oracles, we apply the following transform on the first qubit

$$\begin{aligned} f_0 &\to I \\ f_1 &\to -I \\ f_x &\to Z \\ f_{\bar{x}} &\to -Z \end{aligned}$$

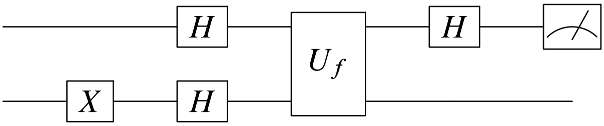


Now we only need to distinguish between the identity gate and the Z gate on the first qubit; we can do this by recalling that:

$$HZH = X$$

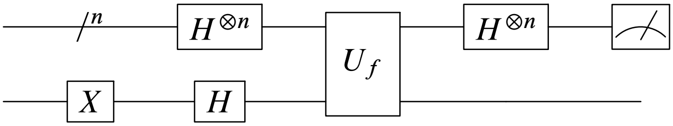
$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

This means that we can turn a phase flip into a bit flip by applying Hadamards before and after the phase flip. If we look at the constant and balanced functions we see that the constant functions will be proportional to I and the balanced will be proportional to X. If we feed in $|0\rangle$ to this register, then in the first case we will only see $|0\rangle$ and in the second case we will see $|1\rangle$. *In other words we will be able to distinguish constant from balanced using a single query of the oracle.*



```
# Import the Cirq Library
import cirq

# Get two qubits, a data qubit and target qubit, respectively
q0, q1 = cirq.LineQubit.range(2)

# Dictionary of oracles
oracles = {'0': [], '1': [cirq.X(q1)], 'x': [cirq.CNOT(q0, q1)],
    'notx': [cirq.CNOT(q0, q1), cirq.X(q1)]}

def deutsch_algorithm(oracle):
   """Yields a circuit for Deustch's algorithm given operations
       implementing
   the oracle."""
   yield cirq.X(q1)
   yield cirq.H(q0), cirq.H(q1)
   yield oracle
   yield cirq.H(q0)
   yield cirq.measure(q0)

# Display each circuit for all oracles
for key, oracle in oracles.items():
   print('Circuit for {}...'.format(key))
   print(cirq.Circuit.from_ops(deutsch_algorithm(oracle)),
       end="\n\n")

# Get a simulator
simulator = cirq.Simulator()

# Execute the circuit for each oracle to distingiush constant from
    balanced
for key, oracle in oracles.items():
   result = simulator.run(
      cirq.Circuit.from_ops(deutsch_algorithm(oracle)),
      repetitions=10
   )
   print('oracle: {:<4} results: {}'.format(key, result))
```

Now let's extend the Deutsch problem to Boolean functions of n Boolean inputs, not just single-input functions as above. We saw that with the Deutsch algorithm we can double our speed, going from two queries classically to one query quantum mechanically.



If we are able to query an n-bit oracle just once and determine whether the function is constant or balanced we have a time complexity of $O(1)$, a significant speedup over $O(n)$ queries. All Boolean functions for one-bit input are either constant or balanced. For Boolean functions with two input bits there are two constant functions, $f(x_0, x_1) = 0$ and $f(x_0, x_1) = 1$, while there are $\binom{4}{2} = 6$ balanced functions. The following code gives you the operations for querying these functions.

```
"""Deustch-Jozsa algorithm on three qubits in Cirq."""

# Import the Cirq library
import cirq

# Get three qubits -- two data and one target qubit
q0, q1, q2 = cirq.LineQubit.range(3)

# Oracles for constant functions
constant = ([], [cirq.X(q2)])

# Oracles for balanced functions
balanced = ([cirq.CNOT(q0, q2)],
         [cirq.CNOT(q1, q2)],
         [cirq.CNOT(q0, q2), cirq.CNOT(q1, q2)],
         [cirq.CNOT(q0, q2), cirq.X(q2)],
         [cirq.CNOT(q1, q2), cirq.X(q2)],
         [cirq.CNOT(q0, q2), cirq.CNOT(q1, q2), cirq.X(q2)])

def your_circuit(oracle):
   """Yields a circiut for the Deustch-Jozsa algorithm on three
       qubits."""
   # phase kickback trick
   yield cirq.X(q2), cirq.H(q2)

   # equal superposition over input bits
   yield cirq.H(q0), cirq.H(q1)

   # query the function
   yield oracle

   # interference to get result, put last qubit into |1>
   yield cirq.H(q0), cirq.H(q1), cirq.H(q2)

   # a final OR gate to put result in final qubit
   yield cirq.X(q0), cirq.X(q1), cirq.CCX(q0, q1, q2)
   yield cirq.measure(q2)


# Get a simulator
simulator = cirq.Simulator()

# Execute circuit for oracles of constant value functions
print('Your result on constant functions')
for oracle in constant:
   result =
       simulator.run(cirq.Circuit.from_ops(your_circuit(oracle)),
       repetitions=10)
   print(result)

# Execute circuit for oracles of balanced functions
print('Your result on balanced functions')
for oracle in balanced:
   result =
       simulator.run(cirq.Circuit.from_ops(your_circuit(oracle)),
       repetitions=10)
   print(result)
```

We can now see how to query an oracle of n-bit boolean inputs to check whether it is constant or balanced.

8.2 The Bernstein-Vazirani Algorithm

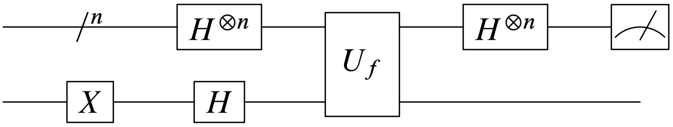
Now let's turn to the Bernstein-Vazirani (BV) algorithm that we considered earlier in this book [29]. As with DJ, the goal of BV is also to ascertain the nature of a black-box Boolean function. While it is true that DJ demonstrates an advantage of quantum over classical computing, if we allow for a small error rate, then the advantage disappears: both classical and quantum approaches are in the order of $O(1)$ time complexity [137].

BV was the first algorithm developed that shows a clear separation between quantum and classical computing even allowing for error, i.e., a true non-deterministic speedup. Here is the BV problem statement:

> Given an unknown function of n inputs:
>
> $$f(x_{n\ \ 1}, x_{n\ \ 2}, ..., x_1, x_0),$$
>
> let a be an unknown non-negative integer less than 2^n. Let $f(x)$ take any other such integer x and modulo-2 sum x multiplied by a. So the output of the function is:
>
> $$a \cdot x = a_0 x_0 \oplus a_1 x_1 \oplus a_2 x_2 ....$$
>
> Find a in one query of the oracle [151].

Just as in DJ, we prepare the states of two sets of qubits: the data register qubits and the target qubit. The data register qubits are set to $|0\rangle$ and the target set to $|1\rangle$. We then apply H to both sets of qubits to put them in superposition. H applied to the data register qubits prepares us to measure in the X basis.

We then apply the unitary U_f, an H to the data register qubits and then measure those qubits. Since we only applied U_f once, the time complexity of BV is $O(1)$.



The Bernstein-Vazirani Algorithm

```
"""Bernstein-Vazirani algorithm in Cirq."""

# Imports
import random

import cirq


def main():
    """Executes the BV algorithm."""
    # Number of qubits
    qubit_count = 8

    # Number of times to sample from the circuit
    circuit_sample_count = 3

    # Choose qubits to use
    input_qubits = [cirq.GridQubit(i, 0) for i in range(qubit_count)]
    output_qubit = cirq.GridQubit(qubit_count, 0)

    # Pick coefficients for the oracle and create a circuit to query
        it
    secret_bias_bit = random.randint(0, 1)
    secret_factor_bits = [random.randint(0, 1) for _ in
        range(qubit_count)]
    oracle = make_oracle(input_qubits,
                         output_qubit,
                         secret_factor_bits,
                         secret_bias_bit)
    print('Secret function:\nf(x) = x*<{}> + {} (mod 2)'.format(
        ', '.join(str(e) for e in secret_factor_bits),
        secret_bias_bit))

    # Embed the oracle into a special quantum circuit querying it
        exactly once
    circuit = make_bernstein_vazirani_circuit(
        input_qubits, output_qubit, oracle)
    print('\nCircuit:')
    print(circuit)

    # Sample from the circuit a couple times
    simulator = cirq.Simulator()
```

```
   result = simulator.run(circuit, repetitions=circuit_sample_count)
   frequencies = result.histogram(key='result', fold_func=bitstring)
   print('\nSampled results:\n{}'.format(frequencies))

   # Check if we actually found the secret value.
   most_common_bitstring = frequencies.most_common(1)[0][0]
   print('\nMost common matches secret factors:\n{}'.format(
      most_common_bitstring == bitstring(secret_factor_bits)))


def make_oracle(input_qubits,
            output_qubit,
            secret_factor_bits,
            secret_bias_bit):
   """Gates implementing the function f(a) = a*factors + bias (mod
       2)."""
   if secret_bias_bit:
      yield cirq.X(output_qubit)

   for qubit, bit in zip(input_qubits, secret_factor_bits):
      if bit:
         yield cirq.CNOT(qubit, output_qubit)




def make_bernstein_vazirani_circuit(input_qubits, output_qubit,
    oracle):
   """Solves for factors in f(a) = a*factors + bias (mod 2) with one
       query."""
   c = cirq.Circuit()

   # Initialize qubits
   c.append([
      cirq.X(output_qubit),
      cirq.H(output_qubit),
      cirq.H.on_each(*input_qubits),
   ])

   # Query oracle
   c.append(oracle)

   # Measure in X basis
   c.append([
      cirq.H.on_each(*input_qubits),
      cirq.measure(*input_qubits, key='result')
   ])

   return c


def bitstring(bits):
   """Creates a bit string out of an iterable of bits."""
   return ''.join(str(int(b)) for b in bits)


if __name__ == '__main__':
    main()
```

Here we initialize the target (or output) register to $|1\rangle$ with an X operator and the data register qubits from state $|0\rangle$ to the $|+\rangle$ / $|-\rangle$ basis by applying H. We then query the oracle, apply H to each of the input qubits and measure the input qubits. This gives us the answer that we were seeking, a, in one query. Thus, no matter how many inputs we have, we can perform this algorithm in $O(1)$ time.

```
"""
=== EXAMPLE OUTPUT for BV ===
Secret function:
f(x) = x*<0, 1, 1, 1, 0, 0, 1, 0> + 1 (mod 2)

Sampled results:
Counter({'01110010': 3})
Most common matches secret factors:
True
"""
```

8.3 *Simon's Problem*

Soon after BV's result, Daniel Simon demonstrated the ability to determine the periodicity of a function exponentially faster on a quantum computer compared with a classical one.

Let us recall that a function can map two different inputs to the same output *but must not* map the same input to two different outputs. In other words, 2:1 is acceptable, but 1:2 is not. For example, the function $f(x) = x^2$ which squares each input is in fact a function; two different inputs, namely 1 and -1 both map to 1, i.e., $f(x)$ is 2:1. See Part III for a review of functions and injectivity, surjectivity and bijectivity.

If we determine that the function is of the 2:1 type, then our next challenge is to investigate the period of the function (see section 8.5 for an explanation of periodicity); this is the core objective of Simon's problem. Here is an outline of the problem in more formal language:

> The problem considers an oracle that implements a function mapping an n-bit string to an m-bit string $f{:}\{0,1\}^n \rightarrow \{0,1\}^m$, with $m \geq n$, where it is promised that f is a 1:1 type function (each input gives a different output) or 2:1 type function (two inputs give the same

> output) with non-zero period $s \in \{0, 1\}^n$ such that for all x, x_0 we have $f(x) = f(x_0)$ if and only if $x_0 = x \oplus s$, where $\oplus$ corresponds to addition modulo-2.[3] The problem is to determine the type of the function f and, if it is 2:1, to determine the period s. [215]

Simon's problem is in the same vein as Deutsch's problem — you are presented with an oracle — a black box function where you can observe the output for specific inputs, but not the underlying function. The challenge is to determine if this black-box process ever sends two different inputs to the same output, and if so, determine how often that occurs. Note that in these algorithms we do not find out what the underlying function in the black box is, only the relationship between the input and the output as seen from outside the box. The actual function may be quite complicated and may require even more steps to analyze than the input/output relationship. We explore the concept of oracles further in this chapter.

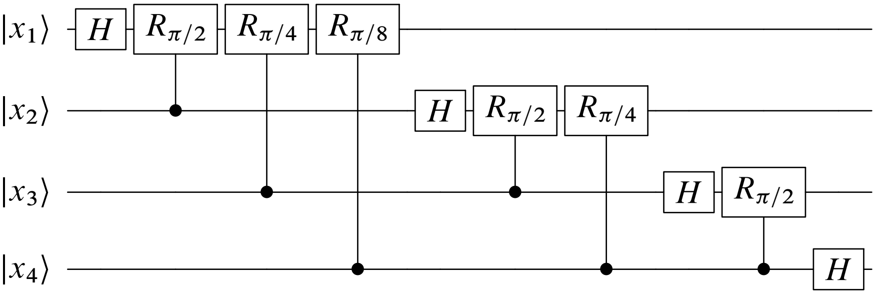
Simon was successful in proving that we can solve this problem to determine the periodicity of a function exponentially faster on a quantum computer compared with a classical one. Shor built on this key result in developing his algorithm for factoring large numbers when he realized that the two problems — finding the periodicity of a function and factoring a large composite number — are in fact isomorphic.

Twenty years after Simon presented his problem, researchers did in fact successfully use a quantum system to determine the periodicity of a function [215]. Check the book's website for sample code for Simon's problem.

8.4 *Quantum Fourier Transform*

In chapter 3, we discussed the Quantum Fourier Transform (QFT) as a method to set up the amplitudes for measurement that will favor the qubit which has the information we require. We can implement a QFT circuit on NISQ hardware in a straightforward manner. Here is the standard circuit diagram for QFT:

[3] When we do addition modulo-2, we equate 2 with 0. So, for example, $1 \oplus 1 = 0$ and we would say, "The sum of 1 and 1 modulo-2 is 0."



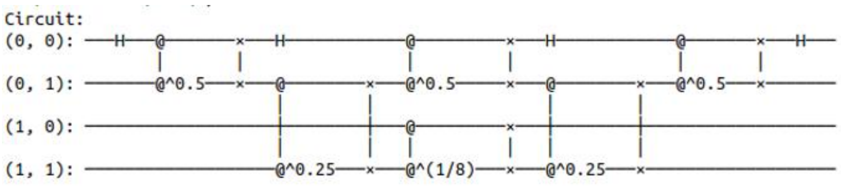
Now let's walk through the code for QFT as we will use this technique in the upcoming section on Shor's algorithm. We'll go through the program step by step, explaining first high-level details and then discussing implementation details.

First, we import the necessary packages for this program including the Cirq library.

```
"""Creates and simulates a circuit for Quantum Fourier Transform
    (QFT)
on a 4 qubit system.
"""

# Imports
import numpy as np

import cirq
```

Next, we define the main function, which simply calls the circuit generation function and then runs it — in this case on the simulator. The program then prints the final state of the wavefunction after the QFT has been applied.

```

def main():
    """Demonstrates the Quantum Fourier transform."""

    # Create circuit and display it
    qft_circuit = generate_2x2_grid_qft_circuit()
    print('Circuit:')
    print(qft_circuit)

    # Simulate and collect the final state
    simulator = cirq.Simulator()
    result = simulator.simulate(qft_circuit)

    # Display the final state
    print('\nFinalState')
    print(np.around(result.final_state, 3))
```



```
Circuit:
(0, 0): ────H───@────────×───H────────────@──────────×───H────────────@────────×───H───
                │        │                │          │                │        │
(0, 1): ────────@^0.5────×───@────────×───@^0.5──────×───@────────×───@^0.5────×───────
                             │        │                  │        │
(1, 0): ─────────────────────┼────────┼───@──────────×───┼────────┼────────────────────
                             │        │   │          │   │        │
(1, 1): ─────────────────────@^0.25───×───@^(1/8)────×───@^0.25───×────────────────────
```

Figure 8.3: Modified QFT circuit including SWAP operations fit for running on a 2x2 grid of qubits with nearest-neighbor interactions.

Next, we define a helper function for building the QFT circuit. This function yields a controlled-R_z rotation as well as a *SWAP* gate on the input qubits.

```
def cz_and_swap(q0, q1, rot):
   """Yields a controlled-RZ gate and SWAP gate on the input
       qubits."""
   yield cirq.CZ(q0, q1)**rot
   yield cirq.SWAP(q0,q1)
```

Finally, we use this helper function to write the entire circuit, which is done in the function below. First, we define a 2 x 2 grid of qubits and label them a through d. In many quantum computing systems there are constraints one which qubits can interact with each other. For example perhaps only nearest-neighbor qubits can interact. Then we cannot apply the standard QFT circuit described above. Instead, a modified QFT circuit including *SWAP* operations needs to be applied, as illustrated in Figure 8.3. This is the circuit we will implement in our example.

We apply a series of Hadamard and control rotation operators as specified by the diagram. In this example we perform the quantum Fourier transform on the $(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ vector, which means acting with the QFT circuit on the ground state $|0000\rangle$.

```
def generate_2x2_grid_qft_circuit():
   """Returns a QFT circuit on a 2 x 2 planar qubit architecture.

   Circuit adopted from https://arxiv.org/pdf/quant-ph/0402196.pdf.
   """
   # Define a 2*2 square grid of qubits
   a, b, c, d = [cirq.GridQubit(0, 0), cirq.GridQubit(0, 1),
             cirq.GridQubit(1, 1), cirq.GridQubit(1, 0)]

   # Create the Circuit
   circuit = cirq.Circuit.from_ops(
      cirq.H(a),
      cz_and_swap(a, b, 0.5),
      cz_and_swap(b, c, 0.25),
      cz_and_swap(c, d, 0.125),
      cirq.H(a),
      cz_and_swap(a, b, 0.5),
      cz_and_swap(b, c, 0.25),
      cirq.H(a),
      cz_and_swap(a, b, 0.5),
      cirq.H(a),
      strategy=cirq.InsertStrategy.EARLIEST
   )

   return circuit
```

Finally, we can run this circuit by calling the main function:

```
if __name__ == '__main__':
    main()
```

The output of this program is shown below:

```
FinalState
[0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j
    0.25+0.j
 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j
     0.25+0.j]
```

Figure 8.4 delineates the process of applying QFT.

8.5 *Shor’s Algorithm*

RSA Cryptography

Suppose Alice would like to send a private message to Bob via the internet. Alice’s message could very well be intercepted by a malicious eavesdropper, Eve, along its journey. This is embarrassing, yet harmless, if Alice is sending Bob a note, but it is an issue if Alice is sending Bob her credit card number. How can we send messages securely via the internet?

Cryptography is the study of the making and breaking of secret codes. Cryptography refers to the writing of secret codes, while cryptanalysis refers to the breaking of those codes. RSA cryptography is a popular style of cryptography that allows for the secure transfer of information via the internet. RSA honors Rivest, Shah and Adelman, three pioneers in its development [188].

The core conjecture of RSA cryptography is that multiplying two large prime numbers is a *trapdoor* function; multiplying two large prime numbers is easy, yet finding the two factors after the multiplication has occurred is hard. Later in this chapter, we’ll see that a fault-tolerant quantum computer will

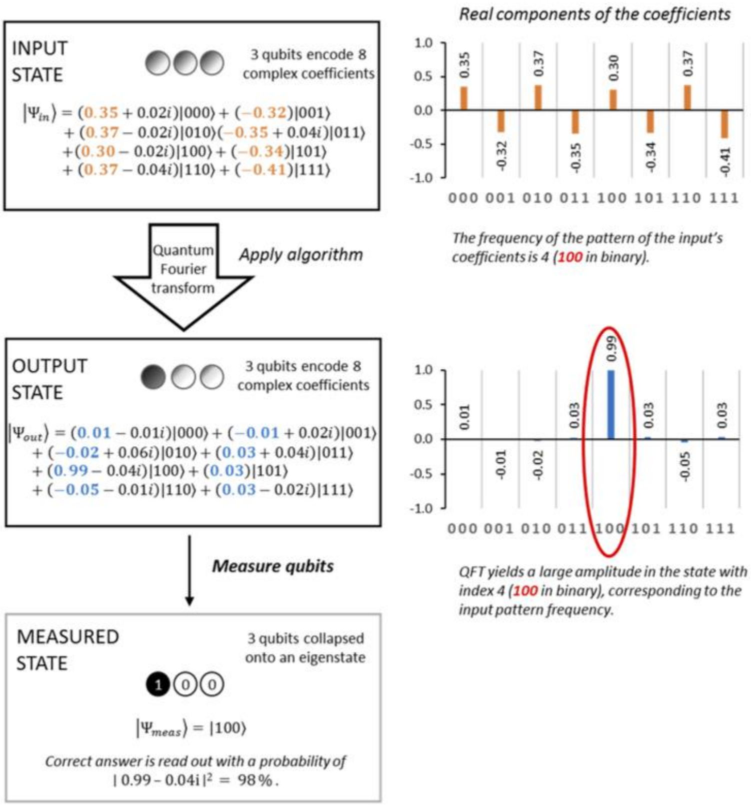

INPUT STATE
3 qubits encode 8 complex coefficients
|Ψ_in⟩ = (0.35 + 0.02i)|000⟩ + (−0.32)|001⟩ + (0.37 − 0.02i)|010⟩(−0.35 + 0.04i)|011⟩ +(0.30 − 0.02i)|100⟩ + (−0.34)|101⟩ + (0.37 − 0.04i)|110⟩ + (−0.41)|111⟩
Real components of the coefficients
1.0
0.5
0.0
-0.5
-1.0
0.35
-0.32
0.37
-0.35
0.30
-0.34
0.37
-0.41
000 001 010 011 100 101 110 111
The frequency of the pattern of the input's coefficients is 4 (100 in binary).
Quantum Fourier transform
Apply algorithm
OUTPUT STATE
3 qubits encode 8 complex coefficients
|Ψ_out⟩ = (0.01 − 0.01i)|000⟩ + (−0.01 + 0.02i)|001⟩ + (−0.02 + 0.06i)|010⟩ + (0.03 + 0.04i)|011⟩ + (0.99 − 0.04i)|100⟩ + (0.03)|101⟩ + (−0.05 − 0.01i)|110⟩ + (0.03 − 0.02i)|111⟩
1.0
0.5
0.0
-0.5
-1.0
0.01
-0.01
-0.02
0.03
0.99
0.03
-0.05
0.03
000 001 010 011 100 101 110 111
QFT yields a large amplitude in the state with index 4 (100 in binary), corresponding to the input pattern frequency.
Measure qubits
MEASURED STATE
3 qubits collapsed onto an eigenstate
1 0 0
|Ψ_meas⟩ = |100⟩
Correct answer is read out with a probability of |0.99 − 0.04i|² = 98%.

Figure 8.4: Measurement process   Source: [159]

have the capacity to surmount the difficulty posed by the factorization process which will put the entire RSA scheme at risk [200]! This leads us to post-quantum cryptography, a fascinating field at the intersection of mathematics, physics and computer science.

In 1994, Peter Shor published his landmark paper establishing a quantum algorithm for the prime factorization of numbers [200]. The problem of factoring a number into primes reduces to finding *a* factor, since if you can find one factor, you can use it to divide the original number and consider the smaller factors. Eventually, using this divide (literally) and conquer strategy, we can completely factor the number into primes.

His technique to find a factor of any number n is to find the period, r, of a certain function f and then use the knowledge of the period of said function to find a factor of the number.

The Period of a Function

The problem of factoring the product of two large prime numbers is, in some sense, equivalent to the problem of finding the *period* of a function. To get an idea of what the period of a function might be, consider raising some number, like 2, to higher and higher powers, and then taking the result modulo a product of two prime numbers such as $91 = 13 \cdot 7$. For example, we have:

$2^0 \pmod{91}$	1
$2^1 \pmod{91}$	2
$2^2 \pmod{91}$	4
$2^3 \pmod{91}$	8
$2^4 \pmod{91}$	16
$2^5 \pmod{91}$	32
$2^6 \pmod{91}$	64
$2^7 \pmod{91}$	37
$2^8 \pmod{91}$	74
$\vdots$	$\vdots$

We see that the numbers cannot grow forever due to the modulo operation. For example, $2^6 \pmod{91} = 64$ and the next highest power $2^7 \pmod{91} = 37$.

8.4 **Exercise** Figure out if the powers of 2 ever cycle back to the number 1 in the table above. More precisely: find the smallest number n beyond 0 such that

$$2^n \pmod{91} = 1$$

Persistence pays off here. If you tried the above exercise, you found that, yes, $2^{12} \pmod{91} = 1$, and that 12 is the smallest number beyond 0 that makes this happen. Was it even obvious that it would return to 1? We refer to the number 12 as the period of the function defined by

$$f(n) := 2^n \pmod{91}$$

In general, for a function defined by

$$f(n) := a^n \pmod{N}$$

for some $a \in \{1, 2, ..., N-1\}$ relatively prime to N, the *period* of the function f is the smallest number n beyond 0 such that $f(n) = 1$ once again. In other words, since at $n = 0$

$$a^0 \pmod{N} = 1$$

we must find the next n that again satisfies

$$a^n \pmod{N} = 1$$

A number a is *relatively prime* to N iff[4] the greatest common divisor of a and N is equal to 1, written $\gcd(a, N) = 1$. We refer to these functions as *modular* functions. This number n is also referred to as the *order* of the element a in the group $(\mathbb{Z}/N\mathbb{Z})^\times$, which denotes the multiplicative group whose underlying set is the subset of numbers in $\{1, 2, ..., N - 1\}$ relatively prime to N, and whose binary operation is multiplication modulo N, as above.

8.5 **Exercise** Check that $(\mathbb{Z}/N\mathbb{Z})^\times$, whose underlying set of elements is the subset of numbers in $\{1, 2, ..., N - 1\}$ relatively prime to N and whose binary operation is multiplication modulo N, is actually a group! For any number N, how many elements does the group $(\mathbb{Z}/N\mathbb{Z})^\times$ have? Can you find a pattern relating the number of elements in $(\mathbb{Z}/N\mathbb{Z})^\times$ to the number N?

The fascinating point we make now is that the difficulty you experienced finding the period of the function $f(x) = 2^n \pmod{97}$ is not unique. Even a (classical) computer would have a difficult time finding the period of this function! Peter Shor realized that we can exploit quantum computing to quickly find the period of such a function [200]. We will now explain how the period of a function can be used as an input to the factorization algorithm that would crack RSA cryptography.

Period of a Function as an Input to a Factorization Algorithm

Suppose we are asked to factor some number N, and that we know how to find the period of any modular function, as described above. Remember that the problem of factoring N reduces to the simpler problem of finding *any* factor of N. So, let's see how we could leverage our ability to find the period of a modular function to find a factor of N:

1. Choose a random number $a < N$.
2. Compute $\gcd(a, N)$ using the extended Euclidean algorithm.
3. If $\gcd(a, N) \neq 1$, i.e., a and N are not relatively prime, a is already a nontrivial factor of N, and so we are done.

[4] Note: we abbreviate *if and only if* as iff throughout the book.

4. Otherwise, find the period r of the modular function

$$f(n) := a^n \ (\mathrm{mod} N)$$

5. If r is an odd number, or if $a^{\frac{r}{2}} = -1 \ (\mathrm{mod} N)$, choose a new random number and start over.
6. Otherwise, classical number theory guarantees that $\gcd(a^{\frac{r}{2}} + 1, N)$ and $\gcd(a^{\frac{r}{2}} - 1, N)$ are both nontrivial factors of N.

8.6 **Exercise** Run the above algorithm to factor the number $N = 21$.

A successful approach to the exercise above is the following:

1. began with $a = 2$, since
2. $\gcd(a, N) = \gcd(2, 21) = 1$,
3. (Step 3 is omitted, since $\gcd(a, N) = \gcd(2, 21) = 1$),
4. the period of the modular function $f(n) := 2^n$ (mod 21) is found to be $r = 6$ and
5. $r = 6$ is neither an odd number, nor does it satisfy the equation

$$a^{\frac{r}{2}} = -1 \ (\mathrm{mod}) N,$$

6. $$\gcd(a^{\frac{r}{2}}+1, N) = \gcd(2^{\frac{6}{2}}+1, 21) = \gcd(8+1, 21) = \gcd(9, 21) = 3$$

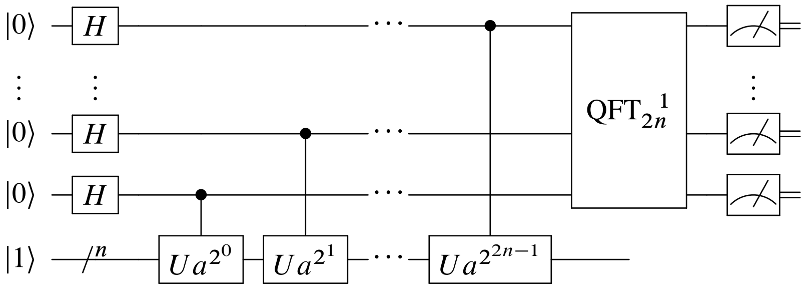
   and

   $$\gcd(a^{\frac{r}{2}}-1, N) = \gcd(2^{\frac{6}{2}}-1, 21) = \gcd(2^3-1, 21) = \gcd(7, 21) = 7$$

   are the two nontrivial factors of $N = 21$.

We can see that being able to find the period of any modular function is the key to factoring. It is the *quantum Fourier transform (QFT)*, described earlier in this book, that allows us to find the period! Shor was inspired by Simon's algorithm and BV in his development of this algorithm. He built on the use of the QFT by BV and the period finding of Simon's approach to then arrive at his number-factoring algorithm.[5] Here is the circuit diagram for Shor's algorithm:

[5] Please see this book's GitHub site for an example of code for Simon's algorithm.



Let us now do a walkthrough of a sample encoding of Shor's algorithm. The following code comes from [234]:

```
"""
toddwildey/shors-python

@toddwildey toddwildey Implemented Shor's algorithm in Python 3.X
    using state vectors

470 lines (353 sloc) 12.1 KB
#!/usr/bin/env python

shors.py: Shor's algorithm for quantum integer factorization"""

import math
import random
import argparse

__author__ = "Todd Wildey"
__copyright__ = "Copyright 2013"
__credits__ = ["Todd Wildey"]

__license__ = "MIT"
__version__ = "1.0.0"
__maintainer__ = "Todd Wildey"
__email__ = "toddwildey@gmail.com"
__status__ = "Prototype"

def printNone(str):
  pass

def printVerbose(str):
  print(str)

printInfo = printNone


#     Quantum Components



class Mapping:
  def __init__(self, state, amplitude):
    self.state = state
    self.amplitude = amplitude
```

```
class QuantumState:
  def __init__(self, amplitude, register):
    self.amplitude = amplitude
    self.register = register
    self.entangled = {}

  def entangle(self, fromState, amplitude):
    register = fromState.register
    entanglement = Mapping(fromState, amplitude)
    try:
      self.entangled[register].append(entanglement)
    except KeyError:
      self.entangled[register] = [entanglement]

  def entangles(self, register = None):
    entangles = 0
    if register is None:
      for states in self.entangled.values():
        entangles += len(states)
    else:
      entangles = len(self.entangled[register])

    return entangles


class QubitRegister:
  def __init__(self, numBits):
    self.numBits = numBits
    self.numStates = 1 << numBits
    self.entangled = []
    self.states = [QuantumState(complex(0.0), self) for x in
        range(self.numStates)]
    self.states[0].amplitude = complex(1.0)

  def propagate(self, fromRegister = None):
    if fromRegister is not None:
      for state in self.states:
        amplitude = complex(0.0)

        try:
          entangles = state.entangled[fromRegister]
          for entangle in entangles:
            amplitude += entangle.state.amplitude *
                entangle.amplitude

          state.amplitude = amplitude
        except KeyError:
          state.amplitude = amplitude

    for register in self.entangled:
      if register is fromRegister:
        continue

      register.propagate(self)
```

```
# Map will convert any mapping to a unitary tensor given each
    element v
# returned by the mapping has the property v * v.conjugate() = 1
#
def map(self, toRegister, mapping, propagate = True):
  self.entangled.append(toRegister)
  toRegister.entangled.append(self)

  # Create the covariant/contravariant representations
  mapTensorX = {}
  mapTensorY = {}
  for x in range(self.numStates):
    mapTensorX[x] = {}
    codomain = mapping(x)
    for element in codomain:
      y = element.state
      mapTensorX[x][y] = element

      try:
        mapTensorY[y][x] = element
      except KeyError:
        mapTensorY[y] = { x: element }

  # Normalize the mapping:
  def normalize(tensor, p = False):
    lSqrt = math.sqrt
    for vectors in tensor.values():
      sumProb = 0.0
      for element in vectors.values():
        amplitude = element.amplitude
        sumProb += (amplitude * amplitude.conjugate()).real

      normalized = lSqrt(sumProb)
      for element in vectors.values():
        element.amplitude = element.amplitude / normalized

  normalize(mapTensorX)
  normalize(mapTensorY, True)

  # Entangle the registers
  for x, yStates in mapTensorX.items():
    for y, element in yStates.items():
      amplitude = element.amplitude
      toState = toRegister.states[y]
      fromState = self.states[x]
      toState.entangle(fromState, amplitude)
      fromState.entangle(toState, amplitude.conjugate())

  if propagate:
    toRegister.propagate(self)

def measure(self):
  measure = random.random()
  sumProb = 0.0

  # Pick a state
  finalX = None
```

```
    finalState = None
    for x, state in enumerate(self.states):
      amplitude = state.amplitude
      sumProb += (amplitude * amplitude.conjugate()).real

      if sumProb > measure:
        finalState = state
        finalX = x
        break

    # If state was found, update the system
    if finalState is not None:
      for state in self.states:
        state.amplitude = complex(0.0)

      finalState.amplitude = complex(1.0)
      self.propagate()

    return finalX

  def entangles(self, register = None):
    entangles = 0
    for state in self.states:
      entangles += state.entangles(None)

    return entangles

  def amplitudes(self):
    amplitudes = []
    for state in self.states:
      amplitudes.append(state.amplitude)

    return amplitudes

def printEntangles(register):
  printInfo("Entagles: " + str(register.entangles()))

def printAmplitudes(register):
  amplitudes = register.amplitudes()
  for x, amplitude in enumerate(amplitudes):
    printInfo(’State #’ + str(x) + ’\’s amplitude: ’ +
        str(amplitude))

def hadamard(x, Q):
  codomain = []
  for y in range(Q):
    amplitude = complex(pow(-1.0, bitCount(x & y) & 1))
    codomain.append(Mapping(y, amplitude))

  return codomain

# Quantum Modular Exponentiation
def qModExp(a, exp, mod):
  state = modExp(a, exp, mod)
  amplitude = complex(1.0)
  return [Mapping(state, amplitude)]
```

```
# Quantum Fourier Transform
def qft(x, Q):
  fQ = float(Q)
  k = -2.0 * math.pi
  codomain = []

  for y in range(Q):
    theta = (k * float((x * y) % Q)) / fQ
    amplitude = complex(math.cos(theta), math.sin(theta))
    codomain.append(Mapping(y, amplitude))

  return codomain
```

Now that we have defined functions for entanglement and QFT, we can define the core period-finding function. Recall that this is the key subroutine that must run on quantum hardware.

```
def findPeriod(a, N):
  nNumBits = N.bit_length()
  inputNumBits = (2 * nNumBits) - 1
  inputNumBits += 1 if ((1 << inputNumBits) < (N * N)) else 0
  Q = 1 << inputNumBits

  printInfo("Finding the period...")
  printInfo("Q = " + str(Q) + "\ta = " + str(a))

  inputRegister = QubitRegister(inputNumBits)
  hmdInputRegister = QubitRegister(inputNumBits)
  qftInputRegister = QubitRegister(inputNumBits)
  outputRegister = QubitRegister(inputNumBits)

  printInfo("Registers generated")
  printInfo("Performing Hadamard on input register")

  inputRegister.map(hmdInputRegister, lambda x: hadamard(x, Q),
      False)
  # inputRegister.hadamard(False)

  printInfo("Hadamard complete")
  printInfo("Mapping input register to output register, where f(x)
      is a^x mod N")

  hmdInputRegister.map(outputRegister, lambda x: qModExp(a, x, N),
      False)

  printInfo("Modular exponentiation complete")
  printInfo("Performing quantum Fourier transform on output
      register")

  hmdInputRegister.map(qftInputRegister, lambda x: qft(x, Q), False)
  inputRegister.propagate()

  printInfo("Quantum Fourier transform complete")
  printInfo("Performing a measurement on the output register")
  y = outputRegister.measure()

  printInfo("Output register measured\ty = " + str(y))

  # Interesting to watch - simply uncomment
  # printAmplitudes(inputRegister)
  # printAmplitudes(qftInputRegister)
  # printAmplitudes(outputRegister)
  # printEntangles(inputRegister)

  printInfo("Performing a measurement on the periodicity register")

  x = qftInputRegister.measure()

  printInfo("QFT register measured\tx = " + str(x))

  if x is None:
    return None

  printInfo("Finding the period via continued fractions")

  r = cf(x, Q, N)

  printInfo("Candidate period\tr = " + str(r))

  return r
```

Now we can define the functions that will run on classical hardware.

```
BIT_LIMIT = 12

def bitCount(x):
  sumBits = 0
  while x > 0:
    sumBits += x & 1
    x >>= 1

  return sumBits

# Greatest Common Divisor
def gcd(a, b):
  while b != 0:
    tA = a % b
    a = b
    b = tA

  return a

# Extended Euclidean
def extendedGCD(a, b):
  fractions = []
  while b != 0:
    fractions.append(a // b)
    tA = a % b
    a = b
    b = tA

  return fractions

# Continued Fractions
def cf(y, Q, N):
  fractions = extendedGCD(y, Q)
  depth = 2

  def partial(fractions, depth):
    c = 0
    r = 1

    for i in reversed(range(depth)):
      tR = fractions[i] * r + c
      c = r
      r = tR

    return c

  r = 0
  for d in range(depth, len(fractions) + 1):
    tR = partial(fractions, d)
    if tR == r or tR >= N:
      return r

    r = tR

  return r

# Modular Exponentiation
def modExp(a, exp, mod):
  fx = 1
  while exp > 0:
    if (exp & 1) == 1:
      fx = fx * a % mod
    a = (a * a) % mod
    exp = exp >> 1

  return fx

def pick(N):
  a = math.floor((random.random() * (N - 1)) + 0.5)
  return a

def checkCandidates(a, r, N, neighborhood):
  if r is None:
    return None

  # Check multiples
  for k in range(1, neighborhood + 2):
    tR = k * r
    if modExp(a, a, N) == modExp(a, a + tR, N):
      return tR

  # Check lower neighborhood
  for tR in range(r - neighborhood, r):
    if modExp(a, a, N) == modExp(a, a + tR, N):
      return tR

  # Check upper neighborhood
  for tR in range(r + 1, r + neighborhood + 1):
    if modExp(a, a, N) == modExp(a, a + tR, N):
      return tR

  return None
```

Now we are ready to define the function that will call all the other functions we have created. This function will iteratively test to see if the period has been found.

```
def shors(N, attempts = 1, neighborhood = 0.0, numPeriods = 1):
  if(N.bit_length() > BIT_LIMIT or N < 3):
    return False

  periods = []
  neighborhood = math.floor(N * neighborhood) + 1

  printInfo("N = " + str(N))
  printInfo("Neighborhood = " + str(neighborhood))
  printInfo("Number of periods = " + str(numPeriods))

  for attempt in range(attempts):
    printInfo("\nAttempt #" + str(attempt))

    a = pick(N)
    while a < 2:
      a = pick(N)

    d = gcd(a, N)
    if d > 1:
      printInfo("Found factors classically, re-attempt")
      continue

    r = findPeriod(a, N)

    printInfo("Checking candidate period, nearby values, and
        multiples")

    r = checkCandidates(a, r, N, neighborhood)

    if r is None:
      printInfo("Period was not found, re-attempt")
      continue

    if (r % 2) > 0:
      printInfo("Period was odd, re-attempt")
      continue

    d = modExp(a, (r // 2), N)
    if r == 0 or d == (N - 1):
      printInfo("Period was trivial, re-attempt")
      continue

    printInfo("Period found\tr = " + str(r))

    periods.append(r)
    if(len(periods) < numPeriods):
      continue

    printInfo("\nFinding least common multiple of all periods")

    r = 1
    for period in periods:
      d = gcd(period, r)
      r = (r * period) // d

    b = modExp(a, (r // 2), N)
    f1 = gcd(N, b + 1)
    f2 = gcd(N, b - 1)

    return [f1, f2]

  return None
```

Finally, we define various flags for command line functionality.

```
def parseArgs():
  parser = argparse.ArgumentParser(description=’Simulate Shor\’s
      algorithm for N.’)
  parser.add_argument(’-a’, ’--attempts’, type=int, default=20,
      help=’Number of quantum attempts to perform’)
  parser.add_argument(’-n’, ’--neighborhood’, type=float,
      default=0.01, help=’Neighborhood size for checking candidates
      (as percentage of N)’)
  parser.add_argument(’-p’, ’--periods’, type=int, default=2,
      help=’Number of periods to get before determining least common
      multiple’)
  parser.add_argument(’-v’, ’--verbose’, type=bool, default=True,
      help=’Verbose’)
  parser.add_argument(’N’, type=int, help=’The integer to factor’)
  return parser.parse_args()

def main():
  args = parseArgs()

  global printInfo
  if args.verbose:
    printInfo = printVerbose
  else:
    printInfo = printNone

  factors = shors(args.N, args.attempts, args.neighborhood,
      args.periods)
  if factors is not None:
    print("Factors:\t" + str(factors[0]) + ", " + str(factors[1]))

if __name__ == "__main__":
  main()
```

So there it is — the famous Shor's algorithm. While we do not yet have the fault-tolerant hardware to run Shor's for any meaningfully large key, it is illustrative of the potential of quantum computing. Although Shor's algorithm is proven to run in polynomial time (i.e., polynomial in the number of bits in the integer to be factored), much work can be done to reduce the constant factors in this polynomial and overall resource requirements. See Gidney and Ekera's work [95] for a discussion of resource requirements in Shor's algorithm.

An example of using this program to factor 15 is shown below. This code (saved in a Python module called "shor.py" on this book's website) is set up to be run from a command line with the number to be factored as an argument. By executing

```
python shor.py 15
```

we see the following output:

```
N = 15
Neighborhood = 1
Number of periods = 2

Attempt #0
Finding the period...
Q = 256 a = 8
Registers generated
Performing Hadamard on input register
Hadamard complete
Mapping input register to output register, where f(x) is a^x mod N
Modular exponentiation complete
Performing quantum Fourier transform on output register
Quantum Fourier transform complete
Performing a measurement on the output register
Output register measured y = 1
Performing a measurement on the periodicity register
QFT register measured x = 192
Finding the period via continued fractions
Candidate period r = 4
Checking candidate period, nearby values, and multiples
Period found r = 4

Attempt #1
Found factors classically, re-attempt

Attempt #2
Found factors classically, re-attempt
```

```
Attempt #3
Finding the period...
Q = 256 a = 2
Registers generated
Performing Hadamard on input register
Hadamard complete
Mapping input register to output register, where f(x) is a^x mod N
Modular exponentiation complete
Performing quantum Fourier transform on output register
Quantum Fourier transform complete
Performing a measurement on the output register
Output register measured y = 2
Performing a measurement on the periodicity register
QFT register measured x = 128
Finding the period via continued fractions
Candidate period r = 2
Checking candidate period, nearby values, and multiples
Period found r = 4

Finding least common multiple of all periods
Factors: 5, 3
```

Here, we see that the quantum part of Shor’s algorithm is executed four times (labeled “‘Attempt #1” through “Attempt #4”). In two of the attempts, the circuit succeeds in finding the period, while in the other two, the factors are found classically by virtue of good luck, so the program re-attempts the quantum part. After finding the period twice, the classical part of Shor’s algorithm ensues, in which the least common multiple of all periods found is computed. From this, the prime factors are determined correctly as 3 and 5.

8.6 *Grover’s Search Algorithm*

In 1996, Lov Grover demonstrated that we can obtain a quadratic speedup in algorithmic search on a quantum computer compared with a classical one [98]. While this is not exponential speedup, it is still significant.

The search problem can be set up as follows. Given a function $f(x)$ such that $f(a^*) = -1$ and all other outputs of the function are 1, find a^*. In other words, we are looking for an exhaustive search algorithm; in particular, we are seeking an algorithmic search protocol. An algorithmic search protocol is one in which we can verify that we have found the item in question by evaluating a function on the search result. So we have exhausted the possibility of finding an analytic approach and now must do a brute force search.

On a classical computer, this would entail an exhaustive search using n operations for some range of $x = \{0, n\}$, or at best $\frac{n}{2}$ steps if we posit that on

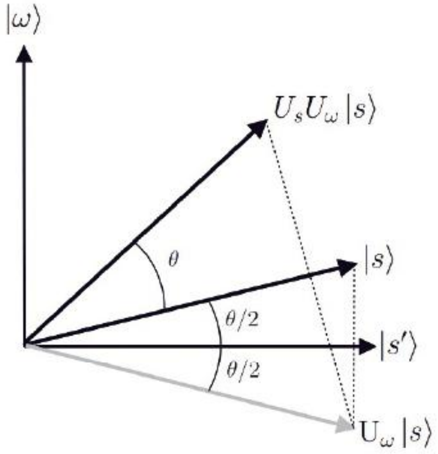


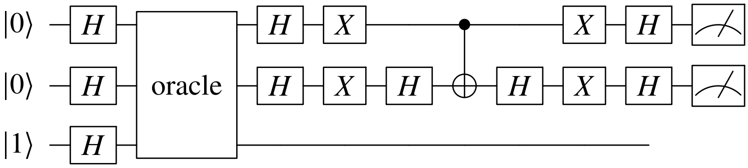
Figure 8.5: Plotting Grover's algorithm as it closes in on the target Source: Wikimedia

average we can find the target after searching half the range. On a quantum computer, however, we can do much better. Instead of n or $\frac{n}{2}$, we can find $a*$ in $O(\sqrt{n})$ steps. Bennett et al. then showed that any such algorithm which solves an algorithmic search problem running on a quantum computer would query the oracle at best $\Omega(\sqrt{n})$; Grover's algorithm is therefore optimal [24].

Grover's algorithm is a bit more involved than DJ and BV. Here we have to apply three unitary operators, the latter two of which we implement in a loop until we find our target. In the manner of David Deutsch, let's call these three operators H, M and XX [66].

As with other algorithms we have examined, we prepare our data input qubits in state $|0\rangle$ and our output qubit in state $|1\rangle$. We then apply H to all data input qubits and to the output qubit.

We now query the oracle:



```
"""Grover's algorithm in Cirq."""

# Imports
import random

import cirq
```

```
def set_io_qubits(qubit_count):
   """Add the specified number of input and output qubits."""
   input_qubits = [cirq.GridQubit(i, 0) for i in range(qubit_count)]
   output_qubit = cirq.GridQubit(qubit_count, 0)
   return (input_qubits, output_qubit)


def make_oracle(input_qubits, output_qubit, x_bits):
   """Implement function {f(x) = 1 if x==x’, f(x) = 0 if x!= x’}."""
   # Make oracle.
   # for (1, 1) it’s just a Toffoli gate
   # otherwise negate the zero-bits.
   yield(cirq.X(q) for (q, bit) in zip(input_qubits, x_bits) if not
       bit)
   yield(cirq.TOFFOLI(input_qubits[0], input_qubits[1],
       output_qubit))
   yield(cirq.X(q) for (q, bit) in zip(input_qubits, x_bits) if not
       bit)


def make_grover_circuit(input_qubits, output_qubit, oracle):
   """Find the value recognized by the oracle in sqrt(N) attempts."""
   # For 2 input qubits, that means using Grover operator only once.
   c = cirq.Circuit()

   # Initialize qubits.
   c.append([
      cirq.X(output_qubit),
      cirq.H(output_qubit),
      cirq.H.on_each(*input_qubits),
   ])

   # Query oracle.
   c.append(oracle)

   # Construct Grover operator.
   c.append(cirq.H.on_each(*input_qubits))
   c.append(cirq.X.on_each(*input_qubits))
   c.append(cirq.H.on(input_qubits[1]))
   c.append(cirq.CNOT(input_qubits[0], input_qubits[1]))
   c.append(cirq.H.on(input_qubits[1]))
   c.append(cirq.X.on_each(*input_qubits))
   c.append(cirq.H.on_each(*input_qubits))

   # Measure the result.
   c.append(cirq.measure(*input_qubits, key=’result’))

   return c


def bitstring(bits):
   return ’’.join(str(int(b)) for b in bits)


def main():
   qubit_count = 2
   circuit_sample_count = 10

   #Set up input and output qubits.
   (input_qubits, output_qubit) = set_io_qubits(qubit_count)

   #Choose the x’ and make an oracle which can recognize it.
   x_bits = [random.randint(0, 1) for _ in range(qubit_count)]
   print(’Secret bit sequence: {}’.format(x_bits))

   # Make oracle (black box)
   oracle = make_oracle(input_qubits, output_qubit, x_bits)

   # Embed the oracle into a quantum circuit implementing Grover’s
       algorithm.
   circuit = make_grover_circuit(input_qubits, output_qubit, oracle)
   print(’Circuit:’)
   print(circuit)

   # Sample from the circuit a couple times.
   simulator = cirq.Simulator()
   result = simulator.run(circuit, repetitions=circuit_sample_count)

   frequencies = result.histogram(key=’result’, fold_func=bitstring)
   print(’Sampled results:\n{}’.format(frequencies))

   # Check if we actually found the secret value.
   most_common_bitstring = frequencies.most_common(1)[0][0]
   print(’Most common bitstring: {}’.format(most_common_bitstring))
   print(’Found a match: {}’.format(
      most_common_bitstring == bitstring(x_bits)))


if __name__ == ’__main__’:
   main()
```

We now run the code and obtain this as an example output:

```
"""
=== EXAMPLE OUTPUT ===
Secret bit sequence: [1, 0]

Sampled results:
Counter({’10’: 10})
Most common bitstring: 10
Found a match: True
"""
```

Summary

In this chapter, we have explored the set of canonical quantum algorithms. These breakthroughs from the 1980s and 1990s established the potential for quantum advantage. While we still do not have the hardware to run Shor’s and Grover's algorithms with meaningful scale, they are powerful reminders of what is to come. In the next chapter we will cover a range of quantum computing methods for the NISQ regime.

CHAPTER 9

Quantum Computing Methods

In this section we will walk through a range of quantum computing programs that can be run on NISQ processors. We will cover methods in optimization, chemistry, machine learning and other areas.

9.1 Variational Quantum Eigensolver

Let us first examine a variational quantum eigensolver (VQE) [170]. We can use a VQE to find the eigenvalues of a large matrix that represents the Hamiltonian of a system. In many cases, we are looking for the lowest eigenvalue, which represents the ground state energy of the system. We can also use VQE and VQE-type algorithms to calculate additional eigenvalues, which represent excited state energies [150, 111]. VQE is a good example of a hybrid classical/quantum approach to solving a problem (for more on VQEs see [170, 231, 165, 227, 202]). While the VQE was initially developed to find ground states of Hamiltonians, we can use it to find the minimum of any given objective function that we can express in a quantum circuit. This broadens the application space significantly for this variational method.

In variational methods, we start with a best guess, or *ansatz*, for the ground state. More specifically we parameterize a quantum state $|\psi(\theta)\rangle$ where θ is a set of parameters. The problem that VQE solves is as follows:

> Given a Hamiltonian H, conventionally coming from a physical system such as molecular hydrogen or water, approximate the ground state energy (minimum eigenvalue of H) by solving the following optimization problem

$$\min_{\theta} \ \langle\psi(\theta)| H |\psi(\theta)\rangle \tag{9.1}$$

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_9

By the variational principle of quantum mechanics, the quantity

$$\langle\psi(\theta)|\, H\, |\psi(\theta)\rangle$$

can never be smaller than the ground state energy. So, by minimizing this quantity, we get an approximation of the ground state energy.

In the VQE algorithm, $|\psi(\theta)\rangle$ is prepared on a quantum computer, so the ansatz is typically developed from parametrized quantum gates; for example, the rotation gates $R_\sigma(\varphi)$ where σ is a Pauli operator, as well as other "static" quantum gates like *CNOT* or control-Z.

For the purposes of VQE, we will assume our Hamiltonian is written as the sum of tensor products of Pauli operators weighted by constant coefficients as per [150].

$$H = \sum_{i=1}^{m} c_i H_i \tag{9.2}$$

Note that the tensor products of Pauli operators form a basis for Hermitian matrices, so in principle any Hamiltonian can be expressed in this way. However, this may lead to a number of terms exponential in the system size. For this general case, different representations are crucial for limiting the number of terms in the Hamiltonian and thus limiting the number of resources required for the quantum algorithm. For the present discussion, we will restrict our attention to Hamiltonians of the form (9.2) where m grows at most polynomially in the system size — that is, $m = O(n^k)$ — which is a reasonable assumption for many physical systems of interest.

The VQE algorithm computes expectation values of each term H_i using a quantum circuit, then adds the total energy classically. The classical optimizer changes the values of the ansatz wavefunction to minimize the total energy. Once an approximate minimum is found, the VQE returns the ground state energy as well as its eigenstate.

Another application of VQEs is error-mitigation. McClean et al. explore this aspect of VQEs:

> Here, we provide evidence for the conjecture that variational approaches can automatically suppress even non-systematic decoherence errors by introducing an exactly solvable channel model of variational state preparation. Moreover, we show how variational quantum-classical approaches fit in a more general hierarchy of measurement and classical computation that

allows one to obtain increasingly accurate solutions with additional classical resources [148].

We recommend the reader explore the use of subspace expansion to achieve error-mitigation in the growing body of literature on this subject [147].

Below, we show a program implementing VQE for a simple Hamiltonian using pyQuil and the Grove library [97].[1] Let us walk through this program in steps and explain each part. First, we import the necessary packages and connect to the quantum virtual machine (QVM).

```
# Imports
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

from pyquil.quil import Program
import pyquil.api as api
from pyquil.paulis import sZ
from pyquil.gates import RX, X, MEASURE

from grove.pyvqe.vqe import VQE
```

We then set up an ansatz, which in this case is the rotation matrix around the x-axis with a single parameter.

```
# Function to create the ansatz
def small_ansatz(params):
   """Returns an ansatz Program with one parameter."""
   return Program(RX(params[0], 0))

# Show the ansatz with an example value for the parameter
print("Ansatz with example value for parameter:")
print(small_ansatz([1.0]))
```

The output of this portion of the program showing the ansatz as a pyQuil circuit is shown below.

```
Ansatz with example value for parameter:
RX(1.0) 0
```

Next, we set up a Hamiltonian; as we stated above, any Hamiltonian can be expressed as a linear combination of tensor products of Pauli operators, as these form a basis for Hermitian matrices. In practice, Hamiltonians must first be converted to qubit operators so that expectation values can be measured

[1] Note that in this book we show code examples in a range of QC frameworks; check the book's online site for code examples in other libraries as one can implement these methods and algorithms in each of the frameworks.

using the quantum computer. If there are m non-trivial, distinct terms in the Hamiltonian (9.2), then there are m distinct expectation values to compute. Each quantum circuit in VQE computes one expectation value, so there are m distinct quantum circuits to run.

For simplicity of instruction, we consider the simple case of one Pauli operator $H = Z$ (note that in this section H refers to a Hamiltonian and not the Hadamard operator). We create an instance of the VQE algorithm using the VQE class imported from Grove and compute the expectation value for an example angle in the ansatz.

```
# Show the ansatz with an example value for the parameter
print("Ansatz with example value for parameter:")
print(small_ansatz([1.0]))

# Create a Hamiltonion H = Z_0
hamiltonian = sZ(0)

# Make an instance of VQE with a Nelder-Mead minimizer
vqe_inst = VQE(minimizer=minimize,
           minimizer_kwargs={'method': 'nelder-mead'})

# Check the VQE manually at a particular angle - say 2.0 radians
angle = 2.0
print("Expectation of Hamiltonian at angle = {}".format(angle))
print(vqe_inst.expectation(small_ansatz([angle]), hamiltonian,
    10000, qvm))
```

To get a picture of the landscape of the optimization problem, we can sweep over a set of values in the range $[0, 2\pi)$. Here, since we have only one parameter in our ansatz, this is computationally inexpensive to do. For larger ansatzes with more parameters, implementing a grid search over all possible values is not feasible, and so classical optimization algorithms must be used to find an approximate minimum.

```
# Loop over a range of angles and plot expectation without sampling
angle_range = np.linspace(0.0, 2.0 * np.pi, 20)
exact = [vqe_inst.expectation(small_ansatz([angle]), hamiltonian,
    None, qvm)
       for angle in angle_range]

# Plot the exact expectation
plt.plot(angle_range, exact, linewidth=2)

# Loop over a range of angles and plot expectation with sampling
sampled = [vqe_inst.expectation(small_ansatz([angle]), hamiltonian,
    1000, qvm)
        for angle in angle_range]

# Plot the sampled expectation
plt.plot(angle_range, sampled, "-o")
```

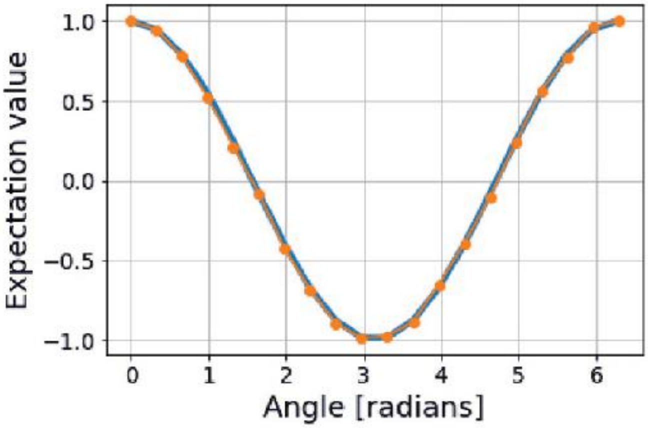


Figure 9.1: Expectation value of the simple Hamiltonian $H = Z$ *at all angles* $\theta \in [0, 2\pi)$ *in the wavefunction ansatz* $|\psi(\theta)\rangle = R_x(\theta)\,|0\rangle$

```
# Plotting options
plt.xlabel('Angle [radians]')
plt.ylabel('Expectation value')
plt.grid()
plt.show()
```

The plot that this section of the program produces is shown in Figure 9.1. Here, we can visually see that the minimum energy (in arbitrary units) of the Hamiltonian appears around the angle $\theta = \pi$ in the wavefunction ansatz. As mentioned, for larger Hamiltonians that require more parameters in the ansatz, enumerating the expectation values for all angles is not feasible. Instead, an optimization algorithm must be used to traverse the optimization landscape and find, ideally, the global minima.

An example of the Nelder-Mead optimization algorithm, implemented in the SciPy Optimize package, is shown below.

```
# Do the minimization and return the best angle
initial_angle = [0.0]
result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle,
    None, qvm=qvm)
print("\nMinimum energy =", round(result["fun"], 4))
print("Best angle =", round(result["x"][0], 4))
```

The output for this final part of the program is

```
Minimum energy = -1.0
Best angle = 3.1416
```

As can be seen, the optimizer is able to find the correct angle $\theta = \pi$ for the global minimum energy $E = \langle\psi| H |\psi\rangle = -1.0$ (in arbitrary units).

VQE with Noise

The VQE algorithm is designed to make effective use of near-term quantum computers. It is therefore important to analyze its performance in a noisy environment such as a NISQ processor. Above, we used the noiseless QVM to simulate circuits in VQE. Now, we can consider a QVM with a particular noise model and run the VQE algorithm again.

A code block setting up a noisy QVM in pyQuil — and demonstrating it is in fact noisy — is shown below. Note that this program is an extension of the previous program and assumes all packages are still imported.

```
# Create a noise model which has a 10% chance of each gate at each
    timestep
pauli_channel = [0.1, 0.1, 0.1]
noisy_qvm = api.QVMConnection(gate_noise=pauli_channel)

# Check that the simulator is indeed noisy
p = Program(X(0), MEASURE(0, 0))
res = noisy_qvm.run(p, [0], 10)
print(res)
```

The example output of this program (measuring the $|1\rangle$ state) demonstrates that the simulator is indeed noisy — otherwise, we would never see the bit 0 measured!

```
"Outcome of NOT and MEASURE circuit on noisy simulator:"
[[0], [1], [1], [1], [0], [1], [1], [1], [1], [0]]
```

Now that we have a noisy simulator, we can run the VQE algorithm under noise. Here, we modify the classical optimizer to start with a larger simplex so we don't get stuck at an initial minimum. Then, we visualize the same landscape plot (energy vs. angle) as before, but now in the presence of noise.

```
# Update the minimizer in VQE to start with a larger initial simplex
vqe_inst.minimizer_kwargs = {"method": "Nelder-mead",
                      "options":
                         {"initial_simplex": np.array([[0.0],
                             [0.05]]),
                          "xatol": 1.0e-2}
                         }

# Loop over a range of angles and plot expectation with sampling
sampled = [vqe_inst.expectation(small_ansatz([angle]), hamiltonian,
    1000, noisy_qvm)
        for angle in angle_range]
```

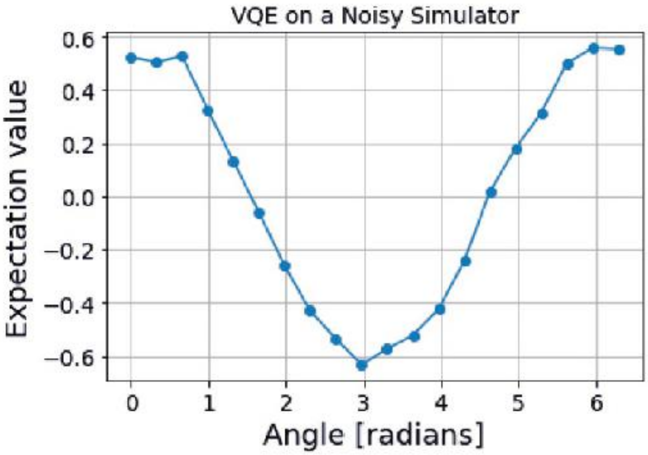


Figure 9.2: Results of running VQE on a simulator with Pauli channel noise.

```
# Plot the sampled expectation
plt.plot(angle_range, sampled, "-o")

# Plotting options
plt.title("VQE on a Noisy Simulator")
plt.xlabel("Angle [radians]")
plt.ylabel("Expectation value")
plt.grid()
plt.show()
```

An example plot produced by this part of the program is shown in Figure 9.2. Here, we note that the landscape generally has the same shape but is slightly distorted. The minimum value of the curve still occurs close to the optimal value of $\theta = \pi$, but the value of the energy here is vertically shifted — the minimum energy here is approximately -0.6 (in arbitrary units) whereas in the noiseless case the minimum energy was -1.0.

However, since the minimum value still occurs around $\theta = \pi$, VQE displays some robustness to noise. The optimal parameters can still be found, and the vertical offset in the minimum energy can be accounted for in classical postprocessing.

Pauli channel noise is not the only noise model we can consider. In the book's online site, this VQE program also demonstrates a noisy simulator with measurement noise. We find that the VQE is robust to measurement noise in the same sense as above — the landscape curve has the same general shape and the minimum value occurs again near $\theta = \pi$.

More Sophisticated Ansatzes

As mentioned, larger Hamiltonians may require an ansatz with more parameters to more closely approximate the ground state wavefunction. In pyQuil, we can increase the number of parameters in our program by adding more gates as follows:

```
# Function for an anstaz with two parameters
def smallish_ansatz(params):
    """Returns an ansatz with two parameters."""
    return Program(RX(params[0], 0), RZ(params[1], 0))

print("Ansatz with two gates and two parameters (with example
    values):")
print(smallish_ansatz([1.0, 2.0]))

# Get a VQE instance
vqe_inst = VQE(minimizer=minimize, minimizer_kwargs={’method’:
    ’nelder-mead’})

# Do the minimization and return the best angle
initial_angles = [1.0, 1.0]
result = vqe_inst.vqe_run(smallish_ansatz, hamiltonian,
    initial_angles, None, qvm=qvm)
print("\nMinimum energy =", round(result["fun"], 4))
print("Best angle =", round(result["x"][0], 4))
```

In the program above, we create an ansatz with two gates and two parameters, print it out, then run the VQE algorithm with this ansatz. An example outcome of the program is

```
"Ansatz with two gates and two parameters (with example values):"
RX(1.0) 0
RZ(2.0) 0

Minimum energy = -1.0
Best angle = 3.1416
```

Here, we see that the minimizer is able to find the exact ground state energy with the new ansatz. This is expected — since we know we can minimize the expectation of the Hamiltonian with only one R_x gate, the second R_z gate is superfluous. For larger, non-trivial Hamiltonians, however, this may not be the case, and more parameters may be needed.

In this section, we use simple trial ansatzes for clarity of presentation. In general, choosing both an appropriate ansatz and good initial starting point for the parameters of the ansatz are critical for successful VQE implementations. Randomly generated ansatzes are likely to have gradients that vanish for large circuit sizes [146], thus making the optimization over parameters exceedingly difficult, if not practically impossible. For these reasons, structured ansatzes such as unitary coupled cluster or QAOA (see Section 9.3) — as opposed to parameterized random quantum circuits — are used in practice.

9.2 *Quantum Chemistry*

We will now explore an application of quantum chemistry, or more generally quantum simulation.[2] In quantum simulation we seek to model the dynamic evolution of the wavefunction under some Hamiltonian H as per Schrödinger's equation

$$i\frac{\partial\,|\psi\rangle}{\partial t} = H\,|\psi\rangle \tag{9.3}$$

where we have set $\hbar = 1$. It is easy to *write* the time evolution operator

$$U(t) = \exp\left(-iHt\right) \tag{9.4}$$

which evolves the initial state $|\psi(0)\rangle$ to a final state at time t via

$$|\psi(t)\rangle = U(t)|\psi(0)\rangle \tag{9.5}$$

However, it is generally very difficult to classically *compute* the unitary time evolution operator $U(t)$ by exponentiating the Hamiltonian of the system, even if the Hamiltonian is sparse. Quantum simulation on a QC give us the opportunity to more easily compute unitary time evolution. Quantum simulation is useful in the same respect that classical simulation of time-dependent processes is useful. Namely, it allows us to analyze the behavior of a complex physical system, compute observable properties, and use both of these these to make new predictions or compare them with experimental results. As an example, O'Malley et al. demonstrated the use of VQE and quantum simulation with QPE to calculate the potential energy surface of molecular hydrogen [165].

Quantum simulation of molecular Hamiltonians is useful for quantum chemistry applications. In the following program, we use Cirq in conjuction with OpenFermion — an open-source package for quantum chemistry that has integration with Cirq [149][3] — to show how we can simulate the evolution of an initial state under a Hamiltonian. In the sample code used here for

[2] Note: this use of the term "quantum simulation" is distinct from the use of a program to simulate the actions of a quantum computer as discussed in chapter 6.

[3] Note that the package OpenFermion-Cirq is used as a bridge between OpenFermion and Cirq.

pedagogic purposes we randomly generate the Hamiltonian and its initial state; we recommend against this in real world conditions for reasons outlined in McClean et al [146].

We now walk through the program in steps. First, we import the necessary packages and define a few constants for our simulation. Namely, we define the number of qubits n, the final simulation time t, and a seed for the random number generator which allows for reproducible results.

```
# Imports
import numpy
import scipy

import cirq
import openfermion
import openfermioncirq

# Set the number of qubits, simulation time, and seed for
    reproducibility
n_qubits = 3
simulation_time = 1.0
random_seed = 8317
```

In the code block below, we generate a random Hamiltonian in matrix form. In order to run any quantum circuits with this Hamiltonian, it first must be written in terms of quantum operators. The next few lines of code use the functionality in OpenFermion to do this.

```
# Generate the random one-body operator
T = openfermion.random_hermitian_matrix(n_qubits, seed=random_seed)
print("Hamiltonian:", T, sep="\n")

# Compute the OpenFermion "FermionOperator" form of the Hamiltonian
H = openfermion.FermionOperator()
for p in range(n_qubits):
   for q in range(n_qubits):
      term = ((p, 1), (q, 0))
      H += openfermion.FermionOperator(term, T[p, q])
print("\nFermion operator:")
print(H)
```

The output of this portion of the program is

```
Hamiltonian:
[[ 0.53672126+0.j   -0.26033703+3.32591737j 1.34336037+1.54498725j]
 [-0.26033703-3.32591737j -2.91433037+0.j -1.52843836+1.35274868j]
 [ 1.34336037-1.54498725j -1.52843836-1.35274868j 2.26163363+0.j ]]

Fermion operator:
(0.5367212624097257+0j) [0^ 0] +
(-0.26033703159240107+3.3259173741375454j) [0^ 1] +
(1.3433603748462144+1.544987250567917j) [0^ 2] +
(-0.26033703159240107-3.3259173741375454j) [1^ 0] +
(-2.9143303700812435+0j) [1^ 1] +
(-1.52843836446248+1.3527486791390022j) [1^ 2] +
(1.3433603748462144-1.544987250567917j) [2^ 0] +
(-1.52843836446248-1.3527486791390022j) [2^ 1] +
(2.261633626116526+0j) [2^ 2]
```

The first section displays the Hamiltonian in matrix form, then the next section displays the matrix in OpenFermion operator form. Here, the OpenFermion notation `[p^ q]` is used to indicate the product of fermionic creation and annihilation operators $a_p^\dagger a_q$ on sites p and q, respectively, which satisfy the canonical commutation relations

$$\{a_p^\dagger, a_q\} = \delta_{pq} \tag{9.6}$$

$$\{a_p, a_q\} = 0 \tag{9.7}$$

Now that we have our Hamiltonian in a usable form, we can begin constructing our circuit. As is common in quantum simulation algorithms [161], we first rotate to the eigenbasis of the Hamiltonian. This is done by (classically) diagonalizing the Hamiltonian, then using OpenFermion to construct a circuit that performs this basis transformation.

```
# Diagonalize T and obtain basis transformation matrix (aka "u")
eigenvalues, eigenvectors = numpy.linalg.eigh(T)
basis_transformation_matrix = eigenvectors.transpose()

# Initialize the qubit register
qubits = cirq.LineQubit.range(n_qubits)

# Rotate to the eigenbasis
inverse_basis_rotation = cirq.inverse(
   openfermioncirq.bogoliubov_transform(qubits,
       basis_transformation_matrix)
)
circuit = cirq.Circuit.from_ops(inverse_basis_rotation)
```

Now we can add the gates corresponding to evolution of the Hamiltonian. Since we are in the eigenbasis of the Hamiltonian, this corresponds to a diagonal operator of Pauli-Z rotations, where the rotation angle is proportional to the eigenvalue and final simulation time. Finally, we change bases back to the computational basis.

```
# Add diagonal phase rotations to circuit
for k, eigenvalue in enumerate(eigenvalues):
   phase = -eigenvalue * simulation_time
   circuit.append(cirq.Rz(rads=phase).on(qubits[k]))
```

```
# Finally, change back to the computational basis
basis_rotation = openfermioncirq.bogoliubov_transform(
   qubits, basis_transformation_matrix
)
circuit.append(basis_rotation)
```

The time evolution operator is now constructed in our quantum circuit. Below, we first obtain a random initial state. Note that this is program is for demonstration purposes. In real world scenarios we will want to use a number of non-random techniques to determine the initial state:

```
# Initialize a random initial state
initial_state = openfermion.haar_random_vector(
   2 ** n_qubits, random_seed).astype(numpy.complex64)
```

Now we compute the time evolution numerically using matrix exponentiation and then simulate it with a QC simulator. After obtaining the final state using both methods, we compute the fidelity (overlap squared) of the two and print out the value.

```

# Numerically compute the correct circuit output
hamiltonian_sparse = openfermion.get_sparse_operator(H)
exact_state = scipy.sparse.linalg.expm_multiply(
   -1j * simulation_time * hamiltonian_sparse, initial_state
)

# Use Cirq simulator to apply circuit
simulator = cirq.google.XmonSimulator()
result = simulator.simulate(circuit, qubit_order=qubits,
                     initial_state=initial_state)
simulated_state = result.final_state

# Print final fidelity
fidelity = abs(numpy.dot(simulated_state,
    numpy.conjugate(exact_state)))**2
print("\nfidelity =", round(fidelity, 4))
```

The output of this section of the code

```
fidelity = 1.0
```

indicates that our quantum circuit evolved the initial state exactly the same as the analytic evolution!

Of course, for larger systems the analytic evolution cannot be computed, and we have to rely solely on a quantum computer. This small proof of principle calculation indicates the validity of this method.

Lastly, we mention that Cirq has the functionality to compile this quantum circuit for Google's Xmon architecture quantum computers, as well as IBM's quantum computers. The code snippet below shows how this is done:

```
# Compile the circuit to Google's Xmon architecture
xmon_circuit = cirq.google.optimized_for_xmon(circuit)
print("\nCircuit optimized for Xmon:")
print(xmon_circuit)

# Print out the OpenQASM code for IBM's hardware
print("\nOpenQASM code:")
print(xmon_circuit.to_qasm())
```

Below, we include the OpenQASM code generated by Cirq for this circuit. The complete circuit diagram and remaining output of this code can be seen by executing this program, which can be found on the book's GitHub site.

```
// Generated from Cirq v0.4.0

OPENQASM 2.0;
include "qelib1.inc";


// Qubits: [0, 1, 2]
qreg q[3];


u2(pi*-1.0118505646, pi*1.0118505646) q[2];
u2(pi*-1.25, pi*1.25) q[1];
u2(pi*-1.25, pi*1.25) q[0];
cz q[1],q[2];
u3(pi*-0.1242949803, pi*-0.0118505646, pi*0.0118505646) q[2];
u3(pi*0.1242949803, pi*-0.25, pi*0.25) q[1];
cz q[1],q[2];
u3(pi*-0.3358296941, pi*0.4881494354, pi*-0.4881494354) q[2];
u3(pi*-0.5219350773, pi*1.25, pi*-1.25) q[1];
cz q[0],q[1];
u3(pi*-0.328242091, pi*0.75, pi*-0.75) q[1];
u3(pi*-0.328242091, pi*-0.25, pi*0.25) q[0];
cz q[0],q[1];
u3(pi*-0.2976584908, pi*0.25, pi*-0.25) q[1];
u3(pi*-0.7937864503, pi*0.25, pi*-0.25) q[0];
cz q[1],q[2];
u3(pi*-0.2326621647, pi*-0.0118505646, pi*0.0118505646) q[2];
u3(pi*0.2326621647, pi*-0.25, pi*0.25) q[1];
cz q[1],q[2];
u3(pi*0.8822298425, pi*0.4881494354, pi*-0.4881494354) q[2];
u3(pi*-0.2826706001, pi*0.25, pi*-0.25) q[1];
cz q[0],q[1];
u3(pi*-0.328242091, pi*0.75, pi*-0.75) q[1];
u3(pi*-0.328242091, pi*-0.25, pi*0.25) q[0];
cz q[0],q[1];
u3(pi*-0.3570821075, pi*0.25, pi*-0.25) q[1];
u2(pi*-0.25, pi*0.25) q[0];
```

```
rz(pi*0.676494835) q[0];
cz q[1],q[2];
u3(pi*0.1242949803, pi*0.9881494354, pi*-0.9881494354) q[2];
u3(pi*-0.1242949803, pi*0.75, pi*-0.75) q[1];
cz q[1],q[2];
u2(pi*-0.0118505646, pi*0.0118505646) q[2];
u2(pi*-0.25, pi*0.25) q[1];
rz(pi*-0.4883581348) q[1];
rz(pi*0.5116418652) q[2];
```

9.3 *Quantum Approximate Optimization Algorithm (QAOA)*

While the previous two quantum computing methods were geared towards physics and chemistry applications, the quantum approximate optimization algorithm (QAOA) is geared towards general optimization problems. Farhi et al. introduced QAOA to handle these kinds of problems [78, 79]. Here, the goal is to maximize or minimize a cost function

$$C(\mathbf{b}) = \sum_{\alpha=1}^{m} C_\alpha(\mathbf{b}) \tag{9.8}$$

written as a sum of m clauses $C_\alpha(\mathbf{b})$ on bitstrings $\mathbf{b} \in \{0, 1\}^n$, or equivalently spins $z_i \in \{-1, +1\}^n$ as there is a bijective map between the bitstrings and spins.[4]

MaxCut is an example of a problem for which we can use QAOA on a regular graph [78]; the cost function can be written in terms of spins as

$$C(\mathbf{z}) = \frac{1}{2} \sum_{\langle i,j \rangle} (1 - z_i z_j) \tag{9.9}$$

Here, the sum is over edges $\langle i, j \rangle$ in a graph, and each clause $(1 - z_i z_j)$ contributes a non-zero term to the cost iff the spins z_i and z_j are anti-aligned (i.e., have different values). A more general case of this problem considers arbitrary *weights* w_{ij} between each edge [243]. This amounts to saying that clauses with larger weights w_{ij} contribute a higher cost. We'll consider this case in the text that follows.

[4] Bits b and spins z are related by the bijective mapping $z = 1 - 2b \iff b = (1 - z)/2$ Thus, any problem on bits can be framed as a problem on spins and vice versa.

By promoting each spin to a Pauli-Z operator (which has eigenvalues ± 1), the cost can be written as a *cost Hamiltonian*

$$C \equiv H_C = \frac{1}{2} \sum_{\langle i,j \rangle} w_{ij} (I - \sigma_z^{(i)} \sigma_z^{(j)}) \tag{9.10}$$

where I is the identity operator and $\sigma_z^{(i)}$ denotes a Pauli-Z operator on the ith spin. This cost Hamiltonian can easily be seen to be diagonal in the computational basis. This is the general input to the quantum approximate optimization algorithm, as we discuss below.

The prescription for QAOA is as follows. Given a cost Hamiltonian $H_C \equiv H$ of the form (9.8), define the unitary operator

$$U(H_C, \gamma) := e^{\ i\gamma H_C} = \prod_{\alpha=1}^{m} e^{\ i\gamma C_\alpha} \tag{9.11}$$

which depends on the parameter γ. Note that the second line follows because each clause C_α is diagonal in the computational basis, hence $[C_\alpha, C_\beta] = 0$ for all $\alpha, \beta \in \{1, ..., m\}$. Further note that this can be interpreted, in light of the previous section, as simulating (i.e., evolving with) the cost Hamiltonian H_C for a *time* γ. We can restrict the "time" to be between 0 and 2π, however, since C has integer values. Thus, we can equally think of the parameter γ as an angle of rotation.

Next, define the operator $B \equiv H_B$, known as a *mixer Hamiltonian*, which is conventionally taken to be

$$B \equiv H_B = \sum_{j=1}^{n} \sigma_x^{(j)} \tag{9.12}$$

where $\sigma_x^{(j)}$ is a Pauli-X operator on spin j. From this, we form the unitary operator

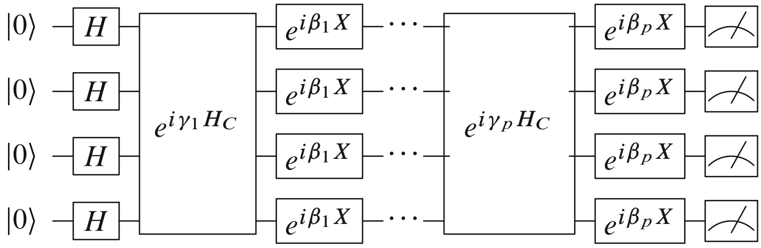
$$U(H_B, \beta) := e^{\ i\beta B} = \prod_{j=1}^{n} e^{\ i\beta \sigma_x^{(j)}} \tag{9.13}$$

Note that the second equality follows because all terms in the Hamiltonian commute with one another. We can view the term $e^{\ i\beta\sigma_x^{(j)}}$ as a rotation about the x axis on spin j by angle 2β. Thus, we can restrict $0 \le \beta < \pi$.

With these definitions, we can state the steps of the quantum part of the QAOA:

1. Start with an initial state that is an equal superposition over all bitstrings (spins)
$$|s\rangle = \frac{1}{\sqrt{2^n}} \sum_{b\in\{0,1\}^n} |b\rangle \tag{9.14}$$
by applying a Hadamard to each qubit $H^{\otimes n}|0\rangle^{\otimes n}$.
2. Evolve with the cost Hamiltonian by implementing $U(H_C, \gamma)$ for an angle γ.
3. Evolve with the mixer Hamiltonian by implementing $U(H_B, \beta)$ for an angle β.
4. Repeat steps (2) and (3) p times with different parameters γ_i, β_i at each step $i = 1, ..., p$ to form the state
$$|\gamma, \beta\rangle := \prod_{i=1}^{p} U(H_B, \beta_i)U(H_C, \gamma_i)|s\rangle \tag{9.15}$$
5. Measure in the computational basis to compute the expectation of H_C in this state:
$$F_p(\gamma, \beta) := \langle\gamma, \beta|H_C|\gamma, \beta\rangle \tag{9.16}$$
6. Use a (classical) optimization algorithm to (approximately) compute the maximum or minimum value of $F_p(\gamma, \beta)$. Alternatively, if you have other methods to determine the optimal angles, you may use these.
7. Sample from the output distribution of the circuit (9.15) to get a set of bitstrings **b**. The most probable bitstrings encode the approximate optima for the cost function.

The full circuit diagram for the quantum circuit in the QAOA is shown below.



The adiabatic theorem states that a system remains in its eigenstate even when subject to a perturbation, as long as that perturbation is slow and gradual enough and there is a gap between the eigenvalue of that state and the rest of the eigenvalues of the system (its spectrum) [39, 40, 115]. In other words if we have a system in a measured state and that state has enough of gap from other possible states of the system, then if we perturb the system slowly enough, it will not jump to another eigenstate. It can be shown using the adiabatic theorem that

$$\lim_{p\to\infty} \max_{\gamma,\beta} F_p(\gamma, \beta) = \max_b C(b). \tag{9.17}$$

That is, given enough parameters γ, β, we can be sure that the exact solution of the problem is attainable. The parameter p can thus be considered a hyperparameter. One form of approximation in the quantum *approximate* optimization algorithm is the finite cutoff for p. Another form of approximation is the ability of the classical optimizer to find the optimum.

However, in particular cases there are provable performance guarantees for $p = 1$ layers. For example, for $p = 1$ on 3-regular graphs, the QAOA always finds a cut that is at least 0.6924 times the size of the optimal cut [78]. Proving more worst-case or average-case performance guarantees is an interesting line of research on the analytic side of QAOA, and developing better classical optimization algorithms is an interesting area on the heuristic side of QAOA.

Example Implementation of QAOA

To get a better idea of how QAOA works, we now turn to an implementation. In this example, we consider the transverse field Ising model as a cost Hamiltonian:

$$H_C = -\sum_{\langle i,j\rangle} J_{ij}\sigma_z^{(i)}\sigma_z^{(j)} - \sum_i h_i\sigma_x^{(i)} \tag{9.18}$$

For simplicity of presentation, we set the transverse field coefficients to zero ($h_i = 0$) and set each interaction coefficient to one ($J_{ij} = 1$). The Hamiltonian can be modified in a straightforward way to generalize it, but these details are not important in a first encounter with QAOA. Another reason for this is that this system is trivial to solve analytically — thus, we can compare the solution found by QAOA to the exact solution. By making these simplifications, our cost Hamiltonian has the form

$$H_C = -\sum_{\langle i,j\rangle} \sigma_z^{(i)}\sigma_z^{(j)} \tag{9.19}$$

The graph (i.e., the arrangement of spins) we will consider is in a nearest neighbors configuration on a 2D grid. We thus need a way to implement the unitary operator

$$U(H_C, \gamma) := e^{-iH_C\gamma} = \prod_{\langle i,j\rangle} e^{i\gamma Z_i Z_j} \tag{9.20}$$

For simplicity, from here on we will substitute Z_i for $\sigma_z^{(i)}$. In order to implement this entire unitary, we need a sequence of gates for implementing each $e^{i\gamma Z_i Z_j}$ term, where i and j are neighbors in the graph. It will be convenient to rescale γ and consider implementing the unitary $e^{i\pi\gamma Z_i Z_j}$. To understand how to do this, note that the operator $Z \otimes Z$ is diagonal in the computational basis, hence $e^{i\pi\gamma Z\otimes Z}$ is just the exponential of each diagonal element (multiplied by $i\pi\gamma$)

$$\exp(i\pi\gamma Z \otimes Z) = \begin{bmatrix} e^{i\pi\gamma} & 0 & 0 & 0 \\ 0 & e^{\ i\pi\gamma} & 0 & 0 \\ 0 & 0 & e^{\ i\pi\gamma} & 0 \\ 0 & 0 & 0 & e^{i\pi\gamma} \end{bmatrix} \tag{9.21}$$

To get some intuition about how to implement this operator in terms of standard gates, note that the controlled-Z gate is diagonal with $C(Z) = \text{diag}(1, 1, 1, -1)$. Writing $-1 = e^{i\pi}$, we see that $C(Z) = \text{diag}(1, 1, 1, e^{i\pi})$, hence

$$C(Z^\gamma) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\pi\gamma} \end{bmatrix} \tag{9.22}$$

This gives us one diagonal term in the final unitary (9.21) that we want to implement. To get the other terms, we can apply X operators on the appropriate qubits. For example,

$$(I \otimes X) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\pi\gamma} \end{bmatrix} (I \otimes X) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{i\pi\gamma} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.23}$$

We can continue in this fashion to get all four diagonal elements, then get the full unitary (9.21) by simply multiplying them together (using the fact that the product of diagonal matrices is diagonal).

In the Cirq code below, we write a function which gives us a circuit for implementing the unitary $e^{\ i\pi\gamma Z_i Z_j}$. We then test our function by printing out the circuit for an example set of qubits i and j with an arbitrary value of γ and ensuring that the unitary matrix of this circuit is what we expect.

```
# Imports
import numpy as np
import matplotlib.pyplot as plt

import cirq
```

```
# Function to implement a ZZ gate on qubits a, b with angle gamma
def ZZ(a, b, gamma):
   """Returns a circuit implementing exp(-i \pi \gamma Z_i Z_j)."""
   # Get a circuit
   circuit = cirq.Circuit()

   # Gives the fourth diagonal component
   circuit.append(cirq.CZ(a, b)**gamma)

   # Gives the third diagonal component
   circuit.append([cirq.X(b), cirq.CZ(a,b)**(-1 * gamma), cirq.X(b)])

   # Gives the second diagonal component
   circuit.append([cirq.X(a), cirq.CZ(a,b)**-gamma, cirq.X(a)])

   # Gives the first diagonal component
   circuit.append([cirq.X(a), cirq.X(b), cirq.CZ(a,b)**gamma,
                   cirq.X(a), cirq.X(b)])

   return circuit

#26.s.one
# Make sure the circuit gives the correct matrix
qreg = cirq.LineQubit.range(2)
zzcirc = ZZ(qreg[0], qreg[1], 0.5)
print("Circuit for ZZ gate:", zzcirc, sep="\n")
print("\nUnitary of circuit:", zzcirc.to_unitary_matrix().round(2),
    sep="\n")
```

The output of this code is as follows:

```
Circuit for ZZ gate:
0: ---@-----------@------------X---@--------X---X---@-------X---
      |           |                |                |
1: ---@^0.5---X---@^-0.5---X-------@^-0.5-------X---@^0.5---X---

Unitary of circuit:
[[0.+1.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.-1.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.-1.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+1.j]]
```

As we can see by comparing with (9.21), this circuit indeed implements the desired unitary operator. Note that the circuit is not optimal — a trivial simplification is removing sequential X operators on qubit 0, and other optimizations are possible. Such optimizations will not concern us here, however.

In the next block of code, we define a 2x2 grid of qubits.

```
ncols = 2
nrows = 2
qreg = [[cirq.GridQubit(i,j) for j in range(ncols)] for i in
    range(nrows)]
```

Then we write functions for implementing the operators $U(H_C, \gamma)$ and $U(H_B, \beta)$.

```
#
# Function to implement the cost Hamiltonian
def cost_circuit(gamma):
   """Returns a circuit for the cost Hamiltonian."""
   circ = cirq.Circuit()
   for i in range(nrows):
      for j in range(ncols):
         if i < nrows - 1:
            circ += ZZ(qreg[i][j], qreg[i + 1][j], gamma)
         if j < ncols - 1:
            circ += ZZ(qreg[i][j], qreg[i][j + 1], gamma)

   return circ

# Function to implement the mixer Hamiltonian
def mixer(beta):
  """Generator for U(H_B, beta) layer (mixing layer)"""
  for row in qreg:
   for qubit in row:
    yield cirq.X(qubit)**beta
```

These functions allow us to construct the entire QAOA circuit. The function below builds this circuit for an arbitrary number p of parameters.

```
# Function to build the QAOA circuit
def qaoa(gammas, betas):
   """Returns a QAOA circuit."""
   circ = cirq.Circuit()
   circ.append(cirq.H.on_each(*[q for row in qreg for q in row]))

   for i in range(len(gammas)):
      circ += cost_circuit(gammas[i])
      circ.append(mixer(betas[i]))

   return circ
```

Now that we can build our QAOA circuit for a given set of parameters, we can compute the expectation of the cost Hamiltonian in the final state (9.16). For simplicity we use Cirq's ability to access the wavefunction to compute this expectation rather than sampling from the circuit itself. The following function shows how we can access the wavefunction after applying a circuit:

```
def simulate(circ):
   """Returns the wavefunction after applying the circuit."""
   sim = cirq.Simulator()
   return sim.simulate(circ).final_state
```

The next function evaluates the expectation using the wavefunction:

```
def energy_from_wavefunction(wf):
   """Computes the energy-per-site of the Ising Model from the
       wavefunction."""
   # Z is a (n_sites x 2**n_sites) array. Each row consists of the
   # 2**n_sites non-zero entries in the operator that is the Pauli-Z
       matrix on
   # one of the qubits times the identites on the other qubits. The
       (i*n_cols + j)th
   # row corresponds to qubit (i,j).
   Z = np.array([(-1)**(np.arange(2**nsites) >> i)
            for i in range(nsites-1,-1,-1)])

   # Create the operator corresponding to the interaction energy
       summed over all
   # nearest-neighbor pairs of qubits
   ZZ_filter = np.zeros_like(wf, dtype=float)
   for i in range(nrows):
      for j in range(ncols):
         if i < nrows-1:
            ZZ_filter += Z[i*ncols + j]*Z[(i+1)*ncols + j]
         if j < ncols-1:
            ZZ_filter += Z[i*ncols + j]*Z[i*ncols + (j+1)]

   # Expectation value of the energy divided by the number of sites
   return -np.sum(np.abs(wf)**2 * ZZ_filter) / nsites
```

Finally, for convenience, we define a function that computes the energy/cost directly from a set of parameters. This function uses the parameters to build a circuit, then gets the wavefunction of the final state and lastly computes the energy/cost using the previous function.

```
def cost(gammas, betas):
   """Returns the cost function of the problem."""
   wavefunction = simulate(qaoa(gammas, betas))
   return energy_from_wavefunction(wavefunction)
```

These functions provide the set up for QAOA, and we could now optimize the parameters to minimize the cost. For instructional purposes, we implement QAOA with $p = 1$ layers and perform a grid search, plotting the 2D cost landscape for each parameter γ and β. The function for the grid search over a range of parameters is given below:

```
def grid_search(gammavals, betavals):
   """Does a grid search over all parameter values."""
   costmat = np.zeros((len(gammavals), len(betavals)))

   for (i, gamma) in enumerate(gammavals):
      for (j, beta) in enumerate(betavals):
         costmat[i, j] = cost([gamma], [beta])

   return costmat
```

Finally, here is the code for using this function within the main script and plotting the cost landscape:

```
# Get a range of parameters
gammavals = np.linspace(0, 1.0, 50)
betavals = np.linspace(0, np.pi, 75)

# Compute the cost at all parameter values using a grid search
costmat = grid_search(gammavals, betavals)

# Plot the cost landscape
plt.imshow(costmat, extent=(0, 1, 0, np.pi), origin="lower",
    aspect="auto")
plt.colorbar()
plt.show()
```

The output of this section of the program is shown in Figure 9.3. As we can see, there is a significant amount of symmetry in the cost landscape. This phenomena is typical in variational quantum algorithms. Apart from the symmetry which arises naturally from the Ising Hamiltonian, the symmetric and periodic form of the cost landscape arises from symmetries in the ansatz circuit. Exploiting these symmetries can help lead classical optimization algorithms to a good solution more quickly.

We can now obtain a set of optimal parameters by taking the coordinates of a minimum in our cost landscape. The following short block of code does this and prints out the numerical value of the cost at these parameters.

```
# Coordinates from the grid of cost values
gamma_coord, beta_coord = np.where(costmat == np.min(costmat))

# Values from the coordinates
gamma_opt = gammavals[gamma_coord[0]]
beta_opt = betavals[beta_coord[0]]
```

Now that we have the optimal parameters, we can run the QAOA circuit with these parameters and measure in the computational basis to get bitstrings that solve our original optimization problem. The function below runs the circuit and returns the measurement results.

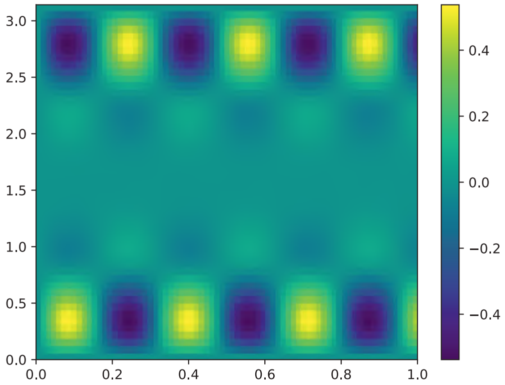


Figure 9.3: Cost landscape of the Ising Hamiltonian computed from one layer of QAOA

```
def get_bit_strings(gammas, betas, nreps=10000):
   """Measures the QAOA circuit in the computational basis to get
       bitstrings."""
   circ = qaoa(gammas, betas)
   circ.append(cirq.measure(*[qubit for row in qreg for qubit in
       row], key='m'))

   # Simulate the circuit
   sim = cirq.Simulator()
   res = sim.run(circ, repetitions=nreps)

   return res
```

Finally, we use this function to sample from the circuit at the optimal parameters found above. Then, we parse the output and print out the two most common bitstrings sampled from the circuit.

```
# Sample to get bits and convert to a histogram
bits = get_bit_strings([gamma_opt], [beta_opt])
hist = bits.histogram(key="m")

# Get the most common bits
top = hist.most_common(2)

# Print out the two most common bitstrings measured
print("\nMost common bitstring:")
print(format(top[0][0], "#010b"))

print("\nSecond most common bitstring:")
print(bin(top[1][0]))
```

A sample output of this portion of the code follows:

```
Most common bitstring:
0b000000000

Second most common bitstring:
0b111111111
```

These bitstrings are exactly the ones we would expect to minimize our cost function! Recall the Ising Hamiltonian we considered, which when written classically has the form

$$C(z) = -\sum_{\langle i,j \rangle} z_i z_j \tag{9.24}$$

where $z_i = \pm 1$ are spins. For our bitstring output, $b = 0$ corresponds to spin up ($z = 1$) and $b = 1$ corresponds to spin down ($z = -1$). Since opposite spins $z_i \neq z_j$ will produce a term with a positive contribution (don't forget the overall minus sign in front!) in the sum, the minimum value of the cost function occurs when all spins are *aligned*. That is, $z_i = z_j$ for all i, j. The bitstrings that we measured correspond to all spins aligned down or all spins aligned up, respectively. Thus, these bitstrings indeed produced the minimum value for our cost function, and QAOA was able to successfully optimize the cost.

For larger optimization problems with more complex cost functions, more layers in the QAOA ansatz (i.e., $p > 1$) may become necessary. More layers means more parameters in the variational quantum circuit, which leads to a harder optimization problem. Such an optimization problem could not be solved by a mere grid search over values, as this quickly becomes intractable. Rather, gradient-based or gradient-free optimization algorithms must be used to compute an approximately optimal set of parameters.

The complete program for this implementation of QAOA is available on the book's online site.

9.4 *Machine Learning on Quantum Processors*

Several groups are exploring the use of QC for machine learning; it is natural to ask whether QC affords us any advantage in this area. Speedup is not the only advantage we should consider in quantum machine learning (QML). There may be opportunity to use a QC to process data directly from a quantum

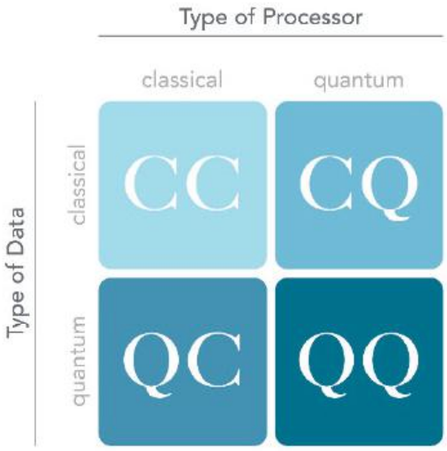


Figure 9.4: Data types and processor types

sensor that retains the full range of quantum information from that sensor. Figure 9.4 points to the potential of matching quantum data with quantum processing. Having a classifier on the QC directly analyzing the datastream for patterns may be better than piping the data to a classical computer.

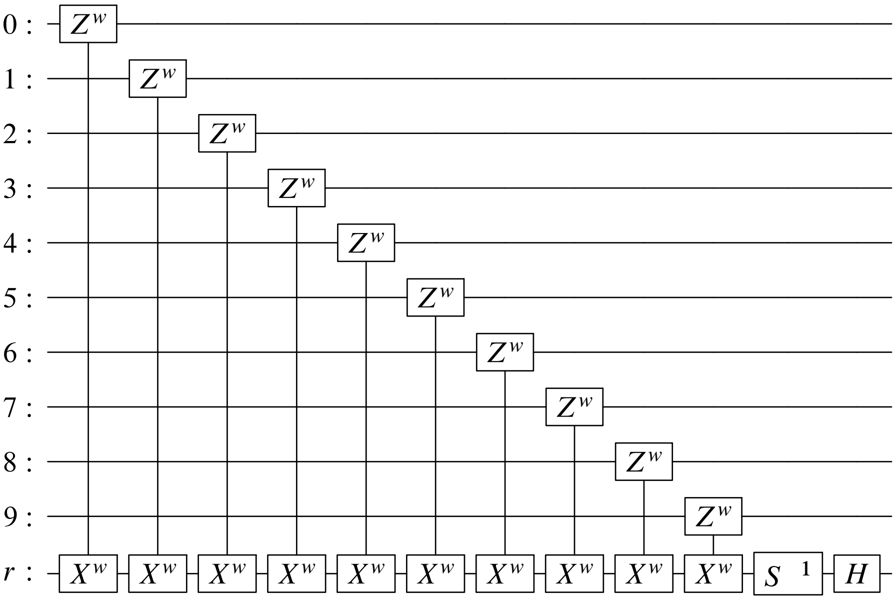
A number of groups have published in this area, including:

1. Alan Aspuru-Guzik and colleagues have explored quantum machine learning as well as hybrid classical-quantum models [51, 189].
2. The Rigetti team has worked on unsupervised machine learning on a classical-quantum hybrid approach [166].
3. Farhi and Neven laid out an approach to classification with neural networks on a quantum processor (QNNs) [82].
4. Wittek and Gogolin explored Markov logic networks on quantum platforms [236].
5. See [31] for additional work in QML and [235] for an online course in QML.

The following QNN code comes from [132]. We begin by defining the QNN:

```
import cirq
import numpy as np

class ZXGate(cirq.ops.eigen_gate.EigenGate,
          cirq.ops.gate_features.TwoQubitGate):
  """ZXGate with variable weight."""
 def __init__(self, weight=1):
   """Initializes the ZX Gate up to phase.

    Args:
       weight: rotation angle, period 2
   """
   self.weight = weight
   super().__init__(exponent=weight) # Automatically handles weights
       other than 1

 def _eigen_components(self):
   return [
      (1, np.array([[0.5, 0.5, 0, 0],
               [ 0.5, 0.5, 0, 0],
               [0, 0, 0.5, -0.5],
               [0, 0, -0.5, 0.5]])),
      (-1, np.array([[0.5, -0.5, 0, 0],
               [ -0.5, 0.5, 0, 0],
               [0, 0, 0.5, 0.5],
               [0, 0, 0.5, 0.5]]))
   ]

 # This lets the weight be a Symbol. Useful for parameterization.
 def _resolve_parameters_(self, param_resolver):
   return ZXGate(weight=param_resolver.value_of(self.weight))

 # How should the gate look in ASCII diagrams-
 def _circuit_diagram_info_(self, args):
   return cirq.protocols.CircuitDiagramInfo(
        wire_symbols=('Z', 'X'),
        exponent=self.weight)

# Total number of data qubits
INPUT_SIZE = 9

data_qubits = cirq.LineQubit.range(INPUT_SIZE)
readout = cirq.NamedQubit('r')

# Initialize parameters of the circuit
params = {'w': 0}

def ZX_layer():
 """Adds a ZX gate between each data qubit and the readout.
 All gates are given the same cirq.Symbol for a weight."""
 for qubit in data_qubits:
   yield ZXGate(cirq.Symbol('w')).on(qubit, readout)


qnn = cirq.Circuit()
qnn.append(ZX_layer())
qnn.append([cirq.S(readout)**-1, cirq.H(readout)]) # Basis
    transformation
```

The QNN circuit we have constructed can be visualized as follows:



The $Z^w - X^w$ notation represents a ZX gate with weight w. The final qubit labeled r is the readout qubit. Notice that the final operations of $S^{\ 1}$ and H perform a basis change so that our measurement at the end will effectively be in the Y-basis.

Next we define functions that let us access the expectation values of Z and the loss function for the QNN:

```
def readout_expectation(state):
  """Takes in a specification of a state as an array of 0s and 1s
  and returns the expectation value of Z on ther readout qubit.
  Uses the Simulator to calculate the wavefunction exactly."""

  # A convenient representation of the state as an integer
  state_num = int(np.sum(state*2**np.arange(len(state))))

  resolver = cirq.ParamResolver(params)
  simulator = cirq.Simulator()

  # Specify an explicit qubit order so that we know which qubit is
      the readout
  result = simulator.simulate(qnn, resolver,
      qubit_order=[readout]+data_qubits,
                       initial_state=state_num)
  wf = result.final_state

  # Since we specified qubit order, the Z value of the readout is the
      most
  # significant bit.
  Z_readout = np.append(np.ones(2**INPUT_SIZE),
      -np.ones(2**INPUT_SIZE))
  # Use np.real to eliminate +0j term
  return np.real(np.sum(wf*wf.conjugate()*Z_readout))

def loss(states, labels):
  loss=0
  for state, label in zip(states,labels):
    loss += 1 - label*readout_expectation(state)
  return loss/(2*len(states))

def classification_error(states, labels):
  error=0
  for state,label in zip(states,labels):
    error += 1 - label*np.sign(readout_expectation(state))
  return error/(2*len(states))
```

Now we generate some data for a toy problem:

```
def make_batch():
  """Generates a set of labels, then uses those labels to generate
        inputs.
  label = -1 corresponds to majority 0 in the sate, label = +1
        corresponds to
  majority 1.
  """
  np.random.seed(0) # For consistency in demo
  labels = (-1)**np.random.choice(2, size=100) # Smaller batch sizes
        will speed up computation
  states = []
  for label in labels:
    states.append(np.random.choice(2, size=INPUT_SIZE,
          p=[0.5-label*0.2,0.5+label*0.2]))
  return states, labels

states, labels = make_batch()
```

Finally, we can do a brute-force search over parameter space to find the optimal QNN:

```
linspace = np.linspace(start=-1, stop=1, num=80)
train_losses = []
error_rates = []
for p in linspace:
  params = {’w’: p}
  train_losses.append(loss(states, labels))
  error_rates.append(classification_error(states, labels))
plt.plot(linspace, train_losses)
plt.xlabel(’Weight’)
plt.ylabel(’Loss’)
plt.title(’Loss as a Function of Weight’)
plt.show()
```

We can plot the loss as a function of the weights to see how the network performs. This is illustrated in Figure 9.5. The minimal loss is about 0.2,

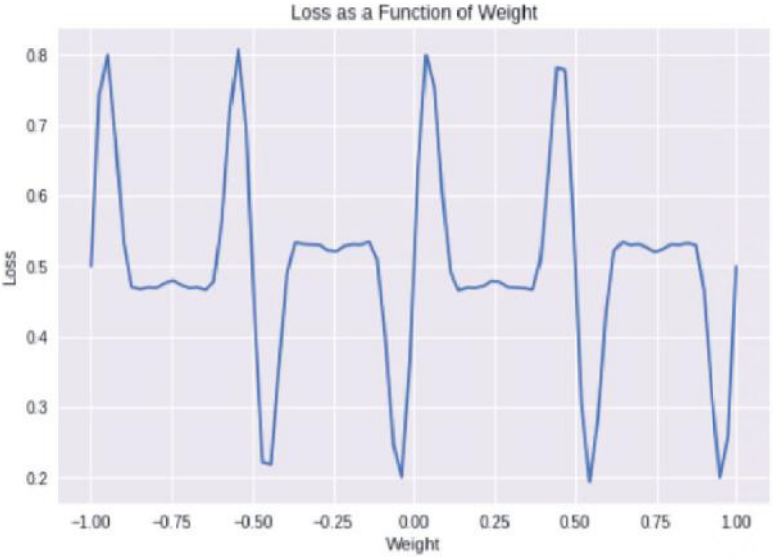


Figure 9.5: Loss function of the QNN plotted against the weight; the weight should be chosen to minimize the loss

which matches what you can obtain by a linear model. A more complicated QNN, as discussed in [82], can do more.

For this type of classification problem, it remains to be seen whether a QNN has an advantage over a classical model. More generally, advantages in quantum machine learning seem to be elusive in the early stages of this field. Until recently, a quantum algorithm for *recommendation systems* achieved an exponential speedup over the best known classical algorithm [118]. Briefly, the general idea of a recommendation system is as follows: Given an incomplete *preference matrix* P of m users and their feedback on n products, output a good recommendation for a particular user. Here, "incomplete" means that entries of the matrix are missing — that is, not every user has provided feedback for every product.

Prior to the work [118], the best classical algorithm had a runtime that scaled linearly in the matrix dimension mn. The quantum recommendation algorithm scales *poly*logarithmically in mn, specifically as

$$O(\text{poly}(\kappa)\text{polylog}(mn))$$

where κ is the condition number of P. While this was a staple of the QML field, a new breakthrough classical algorithm, inspired by the quantum one, also achieved polylogarithmic scaling in the matrix dimension [216].

Depending on one's perspective, this is either a pro or con for quantum machine learning. The pro is that the classical algorithm was directly inspired by the quantum one — without QML, we may have never had this insight. The con is that a staple result of the QML field has been "dequantized." Much research continues to be done in quantum machine learning, for example [106, 197, 222], to explore the possibilities and prospects for this relatively young field.

9.5 *Quantum Phase Estimation*

Quantum phase estimation (QPE), also known as the phase estimation algorithm (PEA), is an algorithm for determining the eigenvalues of a unitary operator. Eigenvalue problems, which have the form

$$A\mathbf{x} = \lambda \mathbf{x} \tag{9.25}$$

where $A \in \mathbb{C}^{2^m \times 2^m}$, $\mathbf{x} \in \mathbb{C}^{2^m}$ and $\lambda \in \mathbb{C}$, are ubiquitous throughout mathematics and physics. In mathematics, applications range from graph theory to partial differential equations. In physics, applications include computing the ground state energy — the smallest eigenvalue of the Hamiltonian of the system — for nuclei, molecules, materials and other physical systems. Moreover, principal component analysis (PCA), an algorithm for reducing the dimensionality of feature vectors in machine learning, has an eigenvalue problem at its core. The applications of (9.25) range across a wide spectrum of disciplines.

In the quantum case, we are concerned with finding the eigenvalues of a unitary operator U. It follows immediately by the definition of unitarity ($U^\dagger U = I$) that eigenvalues of a unitary operator have modulus one: $|\lambda| = 1$. Thus, any eigenvalue λ of a unitary operator can be written in the form

$$\lambda = e^{2\pi i \varphi} \tag{9.26}$$

where $0 \leq \varphi \leq 1$ is called the *phase*. This is the same phase that appears in the name of the algorithm — quantum phase estimation. By estimating φ, we get an estimate of the eigenvalue λ via the equation above.

Suppose φ can be written exactly using n bits[5]

$$\varphi = 0.\varphi_1\varphi_2\cdots\varphi_n \tag{9.27}$$

This is a binary decimal representation of the phase φ. Here, each φ_k for $k = 1, ..., n$ is a binary digit $\varphi_k \in \{0, 1\}$. We can write this equivalently as

$$\varphi = \sum_{k=1}^{n} \varphi_k 2^{\ \ k} \tag{9.28}$$

The key to understanding QPE is to consider the action of controlling the unitary operator on an eigenstate $|\psi\rangle$. Explicitly, let U be a unitary operator which we take as input to the QPE algorithm such that

$$U|\psi\rangle = \lambda|\psi\rangle \tag{9.29}$$

Suppose for now we have the eigenstate $|\psi\rangle$. This is not a requirement for QPE — in fact, it makes the algorithm trivial, for if we knew $|\psi\rangle$ we could just implement $U|\psi\rangle$ on the quantum computer — it just simplifies the explanation. Now, suppose we prepare the equal superposition state (ignoring normalization factors) in the first register and the eigenstate of U in the second register

$$(|0\rangle + |1\rangle) \otimes |\psi\rangle = |0\rangle|\psi\rangle + |1\rangle|\psi\rangle \tag{9.30}$$

Now, as mentioned, we implement a controlled-U operation on this state, which produces the following state:

$$\begin{aligned} |0\rangle|\psi\rangle + |1\rangle U|\psi\rangle &= |0\rangle|\psi\rangle + e^{2\pi i 0.\varphi_1\cdots\varphi_n}|1\rangle|\psi\rangle \\ &= (|0\rangle + e^{2\pi i 0.\varphi_1\cdots\varphi_n}|1\rangle) \otimes |\psi\rangle \end{aligned}$$

Note that the second register goes unchanged. Since $|\psi\rangle$ is an eigenstate of U, it is unaffected by the controlled operation. Why did we do this then? We encoded the information about the phase into the first region. Specifically, the state in the first register picked up the relative phase $e^{2\pi i 0.\varphi_1\cdots\varphi_n}$.

Phase estimation now tells us to implement controlled-U^{2^k} operations for integers $k = 0, ..., n-1$. We already performed the $k = 0$ case above. Consider now the effect of U^2; in particular,

$$U^2|\psi\rangle = \lambda^2|\psi\rangle = e^{2\pi i(2\varphi)}|\psi\rangle = e^{2\pi i 0.\varphi_2\cdots\varphi_n}|\psi\rangle \tag{9.31}$$

[5] This is the case where φ is rational. The general case of φ being irrational (requiring infinitely many bits) is similar, but for simplicity we won't cover it here. See [161] for an explanation.

In the last step, we used the fact that $e^{2\pi i \varphi_1} = 1$ for any $\varphi_1 \in \{0, 1\}$. Thus, by preparing the equal superposition state in the first register, the eigenstate $|\psi\rangle$ in the second register (9.30), then performing a controlled-U^2 operation, we get the state

$$|0\rangle|\psi\rangle + |1\rangle U^2|\psi\rangle = |0\rangle|\psi\rangle + e^{2\pi i 0.\varphi_2 \cdots \varphi_n}|1\rangle|\psi\rangle$$

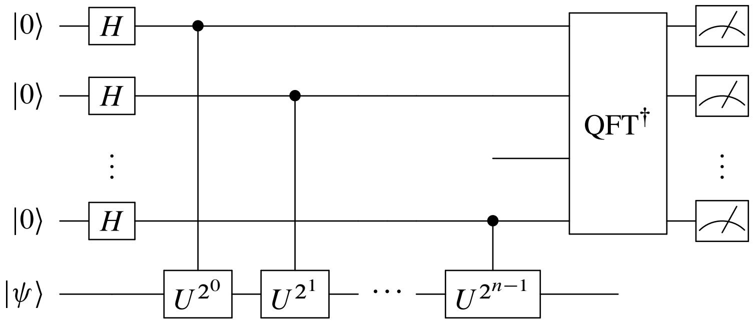
In general, using this same idea, we can see that

$$U^{2^k}|\psi\rangle = \lambda^{2^k}|\psi\rangle = e^{2\pi i (2^k \varphi)}|\psi\rangle = e^{2\pi i 0.\varphi_{k+1} \cdots \varphi_n} \tag{9.32}$$

for $k = 0, ..., n - 1$. Hence, we can transform (9.30) under a controlled-U^{2^k} as

$$|0\rangle|\psi\rangle + |1\rangle|\psi\rangle \longmapsto |0\rangle|\psi\rangle + |1\rangle U^{2^k}|\psi\rangle = (|0\rangle + e^{2\pi i 0.\varphi_{k+1} \cdots \varphi_n}|1\rangle) \otimes |\psi\rangle \tag{9.33}$$

Equation (9.33) is at the heart of the QPE algorithm. In particular, the algorithm says to implement this operation iteratively for $k = 0, ..., n - 1$, using n qubits in the top register with the eigenstate $|\psi\rangle$ in the bottom register. The full circuit for QPE is shown below:



After implementing the series of controlled-U^{2^k} operations, the top register is in the state

$$(|0\rangle + e^{2\pi i 0.\varphi_1 \cdots \varphi_n}|1\rangle) \otimes (|0\rangle + e^{2\pi i 0.\varphi_2 \cdots \varphi_n}|1\rangle) \otimes \cdots \otimes (|0\rangle + e^{2\pi i 0.\varphi_n}|1\rangle) \tag{9.34}$$

To extract the phase information from this state, we use the inverse Fourier transform, which transforms this state to a product state

$$|\varphi_1\rangle \otimes |\varphi_2\rangle \otimes \cdots \otimes |\varphi_n\rangle \tag{9.35}$$

By measuring in the computational basis, we thus learn the bits $\varphi_1, \varphi_2, ..., \varphi_n$, which allow us to construct $\varphi = 0.\varphi_1 \cdots \varphi_n$ and the eigenvalue

$$\lambda = e^{2\pi i \varphi} \tag{9.36}$$

Implemention of QPE

We now turn to an example implementation of QPE using Cirq. Here, we consider computing the eigenvalues of the unitary

$$U = X \otimes Z \tag{9.37}$$

We can see that the eigenvalues of U are, of course, ± 1. We will see that QPE returns these eigenvalues as well.

First, we import the necessary packages

```
# Imports
import numpy as np

import cirq
```

and then define a helper function for converting from bitstrings in binary decimal notation to numeric values:

```
def binary_decimal(string):
   """Returns the numeric value of 0babc... where a, b, c, ... are
       bits.

   Examples:
      0b10 --> 0.5
      0b01 --> 0.25
   """
   val = 0.0
   for (ind, bit) in enumerate(string[2:]):
      if int(bit) == 1:
         val += 2**(-1 -ind)
   return val
```

Now we define the number of qubits in the bottom register of the QPE circuit, create the unitary matrix and classically diagonalize it. We will use these eigenvalues to compare to the ones found by QPE.

```
# Number of qubits and dimension of the eigenstate
m = 2

# Get a unitary matrix on two qubits
xmat = np.array([[0, 1], [1, 0]])
zmat = np.array([[1, 0], [0, -1]])
unitary = np.kron(xmat, zmat)

# Print it to the console
print("Unitary:")
print(unitary)

# Diagonalize it classically
evals, _ = np.linalg.eig(unitary)
```

The output of this portion of the code follows:

```
Unitary:
[[ 0  0  1  0]
 [ 0  0  0 -1]
 [ 1  0  0  0]
 [ 0 -1  0  0]]
```

Now that we have our input to QPE, we can begin building the circuit. As described above, we define the number of qubits in our top register to determine the accuracy of the eigenvalues found. Here, we set this number and define two registers of qubits. Next, we create a circuit and apply the Hadamard gate to each qubit in the top (readout) register.

```
# Number of qubits in the readout/answer register (# bits of
    precision)
n = 2

# Readout register
regA = cirq.LineQubit.range(n)

# Register for the eigenstate
regB = cirq.LineQubit.range(n, n + m)

# Get a circuit
circ = cirq.Circuit()

# Hadamard all qubits in the readout register
circ.append(cirq.H.on_each(*regA))
```

The next step in the QPE algorithm is to implement the series of controlled U^{2^k} operations. We show how this can be done in Cirq for arbitrary two-qubit unitaries written as matrices. First, we create a `TwoQubitMatrixGate` from the unitary matrix, then make a controlled version.

```
# Get a Cirq gate for the unitary matrix
ugate = cirq.ops.matrix_gates.TwoQubitMatrixGate(unitary)

# Controlled version of the gate
cugate = cirq.ops.ControlledGate(ugate)
```

Now that we have the controlled-U gate, we can implement the sequence of transforms with the following code block:

```
# Do the controlled U^{2^k} operations
for k in range(n):
   circ.append(cugate(regA[k], *regB)**(2**k))
```

The last step in QPE is to implement the inverse quantum Fourier transform and measure all qubits in the computational basis. This is done in the following code block.

```
# Do the inverse QFT
for k in range(n - 1):
   circ.append(cirq.H.on(regA[k]))
   targ = k + 1
   for j in range(targ):
      exp = -2**(j - targ)
      rot = cirq.Rz(exp)
      crot = cirq.ControlledGate(rot)
      circ.append(crot(regA[j], regA[targ]))
circ.append(cirq.H.on(regA[n - 1]))

# Measure all qubits in the readout register
circ.append(cirq.measure(*regA, key="z"))
```

Now that we have built our QPE circuit, we can run it and process the results. The code below simulates the circuit and grabs the top two most frequent measurement outcomes. We can obtain φ_1 and φ_2 from each of these to compute our eigenvalues.

```
# Get a simulator
sim = cirq.Simulator()

# Simulate the circuit and get the most frequent measurement outcomes
res = sim.run(circ, repetitions=1000)
hist = res.histogram(key="z")
top = hist.most_common(2)
```

Even though we do not start the second register in an eigenstate of the unitary U, we can think of starting the second register in a linear combination of its eigenstates since the eigenstates of U form an orthonormal basis; in other words, any vector can be expressed as a linear combination of the eigenstates for some coefficients. In particular, we started the second register in the ground state $|0\rangle$, which we can write as

$$|0\rangle = \sum_j c_j |j\rangle \tag{9.38}$$

where $|j\rangle$ are the eigenstates of U. The most probable measurement outcomes are thus those with large $|c_j| = |\langle 0|j\rangle|$.

Now that we have the most frequently measured bitstrings, we can convert these to numerical values of the phase φ and then to eigenvalues. The code below performs these operations and prints out the eigenvalues computed by QPE and the eigenvalues computed by the classical matrix diagonalization algorithm.

```
# Eigenvalues from QPE
estimated = [np.exp(2j * np.pi * binary_decimal(bin(x[0]))) for x in
    top]

# Print out the estimated eigenvalues
print("\nEigenvalues from QPE:")
print(set(sorted(estimated, key=lambda x: abs(x)**2)))

# Print out the actual eigenvalues
print("\nActual eigenvalues:")
print(set(sorted(evals, key=lambda x: abs(x)**2)))
```

The output of this code, shown below, reveals that QPE finds the correct eigenvalues within numerical roundoff precision.

```
Eigenvalues from QPE:
{(1+0j), (-1+1.2246467991473532e-16j)}

Actual eigenvalues:
{1.0, -1.0}
```

The complete program for this implementation can be found on the book's online website. One can change the unitary matrix to compute eigenvalues and compare them to the ones found classically. Additionally, one can change the number of qubits in the top register n to get more bits of precision for more complex unitary operators.

9.6 *Solving Linear Systems*

The problem of solving a linear system of M equations with N variables is ubiquitous in mathematics, science and engineering. The formal statement of the problem is as follows:

> Given an $M \times N$ matrix A and a *solution vector* $\mathbf{b}$, find a vector $\mathbf{x}$ such that

$$A\mathbf{x} = \mathbf{b} \tag{9.39}$$

Linear algebra tells us how to solve this problem classically in the case that A is invertible:[6]

$$\mathbf{x} = A^{\;1}\mathbf{b} \tag{9.40}$$

However, although we can write down the solution immediately, numerically computing $\mathbf{x}$ is intractable for large matrices.

[6] Please consult Part III: Toolkit for a review of these mathematical concepts.

Explicitly computing the inverse of A is generally the most costly method. In practice, most general-purpose numerical solvers use Gaussian elimination and back-substitution, which runs in $O(N^3)$ time. In this discussion we restrict ourselves to square matrices, i.e., $M = N$. Faster classical algorithms are possible: if the matrix A has sparsity s and condition number κ, solving the system to accuracy ϵ can be done by the conjugate gradient algorithm in $O(Ns\kappa \log(1/\epsilon))$ time, which is a considerable speedup compared with $O(N^3)$.

The quantum version of solving systems of linear equations — called the quantum linear systems problem (QLSP) [64] — is similar to the classical approach. Let A be an $N \times N$ Hermitian matrix with unit determinant. Let $\mathbf{b}$ and $\mathbf{x}$ be N-dimensional vectors such that $\mathbf{x} = A \quad {}^{1}\mathbf{b}$. Define the quantum states $|b\rangle$ and $|x\rangle$ on $n = \log_2 N$ qubits

$$|b\rangle = \frac{\sum_i b_i |i\rangle}{|| \sum_i b_i |i\rangle ||_2} \tag{9.41}$$

and

$$|x\rangle = \frac{\sum_i x_i |i\rangle}{|| \sum_i x_i |i\rangle ||_2} \tag{9.42}$$

Here, b_i is the ith component of $\mathbf{b}$, and similarly for x_i.

The goal of the QLSP is as follows: given access to the matrix A (whose elements are accessed by an oracle) and the state $|b\rangle$, output a state $|\bar{x}\rangle$ such that

$$|| |\bar{x}\rangle - |x\rangle ||_2 \leq \epsilon \tag{9.43}$$

with probability greater than $1/2$.

A quantum algorithm for solving the QLSP in time $O(\log(N) s^2 \kappa^2 / \epsilon)$ was discovered by Harrow, Hassidim and Lloyd [103]. The algorithm is commonly known as the HHL algorithm after its developers. Note that the order of HHL is logarithmic, however, this is not always the case in practice. As Aaronson has pointed out:

> ...the HHL algorithm solves Ax = b in logarithmic time, but it does so with four caveats...each of which can be crucial in practice. To make a long story short, HHL is not exactly an algorithm for solving a system of linear equations in logarithmic time. Rather, it's an algorithm for approximately preparing a quantum superposition of the form $|x\rangle$, where x is the solution to a linear system Ax = b, assuming the ability to rapidly prepare

> the state $|x\rangle$, and to apply the unitary transformation $e^{\ iAt}$, and using an amount of time that grows roughly like $\kappa s(log(n))/\epsilon$, where n is the system size, κ is the system's condition number, s is its sparsity and ϵ is the desired error [4].

In the remainder of this section, we explain the mathematics of HHL and then turn to an example implementation. HHL uses several quantum algorithms we have discussed — such as Hamiltonian and quantum phase estimation — as subroutines.

Description of the HHL Algorithm

The HHL algorithm uses three registers of qubits, which we denote as A for ancilla, W for work, and *IO* for input/output. The input to the algorithm is the quantum state $|b\rangle$, defined above, which is input to the *IO* register. The other registers start off in the $|0\rangle$ state, so the entire initial state input to HHL can be written

$$|\psi_0\rangle := |0\rangle_A \otimes |0\rangle_W \otimes |b\rangle_{IO} \tag{9.44}$$

We are also given the matrix A as input. There should be no confusion between the matrix and the register, for we will refer to the former as "matrix A" and the latter as "register A." There are three main steps to the algorithm:

1. Quantum phase estimation with the unitary $U_A := e^{iAt}$, controlled by the W register and U_A applied to the *IO* register.
2. Pauli-Y rotation for a particular angle θ (discussed below) on the A register controlled by the W register.
3. Implement the first step in reverse (known as *uncomputation*) on the W register.

If the A register is measured and one post-selects on the $|1\rangle_A$ outcome, then the state of the *IO* register will be close to $|x\rangle$. We now walk through the steps to show this.

Let the matrix A be written in its eigenbasis

$$A = \sum_j \lambda_j |u_j\rangle\langle u_j| \tag{9.45}$$

For simplicity, we assume for the moment that $|b\rangle$ is one of the eigenvectors of A. That is, $|b\rangle = |u_j\rangle$ for some index j. This assumption will be relaxed momentarily.

We can assume A is Hermitian without loss of generality, since if A is not Hermitian, we can form a Hermitian matrix

$$\tilde{A} := \begin{bmatrix} 0 & A^\dagger \\ A & 0 \end{bmatrix}$$

and perform HHL with $\tilde{A}$. Since A is Hermitian, the operator

$$U_A := e^{iAt} \tag{9.46}$$

is unitary and has eigenvalues $e^{i\lambda_j t}$ and eigenstates $|u_j\rangle$. After the first step of HHL, QPE brings us to the state

$$|\psi_1\rangle := |0\rangle_A \otimes |\tilde{\lambda}_j\rangle_W \otimes |u_j\rangle_{IO} \tag{9.47}$$

Here, $\tilde{\lambda}_j$ is a binary representation of λ_j up to a set precision. Note that we used our assumption $|b\rangle = |u_j\rangle$ when writing the result of QPE.

We now implement the second step of QPE, a controlled-Y rotation $e^{\ i\theta Y}$ for the angle

$$\theta = \arccos \frac{C}{\tilde{\lambda}} \tag{9.48}$$

Here, C is a hyperparameter set by the user of the algorithm. In the example implementation below, we discuss setting the value of C. After this rotation controlled on the O register, we have the state

$$|\psi_2\rangle := \sqrt{1 - \frac{C^2}{\tilde{\lambda}_j^2}}|0\rangle_A \otimes |\tilde{\lambda}_j\rangle_W \otimes |u_j\rangle_{IO} + \frac{C}{\tilde{\lambda}_j}|1\rangle_A \otimes |\tilde{\lambda}_j\rangle_W \otimes |u_j\rangle_{IO} \tag{9.49}$$

We now relax the assumption that $|b\rangle$ is an eigenstate of A. That is, we relax the assumption that $|b\rangle = |u_j\rangle$ for some j. Note that we can write without any assumptions, however, that

$$|b\rangle = \sum_j \beta_j |u_j\rangle \tag{9.50}$$

where $\beta_j = \langle u_j|b\rangle$ are complex coefficients. We can write this because A is Hermitian and so its eigenstates form an orthonormal basis.

We now perform the above analysis (9.45) — (9.49) with $|b\rangle$ expressed in the eigenbasis (9.50). Doing so, we end up with the state

$$|\psi_3\rangle := \sum_{j=1}^{N} \beta_j \left[\sqrt{1 - \frac{C^2}{\tilde{\lambda}_j^2}}|0\rangle_A + \frac{C}{\tilde{\lambda}_j}|1\rangle_A \right] \otimes |\tilde{\lambda}_j\rangle_W \otimes |u_j\rangle_{IO} \tag{9.51}$$

The next step of HHL is to uncompute the W register. Doing so sends $|\tilde{\lambda}_j\rangle_O \rightarrow |0\rangle_O$. Since this state is the all zeros state, we can omit it and write the state after uncomputing as

$$|\psi_4\rangle := \sum_{j=1}^{N} \beta_j \left[\sqrt{1 - \frac{C^2}{\tilde{\lambda}_j^2}}|0\rangle_A + \frac{C}{\tilde{\lambda}_j}|1\rangle_A \right] \otimes |u_j\rangle_{IO} \tag{9.52}$$

This state is in a very useful form, though it may take a careful look to see why. The reason is that

$$A\;\,^{1}|b\rangle = \sum_{j=1}^{N} \frac{\beta_j}{\lambda_j}|u_j\rangle \tag{9.53}$$

Thus, if we measure the A register and post-select on the $|1\rangle_A$ outcome, then (9.52) becomes (ignoring the A register)

$$|\psi_5\rangle := \sum_{j=1}^{N} \frac{\beta_j}{\tilde{\lambda}_j}|u_j\rangle_{IO} \approx |x\rangle. \tag{9.54}$$

Thus, the IO register contains an approximation to $|x\rangle = A\;\,^{1}|b\rangle$.

Note that this solves the quantum linear systems problem exponentially faster than the best known classical algorithm. Like Shor's algorithm and QPE, HHL is a demonstration of potential quantum advantage.

However, note that only a quantum description of the solution vector is output from HHL. For applications that need a full classical description of **x**, this may not be satisfactory. *Quantum state tomography* — which is the measurement and characterization of the wavefunction of a quantum system — can be used to read out each amplitude of $|x\rangle$, but this takes time that scales exponentially in the number of qubits. Fortunately, there exists a number of applications where only certain features of the solution **x** need to be computed, for example the total weight of some subset of the indices.

Now that we have walked through the HHL algorithm, let us turn to an example implementation written in Cirq:

Example Implementation of the HHL Algorithm

In this implementation[7], we consider for simplicity a 2 × 2 system of linear equations. In particular, the matrix A we consider is

[7] Adapted from https://github.com/quantumlib/Cirq/blob/master/examples/hhl.py

$$A = \begin{bmatrix} 4.302134 - 6.015934 \cdot 10 \quad {}^{8}i & 0.235318 + 9.343861 \cdot 10 \quad {}^{1}i \\ 0.235318 - 9.343883 \cdot 10 \quad {}^{1}i & 0.583865 + 6.015934 \cdot 10 \quad {}^{8}i \end{bmatrix} \tag{9.55}$$

and we take the vector $|b\rangle$ as

$$|b\rangle = [0.64510 - 0.47848i \quad 0.35490 - 0.47848i]^T \tag{9.56}$$

Our goal is to use HHL to compute Pauli expectation values $\langle x|\sigma|x\rangle$ where $\sigma \in \{X, Y, Z\}$. We can easily compute these analytically (after classically solving the system) to be

$$\begin{aligned} \langle x|X|x\rangle &= 0.144130 \\ \langle x|Y|x\rangle &= 0.413217 \\ \langle x|Z|x\rangle &= -0.899154 \end{aligned}$$

We will compare the expectation values obtained by HHL to these expectation values.

In our program, we first import the packages we will use:

```
import math
import numpy as np
import cirq
```

and then build up the HHL circuit. Here, we define classes which are new gates in Cirq by inheriting from `cirq.Gate` or related objects. First, we create a gate representing $U_A = e^{iAt}$ which we will use in the QPE steps:

```

class HamiltonianSimulation(cirq.EigenGate, cirq.SingleQubitGate):
    """A gate that implements e^iAt.

    If a large matrix is used, the circuit should implement actual
        Hamiltonian
    simulation, for example by using the linear operators framework
        in Cirq.
    """

    def __init__(self, A, t, exponent=1.0):
        """Initializes a HamiltonianSimulation.

        Args:
            A : numpy.ndarray
                Hermitian matrix that defines the linear system Ax = b.

            t : float
                Simulation time. Hyperparameter of HHL.
        """
        cirq.SingleQubitGate.__init__(self)
        cirq.EigenGate.__init__(self, exponent=exponent)
```

```
      self.A = A
      self.t = t
      ws, vs = np.linalg.eigh(A)
      self.eigen_components = []
      for w, v in zip(ws, vs.T):
         theta = w*t / math.pi
         P = np.outer(v, np.conj(v))
         self.eigen_components.append((theta, P))

   def _with_exponent(self, exponent):
      return HamiltonianSimulation(self.A, self.t, exponent)

   def _eigen_components(self):
      return self.eigen_components
```

Next, we implement the series of controlled unitary operations in QPE, known as the phase kickback portion of the circuit (please refer to chapter 8 for a discussion of phase kickback):

```
class PhaseKickback(cirq.Gate):
   """A gate for the phase kickback stage of Quantum Phase
       Estimation.

   Consists of a series of controlled e^iAt gates with the memory
       qubit as
   the target and each register qubit as the control, raised
   to the power of 2 based on the qubit index.
   """

   def __init__(self, num_qubits, unitary):
      """Initializes a PhaseKickback gate.

      Args:
         num_qubits : int
            The number of qubits in the readout register + 1.

            Note: The last qubit stores the eigenvector; all other
                qubits
                 store the estimated phase, in big-endian.

         unitary : numpy.ndarray
            The unitary gate whose phases will be estimated.
      """
      super(PhaseKickback, self)
      self._num_qubits = num_qubits
      self.U = unitary

   def num_qubits(self):
      """Returns the number of qubits."""
      return self._num_qubits

   def _decompose_(self, qubits):
      """Generator for the phase kickback circuit."""
      qubits = list(qubits)
```

```
    memory = qubits.pop()
    for i, qubit in enumerate(qubits):
       yield cirq.ControlledGate(self.U**(2**i))(qubit, memory)
```

Next, we create a quantum Fourier transform gate, the third and final component of QPE:

```
class QFT(cirq.Gate):
   """Quantum gate for the Quantum Fourier Transformation.

   Note: Swaps are omitted here. These are implicitly done in the
   PhaseKickback gate by reversing the control qubit order.
   """

   def __init__(self, num_qubits):
      """Initializes a QFT circuit.

      Args:
         num_qubits : int
            Number of qubits.
      """
      super(QFT, self)
      self._num_qubits = num_qubits

   def num_qubits(self):
      return self._num_qubits

   def _decompose_(self, qubits):
      processed_qubits = []
      for q_head in qubits:
         for i, qubit in enumerate(processed_qubits):
            yield cirq.CZ(qubit, q_head)**(1/2.0**(i+1))
         yield cirq.H(q_head)
         processed_qubits.insert(0, q_head)
```

Now that we have the three major components of QPE, we can implement the entire algorithm. As above, we make the QPE instance a gate in Cirq:

```
class QPE(cirq.Gate):
   """A gate for Quantum Phase Estimation."""

   def __init__(self, num_qubits, unitary):
      """Initializes an HHL circuit.

      Args:
         num_qubits : int
            The number of qubits in the readout register.

            Note: The last qubit stores the eigenvector; all other
                qubits
                store the estimated phase, in big-endian.

         unitary : numpy.ndarray
            The unitary gate whose phases will be estimated.
   """
      super(QPE, self)
      self._num_qubits = num_qubits
      self.U = unitary

   def num_qubits(self):
      return self._num_qubits

   def _decompose_(self, qubits):
      qubits = list(qubits)
      yield cirq.H.on_each(*qubits[:-1])
      yield PhaseKickback(self.num_qubits(), self.U)(*qubits)
      yield QFT(self._num_qubits-1)(*qubits[:-1])**-1
```

With the unitary set as $U_A = e^{iAt}$, this instance of QPE will make up the first part of the HHL algorithm. The next step is to implement the controlled Pauli-Y rotation, which the following class does:

```
class EigenRotation(cirq.Gate):
   """EigenRotation performs the set of rotation on the ancilla qubit
   equivalent to division on the memory register by each eigenvalue
   of the matrix.

   The last qubit is the ancilla qubit; all remaining qubits are in
       the register,
   assumed to be big-endian.

   It consists of a controlled ancilla qubit rotation for each
       possible value
   that can be represented by the register. Each rotation is an Ry
       gate where
   the angle is calculated from the eigenvalue corresponding to the
       register
   value, up to a normalization factor C.
   """

   def __init__(self, num_qubits, C, t):
      """Initializes an EigenRotation.

      Args:
         num_qubits : int
            Number of qubits.

         C : float
            Hyperparameter of HHL algorithm.

         t : float
            Parameter.
      """
      super(EigenRotation, self)
      self._num_qubits = num_qubits
      self.C = C
      self.t = t
      self.N = 2**(num_qubits-1)
   def num_qubits(self):
      return self._num_qubits

   def _decompose_(self, qubits):
      for k in range(self.N):
         kGate = self._ancilla_rotation(k)

         # xor's 1 bits correspond to X gate positions.
         xor = k ^ (k-1)

         for q in qubits[-2::-1]:
            # Place X gates
            if xor % 2 == 1:
               yield cirq.X(q)
            xor >>= 1

            # Build controlled ancilla rotation
            kGate = cirq.ControlledGate(kGate)

         yield kGate(*qubits)

   def _ancilla_rotation(self, k):
      if k == 0:
         k = self.N
      theta = 2*math.asin(self.C * self.N * self.t / (2*math.pi * k))
      return cirq.Ry(theta)
```

Now that we have built up each component of the HHL algorithm, we can write a function to build the entire circuit, shown below:

```
def hhl_circuit(A, C, t, register_size, *input_prep_gates):
   """Constructs the HHL circuit and returns it.

   Args:
      A : numpy.ndarray
         Hermitian matrix that defines the system of equations Ax =
             b.

      C : float
         Hyperparameter for HHL.

      t : float
         Hyperparameter for HHL
   C and t are tunable parameters for the algorithm.
   register_size is the size of the eigenvalue register.
   input_prep_gates is a list of gates to be applied to |0> to
       generate the
    desired input state |b>.
   """

   # Ancilla register
   ancilla = cirq.GridQubit(0, 0)

   # Work register
   register = [cirq.GridQubit(i + 1, 0) for i in
       range(register_size)]

   # Input/output register
   memory = cirq.GridQubit(register_size + 1, 0)

   # Create a circuit
   circ = cirq.Circuit()

   # Unitary e^{iAt} for QPE
   unitary = HamiltonianSimulation(A, t)

   # QPE with the unitary e^{iAt}
   qpe = QPE(register_size + 1, unitary)

   # Add state preparation circuit for |b>
   circ.append([gate(memory) for gate in input_prep_gates])

   # Add the HHL algorithm to the circuit
   circ.append([
      qpe(*(register + [memory])),
      EigenRotation(register_size+1, C, t)(*(register+[ancilla])),
      qpe(*(register + [memory]))**-1,
      cirq.measure(ancilla)
   ])

   # Pauli observable display
   circ.append([
      cirq.pauli_string_expectation(
         cirq.PauliString({ancilla: cirq.Z}),
         key="a"
      ),
      cirq.pauli_string_expectation(
         cirq.PauliString({memory: cirq.X}),
         key="x"
      ),
      cirq.pauli_string_expectation(
         cirq.PauliString({memory: cirq.Y}),
         key="y"
      ),
      cirq.pauli_string_expectation(
         cirq.PauliString({memory: cirq.Z}),
         key="z"
      ),
   ])

   return circ
```

In this function, we define the three qubit registers for HHL and create an empty circuit. Then we form the unitary $U_A = e^{iAt}$ from the input matrix A for a given time t and form a QPE circuit using that unitary. The next line implements the state preparation circuit to prepare $|b\rangle$ from the ground state. Note that while $|b\rangle$ is assumed as input to HHL, in practice we always need a state preparation circuit.

After this, the HHL circuit is constructed step-by-step; first, we add the QPE circuit, then the controlled-Y rotation, then the inverse QPE circuit. Note that we are using the `**-1` notation in Cirq which makes it very easy to get the inverse of a quantum circuit. Finally, we append Pauli operators to make it easy to compute expectation values.

Now that we have built the circuit for HHL, we can simulate it to run the algorithm. The following function inputs an HHL circuit, simulates it, and prints out the expectation values from the input/output (IO) register after post-selecting the $|1\rangle$ outcome in the ancilla register.

Finally, we write our main function which defines the linear system A, input state $|b\rangle$ and any hyperparameters needed for the HHL algorithm:

```
def main():
    """Runs the main script of the file."""
    # Constants
    t = 0.358166 * math.pi
    register_size = 4

    # Define the linear system
    A = np.array([[4.30213466-6.01593490e-08j,
                0.23531802+9.34386156e-01j],
               [0.23531882-9.34388383e-01j,
                0.58386534+6.01593489e-08j]])

    # The |b> vector is defined by these gates on the zero state
    # |b> = (0.64510-0.47848j, 0.35490-0.47848j)
    input_prep_gates = [cirq.Rx(1.276359), cirq.Rz(1.276359)]

    # Expected expectation values
    expected = (0.144130 + 0j, 0.413217 + 0j, -0.899154 + 0j)

    # Set C to be the smallest eigenvalue that can be represented by
        the
    # circuit.
    C = 2*math.pi / (2**register_size * t)

    # Print the actual expectation values
    print("Expected observable outputs:")
    print("X =", expected[0])
    print("Y =", expected[1])
    print("Z =", expected[2])

    # Do the HHL algorithm and print the computed expectation values
    print("\nComputed: ")
    hhlcirc = hhl_circuit(A, C, t, register_size, *input_prep_gates)
    expectations(hhlcirc)

if __name__ == "__main__":
    main()
```

The output of this program is shown below:

```
Expected observable outputs:
X = (0.14413+0j)
Y = (0.413217+0j)
Z = (-0.899154+0j)

Computed:
X = (0.14413303+0j)
Y = (0.41321677+0j)
Z = (-0.89915407+0j)
```

As can be seen, HHL returns the correct (approximate) expectation values for each Pauli operator, indicating that the final state $|\bar{x}\rangle$ is indeed close to the solution vector $|x\rangle$.

9.7 *Quantum Random Number Generator*

Generating random numbers is crucial for many applications and algorithms, including Monte Carlo methods and cryptography. While classical computers generate *pseudo*random numbers, the random numbers generated by quantum computers are guaranteed to be truly random by the laws of quantum mechanics.

In this section, we consider a simple algorithm for generating truly random numbers on current quantum processors. The algorithm applies a Hadamard gate to a qubit in the ground state, then measures the state of the qubit in the computational basis. As we have seen, the Hadamard gate acting on $|0\rangle$ generates a superposition of computational basis states with equal amplitudes:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{9.57}$$

When this state is measured, there is therefore an equal probability of obtaining the ground and excited states. This can be exploited computationally as a random bit generator.

Let us now walk through an example program for generating random bits in Cirq. This program creates a circuit with one qubit, applies the Hadamard gate and then performs a measurement; the program then iterates the circuit ten times in the simulator.

```python
"""Program for generating random bits in Cirq."""

# Imports
import cirq
```

```
# Helper function for visualizing output
def bitstring(bits):
   return ''.join('1' if e else '0' for e in bits)

# Get a qubit and quantum circuit
qbit = cirq.LineQubit(0)
circ = cirq.Circuit()

# Add the Hadamard and measure operations to the circuit
circ.append([cirq.H(qbit), cirq.measure(qbit, key="z")])

# Simulate the circuit
sim = cirq.Simulator()
res = sim.run(circ, repetitions=10)

# Print the outcome
print("Bitstring =", bitstring(res.measurements["z"]))
```

An example output of this program is shown below:

```
Bitstring = 0011001011
```

Note that this output can be interpreted in several ways, depending on the context. One interpretation is a sequence of random bits, while another is a random bitstring representing, for example, an integer. In base ten, the integer produced in this example output is 203. In this sense, the program can be interpreted as a uniform random number generator in the integer interval $[0, N)$ where N is the number of repetitions of the circuit.

It is also possible to generate a random number in the range $[0, N)$ by using $n = \log_2 N$ qubits. Here, instead of simulating a circuit with one qubit many times, we apply a Hadamard gate on each of the n qubits, then measure. Since there are $2^n = N$ possible bitstrings for n qubits, this will also generate a random bitstring that can be interpreted as an integer in the range $[0, N)$. A program that implements this in Cirq is shown below:

```
"""Program for generating random numbers in Cirq."""

# Imports
import cirq

# Number of qubits
n = 10

# Helper function for visualizing output
def bitstring(bits):
   return ''.join('1' if e else '0' for e in bits)

# Get a qubit and quantum circuit
qreg = [cirq.LineQubit(x) for x in range(n)]
circ = cirq.Circuit()
```

```
# Add the Hadamard and measure operations to the circuit
for x in range(n):
   circ.append([cirq.H(qreg[x]), cirq.measure(qreg[x])])

# Simulate the circuit
sim = cirq.Simulator()
res = sim.run(circ, repetitions=1)

# Print the measured bitstring
bits = bitstring(res.measurements.values())
print("Bitstring =", bits)

# Print the integer corresponding to the bitstring
print("Integer =", int(bits, 2))
```

Here, we use $n = 10$ qubits to generate a random number in the range $[0, 1024)$. An example output of this program is shown below:

```
Bitstring = 1010011100
Integer = 668
```

Here, the integer is the bitstring in base 10. This program produces uniform random numbers in the range [0, 1024) and will produce different integers if run successive times.

9.8 Quantum Walks

Quantum walks have been shown to have computational advantages over classical random walks [11, 81, 9, 116, 56, 57].

In the classical setting, a particle starts out in some initial position (vertex) on a graph $G = (V, E)$ and "walks" to neighboring vertices in a probabilistic manner; these are classical random walks. The final probability distribution of finding the particle at a given vertex V — as well as questions like "how long does it take the particle to reach a particular vertex?" — are interesting and useful quantities to calculate. When certain problems are phrased in terms of random walks, for example the 2-SAT problem, computing these quantities can lead to novel solutions that may not have been known previously.

The simplest example of a classical random walk is a one-dimensional walk on a line. Consider a particle starting at position $x(t = 0) = 0$ at time $t = 0$. At time $t = 1$, the particle moves to the right $(x(1) = 1)$ or left $(x(t = 1) = -1)$ with equal probability. At time $t = 1$, we have $x(2) = x(1) \pm 1$ with equal probability, and in general $x(t) = x(t - 1) \pm 1$ with equal probability. Because steps are made at only discrete increments of time $t = 1, 2, 3, \ldots$, this is known as a *discrete-time* random walk. *Continuous-time* random walks are another model which we discuss below.

Classical random walks are implemented by generating pseudo-random number(s) at each time step. The particle's position is updated at each iteration according to the outcome of the random number generator. Alternatively, an array of values representing probabilities for each position can be stored and updated via a stochastic matrix which determines the time evolution of the system.

In the case of a discrete-time *quantum* walk on a line, the idea is similar but the implementation is different. The quantum walk consists of two registers of qubits: a *position register* P and a *coin register* C. As the name suggests, the position register tracks the probability distribution for the particle to be at a particular position $|0\rangle$, $|1\rangle$, ..., $|N-1\rangle$, where we impose periodic boundary conditions $|N\rangle = |0\rangle$. The coin register is used to update the particle's position at each time step.

The update to the particle's position is given by the *shift operator*

$$S := |0\rangle\langle 0|_C \otimes \sum_i |i-1\rangle\langle i|_P + |1\rangle\langle 1|_C \otimes \sum_i |i+1\rangle\langle i|_P \tag{9.58}$$

That is, if the coin register is in the $|1\rangle$ state, the particle shifts to the left, and if the coin register is in the $|0\rangle$ state, the particle shifts to the right. That is,

$$S|0\rangle_C \otimes |i\rangle_P = |0\rangle_C \otimes |i-1\rangle_P \tag{9.59}$$

and

$$S|1\rangle_C \otimes |i\rangle_P = |1\rangle_C \otimes |i+1\rangle_P \tag{9.60}$$

The coin is "flipped" by applying a single qubit gate — for example the Hadamard gate H to produce an equal superposition state, though "biased" coins can also be used — and then the shift operator is applied. One step of the quantum walk can thus be written

$$U = S(H_C \otimes I_P) \tag{9.61}$$

where H_C is the Hadamard acting on the coin and I_P refers to the identity acting on the particle. T steps of the walk are given by U^T.

In this simplest example of a random walk, numerous differences between the classical and quantum cases can already be seen. For example, starting in the initial state $|0\rangle_C \otimes |0\rangle_P$ will cause the probability distribution to drift "to the left" — that is, the particle is more likely to move to the left — whereas in the classical case the distribution is symmetric. Starting the quantum walk in the state $|1\rangle_C \otimes |0\rangle_P$ will cause the distribution to drift to the right. The reasons for this are constructive and destructive interference of amplitudes, a distinctly quantum phenomena that is not possible in the classical case. Note that the distribution for quantum walk can be made symmetric — by starting in the state $|+\rangle_C \otimes |0\rangle_P$, for example, where $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$.

This simple example is instructive for understanding both how quantum walks work and how quantum walks are different that classical walks. Upon further study, more differences can be seen. For example, the variance of the classical distribution for a discrete-time random walk on a line is $\sigma_c^2 = T$ after T time steps, but in the quantum case it is $\sigma_q^2 = T^2$ [117]. Thus, the quantum walker propagates quadratically faster than the classical one.

For a review of quantum walks, see [117] and [185] and the references therein. For an example of quantum walks applied to graphs, see [9]. Farhi et al. demonstrated speedup for NAND trees with quantum walks [77].

Let us now turn to an example implementation of a quantum walk to get more experience.

Implementation of a Quantum Walk

In this section we provide an example implementation of a continuous time quantum walk (CTQW). We first discuss the transition from discrete to continuous *classical* walk, as this will reveal how we perform a continuous *quantum* walk. In a discrete-time classical walk, probability distributions are stored in a vector **p** which is updated via a stochastic matrix

$$\mathbf{p}(t+1) = M\mathbf{p}(t) \tag{9.62}$$

This operates only at discrete times. To make it continuous, we rewrite this equation as a differential equation

$$\frac{d\mathbf{p}(t)}{dt} = -H\mathbf{p}(\mathbf{t}) \tag{9.63}$$

where H is a time-independent matrix with elements given by

$$\langle i|H|j\rangle = \begin{cases} -\gamma & i \neq j \text{ and } (i,j) \in E \\ 0 & i = j \text{ and } (i,j) \notin E \\ d_i\gamma & i = j \end{cases} \tag{9.64}$$

Here, γ is the constant transition rate from vertex i to vertex j and d_i is the degree (i.e., number of edges) of vertex i.

The solution for this differential equation is known to be $\mathbf{p}(t) = e^{\;Ht}\mathbf{p}(0)$. The step to making this a continuous time *quantum* walk is to treat the matrix H as a Hamiltonian which generates the unitary evolution

$$U(t) = e^{\;iHt} \tag{9.65}$$

which is defined for a continuous, not discrete, spectrum of times t.

Let us now turn to the example implementation using pyQuil[8]. Here we perform a continuous time quantum walk on a *complete* graph with four vertices (nodes), commonly denoted K_4. A complete graph is one in which each vertex is connected to all vertices. We first import the packages we will use for this implementation. We highlight here the use of `networkx`, a common Python package for working with graphs.

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from scipy.linalg import expm

import pyquil.quil as pq
import pyquil.api as api
from pyquil.gates import H, X, CPHASE00
```

We now create a complete graph on four nodes and visualize it:

```
# Create a graph
G = nx.complete_graph(4)
nx.draw_networkx(G)
```

The output of this portion of the program is shown in Figure 9.6. Note that each vertex has an edge connecting it to all other vertices.

The spectrum of a complete graph (i.e., the eigenvalues of the adjacency matrix of a complete graph) is quite simple. It is known from graph theory that one eigenvalue is equal to $N - 1$ (where N is the number of nodes) and the remaining eigenvalues are equal to -1. In the following code block, we get the adjacency matrix of the K_4 graph and diagonalize it to verify the spectrum is what we expect.

```
# Diagonalize the adjacency matrix
A = nx.adjacency_matrix(G).toarray()
eigvals, _ = np.linalg.eigh(A)
print("Eigenvalues =", eigvals)
```

[8] This implementation is adapted from open-source code which can be found at https://github.com/rigetti/pyquil/blob/master/examples/quantum_walk.ipynb.

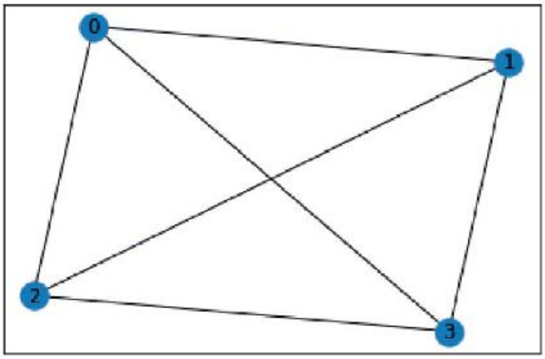


Figure 9.6: A complete graph on four vertices which we implement a continuous time quantum walk on. This graph is denoted K_4.

The output of this code block, shown below, verifies our prediction of the spectrum:

```
Eigenvalues = [-1. -1. -1.  3.]
```

For the CTQW, the usual Hamiltonian is the adjacency matrix A. We modify it slightly by adding the identity, i.e., we take $\mathcal{H} = A + I$. This will reduce the number of gates we need to apply, since the eigenvectors with 0 eigenvalue will not acquire a phase. The code below defines our Hamiltonian:

```
# Get the Hamiltonian
ham = A + np.eye(4)
```

It turns out that complete graphs are Hadamard diagonalizable. This means that we can write

$$H = Q\Lambda Q^\dagger \tag{9.66}$$

where $Q = H \otimes H$ and Λ is the diagonal matrix of eigenvalues. Let's check that this works.

```
# Hadamard gate
hgate = np.sqrt(1/2) * np.array([[1, 1], [1, -1]])

# Form the matrix Q = H \otimes H to diagonalize the Hamiltonian
Q = np.kron(hgate, hgate)

# Print out the Q^\dagger H Q to verify it's diagonal
diag = Q.conj().T.dot(ham).dot(Q)
print(diag)
```

The output of this portion of the program, shown below, verifies that $Q^\dagger HQ$ is indeed diagonal (note that numbers in the first row are numerically zero):

```
[[ 4.00000000e+00 -4.93038066e-32 -4.93038066e-32 4.93038066e-32]
 [ 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

The time evolution operator $U(t) = e^{\ iHt}$ is also diagonalized by the same transformation. In particular, we have

$$Q^\dagger e^{\ iHt} Q = \begin{pmatrix} e^{\ i4t} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9.67}$$

which is exactly a `CPHASE00` gate in pyQuil with an angle of $-4t$. To elaborate on this further, the `CPHASE00`, which we'll denote as $CZ_{00}(\varphi)$ below, has the following action on the computational basis:

$$\begin{aligned} CZ_{00}(\varphi)|00\rangle &= e^{i\varphi}|00\rangle \\ CZ_{00}(\varphi)|01\rangle &= |01\rangle \\ CZ_{00}(\varphi)|10\rangle &= |10\rangle \\ CZ_{00}(\varphi)|11\rangle &= |11\rangle \end{aligned}$$

The circuit to simulate the unitary evolution $U(t) = e^{\ iHt}$ thus consists of a Hadamard gate on each qubit, $CZ_{00}(-4t)$, and then another Hadamard gate on each qubit. The code snippet below defines a function for creating this quantum circuit:

```
# Function for a the continuous time quantum walk circuit on a
    complete graph
def k_4_ctqw(t):
   """Returns a program implementing a continuous time quantum
       walk."""
   prog = pq.Program()

   # Change to diagonal basis
   prog.inst(H(0))
   prog.inst(H(1))

   # Time evolve
   prog.inst(CPHASE00(-4*t, 0, 1))

   # Change back to computational basis
   prog.inst(H(0))
   prog.inst(H(1))
```

```
    return prog
```

Let's compare the quantum walk with a classical random walk. The classical time evolution operator is $e^{(M \quad I)t}$ where M is the stochastic transition matrix of the graph. We obtain M from the adjacency matrix of the graph below:

```
# Stochastic transition matrix for classical walk
M = A / np.sum(A, axis=0)
```

We choose as our initial condition $|\psi(0)\rangle = |0\rangle$, so that the walker starts on the first node. Therefore, due to symmetry, the probability of occupying each of the nodes besides $|0\rangle$ is the same. In the code below, we define the final times to simulate the random walks for and create arrays to store the probability distributions at each final time:

```
# Set up time to simulate for
tmax = 4
steps = 40
time = np.linspace(0, tmax, steps)

# Arrays to hold quantum probabilities and classical probabilities
    at each time
quantum_probs = np.zeros((steps, tmax))
classical_probs = np.zeros((steps, tmax))
```

We can now simulate the continuous-time quantum and classical walks for each final time we have chosen. The code block below performs this simulation and stores the final probability distributions:

```
# Do the classical and quantum continuous-time walks
for i, t in enumerate(time):
    # Get a quantum program
    prog = k_4_ctqw(t)

    # Simulate the circuit and store the probabilities
    wvf = qvm.wavefunction(prog)
    vec = wvf.amplitudes
    quantum_probs[i] = np.abs(vec)**2

    # Do the classical continuous time walk
    classical_ev = expm((M-np.eye(4))*t)
    classical_probs[i] = classical_ev[:, 0]
```

Finally, the code below plots the probabilities for each node at all times:

```
_, (ax1, ax2) = plt.subplots(2, sharex=True, sharey=True)

ax1.set_title("Quantum evolution")
ax1.set_ylabel("Probability")
```

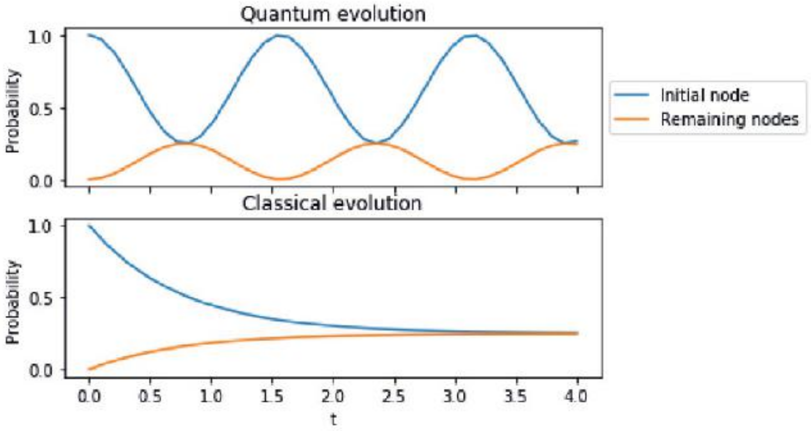


Figure 9.7: Time evolution of the continuous-time quantum and classical walks on a complete graph with four vertices.

```
ax1.plot(time, quantum_probs[:, 0], label='Initial node')
ax1.plot(time, quantum_probs[:, 1], label='Remaining nodes')
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))

ax2.set_title("Classical evolution")
ax2.set_xlabel('t')
ax2.set_ylabel("Probability")
ax2.plot(time, classical_probs[:, 0], label='Initial node')
ax2.plot(time, classical_probs[:, 1], label='Remaining nodes')
```

The output of this code block is shown in Figure 9.7. Here, we see another clear difference between the quantum and classical walks. Namely, in the classical cases, the probability for being in the initial node decays exponentially, whereas in the quantum case it oscillates! This is what we should expect based on our construction of the Hamiltonians for each case — namely, the quantum cases has an i in the exponent e^{iHt} which produces oscillatory behavior, while in the classical case the exponent is real which produces purely exponential decay.

9.9 Summary

In this chapter, we have covered a range of quantum computing methods. We have seen that a QC can be used for optimization, chemical simulation, true random number generation and other techniques. For more algorithms please see and contribute to the Quantum Algorithm Zoo [114]. In the coming chapter we will turn to quantum supremacy, quantum error correction and the road ahead for quantum computing.

As the quantum community eagerly seizes the impending opportunity to experiment with NISQ devices, we must not lose sight of the essential longer-term goal: hastening the onset of the fault-tolerant era.

—John Preskill

CHAPTER 10

Applications and Quantum Supremacy

In this work, we have taken a journey through the quantum computing landscape; we have explored its theoretical foundation, discussed the key research and milestones that advanced the field and covered a range of hardware approaches and quantum computing methods.

On the engineering front, there are still daunting challenges ahead of us to scale to more than 10^6 qubits. Once we achieve fault-tolerant quantum computing, more possibilities open up for applications. In the current NISQ regime, there is plenty of work to be done to explore test cases and prepare for the error-corrected machines.

10.1 Applications

As the quantum computing landscape evolves, a number of QC applications are becoming clear. Check the online site and see [155] and [176] for updates on QC applications.

Quantum Simulation and Chemistry

High-performance classical computers are used today to model new molecular combinations. This work helps researchers develop new materials, novel pharmaceuticals as well as compounds for other applications. Quantum computers will likely give us new capabilities in this domain. Already, the VQE and quantum chemistry simulation methods discussed in chapter 9 have shown promising results. See the following for additional examples: [184, 231, 164].

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_10

Sampling from Probability Distributions

We use distribution sampling in many applications such as pattern recognition and probabilistic inference. With a quantum computer, we can sample from a much larger distribution. This is one reason that distribution sampling is being used to demonstrate quantum supremacy as we describe later in this chapter.

Linear Algebra Speedup with Quantum Computers

There are many application of linear algebra in industry. Matrix inversion, is a common technique that can be used, for example, in computing electromagnetic patterns to design an antenna [176]. The HHL technique which we covered in chapter 9 is one method that may prove valuable for these applications.

Optimization

There are numerous optimization applications in industry, including: delivery truck routing, online ad bidding strategies and mixtures of different chemicals for electric vehicle battery composition. It is becoming clear that quantum computers can be used to optimize these kinds of systems.

Tensor Networks

One promising area of inquiry is the application of quantum computation to tensor networks (TNs). This book's online site contains references to a number of good introductions to TNs.

Various tensor network architectures such as MERA, MPS, TTNs and PEPS are proving to be useful tools to explore questions in physics as well as in other fields such as deep learning networks. Several groups have demonstrated a range of applications of tensor networks [223, 233, 194, 224, 108, 152].

10.2 *Quantum Supremacy*

The term *quantum supremacy*, first coined by Preskill in 2012, refers to a computational task that can be efficiently performed on a quantum computer beyond the capabilities of state-of-the-art classical supercomputer can efficiently implement [175]. We note immediately — as the strong term *supremacy* can generate confusion — that this refers to *any* computational task which meets the criteria, not necessarily a task that is useful.

The algorithm used to demonstrate supremacy does not need to have wide application, just a clear-cut ability to run efficiently on a quantum processor compared with a classical computer where the algorithm is intractable. [36].

In this section, we discuss problems which researchers are considering to demonstrate quantum supremacy. Regardless of the particular problem used for the demonstration, quantum supremacy is a landmark achievement in the history of physics and computer science. While many proof-of-principle quantum computations have been performed on quantum processors, this will be the first to be executed at a large enough scale to reveal an experimentally verifiable computational separation.

This has implications for verifying quantum mechanics through large-scale computation. Indeed, we can think of supremacy experiments as the computational analogue of Bell experiments [104]. Just as Bell experiments refute local hidden variable models, supremacy experiments on an error-corrected QC would refute the Extended Church-Turing Thesis (ECTT) which, as discussed in chapter 4, asserts that any algorithmic process can be simulated efficiently using a probabilistic Turing machine PTM. The ability to carefully control quantum systems of this magnitude is also a crowning achievement of engineering and experimental physics.

In the remainder of the section, we discuss the major problems which are being considered for demonstrating quantum supremacy.

Random Circuit Sampling

Sampling from the output distribution of a quantum circuit is one of the most natural problems to demonstrate quantum supremacy. To simulate this on a classical computer, one must perform the linear algebra and matrix computation to determine the final state of the wavefunction after the execution of the quantum circuit (written as tensor products of unitary operators). However, a quantum computer naturally performs this calculation by simply evolving in time under the physical realizations of the unitary operators.

Classical methods for simulating quantum circuits generally scale exponentially in the number of qubits. Specifically, for the most general, fully entangled, state of n qubits, there are 2^n complex amplitudes to keep track of in the wavefunction. Even for moderate values of n, this quickly reaches current memory limitations on even the most capable supercomputers.

Name	Number of bytes	Number of qubits
Kilobyte (KB)	$2^{10} \approx 10^3$	$n = 6$
Megabyte (MB)	$2^{20} \approx 10^6$	$n = 16$
Gigabyte (GB)	$2^{30} \approx 10^9$	$n = 26$
Terabyte (TB)	$2^{40} \approx 10^{12}$	$n = 36$
Petabyte (PB)	$2^{50} \approx 10^{15}$	$n = 46$
Exabyte (EB)	$2^{60} \approx 10^{18}$	$n = 56$
Zettabyte (ZB)	$2^{70} \approx 10^{21}$	$n = 66$

Table 10.1: Table of prefixes and the number of bytes they correspond to; the last column shows the maximum number of qubits that can be stored for the given memory, assuming the most general state of a qubit with amplitudes stored in double precision. One byte is 8 bits.

Each amplitude in the wavefunction is generally a complex number, which entails storing two real floating-point numbers per amplitude. Suppose that these floating-point numbers are stored in double precision format, i.e., 8 bytes per floating-point number. Under these assumptions, the total memory required to store the wavefunction is

$$2^n \text{ amplitudes} \times 2 \text{ real numbers/amplitude} \times 2^3 \text{ bytes/real number}$$

i.e., 2^{n+4} bytes. Recall that one kilobyte is defined as 2^{10} bytes, one megabyte is 2^{20} bytes, and so on; see Table 10.1.

The leading supercomputers have RAM sizes of petabytes to exabytes.[1] Based on our previous argument for memory requirements of storing wavefunctions, we can estimate the upper range of quantum circuit simulation for any particular classical system.

This is the fundamental idea of using quantum circuit sampling as a candidate problem for demonstrating quantum supremacy [43, 156]. There are now multiple methods for simulating quantum circuits — ranging from explicit construction of the unitary for the circuit to tensor network contractions — but all of them suffer from exponential complexity in the number of qubits.

Let us now consider in more detail random circuit sampling as a demonstration problem of quantum supremacy, following the work of Boixo et al. [160, 36, 142]. Here researchers are considering the problem of sampling from the output distribution of random quantum circuits.

The particular random circuits considered for supremacy experiments are constructed via the following rules [36]:[2]

[1] See https://www.top500.org/ for an up-to-date list of supercomputers and their specs.

[2] Note that we have incorporated Boixo's updated instructions from his GitHub site: https://github.com/sboixo/GRCS

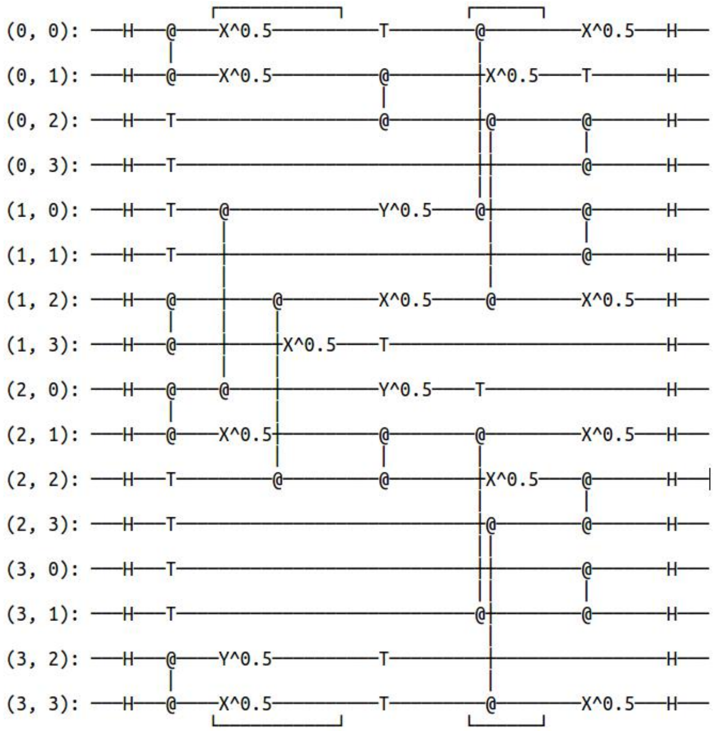
1. Start with a Hadamard gate on each qubit.
2. Apply controlled-Z (*CZ*) operators between neighboring qubits in a two-dimensional grid alternating between horizontal and vertical patterns. Note that in any particular cycle, not *all* neighboring qubits will be connected via a *CZ*, and the number of *CZ* gates can be different in different cycles.[3]
3. Apply single-qubit operators from the set $\{X^{1/2}, Y^{1/2}, T\}$ to the qubits which are not affected by *CZ* gates according to the following criteria:
   - If the previous cycle had a *CZ* gate on a given qubit, apply a randomly-chosen non-diagonal unary gate to that qubit if possible.[4]
   - If the previous cycle had a non-diagonal unary gate on a given qubit, apply a T gate to that qubit if possible.
   - Apply a T gate to a qubit if there are no unary gates in the previous cycles on that qubit (except for the initial Hadamard gate). Note that this rule is an *if* and not *iff*. That is, a T gate may follow another unary gate; the rule simply states that if there is *no* unary gate on that qubit in the previous cycles, then we must place a T gate in the current cycle. The previous two criteria take precedence over this one.
   - If none of the above criteria are satisfied for a given qubit, then a unary gate is not applied to that qubit for the current cycle.
4. Repeat steps (2) and (3) for a given number of cycles (which determines the depth).
5. Measure in the computational or Hadamard (X) basis.

We can now incorporate these rules into a program for building supremacy circuits. An example program in Cirq demonstrating this functionality is provided below.

```
import cirq

# Number of rows in grid of qubits
nrows = 4

# Number of columns in grid of qubits
ncols = 4
```

[3] Note that the term cycle in this context refers to moments in the framework of Cirq. We can think of cycles or moments as the set of operators that are applied simultaneously.

[4] “If possible" means if the circuit has not ended or if there is not a *CZ* gate on that qubit in the current cycle.



```
                   ┌───────────┐           ┌──────┐
(0, 0): ───H───@────X^0.5──────────T────────@─────────X^0.5───H───
               │                            │
(0, 1): ───H───@────X^0.5──────────@────────┼X^0.5────T───────H───
                                   │        │
(0, 2): ───H───T───────────────────@────────┼@────────@───────H───
                                            ││        │
(0, 3): ───H───T────────────────────────────┼┼────────@───────H───
                                            ││
(1, 0): ───H───T────@──────────────Y^0.5────@┼────────@───────H───
                    │                        │        │
(1, 1): ───H───T────┼────────────────────────┼────────@───────H───
                    │                        │
(1, 2): ───H───@────┼────@─────────X^0.5─────@────────X^0.5───H───
               │    │    │
(1, 3): ───H───@────┼────┼X^0.5────T──────────────────────────H───
                    │    │
(2, 0): ───H───@────@────┼─────────Y^0.5────T─────────────────H───
               │         │
(2, 1): ───H───@────X^0.5┼─────────@────────@─────────X^0.5───H───
                         │         │        │
(2, 2): ───H───T─────────@─────────@────────┼X^0.5────@───────H──┤
                                            │         │
(2, 3): ───H───T────────────────────────────┼@────────@───────H───
                                            ││
(3, 0): ───H───T────────────────────────────┼┼────────@───────H───
                                            ││        │
(3, 1): ───H───T────────────────────────────@┼────────@───────H───
                                             │
(3, 2): ───H───@────Y^0.5──────────T─────────┼────────────────H───
               │                             │
(3, 3): ───H───@────X^0.5──────────T─────────@────────X^0.5───H───
                   └───────────┘           └──────┘
```

Figure 10.1: Quantum supremacy circuit on a grid of qubits generated by Cirq. The final cycle of Hadamard gates is for measuring in the X basis, but Z basis measurements may be used as well.

```
# Depth of CZ gates in supremacy circuit
depth = 5

# Generate the supremacy circuit
supremacy_circuit =
    cirq.experiments.generate_supremacy_circuit_google_v2_grid(
   nrows, ncols, depth, seed=123)

print(supremacy_circuit)
```

Here, the number of rows and number of columns are specified for the two-dimensional grid of qubits; the depth, or number of cycles of *CZ* gates, is specified to determine the overall depth of the supremacy circuit. The output of this program is shown in Figure 10.1. Note that in any particular version or implementation of a random circuit sampler, the code module may be following rules that are slightly modified from those stated above.

This circuit of $n = 16$ qubits is easily handled by classical computers, but, as we have argued, the difficulty of classical simulation scales exponentially in n. For a sufficiently large number of qubits n, we outline the steps toward demonstrating quantum supremacy [36]:

1. Generate a supremacy circuit U on n qubits and a given depth d as per above.
2. Sample from the circuit m times with $m \approx 10^3 - 10^6$ to get an output distribution $\{x_1, ..., x_m\}$.
3. Compute $\log 1/p_U(x_j)$ for each $j = 1, ..., m$ with a sufficiently powerful classical computer. Here,

$$p_U(x_j) := |\langle x_j|\psi\rangle|^2 \tag{10.1}$$

   where $|\psi\rangle = U|0\rangle$ is the final state of the supremacy circuit.
4. Compute the quantity

$$\alpha = H_0 - \frac{1}{m}\sum_1^m \log \frac{1}{p_U(x_j)} \tag{10.2}$$

   where $H_0 = \log(2^n) + \gamma$ is the cross-entropy of an algorithm which samples from bit strings uniformly. (Note that the logarithm is the natural logarithm here.) Here, $\gamma \approx 0.577$ is Euler's constant.

Once the quantity α is computed, it is then compared to a similar quantity evaluated on the output distribution p_A of the best classical algorithm A for simulating quantum circuits. Note that the cross-entropy difference, which gives a measure of how well the algorithm A can predict the outcome of a typical random circuit U, is given by

$$\Delta H(p_A) = H_0 - H(p_A, p_U) \tag{10.3}$$

Now consider the expectation value of $\Delta H(p_A)$ over an ensemble of random circuits R and let C hold this value:

$$C := \mathbb{E}_R[\Delta H(p_A)] \tag{10.4}$$

In the Boixo et al. supremacy paper [36], it is shown that quantum advantage is achieved in practice when

$$C \leq \alpha \leq 1 \tag{10.5}$$

Note that $C \to 0$ for large enough circuits, and further that $p_U(x_j)$ can no longer be obtained numerically. This implies by definition that the quantity α can no longer be measured directly. However, it is possible to extrapolate α for larger circuits in order to demonstrate quantum supremacy with random circuit sampling.

Other Problems for Demonstrating Quantum Supremacy

While random circuit sampling is a very natural problem to consider for demonstrating quantum supremacy, it is not the only one. A recent survey paper by Harrow and Montanaro [104] provides a helpful discussion of additional major problems being considered to show quantum-classical separation.

The problem of boson sampling is another candidate for demonstrating quantum supremacy. Originally proposed in [5], boson sampling involves sending n coincident photons into a randomly generated linear-optical network of $m \gg n$ *modes* (beam splitters); this generates a random unitary rotation. Detectors are then used to sample from the distribution of photons, a process which is believed to be classically hard. Boson sampling experiments have been performed with up to five photons and nine modes [209]. Experimental systems are challenged by non-trivial photon loss in the optical network. Additionally, developing more efficient classical sampling techniques is a challenge for quantum supremacy via boson sampling.[5]

Quantum Advantage

Researchers have coined several terms related to the distinction between classical and quantum computing, including quantum advantage and quantum-classical separation. Quantum advantage can refer to a constant or linear speedup compared with classical computing. See W. Zeng's article on terms and measures of quantum-classical computing distinctions [242].

10.3 *Future Directions*

Quantum Error Correction

While today's quantum computers do not yet have sufficient qubits to support full quantum error correction (QEC), there is a growing body of research on QEC with implications for both QC and beyond. Classical computation admits straightforward error correction through the replication of a state across many classical bits. The no-cloning theorem in quantum mechanics, however, prevents us from taking this direct approach in a quantum computer.

[5] This is true for any problem, P, used for quantum supremacy. Namely, if better classical algorithms are developed for P, the threshold for quantum supremacy with P gets pushed back further.

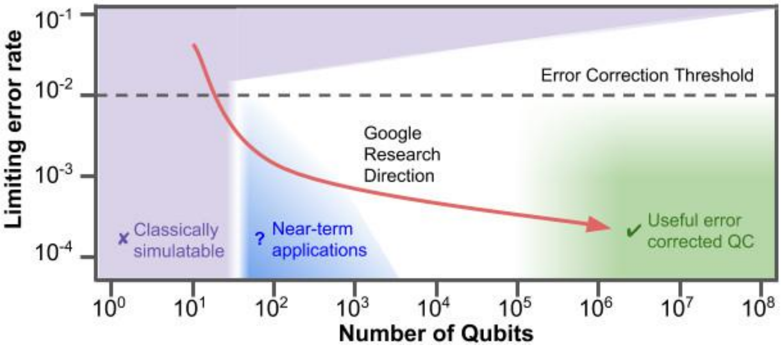


Figure 10.2: Quantum computing roadmap Source: Google

A typical approach to QEC involves a *surface code* which encodes one logical qubit into a topological state of several physical qubits [49, 63, 87]. When we measure these physical qubits we can see a pattern called a *syndrome* which is the result of a particular sequence of errors; a *decoder* can then map the syndrome to a particular error sequence. Such decoding may be amenable to the use of machine learning (see, for example, [20]).

As discussed in the section on VQE, McClean et al. have explored the use of subspace expansion for error mitigation [147]. See the work of Ofek et al. for a discussion of the break-event point of QEC [163]).

Error correction schemes have also emerged from other branches of physics; several researchers have been investigating QEC approaches that derive from the duality framework of Anti-de Sitter/Conformal Field Theory (AdS/CFT) [13]. QEC remains an active area of reseach and is critical in the scaling quantum computing hardware devices.

Doing Physics with Quantum Computers

As we mentioned in the preface of this book, one of the most interesting potential uses of quantum computers is to probe open questions in physics. The duality framework of AdS/CFT gives us an initial mapping between general relativity and quantum mechanics. Susskind and others have speculated on the use of quantum computers to explore this duality [213]. While we are years away from building quantum computers of sufficient scale and fault-tolerance to run such experiments, it is still useful to consider what we might learn from such explorations.

The key principle here is that in quantum computing we are not merely modeling a superposition or entangled state and pointing to it from a classical computer; we are in fact implementing these states, and as such can ask questions about their dynamics.

Conclusion

We anticipate a fast pace of development in this field — both in hardware and software — and predict that many more universities and companies will explore how these platforms can impact their work. We invite the reader to access the book's online companion site

http://www.github.com/jackhidary/quantumcomputingbook

for more resources and updates as the field progresses. As we drive to a fully error-corrected quantum computer, it will certainly be an interesting journey.

Part III

Toolkit

"To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty, of nature ... If you want to learn about nature, to appreciate nature, it is necessary to understand the language that she speaks in."
—Richard Feynman

CHAPTER 11

Mathematical Tools for Quantum Computing I

11.1 *Introduction and Self-Test*

One of the most important discoveries of quantum mechanics in the twentieth century was the observation by John von Neumann in his *Mathematical Foundations of Quantum Mechanics* that all of quantum mechanics can be described by linear algebra [225].

Confident readers may feel that they need not read this chapter and might instead flip through it for formulas, equations and the like. We provide readers with the following provocative questions whose answers we will provide directly afterward.

11.1 **Exercise** ***Self-Test***

1. Is the function

$$T : \mathbb{R} \to \mathbb{R}$$

$$T(x) := x + 1$$

a linear transformation?

2. Does a *binary* operation have anything to do with *binary* code?
3. Which space has a bigger dimension: $\mathbb{R}^4$ or $\mathbb{C}^2$?
4. Of these expressions:

   a. $\langle 0|1\rangle$

   b. $|0\rangle\langle 1|$

   c. $\langle 0|1\rangle\,|0\rangle$

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_11

d. $\langle i|A|j\rangle$ where A is a matrix and i and j are numbers

Which is a number? A vector? A matrix?

5. Give an example of a Hermitian operator whose eigenvalues are not real numbers.

The answers:

1. No, it's hopelessly not linear.
2. No, a binary operation is a special type of function.
3. $\mathbb{R}^4$ and $\mathbb{C}^2$ have the same dimension over $\mathbb{R}$.
4. Here is the classification of the four expressions in Dirac notation:

a. The expression $\langle 0|1\rangle$ is a number. In fact, $\langle 0|1\rangle = 0$.

b. The expression $|0\rangle\langle 1|$ is a matrix, specifically, the matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

c. The expression $\langle 0|1\rangle\,|0\rangle$ is a vector. You can see that $\langle 0|1\rangle$ is the number 0 and we mentioned earlier in the book that $|0\rangle$ is the vector

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

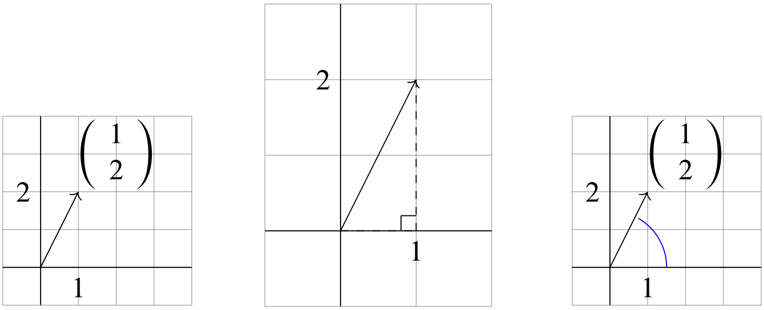
so we can write

$$\langle 0|1\rangle\,|0\rangle = (\langle 0|1\rangle)\,|0\rangle = (0)\,|0\rangle = (0)\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$= \begin{pmatrix} 0\cdot 1 \\ 0\cdot 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

d. This is a number. In particular, it's a clever way to write the entry in the ith row and jth column of the matrix A.

5. There is no such thing! Hermitian operators always have real eigenvalues; try to prove this statement – we will in this section.

Linear algebra can become very complicated. We don't want to discourage you, rather the opposite! We would like to be your ambassadors on a journey into linear algebra and the supporting abstract mathematics that underlies quantum computing. We will develop all of the prerequisite mathematics and offer several examples from the book to relate the tools of linear algebra to quantum computing. That being said, feel free to begin wherever it is that you feel comfortable.


$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$
2
1
2
1
$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$
2
1

(a) Vector in the plane *(b) Length of a vector* *(c) Angle subtended by a vector*

Figure 11.1

11.2 *Linear Algebra*

Vectors and Notation

The vector is one of the central objects of linear algebra. There are several ways to conceptualize a vector. First, we may think of a vector as an ordered collection of numbers (a 1-dimensional array). For example, the following vector is the ordered collection of the numbers 1 and 2:

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \tag{11.2}$$

A vector can be thought of as a geometric object as well. For example, we may plot the vector $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ in the 2-dimensional plane, as in Figure 11.1a.

Thinking geometrically, a vector can have a magnitude (length) and a direction. For example, by the Pythagorean Theorem, the vector in Figure 11.1a has length

$$\sqrt{1^2 + 2^2} \tag{11.3}$$

as indicated by the depiction in Figure 11.1b. We can describe the direction by giving the angle subtended by the arc from the x-axis to the head of the arrow, as in Figure 11.1c.

We often denote a vector with a lowercase bold letter, like $\mathbf{v}$, or, when writing by hand, with an arrow, like $\vec{v}$. We'll use the simple notation v to denote a vector when context makes it clear that v is a vector.

Sometimes, we write a vector more explicitly, like so:

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{11.4}$$

to indicate the number of entries in the vector. So, this vector has n entries. Some people like to denote vectors with square brackets, as below

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \tag{11.5}$$

but it doesn't matter which you use. We prefer round brackets in this discussion.

You may have already encountered the *qubit*, the quantum analog of the classical bit, earlier in the book. We use the example earlier in the book of a qubit that represents the polarization of light, which can be vertical or horizontal, or in some superposition of these states. As discussed in chapter 1, we can denote the state "vertical polarization" with $|0\rangle$ and the state "horizontal polarization" with $|1\rangle$, or equivalently, $|\uparrow\rangle$ and $|\rightarrow\rangle$.

Vectors offer a convenient mathematical notation for these states. For instance, we denote the state "vertical polarization" $|0\rangle = |\uparrow\rangle$ by the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and the state "horizontal polarization" $|1\rangle = |\rightarrow\rangle$ by the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. We refer to a vector of the form $\langle\varphi|$ as a "bra" and to a vector of the form $|\varphi\rangle$ as a "ket." So, "bras" are row vectors and "kets" are column vectors. This bra-ket notation was developed by Paul Dirac and is known as Dirac notation [68].

Basic Vector Operations

Now that we covered vector notation let's discuss what we can *do* with vectors. There are two quite natural operations to consider. The first of these is addition. We can add two vectors in the way you would expect – just add the entries! For example, we add the vectors

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

as follows:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} := \begin{pmatrix} 1+0 \\ 0+1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{11.6}$$

The notation ":=" is used to denote equality that is true by definition and not by happenstance. For example, we would write $1 + 1 = 2$ and not $1+1 := 2$, since the fact that the sum of 1 and 1 is 2 is not by definition and is a consequence of other facts. However, we would write $\mathbb{N} := \{0, 1, 2, 3, ...\}$ to indicate the set of natural numbers, denoted $\mathbb{N}$, is equal to the set $\{0, 1, 2, 3, ...\}$ by definition and not as a consequence of other facts.

11.7 **Exercise** Find the sum of the vectors $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ and $\begin{pmatrix} 4 \\ 2 \end{pmatrix}$.

Note that we cannot add vectors with different numbers of entries. For example, the expression

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{11.8}$$

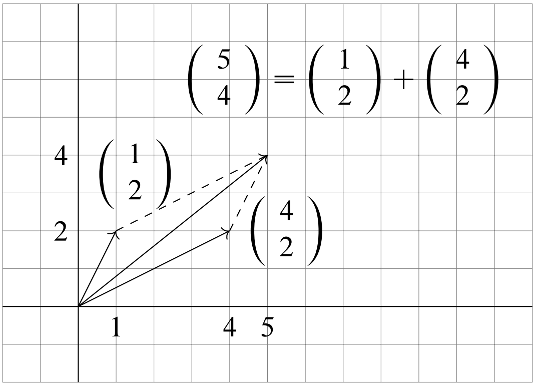
does not make sense. A moment's thought convinces us this should be the case: How would you reasonably add these?[1]

Let us now consider the next operation: a special kind of multiplication called *scalar* multiplication. This operation allows for the multiplication of a vector by a number, also called a *scalar* because it *scales* the vector. Scalar multiplication also works in the way you might expect – just multiply each entry of the vector by the number. For example, we would multiply the vector $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ by the scalar 3 as in Equation 11.9.

$$3 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} := \begin{pmatrix} 3 \cdot 1 \\ 3 \cdot 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \end{pmatrix} \tag{11.9}$$

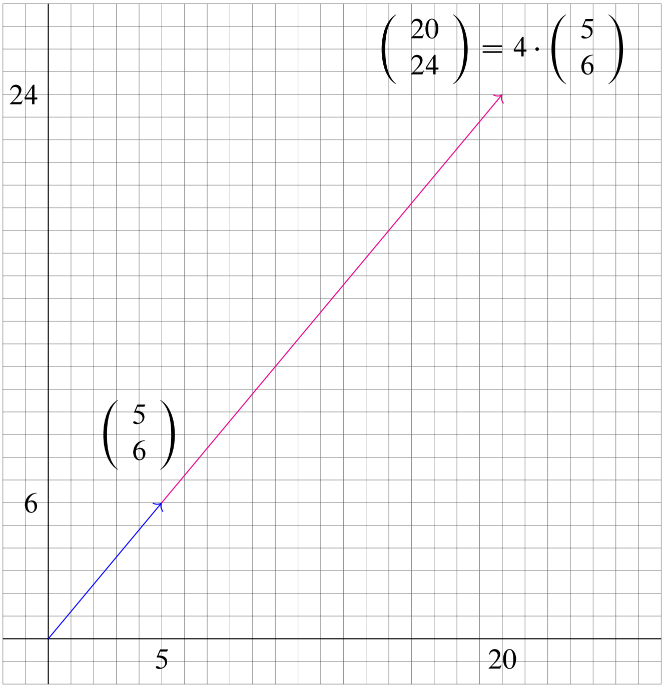
11.10 **Exercise** Perform the scalar multiplication $4 \cdot \begin{pmatrix} 5 \\ 6 \end{pmatrix}$

[1] Occasionally, computer scientists "pad" the vector which has fewer entries with extra zeros to make the addition sensible, but we won't get into this idea here.



$\begin{pmatrix} 5 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 4 \\ 2 \end{pmatrix}$
4
2
$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$
$\begin{pmatrix} 4 \\ 2 \end{pmatrix}$
1
4
5

Figure 11.2: Addition of vectors



$\begin{pmatrix} 20 \\ 24 \end{pmatrix} = 4 \cdot \begin{pmatrix} 5 \\ 6 \end{pmatrix}$
24
$\begin{pmatrix} 5 \\ 6 \end{pmatrix}$
6
5
20

Figure 11.3: Scalar multiplication of a vector

The operations of vector addition and scalar multiplication have natural geometric interpretations as well. Perhaps you recall the "head to tail" method of vector addition from a previous course. It turns out that the vector addition described above algebraically encodes precisely this approach; see Figure 11.2. Scalar multiplication by a number corresponds to "scaling" or "stretching" the vector by that number; see Figure 11.3.

After becoming comfortable with each of these operations, we may mix and match. For example, we can consider multiplying the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ by the scalar $\frac{3}{5}$, yielding the vector

$$\frac{3}{5} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{3}{5} \cdot 1 \\ \frac{3}{5} \cdot 0 \end{pmatrix} = \begin{pmatrix} \frac{3}{5} \\ 0 \end{pmatrix} \tag{11.11}$$

and then multiplying the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ by the scalar $\frac{4}{5}$, yielding the vector

$$\frac{4}{5} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{4}{5} \cdot 0 \\ \frac{4}{5} \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{4}{5} \end{pmatrix} \tag{11.12}$$

and then adding the resulting vectors, yielding

$$\begin{pmatrix} \frac{3}{5} \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{4}{5} \end{pmatrix} = \begin{pmatrix} \frac{3}{5} \\ \frac{4}{5} \end{pmatrix} \tag{11.13}$$

In sum, we refer to the expression

$$\frac{3}{5} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \frac{4}{5} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{3}{5} \\ \frac{4}{5} \end{pmatrix} \tag{11.14}$$

as a *linear combination* of the vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. We'll revisit linear combinations in greater detail later. Using Dirac notation, we can express this as

$$\frac{3}{5} |0\rangle + \frac{4}{5} |1\rangle \tag{11.15}$$

Having read some of chapter 3, you might recognize this expression as a *superposition* of the states $|0\rangle$ and $|1\rangle$. The numbers $\frac{3}{5}$ and $\frac{4}{5}$ in the linear combination above are often called *coefficients*, or in the language of quantum mechanics, *amplitudes*, in the superposition of the states $|0\rangle$ and $|1\rangle$. The square of their absolute values, $\left|\frac{3}{5}\right|^2$ and $\left|\frac{4}{5}\right|^2$, are the probabilities of observing each of the states, $|0\rangle$ and $|1\rangle$, upon measurement. So, in this example, $|1\rangle$ is more likely to be observed than $|0\rangle$. Note that in quantum mechanics, we often use complex numbers as the amplitudes, not real numbers.

11.16 **Exercise** You're invited to verify expression 11.15 is a *bona fide* superposition of states in the sense that, as per Born's rule, the sum of the squares of the absolute values of the coefficients (or amplitudes, in the language of quantum mechanics) $\frac{3}{5}$ and $\frac{4}{5}$ is in fact 1:

$$\left|\frac{3}{5}\right|^2 + \left|\frac{4}{5}\right|^2 = 1$$

11.17 **Exercise** Can you find a superposition of the states $|0\rangle$ and $|1\rangle$ so that each of the states $|0\rangle$ and $|1\rangle$ has equal probability of measurement? Be careful, you have to make sure the squares of the coefficients, or amplitudes, adds up to exactly 1! If you read carefully, you'll find the answer in previous chapters.

11.18 A superposition is a linear combination

This leads to an important observation linking quantum mechanics and linear algebra: a superposition of states can be represented as a linear combination of the vectors representing their states.

The Norm of a Vector

It is quite natural to ask what the length of a vector is given the geometric interpretation described above. For vectors in two-dimensional space, our answer is given by the Pythagorean Theorem. To see this, consider the vector $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$. We may plot this vector in the plane, as in Figure 11.4.

So, by the Pythagorean Theorem, we see that the length of this vector is the square root of the sum of the squares of the entries, i.e., $\sqrt{3^2 + 4^2} = 5$.

11.19 **Exercise** Plot the vector $\begin{pmatrix} 5 \\ 12 \end{pmatrix}$ in the plane. What is the length of this vector?

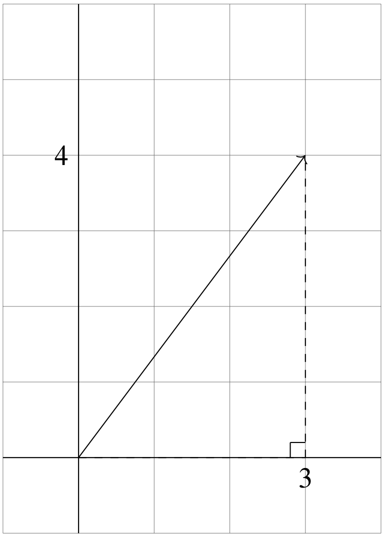


Figure 11.4: Another vector in the plane

What about vectors with three entries? Let's imagine the three-dimensional analog of our previous discussion about lengths of two-dimensional vectors. Consider the vector $\begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}$. We may plot this vector in three-dimensional space, as in Figure 11.5.

The computation of the length of this vector reduces to two applications of the Pythagorean Theorem, and ultimately, in the expression

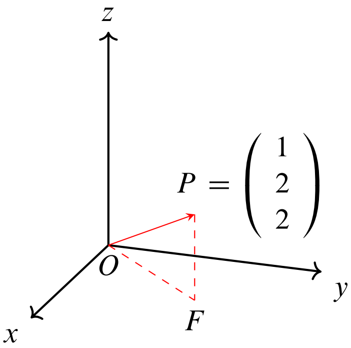


Figure 11.5: Vector in space

$$\sqrt{1^2 + 2^2 + 2^2} = \sqrt{1 + 4 + 4} = \sqrt{9} = 3 \tag{11.20}$$

for the length of the vector $\begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}$.

11.21 **Exercise** Convince yourself that the computation of the length of a three-dimensional vector reduces to two applications of the Pythagorean theorem using the following technique: Think of a point P in three-dimensional space. Now, draw the line segment OP whose endpoints are the origin O and P. From P drop a perpendicular to the floor. Refer to the point beneath P on the floor by the name F. Draw the line segment PF whose endpoints are P and F. Then, draw the line segment OF whose endpoints are O and F. Now, you have formed a triangle with vertices OP, PF, and OF.

To compute the length of the vector pointing from the origin to the point P, we must compute the length of the hypotenuse of the triangle we've just created. We can't do that just yet. We focus attention on the line segment OF. It is a line segment on the floor, a two-dimensional space. Thus, we can compute the length of the line segment OF using the usual Pythagorean Theorem. The length of the line segment PF is simply the height off of the floor of the point P. So, we know the length of two of the sides of our triangle. We can find the third using a second application of the usual Pythagorean Theorem. Draw the picture, and compare to the one in Figure 11.5!

So, we should expect that the length of any vector $\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$ is given by a generalization of the Pythagorean Theorem to n dimensions as

$$\sqrt{v_1^2 + v_2^2 + ... + v_n^2} \tag{11.22}$$

Remarkably, this is the case! You're invited to think about why. The reason is similar to the above exercise. Recalling that the square root is equivalent to the $\frac{1}{2}$ power, and rewriting this expression as

$$\sqrt{v_1^2 + v_2^2 + ... + v_n^2} = \left(v_1^2 + v_2^2 + ... + v_n^2\right)^{\frac{1}{2}} \tag{11.23}$$

reveals an interesting pattern. What is so special about the number 2? Why not do this for other numbers? For example, exchange 2 for 3, yielding

$$\sqrt[3]{v_1^3 + v_2^3 + ... + v_n^3} = \left(v_1^3 + v_2^3 + ... + v_n^3\right)^{\frac{1}{3}} \tag{11.24}$$

These variations on the theme of the length of a vector are known as *norms*.

More generally, given a vector $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$, the L^p-norm[2] of v is given by

11.25 Definition ***L^p-norm of a vector***

$$||v||_p := \left(v_1^p + v_2^p + ... + v_n^p\right)^{\frac{1}{p}}$$

We call the usual length of a vector the L^2-length to emphasize its calculation via the L^2 norm. Likewise, we refer to the length of a vector calculated using the L^p norm as the L^p-length. In what follows, we exclusively consider the L^2 norm, and we will make clear when we are using another norm.

The Dot Product

During our discussion of the addition and scalar multiplication operations we can perform with vectors, the curious reader might have wondered whether there is a natural way to multiply two vectors. Interestingly enough, the natural approach where we multiply the corresponding entries is not desirable. One reason we would like to avoid this type of multiplication for vectors is that this "product" of two non-zero vectors can often be zero.[3] For example, if we take the two non-zero vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ we would have:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 0 \\ 0 \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{11.26}$$

Who cares? One argument for why this is not desirable is that it lacks geometric interpretation.[4] There is a way, however, of multiplying two vectors with a useful geometric interpretation that we will now explore.

[2] We sympathize with the observant reader who notices that the p value of the norm in the definition is noted with a subscript. However, when writing L^p, we use a superscript.

[3] Note that this component-wise product is known as the Hadamard product. While this product is named after Jacques Hadamard, it has no relation to the Hadamard operator beyond the eponymous connection.

[4] Another argument is that in most number systems of interest, the product of two numbers is zero iff at least one of the numbers is zero. Number systems enjoying this property are called *integral domains* and are an important class of objects in the study of abstract algebra. We exploit this property of numbers when solving equations. For example, we solve the equation

Curiously, the product we will focus our attention on takes two vectors of the same number of components as input and yields, not a vector, but rather a number. As stated earlier, we can refer to a number as a *scalar quantity* or *scalar* to remind us that numbers “scale” vectors. To motivate the definition we are about to give, let us return to the notion of the length of a vector. Given a vector $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$, we determined that its L^2-norm is

$$||v||_2 := \left(v_1^2 + v_2^2 + ... + v_n^2\right)^{\frac{1}{2}} \tag{11.27}$$

and so the square of its length is the square of its L^2 norm

$$||v||_2^2 = \left(\left(v_1^2 + v_2^2 + ... + v_n^2\right)^{\frac{1}{2}}\right)^2 = v_1^2 + v_2^2 + ... + v_n^2 \tag{11.28}$$

Inspecting this expression further, we can express

$$||v||_2^2 = v_1^2 + v_2^2 + ... + v_n^2 \tag{11.29}$$

equivalently as

$$v_1 \cdot v_1 + v_2 \cdot v_2 + ... + v_n \cdot v_n \tag{11.30}$$

So

$$||v||_2 = (v_1 \cdot v_1 + v_2 \cdot v_2 + ... + v_n \cdot v_n)^{\frac{1}{2}} \tag{11.31}$$

Squaring both sides, we have

$$||v||_2^2 = v_1 \cdot v_1 + v_2 \cdot v_2 + ... + v_n \cdot v_n \tag{11.32}$$

At the very least, this instructs how we could multiply a vector by itself: multiply the corresponding entries by themselves and then sum these products. More precisely, for a vector $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$, we’re inspired to define the product of v with itself $v \cdot v$ by

$$v \cdot v := v_1 \cdot v_1 + v_2 \cdot v_2 + ... + v_n \cdot v_n \tag{11.33}$$

$x^2 - 1 = 0$ by factoring the left-hand side as $x^2 - 1 = (x + 1)(x - 1)$, and then realize that the only way the product $(x + 1)(x - 1)$ could be zero is if either $x + 1 = 0$ or $x - 1 = 0$. We then conclude $x = -1$ or $x = 1$.

This is simply the square of the L^2 norm of v! Interesting. So, if we define our multiplication like this, we recover the square of the length, or L^2 norm of a vector, in the special case that we multiply the vector by itself. Guided by this example, we define the *dot product* of two vectors with an equal number of entries

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}, v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$$

in the manner below.

11.34 **Definition** ***The dot product of two vectors***

$$u \cdot v := u_1 \cdot v_1 + u_2 \cdot v_2 + ... + u_n \cdot v_n$$

We sometimes refer to the *dot product* of two vectors as the *scalar product* to emphasize that the dot product produces a scalar.

11.35 **Exercise** The reader is invited to verify that in the case that $u = v$, we recover the square of the usual length or L^2, the norm of the vector u.

11.36 **Definition** ***The L^2 norm of a vector as a dot product***

$$v \cdot v = v_1 \cdot v_1 + v_2 \cdot v_2 + ... + v_n \cdot v_n = ||v||_2^2$$

So, the multiplication of two vectors that we've defined is convenient geometrically in that multiplying a vector by itself in this fashion yields a number that can be interpreted as the square of the length of the vector! We'll also see later that vectors whose dot product is zero can be thought of as being *orthogonal* to one another.[5]

[5] The reader may wonder what the difference is between the terms *orthogonal* and *perpendicular*. The term orthogonal is more general than the term perpendicular in that it covers the case where one of the vectors is the zero vector. In this case, we must use the term orthogonal and not perpendicular. Let us recall that the dot product of any vector with the zero vector is zero. However, it doesn't really make sense to say that a vector and the zero vector are

11.3 *The Complex Numbers and the Inner Product*

Complex Numbers

The more experienced reader might be familiar with complex numbers. Complex numbers arise naturally when solving equations. For example, we have no trouble solving the equation $x^2 - 1 = 0$. However, solving the equation $x^2 + 1 = 0$ is more confusing. If we subtract 1 from both sides, we have the equation $x^2 = -1$, and we are now faced with the question of finding a number whose square is negative. There is no *real* number whose square is negative, so we introduce a new number, named i, whose square is -1:

$$i^2 = -1 \tag{11.37}$$

In essence, i is the solution to the equation $x^2 + 1 = 0$, along with $-i$, of course (as you should check).

We refer to i as the *imaginary number*.[6] We can then form *complex numbers* by combining a real part and an imaginary part, like so:

$$0 + i, \quad 1 + i, \quad 2 + 3i, \text{ and } \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i \tag{11.38}$$

Note that the *complex* in *complex numbers* is due to the fact that these numbers consist of two constituent parts, and thus form a complex, not because they're complicated (although they might be to some). In fact, every complex number can be uniquely expressed in the form $a + bi$ for some real numbers a and b, and so we might as well define them as such. We usually denote complex numbers with symbols like z and w, and for a complex number of the form $z = a + bi$, we refer to the real part as $Re(z) = \mathscr{R}(z) = a$ and the imaginary part as $Im(z) = \mathscr{I}(z) = b$. The set of real numbers, $\mathbb{R}$, may be recovered from the set of complex numbers, $\mathbb{C}$, as the numbers $a + bi$ such that $b = 0$, i.e., numbers of the form $a + 0i = a$.

In an earlier chapter, we discussed Born's Rule which states that the square of the *modulus* of the amplitude of a state is the probability of that state resulting after measurement. Let us now review the concept of modulus and the need for it in this context. With real numbers, there is no need to invoke the

perpendicular to each other, since there is really no angle formed between a vector and the zero vector.

[6] We, along with Gauss, lament this unfortunate nomenclature.

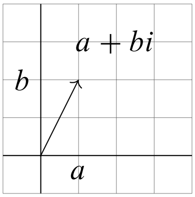


Figure 11.6: Complex number in the plane

modulus, since squaring the number automatically makes it positive. However, when squaring a complex number, the result can be a negative number. Since it is not possible to have a negative probability, we require that we apply the modulus first before squaring. The modulus of a complex number $a + bi$ is defined to be

$$\sqrt{a^2 + b^2} \tag{11.39}$$

Thus, the modulus squared is simply $a^2 + b^2$, which is always a real, positive number, which is what we require for a probability in the measurement of quantum systems.

Now that we are armed with complex numbers, let us examine the properties of vectors whose components are complex, as in Expression 11.40.

$$\begin{pmatrix} 1+i \\ 1-i \end{pmatrix} \tag{11.40}$$

It is a bit more difficult to develop a geometric interpretation for such vectors. How would we plot this vector in space? It has four dimensions![7]

However, each *single* complex number has a natural geometric interpretation as a vector in the plane. To see this, realize that any complex number $a + bi$ may be identified with the vector $\begin{pmatrix} a \\ b \end{pmatrix}$, as in Figure 11.6.

11.41 **Exercise** Where do the real numbers "live" in the complex plane? Can you draw the real line as it is embedded in the complex plane?

[7] We admit that we have not yet given a definition of dimension. For now, it is fine to think of dimension in the intuitive sense: a point is 0-dimensional, a line is 1-dimensional, a plane is 2-dimensional, space is 3-dimensional, etc. We will give a mathematical definition of dimension later!

The Inner Product as a Refinement of the Dot Product

We have discussed how to determine the square of the norm, or the square of the length, of such a vector $\begin{pmatrix} a \\ b \end{pmatrix}$: we compute

$$\left(\sqrt{a^2+b^2}\right)^2 \tag{11.42}$$

Naturally, this is how we define the norm of the complex number $a+bi$ so that it corresponds to the square of the length of its corresponding vector when plotted in the plane. That is, we define the square of the norm of a complex number $a+bi$, which may be thought of as a vector $\begin{pmatrix} a \\ b \end{pmatrix}$, as a^2+b^2.

This definition of the norm of a complex number might confuse you, and rightfully so! If you are not confused, consider this. Every number may be thought of as a vector with only one entry. This is a subtle, but important philosophical point – every number is a vector, in fact.[8] Thinking of a complex number $a+bi$ then as a vector $v = \begin{pmatrix} v_1 \end{pmatrix} = \begin{pmatrix} a+bi \end{pmatrix}$ with only one entry and following the definition of the squared norm of a vector given earlier, we compute the square of the norm of $a+bi$ via

$$||v||_2^2 := \left(\sqrt{v_1^2}\right)^2 = v_1^2 = (a+bi)^2 = (a+bi)(a+bi) \tag{11.43}$$

$$= a^2 + a\cdot bi + bi\cdot a + (bi)^2 = (a^2-b^2) + (2ab)i \tag{11.44}$$

remembering that we multiply complex numbers $(a+bi), (c+di)$ using the usual distributive law of multiplication, and that $i^2=-1$. So, for example, by the distributive property of multiplication, the product of complex numbers $1+2i$ and $3+4i$ is

$$(1+2i)\cdot(3+4i) = (1\cdot3+1\cdot4i+2i\cdot3+2i\cdot4i) = 1\cdot3+(1\cdot4)i+(2\cdot3)i+(2\cdot4)i^2 \tag{11.45}$$

$$= 1\cdot3+(1\cdot4+2\cdot3)i+(2\cdot4)(-1) = (1\cdot3-2\cdot4)+(1\cdot4+2\cdot3)i = -5+10i \tag{11.46}$$

[8] ... and every vector is a matrix! See the discussion of matrices in Section 12.4.

The reader can regard i as x, replacing any x^2 encountered in the computation instead with -1. In fact, this is exactly how algebraists think of this.

11.47 **Exercise** Practice multiplying complex numbers by multiplying

$$(3+4i)(4+5i)$$

Then, multiply

$$(3+4i)(3-4i).$$

Does the second multiplication remind you of something we discussed previously in the Norms section?

In any case, what we have now for the square of the norm

$$||v||_2^2 = (a^2-b^2)+(2ab)i \tag{11.48}$$

does not look at all like what we called the square of the norm a^2+b^2 of the complex number $a+bi$. Even more disturbing is that $(a^2-b^2)+(2ab)i$ is potentially an imaginary number, since $2ab$ is likely nonzero (if both a and b are nonzero). This is disturbing because such an expression cannot be as easily interpreted as a length and we would like the norm of a complex number to be interpretable as its length.[9]

What is going on here?

Let us inspect the expression a^2+b^2 a bit more carefully. Clever readers (who have done the previous exercise) might recognize expression 11.49 (check it using the distributive property).

$$a^2+b^2 = (a+bi)(a-bi) \tag{11.49}$$

The number $a-bi$ is known as the *complex conjugate* of the number $a+bi$. We can also simply call it the *conjugate* when we're well-aware that we're working with complex numbers. We often denote a complex number by z, and the conjugate of z by $\overline{z}$. For example, if we denote $a+bi$ by z, then $\overline{z}=a-bi$. The act of changing a complex number into its conjugate is known as *conjugating*. You might wonder how to conjugate a complex

[9] Refer to the later section on *Hermitian Operators* where we discuss why we can't measure with complex numbers.

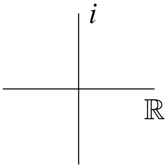


Figure 11.7: The complex plane

number like $1 - i$. The answer is $1 + i$. That is, plus becomes minus and vice versa.

11.50 **Exercise** Conjugate the following: $3 + 4i, 3 - 4i, -1 - i$.

What if we have a real number a and think of it as a vector $\begin{pmatrix} a \end{pmatrix}$ with only one entry? Then, the square of the norm of this vector, following the previous definition, is simply a^2. We may recover the norm of a by taking (the positive) square root, which makes sense since the length of a should be simply a, and we do not encounter any difficulties as we did with complex numbers.

One explanation for this peculiarity is that perhaps we should have been considering real numbers as complex numbers all along. In fact, a previous exercise asking you to realize the real line as "living" in the complex plane makes this explicit. You can check your solution to that exercise by referencing Figure 11.7. What we mean by this is that a real number a can be thought of as a complex number by expressing it as

$$a = a + 0i \tag{11.51}$$

Writing a as $a + 0i$ reveals that the complex conjugate of a is simply a, since $a + 0i$ is $a - 0i = a$. Then, the product of a and its complex conjugate is

$$(a - 0i)(a + 0i) = a \cdot a = a^2 \tag{11.52}$$

In other words, our definition of the square of the norm for real numbers remains the same even if we involve the conjugate!

This observation makes us realize that we should define the square of the norm of any number, complex or not, as the product of that number with its conjugate. More precisely, using the previously introduced notation,

$$||z||_2^2 := \overline{z}z \tag{11.53}$$

We often omit this extensive notation when the context makes clear that we are using the L^2 norm, replacing it with $|z|^2 = \bar{z}z$.

This discussion of how to define the square of the norm of any number instructs our definition of the square of the norm of any vector, complex entries or not. For any vector $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$, we define the square of the norm of v to be:

11.54 **Definition** ***Squared norm of a vector***

$$|v|^2 := \bar{v}_1 \cdot v_1 + \bar{v}_2 \cdot v_2 + ... + \bar{v}_n \cdot v_n$$

11.55 **Exercise** You're encouraged to verify that this definition recovers the previous definition for vectors whose entries are exclusively real numbers, and also that it effectively computes the square of the norm of any real or complex number.

Generalizing this, we define the *inner product* of two vectors (again, with equal number of entries):

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}, v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$$

to be

11.56 **Definition** ***Inner product of two vectors***

$$\langle u, v \rangle := \bar{u}_1 v_1 + \bar{u}_2 v_2 + ... + \bar{u}_n v_n$$

11.57 **Exercise** Similarly, you're invited to verify that this definition recovers the square of the norm of any real or complex number in the case $u = v$ is a real or complex number, i.e., for any real or complex number z,

$\langle z, z\rangle = |z|^2$. You should also verify that the definition of the square of the norm is recovered for a vector whose entries are all real numbers.

It should be noted that the dot product and the inner product agree only if the entries of the vectors we are considering are exclusively real numbers. The inner product generalizes this operation to vectors with complex entries.

Recalling Dirac notation, you will likely notice the uncanny resemblance of the notations $\langle u, v\rangle$ and $\langle u|v\rangle$ – the only difference is the line in the middle! Paul Dirac was probably inspired by the inner product notation when he decided on his notation. That being said, if we have two vectors, written in Dirac notation as $\langle u|$ and $|v\rangle$, we write their inner product as

$$\langle u|v\rangle := \langle u, v\rangle \tag{11.58}$$

So, the notation $\langle u|v\rangle$ literally means "the inner product of the vectors u and v."

So, in what follows, we will assume that our vectors may include complex entries. We will revisit the inner product from a more formal and abstract point of view later on in our description of a *Hilbert space*.

The Polar Coordinate Representation of a Complex Number

We now explore a remarkable connection between complex numbers and geometry. Every complex number can be thought of as living in the two-dimensional plane depicted in Figure 11.7.

We have discussed how a complex number of the form $z := a + bi$ may be thought of as a vector $\begin{pmatrix} a \\ b \end{pmatrix}$. Thinking of z as a vector in two-dimensional space should convince us that it's reasonable to assign a complex number an L^2-norm, i.e., a length. Specifically, the complex number $z = a + bi$, thought of as a vector $\begin{pmatrix} a \\ b \end{pmatrix}$, has L^2 norm given by the Pythagorean Theorem:

$$\sqrt{a^2 + b^2} \tag{11.59}$$

Visualizing complex numbers in this fashion should also make it reasonable to assign a complex number an angle. Specifically, we assign the complex number $a + bi$ the angle, θ, subtended by the arc from the positive real

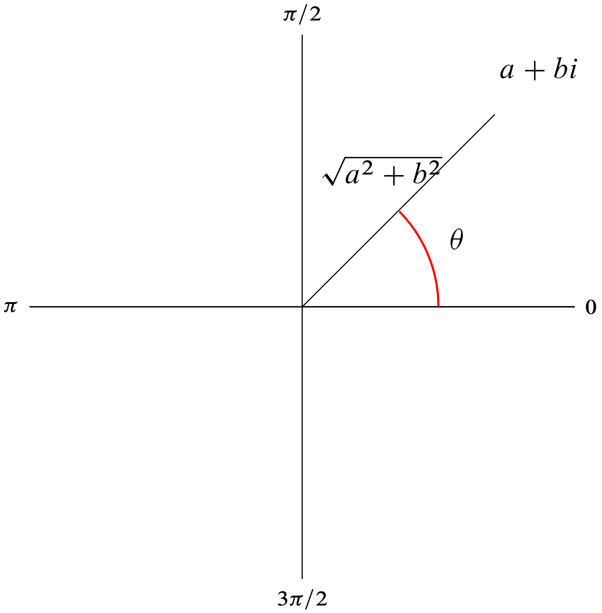


Figure 11.8: Angle subtended by radius

axis to the head of the vector $\begin{pmatrix} a \\ b \end{pmatrix}$ in the counterclockwise direction, as in Figure 11.8.

We can use some simple trigonometry to describe this angle. Let's refer to the angle by θ. Then, θ lives in the right triangle depicted in Figure 11.9 and so satisfies the equation

$$\tan(\theta) = \frac{b}{a} \tag{11.60}$$

If you're shaky on trigonometry, remember that the tangent of an angle θ in a right triangle is the *opposite over the adjacent.* Let's recall the inverse tangent function, known by the name tan ^{1}, or sometimes arctan. We'll use the name arctan.[10] The role of the inverse tangent function is to take a number as input and give an angle as output. For example, arctan(y) means "the angle that makes tangent equal to y." Since

$$\tan(\theta) = \frac{b}{a} \tag{11.61}$$

[10] tan ^{1} could be confused with the reciprocal function of tangent, known as cotangent, which is expressed (tan) $^{1} = \frac{1}{\tan} =$ cot.

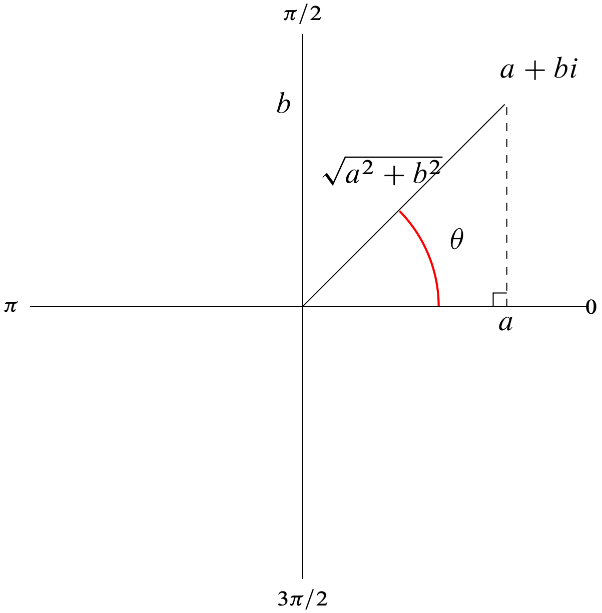


Figure 11.9: Right triangle where the angle lives

applying arctan to both sides yields the equation

$$\arctan\left(\tan(\theta)\right) = \arctan\left(\frac{b}{a}\right) \tag{11.62}$$

So, we can express the angle, θ, in terms of a and b as

$$\theta = \arctan\left(\frac{b}{a}\right) \tag{11.63}$$

All this being said, we can refer to the L^2-norm of the complex number as its radius, and refer to the angle as its... well, angle. In summary, a complex number $a + bi$ has a radius

$$r := \sqrt{a^2 + b^2} \tag{11.64}$$

and an angle

$$\theta := \arctan\left(\frac{b}{a}\right) \tag{11.65}$$

We can also describe a complex number by simply giving a radius and angle. For example, let's try to figure out which complex number is described by the radius 1 and the angle $\theta := \frac{\pi}{4}$ radians (45 degrees). If we write the complex number temporarily as $a + bi$, our problem reduces to determining the numbers a and b given the radius r and the angle θ.

We know from the previous paragraph that the radius r of a complex number of the form $a + bi$ is given by

$$r = \sqrt{a^2 + b^2} \tag{11.66}$$

and we know that the angle θ is given by

$$\theta = \arctan\left(\frac{b}{a}\right) \tag{11.67}$$

If we now apply tan to both sides of 11.67, we obtain

$$\tan(\theta) = \frac{b}{a} \tag{11.68}$$

Since $\theta = \frac{\pi}{4}$,

$$\tan(\theta) = \tan\left(\frac{\pi}{4}\right) = 1 \tag{11.69}$$

So,

$$1 = \tan\left(\frac{\pi}{4}\right) = \tan(\theta) = \frac{b}{a} \tag{11.70}$$

and we see that

$$1 = \frac{b}{a} \tag{11.71}$$

Multiplying by a on both sides reveals $a = b$. Great, so now we know that $a = b$!

Knowing that $a = b$, we look toward the equation

$$r = \sqrt{a^2 + b^2} \tag{11.72}$$

We are told that the radius r is equal to 1, so we know that

$$1 = r = \sqrt{a^2 + b^2} \tag{11.73}$$

and since $a = b$, we may replace b with a yielding instead

$$1 = r = \sqrt{a^2 + b^2} = \sqrt{a^2 + a^2} = \sqrt{2a^2} \tag{11.74}$$

Squaring both sides yields $1 = 2a^2$, and dividing both sides by 2 reveals that $\frac{1}{2} = a^2$. Then, taking the square root of both sides reveals that:

$$a = \sqrt{\frac{1}{2}} \tag{11.75}$$

It turns out that

$$\sqrt{\frac{1}{2}} = \frac{\sqrt{1}}{\sqrt{2}} = \frac{1}{\sqrt{2}} \tag{11.76}$$

so we now know that

$$a = b = \frac{1}{\sqrt{2}} \tag{11.77}$$

Fantastic — we now know our complex number $a + bi$ described by the radius 1 and the angle $\frac{\pi}{4}$ is actually

$$a + bi = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i \tag{11.78}$$

So, we have converted the polar expression of a complex number given by radius 1 and angle $\theta = \frac{\pi}{4}$ to what we refer to as its rectangular expression $\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i$.

11.79 Definition ***Rectangular (or Cartesian) and polar form of a complex number***

In general, we say that a complex number of the form $a + bi$ is given in *rectangular* or *Cartesian* form, and that a complex number described by a radius r and an angle θ as (r, θ) is given in *polar* form.

11.80 Exercise Check that $z = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i$ enjoys the above properties, i.e., z has radius 1 and angle $\frac{\pi}{4}$.

11.81 Exercise Try to convert the Cartesian expression $\frac{\sqrt{3}}{2} + \frac{1}{2}i$ to polar form by determining its radius and angle.

11.82 Exercise To see that Cartesian expressions can sometimes have "ugly" polar expressions, convert $1 + 2i$ to polar coordinates and check that the radius is $\sqrt{5}$ and the angle is $\arctan\left(\frac{2}{1}\right)$, which, unfortunately, cannot be expressed in any more familiar way.[11]

[11] In fact, the arctan of any natural number is irrational.

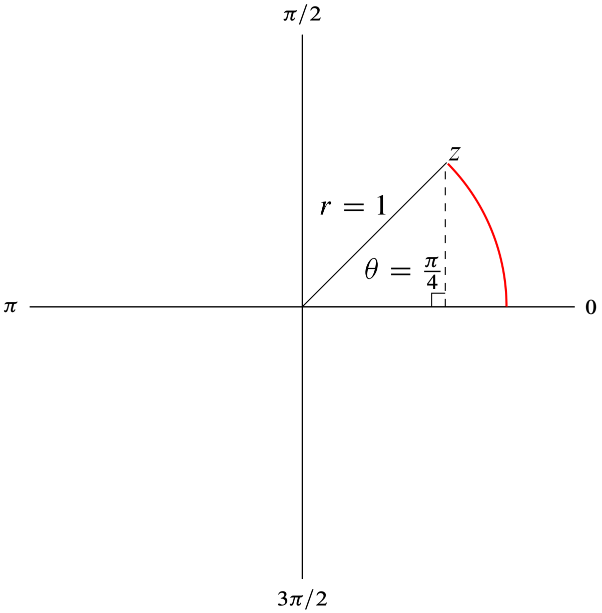


Figure 11.10: Unit complex number with angle $\frac{\pi}{4}$

Recalling a bit of trigonometry, we could have determined this already! Given the radius $r = 1$ and the angle $\theta = \frac{\pi}{4}$ radians, we have the picture in Figure 11.10.

Recalling the definition of sine and cosine, we see that the value for a should be

$$a = r \cdot \cos(\theta) \tag{11.83}$$

and the value for b should be

$$b = r \cdot \sin(\theta) \tag{11.84}$$

We can reformulate a as

$$a = 1 \cdot \cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \tag{11.85}$$

and b as

$$b = 1 \cdot \sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \tag{11.86}$$

11.87 **Exercise** Check that $\frac{\sqrt{2}}{2} = \frac{1}{\sqrt{2}}$ by cross-multiplying.

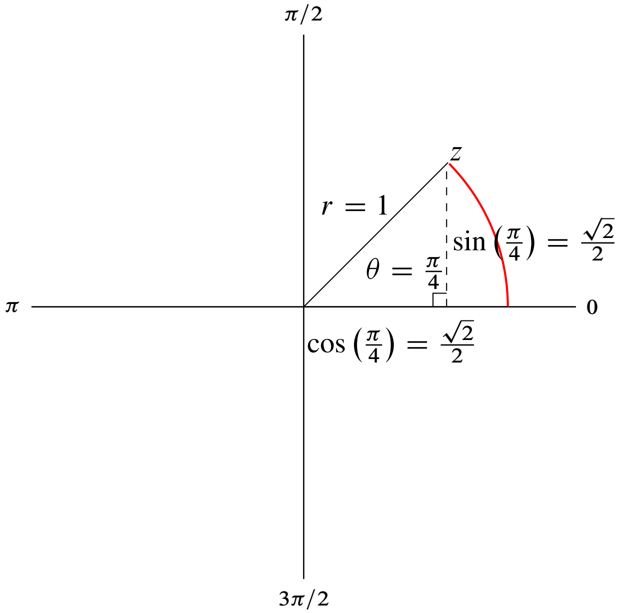

$\pi/2$
z
$r = 1$
$\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$
$\theta = \frac{\pi}{4}$
π
0
$\cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$
$3\pi/2$

Figure 11.11: The trigonometry of a complex number

The above exercise makes us realize we could have determined a and b all along just having recalled some basic trigonometry!

So, we've discussed how a complex number gives rise to a radius and an angle, and conversely, how a radius and an angle give rise to a complex number. We realize now that the descriptions are equivalent!

11.88 **Exercise** Convince yourself that giving a complex number in the form $a + bi$ is equivalent to giving a radius and an angle (r, θ). Recall that we say that the form $a + bi$ is *rectangular* or *Cartesian* (for *Cartesian coordinates*) and that the form (r, θ) is polar.

We can take this one step further with *Euler's formula* (pronounced "oiler"). Euler's formula states that a complex number z with radius 1 and angle θ, i.e., a complex number living on the "unit circle" can be expressed as

$$z = e^{i\theta} = \cos(\theta) + i\sin(\theta) \tag{11.89}$$

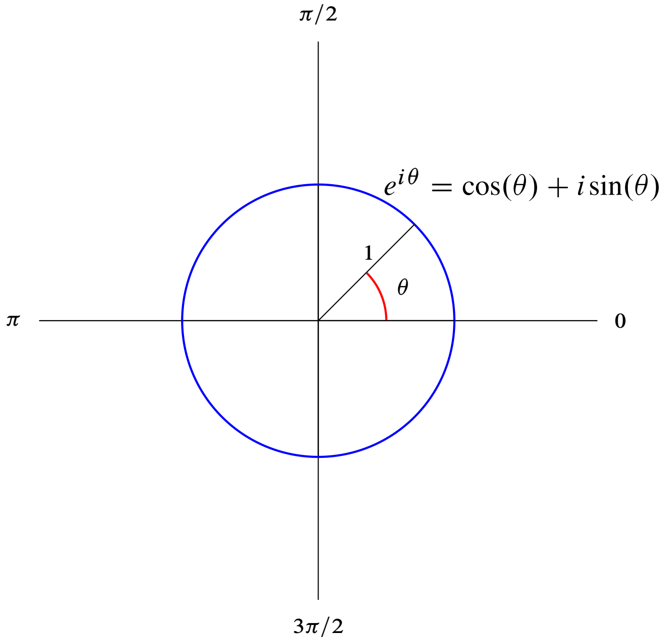


Figure 11.12: Euler's formula

That $z = \cos(\theta) + i\sin(\theta)$ should not be surprising, as it follows our previous discussion, but rather that the number e comes into play here. If you're not familiar with the number e, check out our explanation in chapter 13. This formula is one of the most remarkably beautiful formulas in all of mathematics for many reasons, including the deep connection it illuminates between complex numbers — *a priori*, an algebraic phenomenon — and geometry. We discuss this equation further and prove its validity in our chapter about additional mathematical topics. For now, we'd like for you to take away the following idea:

11.90 Euler's formula

A complex number z with radius 1 and angle θ can be expressed as

$$z = e^{i\theta}$$

That being said, if our complex number z has radius r instead of 1, and angle θ, we may write as $z = re^{i\theta}$, following Euler.

11.91 **Exercise** Convince yourself that a complex number z whose radius is r and whose angle is θ can be expressed $z = re^{i\theta}$ using Euler's formula as above.

We'll exploit this idea later to define a special class of transformations that rotate space by any specified angle!

11.4 *A First Look at Matrices*

Basic Matrix Operations

Having read chapter 3, you encountered matrices, which *a priori* resemble rectangular grids of numbers, e.g., the Pauli X operator σ_x (also known as the NOT operator):

$$\sigma_x = X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{11.92}$$

and the Pauli Z operator σ_z:

$$\sigma_z = Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{11.93}$$

from chapter 3.

Matrices might even involve complex numbers, like the Pauli Y operator σ_y:

$$\sigma_y = Y := \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{11.94}$$

We can multiply matrices by numbers, like you've seen with the Hadamard operator from chapter 3. Just multiply all of the entries by that number, like so:

$$H := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} := \begin{pmatrix} \frac{1}{\sqrt{2}} \cdot 1 & \frac{1}{\sqrt{2}} \cdot 1 \\ \frac{1}{\sqrt{2}} \cdot 1 & \frac{1}{\sqrt{2}} \cdot (-1) \end{pmatrix} \tag{11.95}$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \tag{11.96}$$

We can also add two matrices by adding their corresponding entries, like so:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} := \begin{pmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix} \tag{11.97}$$

A matrix is a representation of a more fundamental object called a *linear transformation*. In fact, so is a vector, since we may think of a vector as an $n \times 1$ matrix. The term *transformation* begs the question *transformation of what?*

11.98 Matrices transform space

A matrix does not merely transform a particular vector or set of vectors; *it transforms an entire vector space*.

To gain an understanding of how a matrix might be thought of as a transformation of space, we will consider a few geometric examples. However, before we can explain these geometric examples, we will need to learn how to multiply a vector by a matrix. This will seem a bit weird at first, but we will explain why it's defined this way later on in this text.

Let

$$\begin{pmatrix} e \\ f \end{pmatrix} \tag{11.99}$$

be a vector and

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{11.100}$$

be a matrix.

We multiply the vector $\begin{pmatrix} e \\ f \end{pmatrix}$ by the matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ like so:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e \\ f \end{pmatrix} := \begin{pmatrix} a \cdot e + b \cdot f \\ c \cdot e + d \cdot f \end{pmatrix} \tag{11.101}$$

If you're observant, you might recognize $a \cdot e + b \cdot f$ resembles the dot product described earlier. In fact, each of the expressions $a \cdot e + b \cdot f$ and $c \cdot e + d \cdot f$ (respectively) are literally the dot products of the vectors $\begin{pmatrix} a \\ b \end{pmatrix}$ and $\begin{pmatrix} e \\ f \end{pmatrix}$, and of $\begin{pmatrix} c \\ d \end{pmatrix}$ and $\begin{pmatrix} e \\ f \end{pmatrix}$ (respectively).[12]

In other words, for vectors with exclusively real entries, the multiplication we've just defined can be thought of as a sequence of dot products!

Recall the Pauli Z matrix:

$$Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{11.102}$$

[12] Actually, if all of a, b, c, d, e, f are strictly real numbers, then these dot products are literally the inner products of the vectors.

and consider the vector:

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{11.103}$$

Recall from the previous description that we write the product of the matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{11.104}$$

and the vector

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{11.105}$$

as

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{11.106}$$

and we multiply them as follows

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} := \begin{pmatrix} 1 \cdot 0 + 0 \cdot 1 \\ 0 \cdot 0 + (-1) \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{11.107}$$

We could say then that the matrix has *transformed* the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ into the vector $\begin{pmatrix} 0 \\ -1 \end{pmatrix}$!

11.108 **Exercise** Figure out where the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is sent via this transformation. More precisely, multiply the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ by the matrix $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ and see where the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ ends up! Do you see a relationship between the transformed vectors and the columns of the matrix? Try to formulate a conjecture!

In fact, we may use Dirac notation to express the relationship between the matrix Z and the states $|0\rangle$ and $|1\rangle$ like so:

$$Z\,|j\rangle = (-1)^j\,|j\rangle \tag{11.109}$$

for all $j \in \{0, 1\}$, as you should check!

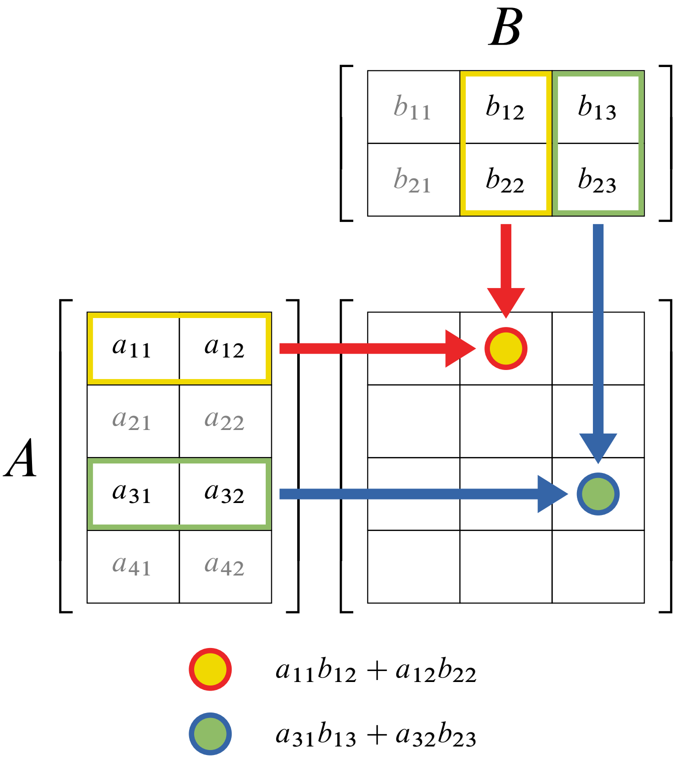


Figure 11.13: Matrix multiplication

11.110 Exercise Learn why the NOT operator $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ is also known as the "bit flip" operator by showing that multiplying the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, which represents the state $|0\rangle$, by the matrix X "flips the state" to $|1\rangle$, i.e., the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

We can likewise multiply two matrices as follows:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} := \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix} \tag{11.111}$$

A helpful graphic depicting matrix multiplication is given in Figure 11.13.

11.112 **Exercise** Realize that the multiplication of a vector by a matrix described earlier can in fact be thought of as the multiplication of two matrices, where we think of the vector as an $n \times 1$ matrix.

Extending the idea of realizing matrix-vector multiplication as simply matrix-matrix multiplication as described in the exercise above, we can also multiply matrices of different dimensions. For example, consider the matrices

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} \tag{11.113}$$

and

$$\begin{pmatrix} g & h & i \\ j & k & l \end{pmatrix} \tag{11.114}$$

where the matrix entry i is simply the letter, not the imaginary number i. Then, we may multiply these matrices like so:

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} \begin{pmatrix} g & h & i \\ j & k & l \end{pmatrix} := \begin{pmatrix} ag + bj & ah + bk & ai + bl \\ cg + dj & ch + dk & ci + dl \\ eg + fj & eh + fk & ei + fl \end{pmatrix} \tag{11.115}$$

It should be noted that the product of a 3×2 matrix and 2×3 matrix yields a 3×3 matrix!

Now that we have seen how to multiply a vector by a matrix, we can see the method for multiplying two matrices, as follows:

$$\begin{pmatrix} g & h & i \\ j & k & l \end{pmatrix} \begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} := \begin{pmatrix} ga + hc + ie & gb + hd + if \\ ja + kc + le & jb + kd + lf \end{pmatrix} \tag{11.116}$$

This should be a surprise – we got a 2×2 matrix by multiplying the two matrices in the opposite order!

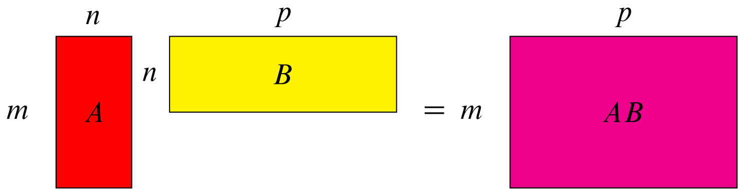


Figure 11.14: Size of the matrix product

11.117 Size of the matrix product

In general, if A is an $m \times n$ matrix and B is an $n \times p$ matrix, then AB is an $m \times p$ matrix.

The picture in Figure 11.14 depicts this phenomenon, exemplified by equations 11.115 and 11.116.

So, when multiplying two matrices, the "inner" dimensions must agree, and the resulting matrix will have dimensions equal to the "outer dimensions" of the two matrices.

When faced with the following expression

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{11.118}$$

we might wonder how to compute it given that we've not discussed how to multiply three matrices at a time. Well, think of an analogous scenario. When asked to compute the product $2 \cdot 3 \cdot 4$ what do we do? Well, you have probably never met anyone capable of multiplying three numbers at a time (at least, we have never met anyone like that). However, we know that it's fair to do either of the following things in an effort to compute the product:

Either we compute:

$$(2 \cdot 3) \cdot 4 \tag{11.119}$$

or we compute:

$$2 \cdot (3 \cdot 4) \tag{11.120}$$

That is, we either multiply 2 and 3 first, and then multiply the result (6) by the remaining 4, or we multiply 3 and 4 first, and then multiply the result (12) by the remaining 2. What is remarkable is that we get the same answer (24) either way!

We can hope that the same is true for matrices. Explore this idea in the following exercise:

11.121 **Exercise** Recognize

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

as the product ZXZ of the matrices $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ and $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Then, compute the product ZXZ by first computing ZX and then multiplying the result by the remaining Z on the right. More precisely, compute

$$(ZX)Z$$

Then, compute

$$Z(XZ)$$

and check that you get the same answer either way! So, you've shown that, at least for these three matrices, matrix multiplication is an *associative operation*.

We'll revisit associative operations later on when we discuss the formal definition of a vector space. We won't prove that matrix multiplication is associative in general here. When we say that matrix multiplication is associative, we mean that for all matrices A, B, C, $(AB)C = A(BC)$ whenever the product makes sense, i.e., the dimensions agree. In fact, we won't bring the issue up again until after we've shown that multiplication of matrices is equivalent to composition of functions, at which point the fact that matrix multiplication is associative will become an obvious fact!

So, when faced with a product like

$$ZX\,|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}|0\rangle \tag{11.122}$$

we first think of the vector on the right as a 2 × 1 matrix. We then realize that we may first compute the product of the two matrices on the left, i.e.,

$$ZX = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \tag{11.123}$$

then apply the resulting matrix to the vector (2 × 1 matrix) $|0\rangle$, like so:

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}|0\rangle = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{11.124}$$

Or we could iteratively apply the matrices to the vector $|0\rangle$, first applying the matrix X

$$X\,|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \tag{11.125}$$

then applying the matrix Z to the resulting vector, like so

$$Z(X\,|0\rangle) = Z(|1\rangle) = Z\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{11.126}$$

Either way, we get the same result!

Please note that when applying a sequence of matrices to a vector, we go from right to left, sometimes said “inside-out,” so in this case, we apply X first, then Z. However, when multiplying the matrices, we go from left to right. What we are realizing is that:

11.127 Multiplying several matrices

The result of applying the composition of two transformations to a space is equivalent to applying them iteratively.

Again, we won’t prove this here, but we promise, it will become obvious once we’ve established the correspondence between matrix multiplication (and thus, multiplication of vectors by matrices) and composition of functions.

The Identity Matrix

Thinking of matrices as transformations of space leads us to believe that there should be a matrix that does not transform the space at all. In other words, it is natural to ask if there is a matrix that has no effect on any of the vectors it multiplies. In two dimensions, the answer is the matrix

$$I_2 := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{11.128}$$

The subscript 2 is there to inform us that this is the identity matrix for 2-dimensional space. We refer to this matrix as *the identity matrix* because it preserves the *identity* of the vectors it acts on.

Let’s see that the matrix I_2 deserves its name. Let’s multiply the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ by the matrix I_2 to see what we get:

$$I_2 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 0 \cdot 0 \\ 0 \cdot 1 + 1 \cdot 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{11.129}$$

It's the same vector! We leave it to you to check that the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ is also preserved under multiplication by I_2.

11.130 **Exercise** Verify that the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ is unchanged under multiplication by I_2.

However, the matrix just described is the identity matrix for 2-dimensional space. What if we want an identity matrix for three-dimensional space? No problem – just make the matrix a little bigger:

$$I_3 := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{11.131}$$

We also call this the identity matrix, although it is specific to three-dimensional space.

11.132 **Exercise** Multiply each of the vectors $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ by this new identity matrix for three-dimensional space and check that it deserves its name.

In fact, each dimension has its own identity matrix, as you might now expect! To build the identity matrix for n-dimensional space, simply create an $n \times n$ matrix with 1's along the diagonal and 0's elsewhere, like so

$$I_n := \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{11.133}$$

This matrix has the property that it preserves any vector of n-dimensions that it acts on, as you should verify.

Transpose, Conjugate and Trace

We have now seen that not all matrices have to be square. For example, consider

$$A := \begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}, B := \begin{pmatrix} g & h \\ i & j \end{pmatrix} \tag{11.134}$$

The matrix A has 3 rows and 2 columns and B has 2 rows and 2 columns, so B is square, while A is not. A very natural operation to consider when thinking of matrices as rectangular grids of numbers is the *transpose*. Here are the transposes, denoted A^T and B^T, of the above matrices:

$$A^T := \begin{pmatrix} a & c & e \\ b & d & f \end{pmatrix}, B^T := \begin{pmatrix} g & i \\ h & j \end{pmatrix} \tag{11.135}$$

What happened? We could describe this operation as *turning rows into columns* and vice versa. Visual learners might recognize that transposing a matrix is reflecting its entries over an imaginary line extending from the upper left-hand corner of the matrix to the lower right-hand corner. We can just as well transpose the transposed matrices, i.e., compute $(A^T)^T$ and $(B^T)^T$. You're invited to check that $(A^T)^T$ is just A again in an exercise below.

11.136 **Exercise** Find the transpose of the following matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

11.137 **Exercise** Check that transposing the already transposed matrix A^T yields A. More precisely, check that $(A^T)^T = A$. So, the operation of transposing inverts itself!

11.138 **Exercise** For which matrices A is A equal to A^T? We call such matrices *symmetric* (for good reason). If a matrix is symmetric and all of its entries are real numbers, we call it *real symmetric*. Real symmetric matrices are very special matrices that deserve quite a bit of attention, as we'll see later.

We can transpose vectors just as well as matrices. To see this, we recognize that we can think of any vector as a matrix. For example, think of the vector

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

as a 2×1 matrix, so its transpose is the 1×2 matrix

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}^T = \begin{pmatrix} 1 & 0 \end{pmatrix} \tag{11.139}$$

11.140 **Exercise** Find the transpose of the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, i.e., find

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix}^T$$

We'd like to mention now why it is that we consider the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and its transpose $\begin{pmatrix} 1 \\ 0 \end{pmatrix}^T = \begin{pmatrix} 1 & 0 \end{pmatrix}$ different objects. Of course, they are visually different, but the difference is more than just in their presentation.

For starters, we cannot multiply the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ by itself, i.e., the expression

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{11.141}$$

does not make sense, as you should recall from our previous discussion about appropriate dimensions for the product of two matrices. Explicitly, the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ has dimensions 2×1, and we see that the product of a 2×1 matrix with a 2×1 matrix does not make sense since the "inner" dimensions, 1 and 2, do not agree.

However, the expression

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{11.142}$$

makes perfect sense, since the inner dimensions are both 2, as you should verify. Thinking of each of $\begin{pmatrix} 1 \\ 0 \end{pmatrix}^T$ and $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ as matrices and following the description of matrix multiplication described above, we have

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1 \cdot 1 + 0 \cdot 0 = 1 \tag{11.143}$$

This result makes sense since we multiplied a 1 × 2 matrix by a 2 × 1 matrix and got a number, i.e., a 1 × 1 matrix!

Another fun operation we can perform on matrices is conjugation. Yes, the same conjugation from earlier! Let's see how this works. Consider the matrix

$$C := \begin{pmatrix} 1+i & 0 \\ 0 & 1-i \end{pmatrix} \tag{11.144}$$

We denote by $\overline{C}$ the conjugate of the matrix C.

$$\overline{C} := \begin{pmatrix} \overline{1+i} & \overline{0} \\ \overline{0} & \overline{1-i} \end{pmatrix} = \begin{pmatrix} 1-i & 0 \\ 0 & 1+i \end{pmatrix} \tag{11.145}$$

To conjugate a matrix, just conjugate each of its entries.

11.146 **Exercise** Figure out when a matrix is equal to its conjugate. Hint: When is a 1 × 1 matrix, i.e., a number, equal to its conjugate?

Of course, viewing a vector as a matrix allows us to conjugate any vector. For example, the conjugate of the vector $\begin{pmatrix} 1+i \\ 1-i \end{pmatrix}$ is the vector

$$\overline{\begin{pmatrix} 1+i \\ 1-i \end{pmatrix}} := \begin{pmatrix} \overline{1+i} \\ \overline{1-i} \end{pmatrix} = \begin{pmatrix} 1-i \\ 1+i \end{pmatrix} \tag{11.147}$$

Let's tie a few concepts together now by expressing the inner product of two vectors using the notation we've developed.

Given two vectors $u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}$ and $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$, we would express their inner product via

$$\langle u, v\rangle := \overline{u_1} \cdot v_1 + \overline{u_2} \cdot v_2 + ... + \overline{u_n} \cdot v_n \tag{11.148}$$

using our earlier notation. Notice that

$$\overline{u_1} \cdot v_1 + \overline{u_2} \cdot v_2 + ... + \overline{u_n} \cdot v_n = \begin{pmatrix} \overline{u_1}, & \overline{u_2}, & \ldots, & \overline{u_n} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{11.149}$$

which we may express using our recently developed notation for transpose as follows:

$$\begin{pmatrix} \overline{u_1}, & \overline{u_2}, & \ldots, & \overline{u_n} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} \overline{u_1} \\ \overline{u_2} \\ \vdots \\ \overline{u_n} \end{pmatrix}^T \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{11.150}$$

and we may further adorn this expression with our new notation for the conjugate of a matrix (and thus a vector) as follows:

$$\begin{pmatrix} \overline{u_1} \\ \overline{u_2} \\ \vdots \\ \overline{u_n} \end{pmatrix}^T \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \overline{u}^T v \tag{11.151}$$

All in all, we may compactly express the inner product of vectors u and v as simply $\overline{u}^T v$ – fantastic! So, in summary, we have the following equivalent expressions for the inner product of two vectors u and v:

$$\langle u, v\rangle = \langle u|v\rangle = \overline{u}^T v \tag{11.152}$$

By convention, we can refer to $\overline{u}^T v$ as $u^\dagger$, so we have

$$\langle u, v\rangle = \langle u|v\rangle = \overline{u}^T v = u^\dagger v \tag{11.153}$$

We can go even further to relate this to Dirac notation! What the above string of equalities reveals is that we may think of the "bra" $\langle u|$ of a vector u as the conjugate-transpose of the vector u, i.e., for a vector $\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}$,

$$\langle u|\,|v\rangle = \overline{u}^T v = \begin{pmatrix} \overline{u_1} \\ \overline{u_2} \\ \vdots \\ \overline{u_n} \end{pmatrix}^T \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{11.154}$$

$$= \begin{pmatrix} \overline{u_1}, & \overline{u_2}, & \ldots, & \overline{u_n} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{11.155}$$

Given that we now have the operations of transposition and conjugation at our disposal, we may apply both to a matrix to yield what is known as its *conjugate transpose*.

11.156 **Exercise** First, conjugate the Pauli Y operator σ_Y, i.e., the matrix $Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$. Then, transpose the result. We call this the conjugate transpose of the matrix Y. Do you notice something special about the relationship between Y and its conjugate transpose?

We'll show now that you could equivalently transpose the matrix Y and then conjugate.

First, we transpose Y, yielding

$$Y^T = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}^T = \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix} \tag{11.157}$$

Then, we conjugate Y^T:

$$\overline{(Y^T)} = \overline{\begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix}} = \begin{pmatrix} \overline{0} & \overline{i} \\ \overline{-i} & \overline{0} \end{pmatrix} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{11.158}$$

Check to see that you got the same thing in the exercise above!

So, in general, we may compute the *conjugate transpose* of a matrix, and we may compute it in either order: conjugate first, then transpose, or transpose first, then conjugate.

We will see later that the idea of the conjugate transpose is important for defining a class of operators called *unitary operators*. A unitary operator is an operator whose inverse is its conjugate transpose, as we will consider further in this section. This is important because quantum states are represented as vectors with norm 1 living in something called a Hilbert space (which we will define rigorously later in this exposition). It turns out that unitary operators have the special property that they preserve the norm of the vectors on which they operate. So, the application of a unitary operator to a vector whose norm is 1 is a vector whose norm is also 1.

There is another important operation that we can perform on a matrix, known as *taking the trace*. Let us recall that the primary diagonal of interest to us in analyzing matrices is the one that runs from the upper left-hand corner to the lower right-hand corner; we refer to this as the *main diagonal*. Given a matrix A, we can find the sum of entries of the main diagonal of A like so:

$$A = \begin{pmatrix} 1 & 3 \\ 8 & 4 \end{pmatrix} \mapsto 1 + 4 = 5 \tag{11.159}$$

We notate this as $Tr(A) = 5$, which can be read, “The trace of A is 5.”

11.160 **Exercise** Compute the trace of the matrix $B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$. Then, check that the trace of B is the trace of B^T. That the trace of a matrix and its transpose are always the same is a theorem of advanced linear algebra, so we content ourselves with this example for now.

The trace has a number of interesting properties including the following:

11.161 Invariance of the Trace

The trace remains invariant among matrices that are similar.

Similar matrices can be thought of as matrices that represent the same linear transformation viewed from different perspectives.[13] Now we are equipped to verify the list of equalities of products of matrices stated in chapter 3.

Recall from earlier chapters that H is the Hadamard operator defined as

[13] Unfortunately, we won’t be able to discuss the idea of *similar matrices* in this exposition, but the name should give some indication of the idea. Two matrices are similar if they are in some sense the same. We can make this idea more precise when we understand the notion of a basis, and what it means to change a basis. The idea is that two matrices are similar iff they differ only by a *change of basis*. This invariance is useful for distinguishing classes of quantum operators.

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

X is the Pauli X operator σ_x, also known as the NOT (or "(qu)bit flip") operator defined as

$$X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Z is the Pauli Z operator σ_z defined as

$$Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Y is the Pauli Y operator σ_y defined as

$$Y := \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

and I is the identity operator, which we'll take to be two-dimensional, so

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

Recall also that for any matrix A, $A^{\dagger}$ denotes the conjugate transpose of A.

11.162 **Exercise**

Verify the following list of equalities of matrices:

- $HXH = Z$
- $HZH = X$
- $HYH = -Y$
- $H^{\dagger} = H$

So, we can say that the Hadamard operator H is *unitary,*[14] since its inverse is its conjugate transpose,

$$H^{\;\;1} = H^{\dagger}$$

[14] We will discuss unitary operators later on in this text.

Matrix Exponentiation

Let us now discuss the exponentiation of matrices. To do this, first we discuss powers of matrices. We can apply the same matrix successively to a vector, as in

$$X(X|\psi\rangle) = X^2|\psi\rangle$$

The notation on the right hand side of this equation suggests we are applying the square of the X operator (matrix) to the vector. Indeed, powers of linear operators (matrices) are defined in this way. In general, the notation A^k for any operator A and any positive integer k means k successive applications of A.

This leads naturally to the notion of the *exponential* of a matrix, denoted $\exp A$ or e^A. The way this is defined is through the Taylor series of e^x, namely

$$e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n.$$

In the same way, we use this to *define* the exponential of a matrix

$$\exp A \equiv e^A := \sum_{n=0}^{\infty} \frac{1}{n!} A^n \tag{11.163}$$

We define $A^0 = I$, the identity matrix, for any matrix A. Although we will not prove it here, it can be shown that the infinite sum in (11.163) converges for *any* matrix A and thus is well-defined.

11.164 **Exercise** Please verify the following.

- $X^2 = I$
- $Y^2 = I$
- $Z^2 = I$
- $H^2 = I$

11.165 **Exercise**

Verify that for any operator A such that $A^2 = I$, the following identity holds:

$$e^{i\theta A} = \cos(\theta) I + i \sin(\theta) A \tag{11.166}$$

For this exercise, it will be useful to recall the Taylor series for cosine and sine. Use this expression for the Pauli matrices X, Y and Z.

11.5 *The Outer Product and the Tensor Product*

The Outer Product as a Way of Building Matrices

Now we'll demonstrate an operation that builds a matrix from two vectors.

Consider the vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}^T = \begin{pmatrix} 1 & 0 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

As discussed earlier, it is perfectly sensible to compute the product

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \end{pmatrix} \tag{11.167}$$

since it is the product of a 2×1 matrix with a 1×2 matrix, i.e., a 2×2 matrix.

11.168 **Exercise** Practice your matrix computation skills and compute the above matrix product.

If you completed the above exercise, you now know that

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 & 1 \cdot 0 \\ 0 \cdot 1 & 0 \cdot 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{11.169}$$

We call the resulting matrix the *outer product* of the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ with itself. We'd like to emphasize that this is just fancy terminology! The outer product is simply the matrix product of two vectors. We can think of the first vector as a 1×2 matrix and the second vector as a 2×1 matrix, remembering that the the product of an $m \times n$ matrix and an $n \times p$ matrix is an $m \times p$ matrix. So, in this case, the product of a 1×2 matrix and a 2×1 matrix yields a 2×2 matrix.

In Dirac notation, we can express the outer product above as $|0\rangle\langle 0|$. Likewise, we may construct the outer products $|0\rangle\langle 1|$, $|1\rangle\langle 0|$, and $|1\rangle\langle 1|$. We leave it to you to compute these in the following exercise:

11.170 **Exercise** Find the outer products $|0\rangle\langle 1|$, $|1\rangle\langle 0|$, and $|1\rangle\langle 1|$.

To help you out, the first of these is:

$$|0\rangle\langle 1| := \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 0 & 1 \cdot 1 \\ 0 \cdot 0 & 0 \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \tag{11.171}$$

Hopefully, you've completed the rest!

With the computation we completed for you above and recalling how it is that we add two matrices, you can now confirm a result written in chapter 3:

$$X := |0\rangle\langle 1| + |1\rangle\langle 0| = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{11.172}$$

The outer product of two vectors is a specific case of the more general concept of a *tensor product*, as we will see now. Before we move on to the tensor product, we would like to discuss how the outer product relates to Dirac notation. Recall that the inner product $\langle u|v\rangle$ of two vectors u and v is in fact expressible as $u^\dagger v$, where $u^\dagger$ denotes the conjugate transpose of u. It should not be too much of a surprise then that the outer product $|u\rangle\langle v|$ is expressible as $uv^\dagger$!

11.173 **Exercise** Check that all of the above computations of outer products $|u\rangle\langle v|$ for vectors u and v could have been thought of as computations of $uv^\dagger$.

In summary:

11.174 Inner and Outer Product and Their Relationships with the Conjugate Transpose

For any two vectors u and v,

$$\langle u|v\rangle = u^\dagger v$$

and

$$|u\rangle\langle v| = uv^\dagger$$

The Tensor Product

Before introducing the operation of a tensor product, let's discuss some terminology for talking about tensor products. As we have seen earlier, a scalar is simply a number.

We can refer to scalars as *0-tensors*, meaning *a tensor of order 0.* Please note that we discourage the use of the word "rank" when referring to the order of tensors, as rank is a term reserved for another term in linear algebra. We can refer to a vector as a 1-tensor. Similarly, we can refer to a matrix as a 2-tensor. A 3-tensor would in fact be a rectangular prism of numbers! Beyond the 3-tensor, we no longer have the ability to give a geometric interpretation. However, many applications in the real world call for tensors of higher order — sometimes in the thousands or millions, as in the case of neural networks. We summarize this terminology in Figure 11.15.

11.175 **Exercise** Why is it appropriate to say that the table in Figure 11.15 is a 2-tensor?

Let us now return to our discussion of the tensor product as a generalization of the outer product. The outer product is the product of two 1-tensors, which produces a matrix (a 2-tensor). However, what happens if we wish to take the tensor product of two tensors of any arbitrary order? We can generalize the outer product of two 1-tensors to the tensor product of any two tensors, A and B, of arbitrary order, like so:

$$A \otimes B$$

To see what we mean by this, consider the two vectors $u := \begin{pmatrix} a \\ b \end{pmatrix}$ and $v := \begin{pmatrix} c & d & e \end{pmatrix}$. Their *tensor product* is the matrix

$$u \otimes v = \begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c & d & e \end{pmatrix} := \begin{pmatrix} a \cdot c & a \cdot d & a \cdot e \\ b \cdot c & b \cdot d & b \cdot e \end{pmatrix} \qquad (11.176)$$

This might remind you of the previously defined outer product of two vectors, and it should, because the tensor product of the two vectors here *is* the outer product.

Let's see the tensor product of two column vectors:

Tensors	**Terminology**	**Example**
0-tensor	scalar	3
1-tensor	vector	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
2-tensor	matrix	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

Figure 11.15: Tensor terminology

$$\begin{pmatrix} r \\ s \\ t \end{pmatrix} \otimes \begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} r \cdot x \\ r \cdot y \\ s \cdot x \\ s \cdot y \\ t \cdot x \\ t \cdot y \end{pmatrix} \tag{11.177}$$

You may have read earlier in the book that we sometimes write $|00\rangle$ to denote the vector $\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$, or $|11\rangle$ to denote the vector $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$. We'd like to point out now that the notation $|00\rangle$ is shorthand for $|0\rangle \otimes |0\rangle$, which is

$$|00\rangle := |0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 \\ 1 \cdot 0 \\ 0 \cdot 1 \\ 0 \cdot 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{11.178}$$

It should be noted that the tensor product cares about whether the vectors are column vectors or row vectors, as we can see from contrasting the previous two examples of tensor products. A quick perusal of the previous examples should convince you that the tensor product of a column vector with a row vector yields a matrix, whereas the tensor product of two column vectors yields another column vector.

11.179 **Exercise** What do you think the tensor product of two row vectors yields?

If you thought about the exercise above, you're hopefully convinced that the tensor product of two row vectors is another row vector.

11.180 Size of tensor product of matrices

The tensor product of an $(a \times b)$ matrix with a $(c \times d)$ matrix is an $(a \cdot c) \times (b \cdot d)$ matrix.

11.181 **Exercise** Check that the aforementioned formula for the dimension of the tensor product of an $a \times b$ matrix with a $c \times d$ matrix specializes, as examples, to the following cases:

- the tensor product of two row vectors
- the tensor product of two column vectors

Note that a row vector can be thought of as matrix whose dimensions are $1 \times m$ *and a column vector can be thought of as a matrix whose dimensions are* $n \times 1$.

Now, we invite you to check that $|11\rangle$ is in fact $|1\rangle \otimes |1\rangle$, as claimed!

11.182 **Exercise** Check that $|11\rangle := |1\rangle \otimes |1\rangle$ is in fact the vector $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$.

Then, check that the intermediate vectors are what they're supposed to be, i.e., $|01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ and $|10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$.

We won't dwell on the idea of the tensor product now. We'd just like to introduce it and get you thinking about it. We'll revisit this idea once we've established the formal definition of vector spaces and linear transformations between them. At that point, we'll realize the tensor product of two vectors is actually the tensor product of the two linear transformations they represent. It will also become clear at that point why it is that the dimension of the tensor product works the way it does.

11.6 *Set Theory*

The Basics of Set Theory

Now we'll spend a bit of time developing some of the prerequisite set theory necessary to continue with this chapter. In particular, we'll need to have an idea of what a function is (this requires a bit of work!) and then we can ascertain what it means for a function to be invertible. Invertible functions are of utmost importance in quantum computing, since the quantum gates we use to construct quantum circuits have to be reversible.

The curious thing is that these gates are represented by matrices, and we'll learn that these matrices are actually just representations of transformations of space, and so are functions themselves. Once we believe that matrices are actually just functions, we have a reasonable notion of what it means for a matrix to be invertible, and thus for a quantum gate to be reversible!

So, don't let the following passages discourage you. A firm mathematical understanding of basic set theory and function theory will guide you well on your way to grasping the underlying ideas of quantum computing.

First, we want to have a notation expressing the notion of the containment of an element in a set. We say that x is an element of a set S, and denote this by $x \in S$. The symbol "$\in$" resembles an "e" for "element," which might help us remember – some people like mnemonics.

We will often abbreviate the phrase "if and only if" with "iff." So, any time you see "iff," think "if and only if." We also often discuss the notion of set containment, or set inclusion when doing mathematics, so let's make sure we have an idea of what this is.

Given two sets A and B, we say that A is a *subset* of B, denoted $A \subset B$, iff for all elements $a \in A$, $a \in B$. In other words, everything in A is also in B.

Sometimes people indicate the inclusion of a set A into a set B by $A \hookrightarrow B$, and we say that "A includes into B." Other ways to say this include "A embeds into B" and "A *injects* into B."

The terminology "A injects into B" is hinting at a property of functions called *injectivity* that we'll investigate soon.

11.183 **Exercise** Check that $\{0\} \subset \{0, 1\}$.

We will find ourselves talking about a few important sets in this chapter, so let us define them. The set of natural numbers,[15] denoted $\mathbb{N}$, is

$$\mathbb{N} := \{0, 1, 2, 3, ...\} \tag{11.184}$$

The set of integers,[16] denoted $\mathbb{Z}$, is

$$\mathbb{Z} := \{..., -3, -2, -1, 0, 1, 2, 3, ...\} \tag{11.185}$$

The set of rational numbers,[17] denoted $\mathbb{Q}$, is

$$\mathbb{Q} := \left\{\frac{p}{q} : p, q \in \mathbb{Z}, q \neq 0\right\} \tag{11.186}$$

The set of real numbers, denoted $\mathbb{R}$, is a bit more complicated to define formally, but it is acceptable to think of a real number as any number that can be approximated to any level of precision by a sequence of rational numbers. Examples of real numbers include

$$0, 1, -1, \frac{3}{4}, \sqrt{2}, e, \pi \tag{11.187}$$

Non-examples of real numbers include

$$i, 1+i, 1-i, \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i \tag{11.188}$$

By including the imaginary number i, we get the complex numbers, denoted $\mathbb{C}$. The set $\mathbb{C}$ is defined as

$$\mathbb{C} := \{a + bi : a, b \in \mathbb{R}\} \tag{11.189}$$

[15] Some people prefer to define the natural numbers excluding 0, like so: $\mathbb{N} = \{1, 2, 3, ...\}$. It makes no difference, really, although computer scientists are inclined to include 0. Some have suggested that the symbol $\mathbb{N}$ be reserved for the set $\{1, 2, 3, ...\}$ and that the symbol $\mathbb{N}_0$ be reserved for the set $\{0, 1, 2, 3, ...\}$. It is a good idea, but it has not yet caught on as far as we know.

[16] You are probably wondering why the set of *integers*, beginning with the letter "i" is denoted with a "z." It turns out that the word "number" is "zahlen" in German, and the Germans are responsible for much of the notation found in number theory and algebra.

[17] Can you figure out why the *rational numbers* are denoted with a "q"? The root of the word "rational" is "ratio," and another word for ratio is "quotient"!

11.190 **Exercise** Check the following set inclusions are actually true

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

You might want to think of each integer x as the rational number $\frac{x}{1}$ to establish the second inclusion in the chain. Then, you might want to think of each real number a as the complex number $a + 0i$ to establish the fourth inclusion.

We can also say things like "B contains A," emphasizing B's containment of A rather than A's containment in B.

We will also begin to use *set-builder notation*[18] to express and describe sets. For example, we can describe the set of all integers whose square is 1 with the notation

$$S := \{x \in \mathbb{Z} : x^2 = 1\} \tag{11.191}$$

We read this as "The set of elements x in the integers (recall, $\mathbb{Z}$ denotes the set of integers) such that the square of x is 1." So, the colon (:) indicates that we should say "such that."

11.192 **Exercise** How many elements are in the set S above? Can you write in set-builder notation the set of all integers whose cube is 1? How many elements are in that set?

11.193 **Exercise** Can you express the set of all complex numbers whose square is -1 in set-builder notation? What about the set of all complex numbers whose fourth power is 1? The set of all complex numbers whose third power is 1? How many are there of each? The answer to the third question is not 1! Remember, you're dealing with complex numbers. This

[18] Note that this notation is also called a *set comprehension*, which is the origin of the term of *list comprehension* in Python and other high-level languages.

is hinting at a classical theorem about the complex numbers known as De Moivre's theorem.[19]

The Cartesian Product

In the development of the definition of vector space, we'll have a desire to understand the notion of a *Cartesian product* of two sets.

The idea is that if we have two sets, say S and T, we'd like to create one unified set from which each of S and T may be identified unambiguously. We call this set the Cartesian product of the sets S and T and denote it by $S \times T$. The notation × was likely chosen to remind us that the number of elements in the Cartesian product of S and T is the number of elements of S *times* the number of elements of T.

Formally, the Cartesian product of the sets S and T is defined

$$S \times T := \{(s,t) : s \in S, t \in T\} \tag{11.194}$$

For example, if we take our two sets to be $S = \{1,2,3\}$ and $T = \{4,5\}$, then, the Cartesian product of S and T is the new set

$$S \times T = \{(s,t) : s \in S, t \in T\} = \{(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)\} \tag{11.195}$$

Notice that $S \times T$ has the promised $2 \times 3 = 6$ elements.

11.196 **Exercise** Think about what the Cartesian product of the set $\mathbb{R}$ with itself is. We call this $\mathbb{R}^2$, i.e., $\mathbb{R}^2 := \mathbb{R} \times \mathbb{R}$. Similarly, we say that $\mathbb{C}^2 := \mathbb{C} \times \mathbb{C}$, and more generally, that

$$\mathbb{R}^n := \underbrace{\mathbb{R} \times \ldots \times \mathbb{R}}_{\text{n times}}$$

and

$$\mathbb{C}^n := \underbrace{\mathbb{C} \times \ldots \times \mathbb{C}}_{\text{n times}}$$

[19] De Moivre's theorem states that the complex numbers z satisfying the equation $z^n = 1$ for a natural number n form the vertices of a regular n-gon in the complex plane. The n complex numbers satisfying the equation $z^n = 1$ are known as the *nth roots of unity*, since 1 is also known as *unity*, and they are the nth roots of 1. For example, the complex numbers satisfying the equation $z^3 = 1$ are the 3rd roots of unity, and form an equilateral triangle in the complex plane. Specifically, they are $1 = e^{0 \cdot \frac{2\pi}{3}}, -\frac{1}{2} + \frac{\sqrt{3}}{2}i = e^{1 \cdot \frac{2\pi}{3}}$, and $-\frac{1}{2} - \frac{\sqrt{3}}{2}i = e^{2 \cdot \frac{2\pi}{3}}$, as you can verify with Euler's formula!

Relations and Functions

Before we venture into the concept of a function, we'll want to discuss relations. The following discussion of relations could accidentally convince you that relations are only interesting in that they are a stepping stone on the path to the definition of a function. This is far from the truth![20]

Maybe you remember writing and seeing things like

$$f(x) = x^2 \tag{11.197}$$

while in school. People would say $f(x)$ is a function. What really is a function though? Intuitively, a function is an *unambiguous* assignment of each element of one set to an element of another set. Of course, we'll want to make this mathematically precise, since it is the backbone of nearly all of the concepts that follow. Actually, we'll learn soon that matrices themselves are functions!

Let's look at some examples and non-examples of functions to get an idea of what they are.

Consider the sets $X := \{1, 2\}$ and $Y := \{3, 4\}$. We can assign each element of X to an element of Y like so

$$1 \mapsto 3$$

$$2 \mapsto 4$$

Introducing notation, we denote the assignment described above by the letter f (for "function") and write

$$f : X \to Y$$

to indicate that f assigns elements of X to elements of Y. We sometimes say

"f maps X to Y"

as well. We refer to the set X as the *domain* of the function f and to the set Y as the *codomain* (sometimes called the "range") of f.

[20] Curious readers should investigate category theory, where the *category* of sets and relations, denoted **Rel**, is an interesting mathematical object in its own right and enjoys an intimate connection with the category of finite-dimensional Hilbert spaces, denoted **Hilb**. For example, every relation $R \subset X \times Y$ of finite sets is naturally associated to a matrix R_{ij}, where $R_{ij} = 1$ iff $(x_i, y_j) \in R$ and is 0 otherwise. Viewing these matrices as having coefficients in the complex numbers $\mathbb{C}$ leads to an interpretation of a relation as a linear transformation between finite-dimensional Hilbert spaces. Category theory offers an enticing framework for computing and other sciences. To learn more about category theory as a unifying language for all of mathematics and science (and even understanding language!), check out Tai-Danae Bradley's blog *Math3ma* [45]. More advanced readers might enjoy the n-Category Café.

We can concisely describe the assignment above with the notation

$$f(1) = 3 \qquad \text{and} \qquad f(2) = 4 \tag{11.198}$$

and say that

"f assigns (maps) the element 1 to the element 3"

and that

"f assigns (maps) the element 2 to the element 4."

Now, for a non-example of a function. Let X and Y be the sets as before, but consider instead the assignment, denoted by the letter f, described by

$$1 \mapsto 3$$

$$1 \mapsto 4$$

$$2 \mapsto 3$$

What is strange is that 1 is now being assigned not only to 3, but also to 4. So, the rule of assignment given by f is ambiguous. It is exactly this ambiguity that we seek to preclude in our formal definition of a function.

We won't give the formal definition of a function just yet though. We would like to emphasize that a function is a special case of a more general phenomenon known as a *relation*. This defers our discussion to that of relations.

More precisely, we would like to define the notion of a relation between two sets. The definition of a relation is quite brief and might surprise you:

11.199 Definition ***Definition of a relation***

A *relation* on two sets X and Y is a subset of their Cartesian product.

That's all? Yes, that's what a relation is. Let's see a few examples of relations to get an idea of what they are.

Consider the subset

$$\mathscr{R} := \{(x, x^2) : x \in \mathbb{R}\} \subset \mathbb{R} \times \mathbb{R} = \mathbb{R}^2 \tag{11.200}$$

By definition, $\mathscr{R}$ is a relation, since it is a subset of the Cartesian product $\mathbb{R} \times \mathbb{R}$ of the set of real numbers $\mathbb{R}$ with itself. But what *is* it?

The following are examples of points which are elements of $\mathscr{R}$, as you should check

$$(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (\sqrt{2}, 2), (\pi, \pi^2), \ldots \tag{11.201}$$

See what's going on?

These are not elements of $\mathscr{R}$

$$(0,1), (1,2), (e,\pi), (7,89), (\sqrt{2},\sqrt{3}), \left(\frac{1}{2},\frac{1}{3}\right)\ldots \tag{11.202}$$

Notice that we say that a point is *in* the relation because we're indicating membership in the set defined by the relation.

11.203 **Exercise** Plot the points of $\mathbb{R}^2$ in the relation $\mathscr{R}$.

If you tried the above exercise, you see now that the set of points in the relation $\mathscr{R}$ is a parabola! What is interesting about $\mathscr{R}$ is that it is even better than a relation – it's a function! In fact, you might recognize it as being described compactly by the equation $f(x) = x^2$. However, the equation $f(x) = x^2$ hides so much of the character of the function f that it's virtually useless to say.

To see what we mean by this, first realize that to say $f(x) = x^2$ gives no indication of the domain or codomain of f. For all we know, f could be mapping from the set $\mathbb{C}$ to $\{0, 1\}$ or whichever pair of sets you like. In fact, if the domain of $f(x) = x^2$ is taken to be $\mathbb{C}$ and the codomain is taken to be $\{0, 1\}$, f isn't a function at all – we'll see why when we give the formal definition of a function!

11.204 **Exercise** Check that the set $\{(0,0), (1,1), (2,2), (3,3)\} \subset \mathbb{Z}\times\mathbb{Z}$ is a relation. Can you describe this relation using set-builder notation?

The point we're trying to make is that we need to give the domain and codomain when specifying a function, not just the rule of assignment.

A *function* f from a set X to a set Y is a relation on X and Y satisfying a special criterion...

Before we give the special criterion, a comment: Recall that to say that f is a relation on X and Y is to say that f is a subset of the Cartesian product $X \times Y$, so we can reasonably talk about elements of the function (thus, relation) f. This type of terminology may sound odd. You've likely not heard of someone saying "(x, y) is an element of the function f" and are probably more accustomed to the phrase $f(x) = y$. However, to say

$f(x) = y$ is literally to say that (x, y) is in the function (and thus, relation) f.

After much deliberation, the special criterion that the points (x, y) in the relation f must satisfy is

11.205 **Definition** ***Definition of a function***

A relation f on a Cartesian product $X \times Y$ of sets X and Y is a *function* iff f satisfies

$$\text{for all } (x_1, y_1), (x_2, y_2) \in f,$$

$$x_1 = x_2 \implies y_1 = y_2$$

The symbol $\implies$ denotes "implies." In other words, if the first coordinates are equal, then the second coordinates have to be equal! We could also phrase this as

$$\text{for all } x_1, x_2 \in X,$$

$$x_1 = x_2 \implies f(x_1) = f(x_2)$$

Let's revisit the non-example we gave earlier and confirm that it is not a function. Recall that we had defined a "function" $f : \{1, 2\} \to \{3, 4\}$ earlier by the rule of assignment

$$1 \mapsto 3$$

$$1 \mapsto 4$$

$$2 \mapsto 3$$

Let's express this as a relation. f is the relation

$$f = \{(1, 3), (1, 4), (2, 3)\} \subset \{1, 2\} \times \{3, 4\} \tag{11.206}$$

11.207 **Exercise** Use the definition of a function to check that f, as given, is not a function. You should also check that $f : \mathbb{C} \to \{0, 1\}$ defined by $f(x) = x^2$ is not a function, as claimed earlier. Hint: Think of f as a relation first – is it even a relation?

Hopefully, you realized that f above is not a function because $(1, 3)$ and $(1, 4)$ are in f and yet despite the fact that their first coordinates are equal (both equal 1), their second coordinates differ (they're each 3 and 4). This is exactly why f violates the definition of a function.

This example, when analyzed alongside the formal definition of function we've given, hopefully illuminates exactly what type of examples the definition is precluding. To request that if the first coordinates of two points in a relation are equal that then their second coordinates must be equal is essentially to ask that no element in the domain is mapped by f to more than one element in the codomain. In other words, the assignment that describes f should be unambiguous.

The curious reader might wonder why we restrict our focus to functions and seldom mention relations. A quick answer is that relations are such a broad class of objects that it's difficult to classify and study them at all. This is not to say that they aren't interesting, just to say that the theory for functions is better understood.

11.208 **Exercise** Consider the function $f : \mathbb{R} \to \mathbb{R}$ described by the rule of assignment

$$f(x) := x^2$$

We are inclined to say that the inverse function of f is the function

$$f^{\ 1}(x) := \sqrt{x}$$

Ah, but if $f^{\ 1}$ is to be a function, it has to have a specified domain and codomain! What are its domain and codomain? Describe the relation that the rule of assignment describing $f^{\ 1}$ defines after finding the domain and codomain. Is it even a function?

This exercise likely confused you, and rightly so! What is confusing is that trying to formulate an inverse for the function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ demands a bit more care. To see what goes wrong, observe that f maps two different elements of the domain $\mathbb{R}$ to the same element of the codomain $\mathbb{R}$:

$$f(1) = 1^2 = 1 \tag{11.209}$$

$$f(-1) = (-1)^2 = 1 \tag{11.210}$$

So, in the process of "inverting" f (which we may think of as "undoing" the squaring operation), we'll have to determine what to do with the number 1 in the codomain. In other words, the question of "Who in the domain did f map to the number 1 in the codomain?" is ambiguous, since both 1 and -1 get mapped to 1 by f. You could answer with either 1 or -1 and you'd be correct! Do you recall now why taking square roots demands that we include "plus or minus"? It's not just dogma!

This phenomenon manifests itself geometrically as the failure of the parabola (the graph of f) to pass the "horizontal line test" as you're invited to explore in the next exercise...

11.211 **Exercise** Draw the graph of the function f and check that it is in fact a parabola. Having done so, see that any horizontal line drawn above the x-axis hits the curve twice. In particular, find the geometric manifestation of the fact that $f(1) = 1$ and $f(-1) = 1$ by drawing a horizontal line at height 1 on your plot.

11.212 **Exercise** You might be wondering now "But wait – then, f isn't even a function because it sends two different things to the same place!" Careful, f *is* a function. The definition of a function disallows the possibility that one thing map to two different things, which is different from what is going on here. Check this using the formal definition of a function!

We'll see that if we impose a condition on the functions that we consider, we'll never encounter this problem while trying to invert them. You can probably already guess that we will require that our functions do not map two different things to the same thing in what follows.

From now on, we'll revert to our usual notation for functions and will likely not need to mention relations. We simply wanted to emphasize the idea that a function is just a specific kind of relation, which instructs its precise mathematical definition.

Important Properties of Functions

We now describe three desirable attributes of functions:[21]

injectivity, surjectivity, bijectivity

We grappled with the idea of a non-injective function earlier while discussing the parabola. Intuitively, an injective function is a function that never maps

[21] The terminology *injective, surjective* and *bijective* can be traced back to *Nicolas Bourbaki*, the pseudonym of a secret society of mainly French mathematicians, including Andre Weil, Jean-Pierre Serre and Alexander Grothendieck, who sought out to reformulate mathematics on an abstract, yet self-contained basis around 1935.

two different things to the same thing. You might know this by the name *one-to-one*.

11.213 Definition ***Injective function***

A function $f : X \to Y$ is *injective* iff

$$\text{for all } x_1, x_2 \in X, \quad f(x_1) = f(x_2) \implies x_1 = x_2.$$

In other words, we ask that f maps two elements to the same element only if those elements are equal. You should check that the function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ earlier is not injective, despite being a function.

11.214 Exercise Prove that if a function f is injective, then for all $x_1, x_2 \in X$, $f(x_1) = f(x_2)$ iff $x_1 = x_2$. Recall the definition of a function!

Sure, $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is not injective. However, if we *restrict* the domain of f to the smaller set $[0, \infty) := \{x \in \mathbb{R} : x \geq 0\}$ and consider instead the function

$$f|_{[0,\infty)} : [0, \infty) \to \mathbb{R} \tag{11.215}$$

(read this "the restriction of f to the set $[0, \infty)$"), we realize that $f|_{[0,\infty)}$ is suddenly injective! Geometrically, this corresponds to chopping off the left side of the parabola and considering only the right side. Graph this to see what we mean. We can express this by saying that we're only considering a single "branch."

Now, let's try to invert this new restricted version of f. Having attempted a previous exercise, you might have struggled to determine the domain and codomain for the inverse function, which we at least know should be something like $\sqrt{x}$. Since the restriction $f|_{[0,\infty)} : [0, \infty) \to \mathbb{R}$ is a function from the domain $[0, \infty)$ to the codomain $\mathbb{R}$, it seems reasonable to request that the inverse function have opposite domain and codomain, i.e., that the domain of the inverse function $\left(f|_{[0,\infty)}\right)^{\ 1}$ should be $\mathbb{R}$ and the codomain should be $[0, \infty)$.

No worries about the codomain: the inverse $\left(f|_{[0,\infty)}\right)^{\ 1} = \sqrt{x}$ certainly has codomain $[0, \infty)$ since the square root of any real number is either 0 or positive, i.e., is an element of the set $[0, \infty)$, as you should check. However, we have an interesting dilemma arising from the issue of what the domain should be.

As you may have already noticed, we cannot take the domain of

$$\left(f|_{[0,\infty)}\right)^{-1} = \sqrt{x} \tag{11.216}$$

to be *all* of $\mathbb{R}$. If we did, we would have to declare how it is that $\left(f|_{[0,\infty)}\right)^{-1}$ operates on numbers like -1... In other words, we would have to take the square root of -1 (and all of the other negative numbers!), which would take us into the imaginary domain.

Since there is no *real* number whose square is -1, we'll have to restrict the domain of the inverse function $\left(f|_{[0,\infty)}\right)^{-1} = \sqrt{x}$ just as well, or else the inverse won't be a function as per our previous definition! Specifically, we remedy our problem of inverting $\left(f|_{[0,\infty)}\right)$ by restricting $\left(f|_{[0,\infty)}\right)^{-1} = \sqrt{x}$ only to the non-negative numbers, i.e., $[0, \infty)$.

In other words, we needed for the function to map *onto* every element of the codomain. In this particular example, we see that the function $f(x) = x^2$ and even its restriction $f|_{[0,\infty)}$ to $[0, \infty)$ does not "hit" everything on the other side. For instance, it misses the number -1, since there is no real number whose square is -1. You can see this geometrically by recognizing that the parabola, and even its right-hand branch, do not ever encroach into negative territory.

We now state a definition clarifying all of this discussion and giving a precise mathematical meaning to the word *onto*.

11.217 **Definition** ***Surjective function***

A function $f : X \to Y$ is *surjective* iff for each element $y \in Y$ there exists an element $x \in X$ such that $f(x) = y$.

Colloquially, we could say that every element in Y is "hit" by some element of X.

Now, the punchline. We set out to make the idea of a function precise and then to give attributes of a function, but we never stated explicitly why we care about these attributes. This next definition is truly a theorem, but we state it as a definition:

11.218 **Definition** ***Bijective function***

A function $f : X \to Y$ is *bijective*, equivalently, *invertible by a function* $f^{-1} : Y \to X$, iff f is injective and surjective.

So, now we know that a function is invertible *by another function* precisely when f is bijective. Invertible functions play an important role in defining what it means for two mathematical objects to be the same. For example, for most purposes, it is often sufficient to consider two sets to be the same iff they have the same number of elements. One way of expressing that two sets have the same number of elements is by stating that there exists an invertible function between them (think about why!). If the sets that we're interested in happen to have a bit more structure and we'd like to assign a meaning to the statement that they're the same, it seems only natural to request that there be an invertible function between them *that preserves their structure*, i.e., it is not sufficient only to have a bijective map between the sets. We'll see later that this idea of a *structure-preserving map* manifests itself in linear algebra as an *invertible linear transformation*.

A few comments: If f is not injective, it is fair to restrict the domain of the function to impose injectivity on it, as we did above with the function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$. But then the function might not be surjective, so we have to invert the function f *only on its image*, where by the *image of f* we mean

$$f(X) := \{f(x) : x \in X\}, \tag{11.219}$$

i.e., the elements $f(x) \in Y$ for some $x \in X$. If we don't invert the function only on its image, the inverse we desire might not end up being a function, as was the case for the squaring function and its presumed inverse, the square root!

11.220 **Exercise** You should check that the image of the squaring function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is in fact $[0, \infty)$.

For a moment, let's recall how it is that we compose two functions.

If you have two functions, say

$$f : \mathbb{R} \to \mathbb{R} \quad \text{defined via} \quad f(x) = x^2 \tag{11.221}$$

and

$$g : \mathbb{R} \to \mathbb{R} \quad \text{defined via} \quad g(x) = x + 1, \tag{11.222}$$

it's natural to ask what their *composition* is, i.e., the result of applying them iteratively.

11.223 Definition *The composition of two functions*

In general, we define the composition of two functions

$$f : X \to Y \qquad \text{and} \qquad g : Y \to Z$$

to be the function $(g \circ f) : X \to Z$ defined via $g(f(x))$.

So, you apply g first, then f. In the case we have above, the composition of $f(x) = x^2$ and $g(x) = x + 1$ is the function

$$(f \circ g)(x) := f(g(x)) = f(x+1) = (x+1)^2 \tag{11.224}$$

11.225 Exercise Check to see that $g \circ f$ is not the same function and is in fact

$$(g \circ f)(x) = x^2 + 1$$

In general, it's quite rare that $f \circ g = g \circ f$ for two functions f and g.

For any set X, there is a special function called *the identity function*, denoted I_X such that for all $x \in X$, $I_X(x) = x$. Given a function $f : X \to Y$, it's natural to ask if there is another function $f^{\ 1} : Y \to X$ such that the following two properties hold: (1) $f^{\ 1} \circ f : X \to Y \to X$ is equal to the identity function I_X on X and (2) $f \circ f^{\ 1} : Y \to X \to Y$ is the identity function I_Y on Y. If we can find such a function $f^{\ 1}$, we say that we have found an inverse function for the function f.

Now, we have a more refined notion of an invertible function:

11.226 Characterization of invertible functions

A function $f : X \to Y$ is invertible iff there exists a function $f^{\ 1} : Y \to X$ such that

$$f^{\ 1} \circ f = I_X \tag{11.227}$$

and

$$f \circ f^{\ 1} = I_Y \tag{11.228}$$

That is, f is invertible iff we can find a function $f^{\ 1}$ whose composition with f in either order yields the identity function! This idea will manifest itself later in our discussion of matrices, when we discuss what it means for a matrix to be invertible.

11.7 *The Definition of a Linear Transformation*

You might have wondered since the beginning of this chapter why it is that we call linear algebra "*linear* algebra."

So far, it's likely not clear at all, since we have not discussed lines or anything obviously linear in any sense. Interestingly, the matrices that have occupied our previous discussion are in fact what we refer to as *linear* transformations. This demands some explanation.

So, what is a linear transformation then? The next (non)example might startle you...

Consider the function (transformation) T from $\mathbb{R} \to \mathbb{R}$ described by

$$x \mapsto x + 1 \tag{11.229}$$

In words, T is the transformation of 1-dimensional space that maps a vector (a number, in this case) to that vector plus one. For example, $T(0) = 0+1 = 1$ and $T(1) = 1 + 1 = 2$. Those familiar with function notation might describe the transformation T as

$$T : \mathbb{R} \to \mathbb{R} \tag{11.230}$$

$$T(x) = x + 1 \tag{11.231}$$

and draw its graph as in Figure 11.16

Well, you might say, "Then T is obviously linear – it's a line!"

Ah, but it's not actually! In fact, the relation above is *affine*, not linear.[22] The reason why this example goes against our intuition is because we have been trained to analyze the graph of a function. Typically, the information encoded by the graph of a function guides our intuition. However, in this case, the graph of T is the set of points

$$\{(x, T(x)) : x \in \mathbb{R}\} \tag{11.232}$$

which does indeed resemble a line when plotted. However, we are not asking that the graph of the function be a line – we're asking that the function

[22] An *affine* transformation is a transformation of the form $Ax := Tx + b$, where T is a linear transformation and b is a vector. In other words, an affine transformation is the result of a linear transformation (i.e., a composition of rotations and dilations) and then a translation. In fact, the reason the transformation $T(x) := x + 1$ is not linear is because it translates by 1! It's a good exercise to determine exactly what A and b are in the definition of Tf and to realize that an affine transformation is linear exactly when the translation vector b is zero.

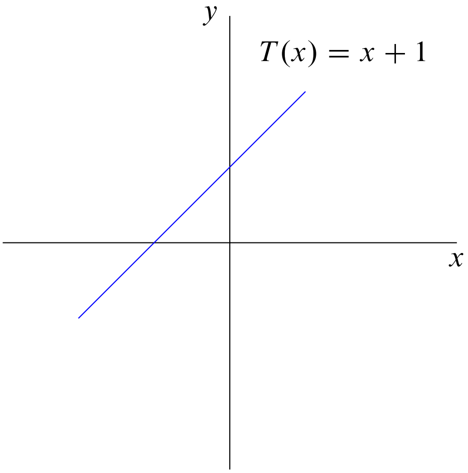


Figure 11.16: The graph of $T(x) = x + 1$

itself be *linear*! You might want to revisit this example after learning the definition of a vector space to see if you can find more reason for why this *should not* be included in our set of linear functions. Try to wrap your head around this for a moment (or three).

Here is the definition of linear that we will take, at least for the case of a transformation that maps from $\mathbb{R}$ to $\mathbb{R}$:

11.233 **Definition** ***The definition of a linear transformation***

A transformation $T : \mathbb{R} \to \mathbb{R}$ (interchangeably, function) is *linear* iff

1. for all $x, y \in \mathbb{R}$, $T(x + y) = T(x) + T(y)$
2. for all $a \in \mathbb{R}$ and for all $x \in \mathbb{R}$, $T(a \cdot x) = a \cdot T(x)$

So, we're requiring that applying T to a sum is the same as applying T to each of the summands and then summing, and that applying T to a scalar multiple is the same as applying T to the multiplicand and then multiplying the result by the scalar. In other words, we require that T preserve addition and scalar multiplication. Often people will say things like *structure-preserving map*. By structure, they mean algebraic structure, i.e., the way things are added and multiplied.

Let's see why the previously defined map $T(x) = x + 1$ is not linear. If T were linear, it would be the case that for all x and y in $\mathbb{R}$, $T(x + y) = T(x) + T(y)$. We see, however, that this is not true:

$$T(x + y) := (x + y) + 1 \tag{11.234}$$

and

$$T(x) + T(y) := (x + 1) + (y + 1) = (x + y) + 2 \neq (x + y) + 1 \tag{11.235}$$

So, $T(x + y) \neq T(x) + T(y)$ in general, and so T does not satisfy our definition of linear!

11.236 **Exercise** Check that T also violates the second property specified in the definition of linear with a specific choice of a and x in $\mathbb{R}$. So, T is hopelessly not linear!

We still claim that the matrices shown earlier are in fact linear transformations. However, to explain this correctly, we'll need to understand in what sense a matrix is a transformation from a more careful and mathematical perspective. In particular, we'll need to turn our attention toward exactly which spaces the matrices are transforming. These spaces are known as *vector spaces*.

11.8 *How to Build a Vector Space From Scratch*

The name *vector space* likely brings to mind a space of vectors. That is pretty much what a vector space is – a space where a bunch of vectors live. We want this space to facilitate all of the usual operations we perform on vectors, such as addition of vectors and scalar multiplication of vectors.

We'll state the precise definition of a vector space and then unpack the terminology.

11.237 **Definition** ***Vector space***

A *vector space* V over a field $\mathbb{F}$ is an abelian group V equipped with an action of the field $\mathbb{F}$ on V.

There are a lot of unfamiliar terms in this definition which we will explain step by step.

First of all, what is a field? It turns out that it will be best to define an abelian group first, since we'll see that any field is an augmentation of an abelian group. Later on in this section we will define the word *action*.

Groups

Recall that the notation

$$x \in G \tag{11.238}$$

denotes "x is an element of G." We also write things like

$$x, y, z \in G \tag{11.239}$$

meaning "x, y and z are elements of G."

We'll state the definition of a group precisely now. After its statement, we'll delve into what all of this really means.

A group $(G, \star)$ is a set G satisfying the following properties:

- Closure: There exists a function

  $$\star : G \times G \to G$$

  which we call the *binary operation* of G.

  Instead of writing the application of the operation $\star$ to a pair of elements g_1, g_2 each in G as $\star(g_1, g_2)$, as we usually do when describing the application of a function to a pair of elements, we write $g_1 \star g_2$ to be succinct.

  For now, it's advisable to think of $\star$ as being a familiar operation like addition or multiplication. For example, it would be weird, but correct, to say that addition is a binary operation that accepts two numbers as the input and outputs a new number. We could write such a thing like this, where we replace the name $\star$ for the binary operation with the symbol $+$

  $$+(x, y) := x + y.$$

  This idea shouldn't be too unfamiliar to computer scientists and programmers – it's like defining a function or method!

- Associativity: For any triplet x, y, z of elements of G,

$$(x \star y) \star z = x \star (y \star z)$$

This is a reasonable assumption that we often make in mathematics. You should be aware of the fact that not every binary operation is associative!

11.240 **Exercise** Can you think of an example of a binary operation that isn't associative? Think of the usual subtraction of two numbers, e.g., $3 - 2 = 1$. Interestingly, this simple operation is not associative! To see this, consider the ambiguous difference $1 - 2 - 3$. Is it $(1 - 2) - 3$ or is it $1 - (2 - 3)$? Check to see that it *does* matter!

- Identity: There exists an element $e \in G$, called the *identity element* such that for any element $x \in G$,

$$x \star e = e \star x = x$$

Again, it's advisable to think of a familiar scenario: you're likely already familiar with the number 0 in the integers equipped with the operation of addition.

11.241 **Exercise** You should check that 0 has the identity property for any set of numbers equipped with addition, i.e., adding any number x and zero in either order yields the number x.

- Inverse: For each element $x \in G$, there exists an inverse element $x^{\ 1} \in G$, with the property that

$$x \star x^{\ 1} = x^{\ 1} \star x = e$$

Inverses in any set of numbers equipped with the operation of addition are the "negatives." To see what we mean, try to find a number such that when you add it to 1 on either side, you get 0. A moment's thought makes us realize that the number we're looking for is -1! So, we would say that -1 is the inverse of the number 1 with respect to addition.

11.242 Definition *The definition of a group*

A *group* $(G, \star)$ is a set G satisfying the following properties:

- Closure: There exists a function
$$\star : G \times G \to G,$$
which we call the *binary operation* of G.
- Associativity: For any triplet x, y, z of elements of G,
$$(x \star y) \star z = x \star (y \star z)$$
- Identity: There exists an element $e \in G$, called the *identity element of* G, such that for any element $x \in G$,
$$x \star e = e \star x = x$$
- Inverse: For each element $x \in G$, there exists an *inverse* element $x^{\ 1} \in G$, with the property that
$$x \star x^{\ 1} = x^{\ 1} \star x = e$$

Now, let's re*group* (sorry, intended...) and think about what this definition really says.

The idea of a group is simple. We begin with a set of things. Then, we define a single binary operation on that set of things, i.e., an operation that accepts two inputs from your set of things and produces another thing in your set. This property is also known as *closure of the set under the operation.*

Next, we require that the operation be *associative*. So, if we have three things to operate on, we may operate on any two first, and then the third (you can't operate on all three at once because you started with a binary operation!).

Afterward, we require that there be a special element in our set called the *identity element.* If we apply the binary operation to this special element and any other element, we get that element back.

Finally, we ask that each element in our set have an *inverse* element, also in the set, so that when we apply the binary operation to that element and its inverse element (in either way), we get the identity element.

To make the group *abelian,*[23] we ask that the operation be *commutative*, in the sense that the operation yields the same result regardless of the order.

[23] The name "abelian" honors the mathematician Niels Henrik Abel.

11.243 **Definition** ***Abelian group***

A group $(G, \star)$ is called *abelian* iff $\star$ is a commutative operation, i.e.,

$$\text{for all } x, y \in G,\ x \star y = y \star x$$

This definition is fairly abstract, so let's pin it down with an example: the integers, denoted $\mathbb{Z}$. This example will hopefully solidify a few of the remarks made during the course of the definition. The integers $\mathbb{Z}$ are the numbers

$$\mathbb{Z} := \{..., -3, -2, -1, 0, 2, 3, ...\} \tag{11.244}$$

There is a natural choice of binary operation $\star : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. It's addition! We define $\star(x, y) := x + y$ (we saw this earlier when explaining the meaning of a binary operation). In other words, $\star$ is just the binary operation of usual integer addition. Then the claim is that this operation does in fact satisfy all of the other properties listed.

Well, associativity is certainly satisfied, since addition of integers is associative (although some might take issue with this, since in some sense associativity is satisfied by fiat). For example, if asked to find the sum $1 + 2 + 3$, we know that we may either compute $1 + 2$ first, then add 3, or compute $2 + 3$ first, then add 1. In symbols, $(1 + 2) + 3 = 1 + (2 + 3)$, and in fact, this is true for all triplets of integers.

The next question is as to what the identity element is here. The answer is 0 because adding 0 and any other integer yields that integer.

Finally, we ask that every integer have an inverse, and this is true, since any integer a has inverse $-a$. Check that for any integer $a \in \mathbb{Z}$,

$$a + (-a) = (-a) + a = 0 \tag{11.245}$$

11.246 **Exercise** Try to figure out why the set of natural numbers

$$\mathbb{N} := \{0, 1, 2, 3, ...\} \tag{11.247}$$

is **not** a group with the operation of usual addition of natural numbers, e.g., $1 + 2 = 3$.

In any case, it turns out that the rational numbers ($\mathbb{Q}$), the real numbers ($\mathbb{R}$) and the complex numbers ($\mathbb{C}$) each form abelian groups under usual addition of numbers.

Now, we have the following observation:

11.248 Key idea

Vectors of numbers form an abelian group!

To see what we mean, let's revisit our definition of the addition of two vectors. Let's focus on what happens in two dimensions. In fact, let's check to see that the set of all vectors with two complex components, denoted

$$V = \mathbb{C}^2 := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} : x, y \in \mathbb{C} \right\} \tag{11.249}$$

does in fact form an abelian group under usual vector addition.

First, we need to verify that this set is closed with respect to the operation of vector addition, i.e., if we take two elements of this set and we add them, that we do in fact get another element of this set (closure). So, let $\begin{pmatrix} x & y \end{pmatrix}^T$ and $\begin{pmatrix} z & w \end{pmatrix}^T$ be any two elements in the set V. We need to check that their sum

$$\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} z \\ w \end{pmatrix} = \begin{pmatrix} x + z \\ y + w \end{pmatrix} \tag{11.250}$$

is in fact another element of the set V.

Since each of x, y, z, w are complex numbers, the sums $x + z$ and $y + w$ are in fact complex numbers too. So, the resulting vector is an element of V after all. Then, V is closed with respect to vector addition.

The next question is as to whether this addition is associative. You're invited to check this and we'll set it up for you. Let

$$\begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} c \\ d \end{pmatrix}, \begin{pmatrix} e \\ f \end{pmatrix} \tag{11.251}$$

be elements of V.

11.252 Exercise Check the associativity of vector addition, i.e., that

$$\left(\begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} c \\ d \end{pmatrix} \right) + \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix} + \left(\begin{pmatrix} c \\ d \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} \right)$$

So now we need an identity element. We can take the vector that has only zero entries

$$e = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{11.253}$$

11.254 Exercise Check that the vector e is in fact an identity element for the set V, i.e., adding e to any vector on either side yields the same vector.

We then ask whether any given vector in V has an inverse element in V with respect to addition. This isn't too bad. If we're given a vector $v := \begin{pmatrix} x & y \end{pmatrix}^T$ in V, the inverse element is $-v := \begin{pmatrix} -x & -y \end{pmatrix}^T$, as you might have expected.

11.255 Exercise Check to see that it is in fact the case that

$$v + (-v) = (-v) + v = e$$

(Remember that "e" is the all-zero vector defined above!)

Now, what about the *abelian* part? Let us recall that abelian refers to commutativity, so let's see if the group of vectors has commutativity for the operation of addition.

11.256 Exercise You should check that adding vectors in V in either order yields the same result, i.e., that they commute.

$$\begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} c \\ d \end{pmatrix}, \begin{pmatrix} e \\ f \end{pmatrix} \tag{11.257}$$

This is true due to the fact that adding complex numbers in either order yields the same result, as you'll see while completing this exercise.

We have now verified that $\mathbb{C}^2$, the set of all two-dimensional vectors with complex entries is an abelian group!

11.258 **Exercise** Prove that $\mathbb{R}^2$ is an abelian group by similar means. Then, prove that the n-dimensional analogue $\mathbb{R}^n$ is also a group. Last, prove the n-dimensional analogue $\mathbb{C}^n$ of $\mathbb{C}^2$ is also an abelian group with respect to usual vector addition.

A comment about notation: After assuming the group is abelian, it's fair to revert to calling the abstract operation $\star$ by the name $+$. This is because the binary operation for *every* abelian group can be realized as some sort of addition of integers! We won't have time to discuss this idea here, however. The inverse of an element x in the abelian group setting is thus fairly referred to by $-x$.

11.259 **Exercise** To see an example of a non-abelian group, consider the Pauli matrices X, Y, Z from earlier, and the identity matrix $I := I_2$. Check that the set of sixteen matrices

$$P := \{\pm I, \pm iI, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\}$$

equipped with the operation of matrix multiplication is a group by verifying the axioms listed above. We call this group *the Pauli group*. It is non-abelian because the matrix product is not commutative in general. For example, $XY \neq YX$.

You should now be wondering what we mean by *an action of a field* $\mathbb{F}$ *on* V, and in particular, what a field is. We'll save the idea of an action for last.

Fields

Having defined an abelian group, the definition of a field is not terribly far off. A field is an abelian group with an additional operation, which is often called "multiplication," that satisfies some mild hypotheses. Fields are not too abstract actually. You've been working with fields all of your life! Examples include the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, the complex numbers $\mathbb{C}$ and more exotic finite fields, like $\mathbb{Z}/2\mathbb{Z} = \{0, 1\}$ equipped with addition and multiplication modulo-2 (also known as $\mathbb{F}_2$), as we will see.

We state the definition of a field now.

A field $\mathbb{F}$ is an abelian group (so, it's a set satisfying all of the properties of a group defined earlier – review these!), where we denote the commutative binary operation by $+$, satisfying the following properties:

- Closure: There exists a binary operation

$$* : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$$

  additional to $+$, which we usually call *multiplication*.

  Similar to the scenario with groups, we often write the application of $*$ to a pair of elements (x, y) of $\mathbb{F}$ as $x * y$ as opposed to $*(x, y)$. As before, you're advised to think of usual multiplication of numbers as you read through these axioms. For example, the rational numbers $\mathbb{Q}$ (all ratios of integers, i.e., fractions) are closed under usual multiplication.

11.260 Exercise Verify that the rational numbers

$$\mathbb{Q} := \left\{ \frac{a}{b} : a, b \in \mathbb{Z}, b \neq 0 \right\}$$

are in fact closed under usual multiplication. So, let

$$\frac{a}{b}, \frac{c}{d} \in \mathbb{Q}$$

(so, $a, b, c, d \in \mathbb{Z}$) and check that their product

$$\frac{a}{b} * \frac{c}{d} := \frac{a * c}{b * d}$$

is actually another rational number. This amounts to confirming the numerator and denominator of the result are each integers and that the product of two integers is an integer (which might even be considered defining properties of the integers!).

- Associativity: For all x, y, z in $\mathbb{F}$,

$$(x * y) * z = x * (y * z)$$

  This is similar to the axiom for groups, but we'd like it to be true for multiplication.

11.261 Exercise Verify that multiplication of rational numbers is associative. So, take three arbitrary rational numbers, say

$$\frac{a}{b}, \frac{c}{d}, \frac{e}{f} \in \mathbb{Q}$$

and show that

$$\left(\frac{a}{b} * \frac{c}{d}\right) * \frac{e}{f} = \frac{a}{b} * \left(\frac{c}{d} * \frac{e}{f}\right)$$

This will amount to the fact that integer multiplication is associative!

- Commutativity: For all $x, y \in \mathbb{F}$,

  $$x * y = y * x$$

  So, multiplication in either order should yield the same product.
- Distributivity: For all $x, y, z \in \mathbb{F}$

  $$x * (y + z) = x * y + x * z$$

  Also,

  $$(y + z) * x = y * x + z * x$$

  This is a bit subtle. We want the distributivity to "work" on both sides. We won't dwell on it though.[24]
- Identity: There exists an element, denoted by 1, in $\mathbb{F}$ such that for any element x in $\mathbb{F}$,

  $$1 * x = x * 1 = x$$

  Again, similar to the group property. The identity element of a field is often *literally* the number 1.
- Inverse: For each nonzero element $x \in \mathbb{F}$, there exists an element $x^{\ 1}$ in $\mathbb{F}$ such that

  $$x * x^{\ 1} = x^{\ 1} * x = 1$$

  Then, the inverse of an element x in the field setting, where we have bona fide multiplication, is referred to $x^{\ 1}$, which should make us think of $\frac{1}{x}$.

  That each element has a multiplicative inverse is actually the defining property of a field.[25]

[24] Technically, we could request the axiom of distributivity *before* the axiom of commutativity, or even request the axiom of distributivity and *not* the axiom of commutativity. Do you see that if we request the axiom of distributivity and not the axiom of commutativity that we really need both equalities in the axiom of distributivity?

[25] There are other algebraic objects called *commutative rings with unity*, which enjoy all of the properties of fields, except the inverse property! If we also remove the commutativity axiom, we have what is called a *division ring*.

11.262 **Exercise** Figure out why the set of all invertible matrices cannot possibly be a field, but has the potential to be a division ring. In other words, which axiom does the set of all invertible matrices violate in the definition of a field? If you don't know what an invertible matrix is, read on to find out!

11.263 **Exercise** You're encouraged to unpack these definitions and verify that the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$ are all fields! To figure out why $\mathbb{C}$ is a field, you'll need to answer the latter portion of the previous exercise regarding multiplicative inverses of complex numbers.

11.264 **Definition** ***The definition of a field***

A *field* $\mathbb{F}$ is

- An abelian group, where we denote the abelian binary operation by $+$, satisfying the following properties:
- Closure: There exists a binary operation, additional to $+$, which we usually call *multiplication*.

$$* : \mathbb{F} \times \mathbb{F} \to \mathbb{F},$$

- Associativity: For all x, y, z in $\mathbb{F}$,

$$(x * y) * z = x * (y * z)$$

- Commutativity: For all $x, y \in \mathbb{F}$,

$$x * y = y * x$$

- Distributivity: For all $x, y, z \in \mathbb{F}$

$$x * (y + z) = x * y + x * z$$

Also,

$$(y + z) * x = y * x + z * x$$

- Identity: There exists an element, denoted by 1, in $\mathbb{F}$ such that for any element x in $\mathbb{F}$,

$$1 * x = x * 1 = x$$

- Inverse: For each nonzero element $x \in \mathbb{F}$, there exists an element $x^{\ 1}$ in $\mathbb{F}$ such that

$$x * x^{\ 1} = x^{\ 1} * x = 1$$

Here's an interesting example of a field not quite like those explored in the previous exercise.[26]

Consider the set $\mathbb{F}_2 := \{0, 1\}$. We equip $\mathbb{F}_2$ with addition and multiplication modulo-2, as indicated by the addition and multiplication tables

[26] This is called the *Galois* field, abbreviated GF_2. Galois was a startlingly brilliant mathematician who lived for only twenty one years, from 1811 to 1832. During his short life, he essentially founded all of Galois theory and group theory in an effort to determine precisely when the solutions of a polynomial equation could be written down explicitly. To understand the issue he was interested in, try to find the exact solutions to the equation $x^5 - x + 1 = 0$.

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

So, a quick look reveals interesting things like $1 + 1 = 0$ happen in $\mathbb{F}_2$. The addition and multiplication tables clearly indicate that closure of each is satisfied. Let's take for granted that associativity is satisfied – it's not difficult to check, just tedious. We can see from the addition and multiplication tables that the (additive) identity element is 0 and that the (multiplicative) identity element is 1.

11.265 **Exercise** Check the inverse property of each of addition and multiplication for $\mathbb{F}_2$ given the addition and multiplication tables.

The final piece in the "definition of a vector space" puzzle is the definition of an action of a field on an abelian group, which we'll discuss now.

An action of a field on an abelian group is an abstraction of the "scalar multiplication" of vectors by numbers defined earlier in the chapter. Here it is formally:

An action of a field $\mathbb{F}$ on an abelian group V is a function $\cdot : \mathbb{F} \times V \to V$ satisfying some properties.

Before listing the properties, we'd like to mention that we'll not write $\cdot(a, v)$ when expressing the application of the function (action) $\cdot$ to a pair (a, v), with a in $\mathbb{F}$ and v in V. Instead, we'll write $a \cdot v$ to be succinct. it's advisable to think of an action as scalar multiplication of a vector v by a number a.

Here are the defining properties of an action:

- Distributivity I: For all $a \in \mathbb{F}$ and all u, v in V,

$$a \cdot (u + v) = a \cdot u + a \cdot v$$

- Distributivity II: For all $a, b \in \mathbb{F}$ and all v in V,

$$(a + b) \cdot v = a \cdot v + b \cdot v$$

These distributivity axioms are subtle. The addition on the left is happening in the field $\mathbb{F}$, while the addition on the right is happening in the group V!

To see what we mean, try this exercise:

11.266 Exercise Let $a = 2$ and $u = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $v = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Check that $a \cdot (u + v) = a \cdot u + a \cdot v$, i.e.,

$$2 \cdot \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) = 2 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 2 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

- Compatibility of the action with multiplication in the field $\mathbb{F}$: For all $a, b \in \mathbb{F}$ and v in V
$$(ab) \cdot v = a \cdot (b \cdot v)$$
This is also subtle. We're asking that multiplying two elements of the field and then applying the resulting element of the field to a vector be the same as iteratively applying those elements of the field. Try this exercise to get a feel for this idea:

11.267 Exercise Let $a = 2, b = 3$ and $v = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Check that $(ab) \cdot v = a \cdot (b \cdot v)$, i.e.,

$$(2 \cdot 3) \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2 \cdot \left(3 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right)$$

- Identity: For all v in V,
$$1 \cdot v = v$$
This is to say that the (multiplicative) identity element of the field should act on the group V in such a way as to not disturb V.

11.268 Exercise Check that the (multiplicative) identity 1 of the field $\mathbb{C}$ acts on the vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ in the way that is requested in the identity clause, i.e., check that

$$1 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

11.269 **Definition** ***Action of a field on an abelian group***

An *action of a field* $\mathbb{F}$ *on an abelian group* V is a function $\cdot : \mathbb{F} \times V \to V$ satisfying:

- Distributivity I: For all $a \in \mathbb{F}$ and all u, v in V,
$$a \cdot (u + v) = a \cdot u + a \cdot v$$
- Distributivity II: For all $a, b \in \mathbb{F}$ and all v in V,
$$(a + b) \cdot v = a \cdot v + b \cdot v$$
- Compatibility of the action with multiplication in the field $\mathbb{F}$ For all $a, b \in \mathbb{F}$ and v in V:
$$(ab) \cdot v = a \cdot (b \cdot v)$$
- Identity: For all v in V,
$$1 \cdot v = v$$

The Definition of a Vector Space

Congratulations! If you've made it to this point (and you've completed and understood the exercises), you understand the *mathematician's* definition of a vector space V over a field $\mathbb{F}$. You'll likely have to review the definitions above and play around with a few examples to get a solid understanding of them, so here are some examples to play with:

11.270 **Exercise** To solidify your understanding of the definition of a vector space, verify that each of the following are examples of vector spaces:

- $\mathbb{Q}$ over itself (yes, it's a vector space!)
- $\mathbb{R}$ over $\mathbb{R}$
- $\mathbb{C}$ over $\mathbb{C}$
- $\mathbb{Q}^2$ over $\mathbb{Q}$
- $\mathbb{R}^2$ over $\mathbb{R}$
- $\mathbb{C}^2$ over $\mathbb{C}$
- $\mathbb{Q}^n$ over $\mathbb{Q}$
- $\mathbb{R}^n$ over $\mathbb{R}$
- $\mathbb{C}^n$ over $\mathbb{C}$

You're now invited to contrast the succinct sentence "A vector space V over a field $\mathbb{F}$ is an abelian group V equipped with an action of $\mathbb{F}$ on V" with the definition found in most standard texts on linear algebra:

A *vector space* V over a field $\mathbb{F}$ is a set satisfying the following properties:

- Associativity of addition: For all $u, v, w \in V$,
$$(u+v)+w = u+(v+w)$$
- Commutativity of addition: For all $u, v \in V$,
$$u+v = v+u$$
- Identity element of addition: There exists an element $0 \in V$ such that for all $v \in V$,
$$v+0 = v$$
- Inverse element of addition: For every $v \in V$, there exists an element $-v \in V$ such that
$$v+(-v) = 0$$
- Compatibility of scalar multiplication and field multiplication: For any $a, b \in \mathbb{F}$ and any $v \in V$,
$$(ab)v = a(bv)$$
- Identity element of scalar multiplication: For any $v \in V$,
$$1v = v$$
where 1 is the multiplicative identity of the field $\mathbb{F}$
- Distributivity of scalar multiplication with respect to vector addition: For all $a \in \mathbb{F}$ and any $u, v \in V$, $a(u+v) = au + av$
- Distributivity of scalar multiplication with respect to field addition: For all a, b in $\mathbb{F}$ and any $v \in V$, $(a+b)v = av + bv$

Even after requiring the properties above, we would still have to define a field axiomatically as a set $\mathbb{F}$ together with two operations, called addition (denoted $+$) and multiplication (denoted $\cdot$), where by an operation we mean an association of any pair of elements of $\mathbb{F}$ to some other element of $\mathbb{F}$. Thus, the definition of a vector space in this approach still requires additional properties:

- Associativity of addition and multiplication: For all $a, b, c \in \mathbb{F}$,
$$a+(b+c) = (a+b)+c \qquad \text{and} \qquad a\cdot(b\cdot c) = (a\cdot b)\cdot c$$

- Commutativity of addition and multiplication: For all $a, b \in \mathbb{F}$,
$$a + b = b + a \qquad \text{and} \qquad a \cdot b = b \cdot a$$
- Additive and multiplicative identity: There exist two elements 0 and 1 in $\mathbb{F}$ such that for all $a \in \mathbb{F}$,
$$a + 0 = a \qquad \text{and} \qquad a \cdot 1 = a$$
- Additive inverses: For every $a \in \mathbb{F}$, there exists an element $-a$ in $\mathbb{F}$ such that
$$a + (-a) = 0$$
- Multiplicative inverses: For every nonzero element a in $\mathbb{F}$, there exists an element $a\ ^{1}$ in $\mathbb{F}$ such that
$$a \cdot a\ ^{1} = 1$$
- Distributivity of multiplication over addition: For all $a, b, c \in \mathbb{F}$,
$$a \cdot (b + c) = a \cdot b + a \cdot c$$

The choice is up to you! Since many linear algebra textbooks do not wish to cover set theory, they are forced to delineate an exhaustive list of properties to define a vector space. With set theory in hand, we state the definition of a vector space once more to remind you:

11.271 **Definition** ***Vector Space***

A *vector space* V over a field $\mathbb{F}$ is an abelian group V equipped with an action of the field $\mathbb{F}$ on V.

Subspaces

Now that we know what a vector space is (make sure you do!), we can define the notion of a *subspace* of a vector space. It's pretty much what it sounds like. You have a space of vectors and then you consider a subcollection of vectors that also satisfies the properties of a vector space.

11.272 **Definition** ***Subspace of a Vector Space***

A subset $S \subset V$ of a vector space V over a field $\mathbb{F}$ is a *subspace* of V iff S is a vector space over $\mathbb{F}$.

The remarkable thing is that instead of having to verify all of the properties of a vector space for S after having done so for V, the verification of the subspace property of a subcollection of a vector space can be checked in three easy steps:

11.273 The subspace lemma

$S \subset V$ is a subspace of a vector space V over a field $\mathbb{F}$ iff

- Identity: $0 \in S$, where this 0 is the additive identity that comes as part of the data of the vector space V
- Additive closure: For all $u, v \in S$, $u + v \in S$
- Closure under scalar multiplication: For all $a \in \mathbb{F}$, $v \in S$, $a \cdot v \in S$

Let's see an example. Consider the vector space $\mathbb{R}^2$ and the x-axis, which we may think of as the set of points

$$X := \left\{ \begin{pmatrix} x \\ 0 \end{pmatrix} : x \in \mathbb{R} \right\} \subset \mathbb{R}^2 \tag{11.274}$$

We claim that $X \subset \mathbb{R}^2$ is a subspace. We'll apply the subspace lemma to prove this.

First of all, the zero vector is in fact part of the x-axis, so the identity property is satisfied.

Second, it's easy to see geometrically that the sum of two vectors on the x-axis is another vector on the x-axis. However, here is a rigorous proof. Let

$$\begin{pmatrix} x_1 \\ 0 \end{pmatrix}, \begin{pmatrix} x_2 \\ 0 \end{pmatrix} \tag{11.275}$$

each be elements of the x-axis (these are like our u and v in the additive closure clause of the subspace lemma). Then

$$\begin{pmatrix} x_1 \\ 0 \end{pmatrix} + \begin{pmatrix} x_2 \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ 0 + 0 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ 0 \end{pmatrix} \tag{11.276}$$

This is another element of the x-axis.

Lastly, we see that if we take a scalar multiple of a vector on the x-axis that it will again be a vector on the x-axis. We give a rigorous proof.

Let a be in $\mathbb{R}$ and $v := \begin{pmatrix} x \\ 0 \end{pmatrix}$ in X. Then, we need to check that $a \cdot v$ is again in X (here, a and v are like the a in the v in the closure under the scalar multiplication clause in the subspace lemma). To see that this is the case, look at

$$a \cdot v = a \cdot \begin{pmatrix} x \\ 0 \end{pmatrix} = \begin{pmatrix} a \cdot x \\ 0 \end{pmatrix} \tag{11.277}$$

This is another element of the x-axis.

So, it is true that the x-axis is a subspace of $\mathbb{R}^2$.

11.278 **Exercise** You're invited to verify that the y-axis, and in fact, *any line through the origin* in the vector space $\mathbb{R}^2$ over $\mathbb{R}$ is in fact a subspace of $\mathbb{R}^2$. By a line through the origin in $\mathbb{R}^2$, we mean the set of points

$$L = \left\{ \begin{pmatrix} x \\ a \cdot x \end{pmatrix} : x \in \mathbb{R} \right\}$$

for a fixed $a \in \mathbb{R}$. Use the subspace lemma like we did above to make your life easier, or else you'll have to verify all of the properties of a vector space again. Why do we emphasize that the line be "through the origin"? What happens if it doesn't pass through the origin?

We close this section with the definition of a linear transformation between arbitrary vector spaces:

11.279 **Definition** ***Definition of a linear transformation between vector spaces***[a]

A *linear transformation* T between vector spaces V and W, each over a field $\mathbb{F}$, is a function $T : V \to W$ satisfying the following properties:

- For all $x, y \in V$, $T(x + y) = T(x) + T(y)$.
- For all $x \in V$ and all $a \in \mathbb{F}$, $T(a \cdot x) = a \cdot T(x)$.

[a] It should be noted that the addition of x and y in the definition above is occurring in the vector space V, whereas the addition of $T(x)$ and $T(y)$ is occurring in W, and V and W may have different definitions of the operation $+$. A similar remark holds for the scalar multiplication occurring in the second criterion.

11.9 *Span, Linear Independence, Bases and Dimension*

Span

A quick comment about some of the notation that follows: We will often write a column vector like $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ as a transposed row vector like so $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ for simplicity of presentation.

The definition of a vector space provided above is helpful, but it would be nice to have a more concrete description. Let's think about the vector space $\mathbb{R}^2$ over $\mathbb{R}$ for a moment so that we can motivate a new definition.

Let's say we want to make a vector in $\mathbb{R}^2$. In other words, we would like to construct any vector in two-dimensional space. A moment's thought brings forth the realization that we can build any vector in two-dimensional space with just two vectors. Let's make this idea more precise. Visualize the vector

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

in $\mathbb{R}^2$.

Suppose we'd like to get from the origin to the point in space to which this vector points. We can travel along the x-axis 1 step, and then travel upward 1 step. We could also travel upward along the y-axis first, and then rightward 1 step. Here is the algebraic manifestation of this concept:

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} = 1 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

In other words, the vector $\begin{pmatrix} 1 & 1 \end{pmatrix}^T$ is 1 of the vector $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ plus 1 of the vector $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$. So, we can build the vector $\begin{pmatrix} 1 & 1 \end{pmatrix}^T$ using the "building blocks" $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ and $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$. Can we make every vector in $\mathbb{R}^2$ in this way?

11.280 **Exercise** Try to build the vector $\begin{pmatrix} 2 & 3 \end{pmatrix}^T$ using some combination of the building blocks $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ and $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$. Can you make the vector $\begin{pmatrix} \frac{1}{2} & \frac{1}{3} \end{pmatrix}^T$? Nobody said we can't use fractional parts of our building blocks! What about the vector $\begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}^T$?

What we're getting at here is the notion of a *spanning set* of vectors. We say that the vectors $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ and $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$ *span* the vector space $\mathbb{R}^2$ over $\mathbb{R}$.

We do have to be a bit more precise here though. What if we were asked to build the vector $\begin{pmatrix} \frac{1}{2} & \frac{1}{3} \end{pmatrix}^T$ and we weren't allowed to use fractional amounts of our building blocks $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ and $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$? So, imagine we can only use integer multiples of the building blocks. Well, then we can't make the vector $\begin{pmatrix} \frac{1}{2} & \frac{1}{3} \end{pmatrix}^T$, at least not with integer multiples of $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ and $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$. So, we really should say a bit more about in what sense the building blocks span the space. Here is the formal definition and some terminology that will be useful for the discussion which follows:

11.281 **Definition** *Spanning set of vectors*

We say that a set of vectors $v_1, ..., v_m$ *spans* a vector space V over a field $\mathbb{F}$ iff for each vector v in V, there exist elements $a_1, ..., a_m$ of the field $\mathbb{F}$ such that

$$a_1 \cdot v_1 + a_2 \cdot v_2 + ... + a_m \cdot v_m = v$$

Then, we define the *span* of a set of vectors:

11.282 **Definition** *Span of a set of vectors*

The *span* of a set of vectors $v_1, ..., v_m$ over a field $\mathbb{F}$ is defined to be:

$$\text{span}(\{v_1, ..., v_m\}) := \{a_1 v_1 + ... + a_m v_m : a_1, ..., a_m \in \mathbb{F}\}.$$

We often abbreviate the phrase "*span* of a set of vectors $v_1, ..., v_m$ over a field $\mathbb{F}$" to simply "span of $v_1, ..., v_m$" when the field $\mathbb{F}$ is understood. We say that a vector v is in the span of some vectors $v_1, ..., v_m$ iff $v \in \text{span}(\{v_1, ..., v_m\})$. We call $a_1, ..., a_m$ the *coefficients* of the $\mathbb{F}$-*linear combination* of the vectors $v_1, ..., v_m$. Furthermore, we say that the set of vectors spanned by the vectors $v_1, ..., v_m$ (equivalently, the set of all $\mathbb{F}$-linear combinations of the vectors $v_1, ..., v_m$) is the *span* of the vectors $v_1, ..., v_m$. So, we can rephrase our definition above to say that a set of vectors $\{v_1, ..., v_m\}$ spans a vector space V over a field $\mathbb{F}$ iff

$$\text{span}(\{v_1, ..., v_m\}) := \{a_1 v_1 + ... + a_m v_m : a_1, ..., a_m \in \mathbb{F}\} = V$$

In other words, every vector in V can be constructed from a sum of $\mathbb{F}$-multiples of the "building blocks" (spanning set) of vectors $v_1, ..., v_m$.

11.283 **Exercise** Can you find another spanning set of vectors for $\mathbb{R}^2$? How many such sets do you think there are? Can you find a spanning set for $\mathbb{R}^3$?

Linear Independence

Having thought about the previous exercise a bit, you're likely convinced that there are several – in fact, infinitely many – spanning sets for $\mathbb{R}^2$, $\mathbb{R}^3$, any $\mathbb{R}^n$ as vector spaces over $\mathbb{R}$, and likewise, any vector space $\mathbb{C}^n$ over $\mathbb{C}$. This is wonderful in that we know that there are several sets of vectors that suffice to build the vector spaces we care about. What is interesting to consider is whether there is a *minimum* set of vectors we can use to build the space we are considering.

For example, we can build all of the vectors in $\mathbb{R}^2$ using the set of *three* (!) vectors

$$\left\{\begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 4 \end{pmatrix}, \begin{pmatrix} 5 \\ 6 \end{pmatrix}\right\} \tag{11.284}$$

We'll try to convince you here by showing that we can at least build the vector $\begin{pmatrix} 10 & 10 \end{pmatrix}^T$ using these three vectors. Before you see what we do you should think of how many of each of these vectors you need to construct $\begin{pmatrix} 10 & 10 \end{pmatrix}^T$. It's not that easy, actually...

$$\begin{pmatrix} 10 \\ 10 \end{pmatrix} = 1 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + (-7) \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} + 6 \cdot \begin{pmatrix} 5 \\ 6 \end{pmatrix} \tag{11.285}$$

11.286 **Exercise** Check that the equality above is true!

This probably seems a bit magical if you tried to figure it out yourself before you looked. It turns out that it's nothing more than solving a bunch of equations though!

Of course, you're probably thinking "...we had a perfectly good set of *two vectors* that we could use to build everything already. Why bother with this set of three more complicated vectors?" Fair question. First, we'll show you that we don't need three vectors at all – only two. In other words, these three vectors span $\mathbb{R}^2$ over $\mathbb{R}$, but we can do with just two. Later on, we'll try to argue that it is occasionally advantageous to use different sets of building blocks at different times – sort of like a change in perspective.

To see that we only need two of these vectors, we'll show that one of them can be built from the others and then we'll revisit the previous example and drive the point home.

Observe that

$$\begin{pmatrix} 5 \\ 6 \end{pmatrix} = (-1) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + 2 \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \tag{11.287}$$

You might have seen this without any calculation. The point is that we can replace any instance of the vector $\begin{pmatrix} 5 & 6 \end{pmatrix}^T$ with this combination

$$(-1) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + 2 \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \tag{11.288}$$

So, we revisit the previous example and apply this substitution:

$$\begin{aligned} \begin{pmatrix} 10 \\ 10 \end{pmatrix} &= 1 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + (-7) \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} + 6 \cdot \begin{pmatrix} 5 \\ 6 \end{pmatrix} \\ &= 1 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + (-7) \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \\ &\qquad + 6 \cdot \left((-1) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + 2 \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right) \end{aligned} \tag{11.289}$$

and then reorganize:

$$\begin{aligned} &= 1 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + (-7) \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} + (-6) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + 12 \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \\ &= 1 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + (-6) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + (-7) \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} + 12 \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \\ &= (-5) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + 5 \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \end{aligned} \tag{11.290}$$

So, we can express our original vector $\begin{pmatrix} 10 & 10 \end{pmatrix}^T$ using only the building blocks $\begin{pmatrix} 1 & 2 \end{pmatrix}^T$ and $\begin{pmatrix} 3 & 4 \end{pmatrix}^T$.

In sum, we say that $\begin{pmatrix} 5 & 6 \end{pmatrix}^T$ is $\mathbb{R}$-*linearly dependent* (even $\mathbb{Z}$-linearly dependent, since the coefficients are all integers!) on the vectors $\begin{pmatrix} 1 & 2 \end{pmatrix}^T$ and $\begin{pmatrix} 3 & 4 \end{pmatrix}^T$ and that $\begin{pmatrix} 10 & 10 \end{pmatrix}^T$ is in the *span* of the vectors $\begin{pmatrix} 1 & 2 \end{pmatrix}^T$

and $\begin{pmatrix} 3 & 4 \end{pmatrix}^T$. Using our previous definition of span, we can state precisely what it means for a vector to be linearly dependent on a set of vectors.

11.291 **Definition** ***Linear dependence***

A vector v in a vector space V over a field $\mathbb{F}$ is $\mathbb{F}$-*linearly dependent* on the vectors $v_1, ..., v_m$ iff v is in span$\{v_1, ..., v_m\}$. We often just use the term *linear dependence* rather than $\mathbb{F}$-*linearly dependence* where the field is understood.

Naturally then, we have a notion of linear *in*dependence of vectors!

11.292 **Definition** ***Linear independence of a set of vectors***

We say that a set of vectors $\{v_1, ..., v_m\}$ is *linearly independent* if none of the vectors in the set are linearly dependent on the others. More precisely, we say that $\{v_1, ..., v_m\}$ is a linearly independent set of vectors iff for each $i \in \{1, ..., m\}$, v_i is not in span$\{v_1, ..., v_{i\quad 1}, v_{i+1}, ...v_m\}$.

11.293 **Exercise** Do we need two vectors to span all of $\mathbb{R}^2$? Can we get away with just having one?

Bases and Dimension

Finally, we can define a *basis* of a vector space. This definition will unlock a matrix-centric view of linear algebra, as we're about to see.

You might be wondering why we bothered defining all of the previous terms. It turns out that the precise definition of a matrix that we desire depends (linearly? We couldn't resist...) on the precise definition of a basis of a vector space. In fact, you might have been frustrated by the idea of a matrix this whole time, wondering, "Where are the numbers in the grid coming from?" If you answered a previous exercise asking you to conjecture what the columns of a matrix tell us, what follows will be quite satisfying!

First, the definition of a basis:

11.294 **Definition** ***The definition of a basis***

A *basis* of a vector space V over a field $\mathbb{F}$ is a linearly independent spanning set of vectors.

That's it? Well, quite a bit is packed into the definitions of *linearly independent* and *spanning*, so you might want to review those!

11.295 **Exercise** Check that the set $\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}$ is in fact a basis for $\mathbb{R}^2$.

We keep saying *a* basis, which implies there are more than one. Hopefully, you convinced yourself earlier that there are infinitely many spanning sets of vectors for $\mathbb{R}^2$. It thus follows that there's an infinite number of linearly independent spanning sets for $\mathbb{R}^2$ as a result. Here are a bunch more:

$$\left\{\begin{pmatrix}2\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}, \left\{\begin{pmatrix}1\\2\end{pmatrix}, \begin{pmatrix}3\\4\end{pmatrix}\right\}, \left\{\begin{pmatrix}-1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\} \tag{11.296}$$

All of these bases have the same number of vectors! With enough playing around (don't just believe us!), we might eventually be convinced that all bases have the same number of vectors, and in fact they do.

This is what many call the fundamental theorem of linear algebra:

11.297 Invariance of basis number

The number of basis vectors for a vector space is the same no matter which basis we choose.

This calls for a definition!

11.298 **Definition** ***Dimension***

The *dimension* of a vector space V is the number of vectors for any basis for V.

Observe that we need the theorem to have a well-defined notion of dimension. If we didn't know that all bases have the same length, we could potentially have two different answers for what the dimension of a vector space is! Believing the theorem, we have a natural notion of finite-dimensionality: a vector space is finite-dimensional iff it has a finite basis.

We won't prove the theorem here, and instead will offer some slight indication for why it should be true. It is likely believable that any set of linearly independent vectors in a finite-dimensional vector space should have size less than or equal to the size of any spanning set of vectors. This fact is known as the *Exchange Lemma*. It is of fundamental importance in linear algebra. You can quickly ascertain the power of the Exchange Lemma, since knowing it and that any two bases are spanning sets already implies they must be of the same length.

11.299 **Exercise** Convince yourself that any two bases for a finite-dimensional vector space have the same size.

This idea has major implications. Suppose we already know the dimension d of a vector space V and that we'd like to find a basis for this vector space. Imagine we already have a set of d vectors that spans the vector space V. Well, then the previous discussion guarantees that this list of d spanning vectors is linearly independent and thus already a basis for V! Likewise, if we have a list of d linearly independent vectors in V, they must span V and are a basis as well.

So if we want to find a basis for a vector space V of dimension d, it suffices to discover a list of d vectors that are either all linearly independent *or* that spans the space V. One or the other is enough, so long as we have d vectors. Unless otherwise mentioned, we will restrict our attention to the finite-dimensional case.

We write the dimension of a vector space V over a field $\mathbb{F}$ as $\dim_{\mathbb{F}}(V)$, but often shorten this to $\dim(V)$ when the field is understood. So, relating this idea to a question asked in the self-test at the outset of the chapter, we would write $\dim_{\mathbb{C}}(\mathbb{C}^2)$ to indicate the dimension of the vector space $\mathbb{C}^2$ considered as a vector space *over the field* $\mathbb{C}$ and $\dim_{\mathbb{R}}(\mathbb{C}^2)$ to indicate the dimension of the vector space $\mathbb{C}^2$ considered as a vector space *over the field* $\mathbb{R}$. The first of these, $\dim_{\mathbb{C}}(\mathbb{C}^2)$, is equal to 2, while the second, $\dim_{\mathbb{R}}(\mathbb{C}^2)$, is equal to 4. They are therefore not the same thing, and we now appreciate that the subscript is occasionally quite important.[27]

[27] This point relates to the question in the self test at the opening of the chapter which asks the reader to compare $\mathbb{R}^4$ and $\mathbb{C}^2$.

11.300 **Exercise** Try to find a basis of length 2 for the vector space $\mathbb{C}^2$ over $\mathbb{C}$ and then a basis of length 4 for the vector space $\mathbb{C}^2$ over $\mathbb{R}$. This should convince you that the subscript is occasionally necessary. Then, think about the question from the self-test again, and figure out why it's a trick question.

Orthonormal Bases

Once we have a basis, we can ask that it satisfy certain conditions. Let's take our cue from a basis we know and love at this point

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} = \{|0\rangle, |1\rangle\} \tag{11.301}$$

What's so nice about this particular basis is that each of the vectors are of unit length (i.e., have L^2 norm 1) and they are what is referred to as *orthogonal* to each other. "Orthogonal" is essentially a fancy word for perpendicular.[28]

11.302 **Definition** ***Orthogonality of vectors***

More precisely, we say that two vectors u, v in a vector space V equipped with an inner product $\langle \cdot, \cdot \rangle$ are *orthogonal* (to each other) iff their inner product is zero, i.e., $\langle u, v \rangle = 0$.

In Dirac notation, we say that two vectors u and v are orthogonal iff $\langle u|v\rangle = 0$. A basis such that each of the elements have unit length and such that they are pairwise orthogonal, is known as an *orthonormal* basis. In the context of quantum computing, we choose a set of orthonormal basis vectors to be our preferred *computational basis vectors*. We can say this a bit more mathematically:

[28] Perpendicular and orthogonal are interchangeable except in the case that either vector is the zero vector, since it does not make sense to say that a vector is perpendicular to the zero vector as there is no angle between them. When comparing two vectors when at least one of the vectors is the zero vector, we use the word orthogonal.

11.303 Definition *Orthonormal basis*

A set of vectors $\mathscr{B} = \{v_1, ..., v_n\}$ in a vector space V is called an *orthonormal basis* iff $\mathscr{B}$ is a basis, and for all $i, j \in \{1, ..., n\}$,

$$\begin{cases} \langle v_i, v_j \rangle = 1 & \text{if } i = j \\ \langle v_i, v_j \rangle = 0 & \text{if } i \neq j \end{cases}$$

We'll learn later that what is written above is an instance of what is known as the *Kronecker Delta* function δ. An important theorem of linear algebra is that every spanning set of vectors in a finite-dimensional vector space can be pruned to form an orthonormal basis. The process yielding the desired orthonormal basis is known as the *Gram-Schmidt process* in honor of Jørgen Pedersen Gram and Erhard Schmidt.

11.304 Gram-Schmidt Orthonormalization

Every spanning set of vectors for a finite-dimensional vector space can be modified to form an orthonormal basis.

Let's relate these ideas to some of the quantum computing you've read earlier. You've read by now that any state is a superposition of the states $|0\rangle$ and $|1\rangle$. The first thing you should check is that the states $|0\rangle$ and $|1\rangle$ are orthogonal to one another:

11.305 Exercise Verify that the vectors $|0\rangle$ and $|1\rangle$ are in fact orthogonal to one another, i.e., $\langle 0|1\rangle = 0$.

Then, you should check that they're even orthonormal!

11.306 Exercise Verify that the vectors $|0\rangle$ and $|1\rangle$ are even orthonormal, i.e., $\langle 0|0\rangle = 1$ and $\langle 1|1\rangle = 1$.

We already know that $\{|0\rangle, |1\rangle\}$ is a basis for $\mathbb{R}^2$. It's not too difficult to believe that $\{|0\rangle, |1\rangle\}$ is also a basis for $\mathbb{C}^2$.

CHAPTER 12

Mathematical Tools for Quantum Computing II

12.1 Linear Transformations as Matrices

We claimed earlier that every linear transformation has an associated matrix, and vice versa. We aim to demonstrate for you that, in fact, *a linear transformation and a matrix are the same thing*. This justifies the study of linear algebra as the study of matrices and operations on them.

Equipped with the definition of a basis, we are prepared to define the matrix associated to a linear transformation between two vector spaces.

Let V and W be vector spaces over some field $\mathbb{F}$. Yes, they have to be vector spaces over the same field – you'll see why! Then, V and W each have a basis, say $\mathscr{B}_V = \{v_1, ..., v_n\}$ and $\mathscr{B}_W = \{w_1, ..., w_m\}$. So, V has dimension n and W has dimension m.

Let T be a linear transformation between the vector spaces V and W. So, $T : V \to W$ is a linear transformation, and thus a function. Since T is a linear transformation from V to W, each basis element v_j for $i \in \{1, ..., n\}$ is mapped into W, i.e., $Tv_j \in W$. Since W is spanned by the vectors $w_1, ..., w_m$ and $Tv_j \in W$, there exist coefficients $a_{1,j}, ..., a_{m,j}$ such that $Tv_j = a_{1,j}w_1 + ... + a_{m,j}w_m$, i.e., we can express each Tv_j as a linear combination of the basis elements $w_1, ..., w_m$.

So, to each basis element v_j of V, we have an associated list of elements $a_{1,j}, ..., a_{m,j}$ of the field $\mathbb{F}$. Maybe you see where we're going with this...

We define the matrix $\mathscr{M}(T : V \to W)_{\mathscr{B}_V, \mathscr{B}_W}$ associated to the linear transformation $T : V \to W$ with respect to the bases $\mathscr{B}_V$ and $\mathscr{B}_W$ entrywise:

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_12

12.1 **Definition** ***Matrix associated to a linear transformation***

$$\left(\mathscr{M}(T : V \to W)_{\mathscr{B}_V, \mathscr{B}_W}\right)_{i,j} := a_{i,j}$$

where $a_{i,j}$ is the coefficient of the ith basis element of $\mathscr{B}_W$ in the linear combination $Tv_j = a_{1,j}w_1 + \ldots + a_{m,j}w_m$ expressing the image Tv_j of the jth basis element of $\mathscr{B}_V$.

12.2 **Exercise** You should wonder why we can't play the same game with *any* transformation between vector spaces. More specifically, why do we only define a matrix for a *linear* transformation between vector spaces? Can you figure this out?

Let's see this construction of a matrix in action. Consider the linear transformation T from $\mathbb{R}^2 \to \mathbb{R}^2$ described by

$$T\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} x+y \\ x-y \end{pmatrix}$$

where we take the basis for $\mathbb{R}^2$ to be the standard basis

$$\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}$$

in each case. So far, we have T as defined, each of V and W are $\mathbb{R}^2$, and each of $\mathscr{B}_V$ and $\mathscr{B}_W$ are $\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}$.

12.3 **Exercise** Check that the transformation T given above is actually a linear transformation.

Now that you've verified that the transformation T is linear, we can proceed to figure out the coefficients $a_{i,j}$ as in the construction above. This amounts to figuring out where T sends the two basis elements. Following the definition of T, we have

$$T\begin{pmatrix} 1 \\ 0 \end{pmatrix} := \begin{pmatrix} 1+0 \\ 1-0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{12.4}$$

Then, we decompose this into a linear combination of the basis vectors like so

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} = 1 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{12.5}$$

At this point, we see that

$$\left(\mathscr{M}(T : V \to W)_{\mathscr{B}_V, \mathscr{B}_W}\right)_{1,1} := a_{1,1} = 1 \tag{12.6}$$

and that

$$\left(\mathscr{M}(T : V \to W)_{\mathscr{B}_V, \mathscr{B}_W}\right)_{2,1} := a_{2,1} = 1, \tag{12.7}$$

We'll abbreviate the fancy notation

$$\left(\mathscr{M}(T : V \to W)_{\mathscr{B}_V, \mathscr{B}_W}\right) \tag{12.8}$$

with $\mathscr{M}(T)$ when the vector spaces V and W and their bases $\mathscr{B}_V$ and $\mathscr{B}_W$ are understood.

12.9 **Exercise** Verify that $a_{1,2} = 1$ and $a_{2,2} = -1$ by determining where the second basis vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ is mapped to and decomposing the resulting vector into a linear combination of the basis vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

After all of this computation, we realize that the matrix associated to the linear transformation $T : V = \mathbb{R}^2 \to W = \mathbb{R}^2$ with respect to the bases

$$\mathscr{B}_V = \mathscr{B}_W = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \tag{12.10}$$

is

$$\mathscr{M}(T) = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{12.11}$$

12.12 **Exercise** To practice building matrices from the description of a linear transformation, construct the matrix associated to the linear transformation

$$T : V = \mathbb{R}^3 \to W = \mathbb{R}^3$$

defined by

$$T\begin{pmatrix} x \\ y \\ z \end{pmatrix} := \begin{pmatrix} 1x + 2y + 3z \\ 4x + 5y + 6z \\ 7x + 8y + 9z \end{pmatrix}$$

with respect to the standard basis

$$\mathscr{B}_V = \mathscr{B}_W = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}$$

for each. You should first check that this is a linear transformation. Then, figure out where each of the basis vectors is mapped. Once you know this, decompose those into a linear combination of the basis vectors and use the coefficients of those linear combinations to build the matrix.

Unless we mention otherwise, we will assume that the vector spaces $\mathbb{R}^n$ and $\mathbb{C}^n$ have the standard bases. You may be frustrated by the definition of matrix multiplication that we gave earlier on. We'd like to demonstrate why it is defined the way it is. It has to do with the intimate relationship between the composition of two transformations (as functions) and the product of the matrices as we have defined it.

Consider the following two transformations $R : \mathbb{R}^2 \to \mathbb{R}^2$ and $S : \mathbb{R}^2 \to \mathbb{R}^2$ defined by

$$R\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} y \\ -x \end{pmatrix} \tag{12.13}$$

and

$$S\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} y \\ x \end{pmatrix}. \tag{12.14}$$

12.15 **Exercise** First, try to understand these transformations geometrically. Then, figure out matrices $\mathscr{M}(R)$ and $\mathscr{M}(S)$ representing each of these transformations (using the standard bases). Afterward, recall how it is that we multiply two matrices, and compute each of the matrix products

$$\mathscr{M}(R) \cdot \mathscr{M}(S) \qquad \text{and} \qquad \mathscr{M}(S) \cdot \mathscr{M}(R) \tag{12.16}$$

They should not be equal, which might be clear from your geometric interpretation!

If you did the above exercise, you now know that R and S have matrices

$$\mathscr{M}(R) = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \text{ and } \mathscr{M}(S) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{12.17}$$

and that the products are

$$\mathscr{M}(R) \cdot \mathscr{M}(S) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \text{ and } \mathscr{M}(S) \cdot \mathscr{M}(R) = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \tag{12.18}$$

Ok, now we're prepared to find the composition of the transformations (functions) R and S. Remember that we work "inside-out." So, S is applied first and R is then applied after S. Thus, we can compute $R \circ S$ as

$$(R \circ S)\begin{pmatrix} x \\ y \end{pmatrix} = R\left(S\begin{pmatrix} x \\ y \end{pmatrix}\right) = R\begin{pmatrix} y \\ x \end{pmatrix} = \begin{pmatrix} x \\ -y \end{pmatrix} \tag{12.19}$$

Next, let us compute $S \circ R$:

$$(S \circ R)\begin{pmatrix} x \\ y \end{pmatrix} = S\left(R\begin{pmatrix} x \\ y \end{pmatrix}\right) = S\begin{pmatrix} y \\ -x \end{pmatrix} = \begin{pmatrix} -x \\ y \end{pmatrix} \tag{12.20}$$

They're not the same! We're getting closer to the analogy. Now, we find the matrix associated to the transformation $R \circ S$. Note where the two basis vectors end up and then express each result as a linear combination of the basis vectors on the other side. Then, the coefficients in these linear combinations help us build the matrix.

$$(R \circ S)\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ -0 \end{pmatrix} = 1 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{12.21}$$

and

$$(R \circ S)\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} = 0 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + (-1) \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{12.22}$$

and

$$(S \circ R)\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix} = (-1) \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{12.23}$$

and

$$(S \circ R)\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -0 \\ 1 \end{pmatrix} = 0 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{12.24}$$

Now, we have everything we need to build the matrices $\mathscr{M}(R \circ S)$ and $\mathscr{M}(S \circ R)$!

12.25 **Exercise** Using the above computations, determine the matrices $\mathscr{M}(R \circ S)$ and $\mathscr{M}(S \circ R)$.

Having done the above exercise, you now see that the matrices associated to the transformations $R \circ S$ and $S \circ R$ are

$$\mathscr{M}(R \circ S) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \text{ and } \mathscr{M}(S \circ R) = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \tag{12.26}$$

and these are the matrices $\mathscr{M}(R) \cdot \mathscr{M}(S)$ and $\mathscr{M}(S) \cdot \mathscr{M}(R)$ we computed earlier! This isn't just a happy coincidence, by the way. This is the motivation for defining the matrix product the way we did.

12.27 The Matrix of a Composition of Linear Transformations

The matrix of a composition of linear transformations is the product of their matrices, for all linear transformations S and T,

$$\mathscr{M}(S \circ T) = \mathscr{M}(S) \cdot \mathscr{M}(T)$$

12.2 *Matrices as Operators*

We refer to a linear transformation that maps from a vector space to itself as a *linear operator* on that space. In fact, all of the linear transformations we've considered (expect for one non-square matrix that we used to explain the transpose operation on matrices) are linear operators! We pay special attention to operators in quantum mechanics.

More particularly, we care about invertible linear operators because we can use them to create quantum gates and thus quantum circuits. Next, we introduce the determinant, a numerical invariant of a matrix encoding the invertibility of the transformation of space that it represents!

An Introduction to the Determinant

Each square matrix has an associated *determinant*, which speaks for the geometry of the linear transformation of space represented by that matrix. It subsequently describes the invertibility of the linear transformation, whether the transformation can be undone.

What do we mean by this idea that a transformation can be undone? Let's see an example of a transformation that can be undone, and then an example that can't.

First, an example of a transformation that can be undone:

Consider the linear transformation R_π from $\mathbb{R}^2$ to $\mathbb{R}^2$ described by the matrix

$$R_\pi := \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{pmatrix} \tag{12.28}$$

which can be found in chapter 3.

12.29 Exercise Review Euler's formula from earlier in this chapter and check that

$$e^{i\pi} = \cos(\pi) + i \sin(\pi) = -1 + 0 \cdot i = -1$$

The above exercise alerts us to the fact that the matrix R_π above can actually be expressed as

$$R_\pi := \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{12.30}$$

12.31 Exercise You're invited to investigate and figure out which transformation of two-dimensional space is described by this matrix. Figure out where the basis vectors are mapped and you should get an idea!

If you tried the above exercise, you probably figured out that the matrix R_π describes a reflection of two-dimensional space over the x-axis. Believing this, it is fairly clear geometrically that this transformation can be undone – just reflect back! Or you could just reflect again, since two reflections over an axis does nothing. You might wonder then why π appears in the name of this matrix. We'll give you a little idea of why. The π is there to signify that this matrix has the effect of rotating the larger space $\mathbb{C}^2$ about the z-axis[1] by π radians, although we'd prefer not to get into this discussion right now. We'd

[1] Refer to the Bloch sphere in chapter 3.

like for you to take away the fact that this is a linear transformation of space that can be undone – that's good for now!

You might have also noticed that the matrix R_π is a special case of the matrix

$$R_\varphi := \begin{pmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{pmatrix} \tag{12.32}$$

mentioned in chapter 3, where the angle φ is π.

Now, let us consider a transformation that *cannot* be undone. Consider the transformation Proj_x from $\mathbb{R}^2$ to $\mathbb{R}^2$ described by the matrix

$$\text{Proj}_x := \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{12.33}$$

12.34 **Exercise** As before, you should try and figure out which transformation of space is described by the matrix

$$\text{Proj}_x := \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

above. The name might give it away...

So, if you tried the above exercise, you're hopefully convinced that this matrix describes the projection of a vector onto the x-axis. For example, applying Proj_x to the vector $\begin{pmatrix} 1 & 1 \end{pmatrix}^T$ yields $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$. Since many vectors also get mapped to $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ by Proj_x we lose information as to what the input was if we just look at the output. For example, this vector $\begin{pmatrix} 1 & 2 \end{pmatrix}^T$ also gets mapped to $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ (and actually, any vector whose first component is 1 will as well!). Since we cannot recover the input from the output and this transformation cannot be undone, it is not invertible.

In essence, this transformation collapses the whole space to the x-axis. This mental image should give the impression that we cannot undo this transformation. In some sense, a higher dimensional space is collapsed into a smaller dimensional one, and so information must be lost.

We don't like such transformations in quantum computing! We'd like for the matrices that represent the quantum logic gates we use to preserve all of the information given to them, and we'd like for these transformations to be reversible. We'll see that we even ask for more of these matrices in what follows.

So, at this point, hopefully you're convinced that not all linear transformations can be undone. It would be nice to have a numerical invariant that encodes the invertibility of a linear transformation. Enter the determinant.

Let's see how to compute the determinant of a 2×2 matrix with an example.

Consider the matrix

$$A := \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{12.35}$$

We compute its determinant like so

$$\det(A) := \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = 1 \cdot 4 - 2 \cdot 3 = 4 - 6 = -2 \tag{12.36}$$

12.37 **Exercise** You're invited to investigate and describe the transformation of two-dimensional space described by this transformation. Figure out where the basis vectors end up and that should give you an idea!

The previous exercise hopefully convinced you that the transformation of two-dimensional space represented by the matrix A is invertible.

Let's follow this example and compute the determinants of the matrices described above

$$\det(R_{\frac{\pi}{2}}) := \begin{vmatrix} 1 & 0 \\ 0 & \text{-1} \end{vmatrix} = 1 \cdot (-1) - 0 \cdot 0 = -1 \tag{12.38}$$

$$\det(\text{Proj}_x) := \begin{vmatrix} 1 & 0 \\ 0 & 0 \end{vmatrix} = 1 \cdot 0 - 0 \cdot 0 = 0 \tag{12.39}$$

If you're observant, you may notice that the matrices corresponding to the invertible transformations have nonzero determinant, while the matrix corresponding to the non-invertible transformation has zero determinant.

12.40 Determinant of a 2×2 matrix

The determinant of a 2×2 matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

is

$$ad - bc$$

You might wonder how to compute the determinant of a 3×3 matrix. Again, we'll demonstrate by example

$$det \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} := \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} \tag{12.41}$$

$$= 1 \cdot \begin{vmatrix} 5 & 6 \\ 8 & 9 \end{vmatrix} - 2 \cdot \begin{vmatrix} 4 & 6 \\ 7 & 9 \end{vmatrix} + 3 \cdot \begin{vmatrix} 4 & 5 \\ 7 & 8 \end{vmatrix} \tag{12.42}$$

$$= 1 \cdot (5 \cdot 9 - 8 \cdot 6) - 2 \cdot (4 \cdot 9 - 7 \cdot 6) + 3 \cdot (4 \cdot 8 - 7 \cdot 5) \tag{12.43}$$

$$= 1 \cdot (45 - 48) - 2 \cdot (36 - 42) + 3 \cdot (32 - 35) \tag{12.44}$$

$$= 1 \cdot (-3) - 2 \cdot (-6) + 3 \cdot (-3) \tag{12.45}$$

$$= -3 + 12 - 9 = 9 - 9 = 0 \tag{12.46}$$

12.47 Equivalent formulations of invertibility

The following are equivalent for a linear transformation T:

- $\det(T) = 0$
- T is not invertible
- The rows of the matrix representing T are linearly dependent
- The columns of the matrix representing T are linearly dependen.

Unfortunately, we will not prove this here. This set of equivalences relating the determinant of a matrix, and thus a linear transformation, to geometric attributes such as invertibility of the transformation should be surprising given the seemingly combinatorial description of the determinant. It might not seem geometric at all from the definition we've given.

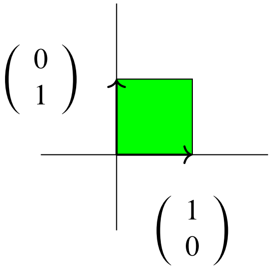

$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$
$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$

Figure 12.1: Unit square in the plane

The Geometry of the Determinant

There is a more geometric definition of the determinant and we would like to give you a small taste of it. Consider the matrix

$$\begin{pmatrix} 1 & 4 \\ 2 & 2 \end{pmatrix}$$

again and observe that it sends the first basis vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ to the vector $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ and it sends the second basis vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ to the vector $\begin{pmatrix} 4 \\ 2 \end{pmatrix}$. This should convince us that the unit square formed by the basis vectors in the first quadrant, as in Figure 12.1, is mapped to the parallelogram whose coordinates are at the points

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 4 \\ 2 \end{pmatrix} \text{ and } \begin{pmatrix} 5 \\ 4 \end{pmatrix}$$

as depicted in Figure 12.2.

12.48 **Exercise** You should draw the parallelogram!

You'll realize that in some sense that the *orientation* of the unit square "flips" upon being transformed to the parallelogram, since the vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, which constitute the unit square in the first quadrant, exchange places to form the parallelogram.

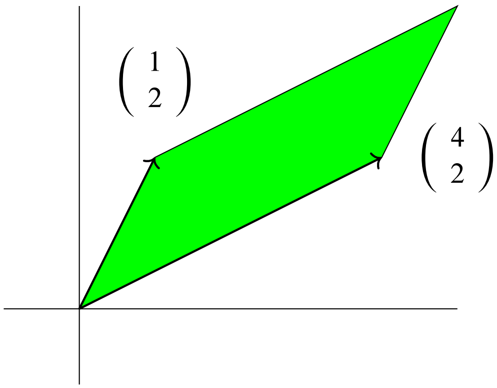

$\begin{pmatrix}1\\2\end{pmatrix}$
$\begin{pmatrix}4\\2\end{pmatrix}$

Figure 12.2: Parallelogram

The unit square has area 1, while the parallelogram has area 2. What does this have to do with the determinant of this matrix, which is -2? Well, that the orientation of the unit square “flips” is encoded in the fact that the determinant is negative. Transformations that preserve orientation have positive determinant, while transformations that flip orientation have negative determinant. Then, the magnitude of the determinant (absolute value) encodes the factor by which the unit parallelogram is magnified by the transformation.

In general, the determinant of the matrix associated to a linear transformation encodes whether that transformation is orientation preserving and then by which factor it “stretches” the unit parallelepiped (higher-dimensional analog of a parallelogram) – quite geometric, actually! Figure 12.2 depicts the parallelogram.

Matrix Inversion

We previously discussed how the determinant of a matrix encodes its invertibility. But how do we actually find the inverse transformation?

First off, when we say an *inverse transformation*, what do we mean? Well, transformation is just another word for function, so presumably what we seek is an inverse function of the transformation we’re given. It might not be so simple to figure out what an inverse function is though. To see what we mean, try to find the inverse transformation for the transformation T of space described by

$$T : \mathbb{R}^3 \to \mathbb{R}^3 \tag{12.49}$$

$$T\begin{pmatrix} x \\ y \\ z \end{pmatrix} := \begin{pmatrix} 1 \cdot x + 2 \cdot y + 3 \cdot z \\ 4 \cdot x + 5 \cdot y + 6 \cdot z \\ 7 \cdot x + 8 \cdot y + 10 \cdot z \end{pmatrix} \tag{12.50}$$

Is it even clear that we *can* invert this transformation? We saw that not all transformations are invertible.

12.51 **Exercise** Find the matrix $\mathscr{M}(T)$ associated to the transformation T and then compute its determinant. Make a claim about the invertibility of the transformation T based on your computation of the determinant. Can we invert T?

12.52 **Exercise** Can we give a reasonable geometric description of this transformation of three-dimensional space from this definition of T?

Recall from our earlier discussion about functions and attributes of functions that we say that a function $f : X \to Y$ is invertible iff there exists an inverse function $f^{\ \ 1} : Y \to X$ such that the compositions $f^{\ \ 1} \circ f : X \to X$ and $f \circ f^{\ \ 1} : Y \to Y$ are the respective identity functions I_X and I_Y. So, we seek a transformation (function) $T^{\ \ 1}$ whose composition with the transformation T is the identity on either side.

If we have such a $T^{\ \ 1}$, we'll know that

$$T \circ T^{\ \ 1} = I_{\mathbb{R}^3} \tag{12.53}$$

and

$$T^{\ \ 1} \circ T = I_{\mathbb{R}^3} \tag{12.54}$$

Now that we know that every linear transformation of space is a matrix and that the composition of two transformations is analogous to the multiplication of their corresponding matrices, we might hope to express the given transformation T as a matrix $\mathscr{M}(T)$. We can then use this matrix description to come up with an inverse matrix to multiply $\mathscr{M}(T)$ to get the identity matrix (which corresponds to the identity function).

In other words, we apply the "matrix operation" to the compositions above yielding

$$\mathscr{M}(T \circ T\ ^{1}) = \mathscr{M}(I_{\mathbb{R}^3}) \tag{12.55}$$

and

$$\mathscr{M}(T\ ^{1} \circ T) = \mathscr{M}(I_{\mathbb{R}^3}) \tag{12.56}$$

The question then is what these matrices are. However, like we said earlier, the matrix of a composition of transformations is the product of their matrices, so we have

$$\mathscr{M}(T)\mathscr{M}(T\ ^{1}) = \mathscr{M}(I_{\mathbb{R}^3}) \tag{12.57}$$

and

$$\mathscr{M}(T\ ^{1})\mathscr{M}(T) = \mathscr{M}(I_{\mathbb{R}^3}) \tag{12.58}$$

We need to find a matrix whose product with the matrix $\mathscr{M}(T)$ on either side is the identity matrix. This is an enormous observation!

We now focus our attention on what it means to find such a matrix. Consider the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{12.59}$$

Let

$$B = \begin{pmatrix} x & y \\ z & w \end{pmatrix} \tag{12.60}$$

be any other matrix, and now suppose that B has the desired inverse property, that $AB = I_2 = BA$.

Well, then to start, we would have

$$AB = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x & y \\ z & w \end{pmatrix} = I_2 \tag{12.61}$$

Let's extract the meaning of

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x & y \\ z & w \end{pmatrix} \tag{12.62}$$

Recall the definition of matrix multiplication. Before we compute the product, you should give it a try!

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} 1 \cdot x + 2 \cdot z & 1 \cdot y + 2 \cdot w \\ 3 \cdot x + 4 \cdot z & 3 \cdot y + 4 \cdot w \end{pmatrix} \tag{12.63}$$

Now, we want this to be equal to the identity matrix, so we need for the following equations and inequations to be satisfied

$$1x + 2z \neq 0$$

$$1y + 2w = 0$$

$$3x + 4z = 0$$

$$3y + 4w \neq 0$$

$$1x + 2z = 3y + 4w$$

The second and third equations are self-evident, but the first, fourth and fifth desire some explanation. We want for the first, fourth and fifth entries to be nonzero and equal so that we can divide by them and force the diagonal entries to be equal to 1 each.

12.64 **Exercise** Try to solve the system of equations and inequations above.

With enough effort, you'll find that the appropriate choice of x, y, z, w is

$$x = 4, y = -2, z = -3, w = 1 \tag{12.65}$$

So, the matrix we seek is

$$B = \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} \tag{12.66}$$

Let's compute the product

$$AB = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} = \begin{pmatrix} -2 & 0 \\ 0 & -2 \end{pmatrix} \tag{12.67}$$

That's not the identity matrix! It's not so bad, actually. Remember that we demanded that the diagonal entries would be nonzero and equal? Now, you'll see why.

Sure, the matrix B doesn't work, but the matrix $\left(-\frac{1}{2}\right) \cdot B$ does (remember how we multiply a matrix by a number)

$$\begin{aligned}
A\left(\left(\frac{1}{-2}\right)B\right) &= \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\left(\left(\frac{1}{-2}\right)\cdot\begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix}\right) \\
&= \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\left(\left(\frac{1}{-2}\right)\cdot\begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix}\right) \\
&= \left(\frac{1}{-2}\right)\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} \\
&= \left(\frac{1}{-2}\right)\begin{pmatrix} -2 & 0 \\ 0 & -2 \end{pmatrix} \\
&= \begin{pmatrix} (\frac{1}{2})\cdot(-2) & (\frac{1}{2})\cdot 0 \\ (\frac{1}{2})\cdot 0 & (\frac{1}{2})\cdot(-2) \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad (12.68)
\end{aligned}$$

So, we have good reason to call the matrix $\left(\frac{1}{2}\right) B$ by the name $A^{\ 1}$!

12.69 Exercise We should check that multiplying on either side yields the identity matrix – do this! That is, verify that $\left(\left(-\frac{1}{2}\right) B\right) A = I$.

Then, we have found the matrix inverting the matrix A, so it's just a matter of determining what linear transformation this inverse matrix represents. Recalling that the entries in a matrix are exactly expressing the coefficients of the images of the basis elements, we can recover the transformation without too much trouble:

12.70 Exercise Try to describe the linear transformation corresponding to the matrix $\left(\frac{1}{2}\right) \cdot B$ using function notation. Fill in the details in the unfinished equation

$$\left(\frac{1}{-2}\right)\cdot B\begin{pmatrix} x \\ y \end{pmatrix} :=$$

Hopefully, you figured out that what should be on the right side of the equation is the vector

$$\left(\frac{1}{-2}\right)\cdot\begin{pmatrix} 4\cdot x + -3\cdot y \\ -2\cdot x + 1\cdot y \end{pmatrix} \qquad (12.71)$$

We've got it! The inverse transformation of the linear transformation

$$A\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} 1\cdot x + 3\cdot y \\ 2\cdot x + 4\cdot y \end{pmatrix} \tag{12.72}$$

which you might recall has the geometric effect of transforming the unit square in the first quadrant into the parallelogram described in the section about determinants, is the transformation

$$\left(\frac{1}{-2}\right)\cdot B\begin{pmatrix} x \\ y \end{pmatrix} := \left(\frac{1}{-2}\right)\cdot\begin{pmatrix} 4\cdot x + -3\cdot y \\ -2\cdot x + 1\cdot y \end{pmatrix} \tag{12.73}$$

That's wonderful and all, but is there a pattern here? Can we extrapolate a general technique from this example? Perhaps you noticed this matrix from our earlier discussion about determinants. Maybe you even got the impression that the -2 appearing in the fraction $\frac{1}{-2}$ in front of B has something to do with the previous discussion, since -2 is the determinant of the matrix B! You'd be correct!

In general, given a 2×2 matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{12.74}$$

its inverse matrix is the matrix

$$A^{\ 1} := \frac{1}{\det(A)}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \tag{12.75}$$

12.76 **Definition** ***Adjugate matrix***

The matrix

$$\begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \tag{12.77}$$

is known as the *adjugate* of A.

In general, given a matrix A, we denote the adjugate of A by $\operatorname{adj}(A)$. What we discovered above can then be phrased as

$$\begin{aligned} A\cdot\big(\det(A)^{\ 1}\cdot\operatorname{adj}(A)\big) &= \det(A)^{\ 1}\cdot(A\cdot\operatorname{adj}(A)) = \det(A)^{\ 1}\cdot(\det(A)\cdot I) \\ &= \big(\det(A)^{\ 1}\cdot\det(A)\big)\cdot I = 1\cdot I = I \end{aligned} \tag{12.78}$$

So, the inverse of a matrix A is the matrix described by $\det(A)^{\ 1}\cdot\operatorname{adj}(A)$. This makes us realize that we need to find a way to determine the adjugate matrix. Remarkably, the adjugate matrix is equal to the transpose of what is known as the *cofactor* matrix of A.

So, what is the cofactor matrix of a matrix? We'll define it and then invite you to verify that the adjugate we built above does in fact satisfy this definition.

The cofactor matrix of a matrix A is the matrix C whose entries $C_{i,j}$ are the determinants of the $(n-1)\times(n-1)$ submatrices of A formed by removing the ith row and jth column of A. We call $C_{i,j}$ the i, jth cofactor of A. We'll demonstrate with an example. Consider the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{12.79}$$

Let's compute the $1, 1$ cofactor $C_{1,1}$. Removing the 1st row and 1st column of A yields the smaller 2×2 matrix

$$\begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix} \tag{12.80}$$

and its determinant is the cofactor $C_{1,1}$ that we're after

$$C_{1,1} := \begin{vmatrix} 5 & 6 \\ 8 & 9 \end{vmatrix} = 5 \cdot 9 - 8 \cdot 6 = 45 - 48 = -3 \tag{12.81}$$

12.82 **Exercise** Compute the cofactors $C_{1,2}$ and $C_{1,3}$.

If you tried the above the exercise, you found that

$$C_{1,2} := \begin{vmatrix} 4 & 6 \\ 7 & 9 \end{vmatrix} = 4 \cdot 9 - 7 \cdot 6 = 36 - 42 = -6 \tag{12.83}$$

and

$$C_{1,3} := \begin{vmatrix} 4 & 5 \\ 7 & 8 \end{vmatrix} = 4 \cdot 8 - 5 \cdot 7 = 32 - 35 = -3 \tag{12.84}$$

12.85 **Exercise** The ambitious reader should go forth and compute the remaining 6 cofactors $C_{2,1}, C_{2,2}, C_{2,3}, C_{3,1}, C_{3,2}, C_{3,3}$ and then construct the cofactor matrix C. Further, transpose this matrix. You'll run into a problem if you try to create the inverse though!

If you try to construct the inverse of this matrix, you'll run into an issue, as stated in the above exercise. Everything works swimmingly until you compute the determinant – it's 0!

There are several ways to discern the determinant of A is 0 and seeing them will tie some ideas together.

First of all, the first column

$$\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} \tag{12.86}$$

of the matrix A is linearly dependent on the other two.

12.87 **Exercise** Can you find the dependence?

With a bit of computation, you'll find that 2 of the second column minus 1 of the third column is the first column

$$\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} = 2 \cdot \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix} + (-1) \cdot \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix} \tag{12.88}$$

If we apply the theorem stated earlier about all of the equivalent ways of expressing that the determinant of a linear transformation is 0, we see that the linear dependence of the first column on the other two is equivalent to the fact that the determinant of A is 0.

Now, we'll perform a computation that will lead to a pretty cool observation connecting the inverse of a matrix, its determinant, and the cofactor matrix we described above. It will start off rather unusual, but if you follow each equality, you'll see the connection!

$$\begin{aligned}\det(A) &= 0 = -3 - 2 \cdot (-6) - 3 \cdot 3 = 1 \cdot C_{1,1} - 2 \cdot C_{1,2} + 3 \cdot C_{1,3} \\ &= (-1)^{1+1} \cdot 1 \cdot C_{1,1} + (-1)^{1+2} \cdot 2 \cdot C_{1,2} + (-1)^{1+3} \cdot 3 \cdot C_{1,3} \\ &= (-1)^{1+1} \cdot A_{1,1} \cdot C_{1,1} + (-1)^{1+2} \cdot A_{1,2} \cdot C_{1,2} \\ &\qquad + (-1)^{1+3} \cdot A_{1,3} \cdot C_{1,3}\end{aligned} \tag{12.89}$$

As you can see, we can compute the determinant by taking an alternating sum of cofactors with the coefficients of the entries of the matrix along that row.

Hopefully, this example convinces you this might be true. To be a bit more convinced, you should try to compute the determinant of A along the second row instead. In fact, you should try to compute the determinant along any row or column you like and see that they all yield the same answer!

12.3 *Eigenvectors and Eigenvalues*

To motivate this next topic, let's play around with the matrix

$$A := \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} \tag{12.90}$$

12.91 **Exercise** First of all, can you describe the transformation of space it represents?

You'll realize that the first basis vector is stretched by a factor of 2, while the second basis vector is stretched by a factor of 3. So, these vectors don't move under this transformation; they are simply scaled. You should contrast this example with the previous example

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{12.92}$$

Can you find a similar set of vectors that are only stretched by the transformation corresponding to B for the transformation corresponding to this matrix? Good luck – we bet you can't find them!

The vectors in the first example, the standard basis vectors, are referred to as *eigenvectors* of the matrix A. In general, a nonzero vector v is an *eigenvector* of a linear transformation T iff there exists $\lambda \in \mathbb{F}$ such that

$$Tv = \lambda v \tag{12.93}$$

In other words, the operation of T on v simply scales the vector v by the scalar λ.

Where did the name eigenvector come from? Well, the Germans are responsible for quite a bit of the terminology and notation of algebra, and "eigen" means "own," as in "one's own" or "characteristic" in German. So, an eigenvector is a vector *characteristic* to the matrix A. This implies that we could very well be able to identify the matrix A given all of its characteristic vectors, eigenvectors.

This is true for the most part. We know from earlier that a linear transformation is described entirely by how it acts on a basis for the space. For the example A above, we're fortunate to know immediately from the description of A how it acts on the *standard* basis, and so we pretty much know everything about A already.

What if the basis was different though? We saw earlier that there are several bases for a vector space, so could it be that we understand the behavior of a matrix on a different basis than the standard basis? Well, sure. Here's an example of such a scenario:

Consider the matrix

$$B = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \tag{12.94}$$

Let's apply this transformation to the vectors $v_1 := \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $v_2 := \begin{pmatrix} 1 \\ 1 \end{pmatrix}$:

$$Bv_1 = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix} = 2 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2 \cdot v_1 \tag{12.95}$$

and

$$Bv_2 = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} = 3 \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 3 \cdot v_2 \tag{12.96}$$

Interesting: so, $Bv_1 = 2 \cdot v_1$ and $Bv_2 = 3 \cdot v_2$.

Now, let's try this out on the vector $v := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Then, we have

$$Bv := B \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \tag{12.97}$$

and notice that if there were a number λ such that

$$\begin{pmatrix} 1 \\ 3 \end{pmatrix} = \lambda \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \lambda \cdot 0 \\ \lambda \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \lambda \end{pmatrix} \tag{12.98}$$

then $1 = 0$. So, there is no such number λ. This is equivalent to saying the vector v is not like the vectors v_1 and v_2, since the transformation does not simply stretch v by some factor.

The vectors v_1 and v_2 are the eigenvectors of the matrix B, while v is not. We refer to the scale factors 2 and 3 as the *eigenvalues* of the eigenvectors v_1 and v_2, respectively. In general, the *eigenvalue* associated to an eigenvector v is the scalar λ appearing in the definition of an eigenvector from earlier

$$Tv = \lambda v \tag{12.99}$$

Collectively, we refer to the eigenvectors and eigenvalues of a transformation as its *eigenstuff*.[2] There can only be two of these eigenvectors for an operator on two-dimensional space, and in general, there could be as many as n for an operator on n-dimensional space.

12.100 **Exercise** Find the eigenstuff for the matrices

$$\text{Proj}_x := \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{12.101}$$

and

$$M := \begin{pmatrix} 2 & 0 \\ -\frac{1}{2} & 3 \end{pmatrix} \tag{12.102}$$

Finding the eigenstuff for Proj_x shouldn't be too hard, while M might pose a bit more of a challenge.

Change of Basis

Let's keep running with the matrix B from our discussion of eigenstuff. Although the standard basis is great, we'd like to demonstrate that changing our basis to the eigenvectors of B might be advantageous for several reasons. Well, we should probably check that the eigenvectors of B do indeed form a basis for the vector space $\mathbb{R}^2$.

12.103 **Exercise** Verify that the eigenvectors

$$v_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, v_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{12.104}$$

actually form a basis for $\mathbb{R}^2$. Remember that this means we have to check that v_1 and v_2 are linearly independent and that v_1 and v_2 span $\mathbb{R}^2$. You should do this by brute force first. Then, think about how you could apply the theorem about the length of any basis for a vector space of a certain dimension afterward.

[2] The eigenstuff is sometimes called the eigensystem of the transformation. We acknowledge (and embrace) the informal choice of terminology.

We know the standard basis consisting of two vectors is indeed a basis for $\mathbb{R}^2$, so that theorem ensures the length of any list of basis elements for $\mathbb{R}^2$ is also two. Knowing this and that the eigenvectors are either linearly independent or spanning therefore guarantees they form a basis by the theorem.

We know from earlier that the matrix describing a linear transformation depends on the bases we choose for the space. Let's change the basis we're using from the standard basis to the basis of eigenvectors and see how the matrix B changes as a result!

To perform this operation, we need to know how the transformation T corresponding to the matrix B acts on the eigenvectors v_1 and v_2 in terms of v_1 and v_2 only, but we've already figured this out! Recall that $Bv_1 = 2 \cdot v_1 = 2 \cdot v_1 + 0 \cdot v_2$ and that $Bv_2 = 3v_2 = 0 \cdot v_1 + 3 \cdot v_2$.

So, we can make the matrix for the transformation T with respect to the basis

$$\mathscr{B}_{\mathbb{R}^2} := \{v_1, v_2\} \tag{12.105}$$

of eigenvectors v_1 and v_2 following the procedure described in an earlier section

$$\left(\mathscr{M}(T : \mathbb{R}^2 \to \mathbb{R}^2)_{\mathscr{B}_{\mathbb{R}^2}, \mathscr{B}_{\mathbb{R}^2}}\right) = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} \tag{12.106}$$

Let's abbreviate all of that with $\mathscr{M}(T)$ like we have before. We can see that this matrix is diagonal; diagonal matrices are quite useful. To see why, try the following exercise.

12.107 **Exercise** Find the square of the matrix $\mathscr{M}(T) = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$ formed with respect to the basis of eigenvectors v_1 and v_2 (sometimes called an *eigenbasis*), compute

$$\mathscr{M}(T)^2 := \mathscr{M}(T) \cdot \mathscr{M}(T)$$

Then, compute

$$\mathscr{M}(T)^3 := \mathscr{M}(T) \cdot \mathscr{M}(T) \cdot \mathscr{M}(T)$$

Then, compute

$$\mathscr{M}(T)^4 := \mathscr{M}(T) \cdot \mathscr{M}(T) \cdot \mathscr{M}(T) \cdot \mathscr{M}(T)$$

What's going on? Can you compute $\mathscr{M}(T)^{100}$?

12.108 **Exercise** Now, try this for the original matrix $B := \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$ formed with respect to the standard basis vectors. That is, compute the square of B. Then, compute B^3. Then, compute B^4. Can you compute B^{100}?

We're willing to bet you figured out how to do the exercise regarding $\mathscr{M}(T)$ and gave up while trying the exercise regarding B — anybody would. The difference between $\mathscr{M}(T)$ and B is that $\mathscr{M}(T)$ is diagonal, while B is not. The failure of B to be diagonal causes considerable difficulties when multiplying, as you learned in the exercise.

The question then becomes: Can we always change our basis to one where the matrix we start with ends up a diagonal matrix?

Actually, no, not always.

12.109 **Exercise** Think about why we can't find a basis of eigenvectors for the transformation described by the matrix

$$T := \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Matrices that have this special property are aptly named *diagonalizable*. It is not so simple to give an explicit characterization of matrices that are diagonalizable, so we won't delve into this right now.

Here we make a connection between quantum mechanics and eigenstuff. Linear operators with eigenvalues that are real numbers form a special class of operators. In our review of quantum mechanics in a previous chapter, we reviewed the measurement postulate of quantum mechanics; this postulate states that any operator associated with a physically measurable property will be Hermitian. We have not seen Hermitian operators just yet, but we will find that they are a class of operators whose eigenvalues are *always* real numbers, and so will be measurable. We'll see what Hermitian operators are and why they have such a remarkable property in a little bit!

12.4 *Further Investigation of Inner Products*

Further investigation of the previously defined inner product reveals some salient properties. The first of which is that of *conjugate symmetry*, which states for all vectors u, v, $\langle u, v\rangle = \overline{\langle v, u\rangle}$. So, exchanging the vectors does not yield the same result, but the conjugate instead. This might seem odd at first glance, but an example should help convince us this should be the case.

Let $u := \begin{pmatrix} i \\ 1 \end{pmatrix}$ and $v := \begin{pmatrix} 2 \\ i \end{pmatrix}$. Then

$$\langle u, v\rangle = \left\langle \begin{pmatrix} i \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ i \end{pmatrix} \right\rangle = \bar{i} \cdot 2 + \bar{1} \cdot i = (-i) \cdot 2 + 1 \cdot i = -2i + i = -i \tag{12.110}$$

and

$$\langle v, u\rangle = \left\langle \begin{pmatrix} 2 \\ i \end{pmatrix}, \begin{pmatrix} i \\ 1 \end{pmatrix} \right\rangle = \bar{2} \cdot i + \bar{i} \cdot 1 = 2 \cdot i + (-i) \cdot 1 = 2i - i = i \tag{12.111}$$

So, unless we conjugate, we don't get the same number. We should conjugate if we exchange the vectors in the inner product.

12.112 **Exercise** Let $u := \begin{pmatrix} i \\ 2 \end{pmatrix}$ and $v := \begin{pmatrix} 1 \\ i \end{pmatrix}$. Find the inner product $\langle u, v\rangle$. Now, find the inner product $\langle v, u\rangle$. Do you see why we have to conjugate?

Conjugate symmetry also guarantees that the inner product of any vector with itself is a real number. To see this, observe that for any vector v, conjugate symmetry ensures that $\langle v, v\rangle = \overline{\langle v, v\rangle}$ (we exchanged v with itself!), so the inner product of v with itself needs to be equal to its conjugate. A complex number $a + bi$ is equal to conjugate $a - bi$ iff $a + bi = a - bi$, which occurs iff $b = -b$. But $b = -b$ only in the case when $b = 0$, i.e., our complex number $a + bi$ was a real number all along.

The inner product is also *linear in the first argument*. So, first of all, for any two vectors u, v and w, the inner product satisfies the following equation:

$$\langle u + v, w\rangle = \langle u, w\rangle + \langle v, w\rangle \tag{12.113}$$

Let's see an example of this phenomenon. Let

$$u = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, v = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, w = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \tag{12.114}$$

Let's compute $\langle u + v, w \rangle$:

$$\begin{aligned} \langle u + v, w \rangle &= \left\langle \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\rangle = \left\langle \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\rangle \\ &:= \overline{1} \cdot 1 + \overline{1} \cdot 2 = 1 \cdot 1 + 1 \cdot 2 = 3 \end{aligned} \tag{12.115}$$

12.116 **Exercise** Now, you compute

$$\langle u, w \rangle + \langle v, w \rangle = \left\langle \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\rangle + \left\langle \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\rangle$$

and check that it equals $1 + 2 = 3$.

We said that the inner product is *linear* in the first argument, so you should be wondering how scalar multiplication manifests itself here. The second part of this linearity in the first argument is that for any scalar a and any vectors u and v, we have

$$\langle a \cdot u, v \rangle = a \cdot \langle u, v \rangle \tag{12.117}$$

To see this property in action, let

$$a = 2, u = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, v = \begin{pmatrix} 3 \\ 4 \end{pmatrix} \tag{12.118}$$

Let's compute the left-hand side

$$\begin{aligned} \langle a \cdot u, v \rangle &= \left\langle 2 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right\rangle = \left\langle \begin{pmatrix} 2 \\ 4 \end{pmatrix}, \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right\rangle \\ &= 2 \cdot 3 + 4 \cdot 4 = 6 + 16 = 22 \end{aligned} \tag{12.119}$$

12.120 **Exercise** Compute the right-hand side, i.e., $a \cdot \langle u, v \rangle$ and check that it's equal to 22.

You might wonder if $\langle u, a \cdot v \rangle = a \cdot \langle u, v \rangle$ for all scalars a and vectors u and v. Check that this is not true in the following exercise.

12.121 **Exercise** Check that it is *not* true that $\langle u, a \cdot v\rangle = a \cdot \langle u, v\rangle$ for all scalars a and vectors u and v by finding a scalar a and two vectors u and v such that $\langle u, a \cdot v\rangle \neq a \cdot \langle u, v\rangle$. In fact, you can prove using the axioms listed later on that for all scalars a and vectors u and v, it is true that $\langle u, a \cdot v\rangle = \overline{a}\langle u, v\rangle$. This property of the inner product is sometimes called *conjugate homogeneity in the second argument.*

The final property of the inner product is known as *positive-definiteness*: for all vectors v, $\langle v, v\rangle \geq 0$ and $\langle v, v\rangle = 0$ iff $v = 0$.

Let's see why this is true from a more abstract perspective. Let $v = \begin{pmatrix} v_1 & v_2 & \dots & v_n \end{pmatrix}^T$ be a vector.

Then

$$\langle v, v\rangle := \overline{v_1} \cdot v_1 + \overline{v_2} \cdot v_2 + ... + \overline{v_n} \cdot v_n \tag{12.122}$$

Recall that the product of a complex number with its conjugate is always a non-negative real number (i.e., greater than or equal to 0), so $\overline{v_i} \cdot v_i$ is a real number for each $i \in \{1, ..., n\}$. Well, then the sum of a (finite) bunch of non-negative real numbers is a real number, so the sum

$$\overline{v_1} \cdot v_1 + \overline{v_2} \cdot v_2 + ... + \overline{v_n} \cdot v_n \tag{12.123}$$

is still a non-negative real number, i.e., is greater than or equal to 0.

12.124 **Exercise** Convince yourself that the expression above is 0 iff the vector v is the zero vector, i.e., has all zero entries.

So, we've shown that for any vector v, $\langle v, v\rangle \geq 0$ and that $\langle v, v\rangle = 0$ iff $v = 0$.

12.125 **Exercise** Notice that we never said that the inner product satisfies linearity in the *second* argument! It is true, however, and your task is to prove it using the axioms that we've already listed and verified. Linearity in the first argument will be crucial for your proof.

The Kronecker Delta Function as an Inner Product

While discussing the idea of an orthonormal basis for a vector space earlier, we mentioned the idea that the mathematical description of an orthonormal basis could be described in terms of the *Kronecker delta function* δ. The Kronecker delta function δ has such a simple description that you can't imagine why it could possibly be named after anyone – although Kronecker isn't complaining.

12.126 **Definition** ***The Kronecker delta function***

For a set $\{1, 2, ..., n\}$

$$\begin{cases} \delta(i,j) = 1 & \text{if } i = j \\ \delta(i,j) = 0 & \text{if } i \neq j \end{cases}$$

This should look familiar! Recall our definition of orthonormal basis vectors from earlier. So, we can refer to an orthonormal set S of vectors as a set of vectors such that the restriction of the inner product to S is the Kronecker delta function!

12.127 **Exercise** Check that the entries of the identity matrix can be described by the Kronecker delta function. More precisely, check that we can define the identity matrix I entry-wise as $I_{ij} := \delta(i,j)$.

12.5 *Hermitian Operators*

A linear operator (interchangeably, matrix) is referred to as *Hermitian* iff it is equal to its conjugate transpose (defined above). Hermitian operators are crucial in quantum mechanics. We often want to measure a quantity associated with some state. Often the quantity we're after is the eigenvalue of an operator, and so we'd like for that eigenvalue to be a real number in order to effectively measure it.

Why We Can't Measure with Complex Numbers

A subtle point worth mentioning is that we can't perform a measurement with complex numbers. Suppose you wanted to figure out which of the numbers 0 or i is bigger? Then, you would either have to decide that $i = 0$, $0 < i$ or that

$i < 0$. Of course, we won't choose $i = 0$ because, if we do, then we could multiply each side of the equation $i = 0$ by $-i$ and find that $1 = 0$! Let's try the choice $0 < i$.

If we choose that $0 < i$, then multiplication by i on each side yields the inequality

$$i \cdot 0 < i \cdot i \tag{12.128}$$

But then

$$0 < i \cdot i = i^2 = -1 \tag{12.129}$$

In summary, we have deduced that $0 < -1$ from our assumption that $0 < i$. Since $0 < -1$ does not make sense, we are left to conclude that our original assumption that $0 < i$ does not make sense.

So, what if we choose that $i < 0$? Subtract i from both sides, revealing

$$i - i < 0 - i \tag{12.130}$$

So,

$$0 < -i \tag{12.131}$$

Multiplying both sides by $-i$ reveals

$$0 \cdot (-i) < -i \cdot (-i) \tag{12.132}$$

But

$$-\, i \cdot (-i) = (-1) \cdot i \cdot (-1) \cdot i \tag{12.133}$$

$$(-1)(-1) \cdot i \cdot i = 1 \cdot i^2 = -1 \tag{12.134}$$

So, the inequality $0 \cdot (-i) < -i \cdot (-i)$ really means $0 < -1$. This does not make sense either, and so we must conclude that the original hypothesis that $i < 0$ does not make sense. So, what we're discovering is that neither choice of order, i.e., neither $0 < i$ nor $i < 0$ makes sense. Assuming either such order reveals that $0 < -1$, which is simply not true. We are left to conclude that there is no acceptable way of ordering the complex numbers.

To make this more precise, suppose you want to come up with an order for the complex numbers that satisfies the following reasonable conditions that we are familiar with from our experience with usual real numbers:

- Trichotomy: For all complex numbers x, y, we have a trichotomy: either $x < y$, $x > y$ or $x = y$.
- Additivity property: For all complex numbers x, y, z, if $x < y$, then $x + z < y + z$.

- Multiplicative property: For all complex numbers x, y, z, if $0 < z$, then $x < y$ implies $xz < yz$.

Now, let's very carefully prove that the choice $0 < i$ contradicts at least one of the above axioms. Since $0 < i$, using the third (multiplicative) property stated above, we see that

$$0 < i \implies 0 \cdot i < i \cdot i = i^2 = -1 \tag{12.135}$$

Although the fact that we have deduced that $0 < -1$ from our assumption that $0 < i$ is disturbing, it technically does not violate any of the axioms above. Using the second (additivity) property above, the inequality $0 < -1$ allows us to deduce

$$0 < -1 \implies 0 + 1 < -1 + 1 = 0 \tag{12.136}$$

So, $1 < 0$. This is also disturbing, but is not a direct violation of any of the axioms above. Now, we invoke the third (multiplicative) property a final time yielding

$$1 < 0 \implies 1 \cdot i < 0 \cdot i = 0 \tag{12.137}$$

Then, we have $i < 0$, but our original assumption was that $0 < i$. This violates the first (trichotomy) property above!

12.138 **Exercise** Try to figure out what goes wrong with defining $i < 0$ in a similar fashion.

So, we really want these measurements to be real numbers, or else we have no way of making sense of them. The question then becomes: Why is it that Hermitian matrices have real eigenvalues?

Hermitian Operators Have Real Eigenvalues

Despite the ease with which we may verify that a matrix is equal to its conjugate transpose, a different but equivalent definition of Hermitian makes the proof that Hermitian matrices have real eigenvalues quite simple. We could also define a linear operator $T : V \to V$ to be Hermitian iff it is *self-adjoint*, where we say:

12.139 **Definition** ***Adjoint of a matrix***

The *adjoint* of a matrix T is a matrix T^* such that for all vectors $u, v \in V$ $\langle Tu, v\rangle = \langle u, T^*v\rangle$.

Then, that T is self-adjoint means that $T = T^*$ and so for all vectors $u, v \in V$, $\langle Tu, v\rangle = \langle u, Tv\rangle$. Notice that the adjoint of an operator is precisely its conjugate transpose! So, to say that an operator is self-adjoint is to say that it equals its conjugate transpose.

12.140 Adjoint is conjugate transpose

The adjoint of a linear operator is its conjugate transpose.

Now, if v is an eigenvector with eigenvalue λ of a Hermitian matrix T, then we may further conclude that

$$\langle Tv, v\rangle = \langle v, Tv\rangle \tag{12.141}$$

That v is an eigenvalue of T with eigenvalue λ means that $Tv = \lambda v$, so we may substitute λv for Tv in the equation $\langle Tv, v\rangle = \langle v, Tv\rangle$, yielding

$$\langle \lambda v, v\rangle = \langle v, \lambda v\rangle \tag{12.142}$$

Our intuition now is to factor out λ, which causes something interesting to happen. By homogeneity of the inner product in the first argument and conjugate homogeneity in the second argument, "factoring out λ" yields the new equation

$$\lambda\langle v, v\rangle = \overline{\lambda}\langle v, v\rangle \tag{12.143}$$

Now, we'd like to "divide by $\langle v, v\rangle$" on both sides and call it a day, but how do we know that it isn't zero? Well, we started by assuming v was an eigenvector of T, and the definition of an eigenvector requests that v not be zero. Then, we apply the definiteness property of the inner product to ensure that, since v is not zero, the inner product of v with itself, i.e., $\langle v, v\rangle$, is not zero either. Thus,

$$\lambda = \overline{\lambda} \tag{12.144}$$

Remember what it means for a complex number to equal its complex conjugate? Exactly — then it's a real number after all!

12.145 **Exercise** It's worth noting that any real symmetric matrix (symmetric matrix with real entries only) is Hermitian. Can you see why?

Hopefully, this foray into the proof of this theorem makes you realize that all of the axioms we request in these definitions are truly necessary! Do you see where we used each and every one of the axioms?

12.6 Unitary operators

You've surely encountered the zoo of matrices describing quantum logic gates within chapter 3 and listed in chapter 14. Those matrices are special because they are *unitary*. Unitary matrices preserve the length of vectors. They are so named because unit vectors remain unit vectors after an application of a unitary matrix.

More precisely,

12.146 Definition ***Unitary Operator***

A linear operator $U : V \to V$ is *unitary* iff for all vectors $u, v \in V$,

$$\langle x, y \rangle = \langle Ux, Uy \rangle$$

i.e., U preserves inner products.

It is interesting to note that we could equivalently define a unitary operator (matrix) $U : V \to V$ to be a matrix whose conjugate transpose is its inverse, i.e.,

$$U^{\dagger} = U^{\quad 1} \tag{12.147}$$

So,

$$U^{\dagger} U = U U^{\dagger} = I \tag{12.148}$$

Nothing we've said so far directly implies that a unitary operator preserves the length of any vector it operates on, so let's verify this in an exercise.

12.149 Exercise Verify that a unitary operator U preserves the length of any vector v that it operates on by using the definition given above. Use the fact that the length of the vector v is equal to $\sqrt{\langle v, v \rangle}$, and compare the original length of v, i.e., $\sqrt{\langle v, v \rangle}$, to the length of v after U has operated, i.e., $\sqrt{\langle Uv, Uv \rangle}$.

12.7 *The Direct Sum and the Tensor Product*

We might want to build bigger vector spaces from collections of vector spaces we already have. There are two popular ways of doing this. One way is the *direct sum*, the other the *tensor product*. We've already seen the tensor product in the context of taking the tensor product of two vectors, or even two matrices. There is a relationship between that previously defined operation and the operation we're about to describe.

The Direct Sum

The first point we'd like to emphasize is that the direct sum operates on *vector spaces*, not numbers, vectors or matrices (although the astute reader already considers all of these to be simply *linear transformations*!). The same is true for the tensor product. Specifically, the direct sum is a binary operation that accepts two vector spaces as input and yields another, usually "bigger," vector space as output. Let's see an example.

Consider the one-dimensional vector space $\mathbb{R}$ and a copy of itself – so, two copies of $\mathbb{R}$.

The direct sum of $\mathbb{R}$ with itself is written

$$\mathbb{R} \oplus \mathbb{R} \tag{12.150}$$

and is defined to be

$$\mathbb{R} \oplus \mathbb{R} := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} : x \in \mathbb{R}, y \in \mathbb{R} \right\} \tag{12.151}$$

Hold on – that's just the Cartesian product $\mathbb{R} \times \mathbb{R}$ of $\mathbb{R}$ and $\mathbb{R}$!

12.152 **Exercise** Recall the definition of the Cartesian product $\mathbb{R} \times \mathbb{R}$ and compare it to the direct sum $\mathbb{R} \oplus \mathbb{R}$.

Why do we have two names for the same thing? Well, they're not actually the same thing because the direct sum $\mathbb{R} \oplus \mathbb{R}$ has more *structure*. In particular, the direct sum $\mathbb{R} \oplus \mathbb{R}$ is a vector space (over $\mathbb{R}$), where the Cartesian product $\mathbb{R} \times \mathbb{R}$ is simply a set.

To further elaborate on this idea, consider the sets $A := \{a, b, c\}$ and $B := \{d, e\}$. We may create their Cartesian product

$$\begin{aligned} A \times B &:= \{(x, y) : x \in A, y \in B\} \\ &= \{(a, d), (a, e), (b, d), (b, e), (c, d), (c, e)\} \end{aligned} \tag{12.153}$$

Now, we ask the question: What is the direct sum $A \oplus B$? Well, at first, we have

$$A \oplus B := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} : x \in A, y \in B \right\} \tag{12.154}$$

simply following the definition from above. A cursory glance might convince us that the issue is that the Cartesian product has row vectors and the direct sum has column vectors, but when discussing sets, it doesn't matter if we're using row or column vectors. The question is illuminated by a further question: How is $A \oplus B$ a vector space? And over which field?

You see, in order to define $A \oplus B$ as a vector space, you have to give it additional *algebraic structure* so that it is an *abelian group* and has a *field action* by some specified field. Currently, it has neither of these. You're invited to ponder this for a moment. How would you define the addition of two elements of $A \oplus B$?

Whatever way you decide has to allow for the addition of the specific elements $\begin{pmatrix} a & d \end{pmatrix}^T$ and $\begin{pmatrix} b & e \end{pmatrix}^T$. Sure, we could say

$$\begin{pmatrix} a \\ d \end{pmatrix} + \begin{pmatrix} b \\ e \end{pmatrix} = \begin{pmatrix} a + b \\ d + e \end{pmatrix} \tag{12.155}$$

but what do $a + b$ and $d + e$ even mean? Remember, a, b, d and e are just letters. They have no numerical meaning whatsoever.

So, the big idea is that it's perfectly reasonable to form Cartesian products of any two sets you like. For example, the Cartesian product of the set

$$\mathbb{COLORS} := \{\text{chartreuse, magenta, periwinkle}\} \tag{12.156}$$

and the set

$$\mathbb{ANIMALS} := \{\text{cat, dog, aardvark}\} \tag{12.157}$$

is the new set

$$\mathbb{COLORS} \times \mathbb{ANIMALS} := \{(x, y) : x \in \mathbb{COLORS}, y \in \mathbb{ANIMALS}\} \tag{12.158}$$

which ends up being

$$\left\{ \begin{array}{c} \text{(chartreuse, cat), (chartreuse, dog), (chartreuse, elephant),} \\ \text{(magenta, cat), (magenta, dog), (magenta, elephant),} \\ \text{(periwinkle, cat), (periwinkle, dog), (periwinkle, elephant)} \end{array} \right\} \tag{12.159}$$

which is just a set, and not a vector space. Whereas the direct sum $\mathbb{R} \oplus \mathbb{R}$ is, *a priori*, the set

$$\mathbb{R} \oplus \mathbb{R} := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} : x \in \mathbb{R}, y \in \mathbb{R} \right\} \tag{12.160}$$

which has a natural interpretation as being a vector space over the field $\mathbb{R}$, since we can define the addition (to get the abelian group structure) to be usual vector addition and the field action to be usual scalar multiplication.

We would also like to mention that since the direct sum of two vector spaces is itself a vector space, it has a dimension. It turns out that the dimension of the direct sum of two vector spaces is the sum of their dimensions, hence the name direct "sum." More specifically, for any two vector spaces V and W both over the same field $\mathbb{F}$,

$$\dim(V \oplus W) = \dim(V) + \dim(W) \tag{12.161}$$

12.162 **Definition** ***The direct sum of two vector spaces***

The *direct sum* of vector spaces V and W is the vector space

$$V \oplus W := \left\{ \begin{pmatrix} v \\ w \end{pmatrix} \right\}$$

12.163 Dimension of direct sum

$$\dim(V \oplus W) = \dim(V) + \dim(W) \tag{12.164}$$

The Tensor Product

Now, we'll discuss the tensor product. What is remarkable about the tensor product of two vector spaces is that it is entirely described by the "tensor product" of the basis elements for each. What we mean by this is that if we have two vectors spaces V and W both over the same field $\mathbb{F}$ and with bases $\mathscr{B}_V = \{v_1, ..., v_m\}$ and $\mathscr{B}_W = \{w_1, ..., w_n\}$, respectively, then the tensor product of V and W, denoted $V \otimes W$ is a vector space with basis given by

$$\mathscr{B}_{V\otimes W} := \{v_1 \otimes w_1, v_1 \otimes w_2, ..., v_1 \otimes w_n, \ldots, v_m \otimes w_1, v_m \otimes w_2, \ldots, \otimes v_m \otimes w_n\} \quad (12.165)$$

You might recognize that all of the possible pairs of tensor products of the basis vectors of V with the basis vectors of W appear in the "tensor product" of the bases of V and of W. In fact, this is always the case! That is, if we have two vector spaces V and W both over the same field $\mathbb{F}$ and we'd like to find their tensor product $V \otimes W$, we can at least determine the basis of $V \otimes W$ by tensoring all of the possible pairs of basis vectors of V and W. Although this is a convenient fact, it does require a little bit of work to prove; it is often referred to as *The Tensor Product Basis Theorem*, emphasizing that it is a result to be proved.

What we would have to show is that, in fact, the proposed basis for $V \otimes W$ consisting of all possible tensor products of pairs of basis vectors from V and W is actually a basis, i.e., all of those tensor products are linearly independent and span $V \otimes W$. This is believable, given that the basis vectors from each of V and W are linearly independent amongst themselves and they span their respective spaces. However, we would like you to think about why this requires a proof!

Anyway, we'll avoid defining the tensor product of two vectors from an axiomatic perspective here, and instead give a basis-centric view of the idea. If you've never seen the tensor product of two vector spaces before, it's fair to think like this for the moment:

Given two vector spaces V and W with bases $\mathscr{B}_V$ and $\mathscr{B}_W$, respectively, we define their *tensor product* $V \otimes W$ to be the vector space with basis equal to the "tensor product" of their bases, i.e., the basis $\mathscr{B}_{V\otimes W}$ consisting of the tensor products of all possible pairs of basis elements of V and W, as described above.

Mathematicians reading this might be frustrated, but we're simply trying to give a working definition. Readers interested in why there might be any fuss about this definition of a tensor product are invited to investigate more mathematical treatments of the tensor product from the perspective of bilinear maps, and even further, category theory![3]

[3] Category theory defines the tensor product of two vector spaces via its *universal property*. This definition offers an elegant and sophisticated view of the tensor product which eases proofs and offers insight into the behavior of the tensor product as it interacts with other vector spaces. This construction also instructs the definition of the tensor product of more general objects than vector spaces, like modules!

Now, a word about notation. You might come across some fancy ways of writing the direct sum or tensor product of several copies of the same vector space. For example,

$$\mathscr{H}^{\bigoplus n} \tag{12.166}$$

is just a fancy way of writing

$$\underbrace{\mathscr{H} \oplus \ldots \oplus \mathscr{H}}_{\text{n times}} \tag{12.167}$$

Another way to state this is: the direct sum of n copies of the vector space $\mathscr{H}$. This can also be written as

$$\bigoplus_{n} \mathscr{H} \tag{12.168}$$

A similar notation is in place for tensor products

$$\mathscr{H}^{\bigotimes n} := \underbrace{\mathscr{H} \otimes \ldots \otimes \mathscr{H}}_{\text{n times}} \tag{12.169}$$

and

$$\bigotimes_{n} \mathscr{H} \tag{12.170}$$

Both of these notations refer to the n-fold tensor product of $\mathscr{H}$, i.e., the tensor product of n copies of the vector space $\mathscr{H}$. These notations for the tensor product of several $\mathscr{H}$s arise when notating a quantum register, i.e., a collection of qubits, as we'll see at the culmination of this chapter with the definition of a Hilbert space.

We can also take the tensor product of two operators. Thinking of each operator as a linear transformation, and thus a matrix, allows us to take their tensor product as we did earlier in this chapter! For example, given the operators on two-dimensional complex space $\mathbb{C}^2$,

$$I := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{12.171}$$

and

$$X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{12.172}$$

we may form their tensor product

$$
\begin{aligned}
I \otimes X &:= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes X = \begin{pmatrix} 1 \cdot X & 0 \cdot X \\ 0 \cdot X & 1 \cdot X \end{pmatrix} \\
&= \begin{pmatrix} 1 \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & 0 \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ 0 \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & 1 \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} 1 \cdot 0 & 1 \cdot 1 \\ 1 \cdot 1 & 1 \cdot 0 \end{pmatrix} & \begin{pmatrix} 0 \cdot 0 & 0 \cdot 1 \\ 0 \cdot 1 & 0 \cdot 0 \end{pmatrix} \\ \begin{pmatrix} 0 \cdot 0 & 0 \cdot 1 \\ 0 \cdot 1 & 0 \cdot 0 \end{pmatrix} & \begin{pmatrix} 1 \cdot 0 & 1 \cdot 1 \\ 1 \cdot 1 & 1 \cdot 0 \end{pmatrix} \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}
\end{aligned}
\tag{12.173}
$$

Summarizing the result of our computation, we've created an operator that now operates on *four*-dimensional complex space $\mathbb{C}^4$ with the following effect

$$
\begin{aligned}
(I \otimes X)(|00\rangle) &= (I \otimes X)(|0\rangle \otimes |0\rangle) \\
&= \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\
&= |0\rangle \otimes |1\rangle = |01\rangle
\end{aligned}
\tag{12.174}
$$

12.175 **Exercise** Find the tensor product $X \otimes I$. Is it the same as $I \otimes X$?

Note that we constructed a 4 × 4 matrix as the tensor product of two 2 × 2 matrices, which makes sense given our earlier discussion of how the tensor product of an $(a \times b)$ matrix and a $(c \times d)$ matrix is a $(a \cdot c) \times (b \cdot d)$ matrix.

Curious readers might try to build the quantum operators discussed in chapter 3 by taking tensor products of two-dimensional operators. Beware: it's not easy! In fact, you cannot build *CNOT* by taking the tensor product of two two-dimensional operators.

12.8 *Hilbert Space*

The modern definition of a quantum computer laid out in the text postulates that a qubit is modeled by a two-dimensional complex Hilbert space, so we include the formal definition here. However, before we can do so, we need to have a brief discussion of the notions of a *metric* on a vector space, *Cauchy* sequences, and finally, the idea of *completeness* of a vector space.

Metrics, Cauchy Sequences and Completeness

By a *metric* on a vector space, we mean that we can measure (hence, "metric") the distance between two vectors u and v by computing the norm of their difference, i.e.,

$$\langle u - v, u - v \rangle \tag{12.176}$$

12.177 **Exercise** Check that the distance between two vectors u and v as defined above is zero iff $u = v$, equivalently, $u - v = 0$.

At first glance, a Cauchy[4] sequence can be thought of as "a sequence of numbers whose terms get as close as we like, given that we look far enough." Before we investigate Cauchy sequences, we should understand the notion of *convergence* of a sequence of numbers. Here is an example of a Cauchy sequence of real numbers $f : \mathbb{N} \to \mathbb{R}$:

$$\begin{array}{c|c}
f(1) & \frac{1}{2} \\
f(2) & \frac{1}{4} \\
f(3) & \frac{1}{8} \\
f(4) & \frac{1}{16} \\
\vdots & \vdots \\
f(n) & \frac{1}{2^n}
\end{array}$$

We can see that these numbers are getting closer and closer to the number 0, and in fact, we can make them as close to 0 as we like by choosing terms sufficiently far along in the sequence.

[4] This sequence is named after Augustin-Louis Cauchy.

12.178 **Exercise** Verify for yourself that we can make the numbers in the sequence above as close to the number 0 as we like by finding the number n such that $f(n) = \frac{1}{2^n}$ is close to 0 within an error of 0.000001. In other words, figure out how far we need to go in the sequence before the terms differ from 0 by only 0.000001 or less. It might help to think of 0.000001 as $\frac{1}{10^6}$.

We say that this sequence of numbers *converges* to the number 0. We can express that the above sequence $f : \mathbb{N} \to \mathbb{R}$ of real numbers defined by $f(n) = \frac{1}{2^n}$ converges to 0 more mathematically by saying,

for all $\epsilon > 0$, *there exists* $N \in \mathbb{N}$ *such that for all* $n \in \mathbb{N}$, *if* $n > N$, *then* $|f(n) - 0| < \epsilon$.

You actually found the "N" in the previous mathematical description of the convergence of a sequence to 0 in the above exercise!

Of course, there is nothing special about this particular sequence, nor the number 0. We can play this game with *any* sequence and number. In general, we say that a sequence $f : \mathbb{N} \to \mathbb{R}$ *converges to a number* L (L stands for "limit") iff

for all $\epsilon > 0$ *there exists* $N \in \mathbb{N}$ *such that for all* $n \in \mathbb{N}$, *if* $n > N$, *then* $|f(n) - L| < \epsilon$.

Again, intuitively, we should think of this as saying "A sequence of numbers converges to a number L iff we can make those numbers get as close to L as we like simply by taking terms further along in the sequence."

Now, a Cauchy sequence is a special type of sequence whose terms, as we said earlier, "get as close as we like, given that we look far enough." Let's revisit the previous sequence

$$\begin{array}{c|c} f(1) & \frac{1}{2} \\ f(2) & \frac{1}{4} \\ f(3) & \frac{1}{8} \\ f(4) & \frac{1}{16} \\ \vdots & \vdots \\ f(n) & \frac{1}{2^n} \end{array}$$

Observe the following phenomenon: How far apart are the terms $f(1)$ and $f(2)$? Well,

$$|f(1) - f(2)| = \left|\frac{1}{2} - \frac{1}{4}\right| = \left|\frac{1}{4}\right| = \frac{1}{4}$$

so they're $\frac{1}{4}$ apart. How far apart are the terms $f(2)$ and $f(3)$? You should check that they're $\frac{1}{8}$ apart. $f(3)$ and $f(4)$? Now, they're $\frac{1}{16}$ apart! Hmm... So, it seems that as we go further along in the sequence, pairs of terms get closer and closer together. Let's make this mathematically precise.

We say that a sequence $f : \mathbb{N} \to \mathbb{R}$ is *Cauchy* iff

for all $\epsilon > 0$, there exists $N \in \mathbb{N}$ such that for all $m, n \in \mathbb{N}$, if $m, n > N$, then $|f(m) - f(n)| < \epsilon$.

12.179 **Exercise** Convince yourself that this precise mathematical formulation of a Cauchy sequence agrees with our intuitive statement about terms getting closer further along in the sequence. Then, check that the sequence $f(n) = \frac{1}{2^n}$ is in fact a Cauchy sequence.

So, we're convinced that the above sequence is Cauchy. Now, we can ask the question as to whether a sequence of numbers converges to a number already in our set. You might be wondering how a sequence of numbers could ever possibly converge to something not already in our set... Well, let's look at the sequence $f(n) = \frac{1}{2^n}$ once more.

Suppose we're only considering numbers greater than 0 for the moment. In other words, we're working with the numbers

$$(0, \infty) := \{x \in \mathbb{R} : x > 0\}$$

So, the numbers in the set we're considering cannot be equal to 0. Well, then the previous sequence has the curious property that it is Cauchy *and* it converges, but not to any number in the space we're working in, because it converges to 0, which is not a number in our set! This is a failure of the set $(0, \infty)$ to be what we refer to as *complete*. Formally, we say that a set is *complete* iff any Cauchy sequence of elements in the set converges to an element in the set. We often relax this statement to "Cauchy sequences converge" when the context is understood.

Now, we can generalize all of these ideas by allowing ourselves to measure the distance between two objects using a more general notion of an absolute value. The absolute value is actually a special case of a more general phenomenon known as a *metric*:

12.180 **Definition** ***Definition of a metric***

A *metric* is a binary function $d : S \times S \to \mathbb{R}$ from the Cartesian product of two copies of a set S to the real numbers satisfying the following properties:

- For all $x, y \in S$, $d(x, y) \geq 0$
- Definiteness: For all $x, y \in S$, $d(x, y) = 0$ iff $x = y$
- Triangle inequality: For all $x, y, z \in S$, $d(x, z) \leq d(x, y) + d(y, z)$

The absolute value $|\cdot| : \mathbb{R} \to \mathbb{R}$ is in fact a metric, as we invite you to check:

12.181 **Exercise** Check that the absolute value $|\cdot| : \mathbb{R} \to \mathbb{R}$ is actually a metric by verifying that it satisfies the above three properties. Also, convince yourself that the third property really should be called the triangle inequality!

But we said that a metric is a more general notion! What's a more general metric? Well, consider the following way of measuring the "distance" between two vectors: Given two vectors u and v in $\mathbb{R}^2$, we could measure the distance between them by taking the (L^2) norm of their difference (as vectors, that is), i.e., $d(u, v) := ||u - v||_2$. But a moment's thought makes us realize that this is simply taking the square root of the inner product of the vector difference with itself, i.e.,

$$||u - v||_2 = \sqrt{\langle u - v, u - v\rangle}$$

There we have it! A very general notion of a metric. We can now define a metric on any vector space with an inner product in the following way. For any vector space V with inner product $\langle \cdot, \cdot \rangle$, define the metric

$$d : V \times V \to \mathbb{R} \tag{12.182}$$

via

$$d(u, v) := \sqrt{\langle u - v, u - v\rangle} \tag{12.183}$$

We invite you to check that this definition does in fact satisfy the criteria for a metric from earlier.

12.184 **Exercise** Check that the metric we defined above

$$d : V \times V \to \mathbb{R}$$

via

$$d(u, v) := \sqrt{u - v, u - v}$$

is actually a metric by checking that it satisfies the criteria laid out earlier.

We say then that the inner product *induces* a metric on the vector space V, or that the metric is *the metric induced by the inner product*.

An Axiomatic Definition of the Inner Product

Let is touch briefly upon the idea of an axiomatic definition of an inner product. The definition given previously in this text serves us well, but it turns out that there are more exotic ways of taking the inner product of two vectors. In fact, we could possibly be working in a vector space where the vectors are... matrices or something some other mathematical object. There are vector spaces, for example, whose "vectors" consist of all continuous functions $f : \mathbb{R} \to \mathbb{R}$, where the inner product of two "vectors" (really, functions!) f and g is defined to be

$$\langle f, g \rangle := \int_0^1 f(x)g(x)dx \tag{12.185}$$

Yes, inner products, and vectors spaces for that matter, can be quite otherworldly. So, we want an axiomatic framework for them that captures the essence.

12.186 **Definition** ***Axiomatic definition of an inner product***

An *inner product* $\langle \cdot, \cdot \rangle$ on a vector space V over a field $\mathbb{F}$ (where $\mathbb{F}$ is either $\mathbb{R}$ or $\mathbb{C}$) is a binary function

$$\langle \cdot, \cdot \rangle : V \times V \to \mathbb{F}$$

satisfying the following properties:

- Conjugate symmetry: For all $u, v \in V$,

$$\langle u, v \rangle = \overline{\langle v, u \rangle}$$

- Linearity in the first argument: For all $a \in \mathbb{F}, u, v, w \in V$,

$$\langle a \cdot u, v \rangle = a \cdot \langle u, v \rangle$$

  and

$$\langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle$$

- Positive-definiteness: For all $v \in V$,

$$\langle v, v \rangle \geq 0$$

  and

$$\langle v, v \rangle = 0 \text{ iff } v = 0$$

12.187 **Exercise** With the axioms laid out above, you're invited to verify that the inner product which we have been using does in fact satisfy these axioms.

Finally, we're able to state the precise definition of a Hilbert space!

The Definition of Hilbert Space

12.188 **Definition** ***Definition of Hilbert space***

A *Hilbert space* is a vector space H over either the field of real or complex numbers equipped with an inner product $\langle \cdot, \cdot \rangle$ that is a complete metric space with respect to the metric induced by the inner product.

The definition of a Hilbert space is the culmination of this chapter, and we'd like to mention briefly how Hilbert spaces arise in quantum computing.[5] We care about Hilbert spaces over the field $\mathbb{C}$ of complex numbers, and the following discussion refers only to such spaces.

12.9 *The Qubit as a Hilbert Space*

One of the central concepts of quantum computing is that a qubit can be represented as a two-dimensional complex Hilbert space.[6] We notate a qubit with the letter H, or sometimes more fancily with $\mathscr{H}$, for "Hilbert."

12.189 A qubit is a vector space

A qubit is represented by a vector space — more specifically, a Hilbert space!

We sometimes call the Hilbert space representing the qubit the *state space*. A *state* in the state space is a vector in the state space with L^2 norm 1. So, a state is on the *Bloch sphere* – refer to chapter 3. For example, the familiar vectors $|0\rangle$ and $|1\rangle$ are states in the 2-dimensional Hilbert space (state space) $\mathscr{H}$, which represents one qubit. In fact, $|0\rangle$ and $|1\rangle$ are an orthonormal basis for $\mathscr{H}$, as you verified earlier in our discussion of orthonormal bases, and so every vector in the state space is a linear combination of these two vectors with L^2 norm 1. In the terminology of quantum mechanics, every state is a superposition of these two states!

We refer to a collection of n qubits as a *quantum register* and often notate it as

$$\underbrace{\mathscr{H} \otimes \mathscr{H} \otimes \ldots \otimes \mathscr{H}}_{\text{n times}} \tag{12.190}$$

It turns out that the tensor product of Hilbert spaces is another Hilbert space, although it will have a greater dimension in general!

12.191 **Exercise** Think about why the dimension of the tensor product of two two-dimensional Hilbert spaces is 4. Then, think about why the dimension of the tensor product of three two-dimensional Hilbert spaces is 8 – not 6! Then, figure out why the dimension of the tensor product of n two-dimensional

[5] The name *Hilbert* space honors the mathematician David Hilbert.

[6] We take our motivation from page 15 of Yuri Manin's seminal paper *[141]*.

Hilbert spaces is in fact 2^n. It might help to think of what the basis for the tensor product space is.

For example, if we have two qubits $\mathscr{H}$ and $\mathscr{H}$, each with orthonormal basis

$$\mathscr{B}_{\mathscr{H}} = \{|0\rangle, |1\rangle\} \tag{12.192}$$

then the quantum register

$$\mathscr{H} \otimes \mathscr{H} \tag{12.193}$$

is a Hilbert space of dimension 4 (as you checked in the exercise above) with basis

$$\begin{aligned} \mathscr{B}_{\mathscr{H}\otimes\mathscr{H}} &= \{|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle\} \\ &= \{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}, \end{aligned} \tag{12.194}$$

where we recall from earlier that $|00\rangle$ is just a convenient renaming of the tensor product $|0\rangle \otimes |0\rangle$, i.e.,

$$|00\rangle := |0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{12.195}$$

and the vectors $|01\rangle, |10\rangle, |11\rangle$ are defined similarly.

12.196 **Exercise** Figure out the basis $\mathscr{B}_{\mathscr{H}^{\otimes 3}}$ for $\mathscr{H}^{\otimes 3}$ where $\mathscr{H}$ has basis $\mathscr{B}_{\mathscr{H}} = \{|0\rangle, |1\rangle\}$. It should consist of $2^3 = 8$ vectors by the previous exercise. Can you figure out the basis for $\mathscr{H}^{\otimes n}$?

12.197 A quantum register is a tensor product

A quantum register consisting of n qubits is a 2^n-dimensional tensor product of vector spaces!

Quantum Computing	**Linear Algebra**	**Example**
qubit	two-dimensional complex Hilbert space	$\mathscr{H} = \text{span}_{\mathbb{C}}\{\|0\rangle, \|1\rangle\}$
n-qubit quantum register	n-fold tensor product of two-dimensional complex Hilbert spaces	$\mathscr{H}^{\otimes n}$
state space	vectors with L^2 norm 1 (Bloch sphere)	$\{v \in \mathscr{H} : \langle v\|v\rangle = 1\}$
definite state	orthonormal basis vector	$\|0\rangle, \|1\rangle$
superposition of states	linear combination of orthonormal basis vectors with L^2 norm 1 (on the Bloch sphere)	$\frac{1}{\sqrt{2}}\|0\rangle + \frac{1}{\sqrt{2}}\|1\rangle$
quantum logic gates	unitary operators	$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
measurement operators	projection operators	$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$

Figure 12.3: Relationship of quantum computing to linear algebra

12.198 Summary of the relationship between quantum computing and linear algebra

- A *qubit* can be represented by a two-dimensional complex *Hilbert space*. The *state* of the qubit is represented by a vector in the Hilbert space.
- More specifically, the vectors representing the states of a qubit have an L^2 *norm of* 1.
- The *quantum register* consisting of n qubits is a 2^n-dimensional complex Hilbert space composed of the *n-fold tensor product of the two-dimensional Hilbert spaces* representing the qubits.
- A *computational basis vector* of the space represents a *definite computational state* of the qubit.
- *A superposition of states is a linear combination of the computational basis vectors.*
- *Quantum logic gates* are *unitary operators*, which act on the state space. Since unitary operators preserve the norm of the vectors they act on, states are transformed to new states and not to just any vector in the space. Thus, we can build quantum circuits using unitary operators.

And so our review of linear algebra culminates with the table in Figure 12.3. Readers wanting a deeper understanding of linear algebra are encouraged to read:

- Sheldon Axler's *Linear Algebra Done Right* [18], where emphasis is placed on the theory and proof technique common to linear algebra, a sophisticated vantage point of matrices, the spectral theorem (which determines when the eigenvectors of a linear operator form a basis), and the determinant are presented.
- Michael Artin's *Algebra* [17], caters to a more mathematical audience. It begins with a wonderful discussion of elementary matrices and their role in Gaussian elimination and offers great perspective for group theory.
- Gilbert Strang's textbook *Linear Algebra and its Applications* [210] is great for applications of linear algebra to the sciences and is supplemented by free MIT OpenCourseWare material.
- See Appendix A and B of Rieffel and Polak's textbook, *Quantum Computing, A Gentle Introduction*, for connections between quantum mechanics and probability theory as well as a treatment of the Abelian hidden subgroup problem [186].
- For readers interested in learning more about abstract algebra, here are additional texts:
    - John Fraleigh's *A First Course in Abstract Algebra* [88]
    - Dummit and Foote's *Abstract Algebra* [73]
    - Joseph Rotman's *Advanced Modern Algebra* [191]
- A whimsical introduction to category theory can be found in the text *Conceptual Mathematics: A First Introduction to Categories* [130] by F. William Lawvere and Stephen H. Schanuel.
- Readers interested in acquiring a refined definition of the tensor product should look up Tai-Danae Bradley's blog *Math3ma* [45].
- More adventurous readers are invited to read Bradley's book *What is Applied Category Theory?* [46] to see that category theory is not simply a rephrasing of mathematics and is in fact quite useful for things like chemistry and natural language processing!
- Emily Riehl's *Category Theory in Context* [187] is a helpful introduction to category theory for advanced undergraduates and graduate students.

CHAPTER 13

Mathematical Tools for Quantum Computing III

13.1 Boolean Functions

13.1 **Definition** ***Boolean function***

A Boolean function f is a function from a Cartesian product

$$\{0,1\}^n \rightarrow \{0,1\}^m$$

where $\{0,1\}^n$ denotes the n-fold Cartesian product of the set $\{0,1\}$ with itself, i.e.,

$$\{0,1\}^n := \underbrace{\{0,1\} \times \ldots \times \{0,1\}}_{\boldsymbol{n} \text{ times}}$$

Cartesian products and functions are discussed in chapter 11, if you want to brush up on these terms. Boolean functions arise naturally in computing. For example, the Deutsch-Jozsa Algorithm discussed in chapter 7 involves the four Boolean functions

$$f_0, f_1, f_x, f_{\overline{x}} \tag{13.2}$$

each with domain and codomain $\{0,1\}$. So, in the case of these four functions, $n = 1$ and $m = 1$ when we compare them to the definition of a Boolean function given above.

Other examples of Boolean functions include NOT, AND, OR and XOR.

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_13

Interest Rate	Number of Times Compounded Annually	Amount in One Year
$\frac{100}{1}$ percent	1 time per year	2 dollars
$\frac{100}{2} = 50$ percent	2 times per year	2.25 dollars
$\frac{100}{3}$ percent	3 times per year	~ 2.37 dollars
$\frac{100}{4}$ percent	4 times per year	~ 2.44 dollars
$\frac{100}{5}$ percent	5 times per year	~ 2.49 dollars
$\frac{100}{100}$ percent	100 times per year	~ 2.70481 dollars
$\frac{100}{\infty}$ percent	∞ times per year	e dollars

Figure 13.1: The Banker's experiment

13.2 *Logarithms and Exponentials*

We recall some of the basic facts about logarithms here. We first consider the *natural logarithm* with a base of the number e. The number e is approximately 2.71 and appears in a menagerie of mathematical and scientific contexts. One interesting way to acquire e is via the following banker's thought experiment: Invest 1 dollar in a bank account at an interest rate of $\frac{100}{1}$ percent annually (once per year) and in one year, you'll earn 1 dollar, and thus have a total of 2 dollars after one year. Now, allow yourself to accrue interest twice a year, but at the compromised rate of $\frac{100}{2} = 50$ percent. After the first six months, you'll earn 50 cents, since 50 cents is 50 percent of the invested 1 dollar. After another six months, you'll earn 50 percent of the 1.50 dollars you have, and end up with 2.25 dollars. This is more than if you compounded only once per year. If you compound three times per year, you have the compromised rate of $\frac{100}{3}$ percent. You earn a little bit more at this rate, as you can check. If you allow yourself to compound "infinitely many times" at the compromised rate of "$\frac{100}{\infty}$" percent, you'll end up with e dollars!

This is summarized in the table in Figure 13.1.

$$\ln := \log_e : (0, \infty) \to (-\infty, \infty) \tag{13.3}$$

where "ln" is the natural logarithm.

The following equivalence guides all of the facts to come:

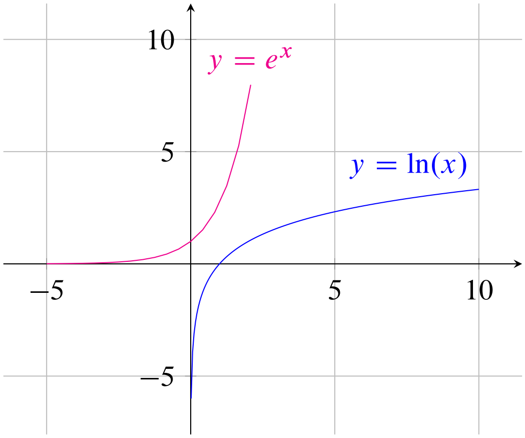


Figure 13.2: Graphs of the logarithm and exponential

13.4 Equivalence of logarithm and exponential

$$\ln(y) = x \iff e^x = y$$

Some first properties of the natural logarithm include

$$\ln(e^x) = x \qquad \text{and} \qquad e^{\ln(x)} = x \tag{13.5}$$

So, the functions $\ln(x)$ and e^x are inverse to one another.

Further, for any two positive real numbers a and b,

$$\ln(a \cdot b) = \ln(a) + \ln(b) \tag{13.6}$$

effectively because

$$e^a e^b = e^{a+b} \tag{13.7}$$

So, logarithms turn products into sums!

Using the above property, we can deduce the following for all positive real numbers a and b and any real number c:

- $\ln\left(\frac{b}{a}\right) = \ln(b) - \ln(a)$

  because $\frac{e^b}{e^a} = e^{b \quad a}$
- $\ln(a^c) = c \cdot \ln(a)$

  because $(e^a)^c = e^{ac}$

We also have a logarithm $\log_2$ with a base of 2. Similar to the natural logarithm,

$$\log_2(y) = x \iff 2^x = y \tag{13.8}$$

In fact, the natural logarithm could be written $\ln(x) = \log_e(x)$, i.e., with an explicitly mentioned base of e. There are logarithms for every positive real number base, so it's nice to be able to *base change*. To change the base of a logarithm from one base b to another base c, we can use the formula

$$\log_b(a) = \frac{\log_c(a)}{\log_c(b)} \tag{13.9}$$

One trick to remember this is that the a is above the b on the left-hand side, and that after changing the base to c, the a remains above the b on the right-hand side.

13.3 *Euler's Formula*

We will give an inkling of a proof via power series (avoiding issues of convergence for brevity) of the mysterious and wonderful formula of Euler, relating the polar coordinates of a complex number of unit norm to the number e, as mentioned in the previous chapter and depicted in Figure 13.3.

13.10 Euler's formula

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

The power series of the (complex-valued) cosine and sine functions are

$$\cos(z) = 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - \frac{z^6}{6!} + ... \tag{13.11}$$

and

$$\sin(z) = z - \frac{z^3}{3!} + \frac{z^5}{5!} - \frac{z^7}{7!} + ... \tag{13.12}$$

The power series of the (complex-valued) exponential function $e^z := \exp(z)$ is

$$e^z = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \frac{z^4}{4!} + ... \tag{13.13}$$

Then, evaluating the power series for the exponential function at the complex number iz yields

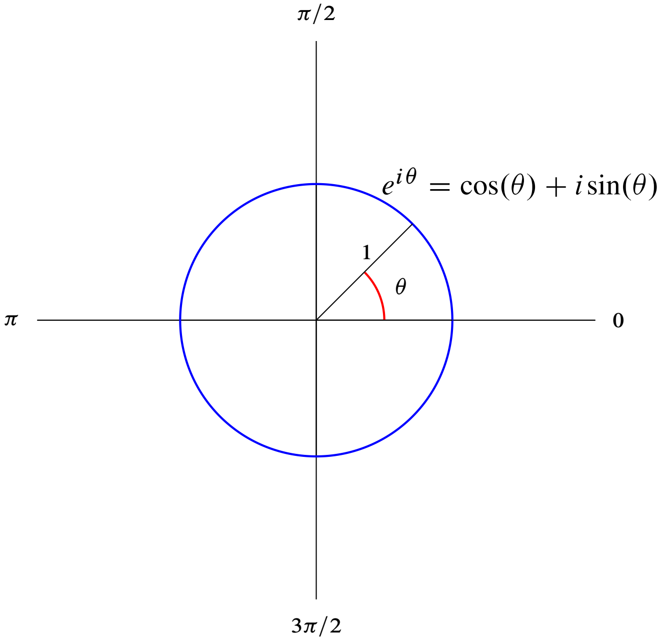


Figure 13.3: Euler's formula

$$\begin{aligned} e^{iz} &= 1 + (iz) + \frac{(iz)^2}{2!} + \frac{(iz)^3}{3!} + \frac{(iz)^4}{4!} + \dots \\ &= 1 + iz + \frac{i^2z^2}{2!} + \frac{i^3z^3}{3!} + \frac{i^4z^4}{4!} + \dots \end{aligned} \tag{13.14}$$

and remembering that $i^2 = -1, i^3 = -i$ and $i^4 = 1$ further yields

$$\begin{aligned} 1 + iz + \frac{i^2z^2}{2!} + \frac{i^3z^3}{3!} + \frac{i^4z^4}{4!} + \dots \\ &= 1 + iz + \frac{-1 \cdot z^2}{2!} + \frac{-iz^3}{3!} + \frac{z^4}{4!} + \dots \\ &= 1 + iz - \frac{z^2}{2!} - \frac{iz^3}{3!} + \frac{z^4}{4!} + \dots \end{aligned} \tag{13.15}$$

and after a little reorganization, we have

$$= \left(1 - \frac{z^2}{2!} + \frac{z^4}{4!} - ...\right) + \left(iz - \frac{iz^3}{3!} + \frac{iz^5}{5!} - ...\right) \tag{13.16}$$

Now, multiplying the power series for $\sin(z)$ by i yields

$$\begin{aligned} i\sin(z) &= i\left(z - \frac{z^3}{3!} + \frac{z^5}{5!} - \frac{z^7}{7!} + \dots\right) = iz - i\frac{z^3}{3!} + i\frac{z^5}{5!} - i\frac{z^7}{7!} + \dots \\ &= iz - \frac{iz^3}{3!} + \frac{iz^5}{5!} - \frac{iz^7}{7!} + \dots \end{aligned} \tag{13.17}$$

Adding the power series of $\cos(z)$ and the newly acquired expression for $i\sin(z)$ yields

$$\cos(z) + i\sin(z) = \left(1 - \frac{z^2}{2!} + \frac{z^4}{4!} - ...\right) + \left(iz - \frac{iz^3}{3!} + \frac{iz^5}{5!} - ...\right) \tag{13.18}$$

which is exactly the expression we found for e^{iz} earlier! Therefore,

$$e^{iz} = \cos(z) + i\sin(z) \tag{13.19}$$

Using Euler's formula, we have Euler's identity:

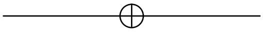
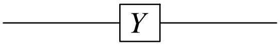
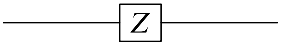
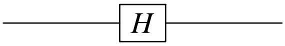
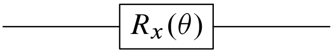
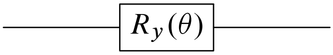
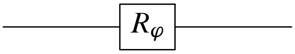
13.20 Euler's Identity

$$e^{i\pi} + 1 = 0$$

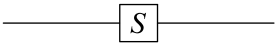
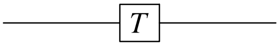
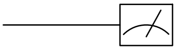
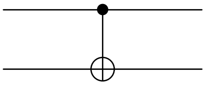
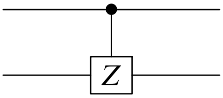
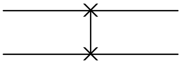
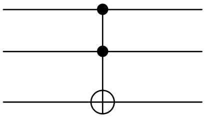
For further resources and additional mathematical tools, please refer to the book's GitHub site.

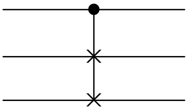
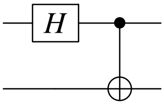
CHAPTER 14

Table of Quantum Operators and Core Circuits

$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	X	
$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	Y	
$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	Z	
$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	H	
$\begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$	$R_x(\theta)$	
$\begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$	$R_y(\theta)$	
$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{pmatrix}$	R_φ	

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0_14

$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	S	
$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$	T	
N/A	Meas.	
$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	*CNOT*	
$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$	*CZ*	
$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	*SWAP*	
$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$	Toffoli	

$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	Fredkin	
$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{pmatrix}$	Bell	
$HXH = Z$ $HZH = X$ $HYH = -Y$ $H^{\dagger} = H = H^{\quad 1}$ $X^2 = Y^2 = Z^2 = I$ $H = (X + Z)/\sqrt{2}$ $H^2 = I$ $SWAP_{12} = C_{12}C_{21}C_{12}$ $C_{12}X_1C_{12} = X_1X_2$ $C_{12}Y_1C_{12} = Y_1X_2$ $C_{12}Z_1C_{12} = Z_1$ $C_{12}X_2C_{12} = X_2$ $C_{12}Y_2C_{12} = Z_1Y_2$ $C_{12}Z_2C_{12} = Z_1Z_2$ $R_{z,1}(\theta)C_{12} = C_{12}R_{z,1}(\theta)$ $R_{x,2}(\theta)C_{12} = C_{12}R_{x,2}(\theta)$	Gate Identities ($C = CNOT$)	

Works Cited

[1] S. Aaronson. Certified randomness from quantum supremacy. Multiple unpublished talks.

[2] S. Aaronson. The Limits of Quantum Computers. *Scientific American*, March 2008.

[3] S. Aaronson. Quantum Complexity Theory. Fall 2010. Massachusetts Institute of Technology: MIT OpenCouseWare. https://ocw.mit.edu. License: Creative Commons BY-NC-SA, 2010.

[4] S. Aaronson. Read the fine print. *Nature Physics*, 11(4):291, 2015.

[5] S. Aaronson and A. Arkhipov. The computational complexity of linear optics. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2011.

[6] S. Aaronson, G. Kuperberg, C. Granade, and V. Russo. The Complexity Zoo. http://complexityzoo.uwaterloo.ca/Complexity_Zoo, 2005.

[7] C. Adami and N. J. Cerf. Quantum computation with linear optics. In *Quantum Computing and Quantum Communications*, pages 391–401. Springer, 1999.

[8] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[9] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani. Quantum walks on graphs. In *Proceedings of*

J. D. Hidary, *Quantum Computing: An Applied Approach*,
https://doi.org/10.1007/978-3-030-23922-0

the thirty-third annual ACM symposium on Theory of computing, pages 50–59. ACM, 2001.

[10] D. Aharonov and U. Vazirani. *Is quantum mechanics falsifiable? A computational perspective on the foundations of quantum mechanics*. Computability: Turing, Gödel, Church, and Beyond. MIT Press, 2013.

[11] Y. Aharonov, L. Davidovich, and N. Zagury. Quantum random walks. *Physical Review A*, 48(2):1687–1690, Aug 1993.

[12] S. Allen, J. Kim, D. L. Moehring, and C. R. Monroe. Reconfigurable and programmable ion trap quantum computer. In *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–3, Nov 2017.

[13] A. Almheiri, X. Dong, and D. Harlow. Bulk locality and quantum error correction in ads/cft. *Journal of High Energy Physics*, 2015(4):163, 2015.

[14] D. Anderson. The Anderson Group Website, JILA, CO. https://jila.colorado.edu/dzanderson/research-area-description/neutral-atom-quantum-computing, 2019.

[15] R. Anderson. Algorithm Analysis & Time Complexity Simplified. July 19, 2017 (https://bit.ly/2xH8mEr), 2017. Original URL: https://medium.com/@randerson112358/algorithm-analysis-time-complexity-simplified-cd39a81fec71.

[16] C. Arnold, J. Demory, V. Loo, A. Lemaître, I. Sagnes, M. Glazov, O. Krebs, P. Voisin, P. Senellart, and L. Lanco. Macroscopic rotation of photon polarization induced by a single spin. *Nature Communications*, 6:6236, 2015.

[17] M. Artin. *Algebra*. Pearson, 2010.

[18] S. Axler. *Linear Algebra Done Right*. Springer, 2014.

[19] D. M. Bacon. *Decoherence, Control, and Symmetry in Quantum Computers*. PhD thesis, University of California at Berkeley, 2001.

[20] P. Baireuther, T. E. O'Brien, B. Tarasinski, and C. W. Beenakker. Machine-learning-assisted correction of correlated qubit errors in a topological code. *Quantum*, 2:48, 2018.

[21] P. Ball. Ion-based commercial quantum computer is a first. *Physics World*, December 17, 2018 (https://bit.ly/2RrpZDf), 2018. Original URL: https://physicsworld.com/a/ion-based-commercial-quantum-computer-is-a-first/.

[22] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical review A*, 52(5):3457, 1995. arXiv: quant-ph/9503016.

[23] P. Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, 22(5):563–591, May 1980.

[24] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani. Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523, 1997.

[25] C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proc. IEEE Int. Conf. Computers, Systems, and Signal Processing*, volume 175, 1984.

[26] C. H. Bennett, 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. *Phys. Rev. Lett.*, 70:1895–1899, Mar 1993.

[27] C. H. Bennett and S. J. Wiesner. Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Phys. Rev. Lett.*, 69:2881–2884, Nov 1992.

[28] H. Bernien, B. Hensen, W. Pfaff, G. Koolstra, M. Blok, L. Robledo, T. Taminiau, M. Markham, D. Twitchen, L. Childress, et al. Heralded entanglement between solid-state qubits separated by three metres. *Nature*, 497(7447):86, 2013.

[29] E. Bernstein and U. Vazirani. Proceedings of the 25th annual ACM symposium on the theory of computing. *ACM, New York*, 11, 1993.

[30] M. K. Bhaskar, D. D. Sukachev, A. Sipahigil, R. E. Evans, M. J. Burek, C. T. Nguyen, L. J. Rogers, P. Siyushev, M. H. Metsch, H. Park, et al. Quantum nonlinear optics with a germanium-vacancy color center in a nanoscale diamond waveguide. *Physical Review Letters*, 118(22):223603, 2017.

[31] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd. Quantum machine learning. *Nature*, 549(7671):195, 2017.

[32] A. Blanco-Redondo, I. Andonegui, M. J. Collins, G. Harari, Y. Lumer, M. C. Rechtsman, B. J. Eggleton, and M. Segev. Topological optical waveguiding in silicon and the transition between topological and trivial defect states. *Physical Review Letters*, 116(16):163901, 2016.

[33] A. Blanco-Redondo, B. Bell, M. Segev, and B. Eggleton. Photonic quantum walks with symmetry protected topological phases. In *AIP Conference Proceedings*, volume 1874, page 020001. AIP Publishing, 2017.

[34] M. Blok, V. Ramasesh, J. Colless, K. O'Brien, T. Schuster, N. Yao, and I. Siddiqi. Implementation and applications of two qutrit gates in supercon-

ducting transmon qubits. Presented at APS March Meeting, 2018.

[35] J. G. Bohnet, B. C. Sawyer, J. W. Britton, M. L. Wall, A. M. Rey, M. Foss-Feig, and J. J. Bollinger. Quantum spin dynamics and entanglement generation with hundreds of trapped ions. *Science*, 352(6291):1297–1301, 2016.

[36] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595–600, Jun 2018. arXiv: 1608.00263.

[37] A. D. Bookatz. QMA-complete problems. *Quantum Information & Computation*, 14(5&6):361–383, 2014.

[38] M. Born. Zur Quantenmechanik der Stoßvorgänge. *Z. Phys.*, 37(12):863–867, 1926.

[39] M. Born. Das adiabatenprinzip in der quantenmechanik. *Zeitschrift für Physik A Hadrons and Nuclei*, 40(3):167–192, 1927.

[40] M. Born and V. Fock. Beweis des adiabatensatzes (1928, English Translation). In L. Faddeev, L. Khalfin, and I. Komarov, editors, *V.A. Fock – Selected Works: Quantum Mechanics and Quantum Field Theory*. Chapman & Hall/CRC, 2004.

[41] D. Boschi, S. Branca, F. De Martini, L. Hardy, and S. Popescu. Experimental realization of teleporting an unknown pure quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.*, 80:1121–1125, Feb 1998.

[42] V. Bouchiat, D. Vion, P. Joyez, D. Esteve, and M. Devoret. Quantum coherence with a single cooper pair. *Physica Scripta*, 1998(T76):165, 1998.

[43] A. Bouland, B. Fefferman, C. Nirkhe, and U. Vazirani. Quantum supremacy and the complex-

ity of random circuit sampling. *arXiv preprint arXiv:1803.04402*, 2018.

[44] P. O. Boykin, T. Mor, M. Pulver, V. Roychowdhury, and F. Vatan. On universal and fault-tolerant quantum computing. *arXiv preprint quant-ph/9906054*, 1999.

[45] T.-D. Bradley. Math3ma. https://www.math3ma.com, 2015.

[46] T.-D. Bradley. What is Applied Category Theory? *arXiv preprint arXiv:1809.05923*, 2018.

[47] S. Bravyi, D. Gosset, and R. Koenig. Quantum advantage with shallow circuits. *Science*, 362(6412):308–311, 2018.

[48] S. Bravyi, D. Gosset, R. Koenig, and M. Tomamichel. Quantum advantage with noisy shallow circuits in 3d. *arXiv preprint arXiv:1904.01502*, 2019.

[49] S. B. Bravyi and A. Y. Kitaev. Quantum codes on a lattice with boundary. *arXiv preprint quant-ph/9811052*, 1998.

[50] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage. Trapped-ion quantum computing: Progress and challenges. *arXiv preprint arXiv:1904.04178*, 2019.

[51] Y. Cao, G. G. Guerreschi, and A. Aspuru-Guzik. Quantum neuron: an elementary building block for machine learning on quantum computers. *arXiv preprint arXiv:1711.11240*, 2017.

[52] J. Carolan, C. Harrold, C. Sparrow, E. Martín-López, N. J. Russell, J. W. Silverstone, P. J. Shadbolt, N. Matsuda, M. Oguma, M. Itoh, et al. Universal linear optics. *Science*, 349(6249):711–716, 2015.

[53] S. Castelletto, B. Johnson, V. Ivády, N. Stavrias, T. Umeda, A. Gali, and T. Ohshima. A silicon carbide room-temperature single-photon source. *Nature Materials*, 13(2):151, 2014.

[54] W.-J. Chen, M. Xiao, and C. T. Chan. Photonic crystals possessing multiple Weyl points and the experimental observation of robust surface states. *Nature Communications*, 7:13038, 2016.

[55] L. Childress and R. Hanson. Diamond nv centers for quantum computing and quantum networks. *MRS Bulletin*, 38(2):134–138, 2013.

[56] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 59–68. ACM, 2003.

[57] A. M. Childs, E. Farhi, and S. Gutmann. An example of the difference between quantum and classical random walks. *arXiv preprint arXiv: quant-ph/0103020v1*, Mar 2001.

[58] K. S. Chou, J. Z. Blumoff, C. S. Wang, P. C. Reinhold, C. J. Axline, Y. Y. Gao, L. Frunzio, M. Devoret, L. Jiang, and R. Schoelkopf. Deterministic teleportation of a quantum gate between two logical qubits. *Nature*, 561(7723):368, 2018.

[59] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Phys. Rev. Lett.*, 74:4091–4094, May 1995.

[60] Cirq Developers. Cirq documentation, 2018. https://cirq.readthedocs.io/en/latest/tutorial.html.

[61] P. J. Coles, S. Eidenbenz, S. Pakin, A. Adedoyin, J. Ambrosiano, P. Anisimov, W. Casper, G. Chennupati, C. Coffrin, H. Djidjev, et al. Quantum algorithm implementations for beginners. *arXiv preprint arXiv:1804.03719*, 2018.

[62] C. M. Dawson and M. A. Nielsen. The Solovay-Kitaev algorithm. *arXiv preprint quant-ph/0505030*, 2005.

[63] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill. Topological quantum memory. *Journal of Mathematical Physics*, 43(9):4452–4505, 2002.

[64] D. Dervovic, M. Herbster, P. Mountney, S. Severini, N. Usher, and L. Wossnig. Quantum linear systems algorithms: a primer. *arXiv preprint arXiv:1802.08227*, 2018.

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

[66] D. Deutsch. Lectures on Quantum Computation. http://www.quiprocone.org/Protected/DD_lectures.htm, 2003.

[67] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 439(1907):553–558, 1992.

[68] P. A. M. Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35(3):416–418, 1939.

[69] D. P. DiVincenzo. Topics in quantum computers. *arXiv preprint cond-mat/9612126*, 1996.

[70] D. P. DiVincenzo. The physical implementation of quantum computation. *Fortschritte der Physik: Progress of Physics*, 48(9-11):771–783, 2000. arXiv: quant-ph/0002077.

[71] M. W. Doherty, N. B. Manson, P. Delaney, F. Jelezko, J. Wrachtrup, and L. C. Hollenberg. The nitrogen-vacancy colour centre in diamond. *Physics Reports*, 528(1):1–45, 2013.

[72] L.-M. Duan and H. Kimble. Scalable photonic quantum computation through cavity-assisted inter-

actions. *Physical Review Letters*, 92(12):127902, 2004.

[73] D. S. Dummit and R. M. Foote. *Abstract Algebra*, volume 3. Wiley Hoboken, 2004.

[74] A. Einstein. Letter from Einstein to D. M. Lipkin. https://bit.ly/2CogEUC, July 1952. Accessed: 2018-05-19, Original URL: http://sethlipkin.com/collectibles/letters/letter1/letter%201%20-%20page%201.jpg.

[75] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Phys. Rev.*, 47:777–780, May 1935.

[76] R. E. Evans, M. K. Bhaskar, D. D. Sukachev, C. T. Nguyen, A. Sipahigil, M. J. Burek, B. Machielse, G. H. Zhang, A. S. Zibrov, E. Bielejec, et al. Photon-mediated interactions between quantum emitters in a diamond nanocavity. *Science*, 362(6415):662–665, 2018.

[77] E. Farhi, J. Goldstone, and S. Gutmann. A quantum algorithm for the hamiltonian NAND tree. *arXiv preprint quant-ph/0702144*, 2007.

[78] E. Farhi, J. Goldstone, and S. Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[79] E. Farhi, J. Goldstone, and S. Gutmann. A quantum approximate optimization algorithm applied to a bounded occurrence constraint problem. *arXiv preprint arXiv:1412.6062*, 2014.

[80] E. Farhi and S. Gutmann. Analog analogue of a digital quantum computation. *Physical Review A*, 57(4):2403, 1998.

[81] E. Farhi and S. Gutmann. Quantum computation and decision trees. *Physical Review A*, 58(2):915–928, Aug 1998. arXiv: quant-ph/9706062.

[82] E. Farhi and H. Neven. Classification with quantum neural networks on near term processors. *arXiv preprint arXiv:1802.06002*, 2018.

[83] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, Jun 1982.

[84] R. P. Feynman. *Feynman Lectures on Computation*. CRC Press, 2000.

[85] M. Fingerhuth, T. Babej, and P. Wittek. Open source software in quantum computing. *PLOS One*, 13(12):e0208561, 2018.

[86] C. Flühmann, T. L. Nguyen, M. Marinelli, V. Negnevitsky, K. Mehta, and J. Home. Encoding a qubit in a trapped-ion mechanical oscillator. *Nature*, 566(7745):513, 2019.

[87] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A*, 86:032324, Sep 2012.

[88] J. B. Fraleigh. *A First Course in Abstract Algebra*. Pearson, 2002.

[89] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, Apr 1982.

[90] M. Freedman, A. Kitaev, M. Larsen, and Z. Wang. Topological quantum computation. *Bulletin of the American Mathematical Society*, 40(1):31–38, 2003. arXiv: quant-ph/0101025.

[91] E. S. Fried, N. P. Sawaya, Y. Cao, I. D. Kivlichan, J. Romero, and A. Aspuru-Guzik. qTorch: The quantum tensor contraction handler. *PloS one*, 13(12):e0208510, 2018.

[92] N. Friis, O. Marty, C. Maier, C. Hempel, M. Holzäpfel, P. Jurcevic, M. B. Plenio, M. Huber, C. Roos, R. Blatt, et al. Observation of entangled

states of a fully controlled 20-qubit system. *Physical Review X*, 8(2):021012, 2018.

[93] A. Frisk Kockum. *Quantum optics with artificial atoms*. Chalmers University of Technology, 2014.

[94] J. M. Gambetta, J. M. Chow, and M. Steffen. Building logical qubits in a superconducting quantum computing system. *NPJ Quantum Information*, 3(1):2, 2017.

[95] C. Gidney and M. Ekera. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *arXiv preprint https://scirate.com/arxiv/1905.09749*, May 2019.

[96] D. Gottesman. The Heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*, 1998.

[97] Grove Developers. Grove documentation, 2018. https://grove-docs.readthedocs.io/en/latest/vqe.html.

[98] L. K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 79:325–328, Jul 1997. arXiv: quant-ph/9706033.

[99] M. Hafezi, E. A. Demler, M. D. Lukin, and J. M. Taylor. Robust optical delay lines with topological protection. *Nature Physics*, 7(11):907, 2011.

[100] M. Hafezi, S. Mittal, J. Fan, A. Migdall, and J. Taylor. Imaging topological edge states in silicon photonics. *Nature Photonics*, 7(12):1001, 2013.

[101] D. Hanneke, J. P. Home, J. D. Jost, J. M. Amini, D. Leibfried, and D. J. Wineland. Realization of a programmable two-qubit quantum processor. *Nature Physics*, 6(1):13, 2010.

[102] G. H. Hardy and J. E. Littlewood. Some problems of diophantine approximation: Part II. The trigonometrical series associated with the elliptic

theta-functions. *Acta Mathematica*, 37:193–239, 1914.

[103] A. W. Harrow, A. Hassidim, and S. Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15):150502, 2009. arXiv: 0811.3171.

[104] A. W. Harrow and A. Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203, 2017.

[105] M. Z. Hasan and C. L. Kane. Colloquium: topological insulators. *Reviews of Modern Physics*, 82(4):3045, 2010.

[106] V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, and J. M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209, 2019.

[107] M. Hayashi and H. Zhu. Secure uniform random-number extraction via incoherent strategies. *Physical Review A*, 97(1):012302, 2018. arXiv: 1706.04009.

[108] P. Hayden, S. Nezami, X.-L. Qi, N. Thomas, M. Walter, and Z. Yang. Holographic duality from random tensor networks. *Journal of High Energy Physics*, 2016(11):9, 2016.

[109] B. Hensen, H. Bernien, A. E. Dréau, A. Reiserer, N. Kalb, M. S. Blok, J. Ruitenberg, R. F. Vermeulen, R. N. Schouten, C. Abellán, et al. Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres. *Nature*, 526(7575):682, 2015.

[110] C. Hepp, T. Müller, V. Waselowski, J. N. Becker, B. Pingault, H. Sternschulte, D. Steinmüller-Nethl, A. Gali, J. R. Maze, M. Atatüre, et al. Electronic structure of the silicon vacancy color center in diamond. *Physical Review Letters*, 112(3):036405, 2014.

[111] O. Higgott, D. Wang, and S. Brierley. Variational quantum computation of excited states. *arXiv preprint arXiv:1805.08138*, 2018.

[112] C.-K. Hong, Z.-Y. Ou, and L. Mandel. Measurement of subpicosecond time intervals between two photons by interference. *Physical Review Letters*, 59(18):2044, 1987.

[113] IBM Q team. IBM Q 16 Rueschlikon backend specification V1.1.0. https://github.com/Qiskit/ibmq-device-information/, 2019.

[114] S. Jordan. Quantum Algorithm Zoo. http://quantumalgorithmzoo.org/, 2011.

[115] M. Katanaev. Adiabatic theorem for finite dimensional quantum mechanical systems. *Russian Physics Journal*, 54(3):342–353, 2011.

[116] J. Kempe. Quantum random walks hit exponentially faster. *arXiv preprint quant-ph/0205083*, 2002.

[117] J. Kempe. Quantum random walks: an introductory overview. *Contemporary Physics*, 44(4):307–327, 2003. arXiv: quant-ph/0303081.

[118] I. Kerenidis and A. Prakash. Quantum recommendation systems. *arXiv preprint arXiv:1603.08675*, 2016.

[119] A. Y. Kitaev. Fault-tolerant quantum computation by anyons. *Annals of Physics*, 303(1):2–30, 2003. arXiv: quant-ph/9707021.

[120] E. Knill, R. Laflamme, and G. J. Milburn. A scheme for efficient quantum computation with linear optics. *Nature*, 409(6816):46, 2001.

[121] D. E. Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24, 1976.

[122] W. F. Koehl, B. B. Buckley, F. J. Heremans, G. Calusine, and D. D. Awschalom. Room temperature co-

herent control of defect spin qubits in silicon carbide. *Nature*, 479(7371):84, 2011.

[123] P. Kok, W. J. Munro, K. Nemoto, T. C. Ralph, J. P. Dowling, and G. J. Milburn. Linear optical quantum computing with photonic qubits. *Reviews of Modern Physics*, 79(1):135, 2007.

[124] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. A quantum engineer's guide to superconducting qubits. *arXiv preprint arXiv:1904.06560*, 2019.

[125] Y. E. Kraus, Y. Lahini, Z. Ringel, M. Verbin, and O. Zilberberg. Topological states and adiabatic pumping in quasicrystals. *Physical Review Letters*, 109(10):106402, 2012.

[126] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien. Quantum computers. *Nature*, 464(7285):45, 2010.

[127] V. Lahtinen and J. K. Pachos. A short introduction to topological quantum computation. *SciPost Physics*, 3(3):021, Sep 2017. arXiv: 1705.04103.

[128] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961.

[129] R. LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019.

[130] F. W. Lawvere and S. H. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, 2009.

[131] D. R. Leibrandt, J. Labaziewicz, V. Vuletić, and I. L. Chuang. Cavity sideband cooling of a single trapped ion. *Physical Review Letters*, 103(10):103001, 2009.

[132] S. Leichenauer. Cirq Bootcamp: QNN Colab, 2018.

[133] H. Levine, A. Keesling, A. Omran, H. Bernien, S. Schwartz, A. S. Zibrov, M. Endres, M. Greiner,

V. Vuletić, and M. D. Lukin. High-fidelity control and entanglement of rydberg-atom qubits. *Physical Review Letters*, 121(12):123603, 2018.

[134] F. Li, X. Huang, J. Lu, J. Ma, and Z. Liu. Weyl points and Fermi arcs in a chiral phononic crystal. *Nature Physics*, 14(1):30, 2018.

[135] C. Liu, M. G. Dutt, and D. Pekker. Single-photon heralded two-qubit unitary gates for pairs of nitrogen-vacancy centers in diamond. *Physical Review A*, 98(5):052342, 2018.

[136] S. Lloyd. A potentially realizable quantum computer. *Science*, 261(5128):1569–1571, 1993.

[137] M. Loceff. *A Course in Quantum Computing*. 2015.

[138] L. Lu, J. D. Joannopoulos, and M. Soljačić. Topological photonics. *Nat. Photonics*, 8:821–829, 2014.

[139] D. Lucas, C. Donald, J. P. Home, M. McDonnell, A. Ramos, D. Stacey, J.-P. Stacey, A. Steane, and S. Webster. Oxford ion-trap quantum computing project. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 361(1808):1401–1408, 2003.

[140] Y. Manin. *Computable and Non-Computable (in Russian)*. Sovetskoye Radio, Moscow, 1980.

[141] Y. I. Manin. Classical computing, quantum computing, and Shor's factoring algorithm. In *Séminaire Bourbaki : volume 1998/99, exposés 850-864*, number 266 in Astérisque, pages 375–404. Société mathématique de France, 2000.

[142] I. L. Markov, A. Fatima, S. V. Isakov, and S. Boixo. Quantum supremacy is both closer and farther than it appears. *arXiv:1807.10749 [quant-ph]*, Jul 2018. arXiv: 1807.10749.

[143] E. Martin-Lopez, A. Laing, T. Lawson, R. Alvarez, X.-Q. Zhou, and J. L. O'brien. Experimental real-

ization of Shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773, 2012.

[144] K. Mattle, H. Weinfurter, P. G. Kwiat, and A. Zeilinger. Dense coding in experimental quantum communication. *Phys. Rev. Lett.*, 76:4656–4659, Jun 1996.

[145] R. Maurand, X. Jehl, D. Kotekar-Patil, A. Corna, H. Bohuslavskyi, R. Laviéville, L. Hutin, S. Barraud, M. Vinet, M. Sanquer, et al. A cmos silicon spin qubit. *Nature Communications*, 7:13575, 2016.

[146] J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven. Barren plateaus in quantum neural network training landscapes. *Nature Communications*, 9(1):4812, 2018.

[147] J. R. McClean, Z. Jiang, N. C. Rubin, R. Babbush, and H. Neven. Decoding quantum errors with subspace expansions. *arXiv preprint arXiv:1903.05786*, 2019.

[148] J. R. McClean, M. E. Kimchi-Schwartz, J. Carter, and W. A. de Jong. Hybrid quantum-classical hierarchy for mitigation of decoherence and determination of excited states. *Physical Review A*, 95(4):042308, 2017.

[149] J. R. McClean, I. D. Kivlichan, K. J. Sung, D. S. Steiger, Y. Cao, C. Dai, E. S. Fried, C. Gidney, B. Gimby, P. Gokhale, et al. Openfermion: the electronic structure package for quantum computers. *arXiv preprint arXiv:1710.07629*, 2017.

[150] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.

[151] N. D. Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007.

[152] A. Milsted and G. Vidal. Tensor networks as conformal transformations. *arXiv preprint arXiv:1805.12524*, 2018.

[153] S. Mittal, S. Ganeshan, J. Fan, A. Vaezi, and M. Hafezi. Measurement of topological invariants in a 2d photonic system. *Nature Photonics*, 10(3):180, 2016.

[154] S. Mittal and M. Hafezi. Topologically robust generation of correlated photon pairs. *arXiv preprint arXiv:1709.09984*, 2017.

[155] M. Mohseni, P. Read, H. Neven, S. Boixo, V. Denchev, R. Babbush, A. Fowler, V. Smelyanskiy, and J. Martinis. Commercialize quantum technologies in five years. *Nature News*, 543(7644):171, 2017.

[156] R. Movassagh. Efficient unitary paths and quantum computational supremacy: A proof of average-case hardness of random circuit sampling. *arXiv preprint arXiv: 1810.04681*, Oct 2018.

[157] Y. Nakamura, Y. A. Pashkin, and J. Tsai. Coherent control of macroscopic quantum states in a single-Cooper-pair box. *Nature*, 398(6730):786, 1999. arXiv: cond-mat/9904003.

[158] Y. Nakamura, Y. A. Pashkin, and J. S. Tsai. Rabi oscillations in a Josephson-junction charge two-level system. *Phys. Rev. Lett.*, 87:246601, Nov 2001.

[159] NAS. *Quantum Computing: Progress and Prospects*. The National Academies Press, Washington, DC, 2018.

[160] C. Neill, P. Roushan, K. Kechedzhi, S. Boixo, S. V. Isakov, V. Smelyanskiy, A. Megrant, B. Chiaro, A. Dunsworth, K. Arya, R. Barends, B. Burkett, Y. Chen, Z. Chen, A. Fowler, B. Foxen, M. Giustina, R. Graff, E. Jeffrey, T. Huang, J. Kelly, P. Klimov, E. Lucero, J. Mutus, M. Neeley, C. Quintana, D. Sank, A. Vainsencher, J. Wenner, T. C. White,

H. Neven, and J. M. Martinis. A blueprint for demonstrating quantum supremacy with superconducting qubits. *Science*, 360(6385):195–199, 2018.

[161] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[162] J. Noh, S. Huang, D. Leykam, Y. D. Chong, K. P. Chen, and M. C. Rechtsman. Experimental observation of optical Weyl points and Fermi arc-like surface states. *Nature Physics*, 13(6):611, 2017.

[163] N. Ofek, A. Petrenko, R. Heeres, P. Reinhold, Z. Leghtas, B. Vlastakis, Y. Liu, L. Frunzio, S. Girvin, L. Jiang, et al. Extending the lifetime of a quantum bit with error correction in superconducting circuits. *Nature*, 536(7617):441, 2016.

[164] J. Olson, Y. Cao, J. Romero, P. Johnson, P.-L. Dallaire-Demers, N. Sawaya, P. Narang, I. Kivlichan, M. Wasielewski, and A. Aspuru-Guzik. Quantum information and computation for chemistry. *arXiv preprint arXiv:1706.05413*, 2017.

[165] P. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, et al. Scalable quantum simulation of molecular energies. *Physical Review X*, 6(3):031007, 2016.

[166] J. Otterbach, R. Manenti, N. Alidoust, A. Bestwick, M. Block, B. Bloom, S. Caldwell, N. Didier, E. S. Fried, S. Hong, et al. Unsupervised machine learning on a hybrid quantum computer. *arXiv preprint arXiv:1712.05771*, 2017.

[167] T. Ozawa, H. M. Price, A. Amo, N. Goldman, M. Hafezi, L. Lu, M. Rechtsman, D. Schuster, J. Simon, O. Zilberberg, et al. Topological photonics. *arXiv preprint arXiv:1802.04173*, 2018.

[168] M. Ozols. Clifford group. *Essays at University of Waterloo, Spring*, 2008. https://bit.ly/2JZe2jO.

[169] D. P. Pappas, J. S. Kline, F. da Silva, and D. Wisbey. Coherence in superconducting materials for quantum computing. https://slideplayer.com/slide/7770332/.

[170] A. Peruzzo et al. A variational eigenvalue solver on a quantum processor. eprint. *arXiv preprint arXiv:1304.3061*, 2013.

[171] W. Pfaff, B. Hensen, H. Bernien, S. B. van Dam, M. S. Blok, T. H. Taminiau, M. J. Tiggelman, R. N. Schouten, M. Markham, D. J. Twitchen, et al. Unconditional quantum teleportation between distant solid-state quantum bits. *Science*, 345(6196):532–535, 2014.

[172] H. Pichler, S.-T. Wang, L. Zhou, S. Choi, and M. D. Lukin. Quantum optimization for maximum independent set using rydberg atom arrays. *arXiv preprint arXiv:1808.10816*, 2018.

[173] S. Prawer and A. D. Greentree. Diamond for quantum computing. *Science*, 320(5883):1601–1602, 2008.

[174] J. Preskill. Lecture notes for Physics 219/Computer Science 219 at Caltech: Quantum Computation. http://www.theory.caltech.edu/people/preskill/ph229, 1997.

[175] J. Preskill. Quantum computing and the entanglement frontier. *arXiv preprint arXiv:1203.5813*, 2012.

[176] J. Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018. arXiv: 1801.00862.

[177] X.-L. Qi and S.-C. Zhang. Topological insulators and superconductors. *Reviews of Modern Physics*, 83(4):1057, 2011.

[178] X. Qiang, X. Zhou, J. Wang, C. M. Wilkes, T. Loke, S. O'Gara, L. Kling, G. D. Marshall, R. Santagati,

T. C. Ralph, et al. Large-scale silicon quantum photonics implementing arbitrary two-qubit processing. *Nature Photonics*, 12(9):534, 2018.

[179] R. Raussendorf and H. J. Briegel. A one-way quantum computer. *Physical Review Letters*, 86(22):5188, 2001.

[180] R. Raussendorf and H. J. Briegel. Computational model underlying the one-way quantum computer. *Quantum Information & Computation*, 2(6):443–486, 2002.

[181] R. Raussendorf, D. Browne, and H. Briegel. The one-way quantum computer–a non-network model of quantum computation. *Journal of Modern Optics*, 49(8):1299–1306, 2002.

[182] R. Raussendorf, D. E. Browne, and H. J. Briegel. Measurement-based quantum computation on cluster states. *Physical review A*, 68(2):022312, 2003.

[183] M. C. Rechtsman, J. M. Zeuner, Y. Plotnik, Y. Lumer, D. Podolsky, F. Dreisow, S. Nolte, M. Segev, and A. Szameit. Photonic floquet topological insulators. *Nature*, 496(7444):196, 2013.

[184] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences*, 114(29):7555–7560, 2017.

[185] D. Reitzner, D. Nagaj, and V. Buzek. Quantum walks. *Acta Physica Slovaca. Reviews and Tutorials*, 61(6), Dec 2011. arXiv: 1207.7283.

[186] E. G. Rieffel and W. H. Polak. *Quantum Computing: A Gentle Introduction*. The MIT Press, 1 edition edition, Mar 2011.

[187] E. Riehl. *Category Theory in Context*. Courier Dover Publications, 2017.

[188] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryp-

tosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[189] J. Romero and A. Aspuru-Guzik. Variational quantum generators: Generative adversarial quantum machine learning for continuous distributions. *arXiv preprint arXiv:1901.00848*, 2019.

[190] S. Rosenblum, Y. Y. Gao, P. Reinhold, C. Wang, C. J. Axline, L. Frunzio, S. M. Girvin, L. Jiang, M. Mirrahimi, M. H. Devoret, et al. A cnot gate between multiphoton qubits encoded in two cavities. *Nature Communications*, 9(1):652, 2018.

[191] J. J. Rotman. *Advanced Modern Algebra*, volume 114. American Mathematical Soc., 2010.

[192] A. Roy and D. P. DiVincenzo. Topological quantum computing. *arXiv preprint arXiv:1701.05052*, 2017.

[193] M. Saffman, T. G. Walker, and K. Mølmer. Quantum information with rydberg atoms. *Reviews of Modern Physics*, 82(3):2313, 2010.

[194] U. Schollwöck. The density-matrix renormalization group. *Reviews of Modern Physics*, 77(1):259, 2005.

[195] C. Schreyvogel, V. Polyakov, R. Wunderlich, J. Meijer, and C. Nebel. Active charge state control of single NV centres in diamond by in-plane Al-Schottky junctions. *Scientific Reports*, 5:12160, 2015.

[196] E. Schrödinger. Discussion of probability relations between separated systems. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 31, pages 555–563. Cambridge University Press, 1935.

[197] M. Schuld and N. Killoran. Quantum machine learning in feature Hilbert spaces. *Physical Review Letters*, 122(4):040504, 2019.

[198] H. M. Sheffer. A set of five independent postulates for Boolean algebras, with application to logical

constants. *Trans. Amer. Math. Soc.*, 14:481–488, 1913.

[199] Y. Shi. Both Toffoli and controlled-NOT need little help to do universal quantum computing. *Quantum Information & Computation*, 3(1):84–92, 2003.

[200] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society, 1994.

[201] J. W. Silverstone, D. Bonneau, J. L. O'Brien, and M. G. Thompson. Silicon quantum photonics. *IEEE Journal of Selected Topics in Quantum Electronics*, 22(6):390–402, 2016.

[202] S. Sim, Y. Cao, J. Romero, P. D. Johnson, and A. Aspuru-Guzik. A framework for algorithm deployment on cloud-based quantum computers. *arXiv preprint arXiv:1810.10576*, 2018.

[203] D. R. Simon. On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 116–123, Washington, DC, USA, 1994. IEEE Computer Society.

[204] S. Simon. Quantum computing...with a twist. *Physics World*, Sep 2010. https://physicsworld.com/a/quantum-computing-with-a-twist/.

[205] A. Sipahigil, R. E. Evans, D. D. Sukachev, M. J. Burek, J. Borregaard, M. K. Bhaskar, C. T. Nguyen, J. L. Pacheco, H. A. Atikian, C. Meuwly, et al. An integrated diamond nanophotonics platform for quantum optical networks. *Science*, page aah6875, 2016.

[206] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik. qHiPSTER: The quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195*, 2016.

[207] R. S. Smith, M. J. Curtis, and W. J. Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[208] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, Mar 1977.

[209] N. Spagnolo, C. Vitelli, M. Bentivegna, D. J. Brod, A. Crespi, F. Flamini, S. Giacomini, G. Milani, R. Ramponi, P. Mataloni, and et al. Efficient experimental validation of photonic boson sampling against the uniform distribution. *Nature Photonics*, 8(8):615–620, Aug 2014. arXiv: 1311.1622.

[210] G. Strang. *Linear Algebra and Its Applications*. Cengage Learning, 2018.

[211] S. Sun, H. Kim, Z. Luo, G. S. Solomon, and E. Waks. A single-photon switch and transistor enabled by a solid-state quantum memory. *arXiv preprint arXiv:1805.01964*, 2018.

[212] S. Sun, H. Kim, G. S. Solomon, and E. Waks. A quantum phase switch between a single solid-state spin and a photon. *Nature Nanotechnology*, 11(6):539–544, 2016.

[213] L. Susskind. Dear qubitzers, GR = QM. *arXiv preprint arXiv:1708.03040*, 2017.

[214] J.-L. Tambasco, G. Corrielli, R. J. Chapman, A. Crespi, O. Zilberberg, R. Osellame, and A. Peruzzo. Quantum interference of topological states of light. *Science Advances*, 4(9):eaat3187, 2018.

[215] M. S. Tame, B. A. Bell, C. Di Franco, W. J. Wadsworth, and J. G. Rarity. Experimental realization of a one-way quantum computer algorithm solving Simon's problem. *Physical Review Letters*, 113(20):200501, 2014. arXiv: 1410.3859.

[216] E. Tang. A quantum-inspired classical algorithm for recommendation systems. *Electronic Colloquium on Computational Complexity (ECCC)*, 25:128, 2018.

[217] T. Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644, Berlin, Heidelberg, 1980. Springer-Verlag.

[218] S. B. van Dam, M. Walsh, M. J. Degen, E. Bersin, S. L. Mouradian, A. Galiullin, M. Ruf, M. IJspeert, T. H. Taminiau, R. Hanson, et al. Optical coherence of diamond nitrogen-vacancy centers formed by ion implantation and annealing. *arXiv preprint arXiv:1812.11523*, 2018.

[219] L. M. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414(6866):883, 2001. arXiv: quant-ph/0112176.

[220] M. Verbin, O. Zilberberg, Y. E. Kraus, Y. Lahini, and Y. Silberberg. Observation of topological phase transitions in photonic quasicrystals. *Physical Review Letters*, 110(7):076403, 2013.

[221] M. Verbin, O. Zilberberg, Y. Lahini, Y. E. Kraus, and Y. Silberberg. Topological pumping over a photonic Fibonacci quasicrystal. *Physical Review B*, 91(6):064201, 2015.

[222] G. Verdon, J. Pye, and M. Broughton. A universal training algorithm for quantum deep learning. *arXiv preprint arXiv:1806.09729*, 2018.

[223] F. Verstraete and J. I. Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. *arXiv preprint cond-mat/0407066*, 2004.

[224] G. Vidal. Entanglement renormalization. *Physical Review Letters*, 99(22):220405, 2007.

[225] J. von Neumann. *Mathematical Foundations of Quantum Mechanics*. Springer, Berlin, 1932.

[226] C. Wang, F.-G. Deng, Y.-S. Li, X.-S. Liu, and G. L. Long. Quantum secure direct communication with high-dimension quantum superdense coding. *Phys. Rev. A*, 71:044305, Apr 2005.

[227] D. Wang, O. Higgott, and S. Brierley. A generalised variational quantum eigensolver. *arXiv preprint arXiv:1802.00171*, 2018.

[228] Y. Wang, X. Zhang, T. A. Corcovilos, A. Kumar, and D. S. Weiss. Coherent addressing of individual neutral atoms in a 3d optical lattice. *Physical Review Letters*, 115(4):043003, 2015.

[229] Z. Wang, Y. Chong, J. D. Joannopoulos, and M. Soljačić. Observation of unidirectional backscattering-immune topological electromagnetic states. *Nature*, 461(7265):772, 2009.

[230] T. Watson, S. Philips, E. Kawakami, D. Ward, P. Scarlino, M. Veldhorst, D. Savage, M. Lagally, M. Friesen, S. Coppersmith, et al. A programmable two-qubit quantum processor in silicon. *Nature*, 555(7698):633, 2018.

[231] D. Wecker, M. B. Hastings, and M. Troyer. Progress towards practical quantum variational algorithms. *Physical Review A*, 92(4):042303, 2015.

[232] D. S. Weiss and M. Saffman. Quantum computing with neutral atoms. *Phys. Today*, 70(7):44, 2017.

[233] S. R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69(19):2863, 1992.

[234] T. Wildey. Shor's Algorithm in Python3. https://github.com/toddwildey/shors-python, 2014.

[235] P. Wittek. Quantum machine learning edX MOOC. https://www.edx.org/course/quantum-machine-learning-2, 2019.

[236] P. Wittek and C. Gogolin. Quantum enhanced inference in Markov logic networks. *Scientific Reports*, 7:45672, 2017.

[237] K. Wright, K. Beck, S. Debnath, J. Amini, Y. Nam, N. Grzesiak, J.-S. Chen, N. Pisenti, M. Chmielewski, C. Collins, et al. Benchmarking an 11-qubit quantum computer. *arXiv preprint arXiv:1903.08181*, 2019.

[238] T.-Y. Wu, A. Kumar, F. Giraldo, and D. S. Weiss. Stern–Gerlach detection of neutral-atom qubits in a state-dependent optical lattice. *Nature Physics*, page 1, 2019.

[239] L. Xiao, X. Zhan, Z. Bian, K. Wang, X. Zhang, X. Wang, J. Li, K. Mochizuki, D. Kim, N. Kawakami, et al. Observation of topological edge states in parity–time-symmetric quantum walks. *Nature Physics*, 13(11):1117, 2017.

[240] J.-S. Xu, K. Sun, Y.-J. Han, C.-F. Li, J. K. Pachos, and G.-C. Guo. Simulating the exchange of majorana zero modes with a photonic system. *Nature Communications*, 7:13194, 2016.

[241] W. Zeng, B. Johnson, R. Smith, N. Rubin, M. Reagor, C. Ryan, and C. Rigetti. First quantum computers need smart software. *Nature News*, 549(7671):149, Sep 2017.

[242] W. J. Zeng. Clarifying quantum supremacy: better terms for milestones in quantum computation. *Medium*, Jan 31, 2019. https://medium.com/@wjzeng/clarifying-quantum-supremacy-better-terms-for-milestones-in-quantum-computation-d15ccb53954f.

[243] L. Zhou, S.-T. Wang, S. Choi, H. Pichler, and M. D. Lukin. Quantum approximate optimization algorithm: Performance, mechanism, and implementation on near-term devices. *arXiv preprint arXiv:1812.01041*, 2018.

[244] O. Zilberberg, S. Huang, J. Guglielmon, M. Wang, K. P. Chen, Y. E. Kraus, and M. C. Rechtsman. Photonic topological boundary pumping as a probe of 4D quantum Hall physics. *Nature*, 553(7686):59, 2018.