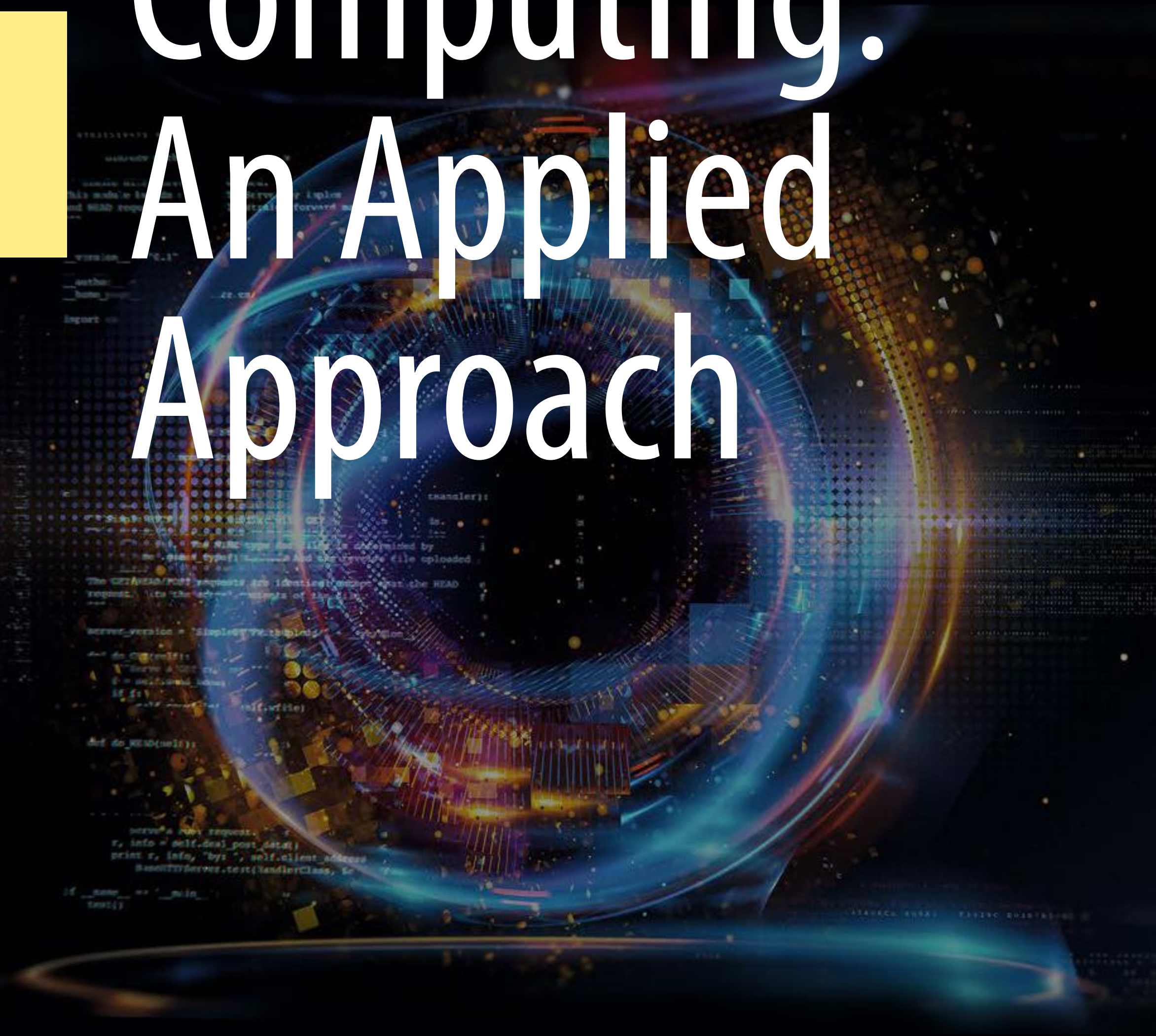
Jack D. Hidary

Quantum Computing: An Applied Approach



Springer

7 ***Teleportation, Superdense Coding and Bell's Inequality*** **81**

7.1 Quantum Teleportation **81**

7.2 Superdense Coding **84**

7.3 Code for Quantum Teleportation and Superdense Communication **85**

7.4 Bell Inequality Test **88**

8 ***The Canon: Code Walkthroughs*** **95**

8.1 The Deutsch-Jozsa Algorithm **97**

8.2 The Bernstein-Vazirani Algorithm **104**

8.3 Simon's Problem **107**

8.4 Quantum Fourier Transform **108**

8.5 Shor's Algorithm **111**

RSA Cryptography **111**

The Period of a Function **113**

Period of a Function as an Input to a Factorization Algorithm **114**

8.6 Grover's Search Algorithm **126**

9 ***Quantum Computing Methods*** **131**

9.1 Variational Quantum Eigensolver **131**

VQE with Noise **136**

More Sophisticated Ansatzes **138**

9.2 Quantum Chemistry **139**

9.3 Quantum Approximate Optimization Algorithm (QAOA) **144**

Example Implementation of QAOA **147**

9.4 Machine Learning on Quantum Processors **154**

9.5 Quantum Phase Estimation **160**

Implemention of QPE **163**

9.6 Solving Linear Systems **166**
Description of the HHL Algorithm **168**
Example Implementation of the HHL Algorithm **170**
9.7 Quantum Random Number Generator **178**
9.8 Quantum Walks **180**
Implementation of a Quantum Walk **182**
9.9 Summary **187**

10 *Applications and Quantum Supremacy* 189
10.1 Applications **189**
Quantum Simulation and Chemistry **189**
Sampling from Probability Distributions **190**
Linear Algebra Speedup with Quantum Computers **190**
Optimization **190**
Tensor Networks **190**
10.2 Quantum Supremacy **190**
Random Circuit Sampling **191**
Other Problems for Demonstrating Quantum Supremacy **195**
Quantum Advantage **196**
10.3 Future Directions **196**
Quantum Error Correction **196**
Doing Physics with Quantum Computers **197**

III Toolkit

11 *Mathematical Tools for Quantum Computing I* 201
11.1 Introduction and Self-Test **201**
11.2 Linear Algebra **203**
Vectors and Notation **203**
Basic Vector Operations **204**
The Norm of a Vector **208**
The Dot Product **211**

CHAPTER 8

The Canon: Code Walkthroughs

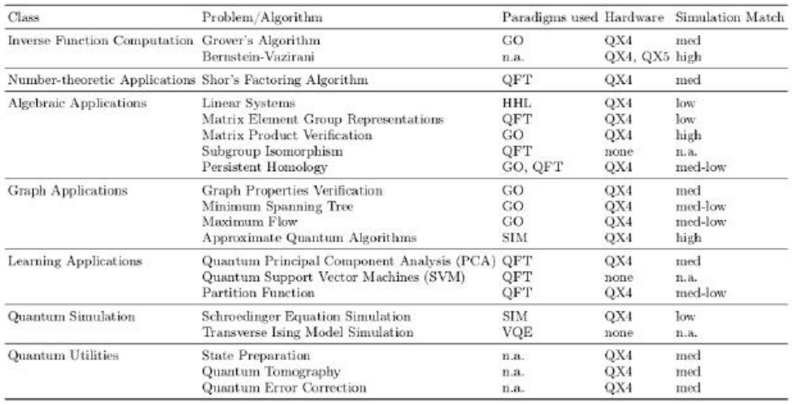
In this chapter, we will walk through a number of fundamental quantum algorithms. We call these algorithms *the canon* as they were all developed in the early years of quantum computing and were the first to establish provable computational speedups with quantum computers. We discussed most of these 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

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

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

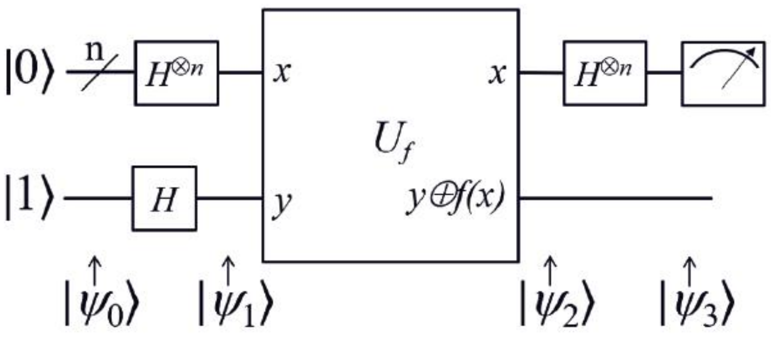


Figure 8.2: The DJ circuit Source: Wikimedia

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

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

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

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 $U_{f_x}(|0\rangle|0\rangle)$ as an example to demonstrate. Recall that $|00\rangle$ is shorthand for $|0\rangle|0\rangle$. Then, by definition,

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

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

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

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

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

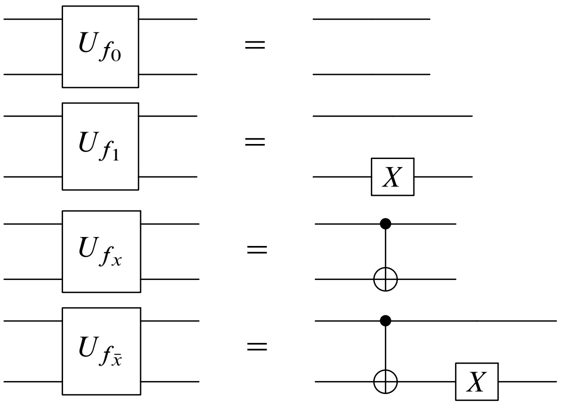
and apply the oracle. Using the linearity of the operator U_f to acquire the second equation and an observation for the third, we can check that

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

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

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

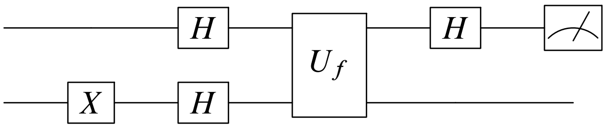


Now we only need to distinguish between the identity gate and the Z gate on the first qubit; we can do this by recalling that:

$$HZH = X$$

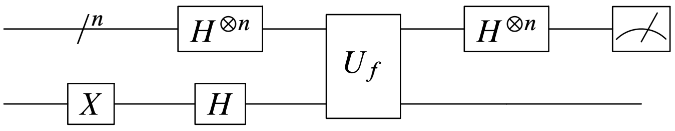
$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

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*

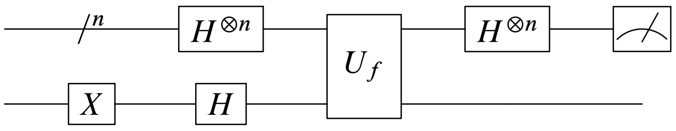
Now let's turn to the Bernstein-Vazirani (BV) algorithm that we considered earlier in this book [29]. As with DJ, the goal of BV is also to ascertain the nature of a black-box Boolean function. While it is true that DJ demonstrates an advantage of quantum over classical computing, if we allow for a small 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:
>
> $$f(x_{n\ \ 1}, x_{n\ \ 2}, \ldots, x_1, x_0),$$
>
> 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:
>
> $$a \cdot x = a_0 x_0 \oplus a_1 x_1 \oplus a_2 x_2 \ldots.$$
>
> 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.

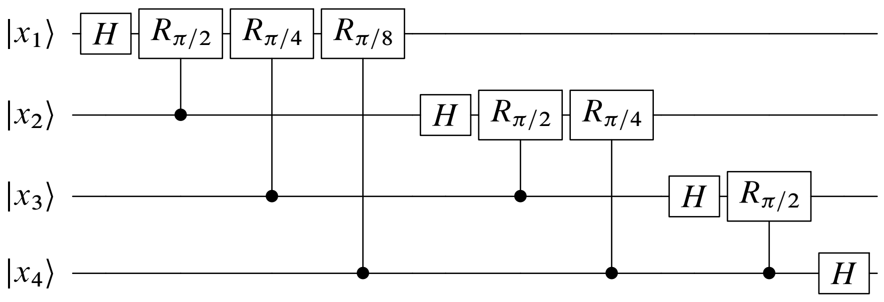
Simon was successful in proving that we can solve this problem to determine the periodicity of a function exponentially faster on a quantum computer compared with a classical one. Shor built on this key result in developing his algorithm for factoring large numbers when he realized that the two problems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



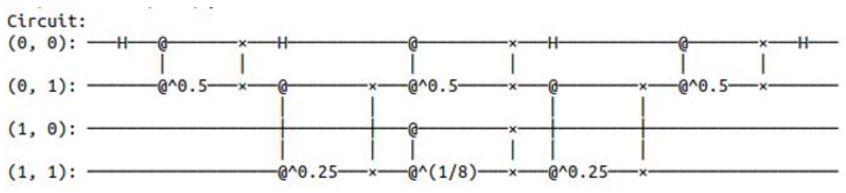
Now let's walk through the code for QFT as we will use this technique in the upcoming section on Shor's algorithm. We'll go through the program step by step, explaining first high-level details and then discussing implementation details.

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.

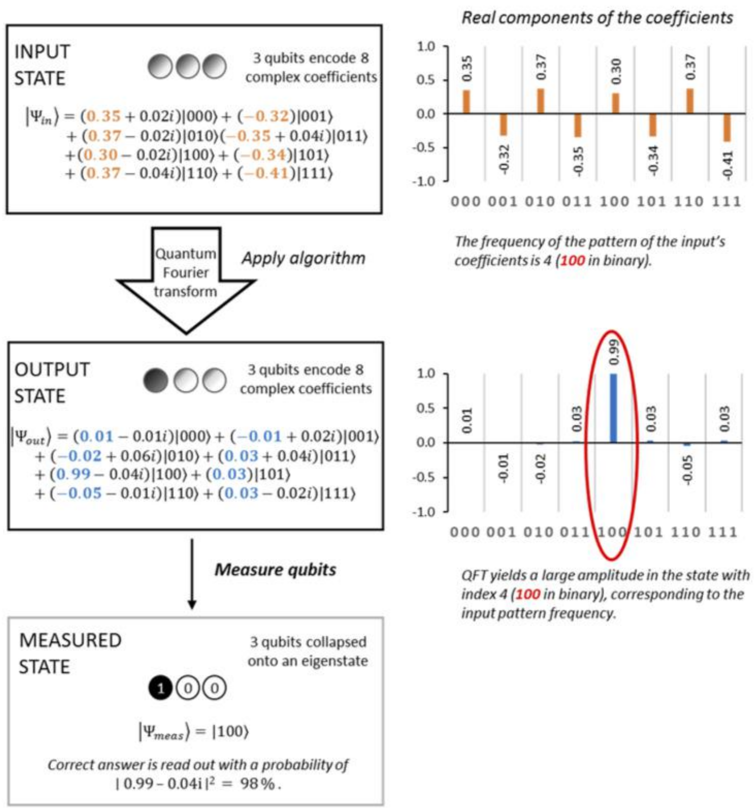
8.5 *Shor's Algorithm*

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
$|\Psi_{in}\rangle = (0.35 + 0.02i)|000\rangle + (-0.32)|001\rangle + (0.37 - 0.02i)|010\rangle(-0.35 + 0.04i)|011\rangle + (0.30 - 0.02i)|100\rangle + (-0.34)|101\rangle + (0.37 - 0.04i)|110\rangle + (-0.41)|111\rangle$
Real components of the coefficients
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
$|\Psi_{out}\rangle = (0.01 - 0.01i)|000\rangle + (-0.01 + 0.02i)|001\rangle + (-0.02 + 0.06i)|010\rangle + (0.03 + 0.04i)|011\rangle + (0.99 - 0.04i)|100\rangle + (0.03)|101\rangle + (-0.05 - 0.01i)|110\rangle + (0.03 - 0.02i)|111\rangle$
0.01
-0.01
-0.02
0.03
0.99
0.03
-0.05
0.03
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
$|\Psi_{meas}\rangle = |100\rangle$
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 $n = 0$

$$a^0 \ (\text{mod} N) = 1$$

we must find the next n that again satisfies

$$a^n \ (\text{mod} N) = 1$$

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 $f(x) = 2^n$ (mod 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.

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

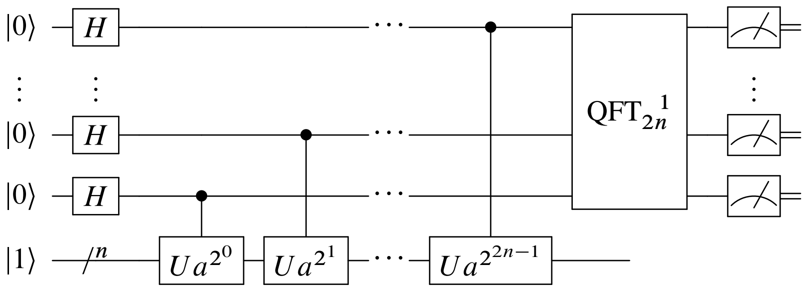
and

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

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

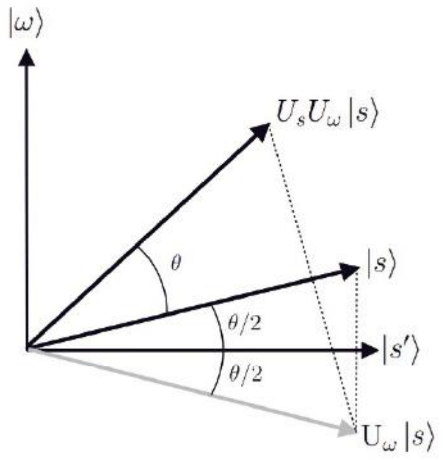


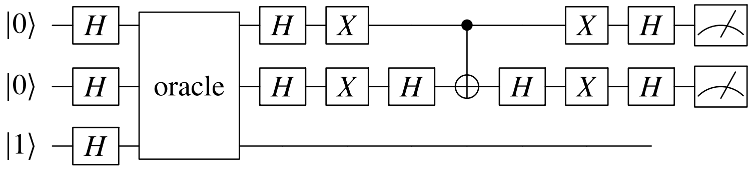
Figure 8.5: Plotting Grover's algorithm as it closes in on the target Source: Wikimedia

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

$$\min_{\theta} \ \langle\psi(\theta)| H |\psi(\theta)\rangle \tag{9.1}$$

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")
```

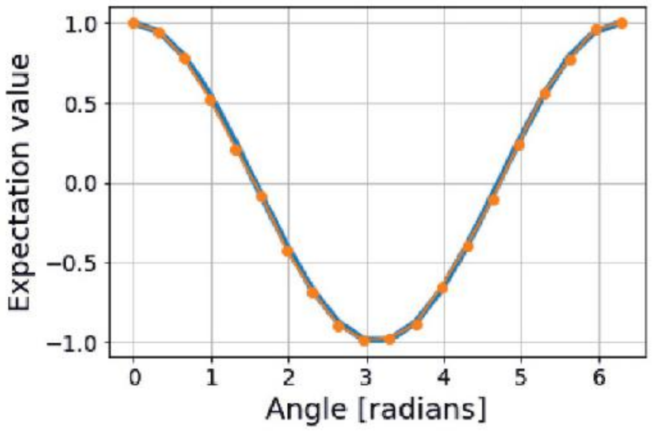


Figure 9.1: Expectation value of the simple Hamiltonian $H = Z$ at all angles $\theta \in [0, 2\pi)$ in the wavefunction ansatz $|\psi(\theta)\rangle = R_x(\theta)|0\rangle$

```
# 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]
```

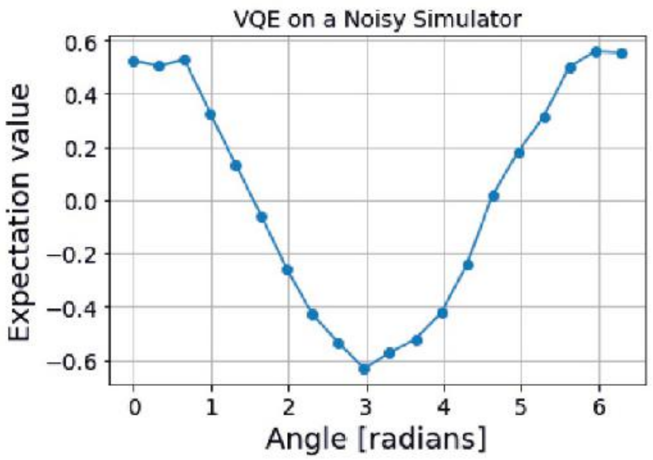


Figure 9.2: Results of running VQE on a simulator with Pauli channel noise.

```
# 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

$$U(t) = \exp(-iHt) \tag{9.4}$$

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

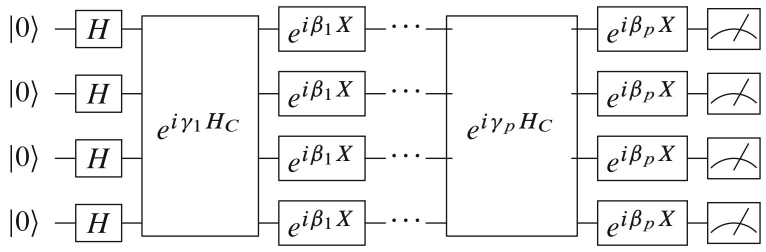
$$U(H_B, \beta) := e^{\ i\beta B} = \prod_{j=1}^{n} e^{\ i\beta\sigma_x^{(j)}} \tag{9.13}$$

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 $0 \leq \beta < \pi$.

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:

```python
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:

```python
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.

```python
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:

```python
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.

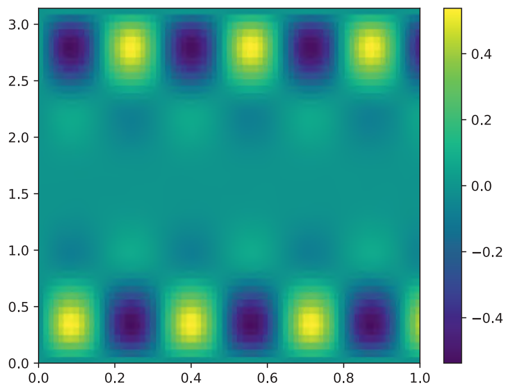


Figure 9.3: Cost landscape of the Ising Hamiltonian computed from one layer of QAOA

```python
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.

```python
# 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

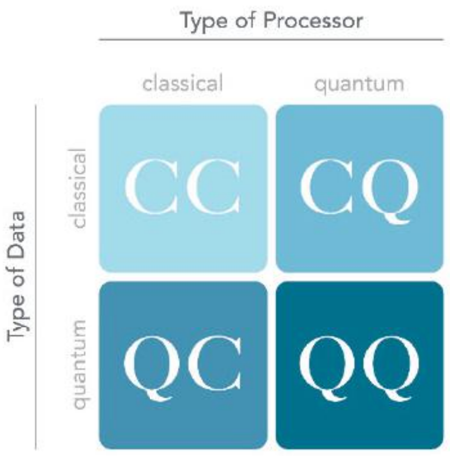


Figure 9.4: Data types and processor types

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.

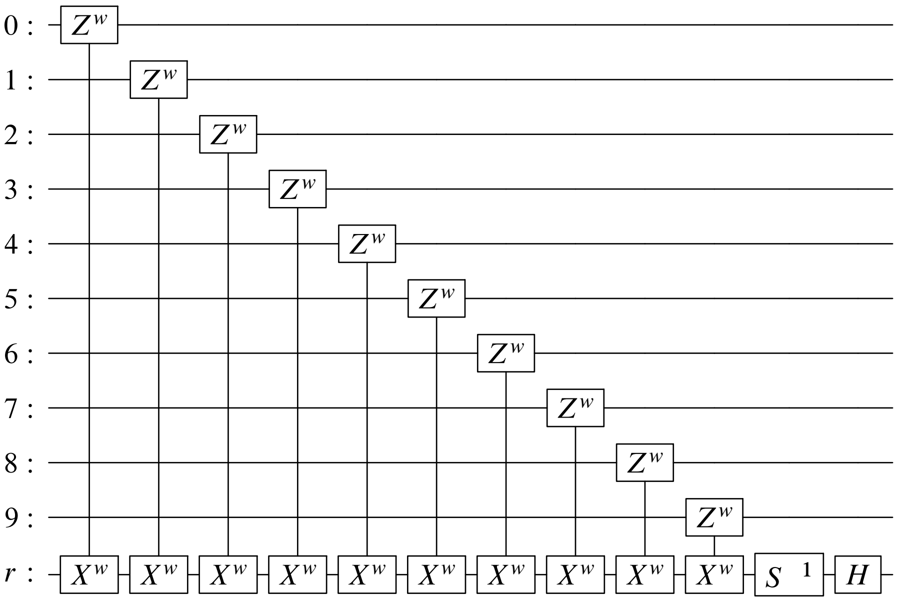
A number of groups have published in this area, including:

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,

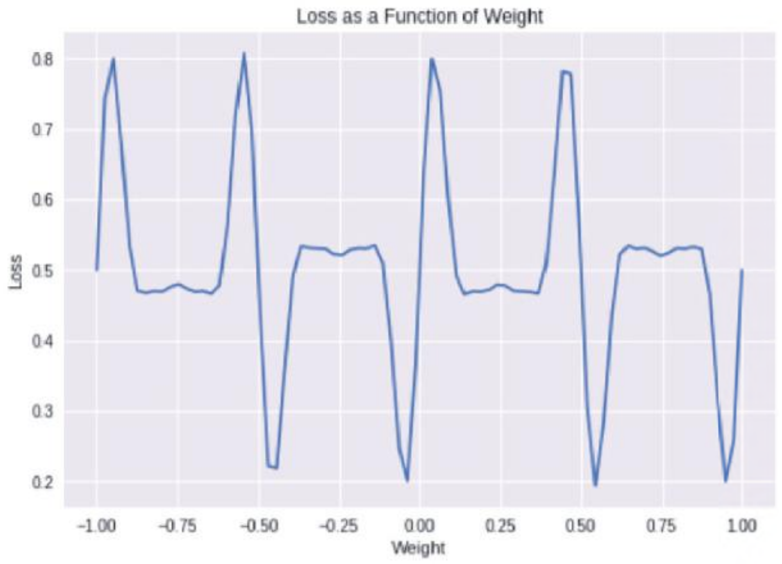


Figure 9.5: Loss function of the QNN plotted against the weight; the weight should be chosen to minimize the loss

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

$$|0\rangle|\psi\rangle + |1\rangle U^2|\psi\rangle = |0\rangle|\psi\rangle + e^{2\pi i 0.\varphi_2\cdots\varphi_n}|1\rangle|\psi\rangle$$

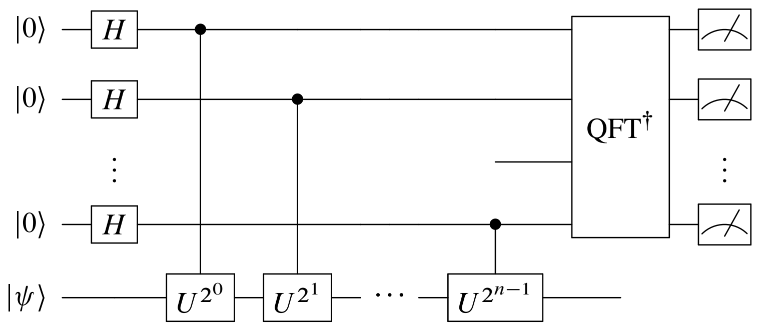
In general, using this same idea, we can see that

$$U^{2^k}|\psi\rangle = \lambda^{2^k}|\psi\rangle = e^{2\pi i(2^k\varphi)}|\psi\rangle = e^{2\pi i 0.\varphi_{k+1}\cdots\varphi_n} \tag{9.32}$$

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 $\mathbf{x} = A\ \ ^1\mathbf{b}$. 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

$$|| |\bar{x}\rangle - |x\rangle ||_2 \le \epsilon \tag{9.43}$$

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 $|\tilde{\lambda}_j\rangle_O \to |0\rangle_O$. 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

$$A \quad ^1|b\rangle = \sum_{j=1}^{N} \frac{\beta_j}{\lambda_j} |u_j\rangle \tag{9.53}$$

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 $|x\rangle = A \quad ^1|b\rangle$.

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

$$A = \begin{bmatrix} 4.302134 - 6.015934 \cdot 10 \ {}^{8}i & 0.235318 + 9.343861 \cdot 10 \ {}^{1}i \\ 0.235318 - 9.343883 \cdot 10 \ {}^{1}i & 0.583865 + 6.015934 \cdot 10 \ {}^{8}i \end{bmatrix} \tag{9.55}$$

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

$$\langle x|X|x\rangle = 0.144130$$
$$\langle x|Y|x\rangle = 0.413217$$
$$\langle x|Z|x\rangle = -0.899154$$

We will compare the expectation values obtained by HHL to these expectation values.

In our program, we first import the packages we will use:

```python
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:

```python
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.

```
"""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 $t = 1, 2, 3, ...$, 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.

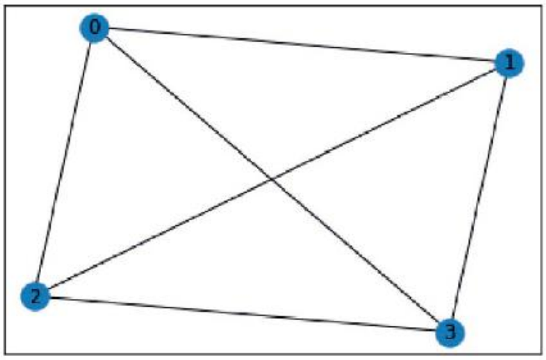


Figure 9.6: A complete graph on four vertices which we implement a continuous time quantum walk on. This graph is denoted K_4.

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")
```

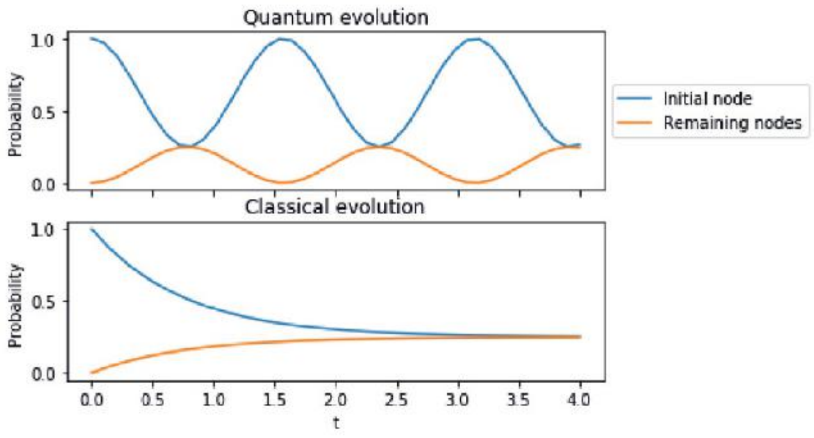


Figure 9.7: Time evolution of the continuous-time quantum and classical walks on a complete graph with four vertices.

```
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.