6 Complex Algorithms

Now that we have convinced ourselves that quantum computers can indeed reach capabilities *beyond classical*, at least on a semi-random circuit, we move on to discuss more meaningful algorithms. The previous sections on simple algorithms prepared us well to explore the complex algorithms in this chapter. We will still use a mix of full matrix and accelerated circuit implementations, depending on which seems best in context. It is recommended that you at least skim Chapter 4 on infrastructure before exploring this chapter.

In this chapter, we develop the quantum Fourier transform (QFT), an important technique used by many complex algorithms, and show it in action by performing arithmetic in the quantum Fourier domain. We discuss phase estimation next, another essential tool, especially when used together with QFT. Armed with these tools, we embark on implementing Shor's famous factorization algorithm.

After this, we switch gears and discuss Grover's search algorithm, along with some derivatives and improvements. We show how combining Grover and phase estimation leads to the interesting quantum counting algorithm. A short interlude on the topic of quantum random walks follows. Quantum walks are a complex topic; we only discuss and implement basic principles.

At a high level, quantum computing appears to have a computational complexity advantage over classical computing for several classes of algorithms and their derivatives. These are algorithms utilizing quantum search, algorithms based on the quantum Fourier transform, algorithms utilizing quantum random walks, and a fourth class, the simulation of quantum systems. We detail the variational quantum eigensolver algorithm (VQE), which allows finding minimum eigenvalues of a Hamiltonian. As an application, we develop a graph maximum cut algorithm by framing the problem as a Hamiltonian. This algorithm was introduced as part of the quantum approximate optimization algorithm (QAOA), which we briefly touch upon. We further explore the Subset Sum problem using a similar mechanism.

We conclude this chapter with an in-depth discussion of the elegant Solovay–Kitaev algorithm for gate approximation, another seminal result in quantum computing.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

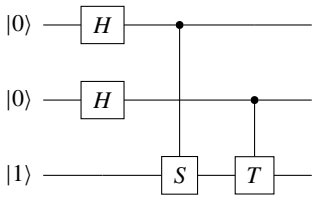


Figure 6.1 A phase kick circuit.

6.1 Phase Kick

In this section we discuss the *phase kick* mechanism, which is the basis for the quantum Fourier transform.

The controlled rotation gates have the interesting property that they can be used in an additive fashion. The basic principle is best explained with a circuit that is commonly known as a *phase kick* circuit. An example is shown in Figure 6.1.

Any number of qubits (the two top qubits in this example circuit) are initialized as $|0\rangle$ and put into superposition with Hadamard gates. A third ancilla qubit starts out in state $|1\rangle$. We apply the controlled S-gate and T-gate, but remember that these gates only add a phase to the $|1\rangle$ part of a state.

Each of the top qubits then connects a controlled rotation gate to the ancilla. In the example above:

- The top qubit controls a 90° rotation gate, the S-gate.
- The second qubit controls a 45° gate, the T-gate.

To express this in code:

```
psi = state.bitstring(0, 0, 1)
psi = ops.Hadamard(2)(psi)
psi = ops.ControlledU(0, 2, ops.Sgate())(psi)
psi = ops.ControlledU(1, 2, ops.Tgate())(psi, 1)
psi.dump()
```

Because of the superposition, the $|1\rangle$ part of each of the top qubits' superpositioned states will activate the rotation of the controlled gate. Using this, we can perform addition of phases. For the example, these are the resulting probability amplitudes and phases:

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

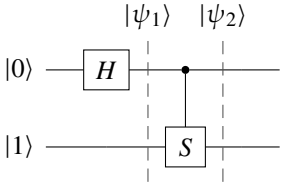
https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Note how the phases add up because they are controlled by a (superpositioned) $|1\rangle$ in the corresponding qubit. Having the top qubit as $|1\rangle$ adds 90°, and having the second qubit as $|1\rangle$ adds 45°. The third qubit is an ancilla. Also, note that we could use arbitrary rotations or fractions of π. We can use this type of circuit and corresponding rotation gates to express numerical computations in terms of phases. Of course, we have to normalize to 2π to avoid overflows.

The ability to add phases in a controlled fashion is powerful and is the foundation of the quantum Fourier transformation, which we will explore in the next section. In preparation for this section, let us briefly look at how to express the rotations mathematically.

- A rotation by 180° as a fraction of 2π is $e^{2\pi i/2^1}$. Expressed as a phase angle this is -1.
- A rotation by 90° as a fraction of 2π is $e^{2\pi i/2^2}$, a phase of i.
- A rotation by 45° as a fraction of 2π is $e^{2\pi i/2^3}$.
- Finally, a rotation by $135° = 90° + 45°$ as a fraction of 2π is $e^{2\pi i(1/2^2+1/2^3)}$.

So why is this circuit called a phase *kick* circuit? To understand this, let us look at a simpler version of the circuit and the corresponding math.



The state $|\psi_1\rangle$ after the Hadamard is:

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

With the Controlled-S operation, the state $|\psi_2\rangle$ becomes

$$\begin{aligned}|\psi_2\rangle &= \frac{1}{\sqrt{2}}\big(|0\rangle|1\rangle + |1\rangle S|1\rangle\big)\\ &= \frac{1}{\sqrt{2}}\big(|0\rangle|1\rangle + |1\rangle e^{i\pi/2}|1\rangle\big).\end{aligned}$$

We pull out the $|1\rangle$ on the right:

$$|\psi_2\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle + e^{i\pi/2}|1\rangle\big)|1\rangle.$$

We observe that the second qubit $|1\rangle$ remains *unmodified*. The trick here is that $|1\rangle$ is an eigenstate of the S-gate. We will elaborate further on eigenvalues in Section 6.4 on phase estimation, but in summary, we found a way to *kick* the phase from qubit 1 to the controlling qubit 0.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

This mechanism also enabled the Bernstein–Vazirani algorithm, covered in Section 3.7. We did not use rotation gates in our implementation, but rather Controlled-Not gates on states in the Hadamard basis. A Controlled-Not gate in this basis corresponds to a simple Z-gate (see also Section 8.4.5), a 180° rotation about the z-axis.

6.2 Quantum Fourier Transform

The quantum Fourier transform (QFT) is one of the foundational algorithms of quantum computing. It is important to note that although it does not speed up classical Fourier analysis of classical data, it does enable other important algorithms, such as phase estimation, which is the approximation of the eigenvalues of an operator. Phase estimation is a key ingredient in Shor's factoring algorithm and others. Let us discuss a few preliminaries first.

6.2.1 Binary Fractions

In Section 6.3 we will learn how to interpret qubits measured as $|0\rangle$ and $|1\rangle$ as bits in a binary number, with the most significant bit being on the left or at the top in circuit diagrams. To convert a binary representation to a decimal number, we added the `bits2val` routine to `lib/helpers.py`.

However, we can also interpret the bits as constituents of a *binary fraction*. We have a choice to interpret the bit order from left to right or right to left and decide which bit should be the least significant qubit, for example:

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

The x_i should be interpreted as binary bits, with values of either 0 or 1. This looks natural in the following mathematical notation:

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

We can define this the other way around, with x_0 being least significant fractional part of a binary fraction, as in $0.x_{n-1} \cdots x_1 x_0$. It is important to note that this is just a notational difference. We will encounter an example of this in the derivation of phase estimation in Section 6.4.

In our code below, the most significant bit represents the largest part of the fraction, for example, 0.5 for the first bit, 0.25 for the second bit, 0.125 for the third, and so

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

on. Again, we can easily revert the order. Given a binary string of states, we use the following routine from file `lib/helpers.py` to compute binary fractions:

```
def bits2frac(bits: Iterable) -> float:
    """For given bits, compute the binary fraction."""

    return sum(bits[i] * 2**(-i-1) for i in range(len(bits)))
```

Computing the fraction from a single 0 will result in:

```
    val = helper.bits2frac((0,))
    print(val)
>> 0
```

The fraction from a single 1:

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

For two bits, the first one will represent the 0.5 part of the fraction, the second part will represent the 0.25 part of the fraction:

```
    val = helper.bits2frac((0, 1))
    print(val)
>> 0.25
    val = helper.bits2frac((1, 0))
    print(val)
>>0.5
    val = helper.bits2frac((1, 1))
    print(val)
>>0.75
```

6.2.2 Phase Gates

We already learned about two different phase gates in Section 2.6.5 on single-qubit gates. We saw the discrete phase gate R_k and the U_1 gate:

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix} \quad \text{and} \quad U_1(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix},$$

with:

$$R_k(0) = U_1(2\pi/2^0),$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

$$R_k(1) = U_1(2\pi/2^1),$$
$$R_k(2) = U_1(2\pi/2^2),$$
$$\vdots$$

Remember Euler's formula, which represents rotation as complex exponentiation:

$$e^{i\phi} = \cos(\phi) + i\,\sin(\phi). \tag{6.1}$$

Applying one of these gates to a state means only the $|1\rangle$ basis state gets a phase. To reiterate, the angles (with *cw* meaning clockwise and *ccw* meaning counterclockwise) are:

$$e^{i\frac{\pi}{2}} = i \Rightarrow 90^\circ ccw,$$
$$e^{i\pi} = -1 \Rightarrow 180^\circ ccw,$$
$$e^{i\frac{3\pi}{2}} = -i \Rightarrow 270^\circ ccw = 90^\circ cw.$$

You might have noticed that the S-gate and T-gate we used in Section 6.1 are also of this form, except they have their rotation angles at the fixed values of $\pi/2$ and $\pi/4$.

6.2.3 Quantum Fourier Transform

We now have all ingredients necessary for the QFT, which we will implement with Hadamard gates and controlled discrete phase gates R_k. Later, for example in Section 6.3 and 6.6, we will also see versions of QFT being implemented with U_1 gates. QFT takes a state $|\psi\rangle$, where each qubit should be interpreted as part of a binary fraction:

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

And converts it to a form where fractional values are encoded as fractional phases. Since we use the phase gates, only the $|1\rangle$ basis state gets a phase. This is how the state looks after we apply a QFT circuit. A detailed derivation for how this state comes about is presented later, in Section 6.4.

$$\begin{aligned} QFT|x_0\ x_1\ \cdots\ x_{n-1}\rangle = \frac{1}{2^{n/2}} &\left(|0\rangle + e^{2\pi i\ 0.x_0}|1\rangle\right) \\ &\otimes \left(|0\rangle + e^{2\pi i\ 0.x_0x_1}|1\rangle\right) \\ &\vdots \\ &\otimes \left(|0\rangle + e^{2\pi i\ 0.x_0x_1x_2\cdots x_{n-1}}|1\rangle\right). \end{aligned}$$

If we interpret the binary fractions in the reverse order, we'd get this state:

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

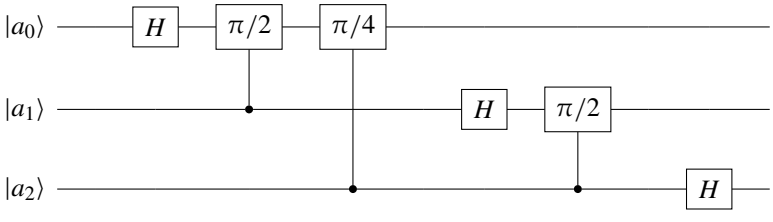
$$\begin{aligned}
&\frac{1}{2^{n/2}}\left(|0\rangle + e^{2\pi i\ 0.x_{n-1}}|1\rangle\right) \\
&\quad\otimes \left(|0\rangle + e^{2\pi i\ 0.x_{n-1}x_{n-2}}|1\rangle\right) \\
&\quad\vdots \\
&\quad\otimes \left(|0\rangle + e^{2\pi i\ 0.x_{n-1}x_{n-2}\cdots x_1x_0}|1\rangle\right).
\end{aligned}$$

Note that the *QFT* is a unitary operation, as it is made up of other unitary operators. Since it is unitary, it has an inverse, and we should explicitly state this important inverse relation:

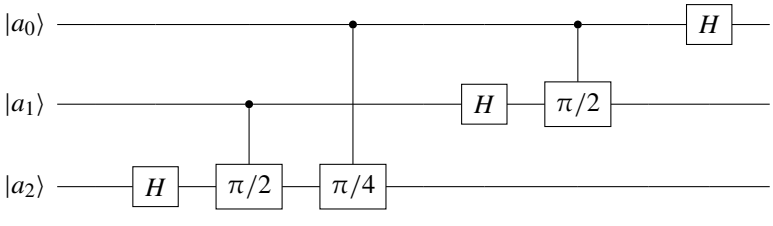
$$\begin{aligned}
QFT^{\dagger}\ &\frac{1}{2^{n/2}}\left(|0\rangle + e^{2\pi i\ 0.x_0}|1\rangle\right) \\
&\quad\otimes \left(|0\rangle + e^{2\pi i\ 0.x_0x_1}|1\rangle\right) \\
&\quad\vdots \\
&\quad\otimes \left(|0\rangle + e^{2\pi i\ 0.x_0x_1x_2\cdots x_{n-1}}|1\rangle\right) \\
&\quad= |x_0\ x_1\ \cdots\ x_{n-1}\rangle. \qquad (6.2)
\end{aligned}$$

This mathematical formulation gives us the blueprint for constructing the circuit. We have to put the qubits into superposition, and we have to apply controlled rotation gates to rotate qubits around according to the scheme of binary fractions.

We can construct the QFT in two directions, depending on our interpretation of the input qubits. We can draw it from top to bottom:



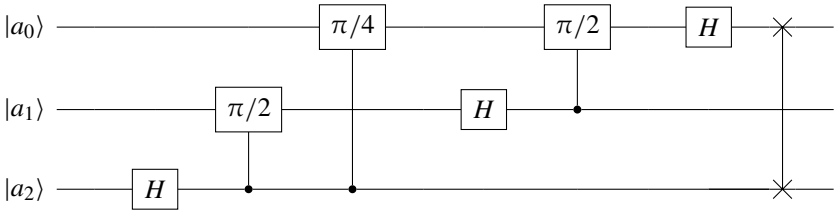
Or from bottom to top:



Or we can implement it one way and add an optional Swap gate to get both possible directions with just one implementation. We will see examples of all of these styles in

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

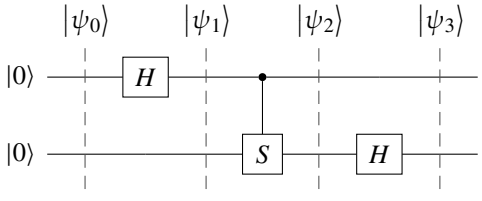
the remainder of this book. Note how we can also switch the controlling and controlled phase gates in the following diagram. Phase gates are symmetric, as shown in Section 3.2.3



An interesting question is that of accuracy. How many fractions do we need in order to achieve a reliable result for a specific algorithm? This is an interesting metric to play around with. Early work on approximate quantum Fourier transform indicates that for Shor's algorithm, you can stop adding rotation gates as the rotation angles become smaller than π/n^2 (Coppersmith, 2002).

6.2.4 Two-Qubit QFT

It can be helpful to explore the QFT on two qubits. There are only four basis states, which may help to develop intuition. We should start with a simple QFT circuit and the input $|00\rangle$. Note that, as with the Controlled-Z gates from Section 3.2, the direction of the Controlled-S gate does *not* matter. Try it out!



We construct this in code with this snippet:

```
psi = state.bitstring(0, 0)
psi = ops.Hadamard()(psi)
psi = ops.ControlledU(0, 1, ops.Sgate())(psi)
psi = ops.Hadamard()(psi, 1)
psi.dump()
```

All states are equally likely with the same probability:

```
|00> (|0>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|01> (|1>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|10> (|2>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|11> (|3>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Let us briefly look at how to compute the results of this circuit. We start with the tensor product of two inputs set to $|0\rangle$ initially.

$$|\psi_0\rangle = |0\rangle \otimes |0\rangle.$$

Apply the Hadamard to the first qubit:

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

Applying the Controlled-S gate has no effect, as it would affect the $|1\rangle$ part of a state, hence $|\psi_2\rangle = |\psi_1\rangle$. Applying the final Hadamard will yield:

$$\begin{aligned}|\psi_3\rangle &= (I \otimes H)\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}|0\rangle\right) \\ &= \frac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big).\end{aligned}$$

Now let's do these calculations with $|0\rangle \otimes |1\rangle = |01\rangle$ as input.

$$\begin{aligned}|\psi_0\rangle &= |0\rangle \otimes |1\rangle, \\ |\psi_1\rangle &= (H \otimes I)\big(|0\rangle \otimes |1\rangle\big) \\ &= \frac{|0\rangle + |1\rangle}{\sqrt{2}}|1\rangle \\ &= \frac{|01\rangle + |11\rangle}{\sqrt{2}}.\end{aligned}$$

Applying the Controlled-S gate will have an effect in this case:

$$|\psi_2\rangle = \frac{|01\rangle + e^{i\pi/2}|11\rangle}{\sqrt{2}}.$$

And the final Hadamard leads to:

$$\begin{aligned}|\psi_3\rangle &= (I \otimes H)\left(\frac{|01\rangle + e^{i\pi/2}|11\rangle}{\sqrt{2}}\right) \\ &= \frac{1}{2}\left(|0\rangle(|0\rangle - |1\rangle) + e^{i\pi/2}|1\rangle(|0\rangle - |1\rangle)\right).\end{aligned}$$

As $e^{i\pi/2} = i$, this results in:

$$|\psi_3\rangle = \frac{1}{2}\big(|00\rangle - |01\rangle + i|10\rangle - i|11\rangle\big).$$

Now let's look the four different inputs in matrix form. Remember that applying the operators in a circuit means we have to multiply the matrices in reverse order:

$$(I \otimes H)\,CS\,(H \otimes I) = \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & i & -i\\ 1 & 1 & -1 & -1\\ 1 & -1 & -i & i\end{bmatrix}.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Applying this gate to the $|0,0\rangle$ base state pulls out the first row, and, correspondingly, $|0,1\rangle$ pulls out the second, matching our results above. Similarly, for the other three basis states and columns 2 and 3:

$$\frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & i & -i\\ 1 & 1 & -1 & -1\\ 1 & -1 & -i & i\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix}$$
$$= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle).$$

And indeed, the other three cases produce amplitudes that correspond to rows 1, 2, and 3 of the above matrix:

```
Input: |01>
|00> (|0>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|01> (|1>):  ampl: -0.50+0.00j prob: 0.25 Phase: 180.0
|10> (|2>):  ampl: +0.00+0.50j prob: 0.25 Phase:  90.0
|11> (|3>):  ampl: +0.00-0.50j prob: 0.25 Phase: -90.0
Input: |10>
|00> (|0>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|01> (|1>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|10> (|2>):  ampl: -0.50+0.00j prob: 0.25 Phase: 180.0
|11> (|3>):  ampl: -0.50+0.00j prob: 0.25 Phase: 180.0
Input: |11>
|00> (|0>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|01> (|1>):  ampl: -0.50+0.00j prob: 0.25 Phase: 180.0
|10> (|2>):  ampl: +0.00-0.50j prob: 0.25 Phase: -90.0
|11> (|3>):  ampl: +0.00+0.50j prob: 0.25 Phase:  90.0
```

In summary, QFT encodes the binary fractional encoding of a state into phases representing the fractions for the basis states. It "rotates around" the states according to the binary fractional parts of each qubit. In the Section 6.3, we will see an immediate application of this: quantum arithmetic. We will combine two states in an additive fashion to enable addition and subtraction in the Fourier domain.

One very important aspect of QFT is that while it enables encoding of (binary) states with phases, on measurement the state would collapse to just one of the basis states. All other information will be lost. The challenge for QFT-based algorithms is to apply transformations such that, on measurement, you can find an algorithmic solution to the problem at hand. In practically all cases, we will apply the inverse QFT to get the state out of superposition to measure a result, following Equation (6.2).

6.2.5 QFT Operator

Here is an implementation of the QFT operator in full matrix form. We put all the input qubits in superposition, then apply controlled rotations for each fractional part to each qubit. There are many different ways to encode this, and all implementations

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

must get the indices into the right order. It usually helps to transpile a circuit into a textual format, such as QASM (defined in Section 8.3.1) or similar, to inspect the indices.

```
def Qft(nbits: int) -> Operator:
  """Make an n-bit QFT operator."""

  op = Identity(nbits)
  h = Hadamard()

  for idx in range(nbits):
    # Each qubit first gets a Hadamard
    op = op(h, idx)

    # Each qubit now gets a sequence of Rk(2), Rk(3), ..., Rk(nbits)
    # controlled by qubit (1, 2, ..., nbits-1).
    for rk in range(2, nbits - idx + 1):
      controlled_from = idx + rk - 1
      op = op(ControlledU(controlled_from, idx, Rk(rk)), idx)

  # Now the qubits need to change their order.
  for idx in range(nbits // 2):
    op = op(Swap(idx, nbits - idx - 1), idx)

  if not op.is_unitary():
    raise AssertionError('Constructed non-unitary operator.')
  return op
```

Computing the inverse of the QFT operator is trivial. QFT is a unitary operator, so the inverse is being computed trivially as the adjoint:

```
  Qft = ops.Qft(nbits)
  [...]
  InvQft = Qft.adjoint()
```

If the QFT is computed via explicit gate applications in a circuit, then the inverse has to be implemented as the application of the inverse gates in reverse order, as outlined in Section 2.13 on reversible computing. We will see examples of this explicit construction shortly.

6.2.6 Online Simulation

It can be helpful to use one of the available online simulators to verify results. We have to be aware that the simulators might not agree on the qubit ordering. For experiments, we can always add Swap gates at the end of a circuit to follow online simulators' qubit ordering. Alternatively, we can also add the Swap gates to the circuits in the online simulators themselves.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

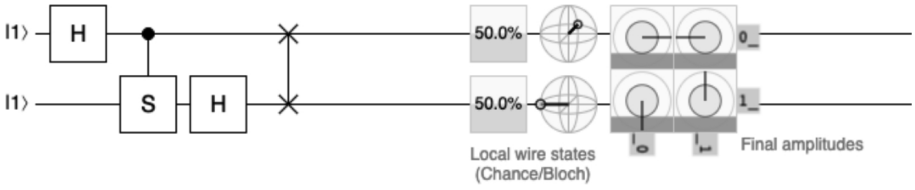


Figure 6.2 A partial screenshot from https://algassert.com/quirk.

A widely used online simulator is Quirk (Gidney, 2021a). Let us construct a simple two-qubit QFT circuit in Quirk, as shown in Figure 6.2. If we look to the very right and reconstruct the phases from the gray circles (blue on the website), we see that state $|00\rangle$ (top left) has a phase of 0 (the direction of the x-axis). The state $|01\rangle$ (top right) has a phase of 180°, the state $|10\rangle$ (bottom left) has a phase of -90°, and the state $|11\rangle$ has a phase of 90°. So it appears that Quirk agrees with our qubit ordering (or we agree with Quirk).

Quirk also shows the state of individual qubits on a Bloch sphere. How does this work, as we are dealing with a two-qubit tensored state, and Bloch spheres only represent single qubits? We talked about the partial trace in Section 2.14, which allows *tracing out* qubits from a state. The result after a trace-out operation is a density matrix. Since it only represents a partial state, it is called a reduced density matrix. In Section 2.9, we showed how to compute the Bloch sphere coordinates from a density matrix. Note that for systems of more than two qubits, all qubits that are not of interest must be traced out, such that only a 2×2 density matrix remains.

Let's give this a try. From the state as shown in Figure 6.2, we trace out qubit 0 and qubit 1 individually and compute the Bloch sphere coordinates:

```
    psi = state.bitstring(1, 1)
    psi = ops.Qft(2)(psi)

    rho0 = ops.TraceOut(psi.density(), [1])
    rho1 = ops.TraceOut(psi.density(), [0])

    x0, y0, z0 = helper.density_to_cartesian(rho0)
    x1, y1, z1 = helper.density_to_cartesian(rho1)

    print('x0: {:.1f} y0: {:.1f} z0: {:.1f}'.format(x0, y0, z0))
    print('x1: {:.1f} y1: {:.1f} z1: {:.1f}'.format(x1, y1, z1))

>>
x0: -1.0 y0: 0.0 z0: -0.0
x1: -0.0 y1:-1.0 z1: -0.0
```

This result seems to agree with Quirk as well. The first qubit is located at -1 on the x-axis of the Bloch sphere (going from the back of the page to the front of the page), and the second qubit is located at -1 on the y-axis (going from left to right).

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

6.3 Quantum Arithmetic

We saw in Section 3.3 how a quantum circuit could be used to emulate a classical full adder, using quantum gates without exploiting any of the unique features of quantum computing, such as superposition or entanglement. It is fair to say that this was a nice exercise demonstrating the universality of quantum computing, but otherwise, a fairly inefficient way to construct a full adder.

In this section, we discuss another algorithm that performs addition and subtraction. This time the math is being developed in the Fourier domain with a technique that was first described by Draper (2000).

To perform addition, we will apply a QFT, some magic, and a final inverse QFT to obtain a numerical result. We explain this algorithm with just a hint of math and lots of code. This implementation uses a different direction from the controller to the controlled qubit as our early QFT operator. This is not difficult to follow; simply inverting the qubits in a register leads to identical implementations. We use explicit angles and the Controlled-U1 gate. The code can be found in the file `src/arith_quantum.py` in the open-source repository.

The first thing we need to specify is the bit width of the inputs a and b. If we want to do n-bit arithmetic, we need to store results as $(n + 1)$ bits to account for overflow.

Our entry point's signature will get the bit width as `n` and the two initial integer values `init_a` and `init_b`, which must fit into the available bits. The parameter `factor` will be 1.0 for addition and -1.0 for subtraction. We'll see shortly how this factor is applied.

```
def arith_quantum(n: int, init_a: int, init_b: int,
                  factor: float = 1.0, dumpit: bool = False) -> None:
```

We instantiate two registers with bitwidth $n + 1$. Because we interpret the bits in reverse order in this example, we have to invert the bits when initializing the registers:

```
  a = qc.reg(n+1, helper.val2bits(init_a, n)[::-1], name='a')
  b = qc.reg(n+1, helper.val2bits(init_b, n)[::-1], name='b')
```

The algorithm performs three basic operations:

- Apply the QFT over the qubits representing a. This encodes the bits as phases on states.
- Evolve a by b. This cryptic sounding step basically performs another set of QFT-like rotations on a using the same controlled-rotation mechanism as with regular QFT. It is not a full QFT though. There are also no initial Hadamard gates as the states are already in superposition. We detail the steps below.
- Perform inverse QFT to decode phases back to bits.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

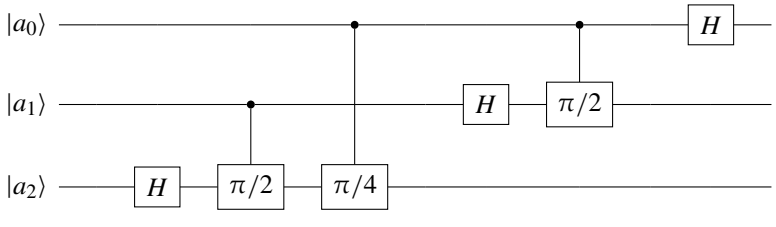
Here is the high level in code:

```
    for i in range(n+1):
        qft(qc, a, n-i)
    for i in range(n+1):
        evolve(qc, a, b, n-i, factor)
    for i in range(n+1):
        inverse_qft(qc, a, i)
```

Let us look at these three steps in detail, using the example of two-qubit additions. We can insert dumpers after each of the loops to dump and visualize the circuit, as described in Section 8.5.6. After the first loop, we produced this circuit, in QASM format (Cross et al., 2017):

```
OPENQASM 2.0;
qreg a[3];
qreg b[3];

h a[2];
cu1(pi/2) a[1],a[2];
cu1(pi/4) a[0],a[2];
h a[1];
cu1(pi/2) a[0],a[1];
h a[0];
```

This sequence corresponds to a standard three-qubit QFT circuit. We can choose to enumerate the qubits from 0 to 2, or 2 to 0; it does not make a real difference, as long as we stay consistent. After the first loop, we constructed a standard QFT circuit.



The middle loop in Figure 6.3 is where the magic happens – the evolve step produces this circuit in QASM format. We explain how this works below.

```
cu1(pi) b[2],a[2];
cu1(pi/2) b[1],a[2];
cu1(pi/4) b[0],a[2];
cu1(pi) b[1],a[1];
cu1(pi/2) b[0],a[1];
cu1(pi) b[0],a[0];
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

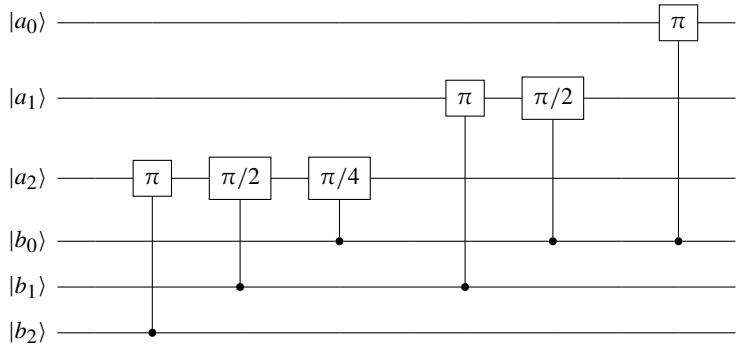
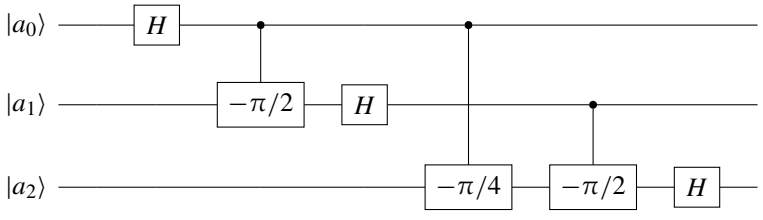
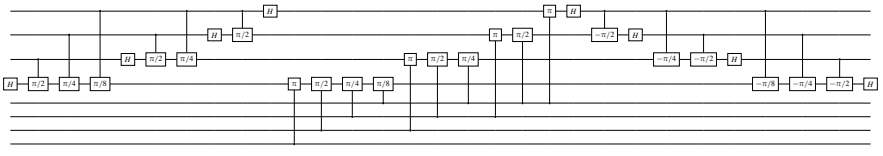


Figure 6.3 The evolve step of quantum arithmetic in the Fourier domain.

The construction of the inverse QFT circuit happens in the third loop. All of the first QFT's gates are inverted and applied in reverse order. Remember that the inverse of the Hadamard gate is another Hadamard gate, and the inverse of a rotation is a rotation by the same angle in the opposite direction.



This is how the combined circuit looks. The gates are too small to read, but the elegant, melodic structure of the whole circuit becomes apparent:[1]



Why and how does this work? Let us first try to explain this mathematically, before explaining it by looking at the state vector. First, remember that QFT takes this state:

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

and changes it into this form again; depending on bit ordering, two forms are possible:

[1] This circuit was generated with our LaTeX transpiler, explained in Section 8.5.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

$$\frac{1}{2^{n/2}}\left(|0\rangle + e^{2\pi i\ 0.x_{n-1}}|1\rangle\right)$$
$$\otimes \left(|0\rangle + e^{2\pi i\ 0.x_{n-1}x_{n-2}}|1\rangle\right)$$
$$\vdots$$
$$\otimes \left(|0\rangle + e^{2\pi i\ 0.x_{n-1}x_{n-2}\cdots x_1x_0}|1\rangle\right).$$

Applying the rotations of the `evolve` step adds the binary fractions of b to a. For example, the first part of above state for a:

$$\left(|0\rangle + e^{2\pi i\ 0.a_{n-1}}|1\rangle\right),$$

becomes:

$$\left(|0\rangle + e^{2\pi i\ 0.(a_{n-1}+b_{n-1})}|1\rangle\right).$$

And so on for all fractional parts. The "trick" for quantum arithmetic is to interpret the qubits not as binary fractions, but as bits of full binary numbers. When interpreted this way, the net result is a full binary addition.

Another way to explain this is to look at the state vector and the evolve circuit itself. Let's assume we want to add 1 in register a and 1 in register b. Before we enter the main loops, the state looks like the following. The first three qubits belong to a, with the qubit 0 now in the role of the least significant qubit. The next three qubits belong to b. This is how we initialize the state, and, as a result, only one state has a nonzero probability:

```
|100100> (|34>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

After the initial QFT, we get this state vector:

```
|543210> after qft
|000100> (|4>):   ampl: +0.35+0.00j prob: 0.12 Phase:   0.0
|001100> (|12>):  ampl: +0.25+0.25j prob: 0.12 Phase:  45.0
|010100> (|20>):  ampl: +0.00+0.35j prob: 0.12 Phase:  90.0
|011100> (|28>):  ampl: -0.25+0.25j prob: 0.12 Phase: 135.0
|100100> (|36>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|101100> (|44>):  ampl: -0.25-0.25j prob: 0.12 Phase: -135.0
|110100> (|52>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
|111100> (|60>):  ampl: +0.25-0.25j prob: 0.12 Phase: -45.0
```

Since we only apply QFT to the first three qubits, we now have eight superimposed states, all with the same probability. Let's now look at the circuit diagram for the evolve phase. The least significant qubit for b is global qubit 3, corresponding to b_0 in the diagram. Because it is set, the evolve circuit controls rotations of a full π to qubit a_0, a rotation of $\pi/2$ to qubit a_1, and a rotation of $\pi/4$ to qubit a_2.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

This works like clockwork, as in, how a clock *actually* works, where smaller gears drive the larger ones. Here we deal with rotations; the higher-order qubits are made to rotate slower than the lower-order qubits:

- The least significant qubit gets a full rotation by π. If it was not set, it will have a phase of π now. If it was set, it will now have a phase of 0.
- For qubit 1 of register a, it gets a rotation of $\pi/2$.
- The most significant qubit of register a rotates by $\pi/4$.

This means, for example, that when adding a number 1 twice, the least significant qubit flip flops between 0 and π, qubit 1 runs in increments of $\pi/2$, and qubit 2 in increments of $\pi/4$. The same scheme works for the higher order qubits in register b. For our example of $1 + 1$, the following are the phases on the state vector after the evolve step:

```
|543210> after evolve
|000100> (|4>):   ampl: +0.35+0.00j prob: 0.12 Phase:   0.0
|001100> (|12>):  ampl: -0.00+0.35j prob: 0.12 Phase:  90.0
|010100> (|20>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|011100> (|28>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
|100100> (|36>):  ampl: +0.35-0.00j prob: 0.12 Phase:  -0.0
|101100> (|44>):  ampl: -0.00+0.35j prob: 0.12 Phase:  90.0
|110100> (|52>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|111100> (|60>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
```

How does this compare if we initialize a with the value 2 instead of 1? Here is that state after initialization. Note the first three qubits are now in state $|010\rangle$:

```
|010100> (|20>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

Here is the state after the initial QFT. We see that the phases are identical to the $1 + 1$ state above after evolving:

```
|543210> after qft
|000100> (|4>):   ampl: +0.35+0.00j prob: 0.12 Phase:   0.0
|001100> (|12>):  ampl: +0.00+0.35j prob: 0.12 Phase:  90.0
|010100> (|20>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|011100> (|28>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
|100100> (|36>):  ampl: +0.35+0.00j prob: 0.12 Phase:   0.0
|101100> (|44>):  ampl: +0.00+0.35j prob: 0.12 Phase:  90.0
|110100> (|52>):  ampl: -0.35+0.00j prob: 0.12 Phase: 180.0
|111100> (|60>):  ampl: -0.00-0.35j prob: 0.12 Phase: -90.0
```

Back to our $1 + 1$ example. After evolve and the inverse QFT, the state corresponding to value $a = 2$ is the only state with nonzero probability, the addition worked.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Remember that the first three qubits correspond to the register a, with qubit 0 acting as the least significant bit.

```
|543210> after inv
|000100> (|4>):   ampl: +0.00+0.00j prob: 0.00 Phase:  90.0
|001100> (|12>):  ampl: +0.00-0.00j prob: 0.00 Phase: -26.6
|010100> (|20>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
|011100> (|28>):  ampl: -0.00-0.00j prob: 0.00 Phase: -180.0
|100100> (|36>):  ampl: -0.00-0.00j prob: 0.00 Phase: -112.5
|101100> (|44>):  ampl: -0.00+0.00j prob: 0.00 Phase: 157.5
|110100> (|52>):  ampl: -0.00+0.00j prob: 0.00 Phase: 112.5
|111100> (|60>):  ampl: -0.00-0.00j prob: 0.00 Phase: -157.5
```

The code for the QFT and evolve functions is straightforward using the `cu1` gate. The indices are hard to follow. It is a good exercise to dump the circuit textually, for example as QASM, and check that the gates are applied in the correct order (as we have done above):

```
def qft(qc: circuit.qc, reg: state.Reg, n: int) -> None:
  qc.had(reg[n])
  for i in range(n):
    qc.cu1(reg[n-(i+1)], reg[n], math.pi/float(2**(i+1)))

def evolve(qc: circuit.qc, reg_a: state.Reg, reg_b: state.Reg,
           n: int, factor: float) -> None:
  for i in range(n+1):
    qc.cu1(reg_b[n-i], reg_a[n], factor * math.pi/float(2**(i)))

def inverse_qft(qc: circuit.qc, reg: state.Reg, n: int) -> None:
  for i in range(n):
    qc.cu1(reg[i], reg[n], -1*math.pi/float(2**(n-i)))
  qc.had(reg[n])
```

Given the insight that rotations in the Fourier domain facilitate addition, it is almost too easy to implement subtraction – we add a factor to b, and, for subtraction, we evolve the state in the opposite direction. This is already implemented in the code above in the `evolve` function.

With the same line of reasoning, multiplication of the form $a + cb$, with c being a constant other than ± 1, also just applies the factor c to the rotations. We have to be careful with overflow because we only accounted for one overflow bit.

You could argue that this is a bit disingenuous, as the rotations are fixed to a given classical factor c. The algorithm does not implement an actual multiplication (as recently proposed in Gidney (2019)) where the factor c is another quantum register used as input to the algorithm. Argued this way, it is true that this multiplication is not purely quantum. Nevertheless, performing multiplication this way has a valid

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

use case. In Section 6.5 on Shor's algorithm, we will deal with known integer values. We can classically compute multiplication results before performing multiplication as above with an unknown quantum state. Perhaps we should call this *semi*-quantum multiplication? Naming is difficult.

In the upcoming discussion on the order-finding algorithm, we will see that for certain, more complex computations, you do indeed have to implement full arithmetic functions in the quantum domain.

To test our code, we check the results with a routine that performs a measurement. We don't actually measure, we just find the state with the highest probability. The input state with the bit pattern representing a has a probability of 1.0. After the rotations and coming out of superposition, the state $a + b$ will also have a probability close to 1.0. Note how we invert the bit order again to get to a valid result.

```
def check_result(psi: state.State, a, b,
                 nbits: int, factor: float = 1.0) -> None:
  """Find most likely result, dump it, compare against expected."""

  maxbits, _ = psi.maxprob()
  result = helper.bits2val(maxbits[0:nbits][::-1])
  if result != a + factor * b:
    print(f'{a} + ({factor} * {b}) != {result}')
    raise AssertionError('Incorrect addition.')
```

Our test program drives this with a few loops, passing `factor` through to `arith_quantum` and `evolve` to allow testing of subtraction and (pseudo) multiplication.

```
def main(argv):
  print('Check quantum addition...')
  for i in range(7):
    for j in range(7):
      arith_quantum(6, i, j, +1.0)

  print('Check quantum subtraction...')
  for i in range(8):
    for j in range(i):  # Note: Results can be 2nd complements.
      arith_quantum(6, i, j, -1.0)

  print('Check quantum multiplication...')
  for i in range(7):
    for j in range(7):
      arith_quantum(6, 0, i, j)
```

Since we're using the circuit implementation with accelerated gates, we can easily handle up to 14 qubits, hence we test with bit widths of 6 (plus one overflow bit per input) for the individual inputs.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

6.3.1 Adding a Constant

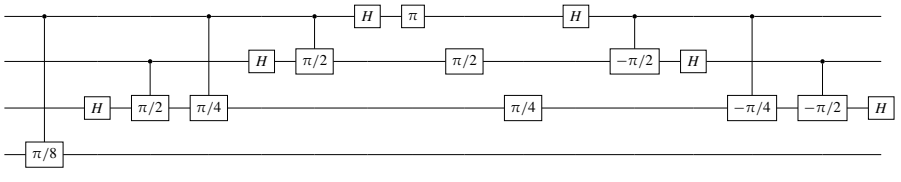
Adding a *known constant* to a quantum register does not require a second quantum register, as it did for the general case. We can simply precompute the rotation angles and directly apply them, as they would have, if they were controlled by a second register. To precompute the required angles:

```
def precompute_angles(a: int, n: int) -> List[float]:
  """Precompute angles used in the Fourier Transform, for fixed a."""

  # Convert 'a' to a string of 0's and 1's.
  s = bin(int(a))[2:].zfill(n)

  angles = [0.] * n
  for i in range(n):
    for j in range(i, n):
      if s[j] == '1':
        angles[n-i-1] += 2**(-(j-i))
    angles[n-i-1] *= math.pi
  return angles
```

We also have to modify the `evolve` step in the quantum addition. Instead of adding controlled gates, we simply add the rotation gates directly. This is the method we will use later in Shor's algorithm as well.

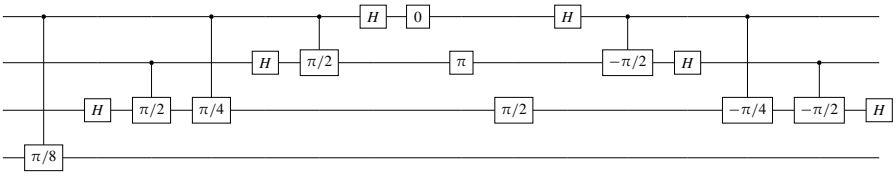
```
    for i in range(n+1):
        qft(qc, a, n-i)

    angles = precompute_angles(c, n)
    for i in range(n):
        qc.u1(a[i], angles[i])

    for i in range(n+1):
        inverse_qft(qc, a, i)
```

For a three-qubit addition of $1 + 1$, the circuit no longer needs the b register and turns into:



https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

This is the same circuit for addition of $1 + 2$, but notice the modified rotation angles in the evolve step:



6.4 Phase Estimation

Quantum phase estimation (QPE) is a key building block for the advanced algorithms presented in this chapter. QPE cannot be discussed without the concepts of eigenvalues and eigenvectors. Let us briefly reiterate what we already know about them.

6.4.1 Eigenvalues and Eigenvectors

We have seen how operators are applied to matrices via matrix-vector multiplication. In the introductory Section 1.6, we also briefly mentioned eigenvalues and eigenvectors, for which the following equation holds, where A is an operator, $|\psi\rangle$ is a state, and λ is a simple (complex) scalar:

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

For example, the identity matrix I has the eigenvalue 1.0. It leaves any vector it is applied to unmodified. Correspondingly, every nonzero, size-compatible vector is an eigenvector of I. Another example would be the Pauli matrices, which have eigenvalues of $+1$ and -1. Note that any multiple of an eigenvector is also an eigenvector. Eigenvalues for a given matrix are found by solving the characteristic equation:[2]

$$\det(A - \lambda I) = 0.$$

In this text, we keep it simple and find eigenvalues of a given matrix with the help of `numpy`:

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

Diagonal matrices are a special case for which finding of eigenvalues is trivial – we can take them right off the diagonal, with the corresponding eigenvectors being the computational bases $(1,0,0,\ldots)^T$, $(0,1,0,\ldots)^T$, and so on. If you are an attentive

[2] With det being the determinant of a matrix. See, for example: https://en.wikipedia.org/wiki/Determinant.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

reader, you will have noticed that the gates we used for phase rotations during quantum Fourier transforms are of similar form:

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix} \quad \text{and} \quad U_1(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix}.$$

6.4.2 Phase Estimation

The definition of *phase estimation* is as follows. *Given a unitary matrix U with an eigenvector $|u\rangle$ and an eigenvalue $e^{2\pi i\phi}$, estimate the value of ϕ.*

The ϕ in this eigenvalue is a factor in the range of 0.0 to 1.0 that computes a fraction of 2π. At a high level, the procedure works in two steps:

1. Encode the unknown phase with a circuit that produces a result that is identical to the result of the QFT discussed earlier in Section 6.2. We interpret the resulting qubits as parts of a binary fraction.
2. Apply $QFT^\dagger$ to compute the phase ϕ.

To detail step one, we define a register with t qubits, where t is determined by the precision we want to achieve. Just as with QFT, we will interpret the qubits as parts of a binary fraction; the more qubits, the more fine-grained fractions of powers of 2 we will be able to add up to the final result. We initialize the register with $|0\rangle$ and put it in superposition with Hadamard gates.

We add a second register representing the eigenvector $|u\rangle$. We then connect this register to a sequence of t instances of the unitary gate U, each one taken to increasing powers of 2 $(1, 2, 4, 8, \ldots, 2^{t-1})$. Similar to QFT, we connect the t register's qubits as controlling gates to the unitary gates. To achieve the powers of 2, we multiply U with itself and accumulate the results. The whole procedure is shown in Figure 6.4 in circuit notation.

Now, the relation to the quantum Fourier rotation gates becomes apparent – higher powers of 2 of U will result in the fractional phase angle being multiplied by increasing powers of 2. Please note the ordering of the qubits and their corresponding powers of 2.

A question to ask is this: Why does $|u\rangle$ have to be initialized with an eigenvector? Wouldn't this procedure work for any normalized state vector $|x\rangle$? The answer is no. This equation holds true only for eigenvectors:

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

This means we can apply U and any power of U to $|u\rangle$ as often as we want to. Since $|u\rangle$ is an eigenvector, it will only be scaled by a number: the complex eigenvalue ϕ with modulus 1 (as we will prove below). Let's develop the details in the next section. If you are not interested in the math, you may jump to Section 6.4.4 on implementation.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

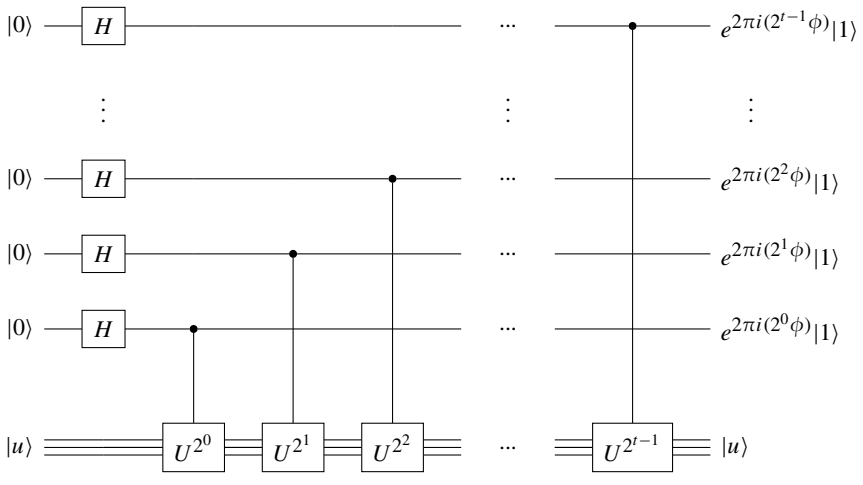


Figure 6.4 The phase estimation circuit.

6.4.3 Detailed Derivation

The first thing to know is that the eigenvalues of a unitary matrix have a modulus of 1, which is easy to prove.

Proof We know that eigenvalues are defined as

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

Squaring the equation yields:

$$\langle xU^\dagger|Ux\rangle = \langle x\lambda^*|\lambda x\rangle.$$

We know that $UU^\dagger = I$, and λ^2 is a factor that we can pull in front of the inner product. State vectors are also normalized with an inner product of 1.0:

$$\begin{aligned}\langle xU^\dagger|Ux\rangle &= (\lambda^*\lambda)\langle x|x\rangle,\\ \langle x|x\rangle &= |\lambda|^2\langle x|x\rangle,\\ 1 &= |\lambda|^2 = |\lambda|.\end{aligned}$$

□

Since $|\lambda| = 1$, we know that the eigenvalues are of this form, with ϕ being a factor between 0 and 1:

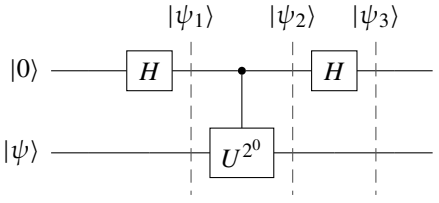
$$\lambda = e^{2\pi i\phi}.$$

In Section 6.2, we used the following notation for binary fractions with t bits of resolution, with ϕ_i being binary bits with values 0 or 1:

$$\begin{aligned}\phi &= 0.\phi_0\phi_1\ldots\phi_{t-1}\\ &= \phi_0\frac{1}{2^1} + \phi_1\frac{1}{2^2} + \cdots + \phi_{t-1}\frac{1}{2^t}.\end{aligned}$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

With these preliminaries, let's see what happens to a state in the following circuit, which is a first small part of the phase estimation circuit. The lower qubits are in state $|\psi\rangle$, which must be an eigenstate of U.



We start by limiting the precision for the eigenvalue of U to just one fractional bit. Once we understand how this works for a single fractional bit, we expand to two, which then makes it easy to generalize.

Let start with U having eigenvalue $\lambda = e^{2\pi i\ 0.\phi_0}$, with only one binary fractional part, corresponding to 2^{-1}. The phase can thus only have a value of 0.0 or 0.5. The state $|\psi_1\rangle$ after the first Hadamard gate is:

$$|\psi_1\rangle = |+\rangle \otimes |\psi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle|\psi\rangle + |1\rangle|\psi\rangle\right).$$

After the Controlled-U operation, the state $|\psi_2\rangle$ will be:

$$\begin{aligned}|\psi_2\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle|\psi\rangle + |1\rangle U|\psi\rangle\right)\\ &= \frac{1}{\sqrt{2}}\left(|0\rangle|\psi\rangle + e^{2\pi i 0.\phi_0}|1\rangle|\psi\rangle\right)\\ &= \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i 0.\phi_0}|1\rangle\right)|\psi\rangle.\end{aligned}$$

We observe that $|\psi\rangle$ remains unmodified, as it should be, since it is an eigenstate of U. We can apply U as often as we want, $|\psi\rangle$ remains unmodified. However, the eigenvalue has turned into a phase of the $|1\rangle$ part of the first qubit. We *kicked* the phase to qubit 0, as described in Section 6.1.

There is still the problem that, on measuring the first qubit, the state might still collapse to $|0\rangle$ or $|1\rangle$ with equal probability of $1/2$, independently of whether the phase value is 0.0 or 0.5. Kicking the phase up does not change the probabilities. To resolve this, we apply another Hadamard gate to the top qubit and obtain state $|\psi_3\rangle$ (omitting the trailing qubit $|\psi\rangle$ for simplicity):

$$\begin{aligned}|\psi_3\rangle &= \frac{1}{\sqrt{2}}H\left(|0\rangle + e^{2\pi i 0.\phi_0}|1\rangle\right)\\ &= \frac{1}{2}\left(1 + e^{2\pi i 0.\phi_0}\right)|0\rangle + \frac{1}{2}\left(1 - e^{2\pi i 0.\phi_0}\right)|1\rangle.\end{aligned}$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

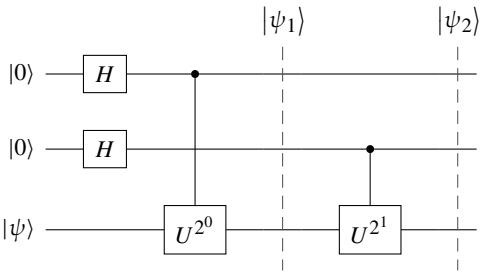


Figure 6.5 Phase estimation with two digits and unitaries.

The term ϕ_0 is a binary digit and can only be 0 or 1. If it is 0, the fraction $0(2^{-1}) = 0$. The factor $e^{2\pi i 0.\phi_0} = e^0$ becomes 1, and $|\psi_3\rangle$ becomes:

$$|\psi_3\rangle = \frac{1}{2}|0\rangle + \frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle - \frac{1}{2}|1\rangle = |0\rangle.$$

If, on the other hand, the digit $\phi_0 = 1$, then $1(2^{-1}) = 1/2$. The factor $e^{2\pi i 0.\phi_0}$ becomes $e^{2\pi i/2} = -1$ and $|\psi_3\rangle$ becomes:

$$|\psi_3\rangle = \frac{1}{2}|0\rangle - \frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle + \frac{1}{2}|1\rangle = |1\rangle.$$

We will now measure either $|0\rangle$ or $|1\rangle$ with certainty, depending on whether ϕ was 0.0 or 0.5.

Now let's move on and consider two (or more) fractional binary parts for the phase $\phi = 0.\phi_0\phi_1$, which can now have values $0.0, 0.25, 0.5$, and 0.75. The corresponding circuit also has two exponentiated U gates and is shown in Figure 6.5. From above, we know that $|\psi_1\rangle$ will have this form:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^3}} \underbrace{\left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1}\right)|1\rangle}_{\text{qubit 0}} \otimes \underbrace{(|0\rangle + |1\rangle)}_{\text{qubit 1}} \otimes \underbrace{|\psi\rangle}_{\text{qubit 2}}.$$

Now let's study the effect of the controlled U^{2^1} on qubit 1. We know that squaring a rotation means doubling the rotation angle:

$$U^2|\psi\rangle = e^{2\pi i(2\phi)}|\psi\rangle.$$

Looking at the fractional representation and the effects of U^2, we see that the binary point shifts by one digit:

$$\begin{aligned} 2\phi &= 2(0.\phi_0\phi_1) \\ &= 2\left(\phi_0 2^{-1} + \phi_1 2^{-2}\right) \\ &= \phi_0 + \phi_1 2^{-1} = \phi_0.\phi_1. \end{aligned}$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

We split the fraction at the decimal point:

$$\begin{aligned} e^{2\pi i(2\phi)} &= e^{2\pi i(\phi_0.\phi_1)} \\ &= e^{2\pi i(\phi_0+0.\phi_1)} \\ &= e^{2\pi i(\phi_0)} e^{2\pi i(0.\phi_1)}. \end{aligned}$$

The term ϕ_0 corresponds to a binary digit; it can only be 0 or 1. This means the first factor corresponds to a rotation by 0 or 2π, which has no effect. The final result is:

$$e^{2\pi i(2\phi)} = e^{2\pi i(0.\phi_1)}.$$

which we can generalize to:

$$e^{2\pi i(2^k\phi)} = e^{2\pi i\ 0.\phi_k}.$$

For our three-qubits circuit above, the final state becomes:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^3}} \underbrace{\left(|0\rangle + e^{2\pi i 0.\phi_0\phi_1}|1\rangle\right)}_{\text{qubit 0}} \otimes \underbrace{\left(|0\rangle + e^{2\pi i 0.\phi_1}|1\rangle\right)}_{\text{qubit 1}} \otimes \underbrace{|\psi\rangle}_{\text{qubit 2}}$$

This is the form that results from applying the *QFT* operator to two qubits! This means we can apply the two-qubit adjoint $QFT^\dagger$ operator to retrieve the binary fractions of $\phi = 0.\phi_0\phi_1$ as qubit states $|0\rangle$ or $|1\rangle$:

$$QFT^\dagger_{0,1}|\psi_2\rangle = |\phi_1\rangle \otimes |\phi_0\rangle \otimes |\psi\rangle.$$

To summarize, we connected the 0th power of 2 to the last qubit in register t and the $(t-1)$'s power of 2 to the first qubit (or the other way around, depending on how we want to interpret the binary fractions). The final state thus becomes:

$$\begin{aligned} &\frac{1}{2^{t/2}} \left(|0\rangle + e^{i\pi 2^{t-1}\phi}|1\rangle\right) \\ &\otimes \left(|0\rangle + e^{i\pi 2^{t-2}\phi}|1\rangle\right) \\ &\quad\vdots \\ &\otimes \left(|0\rangle + e^{i\pi 2^{0}\phi}|1\rangle\right). \end{aligned}$$

Similar to a QFT, we express ϕ in t bits in the fractional notation as the following:

$$\phi = 0.\phi_{t-1}\phi_{t-2}\cdots\phi_0.$$

Multiplying this angle with the powers of two as shown above will shift the digits of the binary representation to the left, and the state after the circuit will be:

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

$$\frac{1}{2^{t/2}}\left(|0\rangle + e^{i2\pi 0.\phi_{t-1}}|1\rangle\right)$$
$$\otimes \left(|0\rangle + e^{i2\pi 0.\phi_{t-1}\phi_{t-2}}|1\rangle\right)$$
$$\vdots$$
$$\otimes \left(|0\rangle + e^{i2\pi 0.\phi_{t-1}\phi_{t-2}\cdots\phi_1\phi_0}|1\rangle\right).$$

The form above is similar to the result of a QFT, where the rotations come out according to how the input qubits were initialized in binary representation. The bit indices are reversed from what we usually see, but this is just a renaming or ordering issue. The final step of phase estimation reverses the QFT. It applies $QFT^\dagger$ to allow reconstruction of the input, which, in our case, was the representation of ϕ in binary fractions. The complete circuit layout is shown in Figure 6.6.

It is important not to confuse the ordering in code and notation. As usual, all textbooks disagree on notation and ordering. This is not that important in our case as we can interpret the binary fraction in the proper order to obtain the desired result.

We can now measure the qubits, interpret them as binary fractions, and combine them to approximate ϕ, as we will show in the implementation. Remember how we used $|1\rangle$ to initialize the ancilla qubit in the phase kick circuit? The underlying mechanism is the same. State $|1\rangle$ is an eigenstate for both the S-gate and T-gate.

6.4.4 Implementation

In code, this may look simpler than the math above. The full implementation can be found in file `src/phase_estimation` in the open source repository. We drive this algorithm from `main()`, reserving six qubits for t and three qubits for the unitary operator. It is instructive to experiment with the numbers. We run ten experiments:

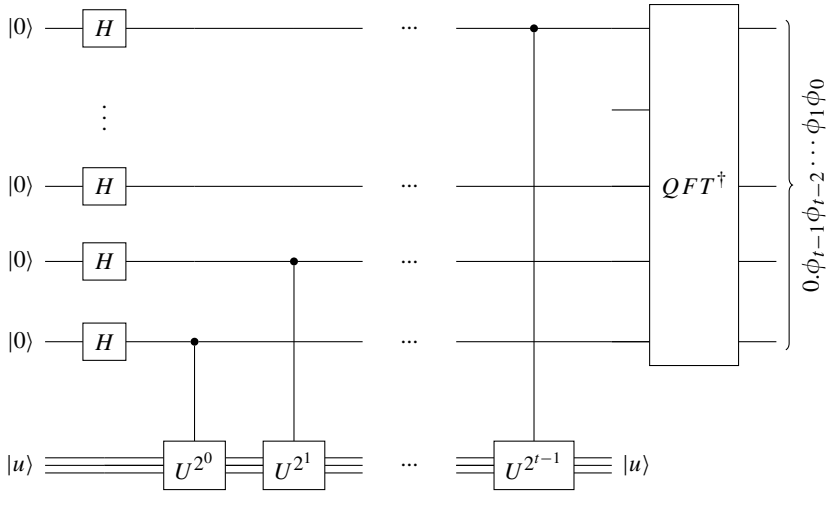


Figure 6.6 Full quantum phase estimation circuit.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
def main(argv):
  nbits = 3
  t = 6
  print('Estimating {} qubits random unitary eigenvalue '
        .format(nbits) + 'with {} bits of accuracy.'.format(t))
  for i in range(10):
    run_experiment(nbits, t)
```

In each experiment, we create a random operator and obtain its eigenvalues and eigenvectors to ensure that our estimates below are close.

```
def run_experiment(nbits: int, t: int = 8):
  """Run single phase estimation experiment."""

  # Make a unitary and find eigenvalue/vector to estimate.
  # We use functions from scipy for this purpose.
  umat = scipy.stats.unitary_group.rvs(2**nbits)
  eigvals, eigvecs = np.linalg.eig(umat)
  u = ops.Operator(umat)
```

We pick eigenvector 0 to use as an example here, but the procedure works for all other pairs of eigenvectors and eigenvalues. To check whether the algorithm works, we compute the to-be-estimated angle `phi` upfront. Since we are assuming the eigenvalue to be of the form $e^{2\pi i\phi}$, as discussed in Section 6.4.2, we divide by `2j*np.pi`. Also, we don't want to deal with negative values. Again, this angle does not participate in the algorithm; we just compute it upfront to confirm later that we indeed computed a correct approximation:

```
  # Pick eigenvalue at eigen_index
  # (any eigenvalue / eigenvector pair will work).
  eigen_index = 0
  phi = np.real(np.log(eigvals[eigen_index]) / (2j*np.pi))
  if phi < 0:
    phi += 1
```

For the overall circuit, note how we initialize the state `psi` with `t` qubits in state $|0\rangle$ tensored with another state that is initialized with an eigenvector.

```
  # Make state + circuit to estimate phi.
  # Pick eigenvector 'eigen_index' to match the eigenvalue.
  psi = state.zeros(t) * state.State(eigvecs[:, eigen_index])
```

After we have this initialized state, we connect it with exponentiated operators, "unpacking" the binary fractions from the phase. We then run the inverse QFT on the resulting state.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
  psi = expo_u(psi, u, t)
  psi = ops.Qft(t).adjoint()(psi)
```

The heart of this circuit is the controlled connection of the operators taken to powers of 2, which is implemented in `expo_u` (naming is difficult):

```
def expo_u(psi: state.State, u: ops.Operator, t: int) -> state.State:
  """Exponentiate U."""

  psi = ops.Hadamard(t)(psi)
  for idx, inv in enumerate(range(t-1, -1, -1)):
    u2 = u
    for _ in range(idx):
      u2 = u2(u2)
    psi = ops.ControlledU(inv, t, u2)(psi, inv)
  return psi
```

All that is left to do is to simulate a measurement by picking the state with the highest probability, computing the binary fraction from this state, and comparing the result against the target value. Since we have limited bits to represent the result, we allow an error margin of 2%. More bits for t will make the circuit run slower but also improve the error margins.

```
  # Find state with highest measurement probability and show results.
  maxbits, maxprob = psi.maxprob()
  phi_estimate = sum(maxbits[i] * 2**(-i-1) for i in range(t))

  delta = abs(phi - phi_estimate)
  print('Phase    : {:.4f}'.format(phi))
  print('Estimate: {:.4f} delta: {:.4f} probability: {:5.2f}%'
        .format(phi_estimate, delta, maxprob * 100.0))
  if delta > 0.02 and phi_estimate < 0.98:
    print('*** Warning: Delta is large')
```

There is the potential of `delta` to be larger than two percent, especially when not enough bits were reserved for `t`. Another interesting error case is when the eigenvalue rounds to 1.0. In this case, all digits after the dot are 0. As a result, the estimated value from the binary fractions will also be 0.0 instead of the correct value of 1.0. The code warns about this case.

The result should look similar to the following output. Note that the highest probability found may not be close to 1.0. This means that when measured on a real, probabilistic quantum computer, we would obtain a fairly noisy result, with the correct solution hopefully showing enough distinction from other measurements.

```
Estimating 3 qubits random unitary eigenvalue with 6 bits of accuracy
Phase    : 0.5180
Estimate: 0.5156 delta: 0.0024 probability: 31.65%
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
Phase   : 0.3203
Estimate: 0.3125 delta: 0.0078 probability:  7.30%
[...]
Phase   : 0.6688
Estimate: 0.6719 delta: 0.0030 probability: 20.73%
```

6.5 Shor's Algorithm

Shor's algorithm for number factorization is the one algorithm that has sparked a tremendous amount of interest in quantum computing (Shor, 1994). The internet's RSA (Rivest, Shamir, Adlemen) encryption algorithm (Rivest et al., 1978) is based on the assumption that number factoring is an intractable problem. If quantum computers could crack this code, it would have severe implications.

Shor's algorithm is complex to implement, at least with the background of the material presented so far. To factor numbers like 15 or 21, it requires a large number of qubits and a very large number of gates, in the order of many thousands.

This looks like a great challenge, so let's dive right in. The algorithm has two parts:

1. It has a classical part, grounded in number theory, which relies on modular arithmetic and a process called order finding.
2. Order finding is intractable when done classically but can be mapped efficiently to a probabilistic quantum algorithm.

Correspondingly, we split the description of the algorithm into two parts. The classical part is discussed in this section, and the quantum part will be discussed in Section 6.6 on order finding.

6.5.1 Modular Arithmetic

Modular arithmetic is a complete arithmetic over integers that wrap around a given number, called the modulus, and considers the remainder. The modulus *mirrors* the C++ or Python percent operators, but it is not quite the same. One definition is,

$$a \equiv b \bmod N \Rightarrow b \equiv qN + a, \text{for some } q.$$

Or, equivalently,

$$a \equiv b \bmod N \Rightarrow a \bmod N \equiv b \bmod N.$$

Two numbers are *congruent* mod N if their modulus is the same, in which case they are in the same equivalence class. Here are examples for a modulus of 12:

$$15 \equiv 3 \bmod 12,$$
$$15 \equiv -9 \bmod 12.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

The numbers $15, 3$, and -9 are in the same mod 12 equivalence class. Note that in Python, applying the % operator would yield `-9 % 12 = 3`. Simple algebraic rules hold:

$$(x + y) \bmod N \equiv x \bmod N + y \bmod N,$$
$$(xy) \bmod N \equiv (x \bmod N)(y \bmod N).$$

We can use these rules to simplify computation with large numbers, for example:

$$(121 + 241) \bmod 12 \equiv 1 + 1 = 2,$$
$$(121 \cdot 241) \bmod 12 \equiv 1 \cdot 1 = 1.$$

6.5.2 Greatest Common Divisor

We will need to compute the *greatest common divisor* (GCD) of two integers. To reiterate, for two numbers, we break the numbers down into their prime factors and find the largest common factor. For example, the GCD of the integers 15 and 21 is 3:

$$15 = 3 \cdot 5,$$
$$21 = 3 \cdot 7.$$

Of course, we compute the GCD with the famous Euclidean algorithm:

```
def gcd(a: int, b: int) -> int:
    while b != 0:
        t = b
        b = a % b
        a = t
    return a
```

6.5.3 Factorization

Now we shall see how to use modular arithmetic and the GCD to factor a large number into two primes. We are only considering numbers that have two prime factors. Why is this important? In general, any number can be factored into several prime factors p_i:

$$N = p_0^{e_0} p_1^{e_1} \cdots p_{n-1}^{e_{n-1}}.$$

But the factoring is most difficult if N has just two prime factors of roughly equal length. This is why this mechanism is used in RSA encryption. Hence, we assume that

$$N = pq.$$

We can rephrase this problem in an interesting way. The problem of factoring a large number N into two primes is equivalent to solving this equation

$$x^2 \equiv 1 \bmod N. \tag{6.3}$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

There are two trivial solutions to this equation: $x = 1$ and $x = -1$. Are there other solutions? In the following, the typical examples for N are 15 and 21. As we will see later, this is mostly determined by the number of qubits we will be able to simulate.

Let us pick 21 as our example integer. We will iterate over all values from 0 to N and see whether we find another x for which above Equation (6.3) holds:

```
1*1 =  1 =  1 mod N
2*2 =  4 =  4 mod N
3*3 =  9 =  9 mod N
4*4 = 16 = 16 mod N
5*5 = 25 =  4 mod N
6*6 = 36 = 15 mod N
7*7 = 49 =  8 mod N
8*8 = 64 =  1 mod N
[...]
```

Indeed, we found another x for which this equation holds, $x = 8$. We can turn the search around and, instead of looking for the n in $n^2 = 1 \bmod N$, we search for the n with a given constant c in $c^n = 1 \bmod N$. This is the mechanism we will use during order finding. Here is an example with $c = 2$:

```
2^0 =  1 =  1 mod N
2^1 =  2 =  2 mod N
2^2 =  4 =  4 mod N
2^3 =  8 =  8 mod N
2^4 = 16 = 16 mod N
2^5 = 32 = 11 mod N
2^6 = 64 =  1 mod N
```

Since:

$$x^2 \equiv 1 \bmod N,$$
$$x^2 - 1 \equiv 0 \bmod N.$$

We can factor this via the quadratic formula into:

$$(x + 1)(x - 1) \equiv 0 \bmod N.$$

The modulo 0 means that N divides this product. We can therefore find the prime factors by computing:

```
factor1 = gcd(N, x+1)
factor2 = gcd(N, x-1)
```

This looks easy but suffers from the "little technical problem" of having to find that number x. In the classical case, our only options are to either iterate over all numbers or pick random values, square them, and check whether we hit a modulo 1 number.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Picking random values means that the birthday paradox applies,[3] and the probability of finding the right value for an N-bit number is roughly $\sqrt{2N}$. The is completely intractable for the large numbers used in internet encryption and numbers with lengths of 1024 bits, 4096 bits, and higher. What now?

6.5.4 Period Finding

We apply the following, and somewhat unexpected, three next steps; later we will find an efficient quantum algorithm for step 2.

Step 1 – Select Seed Number

We pick a random number $a < N$ that does not have a nontrivial factor in common with N. We also say that a and N are *coprime*. This can be tested with the help of the GCD. If their GCD is 1, the two numbers do not have a common factor and are coprime. If a does divide N, we got lucky and have found a factor already.

Step 2 – Find Order

Find the *powers modulo N*, with this sequence:

$$\begin{aligned} a^0 \bmod N &= 1, \\ a^1 \bmod N &= \ldots, \\ a^2 \bmod N &= \ldots, \\ &\vdots \end{aligned}$$

With the help of this function of a, N, and x (with a and N known, x unknown):

$$f_{a,N}(x) = a^x \bmod N.$$

Number theory guarantees that for any coprime a of N, this function will compute a result of 1 for some $x < N$. Once the sequence produces the 1 (for $x > 0$), the sequence of numbers computed so far will repeat itself. Remember that the sequence starts with the exponent of 0, resulting in $a^x = 1$. The length of the sequence, typically named r, is called the *order*, or period, of the function:

$$f_{a,N}(s + r) = f_{a,N}(s).$$

We will see how to construct a quantum algorithm to find the order in the next section. For now, let's just pretend we have an efficient way to compute it.

[3] https://en.wikipedia.org/wiki/Birthday_problem.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Step 3 – Factor

Once we have the order, how does it help us to get the factors of N?

If we find an order r that is an odd number, we give up, throw the result away, and try a different initial value of a in step 1.

If we find an order r that is an even number, we can use what we discovered earlier, namely, we can get the factors if we can find the x in this equation:

$$x^2 \equiv 1 \bmod N.$$

We just found in step 2 above that:

$$a^r \equiv 1 \bmod N.$$

Which we can rewrite as the following, if r is even:

$$\left(a^{r/2}\right)^2 \equiv 1 \bmod N.$$

This means we can now compute the factors similar to above, (with $r =$ `order`) as:

```
factor1 = gcd(N, a ** (order // 2) + 1)
factor2 = gcd(N, a ** (order // 2) - 1)
```

There is another little (as in, actually little) caveat – we do not know whether a given initial value of a will result in an even or odd order. It can be shown that the probability of getting an even order is $1/2$. We might have to run the algorithm multiple times.

These three steps, select seed number, find ordering, and factor, are the core of Shor's algorithm, minus the quantum parts. Let us write some code to explore the concepts this far before explaining quantum order finding in Section 6.6.

6.5.5 Playground

In this section, we will pick random numbers and apply the ideas from above. We still compute the order and derive the prime factors classically. Since our numbers are small, this is still tractable. Let us write a few helper functions first (the full source code is in file `src/shor_classic.,py`).

When picking a random number to play with, we must make sure that it is, indeed, factorizable and not prime:

```
def is_prime(num: int) -> bool:
  """Check to see whether num can be factored at all."""

  for i in range(3, num // 2, 2):
    if num % i == 0:
      return False
  return True
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

The algorithm requires picking a random number to seed the process. This number must not be a relative prime of the larger number, or the process might fail:

```
def is_coprime(num: int, larger_num: int) -> bool:
  """Determine if num is coprime to larger_num."""

  return math.gcd(num, larger_num) == 1
```

Find a random, odd, nonprime number in the range of numbers from `fr` to `to` and also find a corresponding coprime:

```
def get_odd_non_prime(fr: int, to: int) -> int:
  """Get a non-prime number in the range."""

  while True:
    n = random.randint(fr, to)
    if n % 2 == 0:
      continue
    if not is_prime(n):
      return n

def get_coprime(larger_num: int) -> int:
  """Find a number < larger_num which is coprime to it."""

  while True:
    val = random.randint(3, larger_num - 1)
    if is_coprime(val, larger_num):
      return val
```

And finally, we will need a routine to compute the order of a given modulus. This routine is, of course, classical and iterates until it finds the result of 1 that is guaranteed to exist.

```
def classic_order(num: int, modulus: int) -> int:
  """Find the order classically via simple iteration."""

  order = 1
  while True:
    newval = (num ** order) % modulus
    if newval == 1:
      return order
    order += 1
  return order
```

Here is the main algorithm, which we execute many times over randomly chosen numbers. We first select a random a and N, as described above. N is the number we

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

want to factorize, so it must not be prime. The value a must not be a coprime. Once we have those, we compute the order:

```
def run_experiment(fr: int, to: int) -> (int, int):
  """Run the classical part of Shor's algorithm."""

  n = get_odd_non_prime(fr, to)
  a = get_coprime(n)
  order = classic_order(a, n)
```

All that's left is to compute the factors from the even order and to print and check the results:

```
  factor1 = math.gcd(a ** (order // 2) + 1, n)
  factor2 = math.gcd(a ** (order // 2) - 1, n)
  if factor1 == 1 or factor2 == 1:
    return None

  print('Found Factors: N = {:4d} = {:4d} * {:4d} (r={:4})'.
        format(factor1 * factor2, factor1, factor2, order))
  if factor1 * factor2 != n:
    raise AssertionError('Invalid factoring')

  return factor1, factor2
```

We run some 25 tests and should see results like the following. For random numbers up to 9,999, the order can already reach values of up to almost 4,000:

```
def main(argv):
  print('Classic Part of Shor\'s Algorithm.')
  for i in range(25):
    run_experiment(21, 9999)

[...]
Classic Part of Shor's Algorithm.
Found Factors: N = 3629 =  191 *   19 (r=1710)
Found Factors: N = 4295 =    5 *  859 (r=1716)
[...]
Found Factors: N = 2035 =    5 *  407 (r= 180)
Found Factors: N = 9023 = 1289 *    7 (r=3864)
Found Factors: N = 1781 =  137 *   13 (r= 408)
```

In summary, we have learned how to factor a number N into two prime factors based on order finding and modular arithmetic. Order finding for very large numbers

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

is intractable classically, but in the next section we will learn an efficient quantum algorithm for this task. The whole algorithm is quite magical, and it becomes even more so when considering the quantum parts!

6.6 Order Finding

In the last section, we learned how finding the order of a specific function classically lets us efficiently factor a number into its two prime factors. In this section, we discuss an effective quantum algorithm to replace this classical task. We start by stating an objective – finding the phase of a particular operator. Initially, it might not be apparent how this relates to finding the order, but no worries, we develop all the details in the next few sections.

Quantum order finding is phase estimation applied to this operator U:

$$U|y\rangle = |xy \bmod N\rangle. \tag{6.4}$$

Phase estimation needs an eigenvector in order to run correctly. Let us first find the eigenvalues of this operator. We know that eigenvalues are defined as:

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

We use a process similar to the power iteration process. We know that the eigenvalues must be of norm 1; otherwise, the probabilities in the state vector would not add up to 1. Thus we can state:

$$U^k|v\rangle = \lambda^k|v\rangle,$$

and substitute this into the operator of Equation (6.4). This is a key step that is, unfortunately, often omitted in the literature:

$$U^k|y\rangle = \left|x^k y \bmod N\right\rangle.$$

If r is now the order of $x \bmod N$, with $x^r = 1 \bmod N$, then we get this result:

$$U^r|v\rangle = \lambda^r|v\rangle = \left|x^r y \bmod N\right\rangle = |v\rangle.$$

From this we can derive:

$$\lambda^r = 1.$$

This means the eigenvalues of U are the rth *roots of unity*. A root of unity is a complex number that, when raised to some integer power n, yields 1.0. It is defined as:

$$\lambda = e^{2\pi i s/r} \quad \text{for } s = 0, \ldots, r-1.$$

With this result, we will show below that the eigenvectors of this operator are the following for order r and a value s with $0 \leq s < r$:

$$|v_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i k s/r} \left|a^k \bmod N\right\rangle.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

With phase estimation, we can find the eigenvalues $e^{2\pi i s/r}$. The final trick will be to get to the order from the fraction s/r.

There is, of course, a big problem – for the phase estimation circuit, we needed to know an *eigenvector*. Because we do not know the order r, we cannot know any of the eigenvectors. Here comes another smart trick. We do know that the operator in Equation (6.4) is a permutation operator. Following the pattern of modular arithmetic, states are uniquely mapped to other states with order r. In this context, we should interpret states as integers, with state $|1\rangle$ representing decimal 1, and state $|1001\rangle$ representing decimal 9. For all values less than r, this mapping is a 1:1 mapping. For our operator:

$$U|y\rangle = |xy \bmod N\rangle\,.$$

We see that state $|y\rangle$ is multiplied by $x \bmod N$. As we iterate over exponents, this becomes:

$$U^n|y\rangle = \left|x^n y \bmod N\right\rangle.$$

For our example above, with $a = 2$ and $N = 21$, each application multiplies the state of the input register by $2 \bmod N$. We started with $2^0 = 1 = 1 \bmod N$, corresponding to state $|1\rangle$. Then:

$$\begin{aligned}
U|1\rangle &= |2\rangle,\\
U^2|1\rangle &= UU|1\rangle = U|2\rangle = |4\rangle,\\
U^3|1\rangle &= |8\rangle,\\
U^4|1\rangle &= |16\rangle,\\
U^5|1\rangle &= |11\rangle,\\
U^6|1\rangle &= U^r|1\rangle = |1\rangle.
\end{aligned}$$

We can deduce that the first eigenvector of this operator is the *superposition of all states*. This is easy to understand from a simpler example.[4] Assume a unitary gate only permutes between the two states $|0\rangle$ and $|1\rangle$, with:

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

Applying U to the superposition of both these states leads to the following result with an eigenvalue of 1:

$$\begin{aligned}
U\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) &= \frac{U|0\rangle + U|1\rangle}{\sqrt{2}}\\
&= \frac{|1\rangle + |0\rangle}{\sqrt{2}} = \frac{|0\rangle + |1\rangle}{\sqrt{2}}\\
&= 1.0\frac{|0\rangle + |1\rangle}{\sqrt{2}}.
\end{aligned}$$

[4] https://quantumcomputing.stackexchange.com/a/15590/11582.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

For the operator in Equation (6.4), we can generalize to multiple basis states. The superposition of the basis states is an eigenvector of U with eigenvalue 1.0:

$$|u_1\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \left|a^k \bmod N\right\rangle.$$

We also deduced above that the other eigenvalues are of the form:

$$\lambda = e^{2\pi i s/r} \quad \text{for } s = 0, \ldots, r-1.$$

Let's look at the eigenstates where the phase of the kth basis state is proportional to k:

$$|u_1\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i k/r} \left|a^k \bmod N\right\rangle. \tag{6.5}$$

For our example, applying the operator to this eigenvector follows the permutation rules of the operator U ($|1\rangle \rightarrow |2\rangle, |2\rangle \rightarrow |4\rangle, \ldots$):

$$|u_1\rangle = \frac{1}{6}\left(|1\rangle + e^{2\pi i/6}|2\rangle + e^{4\pi i/6}|4\rangle + e^{6\pi i/6}|8\rangle + e^{8\pi i/6}|16\rangle + e^{10\pi i/6}|11\rangle\right),$$

$$U|u_1\rangle = \frac{1}{6}\left(|2\rangle + e^{2\pi i/6}|4\rangle + e^{4\pi i/6}|8\rangle + e^{6\pi i/6}|16\rangle + e^{8\pi i/6}|11\rangle + e^{10\pi i/6}|1\rangle\right).$$

We can pull out the factor $e^{-2\pi i/6}$ to arrive at:

$$\begin{aligned} U|u_1\rangle &= \frac{1}{6} e^{-2\pi i/6}\left(e^{\frac{2\pi i}{6}}|2\rangle + e^{\frac{4\pi i}{6}}|4\rangle + e^{\frac{6\pi i}{6}}|8\rangle + e^{\frac{8\pi i}{6}}|16\rangle + e^{\frac{10\pi i}{6}}|11\rangle + \underbrace{e^{\frac{12\pi i}{6}}}_{=1}|1\rangle\right) \\ &= e^{-2\pi i/6}|u_1\rangle. \end{aligned}$$

Note how the order $r = 6$ now appears in the denominator. To make this general for all eigenvectors, we multiply in a factor s:

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i k s/r} \left|a^k \bmod N\right\rangle.$$

As a result, for our operator, we now get a unique eigenvector for each integer $s = 0, \ldots, r-1$, with the following eigenvalues (note that if we added the minus sign to Equation (6.5) above, the minus sign here would disappear; we can ignore it):

$$e^{-2\pi i s/r}|u_s\rangle.$$

Furthermore, there is another important result from this: if we add up all these eigenvectors, the phases cancel out except for $|1\rangle$ (not shown here; it is voluminous, but not challenging). This helps us because now we can use $|1\rangle$ as the eigenvector input to the phase estimation circuit. Phase estimation will give us the following result:

$$\phi = \frac{s}{r}.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

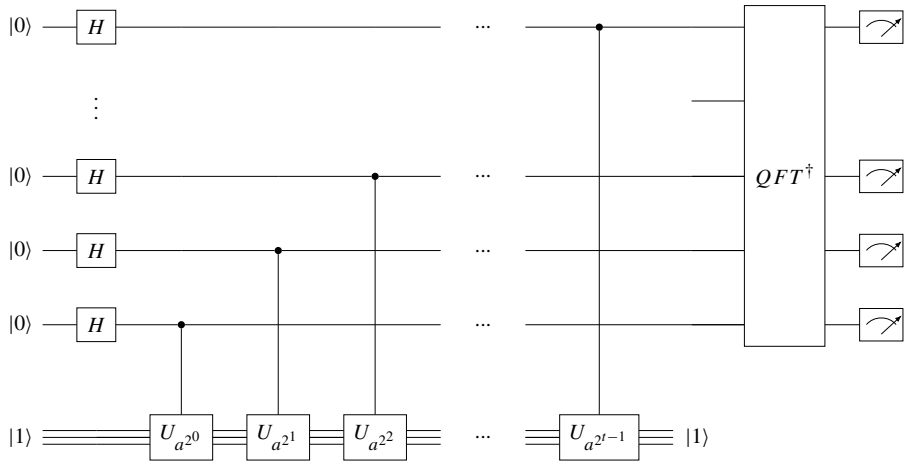


Figure 6.7 Order finding circuit.

But why is it that can we use $|1\rangle$ to initialize the phase estimation? Here[5] is an answer: Phase estimation should work for one eigenvector/eigenvalue pair. But in this case, we initialize the circuit with the sum of all eigenvectors, which we can consider as the *superposition* of all eigenstates. On measurement, the state will collapse to one of them. Which one? We do not know, but we do know from above that it will have a phase $\phi = s/r$. This is all we need to find the order with the method of continued fractions.

With all these preliminaries, we can now construct a phase estimation circuit as shown in Figure 6.7. For a given to-be-factored N, we define the number of bits necessary to represent N as $L = \log_2(N)$. The output of this circuit will be less than N, and we may need up to L output bits. We need to evaluate the unitary operation for at least N^2 values of x to be able to sample the order reliably, so we need $2L$ input bits:

$$\log_2 N^2 = 2 \log_2 N = 2L.$$

When we implement the algorithm, we will also use an ancilla register to store intermediate results from additions with a bit width of $L + 2$. (Typically you reserve a single bit for addition overflow, but we implement controlled addition, which requires an additional ancilla.) In summary, in order to factor a number fitting in L classical bits, we need $L + 2L + L + 2 = 4L + 2$ qubits. To factor 15, which fits into four classical bits, we will need 18 qubits. To factor 21, which fits in five classical bits, we will need 22 qubits. This number differs from what is typically quoted in the literature, which is theoretically closer to $2L + 1$. This discrepancy appears to be an artifact of the implementation details.

The big practical challenge for this circuit is how to implement the large unitary operator U. Our solution is based on a paper from Stephane Beauregard (Beauregard,

[5] https://quantumcomputing.stackexchange.com/q/15589/11582

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

2003) and a reference implementation by Tiago Leao and Rui Maia (Leao, 2021). It is a rather complex implementation, but, fortunately, we have seen most of the building blocks already.

To factor the number 21, we need 22 qubits and more than 20,000 gates. There are lots of QFTs and uncomputations, so the number of gates increases quickly. With our fast implementation, we can still simulate this circuit tractably. The overall implementation is about 250 lines of Python code.

As with all oracles or high-level unitary operators, you might expect some sort of quantum trick, a specially crafted matrix that just happens to compute the modulo exponentiation. Unfortunately, a magical matrix does not exist. Instead, we have to compute the modulo exponentiation explicitly with quantum gates by implementing addition and multiplication (by a constant) in the Fourier domain. We also have to implement the modulo operation, which is something we have not seen before.

We describe the implementation as follows: first, we outline the main routine driving the whole process. Then, we describe the helper routines, e.g., for addition. We have seen most of these before in other sections. Finally, we describe the code that builds the unitary operators and connects them to compute the phase estimate. We then get actual experimental results from the estimated phase with help of continued fractions.

6.6.1 Main Program

The implementation can be found in file `src/order_finding.py` in the open source repository. We get the numbers N and a from command line parameters. With these values, we compute the required bit width and construct three registers:

- `aux` for ancillae.
- `up` is the top register in the circuit shown in Figure 6.7. We will compute the inverse QFT on this register to get the phase estimation.
- `down` is the register that we will connect to the unitary operators. We also initialize it to $|1\rangle$.

```
def main(argv):
  print('Order finding.')

  number = flags.FLAGS.N
  a = flags.FLAGS.a

  # Test some of the basic routines.
  test_preliminaries(a, number)

  # The classical part are handled in 'shor_classic.py'
  nbits = number.bit_length()
  print('Shor: N = {}, a = {}, n = {} -> qubits: {}'
        .format(number,         a, nbits, nbits*4 + 2))
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
qc = circuit.qc('order_finding')

# Aux register for addition and multiplication.
aux = qc.reg(nbits+2, name='q0')

# Register for QFT. This reg will hold the resulting x-value.
up = qc.reg(nbits*2, name='q1')

# Register for multiplications.
down = qc.reg(nbits, name='q2')
```

We follow this with a one-to-one implementation of the circuit diagram in Figure 6.7. We apply Hadamard gates to all of the `up` register qubits and apply the X-gate to the `down` register to initialize it with $|1\rangle$. Note that in order to stay closer to the reference implementation (Leao, 2021), we interpret `down` in a reversed order. Then, we iterate over the number of up bits `(nbits * 2)` and create and connect the unitary gates with the Controlled-Multiply-Modulo (by power of 2) routine `cmultmodn`, which we show below. All of this is then followed by a final $QFT^{\dagger}$:

```
qc.had(up)
qc.x(down[0])
for i in range(nbits*2):
  cmultmodn(qc, up[i], down, aux, int(a**(2**i)), number, nbits)
inverse_qft(qc, up, 2*nbits, with_swaps=1)
```

Finally, we check the results. For the numbers given ($N = 15, a = 4$), we expect a result of 128 or 0 in the `up` register, corresponding to interpretations as binary fractions of 0.5 and 0.0. We will detail the next steps on how to get to the factors at the end of this section. This code snippet differs from the final implementation. Note again that we inverted the bit order with `[::-1]`:

```
# -- Results. An x-value of 128 would result in
#    the correct continuous fractions later.
print('Measurement...')
total_prob = 0.0
for bits in helper.bitprod(nbits*4 + 2):
  prob = qc.psi.prob(*bits)
  if prob > 0.01:
    print('Final x-value. Got: {:3d} Want: 128, probability: {:.3f}'
          .format(
           helper.bits2val(bits[nbits+2 : nbits+2 + nbits*2][::-1]),
           prob.real))
    total_prob += qc.psi.prob(*bits)
    if total_prob > 0.999:
      break

print(qc.stats())
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

And indeed, we will get this result with 50% probability. This is the reality of this algorithm – it is probabilistic. On a real machine, we might find only 1 and N and have to run the algorithm multiple times until we find at least one prime factor. In our infrastructure, of course, we can just peek at the resulting probabilities, no need to run multiple times.

```
[...]
Swap...
Uncompute...
Measurement...
Final x-value. Got:   0 Want: 128, probability: 0.250
Final x-value. Got:   0 Want: 128, probability: 0.250
Final x-value. Got: 128 Want: 128, probability: 0.250
Final x-value. Got: 128 Want: 128, probability: 0.250
Circuit Statistics
  Qubits: 18
  Gates : 10553
```

6.6.2 Support Routines

We use the variable a to compute a modulo number. Since we have to do uncomputation, we need the *modulo inverse* of this number. The modulo inverse of $x \bmod N$ is the number x_{inv}, such that $xx_{inv} = 1 \bmod N$. We can compute this number with the help of the extended Euclidean algorithm (Wikipedia, 2021c):

```
def modular_inverse(a: int, m: int) -> int:
  """Compute Modular Inverse."""

  def egcd(a: int, b: int) -> (int, int, int):
    """Extended Euclidian algorithm."""

    if a == 0:
      return (b, 0, 1)
    else:
      g, y, x = egcd(b % a, a)
    return (g, x - (b // a) * y, y)

  # Modular inverse of x mod m is the number x^-1 such that
  #   x * x^-1 = 1 mod m
  g, x, _ = egcd(a, m)
  if g != 1:
    raise Exception(f'Modular inverse ({a}, {m}) does not exist.')
  else:
    return x % m
```

We will run a large number of QFTs and inverse QFTs. Many of these operations are part of adding a quantum register with a known constant value. As we saw in

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Section 6.3 on quantum arithmetic, this makes the implementation of quantum addition easier. We precompute the angles to apply them directly to the target register:

```
def precompute_angles(a: int, n: int) -> List[float]:
  """Pre-compute angles used in the Fourier transform, for a."""

  # Convert 'a' to a string of 0's and 1's.
  s = bin(int(a))[2:].zfill(n)

  angles = [0.] * n
  for i in range(0, n):
    for j in range(i, n):
      if s[j] == '1':
        angles[n-i-1] += 2**(-(j-i))
    angles[n-i-1] *= math.pi
  return angles
```

We will need circuitry to compute addition, controlled addition, and double-controlled addition. The basic code is similar to quantum arithmetic with a constant that we have seen earlier in Section 6.3.1. We implement constant addition in `add` and controlled addition in `cadd` with `u1` and `cu1`. For the double-controlled addition in `ccadd`, we use the `ccphase` gate outlined below.

```
def add(qc, q, a: int, n: int, factor: float) -> None:
  """Add in fourier space."""

  angles = precompute_angles(a, n)
  for i in range(n):
    qc.u1(q[i], factor * angles[i])

def cadd(qc, q, ctl, a: int, n: int, factor: float -> None):
  """Controlled add in Fourier space."""

  angles = precompute_angles(a, n)
  for i in range(n):
    qc.cu1(ctl, q[i], factor * angles[i])

def ccadd(qc, q, ctl1: int, ctl2: int, a: int, n: int,
          factor: float) -> None:
  """Double-controlled add in Fourier space."""

  angles = precompute_angles(a, n)
  for i in range(n):
    ccphase(qc, factor*angles[i], ctl1, ctl2, q[i])
```

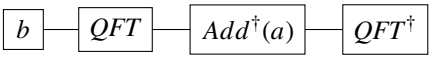
We need a double-controlled phase gate for the double-controlled `ccadd` above. In Section 3.2.7, we learned how to construct a double-controlled gate with the help of

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

their controlled root and adjoint. For rotations around an angle x, the root is just a rotation by $x/2$, and the adjoint of a rotation is a rotation in the other direction:

```
def ccphase(qc, angle: float, ctl1: int, ctl2: int, idx: int) -> None:
  """Controlled controlled phase gate."""

  qc.cu1(ctl1, idx, angle/2)
  qc.cx(ctl2, ctl1)
  qc.cu1(ctl1, idx, -angle/2)
  qc.cx(ctl2, ctl1)
  qc.cu1(ctl2, idx, angle/2)
```

Using the adjoint of the addition circuit, we get $(b-a)$ if $b \geq a$, and $(2^{n-1}-(a-b))$ if $b < a$. So we can use this to subtract and compare numbers. If $b < a$, then the most significant qubit will be $|1\rangle$. We utilize this qubit to control other gates later.



$$= \begin{cases} |b-a\rangle & \text{if } b \geq a, \\ |2^{n-1}-(a-b)\rangle & \text{if } b < a. \end{cases}$$

We implement QFT and $QFT^\dagger$ on the `up` register, this time with an option for swaps (which we actually don't use for this algorithm).

```
def qft(qc, up_reg, n: int, with_swaps: bool = False) -> None:
  """Apply the H gates and Cphases."""

  for i in range(n-1, -1, -1):
    qc.h(up_reg[i])
    for j in range(i-1, -1, -1):
      qc.cu1(up_reg[i], up_reg[j], math.pi/2**(i-j))

  if with_swaps:
    for i in range(n // 2):
      qc.swap(up_reg[i], up_reg[n-1-i])


def inverse_qft(qc, up_reg, n: int, with_swaps: bool = False) -> None:
  """Function to create inverse QFT."""

  if with_swaps == 1:
    for i in range(n // 2):
      qc.swap(up_reg[i], up_reg[n-1-i])

  for i in range(n):
    qc.had(up_reg[i])
    if i != n-1:
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
j = i+1
for y in range(i, -1, -1):
  qc.cu1(up_reg[j], up_reg[y], -math.pi / 2**(j-y))
```

6.6.3 Modular Addition

At this point, we know how to add numbers and to check whether a value has turned negative by checking the sign qubit. This means we should have all necessary ingredients for *modular* addition: we compute $a + b$, and subtract N if $a + b > N$.

We achieve this by adding an ancilla qubit in the initial state $|0\rangle$. We start by adding a and b as before. We also reserve an overflow bit. Then we use the adjoint of the adder to subtract N (a fancy way of saying that we apply a negative factor in the addition routines above). In order to get to the most significant qubit and determine if this result was negative, we have to perform $QFT^{\dagger}$. We connect the most significant qubit and the ancilla with a Controlled-Not gate. It will only be set to $|1\rangle$ if $a + b - N$ is negative.

After this, we go back into the Fourier domain with another QFT. If $a + b - N$ is negative, we use the ancilla qubit to control the addition of N to make the result positive again. The circuit is shown in Figure 6.8.

There is a resulting problem that is not easy to solve – the ancilla qubit is still entangled. It has turned into a junk qubit. We have to find a way to return it to its original state of $|0\rangle$, else it will mess up our results, as junk qubits make a habit of doing.

To resolve this, we use the almost identical circuit again, but with a twist. We observe the following, with register b in the state after the prior modulo operation:

$$(a + b) \bmod N \geq a \Rightarrow a + b < N.$$

This time we run an inverse addition to subtract a from the result above and compute $(a+b) \bmod N - a$. The most significant bit is going to be $|0\rangle$ if $(a+b) \bmod N \geq a$.

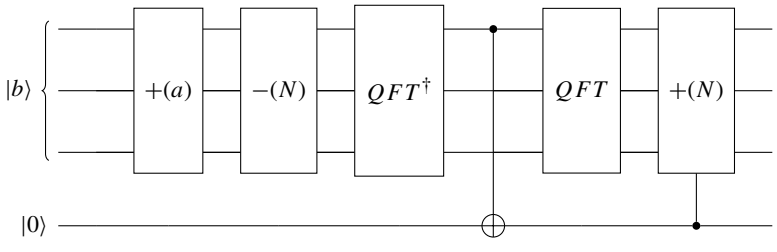


Figure 6.8 First half of the modular addition circuit.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

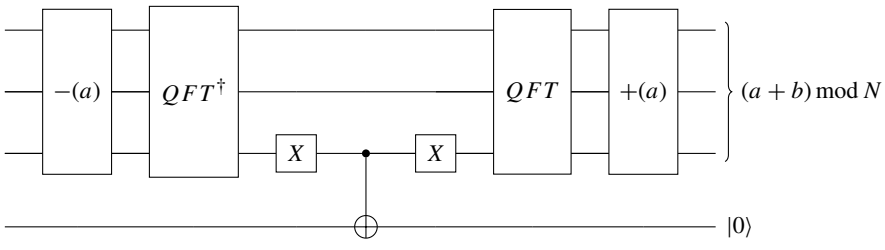


Figure 6.9 Second half of the modulo addition circuit, disentangling the ancilla.

We apply a NOT gate and use it as the controller for a Controlled-Not to the ancilla. With this, the ancilla has been restored. Now we have to undo what we just did. We apply another NOT gate to the most significant qubit, followed by a QFT and an addition of a to revert the initial subtraction. The end result is a clean computation of $(a + b) \bmod N$. In circuit notation, the second half of the circuit is shown in Figure 6.9. In code:

```
def cc_add_mod_n(qc, q, ctl1, ctl2, aux, a, number, n):
  """Circuit that implements doubly controlled modular addition by a."""

  ccadd(qc, q, ctl1, ctl2, a, n, factor=1.0)
  add(qc, q, number, n, factor=-1.0)
  inverse_qft(qc, q, n, with_swaps=0)
  qc.cx(q[n-1], aux)
  qft(qc, q, n, with_swaps=0)
  cadd(qc, q, aux, number, n, factor=1.0)

  ccadd(qc, q, ctl1, ctl2, a, n, factor=-1.0)
  inverse_qft(qc, q, n, with_swaps=0)
  qc.x(q[n-1])
  qc.cx(q[n-1], aux)
  qc.x(q[n-1])
  qft(qc, q, n, with_swaps=0)
  ccadd(qc, q, ctl1, ctl2, a, n, factor=1.0)
```

We will also need the inverse of this procedure. As before, and as explained in the section on uncomputation, we simply apply the inverse gates in the reverse order:

```
def cc_add_mod_n_inverse(qc, q, ctl1, ctl2, aux, a, number, n):
  """Inverse of the double controlled modular addition."""

  ccadd(qc, q, ctl1, ctl2, a, n, factor=-1.0)
  inverse_qft(qc, q, n, with_swaps=0)
  qc.x(q[n-1])
  qc.cx(q[n-1], aux)
  qc.x(q[n-1])
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

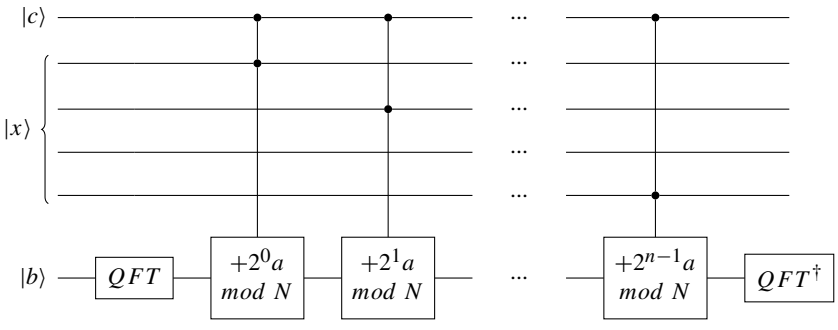


Figure 6.10 Circuit for controlled modular multiplication.

```
qft(qc, q, n, with_swaps=0)
ccadd(qc, q, ctl1, ctl2, a, n, factor=1.0)

cadd(qc, q, aux, number, n, factor=-1.0)
inverse_qft(qc, q, n, with_swaps=0)
qc.cx(q[n-1], aux)
qft(qc, q, n, with_swaps=0)
add(qc, q, number, n, factor=1.0)
ccadd(qc, q, ctl1, ctl2, a, n, factor=-1.0)
```

Uncomputing circuits like this is tedious. In Section 8.5.5 we show how to automate uncomputation in an elegant way.

6.6.4 Controlled Modular Multiplication

The next step is now to build a controlled modular multiplier from the modular adders we just constructed. Our circuit will be controlled by a qubit $|c\rangle$ and take the state $|c,x,b\rangle$ to the state $|c,x,b+(ax) \bmod N\rangle$, if $|c\rangle = |1\rangle$, else it will leave the original state intact.

We perform successive applications of the controlled modular addition gate, as controlled by the individual bits x_i of x. The bit positions correspond to powers of 2 using this identity:

$$(ax) \bmod N =$$
$$(\ldots(((2^0 a x_0) \bmod N + 2^1 a x_1) \bmod N) + \cdots + 2^{n-1} a x_{n-1}) \bmod N.$$

The circuit to compute this expression is shown in Figure 6.10.

As described in the section on uncomputation, in order to eliminate the entanglement with $|b\rangle$, we swap out $|x\rangle$ and uncompute the circuit after the swap. In code, we see three sections. In the first section, it computes the multiplication modulo N. In the second block, it connects the results with controlled gates to swap out $|x\rangle$ to the `aux` register (the `cswap` was introduced in Section 4.3 on quantum circuits). Finally, it uncomputes the results, as we have seen in Section 2.13 on reversible computing.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

This means that we must implement the inverse computation of the first block using the modular inverse.

```
def cmultmodn(qc, ctl, q, aux, a, number, n):
  """Controlled Multiply of q by number, with n bits."""

  print('Compute...')
  qft(qc, aux, n+1, with_swaps=0)
  for i in range(n):
    cc_add_mod_n(qc, aux, q[i], ctl, aux[n+1],
                 ((2**i)*a) % number, number, n+1)
  inverse_qft(qc, aux, n+1, with_swaps=0)

  print('Swap...')
  for i in range(n):
    qc.cswap(ctl, q[i], aux[i])
  a_inv = modular_inverse(a, number)

  print('Uncompute...')
  qft(qc, aux, n+1, with_swaps=0)
  for i in range(n-1, -1, -1):
    cc_add_mod_n_inverse(qc, aux, q[i], ctl, aux[n+1],
                         ((2**i)*a_inv) % number, number, n+1)
  inverse_qft(qc, aux, n+1, with_swaps=0)
```

In summary, the modular multiplication circuit performs:

$$|x\rangle|0\rangle \rightarrow |ax \bmod N\rangle|0\rangle.$$

We shall name this circuit CU_a. There is still a problem – the phase estimation algorithm requires powers of this circuit. Does this mean that we have to multiply this circuit n times with itself to get to $(CU_a)^n$ for each power of 2 as required by phase estimation? Fortunately, we don't. We simply compute a^n classically with:

$$(CU_a)^n = CU_{a^n}.$$

This can be seen at the top level in the code where we iterate over the calls to the modular arithmetic circuit (those expressions containing `2**i`).

6.6.5 Continued Fractions

We are very close to the final result. We mentioned in Section 6.6.1 that an expected result for the `up` register was 128. This was an interpretation of this register as an integer. However, we performed phase estimation, so we have to interpret the bits of the register as binary fractions. A value of 0 corresponds to a phase of 0.0, and a value of 128 corresponds to a phase of 0.5. We also know that phase estimation will give a phase of the following form with order r:

$$\phi = \frac{s}{r}.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Shouldn't this mean that if we could find a fraction of integers approximating this phase, we would have an initial guess for the order r?

To approximate a fractional value to an arbitrary degree of accuracy, we can use the technique of *continued fractions*.[6] Fortunately for us, an implementation of it already exists in the form of a Python library. We include the module:

```
import fractions
```

When we decode the x-register we have to interpret it as a binary fraction (note, again, how we interpret the register bits in a reversed order):

```
    phase = helper.bits2frac(
        bits[nbits+2 : nbits+2 + nbits*2][::-1], nbits*2)
```

We get the lowest denominator from the continued fractions algorithm. We also want to limit the accuracy via `limit_denominator` to ensure we get reasonably sized denominators:

```
  r = fractions.Fraction(phase).limit_denominator(number).denominator
```

With this `r`, we can then follow the explanations on the nonquantum part of Shor's algorithm and seek to compute the factors. We might get 1s, or we might get Ns, which are both useless. With a little luck and by following actual probabilities, we might just find one or two of the real factors.

```
    guesses = [math.gcd(a**(r//2)-1, number),
               math.gcd(a**(r//2)+1, number)]

    print('Final x: {:3d} phase: {:3f} prob: {:.3f} factors: {}'.
          format(intval, phase, prob.real, guesses))
```

6.6.6 Experiments

Let us run just a few examples to demonstrate that this machinery works. To factorize 15 with a value of a of 4, we run a circuit with 10,553 gates and obtain two sets of factors, the trivial ones with 1 and 15, but, Eureka! also the real factors of 3 and 5:

```
.../order_finding -- --a=4 --N=15
Final x-value int:   0 phase: 0.000000 prob: 0.250 factors: [15, 1]
Final x-value int: 128 phase: 0.500000 prob: 0.250 factors: [3, 5]
Circuit Statistics
  Qubits: 18
  Gates : 10553
```

[6] https://en.wikipedia.org/wiki/Continued_fraction.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

To factor 21 with an `a` value of 5, the required number of qubits grows from 18 to 22, thus increasing the number of gates to over 20,000 and the runtime by roughly a factor of 8×. Other than the trivial factors, the routine finds one of the real factors with the value 3.

```
Final x-value int:   0 phase: 0.000000 prob: 0.028 factors: [21, 1]
Final x-value int: 512 phase: 0.500000 prob: 0.028 factors: [1, 3]
Final x-value int: 853 phase: 0.833008 prob: 0.019 factors: [1, 21]
Final x-value int: 171 phase: 0.166992 prob: 0.019 factors: [1, 21]
Final x-value int: 683 phase: 0.666992 prob: 0.019 factors: [1, 3]
Circuit Statistics
  Qubits: 22
  Gates : 20671
```

Finally, factoring 35 with an initial `a` value of 4, uses over 36,000 gates and a runtime of approximately 60 minutes:

```
Final x-value int:    0 phase: 0.000000 prob: 0.028 factors: [35, 1]
Final x-value int: 2048 phase: 0.500000 prob: 0.028 factors: [1, 5]
Final x-value int: 1365 phase: 0.333252 prob: 0.019 factors: [1, 5]
Final x-value int: 3413 phase: 0.833252 prob: 0.019 factors: [7, 5]
Final x-value int:  683 phase: 0.166748 prob: 0.019 factors: [7, 5]
Final x-value int: 2731 phase: 0.666748 prob: 0.019 factors: [1, 5]
Circuit Statistics
  Qubits: 26
  Gates : 36373
```

You may want to experiment and perhaps convert this code to `libq` with the transpilation facilities described in Section 8.5. The code runs significantly faster in `libq`, which allows experimentation with much larger numbers of qubits. As a rough and unscientific estimate – factorization with 22 qubits runs for about two minutes on a standard workstation. After compilation to `libq`, it accelerates because of the sparse representation and takes less than five seconds to complete, a speedup factor of over 25×. Factoring 35 with 26 qubits takes about an hour, but with `libq` it takes about three minutes, a still significant speedup of about 20×.

To summarize, the algorithm as a whole – from the classical parts to the quantum parts and finding the order with continued fraction – is truly magical. No wonder it has gotten so much attention and stands out as one of the key contributors to today's interest in quantum computing.

6.7 Grover's Algorithm

Grover's algorithm is one of the fundamental algorithms of quantum computing (Grover, 1996). It allows searching over N elements in a domain in $O(\sqrt{N})$ time. By

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

"searching" we mean that there is a function $f(x)$ and one (or more) special inputs x' for which:

$$f(x) = 0 \quad \forall x \neq x',$$
$$f(x) = 1 \quad x = x'.$$

The classical algorithm to find x' is of complexity $O(N)$ in the worst case. It needs to evaluate all possible inputs to f. Strictly speaking, $N-1$ steps are required, because once all the elements, including the penultimate one, have returned 0, we know that the last element must be the elusive x'. Being able to do this with complexity $O\left(\sqrt{N}\right)$ is, of course, an exciting prospect.

To understand and implement the algorithm, we first describe the algorithm at a high level in fairly abstract terms. We need to learn two new concepts – *phase inversion* and *inversion about the mean*. Once these concepts are understood, we detail several variants of their implementation. We finally combine all the pieces into Grover's algorithm and run a few experiments.

6.7.1 High-Level Overview

At the high level, the algorithm performs the following steps:

1. Create an equal superposition state $|+ + \cdots +\rangle$ by applying Hadamard gates to an initial state $|00\ldots0\rangle$.
2. Construct a *phase inversion* operator U_f around the special input $|x'\rangle$, defined as:

$$U_f = I^{\otimes n} - 2|x'\rangle\langle x'|.$$

3. Construct an *inversion about the mean* operator $U_\perp$, defined as:

$$U_\perp = 2(|+\rangle\langle +|)^{\otimes n} - I^{\otimes n}.$$

4. Combine $U_\perp$ and U_f into the Grover operator G (in this notation, U_f is applied first):

$$G = U_\perp U_f.$$

5. Iterate k times and apply G to the state. We derive the iteration count k below. The resulting state will be close to the special state $|x'\rangle$:

$$G^k\,|+\rangle^{\otimes n} \sim |x'\rangle.$$

This basically explains the whole procedure. Some of you may look at this, shrug, and understand it right away. For the rest of us, the next sections explain this procedure in great detail, sometimes in multiple different ways. Grover's algorithm is foundational, so we want to make sure we understand and appreciate it fully.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

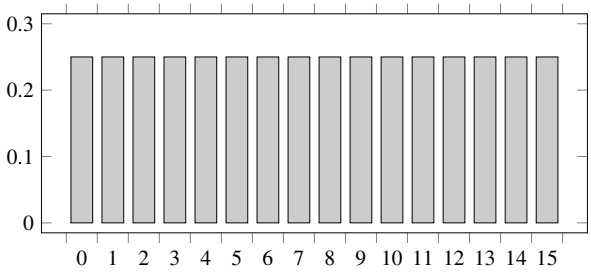


Figure 6.11 Equally distributed probability amplitudes.

6.7.2 Phase Inversion

The first new concept we need to learn about it phase inversion. Let's assume a given state $|\psi\rangle$ with probability amplitudes c_x:

$$|\psi\rangle = \sum_x c_x |x\rangle.$$

For simplicity, let's assume all c_i are equal $1/\sqrt{N}$ (remember that $N = 2^n$ for n qubits). Figure 6.11 shows a bar graph where the x-axis enumerates the states $|x_i\rangle$, and the y-axis plots the height of the corresponding probability amplitudes c_i. It is safe to ignore the *actual* values; we are just trying to make a point.

Let's now further assume that we want to consider one of these input states as special, corresponding to the element $|x'\rangle$ mentioned above. Phase inversion converts the original state into a state where the phase has been *negated* for the special element $|x'\rangle$:

$$|\psi\rangle = \sum_x c_x |x\rangle \quad \rightarrow \quad |\psi\rangle = \sum_{x \neq x'} c_x |x\rangle - c_{x'} |x'\rangle.$$

In the chart in Figure 6.12, we negated the phase of state $|4\rangle$, which should serve as our special state $|x'\rangle$.

To relate this back to the function $f(x)$ we are trying to analyze, we use phase inversion to negate the phase for the special elements only, which we can express with this closed form:

$$|\psi\rangle = \sum_x c_x |x\rangle \quad \rightarrow_{inv} \quad |\psi\rangle = \sum_x c_x (-1)^{f(x)} |x\rangle. \tag{6.6}$$

A key aspect of this procedure is that the function f has to be known. Because how else can we implement and perform this operation? There is an important distinction: Even though an implementation has to know the function, observers who try to reconstruct and measure the function would still have to go through N steps in the classical case, but only $\sqrt{N}$ in the quantum case. This is still different from, say, finding an element meeting certain criteria in a database.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

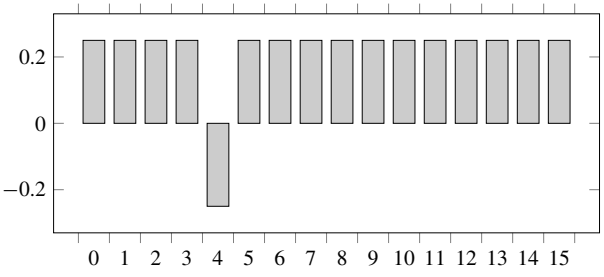


Figure 6.12 Probability amplitudes after phase inversion.

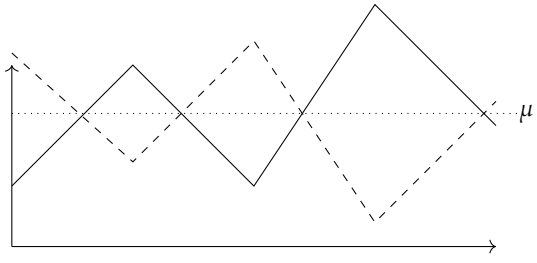


Figure 6.13 An example of (solid line) random data and (dashed line) its inversion about the mean.

6.7.3 Inversion About the Mean

The second new concept is *inversion about the mean*. We can compute the mean μ ("mu") of the probability amplitudes c_x of the original state:

$$\mu = \Big(\sum_x c_x\Big)/N.$$

Inversion about the mean is the process of mirroring each c_x across the mean. To achieve this, we take each value's distance from the mean, which is $\mu - c_x$, and add it to the mean. For values that were above the mean, $\mu - c_x$ is negative, and the value is reflected below the mean. Conversely, for values that were below the mean, $\mu - c_x$ is positive, and the values are being reflected up. An example with a random set of values is shown in Figure 6.13 (solid line).

For each c_i we compute:

$$c_i \rightarrow \mu + (\mu - c_i) = (2\mu - c_i).$$

This reflects each value about the mean. For the example in Figure 6.13, the reflected values are shown with a dashed line. Each amplitude c_i has been reflected about the mean of all amplitudes.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

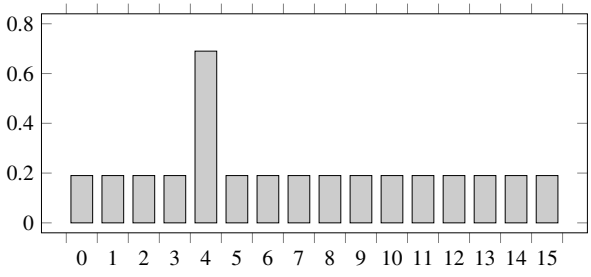


Figure 6.14 Distribution of amplitudes after phase and mean inversion.

$$c_x \to \mu + (\mu - c_x) = (2\mu - c_x),$$
$$\sum_x c_x|x\rangle \to \sum_x (2\mu - c_x)|x\rangle. \tag{6.7}$$

6.7.4 Simple Numerical Example

With these new concepts, we can now describe a step in Grover's algorithm with a simple example with 16 states, as shown in Figure 6.11. Here is how it works:

1. **Initialization.** As seen in Section 6.7.1, we put states in superposition and start with all states being equally likely with amplitude $1/\sqrt{N}$.
2. **Phase inversion.** Apply phase inversion as shown above in Equation (6.6). The phase of the special element becomes negative, thus pushing the mean of all amplitudes down. In our example with 16 states and amplitude $1/\sqrt{16} = 0.25$, the overall mean is roughly pushed down to $(0.25 * 15 - 0.25)/16 = 0.22$.
3. **Inversion around the mean**. This will push down amplitudes of 0.25 to $0.22 + (0.22 - 0.25) = 0.19$, but amplify the special element to a value of $0.22 + (0.22 + 0.25) = 0.69$.

Now rinse and repeat steps 2 and 3. For our artificial amplitude example above, a single step would turn the initial state into the state shown in Figure 6.14.

6.7.5 Two-Qubit Example

Let's make this more concrete and visualize the procedure using an example with two qubits. A geometrical interpretation is shown in Figure 6.15. In a two-qubit system, our special element x' and its corresponding outer product shall be:

$$|x'\rangle = |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{and} \quad |x'\rangle\langle x'| = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The solution $|x'\rangle$ corresponds to the solution space $|\beta\rangle$ in Figure 6.15. The phase inversion operator U_f from step 2 in Section 6.7.1 then becomes the following

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

(note that in the implementation below, we use a different methodology to get this operator):

$$U_f = I - 2|x'\rangle\langle x'| = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

We know how to create an equal superposition state $|s\rangle = |++\rangle$. The state $|x^{\perp}\rangle$ *orthogonal* to $|x'\rangle$ is very close to $|s\rangle$; $|s\rangle$ is *almost* orthogonal to $|x'\rangle$.

$$|s\rangle = H^{\otimes 2}|00\rangle = |++\rangle = \frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad |x^{\perp}\rangle = \frac{1}{\sqrt{3}}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Note that $|x^{\perp}\rangle = |s\rangle - |x'\rangle$ is the equal superposition state $|s\rangle$ with $|x'\rangle$ removed; it corresponds to the axis $|\alpha\rangle$ in Figure 6.15. The state $|\psi\rangle$ in the figure corresponds to the initial $|s\rangle$. It is easy to see how applying the operator U_f inverts the phase of the $|x'\rangle$ component in $|s\rangle$:

$$U_f\,|s\rangle = \frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix}.$$

In Figure 6.15, this corresponds to a reflection of the state $|\psi\rangle$ (which is our $|s\rangle$) about the α-axis. The inversion about the mean operator $U_{\perp}$, as defined in step 3) above, is:

$$\begin{aligned} U_{\perp} &= 2(|+\rangle\langle+|)^{\otimes 2} - I^{\otimes 2} \\ &= 2|s\rangle\langle s| - I^{\otimes 2}. \end{aligned}$$

The operator $U_{\perp}$ reflects $U_f|\psi\rangle$ about the original state $|s\rangle$ into the new state $U_{\perp}U_f|\psi\rangle = |11\rangle$. For our example with just two qubits, a single iteration is all that is needed to move state $|s\rangle$ to $|x'\rangle$. In code:

```
    x = state.bitstring(1, 1)
    s = ops.Hadamard(2)(state.bitstring(0, 0))

    Uf = ops.Operator(ops.Identity(2) - 2 * x.density())
    Ub = ops.Operator(2 * s.density() - ops.Identity(2))
    (Ub @ Uf)(s).dump()
>>
|11> (|3>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

The iteration count of 1 agrees with Equation (6.11), which we will derive next.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

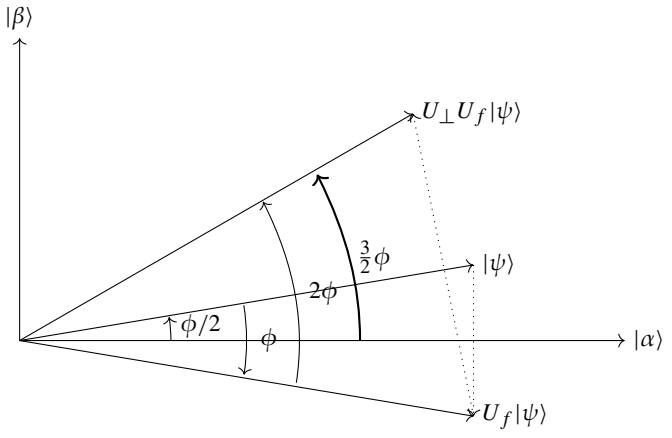


Figure 6.15 Geometric interpretation of a Grover rotation.

6.7.6 Iteration Count

How many iterations k should we perform? How do we know when to stop? It turns out we need exactly k iterations, where:

$$k = \frac{\pi}{4}\sqrt{N}.$$

How do we arrive at this number? First, we define two subspaces: the space of all the states that do not contain a solution and the space of states that are special. Note that in the implementation, we search for just one special element $|x'\rangle$, but here we generalize this derivation to search for M solutions in a population of N elements.

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}}\sum_x |x\rangle,$$
$$|\beta\rangle = \frac{1}{\sqrt{M}}\sum_{x'} |x'\rangle.$$

We can define the whole state as a composite of these two subspaces:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle. \tag{6.8}$$

We can visualize this space in two dimensions, where the x-axis corresponds to state space $|\alpha\rangle$ and the y-axis to solution space $|\beta\rangle$, as shown in Figure 6.15.

Application of phase inversion (let's call the corresponding operator U_f, as before) reflects the state about $|\alpha\rangle$. This, in essence, negates the second part of the superposition, similar to the effect of a Z-gate on a single qubit, where a and b are the probability amplitudes for the subspaces α and β:

$$U_f(a|\alpha\rangle + b|\beta\rangle) = a|\alpha\rangle - b|\beta\rangle.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

The inversion around the mean (let's again call this operator $U_\perp$) then performs another reflection about the vector $|\psi\rangle$. The two reflections amount to a rotation, which means the state remains in the space spanned by $|\alpha\rangle$ and $|\beta\rangle$. Furthermore, the state incrementally rotates towards the solution space $|\beta\rangle$. We have seen in (6.8) above that:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle.$$

We can geometrically position the state vector with simple trigonometry. We define the initial angle between $|\psi\rangle$ and $|\alpha\rangle$ as $\phi/2$. Equation (6.9) is important; we will use it in Section 6.9 on quantum counting:

$$\begin{aligned} \cos\left(\frac{\phi}{2}\right) &= \sqrt{\frac{N-M}{N}}, \\ \sin\left(\frac{\phi}{2}\right) &= \sqrt{\frac{M}{N}}, \\ |\psi\rangle &= \cos\left(\frac{\phi}{2}\right)|\alpha\rangle + \sin\left(\frac{\phi}{2}\right)|\beta\rangle. \end{aligned} \tag{6.9}$$

From Figure 6.16, we can see that after phase inversion and inversion around the mean, the state has rotated by ϕ towards $|\beta\rangle$. The angle between $|\alpha\rangle$ and $|\psi\rangle$ is now $3\phi/2$. We call the combined operator the Grover operator $G = U_\perp U_f$:

$$G|\psi\rangle = \cos\left(\frac{3\phi}{2}\right)|\alpha\rangle + \sin\left(\frac{3\phi}{2}\right)|\beta\rangle.$$

From this, we see that repeated application of the Grover operator G takes the state to:

$$G^k|\psi\rangle = \cos\left(\frac{2k+1}{2}\phi\right)|\alpha\rangle + \sin\left(\frac{2k+1}{2}\phi\right)|\beta\rangle.$$

In order to maximize the probability of measuring $|\beta\rangle$, the term $\sin\left(\frac{2k+1}{2}\phi\right)$ ought to be as close to 1.0 as possible. Taking the arcsin of the expression yields:

$$\begin{aligned} \sin\left(\frac{2k+1}{2}\phi\right) &= 1 \\ \frac{2k+1}{2}\phi &= \pi/2 \\ k &= \frac{\pi}{2\phi} - \frac{1}{2} = \frac{\pi}{4\frac{\phi}{2}} - \frac{1}{2}. \end{aligned} \tag{6.10}$$

Note that the iteration number must be an integer, so the question we face now is what to do with the $-1/2$. We could ignore it, use it for rounding up, or for rounding down. In our implementation, we chose to ignore it. For our examples below, the probabilities for finding the solutions are around 40% or higher, and this term has no impact.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Now let's solve for k. From Equation (6.9), we know that:

$$\sin\left(\frac{\phi}{2}\right) = \sqrt{\frac{M}{N}}.$$

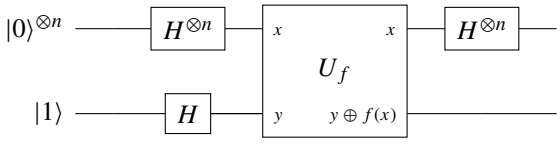
We use the approximation that for small angles, $\sin(x) \approx x$. Substituting in $\frac{\phi}{2} = \sqrt{M/N}$ and $M = 1$ into Equation (6.10), we find the final result for the number of iterations k:

$$k = \frac{\pi}{4}\sqrt{\frac{N}{M}} = \frac{\pi}{4}\sqrt{N}. \tag{6.11}$$

6.7.7 Implementation of Phase Inversion

We will present three different ways to implement phase inversion. We have already seen the mathematical way, which simply computes the operator $U_f = I - 2|x'\rangle\langle x'|$. But how do we construct an actual circuit for this? A second strategy will use an oracle operator, which we suspect can be implemented as a circuit (we also want to demonstrate the utility of the oracle operator one more time). Finally, we develop an actual quantum circuit for phase inversion.

Our second implementation strategy uses a mechanism we have seen before: the oracle operator! The oracle structure is similar to the Deutsch–Jozsa oracle – the input is a whole register of (initially $|0\rangle$, then equal superposition) states:



It is important to note that the bottom ancilla qubit is initialized as $|1\rangle$. The Hadamard gate puts it into state $|-\rangle$. This is important because it means that the input state is transformed by U_f into the desired state:

$$|\psi\rangle = \sum_x c_x|x_i\rangle \quad \rightarrow_{inv} \quad |\psi\rangle = \sum_x c_x(-1)^{f(x)}|x\rangle.$$

How so? State $|-\rangle$ is:

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

Since we use an oracle, all input values are computed in parallel. If $f(x) = 0$, the bottom qubit in state $|-\rangle$ is XOR'ed with 0, which means the state of the qubit remains unmodified:

$$|-\rangle \rightarrow |-\rangle.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

If $f(x) = 1$, the bottom qubit in state $|-\rangle$ is XOR'ed with 1, which means the state changes to:

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

This means it gets a phase:

$$|-\rangle \rightarrow -|-\rangle.$$

For the ancilla, the output is now:

$$(-1)^{f(x)}|-\rangle.$$

The combination of input bits plus the ancilla becomes:

$$\sum_x c_x|x\rangle(-1)^{f(x)}|-\rangle.$$

We can slightly rearrange the terms, ignore the ancilla, and arrive at the exact form we were looking for:

$$|\psi\rangle = \sum_x c_x(-1)^{f(x)}|x\rangle.$$

6.7.8 Phase Inversion Operator

We constructed the phase inversion operator as a giant matrix, which is inefficient for larger numbers of qubits. Here is a more efficient construction with a multi-controlled X-gate. It will be of higher performance, despite the fact that $n - 2$ ancilla qubits are required, as outlined in Section 3.2.8. We are trying to compute a unitary operator U such that:

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle \text{ where } \begin{cases} f(x) = 0 & \forall x \neq x', \\ f(x) = 1 & x = x'. \end{cases}$$

We only want to apply the XOR for the special state $|x'\rangle$ for which $f(x) = 1$. This means we can multi-control the final qubit as shown in Figure 6.16, ensuring that all control bits are $|1\rangle$.

6.7.9 Implementation of Inversion about the Mean

To reiterate, inversion about the mean is this procedure:

$$\sum_x c_x|x\rangle \rightarrow \sum_x (2\mu - c_x)|x\rangle.$$

In matrix form, we can accomplish this by multiplying the state vector with a matrix that has $2/N$ at each element, except for the diagonal elements, which are

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

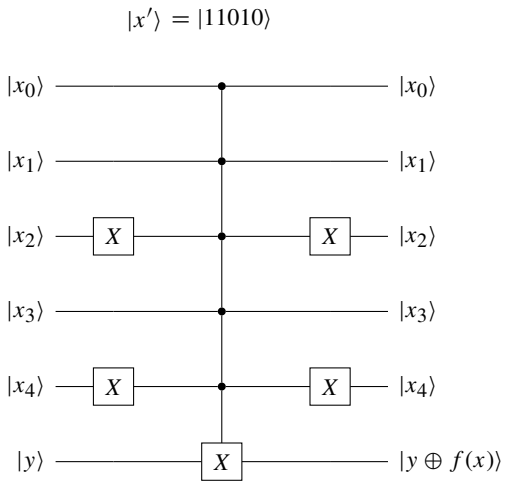


Figure 6.16 Phase inversion circuit.

$2/N - 1$. Note that this matrix is the desired end result of the derivation in the next few paragraphs. It represents this expression, as shown in the introduction of this section:

$$U_\perp = 2(|+\rangle\langle+|)^{\otimes n} - I^{\otimes n}. \tag{6.12}$$

This matrix is also called the *diffusion operator* because its form is similar to the discretized version of the diffusion equation, but we can safely ignore this fun fact here. This is the operator we hope to construct:

$$U_\perp = \begin{pmatrix} 2/N-1 & 2/N & \dots & 2/N \\ 2/N & 2/N-1 & \dots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \dots & 2/N-1 \end{pmatrix}. \tag{6.13}$$

Why do we look for this specific operator? Remember Equation 6.10. We want to construct an operator that performs this transformation.

$$\sum_x c_x|x\rangle \rightarrow \sum_x (2\mu - c_x)|x\rangle.$$

Why does this work? Each row multiplies and adds up each state vector element by $2/N$ before subtracting the one element corresponding to the diagonal. This is the exact definition of the closed form inversion procedure shown in Equation (6.12) above.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

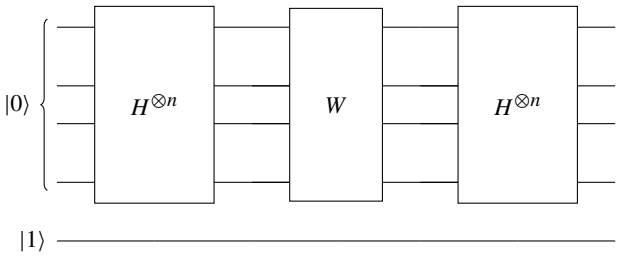


Figure 6.17 Inversion about the mean circuit.

$$\begin{pmatrix} 2/N-1 & 2/N & \dots & 2/N \\ 2/N & 2/N-1 & \dots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \dots & 2/N-1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix}$$
$$= \begin{pmatrix} (2c_0/N + 2c_1/N + \cdots + 2c_{n-1}/N) - c_0 \\ (2c_0/N + 2c_1/N + \cdots + 2c_{n-1}/N) - c_1 \\ \vdots \\ (2c_0/N + 2c_1/N + \cdots + 2c_{n-1}/N) - c_{n-1} \end{pmatrix}$$
$$= \begin{pmatrix} 2\mu - c_0 \\ 2\mu - c_1 \\ \vdots \\ 2\mu - c_{n-1} \end{pmatrix}.$$

How would we arrive at this matrix from what we've learned so far? We've seen the geometrical interpretation above – we can think of inversion about the mean as a reflection around a subspace. Hence, a possible derivation consists of three steps:

1. Ideally, we would want to rotate around the space in equal superposition $|+ + \cdots +\rangle$. But it is hard to construct an operator to do this reflection in this basis. Therefore, we use Hadamard gates to get into the computational basis and construct the reflection there.
2. Coming out of the Hadamard basis, $|+ + \cdots +\rangle$ turns into $|00\ldots0\rangle$. It seems obvious to reflect about $|00\ldots0\rangle$. We could pick another state for reflection, as long as that state is still almost orthogonal to subspace α, but for state $|00\ldots0\rangle$, the inversion operator has an elegant construction (which we show in Section 6.7.10).
3. Transform the basis back to the X-basis with Hadamard gates.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

These three steps define the circuit shown in Figure 6.17. For steps 1 and 3, it is sufficient to apply the Hadamard operators, as we are in the Hadamard basis from the phase inversion before.

For step 2, we will want to leave the state $|00\ldots0\rangle$ alone but reflect all other states. If we think about how states are represented in binary and how matrix-vector multiplication works, we can achieve this by constructing the matrix W, which is easy to derive:

$$W = \begin{pmatrix} 1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \end{pmatrix}$$

$$= 2(P_{|0\rangle})^{\otimes n} - I^{\otimes n} = \begin{pmatrix} 2 & & & \\ & 0 & & \\ & & \ddots & \\ & & & 0 \end{pmatrix} - \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix}.$$

Again, we could pick any state as the axis to reflect about, but the math is elegant and simple when picking the state $|00\ldots0\rangle$. This will become more clear with the derivation immediately below.

Only the first bit in the state vector remains unmodified, and that first bit corresponds to the state $|00\ldots0\rangle$. Remember that the state vector for this state is all 0s, except the very first element, which is a 1. All other states are therefore being negated. In combination, we want to compute the following:

$$H^{\otimes n} W H^{\otimes n} = H^{\otimes n} \begin{pmatrix} 1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \end{pmatrix} H^{\otimes n}$$

$$= H^{\otimes n} \left[\begin{pmatrix} 2 & & & \\ & 0 & & \\ & & \ddots & \\ & & & 0 \end{pmatrix} - I \right] H^{\otimes n}$$

$$= H^{\otimes n} \begin{pmatrix} 2 & & & \\ & 0 & & \\ & & \ddots & \\ & & & 0 \end{pmatrix} H^{\otimes n} - H^{\otimes n} I H^{\otimes n}.$$

Since the Hadamard is its own inverse, the second term reduces to just the identity matrix I. Multiplying in the left and right Hadamard gates:

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

$$= \begin{pmatrix} 2/\sqrt{N} & 0 & \dots & 0 \\ 2/\sqrt{N} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 2/\sqrt{N} & 0 & \dots & 0 \end{pmatrix} H^{\otimes n} - I$$

$$= \begin{pmatrix} 2/N & 2/N & \dots & 2/N \\ 2/N & 2/N & \dots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \dots & 2/N \end{pmatrix} - I.$$

Finally, subtracting the identity I yields a matrix where all elements are 2/N, except the diagonal elements, which are $2/N - 1$:

$$U_\perp = \begin{pmatrix} 2/N - 1 & 2/N & \dots & 2/N \\ 2/N & 2/N - 1 & \dots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 2/N & 2/N & \dots & 2/N - 1 \end{pmatrix}. \tag{6.14}$$

This is the matrix $U_\perp$ we were looking for. Applying this matrix to a state turns each element c_x into $2\mu - c_x$, which is exactly what we wanted from the inversion about the mean procedure, as shown in Equation (6.9)!

6.7.10 Inversion About the Mean Operator

As a third implementation strategy, we can construct a quantum circuit for the inversion about the mean using similar reasoning as for the phase inversion operator (Mermin, 2007).

The main "trick" for constructing an operator for mean inversion is to realize that the direction of the rotation for amplitude amplification does not matter, it can be negative or positive. This means that instead of constructing $W = 2(P_{|0\rangle})^{\otimes n} - I^{\otimes n}$ as before, we construct:

$$W' = I^{\otimes n} - 2(P_{|0\rangle})^{\otimes n} = I^{\otimes n} - 2|00\dots0\rangle\langle00\dots0|.$$

Not to foreshadow the implementation below, but we can verify this by changing this line in `src/grover.py`:

```
<<
  reflection = op_zero * 2.0 - ops.Identity(nbits)
>>
  reflection = ops.Identity(nbits) - op_zero * 2.0
```

We want to build a gate that leaves every state untouched, except $|00\dots0\rangle$, which should get its phase negated. A Z-gate will do this for us. Because the Z-gate must be

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

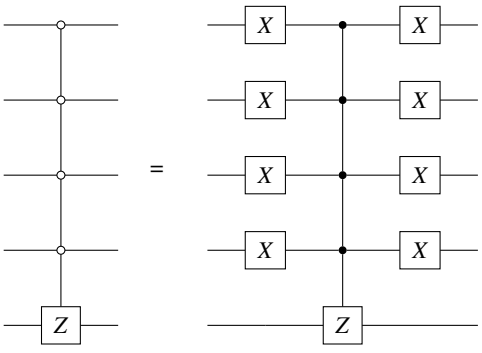


Figure 6.18 Inversion about the mean circuit (omitting leading and trailing Hadamard gates applied to *all* qubits).

controlled to only apply to $|00\ldots0\rangle$, we expect all inputs to be $|0\rangle$. Hence, to control the Z-gate, we sandwich it between X-gates (omitting the left and right Hadamard gates from the construction in Equation (6.13)), as shown in Figure 6.18.

As a result, for the big inversion operator $U_\perp$ from Equation (6.13), the circuit in Figure 6.18 corresponds to the closed form below, which yields $-U_\perp$:

$$H^{\otimes n} X^{\otimes n} (CZ)^{n-1} X^{\otimes n} H^{\otimes n} = -U_\perp.$$

6.7.11 Implementation of Grover's Algorithm

Now let's put all the pieces together (the source code can be found in file `src/grover.py`). The full Grover iteration circuit is shown in Figure 6.19. In code, we first define the function f that we intend to analyze. The `make_f` function creates an array of all 0s, except for one special element set to 1, which corresponds to $|x'\rangle$. The function also creates a function object `func` to convert its parameter, a sequence of address bits, to a decimal index and return the array value at that index. Finally, the function is returned as a callable function object:

```
def make_f(d: int = 3):
  """Construct function that will return 1 for only one bitstring."""

  num_inputs = 2**d
  answers = np.zeros(num_inputs, dtype=np.int32)
  answer_true = np.random.randint(0, num_inputs)

  bit_string = format(answer_true, '0{}b'.format(d))
  answers[answer_true] = 1

  def func(*bits):
    return answers[helper.bits2val(*bits)]

  return func
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

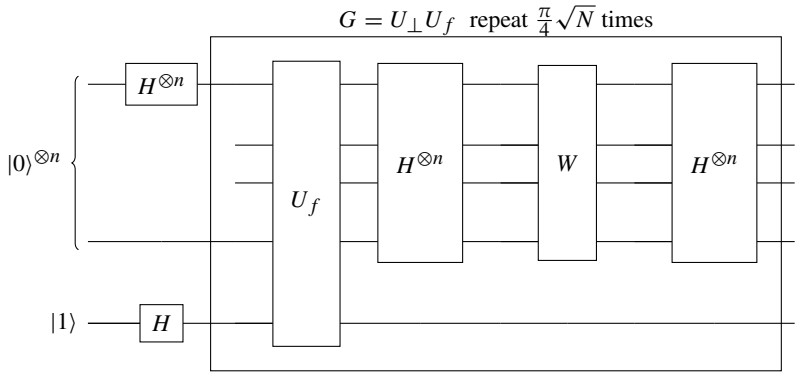


Figure 6.19 Full circuit for Grover iteration.

The circuit's initial state is a register of $|0\rangle$ qubits with an additional ancilla qubit in state $|1\rangle$. Applying the Hadamard gate to all of the qubits puts the ancilla into the state $|-\rangle$:

```
# State initialization:
psi = state.zeros(nbits) * state.ones(1)
for i in range(nbits + 1):
  psi.apply(ops.Hadamard(), i)
```

In order to implement phase inversion, we generate an oracle with the function object we created above. To create the oracle itself, we use our trusty `OracleUf` operator and pass it the function object. Note that using an oracle this way is quite slow, as it utilizes the full matrix implementation. Of course, any given operator can be implemented with quantum gates, but this can become cumbersome. Fortunately for us, this is not the case here, as we showed for the elegant phase inversion operator in Figure 6.18.

```
# Make f and uf. Note:
# We reserve space for an ancilla 'y', which is unused in
# Grover's algorithm. This allows reuse of the Deutsch Uf builder.
#
# We use the Oracle construction for convenience. It is rather
# slow (full matrix) for larger qubit counts. One can construct
# a 'regular' function for the Grover search algorithm, but this
# function is different for each bitstring and that quickly gets
# confusing.
#
f = make_f(nbits)
uf = ops.OracleUf(nbits+1, f)
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Now on to mean inversion. We first construct an all-0 matrix with a single 1.0 at element (0, 0). This is equivalent to building up an `nbits`-dimensional $|0\rangle\langle 0|$ projector:

```
# A projector of all |00...0><0...00| is an all-0 matrix
# with just element (0, 0) set to 1:
#
zero_projector = np.zeros((2**nbits, 2**nbits))
zero_projector[0, 0] = 1
op_zero = ops.Operator(zero_projector)
```

With this, we construct the $2|00\ldots0\rangle\langle 00\ldots0| - I^{\otimes n}$ reflection matrix:

```
reflection = op_zero * 2.0 - ops.Identity(nbits)
```

The full inversion operator $U_{\perp}$ consists of the Hadamard gates bracketing the reflection matrix W. We add an identity gate to account for the ancilla we added earlier for the phase inversion oracle. Finally, we build the full Grover operator $G =$ `grover` as the combination of mean inversion `inversion` with the phase inversion operator `uf`:

```
# Build Grover operator, note Identity() for the ancilla.
# The Grover operator is the combination of:
#    - phase inversion via the uf unitary operator
#    - inversion about the mean (see matrix above)
#
hn = ops.Hadamard(nbits)
reflection = op_zero * 2.0 - ops.Identity(nbits)
inversion = hn(reflection(hn)) * ops.Identity()
grover = inversion(uf)
```

We finally iterate the desired number of times based on the size of the state as discussed above (see Equation (6.10)):

```
iterations = int(math.pi / 4 * math.sqrt(2**nbits))

for _ in range(iterations):
  psi = grover(psi)
```

To check whether we have computed the right result, we perform measurement by peek-a-boo and compare the state with the highest probability to the desired output:

```
# Measurement - pick element with highest probability.
#
# Note: We constructed the oracle with n+1 qubits, to allow
# for the 'XOR-ancilla'. To check the result, we need to
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
  # ignore this ancilla.
  #
  maxbits, maxprob = psi.maxprob()
  result = f(maxbits[:-1])
  print('Got f({}) = {}, want: 1, #: {:2d}, p: {:6.4f}'
        .format(maxbits[:-1], result, solutions, maxprob))
  if result != 1:
    raise AssertionError('Something went wrong, invalid state.')
```

Experimenting with a few bit-widths:

```
def main(argv):
  [...]

  for nbits in range(3, 8):
    run_experiment(nbits)
```

Should produce results like these:

```
Got f((1, 0, 1)) = 1, want: 1, #:  1, p: 0.3906
Got f((1, 0, 1, 1)) = 1, want: 1, #:  1, p: 0.4542
Got f((1, 0, 1, 0, 0)) = 1, want: 1, #:  1, p: 0.4485
Got f((1, 0, 0, 1, 1, 1)) = 1, want: 1, #:  1, p: 0.4818
Got f((0, 1, 0, 1, 0, 0, 0)) = 1, want: 1, #:  1, p: 0.4710
```

6.8 Amplitude Amplification

How should we modify Grover's algorithm to account for multiple solutions? We have to adjust the phase inversion, inversion about the mean, and the iteration count.

Phase inversion for multiple solutions is easy to achieve. We modify the function `make_f` and give it a parameter `solutions` to indicate how many solutions to mark. We also thread this parameter through the code (not shown here, but available in the open-source repository):

```
def make_f(d: int = 3, solutions: int = 1):
  """Construct function that will return 1 for 'solutions' bits."""

  num_inputs = 2**d
  answers = np.zeros(num_inputs, dtype=np.int32)
  for i in range(solutions):
    idx = random.randint(0, num_inputs - 1)

    # Avoid collisions.
    while answers[idx] == 1:
      idx = random.randint(0, num_inputs - 1)
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
    # Found proper index. Populate 'answers' array.
    answers[idx] = 1

  # The actual function just returns an array elements.
  # pylint: disable=no-value-for-parameter
  def func(*bits):
    return answers[helper.bits2val(*bits)]

  # Return the function we just made.
  return func
```

We already derived the proper iteration count in the derivation for Grover's algorithm in Equation (6.11) as:

$$k = \frac{\pi}{4}\sqrt{\frac{N}{M}}.$$

We assumed $M = 1$ there (Section 6.7.6). To account for multiple solutions, we have to adjust the computation of the iteration count and divide by M, which is parameter `solutions` in the code:

```
iterations = int(math.pi / 4 * math.sqrt(2**nbits / solutions))
```

We add a test sequence to our main driver code to check whether any solution can be found, and with what maximal probability. For good performance, we fix the number of qubits at eight and gradually increase the number of solutions from 1 to 32:

```
  for solutions in range(1, 33):
    run_experiment(8, solutions)
```

If we print the number of states with nonzero probability, we find that all of their probabilities are identical, and there are twice as many states with nonzero probability as there are solutions! This is an artifact of our oracle construction and the entanglement with the ancilla qubit. We should get output like the following:

```
Got f((1, 1, 0, 0, 1, 0, 0, 0)) = 1, want: 1, solutions:  1, found 1
↪  with P: 0.4913
Got f((1, 0, 1, 1, 1, 1, 0, 1)) = 1, want: 1, solutions:  2, found 1
↪  with P: 0.2355
Got f((1, 1, 1, 0, 0, 1, 1, 0)) = 1, want: 1, solutions:  3, found 1
↪  with P: 0.1624
Got f((0, 0, 1, 0, 0, 0, 1, 0)) = 1, want: 1, solutions:  4, found 1
↪  with P: 0.1204
Got f((0, 0, 1, 1, 1, 1, 1, 1)) = 1, want: 1, solutions:  5, found 1
↪  with P: 0.0908
Got f((1, 1, 1, 0, 1, 1, 1, 1)) = 1, want: 1, solutions:  6, found 1
↪  with P: 0.0804
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

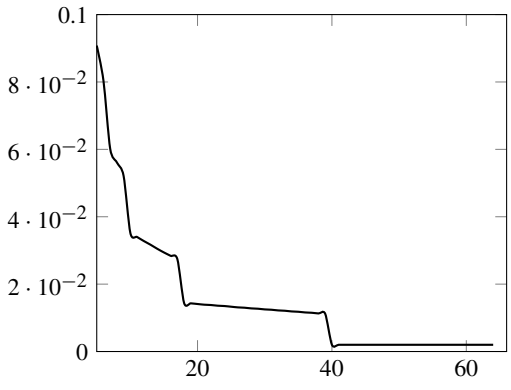


Figure 6.20 Probability of finding a solution when the total number of solutions ranges from 5 up to 64 in a state space of 128 elements.

Note how the probabilities decline rapidly. Let's visualize this with the graph in Figure 6.20. On the x-axis, we have the number of solutions ranging from 5 to 64. On the y-axis, we ignore the first few cases with high probability and set a maximum of 0.1. We can see how the probabilities decline rapidly and drop to 0 after the total number of solutions exceeds 40.

What if there are many more solutions, perhaps even a majority of the state space? To answer this question, Grover's algorithm has been generalized by Brassard et al. (2002) as *Quantum Amplitude Amplification* (QAA).

Grover expected just one special element and initialized the search with an equal superposition of all inputs by applying the algorithm $A = H^{\otimes n}$ to the input (note the unusual use of the term algorithm here). However, we might already have prior knowledge about the state of the system, which we can exploit by preparing the state differently. QAA supports any algorithm A to initialize the input and changes the Grover iteration to a more general form:

$$Q = AWA^{-1}U_f.$$

Operator U_f is the phase inversion operator for multiple solutions, and W is the inversion about the mean matrix we saw in Grover's algorithm. What changes is the derivation of the iteration count k, which has been shown to be proportional to the probability p_{good} of finding a solution (see Kaye et al., 2007, section 8.2), which was M/N (with $M = 1$ in the case of Grover):

$$k = \sqrt{\frac{1}{p_{\text{good}}}}.$$

Let us see how the probabilities improve with the new and improved iteration count. As an experiment,[7] we keep $A = H^{\otimes n}$ and compute the new iteration count as the

[7] Not in open-source, but can be obtained easily by modifying file `grover.py`.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

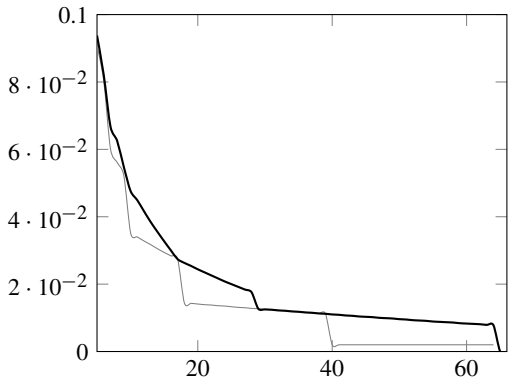


Figure 6.21 Probabilities for amplitude amplification finding 1 out of up to 64 solutions in a state space with 128 elements: (thick black line) amplitude amplification, (light gray line) and Grover's search

following, where we now divide by `solutions` to reflect the probability of finding a solution:

```
iterations = int(math.sqrt(2**nbits / solutions))
```

Figure 6.21 shows the probabilities for the two iterations, where the thick line represents the probabilities obtained with the new iteration count. We see that the situation improves markedly, but the probabilities still drop to 0 at more than 64 solutions. We have twice as many states with nonzero probabilities as there are solutions because of the ancilla entanglement. As soon as we hit half the size of the space, probabilities will drop to 0. A simple way to work around this problem is to just add another qubit. This additional qubit will double the size of the state space and eliminate the problem.

The technique of amplitude amplification requires knowledge of the number of good solutions, as well as their probability distribution. A general technique called *amplitude estimation* can help with this (see Kaye et al., 2007, section 8.2). In the next section, we detail a special case of amplitude estimation, *quantum counting*, which assumes an equal superposition of the search space with algorithm $A = H^{\otimes n}$, similar to Grover.

6.9 Quantum Counting

Quantum Counting is an interesting extension of the search problems we previously solved with Grover's algorithm and amplitude amplification. It combines these search algorithms with phase estimation in interesting ways to solve the problem of not knowing how many solutions M exist in a population of N elements. As we saw in the previous section, amplitude amplification requires knowledge of M to determine

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

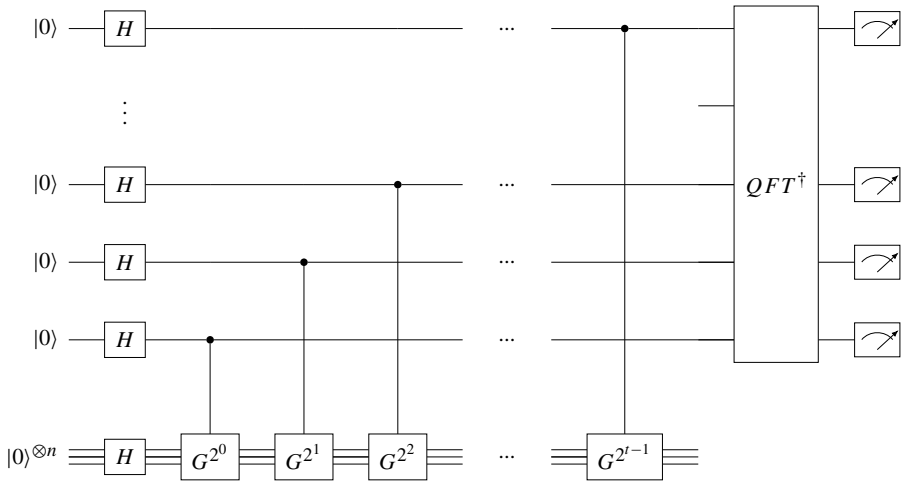


Figure 6.22 Phase estimation for the Grover operator G.

the right iteration count. Quantum counting is a special case of *amplitude estimation* that seeks to estimate this number M. Because it expects an equal superposition of the search space, similar to Grover, with algorithm $A = H^{\otimes n}$, we can reuse much of the Grover implementation in the code below.

As in Grover's algorithm, we partition the state space into a space with no solution $|\alpha\rangle$ and the space with only solutions $|\beta\rangle$:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}}|\alpha\rangle + \sqrt{\frac{M}{N}}|\beta\rangle.$$

Applying the Grover operator amounts to a rotation by an angle ϕ towards the solution space $|\beta\rangle$. You may refer back to Figure 6.15 for a graphical illustration of this process. Since this a counterclockwise rotation, we can express the Grover operator as a standard rotation matrix:

$$G(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}.$$

Rotation matrices are unitary matrices with the eigenvalues:

$$\lambda_{0,1} = e^{\pm i\phi}.$$

We have also learned in the analysis of Grover's algorithm that Equation (6.9), replicated here, holds, with N being the number of elements and M being the number of solutions:

$$\sin\left(\frac{\phi}{2}\right) = \sqrt{\frac{M}{N}}.$$

If we find ϕ, we can estimate M because we already know N. We also know that we can use our friendly neighborhood phase estimator to find ϕ. For this, we build the circuit as shown in Figure 6.22.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Let's translate this circuit into code (the implementation is in file `src/counting.py`). First we define the function that returns 1 for a solution and 0 otherwise. This is the same function we developed for amplitude amplification in Section 6.8:

```
def make_f(d: int = 3, solutions: int = 1) -> Callable:
  """Construct function that will return 1 for 'solutions' bits."""
  [...]
```

Next, we build up the phase inversion operator, just as we did in Section 6.7 on Grover's algorithm. Parameter `nbits_phase` specifies how many qubits to use for the phase estimation, and the parameter `nbits_grover` indicates how many qubits to use for the Grover operator itself. Since this code utilizes the full matrix implementation, we can only use a limited number of qubits. Nevertheless, the more qubits we use for the phase estimation, the more numerically accurate the results will become.

```
def run_experiment(nbits_phase: int, nbits_grover: int,
                   solutions: int) -> None:
  """Run full experiment for a given number of solutions."""

  # Building the Grover operator.
  # A projector of all |00...0><0...00| is an all-0 matrix
  # with just element (1, 1) set to 1:
  #
  n = 2**nbits_grover
  zero_projector = np.zeros((n, n))
  zero_projector[0, 0] = 1
  op_zero = ops.Operator(zero_projector)

  # Construct function (with f(x*) = 1) and corresponding oracle:
  #
  f = make_f(nbits_grover, solutions)
  u = ops.OracleUf(nbits_grover + 1, f)
```

We build up the circuit as in Figure 6.22. Note that the Grover operator needs a $|1\rangle$ ancilla, which we also have to add to the state (not shown in the Figure). We apply a Hadamard to the inputs, including the ancilla:

```
  # The state for the counting algorithm.
  # We reserve nbits_phase for the phase estimation.
  # We also reserve nbits_grover for the Oracle.
  # These numbers could be adjusted to achieve better
  # accuracy.
  #
  # We also add the |1> for the Oracle.
  #
  psi = (state.zeros(nbits_phase) * state.zeros(nbits_grover)
         * state.ones(1))
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
# Apply Hadamard to all the qubits.
for i in range(nbits_phase + nbits_grover + 1):
  psi.apply(ops.Hadamard(), i)
```

Let us construct the Grover operator next. This, again, is very similar to the previous section on Grover's algorithm:

```
# Construct the Grover operator.
reflection = op_zero * 2.0 - ops.Identity(nbits_grover)
hn = ops.Hadamard(nbits_grover)
inversion = hn(reflection(hn)) * ops.Identity()
grover = inversion(u)
```

We follow this with the sequence of exponentiated gates and a final inverse QFT:

```
# Now that we have the Grover operator, we have to perform
# phase estimation. This loop is a copy from phase_estimation.py
# with more comments there.
#
for idx, inv in enumerate(range(nbits_phase - 1, -1, -1)):
  u2 = grover
  for _ in range(idx):
    u2 = u2(u2)
  psi = ops.ControlledU(inv, nbits_phase, u2)(psi, inv)

# Reverse QFT gives us the phase as a fraction of 2*pi.
psi = ops.Qft(nbits_phase).adjoint()(psi)
```

This completes the circuit. We measure and find the state with the highest probability. We reconstruct the phase from the binary fractions and then use Equation (6.9) to estimate M:

```
# Get the state with highest probability and compute the phase
# as a binary fraction. Note that the probability decreases
# as M, the number of solutions, gets closer and closer to N,
# the total number of states.
maxbits, maxprob = psi.maxprob()
phi_estimate = (sum(maxbits[i] * 2**(-i - 1)
                    for i in range(nbits_phase)))

# We know that after phase estimation, this holds:
#
#    sin(phi/2) = sqrt(M/N)
#             M = N * sin(phi/2)^2
#
# Hence we can compute M. We keep the result to 2 digit to visualize
# the errors. Note that the phi_estimate is a fraction of 2*pi, hence
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
  # the 1/2 in above formula cancels out against the 2 and we compute:
  M = round(n * math.sin(phi_estimate * math.pi)**2, 2)

  print('Estimate: {:.4f} prob: {:5.2f}% --> M: {:5.2f}, want: {:2d}'
        .format(phi_estimate, maxprob * 100.0, M, solutions))
```

Let's run some experiments with seven qubits for the phase estimation, and five qubits for the Grover operator. In each experiment, we increase M by 1. For $N = 64$, we let M range from 1 to 10:

```
def main(argv):
  [...]
  for solutions in range(1, 11):
    run_experiment(7, 5, solutions)
```

Running this code should produce output like the following. We can see that our estimates are "in the ballpark" and will round to the correct solution. The solution probability also decreases significantly with higher values for M. It is instructional to experiment with all these parameters.

```
Estimate: 0.0547 prob: 10.05% --> M:  0.94, want:  1
Estimate: 0.0781 prob:  4.56% --> M:  1.89, want:  2
Estimate: 0.8984 prob:  2.85% --> M:  3.15, want:  3
Estimate: 0.8828 prob:  2.36% --> M:  4.14, want:  4
[...]
Estimate: 0.8203 prob:  1.16% --> M:  9.16, want:  9
Estimate: 0.1875 prob:  1.13% --> M:  9.88, want: 10
```

6.10 Quantum Random Walk

A *classical random walk* describes a process of random movement about a given topology, such as randomly moving left or right on a number line, left/right and up/down on a 2-dimensional grid, or moving along the edges of a graph. Random walks appear to accurately model an extensive range of real-world phenomena across disciplines as varied as physics, chemistry, economics, and sociology. In computer science, random walks are effectively used in randomized algorithms; for example, to determine the connectivity of vertices in a graph. Some of these algorithms have lower computational complexity than previously known, nonrandomized algorithms.

Random walks have fascinating properties. For example, assume two random walkers are starting their journey on a 2-dimensional grid. Will the walkers meet again in the future, and if so, how often? The answers are yes and infinitely often.

A *quantum random walk* is the quantum equivalent of a classical random walk (Kempe, 2003). Certain problems, such as the glued tree algorithm developed by

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

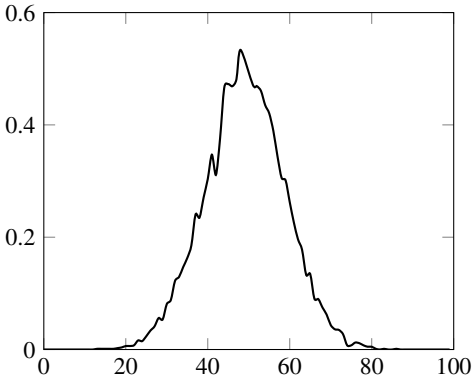


Figure 6.23 Results from a simulated classical random walk, plotting the likelihood of final position after starting in the middle of the range.

Childs et al. (2003, 2009), cannot be tractably computed on a classical machine. Herein lies the great interest in quantum random walks: some of these intractable problems become tractable on a quantum machine. In this section, we touch on basic principles, such as how probabilities propagate through a topology.

6.10.1 1D Walk

Let us start by considering a classical 1-dimensional walk on a number line. For each step, a coin toss determines whether to move left or right. After a number of moves, the probability distribution of the final location will be shaped like a classic bell curve, with the highest probability clustering around the origin of the journey. Figure 6.23 shows the result from a simple experiment,[8] which is available in the open-source repository in `tools/random_walk.py`.

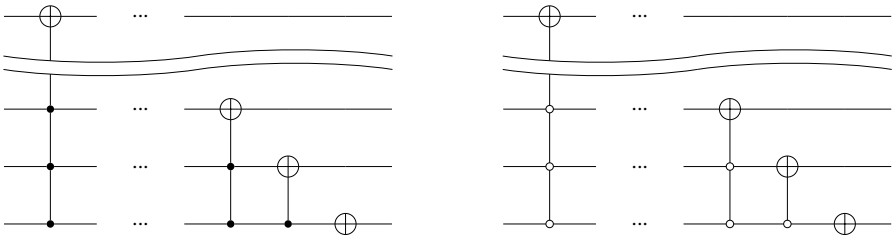
The equivalent quantum walk operates in a similar fashion with coin tosses and movements. Because this is quantum, we exploit superposition and move in both directions at the same time. In short, a quantum random walk is the repeated application of an operator $U = CM$, with C being a coin toss and M being the move operator.

The most straightforward coin toss operator we can think of is, of course, a single Hadamard gate. In this context, the coin is called a *Hadamard coin*. The $|0\rangle$ part of the resulting superposition will control a movement to the left, and the $|1\rangle$ part, the move to the right.

The movement circuits can be constructed the following way, as shown by Douglas and Wang (2009). A number line is of length infinity, which cannot be properly represented. We should assume a circle with N states as underlying topology for the walk. Simple up- and down-counters with overflow and underflow between N and 0

[8] In fairness, the curve simply reflects the random number distribution chosen for the experiment.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press



(a) A n-qubit increment operator (b) A n-qubit decrement operator

Figure 6.24 Increment and decrement operators for quantum walks.

will work as movement operators. We can construct an n-qubit increment circuit as shown in Figure 6.24a, with the corresponding Python code (the full implementation is in file `src/quantum_walk.py`):

```
def incr(qc, idx: int, nbits: int, aux, controller=[]):
  """Increment-by-1 circuit."""

  #  -X--
  #  -o--X--
  #  -o--o--X--
  #  -o--o--o--X--
  #  ...
  for i in range(nbits):
    ctl=controller.copy()
    for j in range(nbits-1, i, -1):
      ctl.append(j+idx)
    qc.multi_control(ctl, i+idx, aux, ops.PauliX(), 'multi-1-X')
```

The analogous n-qubit decrement circuit is easy to construct as well, as shown in Figure 6.24b, with this code:

```
def decr(qc, idx: int, nbits: int, aux, controller=[]):
  """Decrement-by-1 circuit."""

  # Similar to incr, except controlled-by-0's are being used.
  #
  #  -X--
  #  -0--X--
  #  -0--0--X--
  #  -0--0--0--X--
  #  ...
  for i in range(nbits):
    ctl=controller.copy()
    for j in range(nbits-1, i, -1):
      ctl.append([j+idx])
    qc.multi_control(ctl, i+idx, aux, ops.PauliX(), 'multi-0-X')
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

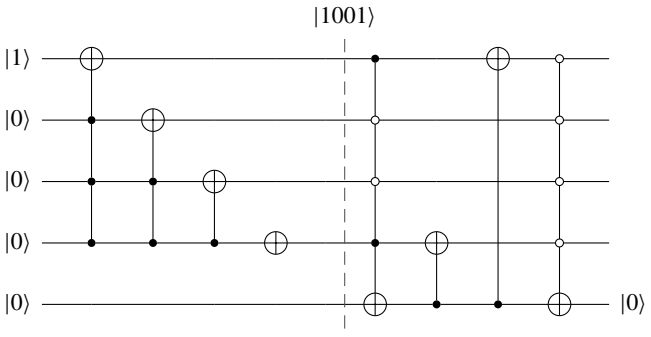


Figure 6.25 A increment modulo 9 operator.

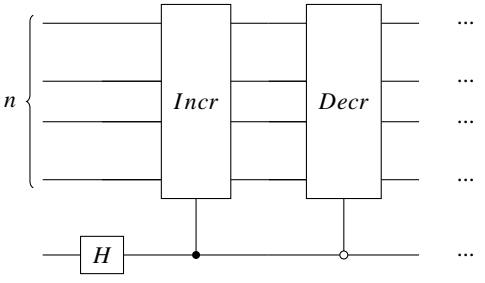


Figure 6.26 A single step for a quantum walk.

For both cases, N is a power of 2. We can construct other types of counters, for example, counters with step size larger than 1, or counters that increment modulo another number. For example, to construct a counter modulo 9, we add gates matching the binary representation of 9 to force a counter reset to 0, as shown in Figure 6.25.

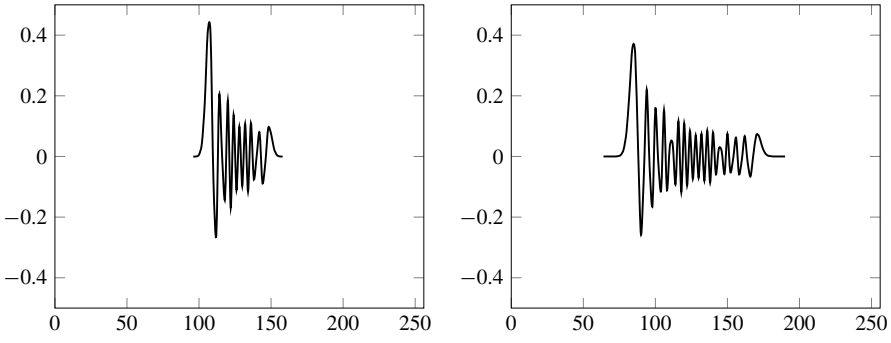
With these tools, we can construct an initial n-qubit quantum circuit *step*, as shown in Figure 6.26. It has to be applied repeatedly to simulate a walk (consisting of more than just a single step).

We can see how to generalize this pattern to other topologies. For example, for a 2D walk across a grid, we can use two Hadamard coins: one for the left or right movement and one for movements up or down. For graph traversals, we would encode a graph's connectivity as an unitary operator. Several other examples can be found in Douglas and Wang (2009).

6.10.2 Walk the Walk

To simulate a given number of steps, we use the following driver code. We initialize the `x` register *in the middle* of the state's number range for n qubits, as binary $0b100\ldots0$. This way, we avoid immediate underflow below zero, and visualizations appear centered. Note how the increment operator is controlled by `coin[0]`, while the

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press



(a) 8 qubits, 32 steps, starting at 0x80 (b) 8 qubits, 64 steps, starting at 0x80

Figure 6.27 Propagating amplitudes after 32 and 64 steps.

decrement operator is controlled by `[coin[0]]`. The former is a standard controlled-by-1 gate, while the latter is a controlled-by-0 gate, as outlined in Section 4.3.7 on multi-controlled gates.

```
def simple_walk():
  """Simple quantum walk."""

  nbits = 8
  qc = circuit.qc('simple_walk')
  qc.reg(nbits, 0x80)
  aux = qc.reg(nbits, 0)
  coin = qc.reg(1, 0)  # Add single coin qubit

  for _ in range(64):
    qc.h(coin[0])
    incr(qc, 0, nbits, aux, [coin[0]])    # ctrl-by-1
    decr(qc, 0, nbits, aux, [[coin[0]]])  # ctrl-by-0
```

What is really happening here? With n qubits, we can represent 2^n states with the corresponding number of probability amplitudes. As we perform step after step, nonzero amplitudes will start to *propagate out* over the state space. Looking at the examples in Figures 6.27b and 6.28b we see that, in contrast to a classical quantum walk, the amplitude distribution spreads out faster and with a very different shape. A series of 32 steps produces a nonzero amplitude in 64 states; the walk progresses in both directions at the same time. The farther away from the origin, the larger the amplitudes become. These are the key properties that quantum algorithms exploit to solve classically intractable problems, such as Childs' welded tree algorithm (Childs et al., 2003). To visualize the effect, we graph the resulting amplitudes:

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

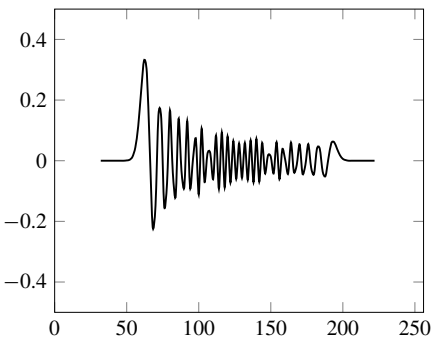
```
for bits in helper.bitprod(nbits):
  idx_bits = bits
  for i in range(nbits+1):
      idx_bits = idx_bits + (0,)
  if qc.psi.ampl(*idx_bits) != 0.0:
    print('{:5.3f}'.format(qc.psi.ampl(*idx_bits0).real))
```

Let us experiment with eight qubits. The starting position should be in the middle of the range of states. With eight qubits, there are 256 possible states, and we initialize with `0x80`, the middle of the range. It is possible to initialize with 0 of course, but that would lead to immediate wraparound effects. The amplitudes after 32, 64, and 96 steps are shown in Figure 6.27a, Figure 6.27b, and Figure 6.28a. The x-axis shows the state space (256 unique states for eight qubits). The y-axis shows each state's amplitude.

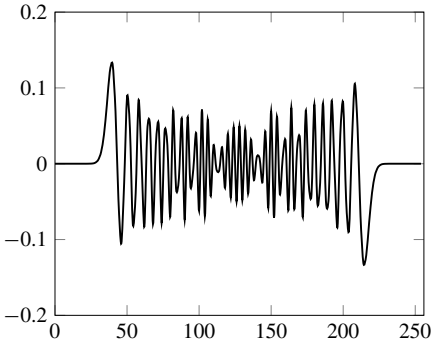
Notice how in the figures the amplitudes progress in a biased fashion. It is possible to create coin operators that are biased to the other side, or even balanced coin operators. Alternatively, we can start in a state different from $|0\rangle$. In the example in Figure 6.28b, we simply initialize the coin state as $|1\rangle$.

There are countless more experiments that you can perform with different coin operators, starting points, initial states, number of qubits, iteration counts, and more complex topologies beyond simple 1D and 2D walks.

It is exciting that if we can express a particular algorithmic reachability problem as a quantum walk circuit, the fast speed of quantum walks and the dense storage of states can lead to quantum algorithms with lower complexity than their corresponding classical algorithms. As an example, the 2010 IARPA program announcement set a challenge of eight complex algorithms to drive scalable quantum software and infrastructure development (IARPA, 2010). Three of these algorithms utilized quantum



(a) 8 qubits, 96 steps, starting at 0x80, with initial state $|0\rangle$



(b) 8 qubits, 96 steps, starting at 0x80, with initial state $|1\rangle$

Figure 6.28 Propagating amplitudes with different initial states.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

walks: the triangle finding algorithm (Buhrman et al., 2005; Magniez et al., 2005), the Boolean formula algorithm (Childs et al., 2009), and the welded tree algorithm (Childs et al., 2003).

6.11 Variational Quantum Eigensolver

This section represents a brief foray into the area of quantum simulation. We discuss the *variational quantum eigensolver* (VQE), an algorithm to estimate the ground state energy of a Hamiltonian.

It is possible to use quantum phase estimation (QPE) for this purpose. However, for realistic Hamiltonians, the number of gates required can reach millions, even billions, making it challenging to keep a physical quantum machine coherent long enough to run the computation. VQE, on the other hand, is a hybrid classical/quantum algorithm. The quantum part requires fewer gates and, therefore, much shorter coherence times when compared to QPE. This is why it created great interest in today's era of Noisy Intermediate Scale Quantum Computers (NISQ), which have limited resources and short coherence times (Preskill, 2018).

There cannot be a book about quantum computing without mentioning the Schrödinger equation at least once. This is that section in this book. So we begin by marveling at the beauty of the equation, although we will not solve it here. The purpose of showing it is to derive the composition of Hamiltonians from eigenvectors and how the variational principle enables the approximation of a minimum eigenvalue. This is followed by a discussion of measurement in different bases. We explain the variational principle next before we detail the hybrid classical/quantum algorithm itself.

6.11.1 System Evolution

In Section 2.15 we described the evolution of a quantum system in postulate 2 as $|\psi\rangle' = U|\psi\rangle$. This is what we have used in this text so far – to change a state we applied a unitary operator to it – and this *discrete time* evolution of a system is sufficient for all the algorithms discussed in previous sections. However, it is a simplification, as time does not move in discrete steps (as far as we know, or perhaps *suspect*).

The following few paragraphs derive a specific form of the time-independent Schrödinger equation. The details are not overly important in the context of this text, and we focus primarily on the final form because that is where the VQE will come into play.

The *time dependent* evolution of a system Ψ is described with the beautiful Schrödinger equation. We show the one-dimensional version only. Again, let's just marvel at this differential equation. We don't have to solve it here:

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m} \frac{\partial^2 \Psi}{\partial x^2} + V\Psi. \tag{6.15}$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

This equation can be transformed into a *time-independent* form:

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V\psi = E\psi. \tag{6.16}$$

In classical mechanics, the total energy of a system, which is the kinetic energy plus the potential V, is called the *Hamiltonian*, denoted as $\mathcal{H}$, not to be confused with our Hadamard operator H.

$$\begin{aligned}\mathcal{H}(x,p) &= \frac{mv^2}{2} + V(x)\\ &= \frac{(mv)^2}{2m} + V(x)\\ &= \frac{p^2}{2m} + V(x).\end{aligned}$$

As a side note, the factor $\hbar$ (the Planck constant) appears here from the famous Heisenberg uncertainty principle for a particle's position x and momentum p_x, with $\Delta x \Delta p_x \geq \hbar/2$. A *Hamiltonian operator* is obtained by the standard substitution with the momentum operator:

$$\begin{aligned}p &\rightarrow -i\hbar\frac{\partial}{\partial x},\\ \hat{\mathcal{H}} &= -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x).\end{aligned}$$

We use this result to rewrite Equation (6.16) as the following, with $\hat{\mathcal{H}}$ being the operator and E being an energy eigenvalue. Note the parallel to the definition of eigenvectors as $A\vec{x} = \lambda\vec{x}$:

$$\hat{\mathcal{H}}\psi = E\psi.$$

The expectation value for the total energy is then:

$$\langle\hat{\mathcal{H}}\rangle = E.$$

This Hamiltonian operator is Hermitian – on measurement, we obtain real values. Hence the eigenvalues must be real. It has a complete set of orthonormal eigenvectors:

$$|E_0\rangle, |E_1\rangle, \ldots, |E_{n-1}\rangle,$$

with corresponding real eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$. We can describe states as linear combinations of the eigenvectors

$$|\psi\rangle = c_0|E_0\rangle + c_1|E_1\rangle + \cdots + c_{n-1}|E_{n-1}\rangle. \tag{6.17}$$

This is the result we were looking for. It is important to note that the c_i are *complex* coefficients, similar to how we described a superposition between $|0\rangle$ and $|1\rangle$. In this case, however, the basis vectors are E_i. For a detailed derivation of the above, see for example Fleisch (2020).

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

6.11.2 The Variational Principle

Assume we are looking for the *ground state energy* E_0 of a system described by a given Hamiltonian. Knowing the ground state energy is important in many fields. For example, in thermodynamics, it describes behavior at temperatures close to absolute zero. In chemistry, it allows drawing conclusions about electron energy levels.

Let us now also assume that we cannot solve the time-independent Schrödinger Equation (6.16). We know that measurement will project the state onto an eigenvector, and the measurement result will be the corresponding eigenvalue. The *variational principle* will give an *upper bound* for E_0 with an expectation value for $\hat{\mathcal{H}}$ as:

$$E_0 \leq \langle\psi|\hat{\mathcal{H}}|\psi\rangle \equiv \langle\hat{\mathcal{H}}\rangle.$$

However, what is this state $|\psi\rangle$? The answer is, *any* state, as long as the state is capable, or close to being capable, of producing an eigenvector for $\hat{\mathcal{H}}$. This state will determine the remaining error for the estimation of E_0. We have to be smart about how to construct it. This is the key idea of the VQE algorithm.

To see how this principle works, let's take our assumed state from above and further assume that λ_0 is the smallest eigenvalue:

$$|\psi\rangle = c_0|E_0\rangle + c_1|E_1\rangle + \cdots + c_{n-1}|E_{n-1}\rangle.$$

Computing $\langle\psi|\hat{\mathcal{H}}|\psi\rangle$, as follows, demonstrates that *any* computed expectation value will be greater or equal to the minimum eigenvalue:

$$\begin{aligned}(c_0^*\langle E_0| + c_1^*\langle E_1| + \cdots + c_{n-1}^*\langle E_{n-1}|)\ \hat{\mathcal{H}}\ (c_0|E_0\rangle + c_1|E_1\rangle + \cdots + c_{n-1}|E_{n-1}\rangle)\\ = |c_0|^2\lambda_0 + |c_1|^2\lambda_1 + \cdots + |c_{n-1}|^2\lambda_{n-1}\\ \geq \lambda_0.\end{aligned}$$

The VQE algorithm works with Hamiltonians that can be written as a sum of a polynomial number of terms of Pauli operators and their tensor products (Peruzzo et al., 2014). This type of Hamiltonian is used in quantum chemistry, the Heisenberg Model, the quantum Ising Model, and other areas. For example, for a helium hydride ion (He-H^+) with bond distance 90 pm, the energy (Hamiltonian) is:

$$\begin{aligned}\hat{\mathcal{H}} = &-3.851II - 0.229I\sigma_x - 1.047I\sigma_z - 0.229\sigma_x I + 0.261\sigma_x\sigma_x\\ &+ 0.229\sigma_x\sigma_z - 1.0467\sigma_z I + 0.229\sigma_z\sigma_x + 0.236\sigma_z\sigma_z.\end{aligned}$$

6.11.3 Measurement in Pauli Bases

So far in this book, we have described measurement as projecting a state onto the basis states $|0\rangle$ and $|1\rangle$. If we recall the Bloch sphere representation as shown in Figure 6.29, measurement projects the state to either the north or south pole of the Bloch sphere, corresponding to a measurement along the z-axis.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

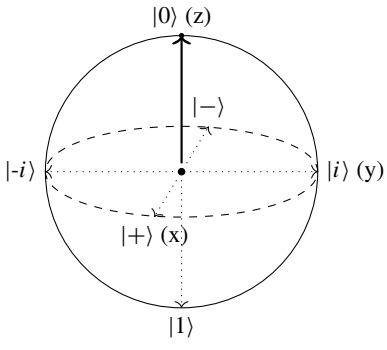


Figure 6.29 Bloch sphere representation with axes x, y, z.

However, what if the current state was aligned with a different axis, such as the x-axis from $|-\rangle$ to $|+\rangle$, or the y-axis pointing from $|\text{-}i\rangle$ to $|i\rangle$? In both cases, a measurement along the z-axis would result in a random toss between $|0\rangle$ and $|1\rangle$.

To measure in a different basis, we should *rotate* the state into the standard basis on the z-axis and perform a standard measurement there. The results can be interpreted along the original bases, and we get the added benefit that we only need a measurement apparatus in one direction.

To get a proper measurement along the x-axis, we could apply the Hadamard gate or rotate over the y-axis. Correspondingly, to get a measurement along the y-axis, we may rotate about the x-axis.

To compute expectation values for states composed of Pauli matrices, we remind ourselves of the basis states in the X, Y, and Z bases:

$$
\begin{aligned}
X: \quad & |+\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}, \qquad |-\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix}, \\
Y: \quad & |i\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\i\end{bmatrix}, \qquad |\text{-}i\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-i\end{bmatrix}, \\
Z: \quad & |0\rangle = \begin{bmatrix}1\\0\end{bmatrix}, \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}.
\end{aligned}
$$

Pauli operators have eigenvalues of -1 and $+1$. Here are the operators applied to basis states with eigenvalues $+1$:

$$
\begin{aligned}
Z|0\rangle &= |0\rangle, \\
X|+\rangle &= |+\rangle, \\
Y|i\rangle &= |i\rangle.
\end{aligned}
$$

These are the same operators applied to basis states with eigenvalues -1:

$$
Z|1\rangle = -|1\rangle,
$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

$$X|-\rangle = -|-\rangle,$$
$$Y|\text{-}i\rangle = \text{-}|\text{-}i\rangle.$$

Let us now talk about expectation values. For a state in the Z-basis with amplitudes c_0^z and c_1^z:

$$|\psi\rangle = c_0^z|0\rangle + c_1^z|1\rangle.$$

Computing the expectation value for Z, in the Z-basis, yields the following, and similar for the X- and Y-bases:

$$\begin{aligned}\langle\psi|Z|\psi\rangle &= \left(c_0^{z*}\langle 0| + c_1^{z*}\langle 1|\right) Z \left(c_0^z|0\rangle + c_1^z|1\rangle\right) \\ &= |c_0^z|^2 - |c_1^z|^2\end{aligned}$$

The values $|c_0^z|^2$ and $|c_1^z|^2$ are the measurement probabilities for $|0\rangle$ and $|1\rangle$. If we run N experiments and measure state $|0\rangle$ n_0 times and state $|1\rangle$ n_1 times:

$$|c_0^z|^2 = \frac{n_0}{N}, \quad |c_1^z|^2 = \frac{n_1}{N}.$$

Then this is the final expectation value for Z; please note the minus sign in this equation:

$$\langle Z\rangle = \frac{n_0 - n_1}{N}.$$

To give an example, let's assume we have a very simple circuit initialized with $|0\rangle$ and with just one Hadamard gate. The state after this gate will be $|+\rangle$, which is on the x-axis. If we now measure N times in the Z-basis, about 50% of the measurements will return $|0\rangle$, and 50% will return $|1\rangle$. The $|0\rangle$ corresponds to eigenvalue 1, the $|1\rangle$ corresponds to eigenvalue -1. Hence, the expectation value is 0:

$$\frac{(+1)N/2 + (-1)N/2}{N} = 0.$$

If we rotated the state into the Z-basis with another Hadamard gate, the expectation value of $|0\rangle$ in the Z-basis would now be 1.0, which corresponds to the expectation value of the state $|+\rangle$ originally in the X-basis.

In our infrastructure, we do not have to make measurements to compute probabilities because we can directly peek at the amplitudes of a state vector. To compute the expectation values for measurements made on Pauli operators with eigenvalues $+1$ and -1 corresponding to measuring $|0\rangle$ or $|1\rangle$, we add this function to our quantum circuit implementation `qc`:

```
def pauli_expectation(self, idx: int):
  """We can compute the Pauli expectation value from probabilities."""

  # Pauli eigenvalues are -1 and +1, hence we can compute the
  # expectation value like this:
  p0, _ = self.measure_bit(idx, 0, False)
  return p0 - (1 - p0)
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Let's run a few experiments to familiarize ourselves with these concepts. What happens to the eigenvectors and eigenvalues for a Hamiltonian constructed from a single Pauli matrix multiplied with a factor? Is the result still unitary, or is it Hermitian?

```
  factor = 0.6
  H = factor * ops.PauliY()
  eigvals = np.linalg.eigvalsh(H)
  print(f'Eigenvalues of {factor} X = ', eigvals)
  print(f'is_unitary: {H.is_unitary()}')
  print(f'is_hermitian: {H.is_hermitian()}')
>>
Eigenvalues of 0.6 X =  [-0.6  0.6]
is_unitary: False
is_hermitian: True
```

Eigenvalues scale with the factor. Hamiltonians are Hermitian, but not necessarily unitary. Let's create a $|0\rangle$ state, show its Bloch sphere coordinates, and compute its expectation value in the Z-basis.

```
  qc = circuit.qc('test')
  qc.reg(1, 0)
  qubit_dump_bloch(qc.psi)
  print(f'Expectation value for 0 State: {qc.pauli_expectation(0)}')
>>
x: 0.00, y: 0.00, z: 1.00
Expectation value for 0 State: 1.0
```

As expected, the current position is on top of the north pole, corresponding to state $|0\rangle$. The expectation value is 1.0; there will be no measurements of the $|1\rangle$ state. Now, if we add just a single Hadamard gate, we will get:

```
x: 1.00, y: 0.00, z: -0.00
Expectation value for |0>: -0.00
```

The position on the Bloch sphere is now on the x-axis, and the corresponding expectation value in the Z-basis is 0; we will measure an equal amount of $|0\rangle$ and $|1\rangle$ states. Of course, to rotate this state back into the Z-basis, we only have to apply another Hadamard gate.

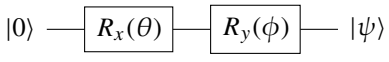
6.11.4 VQE Algorithm

The algorithm itself iterates over these steps:

1. **Ansatz**. Prepare a parameterized initial state $|\psi\rangle$. This is called the *ansatz*.
2. **Measurement**. Measure the expectation value $\langle\psi|\hat{\mathcal{H}}|\psi\rangle$.
3. **Minimize**. Tune the parameters of the ansatz to minimize the expectation value. The smallest value will be the best approximation of the minimum eigenvalue.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

This is best explained by example. Let's focus on the single-qubit case first. We know that we can reach any point on the Bloch sphere with rotations about the x-axis and y-axis. Let's use this simple parameterized circuit as the ansatz:



We will construct multiple instances of ansatzes (which has a fun rhyme to it and is the proper English plural from; the correct German plural Ansätze does not sound quite as melodic). Let's wrap it into code (which is in file `src/vqe_simple.py`):

```
def single_qubit_ansatz(theta: float, phi: float) -> circuit.qc:
  """Generate a single qubit ansatz."""

  qc = circuit.qc('single-qubit ansatz Y')
  qc.qubit(1.0)
  qc.rx(0, theta)
  qc.ry(0, phi)
  return qc
```

Let's further assume a Hamiltonian of this form:

$$H = H_0 + H_1 + H_2 = 0.2X + 0.5Y + 0.6Z.$$

We can compute the minimum eigenvalue of -0.8062 with help from `numpy`:

```
  H = 0.2 * ops.PauliX() + 0.5 * ops.PauliY() + 0.6 * ops.PauliZ()
  # Compute known minimum eigenvalue.
  eigvals = np.linalg.eigvalsh(H)
  print(eigvals)
>>
[-0.8062258  0.8062258]
```

To compute the expectation value, let's create a state $|\psi\rangle$ and compute the expectation value $\langle\psi|\hat{\mathcal{H}}|\psi\rangle$ from two angles `theta` and `phi`:[9]

```
def run_single_qubit_experiment2(theta: float, phi: float):
  """Run experiments with single qubits."""

  # Construct Hamiltonian.
  H = 0.2 * ops.PauliX() + 0.5 * ops.PauliY() + 0.6 * ops.PauliZ()

  # Compute known minimum eigenvalue.
  eigvals = np.linalg.eigvalsh(H)
```

[9] This code segment is different from the open-source version. It is for illustration only.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
  # Build the ansatz with two rotation gates.
  ansatz = single_qubit_ansatz(theta, phi)

  # Compute <psi | H | psi>. Find smallest one, which will be
  # the best approximation to the minimum eigenvalue from above.
  val = np.dot(ansatz.psi.adjoint(), H(ansatz.psi))

  # Result from computed approach:
  print('Minimum: {:.4f}, Estimated: {:.4f}, Delta: {:.4f}'.format(
      eigvals[0], np.real(val), np.real(val - eigvals[0])))
```

Let's experiment with a few different values for `theta` and `phi`:

```
  run_single_qubit_experiment2(0.1, -0.4)
  run_single_qubit_experiment2(0.8, -0.1)
  run_single_qubit_experiment2(0.9, -0.8)
>>
Minimum: -0.8062, Estimated: 0.4225, Delta: 1.2287
Minimum: -0.8062, Estimated: 0.0433, Delta: 0.8496
Minimum: -0.8062, Estimated: -0.2210, Delta: 0.5852
```

It appears we are moving in the right direction. We are getting closer to estimating the lowest eigenvalue, but we are still pretty far away. This particular ansatz is simple enough; we can incrementally iterate over both angles, approximating the minimum eigenvalue with good precision. Just picking random numbers would work as well, up to a certain degree. Obviously we can use techniques like gradient descent to find the best possible arguments more quickly (Wikipedia, 2021d). Let's run 10 experiments with random, single-qubit Hamiltonians, iterating over the angles ϕ and θ in increments of 10 degrees:

```
  [...]
  # iterate over all angles in increments of 10 degrees.
  for i in range(0, 180, 10):
    for j in range(0, 180, 10):
      theta = np.pi * i / 180.0
      phi = np.pi * j / 180.0
  [...]
  # run 10 experiments with random H's.
  [...]
>>
Minimum: -0.6898, Estimated: -0.6889, Delta: 0.0009
Minimum: -0.7378, Estimated: -0.7357, Delta: 0.0020
[...]
Minimum: -1.1555, Estimated: -1.1552, Delta: 0.0004
Minimum: -0.7750, Estimated: -0.7736, Delta: 0.0014
```

In the code above, we explicitly compute the expectation value with two dot products. The key to success here is that the ansatz is capable of creating the minimum

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

eigenvalue's eigenvector (for two qubits). Shende et al. (2004) show how to construct a universal two-qubit gate. However, the challenge is to minimize gates for much larger Hamiltonians, especially on today's smaller machines. How to construct the ansatz is a research challenge. Which specific learning technique to use to accelerate the approximation is another subject of ongoing interest in the field, even though it appears that standard techniques from the field of machine learning work well enough.

6.11.5 Measuring Eigenvalues

In a physical setting, we cannot just multiply the state with the Hamiltonian. We have to measure along the Pauli bases and reconstruct the eigenvalues from the expectation values, as explained above. As mentioned earlier, we assume we can only measure in one direction. Let us once more assume a Hamiltonian of the following form. The factors are important; we have to remember them:

$$\hat{\mathcal{H}} = 0.2X + 0.5Y + 0.6Z.$$

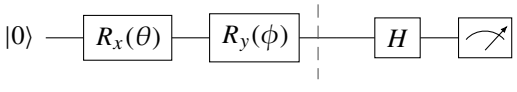
We express the expectation values in the Z-basis with help of gate equivalences. Note how we isolate the Z in the last line, representing the measurement in the Z-basis:

$$\begin{aligned}\langle\psi|\hat{\mathcal{H}}|\psi\rangle &= \langle\psi|0.2X + 0.5Y + 0.6Z|\psi\rangle \\ &= 0.2\langle\psi|X|\psi\rangle + 0.5\langle\psi|Y|\psi\rangle + 0.6\langle\psi|Z|\psi\rangle \\ &= 0.2\langle\psi|HZH|\psi\rangle + 0.5\langle\psi|HS^\dagger ZHS|\psi\rangle + 0.6\langle\psi|Z|\psi\rangle \\ &= 0.2\langle\psi H|Z|H\psi\rangle + 0.5\langle\psi HS^\dagger|Z|HS\psi\rangle + 0.6\langle\psi|Z|\psi\rangle.\end{aligned}$$

In our experimental code, we first construct random Hamiltonians:

```
a = random.random()
b = random.random()
c = random.random()
H = (a * ops.PauliX() + b * ops.PauliY() + c * ops.PauliZ())
```

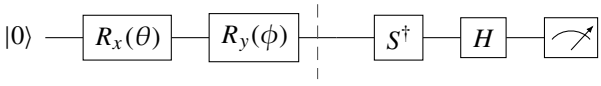
We have to build three circuits. The first is for $\langle\psi|X|\psi\rangle$, which requires an additional Hadamard gate:




```
# X-Basis
qc = single_qubit_ansatz(theta, phi)
qc.h(0)
val_a = a * qc.pauli_expectation(0)
```

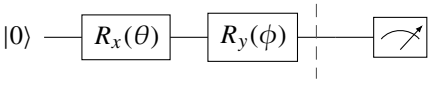
https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Then one circuit for $\langle\psi|Y|\psi\rangle$, which requires a Hadamard gate and an $S^\dagger$ gate:




```
# Y-Basis
qc = single_qubit_ansatz(theta, phi)
qc.sdag(0)
qc.h(0)
val_b = b * qc.pauli_expectation(0)
```

Finally, a circuit for the measurement in the Z-basis $\langle\psi|Z|\psi\rangle$. In this basis we can measure as is, there is no need for additional gates:




```
# Z-Basis
qc = single_qubit_ansatz(theta, phi)
val_c = c * qc.pauli_expectation(0)
```

As before, we iterate over the angles ϕ and θ in increments of, this time, 5 degrees. For each iteration, we take the expectation values `val_a`, `val_b`, and `val_c`, multiply them with the factors we noted above, add up the result, and look for the smallest value.

```
expectation = val_a + val_b + val_c
if expectation < min_val:
  min_val = expectation
```

This value `min_val` should be our estimate. The results are numerically accurate:

```
Minimum eigenvalue: -0.793, Delta: 0.000
Minimum eigenvalue: -0.986, Delta: 0.000
Minimum eigenvalue: -1.278, Delta: 0.000
Minimum eigenvalue: -0.937, Delta: 0.000
[...]
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

6.11.6 Multiple Qubits

How do we extend measurements to more than just one qubit? We begin with the simplest two-qubit Hamiltonians we can think of and extrapolate from there. Let's look at this tensor product and operator matrix:

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

We know that for diagonal matrices, the diagonal elements are eigenvalues, which are $+1$ and -1 in this case. This matrix has two subspaces, which map to these eigenvalues. On measurement, we would get a result of either $+1$ or -1.

Any unitary two-qubit transformation U on this matrix will map to a space with the same eigenvalues of $+1$ and -1. This means we can apply a similar trick as in the one-qubit case and apply the following transformations. Note that these are matrices; we have to multiply from both sides, as in:

$$U^{\dagger}(Z \otimes I)U.$$

We can change any Pauli measurement's basis into $Z \otimes I$. For example, to change the basis for $X \otimes I$ to $Z \otimes I$, we apply a Hadamard gate, just as above, with the operator $U = H \otimes I$. Let's verify this in code:

```
H = ops.Hadamard()
I = ops.Identity()
U = H * I
(ops.PauliZ() * I).dump('Z x I')
(ops.PauliX() * I).dump('X x I')
(U.adjoint() @ (ops.PauliX() * I)).dump('Udag(X x I)')
(U.adjoint() @ (ops.PauliX() * I) @ U).dump('Udag(X x I)U')
>>
Z x I (2-qubits operator)
 1.0       -         -         -
 -         1.0       -         -
 -         -        -1.0       -
 -         -         -        -1.0
X x I (2-qubits operator)
 -         -         1.0       -
 -         -         -         1.0
 1.0       -         -         -
 -         1.0       -         -
Udag(X x I) (2-qubits operator)
 0.7       -         0.7       -
 -         0.7       -         0.7
-0.7       -         0.7       -
 -        -0.7       -         0.7
Udag(X x I)U (2-qubits operator)
 1.0       -         -         -
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
-          1.0        -          -
-          -         -1.0        -
-          -          -         -1.0
```

From this, it is straightforward to construct the operators for a first set of Pauli measurements containing at least one identity operator, as shown in Table 6.1.

But now it gets complicated. The operator for $Z \otimes Z$ is the Controlled-Not $U = CX_{1,0}$! How does this happen? Let look at the matrix $Z \otimes Z$:

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

It needs a few permutations to turn into the form we are looking for, which is $Z \otimes I$. If we apply the Controlled-Not from left and right:

$$CX_{1,0}^{\dagger}(Z \otimes Z)\, CX_{1,0} = (Z \otimes I),$$

we indeed get the result we are looking for:

```
  (ops.Cnot(1, 0).adjoint() @ (ops.PauliZ() * ops.PauliZ()) @
            ops.Cnot(1, 0)).dump()
 >>
1.0        -          -          -
-          1.0        -          -
-          -         -1.0        -
-          -          -         -1.0
```

The operator matrices for $CX_{1,0}$ perform the required permutation; we should not think of this as an actually controlled operation. With these insights, we can now define the remaining 4×4 Pauli measurement operators as shown in Table 6.2.

From here, we can generalize the construction to more than two qubits, similar to Whitfield et al. (2011) for Hamiltonian simulation (which we don't cover here). All we have to do is to surround the multi-Z Hamiltonian with *cascading* Controlled-Not

Table 6.1 Operators for measurements containing an identity.

Pauli Measurement	Operator U
$Z \otimes I$	$I \otimes I$
$X \otimes I$	$H \otimes I$
$Y \otimes I$	$HS^{\dagger} \otimes I$
$I \otimes Z$	$(I \otimes I)\ SWAP$
$I \otimes X$	$(H \otimes I)\ SWAP$
$I \otimes Y$	$(HS^{\dagger} \otimes I)\ SWAP$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Table 6.2 Operators for measurements with no identity.

Pauli Measurement	Operator U
$Z \otimes Z$	$CX_{1,0}$
$X \otimes Z$	$CX_{1,0}\,(H \otimes I)$
$Y \otimes Z$	$CX_{1,0}\,(HS^\dagger \otimes I)$
$Z \otimes X$	$CX_{1,0}\,(I \otimes H)$
$X \otimes X$	$CX_{1,0}\,(H \otimes H)$
$Y \otimes X$	$CX_{1,0}\,(HS^\dagger \otimes H)$
$Z \otimes Y$	$CX_{1,0}\,(I \otimes HS^\dagger)$
$X \otimes Y$	$CX_{1,0}\,(H \otimes HS^\dagger)$
$Y \otimes Y$	$CX_{1,0}\,(HS^\dagger \otimes HS^\dagger)$

gates. For example, for the three-qubit ZZZ, we write the code below (which could be simplified by recognizing that $CX_{1,0}^\dagger = CX_{1,0}$).

```
ZII = ops.PauliZ() * ops.Identity()* ops.Identity()
C10 = ops.Cnot(1, 0) * ops.Identity()
C21 = ops.Identity() * ops.Cnot(2, 1)
C10adj = C10.adjoint()
C21adj = C21.adjoint()
ZZZ = ops.PauliZ() * ops.PauliZ() * ops.PauliZ()

res = C10adj @ C21adj @ ZZZ @ C21 @ C10
self.assertTrue(res.is_close(ZII))
```

Note that the adjoint of the X-gate is identical to the X-gate and the adjoint of a Controlled-Not is a Controlled-Not as well. For $ZZZZ$, or even longer sequences of Z-gates, we build a cascading gate sequence in circuit notation, as shown in Figure 6.30. Now that we have this methodology available, we can measure in any Pauli measurement basis! And just to make sure, you can verify the construction for $ZZZZ$ with a short code sequence like the following:

```
op1 = ops.Cnot(1, 0) * ops.Identity() * ops.Identity()
op2 = ops.Identity() * ops.Cnot(2, 1) * ops.Identity()
op3 = ops.Identity() * ops.Identity() * ops.Cnot(3, 2)

bigop = op1 @ op2 @ op3 @ ops.PauliZ(4) @ op3 @ op2 @ op1
op = ops.PauliZ() * ops.Identity(3)
self.assertTrue(bigop.is_close(op))
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

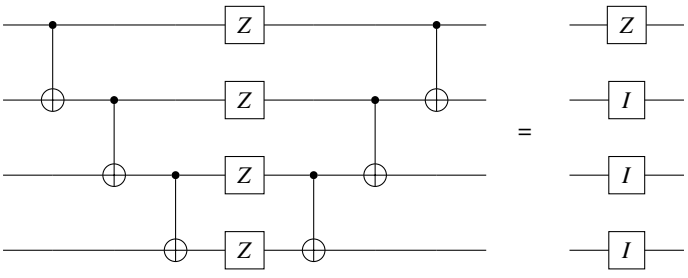


Figure 6.30 Measuring in the $ZZZZ$ basis.

6.12 Quantum Approximate Optimization Algorithm

In this section, we describe the *Quantum Approximate Optimization Algorithm*, or QAOA (pronounced "Quah-Wah"). It was first introduced in the seminal paper by Farhi et al. (2014), which also detailed using QAOA for an implementation of the Max-Cut algorithm. We explore Max-Cut in Section 6.13.

The QAOA technique is related to VQE, which could be considered a subroutine of QAOA. Here we provide only a short overview. There are two operators in QAOA, U_C and U_B. The first operator U_C applies a phase to pairs of qubits with a problem-specific cost function C, which is similar to the Ising formulation below in Section 6.13.1, with Z_i being the Pauli Z-gate applied to qubit i and l being the number of qubits or vertices involved:

$$C = \sum_{j,k}^{l} w_{jk} Z_j Z_k.$$

The operator itself depends on the phase angle γ:

$$U_C(\gamma) = e^{-i\gamma C} = \prod_{j,k} e^{i\gamma w_{jk} Z_j Z_k}.$$

This operator acts on two qubits and thus can be used for problems that can be expressed as weighted graphs.

The second operator U_B depends on a parameter β. It is problem-independent and applies rotations to each qubit with the following, where each X_j is a Pauli X-gate:

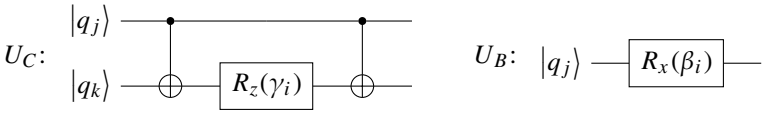
$$U_B(\beta) = e^{-i\beta B} = \prod_j e^{-i\beta X}, \quad \text{where } B = \sum_j X_j.$$

For problems with higher depth, these two operators U_C and U_B are applied repeatedly, each with their own set of hyperparameters γ_i and β_i, on an initial state of $|+\rangle^{\otimes n}$:

$$U_B(\beta_{n-1}) U_C(\gamma_{n-1}) \ldots U_B(\beta_0) U_C(\gamma_0) \; |+\rangle^{\otimes n}.$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

The task at hand is then similar to VQE – find the best possible set of hyperparameters to minimize the expectation value for the cost function $\langle \gamma, \beta | C | \gamma, \beta \rangle$, using well-known optimization techniques, for example, from the area of machine learning. The operators U_C and U_B can be approximated with the following circuits:



We know from Section 6.11 on VQE how to implement this type of search, so we won't expand on this further.

The original QAOA paper showed that for 3-regular graphs, which are cubic graphs with each vertex having exactly three edges, the algorithm produces a cut that is at least 0.7 of the maximum cut, a number that we are roughly able to confirm in our experiments below. Together with VQE, QAOA is an attractive algorithm for today's NISQ machines with limited resources, as the corresponding circuits have a shallow depth (Preskill, 2018). At the same time, QAOA's utility for industrial-sized problems is still under debate (Harrigan et al., 2021).

6.13 Maximum Cut Algorithm

In the previous section, we saw how the VQE finds the minimum eigenvalue and eigenvector for a Hamiltonian. This is an exciting methodology because if we can successfully frame an optimization problem as a Hamiltonian, we can use a VQE to find an optimal solution. This section briefly describes how to construct a class of such Hamiltonians: the Ising spin glass model, representing a multivariate optimization problem. The treatment here is admittedly shallow but sufficient to implement examples – the Max-Cut and Min-Cut algorithms in this section and the Subset Sum problem in Section 6.14.

6.13.1 Ising Formulations of NP Algorithms

The construction of a Hamiltonian with the Pauli σ_z operators is based on Hamiltonians for Ising spin glass (Lucas, 2014). This type of Hamiltonian is written as an unconstrained quadratic programming problem (a multivariate optimization problem). In the Ising model of ferromagnetism, J_{ij} is the *interaction* between a pair i, j of neighboring spins. It takes the values $J_{ij} = 0$ for no interaction, $J_{ij} > 0$ for ferromagnetism, and $J_{ij} < 0$ for antiferromagnetism:

$$\sum_{i}^{N} h_i x_i + \sum_{i,j} J_{ij} x_i y_j,$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

which corresponds to this Hamiltonian:

$$H(x_0, x_1, \ldots, x_n) = -\sum_{i}^{N} h_i \sigma_i^z - \sum_{i,j} J_{ij} \sigma_i^z \sigma_j^z.$$

The term σ_i^z is the application of a Pauli Z-gate on qubit i. The minus sign explains that we can look for a minimum eigenvalue to find a maximum solution. For problems such as Max-Cut, we use σ^z because we want an operator with eigenvalues -1 and $+1$.

With this background, Lucas (2014) details several NP-complete or NP-hard problems for which this approach may work. The list of algorithms includes partitioning problems, graph coloring problems, covering and packing problems, Hamiltonian cycles (including the traveling salesman problem), and tree problems. In the next few sections, we develop a related problem, the graph Max-Cut problem. We will also explore a slightly modified formulation of the Subset Sum problem.

6.13.2 Max-Cut/Min-Cut

For a graph, a *cut* is a partition of the graph's vertices into two nonoverlapping sets L and R. A *maximum cut* is the cut that maximizes the number of edges between L and R. Assigning weights to the graph edges turns the problem into the more general *weighted maximum cut*, which aims to find the cut that maximizes the weights of edges between sets L and R. This is the *Max-Cut* problem we are trying to solve in this section.

Weights can be both positive and negative. The Max-Cut problem turns into a Min-Cut problem simply by changing the sign of each weight. As an example, the maximum cut in Figure 6.31a, a graph of four nodes, is between the sets $L = \{0, 2\}$ and $R = \{1, 3\}$. The nodes are colored white or gray, depending on which set they belong to.

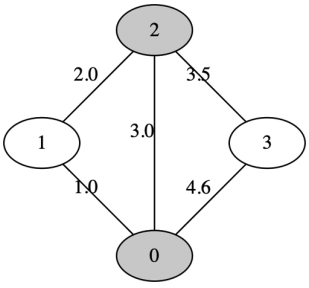
For a graph with just 15 nodes, as shown in Figure 6.31b, the problem becomes unwieldy very quickly. General Max-Cut is NP-complete, we don't know of any polynomial time algorithm to find an optimal solution. This looks like a formidable challenge for a quantum algorithm!

6.13.3 Construct Graphs

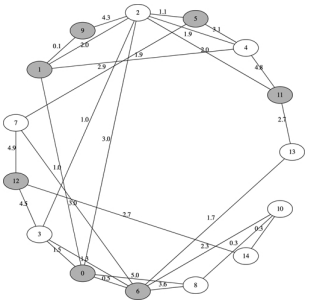
We begin our exploration by constructing a random graph with n vertices with the help of the following code (the implementation is in file `src/max_cut.py`). As usual, the code is designed for simplicity. We number vertices from 0 to $n - 1$ and represent vertices as simple Python tuples `(from_node, to_node, weight)`. A graph is then just a list of these tuples.

The code starts with a triangle of three nodes and then randomly adds new nodes up to limit `num`, ensuring that no double edges are generated.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press



(a) A graph with 4 vertices



(b) A graph with 15 vertices

Figure 6.31 Example graphs with marked Max-Cut sets.

```
def build_graph(num: int = 0) -> (int, List[int]):
    """Build a graph of num nodes."""

    if num < 3:
        raise app.UsageError('Must request graph of at least 3 nodes.')
    weight = 5.0
    nodes = [(0, 1, 1.0), (1, 2, 2.0), (0, 2, 3.0)]
    for i in range(num-3):
        l = random.sample(range(0, 3 + i - 1), 2)
        nodes.append((3 + i, l[0],
                      weight*np.random.random()))
        nodes.append((3 + i, l[1],
                      weight*np.random.random()))
    return num, nodes
```

For debugging and intuition, it helps to visualize the graph. The output of the routine below can be used to visualize the graphs with Graphviz (graphviz.org, 2021). The graphs in Figure 6.31 were produced this way.

```
def graph_to_dot(n: int, nodes: List[int], max_cut) -> None:
    """Convert graph (up to 64 nodes) to dot file."""

    print('graph {')
    print('  {\n    node [  style=filled ]')
    pattern = bin(max_cut)[2:].zfill(n)
    for idx, val in enumerate(pattern):
        if val == '0':
            print(f'    "{idx}" [fillcolor=lightgray]')
    print('  }')
    for node in nodes:
        print('  "{}" -- "{}" [label="{:.1f}",weight="{:.2f}"];'
              .format(node[0], node[1], node[2], node[2]))
    print('}')
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

6.13.4 Compute Max-Cut

Graph nodes are numbered from 0 to $n-1$. We use a binary representation to encode a cut. For example, nodes 0 and 2 in Figure 6.31a are in set L; nodes 1 and 3 are in set R. We align node 0 with index 0 in a binary bitstring (from left to right) and represent the cut as the binary string 1010. We extend this representation to quantum states, associating qubits q_i with graph nodes n_i:

$$\Big|\underbrace{1}_{n_0}\underbrace{0}_{n_1}\underbrace{1}_{n_2}\underbrace{0}_{n_3}\Big\rangle$$

We can compute the Max-Cut exhaustively, and quite inefficiently, given our choice of data structures. For n nodes, we generate all binary bitstrings from 0 to n. For each bitstring, we iterate over the individual bits and build two index sets: indices with a 0 in the bitstring, and indices with a 1 in the bitstring. For example, the bitstring `11001` would create set $L = \{0, 1, 4\}$ and set $R = \{2, 3\}$.

The calculation then iterates over all edges in the graph. For each edge, if one of the vertices is in L and the other in R, there is an edge between sets. We add the edge weight to the currently computed maximum cut and maintain the absolute maximum cut. Finally, we return the corresponding bit pattern as a decimal. For example, if the maximum cut was binary `11001`, the routine returns 25 (this routine will only work with up to 64 bits or vertices).

```
def compute_max_cut(n: int, nodes: List[int]) -> int:
    """Compute (inefficiently) the max cut, exhaustively."""

    max_cut = -1000
    for bits in helper.bitprod(n):
        # Collect in/out sets.
        iset = []
        oset = []
        for idx, val in enumerate(bits):
            iset.append(idx) if val == 0 else oset.append(idx)

        # Compute costs for this cut, record maximum.
        cut = 0
        for node in nodes:
            if node[0] in iset and node[1] in oset:
                cut += node[2]
            if node[1] in iset and node[0] in oset:
                cut += node[2]
        if cut > max_cut:
            max_cut_in, max_cut_out = iset.copy(), oset.copy()
            max_cut  = cut
            max_bits = bits

    state = bin(helper.bits2val(max_bits))[2:].zfill(n)
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
    print('Max Cut. N: {}, Max: {:.1f}, {}  -  {}, |{}>'
          .format(n, np.real(max_cut), max_cut_in, max_cut_out,
                  state))
    return helper.bits2val(max_bits)
```

The performance of this code is, of course, quite horrible but perhaps indicative of the combinatorial character of the problem. On a standard workstation, computing the Max-Cut for 20 nodes takes about 10 seconds; for 23 nodes it takes about 110 seconds. Even considering performance differences between Python and C++ and the relatively poor choice of data structure, it is obvious that the runtime will quickly become intractable for larger graphs.

Note that the solution is symmetric. If a Max-Cut is $L = \{0,1,4\}$ and $R = \{2,3\}$, then $L = \{2,3\}$ and $R = \{0,1,4\}$ is a Max-Cut as well.

6.13.5 Construct Hamiltonian

To construct the Hamiltonian, we iterate over the graph's edges. We build the tensor product with identity matrices for nodes that are not part of the edge and Pauli σ_z matrices for the vertices connected by the edge. This follows the very brief methodology outlined above in Section 6.13.1. We may also use the intuition that Pauli σ_z are "easy" to measure, as we have outlined in Section 6.11 on measuring in the Pauli bases. The Pauli matrix σ_z has eigenvalues $+1$ and -1. An edge can *increase* or *decrease* the energy of the Hamiltonian, depending on which set vertices fall into. This construction *increases* the energy for vertices that are in the same set.

As an example for the construction, for the graph in Figure 6.31a we would build these tensor products for all edges $e_{from,to}$:

$$
\begin{aligned}
e_{0,1} &= 1.0\,(Z \otimes Z \otimes I \otimes I),\\
e_{0,2} &= 3.0\,(Z \otimes I \otimes Z \otimes I),\\
e_{0,3} &= 4.6\,(Z \otimes I \otimes I \otimes Z),\\
e_{1,2} &= 2.0\,(I \otimes Z \otimes Z \otimes I),\\
e_{2,3} &= 3.5\,(I \otimes I \otimes Z \otimes Z).
\end{aligned}
$$

We add up these partial operators to construct the final Hamiltonian, which mirrors Equation (6.17).

$$\mathcal{H} = e_{0,1} + e_{0,2} + e_{0,3} + e_{1,2} + e_{2,3}.$$

Here is the code to construct the Hamiltonian in full matrix form. It iterates over the edges and constructs the full tensor products as shown above:

```
def graph_to_hamiltonian(n: int, nodes: List[int]) -> ops.Operator:
    """Compute Hamiltonian matrix from graph."""

    # Full matrix.
    H = np.zeros((2**n, 2**n))
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
    for node in nodes:
        idx1 = node[0]
        idx2 = node[1]
        if idx1 > idx2:
            idx1, idx2 = idx2, idx1
        op = 1.0
        for _ in range(idx1):
            op = op * ops.Identity()
        op = op * (node[2] * ops.PauliZ())
        for _ in range(idx1 + 1, idx2):
            op = op * ops.Identity()
        op = op * (node[2] * ops.PauliZ())
        for _ in range(idx2 + 1, n):
            op = op * ops.Identity()
        H = H + op
    return ops.Operator(H)
```

As we have described so far, for a graph with n nodes, we would have to build operator matrices of size $2^n \times 2^n$, which does not scale well. Note, however, that both the identity matrix and σ_z are diagonal matrices. Tensor products of diagonal matrices result in diagonal matrices. For example:

$$I \otimes I \otimes Z = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$
$$= \text{diag}(1, -1, 1, -1, 1, -1, 1, -1).$$

If we apply a factor to any individual operator, that factor multiplies across the whole diagonal. Let us look at what happens if we apply σ_z at index $0, 1, 2, \ldots$ in the tensor products (from right to left):

$$I \otimes I \otimes Z = \text{diag}(+1, -1, +1, -1, +1, -1, \underbrace{+1}_{2^0}, \underbrace{-1}_{2^0}),$$
$$I \otimes Z \otimes I = \text{diag}(+1, +1, -1, -1, \underbrace{+1, +1}_{2^1}, \underbrace{-1, -1}_{2^1}),$$
$$Z \otimes I \otimes I = \text{diag}(\underbrace{+1, +1, +1, +1}_{2^2}, \underbrace{-1, -1, -1, -1}_{2^2}).$$

These are power-of-2 patterns, similar to those we have seen in the fast gate apply routines. This means we can optimize the construction of the diagonal Hamiltonian and only construct a diagonal tensor product! The full matrix code is very slow and can barely handle 12 graph nodes. The diagonal version below can easily handle twice

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

as many nodes. C++ acceleration might help to further improve scalability, especially because the calls to `tensor_diag` can be parallelized.

```
def tensor_diag(n: int, fr: int, to: int, w: float):
    """Construct a tensor product from diagonal I, Z matrices."""

    def tensor_product(w1:float, w2:float, diag):
        return [j for i in zip([x * w1 for x in diag],
                               [x * w2 for x in diag]) for j in i]

    diag = [w, -w] if (0 == fr or 0 == to) else [1, 1]
    for i in range(1, n):
    if i == fr or i == to:
        diag = tensor_product(w, -w, diag)
    else:
        diag = tensor_product(1, 1, diag)
    return diag

def graph_to_diagonal_h(n: int, nodes: List[int]) -> np.ndarray:
    """Construct diag(H)."""

    h = [0.0] * 2**n
    for node in nodes:
        diag = tensor_diag(n, node[0], node[1], node[2])
        for idx, val in enumerate(diag):
            h[idx] += val
    return h
```

6.13.6 VQE by Peek-A-Boo

After we constructed the Hamiltonian, we would typically run VQE (or QAOA) to find the minimum eigenvalue. The corresponding eigenstate would encode the Max-Cut in binary form. However, in our simulated case here we don't have to run the expensive VQE, we can just take a peek at the Hamiltonian's matrix. It is diagonal, meaning that the eigenvalues are on the diagonal. The corresponding eigenstate is a state vector with a single 1 at the same row or column in binary encoding as the minimum eigenvalue. For example, for the graph in Figure 6.32, the Hamiltonian is:

$$\begin{aligned}\mathcal{H} = \text{diag}(&49.91, -21.91, -18.67, -5.32, 10.67, -2.67, -41.91, 29.91,\\ &29.91, -41.91, -2.67, 10.67, -5.32, -18.67, -21.91, 49.91).\end{aligned}$$

The minimum value is -41.91 and appears in two places: at index 6, which is binary `0110`, and the complementary index 9, which is binary `1001`. This corresponds to state $|0110\rangle$ and the complementary $|1001\rangle$. This is precisely the Max-Cut pattern in Figure 6.32. We have found the Max-Cut by applying *VQE by peek-a-boo* on a properly prepared Hamiltonian!

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

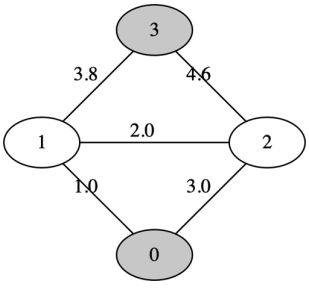


Figure 6.32 Graph with 4 nodes; Max-Cut is $\{0, 3\}, \{1, 2\}$, or `0110` in binary set encoding.

Here is the code to run experiments. It constructs the graph and computes the Max-Cut exhaustively. Then it computes the Hamiltonian and obtains the minimum value and its index off the diagonal.

```
def run_experiment(num_nodes: int):
    """Run an experiment, compute H, match against Max-Cut."""

    n, nodes = build_graph(num_nodes)
    max_cut  = compute_max_cut(n, nodes)
    #
    # These two lines are the basic implementation, where
    # a full matrix is being constructed. However, these
    # are diagonal, and can be constructed much faster.
    #   H       = graph_to_hamiltonian(n, nodes)
    #   diag    = H.diagonal()
    #
    diag     = graph_to_diagonal_h(n, nodes)
    min_idx  = np.argmin(diag)

    # Results...
    if flags.FLAGS.graph:
        graph_to_dot(n, nodes, max_cut)

    if min_idx == max_cut:
        print('SUCCESS: {:+10.2f} |{}>'.format(np.real(diag[min_idx]),
                      bin(min_idx)[2:].zfill(n)))
    else:
        print('FAIL   : {:+10.2f} |{}>  '.format(np.real(diag[min_idx]),
                      bin(min_idx)[2:].zfill(n)), end='')
        print('Max-Cut: {:+10.2f} |{}>'.format(np.real(diag[max_cut]),
                      bin(max_cut)[2:].zfill(n)))
```

Running this code, we find that it does *not always* work; it fails in about 20–30% of the invocations. Our criteria is very strict; to mark a run as successful, we check whether the optimal cut was found. Anything else is considered a failure. However

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

even if the optimal cut was not found, the results are still within 30% of optimal, and typically significantly below 20%. This matches the analysis from the QAOA paper.

For example, running over graphs with 12 nodes may produce output like the following:

```
Max Cut. N: 12, Max: 38.9, [0, 1, 4, 7, 9, 10]-[2, 3, 5, 6, 8, 11],
↪  |001101101001>
SUCCESS :    -129.39 |001101101001>
Max Cut. N: 12, Max: 39.5, [0, 1, 5, 6, 7, 9]-[2, 3, 4, 8, 10, 11],
↪  |001110001011>
SUCCESS :    -117.64 |001110001011>
Max Cut. N: 12, Max: 46.0, [0, 3, 5, 8, 11]-[1, 2, 4, 6, 7, 9, 10],
↪  |011010110110>
FAIL    :    -146.79 |001010110110>  Max-Cut:    -145.05 |011010110110>
[...]
Max Cut. N: 12, Max: 43.7, [0, 1, 3, 4, 7, 8, 9, 10]-[2, 5, 6, 11],
↪  |001001100001>
SUCCESS :    -124.69 |001001100001>
```

It is educational to experiment with the maximum degree of the graph, as it appears that this is one of the factors influencing the failure rate for this algorithm.

6.14 Subset Sum Algorithm

In Section 6.13, we saw how to use QAOA and VQE (by peek-a-boo) to solve an optimization problem. In this section, we explore another algorithm of this type, the so-called Subset Sum problem. Similar to Max-Cut, this problem is known to be NP-complete.

The problem can be stated the following way. Given a set S of integers, can S be divided into two sets, L and $R = S - L$, such that the sum of the elements in L equals the sum of the elements in R:

$$\sum_{i}^{|L|} l_i = \sum_{j}^{|R|} r_j.$$

We will also express this problem with a Hamiltonian constructed very similarly to Max-Cut, except that we will only introduce a single weighted Z-gate for each number in S. In Max-Cut we were looking for a minimal energy state. For this balanced sum problem, we have to look for a zero-energy state because this would indicate energy equilibrium, or equilibrium of the partial sums. Our implementation only decides whether or not a solution exists. It does not find a specific solution.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

6.14.1 Implementation

To start, and since this algorithm is begging to be experimented with, we define relevant parameters as command line options (the implementation is in file `src/subset_sum.py`). The highest integer in S is specified with parameter `nmax`. We will encode integers as positions in a bitstring, or, correspondingly, a state. For integers up to `nmax`, we will need `nmax` qubits. The size $|S|$ of set S is specified with parameter `nnum`. And finally, the number of experiments to run is specified with parameter `iterations`.

```
flags.DEFINE_integer('nmax', 15, 'Maximum number')
flags.DEFINE_integer('nnum',  6,
                     'Maximum number of set elements [1-nmax]')
flags.DEFINE_integer('iterations', 20, 'Number of experiments')
```

The next step is to get `nnum` random, unique integers in the range from 1 to `nmax` inclusive. Other ranges are possible, including negative numbers, but given that we use integers as bit positions, we have to map any such range to the range of 0 to `nmax`.

```
def select_numbers(nmax: int, nnum: int) -> List[int]:
  """Select nnum random, unique numbers in range 1 to nmax."""

  while True:
    sample = random.sample(range(1, nmax), nnum)
    if sum(sample) % 2 == 0:
        return sample
```

The next step is to compute the diagonal tensor product. Note that we only have to check for a single number and a correspondingly weighted (by index i) Z-gate.

```
def tensor_diag(n: int, num: int):
    """Construct tensor product from diagonal matrices."""

    def tensor_product(w1: float, w2: float, diag):
        return [j for i in zip([x * w1 for x in diag],
                               [x * w2 for x in diag]) for j in i]

    diag = [1, -1] if num == 0 else [1, 1]
    for i in range(1, n):
        if i == num:
            diag = tensor_product(i, -i, diag)
        else:
            diag = tensor_product(1, 1, diag)
    return diag
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

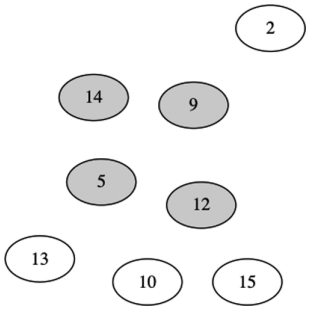


Figure 6.33 A subset partition for a set of eight integers. The partial sums of all elements in the white and gray sets are equal.

The final step for constructing the Hamiltonian is to add up all the diagonal tensor products from the step above. This function is identical to the same function in the Max-Cut algorithm, except for the invocation of the routine to compute the diagonal tensor product itself. If we implemented more algorithms of this type, we would clearly generalize the construction.

```
def set_to_diagonal_h(num_list: List[int],
                      nmax: int) -> np.ndarray:
    """Construct diag(H)."""

    h = [0.0] * 2**nmax
    for num in num_list:
      diag = tensor_diag(nmax, num)
      for idx, val in enumerate(diag):
          h[idx] += val
    return h
```

6.14.2 Experiments

Now on to experiments. We created a list of random numbers in the step above. The next step is to exhaustively compute potential partitions. Similar to Max-Cut, we divide the set of numbers into two sets with the help of binary bit patterns. For each division, we compute the two sums for these two sets. If the results match up, we add the corresponding bit pattern to the list of results. The routine then returns this list, which could be empty if no solution was found for a given set of numbers. A sample set partition is shown in Figure 6.33.

```
def compute_partition(num_list: List[int]):
    """Compute partitions that add up."""

    solutions = []
    for bits in helper.bitprod(len(num_list)):
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
        iset = []
        oset = []
        for idx, val in enumerate(bits):
            (iset.append(num_list[idx]) if val == 0 else
             oset.append(num_list[idx]))
        if sum(iset) == sum(oset):
            solutions.append(bits)
    return solutions
```

Of course we need a small facility to print results:

```
def dump_solution(bits: List[int], num_list: List[int]):
    iset = []
    oset = []
    for idx, val in enumerate(bits):
        (iset.append(f'{num_list[idx]:d}') if val == 0  else
         oset.append(f'{num_list[idx]:d}'))
    return '+'.join(iset) + ' == ' + '+'.join(oset)
```

Finally, we run the experiments. For each experiment, we create a set of numbers, compute the solutions exhaustively, and compute the Hamiltonian.

```
def run_experiment() -> None:
    """Run an experiment, compute H, match against 0."""

    nmax = flags.FLAGS.nmax
    num_list = select_numbers(nmax, flags.FLAGS.nnum)
    solutions = compute_partition(num_list)
    diag = set_to_diagonal_h(num_list, nmax)
```

We perform VQE by peek-a-boo. For Max-Cut we plucked a defined index and value from the diagonal. But what is the right value to look for here? Ultimately, we are looking for a zero-energy state, because this would indicate a balance between sets L and R. Hence we look for zeros on the diagonal of the Hamiltonian. There can be multiple zeros, but as a long as a one single zero can be found, we know there should be a solution. If no solution was found exhaustively, but we still find a zero on the diagonal, we know that we have encountered a false positive. Conversely, if no zero was found on the diagonal, but the exhaustive search did identify a solution, we encountered a false negative. The code below checks for both conditions.

```
    non_zero = np.count_nonzero(diag)
    if non_zero != 2**nmax:
        print('Solution should exist...', end='')
        if solutions:
            print(' Found Solution:',
                  dump_solution(solutions[0], num_list))
            return True
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
        raise AssertionError('False positive found.')
    if solutions:
        raise AssertionError('False negative found.')
    return False
```

As we run the code, we should see a 100% success rate:

```
    for i in range(flags.FLAGS.iterations):
        ret = run_experiment()
[...]
Solution should exist... Found Solution: 13+1+5+3 == 14+8
Solution should exist... Found Solution: 4+9+14 == 12+5+10
Solution should exist... Found Solution: 10+1+14 == 4+12+9
Solution should exist... Found Solution: 12+7+10 == 6+9+14
[...]
Solution should exist... Found Solution: 1+3+11+2 == 5+12
Solution should exist... Found Solution: 13+5+7 == 2+9+14
```

6.15 Solovay–Kitaev Theorem and Algorithm

We now switch gears and discuss the Solovay–Kitaev (SK) theorem and corresponding algorithm (Kitaev et al., 2002). It proves that *any* unitary gate can be approximated from a finite set of universal gates, which in the case of single-qubit gates, are just Hadamard and T-gates. This theorem is one of the key results in quantum computing. A version of the theorem that seems appropriate in our context is the following:

THEOREM 6.1 (Solovay–Kitaev theorem) *Let G be a finite set of elements in $SU(2)$ containing its own inverses, such that $\langle G \rangle$ is dense in SU(2). Let $\epsilon > 0$ be given. Then there is a constant c such that for any U in $SU(2)$ there is a sequence of gates of a length $O(log^c(1/\epsilon))$ such that $||S - U|| < \epsilon$.*

In English, this theorem says that for a given unitary gate U, there is a finite sequence of universal gates that will approximate U up to any precision. To a degree, this should *not* surprise us because if this was not the case, the set of universal gates could hardly be called universal. The produced gate sequences can be quite long, and it appears the field has moved on (Ross and Selinger, 2016, 2021; Kliuchnikov et al., 2015). Nevertheless, the algorithm was seminal and is supremely elegant.

We will study it using the pedagogical review from Dawson and Nielsen (2006) as a guide. We start with a few important concepts and functions. Then we outline the high-level structure of the algorithm before diving deeper into the complex parts and implementation.

6.15.1 Universal Gates

For single qubits, a set of *universal* gates consists of the Hadamard gate H and the T-gate. We call this set "universal" because any point on a Bloch sphere can be reached

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

by a sequence of just these two gates. We prove this below by showing that the SK algorithm, based on just these two gates, can approximate any unitary matrix up to arbitrary precision. We develop the implementation here in a piecemeal fashion, but the full code is available in the open-source repository `src/solovay_kitaev.py`.

6.15.2 SU(2)

One of the requirements for the SK algorithm is that the universal gates involved are part of the $SU(2)$ group, which is the group of all 2×2 unitary matrices with determinant 1. The determinants of both the Hadamard gate and the T-gate are not 1. In order to convert the gates to become members of $SU(2)$, we apply the simple transformation:

$$U' = \sqrt{\frac{1}{\det U}} U.$$

```
def to_su2(U):
    """Convert a 2x2 unitary to a unitary with determinant 1.0."""

    return np.sqrt(1 / np.linalg.det(U)) * U
```

We won't go deeper into $SU(2)$ and the related mathematics of Lie groups. For our purposes, we should simply think of $SU(2)$ in terms of rotations. For a given rotation V, the inverse rotation is $V^\dagger$, with $VV^\dagger = I$. For two rotations U and V, the inverse of UV is $V^\dagger U^\dagger$, with $UVV^\dagger U^\dagger = I$. However, similar to how two perpendicular sides on a Rubik's cube rotate against each other, $UVU^\dagger V^\dagger \neq I$.

6.15.3 Bloch Sphere Angle and Axis

Any 2×2 unitary matrix is of this form:

$$U = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

We can also write such a unitary operator the following way. The notation $\hat{n}$ refers to orthogonal, 3-dimensional axes. The symbol $\vec{\sigma}$ refers to the Pauli matrices:

$$U = e^{i\theta \hat{n} \frac{1}{2}\vec{\sigma}} = I \cos(\theta/2) + i\hat{n}\vec{\sigma} \sin(\theta/2).$$

This means that any unitary matrix can be constructed from a linear combination of the Pauli matrices in a specified coordinate system. Applying these unitary operators is equal to rotations about the axis $\hat{n}$ by angle θ. Only elements on the axis remain untouched by the rotation. With this, we can compute the angle and axis for a given unitary matrix with the following expansions:

$$\begin{aligned} U &= e^{i\theta\hat{n}\cdot\frac{1}{2}\vec{\sigma}} = e^{i\theta/2\hat{n}\cdot\vec{\sigma}} \\ &= I\cos(\theta/2) + \hat{n}\cdot i\vec{\sigma}\ \sin(\theta/2) \\ &= I\cos(\theta/2) + n_1 i\sigma_1 \sin(\theta/2) + n_2 i\sigma_2 \sin(\theta/2) + n_3 i\sigma_3 \sin(\theta/2) \end{aligned}$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

$$
\begin{aligned}
&= \begin{bmatrix} \cos(\theta/2) & 0 \\ 0 & \cos(\theta/2) \end{bmatrix} + \begin{bmatrix} 0 & n_1 i \sin(\theta/2) \\ n_1 i \sin(\theta/2) & 0 \end{bmatrix} \\
&\quad + \begin{bmatrix} 0 & n_2 \sin(\theta/2) \\ -n_2 \sin(\theta/2) & 0 \end{bmatrix} + \begin{bmatrix} n_3 i \sin(\theta/2) & 0 \\ 0 & -n_3 i \sin(\theta/2) \end{bmatrix} \\
&= \begin{bmatrix} \cos(\theta/2) + n_3 i \sin(\theta/2) & n_2 \sin(\theta/2) + n_1 i \sin(\theta/2) \\ -n_2 \sin(\theta/2) + n_1 i \sin(\theta/2) & \cos(\theta/2) - n_3 i \sin(\theta/2) \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.
\end{aligned}
$$

We compute the relevant parameters with simple algebraic transformations:

$$
\begin{aligned}
\theta &= 2 \arccos \frac{a+d}{2}, \\
n_1 &= \frac{b+c}{2i \sin(\theta/2)}, \\
n_2 &= \frac{b-c}{2 \sin(\theta/2)}, \\
n_3 &= \frac{a-d}{2i \sin(\theta/2)}.
\end{aligned}
$$

The corresponding (unoptimized) code is:

```
def u_to_bloch(U):
  """Compute angle and axis for a unitary."""

  angle = np.real(np.arccos((U[0, 0] + U[1, 1])/2))
  sin = np.sin(angle)
  if sin < 1e-10:
    axis = [0, 0, 1]
  else:
    nx = (U[0, 1] + U[1, 0]) / (2j * sin)
    ny = (U[0, 1] - U[1, 0]) / (2 * sin)
    nz = (U[0, 0] - U[1, 1]) / (2j * sin)
    axis = [nx, ny, nz]
  return axis, 2 * angle
```

6.15.4 Similarity Metrics

The *trace distance* is a similarity measure for two states. Typically, this concept is applied for states that are expressed as density matrices, but we may as well adopt it here to measure similarity between operators. For two operators ρ and ϕ, the trace distance is defined as:

$$
T(\rho, \phi) = \frac{1}{2} \operatorname{tr} \left[\sqrt{(\rho - \phi)^\dagger (\rho - \phi)} \right].
$$

In code this looks straightforward:

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
def trace_dist(U, V):
  """Compute trace distance between two 2x2 matrices."""

  return np.real(0.5 * np.trace(np.sqrt((U - V).adjoint() @ (U - V))))
```

There are other similarity metrics. For example, *quantum fidelity*. It is instructive to experiment with these kind of measures to study their impact on the achieved accuracy of our implementation:

$$F(\rho,\phi) = \left(\mathrm{tr}\left[\sqrt{\sqrt{\rho}\;\phi\;\sqrt{\rho}}\right]\right)^2.$$

6.15.5 Pre-computing Gates

The SK algorithm is recursive. At the innermost step, it maps a given unitary operator U against a library of pre-computed gate sequences, picking the resulting gate *closest* to U, as measured by the trace distance (or other similarity metrics).

The process of pre-computing gate sequences can be made quite simple. We only provide a most basic implementation, which is slow, but has the advantage of being easy to understand. There are only two base gates, as shown above. We generate all bitstrings up to a certain length, such as 0, 1, 00, 01, 10, 11, 000, 001, and so on. We initialize a temporary gate with the identity gate I and iterate through each bitstring, multiplying the temporary gate with one of the two basis gates, depending on whether a bit in the bitstring was set to 0 or 1. The function returns a list of all pre-computed gates. For simplicity, we throw away the actual gate sequences; we just keep the resulting unitary matrix, knowing that it was computed from the base gates.

```
def create_unitaries(base, limit):
  """Create all combinations of all base gates, up to length 'limit'."""

  # Create bitstrings up to bitstring length limit-1:
  #  0, 1, 00, 01, 10, 11, 000, 001, 010, ...
  #
  # Multiply together the 2 base operators, according to their index.
  # Note: This can be optimized, by remembering the last 2^x results
  # and multiplying them with base gates 0, 1.
  #
  gate_list = []
  for width in range(limit):
    for bits in helper.bitprod(width):
      U = ops.Identity()
      for bit in bits:
        U = U @ base[bit]
      gate_list.append(U)
  return gate_list
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

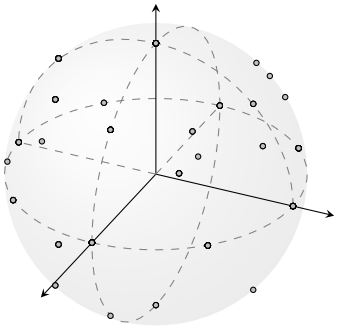


Figure 6.34 Distribution of 256 generated gate sequences applied to state $|0\rangle$. There are many duplicates.

To look up the closest gate, we iterate over the list of gates, compute the trace distance to each one, and return the gate with the minimum distance. Again, this code is kept simple for illustrative purposes. It is horribly slow, but there are ways to speed it up significantly, for example, with KD-trees (Wikipedia, 2021a).

```
def find_closest_u(gate_list, u):
  """Find the one gate in the list closest to u."""

  min_dist, min_u = 10, ops.Identity()
  for gate in gate_list:
    tr_dist = trace_dist(gate, u)
    if tr_dist < min_dist:
      min_dist, min_u = tr_dist, gate
  return min_u
```

Note that the way we generate gate sequences here leads to duplicate gates. For example, when plotting the effects of the generated gates on state $|0\rangle$, we see that the resulting distinct gates are quite sparse on the Bloch sphere, as shown in Figure 6.34.

6.15.6 Algorithm

Now we are ready to discuss the algorithm. We describe it in code, explaining it line by line. Inputs are the unitary operator U we seek to approximate and a maximum recursion depth n.

```
def sk_algo(U, gates, n):
  if n == 0:
    return find_closest_u(gates, U)
  else:
    U_next = sk_algo(U, gates, n-1)
    V, W   = gc_decomp(U @ U_next.adjoint())
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
  V_next = sk_algo(V, gates, n-1)
  W_next = sk_algo(W, gates, n-1)
  return (V_next @ W_next @ V_next.adjoint() @ W_next.adjoint() @
          U_next)
```

The recursion is counting down from an initial value of `n` and stops as it reaches the termination case with `n==0`. At this point, the algorithm looks up the closest pre-computed gate it can find.

```
if n == 0:
  return find_closest_u(gates, U)
```

Starting with this basic approximation, the following steps further improve the approximation by applying sequences of other inaccurate gates. The magic of this algorithms is, of course, that this actually works!

The first recursive step tries to find an approximation of `U`. For example, if `n==1`, the recursion would reach the termination clause and return the closest pre-computed gate.

```
  U_next = sk_algo(U, gates, n-1)
```

The next key steps are now to define $\Delta = UU_{n-1}^{\dagger}$ and to improve the approximation of Δ. We concatenate the two gate sequences for U and $U_{n-1}^{\dagger}$ to obtain the improved approximation. The interesting part here is that we use $U_{n-1}^{\dagger}$. The gate U_{n-1} got us closer to the target. The recursion wants to find out what we did *before* in order to arrive at this gate.

We decompose Δ as a *group commutator*, which is defined as $\Delta = VWV^{\dagger}W^{\dagger}$ with unitary gates V, W. There are an infinite number of such decompositions, but we apply an accuracy criterion to get a *balanced group commutator*. The math motivating this decomposition is beyond this book. We refer to Dawson and Nielsen (2006) and Kitaev et al. (2002) for details. Here we accept the result and show how to implement `gc_decomp()`.

```
  V, W   = gc_decomp(U @ U_next.adjoint())
```

The next recursive steps are then to get *improved* approximations for V, W with the same algorithm and to return a new and improved sequence $UU_{n-1}^{\dagger}$, as:

$$U_n = V_{n-1}W_{n-1}V_{n-1}^{\dagger}W_{n-1}^{\dagger}U_{n-1}.$$

```
  V_next = sk_algo(V, gates, n-1)
  W_next = sk_algo(W, gates, n-1)
  return (V_next @ W_next @
          V_next.adjoint() @ W_next.adjoint() @ U_next
```

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

6.15.7 Balanced Group Commutator

There is an infinite set of group commutator decompositions for a unitary operator U. We are looking for one for which $VWV^\dagger W^\dagger = U$, but with the distance between I and both V and W being smaller than a certain error bound. The difference between U_n and U_{n-1} will then be close to the identity matrix in the algorithm above as well. This condition can be expressed as the following, with $d()$ being a similarity metric and c being a constant:

$$d(I,V), d(I,W) < c\sqrt{\epsilon}.$$

This can be accomplished by considering V as a rotation by angle ϕ about the x-axis of a Bloch sphere and W as a similar rotation about the y-axis. The group commutator $VWV^\dagger W^\dagger$ is then a rotation about the Bloch sphere around axis $\hat{n}$ by an angle θ, satisfying Equation 6.18:

$$\sin(\theta/2) = 2\sin^2(\phi/2)\sqrt{1-\sin^4(\phi/2)}. \tag{6.18}$$

In the next few paragraphs, we will first derive Equation (6.18) and then solve for ϕ. Both V and W were defined as rotations about the x-axis and y-axis:

$$\begin{aligned} V &= R_x(\phi), \\ V^\dagger &= R_x(\phi)^\dagger = R_x(-\phi), \\ U &= VWV^\dagger W^\dagger = R_x(\phi)R_y(\phi)R_x(-\phi)R_y(-\phi). \end{aligned}$$

Similar to how we derived a unitary operator's Bloch sphere angle and axis, we express rotations as:

$$\begin{aligned} R_x(\phi) &= \cos(\phi/2)\, I + i\sin(\phi/2)\, X, \\ R_y(\phi) &= \cos(\phi/2)\, I + i\sin(\phi/2)\, Y. \end{aligned}$$

We can multiply this out and only evaluate the diagonal elements as above as $\cos\left(\frac{\theta}{2}\right) = \frac{a+d}{2}$ to arrive at:

$$\cos(\theta/2) = \cos^4(\phi/2) + 2\cos^2(\phi/2)\sin^2(\phi/2) - \sin^4(\phi/2).$$

We can factor out $\cos^2(\phi/2) + \sin^2(\phi/2)$ for:

$$\begin{aligned} \cos(\theta/2) &= \cos^4(\phi/2) + 2\cos^2(\phi/2)\sin^2(\phi/2) - \sin^4(\phi/2) \\ &= \left(\cos^2(\phi/2) + \sin^2(\phi/2)\right)^2 - 2\sin^4(\phi/2) \\ &= 1 - 2\sin^4(\phi/2). \end{aligned}$$

Using Pythagoras' theorem, we get the form we were looking for:

$$\begin{aligned} \sin^2(\theta/2) &= 1 - \cos^2(\theta/2) \\ &= 1 - \left(1 - 2\sin^4(\phi/2)\right)^2 \end{aligned}$$

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

$$
\begin{aligned}
&= 4\sin^4\left(\phi/2\right) - 4\sin^8\left(\phi/2\right) \\
&= 4\sin^4\left(\phi/2\right)\left(1 - \sin^4\left(\phi/2\right)\right) \\
\Rightarrow \sin\left(\theta/2\right) &= 2\sin^2\left(\phi/2\right)\sqrt{1 - \sin^4\left(\phi/2\right)}.
\end{aligned}
$$

Now on to solving for ϕ. From what we've done so far, we know how to compute θ for an operator. We get rid of the square root in Equation (6.18) by squaring the whole equation. For ease of notation, we substitute x for the left side:

$$
\begin{aligned}
x = \left(\frac{\sin\left(\theta/2\right)}{2}\right)^2 &= \left(\sin^2\left(\phi/2\right)\sqrt{1 - \sin^4\left(\phi/2\right)}\right)^2 \\
x &= \sin^4\left(\phi/2\right)\left(1 - \sin^4\left(\phi/2\right)\right) \\
&= \sin^4\left(\phi/2\right) - \sin^8\left(\phi/2\right) \\
\Rightarrow 0 &= \sin^4\left(\phi/2\right) - \sin^8\left(\phi/2\right) - x \\
&= \sin^8\left(\phi/2\right) - \sin^4\left(\phi/2\right) + x.
\end{aligned}
$$

This is a quadratic equation, which we can solve:

$$
\begin{aligned}
y^2 - y + x &= 0 \\
\Rightarrow \sin^4\left(\phi/2\right) = y &= \frac{1 \pm \sqrt{1 - 4x}}{2} \\
\sin\left(\phi/2\right) &= \sqrt{\sqrt{y}} \\
\phi &= 2\arcsin\left(\sqrt[4]{y}\right).
\end{aligned} \tag{6.19}
$$

Expand y (and remember that $\cos^2(\phi) + \sin^2(\phi) = 1$):

$$
\begin{aligned}
y &= \frac{1 \pm \sqrt{1 - 4x}}{2} \\
&= \frac{1 \pm \sqrt{1 - 4\sin^2\left(\theta/2\right)/4}}{2} \\
&= \frac{1 \pm \cos\left(\theta/2\right)}{2}.
\end{aligned}
$$

Substituting this into Equation (6.19) leads to the final result for ϕ. We ignore the $+$ case from the quadratic equation, as the goal was to arrive at Equation (6.18):[10]

$$
\phi = 2\arcsin\left(\sqrt[4]{\frac{1 - \cos\left(\theta/2\right)}{2}}\right).
$$

The construction proceeds as follows. We assumed that U is a rotation by angle θ about some axis $\hat{x}$. The angle ϕ is the solution to Equation (6.18). We define V, W to be rotations by ϕ, so U must be *conjugate* to the rotation by θ,

[10] We recommend that rigor-sensitive readers please hold their noses here.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

with $U = S(VWV^\dagger W^\dagger)S^\dagger$ for some unitary matrix S. We define $\hat{V} = SVS^\dagger$ and $\hat{W} = SWS^\dagger$ to obtain:

$$U = \hat{V}\hat{W}\hat{V}^\dagger\hat{W}^\dagger.$$

Let's write this in code. First we define the function `gc_decomp`, adding a helper function to diagonalize a unitary matrix. We compute θ and ϕ as described above:

```
def gc_decomp(U):
  """Group commutator decomposition."""

  def diagonalize(U):
    _, V = np.linalg.eig(U)
    return ops.Operator(V)

  # Get axis and theta for the operator.
  axis, theta = u_to_bloch(U)
  # The angle phi comes from eq 6.21 above.
  phi = 2.0 * np.arcsin(np.sqrt(
      np.sqrt((0.5 - 0.5 * np.cos(theta) / 2))))
```

We compute the axis on the Bloch sphere as shown above and construct the rotation operators V and W:

```
  V = ops.RotationX(phi)
  if axis[2] > 0:
    W = ops.RotationY(2 * np.pi - phi)
  else:
    W = ops.RotationY(phi)
```

Finally, we compute S as the transformation from U to the commutator:

```
  Ud = diagonalize(U)
  VWVdWd = diagonalize(V @ W @ V.adjoint() @ W.adjoint())
  S = Ud @ VWVdWd.adjoint()
```

And compute the results as outlined above:

```
  V_hat = S @ V @ S.adjoint()
  W_hat = S @ W @ S.adjoint()
  return V_hat, W_hat
```

6.15.8 Evaluation

For a brief and anecdotal evaluation, we define key parameters and run a handful of experiments. The number of experiments to run is given by `num_experiments`.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

Variable `depth` is the maximum length of the bitstrings we use to pre-compute gates. For a depth value x, $2^x - 1$ gates are pre-computed. Variable `recursion` is the recursion depth for the SK algorithm. It is instructive to experiment with these values to explore the levels of accuracy and performance you can achieve.

```
def main(argv):
  if len(argv) > 1:
    raise app.UsageError('Too many command-line arguments.')

  num_experiments = 10
  depth = 8
  recursion = 4
  print('SK algorithm - depth: {}, recursion: {}, experiments: {}'.
        format(depth, recursion, num_experiments))
```

Next we compute the $SU(2)$ base gates and create the pre-computed gates.

```
  base = [to_su2(ops.Hadamard()), to_su2(ops.Tgate())]
  gates = create_unitaries(base, depth)
  sum_dist = 0.0
```

Finally, we run the experiments. In each experiment, we create a unitary gate U from a randomly chosen combination of rotations. We apply the algorithm and compute and print distance metrics of the results. We also compare the impact of both original and approximated unitary gates on a $|0\rangle$ state. We compute the dot product between the resulting states and show how much it deviates from 1.0, expressed as a percentage. This may give an intuitive measure of the impact of the remaining approximation errors.

```
  for i in range(num_experiments):
      U = (ops.RotationX(2.0 * np.pi * random.random()) @
           ops.RotationY(2.0 * np.pi * random.random()) @
           ops.RotationZ(2.0 * np.pi * random.random()))

      U_approx = sk_algo(U, gates, recursion)
      dist = trace_dist(U, U_approx)
      sum_dist += dist

      phi1 = U(state.zero)
      phi2 = U_approx(state.zero)
      print('[{:2d}]: Trace Dist: {:.4f} State: {:6.4f}%'.
            format(i, dist,
                   100.0 * (1.0 - np.real(np.dot(phi1, phi2.conj())))))
  print('Gates: {}, Mean Trace Dist:: {:.4f}'.
        format(len(gates), sum_dist / num_experiments))
```

This should result in output like the following. With just 255 pre-computed gates (including many duplicates) and a recursion depth of 4, approximation accuracy falls consistently below 1%.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

```
$ bazel run solovay_kitaev
[...]
SK algorithm, depth: 8, recursion: 4. experiments: 10
[ 0]: Trace Dist: 0.0063 State: 0.0048%
[ 1]: Trace Dist: 0.0834 State: 0.3510%
[ 2]: Trace Dist: 0.0550 State: 0.1557%
[...]
[ 8]: Trace Dist: 0.1114 State: 0.6242%
[ 9]: Trace Dist: 0.1149 State: 0.6631%
Gates: 255, Mean Trace Dist:: 0.0698
```

6.15.9 Random Gate Sequences

The following question is interesting. How well does this algorithm perform compared to picking the best approximation from *random* sequences of base gates? We could try to answer this analytically, but we could also run experiments, construct random sequences of basis gates, and find the resulting unitary operator with the minimum trace distance to the original gate. Let's try that.

```
def random_gates(min_length, max_length, num_experiments):
  """Just create random sequences, find the best."""

  base = [to_su2(ops.Hadamard()), to_su2(ops.Tgate())]

  U = (ops.RotationX(2.0 * np.pi * random.random()) @
       ops.RotationY(2.0 * np.pi * random.random()) @
       ops.RotationZ(2.0 * np.pi * random.random()))
  min_dist = 1000
  for i in range(num_experiments):
    seq_length = min_length + random.randint(0, max_length)
    U_approx = ops.Identity()
    for j in range(seq_length):
      g = random.randint(0, 1)
      U_approx = U_approx @ base[g]
    dist = trace_dist(U, U_approx)
    min_dist = min(dist, min_dist)
  phi1 = U(state.zero)
  phi2 = U_approx(state.zero)
  print('Trace Dist: {:.4f} State: {:6.4f}%'.
        format(min_dist,
               100.0 * (1.0 - np.real(np.dot(phi1, phi2.conj())))))
```

It is educational to experiment with this approach. You can find approximated gates with small trace distances, but it appears the impact on basis states is much larger than for the SK algorithm. With longer gate sequences and many more tries, gate sequences can reach low trace distance deltas. However, to reach accuracies as shown above for the SK algorithm, the runtime can be orders of magnitude longer. To answer the

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press

question above about how well the SK algorithm performs – it does *very* well! Here is an example output from a sequence of randomized experiments:

```
Random Experiment, seq length: 10 - 50, tries: 100
Trace Dist: 0.2218 State: 58.4058%
Trace Dist: 0.2742 State: 39.3341%
[...]
Trace Dist: 0.2984 State: 198.4319%
Trace Dist: 0.2866 State: 102.0065%
```

This concludes the section on complex quantum algorithms. For a deeper mathematical treatment of these algorithms and their derivatives, see the Bibliography and relevant publications. For further reading on known algorithms, Mosca (2008) provides a detailed taxonomy and categorization of algorithms. The Quantum Algorithm Zoo lists another large number of algorithms alongside an excellent bibliography (Jordan, 2021). Abhijith et al. (2020) offers high-level descriptions of about 50 algorithm implementations in Qiskit.

https://doi.org/10.1017/9781009099974.008 Published online by Cambridge University Press