qubit by a single multiple of the phase, whereas Figure 7-28 shows we should rotate by eight times its value. This is why we need the exchanges at the very end of Figure 7-32.

How clever is that? The inverse QFT circuit performs the multiple steps needed for our desired periodically varying output, all compressed into a small multipurpose set of operations!

To simplify our preceding explanation we restricted ourselves to a single integer input to the invQFT, but the QPU operations we've used work just as well on a superposition of inputs.

Conclusion

In this chapter you have learned about one of the most powerful QPU primitives, the Quantum Fourier Transform. While the AA primitive allowed us to extract information about discrete values encoded in the phases of our register, the QFT primitive enables us to extract information about *patterns* of information encoded in the QPU register. As we will see in Chapter 12, this primitive is at the core of some of the most powerful algorithms that we can run on a QPU, including Shor's algorithm, which first kick-started mainstream interest in quantum computing.

CHAPTER 8

Quantum Phase Estimation

Quantum phase estimation (also referred to simply as phase estimation) is another QPU primitive for our toolbox. Like amplitude amplification and QFT, phase estimation extracts tangible, readable information from superpositions. Phase estimation is also quite possibly the most challenging primitive we've introduced so far, being conceptually tricky for two reasons:

1. Unlike the AA and QFT primitives, phase estimation teaches us an attribute of an *operation* acting on a QPU register, rather than an attribute of the QPU register state itself.
2. The specific attribute that phase estimation teaches us about a QPU operation, despite being hugely important in many algorithms, *appears* to be useless and arbitrary. Revealing its practical use is a challenge without resorting to some relatively advanced mathematics. But we'll give it a shot!

In this chapter we'll discuss what phase estimation is, try some hands-on examples, and then break it down operation by operation.

Learning About QPU Operations

Programming a problem that we want to solve into a QPU inevitably involves acting on some QPU register with the fundamental operations introduced in Chapters 2 and 3. The idea that we would want a primitive that teaches us something might sound strange—surely if we constructed a circuit, then we know everything there is to know about it! But some kinds of input data can be encoded in QPU operations, so learning more about them can potentially get us a long way toward finding the solutions that we're after.

For example, we will see in Chapter 13 that the HHL algorithm for inverting certain matrices encodes these matrices through judiciously chosen QPU operations. The properties of these operations, which are revealed by quantum phase estimation, teach us something critical and nontrivial about the matrix we need to invert.

Eigenphases Teach Us Something Useful

So precisely what property does phase estimation teach us about QPU operations? It's perhaps easiest to answer this question through an example. Let's take a look at our old friend `HAD`. Recall that when acting on a single qubit register `HAD` transforms $|0\rangle$ and $|1\rangle$ into entirely new states, as shown in Figure 8-1.

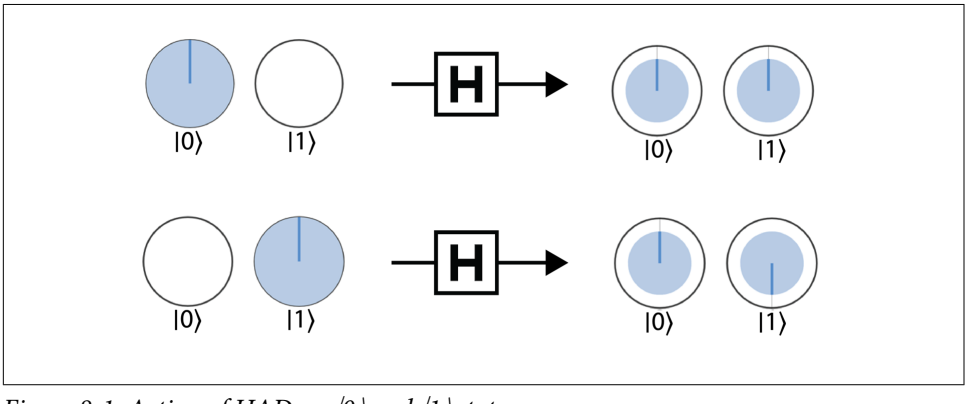


Figure 8-1. Action of HAD on $|0\rangle$ and $|1\rangle$ states

For most other input states `HAD` similarly produces entirely new output states. However, consider `HAD`'s action on the two special input states shown in Figure 8-2.

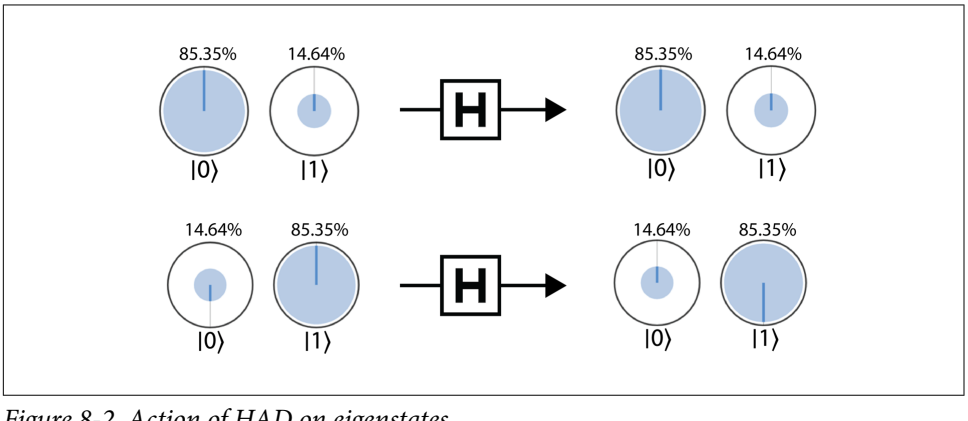


Figure 8-2. Action of HAD on eigenstates

The first input state has both its components in phase with a 14.64% chance of being `READ` to be in state $|1\rangle$, while the second input has its components 180° out of phase and a 14.64% chance[1] of being in state $|0\rangle$.

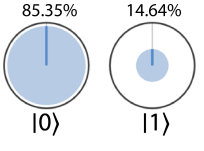
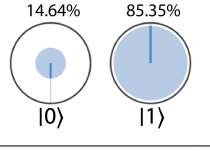
Take careful of note how `HAD` acts on these states. The first is completely unchanged, while the second only acquires a *global* phase of 180° (i.e., the *relative* phase is unchanged). We noted in Chapter 2 that global phases are unobservable, so we can equally say that `HAD` *effectively* leaves this second state unchanged.

States impervious to a certain QPU's operation in this way (except for global phases) are known as the operation's *eigenstates*. Every QPU operation has a distinct and unique set of such special states for which it will only impart an unimportant global phase.

Although the global phases that eigenstates can acquire are unobservable, they do teach us something revealing about the QPU operation producing them. The global phase acquired by a particular eigenstate is known as that eigenstate's *eigenphase*.

As we've just seen, `HAD` has two (and in fact only two) eigenstates, with associated eigenphases as shown in Table 8-1.

Table 8-1. Eigenstates and eigenphases of HAD

Eigenstate	Eigenphase
	0°
	180°

It's worth reiterating that the particular eigenstates (and associated eigenphases) shown in Table 8-1 are specific to `HAD`—other QPU operations will have entirely different eigenstates and eigenphases. In fact, the eigenstates and eigenphases of a QPU operation determine the operation entirely, in the sense that no other QPU operation has the same set.

1 The actual value of this probability is $\sin^2(22.5)$.

What Phase Estimation Does

Now that we're well versed in the language of eigenstates and eigenphases, we can describe what the phase estimation primitive achieves. Phase estimation will help determine the eigenphases associated with the eigenstates of a QPU operation, returning us a superposition of all the eigenphases. This is no mean feat, since global phases are usually unobservable artifacts. The beauty of the phase estimation primitive is that it finds a way of moving information about this global phase into another register—in a form that we *can* `READ`.

Why would we ever *want* to determine the eigenphases of a QPU operation? We'll see in later chapters how useful this can be, but our previous note that they can uniquely characterize a QPU operation should hint that they're powerful things to know.

For readers with experience in linear algebra: eigenstates and eigenphases are the *eigenvectors* and *complex eigenphases* of the unitary matrices representing QPU operations in the full mathematics of quantum computing.

How to Use Phase Estimation

Having an idea of what quantum phase estimation does, let's get our hands dirty and see how to utilize it in practice. Suppose we have some QPU operation U, which acts on n qubits and has some set of eigenstates, which we'll refer to as u_1, u_2, …, u_j. Using phase estimation, we wish to learn the eigenphases associated with each of these eigenstates. Don't forget that the eigenphase associated with the j^{th} eigenstate is always a global phase—so we can specify it by the angle θ_j through which the global phase rotates the register state. Using this notation, we can give a slightly more concise description of the task performed by phase estimation:

> Given a QPU operation U and one of its eigenstates u_j, phase estimation returns (to some precision) the corresponding eigenphase angle θ_j.

In QCEngine we can perform phase estimation using the built-in `phase_est()` function (see Example 8-2 for an implementation of this function in terms of more elementary QPU operations). To call this primitive successfully we need to understand what inputs it expects and how its output should be interpreted. These inputs and outputs are summarized in Figure 8-3.

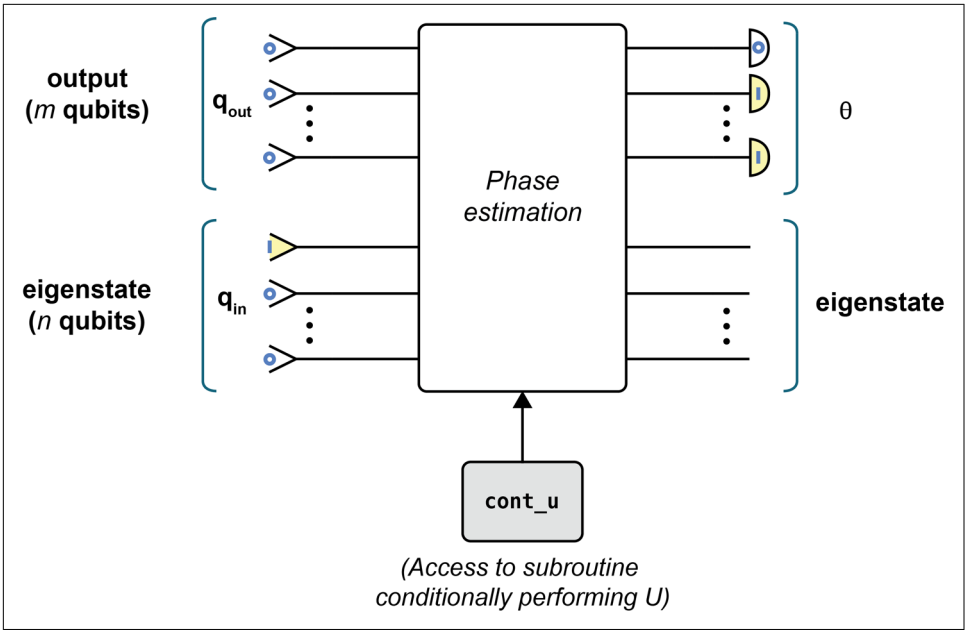


Figure 8-3. Overview of how to use phase estimation primitive

Let's take a more in-depth look at the arguments of `phase_est()`.

Inputs

Phase estimation's signature is:

```
phase_est(qin, qout, cont_u)
```

`qin` and `qout` are both QPU registers, while `cont_u` should be a reference to a function performing the QPU operation we're interested in (although in a particular way, as we'll see shortly):

`qin`
: An n-qubit input register prepared in the eigenstate u_j that we wish to obtain the eigenphase for.

`qout`
: A second register of m qubits, initialized as all zeros. The primitive will ultimately use this register to return a binary representation of the desired angle θ_j corresponding to the eigenstate we input in `qin`. In general, the larger we make m, the greater the precision with which we obtain a representation of θ_j.

`cont_u`
: An implementation of a *controlled* version of the QPU operation U that we are determining the eigenphases of. This should be passed as a function of the form `cont_u(in, out)`, which takes a single qubit `in` that will control whether U is applied to the n-qubit register `out`.

To give a concrete example of using phase estimation, we'll apply the primitive to find an eigenphase of `HAD`. We saw in Table 8-1 that one of `HAD`'s eigenstates has an eigenphase of 180°—let's see whether `phase_est()` can reproduce this result using the sample code in Example 8-1.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=8-1*.

Example 8-1. Using the phase estimation primitive

```
//Specify the size of output register - determines precision
// of our answer
var m = 4;
// Specify the size of input register that will specify
// our eigenstate
var n = 1;
// Setup
qc.reset(m + n);
var qout = qint.new(m, 'output');
var qin = qint.new(n, 'input');
// Initialize output register all zeros
qout.write(0);

// Initialize input register as eigenstate of HAD
qin.write(0);
qin.roty(-135);
// This state will have an eigenphase of 180.
// For eigenphase 0, we would instead use qin.roty(45);

// Define our conditional unitary
function cont_u(qcontrol, qtarget, control_count) {
    // For Hadamard, we only need to know if control_count
    // is even or odd, as applying HAD an even number of
    // times does nothing.
    if (control_count & 1)
        qtarget.chadamard(null, ~0, qcontrol.bits(control_count));
}
// Operate phase estimation primitive on registers
phase_est(qin, qout, cont_u);
// Read output register
qout.read();
```

We specify the eigenstate of interest with `qin`, and initialize `qout` as a four-qubit register of all zeros. When it comes to `cont_u`, it's worth emphasizing that we don't simply pass `HAD`, but rather a function implementing a *controlled* `HAD` operation. As we'll see later in this chapter, the inner workings of phase estimation explicitly require this. Since generating the *controlled* version of any given QPU operation can be nontrivial, `phase_est()` leaves it up to the user to specify a function achieving this. In this specific example we make use of the controlled version of `HAD` built into QCEngine, called `chadamard()`.

Figure 8-4 shows an overview in circle notation of what we can expect from running Example 8-1.

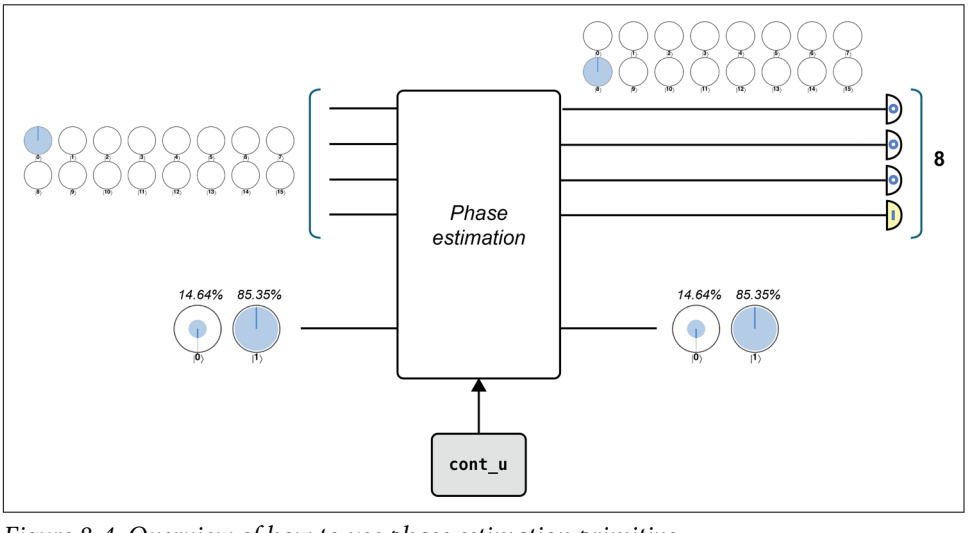


Figure 8-4. Overview of how to use phase estimation primitive

The state of the input register remains the same before and after we apply `phase_est()`, as expected. But what's going on with the output register? We were expecting 180° and we got 8!

Outputs

We obtain our eigenphase by applying `READ` operations to the m qubits of the output register. An important point to note is that the inner workings of phase estimation end up expressing θ_j as a *fraction of* 360°, which is encoded in the output register *as a fraction of its size*. In other words, if the output eigenphase was 90°, which is one quarter of a full rotation, then we would expect a three-qubit output register to yield the binary value for 2, which is also a quarter of the $2^3 = 8$ possible values of the register. For the case in Figure 8-4, we were expecting a phase of 180°, which is half of a full rotation. Therefore, in a four-qubit output register with $2^4 = 16$ values, we expect

the value 8, since that is exactly half the register size. The simple formula that relates the eigenphase (θ_j) with the register value we will read out (R), as a function of the size of the register, is given by Equation 8-1.

Equation 8-1. Relationship between output register (R), eigenphase (θ_j), and size of the QPU register m

$$R = \frac{\theta_j}{360^\circ} \times 2^m$$

The Fine Print

As is always the case with QPU programming, we should take careful note of any limitations we might face. With phase estimation, there are a few pieces of fine print to keep in mind.

Choosing the Size of the Output Register

In Example 8-1 the eigenphase we sought to reveal could be expressed perfectly in a four-qubit representation. In general, however, the precision with which we can determine an eigenphase will depend on the output register size. For example, with a three-qubit output register we can precisely represent the following angles:

Binary	000	001	010	011	100	101	110	111
Fraction of register	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{3}{8}$	$\frac{1}{2}$	$\frac{5}{8}$	$\frac{3}{4}$	$\frac{7}{8}$
Angle	0	45	90	135	180	225	270	315

If we were attempting to use this three-qubit output register to determine an eigenphase that had a value of 150°, the register's resolution would be insufficient. Fully representing 150° would require increasing the number of qubits in the output register.

Of course, endlessly increasing the size of our output register is undesirable. In cases where an output register has insufficient resolution we find that it ends up in a superposition centered around the closest possible values. Because of this superposition, we return the best lower-resolution estimate of our phase only with some probability. For example, Figure 8-5 shows the actual output state we obtain when running phase estimation to determine an eigenphase of 150° with an output register having only three qubits. This complete example can be run online at *http://oreilly-qc.github.io?p=8-2*.

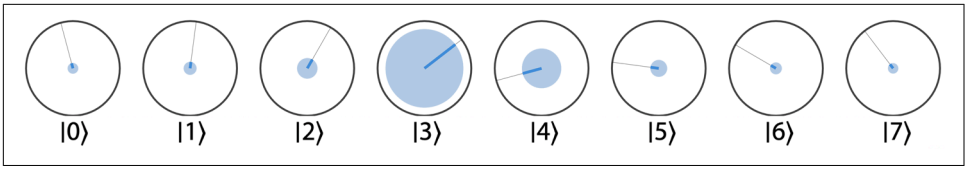


Figure 8-5. Estimating phases beyond the resolution of the output register

The probability of obtaining the best estimate is always greater than about 40%, and we can of course improve this probability by increasing the output register size.

If we want to determine the eigenphase to p bits of accuracy, and we want a probability of error (i.e., not receiving the best possible estimate) of no more than ϵ, then we can calculate the size of output register that we should employ, m, as:

$$m = p + \left\lceil \log\left(2 + \frac{1}{\epsilon}\right)\right\rceil$$

Complexity

The complexity of the phase estimation primitive (in terms of number of operations needed) depends on the number of qubits, m, that we use in our output register, and is given by $O(m^2)$. Clearly, the more precision we require, the more QPU operations are needed. We'll see in "Inside the QPU" on page 164 that this dependence is primarily due to phase estimation's reliance on the invQFT primitive.

Conditional Operations

Perhaps the biggest caveat associated with phase estimation is the assumption that we can access a subroutine implementing a *controlled* version of the QPU operation we want to find the eigenphases of. Since the phase estimation primitive calls this subroutine multiple times, it's critical that it can be performed *efficiently*. How efficiently depends on the requirements of the particular application making use of phase estimation. In general, if our `cont_u` subroutine has a complexity higher than $O(m^2)$, the overall efficiency of the phase estimation primitive will be eroded. The difficulty of finding such efficient subroutines depends on the particular QPU operation in question.

Phase Estimation in Practice

Phase estimation lets us extract the eigenphase associated with a particular eigenstate, requiring us to specify that eigenstate within an input register. This may sound a little contrived—how often would we happen to know the eigenstate, yet need to know the associated eigenphase?

The real utility of phase estimation is that—like all good QPU operations—we can work it in superposition! If we send a *superposition of eigenstates* as an input to the phase estimation primitive, we'll obtain a superposition of the associated eigenphases. The magnitude for each eigenphase in the output superposition will be precisely the magnitude that its eigenstate had in the input register.

This ability of phase estimation to act on superpositions of eigenstates makes the primitive especially useful, since it turns out that *any* state of a QPU register whatsoever can be thought of as a superposition of the eigenstates of any QPU operation.[2] This means that if we set the `cont_u` input of `phase_est()` to be some QPU operation U, and `qin` to be some general register state $|x\rangle$, then the primitive returns details of what eigenphases *characterize* the action of U on $|x\rangle$. Such information is useful in many mathematical applications involving linear algebra. The fact that we can effectively extract all these eigenphases *in superposition* raises the possibility of *parallelizing* these mathematical applications on our QPU (although, as always, with some fine print).

Inside the QPU

The inner workings of phase estimation are worth a peek. Not only does it instructively build on the QFT primitive introduced in Chapter 7, but phase estimation also plays a central role in many QPU applications.

Example 8-2 gives a complete implementation of the `phase_est()` function that we first used in Example 8-1.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=8-2*.

Example 8-2. Implementation of the phase estimation primitive

```
function phase_est(q_in, q_out, cont_u)
{
    // Main phase estimation single run
    // HAD the output register
    q_out.had();

    // Apply conditional powers of u
    for (var j = 0; j < q_out.numBits; j++)
        cont_u(q_out, q_in, 1 << j);
```

2 This fact is by no means obvious, but is possible to demonstrate with the full mathematical machinery of quantum computing. In Chapter 14 we point to more technical resources that give further insight into why this statement is true.

```
    // Inverse QFT on output register
    q_out.invQFT();
}
```

This code implements the circuit shown in Figure 8-6.

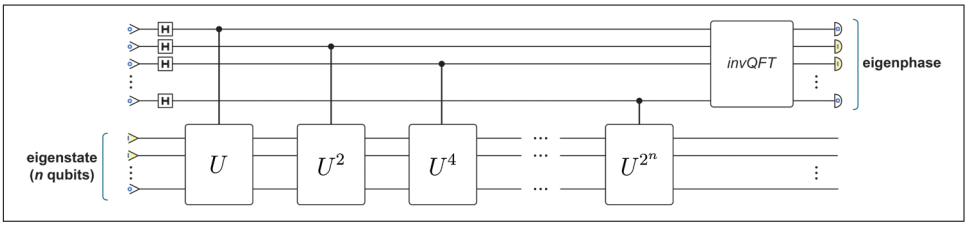


Figure 8-6. Full circuit for implementing phase estimation

Pretty concise! Our task is to explain how this manages to extract the eigenphases of the QPU operation passed to it through the `cont_u` parameter.

The Intuition

Getting eigenphases from a QPU operation *sounds* pretty simple. Since `phase_est()` has access to the QPU operation under scrutiny and one (or more) of its eigenstates, why not simply act the QPU operation on the eigenstate, as shown in Figure 8-7?

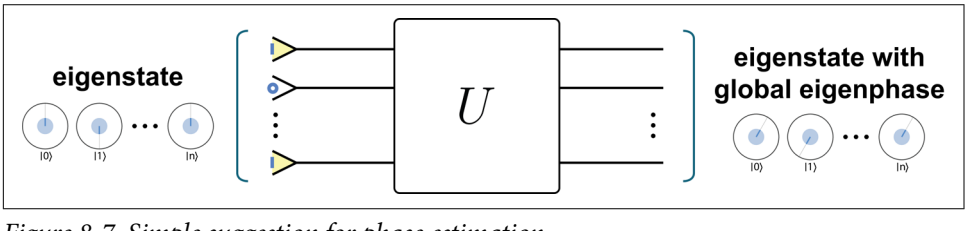


Figure 8-7. Simple suggestion for phase estimation

Thanks to the very definition of eigenstates and eigenphases, this simple program results in our output register being the same input eigenstate, but with the eigenphase applied to it as a global phase. Although this approach *does* represent the eigenphase θ in the output register, we already reminded ourselves earlier that a *global* phase cannot be `READ` out. So this simple idea falls foul of the all-too-familiar problem of having the information we want trapped inside the phases of a QPU register.

What we need is some way of tweaking Figure 8-7 to get the desired eigenphase in a more `READ`able property of our register. Looking back through our growing toolbox of primitives, the QFT offers some promise.

Recall that the QFT transforms periodic phase differences into amplitudes that can be read out. So *if* we can find a way to cause the relative phases in our output register to vary periodically with a frequency determined by our eigenphase, we're home free—we simply apply the *inverse* QFT to read out our eigenphase.

Using two explicit eigenphase angles as examples, Figure 8-8 shows what we're after.

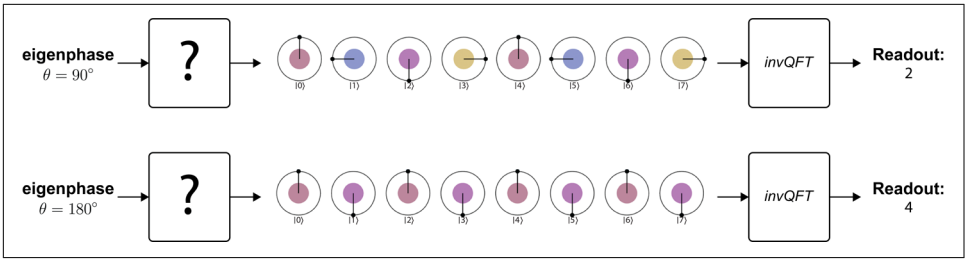


Figure 8-8. Two examples of representing eigenphases in register frequencies

We want to fill in the question marks with a set of QPU operations that produce these results. Say we're trying to determine an eigenphase of 90° (the first example in Figure 8-8), and we want to encode this in our register by having the register's relative phases rotate with a frequency of $\frac{90}{360} = \frac{1}{4}$. Since we have $2^3 = 8$ possible states in our register, this means we want the register to perform two full rotations across its length to give a frequency of $\frac{2}{8} = \frac{1}{4}$. When we perform the invQFT operation on this we, of course, read out a value of 2. From this we could successfully infer the eigenphase: $\frac{2}{8} = \frac{\theta}{360^\circ} \Rightarrow \theta = 90^\circ$.

After ruminating on Figure 8-8, one realizes that we can get the states we need with a few carefully chosen conditional rotations. We simply take an equal superposition of all possible register states and rotate the k^{th} state by k times whatever frequency is desired. That is, if we wanted to encode an eigenphase of θ, we would rotate the k^{th} state by $k\theta$.

This gives just the results we wanted in Figure 8-8, as we show explicitly in Figure 8-9 for the example of wanting to encode a 90° eigenphase in the eight states of a three-qubit register.

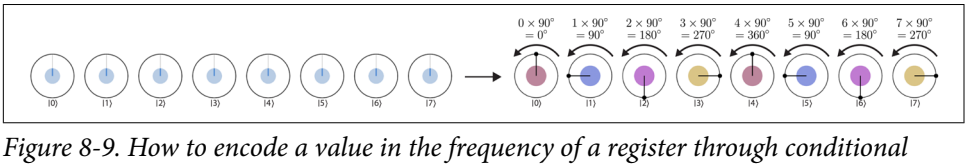


Figure 8-9. How to encode a value in the frequency of a register through conditional rotations

So, we've managed to reframe our problem as follows: *if we can rotate the k^{th} state of a register by k times the eigenphase we're after, then (via the inverse QFT) we'll be able to read it out*.

If this idea of rotating by multiples of a register's value sounds familiar, that's probably because this was precisely the approach that we used to understand the invQFT at the end of Chapter 7. Now, however, we need the register frequencies to be determined by the eigenphase of an operation.

Operation by Operation

As we'll now see, we can build up a circuit to achieve this kind of conditional rotation by combining the `cont_u` subroutine (providing conditional access to *U*) with the *phase-kickback* trick that we introduced in Chapter 3.

Every time we act our QPU operation *U* on its eigenstate we produce a global rotation by its eigenphase. Global phases aren't much good to us as they stand, but we can extend this idea and apply a global phase that is rotated by any integer multiple *k* (specified in another QPU register) of an eigenphase angle. The circuit shown in Figure 8-10 achieves this.

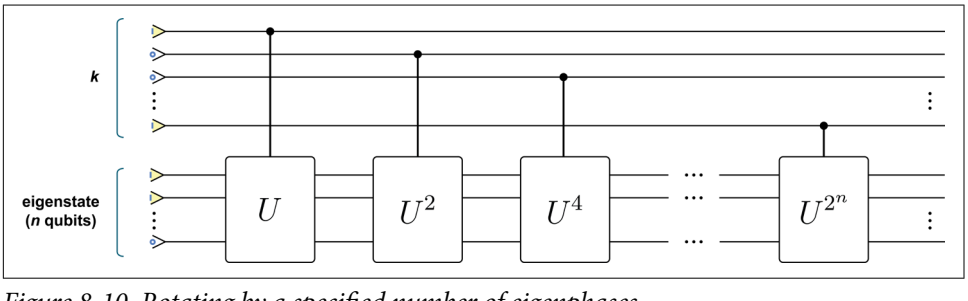


Figure 8-10. Rotating by a specified number of eigenphases

Every time we apply *U* to the eigenstate in the bottom register we rotate by the eigenphase θ. The choice of conditional operations simply performs the number of rotations required by each bit in the binary representation of *k*—resulting in us applying *U* a total of *k* times on our bottom eigenstate register—thus rotating it by $k\theta$.

This circuit allows us to implement just one global phase, for some single value of *k*. To perform the trick that we're asking for in Figure 8-9 we really want to implement such a rotation for all values of *k* in the register in superposition. Instead of specifying a single *k* value in the top register, let's use a uniform superposition of all 2^n possible values, as shown in Figure 8-11.

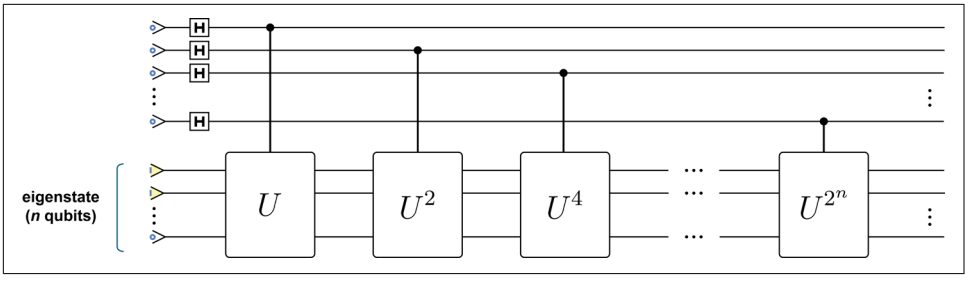


Figure 8-11. Conditionally rotating all states in a register at once

The result of this circuit on the second register is that *if* the first register was in state $|0\rangle$, then the second register would have its global phase rotated by $0 \times \theta = 0°$, whereas if the first register was in state $|1\rangle$, the second would be rotated by $1 \times \theta = \theta$, etc. But after all this, it still seems like we've only ended up with pretty useless (conditional) *global* phases in our *second* register.

What can we say about the state of the *first* register that initially held a superposition of k values? Recall from "QPU Trick: Phase Kickback" on page 51 that at the end of this circuit we can equally think of each state in the first register having its *relative* phase rotated by the specified amount. In other words, the $|0\rangle$ state acquires a relative phase of 0°, the $|1\rangle$ state acquires a relative phase of 90°, etc., which is exactly the state we wanted in Figure 8-9. Bingo!

It might take a few runs through the preceding argument to see how we've used `cont_u` and phase kickback to extract the eigenphase into the first register's frequency. But once we've done this, all we need to do is apply invQFT to the register and `READ` it to acquire the eigenphase we're after. Hence, the full circuit for implementing phase estimation is as shown in Figure 8-12.

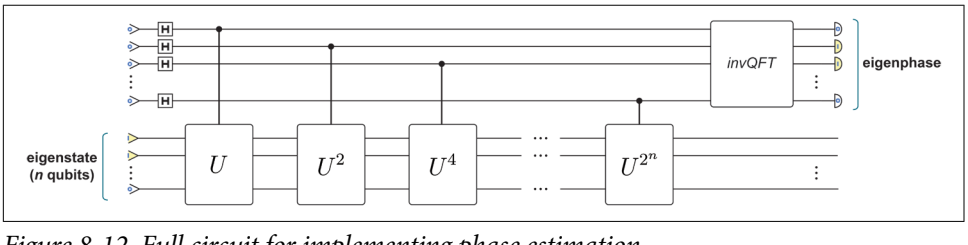


Figure 8-12. Full circuit for implementing phase estimation

We now see why we needed to be able to provide a subroutine for performing a *conditional* version of our QPU operation. Note also that the bottom register will remain in the same eigenstate at the end of the primitive. It's hopefully also now clear why (thanks to invQFT) the size of the upper output register limits the precision of the primitive.

How we perform the powers of `cont_u` can have a huge effect on the efficiency of phase estimation. Naively performing (for example) U^4 by calling `cont_u` four consecutive times is massively inefficient. For this reason, we may want to pass `phase_est()` a subroutine that can also efficiently return the n^{th} power of the QPU operation in question (i.e., `cont_u(n)`).

Conclusion

In this chapter we have explored a new QPU primitive, *phase estimation*. This primitive uses three previously introduced concepts (phase kickback, construction of controlled unitaries, and the invQFT primitive) to achieve a great feat: it can extract information that QPU operations encode in the *global* phases of a register. It does so by transforming the global phase information into relative phase information in a second quantum register, and then applying invQFT to extract that information in a `READ`able format. This operation will prove crucial for some of the machine-learning operations that we will encounter in Chapter 13.

PART III

QPU Applications

With some essential QPU primitives firmly in place, it's time to graduate from "How do QPUs work?" to "How can we use QPUs to do something useful?"

In Chapter 9, we will begin by demonstrating how useful data structures (rather than simple integers) can be represented and stored inside a QPU. Following this, in Chapter 10 we provide recipes to show how our arithmetic primitives from Chapter 5 can be used in a general class of applications that go under the name *quantum search*. We then move on in Chapter 11 to presenting applications in computer graphics, before introducing Peter Shor's famous factoring algorithm in Chapter 12 and finally turning to applications of our QPU primitives to machine learning in Chapter 13.

The search for useful QPU applications is an ongoing one, the focus of intense research by thousands of specialists around the globe. Hopefully this part of the book will prepare and encourage you to take the first steps toward discovering QPU applications in areas that may not even yet have been explored.

CHAPTER 9
Real Data

Fully fledged QPU applications are built to operate on genuine, unaccommodating data. Real data won't necessarily be as simple to represent as the basic integer inputs we've been satisfied with up until now. Thinking of how to represent more complex data within QPUs is thus well worth our effort, and a good data structure can be just as important as a good algorithm. This chapter sets out to answer two questions that we've previously sidestepped:

1. *How should we represent complicated data types in a QPU register?* A positive integer can be represented with simple binary encoding. What should we do with irrational, or even compound types of data, such as a vectors or matrices? This question takes on more depth when considering that superposition and relative phase might allow entirely *quantum* ways of encoding such data types.
2. *How can we read stored data into a QPU register?* So far we've been initializing our input registers by hand, using `WRITE` operations to manually set a register's qubits to binary integers of interest. If we're ever to employ quantum applications on large swathes of data we'll need to read that data into QPU registers from memory. This is a nontrivial requirement as we may want to initialize a QPU register with a superposition of values—something that conventional RAM isn't cut out for.

We'll start by addressing the first of these questions. As we describe QPU representations for increasingly complex types of data, we'll be led to introduce some truly *quantum* data structures and the concept of Quantum Random Access Memory (QRAM). QRAM is a crucial resource for many practical QPU applications.

In coming chapters we rely heavily on the data structures introduced here. For example, the so-called *amplitude encoding* we introduce for vector data is at the heart of every quantum machine-learning application mentioned in Chapter 13.

Noninteger Data

How can we encode noninteger numerical data in a QPU register? Two common methods for representing such values in binary are *fixed point* and *floating point*. Although a floating-point representation is more flexible (able to adapt to the range of values we need to express with a certain number of bits), given the premium we place on qubits and our desire for simplicity, fixed point is a more attractive starting point.

Fixed-point representation splits a register into two sections, one of which encodes the integer part of a number and the other of which encodes the fractional part. The integer part is expressed using a standard binary encoding, with higher-weight qubits representing increasing powers of two. However, in the fractional part of the register, qubits of *decreasing* weight represent *increasing* powers of $\frac{1}{2}$.

Fixed-point numbers are often described using *Q notation* (sadly, the Q is not for quantum). This helps take care of the ambiguity about where the fractional bits in a register end and the integer bits begin. The notation `Q`*n.m* denotes an *n*-bit register with *m* of its bits being fractional (and thus the remaning (*n* – *m*) being integer). We can, of course, use the same notation to specify how we use a QPU register for fixed-point encodings. For example, Figure 9-1 shows an eight-qubit QPU register encoding a value of 3.640625 in Q8.6 fixed-point encoding.

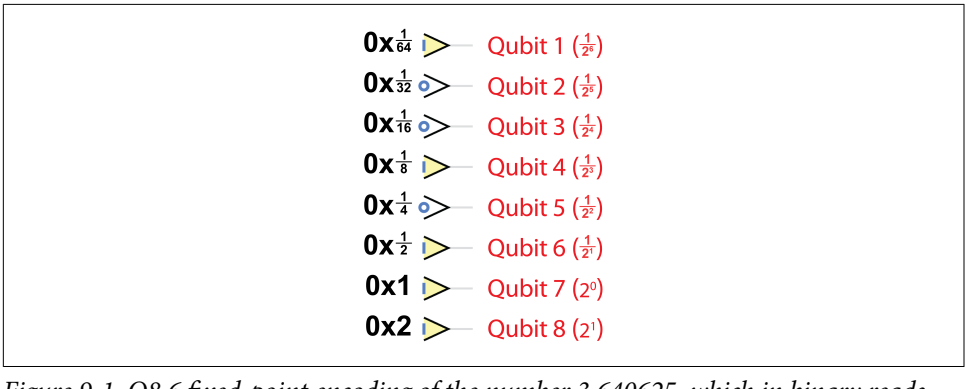


Figure 9-1. Q8.6 fixed-point encoding of the number 3.640625, which in binary reads 11101001

In the preceding example we managed to encode the chosen number precisely in fixed point because $3.640625 = 2^1 + 2^0 + \frac{1}{2^1} + \frac{1}{2^3} + \frac{1}{2^6}$ (how convenient!). Of course, we might not always get so lucky. Whereas increasing the number of bits in the integer side of a fixed-point register increases the *range* of integer values it can encode, increasing the number of bits in the fractional side increases the *accuracy* with which it can represent the fractional component of a number. The more qubits we have in

the fractional part, the more chance there is that some combination of $\frac{1}{2^1}, \frac{1}{2^2}, \frac{1}{2^3}, \ldots$ can accurately represent a given real number.

Although in the coming chapters we'll only point out when we use fixed-point encodings in passing, they're critical to being able to experiment with real data in small QPU registers and so well worth being aware of. When dealing with the various encodings that we have introduced, we must be diligent in keeping track of which particular encoding we're using for data in a given QPU register, so that we interpret the state of its qubits correctly.

Take care—operations using two's complement and fixed-point encodings can often be subject to *overflow*, where the result of a calculation is too large to be represented in a register. This effectively scrambles the output into a meaningless number. Sadly, the only real solution to overflow is to add more qubits to your registers.

QRAM

We can now represent various numerical values in our QPU registers, but how do we actually get these values into them? Initializing input data by hand gets old very quickly. What we really need is the ability to read values from memory, where a binary address allows us to locate stored values. A programmer interfaces with *conventional* Random Access Memory (RAM) through two registers: one initialized with a memory address, and the other uninitialized. Given these inputs, the RAM sets the second register to the binary contents stored at the address specified by the first, as shown in Figure 9-2.

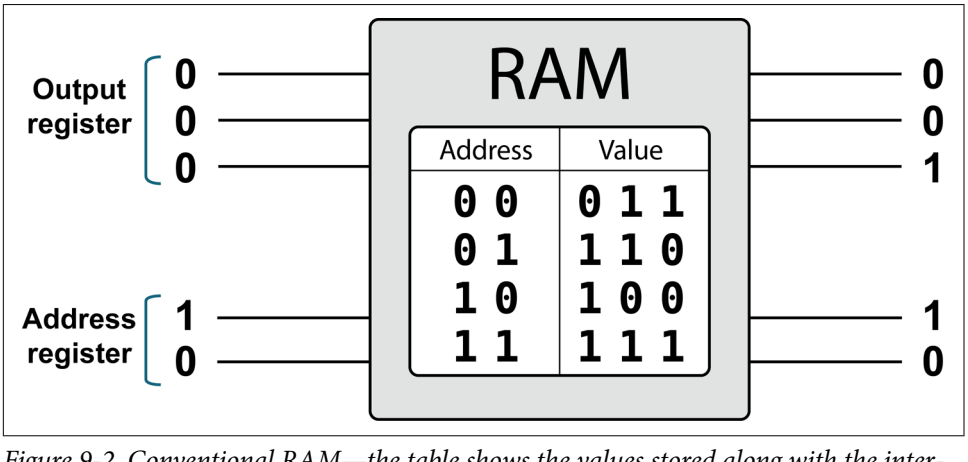


Figure 9-2. Conventional RAM—the table shows the values stored along with the interface used to access them

Can we use conventional RAM to store values for initializing our QPU registers? It's tempting to think so.

If we only want to initialize a QPU register with a single conventional value (whether in two's complement, fixed-point, or just simple binary encoding), RAM works just fine. We simply store the value in question within the RAM, and insert or retrieve it from our QPU register with `write()` and `read()` operations. Under the hood, this is precisely the restricted way in which our QCEngine JavaScript has been interfacing with our QPU registers so far.

For example, the sample code shown in Example 9-1, which takes an array `a` and implements `a[2] += 1;`, is implicitly drawing that array of values from RAM in order to initialize our QPU register. This is illustrated in Figure 9-3.

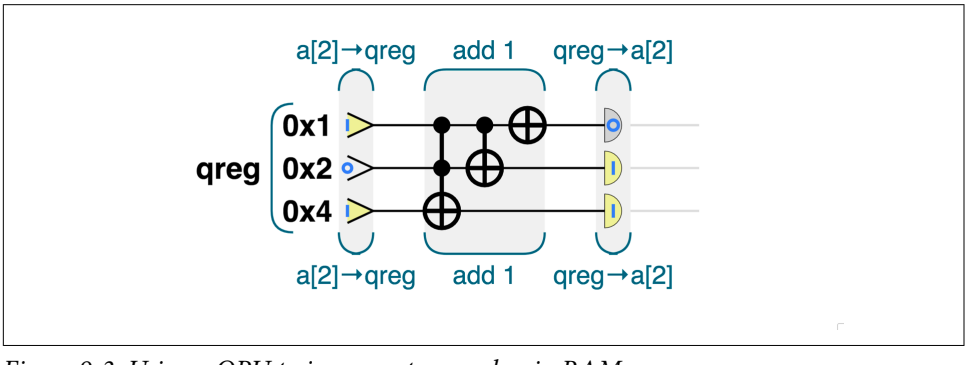


Figure 9-3. Using a QPU to increment a number in RAM

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=9-1*.

Example 9-1. Using a QPU to increment a number in RAM

```
var a = [4, 3, 5, 1];

qc.reset(3);
var qreg = qint.new(3, 'qreg');

qc.print(a);
increment(2, qreg);
qc.print(a);

function increment(index, qreg)
{
    qreg.write(a[index]);
    qreg.add(1);
```

```
    a[index] = qreg.read();
}
```

A point worth noting is that in this simple case, not only is conventional RAM being used to store the integer, but a conventional CPU is performing the array indexing to select and issue the QPU with the array value that we want.

Although using RAM in this way allows us to initialize QPU registers with simple binary values, it has a serious shortcoming. What if we want to initialize a QPU register in a *superposition* of stored values? For example, suppose our RAM stores a value 3 (`110`) at address `0x01`, and a 5 (`111`) at address `0x11`. How do we prepare an input register in a superposition of these two values?

With RAM and its clumsy conventional `write()` there's no way to achieve this. Like their vacuum-tube grandparents, QPUs are in need of a new piece of memory hardware—something fundamentally quantum in nature. Enter QRAM, which allows us to read and write data in a truly quantum way. There are already several ideas for how to physically build QRAM, but it's worth noting that history could very well repeat itself, and excitingly powerful QPUs may exist long before they are complemented with working QRAM hardware.

Let's be a little more precise about what QRAM actually does. Like conventional RAM, QRAM takes two registers as input: an *address* QPU register for specifying a memory address, and an *output* QPU register that returns the value stored at that address. Note that for QRAM these are both qubit registers. This means that we can specify a superposition of locations in the address register and consequently receive a superposition of the corresponding values in the output register, as shown in Figure 9-4.

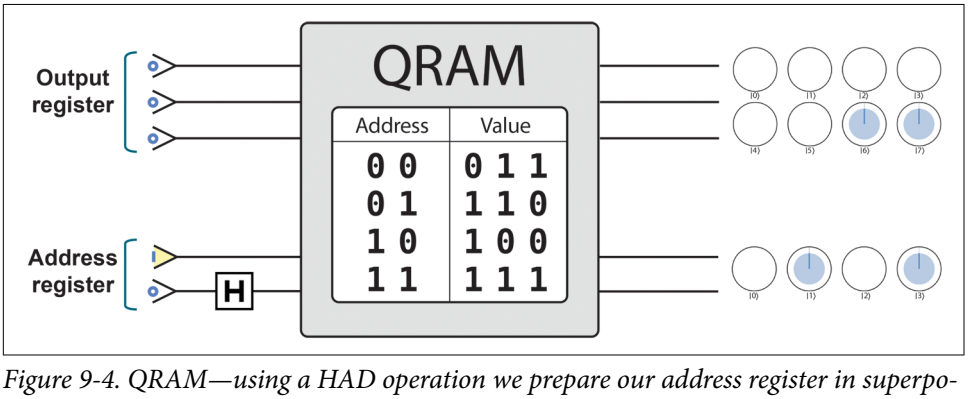


Figure 9-4. QRAM—using a HAD operation we prepare our address register in superposition, and we receive a similar superposition of stored values (shown in circle notation)

Thus, QRAM essentially allows us to read stored values in superposition. The precise amplitudes of the superposition we receive in the output register are determined by the superposition provided in the address register. Example 9-2 illustrates the difference by performing the same increment operation as Example 9-1, as shown in Figure 9-5, but using QRAM to access data instead of a QPU write/read. "A" denotes the register that provides the QRAM with an address (or superposition thereof). "D" denotes the register in which the QRAM returns a corresponding superposition of stored values (data).

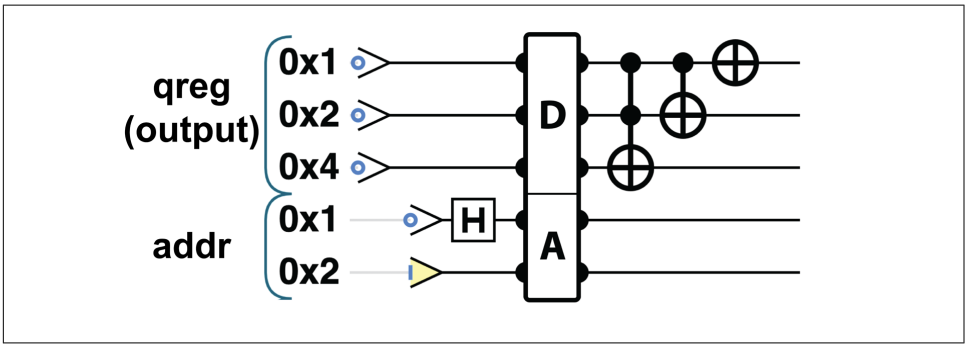


Figure 9-5. Using QRAM to perform the increment

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=9-2*.

Example 9-2. Using a QPU to increment a number from QRAM—the address register can be accessed in a superposition, resulting in the output register being a superposition of the stored values

```
var a = [4, 3, 5, 1];
var reg_qubits = 3;
qc.reset(2 + reg_qubits + qram_qubits());
var qreg = qint.new(3, 'qreg');
var addr = qint.new(2, 'addr');
var qram = qram_initialize(a, reg_qubits);

qreg.write(0);
addr.write(2);
addr.hadamard(0x1);

qram_load(addr, qreg);
qreg.add(1);
```

Can you *write* superpositions back into QRAM? That's not the purpose of it. QRAM allows us to *access* conventionally written digital values in superposition. A *persistent quantum memory* that could store superpositions indefinitely would be a whole different piece of hardware—one that might be even more challenging to build.

This description of QRAM might seem unsatisfyingly vague—just what *is* a piece of QRAM hardware? In this book we don't give a description of how QRAM can be built in practice (just as most C++ books don't give a detailed description of how RAM works). The code samples, such as Example 9-2, run using a simplified simulator model that mimics QRAM's behavior. However, examples of proposed QRAM technology do exist.[1]

Although QRAM will be a critical component in serious QPUs, like much quantum computing hardware, its implementation details are likely subject to change. What matters to us is the idea of a *fundamental resource* behaving in the manner illustrated in Figure 9-4, and the powerful applications that we can be build using it.

With QRAM at our disposal we can begin to build more sophisticated quantum data structures. Of particular interest are those allowing us to represent vector and matrix data.

Vector Encodings

Suppose we wanted to initialize a QPU register to represent a simple vector such as the one shown in Equation 9-1.

Equation 9-1. Example vector for initializing a QPU register

$$\vec{v} = [0, 1, 2, 3]$$

We'll often encounter data of this form in quantum machine-learning applications.

Perhaps the most obvious approach to encoding vector data is to represent each of its components in the state of a distinct QPU register, using an appropriate binary encoding. We'll refer to this (perhaps most obvious) approach as a *state encoding for vectors*. We could state-encode our preceding example vector into four two-qubit registers, as shown in Figure 9-6.

1 See, for example, Giovannetti et al., 2007, and Prakash, 2014.

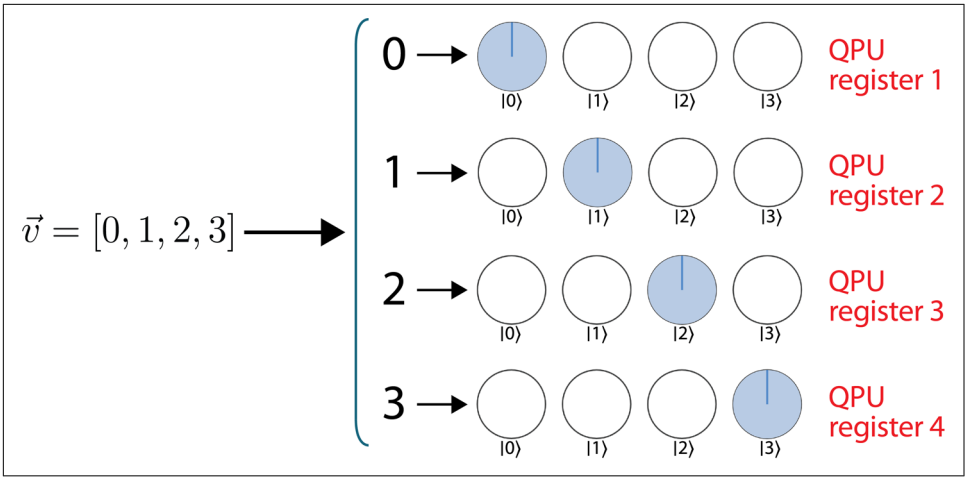


Figure 9-6. Using a state encoding for storing vector data in QPU registers

One problem with this naive state encoding is that it's pretty heavy on qubits—our QPU's scarcest resource. That said, state-encoding conventional vectors does benefit from not requiring any QRAM. We can simply store the vector components in standard RAM and use their individual values to dictate how we prepare each separate QPU register. But this advantage also belies vector state encoding's biggest *drawback*: storing vector data in such a conventional way prevents us from utilizing the unconventional abilities of our QPU. To exploit a QPU's power, we really want to be in the business of manipulating the relative phases of superpositions—something that's hard to do when each component of our vector is essentially treating our QPU as though it were a set of conventional binary registers!

Instead, let's get a little more quantum. Suppose we store the components of a vector in the superposition amplitudes of a *single* QPU register. Since a QPU register of n qubits can exist in a superposition having 2^n amplitudes (meaning we have 2^n circles to play with in circle notation), we can imagine encoding a vector with n components in a QPU register of ceil(log(n)) qubits.

For the example vector in Equation 9-1 this approach would require a two-qubit register—the idea being to find appropriate quantum circuitry to encode the vector data as shown in Figure 9-7.

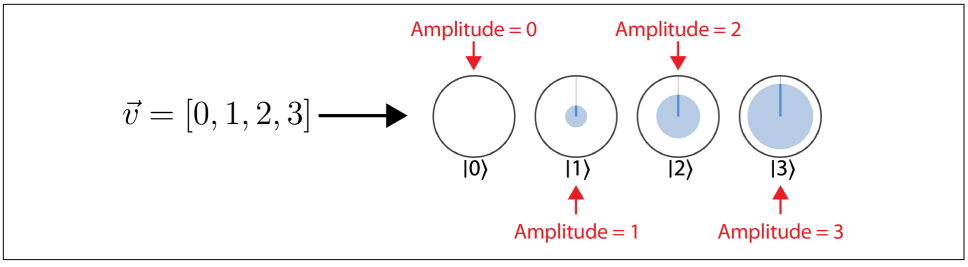
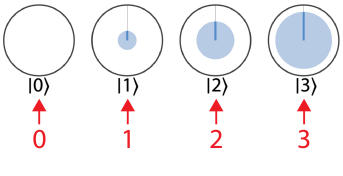
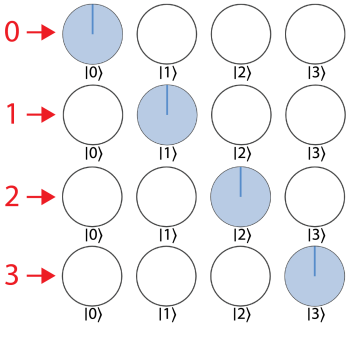
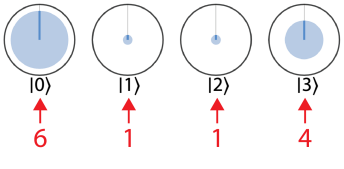
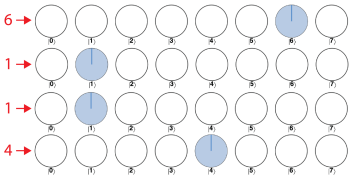
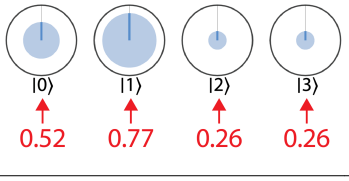
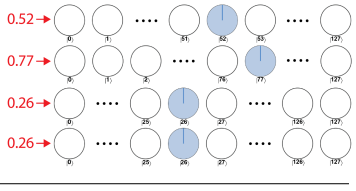


Figure 9-7. The basic idea of amplitude encoding for vectors

We call this uniquely quantum way of encoding vector data an *amplitude encoding for vector data*. It's critical to appreciate the difference between amplitude encoding and the more mundane *state encoding*. Table 9-1 compares these two encodings side by side for various example vector data. In the final example of state encoding, four seven-qubit registers would be needed, each using a Q7.7 fixed-point representation.

Table 9-1. The difference between amplitude and state encoding for vector data

Vector	Amplitude encoding	State encoding
`[0, 1, 2, 3]`		
`[6, 1, 1, 4]`		
`[0.52, 0.77, 0.26, 0.26]`		

We can produce amplitude-encoded vectors in QCEngine using a convenient function called `amplitude_encode()`. Example 9-3 takes a vector of values and a reference to a QPU register (which must be of sufficient size), and prepares that register in an amplitude encoding of the vector.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=9-3*.

Example 9-3. Preparing amplitude-encoded vectors in QCEngine

```
// We have ensured that our input vector has a length
// that is a power of two
var vector = [-1.0, 1.0, 1.0, 5.0, 5.0, 6.0, 6.0, 6.0];

// Create register of right size to amplitude-encode vector
var num_qubits = Math.log2(vector.length);
qc.reset(num_qubits);
var amp_enc_reg = qint.new(num_qubits, 'amp_enc_reg');

// Generate amplitude encoding in amp_enc_reg
amplitude_encode(vector, amp_enc_reg);
```

In this example, we simply provide our vector as a JavaScript array stored in conventional RAM—yet we've stated that amplitude encoding depends on QRAM. How does QCEngine manage to amplitude-encode when it only has access to your laptop's RAM? Although it's possible to generate circuits for amplitude encoding *without* QRAM, it's certainly not possible to do so in an efficient way. QCEngine provides us with a slow, but usable simulation of what we could achieve with access to QRAM.

Limitations of Amplitude Encoding

Amplitude encoding seems like a great idea—it uses fewer qubits and gives us a very quantum way of dealing with vector data. However, any application that leverages it comes with two important caveats.

Caveat 1: Beware of quantum outputs

You may have already spotted the first of these limitations: *quantum superpositions are generally unREADable*. Our old nemesis strikes again! If we spread the components of a vector across a quantum superposition, we can't read them out again. Naturally this isn't such a big deal when we're *inputting* vector data to some QPU program from memory—we presumably know all the components anyway. But very often QPU applications that take amplitude-encoded vector data as an input will also produce amplitude-encoded vector data as an output.

Using amplitude encoding therefore imposes a severe limitation on our ability to `READ` outputs from applications. Fortunately, though, we can still often extract useful information from an amplitude-encoded output. We'll see in later chapters that although we can't learn individual components, we can still learn *global* properties of vectors encoded in this way. Nevertheless, there's no free lunch with amplitude encoding, and its successful use requires care and ingenuity.

If you read about quantum machine-learning applications that solve some conventional machine-learning problem with a fantastic speedup, always be sure to check whether they return a *quantum* output. Quantum outputs, such as amplitude-encoded vectors, limit the usage of applications, and require additional specification of how to extract practical, useful results.

Caveat 2: The requirement for normalized vectors

The second caveat to amplitude encoding is hidden in Table 9-1. Take a closer look at the amplitude encodings of the first two vectors in that table: `[0,1,2,3]` and `[6,1,1,4]`. Can the amplitudes of a two-qubit QPU register *really* take the values `[0,1,2,3]` or the values `[6,1,1,4]`? Unfortunately not. In earlier chapters we've generally eschewed numerical discussion of magnitudes and relative phases in favor of our more readily intuitive circle notation. Although building intuition, this has limited your exposure to an important numerical rule about state amplitudes: *the squares of a register's amplitudes must sum to unity*. This requirement, known as *normalization*, makes sense when we recall that the squared magnitudes in a register are the probabilities of reading different outcomes. Since one outcome must occur—these probabilities, and hence the squares of each amplitude, must sum to one. It's easy to forget about normalization from the comfort of circle notation, but it places an important restriction on what vector data we can amplitude-encode. The laws of physics forbid us from ever creating a QPU register that's in a superposition with amplitudes `[0,1,2,3]` or `[6,1,1,4]`.

To amplitude-encode the two troublesome vectors from Table 9-1 we would first need to normalize them, dividing each component by the summed squares of all components. For example, to amplitude-encode the vector `[0,1,2,3]`, we first divide by `3.74` to yield a normalized vector `[0.00, 0.27, 0.53, 0.80]`—which is now suitable for encoding in the amplitudes of a superposition.

Does normalizing vector data have any adverse effects? It seems as though we've totally altered our data! Normalization actually leaves most of the important information intact (geometrically it rescales the length of a vector, while leaving its direction unchanged). Whether normalized data is just as good as the real thing depends on the requirements of the particular QPU application we plan to use it in. Keep in mind

that we can, of course, always keep track of the normalization factor's numerical value in another register.

Amplitude Encoding and Circle Notation

As we start thinking more concretely about the numerical values of register amplitudes it becomes useful to remind ourselves how amplitudes are represented in circle notation, and highlight a potential pitfall. The filled-in areas in circle notation represent the squared *magnitudes* of the (potentially complex) amplitudes of a quantum state. In cases like amplitude encoding, where we want these amplitudes to assume real-valued vector components, this means that the filled-in areas are given by the *square* of the associated vector component, rather than simply the vector component itself. Figure 9-8 shows how we should properly interpret the circle notation representation of the vector `[0,1,2,3]` after normalization.

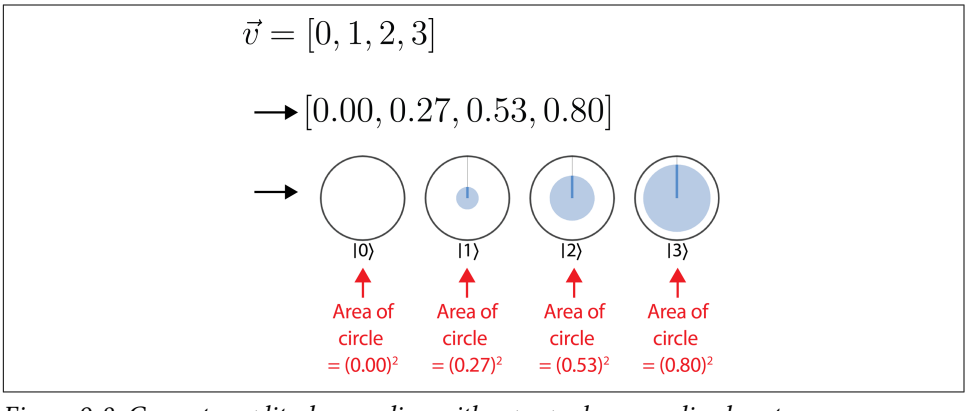


Figure 9-8. Correct amplitude encoding with a properly normalized vector

When estimating numerical values of amplitudes from circle notation, don't forget that the *square* of a state's magnitude (and hence amplitude for real-valued cases) determines the filled area of its circle. Take care to account for the square!

You now know enough about amplitude-encoded vectors to make sense of the QPU applications we'll introduce. But for many applications, especially those from quantum machine learning, we need to go one step further and use our QPU to manipulate not just vectors, but whole *matrices* of data. How should we encode two-dimensional arrays of numbers?

Although the example vectors we've used so far have all involved real components, since the relative phases of a superposition allow its amplitudes to, in general, be complex numbers, it's worth noting that amplitude encoding can easily represent (normalized) complex vectors in a QPU register. This is why amplitude encoding is not referred to as *magnitude* encoding—we can make use of the full amplitude in a superposition to also encode complex values, though we won't do so explicitly in this chapter.

Matrix Encodings

The most obvious way of encoding an $m \times n$ matrix of values would be to employ m QPU registers, each of length $\log_2(n)$, to amplitude-encode each row of the matrix as if it were a vector. Although this is certainly one way of getting matrix data into our QPU, it doesn't necessarily represent the data *as a matrix* very well. For example, it's not at all obvious how this approach would enable fundamental operations such as transposition, or matrix multiplication with amplitude-encoded vectors in other QPU registers.

The best way for us to encode a matrix in QPU register(s) depends on precisely how we want to later use that matrix within QPU applications, and—at least at the time of writing—there are several kinds of matrix encodings that are commonly utilized.

There is, however, one universal requirement we'll have for encoding matrices. Since acting (multiplying) matrices on vectors of data is so common, and since vector data is encoded in QPU registers, it makes sense to think about encoding matrices as QPU operations able to act on vector-holding registers. Representing a matrix as a QPU operation in a meaningful way is a difficult task, and each existing method for achieving this comes with its own important pros and cons. We'll focus on one very popular approach known as *quantum simulation*. But before we do, let's be clear on exactly what it aims to achieve.

How Can a QPU Operation Represent a Matrix?

What does it even mean to say that a QPU operation correctly *represents* a particular matrix of data? Suppose that we worked out how every possible vector that this matrix might act on would be encoded in a QPU register (according to some scheme such as amplitude encoding). If a QPU operation acting on these registers resulted in output registers encoding precisely the vectors we would expect acting on the *matrix* to give, we would certainly feel confident that the QPU operation captured the behavior of our matrix.

When introducing phase estimation in Chapter 8, we noted that a QPU operation is entirely characterized by its eigenstates and eigenphases. Similarly, the eigendecomposition of a matrix characterizes it completely. So, a simpler way to convince

ourselves that a QPU operation faithfully represents a matrix of data is if they both have the same eigendecomposition. By this we mean that the eigenstates of the QPU operation are (the amplitude encodings of) the eigenvectors of the original matrix, and its eigenphases are related to the matrix's eigenvalues. If this is the case, we can be confident that the QPU operation implements the desired matrix's action on amplitude-encoded vectors. Manipulating or studying a QPU operation sharing a matrix's eigendecomposition allows us to reliably answer questions about the encoded matrix.

The term *eigendecomposition* refers to the set of eigenvalues and eigenvectors of a matrix. We can also apply the term to QPU operations, where we mean the set of eigenstates and eigenphases associated with that operation.

Suppose we identify a QPU operation having the same eigendecomposition as a matrix we want to encode. Is that all we need? Almost. When we ask for a *representation of a matrix as a QPU operation*, we don't just want some abstract mathematical description of a suitable QPU operation. Pragmatically speaking, we want a prescription for how to actually perform that operation in terms of the simple single- and multi-qubit operations we introduced in Chapters 2 and 3. Furthermore, we want this prescription to be *efficient*, in the sense that we don't need so many such operations that including matrix data in a QPU application renders it uselessly slow. So for our purposes we can be a little bit more concrete about what we would like:

> A good matrix representation is a procedure for associating a matrix with a QPU operation that can be efficiently implemented through basic single- and multi-qubit operations.

For certain classes of matrices, the procedure of *quantum simulation* provides good matrix representations.

Quantum Simulation

Quantum simulation is actually a catch-all term for a whole class of procedures that can find efficiently implementable QPU operations for representing *Hermitian* matrices.

Hermitian matrices are those for which $H = H^{\dagger}$, where † denotes the *adjoint*. The adjoint is found by taking the transpose and complex conjugate of a (potentially complex) matrix.

As desired, quantum simulation techniques provide a QPU operation having the same eigendecomposition as the original Hermitian matrix. Each of the many methods for performing quantum simulation produce circuits with different resource requirements, and may even enforce different additional constraints on the kinds of matrices that can be represented. However, at a minimum, all quantum simulation techniques require the encoded matrix to be Hermitian.

But doesn't requiring a matrix of real data to be Hermitian render quantum simulation techniques uselessly niche? Well, it turns out that only being able to represent Hermitian matrices is not as restrictive as it might sound. Encoding an $m \times n$ non-Hermitian matrix X can be achieved by constructing a larger $2m \times 2n$ Hermitian matrix H as follows:

$$H = \begin{bmatrix} \mathbf{0} & X \\ X^\dagger & \mathbf{0} \end{bmatrix}$$

Where the **0**s on the diagonal represent $m \times n$ blocks of zeros. This constant one-off additional overhead of creating a larger matrix is usually relatively insignificant.

We'll outline the general high-level approach taken by many different quantum simulation techniques, giving a little more detail on one particular approach as an example. This will involve more mathematics than we've dealt with so far, but only the kind of linear algebra that's par for the course when dealing with matrices.

The basic idea

Despite relying on circle notation, we've noted that the full quantum mechanical description of a QPU register's state is a complex-valued vector. It turns out that the full quantum mechanical description of a QPU operation is a matrix. This might make our goal of encoding matrices as QPU operations sound simple. If, deep down, QPU operations are described by matrices, then just find one that has the same matrix as the data to be encoded! Unfortunately, only a subset of matrices correspond to valid (constructable) QPU operations. In particular, valid QPU operations are described by *unitary* matrices.

A *unitary* matrix U is one for which $UU^\dagger = \mathbb{1}$, where $\mathbb{1}$ is an identity matrix (having ones along the diagonal and zeros elsewhere) of the same size as U. QPU operations need to be described by unitary matrices as this ensures that the circuits realizing them will be reversible (a requirement we noted in Chapter 5).

The good news is that given a Hermitian matrix H, it's possible to construct an associated unitary matrix U through exponentiation: $U = \exp(-iHt)$ is unitary if H is Her-

mitian. Quantum simulation techniques make use of this fact (hence why they're restricted to representing Hermitian matrices). The t appearing in the exponent is the *time* that we apply the QPU operation $U = \exp(-iHt)$ for. We can, for our purposes, consider choosing t to be a hardware implementation detail that we gloss over for simplicity of exposition.

Quantum simulation's task is therefore to efficiently provide a circuit performing this exponentiation of H.

Although we're using it to encode matrix data, quantum simulation is so called because it's primarily used as a technique for *simulating* the behavior of quantum mechanical objects (for example, in molecular or materials simulations). When simulating quantum objects, a certain Hermitian matrix (known in physics as the Hamiltonian) mathematically describes the simulation to be performed, and a QPU operation $\exp(-iHt)$ predicts how the quantum object evolves over time. This technique is heavily used in QPU algorithms for quantum chemistry, see "The Promise of Quantum Simulation" on page 304 for more information.

For some particularly simple Hermitian matrices H it turns out that finding a set of simple QPU operations to implement $\exp(-iHt)$ is relatively easy. For example, if H only has elements on its diagonal, or would only act on a very small number of qubits, then a circuit can easily be found.

It's unlikely, however, that a Hermitian data matrix will satisfy either of these simplicity requirements out of the box, and quantum simulation gives us a way to break down such difficult-to-encode matrices into a number easier-to-encode ones. In the next section we'll outline how this works. Although we don't provide any detailed algorithms for quantum simulation, our high-level description at least helps illustrate the important limitations of these techniques.

How it works

The many approaches to quantum simulation follow a similar set of steps. Given a Hermitian matrix H we proceed as follows:

1. *Deconstruct*. We find a way to split H into a *sum* of some number (n) of other *simpler* Hermitian matrices, $H = H_1 + \ldots H_n$. These matrices $H_1, \ldots, H_n$ are simpler in the sense that they are easier to simulate efficiently in the manner mentioned in the previous section.
2. *Simulate components*. We then efficiently find quantum circuits (in terms of fundamental QPU operations) for these simpler component matrices.

3. *Reconstruction*. We reconstruct a circuit for implementing the full matrix H of interest from the smaller quantum circuits found for its deconstructed components.

For this plan of action to work the two steps that need most clarification are: finding a way to break down a Hermitian matrix into a summation of easy-to-simulate pieces (step 1) and showing how to piece together simulations of each of these smaller pieces into a full-blown simulation of H (step 3). Quantum simulation approaches differ in how they achieve this. Here we summarize one group of methods, known as *product formula methods*.

It's actually easiest to start at the end and first explain how product formula methods perform the final reconstruction of H (step 3, from the preceding list), so let's do that.

Reconstruction

Suppose that we do find some way to write $H = H_1 + \ldots + H_n$, with $H_1, \ldots, H_n$ being Hermitian matrices for which we can easily find QPU operations. If this is the case, we can reconstruct H itself thanks to a mathematical relationship known as the *Lie product formula*. This formula allows the unitary matrix $U = \exp(-iHt)$ to be approximated by performing each of the component QPU operations

$$U_1 = \exp\left(-iH_1\delta t\right), \ldots, U_n = \exp\left(-iH_n\delta t\right)$$

in sequence for very short times δt, and then repeating this whole procedure some number of times m, as illustrated in Figure 9-9.

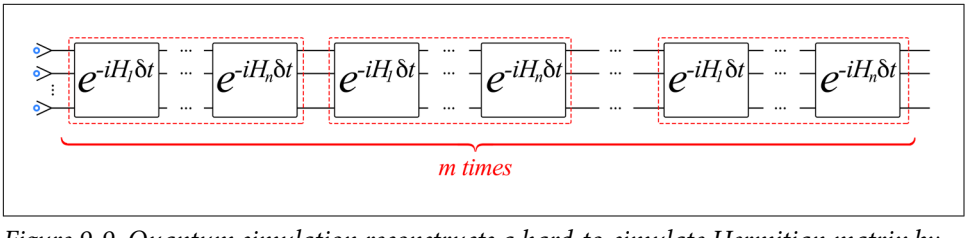


Figure 9-9. Quantum simulation reconstructs a hard-to-simulate Hermitian matrix by repeatedly simulating a sequence of easier-to-simulate Hermitian matrices

Crucially, the Lie product formula shows that *if* we can deconstruct H into matrices having efficient circuits, then we can also approximate $U = \exp(-iHt)$ with an efficient runtime.

Deconstructing H

That takes care of step 3, but how do we deconstruct our matrix as a sum of easy-to-simulate Hermitian matrices in the first place?

Different approaches from the product formula class of quantum simulation techniques perform this deconstruction in different ways. All approaches are quite mathematically involved, and place differing additional requirements on H. For example, one approach that is suitable if H is sparse (and we can efficiently access these sparse components) is to consider H to be an adjacency matrix for a graph. By solving a certain coloring problem on this graph, the different colors we identify group together elements of H that form the easy-to-simulate $H_1, \ldots H_n$ deconstructing it.

Here we use the term *graph* in the mathematical sense, meaning a structure containing a series of vertices connected by edges relating them. A *coloring problem* on a graph is the task of associating one of several available colors to each vertex subject to the constraint that if two vertices are directly connected by an edge, they cannot share the same color. How deconstructing Hermitian matrices relates to graph coloring is not obvious, and results from the problem's underlying mathematical structure.

The cost of quantum simulation

This summary of product formula approaches to quantum simulation hopefully gives you a feel for the lengths one must go to represent matrix data as QPU operations. As we previously mentioned, other quantum simulation approaches also exist, many of which exhibit better performance through either needing smaller circuits or reduced access to the matrix to be encoded. In Table 9-2 we compare the runtimes of some common quantum simulation techniques. Here "runtime" refers the size of the circuit a method produces for simulating a matrix (where circuit size is the required number of fundamental QPU operations). d is a measure of the matrix's sparsity (the maximum number of nonzero elements per row), and ϵ is a measure of the desired precision of the representation.[2]

Table 9-2. Runtimes of selected quantum simulation techniques

Technique	Circuit runtime
Product formula[a]	$O\left(d^4\right)$
Quantum walks	$O(d/\sqrt{\epsilon})$
Quantum signal processing	$O\left(d + \frac{\log(1/\epsilon)}{\log\log(1/\epsilon)}\right)$

[a] It is also possible to slightly improve the runtime of product formula quantum simulation techniques through the use of "higher order" approximations to the Lie product formula used within the approach.

2 Note that these runtimes are also dependent on some other important parameters, like the input matrix and quantum simulation technique we might use. We have excluded these for simplicity in this overview.

CHAPTER 10
Quantum Search

In Chapter 6 we saw how the amplitude amplification (AA) primitive changes differences in phases within a register into detectable variations in magnitude. Recall that when introducing AA, we assumed that applications would provide a subroutine to flip the phases of values in our QPU register. As a simplistic example, we used the `flip` circuit as a placeholder, which simply flipped the phase of a single known register value. In this chapter we will look in detail at several techniques for flipping phases in a quantum state based on the result of nontrivial logic.

Quantum Search (QS) is a particular technique for modifying the `flip` subroutine such that AA allows us to reliably `READ` solutions from a QPU register for a certain class of problems. In other words, QS is really just an application of AA, formed by providing an all-important subroutine[1] marking solutions to a certain class of problems in a register's phases.

The class of problems that QS allows us to solve is those that repeatedly evaluate a subroutine giving a yes/no answer. The yes/no answer of this subroutine is, generally, the output of a conventional boolean logic statement.[2] One obvious problem that can be cast in this form is searching through a database for a specific value. We simply imagine a boolean function that returns a `1` if and only if an input is the database element we're searching for. This was in fact the prototypical use of Quantum Search, and is known, after its discoverer, as *Grover's search algorithm*. By applying the Quantum Search technique, Grover's search algorithm can find an element in a database using only $O(\sqrt{N})$ database queries, whereas conventionally $O(N)$ would be required.

1 In the literature, a function that flips phases according to some logic function is referred to as an *oracle*, in a similar sense to how the term is used in conventional computer science. We opt for slightly less intimidating terminology here, but more thoroughly relate this to prevalent technical language in Chapter 14.

2 We provide a more detailed description of this class of problems at the end of this chapter and in Chapter 14.

However, this assumes an unstructured database—a rare occurrence in reality—and faces substantial practical implementation obstacles.

Although Grover's search algorithm is the best known example of a Quantum Search application, there are many other applications that can use QS as a subroutine to speed up performance. These range from applications in artificial intelligence to software verification.

The piece of the puzzle that we're missing is just how Quantum Search allows us to find subroutines encoding the output of any boolean statement in a QPU register's phases (whether we use this for Grover's database search or other QS applications). Once we know how to do this, AA takes us the rest of the way. To see how we can build such subroutines, we'll need a sophisticated set of tools for manipulating QPU register phases—a repertoire of techniques we term *phase logic*. In the rest of this chapter, we outline phase logic and show how QS can leverage it. At the end of the chapter, we summarize a general recipe for applying QS techniques to various conventional problems.

Phase Logic

In Chapter 5, we introduced a form of quantum logic—i.e., a way of performing logic functions that's compatible with quantum superpositions. However, these logic operations used register values with non-zero *magnitudes* as inputs (e.g. $|2\rangle$ or $|5\rangle$), and similarly would output results as register values (possibly in scratch qubits).

In contrast, the quantum phase logic we need for Quantum Search should output the results of logic operations in the *relative phases* of these registers.

More specifically, to perform phase logic we seek QPU circuits that can achieve *quantum phase logic*, which represents a given logic operation (such as an `AND`, `OR`, etc.) by flipping the phases of values in a register for which the operation would return a `1` value.

The utility of this definition requires a little explaining. The idea is that we feed the phase-logic circuit a state (which may be in superposition) and the circuit flips the relative phases of all inputs that would satisfy the logic operation it represents. If the state fed in is *not* in superposition, this phase flip will simply amount to an unusable global phase, but when a superposition is used, the circuit encodes information in relative phases, which we can then access using the amplitude amplification primitive.

The expression *satisfy* is often used to describe inputs to a logic operation (or a collection of such operations forming a logical *statement*) that produce a 1 output. In this terminology, phase logic flips the phases of all QPU register values that *satisfy* the logic operation in question.

We've already seen one such phase-manipulating operation: the PHASE operation itself! The action of PHASE was shown in Figure 5-13. When acting on a single qubit, it simply writes the logical value of the qubit into its phase (only flipping the phase of the $|1\rangle$ value; i.e., when the output is 1). Although so far we've simply thought of PHASE as a tool for rotating the relative phases of qubits, it satisfies our quantum phase logic definition, and we can also interpret it as a phase-based logic operation.

The difference between binary logic and phase logic is important to understand and a potential point of confusion. Here's a summary:

Conventional binary logic
: Applies logic gates to an input, producing an output

Quantum magnitude logic
: Applies logic gates to a *superposition* of inputs, producing a *superposition* of outputs

Quantum phase logic
: Flips the phase of every input value that would produce a 1 as output; this works when the register is in superposition too

It's perhaps easier to grasp the action of phase logic by seeing some examples in circle notation. Figure 10-1 illustrates how phase-logic versions of OR, NOR, XOR, AND, and NAND operations affect a uniform superposition of a two-qubit register.

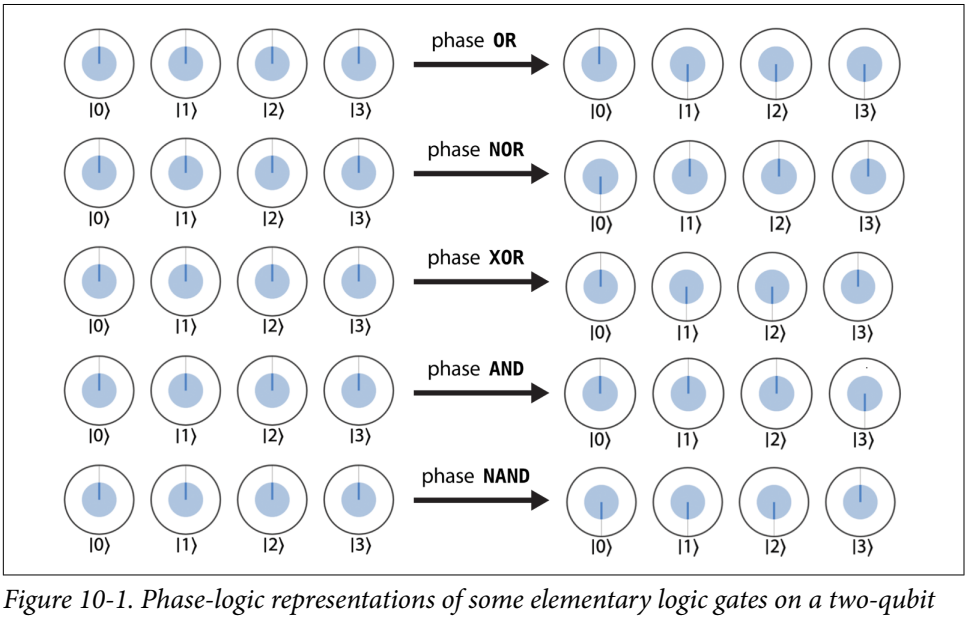


Figure 10-1. Phase-logic representations of some elementary logic gates on a two-qubit register

In Figure 10-1 we have chosen to portray the action of these phase-logic operations on superpositions of the register, but the gates will work regardless of whether the register contains a superposition or a single value. For example, a conventional binary logic `XOR` outputs a value 1 for inputs `10` and `01`. You can check in Figure 10-1 that only the phases of $|1\rangle$ and $|2\rangle$ have been flipped.

Phase logic is fundamentally different from any kind of conventional logic—the results of the logic operations are now hidden in un`READ`able phases. But the advantage is that by flipping the phases in a superposition we have been able to mark *multiple* solutions in a *single* register! Moreover, although solutions held in a superposition are normally unobtainable, we already know that by using this phase logic as the `flip` subroutine in amplitude amplification, we *can* produce `READ`able results.

Building Elementary Phase-Logic Operations

Now that we know what we want phase-logic circuits to achieve, how do we actually build phase-logic gates such as those alluded to in Figure 10-1 from fundamental QPU operations?

Figure 10-2 shows circuits implementing some elementary phase-logic operations. Note that (as was the case in Chapter 5) some of these operations make use of an extra scratch qubit. In the case of phase logic, any scratch qubits will always be initialized[3] in the state $|-\rangle$. It is important to note that this scratch qubit will *not* become entangled with our input register, and hence it is not necessary to uncompute the entire phase-logic gate. This is because the scratch qubits implement the phase-logic gates using the *phase-kickback* trick introduced in Chapter 3.

It's crucial to keep in mind that *input* values to these phase-logic implementations are encoded in the states of QPU registers (e.g. $|2\rangle$ or $|5\rangle$), but *output* values are encoded in the *relative phases*. Don't let the name *phase logic* trick you into thinking that these implementations also take phase values as inputs!

3 For example, the phase-logic `XOR` in Figure 10-2 uses a scratch qubit, which is prepared in the $|-\rangle$ state using a `NOT` and a `HAD` operation.

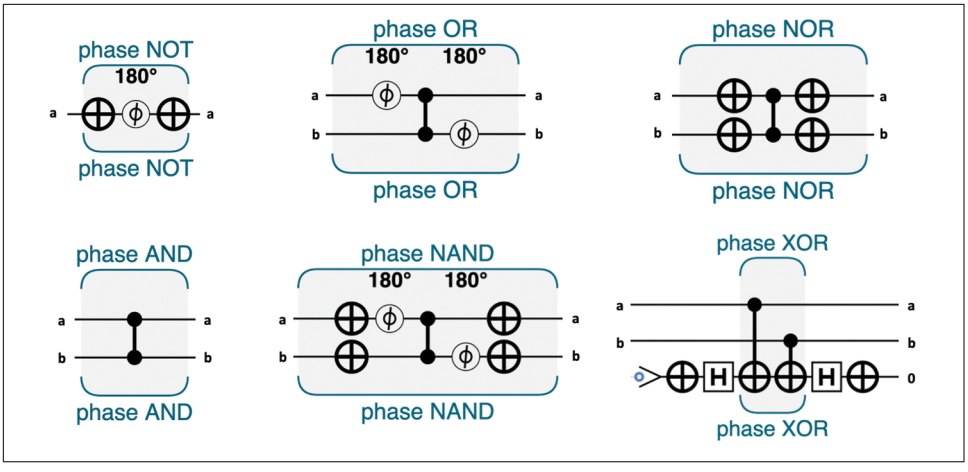


Figure 10-2. QPU operations implementing phase logic

Building Complex Phase-Logic Statements

The kind of logic we want to explore with QS will concatenate together many different elementary logic operations. How do we find QPU circuits for such full-blown, composite phase-logic statements? We've carefully cautioned that the implementations in Figure 10-2 output phases but require magnitude-value inputs. So we can't simply link together these elementary phase-logic operations to form more complex statements; their inputs and outputs aren't compatible.

Luckily, there's a sneaky trick: we take the full statement that we want to implement with phase logic and perform all but the *final* elementary logic operation from the statement using magnitude-based quantum logic, of the kind we saw in Chapter 5. This will output the values from the statement's penultimate operation encoded in QPU register magnitudes. We then feed this into a *phase-logic* implementation of the statement's final remaining logic operation (using one of the circuits from Figure 10-2). Voilà! We have the final output from the whole statement encoded in phases.

To see this trick in action, suppose that we want to evaluate the logical statement `(a OR NOT b) AND c` (involving the three boolean variables `a`, `b`, and `c`) in phase logic. The conventional logic gates for this statement are shown in Figure 10-3.

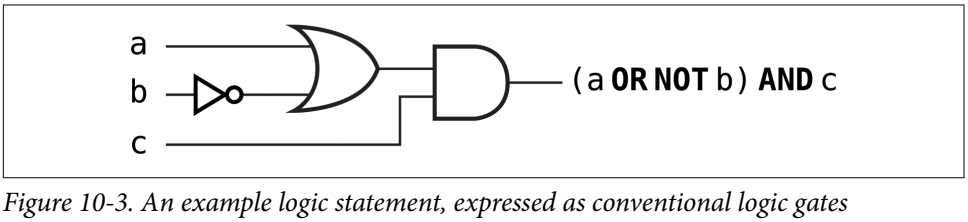


Figure 10-3. An example logic statement, expressed as conventional logic gates

For a phase-logic representation of this statement, we want to end up with a QPU register in a uniform superposition with phases flipped for all input values satisfying the statement.

Our plan is to use magnitude-based quantum logic for the `(a OR NOT b)` part of the statement (all but the last operation), and then use a phase-logic circuit to `AND` this result with `c`—yielding the statement's final output in register phases. The circuit in Figure 10-4 shows how this looks in terms of QPU operations.[4]

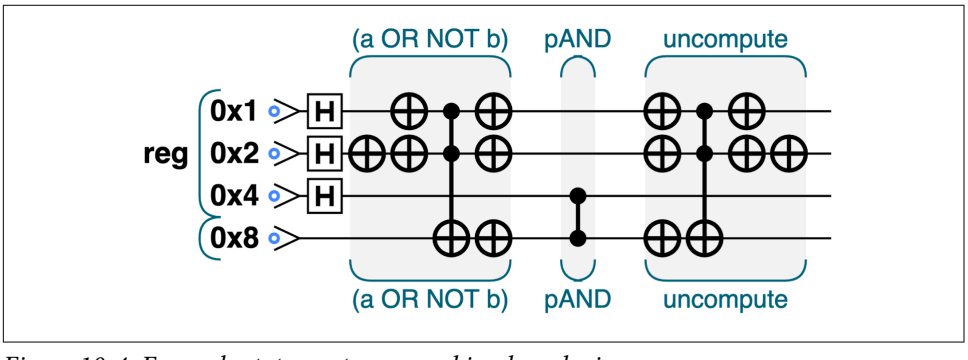


Figure 10-4. Example statement expressed in phase logic

Note that we also needed to include a scratch qubit for our binary-value logic operations. The result of `(a OR NOT b)` is written to the scratch qubit, and our phase-logic `AND` operation (which we denote `pAND`) is then performed between this and qubit `c`. We finish by uncomputing this scratch qubit to return it to its initial, unentangled $|0\rangle$ state.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=10-1*.

Example 10-1. Encoding example in phase logic

```
qc.reset(4);
var reg = qint.new(4, 'reg');

qc.write(0);
reg.hadamard();

// (a OR NOT b)
qc.not(2);
```

4 Note that there are some redundant operations in this circuit (e.g., canceling `NOT` operations). We have kept these for clarity.

```
bit_or(1,2,8);

// pAND
phase_and(4|8);

// uncompute
inv_bit_or(1,2,8);
qc.not(2);

// Logic Definitions

// binary logic OR using ancilla as output
function bit_or(q1,q2,out) {
    qc.not(q1|q2);
    qc.cnot(out,q1|q2);
    qc.not(q1|q2|out);
}

// reversed binary logic OR (to uncompute)
function inv_bit_or(q1,q2,out) {
    qc.not(q1|q2|out);
    qc.cnot(out,q1|q2);
    qc.not(q1|q2);
}

// Phase-logic AND (pAND)
function phase_and(qubits) {
    qc.cz(qubits);
}
```

Running the preceding sample code, you should find that the circuit flips the phases of values |4⟩, |5⟩, and |7⟩, as shown in Figure 10-5.

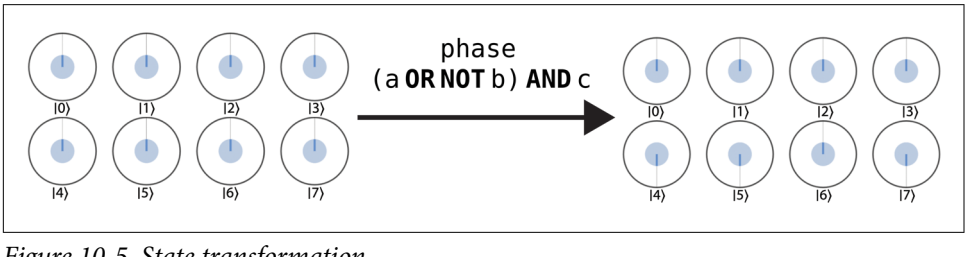


Figure 10-5. State transformation

These states encode the logical assignments `(a=0, b=0, c=1)`, `(a=1, b=0, c=1)`, and `(a=1, b=1, c=1)`, respectively, which are the only logic inputs satisfying the original boolean statement illustrated in Figure 10-3.

Solving Logic Puzzles

With our newfound ability to mark the phases of values satisfying boolean statements, we can use AA to tackle the problem of *boolean satisfiability*. Boolean satisfiability is the problem of determining whether input values exist satisfying a given boolean statement—precisely what we've learned to achieve with phase logic!

Boolean satisfiability is an important problem in computer science[5] and has applications such as model checking, planning in artificial intelligence, and software verification. To understand this problem (and how to solve it), we'll take a look at how QPUs help us with a less financially rewarding but much more fun application of the boolean satisfiability problem: solving logic puzzles!

Of Kittens and Tigers

On an island far, far away, there once lived a princess who desperately wanted a kitten for her birthday. Her father the king, while not opposed to this idea, wanted to make sure that his daughter took the decision of having a pet seriously, and so gave her a riddle for her birthday instead (Figure 10-6).[6]

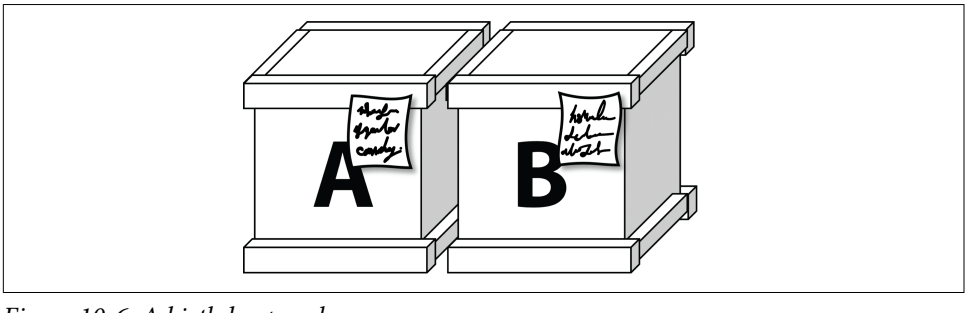


Figure 10-6. A birthday puzzle

On her big day, the princess received two boxes, and was allowed to open at most one. Each box might contain her coveted kitten, but might also contain a vicious tiger. Fortunately, the boxes came labeled as follows:

Label on box A

At least one of these boxes contains a kitten.

5 Boolean satisfiability was the first problem proven to be NP-complete. *N*-SAT, the boolean satisfiability problem for boolean statements containing clauses of *N* literals, is NP-complete when $N > 2$. In Chapter 14 we give more information about basic computational complexity classes, as well as references to more in-depth material.

6 Adaptation of a logic puzzle from the book *The Lady or the Tiger?* by Raymond Smullyan.

Label on box B
    The other box contains a tiger.

"That's easy!" thought the princess, and quickly told her father the solution.

"There's a twist," added her father, having known that such a simple puzzle would be far too easy for her. "The notes on the boxes are either both true, or they're both false."

"Oh," said the princess. After a brief pause, she ran to her workshop and quickly wired up a circuit. A moment later she returned with a device to show her father. The circuit had two inputs, as shown in Figure 10-7.

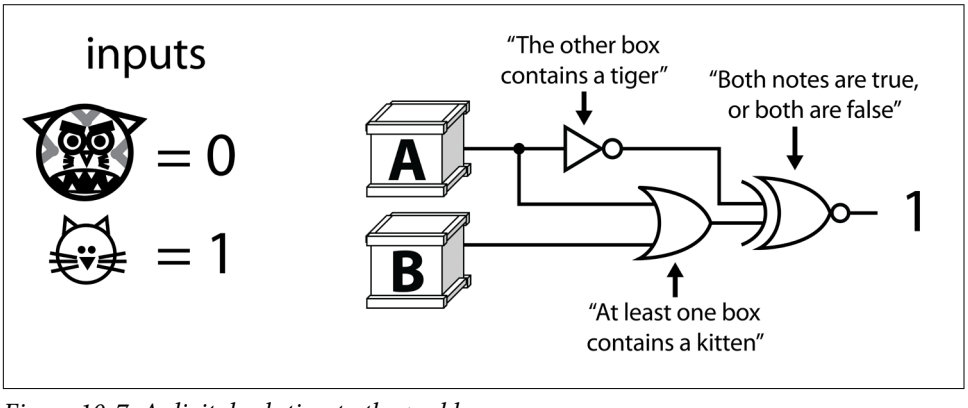


Figure 10-7. A digital solution to the problem

"I've set it up so that 0 means tiger and 1 means kitten", she proudly declared. "If you input a possibility for what's in each box, you'll get a 1 output only if the possibility satisfies all the conditions."

For each of the three conditions (the notes on the two boxes plus the extra rule her father had added), the princess had included a logic gate in her circuit:

- For the note on box A, she used an `OR` gate, indicating that this constraint would only be satisfied if box A *or* box B contained a kitten.
- For the note on box B, she used a `NOT` gate, indicating that this constraint would only be satisfied if box A did *not* contain a kitten.
- Finally, for her father's twist, she added an `XNOR` gate to the end, which would be satisfied (output true) *only* if the results of the other two gates were the same as each other, both true or both false.

"That'll do it. Now I just need to run this on each of the four possible configurations of kittens and tigers to find out which one satisfies all the constraints, and then I'll know which box to open."

"Ahem," said the king.

The princess rolled her eyes, "What now, dad?"

"And… you are only allowed to run your device once."

"Oh," said the princess. This presented a real problem. Running the device only once would require her to guess which input configuration to test, and would be unlikely to return a conclusive answer. She'd have a 25% chance of guessing the correct input to try, but if that failed and her circuit produced a 0, she would need to just choose a box randomly and hope for the best. With all of this guessing, and knowing her father, she'd probably get eaten by a tiger. No, conventional digital logic just would not do the job this time.

But fortunately (in a quite circular fashion), the princess had recently read an O'Reilly book on quantum computation. So, gurgling with delight, she once again dashed to her workshop. A few hours later she returned, having built a new, somewhat larger, device. She switched it on, logged in via a secure terminal, ran her program, and cheered in triumph. Running to the correct box, she threw it open and happily hugged her kitten.

The end.

Example 10-2 is the QPU program that the princess used, as illustrated in Figure 10-8. The figure also shows the conventional logic gate that each part of the QPU circuit ultimately implements in phase logic. Let's check that it works!

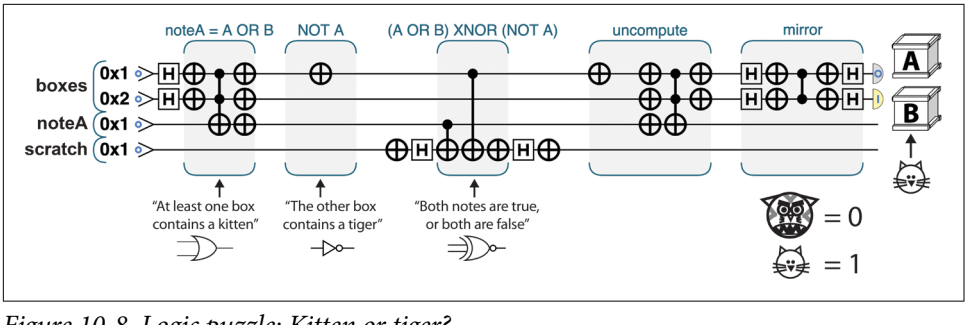


Figure 10-8. Logic puzzle: Kitten or tiger?

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=10-2*.

Example 10-2. Kittens and tigers

```
qc.reset(4);
var boxes = qint.new(2, 'boxes')
var noteA = qint.new(1, 'noteA')
```

```
var anc = qint.new(1,'anc')
qc.write(0);

// Put both boxes into a quantum kitten/tiger superposition
boxes.hadamard();

// Satisfy the note on box A using binary logic
// noteA = A OR B
qc.not(0x1|0x2);
qc.cnot(0x4,0x1|0x2)
qc.not(0x1|0x2|0x4);

// Satisfy the note on box B using binary logic
// NOT A
qc.not(0x1);

// Put the phase-logic scratch qubit into the |+> state
anc.not();
anc.hadamard();

// Satisfy the final condition using phase-logic
// (A OR B) XNOR (NOT A)
qc.cnot(0x8,0x4);
qc.cnot(0x8,0x1);
qc.not(0x8);

// Return the scratch to |0>
anc.hadamard();
anc.not();

// Uncompute all of the binary logic
qc.not(0x1);
qc.nop();
qc.not(0x1|0x2|0x4);
qc.cnot(0x4,0x1|0x2)
qc.not(0x1|0x2);

// Use mirror to convert the flipped phase
boxes.Grover();

// Read and interpret the result!
var result = boxes.read();
var catA = result & 1 ? 'kitten' : 'tiger';
var catB = result & 2 ? 'kitten' : 'tiger';
qc.print('Box A contains a ' + catA + '\n');
qc.print('Box B contains a ' + catB + '\n');
```

After just a single run of the QPU program in Example 10-2, we can solve the riddle! Just as in Example 10-1, we use magnitude logic until the final operation, for which we employ phase logic. The circuit output shows clearly (and with 100% probability)

that the birthday girl should open box B if she wants a kitten. Our phase-logic subroutines take the place of `flip` in an AA iteration, so we follow them with the `mirror` subroutine originally defined in Example 6-1.

As discussed in Chapter 6, the number of times that we need to apply a full AA iteration will depend on the number of qubits involved. Luckily, this time we only needed one! We were also lucky that there *was* a solution to the riddle (i.e., that *some* set of inputs did satisfy the boolean statement). We'll see shortly what would happen if there wasn't a solution.

Remember the subtlety that the `mirror` subroutine increases the magnitude of values that have their phases *flipped* with respect to other ones. This doesn't necessarily mean that the phase has to be negative for this particular state and positive for the rest, so long as it is flipped with respect to the others. In this case, one of the options will be correct (and have a *positive* phase) and the others will be incorrect (with a *negative* phase). But the algorithm works just as well!

General Recipe for Solving Boolean Satisfiability Problems

The princess's puzzle was, of course, a feline form of a boolean satisfiability problem. The approach that we used to solve the riddle generalizes nicely to other boolean satisfiability problems with a QPU:

1. Transform the boolean statement from the satisfiability problem in question into a form having a number of clauses that have to be satisfied simultaneously (i.e., so that the statement is the `AND` of a number of independent clauses).[7]
2. Represent each individual clause using magnitude logic. Doing so will require a number of scratch qubits. As a rule of thumb, since most qubits will be involved in more than one clause, it's useful to have one scratch qubit per logical clause.
3. Initialize the full QPU register (containing qubits representing all input variables to the statement) in a uniform superposition (using `HAD`s), and initialize all scratch registers in the $|0\rangle$ state.
4. Use the magnitude-logic recipes given in Figure 5-25 to build the logic gates in each clause one by one, storing the output value of each logical clause in a scratch qubit.

7 This form is the most desirable, since the final `pAND` combining all the statements in phase logic can be implemented using a single `CPHASE` with no need for extra scratch qubits. However, other forms can be implemented with carefully prepared scratch qubits.

5. Once all the clauses have been implemented, perform a phase-logic `AND` between all the scratch qubits to combine the different clauses.
6. Uncompute all of the magnitude-logic operations, returning the scratch qubits to their initial states.
7. Run a `mirror` subroutine on the QPU register encoding our input variables.
8. Repeat the preceding steps as many times as is necessary according to the amplitude amplification formula given in Equation 6-2.
9. Read the final (amplitude-amplified) result by reading out the QPU register.

In the following sections we give two examples of applying this recipe, the second of which illustrates how the procedure works in cases where a statement we're trying to satisfy *cannot* actually be satisfied (i.e., no input combinations can yield a 1 output).

Hands-on: A Satisfiable 3-SAT Problem

Consider the following 3-SAT problem:

```
(a OR b) AND (NOT a OR c) AND (NOT b OR NOT c) AND (a OR c)
```

Our goal is to find whether any combination of boolean inputs `a`, `b`, `c` can produce a 1 output from this statement. Luckily, the statement is already the `AND` of a number of clauses (how convenient!). So let's follow our steps. We'll use a QPU register of seven qubits—three to represent our variables `a`, `b`, and `c`, and four scratch qubits to represent each of the logical clauses. We then proceed to implement each logical clause in magnitude logic, writing the output from each clause into the scratch qubits. Having done this, we implement a phase-logic `AND` statement between the scratch qubits, and then uncompute each of the magnitude-logic clauses. Finally, we apply a `mirror` subroutine to the seven-qubit QPU register, completing our first amplitude amplification iteration. We can implement this solution with the sample code in Example 10-3, as shown in Figure 10-9.

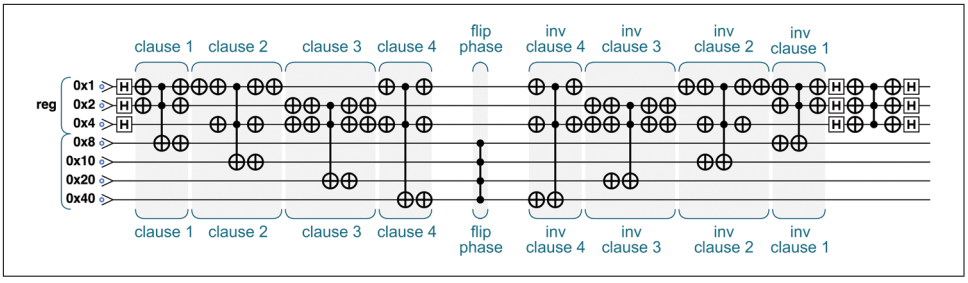


Figure 10-9. A satisfiable 3-SAT problem

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=10-3*.

Example 10-3. Satisfiable 3-SAT problem

```
var num_qubits = 3;
var num_ancilla = 4;

qc.reset(num_qubits+num_ancilla);
var reg = qint.new(num_qubits, 'reg');
qc.write(0);

reg.hadamard();

// clause 1
bit_or(0x1,0x2,0x8);

// clause 2
qc.not(0x1);
bit_or(0x1,0x4,0x10);
qc.not(0x1);

// clause 3
qc.not(0x2|0x4);
bit_or(0x2,0x4,0x20);
qc.not(0x2|0x4);

// clause 4
bit_or(0x1,0x4,0x40);

// flip phase
phase_and(0x8|0x10|0x20|0x40);

// inv clause 4
inv_bit_or(0x1,0x4,0x40);

// inv clause 3
qc.not(0x2|0x4);
```

```
inv_bit_or(0x2,0x4,0x20);
qc.not(0x2|0x4);

// inv clause 2
qc.not(0x1);
inv_bit_or(0x1,0x4,0x10);
qc.not(0x1);

// inv clause 1
inv_bit_or(0x1,0x2,0x8);

reg.Grover();

////////////// Definitions

// Define bit OR and inverse
function bit_or(q1, q2, out)
{
    qc.not(q1|q2);
    qc.cnot(out,q1|q2);
    qc.not(q1|q2|out);
}

function inv_bit_or(q1, q2, out)
{
    qc.not(q1|q2|out);
    qc.cnot(out,q1|q2);
    qc.not(q1|q2);
}

// Define phase AND
function phase_and(qubits)
{
    qc.cz(qubits);
}
```

Using circle notation to follow the progress of the three qubits representing `a`, `b`, and `c`, we see the results shown in Figure 10-10.

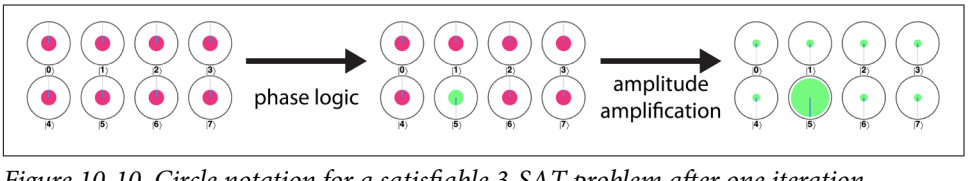


Figure 10-10. Circle notation for a satisfiable 3-SAT problem after one iteration

This tells us that the boolean statement can be satisfied by values `a=1`, `b=0`, and `c=1`.

In this example, we were able to find a set of input values that satisfied the boolean statement. What happens when no solution exists? Luckily for us, NP problems like boolean satisfiability are such that although finding an answer is computationally expensive, checking whether an answer is correct is computationally cheap. If a problem is unsatisfiable, no phase in the register will get flipped and no magnitude will change as a consequence of the `mirror` operation. Since we start with an equal superposition of all values, the final `READ` will give one value from the register at random. We simply need to check whether it satisfies the logical statement; if it *doesn't*, we can be sure that the statement is unsatisfiable. We'll consider this case next.

Hands-on: An Unsatisfiable 3-SAT Problem

Now let's look at an unsatisfiable 3-SAT problem:

```
(a OR b) AND (NOT a OR c) AND (NOT b OR NOT c) AND (a OR c) AND b
```

Knowing that this is unsatisfiable, we expect that no value assignment to variables `a`, `b`, and `c` will be able to produce an output of 1. Let's determine this by running the QPU routine in Example 10-4, as shown in Figure 10-11, following the same prescription as in the previous example.

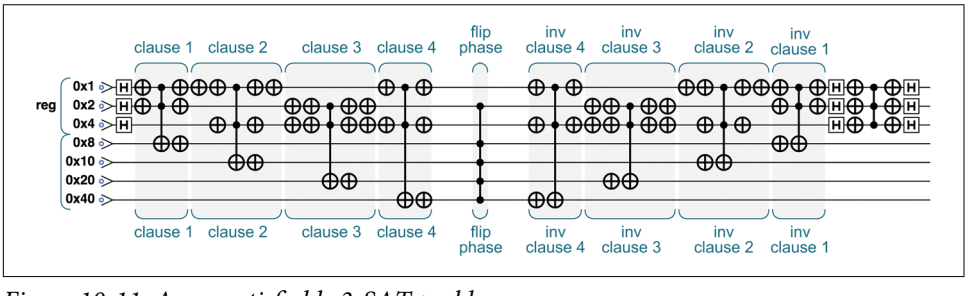


Figure 10-11. An unsatisfiable 3-SAT problem

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=10-4*.

Example 10-4. Example of a 3-SAT unsatisfiable problem

```
// --- 3-SAT - unsatisfiable
var num_qubits = 3;
var num_ancilla = 4;

qc.reset(num_qubits+num_ancilla);
var reg = qint.new(num_qubits, 'reg');
qc.write(0);

reg.hadamard();
```

```
// clause 1
bit_or(0x1,0x2,0x8);

// clause 2
qc.not(0x1);
bit_or(0x1,0x4,0x10);
qc.not(0x1);

// clause 3
qc.not(0x2|0x4);
bit_or(0x2,0x4,0x20);
qc.not(0x2|0x4);

// clause 4
bit_or(0x1,0x4,0x40);

// flip phase
phase_and(0x2|0x8|0x10|0x20|0x40);

// inv clause 4
inv_bit_or(0x1,0x4,0x40);

// inv clause 3
qc.not(0x2|0x4);
inv_bit_or(0x2,0x4,0x20);
qc.not(0x2|0x4);

// inv clause 2
qc.not(0x1);
inv_bit_or(0x1,0x4,0x10);
qc.not(0x1);

// inv clause 1
inv_bit_or(0x1,0x2,0x8);

reg.Grover();

///////////////// Definitions

// Define bit OR and inverse
function bit_or(q1, q2, out) {
    qc.not(q1|q2);
    qc.cnot(out,q1|q2);
    qc.not(q1|q2|out);
}

function inv_bit_or(q1, q2, out) {
    qc.not(q1|q2|out);
    qc.cnot(out,q1|q2);
    qc.not(q1|q2);
}
```

```
// Define phase AND
function phase_and(qubits) {
    qc.cz(qubits);
}
```

So far, so good! Figure 10-12 shows how the three qubits encoding the `a`, `b`, and `c` input values are transformed throughout the computation. Note that we only consider a single AA iteration, since (as we can see in the figure) its effect, or lack thereof, is clear after only one iteration.

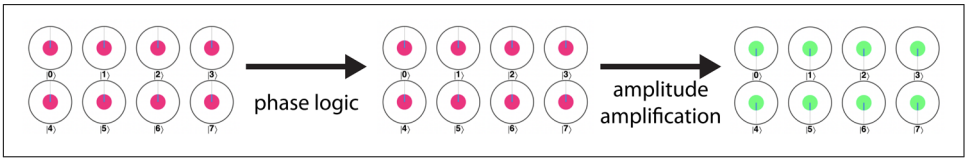


Figure 10-12. Circle notation for an unsatisfiable 3-SAT problem

Nothing has happened! Since none of the eight possible values of the register satisfy the logical statement, no phase has been flipped with respect to any others; hence, the `mirror` part of the amplitude amplification iteration has equally affected all values. No matter how many AA iterations we perform, when `READ`ing these three qubits we will obtain one of the eight values completely at random.

The fact that we could equally have obtained any value on `READ`ing might make it seem like we learned nothing, but whichever of these three values for `a`, `b`, and `c` we `READ`, we can try inputting them into our boolean statement. If we obtain a 0 outcome then we can conclude (with high probability) that the logical statement is *not* satisfiable (otherwise we would have expected to read out the satisfying values).

Speeding Up Conventional Algorithms

One of the most notable features of amplitude amplification is that, in certain cases, it can provide a quadratic speedup not only over brute-force conventional algorithms, but also over the best conventional implementations.

The algorithms that amplitude amplification can speed up are those with one-sided error. These are algorithms for decision problems containing a subroutine outputting a yes/no answer, such that if the answer to the problem is "no," the algorithm always outputs "no," while if the answer to the problem is "yes," the algorithm outputs the answer "yes" with probability $p > 0$. In the case of 3-SAT, which we have seen in the previous examples, the algorithms outputs the the "yes" answer (the formula is satisfiable) only if it finds a satisfiable assignment.

To find a solution with some desired probability, these conventional algorithms must repeat their probabilistic subroutine some number of times (k times if the algorithm has a runtime of $O(k^n poly(n))$). To obtain a QPU speedup, combine them with the quantum routine we simply need to substitute the repeated probabilistic subroutine with an amplitude amplification step.

Any conventional algorithm of this form can be combined with amplitude amplification to make it faster. For example, the naive approach to solving 3-SAT in the earlier examples runs in $O(1.414^n poly(n))$, while the best conventional algorithm runs faster, in $O(1.3(29^n poly(n))$. However, by combining this conventional result with amplitude amplification, we can achieve a runtime of $O(1.1(53^n poly(n))$!

There are a number of other algorithms that can be sped up using this technique. For example:

Element distinctness
: Algorithms which, given a function f acting on a register, can determine whether there exist two distinct elements in the register i, j for which $f(i) = f(j)$. This problem has applications such as finding triangles in graphs or calculating matrix products.

Finding global minima
: Algorithms which, given an integer-valued function acting on a register with N entries, find the index i of the register such that $f(i)$ has the lowest value.

Chapter 14 provides a reference list where many of these algorithms can be found.

CHAPTER 11

Quantum Supersampling

From pixel-based adventure games to photorealistic movie effects, computer graphics has a history of being at the forefront of computing innovations. Quantum Image Processing (QIP) employs a QPU to enhance our image processing capabilities. Although very much in its infancy, QIP already offers exciting examples of how QPUs might impact the field of computer graphics.

What Can a QPU Do for Computer Graphics?

In this chapter we explore one particular QIP application called *Quantum Supersampling* (*QSS*). QSS leverages a QPU to find intriguing improvements to the computer graphics task of *supersampling*, as is summarized in Figure 11-1. Supersampling is a conventional computer graphics technique whereby an image generated by a computer at high resolution is reduced to a lower-resolution image by selectively sampling pixels. Supersampling is an important step in the process of producing usable output graphics from computer-generated images.

QSS was originally developed as a way to *speed up* supersampling with a QPU, and by this metric it failed. However, numerical analysis of the results revealed something interesting. Although the final image quality of QSS (measured as error per pixel) is about the same as for existing conventional methods, the output image has a different kind of advantage.

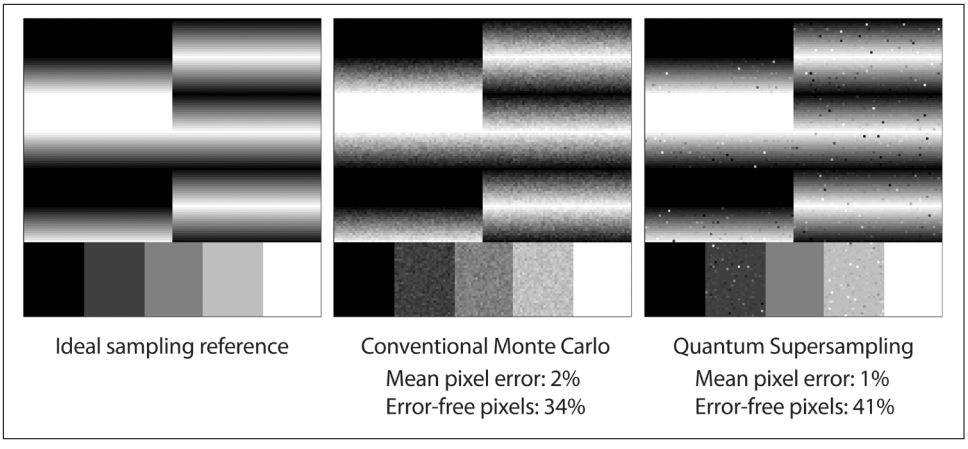


Figure 11-1. Results of QSS (with ideal and conventional cases for comparison); these reveal a change in the character of the noise in a sampled image

Figure 11-1 shows that the average pixel noise[1] in the conventional and QSS sampled images is about the same, but the *character* of the noise is very different. In the conventionally sampled image, each pixel is a little bit noisy. In the quantum image, some of the pixels are *very* noisy (black and white specks), while the rest are perfect.

Imagine you've been given an image, and you're allowed 15 minutes to manually remove the visible noise. For the image generated by QSS the task is fairly easy; for the conventionally sampled image, it's nearly impossible.

QSS combines a range of QPU primitives: quantum arithmetic from Chapter 5, amplitude amplification from Chapter 6, and the Quantum Fourier Transform from Chapter 7. To see how these primitives are utilized, we first need a little more background in the art of supersampling.

Conventional Supersampling

Ray tracing_ is a technique for producing computer-generated images, where increased computational resources allow us to produce results of increasing quality. Figure 11-2 shows a schematic representation of how ray tracing produces images from computer-generated scenes.

1 Here pixel noise is defined as the difference between the sampled and ideal results.

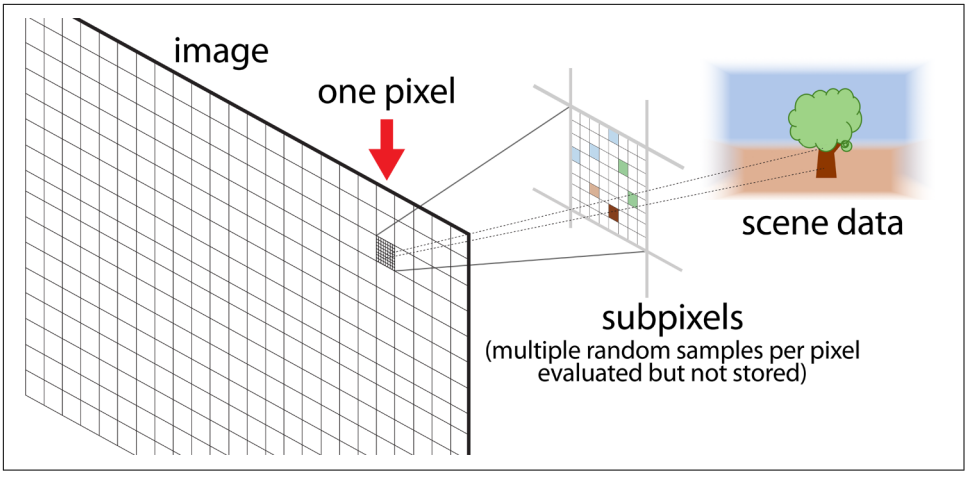


Figure 11-2. In ray tracing, many samples are taken for each pixel of the final image

For each pixel in the final image, a mathematical ray is projected in 3D space from the camera through the pixel and toward the computer-generated scene. The ray strikes an object in the scene, and in the simplest version (ignoring reflections and transparency) the object's color determines the color of the corresponding pixel in the image.

While casting just one ray per pixel would produce a correct image, for faraway details such as the tree in Figure 11-2 edges and fine details in the scene are lost. Additionally, as the camera or the object moves, troublesome noise patterns appear on objects such as tree leaves and picket fences.

To solve this without making the image size ridiculously large, ray-tracing software casts multiple rays per pixel (hundreds or thousands is typical) while varying the direction slightly for each one. Only the *average* color is kept, and the rest of the detail is discarded. This process is called *supersampling*, or *Monte Carlo sampling*. As more samples are taken, the noise level in the final image is reduced.

Supersampling is a task where we perform parallel processing (calculating the results of many rays interacting with a scene), but ultimately only need the *sum* of the results, not the individual results themselves. This sounds like something a QPU might help with! A full QPU-based ray-tracing engine would require far more qubits than are currently available. However, to demonstrate QSS we can use a QPU to draw higher-resolution images by less-sophisticated methods (ones not involving ray-tracing!), so that we can study how a QPU impacts the final resolution-reducing supersampling step.

Hands-on: Computing Phase-Encoded Images

To make use of a QPU for supersampling, we'll need a way of representing images in a QPU register (one that isn't as tricky as full-blown ray tracing!).

There are many different ways to represent pixel images in QPU registers and we summarize a number of approaches from the quantum image processing literature at the end of this chapter. However, we'll use a representation that we refer to as *phase encoding*, where the values of pixels are represented in the phases of a superposition. Importantly, this allows our image information to be compatible with the amplitude amplification technique we introduced in Chapter 6.

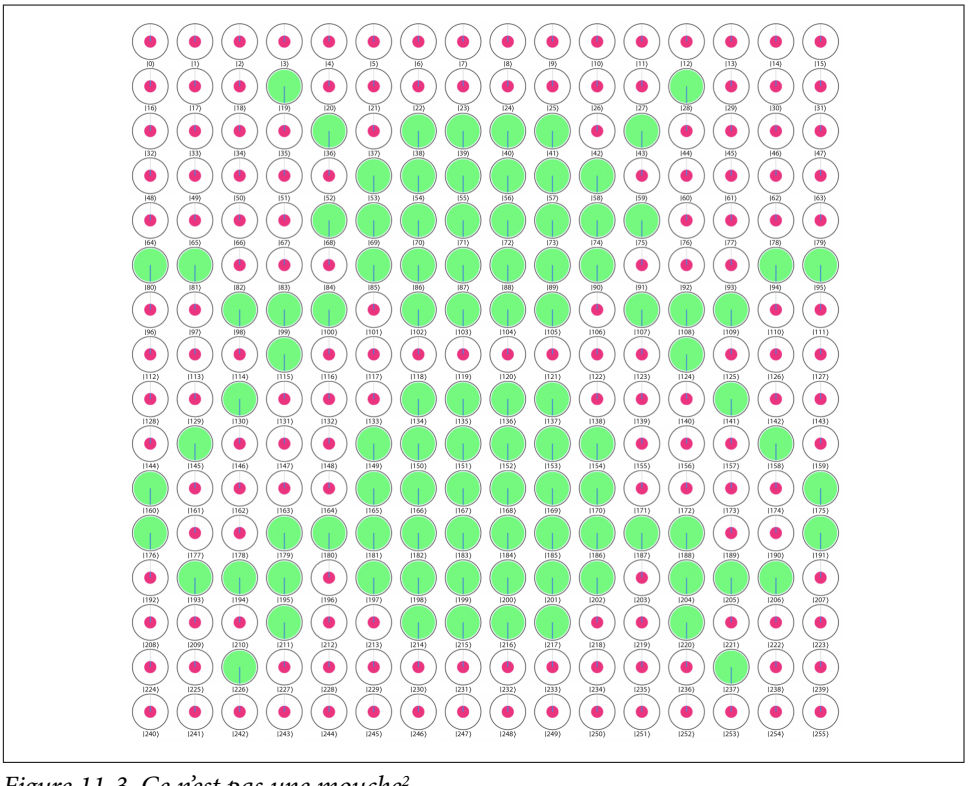


Figure 11-3. Ce n'est pas une mouche[2]

2 In *La trahison des images*, the painter René Magritte highlights the realization that an image of a pipe is not a pipe. Similarly, this phase-encoded *image* of a fly is not the complete quantum state of an actual fly.

Encoding images in the phases of a superposition isn't entirely new to us. At the end of Chapter 4 we used an eight-qubit register to phase-encode a whimsical image of a fly,[3] as shown in Figure 11-3.

In this chapter we'll learn how phase-encoded images like this can be created, and then use them to demonstrate Quantum Supersampling.

A QPU Pixel Shader

A *pixel shader* is a program (often run on a GPU) that takes an x and y position as input, and produces a color (black or white in our case) as output. To help us demonstrate QSS, we'll construct a *quantum* pixel shader.

To begin, Example 11-1 initializes two four-qubit registers, `qx` and `qy`. These will serve as the x and y input values for our shader. As we've seen in previous chapters, performing a `HAD` on all qubits produces a uniform superposition of all possible values, as in Figure 11-4. This is our blank canvas, ready for drawing.

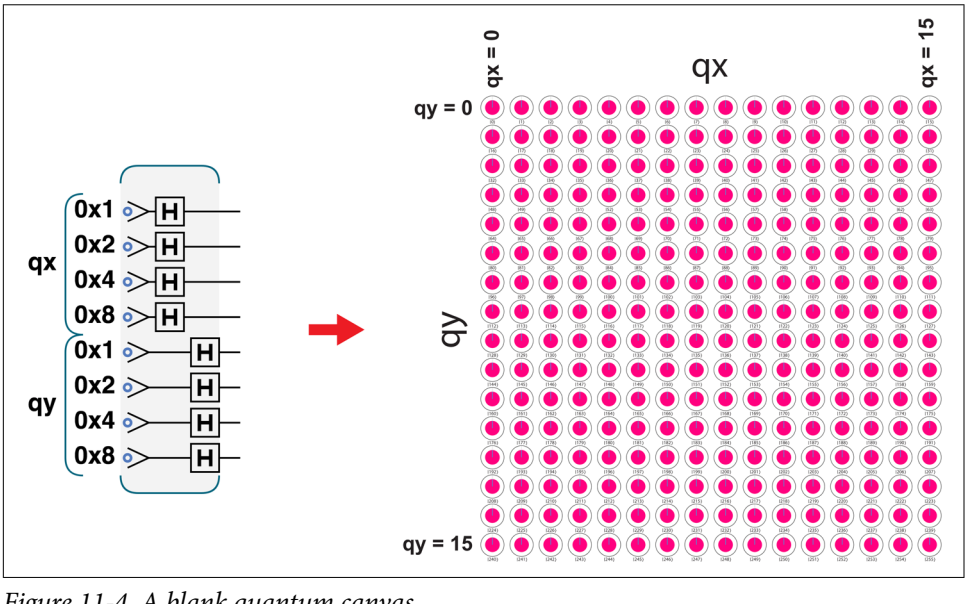


Figure 11-4. A blank quantum canvas

3 The complete code for the fly-in-the-teleporter experiment can be found at *http://oreilly-qc.github.io?p=4-2*. In addition to phase logic, the `mirror` subroutine from Chapter 6 was applied to make the fly easier to see.

Using PHASE to Draw

Now we're ready to draw. Finding concise ways to draw into register phases can get complicated and subtle, but we can start by filling the right half of the canvas simply by performing the operation "if `qx >= 8` then invert phase." This is accomplished by applying a `PHASE(180)` to a single qubit, as demonstrated in Figure 11-5:

```
// Set up and clear the canvas
qc.reset(8);
var qx = qint.new(4, 'qx');
var qy = qint.new(4, 'qy');
qc.write(0);
qx.hadamard();
qy.hadamard();

// Invert if qx >= 8
qc.phase(180, qx.bits(0x8));
```

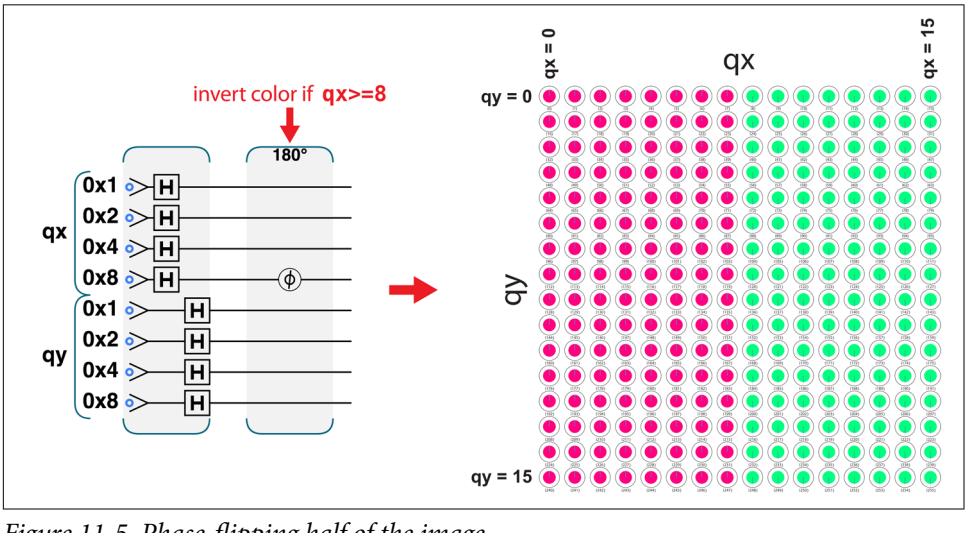


Figure 11-5. Phase-flipping half of the image

In a sense, we've used *just one* QPU instruction to fill 128 pixels. On a GPU, this would have required the pixel shader to run 128 times. As we know only too well, the catch is that if we try to `READ` the result we get only a random value of `qx` and `qy`.

With a little more logic, we can fill a square with a 50% gray dither pattern.[4] For this, we want to flip any pixels where qx and qy are both greater than or equal to 8, *and* where the low qubit of qx is not equal to the low qubit of qy. We do this in Example 11-1, and the results can be seen in Figure 11-6.

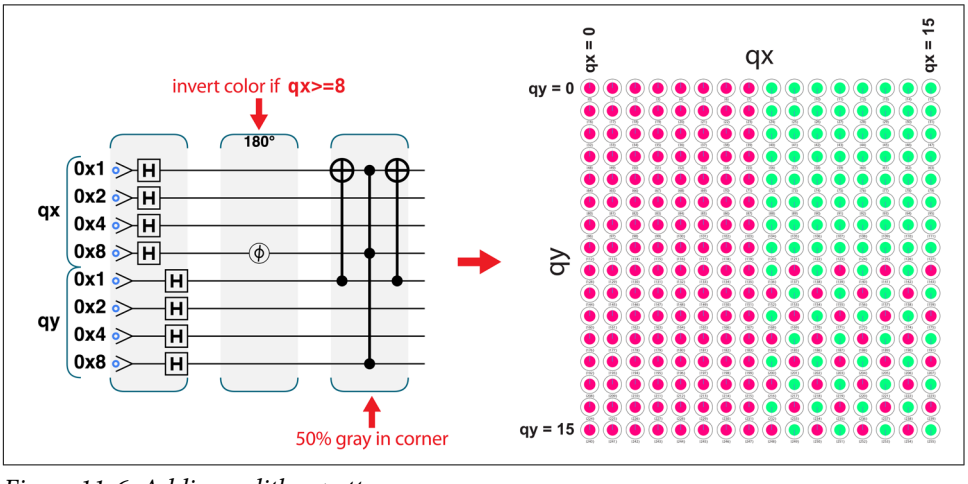


Figure 11-6. Adding a dither pattern

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=11-1*.

Example 11-1. Drawing basics

```
// Clear the canvas
qc.reset(8);
var qx = qint.new(4, 'qx');
var qy = qint.new(4, 'qy');
qc.write(0);
qx.hadamard();
qy.hadamard();

// Invert if qx >= 8
qc.phase(180, qx.bits(0x8));

// 50% gray in the corner
qx.cnot(qy, 0x1);
```

4 Dithering is the process of applying a regular pattern to an image, normally to approximate otherwise unobtainable colors by visually "blending" a limited color palette.

```
qc.cphase(180, qy.bits(0x8, qx.bits(0x8|0x1)));
qx.cnot(qy, 0x1);
```

With a little arithmetic from Chapter 5, we can create more interesting patterns, as shown in Figure 11-7:

```
// Clear the canvas
qc.reset(8);
var qx = qint.new(4, 'qx');
var qy = qint.new(4, 'qy');
qc.write(0);
qx.hadamard();
qy.hadamard();

// fun stripes
qx.subtractShifted(qy, 1);
qc.phase(180, qx.bits(0x8));
qx.addShifted(qy, 1);
```

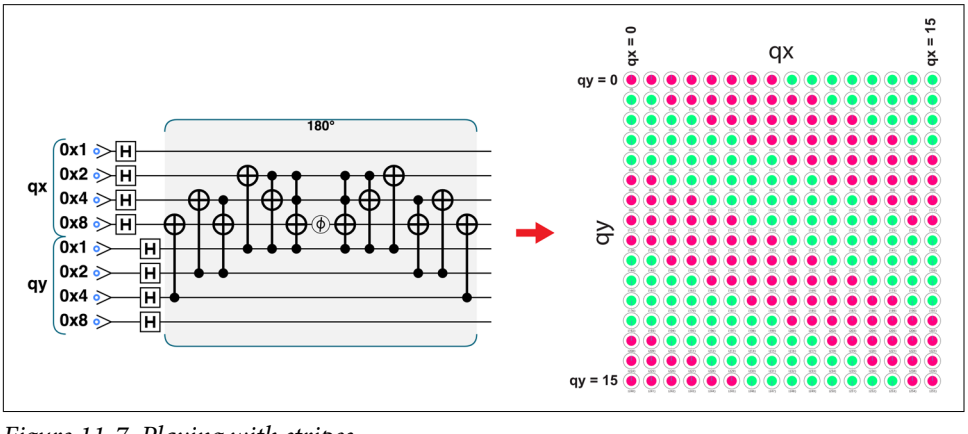


Figure 11-7. Playing with stripes

Drawing Curves

To draw more complex shapes, we need more complex math. Example 11-2 demonstrates the use of the `addSquared()` QPU function from Chapter 5 to draw a quarter-circle with a radius of 13 pixels. The result is shown in Figure 11-8. In this case, the math must be performed in a larger 10-qubit register, to prevent overflow when we square and add `qx` and `qy`. Here we've used the trick learned in Chapter 10 of storing the value of a logical (or mathematical) operation in the phase of a state, through a combination of magnitude and phase-logic operations.

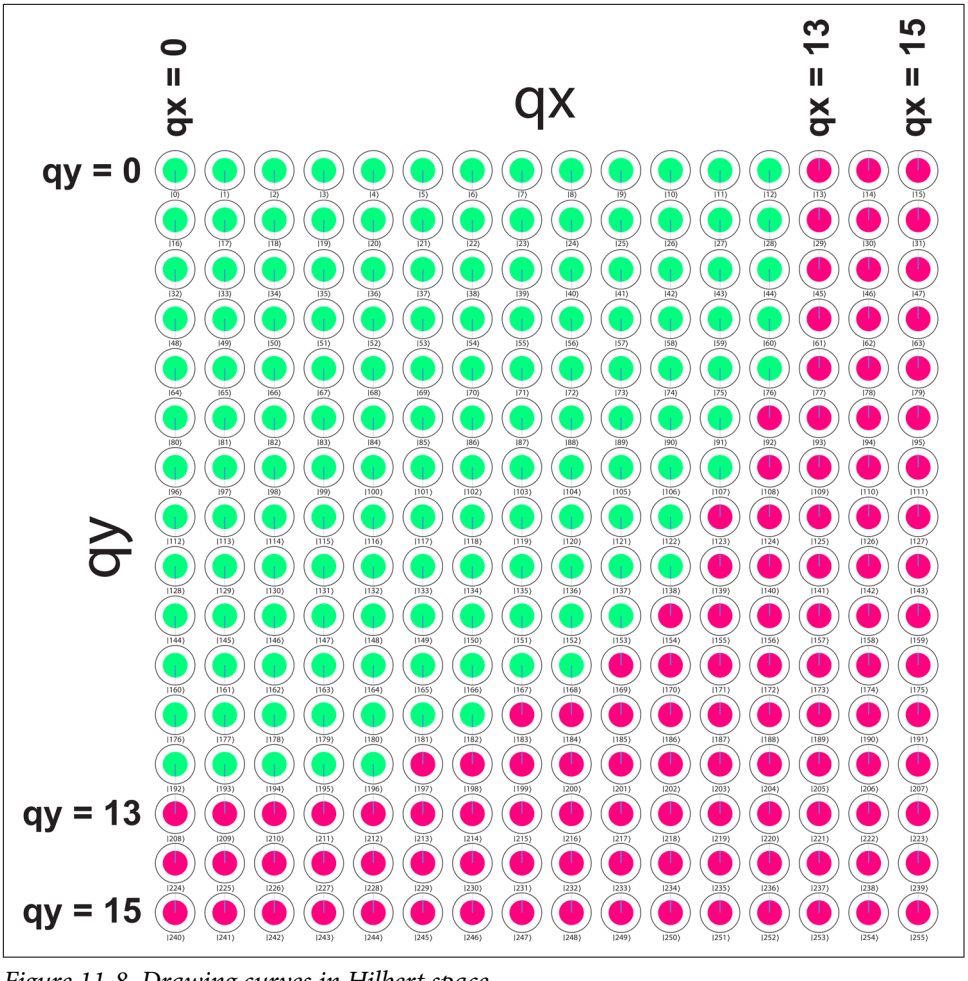


Figure 11-8. Drawing curves in Hilbert space

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=11-2*.

Example 11-2. Drawing curves in Hilbert space

```
var radius = 13;
var acc_bits = 10;

// Clear the canvas
qc.reset(18);
var qx = qint.new(4, 'qx');
var qy = qint.new(4, 'qy');
var qacc = qint.new(10, 'qacc');
```

```
qc.write(0);
qx.hadamard();
qy.hadamard();

// fill if x^2 + y^2 < r^2
qacc.addSquared(qx);
qacc.addSquared(qy);
qacc.subtract(radius * radius);
qacc.phase(180, 1 << (acc_bits - 1));
qacc.add(radius * radius);
qacc.subtractSquared(qy);
qacc.subtractSquared(qx);
```

If the `qacc` register is too small, the math performed in it will overflow, resulting in curved bands. This overflow effect will be used deliberately in Figure 11-11, later in the chapter.

Sampling Phase-Encoded Images

Now that we can represent images in the phases of quantum registers, let's return to the problem of supersampling. Recall that in supersampling we have many pieces of information calculated from a computer-generated scene (corresponding to different rays we have traced) that we want to combine to give a single pixel in our final output image. software verification To simulate this, we can consider our 16×16 array of quantum states to be made of 16 4×4 tiles as shown in Figure 11-9.

We imagine that the full 16×16 image is the higher-resolution data, which we then want reduce into 16 final pixels.

Since they're only 4×4 subpixels, the `qx` and `qy` registers needed to draw into these tiles can be reduced to two qubits each. We can use an overflow register (which we call `qacc` since such registers are often referred to as *accumulators*) to perform any required logical operations that won't fit in two qubits. Figure 11-10 (Example 11-3) shows the circuit for performing the drawing shown in Figure 11-9, tile by tile.

Note that in this program, the variables `tx` and `ty` are digital values indicating which tile of the image the shader is working on. We get the absolute `x` value of a subpixel that we want to draw by adding `qx` and (`tx x 4`), which we can easily compute since `tx` and `ty` are not quantum. Breaking our images into tiles this way makes it easier for us to subsequently perform supersampling.

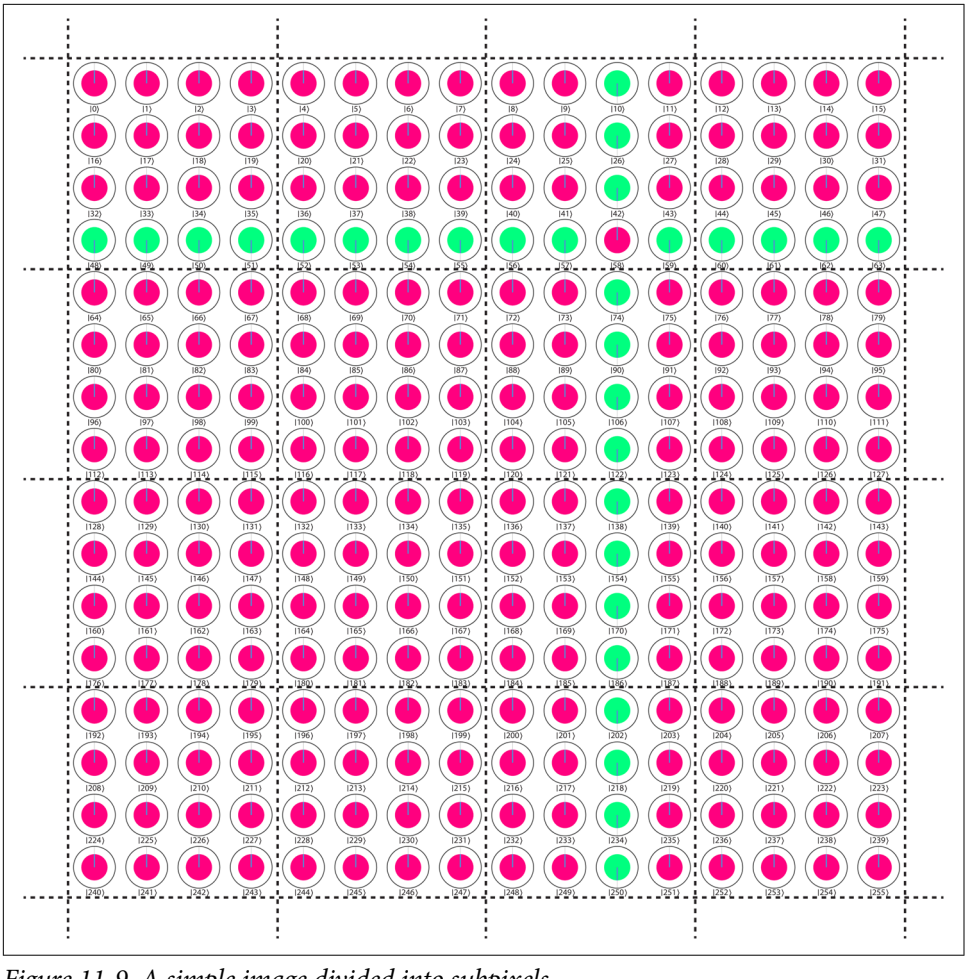


Figure 11-9. A simple image divided into subpixels

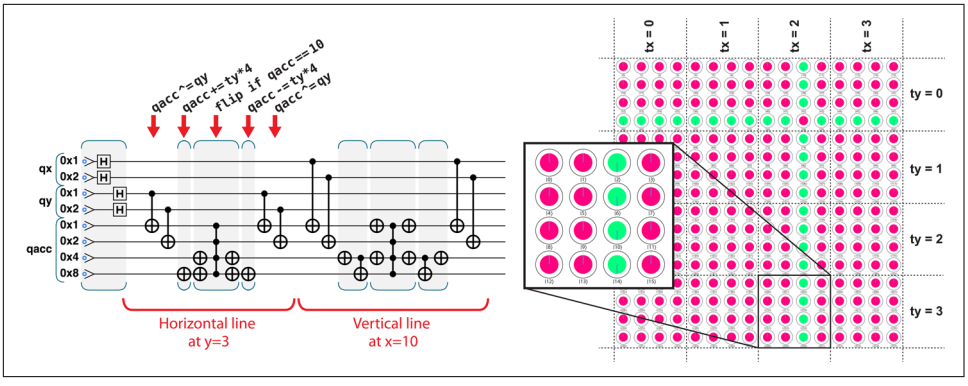


Figure 11-10. Drawing the subpixels of one tile in superposition

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=11-3*.

Example 11-3. Drawing lines using an accumulator

```
// Set up and clear the canvas
qc.reset(8);
var qx = qint.new(2, 'qx');
var qy = qint.new(2, 'qy');
var qacc = qint.new(4, 'qacc');
qc.write(0);
qx.hadamard();
qy.hadamard();

// Choose which tile to draw
var tx = 2;  // tile column
var ty = 1;  // tile row

// Hz line y=3
qacc.cnot(qy)
qacc.add(ty * 4);
qacc.not(~3);
qacc.cphase(180);
qacc.not(~3);
qacc.subtract(ty * 4);
qacc.cnot(qy);

// Vt line x=10
qacc.cnot(qx)
qacc.add(tx * 4);
qacc.not(~10);
qacc.cphase(180);
qacc.not(~10);
```

```
qacc.subtract(tx * 4);
qacc.cnot(qx);
```

A More Interesting Image

With a more sophisticated shader program, we can produce a more interesting test for the quantum supersampling algorithm. In order to test and compare supersampling methods, here we use circular bands to produce some very high-frequency detail. The full source code for the image in Figure 11-11, which also splits the phase-encoded image into tiles as we mentioned earlier, can be run at *http://oreilly-qc.github.io?p=11-A*.

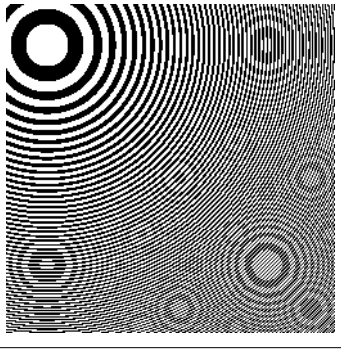


Figure 11-11. A more involved image with some high-frequency detail conventionally rendered at 256×256 pixels (shown here).

It may seem like we've had to use a range of shortcuts and special-case tricks to produce these images. But this is not so different from the hacks and workarounds that were necessary in the very early days of conventional computer graphics.

Now that we have a higher-resolution phase-encoded image broken up into tiles, we're ready to apply the QSS algorithm. In this case, the full image is drawn at 256×256 pixels. We'll use 4,096 tiles, each composed of 4×4 subpixels, and supersample all the subpixels in a single tile to generate one pixel of the final sampled image from its 16 subpixels.

Supersampling

For each tile, we want to estimate the number of subpixels that have been phase-flipped. With black and white subpixels (represented for us as flipped or unflipped phases), this allows us to obtain a value for each final pixel representative of the

intensity of its original constituent subpixels. Handily, this problem is exactly the same as the Quantum Sum Estimation problem from "Multiple Flipped Entries" on page 114.

To make use of Quantum Sum Estimation, we simply consider the quantum program implementing our drawing instructions to be the `flip` subroutine we use in amplitude amplification from Chapter 6. Combining this with the Quantum Fourier Transform from Chapter 7 allows us to approximate the total number of flipped subpixels in each tile. This involves running our drawing program multiple times for each tile.

Note that, without a QPU, sampling a high-resolution image into a lower-resolution one would still require running a drawing subroutine multiple times for each tile, randomizing the values `qx` and `qy` for each sample. Each time we would simply receive a "black" or "white" sample, and by adding these up we would converge on an approximated image.

The sample code in Example 11-4 shows the results of both quantum and conventional supersampling (typically performed in the conventional case by a technique known as Monte Carlo sampling), and we see them compared in Figure 11-12. The *ideal reference* shows the result we would get from perfect sampling, and the QSS lookup table is a tool we discuss in detail in the coming pages.

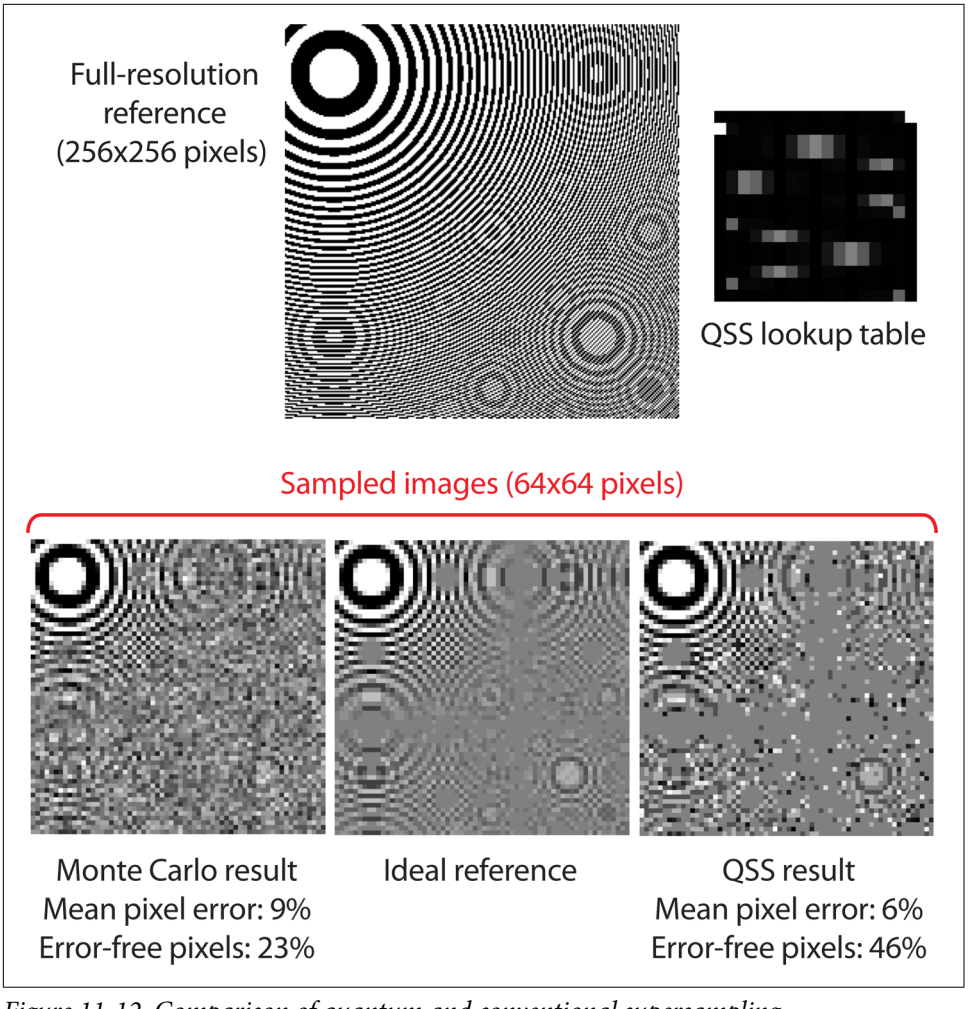


Figure 11-12. Comparison of quantum and conventional supersampling

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=11-4*.

Example 11-4. Supersampling

```
function do_qss_image()
{
    var sp = {};
    var total_qubits = 2 * res_aa_bits + num_counter_bits
                       + accum_bits;

    // Set up the quantum registers
```

```
    qc.reset(total_qubits);
    sp.qx = qint.new(res_aa_bits, 'qx');
    sp.qy = qint.new(res_aa_bits, 'qy');
    sp.counter = qint.new(num_counter_bits, 'counter');
    sp.qacc = qint.new(accum_bits, 'scratch');
    sp.qacc.write(0);

    // For each tile in the image, run the qss_tile() function
    for (var sp.ty = 0; sp.ty < res_tiles; ++sp.ty) {
        for (var sp.tx = 0; sp.tx < res_tiles; ++sp.tx)
            qss_tile(sp);
    }
}

function qss_tile(sp)
{
    // Prepare the tile canvas
    sp.qx.write(0);
    sp.qy.write(0);
    sp.counter.write(0);
    sp.qx.hadamard();
    sp.qy.hadamard();
    sp.counter.hadamard();

    // Run the pixel shader multiple times
    for (var cbit = 0; cbit < num_counter_bits; ++cbit) {
        var iters = 1 << cbit;
        var qxy_bits = sp.qx.bits().or(sp.qy.bits());
        var condition = sp.counter.bits(iters);
        var mask_with_condition = qxy_bits.or(condition);
        for (var i = 0; i < iters; ++i) {
            shader_quantum(sp.qx, sp.qy, sp.tx, sp.ty, sp.qacc,
                           condition, sp.qcolor);
            grover_iteration(qxy_bits, mask_with_condition);
        }
    }
    invQFT(sp.counter);

    // Read and interpret the result
    sp.readVal = sp.counter.read();
    sp.hits = qss_count_to_hits[sp.readVal];
    sp.color = sp.hits / (res_aa * res_aa);
    return sp.color;
}
```

As mentioned at the beginning of this chapter, the interesting advantage we obtain with QSS is not to do with the number of drawing operations that we have to perform, but rather relates to a difference in the *character* of the noise we observe.

In this example, when comparing equivalent numbers of samples, QSS has about 33% lower mean pixel error than Monte Carlo. More interestingly, the number of zero-error pixels (pixels in which the result *exactly* matches the ideal) is almost double that of the Monte Carlo result.

QSS Versus Conventional Monte Carlo Sampling

In contrast to conventional Monte Carlo sampling, our QSS shader never actually outputs a single subpixel value. Instead, it uses the superposition of possible values to estimate the *sum* you would have gotten from calculating them all and adding them up. If you actually need to calculate every subpixel value along the way, then a conventional computing approach is a better tool for the job. If you need to know the sum, or some other characteristic of groups of subpixels, a QPU offers an interesting alternative approach.

The fundamental difference between QSS and conventional supersampling can be characterized as follows:

Conventional supersampling
: As the number of samples increases, the result converges toward the exact answer.

Quantum supersampling
: As the number of samples increases, the probability of getting the exact answer increases.

Now that we have an idea of what QSS can do for us, let's expand a little on how it works.

How QSS Works

The core idea behind QSS is to use the approach we first saw in "Multiple Flipped Entries" on page 114, where combining amplitude amplification iterations with the Quantum Fourier Transform (QFT) allows us to estimate the number of items flipped by whatever logic we use in the `flip` subroutine of each AA iteration.

In the case of QSS, `flip` is provided by our drawing program, which flips the phase of all of the "white" subpixels.

We can understand the way that AA and QFT and work together to help us count flipped items as follows. First, we perform a single AA iteration *conditional on the value of a register of "counter" qubits*. We call this register a "counter" precisely because its value determines how many AA iterations our circuit will perform. If we now use `HAD` operations to prepare our counter register in superposition we will be performing a *superposition* of different numbers of AA iterations. Recall from "Multiple Flipped

Entries" on page 114 that the READ probabilities for multiple flipped values in a register are dependent on the number of AA iterations that we perform. We noted in this earlier discussion that oscillations are introduced depending on the number of flipped values. Because of this, when we perform a superposition of different numbers of AA iterations, we introduce a periodic oscillation across the amplitudes of our QPU register having a frequency that depends on the number of flipped values.

When it comes to READing frequencies encoded in QPU registers, we know that the QFT is the way to go: using the QFT we can determine this frequency, and consequently the number of flipped values (i.e., number of shaded subpixels). Knowing the number of subpixels we've supersampled for a single pixel in our final lower-resolution image, we can then use this to determine the *brightness* we should use for that pixel.

Although this might be a little difficult to parse in text on a page, stepping through the code in Example 11-4 and inspecting the resulting circle notation visualizations will hopefully make the approach taken by QSS more apparent.

Note that the more counter qubits we use, the better the sampling of the image will be, but at the cost of having to run our drawing code more times. This trade-off is true for both quantum and conventional approaches, as can be seen in Figure 11-13.

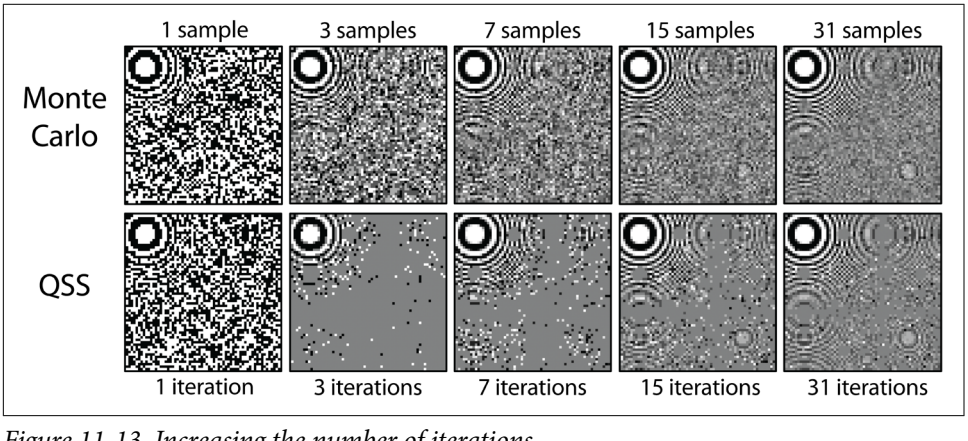


Figure 11-13. Increasing the number of iterations

The QSS lookup table

When we run the QSS algorithm and finish by READing out our QPU register, the number we get will be related but not directly equal to the number of white pixels within a given tile.

The QSS lookup table is the tool we need to look up how many subpixels a given READ value implies were within a tile. The lookup table needed for a particular image doesn't depend on the image detail (or more precisely, the details of the quantum

pixel shader we used). We can generate and reuse a single QSS lookup table for any QSS applications having given tile and counter register sizes.

For example, Figure 11-14 shows a lookup table for QSS applications having a 4×4 tile size and a counter register consisting of four qubits.

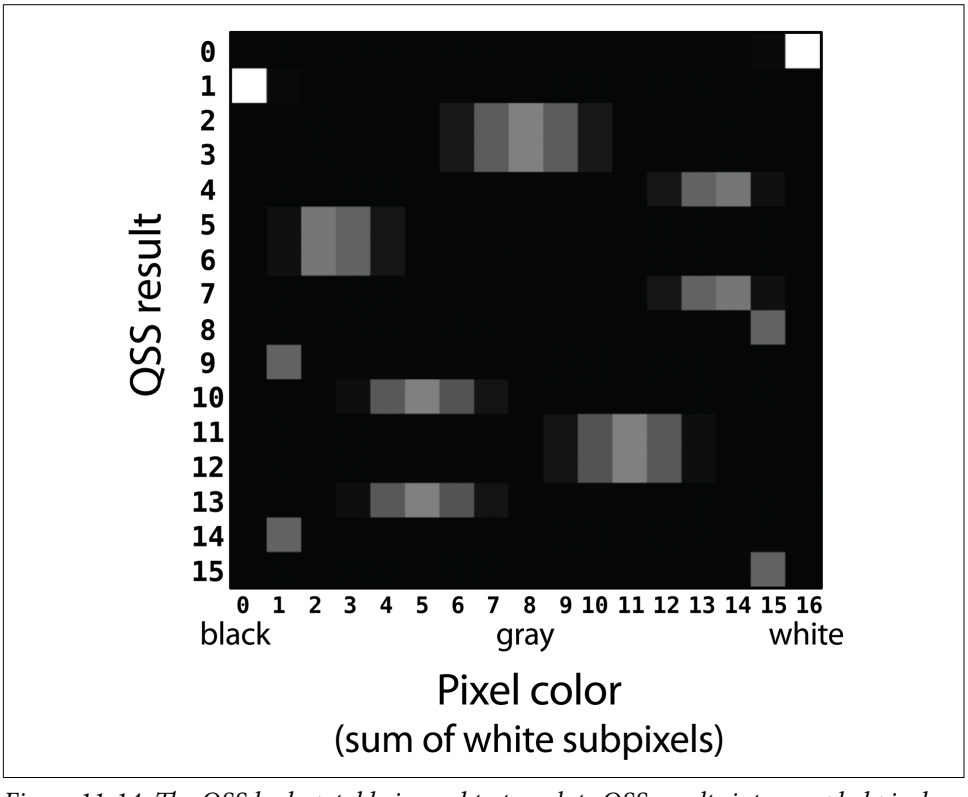


Figure 11-14. The QSS lookup table is used to translate QSS results into sampled pixel brightness

The rows (y-axis) of the lookup table enumerate the possible outcomes we might `read` in the output QPU register from an application of QSS. The columns (x-axis) enumerate the different possible numbers of subpixels within a tile that could lead to these `READ` values. The grayscale colors in the table graphically represent the probabilities associated with the different possible `READ` values (lighter colors here denoting higher probabilities). Here's an example of how we could use the lookup table. Suppose we `read` a QSS result of `10`. Finding that row in the QSS lookup table, we see that this result most likely implies that we have supersampled five white subpixels. However, there's also a nonzero probability we supersampled four or six (or to a lesser extent three or seven) subpixels. Note that some error is introduced as we cannot always uniquely infer the number of sampled subpixels from a `READ` result.

Such a lookup table is used by the QSS algorithm to determine a final result. How do we get our hands on a QSS lookup table for a particular QSS problem? The code in Example 11-5 shows how we can calculate the QSS lookup table. Note that this code doesn't rely on a particular quantum pixel shader (i.e., particular image). Since the lookup table only associated the `READ` value with a given number of white subpixels per tile (regardless of their exact locations), it can be generated without any knowledge of the actual image to be supersampled.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=11-5*.

Example 11-5. Building a QSS lookup table

```
function create_table_column(color, qxy, qcount)
{
    var true_count = color;

    // Put everything into superposition
    qc.write(0);
    qcount.hadamard();
    qxy.hadamard();

    for (var i = 0; i < num_counter_bits; ++i)
    {
        var reps = 1 << i;
        var condition = qcount.bits(reps);
        var mask_with_condition = qxy.bits().or(condition);
        for (var j = 0; j < reps; ++j)
        {
            flip_n_terms(qxy, true_count, condition);
            grover_iteration(qxy.bits(), mask_with_condition);
        }
    }
    invQFT(qcount);

    // Construct the lookup table
    for (var i = 0; i < (1 << num_counter_bits); ++i)
        qss_lookup_table[color][i] = qcount.peekProbability(i);
}
```

The QSS lookup table tells us a lot about how QSS performs for given counter register and tile sizes, acting as a sort of *fingerprint* for the algorithm. Figure 11-15 shows a few examples of how the lookup table changes as (for a fixed counter register size of four qubits) we increase the tile size (and therefore number of subpixels) used in our QSS algorithm.

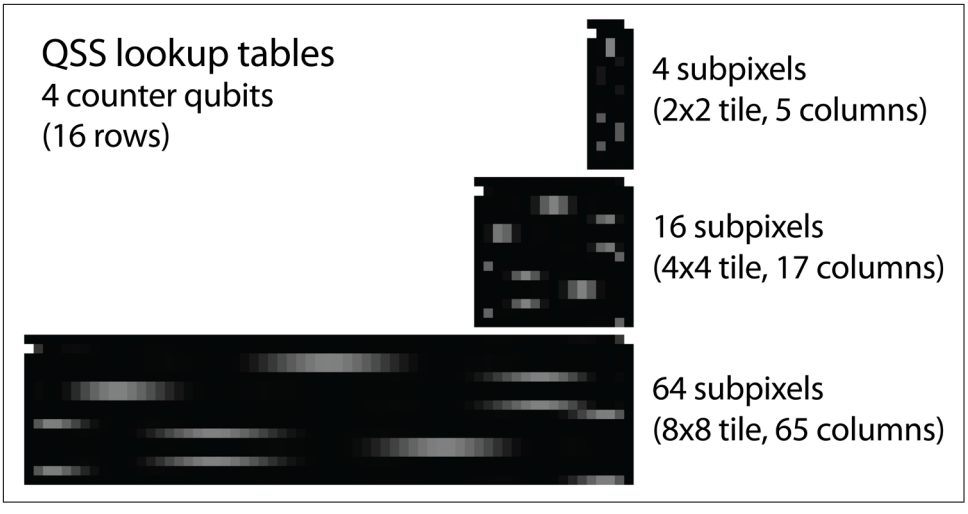


Figure 11-15. Increasing the number of subpixels adds columns

Similarly, Figure 11-16 shows how the lookup table changes as we increase the number of counter qubits used (for a given tile size).

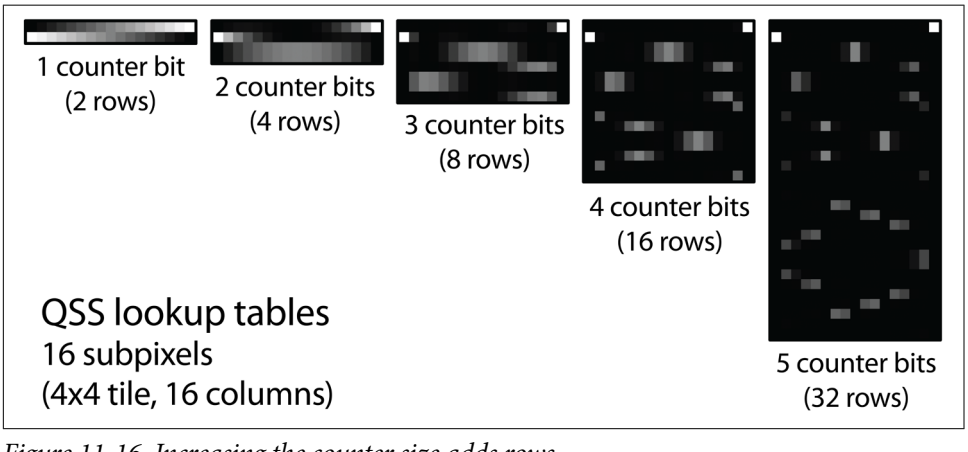


Figure 11-16. Increasing the counter size adds rows

Confidence maps

In addition to being a tool for interpreting `READ` values, we can also use a QSS lookup table to evaluate how confident we are in the final brightness of a pixel. By looking up a given pixel's QSS `READ` value in a row of the table, we can judge the probability that a pixel value we've inferred is correct. For example, in the case of the lookup table in Figure 11-14, we'd be very confident of values `READ` to be `0` or `1`, but much less confident if we'd `READ` values of `2`, `3`, or `4`. This kind of inference can be used to produce a

"confidence map" indicating the likely locations of errors in images we produce with QSS, as illustrated in Figure 11-17.

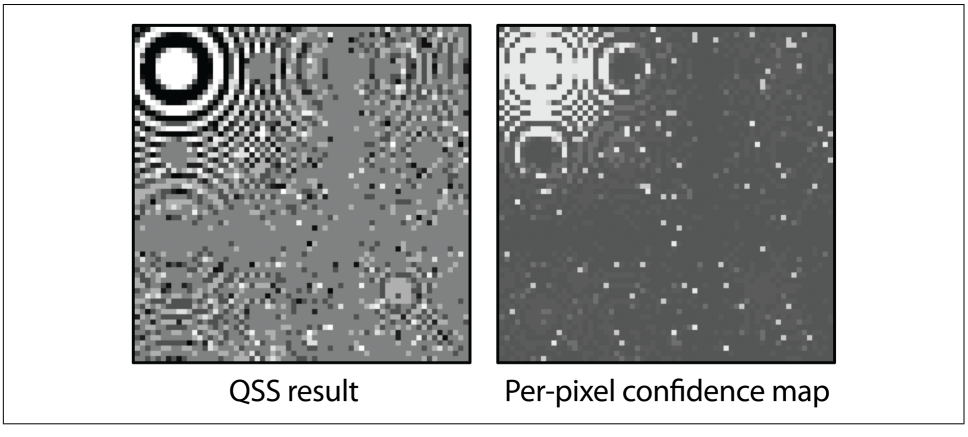


Figure 11-17. QSS result and associated confidence map generated from a QSS lookup table—in the confidence map brighter pixels denote areas where the result is more likely correct

Adding Color

The images we've been rendering with QSS in this chapter have all been one-bit monochrome, using flipped QPU register phases to represent black and white pixels. Although we're big fans of retro gaming, perhaps we can incorporate a few more colors? We *could* simply use the phases and amplitudes of our QPU register to encode a wider range of color values for our pixels, but then the Quantum Sum Estimation used by QSS would no longer work.

However, we can borrow a technique from early graphics cards known as *bitplanes*. In this approach we use our quantum pixel shader to render separate monochrome images, each representing one bit of our image. For example, suppose we want to associate three colors with each pixel in our image (red, green, and blue). Our pixel shader can then essentially generate three separate monochrome images, each of which represents the contribution of one of the three color channels. These three images can undergo supersampling separately, before being recombined into a final color image.

This only allows us eight colors (including black and white); however, supersampling allows us to blend *subpixels* together, effectively producing a 12-bit-per-pixel image (see Figure 11-18). The complete code for this can be found at *http://oreilly-qc.github.io?p=11-6*.

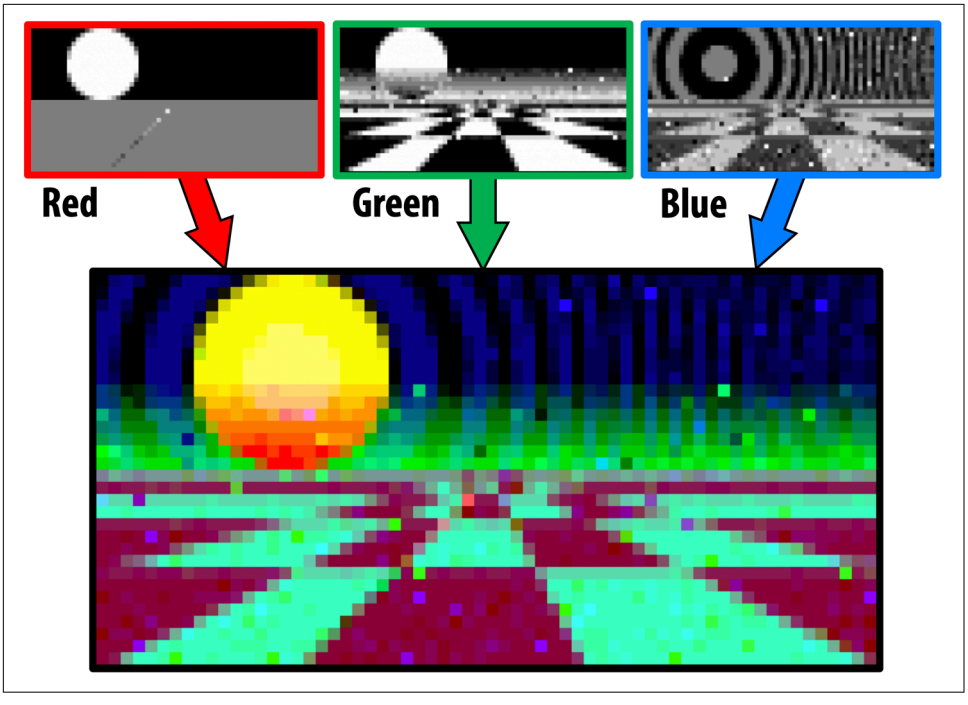


Figure 11-18. Combining supersampled color planes. Space Harrier eat your heart out!

Conclusion

This chapter demonstrated how exploring new combinations of QPU primitives with a little domain knowledge can potentially lead to new QPU applications. The ability to redistribute sampling noise also shows visually how sometimes we have to look beyond speedups to see the advantages of QPU applications.

It's also worth noting that the ability of Quantum Supersampling to produce novel noise redistributions potentially has relevance beyond computer graphics. Many other applications also make heavy use of Monte Carlo sampling, in fields such as artificial intelligence, computational fluid dynamics, and even finance.

To introduce Quantum Supersampling, we used a *phase encoding* representation of images in QPU registers. It is worth noting that Quantum Image Processing researchers have also proposed many other such representations. These include the so-called Qubit Lattice representation,[5] Flexible Representation of Quantum Images

5 Venegas-Andraca et al, 2003.

(FRQI),[6] Novel Enhanced Quantum Representation (NEQR),[7] and Generalized Quantum Image Representation (GQIR).[8] These representations[9] have been used to explore other image processing applications including template matching,[10] edge detection,[11] image classification,[12] and image translation[13] to name but a few.[14]

6 Le et al, 2011.

7 Zhang et al, 2013.

8 Jiang et al, 2015.

9 For a more detailed review of these quantum image representations, see Yan et al, 2015.

10 Curtis et al, 2004.

11 Yuan et al, 2013.

12 Ostaszewski et al, 2015.

13 Wang et al, 2015.

14 For more comprehensive reviews of QIP applications, see Cai et al, 2018 and Yan et al, 2017.

CHAPTER 12
Shor's Factoring Algorithm

If you'd heard about one application of quantum computing before you picked up this book, there's a good chance it was Shor's factoring algorithm.

Quantum computing was mostly considered to be of academic, rather than practical, interest until Peter Shor's 1994 discovery that a sufficiently powerful quantum computer can find the prime factors of a number exponentially faster than any conventional machine. In this chapter, we take a hands-on look at one specific implementation of Shor's QPU factoring algorithm.

Far from being a mere mathematical curiosity, the ability to quickly factorize large numbers can help break the Rivest–Shamir–Adleman (RSA) public-key cryptosystem. Anytime you spin up an `ssh` session you're making use of RSA. Public-key cryptosystems like RSA work by a process wherein a freely available public key can be used by anybody to *encrypt* information. But once encrypted, the information can only be *decrypted* using a secret, privately held key. Public-key cryptosystems are often compared to an electronic version of a mailbox. Imagine a locked box with a slit allowing anyone to post (but not retrieve) a message, and a door with a lock to which only the owner has the key. It turns out that the task of finding the prime factors of a large number N works really well as part of a public-key cryptosystem. The assurance that someone can only use the public key to encrypt and not to decrypt information rests on the assumption that finding the prime factors of N is a computationally infeasible task.

A full explanation of RSA is beyond the scope of this book, but the key point is that if Shor's algorithm provides a way to find the prime factors of a large number N, then it has implications for one of the modern internet's backbone components.

There are other very good reasons to get to grips with Shor's algorithm besides its cryptographic implications, as it's the best-known example of a class of algorithms

solving instances of the so-called *hidden subgroup problem*. In this kind of problem, we want to determine the periodicity of a given periodic function, where the periodicity can be potentially quite complicated. A number of problems in discrete mathematics are instances of the hidden subgroup problem, such as period finding, order finding (which is the underlying difficult problem in factoring), finding discrete logarithms, and many others. A similar procedure to what we'll see in this chapter can also provide solutions to some of these other problems.[1]

Shor's algorithm is another prime example (pun intended) of how our QPU primitives are put to use. We've seen in Chapter 7 that the QFT is perfectly suited to investigating periodic signals, and Shor's algorithm makes heavy use of it.

An especially instructive aspect of Shor's algorithm is that it also ends by leveraging a conventional program to retrieve the desired prime factors from a periodicity learned with the QFT. The algorithm works so well because it accepts the QPU's role as a *coprocessor*, applying quantum ideas only in those parts of the problem to which they are well suited.

Let's take a closer look at the idea and code behind the algorithm.

Hands-on: Using Shor on a QPU

In keeping with the hands-on theme of this book, the sample code in Example 12-1 will allow you to see Shor's factoring algorithm in operation right away, using built-in functions from QCEngine.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=12-1*.

Example 12-1. Complete factoring with Shor

```
function shor_sample() {
    var N = 35;              // The number we're factoring
    var precision_bits = 4; // See the text for a description
    var coprime = 2;         // Must be 2 in this QPU implementation

    var result = Shor(N, precision_bits, coprime);
}

function Shor(N, precision_bits, coprime) {
    // quantum part
```

1 Note that not all problems of this class are known to have solutions of this form. For example, graph isomorphism (which tests the equivalence of graphs under relabeling of vertices) also belongs to the hidden subgroup class, but it is unknown whether an efficient QPU algorithm exists for this problem.

```
    var repeat_period = ShorQPU(N, precision_bits, coprime);
    var factors = ShorLogic(N, repeat_period, coprime);
    // classical part
    return factors;
}

function ShorLogic(N, repeat_period, coprime) {
    // Given the repeat period, find the actual factors
    var ar2 = Math.pow(coprime, repeat_period / 2.0);
    var factor1 = gcd(N, ar2 - 1);
    var factor2 = gcd(N, ar2 + 1);
    return [factor1, factor2];
}
```

As mentioned earlier, the QPU is doing only part of the work here. The `Shor()` function makes calls to two other functions. The first, `ShorQPU()`, leverages our QPU (or a simulation thereof) to help find the *repeat period* of a function, while the rest of the work in `ShorLogic()` is performed with conventional software running on a CPU. We'll dive into each of these functions in more detail in the next section.

The `ShorLogic()` implementation we use in our example is for illustrative purposes only. Although simpler to explain, it will struggle with very large numbers. Full-scale Shor algorithms are the focus of ongoing research.

The remainder of the chapter walks through Shor's algorithm, presented in a standard and approachable way. Take note, however, that the implementation we present is *not* the most efficient realization of Shor's original idea, and actual implementations "in the wild" are likely to vary across QPU hardware.

What Shor's Algorithm Does

Let's start with the `ShorQPU()` function. We're going to assert without proof a useful fact from number theory.[2] Namely, it's possible to solve the problem of finding prime factors p and q of a number $N = pq$ if we are able to solve the seemingly unrelated problem of finding the repeat period of the function $a^x \bmod(N)$, as the integer variable x is varied. Here N is still the number we want to factor; a is called the *coprime*. The value of the coprime can be chosen to be any prime number that we like.

If the idea of finding the repeat period of $a^x \bmod(N)$ sounds obscure, fear not. All it means is that as you change the value of x, eventually the sequence of numbers

2 See Chapter 14 for more in-depth references.

returned by a^x mod(N) repeats itself. The repeat period is just the number of x values between repeats, as you can see in Figure 12-1.

To keep things simple, we'll choose 2 as our coprime. Besides being the smallest prime number, this choice has the advantage that our QPU implementation of a^x can be implemented by simply shifting bits. It's a good choice for the cases we cover in this chapter, but will not be appropriate in other cases.

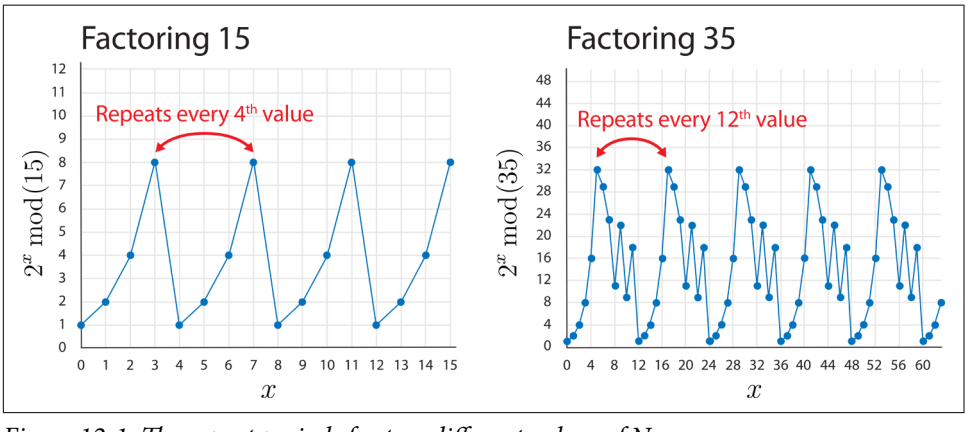


Figure 12-1. The repeat periods for two different values of N

Once we know the repeat period p, then one of N's prime factors *might* be given by $gcd\left(N, a^{p/2}+1\right)$ and the other might be given by $gcd\left(N, a^{p/2}-1\right)$. Again, these are statements that we give here without proof, but which follow from number theory arguments. Here, *gcd* is a function that returns the greatest common divisor of its two arguments. The well-known *Euclidean algorithm* can quickly work out *gcd* for us on a conventional CPU (see the online sample code at *http://oreilly-qc.github.io?p=12-1* for an implementation of *gcd*).

While we *might* be able to find the prime factors from these two *gcd* expressions, this is not guaranteed. Success depends on the chosen value of the coprime a. As previously mentioned, we have chosen 2 as our coprime for illustrative purposes, and as a result there are some numbers that this implementation will fail to factor, such as 171 or 297.

Do We Need a QPU at All?

We've reduced the problem of prime factoring to finding the periodicity p of a^x mod(N). It's actually possible to try to find p with a conventional CPU program. All

we need do is repeatedly evaluate a^x mod(N) for increasing values of x, counting how many values we have tried and keeping track of the return values we get. As soon as a return value repeats, we can stop and declare the period to be the number of values we tried.

This brute-force method for finding the period of a^x mod(N) assumes that if we get the same value back from the function, then we must have gone through one full repeat period p. Although not immediately obvious, a mathematical property of a^x mod(N) is that it can only assume any given value once within one period. Hence the moment we get the same result twice, we know we have completed a full period.

Example 12-2 implements this nonquantum, brute-force approach to finding p.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=12-2*.

Example 12-2. Factoring without a QPU

```
function ShorNoQPU(N, precision_bits, coprime) {
    // Classical replacement for the quantum part of Shor
    varwork = 1;
    var max_loops = Math.pow(2, precision_bits);
    for (var iter = 0; iter < max_loops; ++iter) {
        work = (work * coprime) % N;
        if (work == 1) // found the repeat
            return iter + 1;
    }
    return 0;
}
```

The code shown in Example 12-2 runs quickly on a whole range of numbers. The period-finding loop only needs to run until it finds the first repetition, and then it's done. So why do we need a QPU at all?

Although `ShorNoQPU()` in Example 12-2 might not *look* too costly to evaluate, the number of loops required to find the repeat pattern (which is given by the repeat period of the pattern itself) grows exponentially with the number of bits in N, as shown in Figure 12-2.

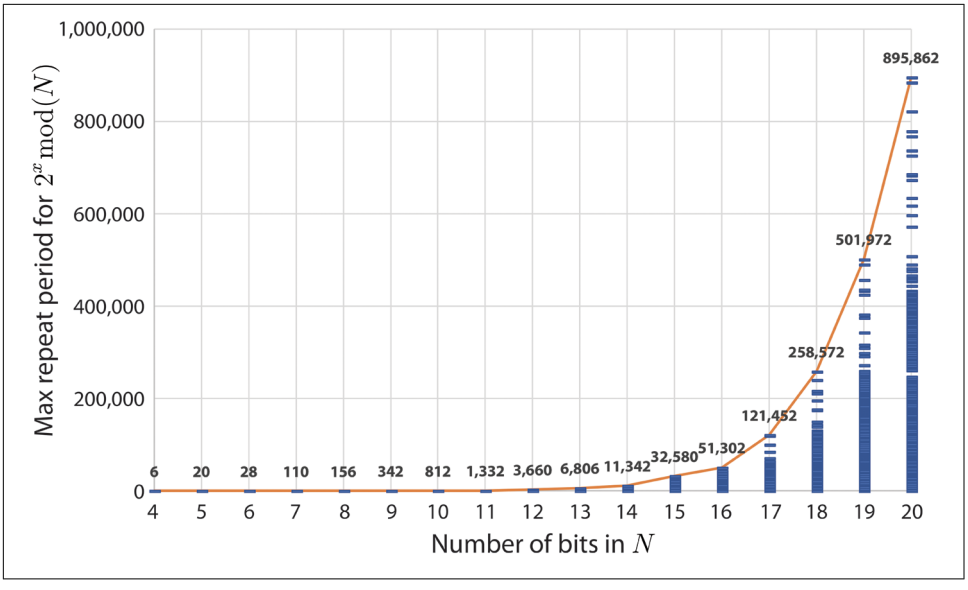


Figure 12-2. The maximum number of loops required to find the repeat period of an integer represented by a bitstring of length N. Each bar also displays a histogram showing the distribution of repeat periods for integers represented by a bitstring of length N.

There are more efficient approaches to finding prime factors on a conventional CPU (such as the *General Number Field Sieve*), but they all run into similar scaling problems as the size of N increases. The runtime of the most efficient algorithm for factoring on a conventional CPU scales exponentially with the input size, whereas the runtime of Shor's algorithm scales polynomially with the input size.

So then, let's crack open the QPU.

The Quantum Approach

Although we'll give a step-by-step walkthrough of how `ShorQPU()` works in the next section, we'll first spend a little time highlighting the creative way it makes use of the QFT.

As we'll see shortly, thanks to some initial `HAD` operations, `ShorQPU()` leaves us with $a^x \bmod(N)$ represented in the magnitudes and relative phases of our QPU register. Recall from Chapter 7 that the QFT implements the Discrete Fourier Transform, and leaves our output QPU register in a superposition of the different frequencies contained in the input.

Figure 12-3 shows what a DFT of $a^x \bmod(N)$ looks like.

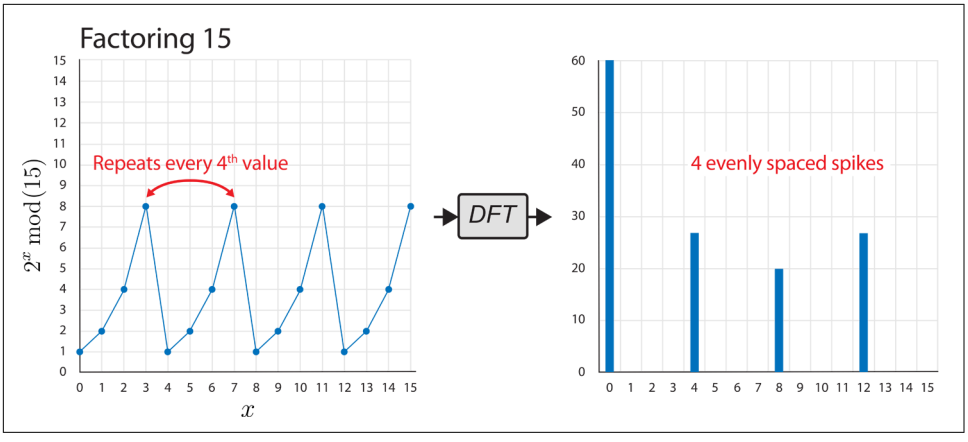


Figure 12-3. Computation performed while factoring 15

This DFT includes a spike at the correct signal frequency of 4, from which we could easily calculate p. That's all well and good for this conventional DFT, but recall from Chapter 7 that if we perform a QFT on the signal, these output peaks occur in superposition within our QPU output register. The value we obtain from the QFT after a `READ` is unlikely to yield the desired frequency.

It may look like our idea for using the QFT has fallen short. But if we play around with different values of N, we find something interesting about the DFT results for this particular kind of signal. Figure 12-4 shows the DFT of $a^x \bmod(N)$ with $N = 35$.

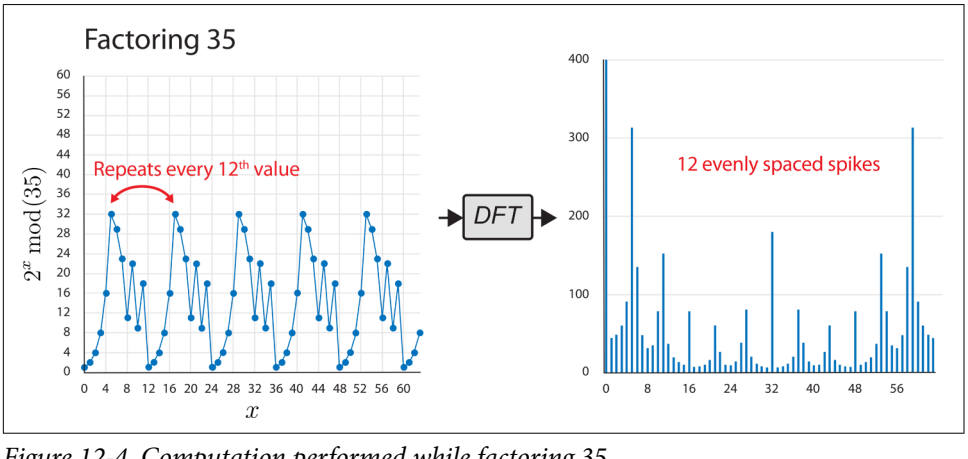


Figure 12-4. Computation performed while factoring 35

In this case, the repeat period of $a^x \bmod(N)$ is 12, and there are also 12 evenly spaced highest-weighted spikes in the DFT (these are the values we're most likely to observe in a readout after the QFT). Experimenting with different N values, we start to notice

a trend: for a pattern with repeat period p, the magnitude of the Fourier Transform will have exactly p evenly spaced spikes, as shown in the figure.

Since the spikes are evenly spaced and we know the size of our register, we can estimate how many spikes there were in the QFT—even though we can't hope to observe more than one of them. (We give an explicit algorithm for doing this shortly.) From our experimenting, we know that this number of peaks is the same as the repeat period, p, of our input signal—precisely what we're after!

This is a more indirect use of the QFT than we covered in Chapter 7, but it demonstrates a crucial point—we shouldn't be afraid to use all the tools at our disposal to experiment with what's in our QPU register. It's reassuring to see that the hallowed programmer's mantra "Change the code and see what happens" remains fruitful even with a QPU.

Step by Step: Factoring the Number 15

Let's take a step-by-step look at using a QPU to factor the number 15. (Spoiler alert: the answer is 3×5.) Our Shor program grows in complexity as we try larger numbers, but 15 is a good place to start. The walkthrough we give will use the settings in Example 12-3 to demonstrate the algorithm's operation.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=12-3*.

Example 12-3. Factoring the number 15

```
var N = 15;             // The number we're factoring
var precision_bits = 4; // See the text for a description of this
var coprime = 2;        // For this QPU implementation, this must be 2

var result = Shor(N, precision_bits, coprime);
```

The first argument, `N`, in this example is set to `15`, the number we want to factor. The second argument, `precision_bits`, is `4`. A larger number of precision bits will generally be more likely to return the correct answer, but conversely requires more qubits and many more instructions to execute. The third argument, `coprime`, will be left at `2`, which is the only value that our simplified QPU implementation of Shor's algorithm supports.

We already know that our main `Shor()` function executes two smaller functions. A QPU program determines the repeat period, and then passes the result to a second

function that determines the prime factors using conventional digital logic on a CPU. The steps taken by the constituent parts of `Shor()` are as follows:

- Steps 1–4 create a superposition of evaluations of a^x mod(N).
- Steps 5–6 implement the QFT trick outlined previously for learning the period p of this signal.
- Steps 7–8 use p in a conventional CPU algorithm to find our prime factors.

We'll walk through these steps for the case of an eight-qubit register. Four of these qubits will be used to represent (superpositions of) the different values of x that we pass to a^x mod(N), while the other four qubits will process and keep track of the values that this function returns.

This means that in total we will need to keep track of $2^8 = 256$ values—256 circles in our circle notation. By arranging all these circles into a 16×16 square, we can visualize the 16 states from each of our 4-qubit registers separately (see Chapter 3 for a recap on reading circle notation in this way). For convenience, we also add tags to these grids of 16×16 circles, showing the probabilities for each value from each of the two registers. Okay, enough talk; let's factor some numbers with a QPU!

Step 1: Initialize QPU Registers

To start the quantum part of the Shor program, we initialize the registers with the digital values 1 and 0 as shown in Figure 12-5. We'll see shortly that starting the `work` register with value 1 is necessary for the way we'll calculate a^x mod(N).

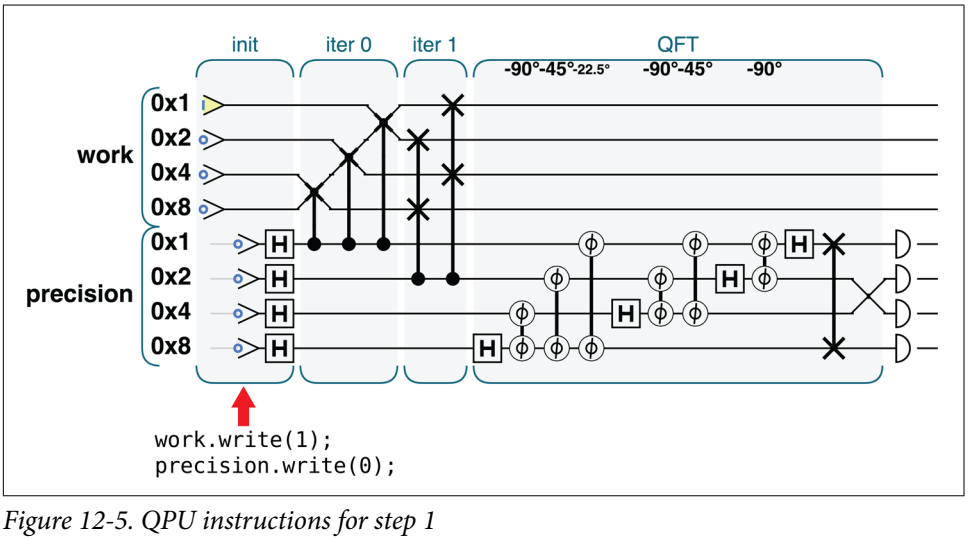


Figure 12-5. QPU instructions for step 1

Figure 12-6 shows the state of our two registers in circle notation after initialization.

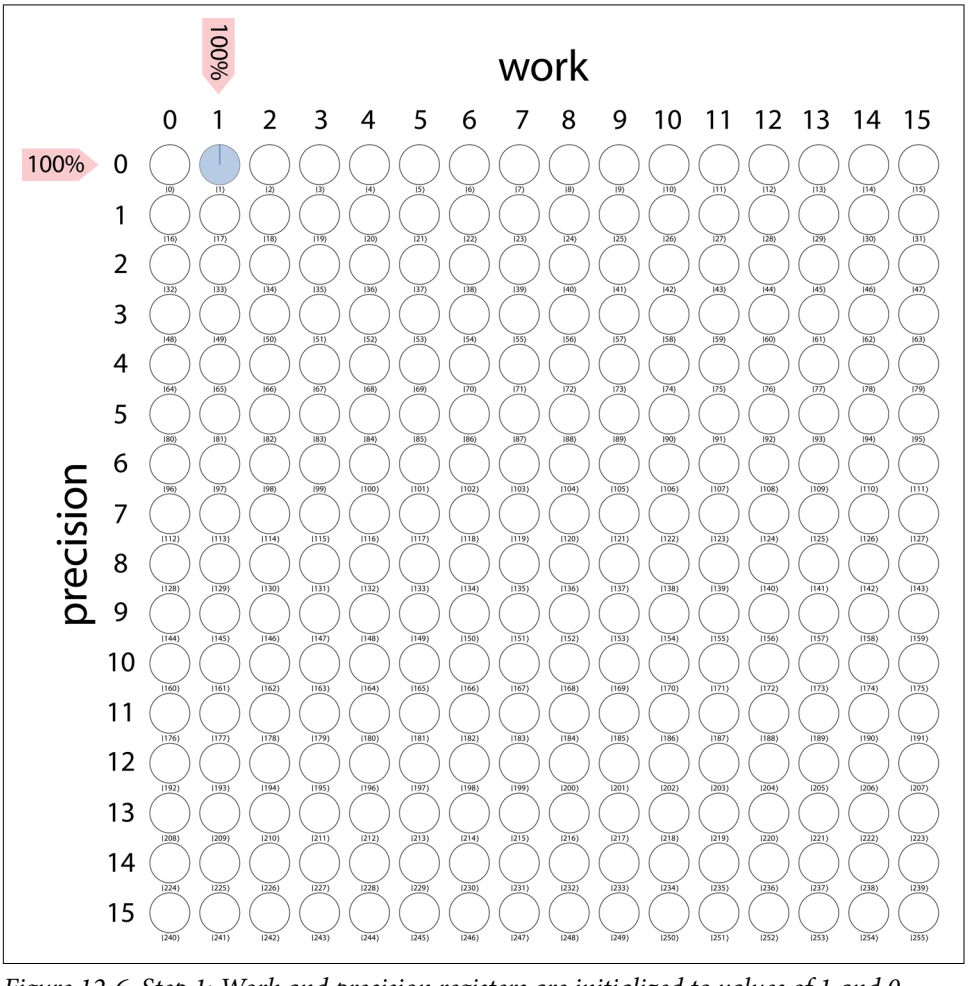


Figure 12-6. Step 1: Work and precision registers are initialized to values of 1 and 0, respectively

Step 2: Expand into Quantum Superposition

The `precision` register is used to represent the x values that we'll pass to the function $a^x \bmod(N)$. We'll use quantum superposition to evaluate this function for multiple values of x in parallel, so we apply `HAD` operations as shown in Figure 12-7 to place the `precision` register into a superposition of all possible values.

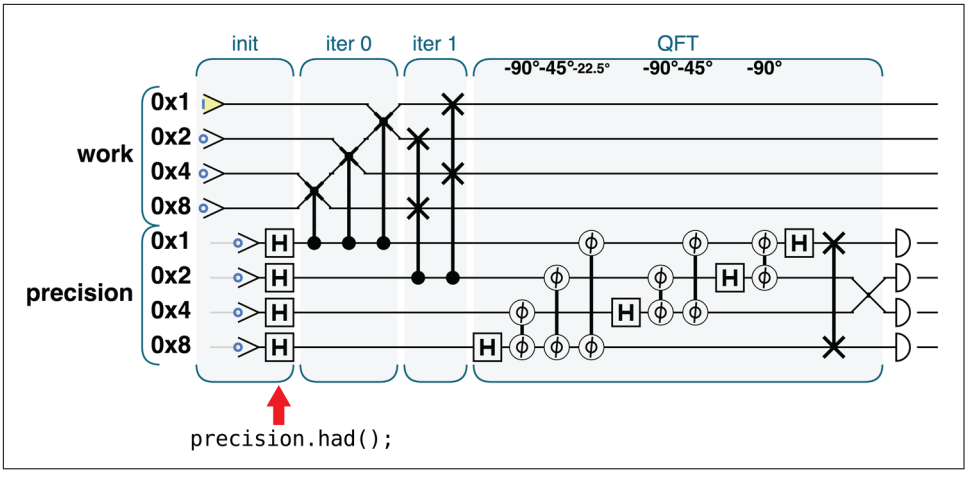


Figure 12-7. QPU instructions for step 2

This way, each row in the circle grid shown in Figure 12-8 is ready to be treated as a separate input to a parallel computation.

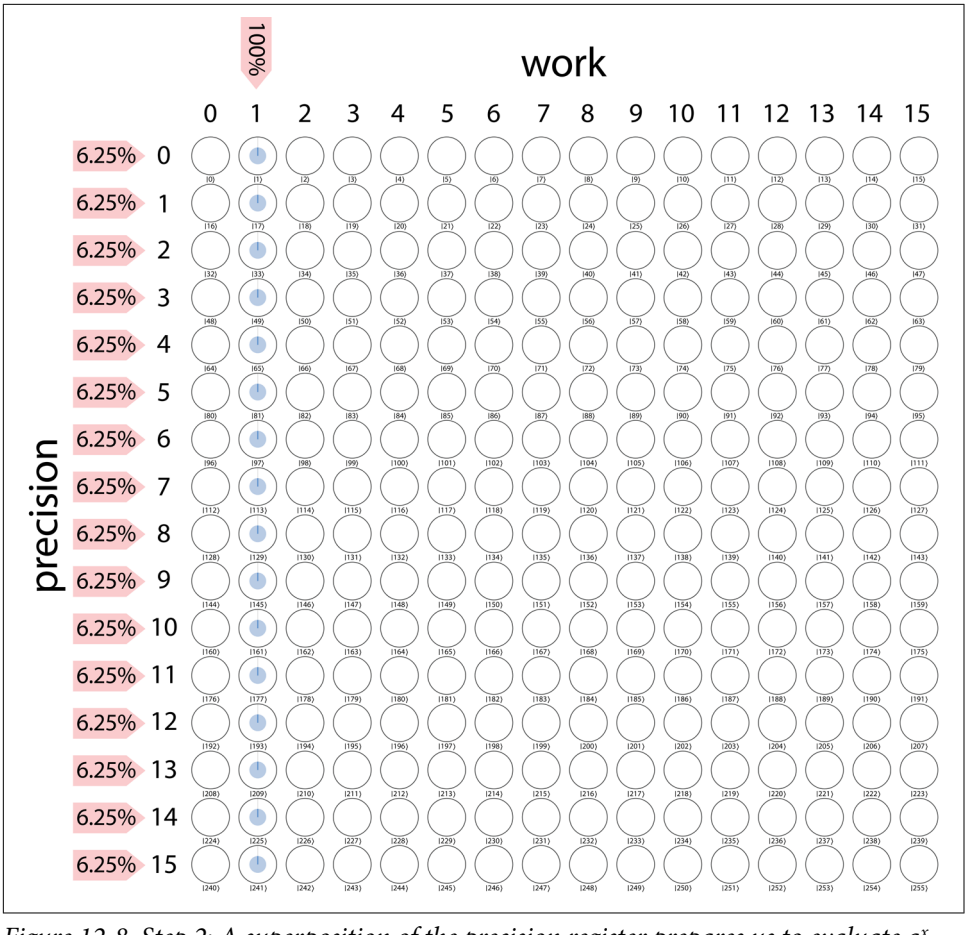


Figure 12-8. Step 2: A superposition of the precision register prepares us to evaluate a^x mod(N) in superposition

Step 3: Conditional Multiply-by-2

We now want to perform our function a^x mod(N) on the superposition of inputs we have within the `precision` register, and we'll use the `work` register to hold the results. The question is, how do we perform a^x mod(N) on our qubit register?

Recall that we have chosen $a = 2$ for our coprime value, and consequently the a^x part of the function becomes 2^x. In other words, to enact this part of the function we need to multiply the `work` register by 2. The number of times we need to perform this multiplication is equal to x, where x is the value represented in binary within our `precision` register.

Multiplication by 2 (or indeed any power of 2) can be achieved on any binary register with a simple bit shift. In our case, each qubit is exchanged with the next highest-weighted position (using the QCEngine `rollLeft()` method). To multiply by 2 a total of x times, we simply condition our multiplication on the qubit values contained in the `precision` register. Note that *we will only use the two lowest-weight qubits of the `precision` register to represent values of x* (meaning that x can take the values 0, 1, 2, 3). Consequently, we only need to condition on these two qubits.

Why is `precision` a four-qubit register if we only need two qubits for x? Although we will never directly use the additional `0x4` and `0x8` qubits that are present in the `precision` register, including them in all consequent calculations effectively *stretches out* the circle-notation patterns that we'll observe within our QPU registers. Shor's algorithm would run just fine without them, but pedagogically, it would be a little harder for us to point out the patterns explaining how the algorithm works.

If the lowest-weight qubit in `precision` has a value 1, then we will need to include one ×2 multiplication to the `work` register, and so we perform a single `rollLeft()` on `work` conditioned on this qubit's value, as shown in Figure 12-9.

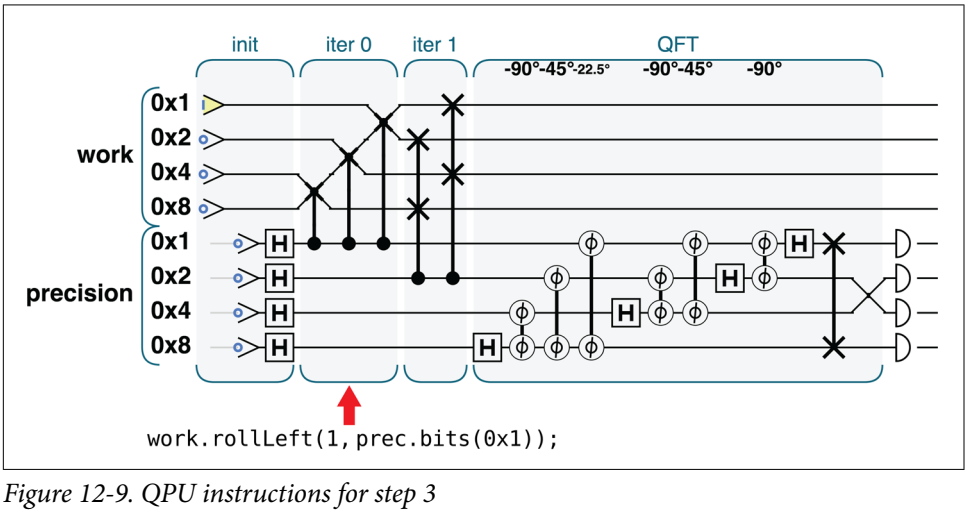


Figure 12-9. QPU instructions for step 3

The result is shown in Figure 12-10.

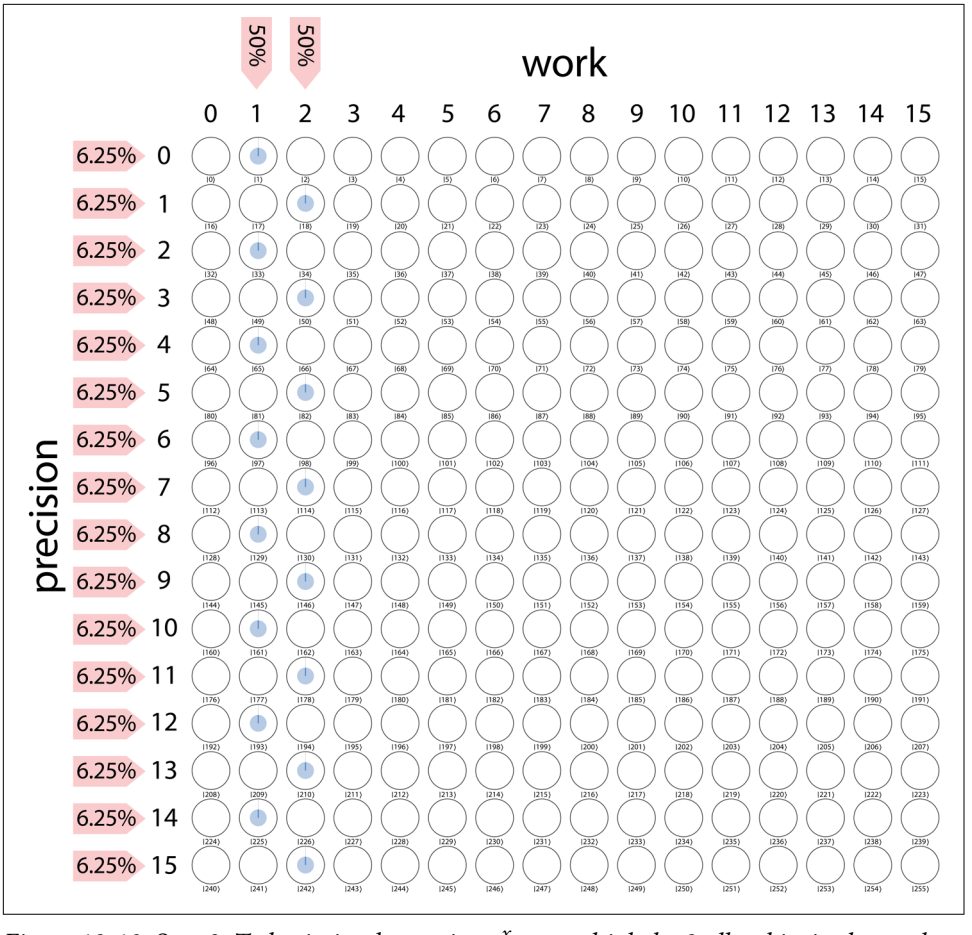


Figure 12-10. Step 3: To begin implementing a^x we multiply by 2 all qubits in the work register, conditioned on the lowest-weight qubit of precision being 1

In QPU programming, using conditional gates as the equivalent of `if/then` operations is extremely useful, as the "condition" is effectively evaluated for all possible values at once.

Step 4: Conditional Multipy-by-4

If the next-highest-weight qubit of the `precision` register is 1, then that implies a binary value of x also requiring another *two* multiplications by 2 on the `work` register. Therefore, as shown in Figure 12-11, we perform a shift by two qubits—i.e., two `rollLeft()` operations—conditional on the value of the `0x2` qubit from the `precision` register.

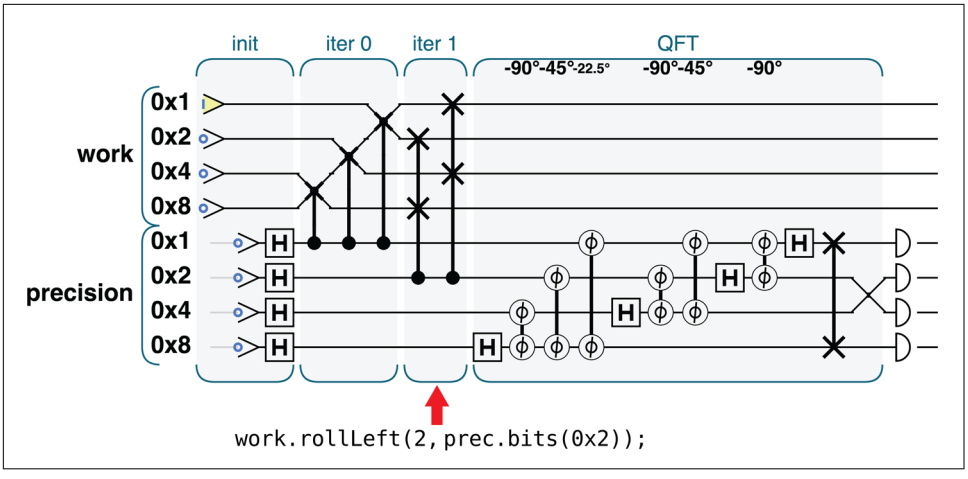


Figure 12-11. QPU instructions for step 4

We now have a value of a^x in the `work` register, for whatever value of x is encoded in the (first two) qubits of the `precision` register. In our case, `precision` is in a uniform superposition of possible x values, so we will obtain a corresponding superposition of associated a^x values in `work`.

Although we've performed all the necessary multiplications by 2, it may seem like we've fallen short of implementing the function $a^x \bmod(N)$ by taking no action to take care of the *mod* part of the function. In fact, for the particular example we've considered, our circuit manages to take care of the modulus automatically. We'll explain how in the next section.

Figure 12-12 shows how we've now managed to compute $a^x \bmod(N)$ on every value of x from the `precision` register in superposition.

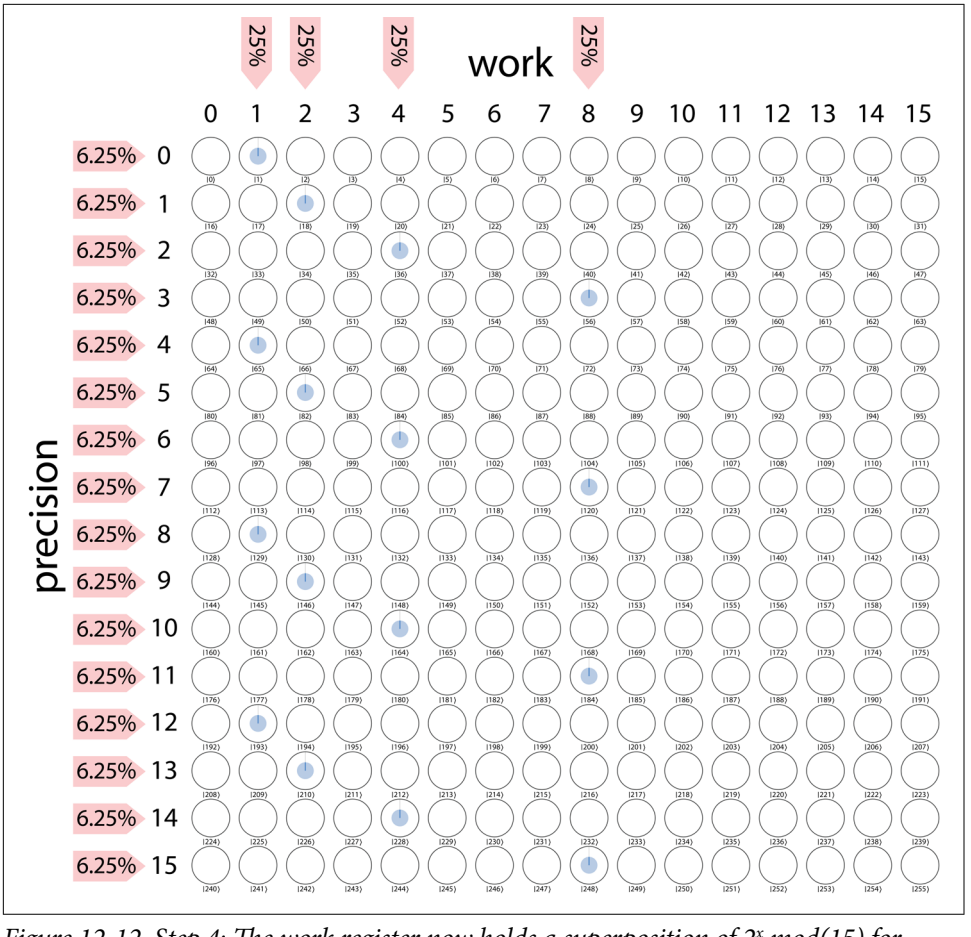


Figure 12-12. Step 4: The work register now holds a superposition of 2^x mod(15) for every possible value of x in the precision register

The preceding circle notation shows a familiar pattern. Having performed the function a^x mod(N) on our QPU register, the superposition amplitudes exactly match the plot of a^x mod(N) that we first produced at the beginning of the chapter in Figure 12-1 (albeit with the axes rotated 90°). We highlight this in Figure 12-13.

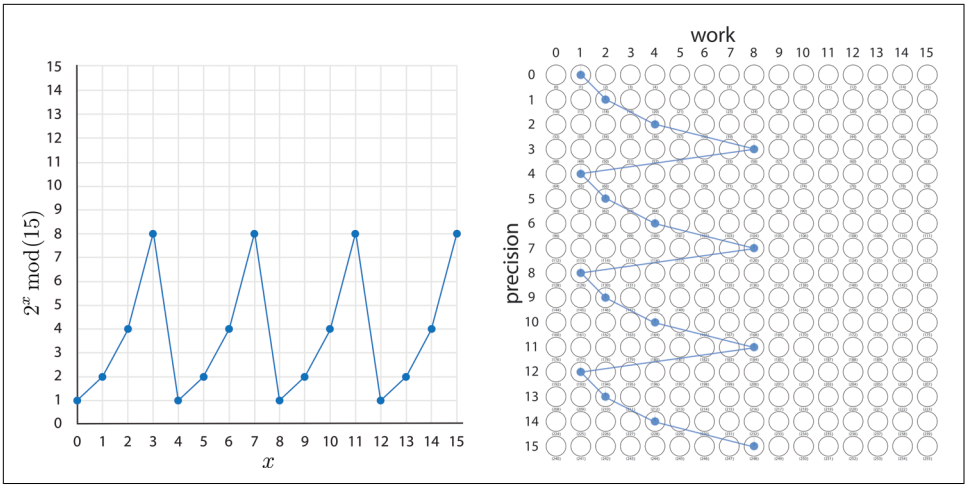


Figure 12-13. Hey, this looks familiar!

We now have the repeating signal for $a^x \bmod(N)$ encoded into our QPU registers. Trying to read either register now will, of course, simply return a random value for the `precision` register, along with the corresponding `work` result. Luckily, we have the DFT-spike-counting trick up our sleeves for finding this signal's repeat period. It's time to use the QFT.

Step 5: Quantum Fourier Transform

By performing a QFT on the `precision` register as shown in Figure 12-14, we effectively perform a DFT on each column of data, transforming the `precision` register states (shown as the rows of our circle-notation grid) into a superposition of the periodic signal's component frequencies.

Looking at the periodic pattern shown in the circle notation of Figure 12-12, you might wonder why we don't need to QFT both registers (after all, it was the `work` register that we applied $a^x \bmod(N)$ to!). Pick a `work` value with nonzero amplitude from the circle-notation grid and look up and down the circles of that column. You should clearly see that as the `precision` register value is varied, we get a periodic variation of amplitude (with a period of 4 in this case). This is the register for which we therefore want to find the QFT.

Reading from top to bottom, Figure 12-15 now resembles the DFT plot we originally saw toward the beginning of the chapter in Figure 12-3, as we highlight in

Figure 12-16. Note in Figure 12-15 that the QFT has also affected the relative phases of the register amplitudes.

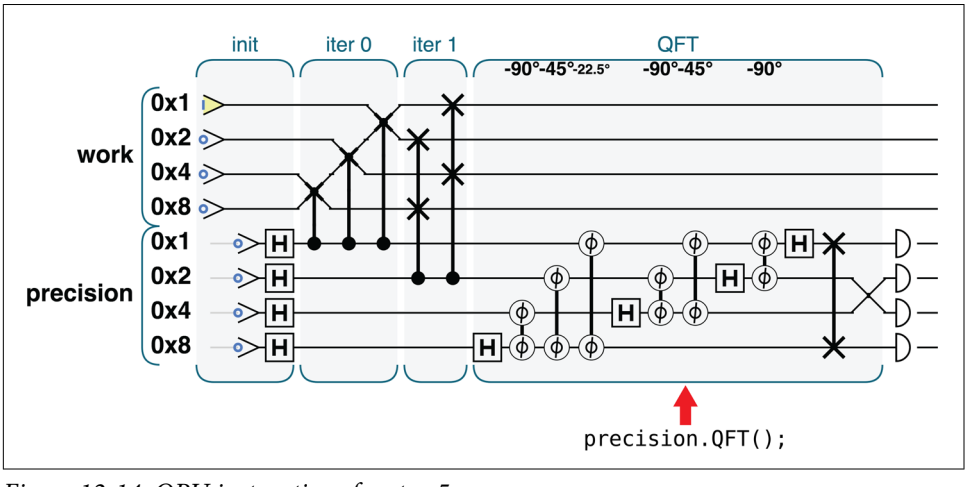


Figure 12-14. QPU instructions for step 5

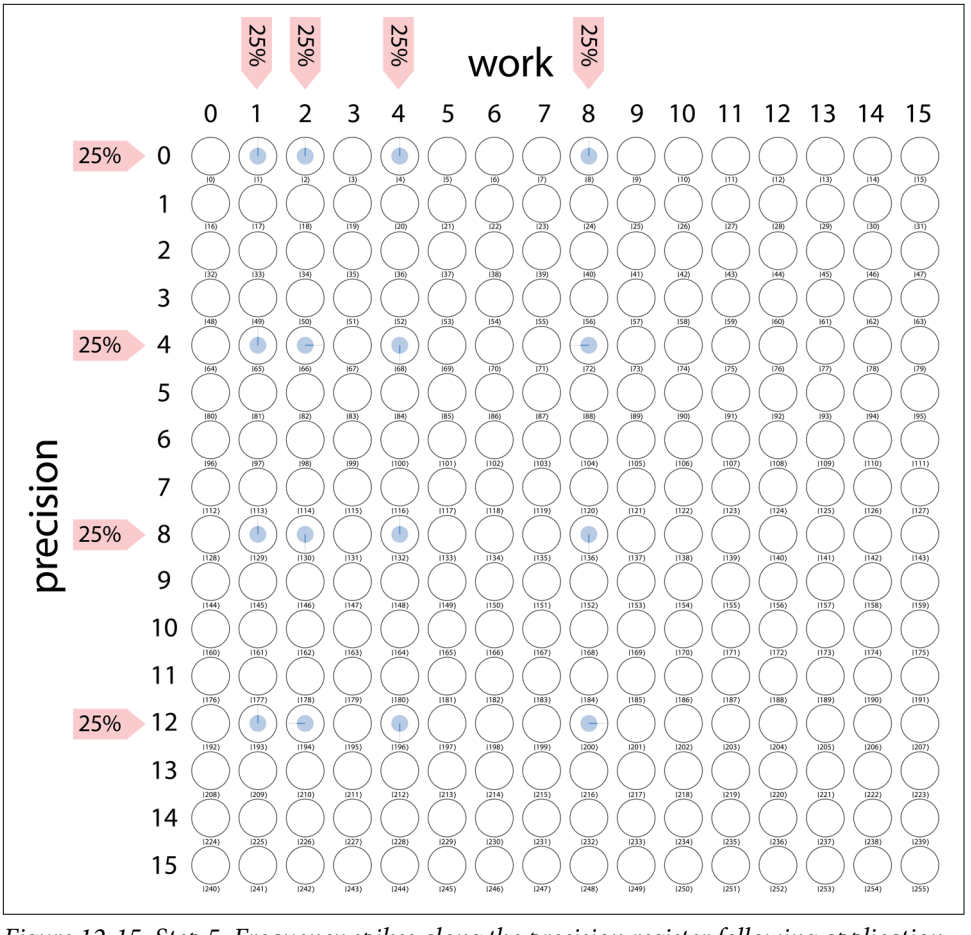


Figure 12-15. Step 5: Frequency spikes along the precision register following application of the QFT

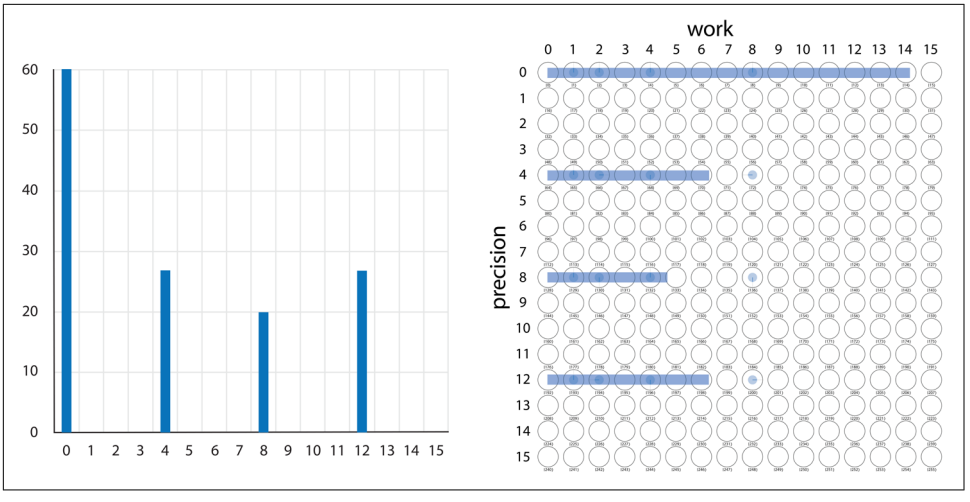


Figure 12-16. Once again, this resembles something we've seen

Each column in Figure 12-16 now contains the correct number of frequency spikes (four in this case). Recall from our earlier discussion that if we can count these spikes, that's all we need to find the factor we seek via some conventional digital logic. Let's `READ` out the `precision` register to get the information we need.

Step 6: Read the Quantum Result

The `READ` operation used in Figure 12-17 returns a random digital value, weighted by the probabilities in the circle diagram. Additionally, the `READ` destroys all values from the superposition that are in disagreement with the observed digital result.

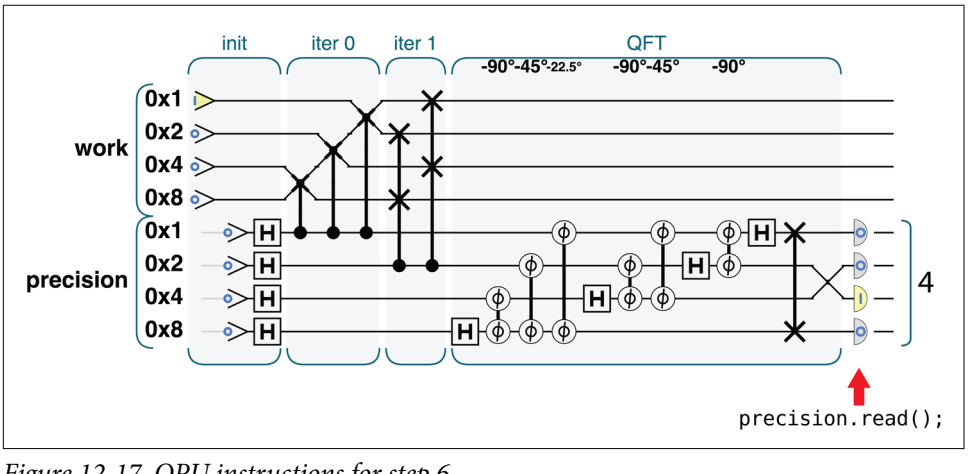


Figure 12-17. QPU instructions for step 6

In the example readout shown in Figure 12-18, the number 4 has been randomly obtained from the four most probable options. The QPU-powered part of Shor's algorithm is now finished, and we hand this `READ` result to the conventional logic function `ShorLogic()` used in the next step.

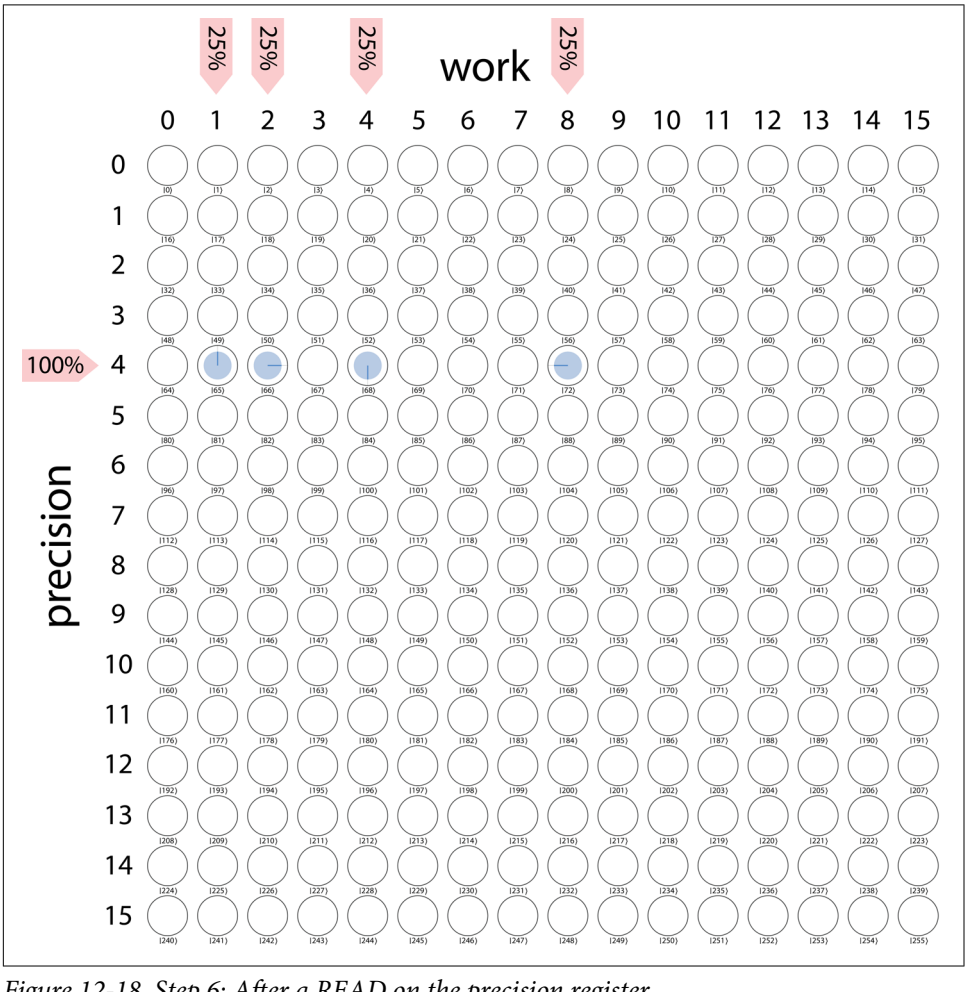


Figure 12-18. Step 6: After a READ on the precision register

Step 7: Digital Logic

Our work up until this point has produced the number 4, although looking back at Figure 12-15, we could equally have randomly received the results 0, 4, 8, or 12.

As noted earlier, given our knowledge that the QFT spikes are evenly distributed across the register, we can determine what periods are consistent with our `READ` value using conventional digital logic. The `estimate_num_spikes()` function in

Example 12-4 explicitly shows the logic for doing this. In some cases, this function may return more than one candidate number of spikes for the DFT plot. For example, if we pass it our `READ` value of 4, then it returns the two values 4 and 8, either of which is a number of spikes in the DFT consistent with our `READ` value.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=12-4*.

Example 12-4. A function to estimate the number of spikes based on the QPU result

```
function estimate_num_spikes(spike, range)
{
    if (spike < range / 2)
        spike = range - spike;
    var best_error = 1.0;
    var e0 = 0, e1 = 0, e2 = 0;
    var actual = spike / range;
    var candidates = []
    for (var denom = 1.0; denom < spike; ++denom)
    {
        var numerator = Math.round(denom * actual);
        var estimated = numerator / denom;
        var error = Math.abs(estimated - actual);
        e0 = e1;
        e1 = e2;
        e2 = error;
        // Look for a local minimum which beats our
        // current best error
        if (e1 <= best_error && e1 < e0 && e1 < e2)
        {
            var repeat_period = denom - 1;
            candidates.push(denom - 1);
            best_error = e1;
        }
    }
    return candidates;
}
```

Since (in this example) we've ended up with two candidate results (4 and 8), we'll need to check both to see whether they give us prime factors of 15. We introduced the method `ShorLogic()`, implementing the *gcd* equations that determine prime factors from a given number of DFT spikes back at the start of the chapter, in Example 12-1. We first try the value 4 in this expression, and it returns the values 3 and 5.

Not all of the available values will lead to a correct answer. What happens if we receive the value 0? This is 25% likely, and in this case the `estimate_num_spikes()` function returns no candidate values at all, so the program fails. This is a common situation with quantum algorithms, and not a problem when we can check the validity of our answer quickly. In this case, we make such a check and then, if necessary, run the program again, from the beginning.

Step 8: Check the Result

The factors of a number can be difficult to find, but once found the answer is simple to verify. We can easily verify that 3 and 5 are both prime and factors of 15 (so there's no need to even try checking the second value of 8 in `ShorLogic()`).

Success!

The Fine Print

This chapter presented a simplified version of what is typically a very complicated algorithm. In our version of Shor's algorithm, a few aspects were simplified for ease of illustration, although at the cost of generality. Without digging too deeply, we here mention some of the necessary simplifications. More information can also be found in the online sample code.

Computing the Modulus

We already mentioned that in our QPU calculation of a^x mod(N) the modulus part of the function was somehow automatically taken care of for us. This was a happy coincidence specific to the particular number we wanted to factor, and sadly won't happen in general. Recall that we multiplied the `work` register by powers of two through the rolling of bits. If we simply start with the value 1 and then shift it 4 times, we should get $2^4 = 16$; however, since we only used a 4-bit number and allowed the shifted bits to wrap around, instead of 16 we get 1, which is precisely what we should get if we're doing the multiplications followed by mod(15).

You can verify that this trick also works if we're trying to factor 21; however, it fails on a larger (but still relatively tiny) number such as 35. What can we do in these more general cases?

When a conventional computer calculates the modulus of a number, such as `1024 % 35`, it performs integer division and then returns the remainder. The number of conventional logic gates required to perform integer division is very (*very*) large, and a QPU implementation is well beyond the scope of this book.

Nevertheless, there is a way we can solve the problem, by using an approach to calculating the modulus that, although less sophisticated, is well suited to QPU operations. Suppose we wanted to find y mod(N) for some value y. The following code will do the trick:

```
y -= N;
if (y < 0) {
    y += N;
}
```

We simply subtract N from our value, and use the sign of the result to determine whether we should allow the value to wrap around or return it to its original value. On a conventional computer this might be considered bad practice. In this case, it gives us exactly what we need: the correct answer, using decrements, increments, and comparisons—all logic we know can be implemented with QPU operations from our discussions in Chapter 5.

A circuit for performing the modulus by this method (on some value `val`) is shown in Figure 12-19.

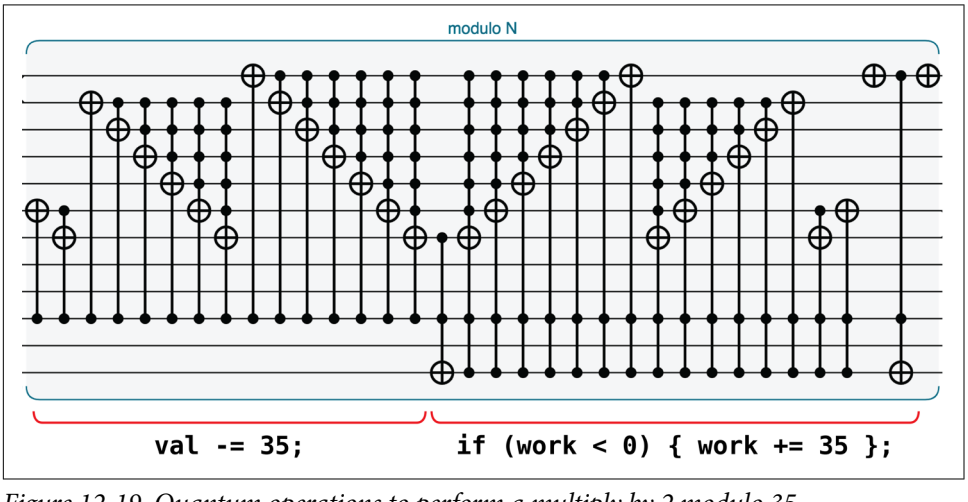


Figure 12-19. Quantum operations to perform a multiply by 2 modulo 35

While this example does use a large number of complicated operations to perform a simple calculation, it is able to perform the modulus in superposition.

The modulus implementation in Figure 12-19 actually uses one additional scratch qubit to stash away the sign bit of the `work` register, for use in the conditional addition.

Time Versus Space

The modulus operation described in the preceding section slows things down considerably for the general factoring case, primarily because it requires the multiply-by-2 operations to be performed one at a time. This increase in the number of operations needed (and therefore overall operation time) destroys our QPU's advantage. This can be solved by increasing the number of qubits used, and then applying the modulus operation a logarithmic number of times. For an example of this, see *http://oreilly-qc.github.io?p=12-A*. Much of the challenge of QPU programming involves finding ways to balance program *depth* (the number of operations) against required number of qubits.

Coprimes Other Than 2

The implementation covered here will factor many numbers, but for some it returns undesired results. For example, using it to factor 407 returns `[407, 1]`. While this is technically correct, we would much prefer the *nontrivial* factors of 407, which are `[37, 11]`.

A solution to this problem is to replace our `coprime=2` with some other prime number, although the quantum operations required to perform non-power-of-2 exponentiation are outside the scope of this book. The choice of `coprime=2` is a useful illustrative simplification.

CHAPTER 13
Quantum Machine Learning

At the time of writing, quantum machine learning (QML) is just about the greatest combination of buzzwords you could hope to synthesize. A lot is written about QML, and the topic is often (confusingly) both overhyped and undersold at the same time. In this section we'll try to give a flavor for how QPUs might transform machine learning, while also being careful to point out the caveats inherent in manipulating quantum data.

Useful QML applications require very large numbers of qubits. For this reason, our overview of QML applications is necessarily very high-level. Such a summary is also fitting given the rapidly changing nature of this nascent field. Although our discussion will be more schematic than pragmatic, it will heavily leverage our hands-on experience of primitives from earlier chapters.

We summarize three different QML applications: *solving systems of linear equations*, *Quantum Principal Component Analysis*, and *Quantum Support Vector Machines*. These have been selected due to both their relevance to machine learning and their simplicity to discuss. These are also applications whose conventional counterparts are hopefully familiar to anyone who has dabbled in machine learning. We only give a brief description of the conventional progenitors of each QML application as it's introduced.

In discussing QML, we'll frequently make use of the following pieces of machine-learning terminology:

Features
: Term used to describe measurable properties of data points available to a machine-learning model for making predictions. We often imagine the possible values of these features to define a *feature space*.

Supervised
: Refers to machine-learning models that must be *trained* on a collection of points in feature space for which correct classes or responses are already known. Only then can the adequately trained model be used to classify (or predict) responses for new points in the feature space.

Unsupervised
: Refers to machine-learning models that are able to learn patterns and structure in training data that does not include known responses.

Classification
: Used to describe supervised predictive models that assign a given point in feature space to one of several discrete classes.

Regression
: Used to describe supervised models predicting some continuously varying *response variable*.

Dimensionality reduction
: One form of unsupervised data preprocessing that can benefit machine-learning models of all types. Dimensionality reduction aims to reduce the number of features needed to describe a problem.

In addition to this terminology, we'll also make use of mathematical descriptions of machine-learning problems. As such, this chapter is slightly more mathematically involved than our previous discourse.

Our first QML application teaches us how a QPU can help solve systems of linear equations.

Solving Systems of Linear Equations

Although systems of linear equations are certainly fundamental to much of machine learning, they also underlie vast areas across all of applied mathematics. The *HHL algorithm* [1] (often referred to simply as HHL) we present for leveraging a QPU to efficiently solve these systems is consequently a fundamental and powerful tool, and we'll see that it's a key building block in *other* QML applications too. HHL has also been considered for applications ranging from modeling electrical effects to streamlining computer graphics calculations.

We begin our summary of the HHL algorithm by recapping the mathematics needed to describe conventional systems of linear equations. We then summarize the distinctly quantum operation of HHL, outlining its performance improvements and—

1 Harrow et al., 2009.

just as importantly—its constraints. Finally, we give a more detailed description of how HHL works "inside the box."

Describing and Solving Systems of Linear Equations

The most concise way of representing a system of linear equations is in terms of matrix multiplication. In fact, for the seasoned equation solver the terms *matrices* and *linear equations* are synonymous. For example, suppose we have a system of two linear equations, $3x_1 + 4x_2 = 3$ and $2x_1 + x_2 = 3$. We can equivalently, and much more concisely, represent these as the single matrix equation shown in Equation 13-1.

Equation 13-1. Using matrices to describe systems of linear equations

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

We can recover our two linear equations through the rules of matrix multiplication. More generally, a system of n linear equations for n variables can be written as a matrix equation containing an $n \times n$ matrix, $\mathbf{A}$, and an n-dimensional vector $\vec{b}$:

$$\mathbf{A}\vec{x} = \vec{b}$$

Here we have also introduced a vector $\vec{x} = [x_1, ..., x_n]$ of the n variables we want to solve for.

In this matrix formulation, the task of solving the system of equations boils down to being able to invert the matrix $\mathbf{A}$. If we can obtain the inverse $\mathbf{A}^{-1}$, then we can easily determine the n unknown variables via Equation 13-2.

Equation 13-2. Solving systems of linear equations by inverting matrices

$$\vec{x} = \mathbf{A}^{-1}\vec{b}$$

Many conventional algorithms exist for finding the inverse of a matrix, and the most efficient rely on the matrix in question possessing certain helpful properties.

The following matrix parameters can affect the performance of both conventional and quantum algorithms:

n

: The *size of the system of linear equations*. Equivalently, this is the dimensionality of $\mathbf{A}$—if we want to solve a system of n linear equations to find n variables, then $\mathbf{A}$ will be an $n \times n$ matrix.

κ

: The *condition number* of the matrix $\mathbf{A}$ representing the system of linear equations. Given the system of equations $\mathbf{A}\vec{x} = \vec{b}$, the condition number tells us how much an error in our specification of $\vec{b}$ affects the error we can expect to find in our solution for $\vec{x} = \mathbf{A}^{-1}\vec{b}$. κ is calculated as the maximum ratio between the relative error in the input $\vec{b}$ and the output $\vec{x} = \mathbf{A}^{-1}\vec{b}$. It turns out that κ can equivalently be found as[2] the ratio $|\lambda_{max}|/|\lambda_{min}|$ of the absolute values of the maximum and minimum eigenvalues of $\mathbf{A}$.

s

: The *sparsity*of the matrix $\mathbf{A}$. This is the number of nonzero entries in $\mathbf{A}$.

ϵ

: The *precision* we require in our solution. In the case of HHL, we'll shortly see that a state $|\vec{x}\rangle$ is output that amplitude-encodes the solution vector $\vec{x}$. Increasing ϵ means increasing the precision with which the values in $\vec{x}$ are represented within these amplitudes.

Our assessment of how efficiently HHL can invert a matrix will be with respect to these parameters. By *efficiency* we mean the *runtime* of the algorithm (a measure of how many fundamental operations it must employ). For comparison, at the time of writing, the leading conventional algorithm for solving systems of linear equations is probably the *conjugate gradient descent* method. This has a runtime of $O(ns\kappa \log(1/\epsilon))$.

Solving Linear Equations with a QPU

HHL (named after its 2009 discoverers Harrow, Hassidim, and Lloyd) employs the primitives we've learned thus far to find (in a particular sense) the inverse of a matrix faster than is possible with conjugate gradient descent. We say *in a particular sense* because HHL solves a distinctly *quantum* version of the problem. HHL provides the solutions to a system of linear equations amplitude-encoded in a QPU register, and as such, they are inaccessibly quantum. Although HHL cannot solve systems of linear

2 The expression for condition number in terms of eigenvalues only holds if $\mathbf{A}$ is a *normal* matrix. All matrices used in HHL are necessarily normal because of its reliance on quantum simulation and Hermitian matrices.

equations in a conventional sense, amplitude-encoded solutions can still be very useful, and are in fact critical building blocks in other QML applications.

What HHL does

Before decomposing HHL into primitives, we'll give an executive summary of its inputs, outputs, and performance.

The inputs and outputs of HHL are as shown in Figure 13-1.

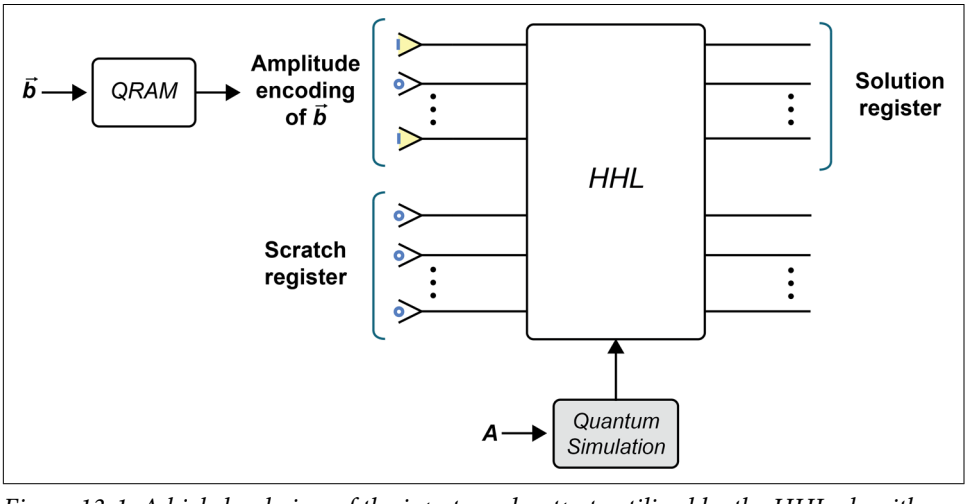


Figure 13-1. A high-level view of the inputs and outputs utilized by the HHL algorithm to solve systems of linear equations.

Inputs. This schematic shows that HHL accepts two sets of input registers, and a matrix (via a quantum simulation):

Scratch register

This contains a number of scratch qubits used by various primitives within HHL, all prepared in the $|0\rangle$ state. Because HHL deals with fixed-point (or floating-point) data and involves nontrivial arithmetic operations (such as taking the square root), we require a *lot* of scratch qubits. This makes HHL difficult to simulate for even the simplest cases.

Amplitude encoding of $\vec{b}$

We also need to provide HHL with the vector $\vec{b}$ from Equation 13-2, amplitude-encoded in a QPU register (in the sense we discussed in Chapter 9). We will denote the state of a register amplitude encoding $\vec{b}$ as $|\vec{b}\rangle$. Note that to prepare

an amplitude encoding of $\vec{b}$ we will need to use QRAM. Thus, HHL fundamentally relies on the existence of QRAM.

QPU operation representing ***A***

Naturally, HHL also needs access to the matrix encapsulating the system of linear equations we wish to solve. The bottom of Figure 13-1 illustrates how HHL requires us to represent **A** as a QPU operation, which we can achieve via the process of quantum simulation outlined in Chapter 9. This means that the matrix **A** must meet the requirements we noted for performing quantum simulation.

Outputs. Figure 13-1 also shows that two registers are output from HHL.

Solution register

The solution vector $\vec{x}$ is amplitude-encoded within a single output QPU register (we denote this state as $|\vec{x}\rangle$). As we've already stressed, this implies that *we cannot access the individual solutions*, since they're *hidden* in the amplitudes of a quantum superposition that we cannot hope to efficiently extract with `READ` operations.

Scratch register

The scratch qubits are returned to their starting state of $|0\rangle$, allowing us to continue using them elsewhere in our QPU.

Here are some examples of ways in which the quantum output from HHL can still be incredibly useful despite its inaccessible nature:

1. Rather than a specification of solutions for all n variables in $\vec{x}$, we may only wish to know some derived property, such as their sum, mean value, or perhaps even whether or not they contain a certain frequency component. In such cases we may be able to apply an appropriate quantum circuit to $|\vec{x}\rangle$, allowing us to `READ` the derived value.
2. If we are satisfied with checking only whether or not the solution vector $\vec{x}$ is equal to one particular suspected vector, then we can employ the *swap test* introduced in Chapter 3 between $|\vec{x}\rangle$ and another register encoding the suspected vector.
3. If, for example, we plan to use the HHL algorithm as a component in a larger algorithm, $|\vec{x}\rangle$ may be sufficient for our needs as is.

Since systems of linear equations are fundamental in many areas of machine learning, HHL is the starting point for many other QML applications, such as regression[3] and data fitting.[4]

Speed and fine print. The HHL algorithm has a runtime[5] of $O\left(\kappa^2 s^2 \epsilon^{-1} \log n\right)$.

In comparison with the conventional method of conjugate gradient descent and its runtime of $O(ns\kappa \log(1/\epsilon))$, HHL clearly offers an exponential improvement in the dependence on the size of the problem (n).

One could argue that this is an unfair comparison, since conventional conjugate gradient descent reveals the full set of solutions to us, unlike the quantum answer generated by HHL. We could instead compare HHL to the best conventional algorithms for determining derived statistics from solutions to systems of linear equations (sum, mean, etc.), which have n and κ dependencies of $O(n\sqrt{\kappa})$, but HHL still affords an exponential improvement in the dependence on n.

Although it's tempting to focus solely on how algorithms scale with the problem size n, other parameters are equally important. Though offering an impressive exponential speedup in terms of n, HHL's performance is worse than conventional competitors once we consider poorly conditioned or less sparse problems (where κ or s becomes important).[6] HHL also suffers if we demand more precision and place importance on the parameter ϵ.

For these reasons, we have the following fine print:

> The HHL algorithm is suited to solving systems of linear equations represented by sparse, well-conditioned matrices.

Additionally, since HHL leverages a quantum simulation primitive, we need to take note of any requirements particular to whichever quantum simulation technique we employ.

Having done our best to give a realistic impression of HHL's usage, let's break down how it works.

3 Kerenidis and Prakash, 2017.

4 Wiebe et al., 2012.

5 Several improvements and extensions have been made to the original HHL algorithm, resulting in runtimes with different trade-offs between the various parameters. Here we focus on the conceptually simpler original algorithm.

6 A more recent result by Childs, et al., 2015, actually manages to improve the dependence of HHL on ϵ to $poly(\log(1/\epsilon))$.

Inside the box

The intuition behind HHL relies on one particular method for finding the inverse of a matrix via its eigendecomposition. Any matrix has an associated set of eigenvectors and eigenvalues. Since our focus in this chapter is machine learning, we'll assume some familiarity with this concept. If the idea is new to you, eigenvectors and eigenvalues are essentially the matrix equivalents of the eigenstates and eigenphases of QPU operations that we introduced while discussing phase estimation in Chapter 8.

In "Phase Estimation in Practice" on page 163 we also noted that any QPU register state can be considered to be some superposition of the eigenstates of any QPU operation. Through its dependence on quantum simulation, HHL is restricted to solving systems of linear equations that are represented by *Hermitian* matrices.[7] For such matrices a similar fact regarding its eigendecomposition is true. Any vector we might want to act a Hermitian matrix **A** on can be expressed in the basis of (i.e., written as a linear combination of) **A**'s eigenvectors.

For example, consider the Hermitian matrix **A** and vector $\vec{z}$ shown in Equation 13-3.

Equation 13-3. Example matrix and vector for introducing eigendecomposition

$$\mathbf{A} = \begin{bmatrix} 2 & 2 \\ 2 & 3 \end{bmatrix}, \quad \vec{z} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The two eigenvectors of this particular matrix **A** are $\vec{v}_1 = [-0.7882, 0.615]$ and $\vec{v}_2 = [-0.615, -0.788]$, with associated eigenvalues $\lambda_1 = 0.438$ and $\lambda_2 = 4.56$, as you can check by confirming that $\mathbf{A}\vec{v}_1 = \lambda_1\vec{v}_1$ and $\mathbf{A}\vec{v}_2 = \lambda_2\vec{v}_2$. Since the example **A** considered here is Hermitian, $\vec{z}$ can be written in the basis of its eigenvectors, and in fact $\vec{z} = -0.788\vec{v}_1 - 0.615\vec{v}_2$. We could simply write this as $\vec{z} = [-0.788, -0.615]$ with the understanding that the components are expressed in the basis of **A**'s eigenvectors.

7 However, as discussed in Chapter 9, we can always extend an $n \times n$ non-Hermitian matrix to a $2n \times 2n$ Hermitian one.

We can also write **A** in its own *eigenbasis*.[8] It turns out that when written this way a matrix is always diagonal, with its main diagonal consisting of its eigenvalues. For our preceding example, **A** is therefore written in its eigenbasis as shown in Equation 13-4.

Equation 13-4. Writing a matrix in its eigenbasis

$$\mathbf{A} = \begin{bmatrix} 0.438 & 0 \\ 0 & 4.56 \end{bmatrix}$$

Expressing **A** in its eigenbasis is very helpful for finding its inverse, because inverting a diagonal matrix is trivial. To do so you simply numerically invert the nonzero values along its diagonal. For example, we can find $\mathbf{A}^{-1}$ as shown in Equation 13-5.

Equation 13-5. Inverting a matrix written in its eigenbasis

$$\mathbf{A}^{-1} = \begin{bmatrix} \frac{1}{0.438} & 0 \\ 0 & \frac{1}{4.56} \end{bmatrix} = \begin{bmatrix} 2.281 & 0 \\ 0 & 0.219 \end{bmatrix}$$

We should note, of course, that this gives us $\mathbf{A}^{-1}$ *expressed in the eigenbasis* of **A**. We can either leave the inverse like this (if we wish to act it on vectors that are also expressed in **A**'s eigenbasis), or rewrite it in the original basis.

So in summary, if we can find the eigenvalues of some general matrix **A**, then we can determine $\vec{x} = \mathbf{A}^{-1}\vec{b}$ as shown in Equation 13-6.

Equation 13-6. General approach for determining the inverse of a matrix via its eigendecomposition

$$\vec{x} = \begin{bmatrix} \frac{1}{\lambda_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\lambda_n} \end{bmatrix} \begin{bmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_n \end{bmatrix} = \begin{bmatrix} \frac{1}{\lambda_1}\tilde{b}_1 \\ \vdots \\ \frac{1}{\lambda_n}\tilde{b}_n \end{bmatrix}$$

Where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of **A** and we use $\tilde{b}_1, \ldots, \tilde{b}_n$ to denote the components of $\vec{b}$ expressed in **A**'s eigenbasis.

8 By this we mean finding the elements **A** must have in order to correctly relate vectors expressed in its eigenbasis.

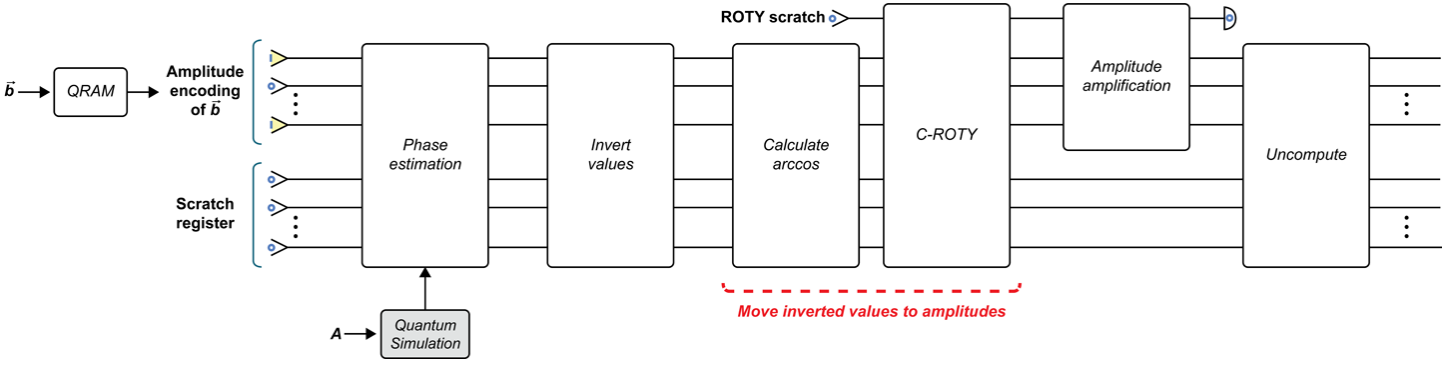


Figure 13-2. Schematic outline of the primitives contained in the HHL algorithm

HHL manages to employ this matrix inversion method using the quantum parallelism of a QPU. The output register from HHL contains an amplitude encoding of precisely the vector shown in Equation 13-6; i.e., one where the amplitude of state $|i\rangle$ is $\tilde{b}_i/\lambda_i$. The schematic in Figure 13-2 shows what's inside the HHL box from Figure 13-1, outlining how HHL uses our familiar QPU primitives to produce the output state in Equation 13-6.

Although we don't explicitly show them in Figure 13-2, HHL will also require other input registers specifying certain configuration parameters needed for the quantum simulation of **A**.

Let's step through each of the primitive components used in Figure 13-2.

1. Quantum simulation, QRAM, and phase estimation. We already know that the phase estimation primitive can efficiently find the eigenstates and eigenphases of a QPU operation. You might suspect that this could help us in using the eigendecomposition approach to inverting a matrix—and you'd be right!

Figure 13-2 shows that we begin by using QRAM to produce a register with an amplitude encoding of $\vec{b}$ and quantum simulation to produce a QPU operation representing **A**. We then feed both of these resources into the phase estimation primitive as shown in Figure 13-3.

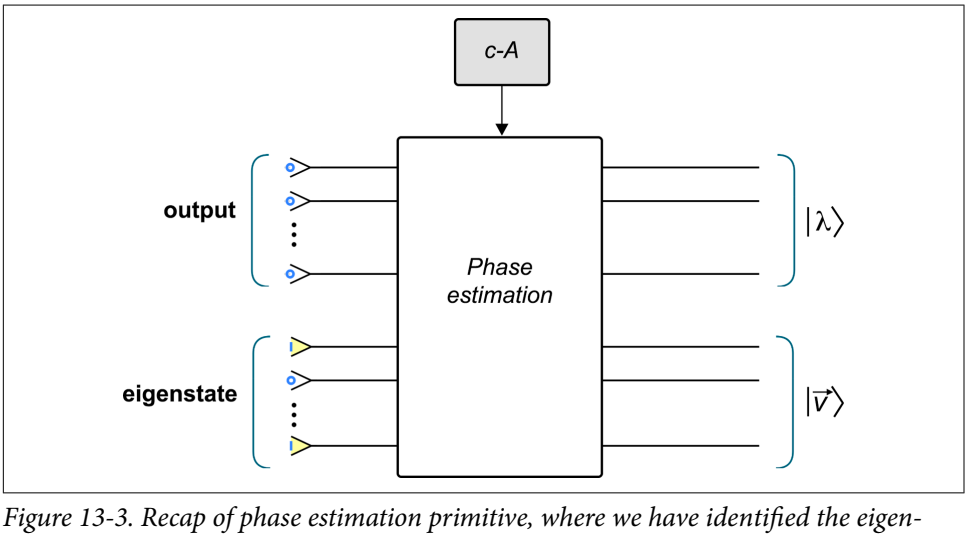


Figure 13-3. Recap of phase estimation primitive, where we have identified the eigenphase obtained in the output register as the eigenvalue of A

Figure 13-3 helps us recall that two input registers are passed to the phase estimation primitive. The lower `eigenstate` register takes an input specifying an eigenstate of the QPU operation for which we would like the associated eigenphase. The upper `output` register (initialized in state $|0\rangle$) produces a representation of the eigenphase, which in our case is an eigenvalue of **A**.

So phase estimation provides *an* eigenvalue of *A*. But how do we get all *n* eigenvalues?

Running phase estimation *n* separate times would reduce HHL's runtime to O(*n*)—no better than its conventional counterparts. We *could* input a uniform superposition in the `eigenstate` register and calculate the eigenvalues in parallel, producing a uniform superposition of them in the `output` register. But suppose we take a slightly different approach and instead send the amplitude encoding of $\vec{b}$ in the `eigenstate` register? This results in the `output` register being a superposition of **A**'s eigenvalues, but one that is entangled with the amplitude encoding of $\vec{b}$ produced in the `output` register.

This will be far more useful to us than just a uniform superposition of the eigenvalues, since now each $|\lambda_i\rangle$ within the entangled state of `eigenstate` and `output` registers has an amplitude of $\tilde{b}_i$ (thanks to the `eigenstate` register). This isn't quite what we want to produce the solution as shown in Equation 13-6, however. The `output` register's *state* represents the eigenvalues of **A**. What we really want is to invert these eigenvalues and—instead of them being held in another (entangled) register—move them into values multiplying the *amplitudes* of the `eigenstate` register. The next steps in HHL achieve this inversion and transfer.

2. Invert values. The second primitive in Figure 13-2 inverts each λ_i value stored in the (entangled) superposition of the `output` and `eigenstate` registers.

At the end of this step the `output` register encodes a superposition of $|1/\lambda_i\rangle$ states, still entangled with the `eigenstate` register. To invert numerical values encoded in a quantum state we actually utilize a number of the arithmetic primitives introduced in Chapter 5. There are many different ways we could build a QPU numerical inversion algorithm from these primitives. One possibility is to use the Newton method for approximating the inverse. Whatever approach we take, as we noted in Chapter 12, division is hard. This seemingly simple operation requires a *significant* overhead in qubit numbers. Not only will the constituent operations require scratch qubits, but dealing with inverses means that we have to encode the involved numerical values in either fixed- or floating-point representations (as well as dealing with overflow, etc.). In fact, the overheads required by this step are a contributing factor to why a full code

sample of even the simplest HHL implementation currently falls outside the scope (and capabilities!) of what we can reasonably present here.[9]

Regardless, at the end of this step, the `eigenstate` register will contain a superposition of $1/\lambda_i$ values.

3. Move inverted values into amplitudes. We now need to move the state-encoded inverted eigenvalues of $\mathbf{A}$ into the *amplitudes* of this state. Don't forget that the amplitude-encoded state of $\vec{b}$ is still entangled with all this, so getting those inverted eigenvalues into the state amplitudes would yield the final line in Equation 13-6, and therefore an amplitude encoding of the solution vector $|\vec{x}\rangle$.

The key to achieving this is applying a `C-ROTY` (i.e., a conditional `ROTY` operation; see Chapter 2). Specifically, we set the target of this conditional operation to a new single scratch qubit (labeled `ROTY scratch` in Figure 13-2), initially in state $|0\rangle$. It's possible to show (although not without resorting to much more mathematics) that if we condition this `ROTY` on the *inverse cosine* of the $1/\lambda_i$ values stored in the `output` register, then the *amplitudes* of all parts of the entangled `output` and `eigenstate` registers where `ROTY scratch` is in the $|1\rangle$ state acquire precisely the $1/\lambda_i$ factors that we're after.

Consequently, this transfer step of the algorithm consists of two parts:

1. Calculate $\arccos\left(1/\lambda_i\right)$ in superposition on each state in the `output` register. This can be achieved in terms of our basic arithmetic primitives,[10] although again with a significant overhead in the number of additional qubits required.
2. Perform a `C-ROTY` between the first register and the `ROTY scratch` (prepared in $|0\rangle$).

At the end of this we will have the state we want *if* `ROTY scratch` is in state $|1\rangle$. We can ensure this if we `READ` the `ROTY scratch` and get a 1 outcome. Unfortunately, this only occurs with a certain probability. To increase the likelihood that we get the needed 1 outcome we can employ another of our QPU primitives, performing amplitude amplification on the `ROTY scratch`.

9 There's something poetic and possibly profound about the fact that attempting to perform *conventional* notions of arithmetic on a QPU can cause such crippling overheads.

10 We actually need to include a constant value in this calculation, and calculate $\arccos\left(C/\lambda_i\right)$ rather than just $\arccos\left(1/\lambda_i\right)$. Since we're not implementing the HHL algorithm fully here we omit this for simplicity.

4. Amplitude amplification. Amplitude amplification allows us to increase the probability of getting an outcome of 1 when we `READ` the single-qubit `ROTY` register. This then increases the chance that we end up with the `eigenstate` register having the desired $\tilde{b}_i/\lambda_i$ amplitudes.

If despite the amplitude amplification a `READ` of `ROTY scratch` still produces the undesired 0 outcome, we have to discard the states of the registers and rerun the whole HHL algorithm from the beginning.

5. Uncompute. Assuming success in our previous `READ`, the `eigenstate` register now contains an amplitude encoding of the solutions $\vec{x}$. But we're not quite done. The `eigenstate` register is still entangled not only with the `output` register, but also with the many other scratch qubits we've introduced along the way. As mentioned in Chapter 5, having our desired state entangled with other registers is troublesome for a number of reasons. We therefore apply the uncomputation procedure (also outlined in Chapter 5) to disentangle the `eigenstate` register.

Bingo! The `eigenstate` register now contains a disentangled amplitude encoding of $\vec{x}$ ready to be used in any of the ways suggested at the start of this section.

HHL is a complex and involved QPU application. Don't feel intimidated if it takes more than one parse to get to grips with; it's well worth the effort to see how a more substantial algorithm masters our QPU primitives.

Quantum Principle Component Analysis

Quantum Principal Component Analysis (QPCA) is a QPU implementation of the eponymous data-processing routine. QPCA not only offers a potentially more efficient approach to this widely used machine-learning task, but can also act as a building block in other QML applications. Like HHL, QPCA relies on QRAM hardware. Before outlining how a QPU allows us to soup up conventional Principal Component Analysis (PCA), we first review PCA itself.

Conventional Principal Component Analysis

PCA is an invaluable tool in data science, machine learning, and beyond. Often used as a preprocessing step, PCA can transform an input set of features into a new, uncorrelated set. The uncorrelated features produced by PCA can be ordered in terms of the amount of the data's variance that they encode. By retaining only some of these new features, PCA is often employed as a *dimensionality reduction* technique. Keeping only the first few principal components allows us to reduce the number of features we need to deal with while retaining as much as possible of the interesting variation within the data.

The process of Principal Component Analysis also goes by many other names in different disciplines, such as the Karhunen-Loève transform or Hotelling transform. It is also equivalent to a *Singular Value Decomposition*.

A common geometrical way to understand the action of PCA is to envision m data points, each described by n features, as being a set of a m points in an n-dimensional *feature space*. In this setting PCA produces a list of n directions in feature space, ordered such that the first is the direction along which there is the most variance in the data, while the second contains the second most variance, and so on. These directions are the so-called *principal components* of our data. This is often illustrated in a simple two-dimensional feature space (i.e., $n = 2$), as shown in Figure 13-4.

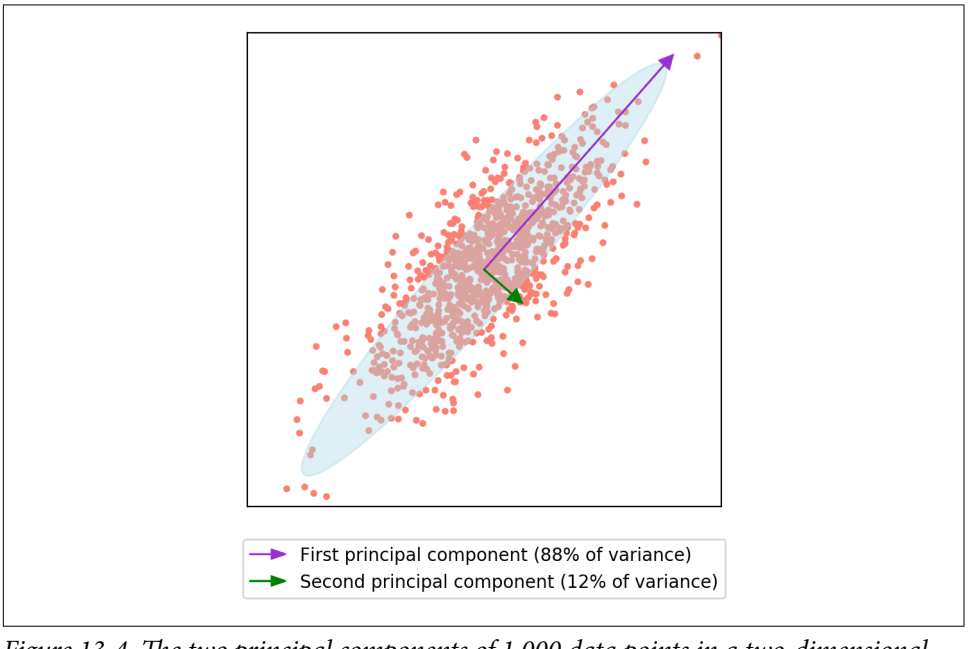


Figure 13-4. The two principal components of 1,000 data points in a two-dimensional feature space. The directions of the arrows show the principal component vectors, while their lengths represent the variance of the data in that direction.

One disadvantage in using the principal components generated by PCA as a new set of features is that they may not have any physical interpretation. However, if we're ultimately interested in building models having the greatest predictive power, this may not be our greatest concern.

Although a geometric description of PCA is useful for building intuition, to actually *compute* principal components we need a mathematical prescription for finding

them. First we calculate the covariance matrix of the dataset in question. If we arrange our data into an $m \times n$ matrix $\mathbf{X}$ (where each row corresponds to one of the m original data points and each column contains values for one of the n different features), then the covariance matrix $\boldsymbol{\sigma}$ is given by:

$$\sigma = \frac{1}{n-1}\mathbf{X}^T\mathbf{X}$$

Conveniently, the principal components are given by finding the eigendecomposition of this covariance matrix. The eigenvectors correspond to the principal component directions, while each associated eigenvalue is proportional to the variance of the data along that principal component. If we want to use PCA for dimensionality reduction, we can then rearrange the eigenvectors in order of decreasing eigenvalue and pick only the top p as our new, reduced set of features.

When performing PCA in practice it is important to *normalize* data before calculating its covariance matrix, as the PCA process is sensitive to the scale of the data. One common normalization technique is finding the deviation of each feature from its mean value and scaling the result by the standard deviation of the data.

The most computationally expensive step in PCA is performing the eigendecomposition of the covariance matrix $\boldsymbol{\sigma}$;. As with HHL, the need to determine eigenvalues immediately brings to mind the phase estimation primitive from Chapter 8. When carefully applied, phase estimation can help us run PCA on a QPU.

PCA with a QPU

We might suspect that something like the following steps could help us find the eigendecomposition that we need for PCA:

1. Represent the covariance matrix of the data as a QPU operation.
2. Perform phase estimation on this QPU operation to determine its eigenvalues.

However, there are a few problems with this proposed approach:

Problem 1: Quantum simulation with $\boldsymbol{\sigma}$
: In the first step we might assume that quantum simulation techniques would help us represent the covariance matrix as a QPU operation, as they did for the matrices involved with HHL. Sadly, covariance matrices rarely satisfy the sparsity requirements of quantum simulation techniques, so we'll need a different way to find a QPU operation representation of $\boldsymbol{\sigma}$.

Problem 2: Input for phase estimation

> In the second proposed step, how do we learn both the eigenvalues *and* eigenvectors that we need? Recall from Figure 13-3 that phase estimation has two input registers, one of which we must use to specify the eigenstate for which we want the associated eigenphase (and hence eigenvalue). But knowing any of the eigenvectors of **σ** is precisely part of the problem we want to solve with QPCA! We got around this seemingly circular problem when using phase estimation in HHL because we were able to use $|\vec{b}\rangle$ in the `eigenstate` input register. Even though we didn't know precisely what eigenstates $|\vec{b}\rangle$ superposed, phase estimation acted on them all in parallel—without us ever needing to learn them. Is there also some kind of clever `eigenstate` input we can use for QPCA?

Remarkably, by introducing one critical trick we can solve both of the problems just described. The trick in question is a way to represent the covariance matrix **σ** in a QPU *register* (not a QPU operation!). How this trick fixes the preceding problems is quite circuitous (and mathematically involved), but we give a brief outline next.

Representing a covariance matrix in a QPU register

The idea of representing a *matrix* in a register is new to us—so far we've gone to some length to represent matrices as QPU *operations*.

We've exclusively used circle notation to describe QPU registers, but have occasionally hinted that a full-blown mathematical description involves using (complex-valued) vectors. However, though we've carefully avoided introducing the notion, there's an *even more general* mathematical description of a QPU register that uses a matrix known as a density operator.[11] The details of density operators are far beyond the scope of this book (though Chapter 14 contains tips and references for starting to learn about them), but the important point for QPCA is that if we have QRAM access to our data, then a trick exists for initializing a QPU register so that its density operator description is precisely the covariance matrix of our data. While encoding matrices in a QPU register's density operator description like this is not normally very useful, for QPCA it affords us the following fixes to the two problems we highlighted earlier.

11 Density operators provide a more general description of a QPU register's state than using a complex vector (or circle notation) because they allow for cases where the register might not only be in superposition, but subject to some statistical uncertainty as to precisely what that superposition is. These quantum states that contain statistical uncertainty are usually called *mixed states*.

The trick QPCA uses to represent a covariance matrix as a density operator works because covariance matrices are always in *Gram form*, meaning that they can be written in the form $\mathbf{V}^T\mathbf{V}$ for some other matrix $\mathbf{V}$. For other matrices it's not such a useful trick.

Fixing problem 1

Having our covariance matrix in a QPU register's density operator allows us to perform a trick where by leveraging the `SWAP` operation (see Chapter 3) we repeatedly perform a kind of partial "mini-SWAP" (partial in a *quantum superposition* sense) between the register encoding $\boldsymbol{\sigma}$ and a second register. Although we won't go into any detail about how to modify `SWAP` to perform this mini-`SWAP` subroutine, it turns out that using it effectively results in a quantum simulation of $\boldsymbol{\sigma}$ being implemented on the second register.[12] This is precisely the result we would normally achieve using more standard quantum simulation techniques,[13] only this mini-`SWAP` approach to generating a QPU operation representing $\boldsymbol{\sigma}$ works efficiently even if $\boldsymbol{\sigma}$ isn't sparse, so long as it is of low rank. Despite this trick requiring us to repeatedly apply `SWAP` operations (and consequently repeatedly reencode $\boldsymbol{\sigma}$ as a QPU register's density operator), it still proves to be efficient.

The rank of a matrix is the number of its columns that are linearly independent. Since the columns of $\boldsymbol{\sigma}$ are the features of our data, saying that a covariance matrix is of "low rank" means that our data is actually well described by some smaller subspace of the full feature space. The number of features we would need to describe this subspace is the rank of $\boldsymbol{\sigma}$.

Fixing problem 2

It transpires that a density operator representation of σ is also precisely the right state for us to use in the phase estimation `eigenstate` input register. If we do this, then the `eigenstate` register output by the phase estimation primitive will encode an eigenvector of σ (i.e., one of the principal components) and the `output` register will encode the associated eigenvalue (i.e., the amount of variance along that principal component). Precisely which principal component, eigenvalue/eigenvector pair we get is randomly determined, but the probability of getting a given principal component is, conveniently, determined by its variance.

12 For more technical detail on how this operation is built and how it enables such a quantum simulation, see Lloyd et al., 2013.

13 Note that this mini-`SWAP` approach to representing a matrix as a QPU operation won't always be better than quantum simulation approaches. It only works well here because covariance matrices happen to be simple to encode in a QPU register's density operator thanks to them being in Gram form.

With these proposed fixes, a full schematic for QPCA is as shown in Figure 13-5.

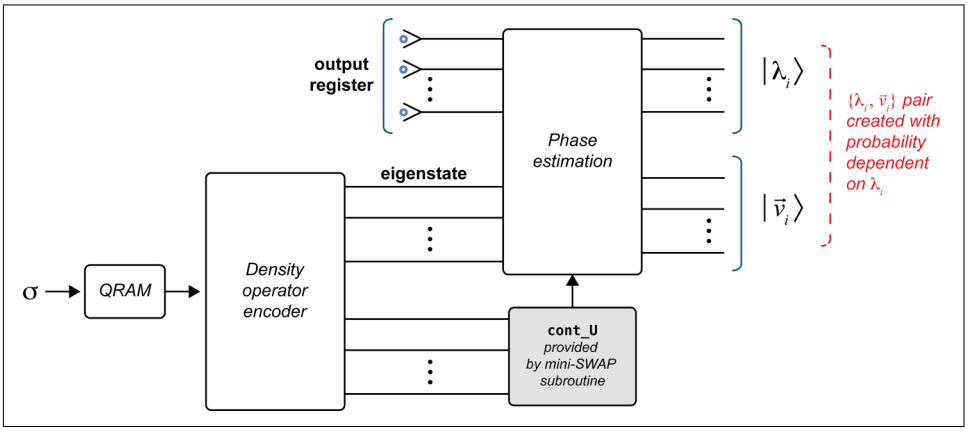


Figure 13-5. Schematic for QPCA

Figure 13-5 illustrates how the mini-SWAP subroutine's ability to perform quantum simulation with a density operator allows us to use it as the `cont_U` input to phase estimation. Note that the density operator encoder must be run multiple times and its output passed to the mini-SWAP subroutine, which provides the (conditional) QPU operation required by the phase estimation primitive (see Chapter 8). Also note that one run of the density operator encoder is input into the `eigenstate` register of the phase estimation primitive.

We've used the term *density operator encoder* to denote the process that allows us to represent our covariance matrix as a register's density operator. We won't go into detail about how the encoder works here, but Figure 13-5 summarizes how access to such an ability allows us to turn the phase estimation primitive to the task of PCA.

The output

After all this, we have an algorithm returning all the information we need about one (randomly chosen) principal component of our data—most likely to be the component having highest variance (exactly the ones we often care about when using PCA). But both the principal components and their variances are stored in QPU registers, and we must assert our usual important caveat that *our solutions are output in a quantum form*. Nevertheless, as was the case with HHL, it may still be possible to `READ` useful derived properties. Furthermore, we could pass the QPU register states onward to other QPU applications, where their quantum nature is likely advantageous.

Performance

Conventional algorithms for PCA have runtimes of $O(d)$, where d is the number of features we wish to perform PCA on.

In contrast, QPCA has a runtime of $O(R \log d)$, with R being the lowest-rank acceptable approximation to the covariance matrix $\boldsymbol{\sigma}$ (i.e., the smallest number of features that allow us to still represent our data to an acceptable approximation). In cases where $R < d$ (the data is well described by the low-rank approximation of our principal components), QPCA gives an exponential improvement in runtime over its conventional counterpart.

The runtime of QPCA qualifies our earlier assertion that it only offers an improvement for *low-rank* covariance matrices. This requirement is not as restrictive as it may seem. We are (presumably) normally using PCA on data that we expect is amenable to such a low-rank approximation—otherwise we'd be remiss to consider representing it with a subset of its principal components.

Quantum Support Vector Machines

Quantum support vector machines (QSVMs) demonstrate how QPUs can implement *supervised* machine-learning applications. Like a conventional supervised model, the QSVM we describe here must be trained on points in feature space having known classifications. However, QSVM comes with a number of unconventional constraints. First, a QSVM requires that training data be accessible in superposition using QRAM. Furthermore, the parameters that describe our learned model (used for future classification) are produced amplitude-encoded in a QPU register. As always, this means that we must take special care in how we plan to utilize a QSVM.

Conventional Support Vector Machines

Support vector machines (SVMs) are a popular type of supervised classifier finding wide application. As with other *linear classifiers*, the idea of SVMs is to use training data to find hyperplanes in feature space that separate different output classes of the problem. Once an SVM has learned such hyperplanes, a new data point in feature space can be classified by checking on which sides of these hyperplanes it lies. For a simple example, suppose we only have two features (and therefore a two-dimensional feature space), and furthermore that there are only two possible output classes the data can assume. In that case the hyperplane we seek is a line, as shown in Figure 13-6, where the x- and y-axes represent values of the two features, and we use blue and red markers to represent training data from the two output classes.

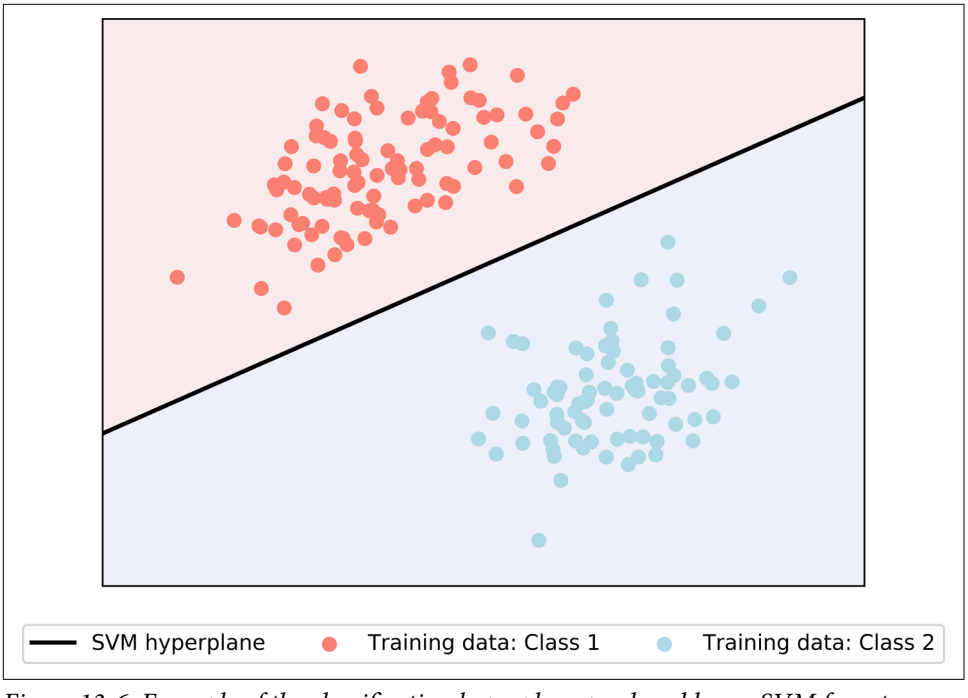


Figure 13-6. Example of the classification hyperplane produced by an SVM for a two-class classification problem with two features

How does an SVM learn a suitable hyperplane from training data? Geometrically we can think of the SVM training process as considering *two* parallel hyperplanes separating the training data classes, and trying to maximize the distance between this pair of hyperplanes. If we then choose a third hyperplane halfway between these as our classifier, we will have the classification hyperplane with the maximum *margin* between the training classes. The optimized hyperplane learned by the SVM is mathematically described[14] by a normal vector $\vec{w}$ and offset b. Having learned this description of the hyperplane, we predict the class of a new data point $\vec{x}$ according to the following rule:

$$\text{class} = \text{sign}\left(\vec{w} \cdot \vec{x} - b\right)$$

This mathematically determines on which side of the hyperplane the new point lies.

14 These parameters describe the hyperplane through the equation $\vec{w} \cdot \vec{x} - b = 0$, where $\vec{x}$ is a vector of feature values.

Although the preceding description of finding optimal hyperplanes is perhaps easier to visualize, for computational purposes it's common to consider a so-called *dual formulation* of the SVM training process.[15] The dual form is more useful for describing QSVMs, and requires us to solve a quadratic programming problem for a set of parameters $\vec{\alpha} = [\alpha_1, ..., \alpha_m]$. Specifically, finding an SVM's optimal hyperplane is equivalent to finding the $\vec{\alpha}$ maximizing the expression in Equation 13-7.

Equation 13-7. Dual description of the SVM optimization problem

$$\sum_i \alpha_i y_i - \frac{1}{2}\sum_i \sum_j \alpha_i \vec{x}_i \cdot \vec{x}_j \alpha_j$$

Here $\vec{x}_i$ is the i^{th} training data point in feature space and y_i is the associated known class. The set of inner products $\vec{x}_i \cdot \vec{x}_j = K_{ij}$ between the training data points take on a special role if we generalize SVMs (as we'll discuss briefly in the next section) and are often collected into a matrix known as the *kernel matrix*.

Finding an $\vec{\alpha}$ satisfying this expression subject to the constraints that $\sum_i \alpha_i = 0$ and $y_i \alpha_i \geq 0 \forall i$ gives us the information needed to recover the optimal hyperplane parameters $\vec{w}$ and b. In fact, we can classify new data points directly in terms of $\vec{\alpha}$ according to Equation 13-8, where the intercept b can also be calculated from the training data.[16]

Equation 13-8. Classification rule for an SVM in the dual representation

$$\text{sign}\left(\sum_i \alpha_i \vec{x}_i \cdot \vec{x} - b\right)$$

The key thing to note here for our forthcoming discussion of a *quantum* SVM is that classifying a new data point $\vec{x}$ requires us to calculate its inner product with every training data point, $\vec{x} \cdot \vec{x}_i$.

We won't delve any further into the detailed derivation or usage of these equations; instead we'll see how the calculations they require can be performed much more efficiently if we have access to a QPU.

15 Optimization problems often have such a dual form that can be easier to deal with. It's worth noting that sometimes these dual forms can contain subtle differences from the original (*primal*) optimization problem.

16 In fact, b can be determined from a single piece of training data lying on one of the two parallel hyperplanes defining the margin. The points lying on these defining hyperplanes are known as the *support vectors*.

SVM generalizations

First, though, it's worth noting that the kind of SVMs we have described so far are restricted to certain classification problems. To begin with, our discussion has assumed that the data to be modeled is *linearly separable*; i.e., that there definitely exists a hyperplane that could completely and unambiguously separate data from the two classes. Often, of course, this might not be the case, and while linearly separating the data could still provide a good fit, data from different classes may somewhat overlap in the feature space, as exemplified in Figure 13-7.

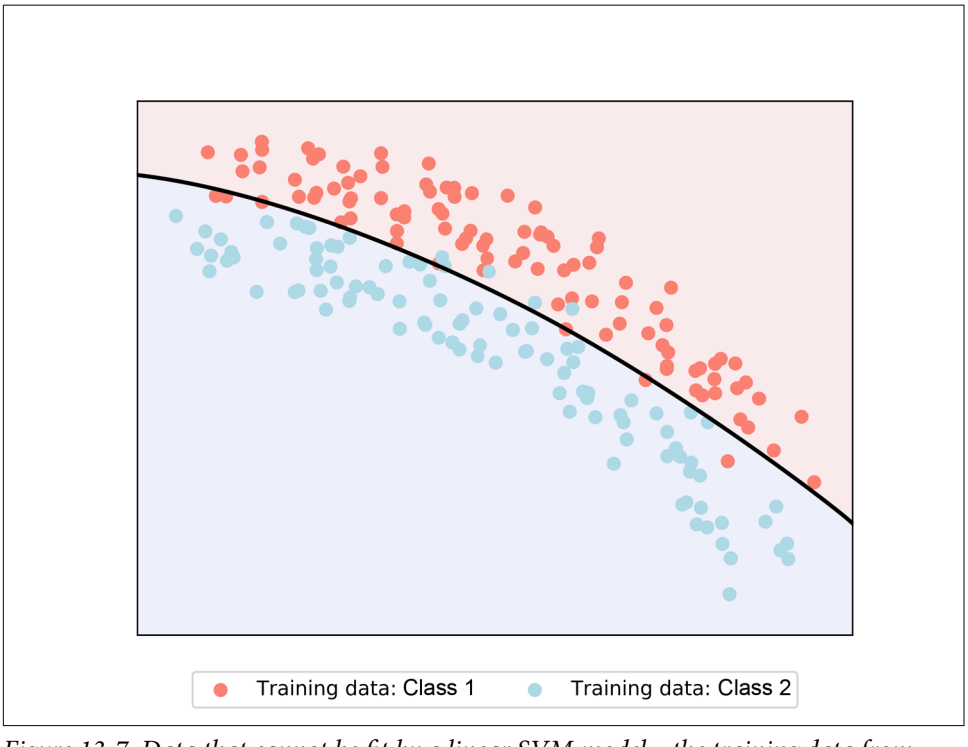


Figure 13-7. Data that cannot be fit by a linear SVM model—the training data from different classes overlap in feature space, and furthermore the correct decision boundary is clearly nonlinear.

SVMs deal with this possibility through the introduction of so-called *soft margins* in the training process. Although we won't expand on them here, it's worth noting that the QPU speedups we outline shortly persist for soft-margin SVMs.

Even soft-margin SVMs are still restricted to a linear separation of the data, though. In some cases a simple hyperplane may not do a good job of separating classes. When this is the case, we can try embedding our feature space in an even higher-dimensional space. If we perform this embedding carefully, we may be able to find a

hyperplane in the higher-dimensional space that *does* effectively segregate the data from different classes. In other words, we use the projection of an $n + m$-dimensional hyperplane into an n-dimensional feature space, since a linear $n + m$-dimensional hyperplane can have a nonlinear n-dimensional projection. Such a nonlinear SVM is shown in Figure 13-7. Though this may sound like a complex modification to our SVM training process, it turns out we can very easily implement this kind of nonlinear generalization. By replacing the kernel matrix of inner products appearing in Equation 13-7 with a carefully chosen alternative kernel matrix, we're able to encapsulate higher-dimensional, nonlinear margins. This extension is often referred to as training an SVM with a *nonlinear kernel*.

SVM with a QPU

There exist QPU algorithms improving the performance of both the training of SVM models and classification with a trained model. Here we will only outline using a QPU to efficiently train a QSVM model. The best classical algorithms for training conventional SVMs have runtimes of $O(poly(m,n))$, where m is the number of training data points and n is the number of features describing each point. In contrast, a QSVM can be trained with runtime $O(log(mn))$.

Using a QPU to train a quantum SVM

Obtaining a quantum advantage for training SVMs is contingent on us being content with a fully quantum model. What we mean by this is that a trained QSVM is a set of QPU registers containing *amplitude encodings* of the hyperplane parameters $\vec{w}$ and b. Although these parameters are locked in superpositions, we'll see that they can still be used to classify new points in feature space, so long as the data for these new points can be accessed in superposition via a QRAM.

Training an SVM on a set of m training data points, $\left\{\vec{x}_i, y_i\right\}_{i=1}^{m}$, requires us to find optimal values of the $\vec{\alpha}$ solving the quadratic programming problem stated in Equation 13-7. This seems like a tricky problem to speed up with a QPU. However, there is a reformulation of the SVM training problem known as a *Least Squares Support Vector Machine* (LS-SVM) that casts the building of an SVM classifier into a least-squares optimization problem.[17] As a consequence, to find the $\vec{\alpha}$ required by the SVM dual formulation, we are now presented with a linear system of equations, the solution of

17 There are some subtle differences between SVM and LS-SVM models, although the two have been shown to be equivalent under certain reasonable conditions. See, for example, Ye and Xiong, 2007.

which is given by the matrix equation shown in Equation 13-9 (where $\vec{y}$ is a vector containing the training data classes).

Equation 13-9. LS-SVM equation

$$\begin{bmatrix} b \\ \vec{\alpha} \end{bmatrix} = \mathbf{F}^{-1} \begin{bmatrix} 0 \\ \vec{y} \end{bmatrix}$$

This looks more like something amenable to a QPU speedup, via the HHL algorithm from "Solving Systems of Linear Equations" on page 262. The matrix **F** is constructed from the kernel matrix K of training data inner products as outlined in Equation 13-10.

Equation 13-10. How the F matrix is built from the kernel matrix

$$\mathbf{F} = \begin{bmatrix} 0 & 1^T \\ 1 & K + \frac{1}{\gamma}\mathbb{1} \end{bmatrix}$$

Here, γ is a real number hyperparameter of the model (that in practice would be determined by cross-validation), $\mathbb{1}$ is the identity matrix, and we use 1 to denote a vector of m ones. Once we've used **F** to obtain values of $\vec{\alpha}$ and b, we can classify new data points using the LS-SVM just as we did with the standard SVM model, by using the criterion in Equation 13-8.

Using the HHL algorithm to efficiently solve Equation 13-9 and return registers' amplitude encoding $|b\rangle$ and $|\vec{\alpha}\rangle$ requires us to address a few key concerns:

*Concern 1: Is **F** suitable for HHL?*
: In other words, is the matrix **F** of the type we can invert with HHL (i.e., Hermitian, sufficiently sparse, etc.)?

Concern 2: How can we act $\mathbf{F}^{-1}$ on $\left[0, \vec{y}\right]$?
: If we *can* calculate $\mathbf{F}^{-1}$ using HHL, then how do we ensure it correctly acts on the vector $\left[0, \vec{y}\right]$ as Equation 13-9 requires?

Concern 3: How do we classify data?
: Even if we address concerns 1 and 2, we still need a way to make use of the obtained quantum representations of b and $\vec{\alpha}$ to train data presented to us in the future.

Let's address each of these concerns in turn to help convince ourselves that the training of a QSVM can indeed leverage the HHL algorithm.

Concern 1: Is F suitable for HHL?. It's not too tricky to see that the matrix **F** is Hermitian, and so we can potentially represent it as a QPU operation using the quantum simulation techniques from Chapter 9. As we previously noted, being Hermitian, although necessary, is not sufficient for a matrix to be efficiently used in quantum simulation. However, it's also possible to see that a matrix of the form **F** can be decomposed into a sum of matrices, each of which satisfies all the requirements of quantum simulation techniques. So it turns out that we *can* use quantum simulation to find a QPU operation representing **F**. Note, however, that the nontrivial elements of **F** consist of inner products between the training data points. To efficiently use quantum simulation for representing **F** as a QPU operation we will need to be able to access the training data points using QRAM.

Concern 2: How can we act $\mathbf{F}^{-1}$ on $\left[0, \vec{y}\right]$?. If we're using HHL to find $\mathbf{F}^{-1}$ then we can take care of this concern during the phase estimation stage of the algorithm. We input an amplitude encoding of the vector of training data classes, $|\vec{y}'\rangle$, into the phase estimation's `eigenstate` register. Here we use $|\vec{y}'\rangle$ to denote a QPU register state amplitude-encoding the vector $\left[0, \vec{y}\right]$. This, again, assumes that we have QRAM access to the training data classes. By the same logic we described when originally explaining HHL in "Solving Systems of Linear Equations" on page 262, $|\vec{y}'\rangle$ can be thought of as a superposition of the eigenstates of F. As a consequence, when we follow through the HHL algorithm we will find that $|F^{-1}\vec{y}\rangle$ is contained in the final output register, which is equal to precisely the solutions that we desire: $|b, \vec{\alpha}\rangle$. So we don't have to do anything fancy—HHL can output $\mathbf{F}^{-1}$ acted on the required vector, just as it did when we originally used it for solving systems of linear equations more generally.

Concern 3: How do we classify data?. Suppose we are given a new data point $\vec{x}$, which we want to classify with our trained QSVM. Recall that a "trained QSVM" is really just access to the state $|b, \vec{\alpha}\rangle$. We can perform this classification efficiently so long as we have QRAM access to the new data point $\vec{x}$. Classifying the new point requires computing the sign given by Equation 13-8. This, in turn, involves determining the inner products of $\vec{x}$ with all the training data points $\vec{x}_i$, weighted by the LS-SVM dual hyperplane parameters α_i. We can calculate the requisite inner products in superposition and assess Equation 13-8 as follows. First we use our trained LS-SVM state $|b, \vec{\alpha}\rangle$ in the `address` register of a query to the QRAM holding the training data. This gives us a superposition of the training data with amplitudes containing the α_i. In another register we perform a query to the QRAM containing the new data point $\vec{x}$.

Having both of these states, we can perform a special *swap-test subroutine*. Although we won't go into full detail, this subroutine combines the states resulting from our two QRAM queries into an entangled superposition and then performs a carefully constructed swap test (see Chapter 3). Recall that as well as telling us whether the states of two QPU registers are equal or not, the exact success probability p of the `READ` involved in the swap test is dependent on the *fidelity* of the two states—a quantitative measure of precisely how close they are. The swap test we use here is carefully constructed so that the probability p of `READ`ing a 1 reflects the sign we need in Equation 13-8. Specifically, $p < 1/2$ if the sign is +1 and $p >= 1/2$ if the sign is –1. By repeating the swap test and counting 0 and 1 outcomes, we can estimate the value of the probability p to a desired accuracy, and classify the data point $\vec{x}$ accordingly.

Thus, we are able to train and use a quantum LS-SVM model according to the schematic shown in Figure 13-8.

Without delving into mathematical details, like those of the swap-test subroutine, Figure 13-8 gives only a very general overview of how LS-SVM training proceeds. Note that many details of the swap-test subroutine are not shown, although the key input states are highlighted. This gives some idea of the key roles the QPU primitives we've introduced throughout the book play.

Our QSVM summary also provides an important take-home message. A key step was recasting the SVM problem in a format that is amenable to techniques a QPU is well suited for (in particular, matrix inversion). This exemplifies a central ethos of this book—through awareness of *what* a QPU can do well, domain expertise may allow the discovery of new QPU applications simply by casting existing problems in QPU-compatible forms.

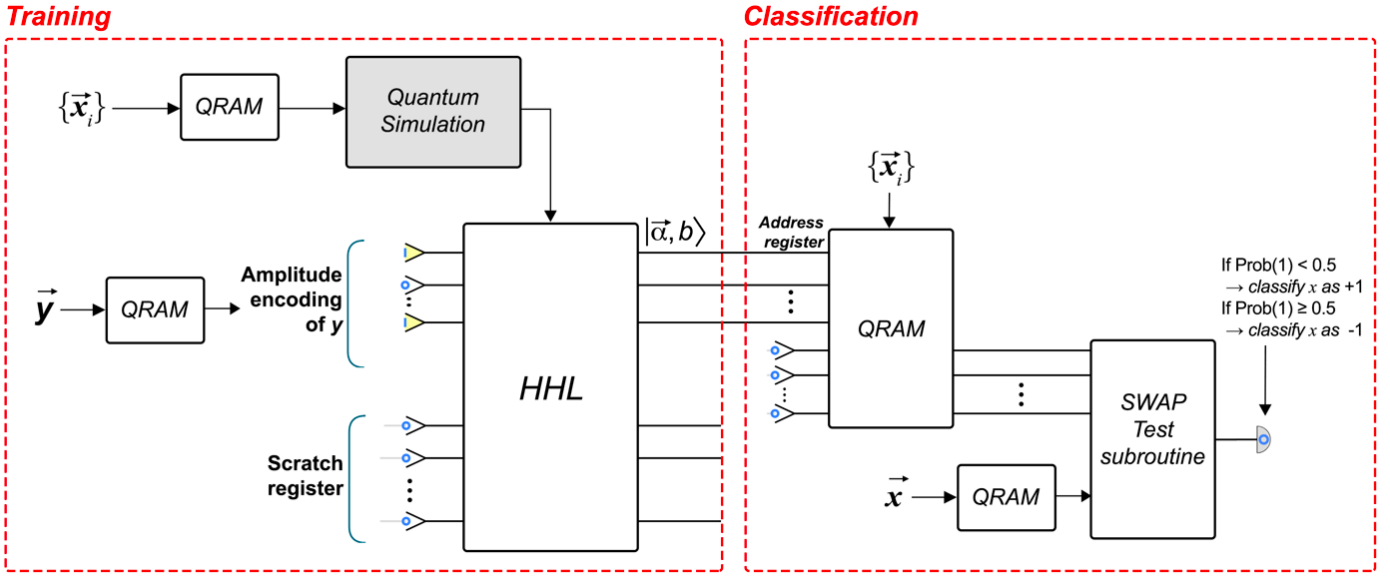


Figure 13-8. Schematic showing the stages in training and classifying with a QSVM

Other Machine Learning Applications

Quantum machine learning is still an extremely dynamic area of research. In this chapter we presented three canonical examples, but new developments in QML are constantly being made. At the same time, *conventional* approaches to machine learning are being inspired by QPU algorithms—in fact, within the time period of us writing this book, all three of the QML applications presented in this chapter have inspired the development of conventional algorithms with similar runtime improvements.[18] This should not shake your confidence in QML's potential; these results were unlikely to have been discovered without the inspiration of their QPU counterparts, showing that QML applications have far-reaching and unexpected consequences.[19] There are also many other QML applications that we simply didn't have space to properly mention. These include efficient QPU applications for linear regression,[20] unsupervised learning,[21], Boltzmann machines,[22] semidefinite programming,[23] and quantum recommender systems[24] (quantum recommender systems have also inspired improvements in conventional algorithms[25]).

18 HHL: Chia et al., 2018; QPCA and QSVM: Tang, 2018.

19 In fact, it's not obvious how practical these quantum-inspired algorithms will be in actual usage. See for example Arrazola et al., 2019.

20 Chakraborty et al., 2018.

21 Lloyd et al., 2013.

22 Wiebe et al., 2014.

23 Brandão et al., 2017.

24 Kerenidis and Prakash, 2016.

25 Tang, 2018.

PART IV
Outlook

CHAPTER 14

Staying on Top: A Guide to the Literature

We hope you've enjoyed tinkering with the computational problems we presented in this book! Before closing, we'll briefly introduce a few subjects we didn't have space to go into previously, and provide pointers on where to go to learn more about these and other topics in quantum computing. We won't go into too much depth here; the aim here is rather to link what you've learned so far to material reaching beyond the scope of this book. Let the intuition you've built here be only the first step in your exploration of quantum programming!

From Circle Notation to Complex Vectors

The $|x\rangle$ notation that we use throughout the book to refer to states in a quantum register is called *bra-ket notation* or, sometimes *Dirac notation*, in honor of the 20th-century physicist of the same name. Throughout the quantum computing literature, this—rather than circle notation—is the notation used to represent quantum states. In Chapter 2 we hinted at the equivalence between these two notations, but it's worth saying a little more to set you on your way. A general superposition within a single-qubit register can be expressed in Dirac notation as $\alpha|0\rangle + \beta|1\rangle$, where α and β are the states' *amplitudes*, represented as complex numbers satisfying the equation $|\alpha|^2 + |\beta|^2 = 1$. The magnitude and relative phase of each value in the circle notation we've been using are given by the *modulus* and *argument* of the complex numbers α and β, respectively. The probability of a `READ` outcome for a given binary output value from a QPU register is given by the *squared modulus* of the complex number describing the amplitude of that value. For example, in the preceding single-qubit case, $|\alpha|^2$ would give the probability of `READ`ing a 0 and $|\beta|^2$ would give the probability of `READ`ing a 1.

The complex vectors describing QPU register states satisfy some very specific mathematical properties, meaning that they can be said to exist in a structure known as a *Hilbert space*. You likely don't need to know that much about Hilbert space, but will hear the term used a lot—mostly simply to refer to the collection of possible complex vectors representing a given QPU register.

In the case of single qubits, a common way to parameterize α and β is as $\cos\theta|0\rangle + e^{i\phi}\sin\theta|1\rangle$. In this case, the two variables θ and ϕ can be interpreted as angles on a sphere, which many references will refer to as the *Bloch* sphere. As mentioned in Chapter 2, the Bloch sphere provides a visual representation of single-qubit states. Unlike circle notation, though, it's unfortunately difficult to use the Bloch sphere to visualize registers with more than one qubit.

Another complication regarding qubit states that we didn't cover in the book is the so-called *mixed states*, which are represented mathematically by so-called *density operators* (although we did mention density operators briefly in Chapter 13). These are a statistical mixture of the kind of *pure states* that we've been working with in our quantum registers throughout the book (i.e., how you should describe a qubit if you're not sure precisely *what* superposition it's in). To some extent it is possible to represent mixed states in circle notation, but if we have a QPU with error correction (more on that later), pure states are enough to get started with QPU programming.

As visualizations of many-qubit registers are not common in most textbooks and academic references, quantum registers are most often represented solely by complex vectors,[1] with the length of the vector needed for representing n qubits being 2^n (just as the number of circles needed in circle-notation to represent n qubits was 2^n. When writing down the amplitudes of an n-qubit register's state in a column vector, the amplitude of the state $|00\ldots0\rangle$ is conventionally placed at the top with the remaining possible states following below in ascending binary order.

QPU operations are described by unitary matrices acting on these complex vectors. The order in which the matrices are written is right to left (exactly opposite of how we would write a quantum circuit diagram—left to right), so that the first matrix that acts on our complex vector corresponds to the first (leftmost) gate in an associated circuit diagram. Equation 14-1 shows a simple example, where a `NOT` gate (often also referred to as `X` in the literature) is applied to an input qubit in the state $\alpha|0\rangle + \beta|1\rangle$. We can see how it flips the values of α and β, as expected.

1 Recall from Chapter 13, though, that mixed states are represented by matrices, called *density operators*, instead of vectors.

Equation 14-1. NOT gate acting on qubit in standard complex-vector notation

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

Single-qubit gates are represented as 2×2 matrices since they transform vectors with two entries, corresponding to the single-qubit register values of $|0\rangle$ and $|1\rangle$. Two-qubit gates are represented by 4×4 matrices and, in general, n-qubit gates are represented by $2^n \times 2^n$ matrices. In Figure 14-1 we show the matrix representations of some of the most commonly used single- and two-qubit gates. If you have an understanding of matrix multiplication and really want to test your understanding, you might try to predict the action of these operations on different input states and see whether your predictions match what you see in QCEngine's circle notation.

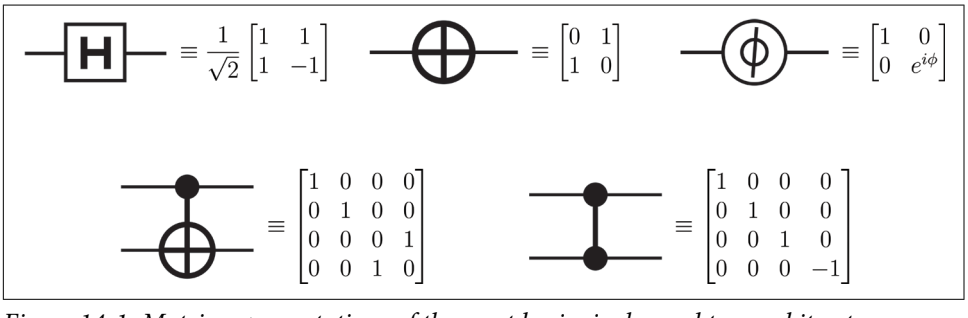


Figure 14-1. Matrix representations of the most basic single- and two-qubit gates

Some Subtleties and Notes on Terminology

There are a few subtleties we wanted to mention with regard to to the terminology used in this book.

- Throughout the book we've referred to pre-quantum computing as "conventional." You may also hear people use the term *classical* to refer to traditional binary computing that doesn't operate with quantum registers.
- We have often used so-called *scratch qubits* to help perform some aspects of the quantum computation. These are instead referred to as *ancilla qubits* in many quantum computing resources.
- In Chapter 2 we introduced the `PHASE` gate, which takes an angle as an input parameter. In this book we have used *degrees* to represent angles, ranging from 0° to 360°. It is common in the quantum computing literature to specify angles in *radians*. Radians are an angular unit corresponding to an angle at the center of the circle such that its arc has the same length as its radius. The following table shows commonly used angles in both units:

Degrees	0°	45°	90°	135°	180°	225°	270°	315°
Radians	0	$\frac{\pi}{4}$	$\frac{\pi}{2}$	$\frac{3\pi}{4}$	π	$\frac{5\pi}{4}$	$\frac{3\pi}{2}$	$\frac{7\pi}{4}$

- There are three cases in which the `PHASE` gate receives a special name:

Angle (radians)	$\frac{\pi}{4}$	$\frac{\pi}{2}$	π
Name	T	S	Z

- In Chapter 6, we introduced the AA primitive. This primitive (more specifically, the `mirror` operation) allows us to *amplify the amplitude* of marked states in our register, thereby increasing the probability of `READ`ing out that register. Although this terminology may seem straightforward enough, it's worth noting that in the scholarly literature, these iterations are usually denoted as *Grover iterations*, while the expression *amplitude amplification* is reserved for a general class of algorithms that can use Grover iterations in order to improve their success probabilities.
- The particular configuration of a QPU register (eg: what superposition or entangled arrangement it might exist in) is commonly referred to as the *state* of the register. This terminology arises from the fact that a register's configuration is really described by a quantum state in the mathematical sense introduced briefly above (even if we we choose to visualize the register more conveniently using circle notation).
- When describing an N qubit QPU register in some superposition of its 2^N possible integer values, we have often referred to each of these possibilities (and their asociated amplitudes) as *values* within the superposition. An equivalent, more commonplace, expression is *term*. So one might talk (for example) about the amplitude of "*the $|4\rangle$ term*" in a QPU register's superposition. If we're thinking of QPU register's in terms of their proper Dirac notation representation, then this terminology makes sense, as $|4\rangle$ (and it's amplitude) really is a *term* in a mathematical expression. We avoided using the expression *term* throughout most of the book, simply because of the unavoidable mathematical connotations.
- QPU operations are sometimes referred to as *quantum gates*, with reverance to the logic gates of conventional computation. You can consider the terms "QPU operation" and "quantum gate" to be snynonymous. Collections of quantum gates form a quantum circuit.

Measurement Basis

There's a widespread concept in quantum computing that we've carefully managed to avoid mentioning throughout the book—that of *measurement basis*. Understanding measurement more thoroughly in quantum computing really involves getting to grips with the full mathematical machinery of quantum theory, and we can't hope to begin to do so in this short space. The goal of this book is to give you an intuitive "in" to the concepts needed to program a QPU, and for more in-depth discussion we refer the interested reader to the recommended resources at the end of this chapter. That said, we'll try to briefly give an insight into how the core idea of measurement basis relates to the concepts and terminology we've used.

Wherever we used the `READ` operation, we always assumed it would give us an answer of 0 or 1. We saw that these two answers correspond to the states $|0\rangle$ and $|1\rangle$, in the sense that these are the states which will always give a `0` or `1` outcome, respectively. These states are, more technically, the eigenstates[2] of the `PHASE(180)` operation (also sometimes called a `Z` gate). Whenever we "`READ` in the `Z` basis," as we have implicitly done throughout the book, the complex vector describing our QPU register will, after the `READ`, end up in one of these two states. We say that we've *projected* our QPU register onto one of these states.[3] Although we've so far only thought about measurements in the `Z` basis, this is not the only option.

Different measurement bases are like different *questions* we are asking our QPU state. The possible questions we can ask with quantum `READ` operations are which of the eigenstates from certain QPU operations our system is in. This may sound incredibly abstract, but in quantum physics these operations and their eigenstates do have physical meanings, and understanding their precise nature requires a more in-depth understanding of the underlying physics. Since so far we've only been measuring in the basis of `PHASE(180)`, we've actually always been asking the question, "is the QPU register in the eigenstate of `PHASE(180)` corresponding to the eigenvalue `+1`, or is it in the state corresponding to the eigenvalue `-1`?" Even if the QPU register is in a superposition of these possibilities, after the `READ` it will assume one of them.

Performing a `READ` in another basis means asking which eigenstate of some *other* QPU operation, *U*, our QPU register is in. After the `READ` our QPU register will end up in (i.e., be projected onto) one of the eigenstates of `U`.

2 For a refresher on eigenstates, see Chapter 8.

3 The term *project* here has a mathematical meaning. Projection operators in mathematics "select out" certain vectors from a linear combination. In quantum mechanics, the action of a `READ` in the `Z` basis is to "project" the complex vector representing our QPU register onto one of the complex vectors representing $|0\rangle$ or $|1\rangle$.

Since a QPU register's state can be thought of as a superposition of the eigenstates of any QPU operation (as we described in Chapter 13), writing out the complex vector representation of a QPU register state in the eigenbasis of some operation `U` allows us to work out the various `READ` probabilities in the `U` measurement basis. Another interesting aspect of this whole measurement basis business is that states that *always* have the same measurement outcome (with 100% probability) when measured in one basis may not be so definite in a different basis, possibly only *probabilistically* yielding each outcome. For example, we've seen multiple times that when reading out the state $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, we have a 50% chance of getting 0 and 50% chance of getting 1 when measuring in the `Z` basis. However, if we're measuring in the `X` (`NOT`) basis, we will always get 0; this is because $|+\rangle$ happens to be an eigenstate of `X` and therefore, when considered in the measurement basis of `X`, our state isn't in a superposition at all.

Gate Decompositions and Compilation

Controlled operations have played an important role throughout the book, and on occasion, you may have been left wondering how we would implement these operations. Cases that might particularly require elucidation are:

1. Controlled operations where the operation acting on the target qubit is something other than `NOT` or `PHASE`
2. Gates with controls on several qubits at once

For compactness of notation we've often drawn these kinds of operations as single diagrammatic units in our circuit diagrams. However, in general, they will not correspond to native instructions on QPU hardware and they'll need to be implemented in terms of more fundamental QPU operations.

Fortunately, more complex conditional operations can be written as series of single-qubit and two-qubit operations. In Figure 14-2 we show a general decomposition of a controlled QPU operation (corresponding to a general unitary matrix in the mathematics of quantum computing). The constituent operations in this decomposition need to be chosen such that A, B, and C satisfy $U = e^{i\alpha}AXBXC$ (where X is the `NOT` gate) and acting all three operations directly one after the other, $A \cdot B \cdot C$, has no overall effect on our QPU register state.

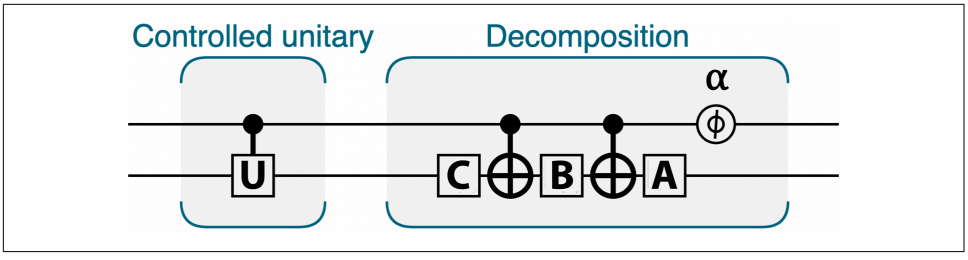


Figure 14-2. General decomposition of a controlled unitary

If we can find operations A, B, C, and α satisfying these requirements, then we can conditionally perform our desired operation U. Sometimes there may be more than one possible way to decompose a conditional operation according to this prescription.

What about conditional operations that are conditioned on *more than one qubit at once*? As an example, Figure 14-3 shows three different decompositions for implementing the `CCNOT` (Toffoli) gate, which is conditioned on two qubits.

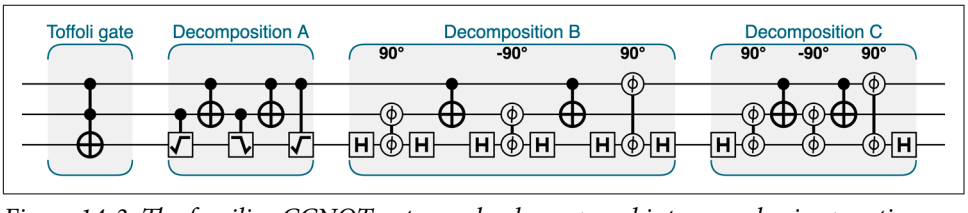


Figure 14-3. The familiar CCNOT gate can be decomposed into more basic operations

You may notice that all three of these decompositions follow the same pattern. This is not only the case for the `CCNOT`, any controlled-controlled operation (i.e., an operation conditioned on two other qubits) will have a similar decomposition. Figure 14-4 shows the general way we can implement a QPU operation controlled on two qubits, if we can find a QPU operation V satisfying $V^2 = U$ (i.e., where applying V twice to a register is the same as if we had applied U).

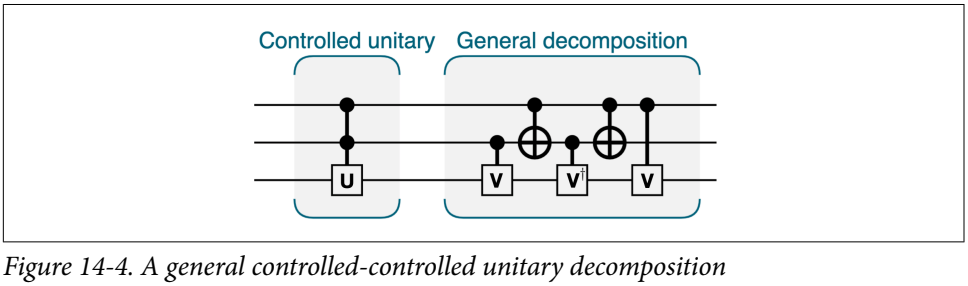


Figure 14-4. A general controlled-controlled unitary decomposition

Note that if we need help constructing the controlled-V operations in Figure 14-4, we can always refer back to Figure 14-3.

Finding the optimal decompositions of QPU algorithms in terms of simple QPU primitives and operations is not easy. The field of *quantum compiling* focuses on finding fast implementations of quantum algorithms. We'll mention a few more details about quantum compiling later in the chapter, and you can find some examples of quantum compiling optimizations online at *http://oreilly-qc.github.io?p=14-GD*.

Gate Teleportation

In Chapter 4, we used the quantum teleportation protocol to introduce ideas and notation that we used later throughout the book. While the teleportation of *information* does not feature heavily in most QPU applications, the ability to teleport *QPU operations* often does. This allows two parties to perform an operation on a quantum register, even if no one of the two parties has access to the state and operation in the same place. Like the teleportation protocol we saw in Chapter 4, this trick requires the use of a pair of entangled qubits.

A simple example of a gate teleportation protocol can be found online at *http://oreilly-qc.github.io?p=14-GT*.

QPU Hall of Fame

For most of the book, we have abstained from referring the reader to academic references. Here we've compiled a small list of the references that first introduced many of the ideas and algorithms that we've discussed in detail:

- Feynman (1982), "Simulating Physics with Computers"
- Deutsch (1985), "Quantum theory, the Church-Turing principle and the universal quantum computer"
- Deutsch (1989), "Quantum Computational Networks"
- Shor (1994), "Algorithms for Quantum Computation: Discrete Log and Factoring"
- Barenco et al. (1995), "Elementary gates for quantum computation"
- Grover (1996), "A fast quantum mechanical algorithm for database search"
- Brassard et al. (1998), "Quantum Counting"
- Brassard et al. (1998), "Quantum Amplitude Amplification and Estimation"
- Lloyd et al. (2009), "Quantum Algorithm for Solving Linear Systems of Equations"

Further references and resources can also be found in the lists of selected books and lecture notes at the end of the chapter.

The Race: Quantum Versus Conventional Computers

When discussing various QPU applications we've often been interested in comparing the performance of our newly learned QPU algorithms with their conventional counterparts. Although we've made comparisons on a case-by-case basis, interesting general comparisons can be made between the capabilities of quantum and conventional computers in terms of computational complexity. The computational complexity of a problem in computer science is given (roughly speaking) by the resources required to run the most efficient algorithm solving the problem. Computational complexity theory studies the classification of difficult problems according to their computational complexity.[4] For example, some problems are classified as *P*, which means that the resources required to find a solution to the problem scale polynomially with the size of the problem (e.g., matrix diagonalization). *NP* refers to the class of problems that have a *correct* solution that can be *checked* in polynomial time, but where these solutions cannot necessarily be *found* in polynomial time. This class has a twin, *co-NP*, which corresponds to the problems for which an *incorrect* answer to the problem can be verified in polynomial time. For example, the problem of factoring that we discussed in Chapter 12 is in both the NP and co-NP classes—it's easy to *check* whether a pair of numbers are (or aren't) prime factors, but not so easy to find them in the first place. Whether P=NP is a notorious open problem, well known thanks to multiple references in popular culture and the small matter of a $1M prize awaiting anyone who can address the question! It is widely suspected, with good reason, that the two classes are not equal. The NP-complete class that we mentioned in Chapter 10 corresponds, in some sense, to the most difficult problems in the NP class. Any other problem in NP can be reduced to one of these NP-complete problems, and if a polynomial-time solution is found for any NP-complete problem, all the problems in NP will also become solvable in polynomial time and change class to P.

The excitement surrounding quantum computing is mainly due to the fact that it appears able to reduce the computational complexity of certain problems. Note that this is not a blanket speedup of any conventional computing problem; only very specific classes of algorithms are currently known to enjoy quantum speedups. This reduction can be polynomial (e.g., from higher-order polynomial to lower-order), as is the case for the 3-SAT problem we looked at in Chapter 10, or superpolynomial (e.g., from exponential to polynomial), as is the case for factoring. Where can quantum computers provide a superpolynomial speedup? It turns out that problems that are *both* in NP and co-NP are prime suspects (pun intended) to have exponentially

4 A full description of its 500+ classes can be found in the Complexity Zoo.

faster QPU algorithms, and in fact, most of the algorithms for which such a speedup has been achieved belong to the intersection of these two classes.

Another important development that we initially mentioned in Chapter 13 is worth reiterating here. There have been a number of instances where completely conventional but quantum-inspired algorithms have emerged after quantum algorithms had been developed for those same problems. Therefore, research and understanding of quantum algorithms can lead to advances in conventional computing as well!

A Note on Oracle-Based Algorithms

Three of the earliest quantum algorithms shown to have a speedup on quantum versus conventional computers required calls to an *oracle*. An oracle provides information about a variable or a function, without revealing the variable or functions itself. The task in these early algorithms was to determine the variable or function used by the oracle in as few calls as possible. The required number of oracle calls in such a problem (and how this number scales with problem size) is usually referred to as the *query complexity*.

While these algorithms did not provide a useful computational advantage, they were crucial in the development of quantum computing, as they built understanding of the capabilities of quantum computers and eventually inspired researchers such as Peter Shor to develop more useful algorithms. Due to their pedagogical and historical importance, we briefly mention these early algorithms here, each of which is now known by the name of its inventor.

QCEngine code samples are also provided online at *http://oreilly-qc.github.io* to help you explore these pioneering quantum algorithms.

Deutsch-Jozsa

Oracle
: Takes a binary string of n bits and outputs a single bit that is the result of applying a function f to the binary string. We are promised that the function f is either constant, in which case the output bit will always be the same, or balanced, in which case there will be the same number of 0 and 1 outputs.

Problem
: Decide with absolute certainty whether f is constant or balanced in as few queries to the oracle as possible.

Query complexity
: Conventionally, we will need to make $2^{n-1} + 1$ queries to the oracle to be absolutely sure about the nature of the function. With a QPU, we can solve the problem with zero probability of error *in a single quantum query!*

This algorithm can be run online at *http://oreilly-qc.github.io?p=14-DJ*.

Bernstein-Vazirani

Oracle
: Takes a binary string, x, of n bits and outputs a single binary number. The output is obtained by $\sum_i x_i \cdot s_i$, where s is a secret string used by the oracle.

Problem
: Find the secret string s.

Query complexity
: Conventionally, we require n oracle queries, one to learn each of the input bits. However, using a QPU we can solve the problem with a single query.

This algorithm can be run online at *http://oreilly-qc.github.io?p=14-BV*.

Simon

Oracle
: Takes an n-bit binary string, x, and outputs a single integer. All the possible input strings are paired through a secret string s, such that two strings (x,y) will result in the same output if and only if $y = x \oplus s$ (where $\oplus$ denotes bitwise addition modulo 2).

Problem
: Find the secret string s.

Query complexity
: A conventional deterministic algorithm will require at least $2^{n-1} + 1$ oracle queries. By using Simon's quantum algorithm, we can find the solution in a number of calls that scales linearly in n, rather than exponentially.

This algorithm can be run online at *http://oreilly-qc.github.io?p=14-S*.

Quantum Programming Languages

A topic we have not touched upon is that of *quantum programming languages*—i.e., programming languages specifically developed with the quirks of quantum computing in mind. The applications and algorithms we've covered have been described using basic QPU operations (analogous to conventional binary logic gates), which we have orchestrated and controlled from a conventional programming language. This approach was followed for two reasons. Firstly, our goal has been to allow you to get hands-on and experiment with the abilities of a QPU. At the time of writing, quantum programming is arguably too nascent a field, with no existing universal standards. Secondly, the majority of quantum computing resources available to take you

beyond this book (other books, lecture notes, and online simulators) are all written in terms of the same basic QPU operations we have centered our discussion around.

On the topic of developing a quantum programming stack, one area that has seen a lot of recent interest and development is *quantum compiling*. Due to the characteristics of quantum error-correction codes, some QPU operations (such as `HAD`) are much easier to implement in a fault-tolerant manner than others (such as `PHASE`(45), also referred to as a `T` gate). Finding ways to compile quantum programs that account for such implementation constraints but don't adversely impact the speedup offered by a QPU is the task of quantum compiling. Literature surrounding quantum compiling often mentions the *T count* of a program, referring to the total number of difficult-to-perform `T` gates required. A common topic is also the preparation and distillation of so-called *magic states*,[5] particular quantum states that (when perfectly prepared) allow us to implement the elusive `T` gate.

At the time of writing, the study of quantum programming languages and quantum software toolchains would benefit greatly from input from experts in conventional computing—especially in the areas of debugging and verification. Some good starting points on the topic are listed here:

- Huang and Martonosi (2018), "QDB: From Quantum Algorithms Towards Correct Quantum Programs"
- Green et al. (2013), "Quipper: A Scalable Quantum Programming Language"
- Altenkirch and Grattage (2005), "A Functional Quantum Programming Language"
- Svore (2018), "Q#: Enabling Scalable Quantum Computing and Development with a High-Level Domain-Specific Language"
- Hietala et al. (2019), "Verified Optimization in a Quantum Intermediate Representation"
- Qiskit: An open-source software development kit (SDK) for working with OpenQASM and the IBM Q quantum processors
- Aleksandrowicz et al. (2019), "Qiskit: An Open-source Framework for Quantum Computing"

The Promise of Quantum Simulation

At the time of writing (circa 2019), quantum simulation is being proposed as the *killer app* of quantum computing. In fact, there are already some quantum algorithmic pro-

5 *Magic* is a technical term in quantum computing. As well as a magical one.

posals to address quantum chemistry problems that are intractable with classical computers. Some of the problems that could be solved using quantum simulation routines are:

Nitrogen fixation
: Find a catalyst to convert nitrogen to ammonia at room temperature. This process can be used to lower the cost of fertilizer, addressing hunger in third-world countries.

Room temperature superconductor
: Find a material that superconducts at room temperature. This material would allow for power transmission with virtually no losses.

Catalyst for carbon sequestration
: Find a catalyst to absorb carbon from the atmosphere, reducing the amount of CO_2 and slowing global warming.

It is clear that the potential for social and economic change that quantum computers can bring is unparalleled compared to most other technologies, which is one of the reasons there is such an interest in bringing about this technology.

Error Correction and NISQ Devices

The discussions of QPU operations, primitives, and applications in this book have all assumed (or simulated) the availability of *error-corrected* (also called logical) qubits. As with conventional computation, errors in registers and operations can quickly ruin quantum computation. A large body of research exists into quantum error-correction codes designed to counteract this effect. A remarkable result in the study of quantum error correction is the *threshold theorem*, which shows that if the rate of errors in a QPU falls below a certain threshold, then quantum error-correction codes will allow us to suppress the errors at the cost of only a small overhead to our computation. QPU applications only maintain their advantage over conventional algorithms under such low-noise conditions, and therefore likely require error-corrected qubits.

That said, at the time of writing there is a trend to search for QPU algorithms that might provide speedups over conventional computers even when run on Noisy Intermediate-Scale Quantum (NISQ) devices. These devices are understood to be composed of noisy qubits that do not undergo error correction. The hope is that algorithms might exist that are themselves intrinsically tolerant to QPU noise. However, circa 2019, no such algorithms capable of solving useful problems are known to exist.

Where Next?

In this book, we hope to have provided you with the conceptual tools and QPU intuition to further explore the fascinating topic of quantum computing. With the foun-

dation you now have, the references we list in this section are good places to take your understanding of QPUs to the next stage. Be warned (but not dissuaded!) that these references often freely use the more advanced levels of linear algebra and other mathematics that we've aimed to avoid in this text.

Books

- Nielsen and Chuang (2011), *Quantum Computation and Quantum Information*
- Mermin (2007), *Quantum Computer Science: An Introduction*
- Aaronson (2013), *Quantum Computing Since Democritus*
- Kitaev et al. (2002), *Classical and Quantum Computation*
- Watrous (2018), *The Theory of Quantum Information*
- Kaye et al. (2007), *An Introduction to Quantum Computing*
- Wilde (2013), *Quantum Information Theory*

Lecture Notes

- Aaronson, Quantum Information Science lecture notes
- Preskill, lecture notes on Quantum Computation
- Childs, lecture notes on quantum algorithms
- Watrous, Quantum Computation, lecture notes

Online Resources

- Vazirani, Quantum Mechanics and Quantum Computation
- Shor, Quantum Computation
- Quantum Algorithm Zoo

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

About the Authors

Eric R. Johnston ("EJ") is the creator of the QCEngine simulator, and also an acrobat and competitive gymnast. As an engineer, EJ values surprise and whimsy above all other things. He studied Electrical Engineering and Computer Science at U. C. Berkeley, and worked as a researcher in Quantum Engineering at the University of Bristol Centre for Quantum Photonics. EJ spent two decades at Lucasfilm as a software engineer for video games and movie effects, along with occasional motion-capture stunt performance. Currently, he works as a senior quantum engineer in Palo Alto, California.

Nicholas Harrigan is a physicist, programmer, and easily excitable science communicator. He received his doctorate from Imperial College London for research into quantum computing and the foundations of quantum mechanics. His work on quantum mechanics has moderately convinced him that the moon is still there when he doesn't look at it. Nicholas has since worked at the University of Bristol, and spent time as a data scientist and teacher. He now works as a quantum architect in a quantum computing startup. Nicholas is also a keen climber who finds constant motivation from the Unix command `yes`.

Mercedes Gimeno-Segovia is a quantum physicist whose main scientific goal is to develop the next generation of quantum technologies. Mercedes has always been fascinated by the inner workings of conventional computers and decided to merge that interest with her passion for quantum physics; she received her PhD from Imperial College London for her work on the first photonic quantum architecture compatible with the silicon industry. As Director of Quantum Architecture at PsiQuantum, she is working on the design of a general-purpose quantum computer. When not thinking about physics, Mercedes can be found playing the violin, running (preferably on trails) and reading.

Colophon

The animal on the cover of *Programming Quantum Computers* is the musky octopus (*Eledone moschata*), a sea creature found at depths of up to ¼ mile in the Mediterranean Sea and the coastal waters of western Europe.

Its smooth skin and musky odor make the musky octopus easy to identify. As seen on the cover, the skin is beige to gray-brown, and marked with dark brown spots. At night, these tentacles appear fringed with an iridescent blue border. Its eight tentacles are relatively short and carry only one row of suckers.

Unlike other octopuses, which mainly take shelter in rock crevices or vegetation, the musky octopus burrows into the sediment of the continental platform.

The musky octopus has a carnivorous diet, aided by its parrot-like beak, and mostly dines on crustaceans, mollusks, and small fish. Its siphon, or funnel, allows it to propel itself after prey or away from predators, emitting a cloud of black ink as defense against the latter.

While the musky octopus's current conservation status is designated as of Least Concern, many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from Dover's *Animals*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.