Chapter 1. Introduction

Whether you're an expert in software engineering, computer graphics, data science, or just a curious computerphile, this book is designed to show how the power of quantum computing might be relevant to you, by actually allowing you to start using it.

To facilitate this, the following chapters do *not* contain thorough explanations of quantum physics (the laws underlying quantum computing) or even quantum information theory (how those laws determine our abilities to process information). Instead, they present working examples providing insight into the capabilities of this exciting new technology. Most importantly, the code we present for these examples can be tweaked and adapted. This allows you to learn from them in the most effective way possible: by getting hands-on. Along the way, core concepts are explained as they are used, and only insofar as they build an intuition for writing quantum programs.

Our humble hope is that interested readers might be able to wield these insights to apply and augment applications of quantum computing in fields that physicists may not even have heard of. Admittedly, hoping to help spark a quantum revolution isn't *that* humble, but it's definitely exciting to be a pioneer.

Required Background

The physics underlying quantum computing is full of dense mathematics. But then so is the physics behind the transistor, and yet learning C++ need not involve a single physics equation. In this book we take a similarly *programmer-centric* approach, circumventing any significant mathematical background. That said, here is a short list of knowledge that may be helpful in digesting the concepts we introduce:

- Familiarity with programming control structures (`if`, `while`, etc.). JavaScript is used in this book to provide lightweight access to samples that can be run online. If you're new to JavaScript but have some prior programming experience, the level of background you need could likely be picked up in less than an hour. For a more thorough introduction to JavaScript, see *Learning JavaScript* by Todd Brown (O'Reilly).
- Some relevant programmer-level mathematics, necessitating:
    - An understanding of using mathematical functions
    - Familiarity with trigonometric functions
    - Comfort manipulating binary numbers and converting between binary and decimal representations

    - A comprehension of the basic meaning of complex numbers
- A very elementary understanding of how to assess the computational complexity of an algorithm (i.e., *big-o* notation).

One part of the book that reaches beyond these requirements is Chapter 13, where we survey a number of applications of quantum computing to machine learning. Due to space constraints our survey gives only very cursory introductions to each machine-learning application before showing how a quantum computer can provide an advantage. Although we intend the content to be understandable to a general reader, those wishing to really experiment with these applications will benefit from a bit more of a machine-learning background.

This book is about programming (not building, nor researching) quantum computers, which is why we can do without advanced mathematics and quantum theory. However, for those interested in exploring the more academic literature on the topic, Chapter 14 provides some good initial references and links the concepts we introduce to mathematical notations commonly used by the quantum computing research community.

What Is a QPU?

Despite its ubiquity, the term “quantum computer” can be a bit misleading. It conjures images of an entirely

new and alien kind of machine—one that supplants all existing computing software with a futuristic alternative.

At the time of writing this is a common, albeit huge, misconception. The promise of quantum computers stems not from them being a *conventional computer killer*, but rather from their ability to dramatically extend the kinds of problems that are tractable within computing. There are important computational problems that are easily calculable on a quantum computer, but that would quite literally be impossible on any conceivable standard computing device that we could ever hope to build.[1]

But crucially, these kinds of speedups have only been seen for certain problems (many of which we later elucidate on), and although it is anticipated that more will be discovered, it's highly unlikely that it would ever make sense to run *all* computations on a quantum computer. For most of the tasks taking up your laptop's clock cycles, a quantum computer performs no better.

In other words—from the programmer's point of view—a quantum computer is really a *co*-processor. In the past, computers have used a wide variety of co-processors, each suited to their own specialties, such as floating-point arithmetic, signal processing, and real-time graphics. With this in mind, we will use the term *QPU* (Quantum Processing Unit) to refer to the device on which our code samples run. We feel this reinforces the important context within which quantum computing should be placed.

As with other co-processors such as the GPU (Graphics Processing Unit), programming for a QPU involves the programmer writing code that will primarily run on the CPU (Central Processing Unit) of a normal computer. The CPU issues the QPU co-processor commands only to initiate tasks suited to its capabilities.

A Hands-on Approach

Hands-on samples form the backbone of this book. But at the time of writing, a full-blown, general-purpose QPU does not exist—so how can you hope to ever run our code? Fortunately (and excitingly), even at the time of writing a few prototype QPUs *are* currently available, and can be accessed on the cloud. Furthermore, for smaller problems it's possible to *simulate* the behavior of a QPU on conventional computing hardware. Although simulating larger QPU programs becomes impossible, for smaller code snippets it's a convenient way to learn how to control an actual QPU. The code samples in this book are compatible with both of these scenarios, and will remain both usable and pedagogical even as more sophisticated QPUs become available.

There are many QPU simulators, libraries, and systems available. You can find a list of links to several well-supported systems at *http://oreilly-qc.github.io*. On that page, we provide the code samples from this book, whenever possible, in a variety of languages. However, to prevent code samples from overwhelming the text, we

provide samples only in JavaScript for QCEngine. QCEngine is a free online quantum computation simulator, allowing users to run samples in a browser, with no software installation at all. This simulator was developed by the authors, initially for their own use and now as a companion for this book. QCEngine is especially useful for us, both because it can be run without the need to download any software and because it incorporates the *circle notation* that we use as a visualization tool throughout the book.

A QCEngine Primer

Since we'll rely heavily on QCEngine, it's worth spending a little time to see how to navigate the simulator, which you can find at *http://oreilly-qc.github.io*.

RUNNING CODE

The QCEngine web interface, shown in Figure 1-1, allows you to easily produce the various visualizations that we'll rely on. You can create these visualizations by simply entering code into the QCEngine code editor.

Figure 1-1. The QCEngine UI

To run one of the code samples from the book, select it from the drop-down list at the top of the editor and click the Run Program button. Some new interactive UI

elements will appear for visualizing the results of running your code (see Figure 1-2).

Quantum circuit visualizer

: This element presents a visual representation of the circuit representing your code. We introduce the symbols used in these circuits in Chapters 2 and 3. This view can also be used to interactively step through the program (see Figure 1-2).

Circle-notation visualizer

: This displays the so-called *circle-notation* visualization of the QPU (or simulator) register. We explain how to read and use this notation in Chapter 2.

QCEngine output console

: This is where any text appears that may have been printed from within your code (i.e., for debugging) using the `qc.print()` command. Anything printed with the standard JavaScript `console.log()` function will still go to your web browser's JavaScript console.

Figure 1-2. QCEngine UI elements for visualizing QPU results

DEBUGGING CODE

Debugging QPU programs can be tough. Quite often the easiest way to understand what a program is doing is to slowly step through it, inspecting the visualizations at each step. Hovering your mouse over the circuit visualizer, you should see a vertical orange line appear

at a fixed position and a gray vertical line wherever in the circuit your cursor happens to be. The orange line shows which position in the circuit (and therefore the program) the circle-notation visualizer currently represents. By default this is the end of the program, but by clicking other parts of the circuit, you can have the circle-notation visualizer show the configuration of the QPU at that point in the program. For example, Figure 1-3 shows how the circle-notation visualizer changes as we switch between two different steps in the default QCEngine program.

Figure 1-3. Stepping through a QCEngine program using the circuit and circle-notation visualizers

Having access to a QPU simulator, you're probably keen to start tinkering. Don't let us stop you!
In Chapter 2 we'll walk through code for increasingly complex QPU programs.

Native QPU Instructions

QCEngine is one of several tools allowing us to run and inspect QPU code, but what does QPU code actually look like? Conventional high-level languages are commonly used to control lower-level QPU instructions (as we've already seen with the JavaScript-based QCEngine). In this book we'll regularly cross between these levels. Describing the programming of a QPU with distinctly quantum machine-level operations helps us get

to grips with the fundamental novel logic of a QPU, while seeing how to manipulate these operations from higher-level conventional languages like JavaScript, Python, or C++ gives us a more pragmatic paradigm for actually writing code. The definition of new, bespoke, *quantum* programming languages is an active area of development. We won't highlight these in this book, but references for the interested reader are offered in Chapter 14.

To whet your appetite, we list some of the fundamental QPU instructions in Table 1-1, each of which will be explained in more detail within the chapters ahead.