A Hitchhikers Guide to Practical Quantum Computing for High Energy Physics

Vincent Croft, Elias Combarro, Michele Grossi

November 2024

Chapter 1

Introduction - Shared Concepts

Chapter 2

Wave Particle Duality... without the Waves: A Quantum Friendly Introduction to the Standard Model of Particle Physics

Chapter 3

The Shortest of Circuits: A physics friendly introduction to quantum computing with circuits

We have mentioned that quantum computing relies on quantum phenomena such as superposition, entanglement, and interference to perform computations. But what does this really mean? To make this explicit, we need to define a particular computational model that allow us to describe mathematically how to take advantage of all these properties.

There are many such models, including quantum Turing machines, measurement based quantum computing (also known as one-way quantum computing), or adiabatic quantum computing, and all of them are equivalent in power. However, the most popular one — and the one that we will be using today is the quantum circuit model.

Every computation has three elements: data, operations, and output. In the quantum circuit model, these correspond to some concepts that you may have already heard about: qubits, quantum gates, and measurements. Through the remainder of this chapter, we will briefly review all of them, highlighting some special details that will be of particular importance when talking about quantum machine learning and quantum optimization algorithms; at the same time, we will show the notation that will be used throughout the book. But before committing to that, let us have a quick overview of what a quantum circuit is.

One of the advantages of using a computational model is that you can forget about the particularities of the physical implementation of your computer and focus instead on the properties of the elements on which you store information and the operations you can perform on them. For instance, we could define a qubit as a (physical) quantum system that is capable of being in two different states. In practice, it could be a photon with two possible polarizations, a particle with two possible values for its spin, or a superconducting circuit, whose current can be flowing in one of two directions. When using the quantum circuit model, we can forget about those implementation details and just define a qubit... as a mathematical vector!

3.1 What is a qubit?

In fact, a qubit (short for quantum bit, sometimes also written as qbit, Qbit or even q-bit) is the minimal information unit in quantum computing. In the same way that a bit (short for binary digit) can be in state 0 or in state 1, a qubit can be in state $|0\rangle$ or in state $|1\rangle$ using standard Dirac notation.

The possible values for the state of a single qubit are vectors that live in a complex vector space of dimension 2 (in fact, they live in what is called a Hilbert space, but since we will be working today only with finite dimensions, there is no real difference). Thus we shall fix the vectors —0⟩ and —1⟩ as elements of a special basis, which we will refer to as the computational basis. We will represent these vectors, constituents of the computational basis, as the column vectors

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

and hence

$$a\,|0\rangle + b\,|1\rangle = a\begin{pmatrix} 1 \\ 0 \end{pmatrix} + b\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

If we are given a qubit and we want to determine or, rather, estimate its state, all we can do is perform a measurement and get one of two possible results: 0 or 1. We have nonetheless seen how a qubit can be in infinitely many states, so how does the state of a qubit determine the outcome of a measurement? As you likely already know, in quantum physics, these measurements are not deterministic, but probabilistic. In particular, given any qubit $a\,|0\rangle + b\,|1\rangle$, the probability of getting 0 upon a measurement is $|a|^2$, while that of getting 1 is $|b|^2$. Naturally, these two probabilities must add up to 1, hence the need for the normalization condition $|a|^2 + |b|^2 = 1$

If upon measuring a qubit we get, let's say, 0, we then know that, after the measurement, the state of the qubit is $|0\rangle$, and we say that the qubit has collapsed into that state. If we obtain 1, the state collapses to $|0\rangle$. Since we are obtaining results that correspond to $|0\rangle$ and $|1\rangle$, we say that we are measuring in **the computational basis** and a qubit is, mathematically, just a 2-dimensional vector that satisfies a normalization condition.

3.2 Gates

Since a qubit is, fundamentally, a quantum system, its evolution follows the laws of quantum mechanics. More precisely, if we suppose that our system is isolated from its environment, it obeys the **Schrödinger equation**. However, the only thing that you really need to know about quantum mechanics at this stage is that its solutions are always a special type of linear transformations. For the purposes of the quantum circuit model, since we are working in finite-dimensional spaces and we have fixed a basis, the operations can be described by matrices that are applied to the vectors that represent the states of the qubits.

3.2.1 Unitarity

But not any kind of matrix does the trick. According to quantum mechanics, the only matrices that we can use **unitary matrices**, which are matrices U such that

$$U^\dagger U = UU^\dagger = I$$

where I is the identity matrix and $U^\dagger$ is the adjoint of U, that is, the matrix obtained by transposing U and replacing each element by its complex conjugate. This means that any unitary matrix U is invertible and its inverse is given by $U^\dagger$. In the context of the quantum circuit model, the operations represented by these matrices are called quantum gates.

3.2.2 X Gate (Not)

When we have just one qubit, our unitary matrices need to be of size 2×2 because the state vector is of dimension 2. Thus, the simplest example of a quantum gate is the identity matrix of dimension 2, which transforms the state of the qubit by... well, by not transforming it at all. A less boring example is the X gate, whose matrix is given by

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

The X gate is also called the NOT gate, because its action on the elements of the computational basis is:

$$X\left|0\right\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \left|1\right\rangle$$

and

$$X\left|1\right\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \left|0\right\rangle$$

which is exactly what the NOT gate does in classical digital circuits.

3.2.3 Hadamard gate

A quantum gate with no classical analog is the **Hadamard** or H gate, given by

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

This gate is extremely useful in quantum computing, for it can create superposition. To be precise, if we apply the H gate on a qubit in state $|0\rangle$, we obtain

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

This state is so important that it has its own name and symbol. It is called the plus state and it is denoted by $|+\rangle$. In a similar way, we have that

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

is called the minus state denoted by $|-\rangle$

3.2.4 Z gate

Now we can start to apply several gates to the same qubit one after the other constructing simple circuits. E.g. the circuit HXH transforms a $|0\rangle$ state into $|0\rangle$ but $|1\rangle$ into $-|1\rangle$ this operation is also very important, and, of course, it has its own name: we call it the Z gate. From its action on $|0\rangle$ and $|1\rangle$, we can tell that its matrix will be

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

something that we could have also deduced by multiplying the matrices of the gates H, X, and H one after the other.

3.2.5 Pauli Matrices

Since there are X and Z gates, you may be wondering if there is also a Y gate. Indeed, there is one, given by matrix

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

The set I, X, Y, Z, are of course also known as the set of Pauli matrices, as seen in your particle theory lectures. They are of great importance in quantum computing. One of its many interesting properties is that it constitutes a basis of the vector space of 2×2 complex matrices.

3.2.6 More gates

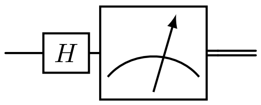
Other important one-qubit gates include the S and T gates, whose matrices are

$$S = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{pmatrix}, T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$$

But, of course, there is an (uncountably!) infinite number of 2-dimensional unitary matrices and we cannot just list them all here. What we will do instead is introduce a beautiful geometrical representation of single-qubit states, and, with it, we will explain how all one-qubit quantum gates can, in fact, be understood as certain kinds of rotations on the **Bloch Sphere**.

3.3 Constructing Circuits

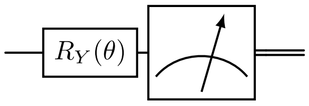
To put together everything that we have learned, we are going to create our very first complete quantum circuit. It looks like this:



It doesn't seem very impressive, but let's analyze it part by part. As you know, following convention, the initial state of our qubit is assumed to be $|0\rangle$, so that's what we have before we do anything. Then we apply the H gate, so the state changes to $\sqrt{1/2}\,|0\rangle + \sqrt{1/2}\,|1\rangle$. Finally, we measure the qubit. The probability of obtaining 0 will be $|\sqrt{1/2}|^2 = 1/2$ and that of getting 1 will also be 1/2, so we have created a circuit that — at least in theory — generates random bits following a perfectly uniform distribution.

3.3.1 Rotations

We can modify the previous circuit to obtain any distribution over 0 and 1 that we desire. If we want the probability of measuring 0 to be $p \in [0, 1]$, we just need to consider $\theta = 2 \arccos \sqrt{p}$ and the following circuit:



With this we can now perform arbitary operations on each qubit of our system. But not yet associate these qubits with each other.

3.4 Entangling gates

So far, we have worked with qubits in isolation. But the real power of quantum computing cannot be unleashed unless qubits can talk to each other. We will start by considering the simplest case of quantum systems in which there is qubit interaction: two-qubit systems. Of course, in a two-qubit system, each of the qubits can be in state $|0\rangle$ or in state $|1\rangle$. Thus, for the two qubits, we have four possible combinations: both are in state $|0\rangle$, the first one is in state $|0\rangle$ and the second one in state $|1\rangle$, the first one is in state $|1\rangle$ and the second one in state $|0\rangle$, or both are in state $|1\rangle$. These four possibilities form a basis (called the computational basis) of a 4-dimensional space and we denote them, respectively, by

$$|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle$$

Hence, the four basis states can be represented by four-dimensional column vectors given by

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

Usually, we omit the $\otimes$ symbol and just write

$$|0\rangle|0\rangle, |0\rangle|1\rangle, |1\rangle|0\rangle, |1\rangle|1\rangle$$

or

$$|00\rangle, |01\rangle, |10\rangle, |11\rangle$$

or even

$$|0\rangle, |1\rangle, |2\rangle, |3\rangle$$

As we have mentioned, these four states constitute a basis of the vector space of possible states for a two-qubit system. The general expression for the state of such a system is

$$|\psi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle$$

where a_{00}, a_{01}, a_{10}, and a_{11} are complex numbers representing the amplitudes.

If we measure in the computational basis both qubits at this generic state that we are considering, we will obtain 00 with probability $|a_{00}|^2$, 01 with probability $|a_{01}|^2$, 10 with probability $|a_{10}|^2$, and 11 with probability $|a_{11}|^2$. In all those cases, the state will collapse to the state corresponding to the outcome of the measurement, just as with one-qubit systems.

Let's now say that we only measure one of the qubits. What happens then? Suppose that we measure the first qubit. Then, the probability of obtaining 0 will be $|a_{00}|^2 + |a_{01}|^2$, which is the sum of the probabilities of all the outcomes in which the first qubit can be 0. If we measure the first qubit and the result turns out to be 0, the system will not collapse completely, but it will remain in the state

$$\frac{a_{00} |00\rangle + a_{01} |01\rangle}{\sqrt{|a_{00}|^2 + |a_{01}|^2}}$$

where we have divided by $\sqrt{|a_{00}|^2 + |a_{01}|^2}$ to keep the state normalized.

Of course, the operations that we can conduct on two-qubit systems need to be unitary. Thus, two-qubit quantum gates are 4×4 unitary matrices that act on 4-dimensional column vectors. The simplest way to construct such matrices is by taking the tensor product of two one-qubit quantum gates. Namely, if we consider two one-qubit gates U_1 and U_2 and two one-qubit states $|\psi_1\rangle$ and $|\psi_2\rangle$, we can form a two-qubit gate $U_1 \otimes U_2$ that acts on $|\psi_1\rangle \otimes |\psi_2\rangle$ as

$$(U_1 \otimes U_2)(|\psi_1\rangle \otimes |\psi_2\rangle) = (U_1 |\psi_1\rangle) \otimes (U_2 |\psi_2\rangle)$$

By linearity, we can extend $U_1 \otimes U_2$ to any combination of two-qubit states and we can associate a matrix to $U_1 \otimes U_2$. In fact, said matrix is given by the tensor product of the matrices associated to U_1 and U_2. Now it is easy (and a good excercise) to verify that this operation is indeed unitary and, hence, deserves the name of quantum gate. It's easy to see that tensor products of gates occur naturally when we have circuits with two qubits and pairs of individual one-qubit gates are acting on each of them.

You may complain that we haven't done anything new so far. And you would be right! In fact, quantum gates that are obtained as the tensor product of one-qubit gates can be seen as operations on isolated qubits that just happen to be applied at the same time. But wait and see! In the next subsection, we will introduce a completely different way of acting on two-qubit systems.

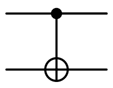
3.4.1 CNOT

In the two-qubit case, probably the most important unitary matrix that cannot be written as the tensor product of other simple matrices is the controlled-NOT (or controlled-X) gate, usually called the CNOT gate, given by the unitary matrix

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

It's a good exercise to compute the operation of this matrix in different circumstances has the effect that that the value of the second qubit is flipped if and only if the value of the first qubit is 1. Or, to put it in other words, the application of a NOT gate on the second qubit (that we call the target) is controlled by the first qubit. Now the name of this gate makes much more sense, doesn't it?

In a quantum circuit, the CNOT gate is represented as follows



Notice that the control qubit is indicated by a solid black circle and the target qubit is indicated by the $\otimes$ symbol (the symbol for an X gate can also be used instead of $\otimes$).

The most prominent use of the CNOT gate is, without a doubt, the ability to create **entanglement**.

3.4.2 Entanglement

Oddly enough, in order to define when a quantum system is entangled, we first need to define when it is not entangled. We say that a state $|\psi\rangle$ is a product state if it can be written as the tensor product of two other states $|\psi_1\rangle$ and $|\psi_2\rangle$, each of at least one qubit, as in

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$$

if $|\psi\rangle$ is not a product state, we say that it is entangled. For example, $|01\rangle$ is a product state, because we know that it is just another way of writing $|0\rangle \otimes |1\rangle$. Also, $\sqrt{1/2}(|00\rangle + |10\rangle)$ is a product state, because we can factor $|0\rangle$ on the second qubit. On the other hand, $\sqrt{1/2}(|00\rangle + |11\rangle)$ is an entangled state. No matter how hard you try, it is impossible to write it as a product of two one-qubit states.

When measured, entangled states can show correlations that go beyond what can be ex- plained with classical physics. For instance, if we have the entangled state $\sqrt{1/2}(|00\rangle + |11\rangle)$ and we measure the first qubit, we can obtain 0 or 1, each with probability 1/2. However, if we measure the second qubit afterwards, the result will be completely determined by the value obtained when measuring the first qubit and, in fact, will be exactly the same. If we invert the order and measure first the second qubit, then the result will be 0 or 1, with equal probability. But, in this case, the result of a subsequent measurement of the first qubit will be completely determined!

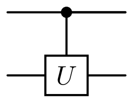
This phenomena is the basis for Chapter 5: Quantum Observables at Colliders, And, very importantly for us, entanglement is one of the most powerful resources available in quantum computing.

3.4.3 Controlled Gates

You may be wondering if, in addition to a controlled-X (or CNOT) gate, there are also controlled-Y, controlled-Z, or controlled-H gates. The answer is a resounding yes and, in fact, for any quantum gate U, it is possible to define a controlled-U (or, simply, CU) gate whose action on the computational basis is

$$CU\,|00\rangle = |00\rangle\,, CU\,|01\rangle = |01\rangle\,, CU\,|10\rangle = |1\rangle\,U\,|0\rangle\,, CU\,|11\rangle = |1\rangle\,U\,|1\rangle$$

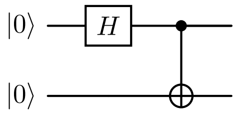
The circuit representation of a CU gate is similar to the one that we use for the CNOT gate, namely



where the solid black circle indicates the control and the box with U inside indicates the target. Constructing a controlled gate is simpler than it seems, provided your quantum computer already implements rotation gates and the two-qubit CNOT gate.

3.4.4 Soldering it up

To finish up let's create entangled states with the help of the CNOT gate. Consider the following circuit:



Initially, the state of the system is —00⟩. After we apply the H gate, we get into the state $\sqrt{1/2}(|00\rangle + |10\rangle)$ Finally, when we apply the CNOT gate, the state changes to $\sqrt{1/2}(|00\rangle + |11\rangle)$ which is indeed an entangled state.

The state $\sqrt{1/2}(|00\rangle + |11\rangle)$ is known as a **Bell state** of which there are 4 the others being $\sqrt{1/2}(|00\rangle - |11\rangle)$, $\sqrt{1/2}(|10\rangle + |01\rangle)$ and $\sqrt{1/2}(|10\rangle - |01\rangle)$ All of them are entangled, and they can be prepared with circuits similar to the preceding one.

3.4.5 Multiple qubits

Now that we have mastered working with two-qubit systems, it will be fairly straightforward to generalize all the notions that we have been studying to the case in which the number of qubits in our circuits is arbitrarily big. You know the drill: we will start by defining, mathematically, what a multi-qubit system is, we will then learn how to measure it and, finally, we will introduce quantum gates that act on many qubits at the same time.

Since n-qubit states are represented by 2^n-dimensional column vectors, n-qubit gates can be identified with $2^n 2^n$ unitary matrices. Similar to the two-qubit case, we can construct n-qubit gates by taking the tensor product of gates on a smaller number of qubits. Namely, if U_1 is an n_1-qubit gate and U_2 is an n_2-qubit gate, then $U_1 \otimes U_2$ is an $(n_1 + n_2)$-qubit gate and its matrix is given by the tensor product of the matrices U_1 and U_2.

However, there are n-qubit gates that cannot be constructed as tensor products of smaller gates. One such example is the **Toffoli** or **CCNOT** gate, a three-qubit gate that acts on the computational basis as

$$CCNOT \left|x\right\rangle \left|y\right\rangle \left|z\right\rangle = \left|x\right\rangle \left|y\right\rangle \left|z \oplus (x \wedge y)\right\rangle$$

where $\oplus$ is the XOR function and $\wedge$ is the symbol for the AND Boolean function. Thus, CCNOT applies a doubly controlled (in this case, by the first two qubits) NOT gate to the third qubit — hence the name!

The Toffoli gate is important because, using it and with the help of auxiliary qubits, we can construct any classical Boolean operator. This shows that, with quantum circuits, we can simulate the behavior of any classical digital circuit at the cost of using some additional ancillary qubits, since any Boolean function can be built with just negations and conjunctions. This is somewhat surprising, because we know that all quantum gates are invertible, while not all Boolean functions are. It then follows that we could make all of our digital circuits reversible just by implementing a classical version of the Toffoli gate!

Chapter 4

Quantum Machine Learning

4.1 Circuit Engineering

We are all very much looking forward to having a "Q1 Pro" quantum chip in our laptops, but — much to our regret — the technology is not there just yet. Nevertheless, we do have some actual quantum computers that, with their limitations, are able to execute quantum algorithms. And, furthermore, our good old classical computers can actually do a very decent job at simulating ideal quantum computers, at least for a low number of qubits. In this chapter, we will explore the tools that allow us to implement quantum algorithms using the quantum circuit model and run them on our laptops. We will begin by going through some excercises using one of the most widely-used quantum software frame-works available to date. PennyLane. PennyLane (https://pennylane.ai/) is a quantum framework built specifically for quantum machine learning, but it can perfectly be used as a general-purpose quantum computing framework. It is quite a newcomer to the quantum programming scene and it is being developed at Xanadu. Like Qiskit, it uses Python as a host language. Any quantum algorithms written in PennyLane can be sent to real quantum computers and executed in a broad collection of simulators. PennyLane is one of the best frameworks out there when it comes to interoperability. Thanks to a wide collection of plugins, you can export PennyLane circuits to other frameworks and execute them there — taking advantage of some of the features that these other frameworks may have. When it comes to machine learning, PennyLane provides some built-in tools, but it is also highly inter-operable with classical machine learning frameworks such as scikit-learn, Keras, TensorFlow, and PyTorch.

The way quantum circuits are built in PennyLane is fundamentally different to the way they are constructed in other software such as Qiskit. In Qiskit, if we wanted to implement a quantum circuit, we would initialize a QuantumCircuit object and manipulate it with some methods; some of these methods would be used to add gates to the circuit, some to perform measurements, and some to specify where we wanted to extract information about the state of the circuit.

In PennyLane, on the other hand, if you want to run a circuit, you need two elements: a Device object and a function that specifies the circuit. To put it in simple terms, a Device object is PennyLane's virtual analog of a quantum device. It is an object with methods that allow it to run any circuit that it is given (through a simulator, through an interface with other platforms, or however it may be!). For example, if we have a circuit and we want to run it on the default.qubit simulator (more on that later in this section) using two qubits, we will need to use this device:

```
dev = qml.device('default.qubit', wires = 2)
```

Notice, by the way, how the number of qubits available is a property of the device object itself.

Now that we have a device, we need to define the specification of our circuit. As we mentioned earlier, that is as easy as defining a function. In this function, we will execute instructions that will correspond to the actions of the quantum gates that we want to use. Lastly, the output of the function will be whichever information we want to get out of the circuit — whether it be the state of the circuit, some measurement samples, or whatever it may be. Of course, the output that we can get will depend on the device that we are using.

Let us illustrate this with an example:

```
def qc():
    qml.PauliX(wires = 0)
    qml.Hadamard(wires = 0)
    return qml.state()
```

Here we have a very basic circuit specification. In this circuit, we get the state vector (with `qml.state()`), after we first apply an X gate on the first qubit and then an H gate on the first qubit too. We do this by calling, in sequence, `qml.PauliX` and `qml.Hadamard`, specifying the wires on which we want the gates to act. In most non-parametrized gates, wires is the first positional argument, and it does not have a default value, so you need to provide one. In the case of single-qubit gates, this value must be an integer representing the qubit on which the gate is meant to act. Analogously, for multi-qubit gates, wires must be a list of integers.

In regard to rotation gates, we can apply R_X, R_Y, and R_Z parametrized by theta on a wire w using the instructions `qml.RX(phi=theta, wires=w)`, `qml.RY(phi=theta, wires=w)`, and `qml.RZ(phi=theta, wires=w)` respectively. In addition, the universal single-qubit gate $U(\theta, \phi, \lambda$ can be applied on a wire w calling `qml.U3(theta, phi, lambd, w)`.

Lastly, the controlled Pauli gates can be applied on a pair of qubits $w = [w0, w1]$ using the instructions `qml.CNOT(w)`, `qml.CY(w)` and `qml.CZ(w)`. The first wire, `w0`, is meant to be the control qubit, while the second one must be the target. Controlled X, Y, and Z rotations parametrized by an angle theta can be added with the instructions `qml.CRX(theta, w)`, `qml.CRY(theta, w)`, and `qml.CRZ(theta, w)` respectively.

In any case, we now have a two-qubit device dev and we have a circuit function qc. How do we assemble these two together and run the circuit? Easy, all we have to do is execute the following:

```
qcirc = qml.QNode(qc, dev) # Assemble the circuit & the device.
qcirc() # Run it!
```

If we run this, we will get the following result,

```
tensor([ 0.70710678+0.j, 0. +0.j, -0.70710678+0.j,
0. +0.j], requires_grad=True)
```

which makes perfect sense, for we know that

$$(HX \otimes I \left|00\right\rangle = (H \otimes I)\left|10\right\rangle = \frac{1}{\sqrt{2}}(\left|00\right\rangle - \left|10\right\rangle) \approx (0.7071\ldots)(\left|00\right\rangle - \left|10\right\rangle).$$

Notice how, consistent with PennyLane's convention for labelling states, the state vector is returned as a list with the amplitudes of the states in the computational basis. The first element corresponds to the amplitude state $\left|0\cdots 0\right\rangle$, the second one to that of $\left|0\cdots 01\right\rangle$, and so on.

But I'm lazy, all this process of defining a function and assembling it with a device seems overwhelmingly exhausting. Thankfully, the folks at PennyLane were kind enough to provide a shortcut. If you have a device dev and want to define a circuit for it, you could just do the following:

```
@qml.qnode(dev) # We add this decorator to use the device dev.
def qcirc():
    qml.PauliX(wires = 0)
    qml.Hadamard(wires = 0)
    return qml.state()

# Now qcirc is already a QNode. We can just run it!
qcirc()
```

Now that is much cuter! By placing the `@qml.qnode(dev)` decorator before the definition of our circuit function, it automatically became a QNode without us having to do anything else.

We have seen how circuits in PennyLane are implemented as simple functions, and this begs the question: are we allowed then to use parameters in these functions? The answer is a resounding yes. Let us say that we want to construct a one-qubit circuit, parametrized by a certain theta, which performs an X-rotation by this parameter. Doing so is as easy as this:

```
dev = qml.device('default.qubit', wires = 1)
@qml.qnode(dev)
def qcirc(theta):
    qml.RX(theta, wires = 0)
    return qml.state()
```

And, with this, for any value theta of our choice, we can run `qcirc(theta)` and get our result. This way of handling parameters is very handy and convenient. Of course, you can use loops and conditionals dependent on circuit parameters within the definition of a circuit. The possibilities are endless!

If at any point you need to draw a circuit in PennyLane, that is not an issue: it is fairly straightforward. Once you have a quantum node `qcirc`, you can pass this node to the `qml.draw` function. This will itself return a function, `qml.draw(qcirc)`, which will take the same arguments as qcirc and will give you a string that draws the circuit for each choice of those arguments.

So far, we have only performed simulations that return the state vector of the circuit at the end of its executions, but, naturally, that is just one of the many options that PennyLane provides. These are some, but not all, of the return values that we can have in a circuit function:

- If we want to get the state of the circuit at the end of its execution, we can, as we have seen, return `qml.state()`.
- If we wish to get a list with the probabilities of each state in the computational basis of a list of wires w, we can return `qml.probs(wires = w)`.
- We can get a sample of measurements in the computational basis of some wires w by returning qml.sample(wires = w); the wires argument is optional (if no value is provided, all qubits are measured). When we get a sample, we have to specify its size by either setting a shots argument when invoking the device or by setting it when calling the QNode.

We will explore some additional possibilities for return values shortly. But for now we already know how to get the state of the circuit. In order to illustrate the other return values that we may use, let us execute the following piece of code:

```
dev = qml.device('default.qubit', wires = 3)

# Get probabilities
@qml.qnode(dev)
def qcirc():
    qml.Hadamard(wires = 1)
    return qml.probs(wires = [1, 2]) # Only the last 2 wires.
prob = qcirc()
print("Probs. wires [1, 2] with H in wire 1:", prob)

# Get a sample, not having specified shots in the device.
@qml.qnode(dev)
def qcirc():
    qml.Hadamard(wires = 0)
    return qml.sample(wires = 0) # Only the first wire.
s1 = qcirc(shots = 4) # We specify the shots here.
```

```
print("Sample 1 after H:", s1)

# Get a sample with shots in the device.
dev = qml.device('default.qubit', wires = 2, shots = 4)
@qml.qnode(dev)
def qcirc():
    qml.Hadamard(wires=0)
    return qml.sample() # Will sample all wires.
s2 = qcirc()
print("Sample 2 after H x I:", s2)
```

The output we got with this execution is the following (the samples returned in your case will probably be different):

```
Probs. wires [1, 2] with H in wire 1: [0.5 0. 0.5 0. ]
Sample 1 after H: [0 1 0 0]
Sample 2 after H x I: [[1 0], [0 0], [0 0], [1 0]]
```

There might be a bit to unpack here. So, first of all, we are returned a list of probabilities; these are, in accordance with PennyLane's conventions, the probabilities of getting 00, 01, 10, and 11. In these possible outcomes, the first (leftmost) bit represents the outcome of the first measured qubit: in our case, since we are measuring wires [1, 2], the second wire of the circuit, wire 1. The second (rightmost) bit represents the outcome of the second measured qubit: in our case, the third wire of the circuit. For example, the first number in the list of probabilities represents the probability of getting 00(that is, 0in both wires). The second number in the list would be the probability of getting 01(0in wire 1 and 1 in wire 2). And so on.

Lastly, in the next two examples, we are getting some measurement samples. In the first case, we specify that we want to measure only the first qubit (wire 0), and, when we call the QNode, we ask for 4 shots; since we hadn't specified a default number of shots when defining the device, we need to do it in the execution. And with that, we have a sample of the first qubit. In our case, the results were first 0, then 1, and then two more zeros. In the last example, we define a two-qubit circuit and we measure all the wires. We already specified a default number of shots (4) when we defined the device, so we don't need to do it when calling the QNode. And, upon execution, we are given a sample of measurements. Each item in the list corresponds to a sample. Within each sample, the first element gives the result of measuring the first qubit of the circuit, the second element the result of measuring the second qubit, and so on it would go. For example, in our case, we see that in the first measurement we obtained 1 on the first qubit and 0 on the second one.

So far, we have been working with devices based on the default.qubit simulator, which is a Python-based simulator with some basic functionalities. We will introduce more simulators when we dive into the world of quantum machine learning. For now, however, you should at least know about the existence of the lightning.qubit simulator, which relies on a C++ backend and provides a significant boost in performance, especially for circuits with a large number of qubits. Its usage is analogous to that of the default.qubit simulator. Furthermore, there is a lightning.gpu simulator that can enable the Lightning simulator to rely on your GPU. It can be installed as a plugin however such technology is currently in a massive state of flux as the community and software develops.