

Getting Started With  
**Quantum Bayesian Networks**

In Python And Qiskit

**Dr. Frank Zickert**



Copyright © 2022 Dr. Frank Zickert

PUBLISHED BY PYQML

[www.pyqml.com](http://www.pyqml.com)

The contents of this book, unless otherwise indicated, are Copyright © 2022 Dr. Frank Zickert, [pyqml.com](http://pyqml.com). All rights reserved. Books like this are made possible by the time invested by the authors. If you received this book and did not purchase it, please consider making future books possible by buying a copy at <https://www.pyqml.com> today.

Release 0.1, October 2022

# Contents

1	Introduction	4
1.1	You Don't Need To Be A Mathematician	6
1.2	Applied Quantum Computing Is a tool for statistical modeling	9
1.3	Bayesian Networks	10
2	Preparation	13
2.1	Configuring Your Quantum Machine Learning Workstation	13
2.2	Quantum Circuit	17
2.3	The Quantum Superposition	20
2.4	Qubit Operators	21
3	Variables	32
3.1	Marginal Probability	32
3.2	Joint probability	35
3.3	Conditional Probability	36
4	Complete QBN	41
5	Conclusion	48

# 1. Introduction

In the recent past, we have witnessed how algorithms learned to drive cars and beat world champions in chess and Go. Machine learning is being applied to virtually every imaginable sector, from military to aerospace, from agriculture to manufacturing, and from finance to healthcare.

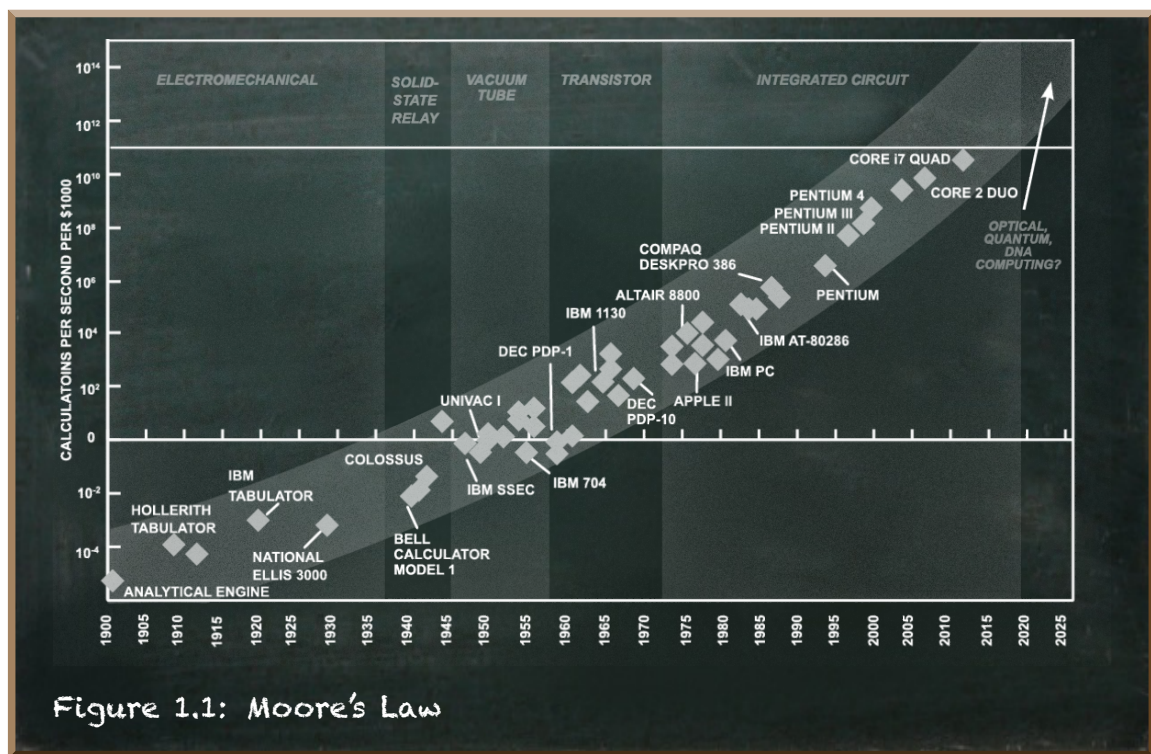


Figure 1.1: Moore's Law

But these algorithms become increasingly hard to train because they consist of billions of parameters. Perhaps we are at the end of technological progress as we know it. Moore's law comes to an end.

What appears to result in a slow-down of technological progress might instead result in a disruptive change.

Quantum computing promises to be the most disruptive technology of the 21st century. Unlike classical computers based on sequential information processing, quantum computing uses the properties of quantum physics: superposition, entanglement, and interference. But rather than increasing the available computing capacity, it reduces the capacity needed to solve a problem. Quantum computers promise to solve such problems that are intractable with current computing technologies.

When this disruption happens, you'd better be prepared. If you wait until the change happens, it might be too late. Even if it is not too late to be left behind, you'd certainly miss the first-mover advantage and belong to this shift's winners. It's now up to you to set the course for whether this becomes a threat to your career or a once-in-a-lifetime opportunity to gain a competitive advantage.

Of course, quantum computing is nothing you learn in a day. Quantum computing requires us to change the way we think about computers. It requires a whole new set of algorithms. Algorithms that encode and use quantum information. This includes machine learning algorithms.

And it requires a new set of developers. Developers who understand machine learning and quantum computing. Developers capable of solving practical problems that have not been solved before. A rare type of developer. The ability to solve quantum machine learning problems already sets you apart from all the others. And, of course, the field of quantum computing is still in the process of evolving. But if you wait until it becomes mainstream, you've lost the opportunity to turn your knowledge into a competitive advantage. Instead, you're pushed to keep up so you don't get left behind. Then it becomes a threat.

If you want to get started with quantum machine learning, you'll love quantum Bayesian networks (QBN). They are intuitive, and thus, easy to understand. Yet, they use the fundamental quantum computing concepts. Therefore, you can learn the concepts hands-on.



## 1.1 You Don't Need To Be A Mathematician

Scientific papers and textbooks about quantum computing are full of mathematical formulae. Even blog posts on quantum computing are loaded with mathematical jargon. It starts with the first concept you encounter. The quantum superposition:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \text{ with } \alpha^2 + \beta^2 = 1$$

Figure 1.2: Quantum Superposition

As a non-mathematician, this formula might already be too much. If you're not familiar with the used Dirac-notation ( $|\psi\rangle$ ) or if you're not used to working with vectors, then such a formula is as good as Egyptian hieroglyphs:



Figure 1.3: Hieroglyphs

I am ambivalent about mathematics.

For most, including me, it is a real struggle. It starts with the implicit meaning of symbols. Moreover, many equations leave out the steps in between. So, it is impossible to follow the derivation of the equation. Of course, the remaining part of the equation may suffice to describe the critical relationship between two terms, but it doesn't foster a deep understanding of the topic.

Don't get me wrong. Math is a great way to describe technical concepts. Math is a concise yet precise language. Our natural languages, such as English, by contrast, are lengthy and imprecise. It takes a whole book full of natural language to explain a small collection of mathematical formulae.

But most of us are far better at understanding natural language than math. We learn our mother tongue as a young child and we practice it every single day. We even dream in our natural language. I couldn't tell if some fellows dream in math, though.

For most of us, math is, at best, a foreign language. Only the fewest people look at an unknown equation and understand it immediately. By contrast, most of us have to work through equations thoroughly to understand their meaning. Long story short, a mathematical equation condenses a lot of information into very few symbols. But just because you can write it down more quickly than a 400-page book doesn't mean that the reader understands it as quickly.

I am convinced that it will take more time if the reader is left alone with it. When we're about to learn something new, it is easier for us if we use our mother tongue. It is hard enough to grasp the meaning of the new concept. If we're taught in a foreign language, it is even harder. If not impossible.

Of course, math is the native language of quantum mechanics and quantum computing, if you will. But why should we teach quantum computing only in its own language? Shouldn't we try to explain it in a way more accessible to the learner? I'd say "absolutely"!

Teaching something in the language of the learner doesn't mean we should not have a look at the math. We should! But, we use math when its precision helps us to explain how things work. But we do not use math as an excuse to not explain an important concept. Moreover, math is not the only precise language we have. We have languages that are as precise as mathematical formulae. And nowadays, these languages come almost natural to many. These languages are programming languages.

I do not mean the syntax of a specific programming language. Rather, I refer to a way of thinking almost all programming languages share. From Python to Java, from Javascript to Ruby, even from C to Cobol. All these languages build upon boolean logic. Regardless of programming language, a programmer works a lot with boolean logic.

Most prominently, boolean logic appears in conditional statements: if then else.

Listing 1.1

```
1 if x and y: # A statement to evaluate in boolean logic
2     doSomething () # if the statement evaluates to True
3 else:
4     doSomethingElse () #otherwise
```

The if-part of a conditional statement is pure boolean logic. Often, it contains the basic boolean operators `not`, `and`, and `or`.

If some statement is true, then its negation is false. Conversely, if a statement is false, then its negation is true. If a statement consists of two parts  $P$  and  $Q$ , then  $P$  and  $Q$  is only true if  $P$  is true and  $Q$  is true. But  $P$  or  $Q$  is true if either  $P$  or  $Q$  is true.

Here are a few examples of boolean logic in Python.

Listing 1.2: Boolean logic in Python

```
1 P = True
2 Q = False
3
4 print('not P is {}'.format(not P))
5 print('P and Q is {}'.format(P and Q))
6 print('P or Q is {}'.format(P or Q))
7 print('P and not Q is {}'.format(P and not Q))
```

Truth tables are the most common form to represent boolean logic. In the next table, we make use of some mathematic symbols to represent `not` ( $\neg$ ), `and` ( $\wedge$ ), and `or` ( $\vee$ ). But while the representation of a concept may differ when you describe it in Python or in math, the concept itself is the same. So, you don't need to be a mathematician to understand boolean logic. Actually, you don't need to be a programmer, either. You only need to be capable of pure logical reasoning.

In the following truth table, we have two variables,  $P$  and  $Q$ . Each variable is either true ( $T$ ) or false ( $F$ ). Depending on the combination of their values, we can deduce the value of any boolean statement. The following figure depicts the truth table for  $P$ ,  $Q$ , `not P`, `not Q`, `not P and not Q`, `not (not P and not Q)`, and  $P$  or  $Q$ .

In boolean logic, we only need very few basic operators. In fact, the “not and” (NAND) operator itself is functional complete. You can compose any arbitrary behavior out of this one operator. The main reason why there are only



P	Q	$\neg P$	$\neg Q$	$\neg P \wedge \neg Q$	$\neg (\neg P \wedge \neg Q)$	$P \vee Q$
T	T	F	F	F	T	T
T	F	F	T	F	T	T
F	T	T	F	F	T	T
F	F	T	T	T	F	F

Figure 1.4: Truth table

a few operators is that the underlying value is pretty simple. Logically, it is a boolean value that is either true or false. Computationally, it is either 1 or 0. And technically it is either current on or off.

In the end, the task in classical programming is to compose these basic operators to create increasingly complex programs.

## 1.2 Applied Quantum Computing Is a tool for statistical modeling

So, the concept underlying classical computing is logical operators. To succeed, you'll need to learn how to compose them.

And, of course, there is also an important concept underlying quantum computing that you will need to understand if you want to use quantum computing. But, it's not physics.

In quantum computing, we talk about quantum superposition — a complex linear combination of basis states — and quantum entanglement — the perfect correlation of distant particles. These are spectacular phenomena that define the subatomic world of quantum mechanics. And to understand them, you certainly need a degree in theoretical physics and a passion for mathematics.

The critical point, however, is that you don't need to understand quantum mechanics to use quantum computers, just as you don't need to know how transistors work if you want to use a classical computer.

If you want to use a computer — and by use, I mean to solve problems with it — whether classical or quantum, you have to program it.

Classical programming is about creating step-by-step instructions to solve a

problem. You use logical reasoning to develop rules that decide which path to take when a decision is at hand.

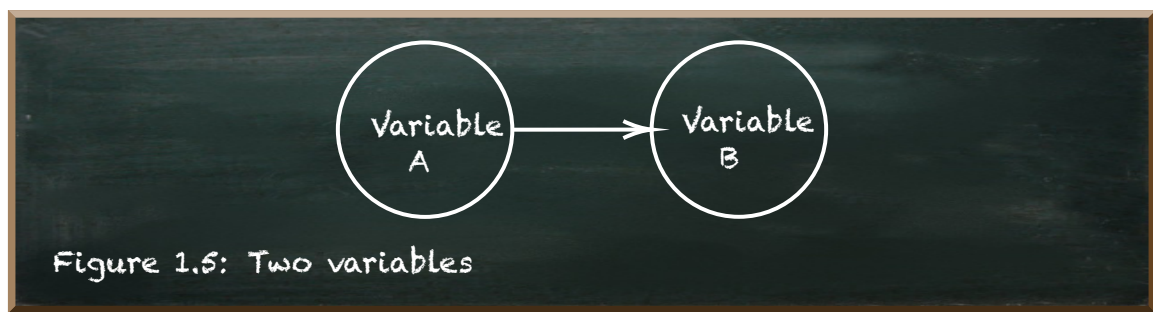
By contrast, quantum computing is a tool for statistical modeling. We encode our statistical assumptions about the world in an executable model — the quantum circuit. That circuit enables us, data scientists, to conduct analyses and infer relationships between variables, discover insights, and to make statistical predictions about the world.

So, programming a quantum computer is not about writing software that behaves the way you want it to. The process of creating such a statistical model is very different from writing software. Therefore, it is best not to think about “writing software” at all. Instead, we should refer to the data science process of applying statistical analysis to data sets. This process is all about variables and their relationship to each other. First, you need to identify

- (a) explanatory (independent, predictive) variables that explain variations in
- (b) response (dependent, outcome) variables.

Then you need to determine the form of the relationships between the explanatory and response variables and code them in your model.

The following figure depicts two variables, *A* and *B*. *A* is an independent variable. It can have any arbitrary value. If it is a boolean variable, for instance, it can be either true or false. But we don’t know much about it, yet. *B*, however, is a dependent variable. It can also be anything. Yet, we know that its value depends somehow on *A*’s value.

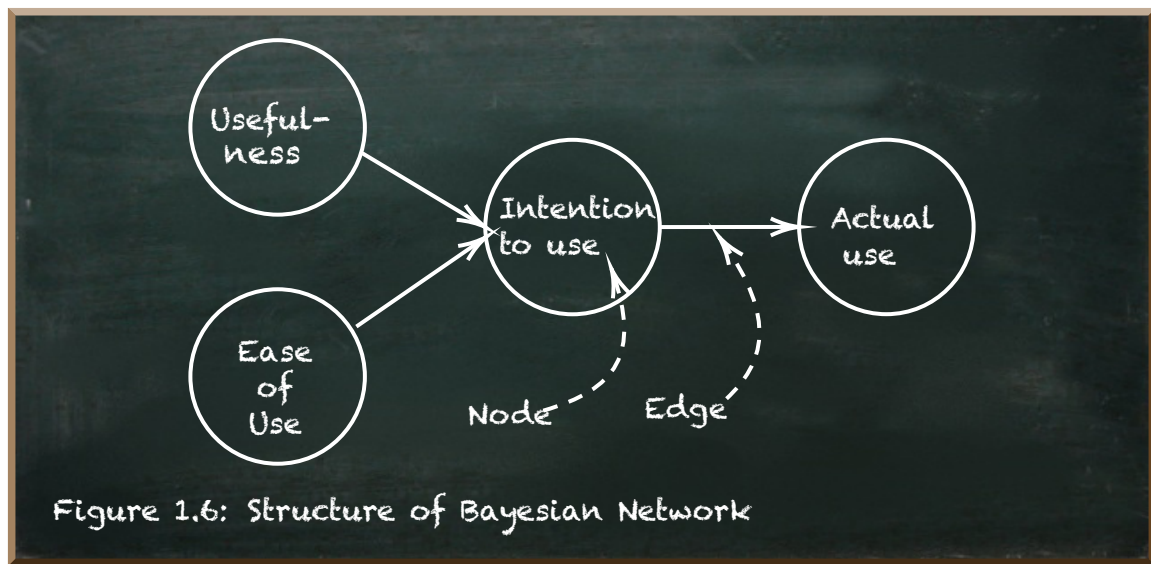


## 1.3 Bayesian Networks

Understanding variables is the building block for more advanced models—Bayesian networks.

Bayesian networks are probabilistic models that model knowledge about

an uncertain domain. We represent Bayesian networks as directed acyclic graphs with nodes and edges as depicted in the following figure.



This figure contains a version of the technology acceptance model (based on Davis, F. D. (1989), “Perceived usefulness, perceived ease of use, and user acceptance of information technology”, *MIS Quarterly*, 13 (3): 319–340). It models how users come to accept and actually use a technology. The model suggests that when users are presented with a new technology, a number of factors influence their decision about whether they will use it.

The actual use is the variable that tells us whether people use a technology or not. This decision is influenced by the user’s intention to use it. The intention again depends on

- a) the (perceived) usefulness of the system—that is whether a person believes that using the system would enhance her job performance; and
- b) the (perceived) ease of use—that is whether a person believes using a particular system would be free from effort.

In the graphical representation, the nodes represent random variables, such as the ease-of-use or the actual use. The edges correspond to the direct influence of one variable on another. In other words, the edges define the relationship between two variables. The directions of the arrows are important, too. The node connected to the tail of the arrow is the parent node. The node connected to the head is the child node. The child node depends on the parent node.

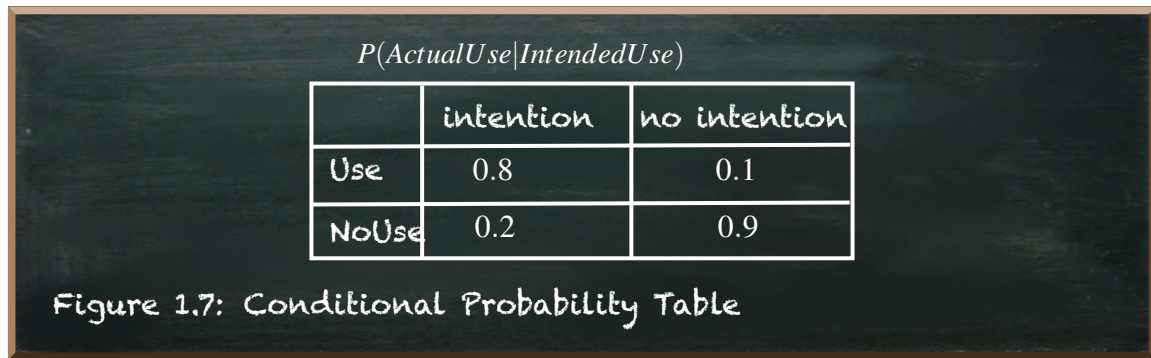
Moreover, Bayesian networks are probabilistic models that model knowledge about an uncertain domain. So, even though we do not know whether a per-

son is going to use a system, we can calculate the corresponding probability based on our observations and our beliefs.

We express our observations and beliefs in conditional probability tables (CPT) for discrete variables and conditional probability distributions (CPD) for continuous variables.

For instance, we might believe that if someone intends to use the system will actually use the system with a probability of 80%. This value might result from pure belief as well as from well-researched studies. Of course, the more evidence we have the better. But, Bayesian networks are powerful tools not least because they work even without verified data. Of course, we need to be careful with the interpretation of any results we obtain from them.

Bayesian networks do not represent the real world but our beliefs of it.



$P(\text{ActualUse}|\text{IntendedUse})$

	intention	no intention
Use	0.8	0.1
NoUse	0.2	0.9

Figure 1.7: Conditional Probability Table

As we see, the CPT of the variable representing the actual use contains different values. Besides the probability of 80% given the user intends to use the system, it also contains the probability of actual use given the user does not intend to use the system (here 10%). Furthermore, it also contains the probabilities of not using the system. These values result from subtracting the other probabilities from 100%.

## 2. Preparation

### 2.1 Configuring Your Quantum Machine Learning Workstation

Even though this book is about quantum machine learning, I don't expect you to have a quantum computer at your disposal. Thus, we will run most of the code examples in a simulated environment on your local machine. But we will need to compile and install some dependencies first.

We will use the following software stack:

- Unix-based operating system (not required but recommended)
- Python, including pip
- Jupyter (not required but recommended)
- Qiskit

For all examples, we use Python as our programming language. Python is easy to learn. Its simple syntax allows you to concentrate on learning quantum machine learning rather than spending your time with the specificities of the language.

Most importantly, quantum computing tools, such as Qiskit and Cirq, are available as Python SDKs.

Jupyter notebooks are a great way to run quantum machine learning experiments. They are a de facto standard in the machine-learning and quantum computing communities. A notebook is a file format (`.ipynb`). The [Jupyter](#)

[Notebook app](#) lets you edit your file in the browser while running the Python code in interactive Python kernels. The kernel keeps the state in memory until it is terminated or restarted. This state contains the variables defined during the evaluation of code.

A notebook allows you to break up long experiments into smaller pieces you can execute independently. You don't need to rerun all the code every time you make a change. But you can interact with it.

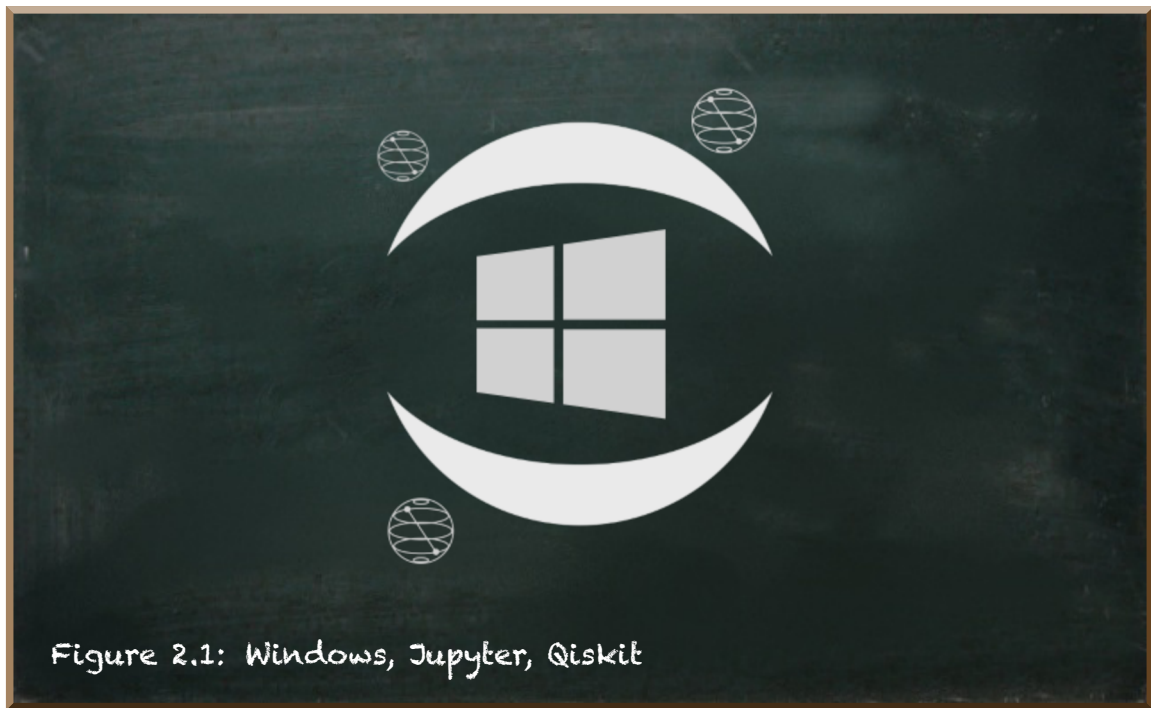
Like most programming languages, Python has its package installer. This is `pip`. It installs packages from the Python Package Index (PyPI) and other indexes. By default, it installs the packages in the same base directory shared among all your Python projects. Thus, it makes an installed package available to all your projects. This seems to be good because you don't need to install the same packages repeatedly.

However, if any two of your projects require different versions of a package, you'll be in trouble because there is no differentiation between versions. You would need to uninstall one version and install another whenever you switch working on either one of the projects.

This is where virtual environments come into play. Their purpose is to create an isolated environment for each of your Python projects. It's no surprise, using Python virtual environments is the best practice.

An Ubuntu Linux environment is highly recommended when working with quantum machine learning and Python because all the tools you need can be installed and configured quickly. If you run Windows, I strongly recommend to install and use the Windows Subsystem for Linux (WSL2).





The Windows Subsystem for Linux lets you run a full Ubuntu Linux inside Windows. Windows 10 must be updated to version 2004 and Intel's virtualization technology must be enabled in BIOS settings.

In the first step, we need to activate the Windows Subsystem for Linux optional feature. Open PowerShell as Administrator and run the following command:

```
dism.exe /online /enable-feature  
/featurename:Microsoft-Windows-Subsystem-Linux  
/all /norestart
```

In the next step, we update the subsystem to WSL2. Download the latest kernel update for your system from <https://aka.ms/wsl2kernel> and install the MSI package.

Now, we enable the Virtual machine platform and set WSL2 as the default version.

```
dism.exe /online /enable-feature  
/featurename:VirtualMachinePlatform  
/all /norestart  
wsl --set-default-version 2
```

Finally, we can install a Linux distribution as if it was a normal program.

Open the Microsoft store, search for “Ubuntu 20.04 LTS”, and install the program. Once the installation finishes, you can start Ubuntu from your start menu. On the first start, you need to create a new Unix user and specify a password.

Now, you can proceed with the installation of the libraries and packages in Ubuntu.

We accomplish all steps by using the Linux terminal. To start, open up your command line and update the apt-get package manager.

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install -y build-essential wget python3-dev \
    libreadline-gplv2-dev libncursesw5-dev libssl-dev \
    libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev \
    libffi-dev
```

The next step downloads and installs Python 3.10.7 (the latest stable release at the time of writing).

```
$ mkdir /tmp/Python310
$ cd /tmp/Python310
$ wget https://www.python.org/ftp/python/3.10.7/Python-3.10.7.tar.xz
$ tar xvf Python-3.10.7.tar.xz
$ cd /tmp/Python310/Python-3.10.7
$ ./configure
$ sudo make altinstall
```

If you want to have this Python version as the default, run

```
$ sudo ln -s /usr/local/bin/python3.8 /usr/bin/python
```

Python is ready to work. Let’s now install and update the Python package manager pip:

```
$ wget https://bootstrap.pypa.io/get-pip.py && python get-pip.py
$ pip install --upgrade pip
```

You might need to restart your machine to recognize pip as a command.

As mentioned, we install all the Python packages in a virtual environment. So, we need to install virtualenv:

```
$ sudo apt-get install python3-venv
```

To create a virtual environment, go to your project directory and run venv.

The following parameter (here `env`) specifies the name of your environment.

```
$ python -m venv env
```

You'll need to activate your environment before you can start installing or using packages.

```
$ source env/bin/activate
```

When you're done working on this project, you can leave your virtual environment by running the command `deactivate`. If you want to reenter, call `source env/bin/activate` again.

We're now ready to install the packages we need.

Install [Jupyter](#):

```
$ pip install jupyter notebook jupyterlab --upgrade
```

Install [Qiskit](#) and Matplotlib

```
$ pip install qiskit matplotlib
```

If you don't install Qiskit in the virtual environment, you should add the `--user` flag. Otherwise, the installation might fail due to missing permissions.

You're now ready to start. Open up JupyterLab with

```
$ jupyter lab
```

## 2.2 Quantum Circuit

The fundamental unit of Qiskit is the quantum circuit. A quantum circuit is a model for quantum computation. The program, if you will.

The following code listing defines a quantum circuit and executes it using a local simulator. It serves as a blueprint for all the circuits we will be running today.

Listing 2.1

```
1 from qiskit import QuantumCircuit, execute, Aer
2 from qiskit.visualization import plot_histogram
3
4 # Create a quantum circuit with one qubit
5 qc = QuantumCircuit(1)
6
7 # YOUR CODE GOES HERE
8 qc.h(0)
9
10 # measure the qubit
11 qc.measure_all()
12
13 # Tell Qiskit how to simulate our circuit
14 backend = Aer.get_backend('qasm_simulator')
15
16 # Do the simulation, returning the result
17 results = execute(qc, backend, shots=1000).result()
18
19 # get the probability distribution
20 counts = results.get_counts()
21
22 # Show the histogram
23 plot_histogram(counts)
```

First, we create a `QuantumCircuit` and pass the number of qubits we want it to have (line 5).

Second, we add some instructions to the circuit (line 8). We will learn about the instructions in a minute. Right now, only remember that your individual code goes here.

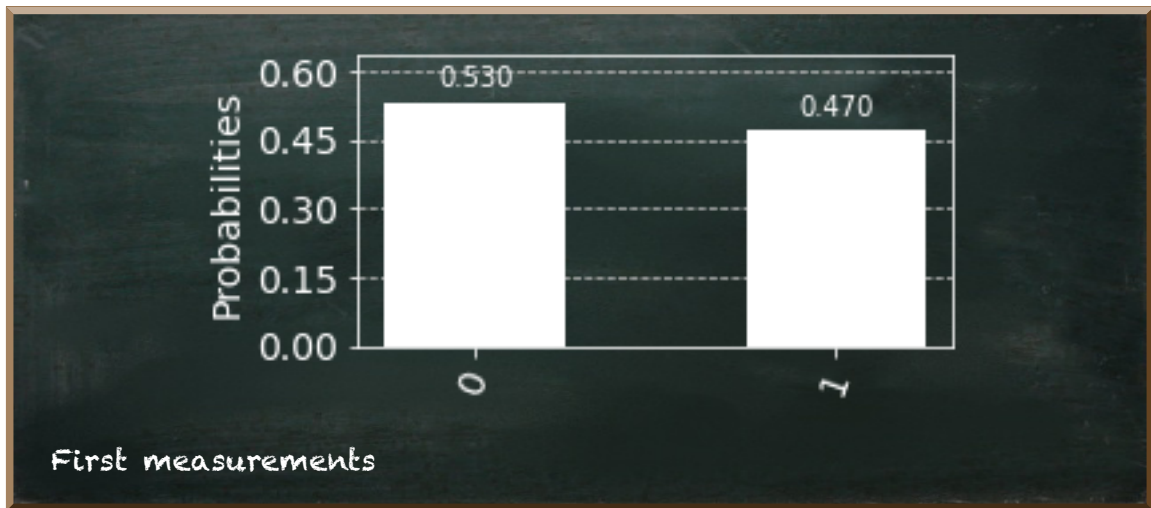
Third, we measure all the qubits (line 11). The theory tells us that a qubit is in the state of superposition only as long as you do not look at it. But, first, that's not entirely true. And second, who cares about the theory, right? So, let's just say we collect the values from the qubit. We always do this at the very end of the quantum circuit.

Fourth, we boot our quantum computer. Unfortunately not. But we create a backend that simulates a quantum computer (line 14).

Fifth, we run the `QuantumCircuit` using the backend for 1,000 times (`shots`) and obtain the result (line 17).

Finally, we take the counts from the result (line 20) and display them in a histogram (line 23).

The following figure depicts the result.



Out of the 1,000 times, we executed the circuit, we observed the qubit as a 0 530 times and a 1 470 times.

When you run the code, the exact values will differ. They differ because the qubit is a probabilistic system. Whether you measure it as a 0 or a 1 depends on chance—and even more importantly—the qubit state. But before we learn more about the quantum state, we define a function to take the parts of running the circuit that we can reuse.

Listing 2.2: Convenience function to handle QuantumCircuits

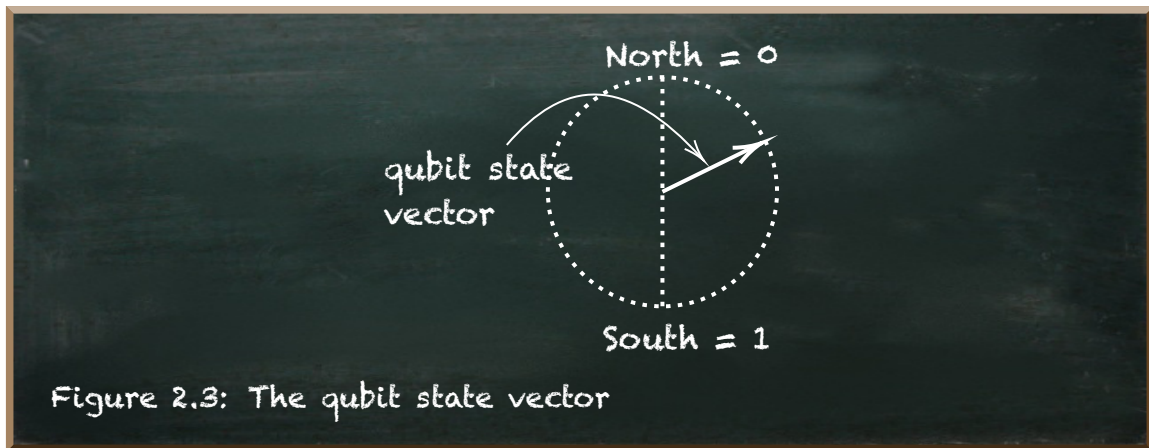
```
1 from qiskit import QuantumCircuit, Aer, execute
2 from qiskit.visualization import plot_histogram
3
4 def run_circuit(qc, simulator='qasm_simulator', shots=1000, hist=True):
5     # Tell Qiskit how to simulate our circuit
6     backend = Aer.get_backend(simulator)
7
8     # execute the qc
9     results = execute(qc, backend, shots=shots).result().get_counts()
10
11     # plot the results
12     return plot_histogram(results, figsize=(8,4)) if hist else results
```

This function takes a `QuantumCircuit` (steps 1-3 from above), runs it, and shows the resulting histogram steps 4-6 from above).

## 2.3 The Quantum Superposition

Like all quantum mechanical systems, a quantum bit (qubit) is in a state of superposition. This is a complex (as in complex number) linear combination of the basis states. A basis state, in turn, is a value that the qubit can assume once we measure it. A qubit has two basis states  $|0\rangle$  and  $|1\rangle$ , with the state  $|0\rangle$  corresponds to measuring the qubit as 0 and  $|1\rangle$  as 1.

This peculiar specific notation is known as the Dirac notation. It is a short way of writing vectors. A vector is a graphical object that has a length and a direction. If drawn in a coordinate system, the vector starts in the center and ends at the point specified by the numbers in the vector. The following figure depicts such a vector that represents the qubit state.

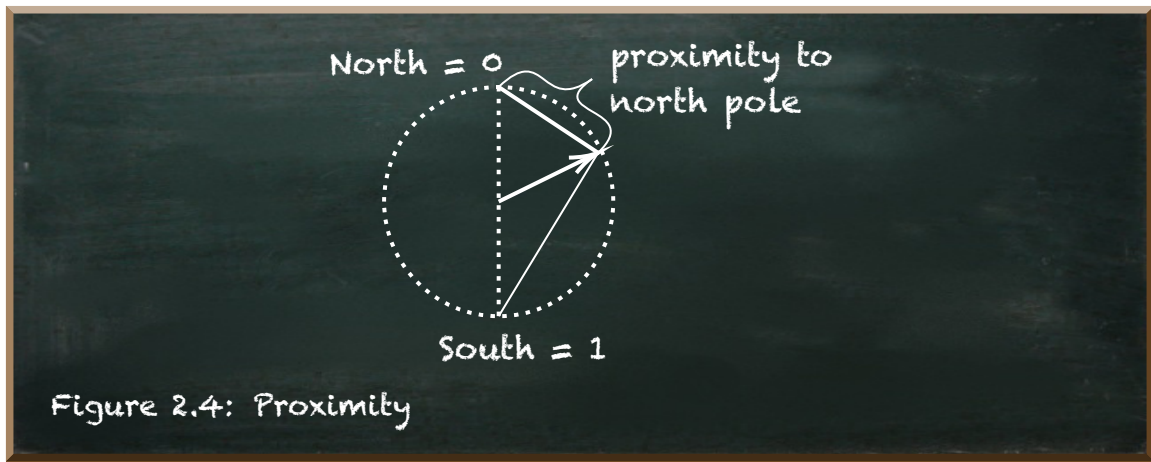


The qubit state is a vector from the center to the circle surface. So, of course, all states share the same origin because there is only one center. And, all state vectors share the same length because a circle has a constant radius. The essential characteristic of a qubit state vector is the point where it touches the surface. This point—more specifically—its distances to the opposing poles determine the probability of measuring the qubit as either 0 or 1.

The farther away this point is from the south pole, the more likely it is to measure the qubit as 0. And, the farther away this point is from the north pole, the more likely it is to measure the qubit as 1. More specifically, the probability of measuring the qubit as 0 is the square of the distance between the point where the qubit state touches the surface and the south pole.



In other words, the closer a point to a pole, the higher the probability of measuring the qubit accordingly as depicted in the following image.



## 2.4 Qubit Operators

Similar to classical computing, we use operators to manipulate the state of the system. Like classical operators work with truth values, quantum operators change the qubit state vector.

As we just learned, all qubit state vectors start in the center of the circle and end at the surface of the circle around the center. So, the only thing single-qubit operators do is to move the quantum state vector around that circle. Multi-qubit operators work a little bit differently. We will cover this in a minute.

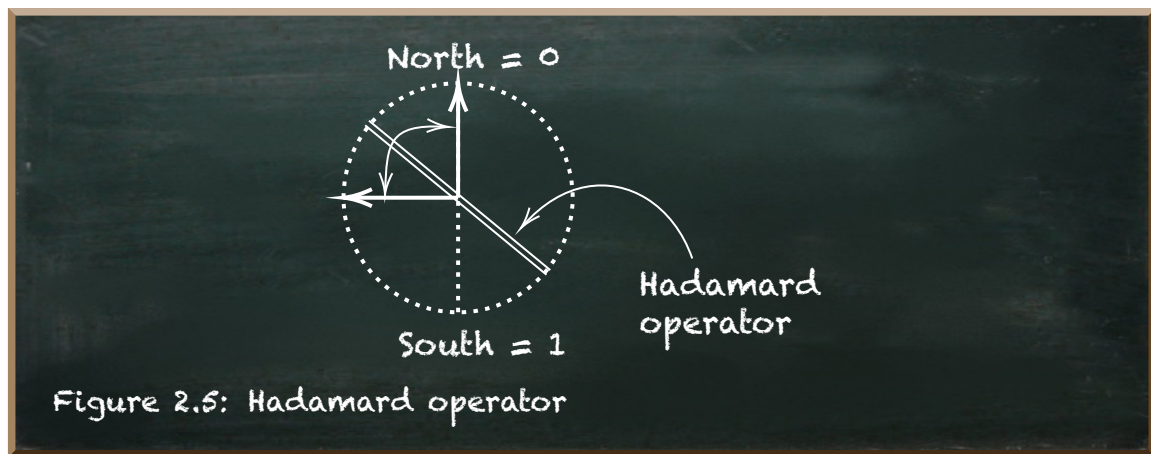
So, there are two ways of moving the qubit state vector.

1. Mirroring
2. Rotating

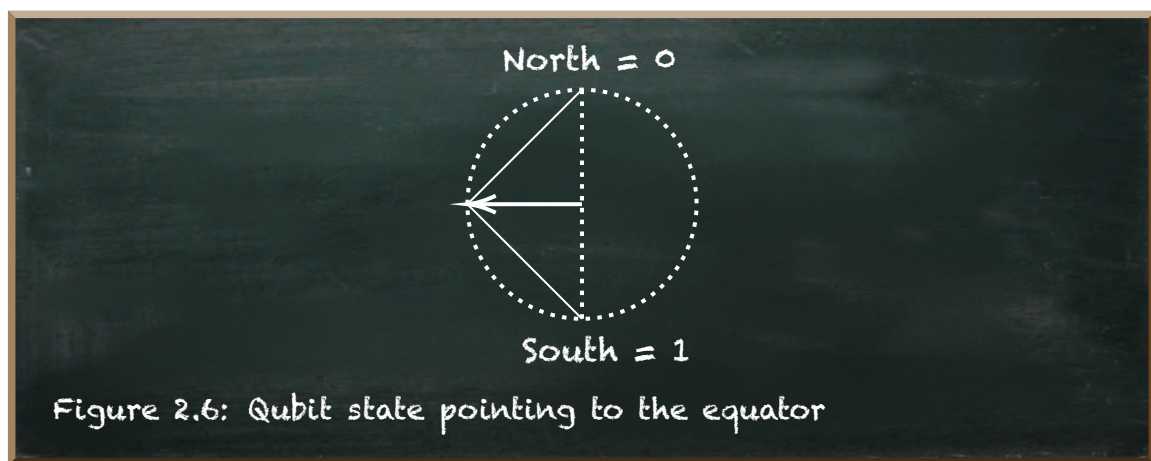
For instance, the Hadamard operator mirrors the qubit state by the diagonal from the top-left to the bottom-right as depicted in the following figure.

So, a qubit whose state vector points to the north pole gets mirrored to point to the equator of the circle.

Since Qiskit initializes qubits in a state that points to the north pole, it explains why we see equal probabilities of measuring the qubit as 0 or 1 in our example we created above. The proximities from the equator to either pole are the same. So are the measurement probabilities.



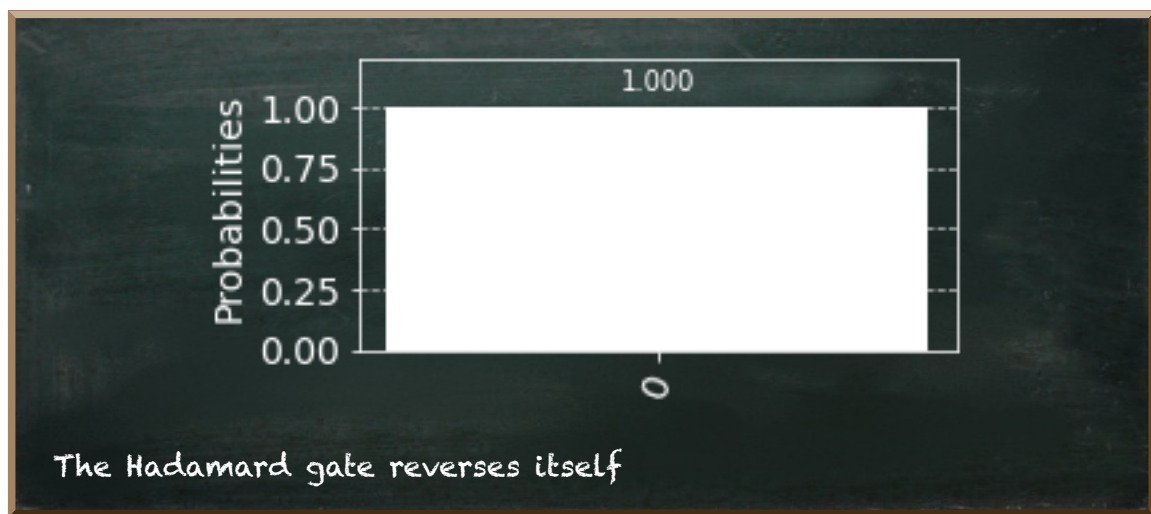
The following visualization explains further characteristics of the Hadamard operator. For instance, if you apply it twice, it reverts itself — no matter its original state. Of course, it does because mirroring twice results in the original.



So, let's create and run a circuit with two Hadamard gates.

Listing 2.3: Applying the Hadamard gate twice

```
1 # Create a quantum circuit with one qubit
2 qc = QuantumCircuit(1)
3
4 # YOUR CODE GOES HERE
5 qc.h(0)
6 qc.h(0)
7
8 # measure the qubit
9 qc.measure_all()
10 run_circuit(qc)
```

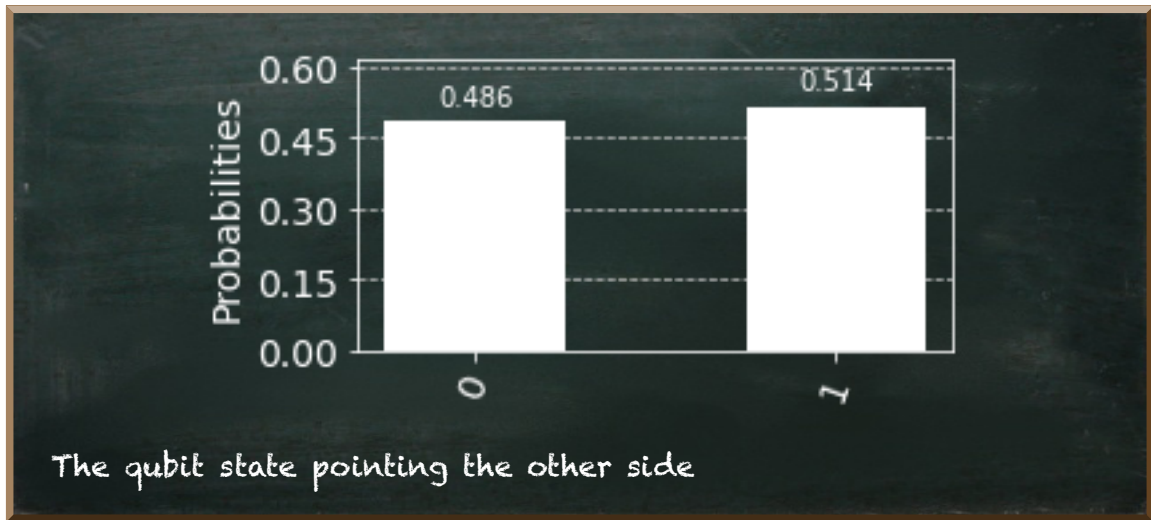


We always measure the qubit as a 0. This is the case when the qubit state vector points to the north pole. In the corresponding basis state  $|0\rangle$  the qubit has a probability of 100% to result in 0.

Moreover, if you apply the Hadamard operator on a qubit that points to the south pole ( $|1\rangle$ ) it also points to the equator but in the opposite direction. But since the proximities to the poles don't care about the direction, there is no difference in the measurement results.

Listing 2.4: Applying the Hadamard gate twice

```
1 # Create a quantum circuit with one qubit
2 qc = QuantumCircuit(1)
3
4 # YOUR CODE GOES HERE
5 qc.x(0)
6 qc.h(0)
7
8 # measure the qubit
9 qc.measure_all()
10 run_circuit(qc)
```

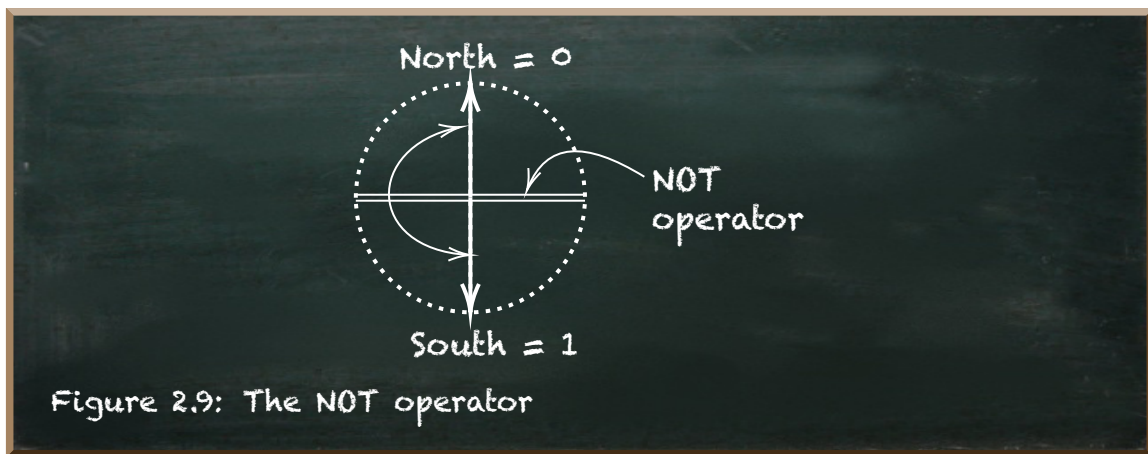


We secretly used another quantum operator here—that is as common as the Hadamard operator. This is the NOT operator that mirrors the quantum state on the X-axis. Therefore, its function name is `x`.

So, of course, the NOT operator reverses itself, too. But if the qubit state points to the equator, the NOT operator has no effect.

Quantum operators are surprisingly intuitive—if you look at them appropriately. Of course, this explanation in plain English lacks the precision and conciseness of a mathematical equation. Yet, I am truly convinced that this explanation is more helpful to get started with quantum computing than all equations bundled together.

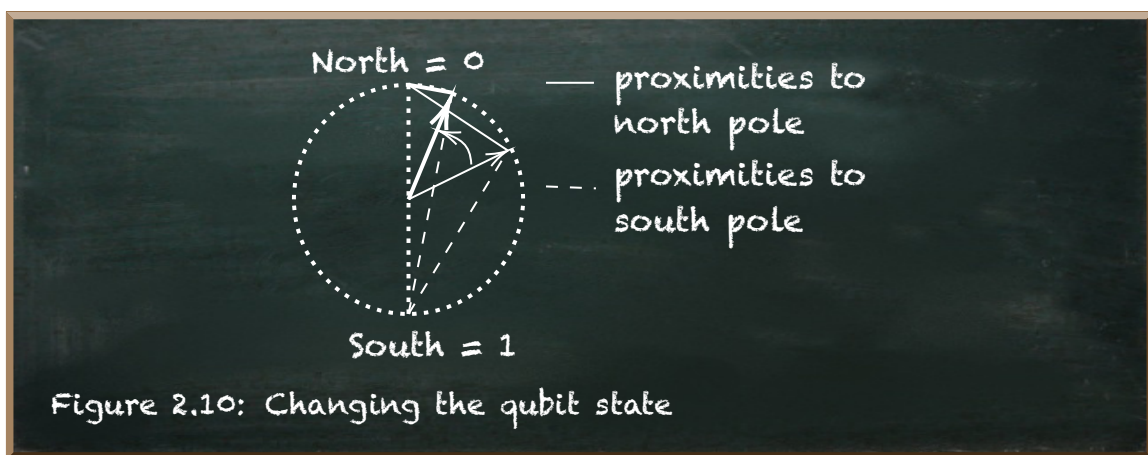
Besides mirroring, we can also rotate the qubit state vector. Given an arbitrary quantum state vector that we define as  $|\psi\rangle$  (“psi”), when we rotate it by an angle  $\theta$  (“theta”), the result is another quantum state vector.



It is worth mentioning that  $\theta$  is just an arbitrary name for the angle, and  $\psi$  is the arbitrary name of an arbitrary qubit state. Since qubit states are vectors, we use the Dirac notation, so it is  $|\psi\rangle$ .

The important thing to note is that  $\theta$  controls the proximities of the vector head to poles of the circle. And these proximities represent the probability amplitudes (that we call  $\alpha$  “alpha” and  $\beta$  “beta”) whose squares are the probabilities of measuring 0 or 1 respectively. So  $\alpha^2$  denotes the probability of measuring  $|\psi\rangle$  as 0.  $\beta^2$  denotes the probability of measuring it as 1.

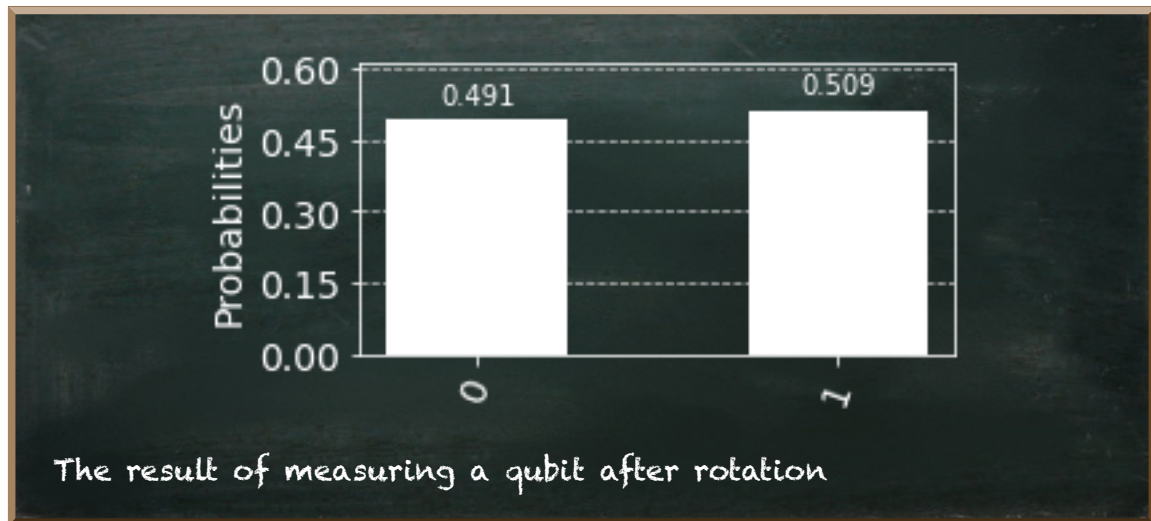
So while the proximity to one pole increases as we rotate the qubit state, the proximity to the other pole decreases correspondingly.



Let's put this into code.

Listing 2.5: Applying the Hadamard gate twice

```
1 from math import pi
2
3 # Create a quantum circuit with one qubit
4 qc = QuantumCircuit(1)
5
6 # rotate the qubit state vector
7 qc.ry(pi/2, 0)
8
9 # measure the qubit
10 qc.measure_all()
11 run_circuit(qc)
```



In Qiskit, we use the `ry` function. It takes two arguments, the angle `theta` and the position of the qubit we apply the operator on.

In our case, we use  $\pi/2$  as `theta`'s value. If you recall a little basic math, you'll remember that the circumference of a circle in radians is  $2\pi$ . So,  $\frac{\pi}{2}$  is a quarter rotation. Since Qiskit initializes qubits in the state pointing to the north pole, a quarter rotation results in a state pointing to the equator. A qubit in this state has equal probabilities to result in 0 and 1.

Of course, you may not always want work with an angle in radians. So, suppose you want to put the qubit into a state where it exhibits a specific probability to result in 1.

The following function `prob_to_angle` implements a function in Python that



takes a probability to measure the qubit as a 1 and returns the corresponding angle theta.

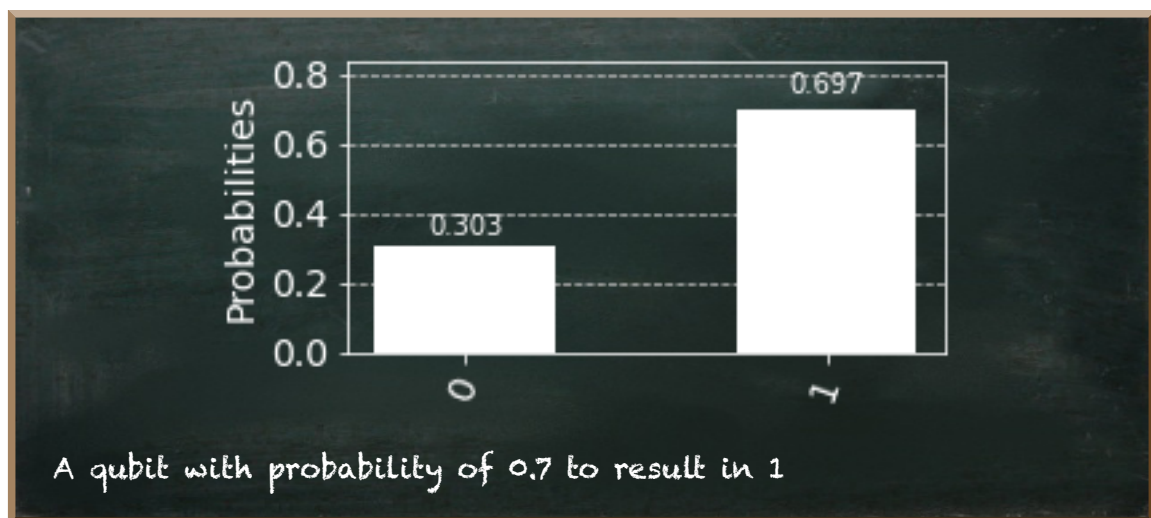
Listing 2.6

```
1 from math import asin, sqrt
2
3 def prob_to_angle(theta):
4     return 2*asin(sqrt(theta))
```

Let's use this function to set the probability of measuring our qubit as a 1 to 70%.

Listing 2.7: Applying the Hadamard gate twice

```
1 from math import pi
2
3 # Create a quantum circuit with one qubit
4 qc = QuantumCircuit(1)
5
6 # rotate the qubit state vector to represent 0.7
7 qc.ry(prob_to_angle(0.7), 0)
8
9 # measure the qubit
10 qc.measure_all()
11 run_circuit(qc)
```



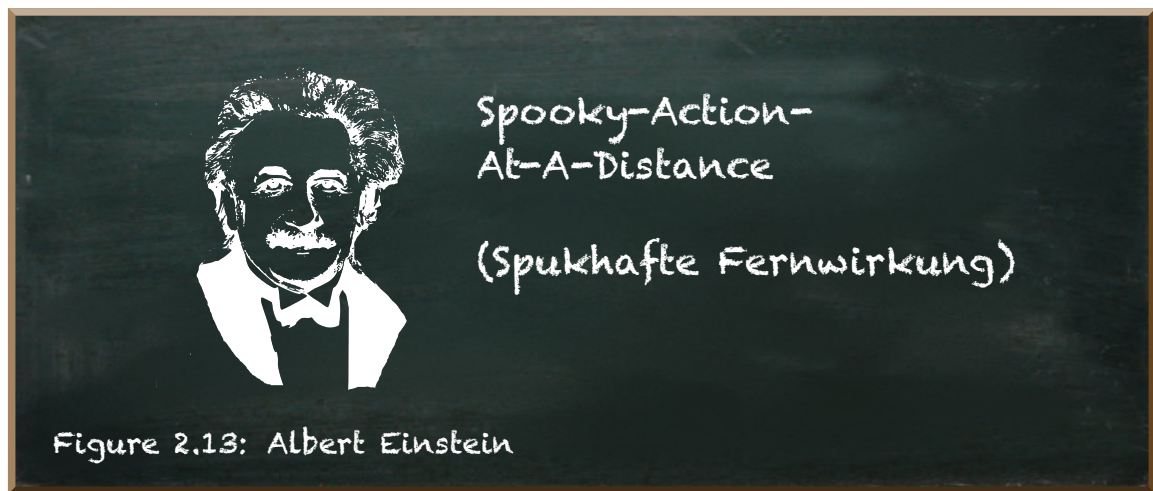
With the  $R_Y$  operator and the `prob_to_angle` function, you can effectively create a probabilistic model using qubits.

It is the building block to model your assumptions about the world in a quantum circuit and, therefore, to conduct analyses and infer relationships between variables. The paramount takeaway is that quantum bits (qubits) are inherently probabilistic. You measure them as either 0 or 1. Which of both you measure depends on the invisible qubit state and chance.

This chance is a helpful tool when designing algorithms. For instance, when there are many equally good options to choose from, it is much easier to have the algorithm flip a coin rather than describe an appropriate deterministic rule.

Moreover, randomization is the foundation of many experiments. It helps mitigate biases (such as selection bias) related to significant known and unknown confounders and contributes to the validity of statistical analyses.

Once you get a little experience, working with single-qubit operators becomes pretty intuitive. The real challenge to our minds is multi-qubit operators. This is because a multi-qubit operator entangles two qubits.



Quantum entanglement is an incredible phenomenon. Two particles – no matter how far apart – share a common state. As soon as you look at one of the particles, the other one instantly jumps to another state, which corresponds to what you measured at the first qubit.

In a letter to Max Born in 1947, Albert Einstein formulated this phenomenon vividly as “spukhafte Fernwirkung” (spooky action at a distance).

If I were a physicist, I’d be concerned. Quantum mechanics is really hard to understand. In the words of Richard Feynman – one of the fathers of

Figure 2.14: If you think you understand quantum mechanics, you don't understand quantum mechanics

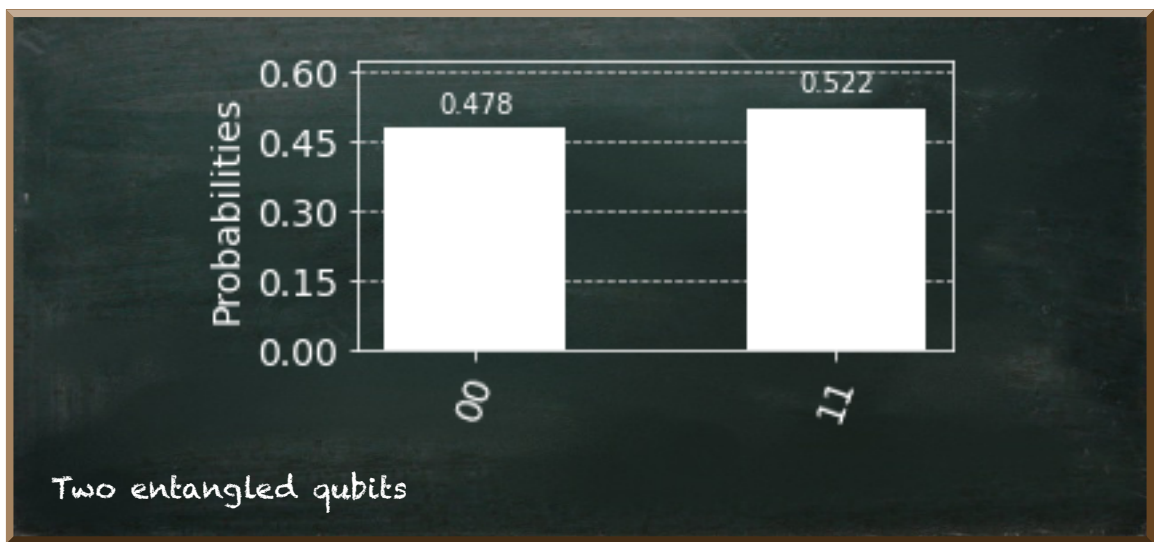
The basic operator we use to entangle two qubits is the controlled-NOT operator (alternatively CNOT,  $c_x$ ) that takes two qubits, a control qubit and a target qubit.

The controlled-NOT operator applies the NOT ( $\times$ ) operator on the target qubit only if the control qubit is in state  $|1\rangle$ .

The following code listing shows this operator in practice.

Listing 2.8: Applying the Hadamard gate twice

```
1 # Create a quantum circuit with two qubits
2 qc = QuantumCircuit(2)
3
4 # apply the Hadamard gate on qubit 0
5 qc.h(0)
6
7 # apply the CNOT gate with qubit 0 as the control and qubit 1 as the
  target qubit
8 qc.cx(0,1)
9
10 # measure the qubit
11 qc.measure_all()
12 run_circuit(qc)
```



Let's first clarify what we see in this histogram. Each bar indicates the probability of measuring the two-qubit system in the state denoted at the bottom axis. We see two possible measurements: 00 and 11. These numbers read from right to left. So, the digit at the right represents the measure of the qubit at position 0 and the left digit represents the qubit at position 1. In this case, it doesn't yet matter because both digits equal.

We can easily reason about the effect of the CNOT operator. We can even visualize it in a truth table as depicted below.

A	B	A	$A \oplus B$
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Figure 2.16: Truth table of the CNOT gate

In our example, the control qubit (at position 0) is in a state where the outcomes 0 and 1 are equally likely. As mentioned before, the qubit is in a complex linear combination of its basis states  $|0\rangle$  and  $|1\rangle$ . A prominent (yet not very accurate) interpretation of this is that the qubit is in both states concurrently.

In that notion, by applying the CNOT operator, we apply the NOT operator only on that part of the qubit where the control qubit is in state  $|1\rangle$ . Since the target qubit is initialized in state  $|0\rangle$ , we flip it to  $|1\rangle$  resulting in the combined state  $|11\rangle$ .

By contrast, we do nothing if the target qubit is in state  $|0\rangle$ . So, the target qubit remains in its initial state  $|0\rangle$ , resulting in the combined state  $|00\rangle$ .

As a result,  $|00\rangle$  and  $|11\rangle$  are the only states we observe the two qubits in.

From a computational basis, the CNOT operator (and all other entangling gates) let us apply other quantum operators on a subset of all possible states, to wit the states where the control qubit is in state  $|1\rangle$ .

If you are familiar with `if then else` constructs, you won't find it too hard to understand the CNOT operator and practical quantum entanglement.



## 3. Variables

The single most important concept in statistical modeling you'll need to understand is variables, events, and their probabilities.

A variable is any characteristics, number, or quantity that can be measured or counted. So, it can obtain different values. The simplest of all is a Boolean variable that can only have two values, true or false. An event, in this context, is the variable assuming one of the possible values. So, for instance, our variable has the value true. Finally, each event occurs with a probability.

And, there are different types of probabilities.

- The Marginal Probability is the absolute probability of an event.
- The Joint Probability is the probability of two events occurring together.
- The Conditional Probability is the probability of one event given the knowledge that another event occurred.

### 3.1 Marginal Probability

We start with letting a qubit represent the marginal probability of one event. A marginal probability is the absolute probability of the event irrespective of any further information. If we have multiple states where the event occurs, then the marginal probability is the sum of all the corresponding probabilities.



Suppose a qubit represents a variable that can have only one of two values. It can be 0 or 1. Each value occurs with a certain probability.

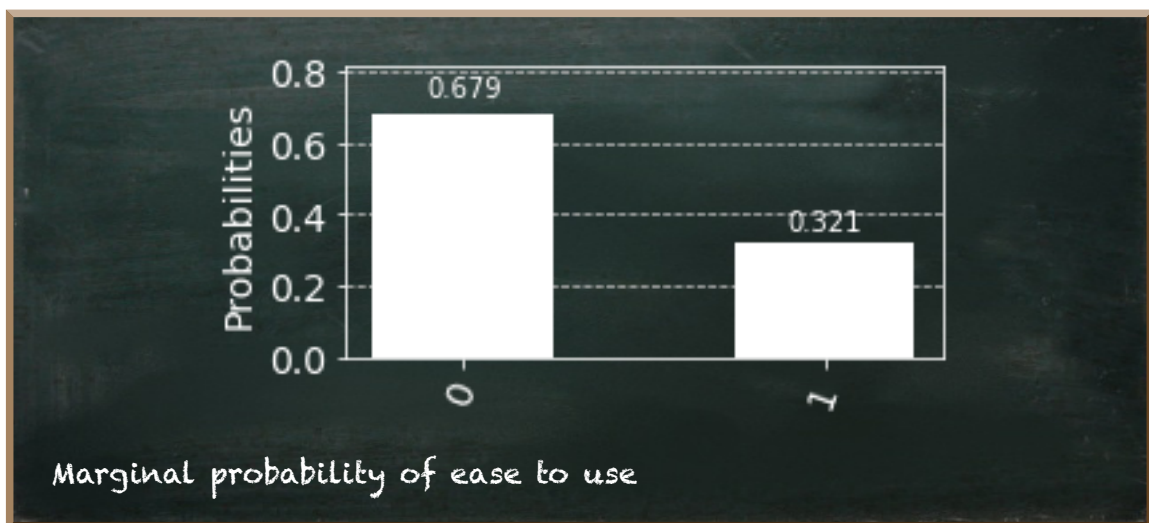
$p$  denotes the probability of the variable being 1. Since there are only two possible values, we know that whenever the value is not 1 it must be 0 instead. And since all probabilities must add up to 1 (= 100%), we know that the probability of 0 is  $1 - p$ .

Recall our above example of a Bayesian network (figure 1.6). Suppose the ease-of-use of a particular system is 0.3. That means that three out of ten persons find the system easy to use.

Now, we say that the qubit at position 0 represents this variable and the value of 1 corresponds to the system being easy to use.

Listing 3.1: Representing the marginal probability

```
1 # Specify the marginal probability of the event
2 # easy_to_use
3 easy_to_use = 0.3
4
5 qc = QuantumCircuit(1)
6
7 # Apply the marginal probability
8 qc.ry(prob_to_angle(easy_to_use), 0)
9
10 qc.measure_all()
11
12 run_circuit(qc)
```



Except for a small difference due to the empirical nature of our simulation, we measure our variable as 1 with a probability of 30%.

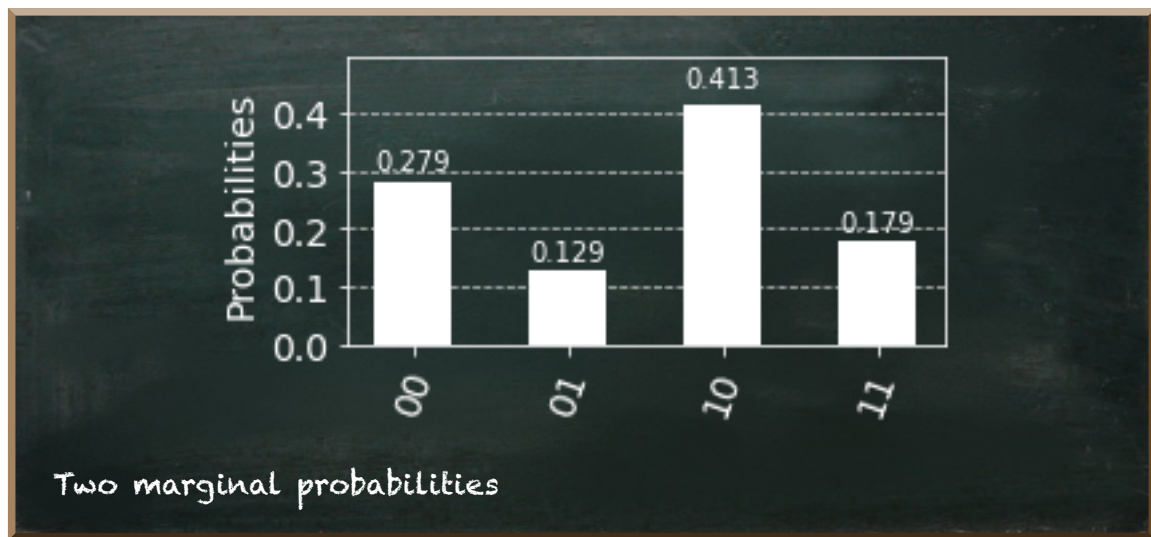
We used the `prob_to_angle` function in combination with the  $R_Y$  operator. This is an effective way to model marginal probabilities. In Bayesian networks, we need to specify the marginal probability for those variables that do not depend on any other variable.

In our Bayesian network, there is another variable that does not depend on any other. This is the *perceived usefulness*. So, let's say 60% of the users think that this particular software is useful.

So, we extend our model with another qubit (position 1) to represent *perceived usefulness*.

Listing 3.2: Representing the marginal probability

```
1 # Specify the marginal probabilities
2 easy_to_use = 0.3
3 usefulness = 0.6
4
5 qc = QuantumCircuit(2)
6
7 # Apply the marginal probabilities
8 qc.ry(prob_to_angle(easy_to_use), 0)
9 qc.ry(prob_to_angle(usefulness), 1)
10
11 qc.measure_all()
12
13 run_circuit(qc)
```

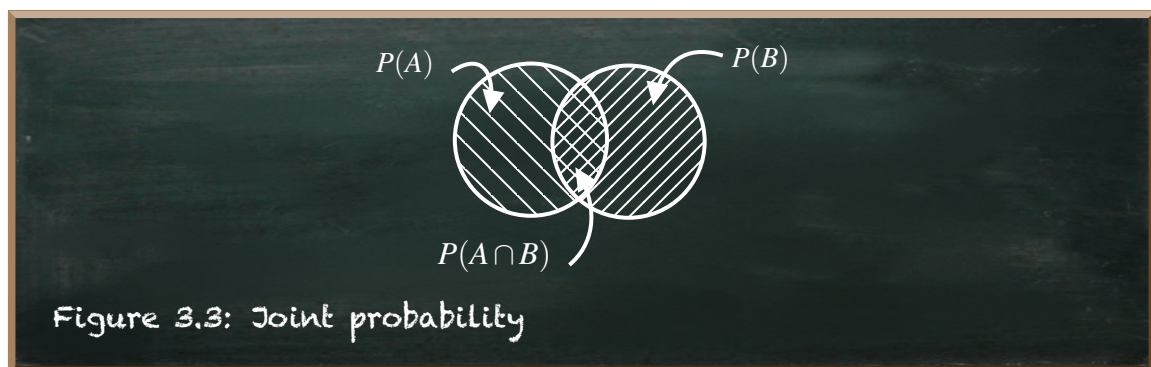


If you only look at the qubit at position 1 (the left digit in the state string), you see that it is 1 with a probability of  $0.413 + 0.179 = 0.602$ .

This illustrates the meaning of marginal probabilities. We only look at the value of one distinct variable irrespective of any other variables and their values.

## 3.2 Joint probability

But, of course, we can also look at multiple variables concurrently. For instance, we could ask for the probability that a user finds the system easy to use and useful. The answer is given by the joint probability that as depicted in the following figure.



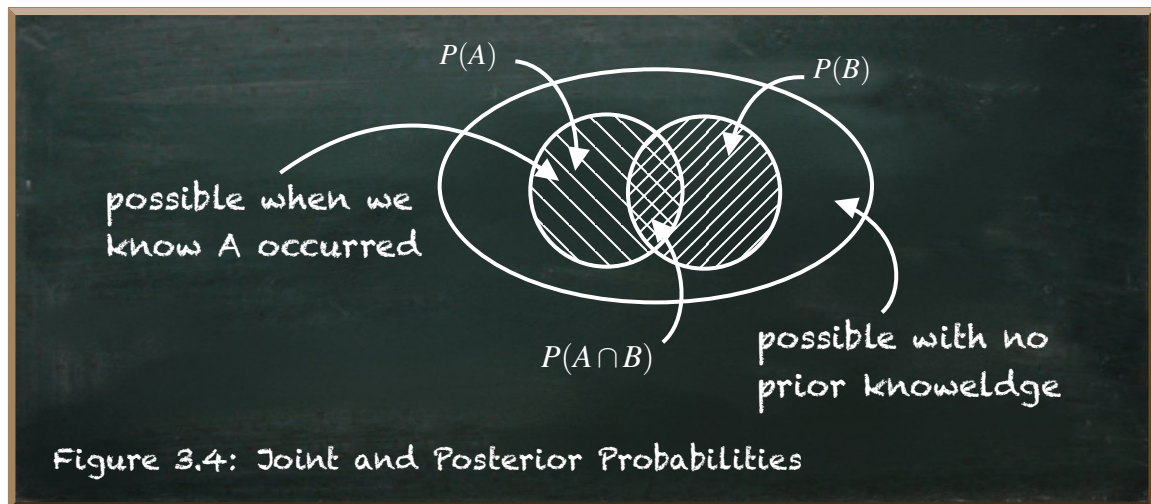
Mathematically, we multiply the probability of both variables.

In the above example, this is 0.179. So, it is very close to the mathematical value of  $0.3 \times 0.6 = 0.18$ .

In a quantum circuit, we get the joint probability if we measure both qubits and count the portion where both qubits are 1.

### 3.3 Conditional Probability

Graphically, the posterior probability is almost the same as the joint probability. In fact, the area representing the positive cases is the same. It is the overlap of event A and event B. But the base set is different. While we consider all possible cases when calculating the joint probability, we only consider the cases where one event occurred when calculating the posterior probability.



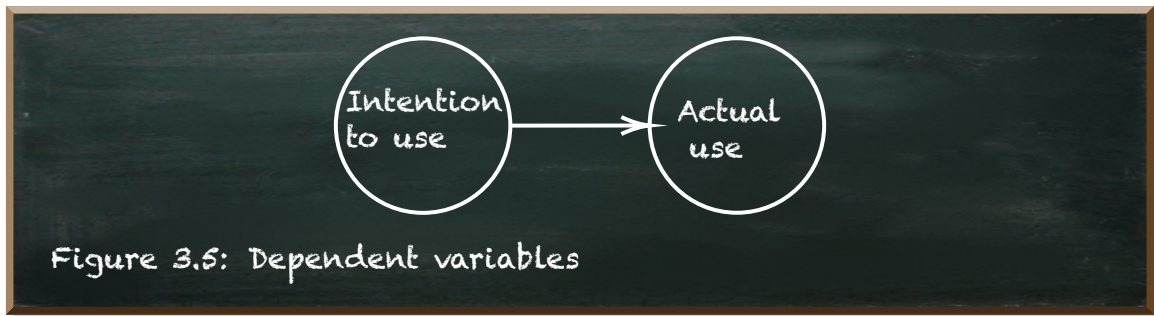
So, a conditional probability is a probability of an event (our hypothesis) given the knowledge that another event occurred (our evidence). To model such a relationship, we make use of quantum entanglement. Do you remember? Entangling operators apply an operator on the target qubit only if the control qubit is in state  $|1\rangle$ . In the previous chapter, we got to learn the CNOT operator that applies a NOT gate on the target qubit.

But you can apply any operator, if you want to. So, for instance, the controlled  $R_Y$  operator (also  $CR_Y$ ,  $cry$ ) applies the  $R_Y$  operator on the target qubit.

Let's look at the relation between the intention to use and the actual use.

If we only regard these two variables, then the intention to use is an independent variable. So, we specify a marginal probability for it, say 0.4.

Now, there are two different events:



1. The user does not intend to use the system (represent by  $|0\rangle$ )
2. The user does intend to use the system (represent by  $|1\rangle$ )

Recall the CPT we used before.

$P(\text{ActualUse}|\text{IntendedUse})$

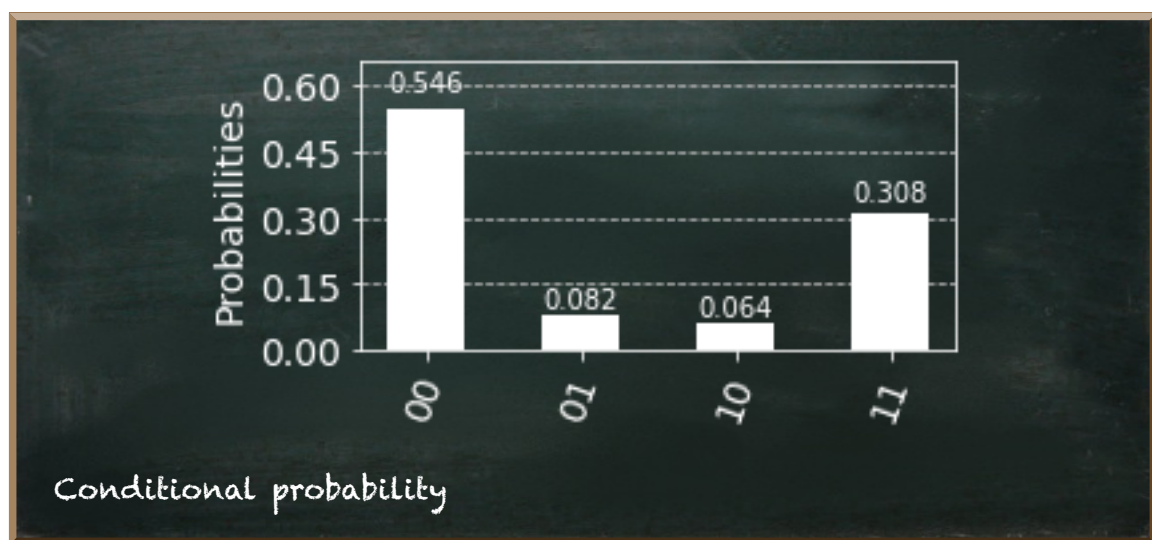
	intention	no intention
Use	0.8	0.1
NoUse	0.2	0.9

Figure 3.6: Conditional Probability Table

The following code listing implements this relation in a quantum circuit.

Listing 3.3: Representing the marginal probability

```
1 # Specify the marginal probability
2 intention = 0.4
3
4 # Specify the conditional probabilities
5 use_intention = 0.8
6 use_nointention = 0.1
7
8 qc = QuantumCircuit(2)
9
10 # Apply the marginal probability
11 qc.ry(prob_to_angle(intention), 0)
12
13 # apply the conditional probability
14 # when the user intends to use the system
15 qc.cry(prob_to_angle(use_intention), 0, 1)
16
17 # apply the conditional probability
18 # when the user does not intend to use the system
19 qc.x(0)
20 qc.cry(prob_to_angle(use_nointention), 0, 1)
21 qc.x(0)
22
23 qc.measure_all()
24
25 counts = run_circuit(qc, hist=False)
26 plot_histogram(counts)
```





First, we specify the marginal and the conditional probabilities (lines 1-8). Then, we initialize the quantum circuit with two qubits (one qubit for each variable) (line 8), and apply the marginal probability representing the intention to use the system (line 11).

The interesting stuff happens afterward.

Next, we apply the conditional probability on the actual use if the user intends to use the system (line 15). The underlying rationale is that the control qubit (position 0) is in state  $|1\rangle$  if the user intends to use the system. So, the controlled  $R_Y$  operator applies the probability on the target qubit only in this case, but doesn't do anything if the control qubit is in state  $|0\rangle$ .

Finally, we apply the conditional probability on the actual use if the user does not intend to use the system (lines 19-21). We also use the controlled  $R_Y$  operator. But, this time, we encapsulated it into NOT operators that we apply on the control qubit. The first not operator flips the states  $|0\rangle$  and  $|1\rangle$  of the control qubit. Therefore, we can say that the state  $|1\rangle$  now represents the case that the user does not intend to use the system. The following controlled  $R_Y$  operator therefore applies the rotation on the target qubit only in this case. The second NOT operator flips the control qubit back so that  $|1\rangle$  represents the intention to use the system and  $|0\rangle$  the opposite.

Listing 3.4: The resulting probabilities

```
1 print("P(Intention): {}".format(
2     round((counts['01']+counts['11'])/1000, 2)
3 ))
4
5 print("P(Use|No Intention): {}".format(
6     round(counts['10']/(counts['10']+counts['00']), 2)
7 ))
8
9 print("P(Use|Intention): {}".format(
10     round(counts['11']/(counts['11']+counts['01']), 2)
11 ))
12
13 print("P(Use): {}".format(
14     round((counts['10']+counts['11'])/1000, 2)
15 ))
```

$P(\text{Intention}): 0.39$ $P(\text{Use} \text{No Intention}): 0.1$ $P(\text{Use} \text{Intention}): 0.79$ $P(\text{Use}): 0.37$
--

When we look at the results, we first check whether the marginal probability of the control qubit (the intention to use the system) matches the expected value.

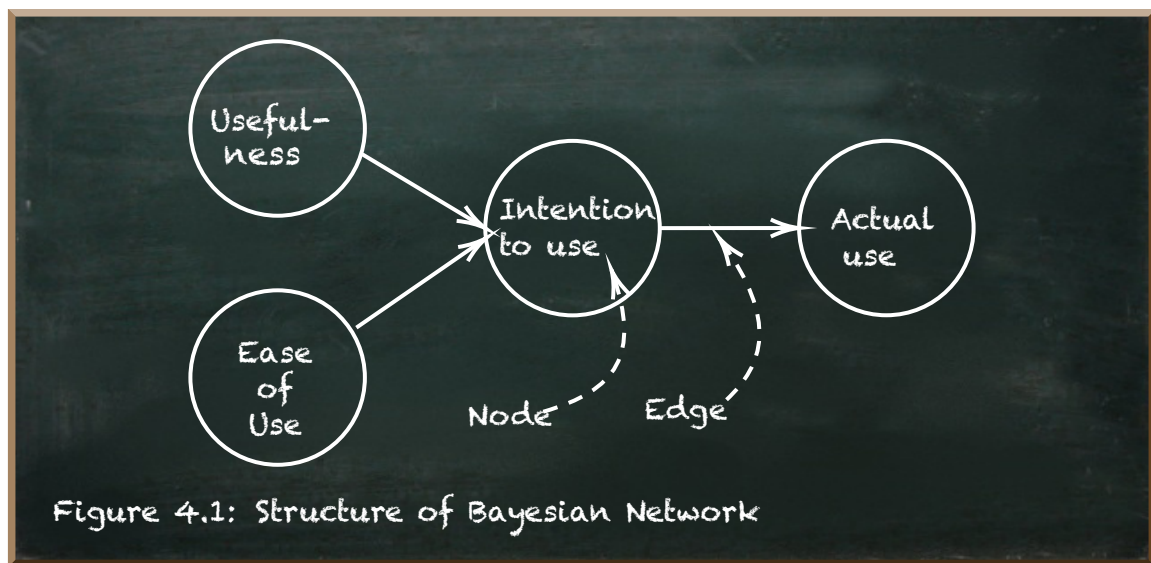
We see that the sum of the 01 and 11 measurements is around 0.4 - the specified value. Further, we can check that the conditional probabilities also match.

The result 10 is about 10% of the sum of the results 00 and 10. That is the use of the system even though the user does not intend to use it. Accordingly, the result 11 is about 80% of the sum of the results 01 and 11. These are the results that represent users with the intention to use the system.

Finally, we can easily calculate the marginal probability of actual use. It is around 0.36.

## 4. Complete QBN

We're now ready to create the entire quantum Bayesian network as depicted in the following figure.



Our QBN consists of four variables. Therefore, we use four qubits. In the following code listing, we use two additional classes, the `QuantumRegister` to hold our qubits and a `ClassicalRegister` to receive the measurements. Technically, there is no difference to the way we specified the `QuantumCircuit` before, but the registers allow us to access the qubits more easily. Something we will need to do in a minute.

Listing 4.1: Specify the quantum circuit

```

1 from qiskit import QuantumRegister, ClassicalRegister
2 qr = QuantumRegister(4)
3 cr = ClassicalRegister(4)
4 qc = QuantumCircuit(qr, cr)

```

There are two independent variables, ease of use and perceived usefulness. We model these using their marginal probabilities.

Listing 4.2: Specify the marginal probabilities

```

1 # Specify the marginal probabilities
2 easy_to_use = 0.3
3 usefulness = 0.6
4
5 # Apply the marginal probabilities
6 qc.ry(prob_to_angle(easy_to_use), 0)
7 qc.ry(prob_to_angle(usefulness), 1)

```

In the next step, we need to specify the intention to use. In contrast to our previous example, this is not an independent but a dependent variable. It depends on the two variables ease of use and perceived usefulness. The following CPT depicts the corresponding probabilities.

$P(\text{ActualUse}|\text{IntendedUse})$

	Easy to use	Not easy to use
Useful	0.9	0.7
NotUseful	0.2	0.1

Figure 4.2: CPT of intention to use

Let's now encode these conditional probabilities in our quantum circuit.

Listing 4.3: The conditional probabilities of intention to use

```

1 # Specify the conditional probabilities
2 intention_easy_useful = 0.9
3 intention_noeasy_useful = 0.7
4 intention_easy_nouseful = 0.2
5 intention_noeasy_nouseful = 0.1
6
7 # easy to use and useful
8 qc.mcry(prob_to_angle(intention_easy_useful), [qr[0], qr[1]], qr[2])
9
10 # not easy to use but useful
11 qc.x(0)
12 qc.mcry(prob_to_angle(intention_noeasy_useful), [qr[0], qr[1]], qr[2])
13 qc.x(0)
14
15 # easy to use but not useful
16 qc.x(1)
17 qc.mcry(prob_to_angle(intention_easy_nouseful), [qr[0], qr[1]], qr[2])
18 qc.x(1)
19
20 # not easy to use and not useful
21 qc.x(0)
22 qc.x(1)
23 qc.mcry(prob_to_angle(intention_noeasy_nouseful), [qr[0], qr[1]], qr[2])
24 qc.x(0)
25 qc.x(1)

```

There are four different situations, each associated with a conditional probability. In the first, both independent variables (ease of use and usefulness) are  $|1\rangle$ . Therefore, we do not need to use any NOT operator. But, we use a new operator, the multi-controlled  $R_Y$  operator (`mcry`). It works just like the controlled  $R_Y$  operator (`cry`) with the only difference that it takes multiple control qubits. So, it applies the  $R_Y$  rotation only if all control qubits are in state  $|1\rangle$ .

The first argument to this operator is the rotation angle `theta`. The second is a list of the control qubits. This time, however, we must provide the actual qubits instead of their positions as the argument. This is the reason why we put the qubits into a `QuantumRegister`. This structure allows us to access the qubits. The third parameter is the target qubit.

In the next block, we apply the conditional probability associated with a system that is not easy to use but useful. Accordingly, we flip the qubit that rep-

resents ease of use from  $|0\rangle$  to  $|1\rangle$  before we apply the `mcry` operator. Afterward, we undo this flip by applying the NOT operator again.

We do the same thing for the situation when the system is easy to use but not useful by encapsulating the `mcry` operator into NOT operators we apply on the qubit at position 1.

Finally, we apply NOT gates on both control qubits to model the conditional probability of the case when the system is neither easy to use nor useful.

In the next code listing, we model the conditional probability between the intention to use the system and the actual use. This is pretty much the same code we used before.

Listing 4.4: The conditional probability on the actual use

```

1 # Specify the conditional probabilities
2 use_intention = 0.8
3 use_nointention = 0.1
4
5 # apply the conditional probability
6 # when the user intends to use the system
7 qc.cry(prob_to_angle(use_intention), 2, 3)
8
9 # apply the conditional probability
10 # when the user does not intend to use the system
11 qc.x(2)
12 qc.cry(prob_to_angle(use_nointention), 2, 3)
13 qc.x(2)

```

In the last step, as depicted in the following listing, we measure our qubits. This involves the `ClassicalRegister`

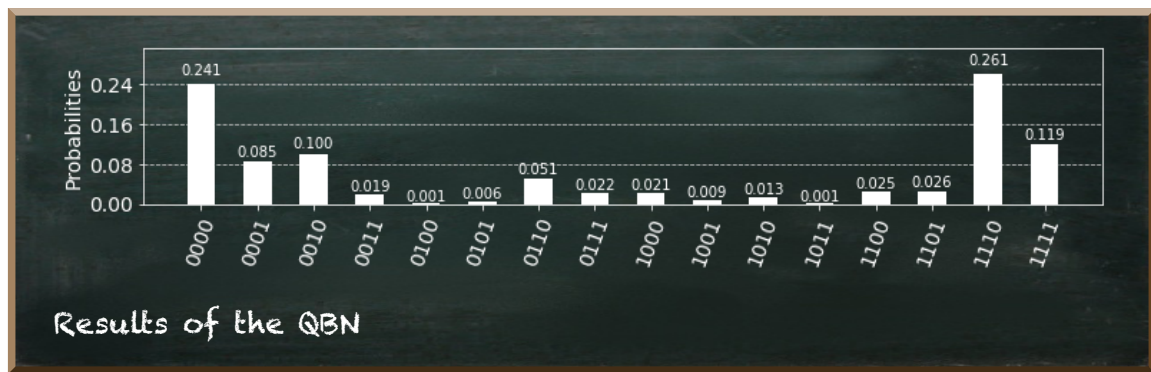
Listing 4.5: Measuring and running the circuit

```

1 qc.measure(qr, cr)
2
3 counts = run_circuit(qc, hist=False)
4 plot_histogram(counts)

```





The resulting histogram contains 16 different results. These are all the possible combinations. Each qubit represents the state of one variable. So, for instance, the qubit at the left represents the actual use.

By summing up the respective counts, we can obtain the marginal probability of the system being used.

Listing 4.6: The marginal probability of the system being used

```

1 print("P(Use): {}".format(
2     round(sum(
3         counts[key] for key in filter(
4             lambda cnt: cnt[0] == '1',
5             counts.keys()
6         )
7     )/1000, 2)
8 ))

```

P(Use): 0.47

If you were only interested in the marginal probability of the actual use, you could as well only measure this qubit you're interested in as depicted in the following listing.

Listing 4.7: The complete code (part 1)

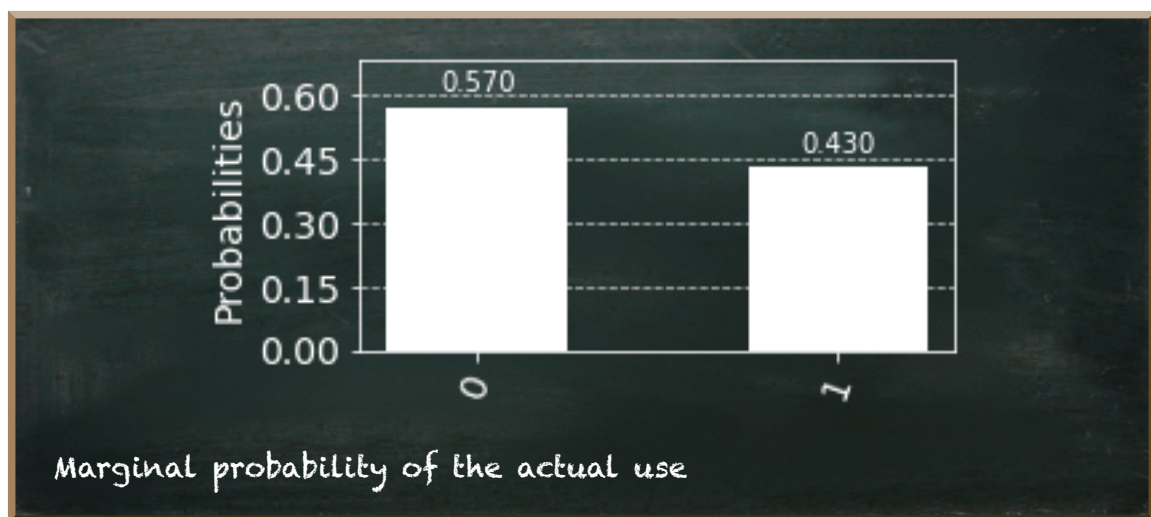
```
1 qr = QuantumRegister(4)
2 cr = ClassicalRegister(1)
3 qc = QuantumCircuit(qr, cr)
4
5 # Specify the marginal probabilities
6 easy_to_use = 0.3
7 usefulness = 0.6
8
9 # Apply the marginal probabilities
10 qc.ry(prob_to_angle(easy_to_use), 0)
11 qc.ry(prob_to_angle(usefulness), 1)
12
13
14 # Specify the conditional probabilities
15 intention_easy_useful = 0.9
16 intention_noeasy_useful = 0.7
17 intention_easy_nouseful = 0.2
18 intention_noeasy_nouseful = 0.1
19
20 # easy to use and useful
21 qc.mcry(prob_to_angle(intention_easy_useful), [qr[0], qr[1]], qr[2])
22
23 # not easy to use but useful
24 qc.x(0)
25 qc.mcry(prob_to_angle(intention_noeasy_useful), [qr[0], qr[1]], qr[2])
26 qc.x(0)
27
28 # easy to use but not useful
29 qc.x(1)
30 qc.mcry(prob_to_angle(intention_easy_nouseful), [qr[0], qr[1]], qr[2])
31 qc.x(1)
32
33 # not easy to use and not useful
34 qc.x(0)
35 qc.x(1)
36 qc.mcry(prob_to_angle(intention_noeasy_nouseful), [qr[0], qr[1]], qr[2])
37 qc.x(0)
38 qc.x(1)
```

Listing 4.8: The complete code (part 2)

```

1 # Specify the conditional probabilities
2 use_intention = 0.8
3 use_nointention = 0.1
4
5 # apply the conditional probability
6 # when the user intends to use the system
7 qc.cry(prob_to_angle(use_intention), 2, 3)
8
9 # apply the conditional probability
10 # when the user does not intend to use the system
11 qc.x(2)
12 qc.cry(prob_to_angle(use_nointention), 2, 3)
13 qc.x(2)
14
15
16 # only measure the qubit representing the actual use
17 qc.measure(qr[3], cr[0])
18
19 counts = run_circuit(qc, hist=False)
20 plot_histogram(counts)

```



By ignoring the values of the other qubits, the resulting histogram summarizes the marginal probability of the actual use of the system.

## 5. Conclusion

Quantum computing builds upon the strange phenomena of the subatomic only a sublime group of physicists understand. But how could someone use a technology he doesn't understand?

I don't know about you, but I do this every day. I don't know how a microwave oven works.

"That's too bold! This is a consumer device," you say?

Fair enough. How about a classical computer?



I program classical computers for a living. I develop code to solve practical problems. So, I'd say I use computers beyond the everyday Cletus. But, do be honest: I don't know how they work physically.

Quantum Bayesian networks are a simple yet powerful machine learning tool. And, they are handy to get started with quantum machine learning because they share the probabilistic perspective that is paramount in quantum computing.

We experienced how a Bernoulli distribution represents a qubit in superposition. And, we saw that we could use entanglement to model the dependencies between variables in the Bayesian network.

Do you want to learn more about quantum machine learning and quantum Bayesian networks? In the first volume of my book [Hands-On Quantum Machine Learning With Python](https://www.pyqml.com) (<https://www.pyqml.com>), we build up a QBN from scratch, train it to account for missing values, and use it for inference.

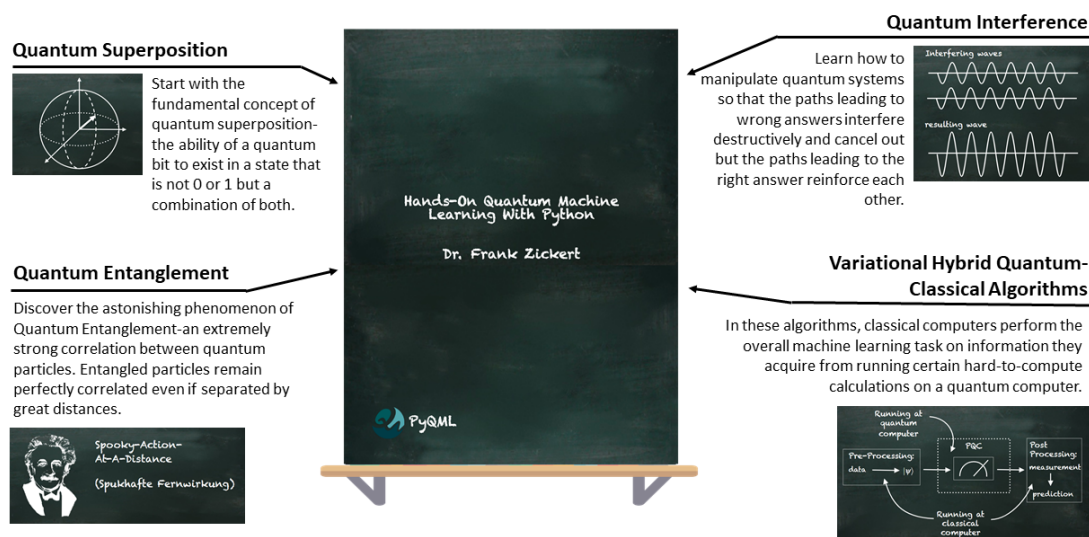


Figure 5.2: Volume 1: Getting Started

This brief tutorial shows that you don't need to be a mathematician or physicist to become an expert in quantum computing. Quantum Bayesian networks are just the very beginning of your quantum computing journey. There are many more interesting and powerful algorithms out there, such as the Variational Quantum Eigensolver and the Quantum Approximate Optimization Algorithm. This is what I cover in the second volume of [Hands-On](https://www.pyqml.com)

## Quantum Machine Learning With Python (<https://www.pyqml.com>).

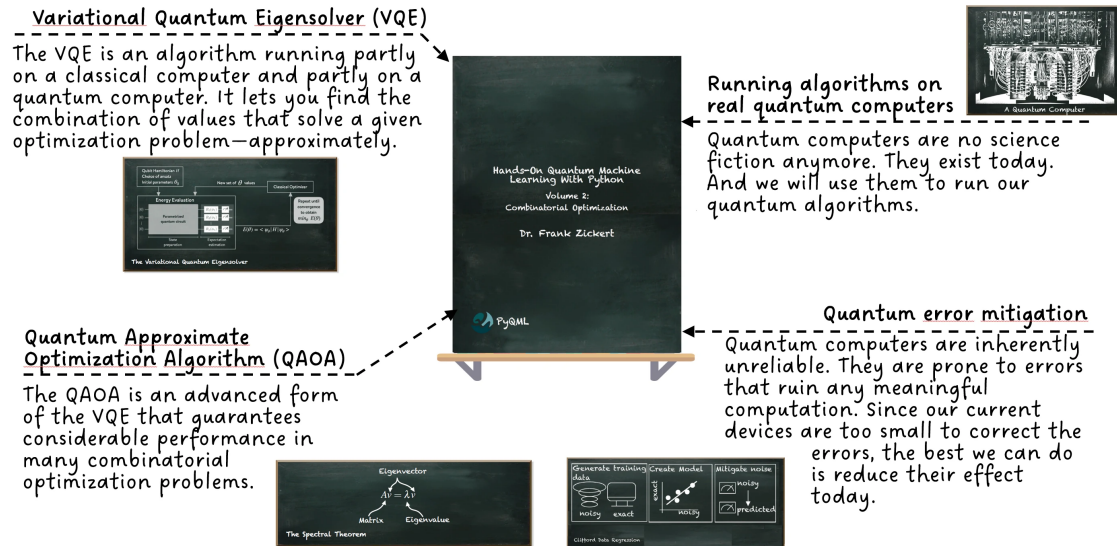


Figure 5.3: Volume 2: Combinatorial Optimization

I hope this tutorial raised your interest in this new kind of computation that is eventually revolutionizing the way we solve complex problems.



# Getting Started With Quantum Bayesian Networks

Dr. Frank Zickert

You're interested in quantum computing and machine learning... But you don't know how to get started? Let me help.



If you can't explain it simply, you don't understand it well enough.

Albert Einstein.

Whether you just get started with quantum computing and machine learning or you're already a senior machine learning engineer, this book is your comprehensive guide to get started with Quantum Bayesian Networks. Quantum Bayesian networks are the best way to start your quantum machine learning journey because they share the probabilistic perspective that is paramount in quantum computing.

This book offers a practical, hands-on exploration of Quantum Bayesian Networks.

