Hands-On Quantum Machine Learning With Python

Volume 1: Get Started

Dr. Frank Zickert

PyQML

Contents

1. Introduction

Welcome to **Hands-On Quantum Machine Learning With Python**. This book is your comprehensive guide to get started with "*Quantum Machine Learning*" – the use of **quantum computing** for the computation of **machine learning** algorithms.

Hands-On Quantum Machine Learning With Python strives to be the perfect balance between theory taught in a textbook and the actual hands-on knowledge you'll need to implement real-world solutions.

Inside this book, you will learn the basics of quantum computing and machine learning in a practical and applied manner. And you will learn to use state-of-the-art quantum machine learning algorithms.

By the time you finish this book, you'll be well equipped to apply quantum machine learning to your projects. Then, you will be in the pole position to become a "*Quantum Machine Learning Engineer*" – the job to become the sexiest job of the 2020s.

1.1 Who This Book Is For

This book is for **developers**, **programmers**, **students**, and **researchers** who have at least some programming experience and want to become proficient in quantum machine learning.

Don't worry if you're just getting started with quantum computing and machine learning. We will begin with the basics, and we don't assume prior

knowledge of machine learning or quantum computing. So you will not get left behind.

If you have experience in machine learning or quantum computing, the respective parts may repeat concepts you're already familiar with. However, this may make learning the corresponding new topic easier and provide a slightly different angle to the known.

This book offers a practical, hands-on exploration of quantum machine learning. Rather than working through tons of theory, we will build up practical intuition about the core concepts. We will acquire the exact knowledge we need to solve practical examples with lots of code. Step by step, you will extend your knowledge and learn how to solve new problems.

Of course, we will do some math. Of course, we will cover a little physics. But I don't expect you to hold a degree in any of these two fields. We will go through all the concepts we need. While this includes some mathematical notation and formulae, we keep it at the minimum required to solve our practical problems.

The theoretical foundation of quantum machine learning may appear overwhelming at first sight. But, be assured that it is not harder than learning a new programming language when put into the proper context and explained conceptually. And this is what's inside **Hands-On Quantum Machine Learning With Python**.

Of course, we will write code. A lot of code. Do you know a little Python? Great! If you don't know Python but another language, such as Java, Javascript, or PHP, you'll be fine, too. If you know programming concepts (such as if-then else-constructs and loops), then learning the syntax is a piece of cake. If you're familiar with functional programming constructs, such as map, filter, and reduce, you're already well equipped. If not, don't worry. We will get you started with these constructs, too. We don't expect you to be a senior software developer. We will go through all the source code—line by line.

By the time you finish the first few chapters of this book, you will be proficient with doing the math, understanding the physics, and writing the code you need to graduate to the more advanced content.

This book is not just for beginners. There is a lot of advanced content in here, too. Many chapters of **Hands-On Quantum Machine Learning With Python** cover, explain, and apply quantum machine learning algorithms developed in the last two years. You can directly apply the insights this book provides in your job and research. The time you save by reading through Hands-On Quantum Machine Learning With Python will more than pay for itself.

1.2 Book Organization

Machine learning and quantum computing rely on math, statistics, physics, and computer science. This is a lot of theory. Covering it all upfront would be pretty exhaustive and fill at least one book without any practical insight.

However, without understanding the underlying theoretical concepts, the code examples on their own do not provide many practical insights, either. While libraries free you from tedious implementation details, the code, even though short, does not explain the core concepts.

This book provides the theory needed to understand the code we're writing to solve a problem. For one thing, we cover the theory when it applies, and we need it to understand the background of what we are doing. Secondly, we will embed the theory into solving a practical problem and directly see it in action.

As a result, the theory spreads among all the chapters, from simple to complex. You may skip individual examples if you like. But you should have a look at the theoretical concepts discussed in each chapter.

We start with a Variational Hybrid Quantum-Classical Algorithm to solve a binary classification task. First, we have a detailed look at binary classification in chapter 2. Then, in chapter 3, we introduce the basic concept of the quantum bit, the quantum state, and how measurement affects it. Based on these concepts, we build our first Parameterized Quantum Circuit and use it to solve our binary classification task. Such a hybrid algorithm combines the quantum state preparation and measurement with classical optimization.

Then, we learn how to work with single qubits (chapter 5) and with multiple qubits (chapter 6). And we explore the astonishing phenomenon of quantum entanglement. This serves as our basis to develop a Quantum Naïve Bayes classifier (chapter 7). In chapter 8, we dive deep into the specificities of how quantum computing is different from classical computing and how we solve problems the quantum way. It enables us to create and train a quantum Bayesian network (chapter 9).

By now, you'll be experienced in dealing with the probabilistic nature of qubits. It's time to take it one step further. We learn about the qubit phase (chapter 11) and how we can use it to tap the potential of quantum systems (chapter 12). We use it to search the relatives of a passenger on board the Titanic (chapter 13) and approximate a variable's distribution in our quantum Bayesian network. (chapter 14).

You can find the complete source code of this book this Github repository.

1.3　Why Should I Bother With Quantum Machine Learning?

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.

But these algorithms become increasingly hard to train because they consist of billions of parameters. Quantum computers promise to solve such problems intractable with current computing technologies. Moreover, their ability to compute multiple states simultaneously enables them to perform an indefinite number of superposed tasks in parallel. An ability that promises to improve and to expedite machine learning techniques.

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.

But 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.

Quantum machine learning promises to be disruptive. Although this merger of machine learning and quantum computing, both areas of active research, is largely in the conceptual domain, there are already some examples where it is being applied to solve real-life problems. Google, Amazon, IBM, Microsoft, and a whole fleet of high-tech startups strive to be the first to build and sell quantum machine learning systems.

The opportunity to study a technology right when it is about to prove its supremacy is a unique opportunity. Don't miss it.

1.4 Quantum Machine Learning - Beyond The Hype

If there were two terms in computer science that I would describe as overly hyped and poorly understood, I would say *machine learning* and *quantum computing*.

Quantum Machine Learning is the use of *quantum computing* for the computation of *machine learning* algorithms. Could it be any worse?

Figure 1.1: Which future will it be?
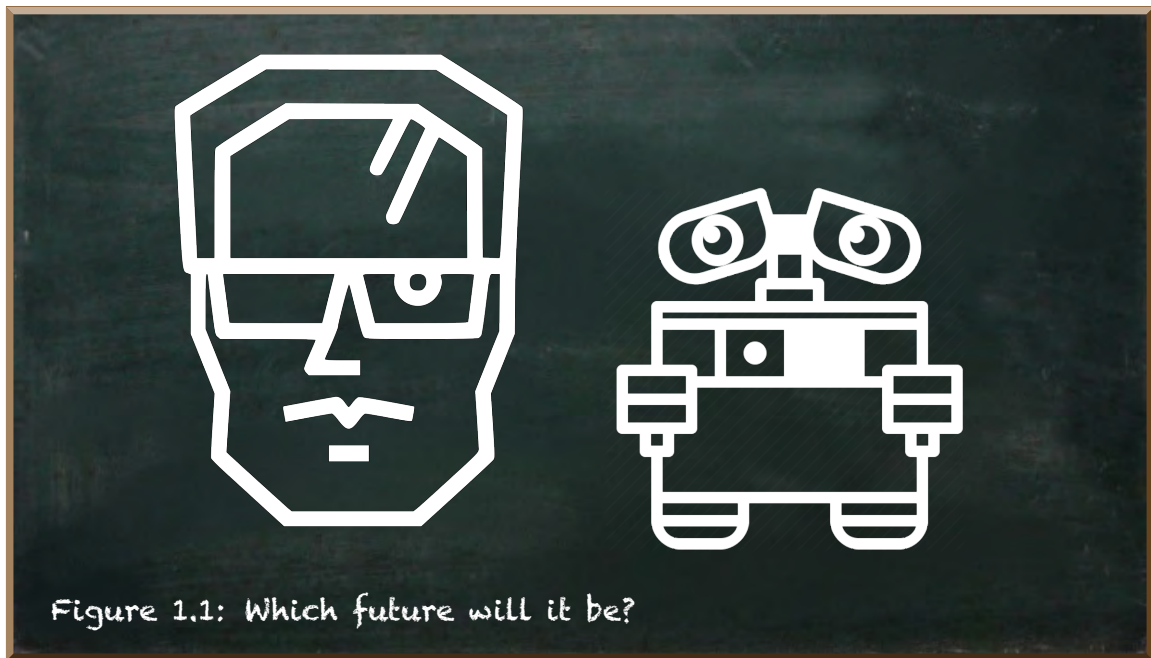
There are many anecdotes on these two technologies. They start at machines that understand the natural language of us humans. And they end at the advent of the Artificial General Intelligence that either manifests as the Terminator-like apocalypse or the Wall-E-like utopia.

Don't fall for the hype! An unbiased and detailed look at a technology helps not to fall for the hype and the folklore. Let's start with machine learning.

1.4.1 What is Machine Learning?

"Machine learning is a thing-labeler, essentially." – Cassie Kozyrkov, Chief Decision Scientist at Google, source –

With machine learning, we aim to put a label onto a yet unlabeled thing. And

there are three main ways of doing it: classification, regression, and segmentation.

In **classification**, we try to predict the discrete label of an instance. Given the input and a set of possible labels, which one is it? Here's a picture. Is it a cat or a dog?

Figure 1.2: Is it a cat or a dog?
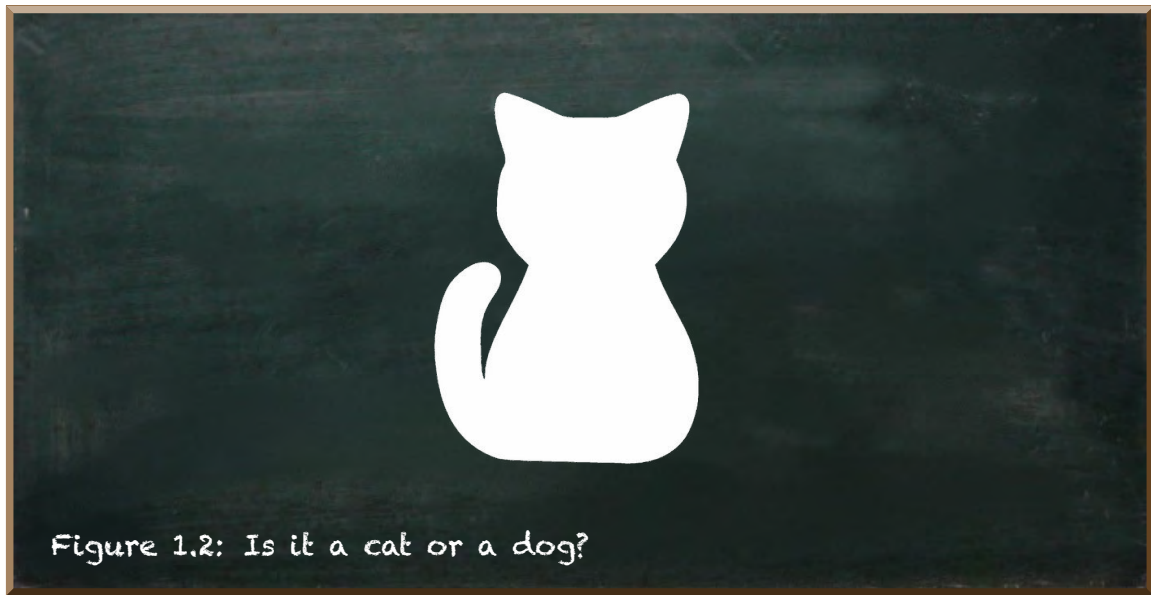
Regression is about finding a function to predict the relationship between some input and the dependent continuous output value.

Figure 1.3: Effective tax rate by gross income
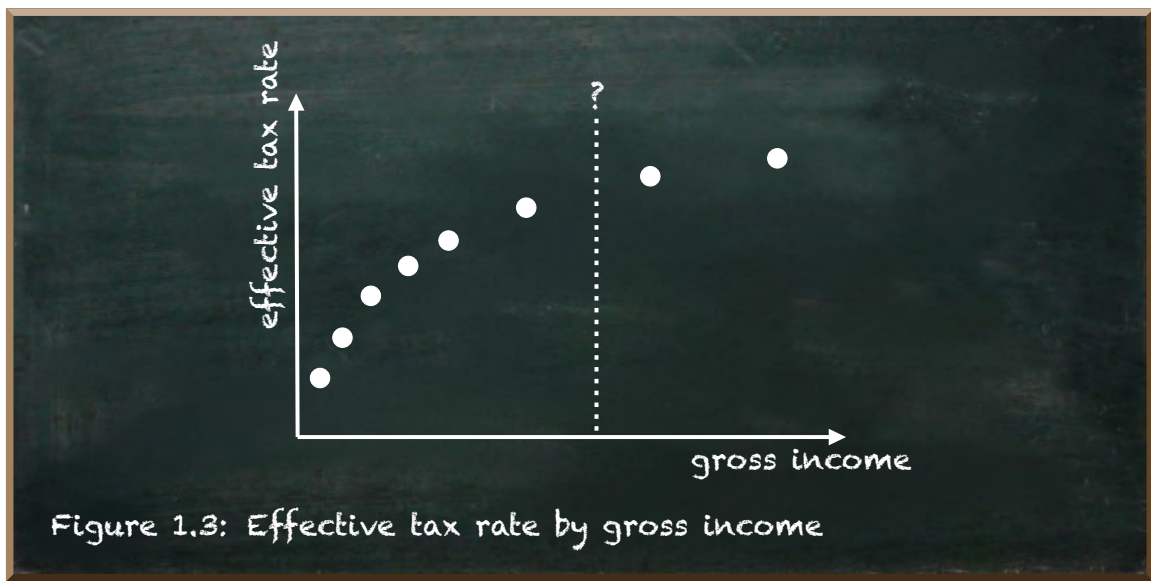
For example, given that you know your friends' income and the effective tax rates, can you estimate your tax rate given your income even though you don't know the actual calculation?

And **segmentation** is the process of partitioning the population into groups with similar characteristics, which are thus likely to exhibit similar behavior. Given that you produce an expensive product, such as yachts, and a population of potential customers, whom do you want to try to sell to?

Figure 1.4: Customer Segmentation
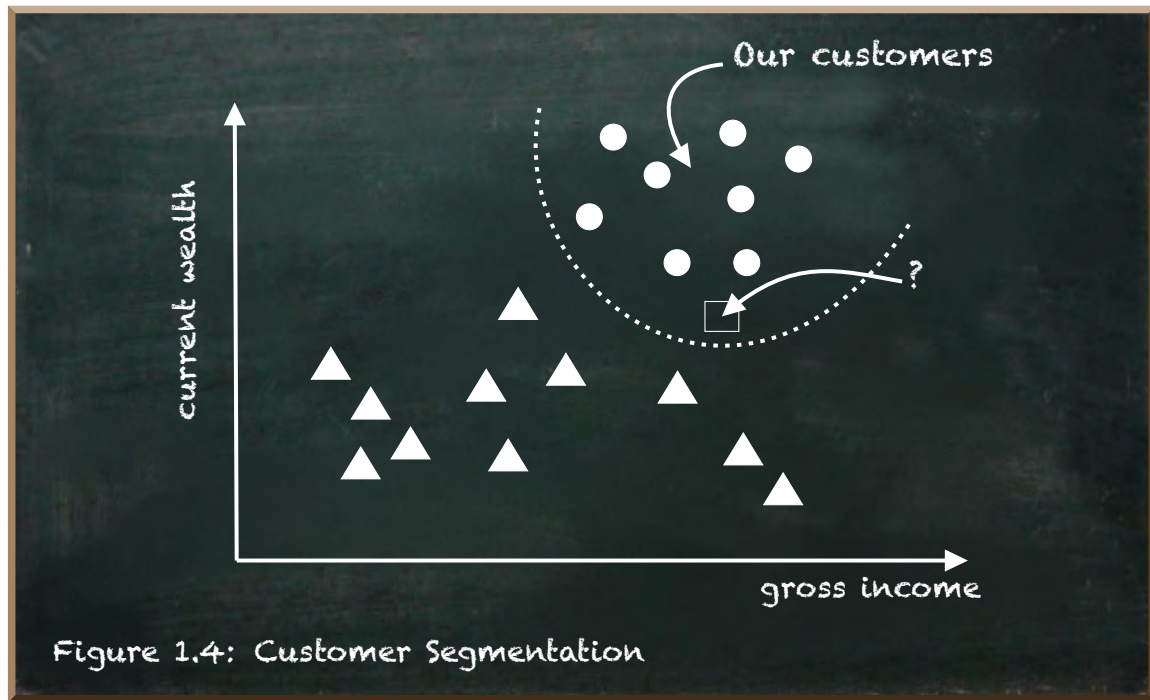
1.4.2 What is Quantum Computing?

Quantum computing is a different form of computation. It uses three fundamental properties of quantum physics: superposition, interference, and entanglement.

Superposition refers to the quantum phenomenon where a quantum system can exist in multiple states concurrently.

> (!) The quantum system does not exist in multiple states concurrently. It exists in a complex linear combination of a state 0 and a state 1. It is a different kind of combination that is neither "or" nor is it "and." We will explore this state in-depth in this book.

Figure 1.5: The quantum superposition
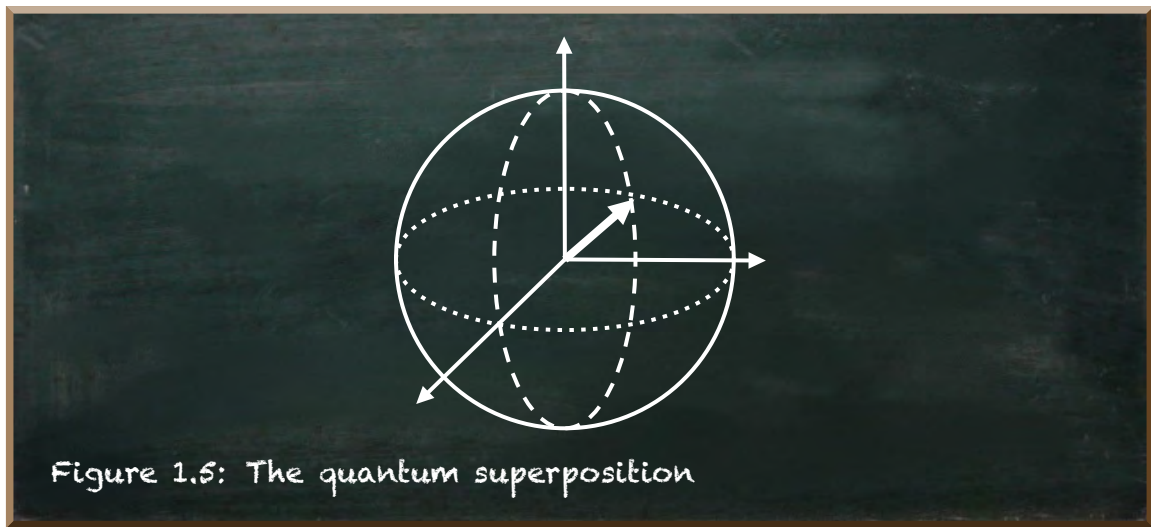
Quantum **interference** is what allows us to bias quantum systems toward the desired state. The idea is to create a pattern of interference where the paths leading to wrong answers interfere destructively and cancel out, but the paths leading to the correct answer reinforce each other.

Figure 1.6: Interference of waves
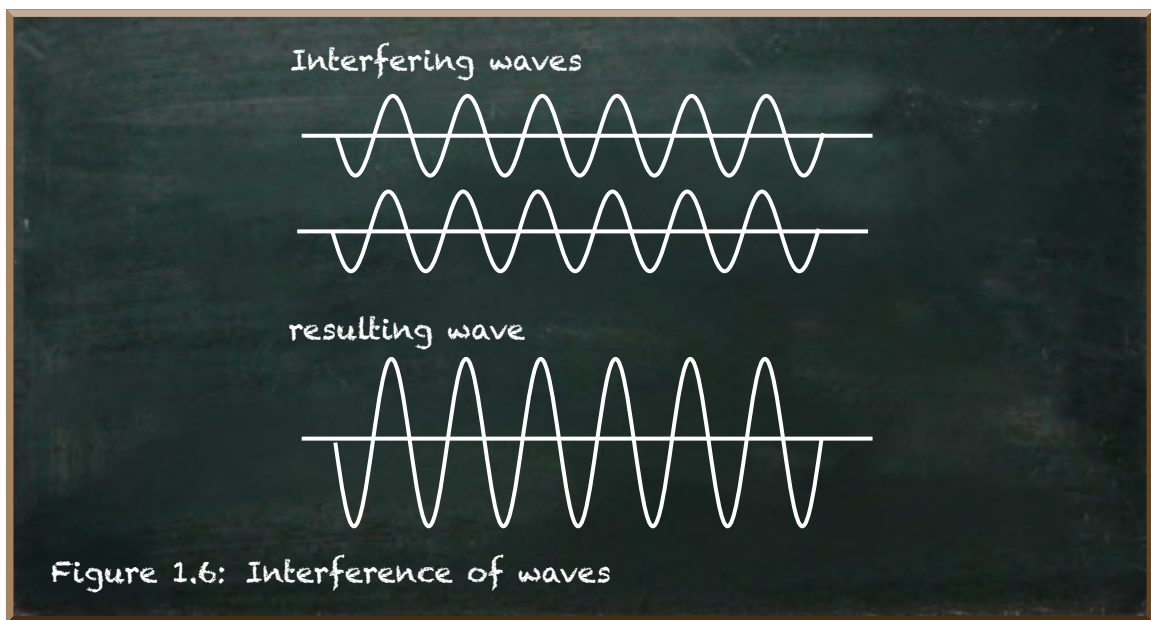
Entanglement is an extremely strong correlation between quantum particles. Entangled particles remain perfectly correlated even if separated by great distances.

Figure 1.7: Entanglement
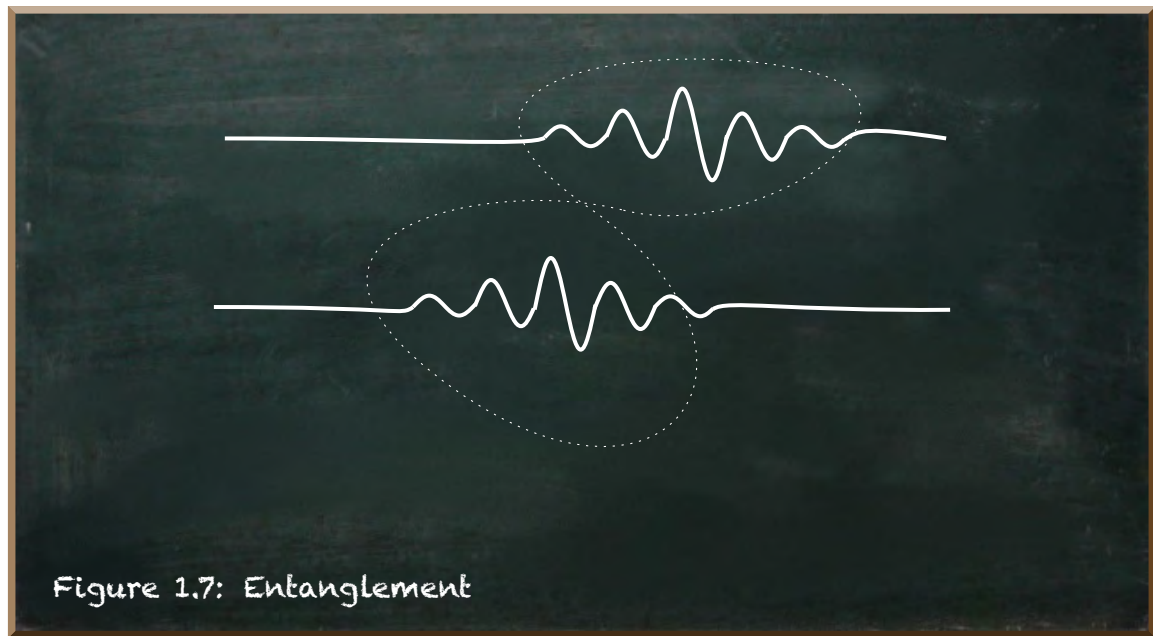
Do you see the Terminator already? No? Maybe Wall-E? No *again*?

Maybe it helps to look at how these things work.

1.4.3 How Does Machine Learning Work?

There are myriads of machine learning algorithms out there. But every one of these algorithms has three components:

- The **representation** depicts the inner architecture the algorithm uses to represent the knowledge. It may consist of rules, instances, decision trees, support vector machines, neural networks, and others.
- The **evaluation** is a function to evaluate candidate algorithm parameterizations. Examples include accuracy, prediction and recall, squared error, posterior probability, cost, margin, entropy, and others.
- The **optimization** describes the way of generating candidate algorithm parameterizations. It is known as the search process – for instance, combinatorial optimization, convex optimization, and constrained optimization.

The first step of machine learning is the development of the architecture, the representation. The architecture specifies the parameters whose values hold the representation of the knowledge. This step determines how suited the solution will be to solve a specific problem. More parameters are not always better. For example, if a linear function can solve our problem, trying to solve it with a solution that consists of millions of parameters is likely to fail. On

the other hand, an architecture with very few parameters may be insufficient to solve complex problems such as natural language understanding.

Figure 1.8: A generalized notion of machine learning
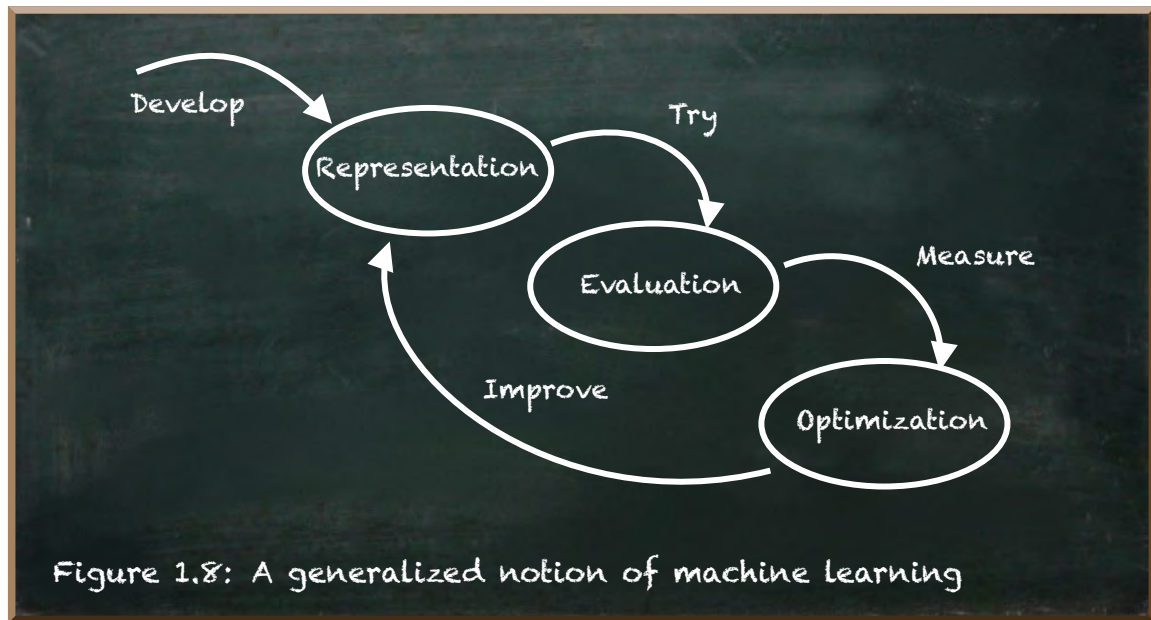
Once we settled for the architecture to represent the knowledge, we train our machine learning algorithm with examples. Depending on the number of parameters, we need many examples. Next, the algorithm tries to predict the label of each instance. Finally, we use the evaluation function to measure how well the algorithm performed.

The optimizer adjusts the representation to parameters that promise better performance concerning the measured evaluation. It may even involve changing the architecture of the representation.

Learning does not happen in giant leaps. Instead, it takes tiny steps. To yield a good performance and depending on the complexity of the problem, it takes several iterations of this general process until the machine can put the correct label on a thing.

1.4.4 What Tasks Are Quantum Computers Good At?

The world of quantum mechanics is different from the physics we experience in our everyday situations. So is the world of quantum computing different from classical (digital) computing.

What makes quantum computing so powerful isn't its processing speed. It is rather slow. What makes quantum computing so powerful isn't its memory,

either. It is absurdly tiny. We're talking about a few quantum bits.

What makes quantum computing so powerful is the algorithms it makes possible because these algorithms exhibit different complexity characteristics than their classical equivalents. To understand what that means, let's have a brief look at complexity theory. Complexity theory is the study of the computational effort required to run an algorithm.

For instance, the computational effort of addition is $\mathcal{O}(n)$. This means that the effort of adding two numbers increases linearly with the size (digits) of the number. The computational effort of multiplication is $\mathcal{O}(n^2)$. The effort increases by the square of the number size. These algorithms are said to be solvable in polynomial time.

But these problems are comparably simple. For example, the best algorithm solving the problem of factorization, finding the prime factors of an n-digit number, is $\mathcal{O}(e^{n^{1/3}})$. It means that the effort increases exponentially with the number of digits.

Figure 1.9: Graphs of common complexity functions
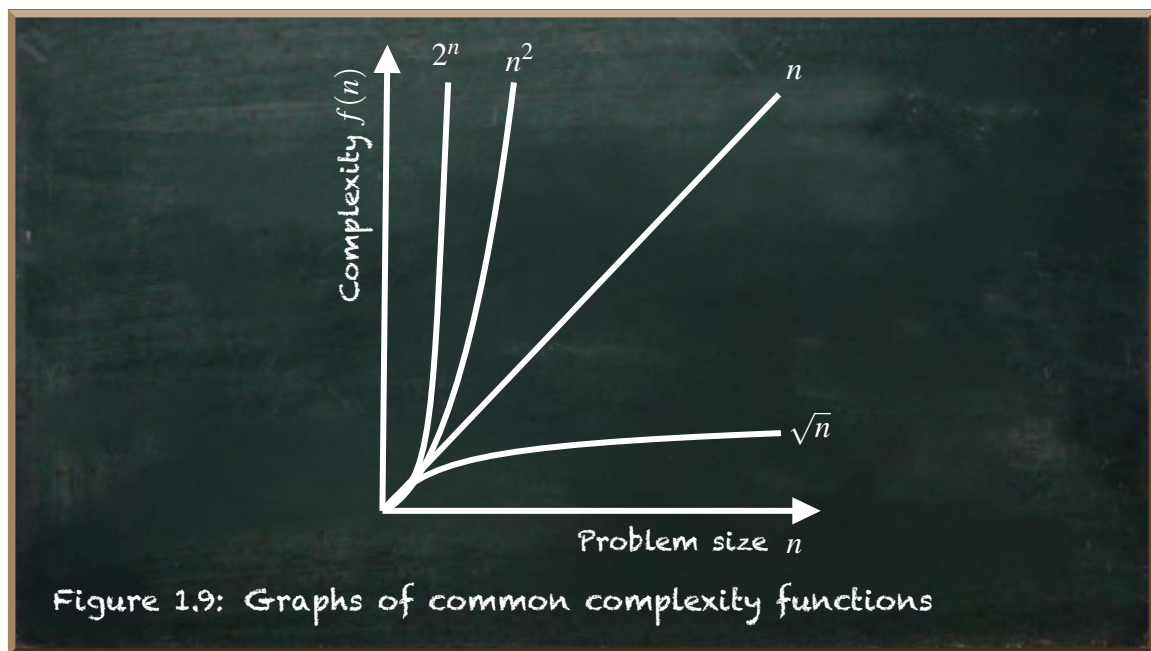
The difference between $\mathcal{O}(n^2)$ and $\mathcal{O}(e^{n^{1/3}})$ complexity must not be underestimated. While your smartphone can multiply numbers with 800 digits in a few seconds, the factorization of such numbers takes about 2,000 years on a supercomputer.

A proper quantum algorithm (such as Shor's algorithm) can use superposition to evaluate all possible factors of a number simultaneously. And rather than calculating the result, it uses interference to combine all possible an-

swers in a way that yields a correct answer. This algorithm solves a factorization problem with $\mathcal{O}\left((\log n)^2 (\log \log n)(\log \log \log n)\right)$ complexity. This is a polynomial complexity! So is multiplication.

Quantum computing is powerful because it promises to solve certain types of mathematical calculations with reduced complexity.

Do you see the Terminator or Wall-E now? Not yet?

1.4.5   The Case For Quantum Machine Learning

Quantum machine learning is the use of quantum computing for the computation of machine learning algorithms.

We have learned that machine learning algorithms contain three components: representation, evaluation, and optimization.

When we look at the representation, current machine learning algorithms, such as the Generative Pre-trained Transformer 3 (GPT-3) network, published in 2020, come to mind. GPT-3 produces human-like text, but it has 175 billion parameters. In comparison, the IBM Q quantum computer has 27 quantum bits, only. Thus, even though quantum bits store a lot more information than a classical bit does (because it is not either 0 or 1), quantum computers are far away from advancing machine learning for their representation ability.

During the evaluation, the machine learning algorithm tries to predict the label of a thing. Classically, this involves measuring and transforming data points. For instance, neural networks rely on matrix multiplications. These are tasks classical computers are good at. However, if you have 175 billion parameters, then calculating the resulting prediction takes quite many matrix multiplications.

Finally, the algorithm needs to improve the parameters in a meaningful way. The problem is to find a set of parameter values that result in better performance. With 175 billion parameters, the number of combinations is endless.

Classical machine learning employs heuristics that exploit the structure of the problem to converge to an acceptable solution within a reasonable time. However, despite the use of even advanced heuristics, training the GPT-3 would require 355 years to train on a single GPU (Graphics Processing Unit) and cost $4.6 million. To get a feeling of what reasonable means in this context.

The main characteristic of quantum computing is the ability to compute multiple states concurrently. A quantum optimization algorithm can combine

all possible candidates and yield those that promise good results. Therefore, quantum computing promises to be exponentially faster than classical computers in the optimization of the algorithm. But this does not mean we only look at the optimization. Instead, the optimization builds upon running an evaluation, and the evaluation builds upon the representation. Thus, tapping the full potential of quantum computing to solve the machine learning optimization problem requires the evaluation and the representation to integrate with the quantum optimizer.

Keeping in mind what classical machine learning algorithms can do today. If we expect quantum computing to reduce the complexity of training such algorithms by magnitudes, then the hype becomes understandable because we are "only" magnitudes away from things like Artificial General Intelligence.

But of course, building Artificial General Intelligence requires more than computation. It needs data. And it needs the algorithms.

The development of such algorithms is one of the current challenges in quantum machine learning. But there's another aspect to cope with in that challenge. That aspect is that we are in the NISQ era.

1.5 Quantum Machine Learning In The NISQ Era

Quantum computing is a different form of computation. As we just learned, a form can change the complexity of solving problems, making them tractable. But this different form of computation brings its challenges.

Digital computers need to distinguish between two states: 0 and 1. The circuits need to tell the difference between high voltage and low voltage. Whenever there is a high voltage, it is 1 and if there is a lower voltage, it is 0. This discretization means that errors must be relatively large to be noticeable, and we can implement methods for detecting and correcting such errors.

Unlike digital computers, quantum computers need to be very precise because they keep a continuous quantum state. Quantum algorithms base on specific manipulations of continuously varying parameters. In quantum computers, errors can be arbitrarily small and impossible to detect, but still, their effects can build up to ruin a computation. This fragile quantum state is very vulnerable to the noise coming from the environment around the quantum bit. For example, noise can arise from control electronics, heat, or impurities in the quantum computer's material itself and cause serious computing errors that may be difficult to correct.

But to keep the promises quantum computers make, we need fault-tolerant devices. We need devices to compute Shor's algorithm for factoring. We need machines to execute all the other algorithms that we know in theory that solve problems intractable for digital computers.

But such devices require millions of quantum bits. This overhead is required for error correction since most of these sophisticated algorithms are extremely sensitive to noise. Current quantum computers have up to 27 quantum bits. Even though IBM strives for a 1000-quantum bits computer by 2023, we expect the quantum processors in the near term to have between 50 and 100 quantum bits. Even if they exceed these numbers, they remain relatively small and noisy. These computers can only execute short programs since the longer the program is, the more noise-related output errors will occur.

Nevertheless, programs that run on devices beyond 50 quantum bits become extremely difficult to simulate on classical computers already. These relatively small quantum devices can do things infeasible for a classical computer.

And this is the era we're about to enter. The era when we can build quantum computers that, while not being fault-tolerant, can do things classical computers can't. We describe this era by the term "Noisy Intermediate-Scale Quantum" - **NISQ**.

Noisy because we don't have enough qubits to spare for error correction. And "Intermediate-Scale" because the number of quantum bits is too small to compute sophisticated quantum algorithms but large enough to show quantum advantage or even supremacy.

The current era of NISQ-devices requires a different set of algorithms, tools, and strategies.

For instance, Variational Quantum-Classical Algorithms have become a popular way to think about quantum algorithms for near-term quantum devices. In these algorithms, classical computers perform the overall machine learning task on information they acquire from running the hard-to-compute calculations on a quantum computer.

The quantum algorithm produces information based on a set of parameters provided by the classical algorithm. Therefore, they are called Parameterized Quantum Circuits (PQCs). They are relatively small, short-lived, and thus suited for NISQ-devices.

1.6 I learned Quantum Machine Learning The Hard Way

I did not have the fortune to take a quantum computing class in college, not to speak of a course in quantum machine learning. At the time, it wouldn't have been much fun either. In the early 2000s, quantum computing was just about to take the step from a pure theory to evaluation in research labs. It was a field for theoretical physicists and mathematicians.

At the time, I haven't even heard about it. When I did for the first time, I think it was around 2008, researchers had successfully entangled qubits and were able to control them. Then, of course, Star Trek-like transportation came to mind when I heard two physically apart particles could share a state so that it was possible to change the state of one particle by observing the other.

Yet, until around 2014, I did not pay much attention. I was too busy writing my doctoral dissertation about assessing the effort caused by the requirements in a software development project. When I returned to everyday life, I was just right in time to experience the end of the second AI winter and the advent of practical machine learning. What had been theory thus far became a reality now.

When I got into machine learning, the field was already quite evolved. Libraries such as Scikit-Learn, later Keras, TensorFlow, and PyTorch made machine learning algorithms convenient. So even though my favorite books were published sometime later, there were already many good books and learning material available.

> **i** My favorite books are Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurélien Géron, released in 2017, and Deep Learning with Python by Francois Chollet, released in 2018.

But the models we're developing today become increasingly hard to train. As mentioned before, Open AI's GPT-3 model that uses deep learning to produce human-like text would require 355 years on a single GPU. Thus, it is hard to believe that we can reach the upcoming milestones classically. This insight brought quantum computing back into my focus. Quantum computing promises to reduce the computational complexity of specific algorithms by magnitudes. It promises to solve tasks in a few seconds classical computers would need thousands of years for. It may even prevent us from the next AI

winter caused by the inability to reach the following milestones of machine learning.
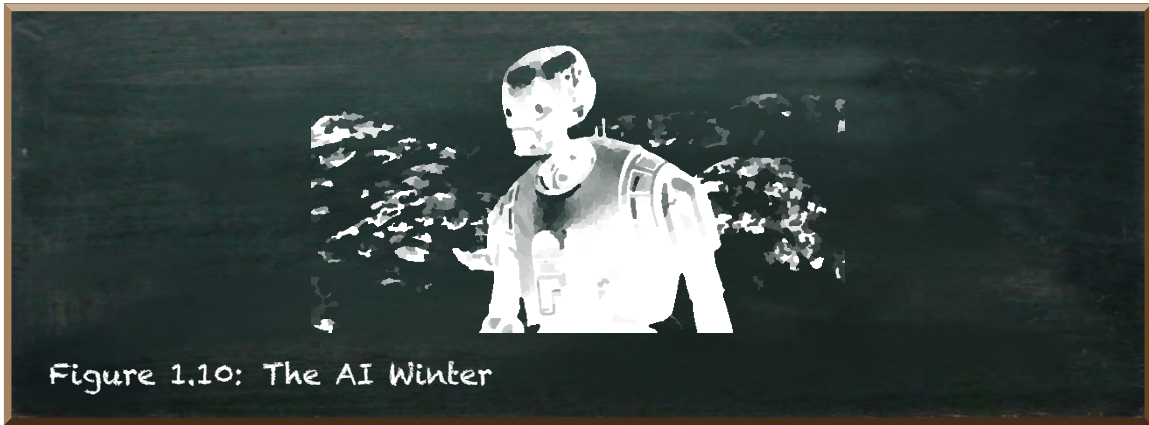
Figure 1.10: The AI Winter

In 2018, I started to deep dive into quantum machine learning. Scientific papers and a few academic books were all I could find. And these did not cover quantum machine learning but quantum computing in general. So I was happy about every little piece.

These quantum computing publications left me scratching my head. Most of the papers are pretty heavy on math and assume you're familiar with much physical jargon. I could not even find an appropriate starting point or guidance on how to structure my learning efforts.

Frustrated with my failed attempts, I spent hours searching on Google. Finally, I hunted for quantum tutorials, only to come up empty-handed.

I could see the potential value of quantum computing for machine learning. Yet, I couldn't see how all these parts of quantum computing fit together. Entry-level material was hard to find. And practical guides were simply not existent. I wanted to get started, but I had nothing to show for my effort, except for a stack of quantum computing papers on my desk that I hardly understood.

Finally, I resorted to learning the theory first. Then, I heard about Qiskit, the IBM quantum SDK for Python. Its documentation was relatively poor at the time, especially if you were not familiar with all the physical jargon and its underlying theory. But it let me experience what some of these things like superposition, entanglement, and interference meant practically.

This practical knowledge enabled me to connect quantum computing with the algorithms I knew from machine learning. I found my way to quantum machine learning success through myriads of trial-and-error experiments,

countless late nights, and much endurance. I believe that painstakingly working everything out in small pieces impacted how I understand quantum machine learning. Again, though, I would recommend not taking the same path.

My takeaways are:

- You don't need to cram all the theory before you start applying it.
- You don't need to work through tons of equations.
- You don't need to be a mathematician to master quantum machine learning.
- You don't need to be a physicist to understand quantum machine learning.
- You'll do great as a programmer, an engineer, a data scientist, or any other profession.
- But quantum machine learning is taught the wrong way.

When I started studying the quantum part of quantum machine learning, I took a deep dive into theory and math. Because this is what most quantum computing resources focus on.

Of course, it is desirable to have an understanding of the underlying math and the theory. But more importantly, you need to have a sense of what the concepts mean in practice. You need to know what you can do and how you need to do it. But you don't need to know how it works physically.

Don't get me wrong. In quantum machine learning, theory and math are essential. But if you don't use the theoretical knowledge and apply it to solve real-world tasks, then you'll have a hard time finding your space in the quantum machine learning world. So it would be best if you became a quantum machine learning practitioner from the very beginning. In contrast to the days when I started, today, there are quite a few resources available. But most of them fall into one of the following categories.

- Theoretical papers with lots of equations prove some quantum speedup of an algorithm. Yet, they don't show any code.
- Textbooks on quantum computing explain the concepts. But they are short on showing how to use them for a purpose.
- Blog posts show you an actual algorithm in code. But they don't relate the code to any underlying concept. While you see it works, you don't learn anything about why and how it works.

By no means do I want to say these resources are not worth reading. But none of these resources are helpful to learn how to apply quantum machine learning. For someone just about to start with quantum machine learning, you

would need to invest a lot of time and effort for little to no practical return.

There is a fundamental disconnect between theory and practice. There's a gap I want to help to fill with Hands-On Quantum Machine Learning with Python so you can learn in a more efficient–a better way.

This is the book I wish I had when I first started studying quantum machine learning. Inside this book, you'll find practical walkthroughs and hands-on tutorials with lots of code. The book introduces new theory just in time you need it to take the next step. You'll learn a lot of theory. But you're not left alone with it. We directly apply our newly acquired knowledge to solve an actual problem.

We will not only implement different quantum machine learning algorithms, such as Quantum Naïve Bayes and Quantum Bayesian Networks. But we also use them to solve actual problems taken from Kaggle.

By the time you finish this book, you'll know these algorithms, what they do, why you need them, how they work, and most importantly, how to use them.

Hands-On Quantum Machine Learning With Python strives to be the perfect balance between theory taught in a textbook and the actual hands-on knowledge you'll need to implement real-world solutions.

This book is your comprehensive guide to get started with Quantum Machine Learning–the use of quantum computing for machine learning tasks.

1.7 Quantum Machine Learning Is Taught The Wrong Way

The literature on quantum computing is full of physical jargon and formulae. Let's take the Variational Quantum Eigensolver (VQE), for instance.

> VQE can help us to estimate the energy of the ground state of a given quantum mechanical system. This is the upper bound of the lowest eigenvalue of a given Hamiltonian. It builds upon the variational principle that is described as: $\langle \Psi_\lambda | H | \Psi_\lambda \rangle >= E_0$

If you don't hold a degree in physics, the first and natural reaction is to put the article away.

"Well, nice try. Maybe the whole topic is not for me", you think. "Maybe, quantum computing is beyond my reach".

Don't give up that fast. Physicists and mathematicians discovered most of the stuff in quantum computing. Of course, they build upon the knowledge of their peers when they share insights and teach their students. So it is reasonable that they use the terms they are familiar with.

You wouldn't use the vocabulary of a bartender to explain programming and machine learning either, would you? But maybe, we should.

It is reasonable to assume a certain kind of knowledge when we talk or write about something. But should we restrain students of other, nearby disciplines from learning the stuff? For example, why shouldn't we support a computer scientist or a software engineer in learning quantum computing?

I've got a clear opinion. I believe anyone sincerely interested in quantum computing should be able to learn it. There should be resources out there catering to the student's needs, not to the teacher's convenience. But, of course, this requires a teacher to explain the complex stuff in allegedly simple language.

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

Figure 1.11: Albert Einstein
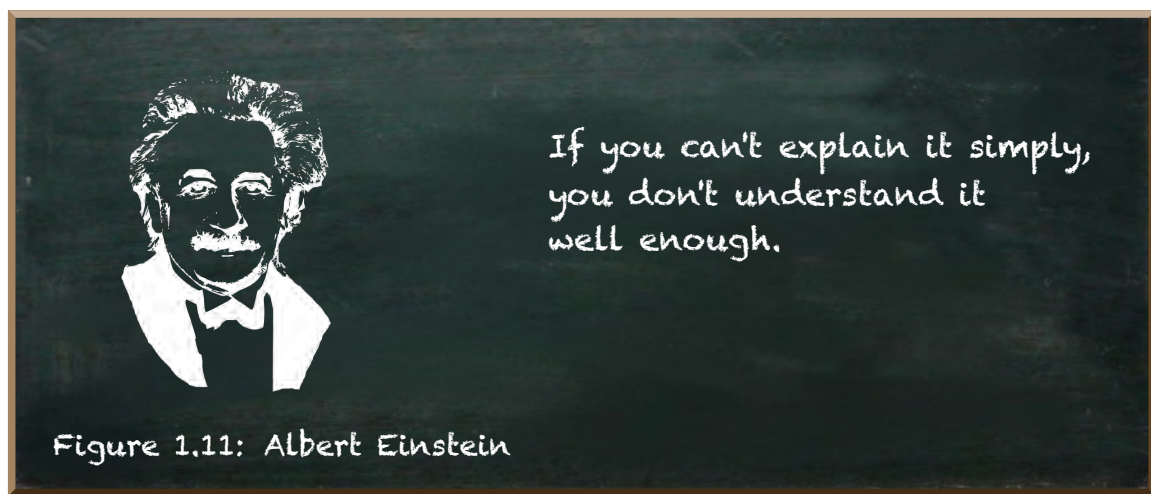
I wouldn't dare to say I understood quantum computing well enough to explain it with the vocabulary bartenders use. But I'd give it a shot explaining it to a computer scientist and a software engineer. I don't see a reason to restrict this field to physicists only.

Of course, it is desirable to understand the underlying theory of quantum mechanics. Of course, it is desirable to be able to do the math. But, more importantly, you need to understand how to solve a certain problem.

In quantum computing, we use quantum superposition, entanglement, and interference to solve tasks. These are astonishing and maybe counter-intuitive phenomena. But no matter how weird they may appear, quantum

mechanical systems adhere to a particular set of physical laws. And these laws make the systems behave in specific ways. How deep do you have to know the physical laws? How much quantum math do you need?

I don't believe anyone (including me) understands how a classical computer works. Yet, we all use them. We even program them! I learned how to code a classical computer because my teachers explained it to me in a way I could understand back then.

My high-school teacher explained the concepts of data types and algorithms in an applied way. He taught me how they work and what they are good for. So even though–or maybe because–we didn't go through electro-mechanical circuits and information theory, I learned to program.

"Maybe quantum computing is different," you say? "Maybe, the stuff in there is too complex to learn without a degree in physics!"

The theoretical foundation of quantum machine learning may appear overwhelming at first sight. But, be assured, when put into the proper context and explained conceptually, it is not more complicated than learning a new programming language.

I genuinely believe developers, programmers, and students who have at least some programming experience can become proficient in quantum machine learning. However, teaching quantum machine learning the right way requires a different approach–a practical approach.

Rather than working through tons of theory, a good approach builds up practical intuition about the core concepts. I think it is best to acquire the exact theoretical knowledge we need to solve practical examples.

Quantum machine learning relies on math, statistics, physics, and computer science. Covering it all upfront would be pretty exhaustive and fill at least one book without any practical insight. However, without understanding the underlying theoretical concepts, code examples on their own do not provide valuable insights, either.

This books combines practical examples with the underlying theory.

1.8 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)

1.8.1 Python

For all examples inside **Hands-On Quantum Machine Learning With Python**, 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, machine learning tools, such as PyTorch and Tensorflow, as well as quantum computing tools, such as Qiskit and Cirq, are available as Python SDKs.

1.8.2 Jupyter

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.

1.8.3 Libraries and Packages

We will use the following libraries and packages:

- Scikit-learn
- Pandas
- Qiskit

Scikit-learn is the most helpful library for machine learning in Python. It contains a range of supervised and unsupervised learning algorithms. Scikit-learn builds upon a range of other handy libraries, such as:

- NumPy: Work with n-dimensional arrays
- SciPy: Fundamental library for scientific computing
- Matplotlib: Comprehensive 2D/3D plotting
- IPython: Enhanced interactive console
- Sympy: Symbolic mathematics

Pandas provides convenient data structures and analysis tools. Qiskit is IBM's quantum computing SDK.

1.8.4 Virtual Environment

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.

1.8.5 Configuring Ubuntu For Quantum Machine Learning with Python

In this section, we go through the installation on Ubuntu Linux. 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.

Other Linux distributions (such as Debian) or MacOS (that also builds upon Unix) are also ok. But there are a few more aspects to consider.

All the code should work on Windows, too. However, the configuration of a

Windows working environment can be a challenge on its own. Fortunately, there is a way out. So, if you have a Windows operating system, look at the next section 1.8.6 before you continue with the following instructions.

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 3.8.5 (the latest stable release at the time of writing).

```
$ mkdir /tmp/Python38
$ cd /tmp/Python38
$ wget https://www.python.org/ftp/python/3.8.5/Python-3.8.5.tar.xz
$ tar xvf Python-3.8.5.tar.xz
$ cd /tmp/Python38/Python-3.8.5
$ ./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

```
$ pip install qiskit
```

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.

Install further dependencies required of Qiskit and Scikit-Learn. If you don't use a virtual environment, use the `--user` flag here, too.

```
$ pip install numpy scipy matplotlib ipython pandas sympy nose seaborn
```

Install Scikit-Learn, with the `--user` flag if you're not using a virtual environment.

```
$ pip install scikit-learn
```

Install drawing libraries:

```
$ pip install pylatexenc ipywidgets qutip
```

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

```
$ jupyter lab
```

1.8.6 How To Setup JupyterLab For Quantum Computing — On Windows

If you're a Python developer, there's no way around a Unix-based operating system. Python is a language to write software that's usually supposed to run at a server. And most servers run some kind of Linux.

Figure 1.12: Windows, Jupyter, Qiskit

Consequently, the default configuration in Python caters to the specificities of a Unix-based system. While Python works on Windows, too, it requires a lot more attention to get all the configuration details right. Starting from the path separator that is not a slash but a backslash (\) to the different charset (`windows−1252`), to different commands (e.g. `del /s /q` instead of `rm`), Windows differs in quite a few aspects.

While Linux is great for developing, you may prefer Windows in other situations. Maybe you don't even have a choice. Your working computer simply runs Windows. Full stop.

Fortunately, there's a solution - at least if you're running Windows 10. Windows 10 contains WSL2, the Windows Subsystem for Linux. It 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.

You can proceed with the installation of the libraries and packages as described in the previous section 1.8.5.

2. Binary Classification

2.1 Predicting Survival On The Titanic

The sinking of the Titanic is one of the most infamous shipwrecks in history.

On April 15, 1912, the Titanic sank after colliding with an iceberg. Being considered unsinkable, there weren't enough lifeboats for everyone on board. As a result, 1502 out of 2224 passengers and crew members died that night.

Of course, the 722 survivors must have had some luck. But it seems as if certain groups of people had better chances to survive than others. Therefore, the Titanic sinking has also become a famous starting point for anyone interested in machine learning.

If you have some experience with machine learning, you'll probably know the legendary Titanic ML competition provided by Kaggle.

If you don't know Kaggle yet, Kaggle is among the world's largest data science communities. It offers many exciting datasets, and therefore, it is an excellent place to get started.

The problem to be solved is simple. Use machine learning to create a model that, given the passenger data, predicts which passengers survived the Titanic shipwreck.

2.2 Get the Dataset

To get the dataset, you'll need to create a Kaggle account (it's free) and join the competition. Even though Kaggle is all about competitions, you don't need to take part in them actively by uploading your solution.
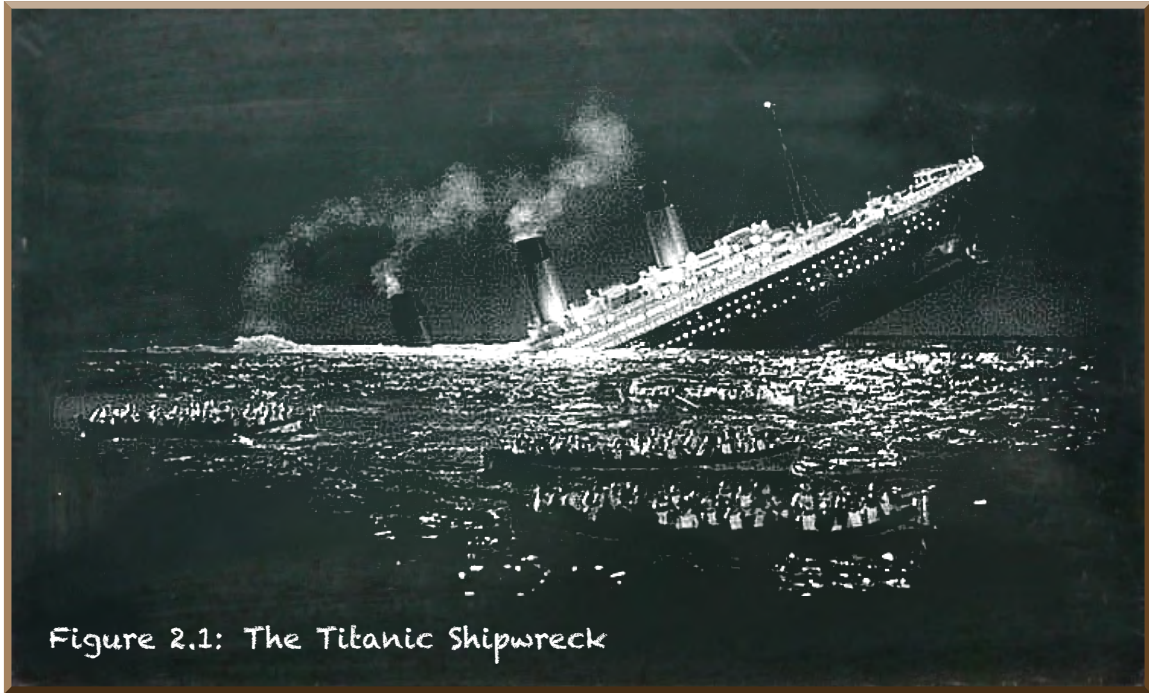
Figure 2.1: The Titanic Shipwreck

When you join a competition, you need to accept and abide by the rules that govern how many submissions you can make per day, the maximum team size, and other competition-specific details.

You'll find the competition data in the Data tab at the top of the competition page. Then, scroll down to see the list of files.

There are three files in the data:

- `train.csv`
- `test.csv`
- `gender_submission.csv`

The file `train.csv` contains the data of a subset of the Titanic's passengers. This file is supposed to serve your algorithm as a basis to learn whether a passenger survived or not.

The file `test.csv` contains the data of another subset of passengers. It serves to determine how well your algorithm performs.

The `gender_submission.csv` file is an example that shows how you should structure your predictions if you plan to submit them to Kaggle. Since we're here to start learning and not yet be ready to compete, we'll skip this file.

Download the files `train.csv` and `test.csv`.

2.3 Look at the data

The first thing we need to do is to load the data. We use *Pandas* for that. It is renowned in the machine learning community for data processing. It offers a variety of useful functions, such as a function to load `.csv`-files: `read_csv`.

Listing 2.1: Load the data from the csv-files

```
1  import pandas as pd
2
3  train = pd.read_csv('./data/train.csv')
4  test = pd.read_csv('./data/test.csv')
```

We loaded our data into `train` and `test`. These are *Pandas* `DataFrame`s.

A `DataFrame` keeps the data in a two-dimensional structure with labels. Such as a database table or a spreadsheet. It provides a lot of valuable attributes and functions out of the box.

For instance, the `DataFrame`'s attribute `shape` provides a tuple of two integers that denote the number of rows and columns.

Let's have a look:

Listing 2.2: The shapes of the Titanic datasets

```
1  print('train has {} rows and {} columns'.format(*train.shape))
2  print('test has {} rows and {} columns'.format(*test.shape))
```

```
train has 891 rows and 12 columns
test has 418 rows and 11 columns
```

We can see we have 891 training and 418 testing entries. But, more interestingly, the `train` dataset has one more column than the `test` dataset.

The `DataFrame`'s `info()` method shows some more detailed information. Have a look at the `train` dataset.

Listing 2.3: The structure of the train dataset

```
1  train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

The `info` method returns a list of the columns: their index, their names, how many entries have actual values (are not `null`), and the type of values.

Let's have a look at the `test` dataset, too.

Listing 2.4: The structure of the test dataset

```
1  test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  418 non-null    int64
 1   Pclass       418 non-null    int64
 2   Name         418 non-null    object
 3   Sex          418 non-null    object
 4   Age          332 non-null    float64
 5   SibSp        418 non-null    int64
 6   Parch        418 non-null    int64
 7   Ticket       418 non-null    object
 8   Fare         417 non-null    float64
 9   Cabin        91 non-null     object
 10  Embarked     418 non-null    object
dtypes: float64(2), int64(4), object(5)
memory usage: 36.0+ KB
```

When comparing both info, we can see the `test` dataset misses the column `Survived`, indicating whether a passenger survived or died.

As Kaggle notes, they use the `test` dataset to evaluate the submissions. If they provided the correct answer, it wouldn't be much of a competition anymore, would it? It is our task to predict the correct label.

Since we do not plan to submit our predictions to Kaggle to evaluate how our algorithm performed, the `test` dataset is quite useless for us.

So, we concentrate on the `train` dataset.

The `info` output is relatively abstract. Wouldn't it be good to see some actual data? No problem. That's what the `head` method is for.

The `head` method shows the column heads and the first five rows. So, with this impression, let's go through the columns. You can read an explanation on the Kaggle page, too.

Listing 2.5: Look at the data

```
1  train.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Each column represents one feature of our data. The `PassengerId` is a consecutive number identifying each row. `Survived` is the indicator of whether the passenger survived (0 = No, 1 = Yes). `Pclass` is the ticket class (1 = 1st, 2 = 2nd, 3 = 3rd). Then we have self-explanatory `Name`, `Sex`, and `Age`.

`SibSp` denotes the number of this passenger's siblings or spouses aboard the Titanic. `Parch` indicates the number of this passenger's parents or children aboard the Titanic.

Then, there is the `Fare` the passenger paid, the `Cabin` number, and the port of embarkation (`embarked`) (C = Cherbourg, Q = Queenstown, S = Southampton).

2.4 Data Preparation and Cleaning

Our data have different types. There are numerical data, such as `Age`, `SibSp`, `Parch`, and `Fare`. There are categorical data. Some of the categories are represented by numbers (`Survived`, `Pclass`). Some are represented by text (`Sex` and `Embarked`). And there is textual data (`Name`, `Ticket`, and `Cabin`).

This is quite a mess for data we want to feed into a computer. Furthermore, when looking at the result of `train.info()`, you can see that the counts vary for different columns. While we have 891 values for most columns, we only have 714 for `Age`, 204 for `Cabin`, and 889 for `Embarked`.

Before we can feed our data into any machine learning algorithm, we need to clean up.

2.4.1 Missing Values

Most machine learning algorithms don't work well with missing values. There are three options of how we can fix this:

- Get rid of the corresponding rows (removing the passengers from consideration)
- Get rid of the whole column (remove the entire feature for all passengers)
- Fill the missing values (for example, with zero, the mean, or the median)

Listing 2.6: Cope with missing values

```
1  # option 1
2  # We only have two passengers without it. This is bearable
3  train = train.dropna(subset=["Embarked"])
4
5  # option 2
6  # We only have very few information about the cabin, let's drop it
7  train = train.drop("Cabin", axis=1)
8
9  # option 3
10 # The age misses quite a few times. But intuition
11 # says it might be important for someone's chance to survive.
12 mean = train["Age"].mean()
13 train["Age"] = train["Age"].fillna(mean)
14
15 train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 889 entries, 0 to 890
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  889 non-null    int64
 1   Survived     889 non-null    int64
 2   Pclass       889 non-null    int64
 3   Name         889 non-null    object
 4   Sex          889 non-null    object
 5   Age          889 non-null    float64
 6   SibSp        889 non-null    int64
 7   Parch        889 non-null    int64
 8   Ticket       889 non-null    object
 9   Fare         889 non-null    float64
 10  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(4)
memory usage: 83.3+ KB
```

We can accomplish these things easily using DataFrame's dropna(), drop(), and fillna() methods. There is no one best option in general. But you should carefully consider the specific context.

There are only two passengers whose port of embarkation we don't know.

These account for less than 1% of our data. If we disregard these two passengers entirely, we won't see completely different results. Thus, we drop these rows (line 3) with the `dropna`-method.

The `dropna`-method takes the column (`"Embarked"`) as a named parameter `subset`. This parameter specifies the columns that determine whether to remove the row (passenger). If at least one value of these columns is missing, the row gets removed.

The situation is different concerning the `Cabin`. We only have this information for 204 out of 991 passengers. It is questionable if this is enough to draw any information from. We don't know why these values miss. Even if we found the `Cabin` to be highly correlated with the survival of a passenger, we wouldn't know whether this correlation can be generalized to all passengers or whether there is a selection bias, meaning that the fact that we know the `Cabin` depends on some other aspect.

We drop the whole column with the method `drop`. Then, we provide the column (`Cabin`) we want to remove as a positioned argument. The value 1 we provide as a named argument `axis` specifies that we want to remove the whole column.

Next, we know the `Age` of 714 passengers. Removing all the passengers from consideration whose `Age` we don't know doesn't seem to be an option because they account for about 22% of our data, quite a significant portion. Removing the whole column doesn't seem to be a good option either. First, we know the `Age` of most of the passengers, and intuition suggests that the `Age` might be influential for someone's chance to survive.

We fill the missing values with the `fillna` method (line 13). Since we want to fill only the missing values in the `Age` column, we call this function on this column and not the whole `DataFrame`. We provide as an argument the value we want to set. This is the mean age of all passengers we calculated before (line 12).

Great. We now have 889 rows, ten columns, and no missing data anymore.

2.4.2 Identifiers

The goal of machine learning is to create an algorithm that can predict data. Or, as we said before: to put a label on a thing. While we use already labeled data when building our algorithm, the goal is to predict labels we don't know yet.

We don't tell our algorithm how it can decide which label to select. Instead,

we say to the algorithm, "here is the data. Figure it out yourself." That being said, an intelligent algorithm may be able to memorize all the data you provide it with. This is referred to as overfitting. The result is an algorithm performing well on known data but poorly on unknown data.

If our goal was only to predict labels we already know, the best thing we could do is memorize all passengers and whether they survived. But if we want to create an algorithm that performs well even on unknown data, we need to prevent memorization.

We have not even started building our algorithm. Yet, the features we use in our algorithm affect whether the algorithm can memorize data because we have potential identifiers in our data.

When looking at the first five entries of the dataset, three columns appear suspicious: the `PassengerId`, the `Name`, and the `Ticket`.

The `PassengerId` is a consecutive number. Therefore, there should be no connection between how big the number is and whether a passenger survived.

Neither should the name of a passenger or the number on a ticket be a decisive factor for survival. Instead, these are data identifying single passengers. Let's validate this assumption.

Let's have a look at how many unique values are in these columns.

Listing 2.7: Unique values in columns

```python
print('There are {} different (unique) PassengerIds in the data'
    .format(train["PassengerId"].nunique()))
print('There are {} different (unique) names in the data'
    .format(train["Name"].nunique()))
print('There are {} different (unique) ticket numbers in the data'
    .format(train["Ticket"].nunique()))
```

```
There are 889 different (unique) PassengerIds in the data
There are 889 different (unique) names in the data
There are 680 different (unique) ticket numbers in the data
```

`Name` and `PassengerId` are perfect identifiers. Therefore, each of the 889 rows in our dataset has a unique value.

And there are 680 different `Ticket` numbers. A possible explanation for the

`Ticket` not to be a perfect identifier may be family tickets. Yet, a prediction based on this data appears to support memorization rather than learning transferable insights.

We remove these columns.

Listing 2.8: Remove identifying data

```
1  train = train.drop("PassengerId", axis=1)
2  train = train.drop("Name", axis=1)
3  train = train.drop("Ticket", axis=1)
4
5  train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 889 entries, 0 to 890
Data columns (total 8 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   Survived  889 non-null     int64
 1   Pclass    889 non-null     int64
 2   Sex       889 non-null     object
 3   Age       889 non-null     float64
 4   SibSp     889 non-null     int64
 5   Parch     889 non-null     int64
 6   Fare      889 non-null     float64
 7   Embarked  889 non-null     object
dtypes: float64(2), int64(4), object(2)
memory usage: 62.5+ KB
```

2.4.3 Handling Text and Categorical Attributes

Many machine learning algorithms work with numbers, nothing but numbers. If we want to use textual data, we need to translate it into numbers.

Scikit-Learn provides a transformer for this task called `LabelEncoder`.

Listing 2.9: Transforming textual data into numbers

```
1  from sklearn.preprocessing import LabelEncoder
2  le = LabelEncoder()
3
4  for col in ['Sex', 'Embarked']:
5    le.fit(train[col])
6    train[col] = le.transform(train[col])
7
8  train.head()
```

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	1	22.0	1	0	7.2500	2
1	1	1	0	38.0	1	0	71.2833	0
2	1	3	0	26.0	0	0	7.9250	2
3	1	1	0	35.0	1	0	53.1000	2
4	0	3	1	35.0	0	0	8.0500	2

First, we import the LabelEncoder (line 1) and initialize an instance (line 2). Then, we loop through the columns with textual data (Sex and Embarked) (line 4). For each column, we need to fit the encoder to the data in the column (line 5) before we can transform the values (line 6).

Finally, let's have another look at our DataFrame. You can see that both, Sex and Embarked are now numbers (int64). In our case, 0 represents male, and 1 represents female passengers. But when you rerun the transformation, you may yield different assignments.

2.4.4 Feature Scaling

Machine learning algorithms usually work with numbers with identical scales. If numbers have different scales, the algorithm may consider those with higher scales to be more important.

Even though all our data is numerical, it is not yet uniformly scaled. For example, the values of most of the columns range between 0 and 3. But Age and Fare have far bigger scales.

The max method returns the maximum value in a column. As we can see, the oldest passenger was 80 years old, and the highest fare was about 512.

Listing 2.10: The maximum values

```
1 print('The maximum age is {}'.format(train["Age"].max()))
2 print('The maximum fare is {}'.format(train["Fare"].max()))
```

```
The maximum age is 80.0
The maximum fare is 512.3292
```

A common way to cope with data of different scales is min-max-scaling (also known as normalization). This process shifts and rescales values so that they end up ranging from 0 to 1. It subtracts the minimum value from each value and divides it by the maximum minus the minimum value.

Scikit−Learn provides the MinMaxScaler transformer to do this for us.

Listing 2.11: Normalization of the data.

```
1 from sklearn.preprocessing import MinMaxScaler
2
3 scaler = MinMaxScaler()
4 scaler.fit(train)
5 train = scaler.transform(train)
6
7 print('The minimum value is {} and the maximum value is {}'
8    .format(train.min(), train.max()))
```

```
The minimum value is 0.0 and the maximum value is 1.0
```

Again, we first import the transformer (line 1) and initialize it (line 3). Then, we fit the transformer to our data (line 4) and transform it (line 5).

As a result, all the data in our dataset range between 0.0 and 1.0.

> ! The scaler returns a *Numpy*-array instead of a Pandas *DataFrame*.

2.4.5 Training and Testing

We already mentioned the goal of building an algorithm that performs well on data it already knows and predicts the labels of yet unknown data. That's why it is essential to separate the data into a training and a testing set. We use the training set to build our algorithm. And we use the testing set to validate its performance.

Even though Kaggle provides a testing set, we skipped it for not including the Survived column. This is because we would need to ask Kaggle every time we wanted to validate it. To keep things simple and do the validation on our own, we rather spare some rows from the Kaggle training set for testing.

Separating a test set is quite simple. *Scikit-learn* provides a useful method for that, too. This is `train_test_split`.

Further, we need to separate the input data from the resulting label we want to predict.

Listing 2.12: Separating input from labels and training from testing sets

```python
from sklearn.model_selection import train_test_split

input_data = train[:, 1:8]
labels = train[:, 0]

train_input, test_input, train_labels, test_labels = train_test_split(
    input_data, labels, test_size = 0.2)

print('We have {} training and {} testing rows'.format(train_input.shape
    [0], test_input.shape[0]))
print('There are {} input columns'.format(train_input.shape[1]))
```

```
We have 711 training and 178 testing rows
There are 7 input columns
```

We separate the input columns from the labels with Python array indices (lines 3-4). The first column (position 0) contains the Survived flag we want to predict. The other columns have the data we use as input.

`train_test_split` separates the training from the testing data set. The parameter `test_size = 0.2` (= 20%) specifies the portion we want the testing set to have.

We can see that our training data set consists of 711 entries. Accordingly, our testing set consists of 178 entries. We have input seven columns and single-column output. Let's save our prepared data to use in the future without needing to repeat all these steps.

Listing 2.13: Save the data to the filesystem

```python
import numpy as np

with open('data/train.npy', 'wb') as f:
    np.save(f, train_input)
    np.save(f, train_labels)

with open('data/test.npy', 'wb') as f:
    np.save(f, test_input)
    np.save(f, test_labels)
```

2.5 Baseline

Now, we have our input data and the resulting labels. And we have it separated into a training and a testing set. The only thing left is our algorithm.

Our algorithm should predict whether a passenger survived the Titanic shipwreck. This is a classification task since there are distinct outcome values. Specifically, it is a binary classification task because there are precisely two possible predictions (survived or died).

Before developing a quantum machine learning algorithm, let's implement the simplest algorithm we can imagine: a classifier that guesses.

Listing 2.14: A random classifier

```python
import random
random.seed(a=None, version=2)

def classify(passenger):
    return random.randint(0, 1)
```

We import the random number generator (line 1) and initialize it (line 2).

Our classifier is a function that takes passenger data as input and returns either 0 or 1 as output. Similar to our data, 0 indicates the passenger died and 1 the passenger survived.

To use the classifier, we write a Python function that runs our classifier for each item in the training set.

Listing 2.15: The classification runner

```python
1 def run(f_classify, x):
2   return list(map(f_classify, x))
```

This function takes the classifier-function as the first argument (we can replace the classifier later) and the input data (as x) as the second parameter (line 1).

It uses Python's map function to call the classifier with each item in x and return an array of the results.

Let's run it.

Listing 2.16: Run the classifier

```python
1 result = run(classify, train_input)
```

```
[0, 1, 0, ... 0, 1, 1]
```

When we run the classifier with our train_input, we receive a list of predictions.

Since our goal is to predict the actual result correctly, we need to evaluate whether the prediction matches the actual result.

Let's have a look at the accuracy of our predictions.

Listing 2.17: Evaluate the classifier

```python
1 def evaluate(predictions, actual):
2   correct = list(filter(
3     lambda item: item[0] == item[1],
4     list(zip(predictions,actual))
5   ))
6   return '{} correct predictions out of {}. Accuracy {:.0f} %' \
7     .format(len(correct), len(actual), 100*len(correct)/len(actual))
8
9 print(evaluate(run(classify, train_input), train_labels))
```

```
347 correct predictions out of 711. Accuracy 49 %
```

We define another function named `evaluate`. It takes the predictions of our algorithm and the actual results as parameters (line 1).

The term `list(zip(predictions,actual))` (line 4) creates a list of 2-item lists. The 2-item lists are pairs of a prediction and the corresponding actual result.

We `filter` these `items` from the list where the prediction matches the actual result (`lambda item: item[0] == item[1]`) (line 3). These are the correct predictions. The length of the list of correct predictions divided by the total number of passengers is our `Accuracy`.

Great! We are already correct in half of the cases (more or less). This is not a surprise when guessing one out of two possible labels.

But maybe we can do even better? I mean without any effort. We know that more people died than survived. What if we consistently predicted the death of a passenger?

Listing 2.18: Always predict a passenger died

```
1  def predict_death(item):
2    return 0
3
4  print(evaluate(run(predict_death, train_input), train_labels))
```

```
436 correct predictions out of 711. Accuracy 61 %
```

We're up to an accuracy of 61% of our predictions. Not too bad. This value that is the ratio between the two possible actual values, is the prevalence.

Let's consider a different task for a moment. Let's say you're a doctor, and your job is to predict whether a patient has cancer. Only 1% of your patients have cancer. If you expected no cancer all the time, your accuracy would be astonishing 99%! But you would falsely diagnose the patients that have cancer. And for the resulting lack of treatment, they're going to die.

Maybe the accuracy of the predictions alone is not a good measure to evaluate the performance of our algorithm.

2.6 Classifier Evaluation and Measures

As we mentioned in section 1.4.3, the evaluation is one central part of every machine learning algorithm. It may seem trivial at first sight. Yet, deciding on the right measure is a crucial step. When you optimize your algorithm towards better performance, you will inevitably optimize the scores in your evaluation function.

We will get to know more sophisticated evaluation functions in this book. But right now, we keep it simple. For example, a better way to evaluate the performance of a classifier is to look at the confusion matrix.

Figure 2.2: Confusion matrix
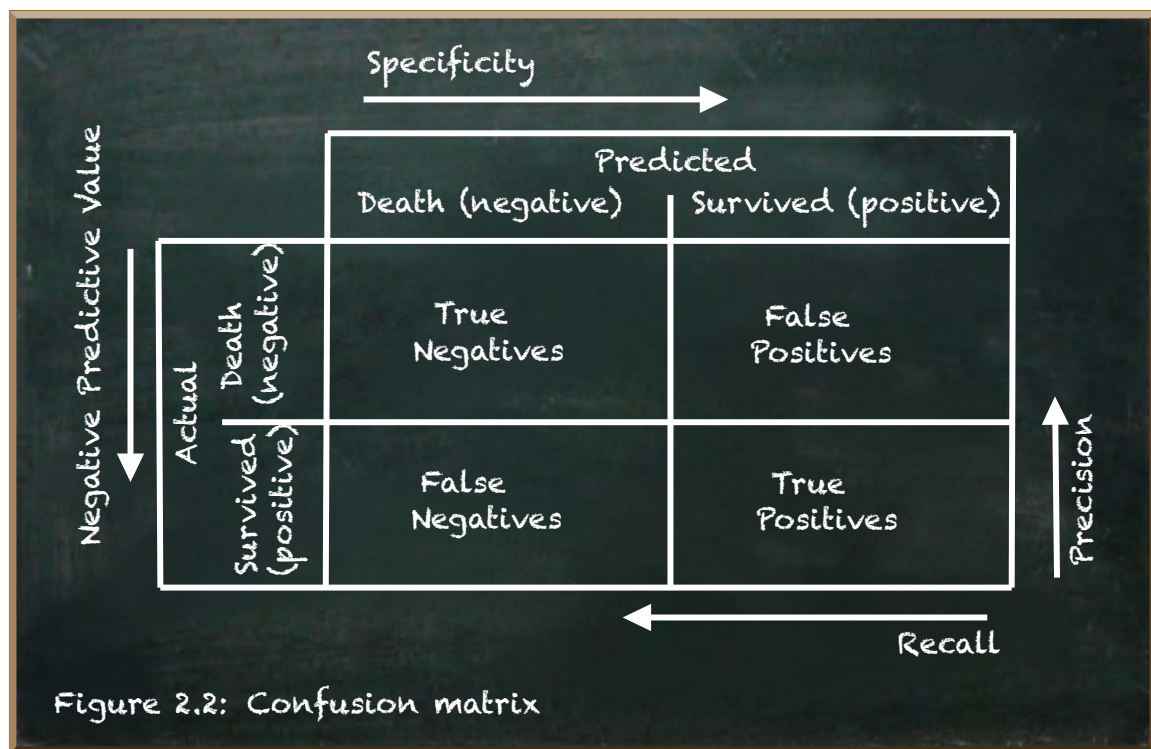
The general idea is to compare the predictions with the actual values. So, for example, in binary classification, there are two possible actual values: `true` or `false`. And there are two possible predictions: `true` or `false`.

There are four possibilities:

- **True Negatives** (TN): a passenger who died was correctly predicted
- **False Positives** (FP): a passenger who died was wrongly predicted to survive
- **False Negatives** (FN): a passenger who survived was wrongly predicted to die
- **True Positive** (TP): a passenger who survived was correctly predicted

Let's have a look at the confusion matrix of the `predict_death` classifier.

Listing 2.19: Confustion matrix of the predict death classifier

```
1  from sklearn.metrics import confusion_matrix
2
3  predictions = run(predict_death, train_input)
4  confusion_matrix(train_labels, predictions)
```

```
array([[436,    0],
       [275,    0]])
```

Scikit-Learn provides the `confusion_matrix` method that we import (line 1). It takes the actual values as first and the predictions as the second parameter (line 3).

It returns a two-dimensional array. The first row shows the true negatives (TN) and the false positives (FP). And, the second row shows the false negatives (FN) and the true positives (TP).

We can define the accuracy we measured thus far as:

$$Accuracy = \frac{\sum TruePositives + \sum TrueNegatives}{TotalPopulation} \qquad (2.1)$$

It does not care whether there is a systematic error, such as the algorithm's inability to correctly predict a passenger who survived (true positives), as long as it performs well at correctly predicting passengers who die (true negatives).

The confusion matrix offers us more detailed measures of our classifier performance. These are:

- precision
- recall
- specificity
- negative predictive value (NPV)

The precision is the "accuracy of the positive predictions." It only looks at the positive predictions. These are predictions that the passenger survived.

$$Precision = \frac{\sum TruePositives}{\sum AllPredictedPositives} \qquad (2.2)$$

Let's have a look at the code:

Listing 2.20: The precision score

```
1  from sklearn.metrics import precision_score
2  print('The precision score of the predict_death classifier is {}'
3    .format(precision_score(train_labels, predictions)))
```

```
  The precision score of the predict_death classifier is 0.0
```

Scikit-Learn provides a function to calculate the `precision_score`. It takes the list of actual values and the list of predicted values as input.

Since we did not have a single positive prediction, our precision is not defined. *Scikit-Learn* interprets this as a score of `0.0`.

The recall is the "accuracy of the actual positives." It only looks at the actual positives.

$$Recall = \frac{\sum TruePositives}{\sum AllActualPositives} \tag{2.3}$$

In Python, it is:

Listing 2.21: The recall score

```
1  from sklearn.metrics import recall_score
2  print('The recall score of the predict_death classifier is {}'
3    .format(recall_score(train_labels, predictions)))
```

```
  The recall score of the predict_death classifier is 0.0
```

Even though recall is defined (the number of actual positives is greater than 0), the score is 0.0 again because our classifier did not predict a single survival correctly. It is not a surprise when it always predicts death.

The specificity is the "accuracy of the actual negatives." It only looks at actual negatives (deaths).

$$Specificity = \frac{\sum TrueNegatives}{\sum AllActualNegatives} \tag{2.4}$$

And the "negative predictive value" (NPV) is the "accuracy of the negative predictions."

$$NegativePredictiveValue(NPV) = \frac{\sum TrueNegatives}{\sum AllPredictedNegatives} \tag{2.5}$$

These two functions are not provided out of the box. But with the values we get from the confusion matrix, we can calculate them easily:

Listing 2.22: The specificity and the npv

```
 1  def specificity(matrix):
 2    return matrix[0][0]/(matrix[0][0]+matrix[0][1]) if (matrix[0][0]+matrix
      [0][1] > 0) else 0
 3
 4  def npv(matrix):
 5    return matrix[0][0]/(matrix[0][0]+matrix[1][0]) if (matrix[0][0]+matrix
      [1][0] > 0) else 0
 6
 7  cm = confusion_matrix(train_labels, predictions)
 8
 9  print('The specificity score of the predict_death classifier is {:.2f}'.
      format(specificity(cm)))
10  print('The npv score of the predict_death classifier is {:.2f}'.format(
      npv(cm)))
```

```
The specificity score of the predict_death classifier is 1.00
The npv score of the predict_death classifier is 0.61
```

The function `specificity` takes the confusion matrix as a parameter (line 1). It divides the true negatives (`matrix[0][0]`) by the sum of the true negatives and the false positives (`matrix[0][1]`) (line 2).

The function `npv` takes the confusion matrix as a parameter (line 4) and divides the true negatives by the sum of the true negatives and the false negatives (`matrix[1][0]`).

These four scores provide a more detailed view of the performance of our classifiers.

Let's calculate these scores for our random classifier as well:

Listing 2.23: The scores of the random classifier

```
1  random_predictions = run(classify, train_input)
2  random_cm = confusion_matrix(train_labels, random_predictions)
3
4  print('The precision score of the random classifier is {:.2f}'
5    .format(precision_score(train_labels, random_predictions)))
6  print('The recall score of the random classifier is {:.2f}'
7    .format(recall_score(train_labels, random_predictions)))
8  print('The specificity score of the random classifier is {:.2f}'
9    .format(specificity(random_cm)))
10 print('The npv score of the random classifier is {:.2f}'
11   .format(npv(random_cm)))
```

```
The precision score of the random classifier is 0.38
The recall score of the random classifier is 0.49
The specificity score of the random classifier is 0.50
The npv score of the random classifier is 0.61
```

While the `predict_death` classifier exhibits a complete absence of precision and recall, it has excellent specificity. It reaches an NPV score that matches the percentage of negatives in our test dataset (the prevalence).

The random classifier produces balanced scores. You'll get a little bit different scores every time you run the classifier. But the values seem to stay in certain ranges. While the precision of this classifier is usually below `0.4` the npv is above `0.6`.

The confusion matrix and related measures give you much information. But sometimes, you need a more concise metric. For example, the evaluation function in a machine learning algorithm must return a single measure to optimize.

And this single measure should unmask a classifier that does not add any value.

2.7 Unmask the Hypocrite Classifier

Even though the `predict_death` classifier does not add any insight, it outperforms the random classifier concerning overall accuracy. This is because it

exploits the prevalence, the ratio between the two possible values, not being 0.5.

The confusion matrix reveals more details on certain areas. For example, it shows that the `predict_death` classifier lacks any recall and predicts actual positives. This is no surprise since it always predicts death.

But having a whole set of metrics makes it difficult to measure real progress. How do we recognize that one classifier is better than another? How do we even identify a classifier that adds no value at all? How do we identify such a hypocrite classifier?

Let's write a generalized hypocrite classifier and see how we can unmask it.

Listing 2.24: A hypocrite classifier

```
1  def hypocrite(passenger, weight):
2      return round(min(1,max(0,weight*0.5+random.uniform(0, 1))))
```

The `hypocrite` classifier takes the `passenger` data and a `weight` value. The `weight` is a number between −1 and 1. It denotes the classifier's tendency to predict death (negative values) or survival (positive values).

The formula `weight*0.5+random.uniform(0, 1)` generates numbers between −0.5 and 1.5. The `min` and `max` functions ensure the result to be between 0 and 1. The `round` function returns either 0 (death) or 1 (survival).

Depending on the weight, the chances to return one or the other prediction differs.

If `weight` is −1, it returns `-1*0.5+random.uniform(0, 1)`, a number between −0.5 and 0.5. A number almost always rounding to 0 (predicted death).

If `weight` is 0, the formula returns `-1*0+random.uniform(0, 1)`. This is our random classifier.

If `weight` is 1, it returns `1*0.5+random.uniform(0, 1)`, a number that is always greater than 0.5 and thus, rounding to 1(predicted survival).

We can choose the tendency from −1 to 1. −1 always predicts death, 0 is entirely random, 1 always predicts survival.

Let's have a look at how the predictions vary. We pass the weight as a hyperparameter. Try different values, if you like.

Listing 2.25: The scores of the hypocrite classifier

```
 1  w_predictions = run(lambda passenger: hypocrite(passenger, -0.5),
        train_input)
 2  w_cm = confusion_matrix(train_labels, w_predictions)
 3
 4  print('The precision score of the hypocrite classifier is {:.2f}'
 5      .format(precision_score(train_labels, w_predictions)))
 6  print('The recall score of the hypocrite classifier is {:.2f}'
 7      .format(recall_score(train_labels, w_predictions)))
 8  print('The specificity score of the hypocrite classifier is {:.2f}'
 9      .format(specificity(w_cm)))
10  print('The npv score of the hypocrite classifier is {:.2f}'
11      .format(npv(w_cm)))
```

```
The precision score of the hypocrite classifier is 0.38
The recall score of the hypocrite classifier is 0.22
The specificity score of the hypocrite classifier is 0.77
The npv score of the hypocrite classifier is 0.61
```

If you run the `hypocrite` classifier a few times, you may get a feeling for its performance. But let's create a visualization of it.

The following code runs the hypocrite classifier for different values of `weight`.

The range of allowed `weight`s is between -1 and 1. We divide this range into 40 (`cnt_steps`) steps (line 4). We create lists of the indices (`steps=[0, 1, ..., 38, 39]`, line 7) and of the `weights` at every step (`weights=[-1, -0.95, ... 0.9, 0.95, 1.0]`, lines 10-13).

We run the `hypocrite` classifier for every step (lines 17-19) and put the results into `l_predictions` (line 16). Based on the predictions and the actual results, we calculate the confusion matrix for every step (line 26) and store them in `l_cm` (line 25).

Listing 2.26: Run the hypocrite classifiers

```python
import numpy as np

# number of steps to consider between −1 and 1
cnt_steps = 40

# a list of the step numbers [0, 1, ..., 38, 39]
steps = np.arange(0, cnt_steps, 1).tolist()

# list of the weights at every step [−1, −0.95, ... 0.9, 0.95, 1.0]
weights = list(map(
  lambda weight: round(weight, 2),
  np.arange(−1, 1+2/(cnt_steps−1), 2/(cnt_steps−1)).tolist()
))

# list of predictions at every step
l_predictions = list(map(
  lambda step: run(
    lambda passenger: hypocrite(passenger, weights[step]),
    train_input
  ),
  steps
))

# list of confusion matrices at every steo
l_cm = list(map(
  lambda step: confusion_matrix(train_labels, l_predictions[step]),
  steps
))
```

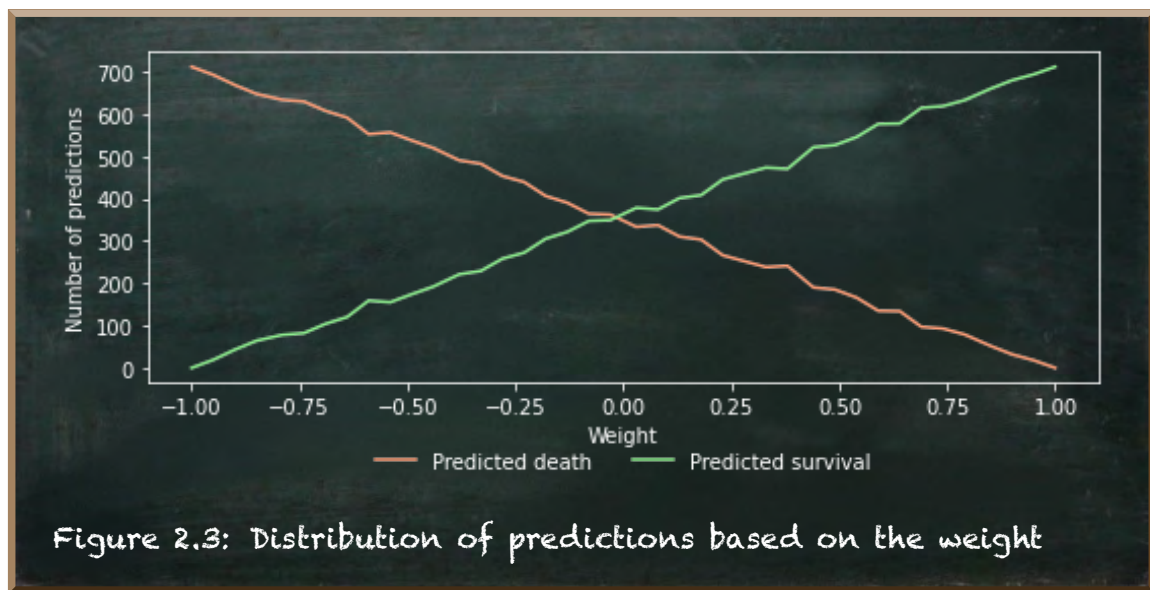The next piece of code takes care of rendering the two graphs.

The green graph depicts the number of predicted survivals at a step. The red graph shows the number of expected deaths.

Listing 2.27: Plot the distribution of predictions

```python
import matplotlib.pyplot as plt
import matplotlib

# create a graph for the number of predicted deaths
deaths, = plt.plot(
    weights, # point at x-axis
    list(map(lambda cur: l_cm[cur][0][0]+l_cm[cur][1][0], steps)),
    'lightsalmon', # color of the graph
    label='Predicted death'
)

# create a graph for the number of predicted survivals
survivals, = plt.plot(
    weights, # point at x-axis
    list(map(lambda cur: l_cm[cur][0][1]+l_cm[cur][1][1], steps)),
    'lightgreen', # color of the graph
    label='Predicted survival'
)

plt.legend(handles=[deaths, survivals],loc='upper center',
    bbox_to_anchor=(0.5, -0.15), framealpha=0.0, ncol=2)
plt.xlabel("Weight")
plt.ylabel("Number of predictions")
plt.show()
```


Figure 2.3: Distribution of predictions based on the weight

We can see that the `hypocrite` classifier generates the expected tendency in its predictions. At `weight=−1`, it always predicts death, at `weight=0` it is 50:50, and at `weight=1` it always predicts survival.

Let's see how the different `hypocrite` classifiers perform at the four metrics depending on the `weight`.
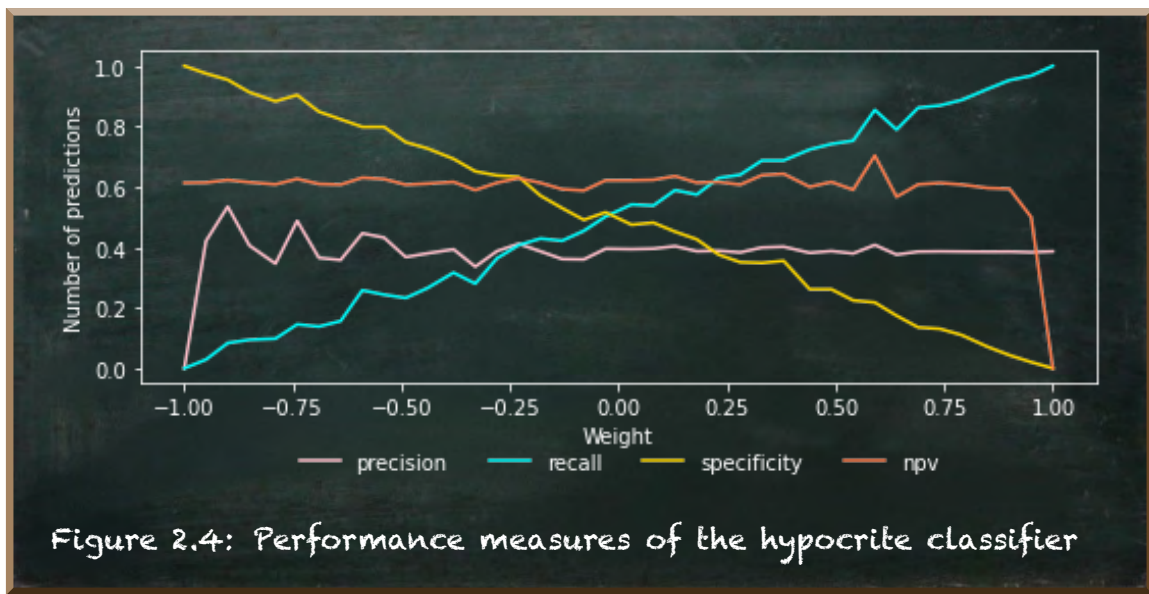
Listing 2.28: Metrics of the hypocrite classifier

```python
l_precision = list(map(lambda step: precision_score(train_labels,
    l_predictions[step]),steps))
l_recall = list(map(lambda step: recall_score(train_labels, l_predictions
    [step]),steps))
l_specificity = list(map(lambda step: specificity(l_cm[step]),steps))
l_npv = list(map(lambda step: npv(l_cm[step]),steps))
```

In these four lines, we calculate the four metrics at each step. Let's visualize them.

Listing 2.29: Plot the performance measures

```python
m_precision, = plt.plot(weights, l_precision, 'pink', label="precision")
m_recall, = plt.plot(weights, l_recall, 'cyan', label="recall")
m_specificity, = plt.plot(weights, l_specificity, 'gold', label="
    specificity")
m_npv, = plt.plot(weights, l_npv, 'coral', label="npv")

plt.legend(
  handles=[m_precision, m_recall, m_specificity, m_npv],
  loc='upper center',
  bbox_to_anchor=(0.5, −0.15),
  framealpha=0.0,
  ncol=4)

plt.xlabel("Weight")
plt.ylabel("Number of predictions")
plt.show()
```

Figure 2.4: Performance measures of the hypocrite classifier

These graphs show some exciting characteristics. `specificity` and `recall` are directly related to the classifier's tendency to predict death (higher `specificity`) or to predict survival (higher `recall`).

Except for the edge cases where all predictions are death, or all are survival, the values for `precision` and `npv` seem to be horizontal lines. `precision` relates to the prevalence of 39% survivals in our data and `npv` to the prevalence of 61% deaths.

Listing 2.30: Calculating the mean of the measures

```
1  l_mean = list(map(lambda step: sum(step)*0.25, zip(l_precision, l_recall,
        l_specificity, l_npv)))
2  m_mean, = plt.plot(weights, l_mean, 'pink', label="Mean of the measures")
3
4  plt.legend(handles=[m_mean],loc='upper center',
5    bbox_to_anchor=(0.5, -0.15),framealpha=0.0)
6  plt.ylim(0, 1)
7  plt.show()
```

Figure 2.5: The mean of the measures discloses the absence of any information gain
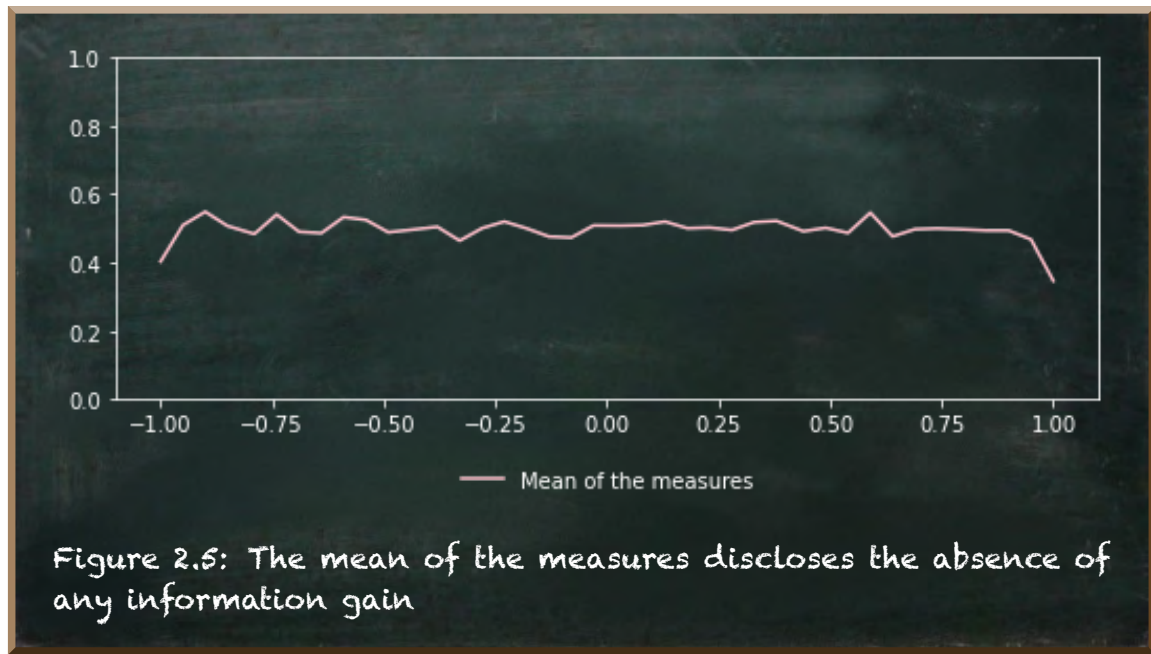
When looking at the mean of all four measures, we see an almost flat line. Its drops at the edges are due to `precision` and `npv` being `0` there because there are no predicted survivals (left edge) respectively no predicted deaths (right edge) to calculate some measures.

This line indicates that the overall level of information provided by all hypocrite classifiers is equal. And the level is about 0.5. That is the baseline for a binary classifier, for there are only two possible outcomes.

Even though specific types of `hypocrite` classifiers are able to trick a single measure (like `accuracy`, `recall`, `precision`, or `npv`) by exploiting the prevalence, when looking at all complementary measures at once, we can unmask the `hypocrite` classifier.

However, this does not imply that the mean of these measures is the best measure to evaluate the performance of your classifier with. Depending on your task at hand, you may, for instance, favor `precision` over `recall`. Rather, the implication is that you should look at the overall level of information provided by the classifier, too. You should not let yourself be dazzled by the classifier's performance at a single measure.

Finally, let's create a reusable function that calculates the measures for us and displays the results.

Listing 2.31: A reusable function to unmask the hypocrite classifier

```
 1 def classifier_report(name, run, classify, input, labels):
 2   cr_predictions = run(classify, input)
 3   cr_cm = confusion_matrix(labels, cr_predictions)
 4
 5   cr_precision = precision_score(labels, cr_predictions)
 6   cr_recall = recall_score(labels, cr_predictions)
 7   cr_specificity = specificity(cr_cm)
 8   cr_npv = npv(cr_cm)
 9   cr_level = 0.25*(cr_precision + cr_recall + cr_specificity + cr_npv)
10
11   print('The precision score of the {} classifier is {:.2f}'
12     .format(name, cr_precision))
13   print('The recall score of the {} classifier is {:.2f}'
14     .format(name, cr_recall))
15   print('The specificity score of the {} classifier is {:.2f}'
16     .format(name, cr_specificity))
17   print('The npv score of the {} classifier is {:.2f}'
18     .format(name, cr_npv))
19   print('The information level is: {:.2f}'
20     .format(cr_level))
```

Let's use this function to get a report of our random classifier.

Listing 2.32: The report of the random classifier

```
1 classifier_report(
2   "Random PQC",
3   run,
4   classify,
5   train_input,
6   train_labels)
```

```
The precision score of the Random PQC classifier is 0.38
The recall score of the Random PQC classifier is 0.51
The specificity score of the Random PQC classifier is 0.47
The npv score of the Random PQC classifier is 0.61
The information level is: 0.49
```

3. Qubit and Quantum States

In this chapter, we start with the very basics of quantum computing–the quantum bit. And we will write our first quantum circuit. A quantum circuit is a sequence of quantum bit transformations–the quantum program. Let's start with the basics.

3.1 Exploring the Quantum States

The world of quantum mechanics is different. A quantum system can be in a state of superposition. A popular notion of superposition is that the system is in different states concurrently unless you measure it.

For instance, the spin of a particle is not up **or** down, but it is up **and** down at the same time. But when you look at it, you find it either up **or** down.

Or, let's say you flip a quantum coin. In the air, it has both values, heads **and** tails. If and only if you catch it and look at it, it decides for a value. Once landed, it is a normal coin with heads up **or** tails up.

Another notion of superposition is that the system is truly random and therefore distinguishes it from the systems we know. Tossing a (normal) coin, for instance, seems random because whenever you do it, the conditions are slightly different. And even tiny differences can change the outcome from heads to tails. The coin is *sensitive dependent to initial conditions*.

If we were able to measure all conditions precisely, we could tell the outcome. In classical mechanics, there is no randomness. Things in our every-

day world, such as the coin, seem random. But they are not. If measured with infinite precision, randomness would disappear. By contrast, a quantum system is truly random.

Maybe you wonder: Ok, it's *random*. *Where's the big deal*?

The big thing is the consequences. In a classic system, a system *sensitive dependent to initial conditions*, the answer to a question is already determined before we ask it.

Rather than watching the baseball match tonight, you spend the evening with your friends. When you return home, even though you don't know the results, the match is over, and there is a definite result. There could be different results, but you simply don't know the result until you look at it.

Contrarily, in a quantum system, the answer to a question is not determined up until the time you ask it. And since it is not determined yet, you still can change the probabilities of measuring distinct states.

Do you have doubts? Good! Not even Einstein liked this notion. It led him to his famous statement of *God does not play dice*.
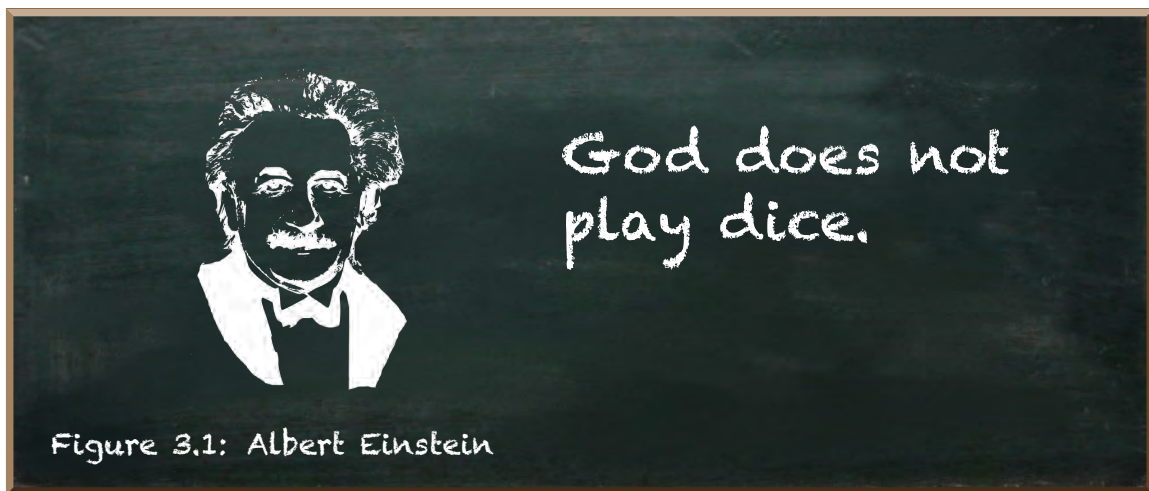
Figure 3.1: Albert Einstein

Many physicists, including Einstein, proposed the quantum state, though hidden, to be a well-defined state. This is known as the *hidden variable theory*.

There are statistically distinct behaviors between a system following the hidden variable theory and a quantum system following the superposition principle. And experiments showed that the quantum mechanical predictions were correct.

For now, let's accept the quantum state is something different. Later in this

book, we will have a closer look at it. And its consequences. But this requires a little more theory and math.

We turn to the quantum computer. Let's say you have a quantum bit. We call it qubit. Unless you observe its value, it is in a superposition state of 0 and 1. Once you observe its value, you'll get 0 or 1.

The chances of a qubit to result in either one value don't need to be 50:50. It can be 25:75, 67:33, or even 100:0. It can be any weighted probability distribution.

The probability distribution a qubit has when observed depends on its state. The quantum state.

In quantum mechanics, we use vectors to describe the quantum state. A popular way of representing quantum state vectors is the **Dirac** notation's "ket"-construct that looks like $|\psi\rangle$. In Python, we don't have vectors. But we have arrays. Luckily, their structures are similar.

Let's have a look. We start with the simplest case. Let's say we have a qubit that, when observed, always has the value 0. If you argued this qubit must have the value 0 even before it is observed, you wouldn't be completely wrong. Yet, you'd be imprecise. Before it is observed, this qubit has the probability of $1 (= 100\%)$ to have the value 0 when observed.

These are the equivalent representations (ket, vector, array) of a qubit that always results in 0 when observed:

$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and in Python `[1, 0]`.

Accordingly, the following representations depict a qubit that always results in 1 when observed:

$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and in Python `[0, 1]`.

Ok, enough with the theory for now. Let's have a look at the code of such a qubit.

If you haven't configured your workstation yet, have a look at the brief explanation of how to set up the working environment (section 1.8).

Now, open the *Jupyter* notebook and test whether *Qiskit* works.

Listing 3.1: Verify Qiskit version

```
1  import qiskit
2  qiskit.__qiskit_version__
```

```
{'qiskit-terra': '0.16.4',
 'qiskit-aer': '0.7.4',
 'qiskit-ignis': '0.5.2',
 'qiskit-ibmq-provider': '0.11.1',
 'qiskit-aqua': '0.8.2',
 'qiskit': '0.23.5'}
```

If you get a response like this, *Qiskit* works. Great! We're ready to create our first qubit.

Listing 3.2: The first qubit

```
1  from qiskit import QuantumCircuit
2
3  # Create a quantum circuit with one qubit
4  qc = QuantumCircuit(1)
5
6  # Define initial_state as |1>
7  initial_state = [0,1]
8
9  # Apply initialization operation to the qubit at position 0
10 qc.initialize(initial_state, 0)
```

The fundamental unit of *Qiskit* is the quantum circuit. A quantum circuit is a model for quantum computation. The program, if you will. Our circuit consists of a single qubit (line 4).

We define `[0,1]` as the `initial_state` of our qubit (line 7) and initialize the first and only qubit (at position `0` of the array) of our quantum circuit with it (line 10).

Remember `[0,1]`? This is the equivalent to $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. And in plain English, it is a qubit resulting in the value 1 when observed.

This is it. It's now time to boot our quantum computer. In case you don't

have one, no problem. We can simulate it. (In case you have one: *"Cool, let me know"*).

Listing 3.3: Prepare the simulation backend

```
1  from qiskit import execute, Aer
2
3  # Tell Qiskit how to simulate our circuit
4  backend = Aer.get_backend('statevector_simulator')
5
6  # Do the simulation, returning the result
7  result = execute(qc,backend).result()
```

Qiskit provides the `Aer` package (that we import at line 1). It provides different backends for simulating quantum circuits. The most common backend is the `statevector_simulator` (line 4).

The `execute` function (that we import at line 1, too) runs our quantum circuit (`qc`) at the specified `backend`. It returns a `job` object that has a useful method `job.result()`. This returns the `result` object once our program completes it.

Let's have a look at our qubit in action.

Qiskit uses Matplotlib to provide useful visualizations. A simple histogram will do. The `result` object provides the `get_counts` method to obtain the histogram data of an executed circuit (line 5).

The method `plot_histogram` returns a *Matplotlib* figure that Jupyter draws automatically (line 8).

We see we have a 100% chance of observing the value 1.

Listing 3.4: The measured qubit

```
1  from qiskit.visualization import plot_histogram
2  import matplotlib.pyplot as plt
3
4  # get the probability distribution
5  counts = result.get_counts()
6
7  # Show the histogram
8  plot_histogram(counts)
```

Figure 3.2: The qubit state
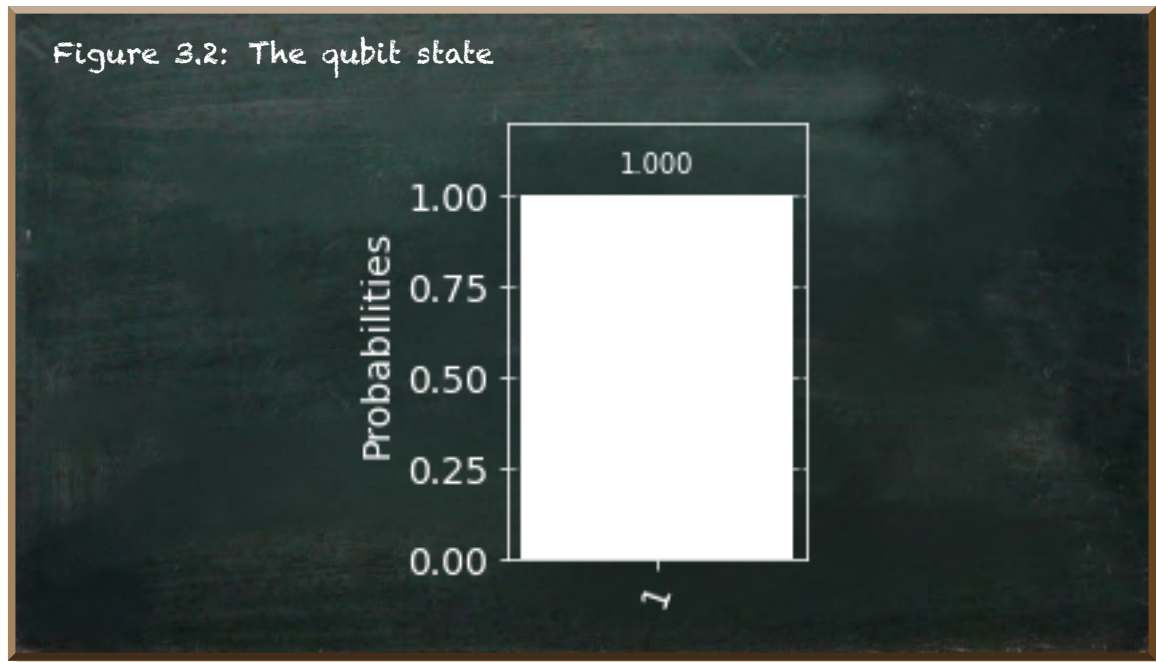
Now, let's move on to a more advanced case. Say, we want our qubit to result in either 0 or 1 with the same probability (50:50).

In quantum mechanics, there is the fundamental principle superposition. It says any two (or more) quantum states can be added together ("superposed"), and the result will be another valid quantum state.

Wait! We already know two quantum states, $|0\rangle$ and $|1\rangle$. Why don't we add them? $|0\rangle$ and $|1\rangle$ are vectors. Adding two vectors is straightforward.

A vector is a geometric object that has a magnitude (or length) and a direction. Usually, they are represented by straight arrows, starting at one point on a coordinate axis and ending at a different point.

You can add two vectors by placing one vector with its tail at the other vector's head. The straight line between the yet unconnected tail and the yet unconnected head is the sum of both vectors. Have a look at the figure 3.3.

Mathematically, it is as easy.

Let $\vec{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ and $\vec{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$ be two vectors.

The sum of $\vec{u}$ and $\vec{v}$ is:

$$\vec{u} + \vec{v} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \end{bmatrix} \tag{3.1}$$

Figure 3.3: Adding two vectors
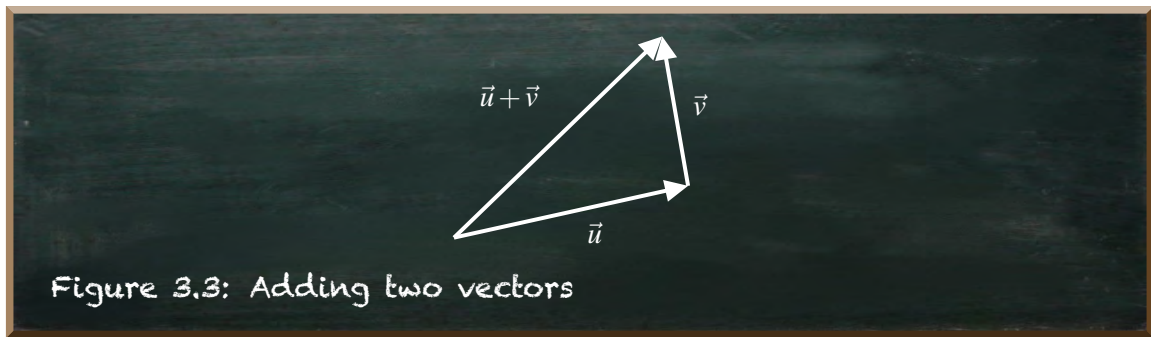
Accordingly, our superposed state should be ψ^*:

$$|\psi\rangle = \underbrace{|0\rangle + |1\rangle}_{superposition} = \begin{bmatrix} 1+0 \\ 0+1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad (3.2)$$

$^*\psi$ ("psi") is a common symbol used for the state of a quantum system.

We have a computer in our hands. Why don't we try it?

Listing 3.5: First attempt to superpose two states

```
1  # Define state |psi>
2  initial_state = [1, 1]
3
4  # Redefine the quantum circuit
5  qc = QuantumCircuit(1)
6
7  # Initialise the 0th qubit in the state `initial_state`
8  qc.initialize(initial_state, 0)
9
10 # execute the qc
11 results = execute(qc,backend).result().get_counts()
12
13 # plot the results
14 plot_histogram(results)
```

```
QiskitError: 'Sum of amplitudes-squared does not equal one.'
```

It didn't quite work. It tells us: `QiskitError: 'Sum of amplitudes−squared does not equal one.'`.

The amplitudes are the values in our array. They are proportional to probabilities. And all the probabilities should add up to exactly 1 (100%). We need to add weights to the quantum states $|0\rangle$ and $|1\rangle$. Let's call them α and β.

We weight $|0\rangle$ with α and $|1\rangle$ with β. Like this:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} 1\cdot\alpha + 0\cdot\beta \\ 0\cdot\alpha + 1\cdot\beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Amplitudes are proportional to probabilities. We need to normalize them so that $\alpha^2 + \beta^2 = 1$. If both states $|0\rangle$ and $|1\rangle$ should have the same weight, then $\alpha = \beta$. And therefore, we can solve our equation to α:

$$\alpha^2 + \alpha^2 = 1 \Leftrightarrow 2\cdot\alpha^2 = 1 \Leftrightarrow \alpha^2 = \frac{1}{2} \Leftrightarrow \alpha = \frac{1}{\sqrt{2}}$$

And we insert the value for both α and β (both are equal). Let's try this quantum state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

The corresponding array in Python is: `[1/sqrt(2), 1/sqrt(2)]`. Don't forget to import `sqrt`.

Listing 3.6: Weighted initial state

```python
from math import sqrt

# Define state |psi>
initial_state = [1/sqrt(2), 1/sqrt(2)]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```

Figure 3.4: The qubit state
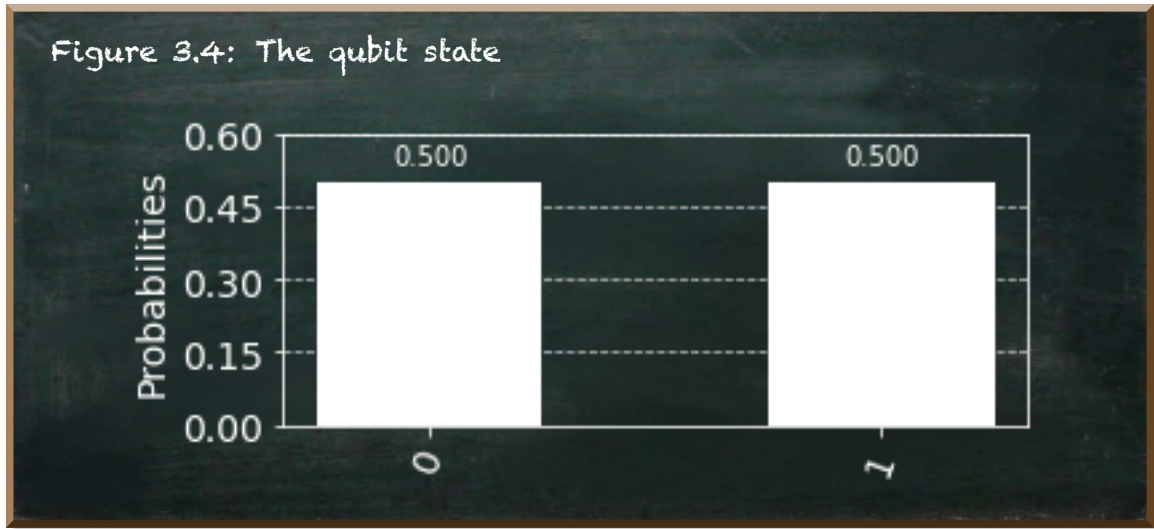
What is the state of qubit that has a 25% chance of resulting in 0 and 75% of resulting in 1?

The solution is solving the following equation system.

Equation 3.3. This is the definition of a qubit in superposition. This qubit, when observed, has the probability of α^2 to result in 0 and β^2 to result in 1.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{3.3}$$

Equation 3.4. This is the required normalization. It requires the sum of the squared amplitudes (α and β) to equal 1.

$$\alpha^2 + \beta^2 = 1 \tag{3.4}$$

Let's regard the probabilities 25% and 75% as fractions and equate them to α^2 and β^2, respectively.

$$\alpha^2 = \frac{1}{4} \Leftrightarrow \alpha = \frac{1}{2} \tag{3.5}$$

and

$$\beta^2 = \frac{3}{4} \Leftrightarrow \beta = \frac{\sqrt{3}}{2} \tag{3.6}$$

Now, we insert 3.5 and 3.6 into equation 3.3:

$$|\psi\rangle = \frac{1}{2}|0\rangle + \frac{\sqrt{3}}{2}|1\rangle = \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix}$$

In Python, the array `[1/2, sqrt(3)/2]` represents the vector $\begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix}$

Now, let's open our Jupyter notebook and test our calculation.

Listing 3.7: The qubit with a probability of 0.25 to result in 0

```python
from qiskit import QuantumCircuit, execute, Aer
from qiskit.visualization import plot_histogram
from math import sqrt

qc = QuantumCircuit(1)
initial_state = [1/2, sqrt(3)/2] # Here, we insert the state
qc.initialize(initial_state, 0)
backend = Aer.get_backend('statevector_simulator')
result = execute(qc,backend).result()
counts = result.get_counts()
plot_histogram(counts)
```


Figure 3.5: The qubit measurement probabilities
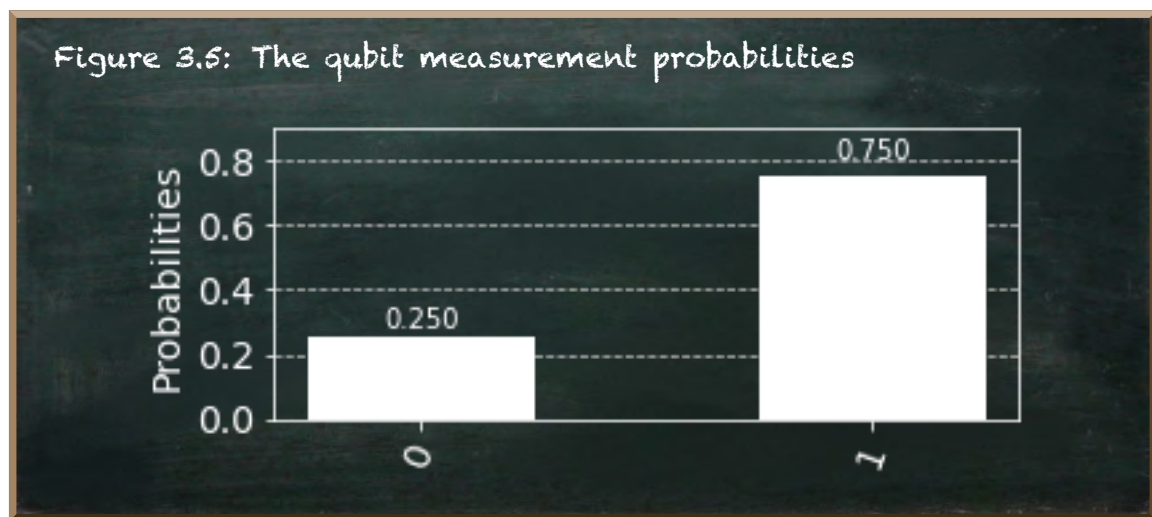
Phew. In this chapter, we introduced quite a few terms and equations just to scratch on the surface of quantum mechanics. But the actual source code is pretty neat, isn't it?

We introduced the notion of the quantum state. In particular, the state of a binary quantum system. The quantum bit or qubit.

Until we observe a qubit, it is in superposition. Contrary to a classical bit that can be either 0 or 1, a qubit is in a superposition of both states. But once you observe it, there are distinct probabilities of measuring 0 or 1.

This means that multiple measurements made on multiple qubits in identical states will not always give the same result. The equivalent representations of a quantum bit that, when observed, has the probability of α^2 to result in 0 and β^2 to result in 1 are:

$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$, with $\alpha^2 + \beta^2 = 1$. In Python, the array `[alpha, beta]` denotes this state.

3.2 Visual Exploration Of The Qubit State

The qubit is a two-dimensional quantum system. Each dimension is denoted by a standard basis vector:

$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ in Python `[1, 0]` and

$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, in Python `[0, 1]`.

The state of the qubit is represented by the superposition of both dimensions. This is the qubit state vector $|\psi\rangle$ ("psi").

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{3.7}$$

In Python, $|\psi\rangle$ is the array `[alpha, beta]`.

But $|\psi\rangle$ must be normalized by:

$$\alpha^2 + \beta^2 = 1 \tag{3.8}$$

Although normalizing the qubit state vector is not a difficult task, doing the math over and over again is quite cumbersome.

But maybe, there's another way, an easy way. Let's first have a look at a graphical representation of the qubit state $|\psi\rangle$ in the following figure 3.6.

In this representation, both dimensions reside at the vertical axis but in opposite directions. The top and the bottom of the system correspond to the standard basis vectors $|0\rangle$ and $|1\rangle$, respectively.

Figure 3.6: 2-dimensional qubit system
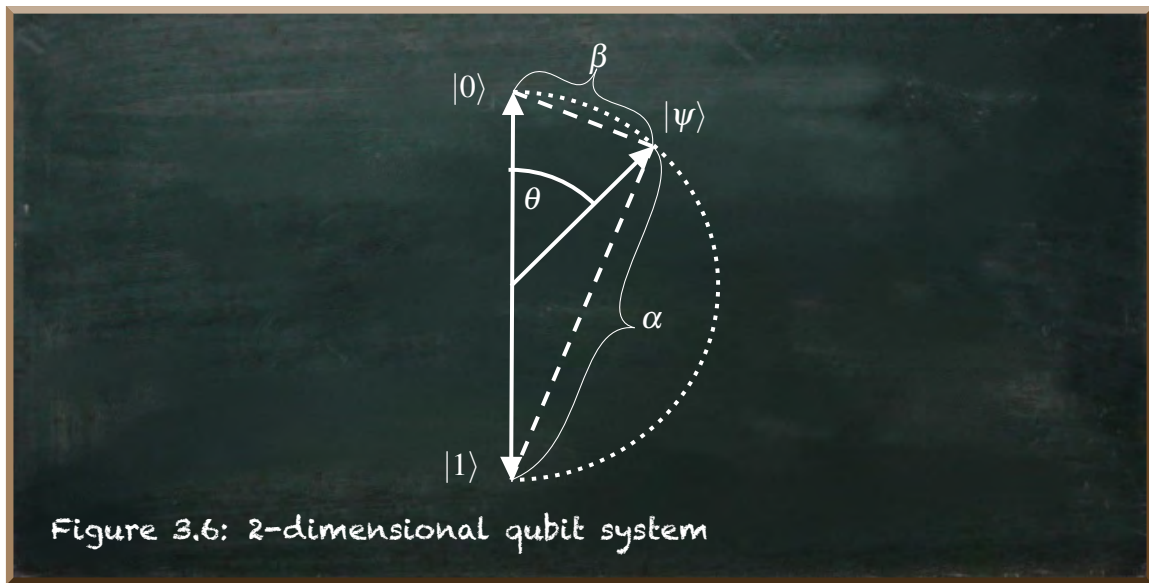
> (i) When there are two dimensions, the usual way is to put the two dimensions orthogonal to each other. While using one axis to represent both dimensions is rather an unusual representation for a two-dimensional system, it is well suited for a quantum system. But more on this later.

Let's have a look at the arbitrary qubit state vector $|\psi\rangle$ in this figure 3.6.

Since qubit state vectors are normalized, $|\psi\rangle$ originates in the center and has the magnitude (length) of $\frac{1}{2}$. Due to this equal magnitude, all state vectors end at the pointed circle. So does $|\psi\rangle$.

The angle between the state vector $|0\rangle$ and $|\psi\rangle$, named θ ("theta"), controls the proximities of the vector head to the top and the bottom of the system (dashed lines).

These proximities represent the probabilities of

- α^2 of measuring $|\psi\rangle$ as 0
- and β^2 of measuring it as 1.

> (!) The proximities α and β are at the opposite sides of the state's probability ($|\psi\rangle$) they describe. α is the proximity (or distance) to $|1\rangle$ because with increasing distance to $|1\rangle$ the probability of

measuring 0 increases.

Thus, by controlling the proximities, the angle θ also controls the probabilities of measuring the qubit in either state 0 or 1.

Rather than specifying the relation between α and β and then coping with normalizing their values, we can specify the angle θ and use the required normalization to derive α and β from it.

3.3 Bypassing The Normalization

The angle θ controls the probabilities of measuring the qubit in either state 0 or 1. Therefore, θ also determines α and β.

Have a look at figure 3.6 again.

Any valid qubit state vector must be normalized:

$$\alpha^2 + \beta^2 = 1 \tag{3.9}$$

This implies all qubit state vectors have the same magnitude (length). Since they all originate in the center, they form a circle with a radius of their magnitude (that is half of the circle diameter).

In such a situation, **Thales' theorem** states, if

- A, B, and C are distinct points on a circle (condition 1)
- where the line AC is a diameter (condition 2)
- then the angle $\angle ABC$ (the angle at point B) is a right angle.

In our case, the heads of $|0\rangle$, $|1\rangle$, and $|\psi\rangle$ represent the points A, B, and C, respectively (satisfy condition 1). The line between $|0\rangle$ and $|1\rangle$ is the diameter (satisfy condition 2). Therefore, the angle at the head of $|\psi\rangle$ is a right angle.

Now, the **Pythagorean theorem** states the area of the square whose side is opposite the right angle (hypotenuse, c) is equal to the sum of the areas of the squares on the other two sides (legs a, b).

$$c^2 = a^2 + b^2 \tag{3.10}$$

When looking at figure 3.6, again, we can see that α and β are the two legs of the rectangular triangle and the diameter of the circle is the hypotenuse.

Therefore, we can insert the normalization equation 3.9

$$c = \sqrt{\alpha^2 + \beta^2} = \sqrt{1} = 1 \tag{3.11}$$

The diameter c is two times the radius, thus two times the magnitude of any vector $|\psi\rangle$. The length of $|\psi\rangle$ is thus $\frac{c}{2} = \frac{1}{2}$.

Since all qubit state vectors have the same length, including $|0\rangle$ and $|1\rangle$, there are two isosceles triangles ($\triangle M|0\rangle|\psi\rangle$ and $\triangle M|\psi\rangle|1\rangle$).

Have a look at the following figure 3.7.

Figure 3.7: Two inner isosceles triangles and an outer rectangular triangle
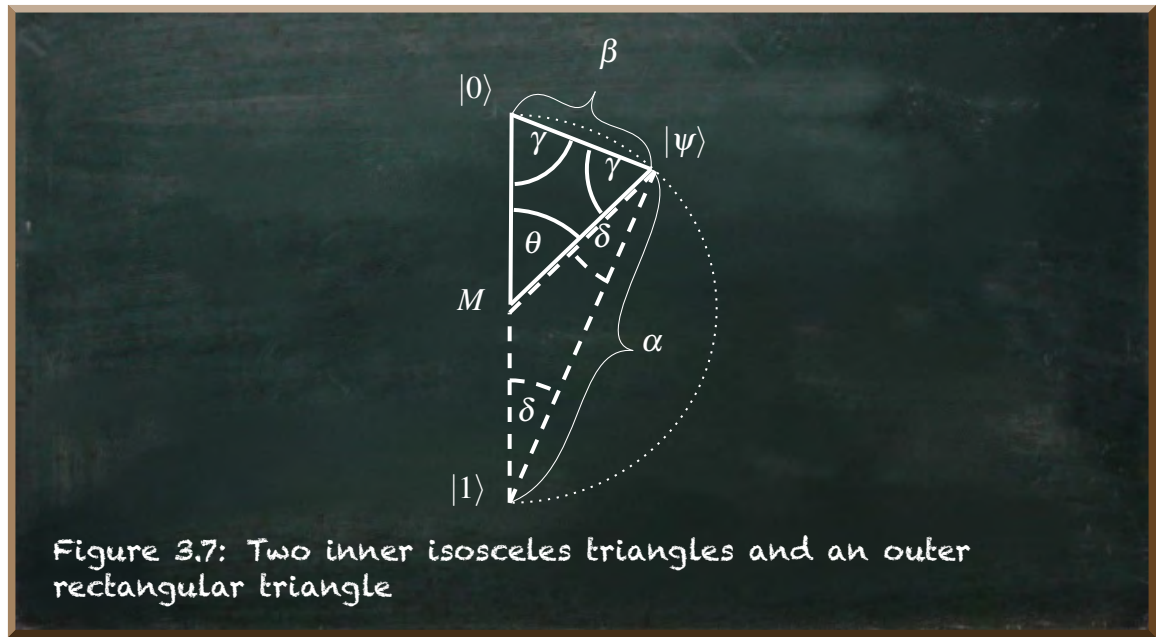
You can see the two isosceles triangles. The angles in isosceles triangles at the equal legs are equal, as denoted by γ and δ.

Further, the sum of all three angles in a triangle is 180°. Therefore,

$$\theta + 2\gamma = 180^o \tag{3.12}$$

Let's solve this after γ

$$\gamma = \frac{180^o - \theta}{2} = 90^o - \frac{\theta}{2} \tag{3.13}$$

In a rectangular triangle (the outer one), trigonometric identity says the sine of an angle is the length of the opposite leg divided by the length of the hypotenuse. In our case, this means:

$$\sin \gamma = \frac{\alpha}{1} = \alpha \tag{3.14}$$

Now, we insert equation 3.13:

$$\sin\left(90^o - \frac{\theta}{2}\right) = \alpha \tag{3.15}$$

With $sin(90^o - x) = \cos x$, we can see:

$$\alpha = \cos\frac{\theta}{2}$$

This is the first variable we aimed to calculate.

The further derivation works accordingly and is straightforward. At the center (M), the (unnamed) angle inside the dashed triangle is $180^o - \theta$.

$$(180^o - \theta) + 2\delta = 180^o \Leftrightarrow \delta = \frac{\theta}{2} \tag{3.16}$$

Again, we use the trigonometric identity. This time it implies:

$$\sin\delta = \frac{\beta}{1} = \beta \tag{3.17}$$

Finally, we insert 3.16:

$$\sin\frac{\theta}{2} = \beta \tag{3.18}$$

This is the second variable to calculate.

We calculated α and β. We can insert it into the definition of the qubit superposition.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{3.19}$$
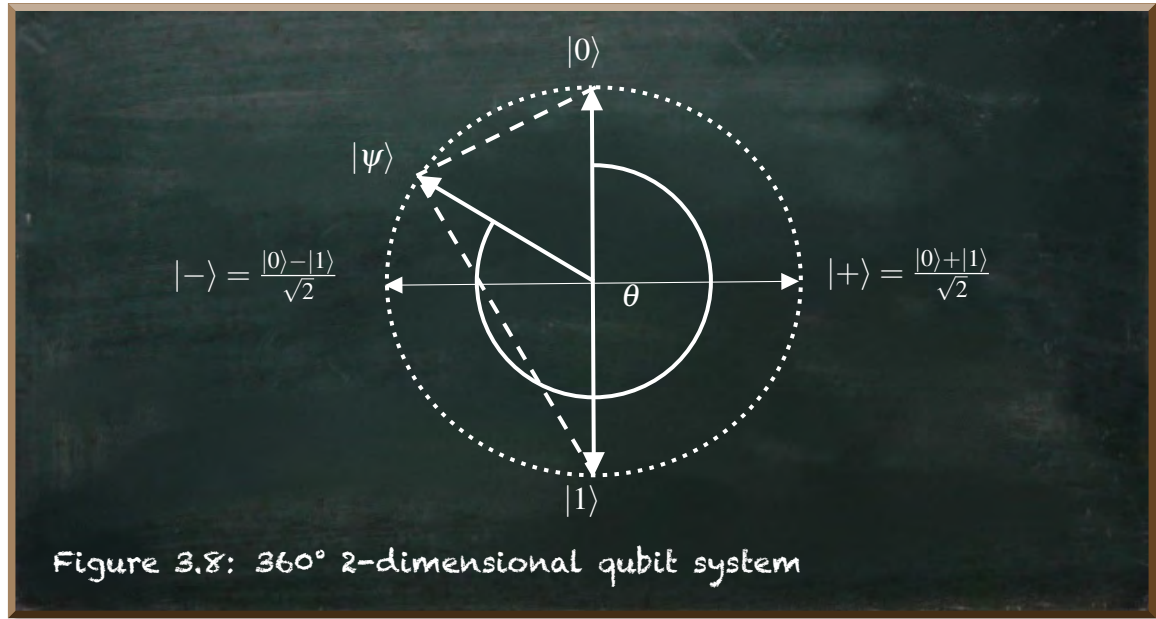
The result is

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix} \tag{3.20}$$

In Python the two-field array `[cos(theta/2), sin(theta/2)]` denotes this state.

α and β describe the proximity to the top and the bottom of the system, respectively. θ is the angle between the standard basis vector: $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and the qubit state vector $|\psi\rangle$ it represents.

There's one problem left. For $\theta \in \mathbb{R}$, what if $\pi < \theta < 2\pi$? Or in plain English, what if the θ denotes a vector pointing to the left side of the vertical axis?

Figure 3.8 shows this situation.

Figure 3.8: 360° 2-dimensional qubit system

Mathematically, we don't have a problem. Since we square α and β, their signs ($+$ or $-$) are irrelevant for the resulting probabilities.

But what does it mean? How can either α^2 or β^2 be negative, as the figure indicates? The answer is i. i is a complex number whose square is negative: $i^2 = -1$.

And if α and β are complex numbers ($\alpha, \beta \in \mathbb{C}$), their squares can be negative.

This entails a lot of consequences. And it raises a lot of questions. We will unravel them one by one in this book. For now, we interpret all vectors on the left-hand side of the vertical axis to have a negative value for β^2 ($\beta^2 < 0$).

While such a value lets us distinguish the qubit state vectors on both sides of the vertical axis, it does not matter for the resulting probabilities.

For instance, the state $|\psi\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$ yields the same probability of measuring 0 or 1. It resides on the horizontal axis. And so does $|\psi\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$

Although these states share the same probabilities, they are different. And the angle θ differentiates between them.

$\theta = \frac{\pi}{2}$ specifies $|\psi\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$ that is also known as $|+\rangle$.

And $\theta = \frac{3}{2}\pi$ or $\theta = -\frac{\pi}{2}$ specifies $|\psi\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$ that is also known as $|-\rangle$.

One of the consequences mentioned above of α^2 or β^2 being negative is that our normalization rule needs some adjustments.

We need to change the normalization equation 3.8 to:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{3.21}$$

This section contained a lot of formulae. The important takeaway is we can specify quantum states that yield certain probabilities of measuring 0 and 1 by an angle θ. It saves us from doing the normalization manually.

Let's have a look.

Listing 3.8: Using theta to specify the quantum state vector

```python
from math import pi, cos, sin
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram

def get_state (theta):
  """returns a valid state vector"""
  return [cos(theta/2), sin(theta/2)]

# play with the values for theta to get a feeling
theta = -pi/2 # affects the probabilities


# create, initialize, and execute the quantum circuit
qc = QuantumCircuit(1)
qc.initialize(get_state(theta), 0)
backend = Aer.get_backend('statevector_simulator')
result = execute(qc,backend).result()
counts = result.get_counts()

# Show the histogram
plot_histogram(counts)
```

Figure 3.9: Theta specifies the qubit state |->
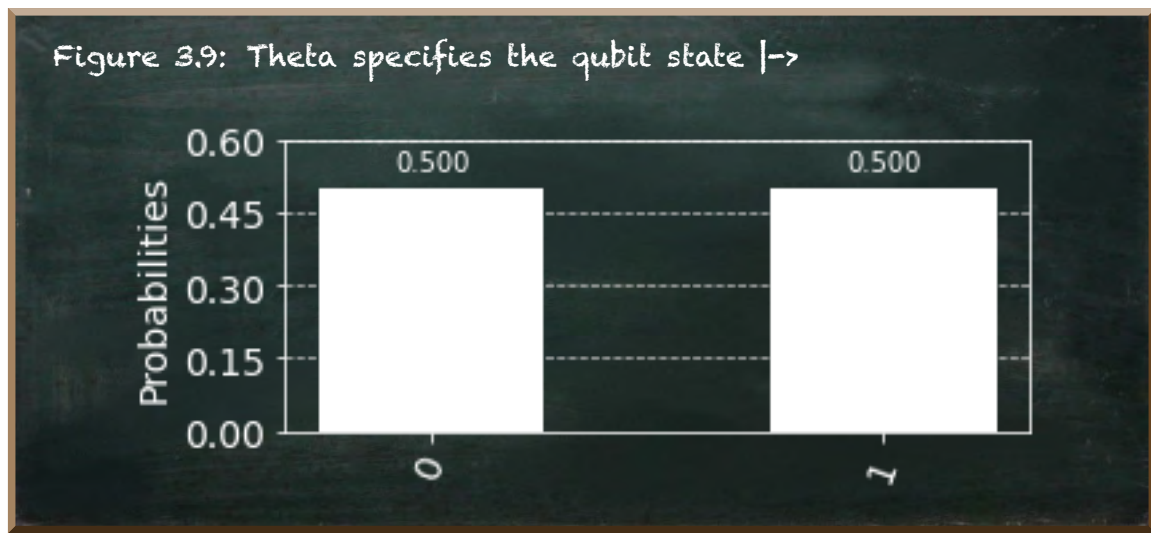
In this piece of code, we introduced the function `getState` (line 5). It takes `theta` as a parameter and returns the array `[cos(theta/2), sin(theta/2)]`. This is the vector we specified in the equation 3.20.

# 3.4 Exploring The Observer Effect

A qubit is a two-level quantum system that is in a superposition of the quantum states $|0\rangle$ and $|1\rangle$ unless you observe it. Once you observe it, there are distinct probabilities of measuring `0` or `1`. In physics, this is known as the observer effect. It says the mere observation of a phenomenon inevitably changes that phenomenon itself. For instance, if you measure the temperature in your room, you're taking away a little bit of the energy to heat up the mercury in the thermometer. This loss of energy cools down the rest of your room. In the world we experience, the effects of observation are often negligible.

But in the sub-atomic world of quantum mechanics, these effects matter. They matter a lot. The mere observation of a quantum bit changes its state from a superposition of the states $|0\rangle$ and $|1\rangle$ to either one value. Thus, even the observation is a manipulation of the system we need to consider when developing a quantum circuit.

Let's revisit the quantum circuit from section 3.1. Here's the code and the result if you run it:

Listing 3.9: A circuit without measurement

```
1  from qiskit import QuantumCircuit, execute, Aer
2  from qiskit.visualization import plot_histogram
3  from math import sqrt
4
5  # Create a quantum circuit with one qubit
6  qc = QuantumCircuit(1)
7
8  # Define state |Psi>
9  initial_state = [1/sqrt(2), 1/sqrt(2)]
10
11 # Apply initialization operation to the qubit at position 0
12 qc.initialize(initial_state, 0)
13
14 # Tell Qiskit how to simulate our circuit
15 backend = Aer.get_backend('statevector_simulator')
16
17 # Do the simulation, returning the result
18 result = execute(qc,backend).result()
19
20 # Get the data and display histogram
21 counts = result.get_counts()
22 plot_histogram(counts)
```

Figure 3.10: Probabilities of measuring a qubit
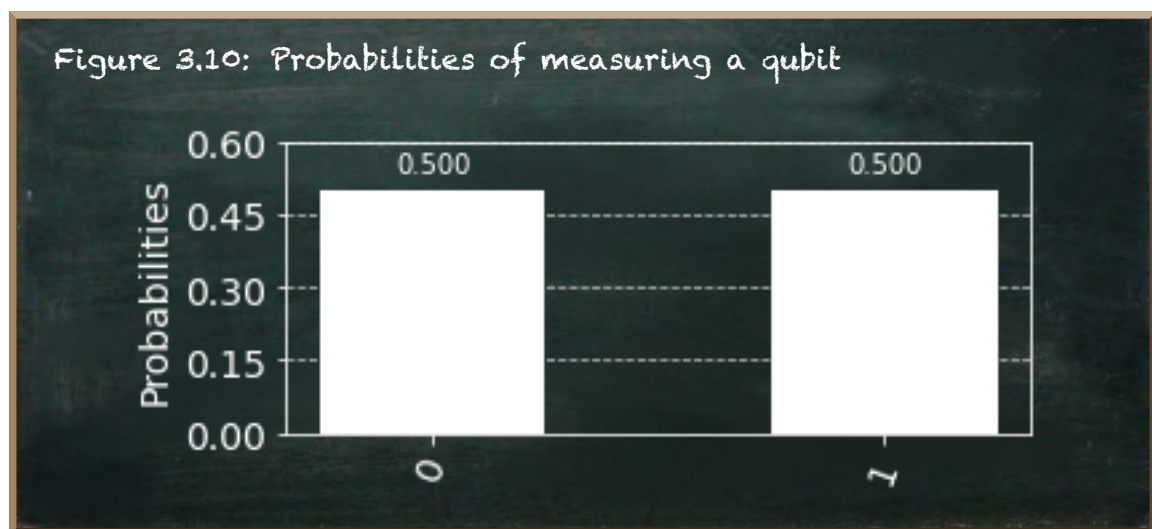
Our circuit consists of a single qubit (line 6). It has the initial state `[1/sqrt(2), 1/sqrt(2)]` (line 9) that we initialize our quantum circuit with (line 12).

Here are the *Dirac* and the vector notation of this state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

We add a simulation backend (line 15), execute the circuit, and obtain the result (line 18). The `result` object provides the `get_counts` function that provides the probabilities for the resulting (observed) state of our qubit.

Let's have a look at our circuit. The `QuantumCircuit` provides the `draw` function that renders an image of the circuit diagram. Provide `output=text` as a named parameter to get an ASCII art version of the image.

Listing 3.10: Draw the circuit

```
1 qc.draw(output='text')
```


Figure 3.11: The qubit state
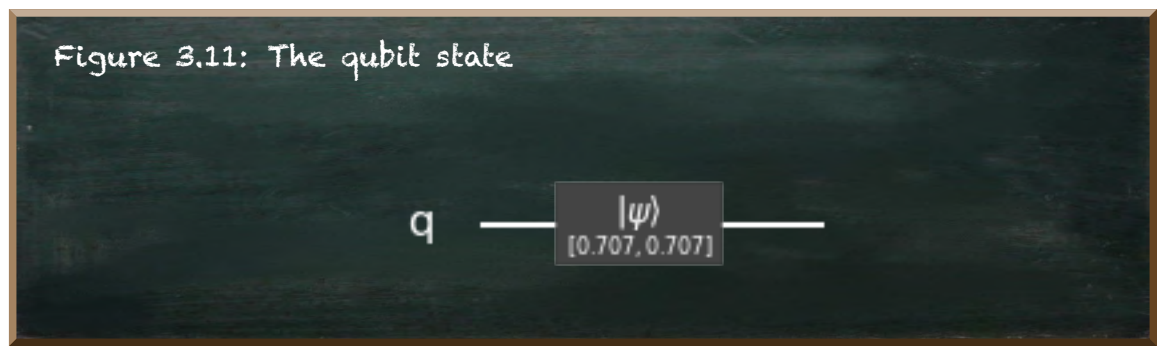
This drawing shows the inputs on the left, outputs on the right, and operations in between.

What we see here is our single qubit (`q`) and its initialization values ($\frac{1}{\sqrt{2}} = 0.707$). These values are the input and the output of our circuit. When we execute this circuit, our `result` function evaluates the quantum bit in the superposition state of $|0\rangle$ and $|1\rangle$. Thus, we have a 50:50 chance to catch our qubit in either one state.

Let's see what happens if we observe our qubit as part of the circuit.

Listing 3.11: Circuit with measurement

```
1  qc = QuantumCircuit(1)
2  qc.initialize(initial_state, 0)
3
4  # observe the qubit
5  qc.measure_all()
6
7  # Do the simulation, returning the result
8  result = execute(qc,backend).result()
9  counts = result.get_counts()
10 plot_histogram(counts)
```


Figure 3.12: Measuring the qubit inside the circuit
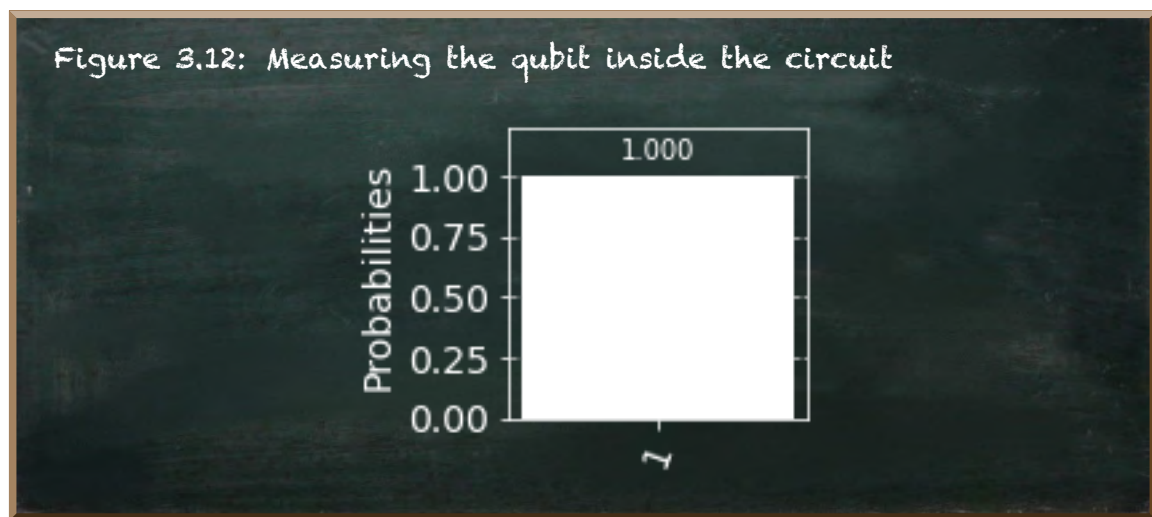
"Whoa?!"

We get a 100% probability of resulting state 1. That can't be true. Let's rerun the code. I know, doing the same things and expecting different results is a sign of insanity.

Listing 3.12: Another circuit with measurement

```
1  qc = QuantumCircuit(1)
2  qc.initialize(initial_state, 0)
3  qc.measure_all()
4  result = execute(qc,backend).result()
5  counts = result.get_counts()
6  plot_histogram(counts)
```

Figure 3.13: Measuring the qubit inside the circuit, again
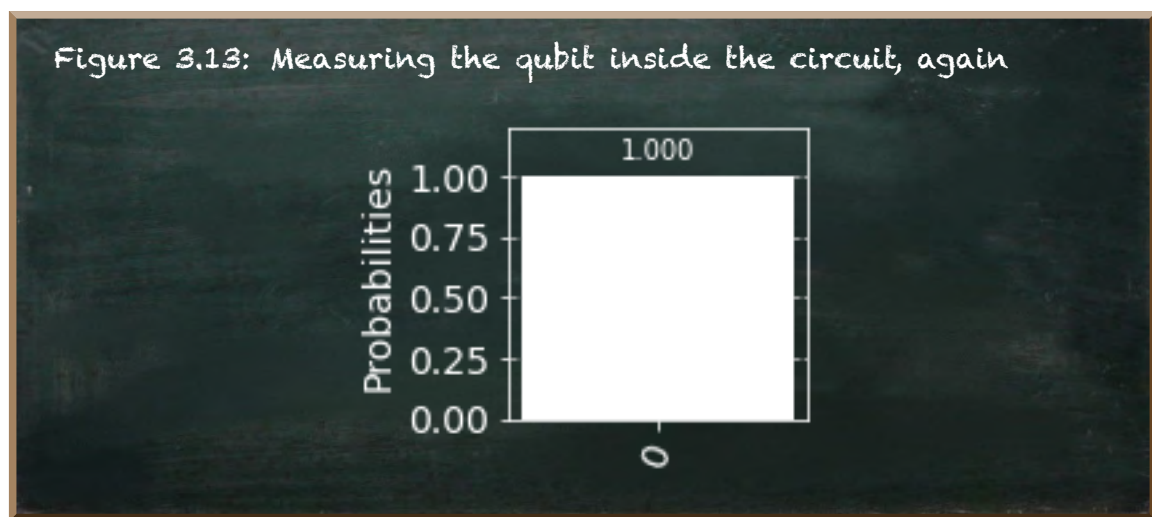
Again. 100% probability of measuring ... wait ... it's state 0.

No matter how often you run this code, you'll always get a 100% probability of either 0 or 1. In fact, if you reran the code many, many times and counted the results, you'd see a 50:50 distribution.

Sounds suspicious? Yes, you're right. Let's have a look at our circuit.

Listing 3.13: Draw a circuit with measurement

```
1 qc.draw(output='text')
```


Figure 3.14: A circuit with measurement
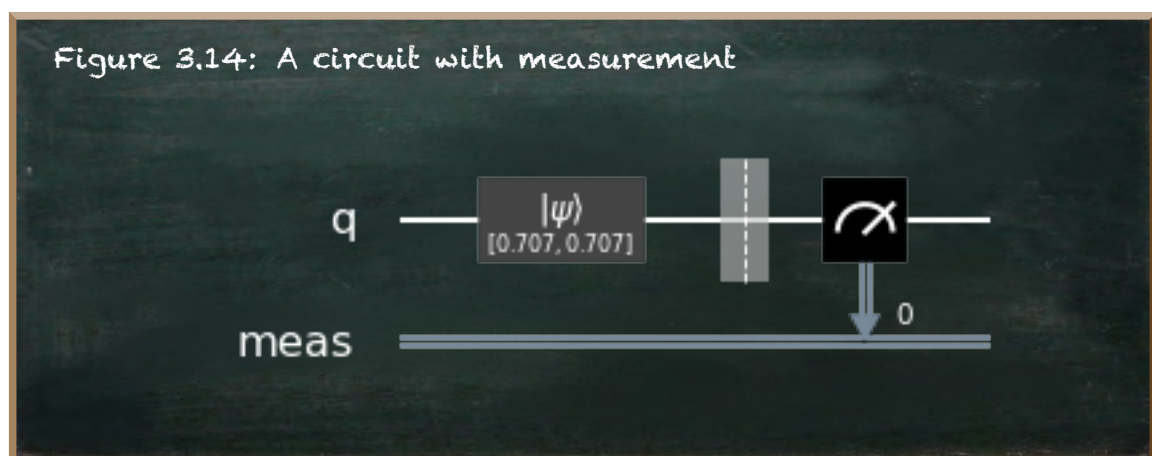
Our circuit now contains a measurement. That is an observation. It pulls our qubit out of a superposition state and lets it collapse into either 0 **or** 1. When

we obtain the result afterward, there's nothing quantumic anymore. It is a distinct value. And this is the output (to the right) of the circuit.

Whether we observe a 0 or a 1 is now part of our quantum circuit.

> ⚠ The small number at the bottom measurement line does not depict a qubit's value. It is the measurement's index that indicates the classical bit that receives the measurement.

Sometimes, we refer to measurement as collapsing the state of the qubit. This notion emphasizes the effect a measurement has. Unlike classical programming, where you can inspect, print, and show values of your bits as often as you like, in quantum programming, measurement has an effect on your results.

If we constantly measured our qubit to keep track of its value, we would keep it in a well-defined state, either 0 or 1. Such a qubit wouldn't be different from a classical bit. Our computation could be easily replaced by a classical computation. In quantum computation, we must allow the qubits to explore more complex states. Measurements are therefore only used when we need to extract an output. This means that we often place all measurements at the end of our quantum circuit.

In this section, we had a look at the simplest quantum circuit. We initialize a single qubit and observe it. But it effectively demonstrates the observer effect in quantum computing. It is something we need to keep in mind when we start manipulating our qubits.

3.5 Parameterized Quantum Circuit

In chapter 2, we created different hypocrite classifiers. These are classifiers solely building upon chance when predicting the label of a thing. While such a classifier can yield seemingly good performance in a single measure, such as precision, it does not reach an average far beyond 0.5 four the four measures that directly result from the confusion matrix (precision, recall, specificity, and NPV).

In this section, we use a quantum circuit to solve our binary classification task. This quantum circuit is a **Parameterized Quantum Circuit** (PQC). A PQC is a quantum circuit that takes all data it needs as input parameters. Therefore it has its name parameterized. It predicts the label of the thing

based on these parameters.

The following image 3.15 depicts the simple PQC we are about to build in this section.

Figure 3.15: A PQC binary classifier
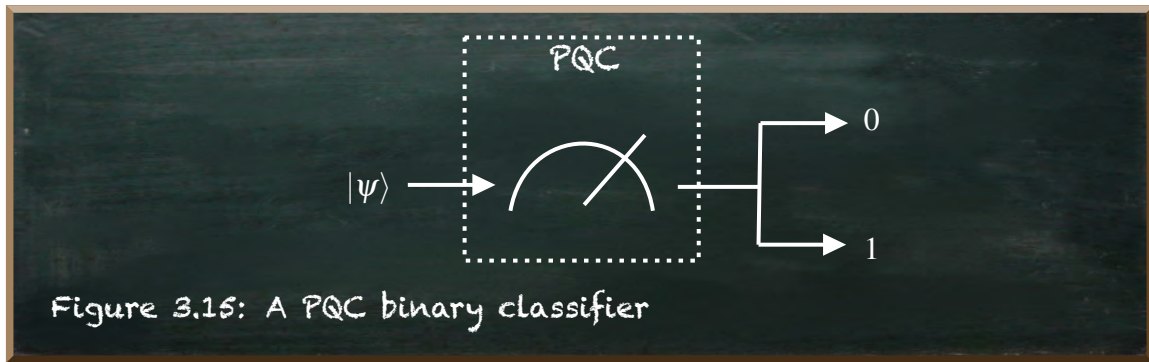
This PQC takes a single quantum state (ψ) as its input. It measures the state and provides its prediction as output.

We created such a quantum circuit in the last section 3.4, already.

Here's the source code.

Listing 3.14: A simple PQC binary classifier

```
1 qc = QuantumCircuit(1)
2 initial_state = [1/sqrt(2), 1/sqrt(2)]
3 qc.initialize(initial_state, 0)
4 qc.measure_all()
```

In fact, this circuit outputs either `0` or `1`, each with a probability of 50%. It sounds a lot like the random classifier we created in section 2.5.

Let's wrap this circuit into a function we can use with the `run` and `evaluate` functions we created in that section to see whether it behaves similarly.

Listing 3.15: The parameterized quantum circuit classifier

```python
from qiskit import execute, Aer, QuantumCircuit
from math import sqrt
from sklearn.metrics import recall_score, precision_score,
    confusion_matrix

def pqc_classify(backend, passenger_state):
  """backend -- a qiskit backend to run the quantum circuit at
  passenger_state -- a valid quantum state vector"""

  # Create a quantum circuit with one qubit
  qc = QuantumCircuit(1)

  # Define state |Psi> and initialize the circuit
  qc.initialize(passenger_state, 0)

  # Measure the qubit
  qc.measure_all()

  # run the quantum circuit
  result=execute(qc,backend).result()

  # get the counts, these are either {'0': 1} or {'1': 1}
  counts=result.get_counts(qc)

  # get the bit 0 or 1
  return int(list(map(lambda item: item[0], counts.items())))[0])
```

The first difference to notice is the function takes two parameters instead of one (line 5). The first parameter is a *Qiskit* `backend`. Since the classifier will run a lot of times in a row, it makes sense to reuse all we can. And we can reuse the `backend`.

The second parameter differs from the classifiers thus far. It does not take the passenger data but a quantum state vector (`passenger_state`) as input. This is not a problem right now since all the hypocrite classifiers we developed so far ignored the data anyway.

The function creates a quantum circuit with one qubit (line 12), initializes it with the `passenger_state` (line 15), measures the qubit (line 18), executes the quantum circuit (line 21), and retrieves the counts from the `result` (line 24). All these steps did not change.

But how we return the counts is new (line 27). `counts` is a Python dictionary.

It contains the measurement result (either 0 or 1) as a key and the probability as the associated value. Since our quantum circuit measures the qubit, it collapsed to a finite value. Thus, the measurement probability is always 1. Consequently, `counts` is either `{'0': 1}` or `{'1': 1}`.

All we're interested in here is the key. And this is what we return.

We start (from inner to outer) with the term `counts.items()`. It transforms the Python dictionary into a list of tuples, like `[('0', 1)]`. Since we only have one key in the dictionary, there is only one tuple in the list. The important point is to get the tuple rather than the dictionary's key-value construct because we can access a tuple's elements through the index.

This is what we do in the function `lambda: item: item[0]`. It takes a tuple and returns its first element. We do this for every item in the list (even though there is only one item) by using `list(map(...))`. From this list, we take the first (and only) item (either `'0'` or `'1'`) and transform it into a number (`int(...)`).

Before we can run it, we need to load the prepared passenger data.

Listing 3.16: Load the data

```
1  import numpy as np
2
3  with open('data/train.npy', 'rb') as f:
4      train_input = np.load(f)
5      train_labels = np.load(f)
6
7  with open('data/test.npy', 'rb') as f:
8      test_input = np.load(f)
9      test_labels = np.load(f)
```

The following code runs the `pqc_classifier` with the initial state with a probability of 0.5 to measure 0 or 1, respectively (line 5).

Further, we create a backend (line 2) and provide it as a parameter to be reused (line 8).

Listing 3.17: The scores of the random quantum classifier

```
1  # Tell Qiskit how to simulate our circuit
2  backend = Aer.get_backend('statevector_simulator')
3
4  # Specify the quantum state that results in either 0 or 1
5  initial_state = [1/sqrt(2), 1/sqrt(2)]
6
7  classifier_report("Random PQC",
8      run,
9      lambda passenger: pqc_classify(backend, initial_state),
10     train_input,
11     train_labels)
```

```
The precision score of the Random PQC classifier is 0.39
The recall score of the Random PQC classifier is 0.49
The specificity score of the Random PQC classifier is 0.51
The npv score of the Random PQC classifier is 0.61
The information level is: 0.50
```

When we run the `pqc_classify` classifier with the initial state, we can see that it yields identical scores as the random classifier did.

But how these two classifiers create the results is entirely different.

The classic "random" classifier uses the function `random` and initializes it, as depicted by the following code snippet.

Listing 3.18: Initialization of classical (pseudo-)random

```
1  import random
2  random.seed(a=None, version=2)
```

We provide `None` as the randomness source (`a`). This implies that the function takes a value from the operating system. Thus, it appears random, but it is not. If we knew the value it gets from the operating system or specified a distinct value ourselves, we could reproduce the exact predictions.

That's why Python's `random` function generates pseudo-random (see Python-docs) numbers.

By contrast, the PQC generates truly random results (when running on a real quantum computer). This is following one of the interpretations of the quantum state of superposition that we discussed in section (3.1).

Nevertheless, we have not used anything quantumic yet, making us see the difference between classical pseudo-random and quantumic genuinely random.

3.6 Variational Hybrid Quantum-Classical Algorithm

The PQC binary classifier we created in the previous section 3.5 is as good as the random classifier or as poor because it does not increase the information level.

This is going to change now. So far, we always feed the PQC with the same initial state: $|\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$, with the corresponding array in Python: `[1/sqrt(2), 1/sqrt(2)]`.

This state does not take into account the passenger data at all. It is a hypocrite classifier, such as the classifiers we build in section 2.7. Hypocrite classifiers solely use chance when predicting the label of a thing. While such a classifier can yield seemingly good performance in a single metric, such as precision, it does not reach an average above 0.5 for the four metrics that directly result from the confusion matrix (precision, recall, specificity, and NPV). Thus, it does not provide any information gain.

To improve our classifier, we need to use the passenger data. However, even though we prepared the passenger data into normalized numerical data, it does not fit the quantum state vector we need to feed into our PQC. Therefore, we need to pre-process our passenger data to be computable by a quantum computer.

We implicitly post-processed the results as part of the `return` statement, as shown in the following snippet.

Listing 3.19: Return statement of pqc-classify

```
1 def pqc_classify(backend, passenger_state):
2   # ...
3
4   # get the bit 0 or 1
5   return int(list(map(lambda item: item[0], counts.items())))[0])
```

Since we have a binary classification task, our prediction is 0 or 1. Thus, our post-processing is limited to transforming the output format. But in any other setting, post-processing may involve translation from the output of the quantum circuit into a useable prediction.

Altogether, we wrap the PQC into a process of classical pre-processing and post-processing. This is an algorithm with an outer structure running at a classical computer and an inner component running on a quantum computer. It is a **Variational Hybrid Quantum-Classical Algorithm**, and it is a popular approach for near-term quantum devices.

Figure 3.16 shows the overall architecture of our simple Variational Hybrid Quantum-Classical Algorithm.

Figure 3.16: A Variational Hybrid Quantum-Classical Algorithm
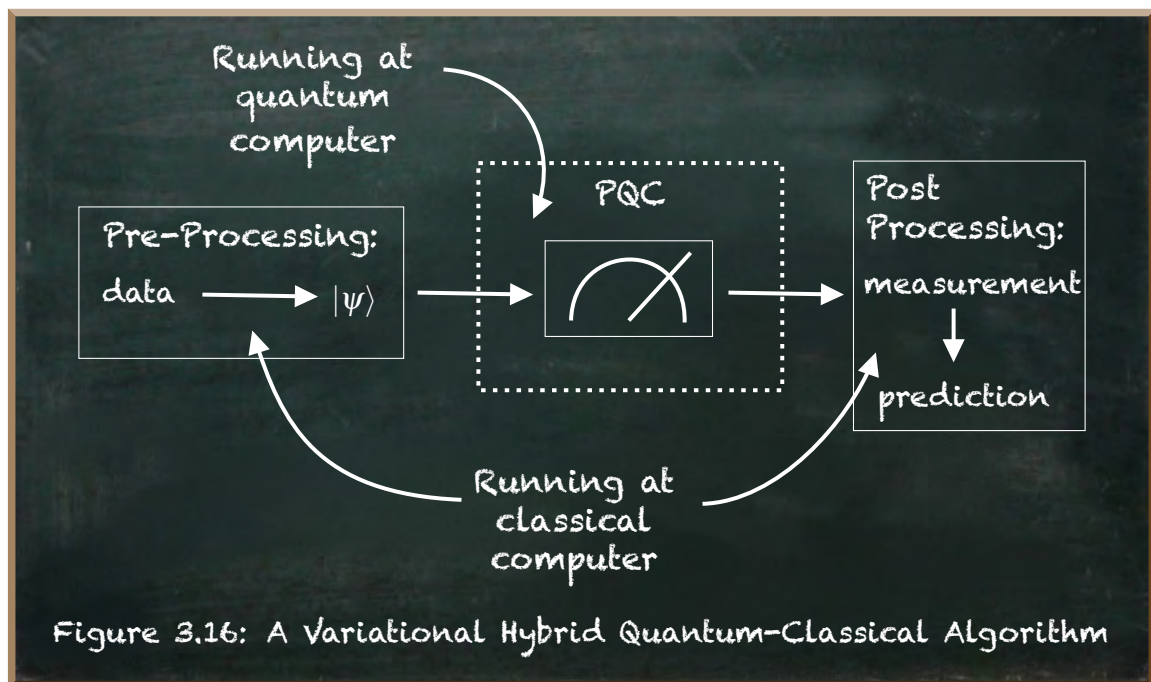
The data is pre-processed on a classical computer to determine a set of parameters for the PQC. In our simple case, this is the quantum state vector $|\psi\rangle$.

The quantum hardware uses the initial quantum state, works with it, and performs measurements. All its calculations are parameterized. So, they are relatively small and short-lived. In our case, we only measure the quantum state. We do not use any other parameters beyond $|\psi\rangle$.

Finally, the measurement outcomes are post-processed by the classical computer to generate a prediction.

The overall algorithm consists of a closed-loop between the classical and quantum components.

Let's separate our code thus far into the three parts:

- Pre-processing
- PQC
- Post-processing

Listing 3.20: Pre-processing template

```python
def pre_process(passenger):
    """

    passenger -- the normalized (array of numeric data) passenger data
    returns a valid quantum state
    """
    quantum_state = [1/sqrt(2), 1/sqrt(2)]
    return quantum_state
```

The function `pre_process` takes the passenger data as an array of numeric data.

It returns a valid quantum state vector. In this first version, it returns the balanced state of measuring `0` or `1` with equal probabilities.

Listing 3.21: The parameterized quantum circuit

```python
def pqc(backend, quantum_state):
    """
    backend -- a qiskit backend to run the quantum circuit at
    quantum_state -- a valid quantum state vector
    returns the counts of the measurement
    """

    # Create a quantum circuit with one qubit
    qc = QuantumCircuit(1)

    # Define state |Psi> and initialize the circuit
    qc.initialize(quantum_state, 0)

    # Measure the qubit
    qc.measure_all()

    # run the quantum circuit
    result=execute(qc,backend).result()

    # get the counts, these are either {'0': 1} or {'1': 1}
    counts=result.get_counts(qc)

    return counts
```

The function `pqc` is the PQC. It takes a quantum `backend` and a valid `quantum_state` as input parameters.

It prepares and runs the quantum circuit before it returns the `counts` of its measurements.

Listing 3.22: Post-processing

```python
def post_process(counts):
    """
    counts -- the result of the quantum circuit execution
    returns the prediction
    """
    return int(list(map(lambda item: item[0], counts.items()))[0])
```

The function `post_process` takes the `counts` as input and returns the prediction (see section 3.5 for the detailed explanation of how to transform the `counts` dictionary into the prediction).

Let's put it all together.

Listing 3.23: The scores of the random quantum classifier

```
1  # Tell Qiskit how to simulate our circuit
2  backend = Aer.get_backend('statevector_simulator')
3
4  classifier_report(
5      "Variational",
6      run,
7      lambda passenger: post_process(pqc(backend, pre_process(passenger))),
8      train_input,
9      train_labels)
```

```
The precision score of the Variational classifier is 0.39
The recall score of the Variational classifier is 0.50
The specificity score of the Variational classifier is 0.53
The npv score of the Variational classifier is 0.63
The information level is: 0.51
```

We first create the `statevector_simulator` backend we can reuse for all our predictions (line 2).

We use the `classifier_report` wrapping function we developed in section 2.7.

Besides an arbitrary name it uses in the output (line 5), the primary input is the classifier we provide (line 6).

We provide an anonymous (`lambda`) function (a function without a name) as our classifier. It takes a single parameter `passenger` and runs (from inner to outer) the `pre_process` function with the `passenger` as a parameter. Finally, we put the result alongside the `backend` into the `pqc` function whose result we put into the `post_process` function.

When we run the pqc classifier with the initial state, we can see that it yields the identical scores as the random classifier.

Now, it's finally time to build a real classifier. One that uses the actual passenger data to predict whether the passenger survived the Titanic shipwreck or not.

Let's start at the end. The current post-processing already returns either `0` or `1`. This fits our required output since `0` represents a passenger who died and `1` means the passenger survived.

The current PQC measures the provided quantum state vector and returns the `counts`. We could leave it unchanged if we provided input a vector whose probability corresponds to the passenger's actual likelihood to survive.

The passenger data consists of an array of seven features. We already transformed all features into numbers between `0` and `1` (section 2.4).

Thus, the pre-processing task is to translate these seven numbers into a quantum state vector whose probability corresponds to the passenger's actual likelihood to survive.

Finding such a probability is the innate objective of any machine learning algorithm.

Our data consists of seven features. The central assumption is that these features determine or at least affected whether a passenger survived or not. If that weren't the case, we wouldn't be able to predict anything reliably. So let's assume the features determine survival.

The question then is, **how do these seven features determine survival?** Is one feature more important than another? Is there a direct relationship between a feature and survival? Are there any interdependencies between the features, such as if A then B indicates survival? But if not A, then B is irrelevant, but C is essential.

But before we use sophisticated tools (such as Bayesian networks) that can discover complex structures of how the features determine the outcome, we start simple.

We assume all features are independent of each other, and each feature contributes more or less to the survival or death of the passenger.

Therefore, we say the overall probability of survival $P(survival)$ is the sum of each feature's value F times the feature's weight μ_F ("mu").

$$P(survival) = \sum (F \cdot \mu_F) \tag{3.22}$$

Let's have a look at what this means in Python.

Listing 3.24: weigh a passenger's feature

```
1  def weigh_feature(feature, weight):
2      """
3      feature -- the single value of a passenger's feature
4      weight -- the overall weight of this feature
5      returns the weighted feature
6      """
7      return feature*weight
```

The `weigh_feature` function calculates and returns the term $F \cdot \mu_F$. Thus, this function calculates how much a passenger's feature the age contributes to this passenger's overall probability of survival. The higher the weighted value, the higher the chance.

Next, we need to add all the weighted features to calculate the overall probability.

Listing 3.25: Calculate the overall probability

```
1   from functools import reduce
2
3   def get_overall_probability(features, weights):
4       """
5       features -- list of the features of a passenger
6       weights -- list of all features' weights
7       """
8       return reduce(
9         lambda result, data: result + weigh_feature(*data),
10        zip(features, weights),
11          0
12      )
```

The function `get_overall_probability` takes two parameters. First, it takes the list of a passenger's feature values. This is a passenger's data. Second, it takes the list of the feature weights.

We construct a list of tuples for each feature (line 10) containing the feature and its weight. Python's `zip` function takes two separate lists and creates the respective tuple for every two elements in the lists.

We `reduce` this list of `(feature, weight)` into a single number (line 8). Then, we call the `weight_feature`-function for each of the tuples and add up the results (line 9), starting with the value `0` (line 11).

Now, we need to calculate the weights of the features. These are similar across all passengers. We build the weights upon the correlation coefficients.

The correlation coefficient is a measure of the relationship between two variables. Each variable is a list of values. It denotes how much the value in one list increases as the value of the other list increases. The correlation coefficient can take values between -1 and 1.

- A correlation coefficient of 1 means that there is a proportional increase in the other for every increase in one variable.
- A correlation coefficient of -1 means that there is a proportional decrease in the other for every increase in one variable.
- A correlation coefficient of 0 means that the two variables are not linearly related.

We calculate the correlation coefficient for each feature in our dataset in relation to the list of `labels`. In the following code, we separate our dataset into a list of the columns (line 4).

The term `list(map(lambda passenger: passenger[i], train_input` transforms each passenger's data into its value at the position `i`. And we do this `for i in range(0,7)`. It means we do this for each column.

Listing 3.26: Calculate the correlation coefficients

```
1  from scipy.stats import spearmanr
2
3  # separate the training data into a list of the columns
4  columns = [list(map(lambda passenger: passenger[i], train_input)) for i
       in range(0,7)]
5
6  # calculate the correlation coefficient for each column
7  correlations = list(map(lambda col: spearmanr(col, train_labels)[0],
       columns))
8  correlations
```

```
[-0.33282978445145533,
 -0.539340557551996,
 -0.029337576985579865,
 0.10244706581397216,
 0.15946021387370407,
 0.3222967880289113,
 -0.16443725432119416]
```

There are different types of correlation coefficients. The most frequently used are the Pearson and Spearman correlation methods.

The Pearson correlation is best suited for linear continuous variables, whereas the Spearman correlation also works for monotonic ordinal variables. Since we have some categorical data (`Pclass`, `Sex`, and `Embarked`), we use the Spearman method to calculate the correlation coefficient.

Scipy provides the function `spearmanr` for us. We call this function for each column and the `train_labels` (line 7). The function returns two values, the correlation coefficient and the p-value. We're only interested in the first (at index `0`).

The correlation coefficients range from -0.58 to 0.32.

Let's put this all together in the pre-processing.

Listing 3.27: The weighting pre-processing

```python
from math import pi, sin, cos

def get_state (theta):
  """returns a valid state vector from angle theta"""
  return [cos(theta/2), sin(theta/2)]

def pre_process_weighted(passenger):
  """
  passenger -- the normalized (array of numeric data) passenger data
  returns a valid quantum state
  """

  # caluclate the overall probability
  mu = get_overall_probability(passenger, correlations)

  # theta between 0 (|0>) and pi (|1>)
  quantum_state = get_state((1-mu)*pi)

  return quantum_state
```

We use the function `get_state` from section 3.2. It takes the angle `theta` and returns a valid quantum state. An angle of 0 denotes the state $|0\rangle$ which is the probability of 100% measuring `0`. An angle of π denotes the state $|1\rangle$ that is the probability of 100% measuring `1`.

Accordingly, we multiply the overall probability we calculate at line 14 with

`pi` to specify an angle up to π (line 17). Since the correlation coefficients are between -1 and 1 and most of our coefficients are negative, a value of μ towards -1 implies the passenger died. Thus, we reverse the angles by calculating `(1-mu)*pi`.

Now, we're ready to run the classifier. Let's feed it into the `classifier_report` wrapping function.

Listing 3.28: Run the PQC with the weighted pre-processing

```
1  backend = Aer.get_backend('statevector_simulator')
2
3  classifier_report("Variational",
4      run,
5      lambda passenger: post_process(pqc(backend, pre_process_weighted(
         passenger))),
6      train_input,
7      train_labels)
```

```
The precision score of the Variational classifier is 0.70
The recall score of the Variational classifier is 0.61
The specificity score of the Variational classifier is 0.84
The npv score of the Variational classifier is 0.78
The information level is: 0.73
```

We achieve an overall information level of about 0.73 to 0.77. Not too bad, is it? But before we're starting to party, we need to test our classifier. We "trained" the classifier with the training data. So it had seen the data before.

Let's run the classifier with the test dataset.

Listing 3.29: Test the PQC-based classifier on data it has not seen before

```
1  classifier_report("Variational-Test",
2      run,
3      lambda passenger: post_process(pqc(backend, pre_process_weighted(
         passenger))),
4      test_input,
5      test_labels)
```

```
The precision score of the Variational-Test classifier is 0.67
The recall score of the Variational-Test classifier is 0.68
The specificity score of the Variational-Test classifier is 0.78
The npv score of the Variational-Test classifier is 0.78
The information level is: 0.73
```

The overall information level is somewhere between 0.71 and 0.76. This is only slightly lower than the value we get when running it on the training data. The algorithm seems to generalize (to a certain extent).

Most importantly, in comparison to the hypocrite classifiers, we see a significant increase in the information level. Therefore, this classifier provides real information.

It is our first working Variational Hybrid Quantum-Classical Classifier.

4. Probabilistic Binary Classifier

In our first simple Variational Hybrid Quantum-Classical Binary Classification Algorithm, we developed in the previous section , we used a Parameterized Quantum Circuit (PQC) that did nothing but measuring a quantum state. While quantum systems bring inherent randomness and allow us to work with probabilities, we did not yet use this characteristic because we determined the resulting probability of measuring either 0 or 1 upfront in a classical program.

In the following two chapters, we go one step further. We create a probabilistic binary classifier that calculates the resulting likelihood inside the PQC. We build a Variational Hybrid quantum-classical Naïve Bayes Classifier. It builds upon Bayes' Theorem. Starting with an initial prior probability, we update the resulting probability inside the PQC based on the evidence given by the passenger data.

Don't worry if you're not familiar with Bayes Theorem and the Naïve Bayes classifier. We'll cover all the basics in this chapter.

We use the Titanic shipwreck data to discover Bayes' Theorem and the Naïve Bayes classifier with actual data. We load the original data here because it is easier to work with manually.

Listing 4.1: Load the raw data

```
1 import pandas as pd
2 train = pd.read_csv('./data/train.csv')
```

The following table depicts the first five rows and the data in the `train` *Pandas* dataframe. See section 2.3 for more details on the dataset.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

4.1   Towards Naïve Bayes

"*Did a passenger survive the Titanic shipwreck?*"

A probabilistic classifier predicts the label of a thing based on its probability. So, to answer the question above, we need to know what the chance to survive is?

Let's calculate it. In the following snippet, we create a list of all survivors (line 2). First, we use the *Pandas* chaining operation (`train.Survived`) to access a column. Then, we use *Pandas'* `eq()` function and chain it to the column. It selects the rows whose values match the provided value (`1` for survival).

The probability of surviving is the number of survivors divided by the total number of passengers (line 5).

Listing 4.2: Calculating the probability to survive the Titanic shipwreck

```python
# list of all survivors
survivors = train[train.Survived.eq(1)]

# calculate the probability
prob_survival = len(survivors)/len(train)
print('P(Survival) is {:.2f}'.format(prob_survival))
```

```
P(Survival) is 0.38
```

Given our dataset, the probability of survival is 38%. Thus, I'd rather say the passenger died than survived.

This is a probabilistic classifier already. It is the "predict_death" classifier we created in section 2.5 and discussed in section 2.7. Even though it is a hypocrite classifier because it does not consider the individual passenger when predicting survival, this classifier yields a higher precision than a purely random classifier does.

What if the passenger had a second-class ticket? What was this passenger's probability of surviving?

Let's have a look. In the following snippet, we create a list of passengers with a second-class ticket (`train.Pclass.eq(2)`, line 2).

We divide the survivors of this subset (`secondclass.Survived.eq(1)`) by the total number of passengers with a second-class ticket (line 4).

Listing 4.3: Calculating the probability to survive if the passenger had a second-class ticket

```
1  # list of all passengers with a second class ticket
2  secondclass = train[train.Pclass.eq(2)]
3
4  prob_survival_secondclass = len(secondclass[secondclass.Survived.eq(1)])/
       len(secondclass)
5  print('P(Survived|SecondClass) is {:.2f}'.format(
       prob_survival_secondclass))
```

```
  P(Survived|SecondClass) is 0.47
```

Second-class passengers had a probability of surviving of 47%. Thus, those passengers had a much better chance to survive than the average passenger. Mathematically, the term $P(Survived|SecondClass)$ describes a conditional probability. In general, a conditional probability consists of a *Hypothesis* whose probability it denotes, and some *Evidence* we observed.

$$P(Hypothesis|Evidence) \tag{4.1}$$

This notion of a conditional probability is already an important part of a Bayesian classifier. While a hypocrite classifier sticks with its prediction ignoring all evidence, the Bayesian classifier updates our belief about a hypothesis given the evidence.

What if the passenger was female?

Listing 4.4: Calculating the probability to survive if the passenger was female

```
1  #list of all females
2  females = train[train.Sex.eq("female")]
3
4  prob_survival_female = len(females[females.Survived.eq(1)])/len(females)
5  print('P(Survived|Female) is {:.2f}'.format(prob_survival_female))
```

```
  P(Survived|Female) is 0.74
```

Females had an even better chance to survive. And what if we know that the passenger was female and had a second-class ticket?

Listing 4.5: Calculating the probability to survive if the passenger was female and had a second-class ticket

```
1  #list of all females with a second class ticket
2  secondclass_female = secondclass[secondclass.Sex.eq("female")]
3  prob_survival_secondclass_female = len(secondclass_female[
       secondclass_female.Survived.eq(1)])/len(secondclass_female)
4  print('P(Survived|SecondClass,Female) is {:.2f}'.format(
       prob_survival_secondclass_female))
```

```
  P(Survived|SecondClass,Female) is 0.92
```

92% of the female passengers with a second-class ticket survived. So if I were to predict the survival of such a passenger, I'd say she survived.

A probabilistic classifier can be a powerful tool. For example, based on only two features, we got an almost precise result on the chances to survive for a particular class of passengers.

The problem is, though, there are myriads of possible types of passengers—a different type for each possible combination of all the features. For one thing, calculating all of them upfront is cumbersome. The other and even worse, when we consider all features, the number of passengers per class might be only one. Thus, we would create an algorithm that memorizes the training data rather than generalizing to yet unknown data.
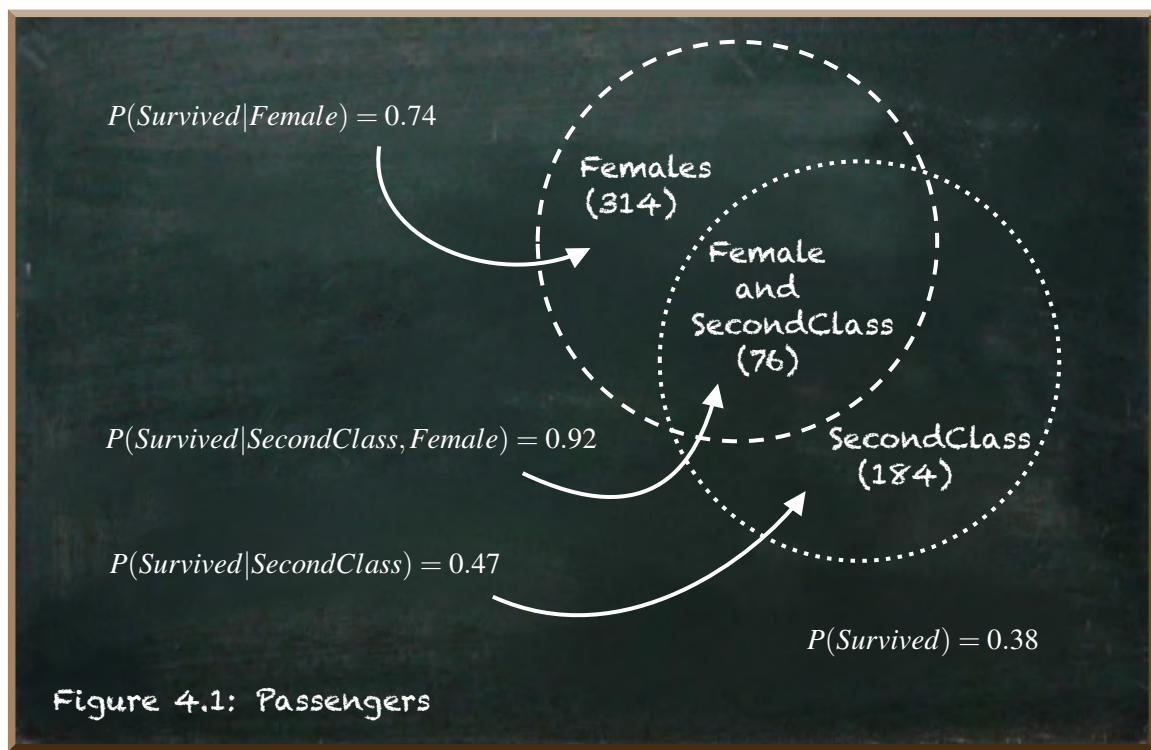
In the example above, we calculated the probability of $P(Survived|SecondClass, Female)$ solely based on the knowledge we gathered from female passengers with a second-class ticket. We excluded everything we know about female passengers in other classes or male passengers.

Listing 4.6: Counting passengers

```
1 print('There are {} female passengers the dataset'.format(len(females)))
2 print('There are {} passengers with a second−class ticket in the dataset'.
      format(len(secondclass)))
3 print('There are {} female passengers with a second−class ticket in\nthe
      dataset'.format(len(secondclass_female)))
```

```
There are 314 female passengers the dataset
There are 184 passengers with a second class ticket in the dataset
There are 76 female passengers with a second class ticket in
the dataset
```

The following image 4.1 illustrates these subsets of passengers.

$P(Survived|Female) = 0.74$

Females (314)

Female and SecondClass (76)

$P(Survived|SecondClass, Female) = 0.92$

SecondClass (184)

$P(Survived|SecondClass) = 0.47$

$P(Survived) = 0.38$

Figure 4.1: Passengers

We see, we only consider 76 passengers (out of 891 in the dataset) when calculating the probability to survive for female passengers with a second-class ticket. Thus, the focus narrows down very quickly.

But maybe there's an alternative. Maybe we can derive the probability of $P(Survived|SecondClass, Female)$ differently. Indeed, we can. This is where Naïve Bayes comes into play. In simple terms, Naïve Bayes assumes that the presence of a particular feature in a dataset is unrelated to the presence of any other feature. In our case, it implies a passenger being female is unrelated to the ticket class.

But how can we calculate $P(Survived|SecondClass, Female) = 0.92$ from $P(Survived|Female) = 0.74$ and $P(Survived|SecondClass) = 0.47$?

This is where Bayes' Theorem comes into play and helps us.

4.2   Bayes' Theorem

Bayes' Theorem describes a way of finding a conditional probability when you know certain other probabilities. The following equation 4.2 denotes Bayes' Theorem mathematically:

$$\underbrace{P(Hypothesis|Evidence)}_{posterior} = \underbrace{P(Hypothesis)}_{prior} \cdot \underbrace{\frac{P(Evidence|Hypothesis)}{P(Evidence)}}_{modifier} \qquad (4.2)$$

Bayes' Theorem says we can calculate the "*posterior*" probability from a "*prior*" probability and some evidence-related "*modifier*".

The "*posterior*" denotes what we believe about $Hypothesis$ **after** gathering the new information about the $Evidence$. It is a conditional probability such as we discussed above. The "*prior*" probability denotes what we believed about $Hypothesis$ **before** we gathered the new information. It is the overall probability of our $Hypothesis$.

The *modifier* of the new information denotes the relative change of our belief about $Hypothesis$ caused by the $Evidence$.

This *modifier* is the quotient of the backward probability ($P(Evidence|Hypothesis)$) and the probability of the new piece of information ($P(Evidence)$). The backward probability (the numerator of the *modifier*) answers the question, "what is the probability of observing this evidence in a world where our hypothesis is true?" The denominator is the probability of observing the evidence on its own.

Thus, when you see the evidence often in a world where the hypothesis is true, but rarely on its own, this evidence seems to support the hypothesis. On the contrary, if you usually see the evidence everywhere but you don't see it in a world where the hypothesis is true, then the evidence opposes the hypothesis.

The farther the *modifier* is away from 1, the more it changes the probability. A *modifier* of precisely 1 would not change the probability at all. Let's define the value of the *informativeness* as the *modifier*'s distance to 1.

$$Informativeness = |\frac{P(Evidence|Hypothesis)}{P(Evidence)} - 1|$$

If we have one hypothesis H and multiple pieces of evidence $E_1, E_2, \ldots, E_n$, then we have n modifiers $M_1, M_2, \ldots, M_n$:

$$\underbrace{P(H|E_1, E_2, \ldots, E_n)}_{posterior} = \underbrace{\frac{P(E_1|H)}{P(E_1)}}_{M_1} \cdot \underbrace{\frac{P(E_2|H)}{P(E_2)}}_{M_2} \cdot \ldots \cdot \underbrace{\frac{P(E_n|H)}{P(E_n)}}_{M_n} \cdot \underbrace{P(H)}_{prior} \qquad (4.3)$$

What does that mean in practice?

Our *Hypothesis* is a passenger survived the Titanic shipwreck. We have two pieces of evidence *Female* and *SecondClass*.

- $P(Survived)$ is the overall probability of a passenger to survive.
- $P(Female)$ is the probability of a passenger to be female,
- and $P(SecondClass)$ is the probability of a passenger holding a second-class ticket.
- $P(Female|Survived)$ denotes how likely a passenger who survived is female.
- And $P(SecondClas|Survived)$ denotes how likely a passenger who survived had a second-class ticket.

The following equation 4.4 depicts how to calculate the probability of a female passenger with a second class ticket to survive:

$$P(Survived|SecCl, Female) = \frac{P(SecCl|Survived)}{P(SecCl)} \cdot \frac{P(Female|Survived)}{P(Female)} \cdot P(Survived)$$

$$(4.4)$$

Let's have a look at the Python code.

Listing 4.7: Calculating the posterior probability

```
1  # calculate the backwards probability of a survivor having a
2  # second-class ticket
3  p_surv_seccl = len(survivors[survivors.Pclass.eq(2)])/len(survivors)
4
5  # calculate the modifier and the informativeness of the second-class
       ticket
6  m_seccl = p_surv_seccl / (len(secondclass)/len(train))
7  i_seccl = abs(m_seccl-1)
8  print('The modifier of the second-class ticket is {:.2f}.\nThe
       informativeness is {:.2f}'.format(m_seccl, i_seccl))
9
10 # calculate the backwards probability of a survivor being female
11 p_surv_female = len(survivors[survivors.Sex.eq("female")])/len(survivors)
12
13 # calculate the modifier and the informativeness of being female
14 m_female = p_surv_female / (len(females)/len(train))
15 i_female = abs(m_female-1)
16 print('The modifier of being female is {:.2f}.\nThe informativeness is
       {:.2f}'.format(m_female,i_female))
17
18 # calculate the posterior probability
19 posterior = m_seccl * m_female * prob_survival
20 print('\nP(Survived|SecondClass,Female) is {:.2f}'.format(posterior))
```

```
The modifier of the second class ticket is 1.23.
The informativeness is 0.23
The modifier of being female is 1.93.
The informativeness is 0.93


P(Survived|SecondClass,Female) is 0.91
```

First, we calculate the *modifier* of the second class ticket (line 6) and of being female (line 14). As we can see, the *modifier* is a positive number that scales our prior probability. Thus, we can see that both evidences increase the chance a passenger survived because they are greater than 1. And we can see the *informativeness* of being female is higher than the *informativeness* of a second-class ticket because it has a bigger effect on the prior probability.

The Bayesian probability $P(Survived|SecondClass, Female) = 0.91$ does not exactly match the forward probability (0.92) we calculated earlier. The reason

is the assumed independence of conditions. However, the Bayesian probability comes close enough.

The first question to arise usually is why $P(Evidence|Hypothesis)$ is easier to estimate than $P(Hypothesis|Evidence)$? If I don't know what $P(Hypothesis|Evidence)$ is, how am I supposed to know what $P(Evidence|Hypothesis)$ is?

The explanation usually involves the more constrained perspective of $P(Hypothesis|Evidence)$. In our case, the probabilities $P(Female|Survived)$ and $P(SecondClas|Survived)$ are narrowed down to the survivors. We calculated them from the respective survivors subset (lines 3 and 11).

In a real-world setting, we could retrieve these data by surveying the survivors. The forward probability of $P(Survived|SecondClass, Female)$ requires a representative list of all the passengers.

This explanation, however, does not explain why we use the formula in the case where we have such a list. In our case, it has simple, practical reasons. As mentioned, if we calculated the forward probabilities directly, we would need to do it for every single class of passengers. This is a number that grows exponentially with the number of features. For instance, if we have seven (useful) features and each feature has only two values (some have many more), there are $2^7 = 128$ classes of passengers to consider. If only two of them (ticket class and port of embarkation) have three possible values, we're up to 288 classes ($2^5 * 3^2$).

By contrast, Bayes' Theorem lets us add a feature by calculating its *modifier*. This is a quotient of two probabilities. With seven features, we need to calculate $2 * 2 * 7 = 28$ probabilities. If two features have three rather than two possible values, we need $2 * (2 * 5 + 3 * 2) = 32$ probabilities. This number grows linearly, only.

Do you object: "I *have a computer capable of running this number of calculations*"?

While you're certainly right about a problem with seven features, you might be wrong about a problem with 20 or 100 features. But even for the problem at hand, if we considered every single group of passengers, we still had the problem of too small groups that result in memorization rather than generalizable learning.

Finally, a Naïve Bayes classifier works well with missing data. Because if you don't have certain evidence, it is no problem to leave it unconsidered. You update your belief in the resulting probability based on the evidence you have.

4.3 Gaussian Naïve Bayes

So far, we have considered categorical data. There are two genders in our dataset. There are three classes of tickets. These features have distinct values we can treat as such.

But how about features such as the age or the paid fare. One way is to transform numerical features into their categorical counterparts. The question is how and where to separate the categories from each other. For instance, a 29-year old passenger has a different age than a 30-year old. But they are somewhat similar when compared to a 39-year old passenger. But when we split by tens, we would put the 30-year old and the 39-year old passengers together and separate them from the 29-year-old.

The other option we have is to treat numerical features as continuous distributions. A continuous distribution cannot be expressed in tabular form. Instead, we use an equation to describe a continuous probability distribution. A common practice is to assume normal (Gaussian) distributions for numerical variables. The following equation 4.5 denotes the general form of the Gaussian density function

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{4.5}$$

The parameter μ_x is the mean of the evidence's probability distribution. The parameter σ_x is its standard deviation.

The following image depicts such a distribution.

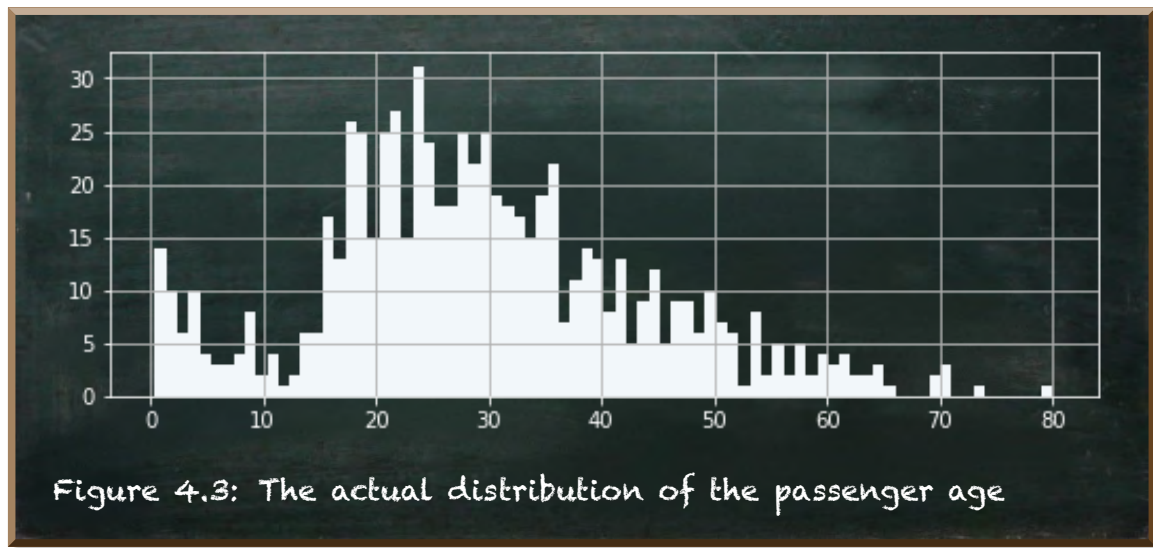Figure 4.2: Gaussian probability distribution

Before we use this formula and calculate how the age affects the chances to

survive, let's first have a look at the actual distribution of the passenger age.

Listing 4.8: The actual distribution of the passenger age

```
1 train["Age"].hist(bins=80)
```


Figure 4.3: The actual distribution of the passenger age

Pandas lets us quickly create a histogram of each series in the `DataFrame`. A histogram is a representation of the distribution of data. The named parameter `bins=80` tells *Pandas* the number of data points (on the x-axis) the histogram should have.

While it is not perfectly normal distributed, we can see many passengers in the center between 15 and 35. Next, we calculate the mean and the standard deviation.

Listing 4.9: Calculating the mean and the standard deviation of the passenger age

```
1 age_mean = train["Age"].mean()
2 age_std = train["Age"].std()
3 print('The average passenger age is {:.1f}. The standard deviation is
      {:.1f}'.format(age_mean, age_std))
```

```
The average passenger age was 29.7. The standard deviation was 14.5
```

Pandas also provides convenience functions to calculate the mean and the standard deviation of a data series, such as a column in a `DataFrame`.

Now, we're ready to calculate the *modifier* of a certain age. We can use our formula from above. Let's calculate the informativeness of an age of 29 years.

$$Modifier_{Age=29} = \frac{P(Age = 29|Survived)}{P(Age = 29)} \tag{4.6}$$

Again, we use the backward probability. To calculate $P(Age = 29|Survived)$, we need to use the age distribution among the survivors.

Listing 4.10: Calculating modifier and informativenesses of the age of 29

```python
from math import exp, sqrt, pi

def density(mu,sigma,age):
    return 1/(sigma*sqrt(2*pi))*exp(-0.5*((age-mu)/sigma)**2)

survivor_age_mean = survivors["Age"].mean()
survivor_age_std = survivors["Age"].std()
print('The average survivor age is {:.1f}. The standard deviation is {:.1
    f}'.format(survivor_age_mean, survivor_age_std))

# calculate the Informativeness of the age of 29
p_surv_age29 = density(survivor_age_mean, survivor_age_std, 29)
p_age29 = density(age_mean, age_std, 29)
m_age29 = p_surv_age29 / p_age29
i_age29 = abs(m_age29-1)
print('The modifier of the age of 29 is {:.2f}.'.format(m_age29))
print('Its informativeness is {:.2f}.'.format(i_age29))
```

```
The average survivor age is 28.3. The standard deviation is 15.0
The modifier of the age of 29 is 0.97.
Its informativeness is 0.03.
```

We create a convenience function `density` to calculate $P(x)$ (lines 3-4). We use this function to calculate $P(Age = 29|Survived)$ (line 11) and $P(Age = 29)$ (line 12). We calculate the *modifier* as the quotient of both (line 13).

We see that the age of 29 does not have a great effect on the probability to survive. Its *modifier* is close to 1 and thus, its *informativeness* is pretty small (`0.03`).

Let's calculate the informativeness of 70-year-old and 5-year-old passengers as a comparison.

Listing 4.11: Calculating informativenesses of other ages

```
 1  # calculate the Informativeness of the age of 70
 2  p_surv_age70 = density(survivor_age_mean, survivor_age_std, 70)
 3  p_age70 = density(age_mean, age_std, 70)
 4  m_age70 = p_surv_age70 / p_age70
 5  i_age70 = abs(m_age70−1)
 6  print('The modifier of the age of 70 is {:.2f}.'.format(m_age70))
 7  print('Its informativeness is {:.2f}.\n'.format(i_age70))
 8
 9  # calculate the Informativeness of the age of 5
10  p_surv_age5 = density(survivor_age_mean, survivor_age_std, 5)
11  p_age5 = density(age_mean, age_std, 5)
12  m_age5 = p_surv_age5 / p_age5
13  i_age5 = abs(m_age5−1)
14  print('The modifier of the age of 5 is {:.2f}.'.format(m_age5))
15  print('Its informativeness is {:.2f}.'.format(i_age5))
```

```
The modifier of the age of 70 is 0.94.
Its informativeness is 0.06.

The modifier of the age of 5 is 1.22.
Its informativeness is 0.22.
```

The *modifier* of the age of 70 is not much different from the *modifier* of the age of 29. But an age of 5 years resulted in an increased probability of survival.

5. Working with Qubits

5.1 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 5.1: Hieroglyphs

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:

Don't get me wrong. Math is a great way to describe technical concepts. Math is 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.

When we're about to learn something new, it is easier for us to 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 learner's language 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.

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. Thus, regardless of programming language, a programmer works a lot with boolean logic.

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

Listing 5.1: If then else in Python

```python
if x and y:# A statement to evaluate in boolean logic
  doSomething () # if the statement evaluiates to True
else:
  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`. For example, 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 three examples of boolean logic in Python.

Listing 5.2: Boolean logic in Python

```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))
```

```
not P is False
P and Q is False
P or Q is True
P and not Q is True
```

While Python uses these exact keywords, in math, symbols represent these operators:

- $\neg$ means `not`
- $\wedge$ means `and`
- $\vee$ means `or`

If you're not a mathematician, these symbols and all the other symbols you encounter on your quantum machine learning journey may appear cryptic. But while the representation of a concept may differ when you describe it in Python or math, the concept itself is the same.

You don't need to be a mathematician to understand boolean logic. You don't need to be a programmer, either, because we can even describe the boolean logic by truth tables.

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. For instance, the following figure 5.2 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`.

Figure 5.2: Truth table

This truth table reveals that `P or Q` is equivalent to `not (not P and not Q)`. This logical equivalence tells us that we do not even need the operator `or`. We could replace it by `not (not P and not Q)`.

But `P or Q` is concise and much easier to understand.

"What if there was no or operator in our programming language?"

The savvy programmer would write her custom operator.

Listing 5.3: A reimplementation of or

```
1 def my_or(p, q):
2     return not (not p and not q)
3
4 print('P | Q | P or Q')
5 print('--------------')
6 print('T | T | {}'.format(my_or(True, True)))
7 print('T | F | {}'.format(my_or(True, False)))
8 print('F | T | {}'.format(my_or(False, True)))
9 print('F | F | {}'.format(my_or(False, False)))
```

```
 P | Q | P or Q
 --------------
 T | T | True
 T | F | True
 F | T | True
 F | F | False
```

This is what programming is all about. Programmers write functions that produce a particular behavior. They use and combine these functions to create even more functions that exhibit even complex behavior. The whole program they write comes down to a set of functions savvily combined. Programmers have their compiler (or interpreter) to translate the higher-level functions down to basic boolean logic. And this basic boolean logic can be performed using electrical switches. The switches and their combination are called gates. When we connect gates, they form a circuit.

At a discrete interval, the computer sends a pulse of electricity through the circuit. If we receive a pulse of electricity at the appropriate time, we interpret it as 1 (true). If we don't receive a pulse, we interpret it as 0(false).

Despite the name, there is nothing circular about circuits. They are linear and are read from left to right. Let's look at an example that corresponds to the boolean functions that we looked at earlier.

The following figure 5.3 depicts the circuit diagram of not (not P and not Q). The circuit receives the input from the left and outputs it to the right.

Figure 5.3: Classical circuit
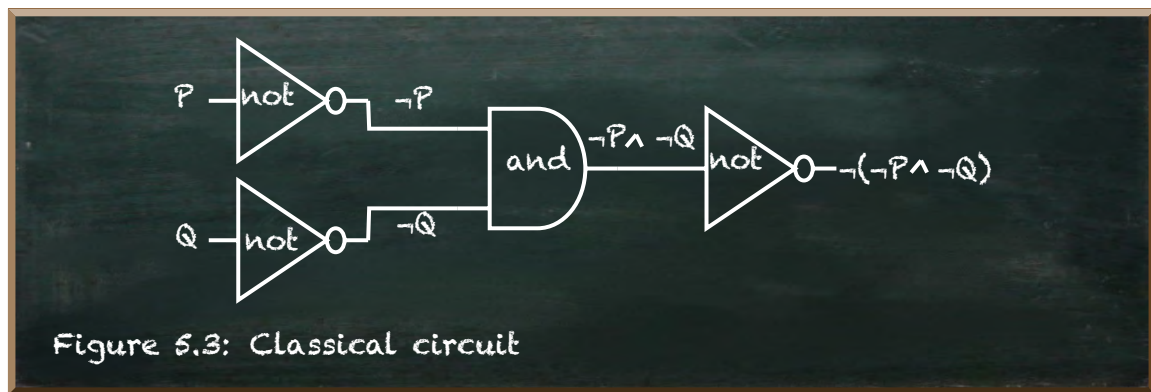
Such gates and circuits are the building blocks of any modern computer. This includes quantum computers. While the world of quantum mechanics is different, the world of quantum computing is surprisingly similar.

Don't let yourself be dazzled by all the mathematical formulae. They are representations of concepts. Not more, not less.

Let's return to our introductory formula:

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

It is the mathematical notation of the quantum state $|\psi\rangle$ ("psi"). While the state of a classical bit is boolean (either 0 meaning false or 1 meaning true),

the state of the quantum bit (qubit) is the superposition of the quantum states $|0\rangle$ and $|1\rangle$ weighted by α and β.

In this state of superposition, the quantum system is neither 0 nor 1 unless you measure it. Only when you measure the qubit, the state collapses to either 0 or 1. The squares of the two weights (α^2) and (β^2) denote the probabilities of measuring either 0 or 1. The larger α is, the higher the probability of measuring 0. Respectively, the larger β is, the higher the probability of measuring 1.

The formula says something more. It says the quantum state is the vector of the two weights $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$.

A vector is a geographical object that has a length (magnitude) 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.

Figure 5.4: A vector
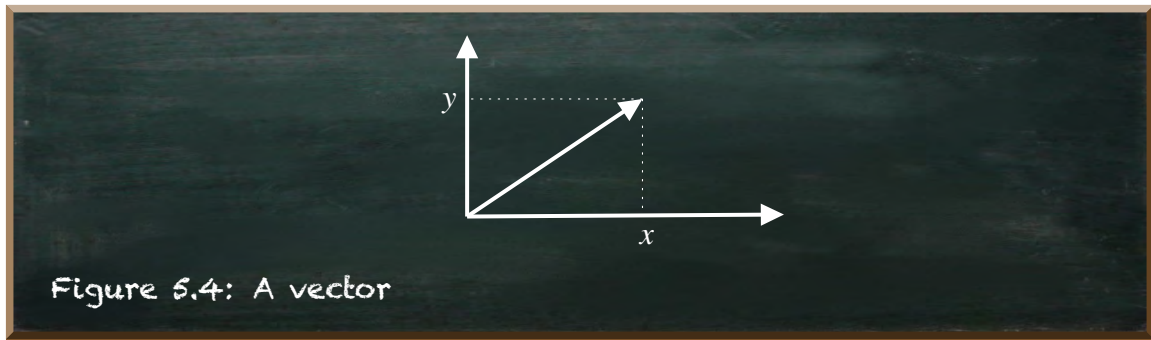
In Python, a vector is an array. Thus, the state of a qubit is the array `[alpha, beta]`. `alpha` and `beta` are numerical variables. The quantum state is an array of two numbers.

But an array of two numbers is a much more complex datatype than a boolean value is. A boolean is either `True` or `False`. You can transform boolean values with simple operators, such as `not`, `and`, and `or`. You can reason about the transformation of boolean values in a truth table.

But how do you transform an array of two numbers? And how can you reason about such transformations?

The apparent answer is math. But it is not the only possible answer. So, let's use Python for that.

Listing 5.4: Reversing the qubit states

```python
from math import sqrt

# define the initital states
psi = [0.5, sqrt(3)/2]
always_0 = [1, 0]
always_1 = [0, 1]

def transform(name, state, f):
  print ('{}: [{:.2f}, {:.2f}] result: [{:.2f}, {:.2f}]'.format(name, *
    state, *f(state)))

def reverse_state(arr):
  return list(reversed(arr))

print("----------- Reversed states: ----------")
transform("psi", psi, reverse_state)
transform("|0>", always_0, reverse_state)
transform("|1>", always_1, reverse_state)
```

```
----------- Reversed states: ----------
 psi: [0.50, 0.87]  result: [0.87, 0.50]
 |0>: [1.00, 0.00]  result: [0.00, 1.00]
 |1>: [0.00, 1.00]  result: [1.00, 0.00]
```

We start with the initialization of three states. Each state is an array of two numbers. The state `psi` has the values $\frac{1}{2}$ and $\frac{\sqrt{3}}{2}$ (line 4). The probability of measuring 0 in this state is $(\frac{1}{2})^2 = \frac{1}{4} = 0.25$. The probability of measuring 1 is $(\frac{\sqrt{3}}{2})^2 = \frac{3}{4} = 0.75$.

The state `always_0` has the values 1 and 0. The probability of measuring 0 in this state is $1^2 = 1$ (line 5). The probability of measuring 1 is $0^2 = 0$. When we measure a qubit in this state, we always measure it as 0. The state `always_1` is the respective opposite. We consistently measure it as 1 (line 6).

Next, we create a convenience function `transform` (lines 8-9). Did I tell you that writing functions to make things easier is what programming is all about? This is an example. The function takes the name of the quantum state (an arbitrary string to show), the `state`, and a function `f`. `transform` prints to the console the original state and the state after having applied the function `f` on it.

Finally, we create a function `reverse_state` we can feed into `transform` (lines 11-12). `reverse_state` calls Python's default `reversed` function that returns an array of the same length in the opposite order.

In the output, we can see that the numbers in the state arrays have switched their positions. Thus, the probability of measuring `0` or `1` switched, respectively. The reversed `psi` has a `0.75` chance of measuring `0` and a `0.25` chance of measuring `1`. The reversed `always_0` is similar to the original `always_1`.

These are only three possible states. Listing all possible states in a kind of truth table is impossible. But I think the behavior of the `reverse_state` function is quite clear. It is the behavior of the X-gate in quantum computing. It is one of the fundamental transformations of the quantum state.

Let's have a look at this gate in practice. We use IBM's quantum computing SDK Qiskit.

Listing 5.5: The measured qubit

```python
from qiskit import execute, Aer, QuantumCircuit
from qiskit.visualization import import plot_histogram

# Create a quantum circuit with one qubit
qc = QuantumCircuit(1)

 # Define initial_state
qc.initialize(psi, 0)

# Apply the X—gate
qc.x(0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# Do the simulation, returning the result
result = execute(qc,backend).result()
counts = result.get_counts()
plot_histogram(counts)
```

Figure 5.5: The effect of the X-gate
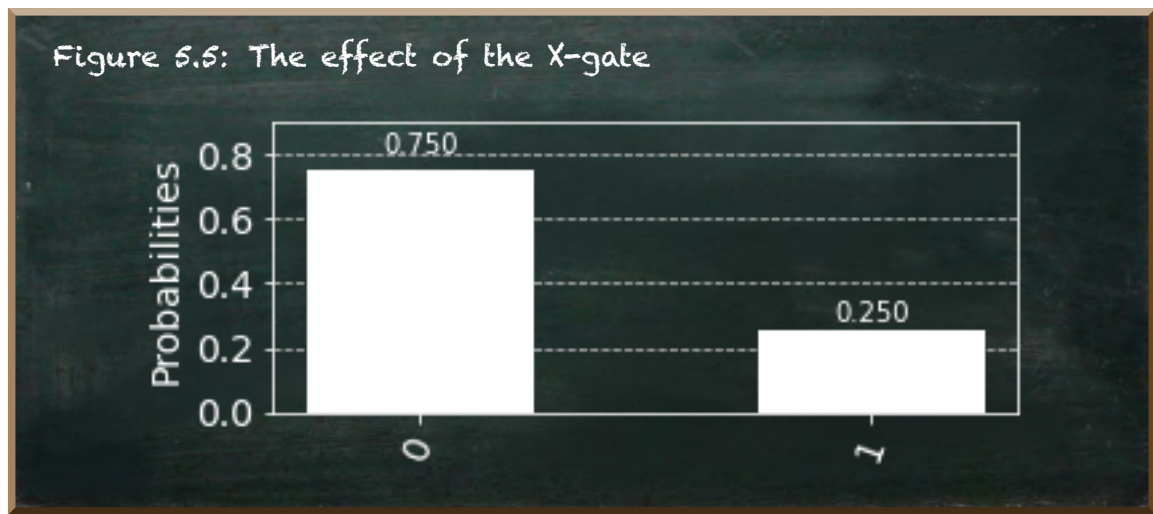
The fundamental unit of *Qiskit* is the quantum circuit. A quantum circuit is a model for quantum computation. The program, if you will. Our circuit consists of a single one qubit (line 5).

We initialize our qubit with the state `psi` (line 8), and we apply the X-gate on it (line 11).

Qiskit provides the `Aer` package (that we import at line 1). In addition, it offers different backends for simulating quantum circuits. The most common backend is the `statevector_simulator` (line 14).

The `execute` function (that we import at line 1) runs our quantum circuit (`qc`) at the specified `backend`. It returns a `job` object that has a useful method `job.result()` (line 17). This returns the `result` object once our program completes it.

Qiskit uses Matplotlib to provide insightful visualizations. A simple histogram will do. The `result` object provides the `get_counts` method to obtain the histogram data of an executed circuit (line 18).

The method `plot_histogram` returns a *Matplotlib* figure that Jupyter draws automatically (line 19).

We see we have a 75% chance of observing the value `0` and a 25% chance of observing the value `1`–The exact opposite of the initial state.

You can run the circuit with different initial states to get a better feeling for this gate.

In general, quantum circuits are not different from classical circuits. We can represent them in a diagram. *Qiskit*'s `QuantumCircuit` class provides the `draw`

method that does the job for us.

Listing 5.6: The measured qubit

```
1 qc.draw('mpl')
```

Figure 5.6: A circuit with an X-gate
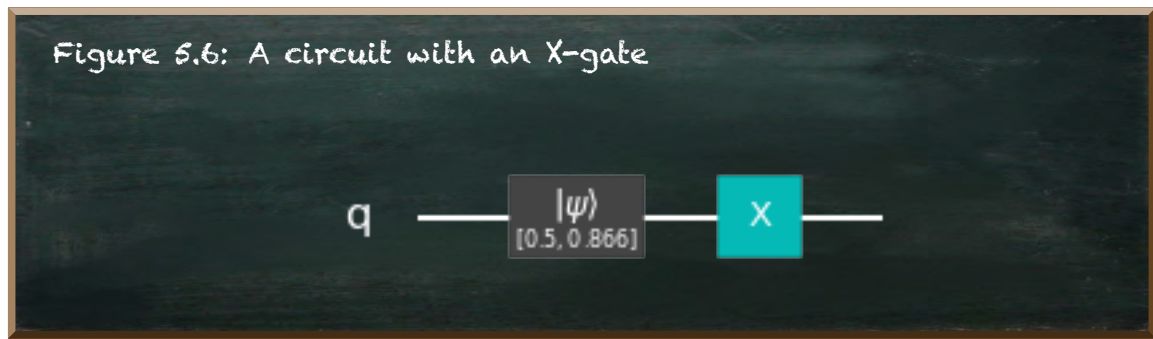
We can see our only qubit (`q`), its initialization with the array `[0.5, 0.866]`, and the applied X-gate.

You've completed the first step towards quantum computing mastery without being a mathematician. Getting a conceptual understanding of quantum gates as the quantumic peers of classical circuit gates does not depend on math. The combination of plain English and a little bit of Python is well-suited. And for many, this combination is much more accessible.

But, math remains paramount to quantum computing. So, if you want to gain a deep understanding of the concepts, you'll cope with the mathematical formulae sooner or later. And as I said, math is a great way to describe technical concepts.

Let's have a look at the underlying math of the X-gate. Don't worry. I don't expect you to be a mathematician. A little affinity to algebra (that is, the study of mathematical symbols and the rules from manipulating them) doesn't hurt, though.

So far, we used Python's built-in function `reversed`. While this is convenient, we do not see how it works internally. So let's use another function–a self-made function.

Listing 5.7: Self-made reverse function

```python
def adjust_weight(state, weights):
    return state[0]*weights[0]+state[1]*weights[1]

print ('reversed psi: [{:.2f}, {:.2f}]'.format(
    adjust_weight(psi, [0,1]),
    adjust_weight(psi, [1,0])
))
```

```
reversed psi: [0.87, 0.50]
```

We define a function `adjust_weight`(line 1). It takes a quantum `state` and `weights`. Both are arrays with two items. It multiplies the values at position `0`, and it multiplies the values at position `1`. It returns the sum of these two products (line 2).

We can use this function to reverse `psi`. For `adjust_weight` returns a single number, we call it twice to get back an array of two items (lines 5 and 6). In this example, we do not explicitly create an array, but we directly print these values to the console (line 4).

In both calls, we provide the original `psi` as the `state` parameter. For the first call, whose result is the first number of the reversed psi, we provide `[0,1]` as `weights`. It means we get the sum of `0` times the first number of `psi` and `1` time the second number of `psi`. This sum is the second number of `psi`.

For the second call, whose result is the second number of the reversed psi, we provide `[1,0]` as `weights`. This is `1` time the first number of `psi` and `0` times the second number of `psi`. This equals the first number of `psi`.

With these weights, we have effectively switched the places of the numbers of `psi`.

In math, this is matrix multiplication. The general formula for multiplying a matrix M and a vector v is:

$$M \cdot |v\rangle = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} a \cdot v_0 + b \cdot v_1 \\ c \cdot v_0 + d \cdot v_1 \end{bmatrix}$$

a and b are the weights we used to calculate the first number of the resulting vector. c and d are the weights for the second number, respectively.

Mathematically, the X-gate quantum operator is the matrix: $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

Let's apply this operator to our three exemplary states:

Reversing the state $|0\rangle$ results in $|1\rangle$:

$$X \cdot |0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

Reversing the state $|1\rangle$ results in $|0\rangle$:

$$X \cdot |1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \cdot 0 + 1 \cdot 1 \\ 1 \cdot 0 + 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

And, applying the matrix at $|\psi\rangle$ results in its reversal, too:

$$X \cdot |\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} 0 \cdot \frac{1}{2} + 1 \cdot \frac{\sqrt{3}}{2} \\ 1 \cdot \frac{1}{2} + 0 \cdot \frac{\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix}$$

In classical computing, we have a small set of boolean operators whose behavior we can easily represent in truth tables. But in quantum computing, matrices denote the operators called gates. And there are myriads of possible matrices we can apply. Math is a concise yet precise way to describe these operators. But you don't need to be a mathematician to use these operations.

Of course, it is desirable to understand the underlying math of a gate when you apply it. But more importantly, you need to have an understanding of what the gate does. If you know what the X-gate does, you don't need to cope with the math all the time.

5.2 Quantumic Math – Are You Ready For The Red Pill?

After this, the Matrix is no longer cryptic symbols falling from the top, but you'll see the woman in the red dress…

… at least concerning the Hadamard gate.

> "You take the blue pill — the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill — you stay in Wonderland, and I show you how deep the rabbit-hole goes." *Morpheus, The Matrix*

Figure 5.7: Can you see the woman in the red dress?
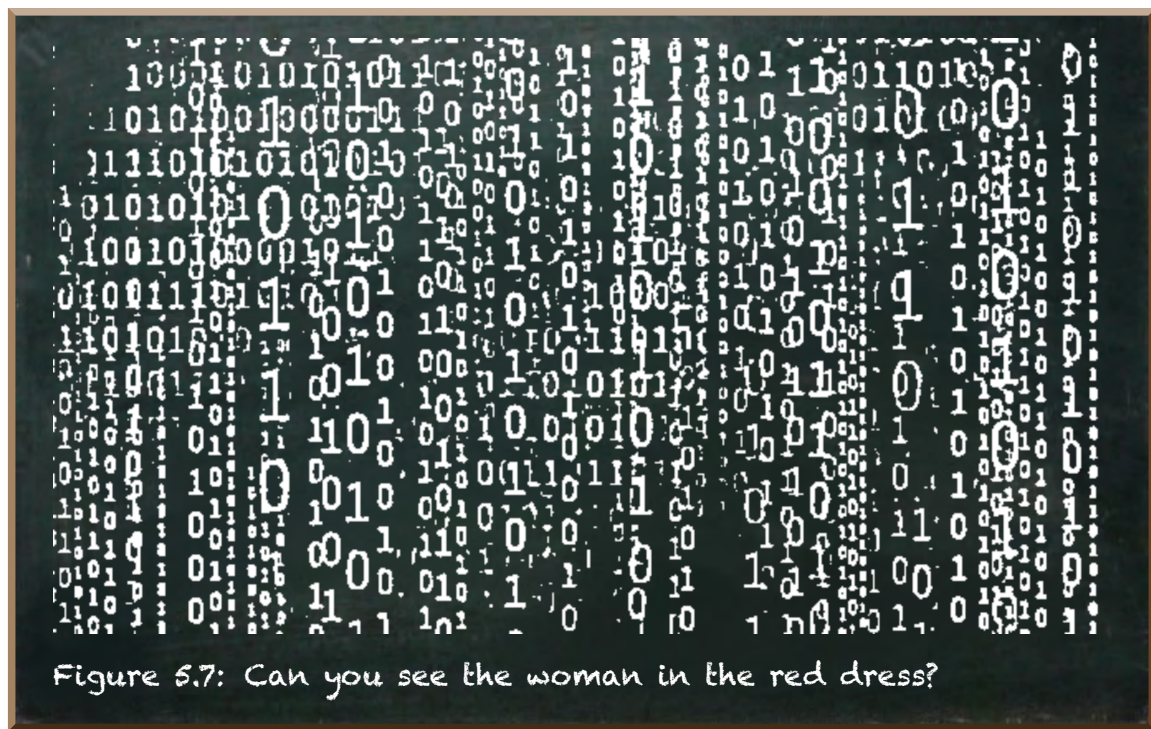
A qubit resembles the idea of the spin of an electron. It is in a state of superposition. While the electron's superposition consists of the states up and down, the qubit's superposition consists of the states $|0\rangle$ and $|1\rangle$.

A popular notion of superposition is that the system is in different states concurrently unless you measure it. But, when you look at the electron, you find it either up or down. When you look at the qubit, it is either 0 or 1. Another notion is that the system is truly random and not just *sensitive dependent on initial conditions* (see 3.1). But superposition does not mean **and**. And it does not mean **or**. It is a combination of states that does not map onto classical concepts.

> "This is your last chance. After this, there is no turning back."
> *Morpheus, The Matrix*

The basic model of superposition is given by a vector space. A vector space is a collection of all valid qubit state vectors along with the operations you can perform on them. We got to know the qubit state vector by the following equation:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$, with $\alpha^2 + \beta^2 = 1$. In Python, the array `[alpha, beta]` denotes this vector.

> ⚠ α and β are the probability amplitudes. They are not probabili-
> ties. They can be positive or negative. But their squares α^2 and
> β^2 denote the probabilities.

When we measure a qubit, it will collapse to either one of the possible mea-
surements. The number of possible measurements determines the dimen-
sion of this underlying vector space. There are two possible measurements
of a qubit, 0 or 1. Thus, the vector space is two-dimensional. All vectors in
this vector space consist of two numbers. These are the probability ampli-
tudes α and β as in the vector $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$.

When we measure a qubit, we observe it as either 0 or 1. We know the state
$|0\rangle$ says our qubit will result in the value 0 when observed. And $|1\rangle$ says our
qubit will result in the value 1 when observed. In general, $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$
says our qubit will result in the value 0 with the probability of α^2 and 1 with
the probability of β^2

The probability is a single number, called a scalar. How can we obtain this
scalar from a qubit state? There's one way of vector multiplication that pro-
duces a scalar. This is called the inner product. And it results from multiply-
ing a column vector such as $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ with a row vector, such as $\begin{bmatrix} 1 & 0 \end{bmatrix}$

In section 3.1, we introduced the **Dirac** notation and its "ket"-construct that
denotes a column vector. For instance, $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Now, we introduce the
"bra"-construct ($\langle 0|$). The bra is a row vector, such as $\langle 0| = \begin{bmatrix} 1 & 0 \end{bmatrix}$

The inner product is defined as:

$$\langle a|b\rangle = \begin{bmatrix} a_0 & a_1 & \ldots & a_n \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = a_0 \cdot b_0 + a_1 \cdot b_1 + \cdots a_n \cdot b_n$$

We can use the inner product to obtain the probability amplitude of measur-
ing a particular value from a qubit state. And its square denotes the probabil-
ity.

So, what's the probability of measuring 1 from the state $|0\rangle$? Let's build the

inner product to find out:

$$(\langle 1|0\rangle)^2 = \left(\begin{bmatrix} 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)^2 = (0 \cdot 1 + 1 \cdot 0)^2 = 0^2 = 0$$

And what's the probability of measuring 0?

$$(\langle 0|0\rangle)^2 = \left(\begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)^2 = (1 \cdot 1 + 0 \cdot 0)^2 = 1^2 = 1$$

This also works for an arbitrary state vector $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$. The probability of measuring 1 is:

$$(\langle 1|\psi\rangle)^2 = \left(\begin{bmatrix} 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \right)^2 = (0 \cdot \alpha + 1 \cdot \beta)^2 = \beta^2$$

And what's the probability of measuring $|\psi\rangle$ as 0?

$$(\langle 0|\psi\rangle)^2 = \left(\begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \right)^2 = (1 \cdot \alpha + 0 \cdot \beta)^2 = \alpha^2$$

Great! Even though this is quite mathematical, it illustrates how we can obtain a value from our quantum state by multiplying our state vector with a row vector. In layman's terms, the "bra-ket" $\langle e|\psi\rangle$ denotes the probability amplitude of measuring $|\psi\rangle$ as e. Its square represents the probability.

In the previous section 5.1, we got to know the matrix multiplication. We learned that when we multiply a matrix with a vector, the result is another vector:

$$M \cdot |v\rangle = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} a \cdot v_0 + b \cdot v_1 \\ c \cdot v_0 + d \cdot v_1 \end{bmatrix}$$

We saw that the X-gate quantum operator $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ switches the amplitudes of the quantum state.

The X-gate applied to $|0\rangle$ results in $|1\rangle$:

$$X \cdot |0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

The X-gate applied to $|1\rangle$ results in $|0\rangle$:

$$X \cdot |1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \cdot 0 + 1 \cdot 1 \\ 1 \cdot 0 + 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

In the **Dirac** notation, a ket and a bra arranged like $|a\rangle\langle b|$ denotes the outer product. Therefore, we can interpret the outer product as a matrix multiplication:

$$|a\rangle\langle b| = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_0 & b_1 & \ldots & b_n \end{bmatrix} = \begin{bmatrix} a_0 \cdot b_0 & a_0 \cdot b_1 & \ldots & a_0 \cdot b_n \\ a_1 \cdot b_0 & a_1 \cdot b_1 & \ldots & a_1 \cdot b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_0 & a_n \cdot b_1 & \ldots & a_n \cdot b_n \end{bmatrix}$$

Therefore, the term $|a\rangle\langle b|$ denotes a matrix. And, we can write our matrices in terms of vectors:

$$X = |0\rangle\langle 1| + |1\rangle\langle 0| = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

In layman's terms (and only for specific base cases), the "ket-bra" $|a\rangle\langle b|$ turns your $|b\rangle$ into $|a\rangle$.

Accordingly, the X-gate turns $|1\rangle$ into $|0\rangle$ (because of $|0\rangle\langle 1|$) and it turns $|0\rangle$ into a $|1\rangle$ (because of $|1\rangle\langle 0|$).

We have talked a lot about the state of quantum superposition. But whenever we worked with a qubit in such a state, we initialized the qubit with the corresponding probability amplitudes α and β. But what if we wanted to put a once measured qubit back into superposition?

Now, we have some means to do it. What do you think about this?

$$H = |+\rangle\langle 0| + |-\rangle\langle 1|$$

According to our notion, it means we turn the state $|0\rangle$ into $|+\rangle$ and we turn the state $|1\rangle$ into $|-\rangle$.

Do you remember the states $|+\rangle$ and $|-\rangle$? We introduced them in section 3.2. They are defined as:

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}$$

These states yield the same probability of measuring 0 or 1. This is because they reside on the horizontal axis. But although these states share identical probabilities, they are different because the amplitude of state $|-\rangle$ is negative.

Let's have a look at this operator.

$$
\begin{aligned}
H &= |+\rangle\langle 0| + |-\rangle\langle 1| \\
&= \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & \frac{-1}{\sqrt{2}} \end{bmatrix} \\
&= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \\
&= \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
\end{aligned}
\tag{5.1}
$$

This operator is known as the Hadamard gate, or H-gate. It allows us to move away from the basis state vectors $|0\rangle$ and $|1\rangle$. It puts the qubit into a balanced state of superposition.

In short, it has the matrix:

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

> **ⓘ Why do we need to distinguish these two states?**
>
> In section 1.5, we mentioned the high precision with which quantum computers must work because quantum algorithms build on precise manipulations of continuously varying parameters. Therefore, even the noise caused by heat can ruin the computation.
>
> This is problematic because the computers we can build thus far are, essentially, expensive electric heaters that happen to perform a small amount of computation as a side effect.
>
> Our computers operate in a way that depends on the intentional loss of some information. When we look at the and operator, we get an output of 1 if both input values are 1. In all other cases, we get a 0. Given the output of 0, we have no way of knowing what the input was.

In the process of performing such an operator, the computer de-structively overwrites its input. Then, it physically destroys the old information by pushing it out into the computer's thermal environment. Thus, it becomes entropy that manifests as heat.

Quantum computers operate at shallow temperatures - below 1 kelvin or -273°C. As a result, quantum computers must be very energy efficient. Not because energy is a valuable resource. But because any loss of energy inevitably overheats the computer.

It is possible to carry out computations without losing informa-tion and thus, without producing heat. This is known as re-versible computation.

Enabling our H-operator to distinguish between the input states $|0\rangle$ and $|1\rangle$, it becomes reversible and, thus, suited for a quantum computer.

Why are there no states $\frac{-|0\rangle-|1\rangle}{\sqrt{2}}$ and $\frac{-|0\rangle+|1\rangle}{\sqrt{2}}$?

Let's say you have a qubit in state $\frac{-|0\rangle-|1\rangle}{\sqrt{2}}$. What does this mean? It means that α and β as in $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ are both negative $-\frac{1}{sqrt2}$.

α and β are the probability amplitudes. Their squares are the probabilities of measuring 0 or 1. Therefore, we get the same probabilities for $\alpha = \frac{1}{sqrt2}$ and $\alpha = -\frac{1}{sqrt2}$ (β accordingly).

Thus, there is no way to tell the difference between the states $\frac{-|0\rangle-|1\rangle}{\sqrt{2}}$ and $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$. And there is no way to tell the difference be-tween $\frac{-|0\rangle+|1\rangle}{\sqrt{2}}$ and $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$

> But how about $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ and $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$? Aren't these indistinguishable, too?
>
> Our newly introduced operator H proves the difference. While these two states do not differ in terms of their probabilities, they differ in computation. This is because they originate from two different inputs.

Let's see the *Hadamard* gate in action.

Listing 5.8: The Hadamard gate

```python
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Create a quantum circuit with one qubit
qc = QuantumCircuit(1)

# Define initial_state as |0>
initial_state = [1,0]
qc.initialize(initial_state, 0)

# apply the Hadamard gate to the qubit
qc.h(0)

 # Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# Do the simulation, returning the result
result = execute(qc,backend).result()

# get the probability distribution
counts = result.get_counts()

# Show the histogram
plot_histogram(counts)
```

Figure 5.8: The Hadamard gate

We have used and discussed most lines of this code listing before. However, you should note, we initialize our qubit with the state $|0\rangle$, in Python [1, 0]. The only new thing is the *Hadamard* gate we apply to our qubit at position 0 (line 13).

We can see that even though we initialized the qubit with the state $|0\rangle$, we measure it with a 50% probability for 0 and 1, each.

We mentioned the reversibility of the *Hadamard* gate. The *Hadamard* gate reverses itself.

In this code snippet, we initialize the qubit with state $|1\rangle$ (line 9). We apply the *Hadamard* gate two times. It results in a 100% chance of measuring 1. Exactly what the state $|1\rangle$ denotes.

The *Hadamard* gate is a fundamental quantum gate. It shows up everywhere in quantum computing. It turns a qubit from the state $|0\rangle$ into $|+\rangle$ and a qubit from the state $|1\rangle$ into the state $|-\rangle$. And it reverses these transformations.

Listing 5.9: The Hadamard gate reverses itself

```python
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Create a quantum circuit with one qubit
qc = QuantumCircuit(1)

# Define initial_state as |1>
initial_state = [0, 1]
qc.initialize(initial_state, 0)

# apply the Hadamard gate to the qubit
qc.h(0)

# apply the Hadamard gate again to reverse it
qc.h(0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# Do the simulation, returning the result
result = execute(qc,backend).result()

# get the probability distribution
counts = result.get_counts()

# Show the histogram
plot_histogram(counts)
```


Figure 5.9: The Hadamard gate reverses itself

5.3   If You Want To Gamble With Quantum Computing...

...ensure the probabilities to favor you

Are you into gambling? If yes, quantum computing is for you.

Figure 5.10: Gambling without a quantum computer

Because when you measure a qubit, what you observe depends on chance. Unless you measure it, the qubit is in a state of superposition of the states $|0\rangle$ and $|1\rangle$. But once you measure it, it will be either 0 or 1. If you measure a hundred qubits in the same state, you don't get the same result a hundred times. Instead, you'll get a list of 0s and 1s. The proportion of 0s and 1s you get corresponds to the probability distribution the qubit state entails.

In the last section 5.2, we got to know the *Hadamard* gate. It allows us to put a qubit into superposition. For instance, if you start with a qubit in the state

$|0\rangle$, applying the *Hadamard* gate results in a qubit in the state $|+\rangle$.

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

The resulting probability amplitudes for both states $|0\rangle$ and $|1\rangle$ are $\frac{1}{\sqrt{2}}$. Their squares denote the probabilities of measuring 0, respectively 1. Both probabilities are $\frac{1}{2}$. So, we got a 50:50 chance.

If you were to bet on either one outcome, there would be no good advice. But, if you played long enough, you'd end up with the same number of wins and losses–a fair game.

But if you were a casino, offering such a fair game wouldn't earn you any money. Instead, you'd need to increase your chance of winning. This is what casinos do. And this is the origin of the phrase "the bank always wins". For instance, the Wheel of Fortune and the popular slot machines disadvantage the players the most. These games have a house edge of 10 percent or more. But even in Blackjack, the fairest game if played optimal, there's a house edge of about 1 percent.

Listing 5.10: Weighted initial state

```python
from math import sqrt
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Define state |psi>
initial_state = [sqrt(0.4), sqrt(0.6)]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```

Figure 5.11: Probabilities of a weighted initial state
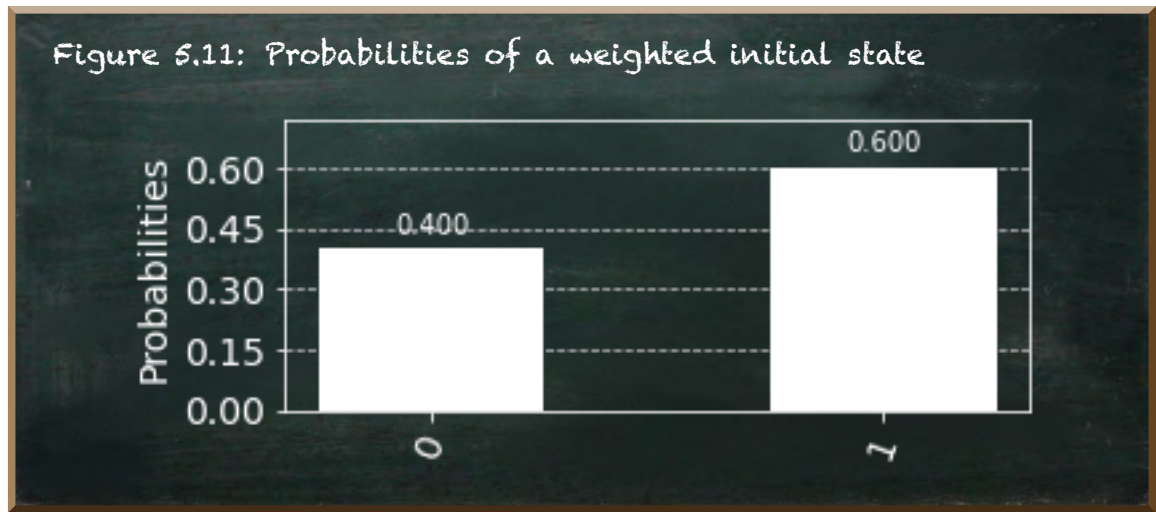
Let's say the casino wins when we measure 1, and the player wins when we measure 0. As the casino, we want to increase the chance of winning by 10% to win in 60% of the cases.

We already know one way. We can specify the probability amplitudes of the qubit during its initialization. For the probabilities are the squares of the probability amplitudes, we need to provide the square roots of the probabilities we want to specify (line 7).

But how can we change the probabilities of measuring 0 or 1 outside of the initialization?

In section 3.2, rather than specifying the exact probabilities, we controlled the probabilities by an angle θ (theta). This is the angle between the basis state vector $|0\rangle$ and the qubit state $|\psi\rangle$. θ controls the proximities of the vector head to the top and the bottom of the system (dashed lines). And these proximities represent the probability amplitudes whose squares are the probabilities of measuring 0 or 1 respectively. α^2 denotes the probability of measuring $|\psi\rangle$ as 0. β^2 indicates the probability of measuring it as 1.

We can deduct the values of α and β and thus the state $|\psi\rangle$:

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix}$$

In the previous section 5.2, we learned how we could use matrices to transform the state of a qubit. And we used the layman's interpretation that the "ket-bra" $|a\rangle\langle b|$ turns our qubit from the state $|b\rangle$ into the state $|a\rangle$.

So, why don't we use this interpretation to rotate our qubit state? θ is the an-

Figure 5.12: 2-dimensional qubit system
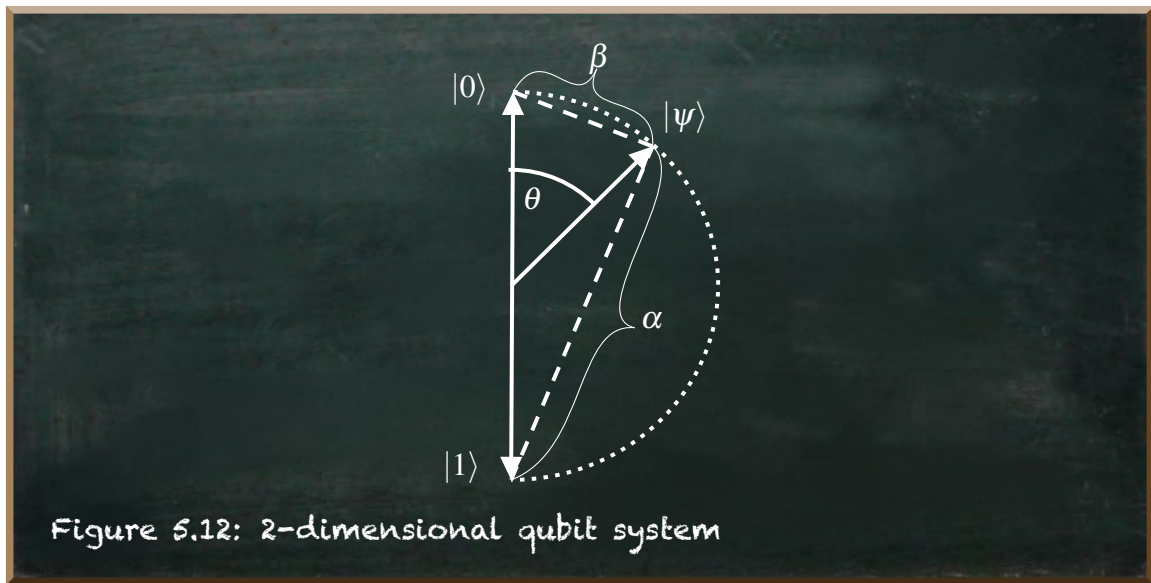
gle between the state $|0\rangle$ and the qubit state vector $|\psi\rangle$. Consequently, rotating $|0\rangle$ by θ turns it into $|\psi\rangle$. The ket-bra $|\psi\rangle\langle 0|$ denotes this part of our transformation.

The qubit state we name $|\psi'\rangle$ in the following image 5.13 depicts the rotation of the state $|1\rangle$ by θ. The ket-bra $|\psi'\rangle\langle 1|$ denotes this second part of our transformation.

Figure 5.13: Qubit rotation
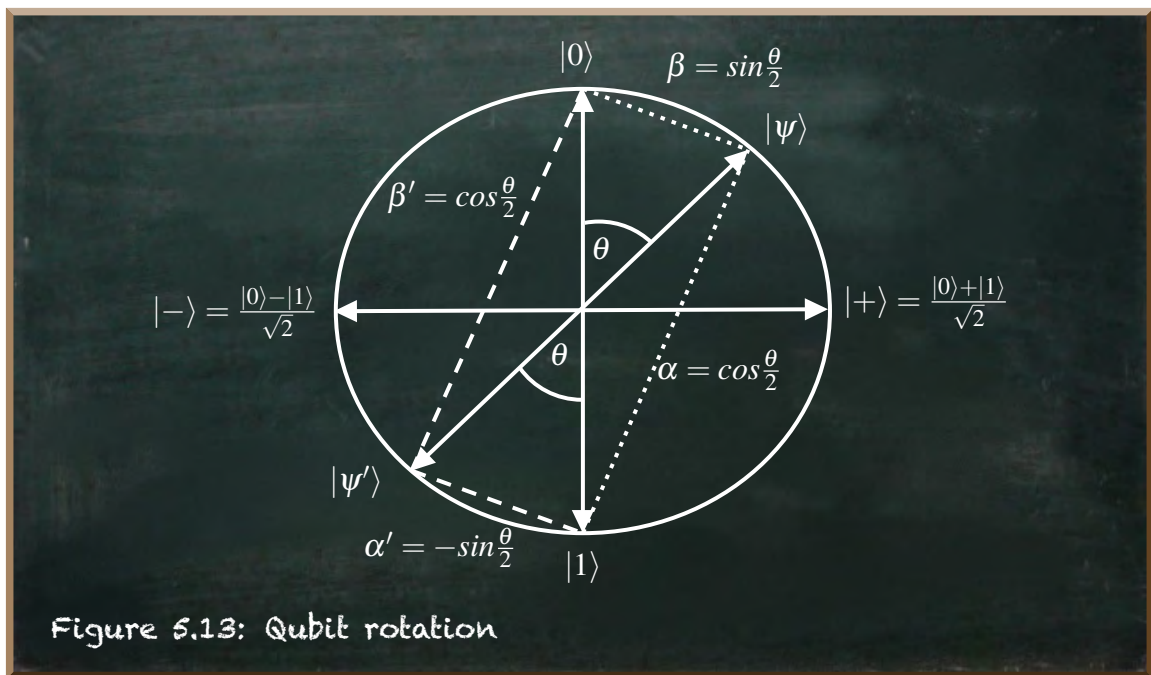
The following equation describes the rotation of our qubit:

$$R_y = |\psi\rangle\langle 0| + |\psi'\rangle\langle 1| = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} -sin\frac{\theta}{2} \\ cos\frac{\theta}{2} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$$

This matrix is known as the rotation matrix. The only quantum-specific here is that we take the *sin* and *cos* of $\frac{\theta}{2}$ rather than θ. The reason for this is the specific way we represent our qubit with the states $|0\rangle$ and $|1\rangle$ opposing each other on the same axis.

> (i) Usually, the rotation matrix implies a counter-clockwise rota-
> tion because in a standard representation, increasing angles
> "open" counter-clockwise. But the qubit state vector "opens"
> clockwise starting from the state $|0\rangle$. Therefore, the rotation
> matrix implies a clockwise rotation.

Another question that arises is why there is a $-sin\frac{\theta}{2}$ in the formula?

When you look at the figure 5.13, you can see that the qubit state $|\psi'\rangle$ ends at the left-hand side. The probabilities of states on that side equal the probabilities of states on the right-hand side (if mirrored on the vertical axis). But in the previous section 5.2, we also learned the importance of reversible transformations. So, we need to distinguish a clockwise rotation from a counter-clockwise rotation. As we need to distinguish whether we applied the *Hadamard* gate on the state $|0\rangle$ (resulting in $|+\rangle$) or on the state $|1\rangle$ (resulting in $|-\rangle$). It is the same justification.

But why do we specify a negative value for α' and not for β'? In section 3.2, we said we would interpret all vectors on the left-hand side of the vertical axis to have a negative value for β. While this is true, there is, in fact, no way to tell the difference between the states $\frac{-\alpha|0\rangle+\beta|1\rangle}{\sqrt{2}}$ and $\frac{\alpha|0\rangle-\beta|1\rangle}{\sqrt{2}}$. And when we look at a rotation matrix in a classical, two-dimensional vector space with orthogonal axes, we can see that it is the value for α' that is in the negative area, not the value for β'.

As you can see, the vector $|\psi'\rangle$ ends in the negative area of X (it is left to the y-axis). The distance to the y-axis is $sin\theta$. Therefore, the upper value (representing the x-coordinate) is negative.

Using the same rotation matrix for our quantum system, we use a formula many mathematicians are familiar with.

Figure 5.14: Rotation matrix
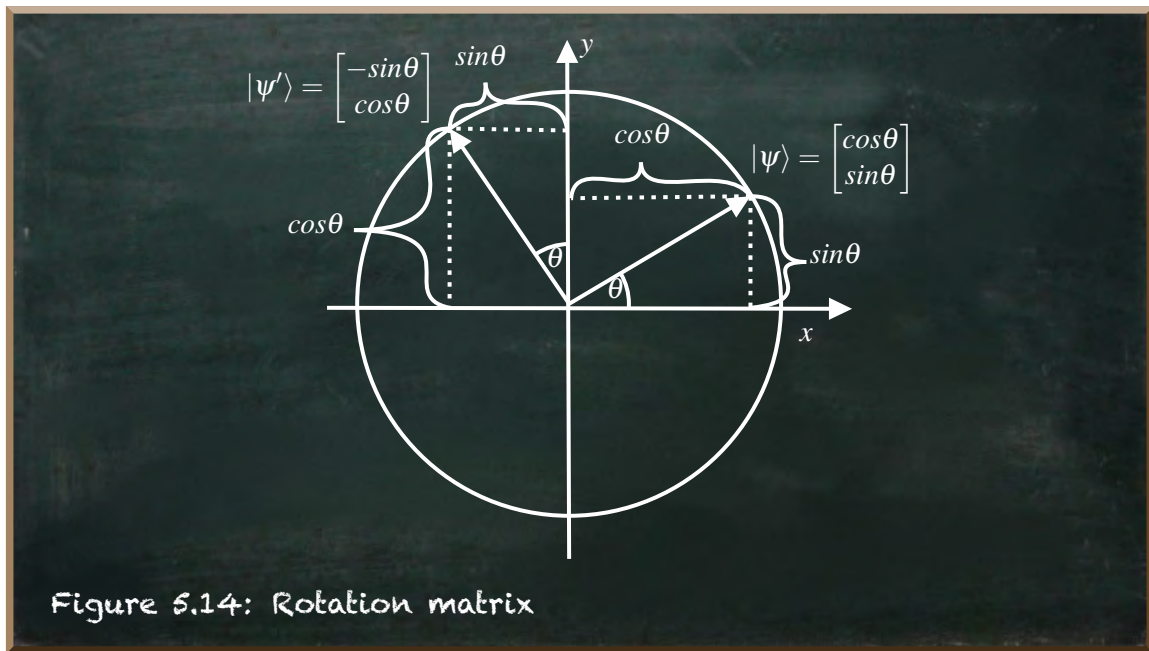
Let's have a look at our transformation in action.

Listing 5.11: Rotate the qubit state

```python
from math import pi

# Define state |0>
initial_state = [1, 0]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# Rotate the state by a quarter of a half circle.
qc.ry(pi/4,0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```

Figure 5.15: Probabilities of the rotated qubit state
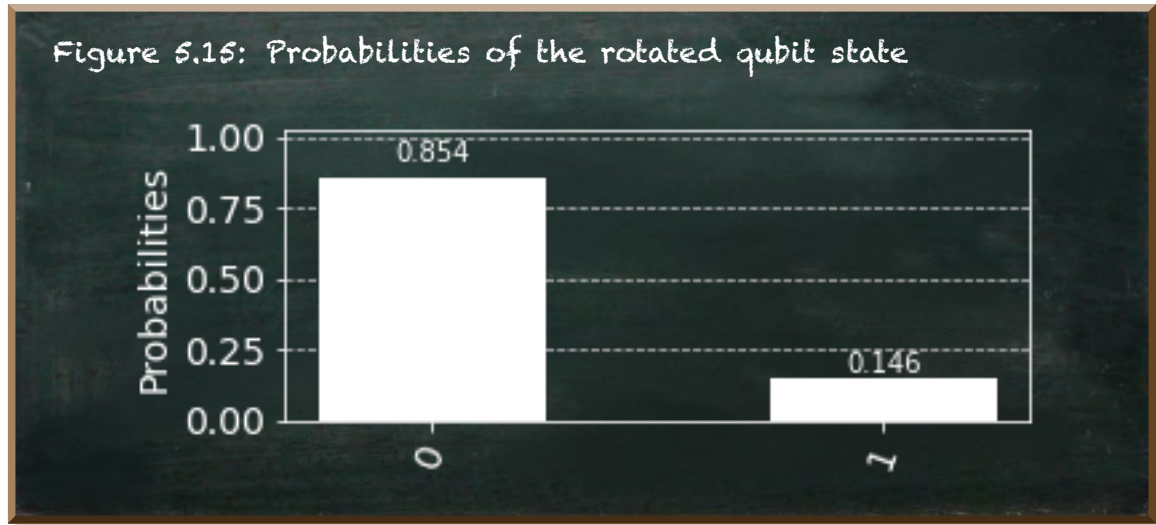
The Qiskit `QuantumCircuit` object provides the `ry` function (line 13). `ry` is for R_y gate. Because it rotates the qubit around the y-axis of the quantum system, this function takes the angle θ (in Radians) as the first parameter. The value of `2*pi` denotes a full rotation of 360°. The second parameter of the function is the position of the qubit to apply the gate to.

However, you need to be careful. The angle θ does not stop when it "reaches" the state $|1\rangle$. You can rotate your qubit state beyond it. Then, rather than increasing the probability of measuring 1 you decrease it.

The R_y gate is easily reversible. Apply another R_y gate with $-\theta$ as the parameter.

We started with the goal to increase the casino's chance to win by 10%. What is 10% in terms of the angle θ?

θ denotes the angle between the basis state $|0\rangle$ and $|\psi\rangle$. From our quantum state formula, $|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix}$, we can see that we have a probability amplitude for the state $|1\rangle$ of $sin\left(\frac{\theta}{2}\right)$. Thus, the probability of measuring a qubit in the state $|\psi\rangle$ as a 1 is the squared probability amplitude.

$$sin^2\left(\frac{\theta}{2}\right) = P_1(\psi) \tag{5.2}$$

Let's solve this equation for the angle θ.

$$sin\left(\frac{\theta}{2}\right) = \sqrt{P_1(\psi)}$$
$$\frac{\theta}{2} = sin^{-1}\sqrt{P_1(\psi')} \tag{5.3}$$
$$\theta = 2 \cdot sin^{-1}\sqrt{P_1(\psi')}$$

This formula shows the angle θ that represents the probability of measuring $|\psi\rangle$ as a 1.

The following function `prob_to_angle` implements this equation in Python. It takes a probability to measure the qubit as a 1 and returns the corresponding angle θ.

Listing 5.12: Calculate the angle that represents a certain probability

```python
from math import asin

def prob_to_angle(prob):
    """
    Converts a given P(psi) value into an equivalent theta value.
    """
    return 2*asin(sqrt(prob))
```

Let's use this function to set the probability of measuring our qubit as a 1 to 60%.

We initialize our qubit with the state $|0\rangle$ (line 4). Then, we apply the R_y gate on the qubit and pass as the first parameter the result of calling `prob_to_angle` with the probability value of `0.6` (line 13). The rest of the code remains unchanged.

As a result, we see a 60% chance to measure the qubit as the value 1. We have found an effective way to control the probabilities of measuring 0 and 1, respectively.

Let's see what happens if we apply the R_y gate on a qubit in another state, for instance, in

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

Listing 5.13: Rotate the qubit state by 0.8

```python
from math import pi, sqrt

# Define state |0>
initial_state = [1,0]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# Rotate the state by 60%
qc.ry(prob_to_angle(0.6), 0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```


Figure 5.16: Probabilities of the rotated qubit state
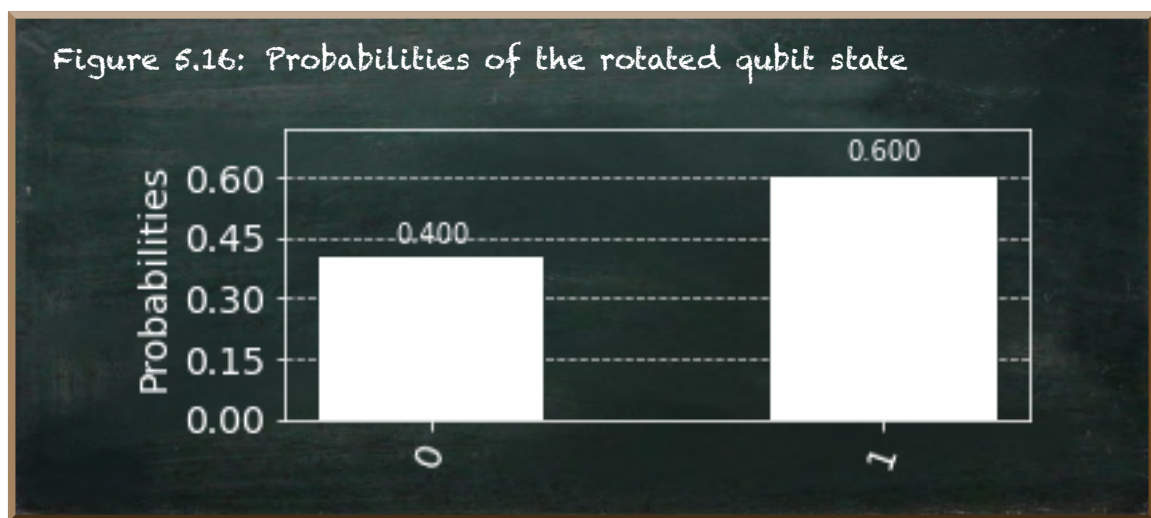
In the following example, we initialize the qubit in the state $|+\rangle$. It has a probability of 50% measuring the qubit in either state 0 or 1 (line 4). And we rotate it by the angle we calculate from the probability of 10% (line 13).

Listing 5.14: Rotate the qubit state with initial state

```python
from math import pi, sqrt

# Define state |+>
initial_state = [1/sqrt(2), 1/sqrt(2)]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# Rotate the state by 10%
qc.ry(prob_to_angle(0.1), 0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```


Figure 5.17: Probabilities of the rotated qubit state
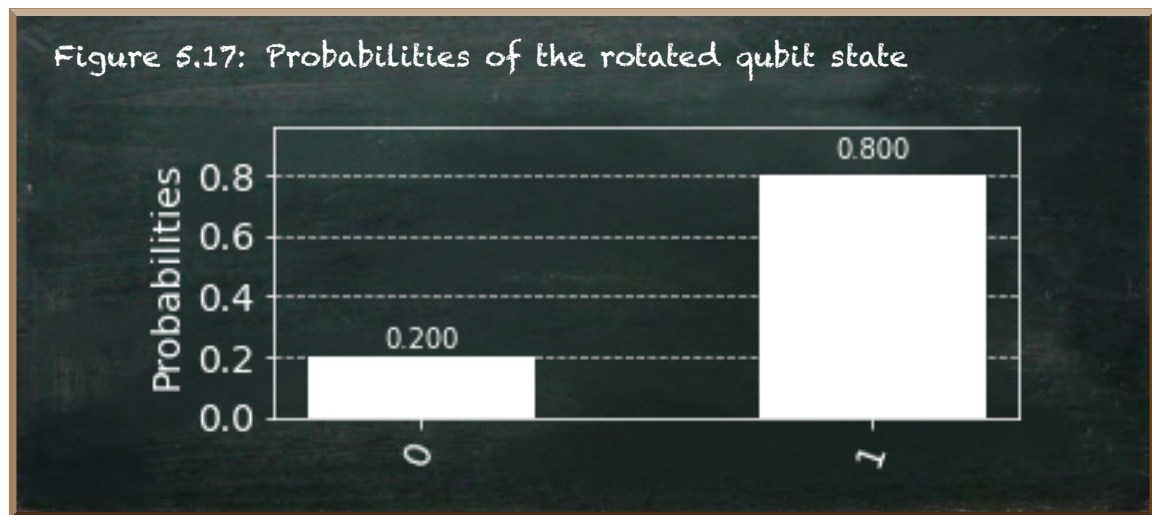
Wait, this is not correct. We get an 80% chance of measuring the qubit as a 1. But we would have expected only 60%.

The problem is how we calculated the angle θ from the probability it repre-

sents. θ is the angle between the vector $|\psi\rangle$ and the basis state vector $|0\rangle$. But the gradients of trigonometric functions (such as sine and arcsine) are not constant. Thus, the probability an angle represents that starts at the top of the circle (state $|0\rangle$) is another probability that the same angle represents that starts at the horizontal axis such as the state $|+\rangle$.

We can fix this. We calculate the overall angle θ that represents the sum of the prior probability and the probability we want our qubit to change ($2 * asin\sqrt{prob + prior}$). We subtract from it the angle that represents the prior ($-2 * asin\sqrt{prior}$). The result is the angle that represents the probability change at the current state of the qubit.

Listing 5.15: Rotate the qubit state correctly

```python
from math import asin

def prob_to_angle_with_prior(prob, prior):
    """
    Converts a given P(psi) value into an equivalent theta value.
    """
    return 2*asin(sqrt(prob+prior))-2*asin(sqrt(prior))

# Define state |+>
initial_state = [1/sqrt(2), 1/sqrt(2)]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# Rotate the state by 10%
qc.ry(prob_to_angle_with_prior(0.1, 0.5), 0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```

Figure 5.18: Probabilities of the rotated qubit state
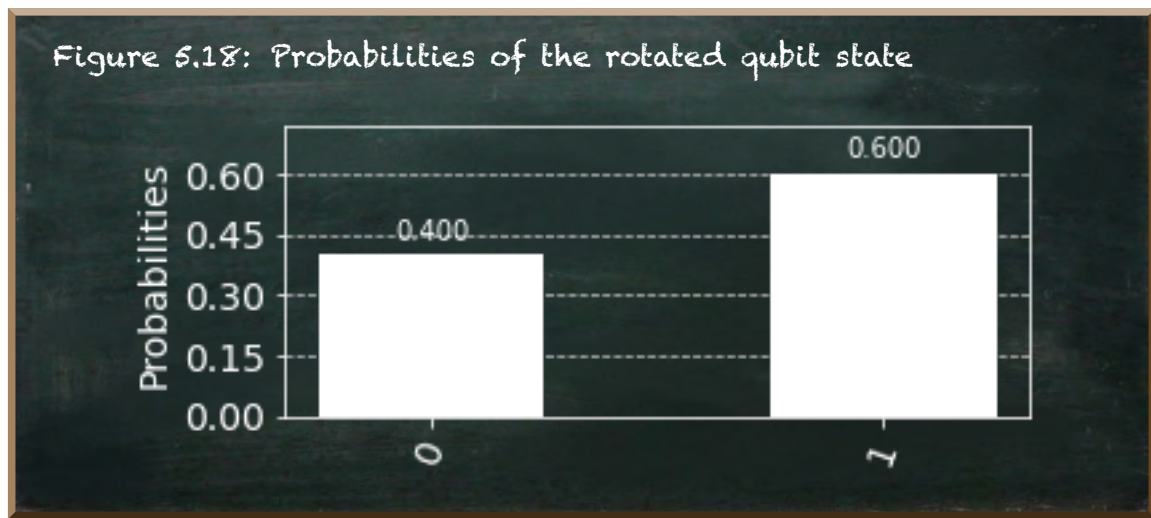
We write a new function `prob_to_angle_with_prior` (lines 3-7). This function takes the probability we want our qubit to change as the first parameter. And it takes the prior probability of the qubit as the second parameter.

When we run the code, we see the result we expected.

Rotating the qubit around the y-axis allows you to control the probabilities of measuring 0 and 1 by the angle θ. And you can represent θ by the change of probability of measuring the qubit as 1 ($P_1(\psi')$) and by the prior probability of measuring 1 ($P_1(\psi)$)

$$\theta = 2 \cdot sin^{-1}\sqrt{P_1(\psi)} - 2 \cdot cos^{-1}\sqrt{P_1(\psi')}$$

But for once, this is not an all-around carefree way to push probabilities in a certain direction. While you can rotate the angle θ further and further, the effect that it has on the resulting probabilities depend on the direction of your qubit state vector $|\psi\rangle$. If the vector $|\psi\rangle$ points to the right-hand side of the y-axis, rotating it by θ increases the probability of measuring 1. But if the vector $|\psi\rangle$ points to the left-hand side of the y-axis, rotating it by θ decreases the probability of measuring 1.

In quantum computing, where you go always depends on where you come from.

To the second, even more importantly, from a practical perspective, if you want to change the measurement probabilities by a certain percentage, you need to know the prior probabilities. You need to know the state the qubit is in. Remember, measuring the qubit collapses it to either 0 or 1. Measuring destroys the qubit superposition. But, if you're not allowed to measure the qubit, how could you specify the prior probability?

In the trivial examples we used in this section, we can keep track of the qubit states by hand. But for any meaningful quantum circuit, this becomes impossible. Thus, the advantage of a quantum circuit over a classical algorithm builds upon the qubit's ability to explore states we can't trace classically.

To succeed beyond the traceable manipulation of qubit states, we need to work with multiple qubits concurrently. When we combine qubits, more sophisticated transformations become possible. In the next chapter, we will explore what it means to work with multiple qubits.

6. Working With Multiple Qubits

6.1 Hands-On Introduction To Quantum Entanglement

Figure 6.1: Albert Einstein
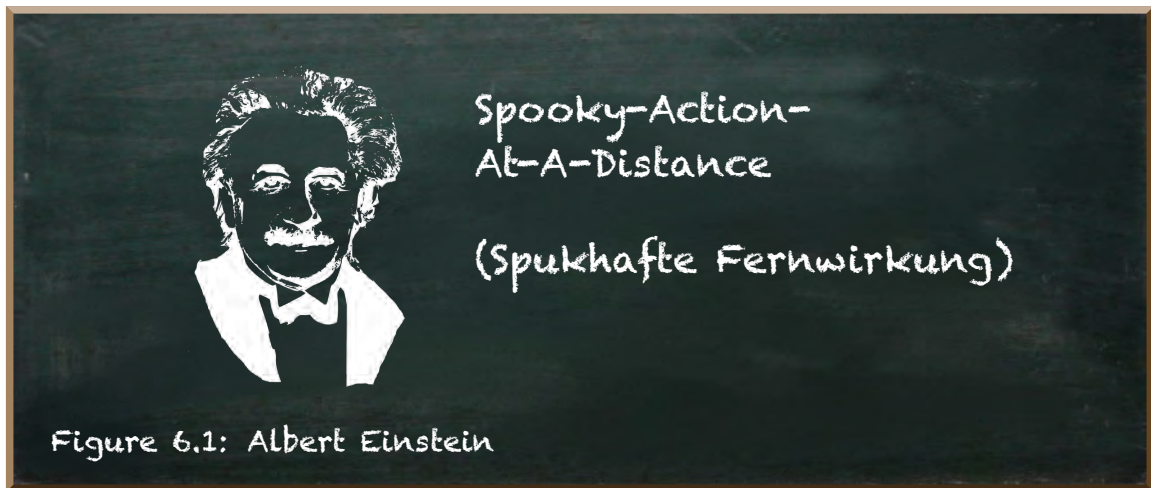
Thus far, you may wonder what the big deal with quantum computing is. Why does everyone seem to be freaking out about this?

Some emphasize the notion of the qubit being in the two states concurrently. And that's so different from anything we know. In the world we live in, the world we experience, there is no such thing that is in two mutually exclusive states at the same time.

Others counter this notion is wrong. The qubit is not 0 and 1 at the same time.

Rather, it is a truly random system. And that's so different from anything we know. In the world we live in, the world we experience, there is no truly random thing. Everything is sensitively dependent on initial conditions. If you were able to measure everything with absolute precision, randomness would disappear.

Again others object this notion is wrong, too. But the whole concept of quantum superposition is so different from anything we know. In the world we live in, the world we experience, there is nothing comparable. So any analogy to something we know is simply inadequate.

But thus far, the only aspect of the quantum superposition we covered is the probabilities of measuring it as either 0 or 1. Yes, it is interesting. It may even seem a little strange. But a system whose value depends on chance is not unimaginable. Thus far, it doesn't matter if the qubit is in both states concurrently, purely random, or something completely different. Thus far, it is a probabilistic system. Not more. Not less.

But, so far, we only considered a single qubit. It's going to change if we start to work with multiple qubits.

We already know some operations that work with multiple **classical** bits. For instance, and and or. A look at the truth tables discloses they are inappropriate for qubit transformation. They are irreversible.

Figure 6.2: Truth table of AND and OR

While there are two input values (P and Q), either and or or on its own has only one output value. It is impossible to reconstruct the two input bits if you only got one output bit as information. Thus, when we work with two qubits, any operator (transformation gate) must have two inputs and two outputs.

Can we use and and or as the two outputs?

No, we can't. These two operations don't allow us to tell the difference between the states in which either one of P and Q is true and the other false.

Let's try something different. We replace the and-operator with the plain and

unchanged value of P. We can now tell the difference between the two states

- P is true, and Q is false
- P is false, and Q is true.

Figure 6.3: Truth table of AND and P

But we can't tell the difference between the state when P and Q are true and the state when only P is true anymore.

The reason is, both operations and and or, are imbalanced. And is false in three cases. Or is true in three cases. The other output bit of the transformation would need to tell the difference between the three cases. That's impossible for a single bit of information.

So, we also need to replace or. We can use the "exclusive or" (XOR) operator instead. In math, the symbol $\oplus$ represents the "exclusive or" operator. It is true, for precisely one of its inputs is true. Otherwise, it is false. The following truth table depicts the definition of "exclusive or."

Figure 6.4: Truth table of exclusive or

The combination of P and $P \oplus Q$ is reversible. It is not just reversible, but it inverses itself. If we apply it twice, the output is P and Q again.

At first sight, there's nothing special about this transformation. We can even draw a classical circuit diagram.

Figure 6.5: The combination of P and P XOR Q is reversible

Figure 6.6: A classical circuit with P, P XOR Q

The dot is the fan-out operation. In this classical circuit, where the lines are wires, it copies the value of P. We interpret voltage at the wire as 1 and the absence of voltage as 0. If we connect a wire with another wire, it receives the same output at both ends. One wire coming from P connects to the XOR gate. The other serves as the output. It is the unchanged value of P.

In quantum computing, the situation is different. First, it is impossible to copy a qubit (we'll cover this topic later in this chapter). Second, the transformation does not provide an unchanged P as an output.

Therefore, we use a different representation of this transformation in quantum computing.

Figure 6.7: The quantum CNOT gate

While there is a fan-out at the P qubit, it does not imply copying the qubit. Instead, the fan-out indicates that the qubit P controls the transformation of the target qubit Q. To understand what this means, let's apply a different perspective on the truth table. We split it into two blocks.

Figure 6.8: Truth table of the CNOT gate

In the first block, P is 0, and $P \oplus Q$ is equal to Q. So, nothing changes at all.

But in the second block, when P is 1, $P \oplus Q$ is equal to $\neg Q$ ("not Q"). In other words, if P is 1, we apply the quantum X-gate on Q.

The qubit P controls whether we apply an X-gate on the qubit Q. Therefore, this gate is named the "controlled not" or CNOT-gate. In Qiskit, it is the cx-gate.

The following code shows the CNOT-gate in action.

Listing 6.1: Apply the CNOT-gate with |0> as control qubit

```python
from math import sqrt
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram

# Redefine the quantum circuit
qc = QuantumCircuit(2)

# Initialise the qubits
qc.initialize([1,0], 0)
qc.initialize([1,0], 1)

# Apply the CNOT-gate
qc.cx(0,1)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```


Figure 6.9: Result of the CNOT-gate with |0> as control qubit
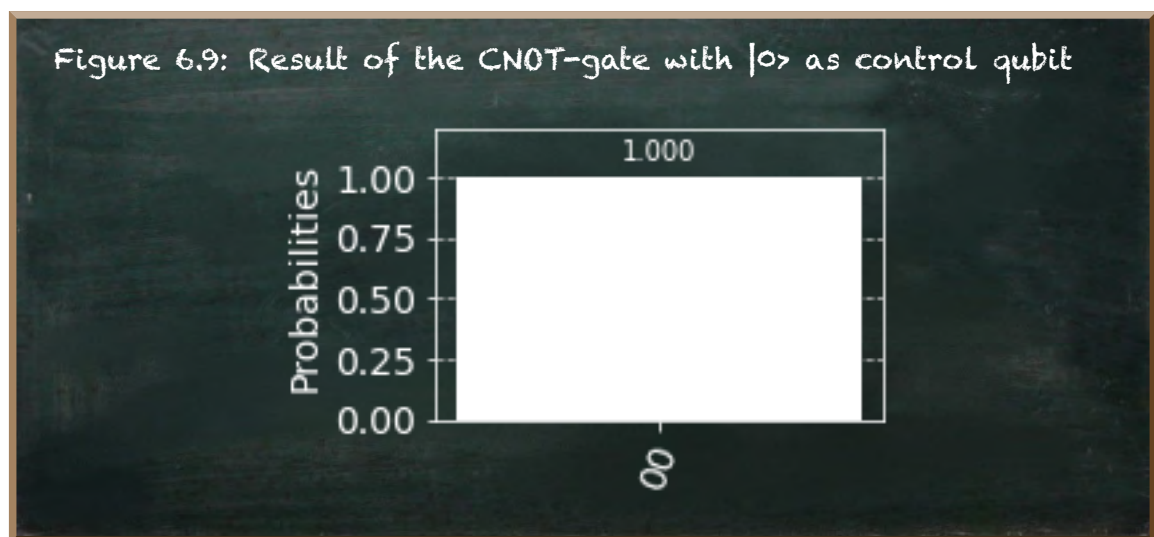
When we initialize both qubits with $|0\rangle$ (lines 9-11) before we apply the CNOT-gate (line 13), we always measure 00. Nothing happens.

When we initialize the control qubit with $|1\rangle$ and the target qubit with $|0\rangle$, we always measure 11.

Listing 6.2: Apply the CNOT-gate with |1> as control qubit

```python
# Redefine the quantum circuit
qc = QuantumCircuit(2)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize([0,1], 0)
qc.initialize([1,0], 1)

# Apply the CNOT-gate
qc.cx(0,1)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```


Figure 6.10: Result of the CNOT-gate with |1> as control qubit

When we only look at the basis states, there is still nothing special going on here. The result equals the result that a classical circuit produces.

But it becomes interesting when the control qubit is in a state of superposi-

tion. We initialize both qubits in the state $|0\rangle$, again. Then, the Hadamard
gate puts the qubit Q into the state $|+\rangle$. When measured, a qubit in this state
is either 0 or 1, with a probability of 50% each. The following figure depicts
the quantum circuit diagram.

Figure 6.11: A CNOT-gate with a control qubit in state |+>
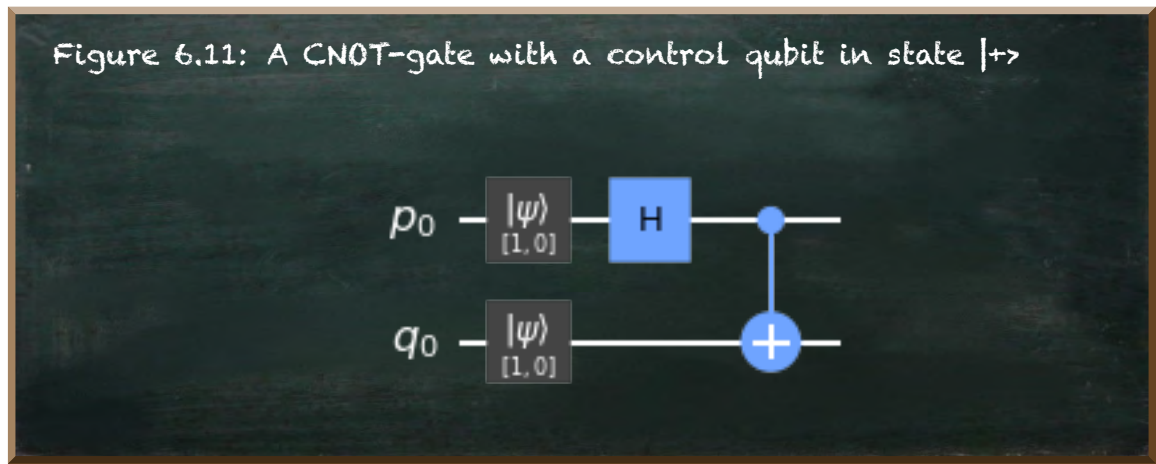
Listing 6.3: Apply the CNOT-gate with |+> as control qubit

```
 1  # Redefine the quantum circuit
 2  qc = QuantumCircuit(2)
 3
 4  # Initialise the 0th qubit in the state `initial_state`
 5  qc.initialize([1,0], 0)
 6  qc.initialize([1,0], 1)
 7
 8  # Apply the Hadamard gate
 9  qc.h(0)
10
11  # Apply the CNOT−gate
12  qc.cx(0,1)
13
14  # Tell Qiskit how to simulate our circuit
15  backend = Aer.get_backend('statevector_simulator')
16
17  # execute the qc
18  results = execute(qc,backend).result().get_counts()
19
20  # plot the results
21  plot_histogram(results)
```

Figure 6.12: Result of the CNOT-gate with $|+\rangle$ as control qubit
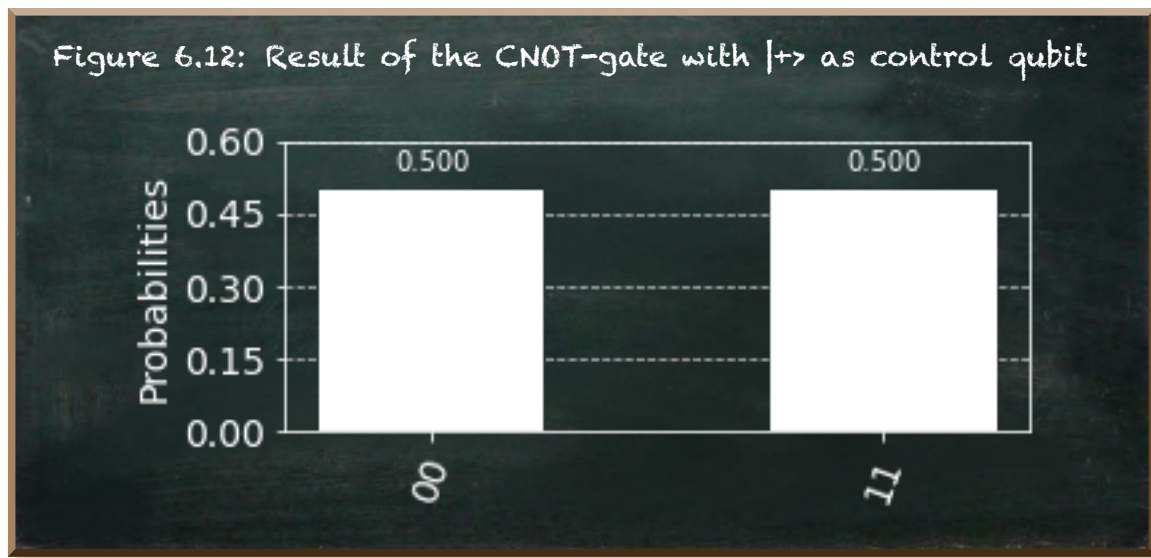
We measure the control qubit P as either 0 or 1 with a probability of 50% each. This is exactly what we expect for a qubit in the state $|+\rangle$. And, we measure the target qubit Q as either 0 or 1, too. Its value perfectly matches the first qubit.

"*Of course it does!*" you may think. If the control qubit P is 0, we leave the target qubit Q untouched in its state $|0\rangle$. We measure it as 0. But if the control qubit is 1, we apply the X-gate on the qubit Q. We turn it from $|0\rangle$ into $|1\rangle$ and measure it as 1.

"*There is a true causal relationship,*" you may think, "*just like in a classical circuit.*" If it was, what if we measured the target qubit Q first?

The control qubit P is in a state of superposition unless you measure it. It controls whether we apply an X-gate on the second qubit Q. If there were a cause and an effect, how could we see the effect before the cause?

Let's have a look.

Listing 6.4: Measure the controlled qubit first

```python
from qiskit import ClassicalRegister, QuantumRegister

# Prepare a register of two qubits
qr = QuantumRegister(2)

# Prepare a register of two classical bits
cr = ClassicalRegister(2)

# Redefine the quantum circuit
qc = QuantumCircuit(qr, cr)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize([1,0], 0)
qc.initialize([1,0], 1)

# Apply the Hadamard gate
qc.h(0)

# Apply the CNOT-gate
qc.cx(0,1)

# Measure the qubits to the classical bits, start with the controlled
    qubit
qc.measure(qr[1], cr[1])
qc.measure(qr[0], cr[0])

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('qasm_simulator')

# execute the qc
results = execute(qc,backend,shots = 1000).result().get_counts(qc)

# plot the results
plot_histogram(results)
```

Figure 6.13: Result of measuring the controlled qubit first
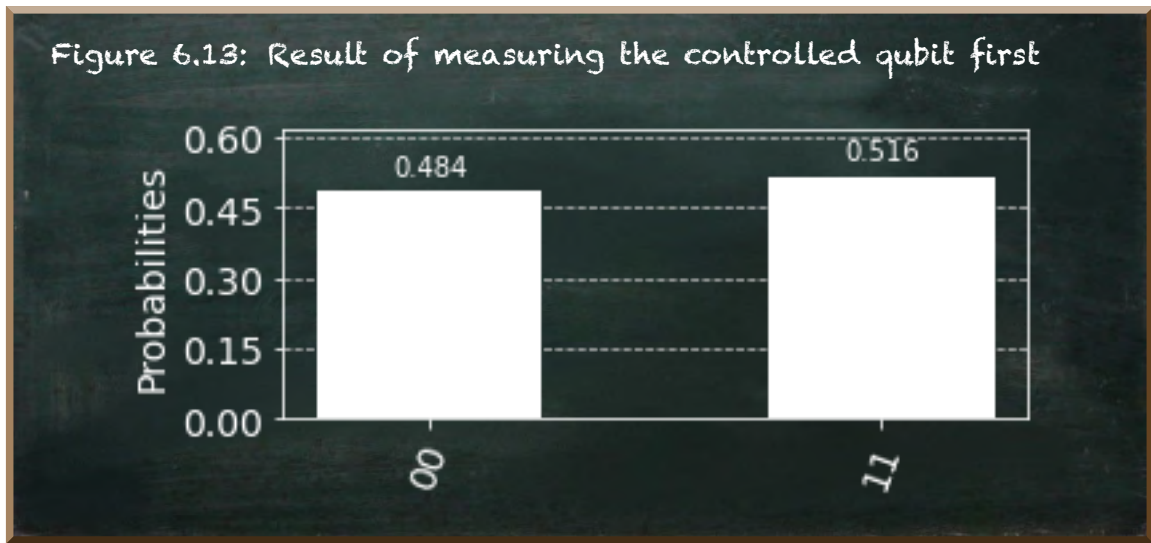
To specify the order of measurement, we need to edit our code a little bit. First, we create two registers (lines 4 and 7) and initialize the `QuantumCircuit` with them (line 10). A `QuantumRegister` is a collection of qubits. A `ClassicalRegister` is a collection of regular bits. The registers allow us to combine these two kinds of bits in a single circuit. The classical bits take the measurement results of the qubits (lines 23-24).

This time, we choose another backend simulator. We use the `qasm_simulator` (line 27) because it supports multiple executions of a quantum circuit. The `statevector_simulator` we used thus far is ideal for the examination of qubits in a state of superposition. But it only supports a single execution of the circuit. But this time, we include the measurement in our circuit that collapses the state of superposition. As a result, we receive a single pair of regular bits whose measurement probability is always 100%.

To investigate the probability of the underlying quantum system, we have to execute the circuit several times, for which we use the `qasm_simulator`. The parameter `shots=1000` (line 30) specifies the number of executions we want to run.

Since we do not calculate the real probabilities but retrieve them empirically, the result is not entirely accurate. But it is close enough.

The measured values of both qubits stay perfectly aligned.

If we measured only the target qubit, it would appear to be random despite its initialization with the state $|0\rangle$. But once we look at the control qubit, we see that both values are equal. Always.

It does not matter which qubit we measure first. It seems as if the other qubit knows the outcome and chooses its state accordingly. The measurement of one qubit affects the other qubit. But it only appears that way.

In a classical circuit, the first bit remains unchanged. Wires connect both bits physically and there is a clear causal relationship. The input voltage (or its absence) of the control bit determines the output voltage of both wires. It directly determines the output of the directly connected wire, and it determines the output of the "exclusive or" wire (together with the other input voltage).

But unlike its classical counterpart, the CNOT-quantum gate does not output an unchanged qubit P alongside the qubit Q whose output is $P \oplus Q$. It outputs an **entangled** pair of qubits. They are in a state of superposition. Once you measure any part of this entangled quantum system, the whole system collapses.

This happens without any information exchange between the entangled qubits. Because qubits are not physically connected. We could even separate them by a large distance, and still, measuring one qubit would collapse them both. Einstein did not believe this. He termed this phenomenon as "*spooky action at a distance.*"

But this is the point. The CNOT-gate does not change the value of the target qubit Q depending on the control qubit P. But it entangles the two qubits. It puts them into a shared state–**an entangled state**.

While we can describe the quantum state of an entangled system as a whole, we can't describe it independently per single qubit anymore.

"***But how is that useful at all?***" you ask?

It is helpful because it enables us to construct a quantum system beyond a single qubit. A qubit is a probabilistic system that collapses to either 0 or 1. Entangled qubits can collapse to a broader range of values.

Even more importantly, a set of entangled qubits can represent the problem at hand more accurately. A set of qubits can represent the structure of this problem to be solved. Once the quantum circuit concerts all qubits in a way that represents the problem, then a single measurement collapses the whole system. And the measured values disclose the solution.

In classical computing, we think a lot about cause and effect. Given some input, which transformations do we need to apply to produce a certain output? The desired output.

In quantum computing, we think about the structure of the problem. Given the specificities of the problem, which transformations do we need to concert the qubits so that, when measured, results in the solution?

Working with multiple qubits and entanglement are fundamental building blocks of quantum computing. And they have many facets.

In the remainder of this chapter, we shed some light on the underlying math, the theoretical consequences, and the proof that the quantum superposition is, in fact, different from a classical system that appears random but is sensitively dependent on initial conditions. While I try to explain all these topics as practical and accessible, they remain pretty theoretical.

If you feel ready for this deep dive into entanglement, then just read on. However, if you prefer to continue with the practical consequences of entanglement on quantum machine learning algorithms, then you may want to jump to section 6.3. In that case, I recommend you come back to this chapter later. While the theoretic background is not necessary to apply quantum gates on multiple qubits, it undoubtedly fosters a deeper understanding and prepares you for the upcoming challenges.

6.2 The Equation Einstein Could Not Believe

Albert Einstein colorfully rejected the idea of quantum entanglement as "spooky-action-at-a-distance."

In layman's terms, quantum entanglement is the ability of distributed particles to share a state–a state of quantum superposition, to be precise.

Doesn't it sound spooky? Maybe we should refresh the notion of superposition.

Particles have a spin. Up or down. The direction of the spin is not determined until you measure it. But once you measure it, it will instantly collapse to either one spin direction for you to observe. This is the superposition of a single particle.

Quantum entanglement says two particles can share a state of superposition. Their spins correlate. Once you measure one particle's spin, the state of the other particle changes immediately.

Doesn't it sound spooky? Maybe we should talk about scales.

When we say the two particles are distributed, then they can be direct neighbors within the same atom. They can be a few feet away from each other. But they can also be light-years apart. It doesn't matter!

When we say the state of the particle changes instantly, we mean instantly. Not after a few seconds. Not after a tiny fraction of a second. But instantly.

The two particles can be light-years away from each other, yet when we measure one of them, the other changes its state simultaneously.

Sounds spooky, right?

"But how do we know?"

We have not tested such a setting with particles light-years away. But we know the underlying math.

Long before the first experiment provided evidence, a group of geniuses developed formulae that predicted how an entangled pair of particles would behave. Einstein was one of them. And while he was able to understand the language of math like no one else could (very few could, maybe), he didn't like what math told him this time.

6.2.1 Single Qubit Superposition

In quantum mechanics, we use vectors to describe the quantum state. A popular way of representing quantum state vectors is the Dirac notation's "ket"-construct that looks like $|\psi\rangle$.

In a quantum system with two values that we could measure, such as the particle spin that can be up or down, or the quantum bit (qubit) that can be either 0 or 1, there are two basis vectors.

For the quantum bit, these are: $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

The quantum superposition is a combination of these two basis states.

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

The values α and β are the probability amplitudes. Their squares denote the probabilities of measuring the qubit as a 0 (α^2) or a 1 (β^2). The larger α, the larger the probability is to measure the qubit as 0. The larger β, the larger the probability is to measure the qubit as 1.

Since the probabilities must add up to 1, we can say that their sum must be 1.

$$|\alpha|^2 + |\beta|^2 = 1$$

6.2.2 Quantum Transformation Matrices

In quantum mechanics, we also use vectors to transform qubit states. The Dirac notation's "bra"-construct ($\langle 0|$) represents a row vector. When we multiply a column vector with a row vector, we build the outer product. It results in a matrix, like this

$$|a\rangle\langle b| = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_0 & b_1 & \cdots & b_n \end{bmatrix} = \begin{bmatrix} a_0 \cdot b_0 & a_0 \cdot b_1 & \cdots & a_0 \cdot b_n \\ a_1 \cdot b_0 & a_1 \cdot b_1 & \cdots & a_1 \cdot b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_0 & a_n \cdot b_1 & \cdots & a_n \cdot b_n \end{bmatrix}$$

So, we can create matrices from vectors. For instance, we can make three simple matrices.

- The Identity (I) matrix

$$I = |0\rangle\langle 0| + |1\rangle\langle 1| = \begin{bmatrix} 1 \cdot 1 & 1 \cdot 0 \\ 0 \cdot 1 & 0 \cdot 0 \end{bmatrix} + \begin{bmatrix} 0 \cdot 0 & 0 \cdot 1 \\ 1 \cdot 0 & 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- The Not (X) matrix

$$X = |0\rangle\langle 1| + |1\rangle\langle 0| = \begin{bmatrix} 1 \cdot 0 & 1 \cdot 1 \\ 0 \cdot 0 & 0 \cdot 1 \end{bmatrix} + \begin{bmatrix} 0 \cdot 1 & 0 \cdot 0 \\ 1 \cdot 1 & 1 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- The Hadamard (H) matrix

$$H = |+\rangle\langle 0| + |-\rangle\langle 1|$$

$$= \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & \frac{-1}{\sqrt{2}} \end{bmatrix}$$

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

$$= \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

6.2.3 Transforming Single Qubits

When we multiply a matrix with a column vector (our quantum state), the result is another column vector, like this:

$$M \cdot |v\rangle = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} a \cdot v_0 + b \cdot v_1 \\ c \cdot v_0 + d \cdot v_1 \end{bmatrix}$$

When we multiply the I-gate matrix with a vector, we get the unchanged vector as the output.

$$I \cdot |\psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 1 \cdot \alpha + 0 \cdot \beta \\ 0 \cdot \alpha + 1 \cdot \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

The X-gate matrix flips the probability amplitudes of the vector.

$$X \cdot |\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 \cdot \alpha + 1 \cdot \beta \\ 1 \cdot \alpha + 0 \cdot \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

The H-gate puts a qubit from a basis state into superposition.

$$H \cdot |0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 - 1 \cdot 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

6.2.4 Two-Qubit States

Now, let's say we have two qubits. Let's call them $|a\rangle$ and $|b\rangle$. Each of the two qubits has its own probability amplitudes: $|a\rangle = a_0|0\rangle + a_1|1\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$ and $|b\rangle = b_0|0\rangle + b_1|1\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$. When we look at these two qubits concurrently, there are four different combinations of the basis states. Each of these combinations has its probability amplitude. These are the products of the probability amplitudes of the two corresponding states.

- $a_0|0\rangle b_0|0\rangle$
- $a_0|0\rangle b_1|1\rangle$
- $a_1|1\rangle b_0|0\rangle$
- $a_1|1\rangle b_1|1\rangle$

These four states form a quantum system on their own. Therefore, we can represent them in a single equation. While we are free to choose an arbitrary name for the state, we use $|ab\rangle$ because this state is the collective quantum state of $|a\rangle$ and $|b\rangle$.

$$|ab\rangle = |a\rangle \otimes |b\rangle = a_0 b_0|0\rangle|0\rangle + a_0 b_1|0\rangle|1\rangle + a_1 b_0|1\rangle|0\rangle + a_1 b_1|1\rangle|1\rangle$$

In this equation, $|ab\rangle$ is an arbitrary name. The last term is the four combinations reordered to have the amplitudes at the beginning. But what does $|a\rangle \otimes |b\rangle$ mean?

The term $|a\rangle \otimes |b\rangle$ is the tensor product of the two vectors $|a\rangle$ and $|b\rangle$.

The tensor product (denoted by the symbol $\otimes$) is the mathematical way of calculating the amplitudes. In general, the tensor product of two vectors v and w is a vector of all combinations. Like this:

$$\text{With } v = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_n \end{bmatrix} \text{ and } w = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} \text{ then } v \otimes w = \begin{bmatrix} v_0 w_0 \\ v_0 w_1 \\ \vdots \\ v_0 w_n \\ v_1 w_0 \\ v_1 w_1 \\ \vdots \\ v_1 w_n \\ \vdots \\ v_n w_n \end{bmatrix}$$

For our system of two qubits, it is $|a\rangle \otimes |b\rangle = \begin{bmatrix} a_0 \cdot \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \\ a_1 \cdot \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{bmatrix}.$

The tensor product $|a\rangle \otimes |b\rangle$ is the explicit notation of $|a\rangle|b\rangle$. Both terms mean the same.

We can represent a qubit system in a column vector or as the sum of the states and their amplitudes.

$$|ab\rangle = |a\rangle \otimes |b\rangle = a_0 b_0 |0\rangle|0\rangle + a_0 b_1 |0\rangle|1\rangle + a_1 b_0 |1\rangle|0\rangle + a_1 b_1 |1\rangle|1\rangle = \begin{bmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{bmatrix}$$

This representation of the qubit state is similar to the representation of the single-qubit state $|\psi\rangle$. The only difference is the larger number of dimensions the two-qubit system has. It has four basis state vectors instead of two.

All the rules that govern a single qubit apply to a system that consists of two qubits. It works similarly. Accordingly, the sum of all probabilities (remem-

ber the probability of a state is the amplitude square) must be 1:

$$|a_0b_0|^2 + |a_0b_1|^2 + |a_1b_0|^2 + |a_1b_1|^2 = 1$$

Unsurprisingly, working with a two-qubit system works similar to working with a one-qubit system, too. The only difference is, again, the larger number of dimensions the vectors and matrices have.

6.2.5 Two-Qubit Transformations

Let's say we want to apply the H-gate to the first qubit $|a\rangle$ and the X-gate to the second qubit $|b\rangle$ as depicted in the following figure.

Figure 6.14: Two-qubit transformation circuit
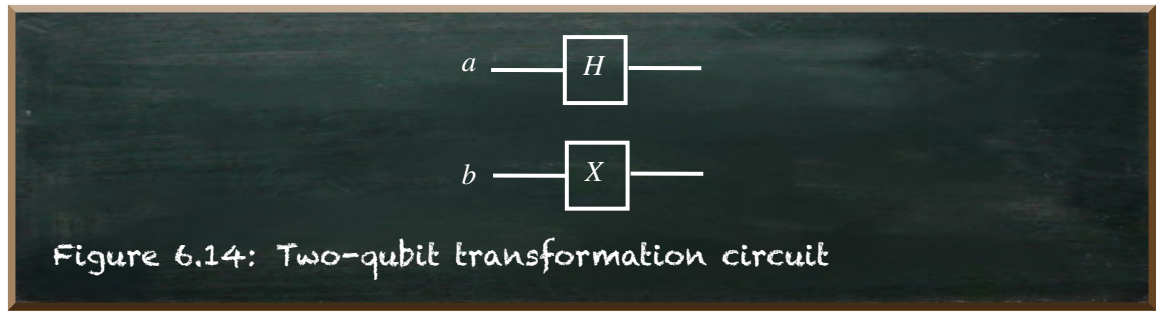
As we mentioned above, we can express the application of a gate by prepending the transformation matrix to the vector, like $M \cdot v$. In our specific example, we prepend a matrix to each of the vectors, like $H|a\rangle \otimes X|b\rangle$. Further, the tensor product is associative. This means we can regroup the terms as follows:

$$H|a\rangle \otimes X|b\rangle = (H \otimes X)(|a\rangle \otimes |b\rangle) = (H \otimes X)|ab\rangle$$

So, let's calculate the matrix denoted by $H \otimes X$.

We can see that the matrix of a two-qubit transformation gate has four times four dimensions. It corresponds to the four dimensions the two-qubit state vector has.

Except for the larger number of dimensions, there is nothing extraordinary

going on here. We can prepend this matrix to a two-qubit system.

$$
\begin{aligned}
H \otimes X &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
&= \frac{1}{\sqrt{2}} \begin{bmatrix} X & X \\ X & -X \end{bmatrix} \\
&= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & -1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \end{bmatrix}
\end{aligned}
\tag{6.1}
$$

Working with matrices of this size by hand is cumbersome. Fortunately, we have a computer to calculate the matrices and the tensor products for us.

Listing 6.5: Calculate the transformation matrix

```python
from qiskit import QuantumCircuit, Aer, execute

# Create a quantum circuit with one qubit
qc = QuantumCircuit(2)

# apply the Hadamard gate to the qubit
qc.i(0)
qc.h(1)

backend = Aer.get_backend('unitary_simulator')
unitary = execute(qc,backend).result().get_unitary()

# Display the results
unitary
```

First, we create the `QuantumCircuit` with two qubits (line 4). Then, we apply the X-gate to the one qubit and the H-gate to the other (lines 7-8).

> ! Qiskit orders the qubits from back to front with regard to the matrix calculation, so we need to switch the positions.

This time, we use a different Qiskit simulator as the backend, the `UnitarySimulator` (line 10). This simulator executes the circuit once and returns the final transformation matrix of the circuit itself. Note that this simulator does not contain any measurements.

The result is the matrix our circuit represents.

What if we only wanted to apply the H-gate to one of the qubits and leave the other unchanged? How would we calculate such a two-qubit transformation matrix?

We can use the I-gate as a placeholder when we calculate the tensor product. If we want to apply the H-gate to the first qubit and leave the second qubit unchanged, we calculate the transformation matrix as follows:

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} I & I \\ I & -I \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

These two-qubit gates transform qubits in a single step, but the qubits remain independent from each other. As a result, we unintentionally introduced another constraint.

When we look at the formula of the two-qubit state again, more specifically at the amplitudes, we can see that the product of the outer states' amplitudes ($|0\rangle|0\rangle$ and $|1\rangle|1\rangle$) equals the product of the inner states' amplitudes ($|0\rangle|1\rangle$ and $|1\rangle|0\rangle$), as shown in the following equation.

$$a_0 b_0 \cdot a_1 b_1 = a_0 b_1 \cdot a_1 b_0$$

This constraint results from how we create the two-qubit system as the combination of two independent single qubits. We even worked with these two qubits, yet only as independent qubits.

The term $(H \otimes X)|ab\rangle$ from our equation above explicitly shows the transfor-

mation we apply to $|ab\rangle$. This is

$$(H \otimes X)|ab\rangle = \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \end{bmatrix} |ab\rangle$$

6.2.6 Entanglement

What if we constructed the two-qubit system differently? When we discard the factors, the four basis states consist of $(a_0, b_0, ...)$ and replace them with general variables. We can state the following equation for an arbitrary two-qubit system.

$$|\psi\rangle = \alpha|0\rangle|0\rangle + \beta|0\rangle|1\rangle + \gamma|1\rangle|0\rangle + \delta|1\rangle|1\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}$$

We're holding on to the normalization of the sum of all probabilities must be 1.

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$$

But we do **not** insist that $\alpha\delta = \beta\gamma$.

In the last section, we got to know the CNOT-gate. It applies the X-gate to the target qubit only if we measure the control qubit as a 1.

Figure 6.15: The quantum CNOT gate
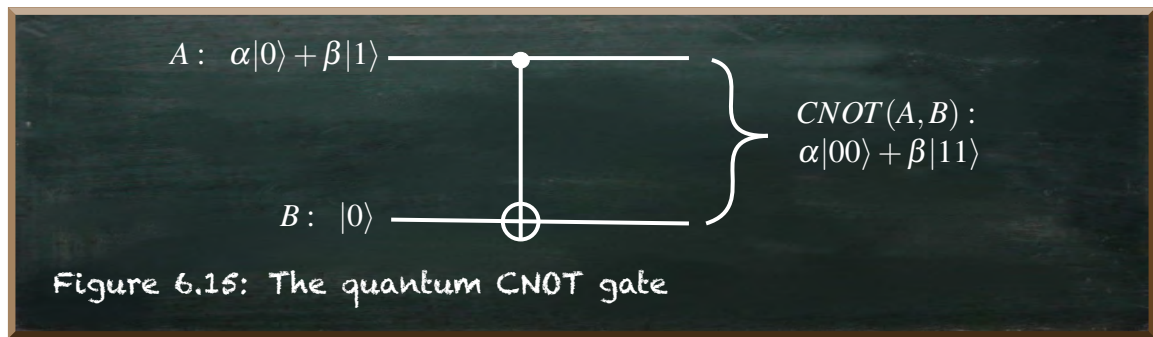
We can create the CNOT-gate from the two-qubit identity matrix by inter-

changing the order of the last two elements, like this:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The CNOT-gate takes two inputs and gives two outputs. The first input is called the control qubit. The second input is called the target qubit.

The result of the CNOT-gate is pretty forward if the control qubit is in a basis state $|0\rangle$ or $|1\rangle$. If the control qubit is $|0\rangle$, then nothing happens. The output equals the input. If the control qubit is $|1\rangle$, then the CNOT-gate applies the X-gate (NOT-gate) on the target qubit. It flips the state of the target qubit.

The following figure depicts the truth table of the CNOT-gate.

Figure 6.16: Truth table of the CNOT gate

It gets interesting when the control qubit is in superposition. For instance, when we apply the Hadamard gate to the first qubit before we apply the CNOT-gate.

Figure 6.17: A CNOT-gate with a control qubit in state $|+\rangle$
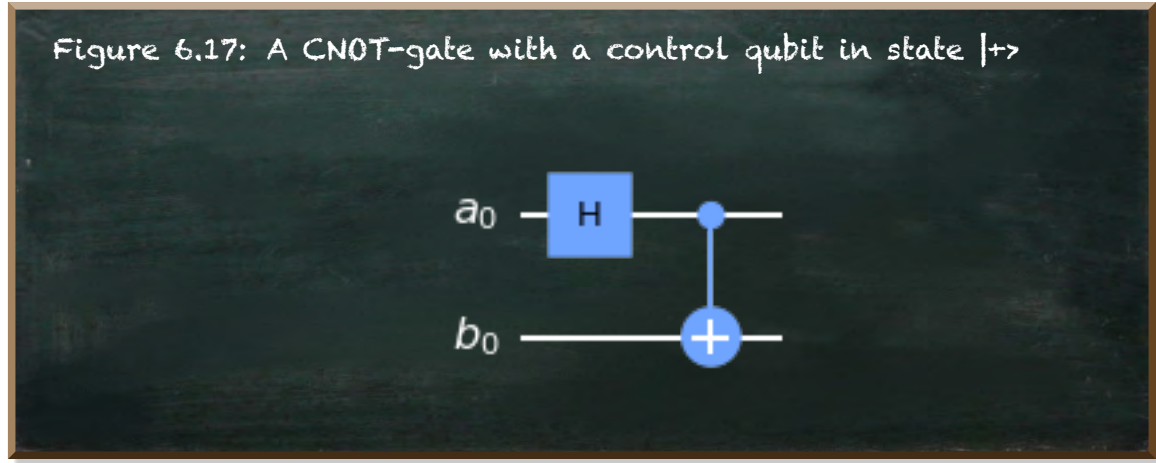
The following equation denotes the state of our two-qubit system.

$$CNOT \cdot (H \otimes I)|00\rangle$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

(6.2)

To calculate the subsequent application of multiple gates, we need to multiply the matrices from back to front. Thus, we start with the $CNOT$-gate, followed by the Hadamard gate. Unlike the multiplication of numbers, the order is essential when multiplying matrices.

When we read the term $CNOT \cdot (H \otimes I)|00\rangle$ from back to front, we start with the initial state ($|00\rangle$), apply the Hadamard gate to the first qubit, and apply the $CNOT$-gate to the combined two-qubit system.

In the next step, we replace the gates with the respective matrices (we derived them before in this section) and the initial state by the corresponding vector. Then, we calculate the overall transformation matrix before we apply it to the state vector in the last step.

Further, we can rewrite the vector as the sum of the weighted (by the amplitudes) basis states. We omit the states $|a_0\rangle|b_1\rangle$ and $|a_1\rangle|b_0\rangle$ for their amplitudes are 0. For clarity, we named the different basis states $|a_0\rangle$ and $|b_0\rangle$ rather than

simply $|0\rangle$.

$$|\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}|a_0\rangle\,(|b_0\rangle) + \frac{1}{\sqrt{2}}|a_1\rangle(|b_1\rangle)$$

Usually, we measure both qubits concurrently. The probability amplitudes tell us what to expect in this case. We will either get 00 or 11, each with a probability of $\frac{1}{2}$. But we can also measure a single qubit.

In the above equation, we disregard the terms within the parentheses for a second. This represents the case when we only measure the first qubit $|a\rangle$. We measure it with a probability of $\frac{1}{2}$ as 0 and with the same probability as 1. Once we measure it, the other qubit $|b\rangle$ jumps into the state given by the term inside the respective parentheses. So, if we measure $|a\rangle$ as 0, $|b\rangle$ jumps to the state $|b_0\rangle$. And if we measure $|a\rangle$ as 1, $|b\rangle$ jumps to the state $|b_1\rangle$.

When we measure $|a\rangle$, then $|b\rangle$ changes its state. It is important to note that $|b\rangle$ does not collapse once you measure $|a\rangle$. It does not collapse into 0 or 1 but jumps to another quantum state $|0\rangle$ or $|1\rangle$. Thus, measuring one of two entangled qubits collapses a subsystem. The unmeasured rest jumps to an unentangled quantum state.

The distinction between jumping to $|0\rangle$ or collapsing into 0 seems to be somewhat technical because once you measure $|0\rangle$ you inevitably get 0. But the state the unmeasured qubit jumps to can be any quantum state.

Let's edit our example a little bit. Instead of leaving the controlled qubit unchanged (applying the I-gate), we apply the R_Y-gate to it. We introduced the R_Y-gate in the last section. The R_Y-gate rotates the qubit state around the y-axis by a given angle. The following equation shows its transformation matrix.

$$R_y = |\psi\rangle\langle 0| + |\psi'\rangle\langle 1| = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} -sin\frac{\theta}{2} \\ cos\frac{\theta}{2} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$$

Let's rotate the controlled qubit by the angle by $\frac{\pi}{3}$, resulting in the circuit depicted in this figure.

Figure 6.18: The CNOT-gate can result in any quantum state
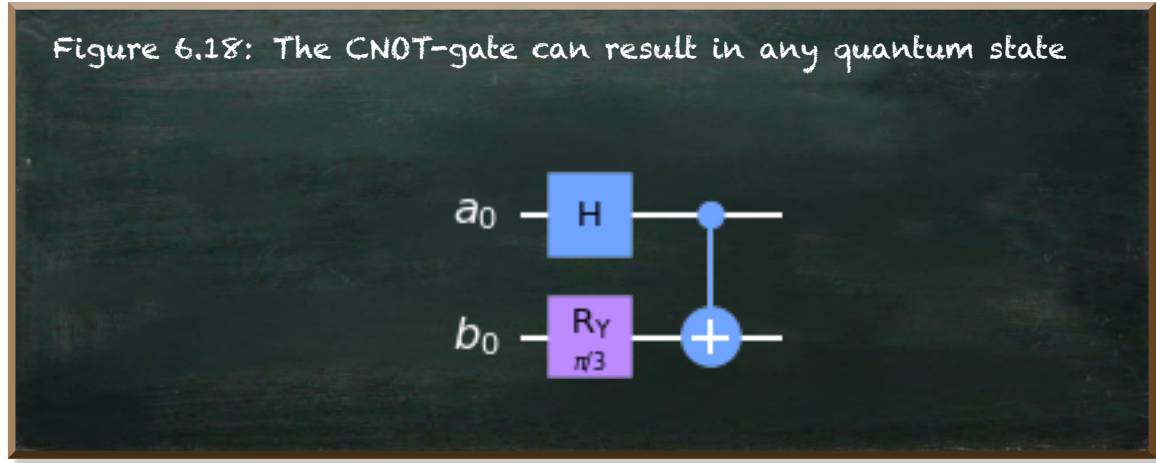
As before, we calculate the transformation matrix by matrix multiplication. The following equation shows the resulting vector.

$$CNOT \cdot (H \otimes R_Y(\frac{\pi}{3}))|00\rangle$$

$$= \begin{bmatrix} \sqrt{\frac{3}{8}} & -\frac{1}{\sqrt{8}} & \sqrt{\frac{3}{8}} & -\frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} & \sqrt{\frac{3}{8}} & \frac{1}{\sqrt{8}} & \sqrt{\frac{3}{8}} \\ -\frac{1}{\sqrt{8}} & \sqrt{\frac{3}{8}} & \frac{1}{\sqrt{8}} & -\sqrt{\frac{3}{8}} \\ \sqrt{\frac{3}{8}} & \frac{1}{\sqrt{8}} & -\sqrt{\frac{3}{8}} & -\frac{1}{\sqrt{8}} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{\frac{3}{8}} \\ \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} \\ \sqrt{\frac{3}{8}} \end{bmatrix}$$

(6.3)

We can easily see the resulting state is entangled. The outer probability amplitudes of the resulting vectors are both $\sqrt{\frac{3}{8}}$. Their product is $\frac{3}{8}$. The inner amplitudes are both $\frac{1}{\sqrt{8}}$. Their product is $\frac{1}{8}$. So, of course, $\frac{3}{8} \neq \frac{1}{8}$.

Again, we write this quantum state as the sum of its basis state.

$$\sqrt{\frac{3}{8}}|a_0\rangle|b_0\rangle + \frac{1}{\sqrt{8}}|a_0\rangle|b_1\rangle + \frac{1}{\sqrt{8}}|a_1\rangle|b_0\rangle + \sqrt{\frac{3}{8}}|a_1\rangle|b_1\rangle$$

We want to pull out the common factor $(\frac{1}{\sqrt{2}})$ and the state of the qubit $|a\rangle$. So,

let's rewrite this equation.

$$\frac{1}{\sqrt{2}}|a_0\rangle \left(\frac{\sqrt{3}}{2}|b_0\rangle + \frac{1}{2}|b_1\rangle \right) + \frac{1}{\sqrt{2}}|a_1\rangle \left(\frac{1}{2}|b_0\rangle + \frac{\sqrt{3}}{2}|b_1\rangle \right)$$

Both states $|a_0\rangle$ and $|a_1\rangle$ have the same probability amplitude of $\frac{1}{\sqrt{2}}$. Thus, we have a 50:50 chance of measuring the qubit $|a\rangle$ as 0 or 1. When we measure $|a\rangle$ as 0, then the qubit $|b\rangle$ jumps to the state $\frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$. Then, we have a probability of $\left(\frac{\sqrt{3}}{2} \right)^2 = \frac{3}{4}$ of measuring $|b\rangle$ as 0 and a probability of $\left(\frac{1}{2} \right)^2 = \frac{1}{4}$ of measuring it as 1.

But when we measure $|a\rangle$ as 1, we have the exact opposite probabilities when measuring $|b\rangle$. We get a probability of $\left(\frac{1}{2} \right)^2 = \frac{1}{4}$ of measuring $|b\rangle$ as 0 and a probability of $\left(\frac{\sqrt{3}}{2} \right)^2 = \frac{3}{4}$ of measuring it as 1.

When we measure entangled qubits individually, we do not collapse both qubits to finite values. But when we measure one qubit, the other jumps to an unentangled quantum state. It is not restricted to result in a basis state. It can jump to any valid quantum state.

When we measure the entangled qubits individually, what we measure might appear random. Only when we look at both measurements, we see their entangled state perfectly correlates them. This is because the entangled information qubits hold does not reside in either of the qubits individually. Instead, an entangled two-qubit system keeps its information non-locally in the correlations between the two qubits.

If the condition $\alpha\delta \neq \beta\gamma$ holds, then two qubits - or particles - are entangled. They share a state of quantum superposition. We can't represent their state by two individual states anymore. But their state is:

$$|\psi\rangle = \alpha|0\rangle|0\rangle + \beta|0\rangle|1\rangle + \gamma|1\rangle|0\rangle + \delta|1\rangle|1\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}$$

Once we measure one of the particles, the other inevitably changes its state. The two particles can be far apart. It doesn't matter how far. The other particle changes its state instantly. And, instantly means instantly.

This contradicted Einstein's notion of local realism. Therefore, he rejected the idea of entangled particles as "spukhafte Fernwirkung oder Telepathie". Translated into English, it is "spooky-action-at-a-distance or telepathy."

It was not until after Einstein's death that the first experimental evidence supported the theory of quantum entanglement.

6.3 Quantum Programming For Non-mathematicians

Figure 6.19: The controlled RY
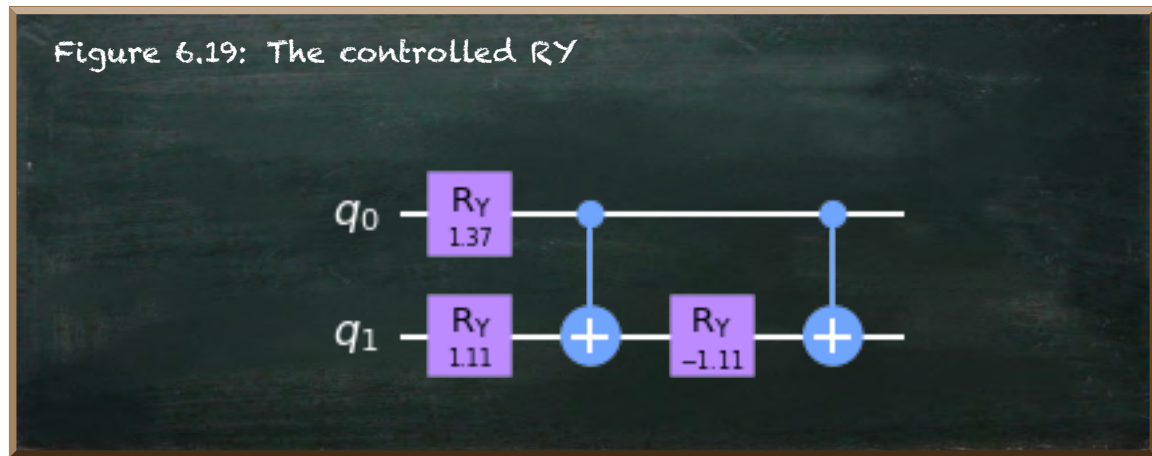
In the previous two chapters, we learned a lot about quantum computing. We learned how to work with a single qubit. We got to know different qubit gates. The Hadamard-gate, the NOT-gate, and the rotation-gate (R_Y).

We learned how to work with multiple qubits. We looked at entanglement and how we can use the CNOT-gate to entangle two qubits.

Thus far, we have paid attention to the concepts and the underlying math. But I meant it when I said you don't need to be a mathematician to master quantum machine learning. It is now time to look at quantum computing from the programmer's perspective and work with qubits practically. So we leave aside all the theory and math. Ok, we still need a little math to calculate probabilities. But that's it.

The only thing to understand is the 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

We will create and run quite a few circuits in this section. Therefore, here's a helper function that takes a configured `QuantumCircuit` instance, runs it and

returns the histogram.

Listing 6.6: The run-circuit helper function

```
1  from qiskit import QuantumCircuit, Aer, execute
2  from qiskit.visualization import plot_histogram
3  import matplotlib.pyplot as plt
4
5  def run_circuit(qc,simulator='statevector_simulator', shots=1, hist=True):
6
7    # Tell Qiskit how to simulate our circuit
8    backend = Aer.get_backend(simulator)
9
10   # execute the qc
11   results = execute(qc,backend, shots=shots).result().get_counts()
12
13   # plot the results
     return plot_histogram(results, figsize=(18,4)) if hist else results
```

We specify a rather broad figure size (`figsize=(18,4)`) of the histogram (line 13) to get some space in it to display all the different states. Further, while we work with the default `statevector_simulator` (line 5), we can also specify another simulator to use as our backend (line 7). Finally, we take the number of shots (how many times should the simulator run the circuit to obtain precise results) as a parameter (line 5).

Before we start calculating Bayes' posterior probability, let's look at the structure of the histogram Qiskit creates for us. In the following example, we create a `QuantumCircuit` with four qubits (line 1). Then, we apply the Hadamard-gate to each qubit (by providing a list of the qubit positions) (line 2) before we run the circuit (line 3).

Listing 6.7: Create an exemplary histogram

```
1  qc = QuantumCircuit(4)
2  qc.h([0,1,2,3])
3  run_circuit(qc)
```

Figure 6.20: An exemplary histogram
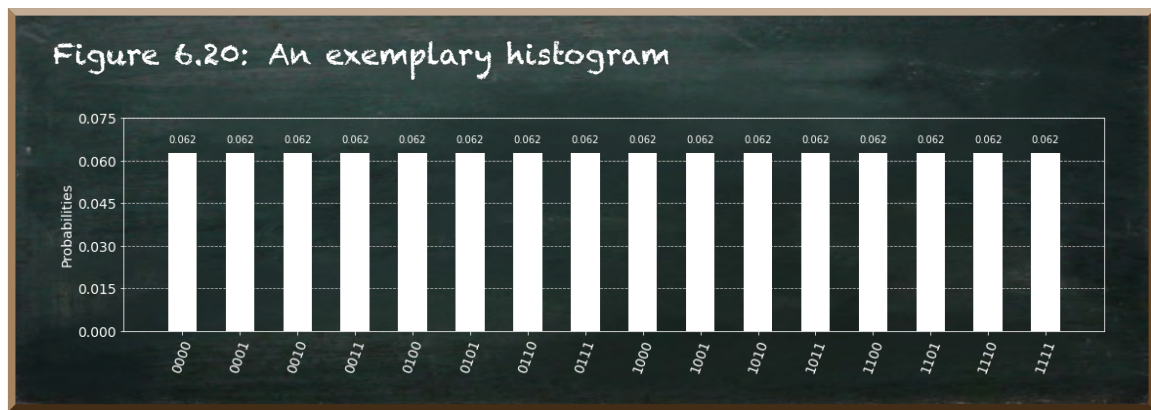
Each column in the histogram represents a state. A state is a combination of the qubits' values. In our case, a state is made up of four qubits. We can measure each of these qubits as either 0 or 1. The bar and the number above the bar indicate the measurement probability of this state.

In our case, there are 16 states, and they all have the same probability of 0.062, respectively $\frac{1}{16}$.

The numbers below the bar indicate the values of the four qubits in the given state. They are read from the top (qubit at position 0) to the bottom (qubit at position 3). If you rotated the numbers clockwise to read them better, you would need to read them from the right (qubit 0) to the left (qubit 3).

Further, the states are ordered. As if the four qubits made up a binary digit, qubit 0 is the lower bit at the right-hand side, and qubit 3 is the upper bit at the left-hand side. As a result, all the states where qubit 3 is measured as 1 reside at the right half of the histogram. Thus, if you want states to be next to each other, make sure their value for the highest qubit is the same.

For better readability, let's consider a circuit with a single Hadamard gate.

Listing 6.8: A single Hadamard gate

```
1 qc = QuantumCircuit(4)
2 qc.h(0)
3 run_circuit(qc)
```

Figure 6.21: Result of a single Hadamard gate
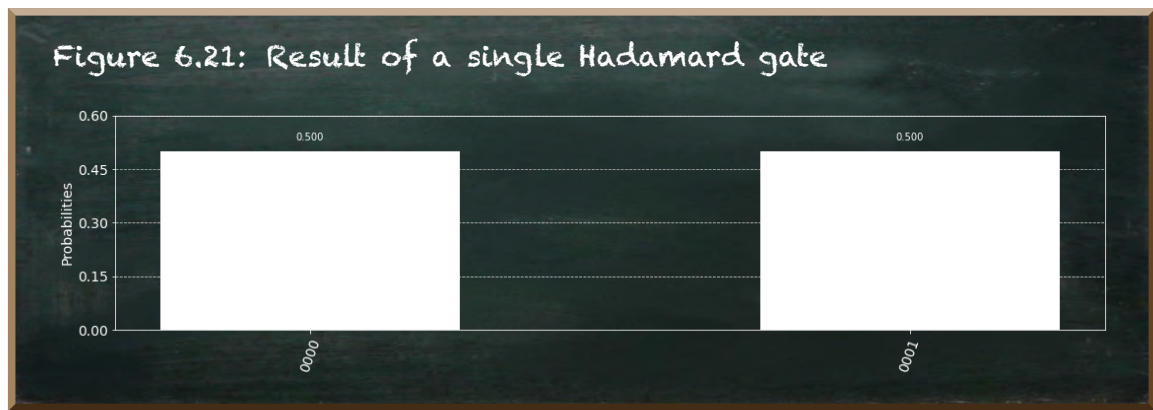
Qiskit initializes qubits in the state $|0\rangle$. When we apply the Hadamard-gate on a qubit in this state, it cuts the total probability of 1.0 into two halves. Thus, we get two states that differ in the value for qubit 0. Each of these states has a probability of 0.5. Since the binary value of 0001 is greater than 0000, the state 0001 is on the right-hand side.

6.3.1 Representing a 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.

In the figure with one Hadamard-gate, there is only one state where qubit 0 is 1. Therefore, the marginal probability is 0.5. In the figure with four Hadamard gates, there are eight states where qubit 0 is 1. The marginal probability of qubit 0 being 1 is the sum of all these states' probabilities. It is 0.5, too.

The Hadamard-gate splits the overall probability into equal halves. But a marginal probability can be any value between 0.0 and 1.0.

In section 5.3, we introduced the R_Y-gate. It takes a parameter we can use to specify the exact probability. For the R_Y-gate takes an angle θ as its parameter, not a probability, we need to convert the probability into an angle before passing it to the gate. This is what the function `prob_to_angle` does for us.

Listing 6.9: Calculate the angle that represents a certain probability

```python
from math import asin, sqrt

def prob_to_angle(prob):
    """
    Converts a given P(psi) value into an equivalent theta value.
    """
    return 2*asin(sqrt(prob))
```

Now, we can create and run a circuit with an arbitrary marginal probability between 0 and 1. Let's start with a probability of 0.4 of measuring 1.

We apply the R_y gate on the qubit and pass it as the first parameter to call `prob_to_angle` with the probability value of `0.4` (line 2).

Listing 6.10: Specify the marginal probability

```python
# Specify the marginal probability
event_a = 0.4

qc = QuantumCircuit(4)

# Set qubit to prior
qc.ry(prob_to_angle(event_a), 0)

run_circuit(qc)
```


Figure 6.22: The marginal probability
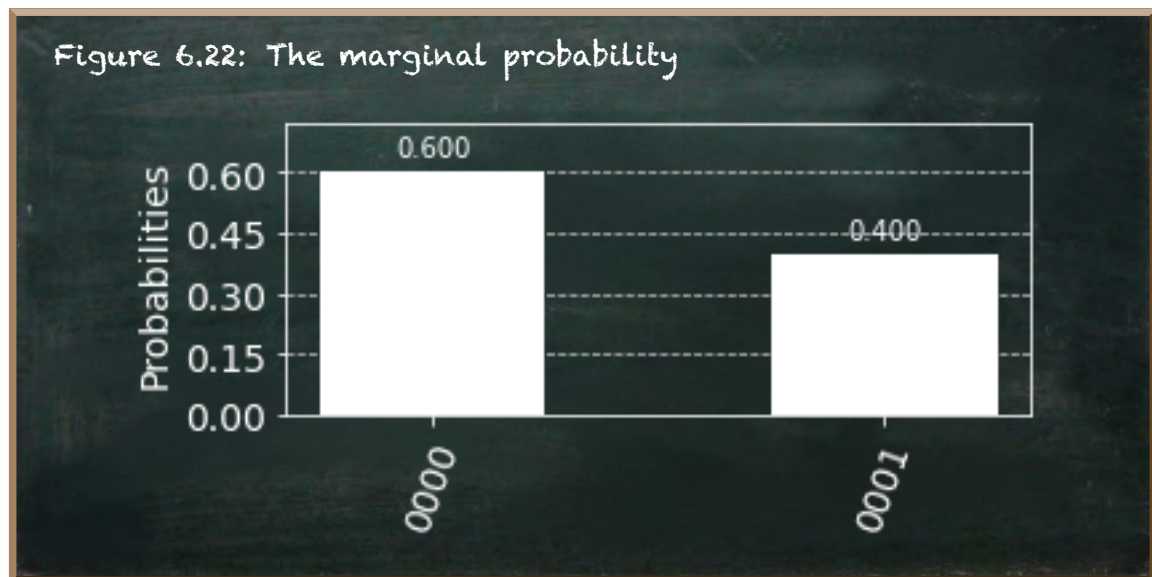
Similar to the Hadamard-gate, the R_Y-gate cuts the probability into two parts. But it provides us with a tool to control the size of the two parts.

6.3.2 Calculate the joint probability

In the next step, we want to calculate the joint probability of two events. Both events have marginal probabilities between 0.0 and 1.0. Just like any other probability.

The following figure depicts the joint probability of two variables.

Figure 6.23: The joint probability
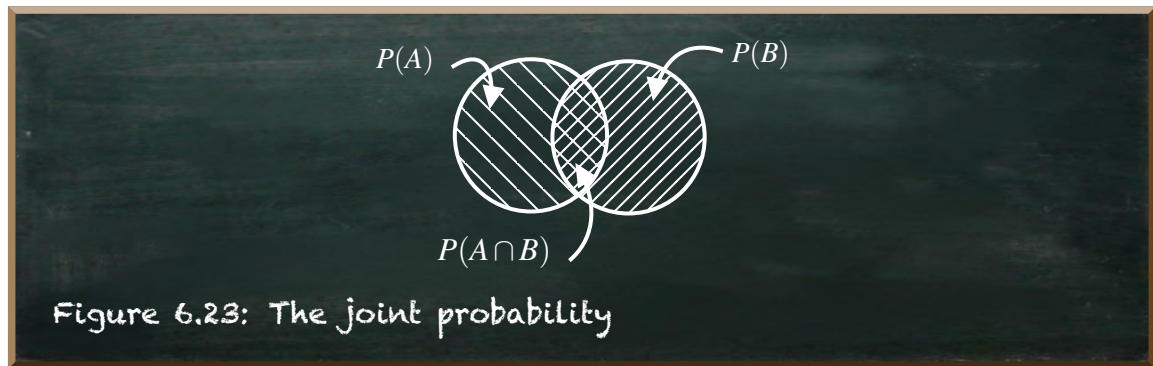
Mathematically, we can calculate the joint probability by multiplying both marginal probabilities. Let's say event B has a probability of 0.8. We expect a probability of $0.4 * 0.8 = 0.32$.

Let's try it with Qiskit.

Listing 6.11: Represent two marginal probabilities with a single qubit

```
1  # Specify the marginal probabilities
2  event_a = 0.4
3  event_b = 0.8
4
5  qc = QuantumCircuit(4)
6
7  # Set qubit to prior
8  qc.ry(prob_to_angle(event_a), 0)
9
10 # Apply modifier
11 qc.ry(prob_to_angle(event_b), 0)
12
13 run_circuit(qc)
```

Figure 6.24: Result of using a single qubit

This didn't work. We're not even close to the target probability. Instead, we get a probability of 0.952.

Figure 6.25: The function f(x)=arcsin(sqrt(x))
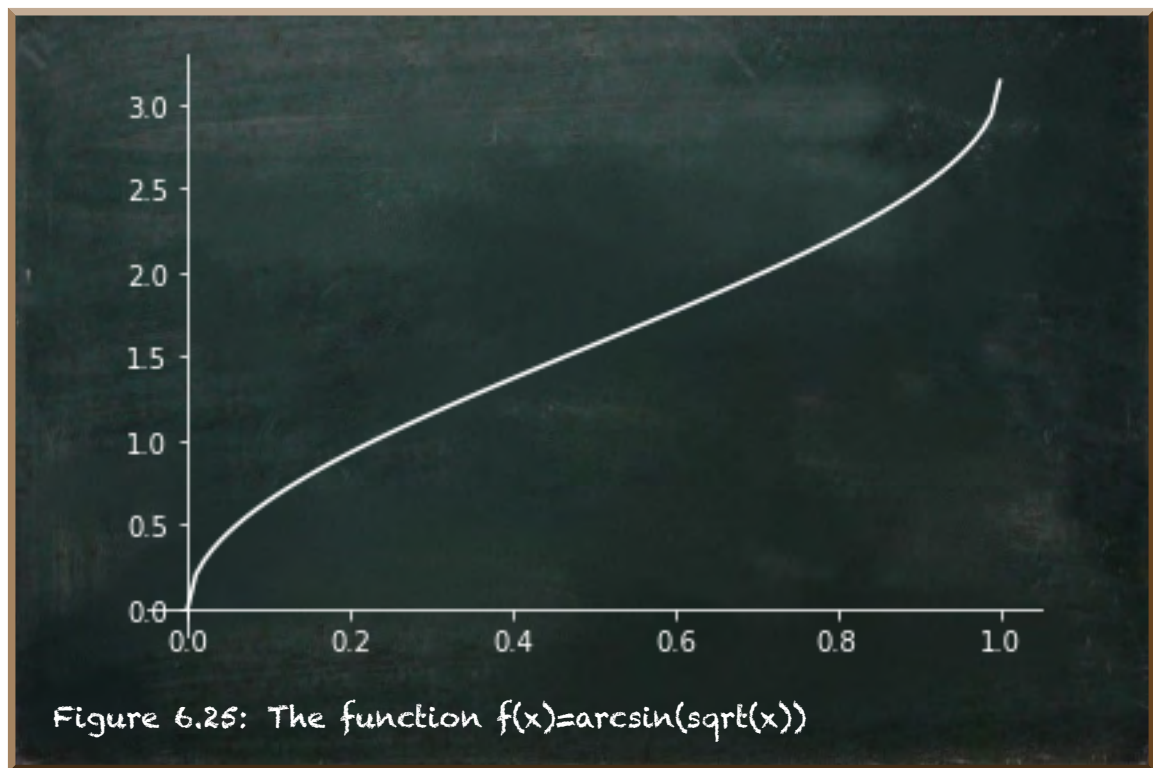
The problem is the calculation of the angle θ inside the `prob_to_angle`-function. We calculate the angle as the arcsine of the target probability's square root.

Let's have a closer look at this function. The following figure depicts the shape of $f(x) = arcsin(\sqrt{x})$

The first thing to note is that the function is defined in the interval between 0 and 1. For negative values, the square root is not defined. For values above 1, the arcsine is not defined.

The second thing to note is the curve of the function. The `prob_to_angle`-function assumes the qubit to be in the basis state $|0\rangle$. θ– that is the angle we calculate–is the angle between the vector $|\psi\rangle$–that is the target state–and the basis state vector $|0\rangle$–that is the state we start from. If we started from another state, we would need to incorporate this state in the calculation of θ. We would need to start at the respective point on the curve. It makes a difference if you calculate a step at the beginning of the curve (there is a high gradient) and calculate a step in the middle of the curve.

But if we incorporated the current point (that represents the probability of event A) on the curve into the calculation, we would do the whole calculation of the joint probability outside of our quantum circuit. This is not what we aim at.

Let's give it another try. If the `prob_to_angle`-function assumes the qubit to be in the basis state $|0\rangle$, why don't we set the probability of event B on a new qubit? The difference is in line 11, where we apply the rotation about the $P(B)$ on the qubit 1 instead of the qubit 0.

Listing 6.12: Each marginal probability uses a qubit

```
1  # Specify the marginal probabilities
2  event_a = 0.4
3  event_b = 0.8
4
5  qc = QuantumCircuit(4)
6
7  # Set qubit to event_a
8  qc.ry(prob_to_angle(event_a), 0)
9
10 # Set fresh qubit to event_b
11 qc.ry(prob_to_angle(event_b), 1)
12
13 run_circuit(qc)
```

Figure 6.26: Result of using a fresh qubit
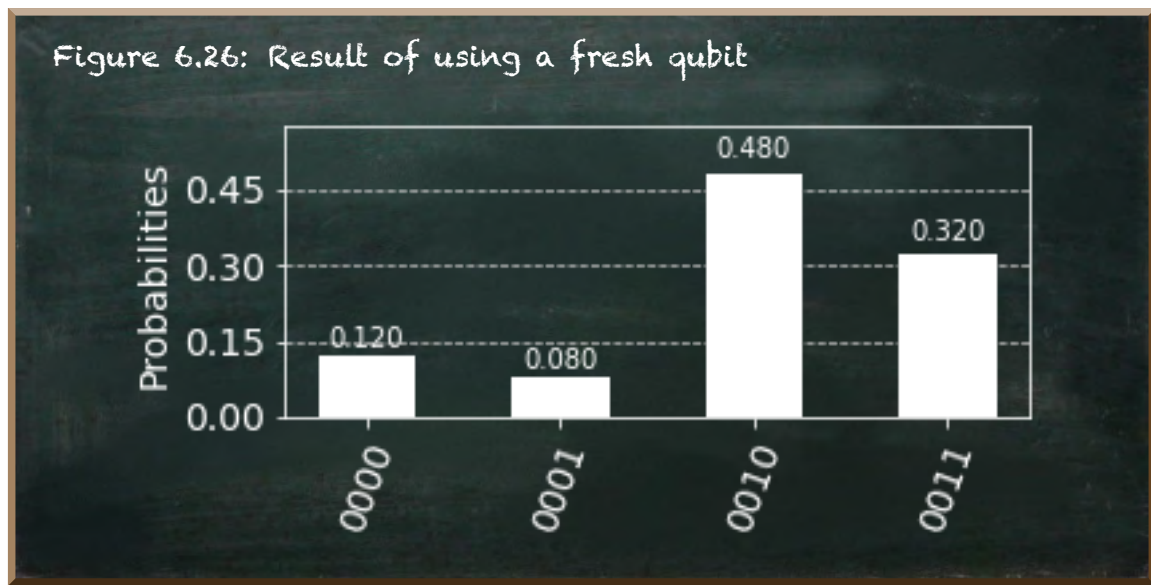
We see that the state 0011 (qubit 0 is 1 and qubit 1 is 1) denotes the correct probability of 0.32. The underlying rationale is quite simple.

All the states where qubit 0 is 1, (0001 and 0011) add up to 0.4–the marginal probability representing event A. The states where qubit 1 is 1 (0010 and 0011) add up to 0.8–the marginal probability representing event B. Since both rotations are independent of each other, the state 0011 represents the overlap of both probabilities, with $0.4 * 0.8 = 0.32$.

For completion, the probability of the other states are:

- state 0000: $(1.0 - 0.4) * (1.0 - 0.8) = 0.6 * 0.2 = 0.12$
- state 0001: $0.4 * (1.0 - 0.8) = 0.4 * 0.2 = 0.08$
- state 0010: $(1.0 - 0.4) * 0.8 = 0.6 * 0.8 = 0.48$

To get the joint probability, we need to measure both qubits and count the portion where both qubits are 1.

We're interested in a single one probability. Wouldn't it be good if a single one qubit represented it?

This is where entanglement comes in handy. Do you remember the CNOT-gate? It is a two-qubit gate. The first qubit is the control qubit. If that is 1, then the gate applies the X-gate (NOT-gate) on the second qubit. If the control qubit is 0, then the second qubit remains unchanged.

Let's first look at the code.

Listing 6.13: A controlled RY-gate

```
 1  # Specify the marginal probabilities
 2  event_a = 0.4
 3  event_b = 0.8
 4
 5  qc = QuantumCircuit(4)
 6
 7  # Set qubit to prior
 8  qc.ry(prob_to_angle(event_a), 0)
 9
10  # Apply half of the modifier
11  qc.ry(prob_to_angle(event_b)/2, 1)
12
13  # entangle qubits 0 and 1
14  qc.cx(0,1)
15
16  # Apply the other half of the modifier
17  qc.ry(-prob_to_angle(event_b)/2, 1)
18
19  # unentganle qubits 0 and 1
20  qc.cx(0,1)
21
22  run_circuit(qc)
```


Figure 6.27: The result of a controlled rotation
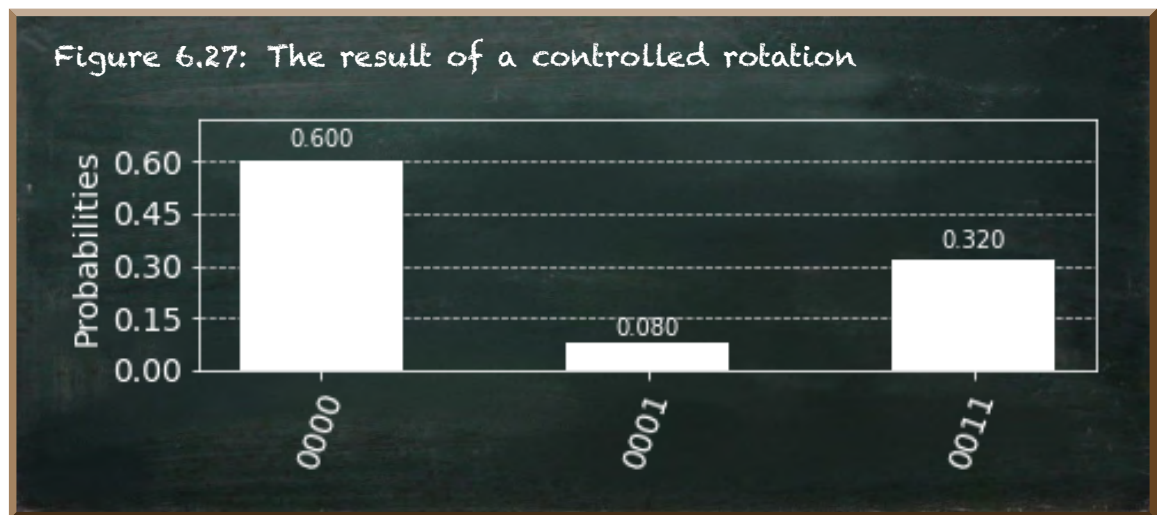
The result shows a probability of 0.32 for measuring qubit 1 as 1. We only have to measure a single qubit to get the joint probability we are looking for.

But how does it work?

As before, we apply the marginal probability of event *A* (line 8). Next, we apply **half** of the marginal probability of event *B*. The following figure shows the state the system would have if we stopped here.

Figure 6.28: Applying half of the probability
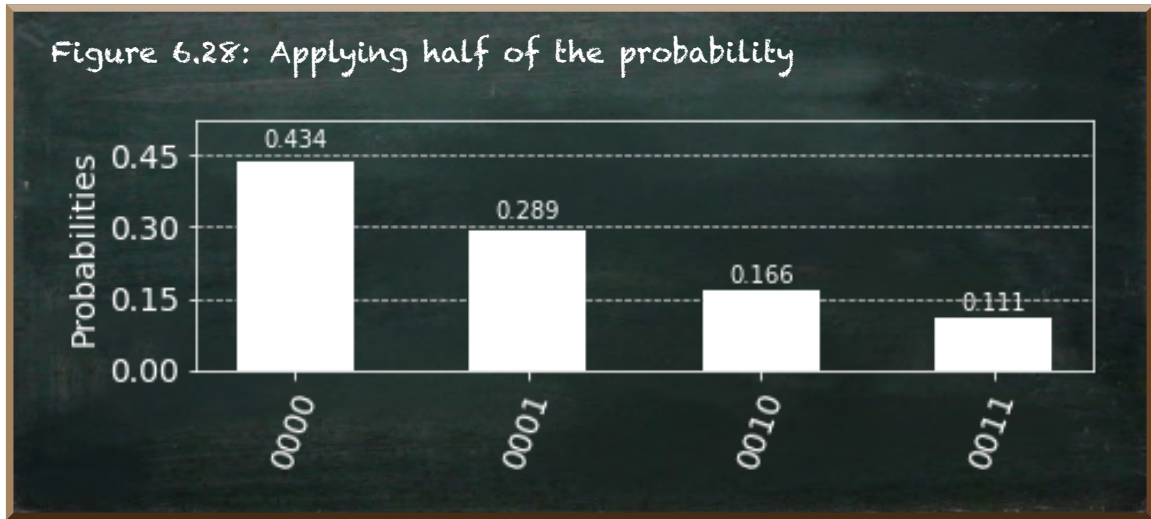
The resulting probabilities are quite a mess. But what we can see is that we split the states where qubit 0 is 0 into two parts: the states 0000 and 0010. And we did the same for the states where qubit 0 is 1.

Next, we apply the CNOT-gate (line 14). Let's see what it does.

Figure 6.29: The CNOT-gate switches probabilities
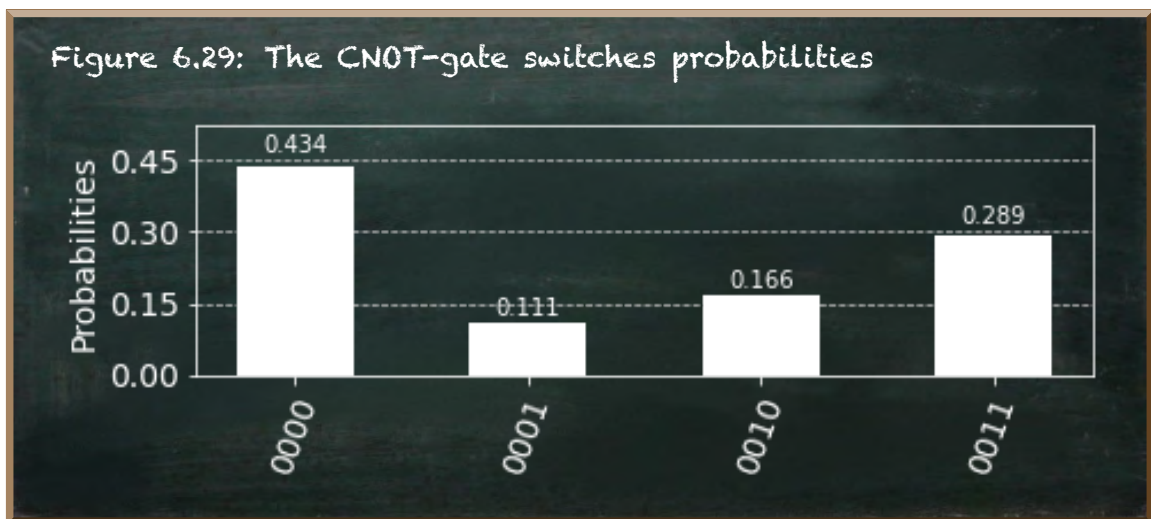
The CNOT-gate does not change the probability values. But it switches the states that have these probabilities.

The states 0000 and 0010 keep their probabilities because the CNOT-gate does not do anything if the control qubit (here qubit 0) is 0. By contrast, the states 0001 and 0011 switch their probabilities–just like we said. If the control qubit is 1 the CNOT-gate acts like an X-gate on the target qubit (here qubit 1).

Essentially, what we did is we say the state where $P(A)$ and $\frac{P(B)}{2}$ overlap should switch its probability with the state where the $P(A)$ overlaps with $1 - \frac{P(B)}{2}$.

We apply the second half of $P(B)$ but with a minus sign (line 17). Here's the effect.

Figure 6.30: Applying the other half of the probability
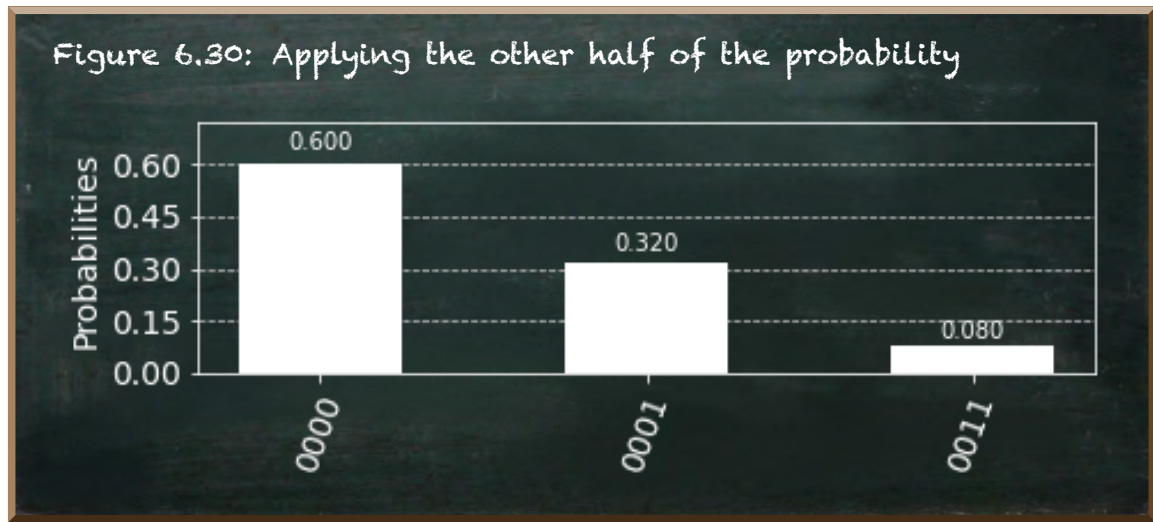
The correct value of the joint probability appears. Since we apply the negative half of $P(B)$, we undo the split of the probabilities when qubit 0 is 0 that we did with the first application of the RY-gate. The state of 0010 disappears. It is now part of the state 0000 again.

When qubit 0 is 1, we move half of $P(B)$ from state 0011 back to state 0001. But remember, these two states switched their probabilities before. That means, instead of undoing the effect of the first R_Y-gate, we add the second half of $P(B)$.

In the last step, we apply the CNOT-gate again. This leaves the state 0000 untouched for the control qubit 0 is 0. But when qubit 0 is 1 it switches the value of qubit 1. Thus, the states 0001 and 0011 switch their probabilities again.

As a result, the state 0011 has the resulting joint probability of 0.32. Since this state is the only state where qubit 1 is 1, we get the joint probability by measuring a single qubit.

This part of the circuit is also known as the controlled R_Y-gate. Qiskit provides

a function for this out of the box. Let's have a look. It has the same effect.

Listing 6.14: The controlled RY-gate of Qiskit

```python
# Specify the marginal probabilities
event_a = 0.4
event_b = 0.8

qc = QuantumCircuit(4)

# Set marginal probability
qc.ry(prob_to_angle(event_a), 0)

# Apply the controlled RY−gate
qc.cry(prob_to_angle(event_b), 0, 1)

run_circuit(qc)
```


Figure 6.31: Result of the CRY-gate
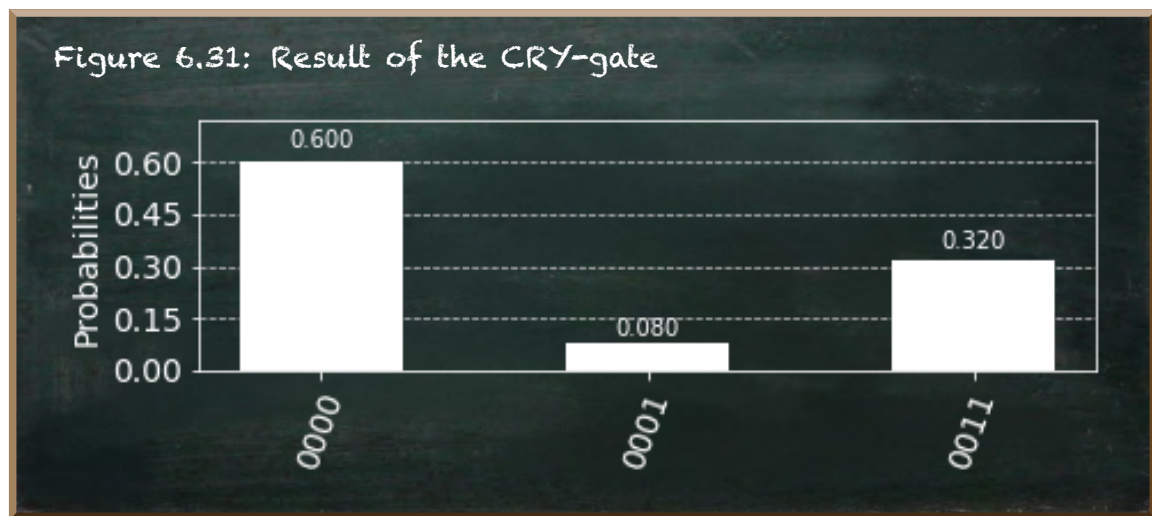
In summary, the controlled R_Y-gate works similarly to the CNOT-gate. But it applies an R_Y-gate on the target qubit if the control qubit is 1, rather than using the X-gate. If the control qubit is 0, nothing happens.

From the perspective of the resulting states, the controlled R_Y-gate splits the state(s) where the control qubit is 1 into two parts. But it leaves untouched the state(s) where the control qubit is 0. Thus, in contrast to applying the R_Y-gate on a new qubit that splits all states, the controlled R_Y-gate provides fine control over the states you want to work with. Consequently, you transform the target qubit only in the cases that matter.

6.3.3 Calculate the conditional probability

Figure 6.32: Calculating the posterior
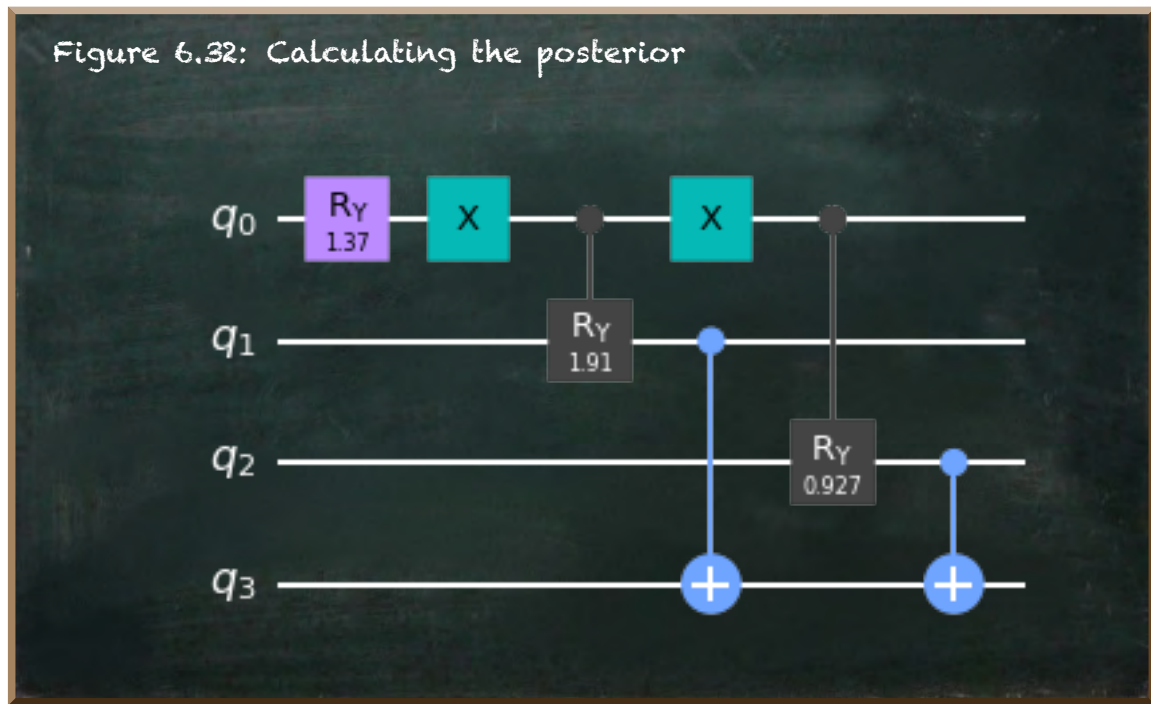
The calculation of the joint probability of two events works pretty well. It answers the question of how likely it is for two independent events to occur concurrently. In the next step, we aim to calculate the conditional probability of an event given that we know another event occurred. This is also known as the posterior probability.

Graphically, the conditional probability is almost the same as the joint probability. 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 conditional probability.

Bayes' Theorem tells us how to calculate the conditional probability. We covered Bayes' Theorem in-depth in section 4.2. Therefore, here's only a very brief recap so that you don't need to flip too many pages all the time.

Bayes' Theorem describes a way of finding a conditional probability. A conditional probability is a probability of an event (our hypothesis) given the knowledge that another event occurred (our evidence). Bayes tells us we can calculate the conditional probability of $P(Hypothesis|Evidence)$ as the product of the marginal probability of the hypothesis ($P(Hypothesis)$, called the prior probability) and a modifier. This modifier is the quotient of the "backward"

Figure 6.33: Joint and posterior probabilities
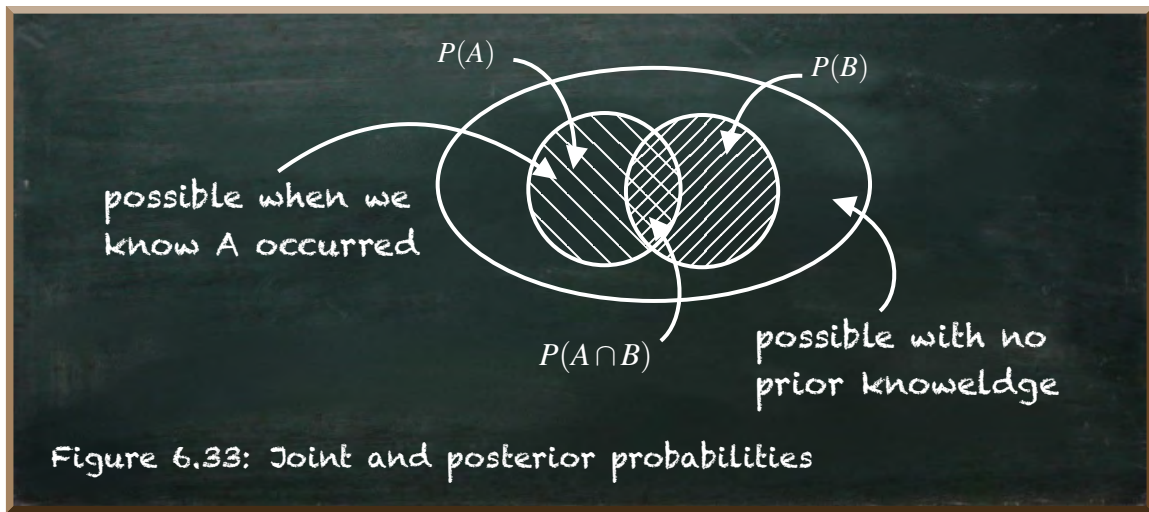
conditional probability ($P(Evidence|Hypothesis)$) and the marginal probability of the new piece of information ($P(Evidence)$). The backward probability (the numerator of the modifier) answers the question, "what is the probability of observing this evidence in a world where our hypothesis is true?" The denominator is the probability of observing the EvidenceEvidence on its own.

The following equation depicts Bayes' Theorem mathematically:

$$\underbrace{P(Hypothesis|Evidence)}_{posterior} = \underbrace{P(Hypothesis)}_{prior} \cdot \underbrace{\frac{P(Evidence|Hypothesis)}{P(Evidence)}}_{modifier}$$

Listing 6.15: Calculate the conditional probability for a modifier < 1

```python
# Specify the prior probability and the modifier
prior = 0.4
modifier = 0.9

qc = QuantumCircuit(4)

# Set qubit to prior
qc.ry(prob_to_angle(prior), 0)

# Apply the controlled RY-gate
qc.cry(prob_to_angle(modifier), 0, 1)

run_circuit(qc)
```

Figure 6.34: The posterior probability
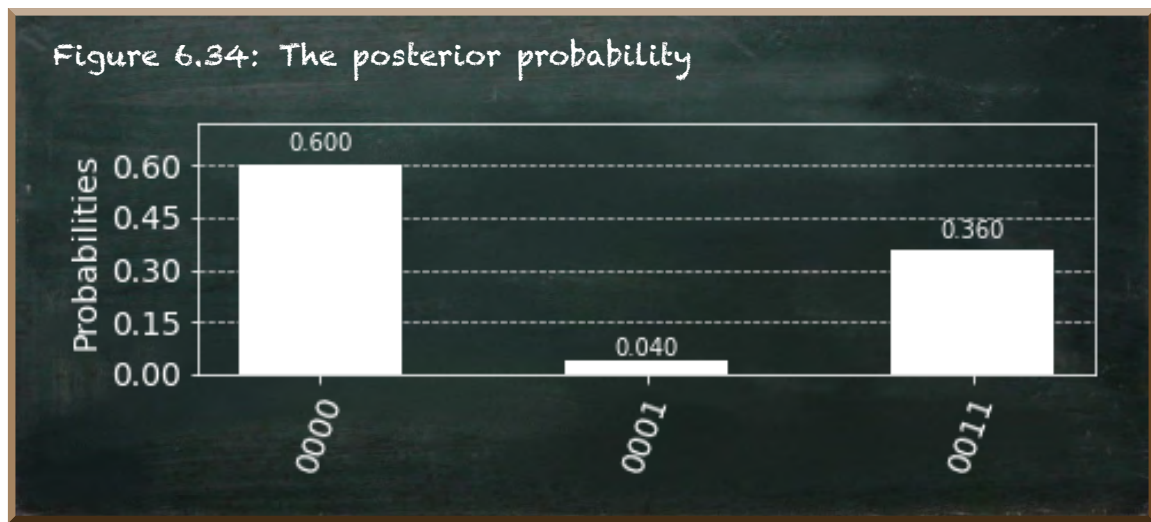
The modifier can be any positive number. But most likely, it is a number close to 1. If the modifier was exactly 1, it would mean the *prior* is equal to the *posterior* probability. Then, the *Evidence* would not have provided any information.

In the first case, let's say the modifier is a number between 0.0 and 1.0. We can use the quantum circuit we created to calculate a conditional probability.

Qubit 1 shows the resulting conditional probability of 0.36. Let's have a look at what happens for a modifier greater than 1.0.

Listing 6.16: A modifier greater than 1

```python
# Specify the prior probability and the modifier
prior = 0.4
modifier = 1.2

qc = QuantumCircuit(4)

# Set qubit to prior
qc.ry(prob_to_angle(prior), 0)

# Apply modifier
qc.cry(prob_to_angle(modifier), 0,1)

run_circuit(qc)
```

```
ValueError: math domain error
```

We get a math domain error. Of course, we do because the function `prob_to_angle` is only defined for values between 0 and 1. For values greater than `1.0`, the arcsine is not defined. The arcsine is the reverse of the sine function. Its gradient at `0.0` and `1.0` tend to infinity. Therefore, we can't define the function for values greater than `1.0` in a meaningful way.

Let's rethink our approach. If the modifier is greater than `1.0`, it increases the probability. The resulting probability must be bigger than the *prior* probability. It must be greater by exactly $(modifier - 1) \cdot prior$.

The transformation gates let us cut the overall probability of `1.0` into pieces. Why don't we separate the *prior* not once but twice? Then, we apply the reduced *modifier* $(modifier - 1)$ on one of the two states representing the *prior*. The sum of the untouched *prior* and the applied reduced *modifier* should be the conditional probability.

In the following code, we apply the prior to qubit 0 (line 8) and to qubit 1 (line 11). Then, we apply the reduced modifier to qubit 2 through an R_Y-gate controlled by qubit 0.

Listing 6.17: Working with a reduced modifier

```python
# Specify the prior probability and the modifier
prior = 0.4
modifier = 1.2

qc = QuantumCircuit(4)

# Apply prior to qubit 0
qc.ry(prob_to_angle(prior), 0)

# Apply prior to qubit 1
qc.ry(prob_to_angle(prior), 1)

# Apply modifier to qubit 2
qc.cry(prob_to_angle(modifier-1), 0,2)

run_circuit(qc)
```

Figure 6.35: Result of working with a reduced modifier
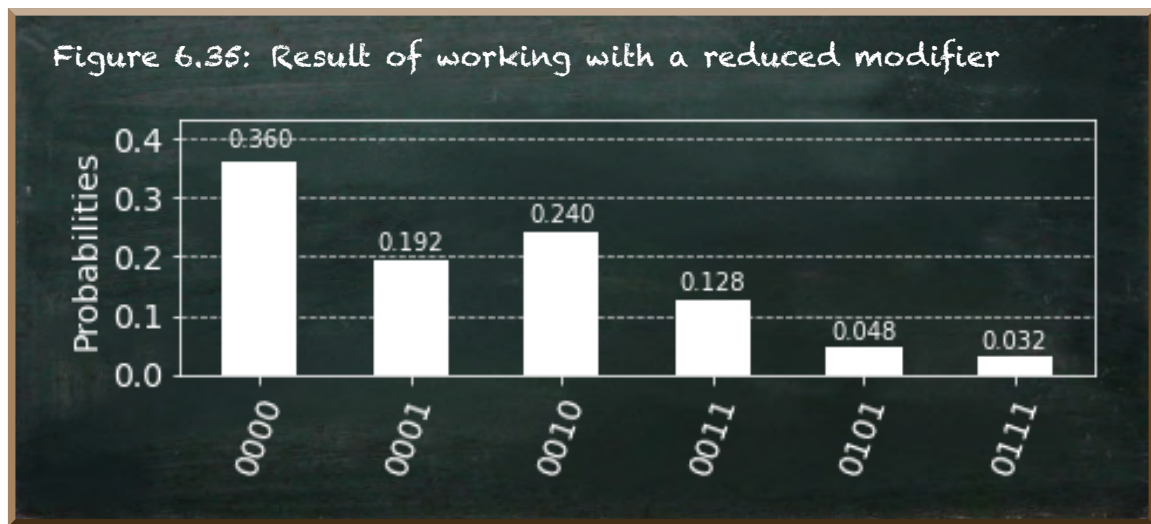
We get six different states. Our conditional probability should be the sum of the states where qubit 1 is 1 plus the sum of the states where qubit 2 is 1. These are the four states on the right-hand side. Let's add them:

$$0.240 + 0.128 + 0.048 + 0.032 = 0.448$$

This didn't work. The expected result is $0.4 + 0.4 * 0.2 = 0.48$. What happened?

The problem is the case where both qubits 1 and 2 are 1. This is the state 0111. In order to get the correct conditional probability, we would need to count this state twice: $0.448 + 0.032 = 0.48$.

Listing 6.18: The overlap when applying the prior twice

```
1  # Specify the prior probability and the modifier
2  prior = 0.4
3  modifier = 1.2
4
5  qc = QuantumCircuit(4)
6
7  # Apply prior to qubit 0
8  qc.ry(prob_to_angle(prior), 0)
9
10 # Apply prior to qubit 1
11 qc.ry(prob_to_angle(prior), 1)
12
13 run_circuit(qc)
```

Figure 6.36: The overlap when applying the prior twice
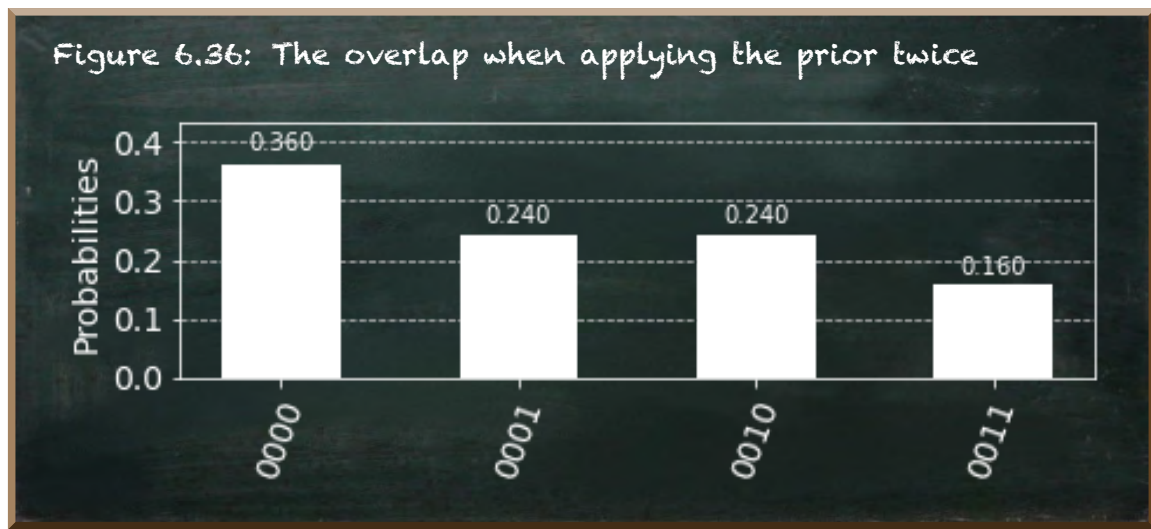
This problem originated when we applied the *prior* probability for the second time. We aimed at two states, each representing the *prior*. When we look at the result, we can see that, in fact, the probability of measuring qubit 0 as 1 is `0.4` (the *prior*), and the probability of measuring qubit 1 as 1 is `0.4`, too. But we also see that these probabilities are not independent of each other. But they overlap in the state 0011.

When we apply the *prior* to qubit 1, we need to leave the states where qubit 0 is 1 untouched.

Have a look.

Listing 6.19: Applying the prior to qubit 1 from the remainder

```
 1  # Specify the prior probability and the modifier
 2  prior = 0.4
 3  modifier = 1.2
 4
 5  qc = QuantumCircuit(4)
 6
 7  # Apply prior to qubit 0
 8  qc.ry(prob_to_angle(prior), 0)
 9
10  # Apply prior to qubit 1
11  qc.x(0)
12  qc.cry(prob_to_angle(prior/(1−prior)), 0, 1)
13  qc.x(0)
14
15  run_circuit(qc)
```

Figure 6.37: Applying the prior to qubit 1 from the remainder
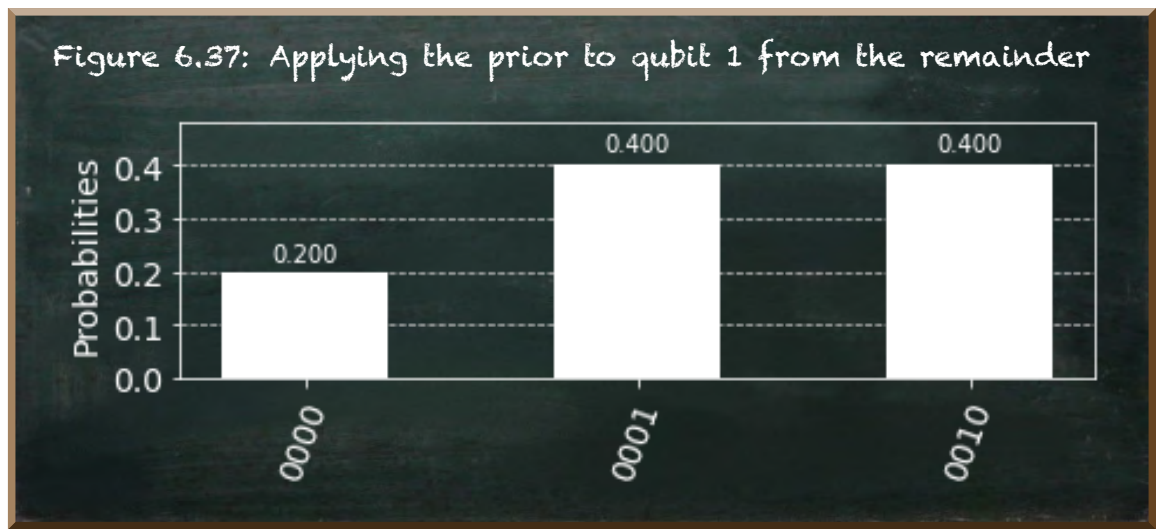
Three lines do the trick:

Listing 6.20

```
1  qc.x(0)
2  qc.cry(prob_to_angle(prior/(1−prior)), 0, 1)
3  qc.x(0)
```

Let's go through these lines step by step. In the first step, we apply the NOT-gate to qubit 0. It switches the probabilities of the states where qubit 0 is 0 with those where qubit 0 is 1.

Figure 6.38: Probabilities after the first X-gate
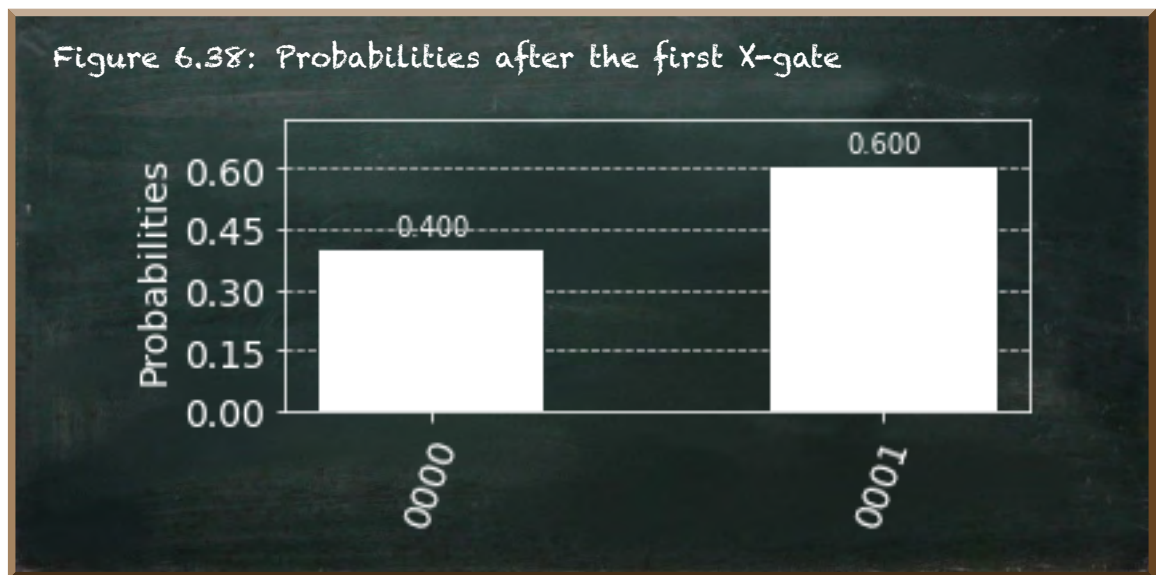
The figure depicts the state after the first NOT-gate.

We set the *prior* (`0.4`) as the probability of measuring qubit 0 as `1`. The NOT-gate reverses this. Now, we have the probability of `0.4` of measuring qubit 0 as `0`.

This also means we measure the remainder (`0.6`) when qubit 0 is `1`. Simply put, the NOT-gate is our way of saying: "Let's proceed to work with the remainder, not the prior".

This is the preparation for our next step. The controlled R_Y-gate.

Listing 6.21

```
1  qc.cry(prob_to_angle(prior/(1-prior)), 0, 1)
```

We only apply a rotation of qubit 1 when qubit 0 is `1`. This is the case only for the remainder. The rest is not `1.0`, but it is `1.0-prior`. We modify the probability we use in the controlled R_Y-gate. By specifying the size of the remainder as the denominator, we account for the smaller size.

The figure below depicts the state after the controlled R_Y-gate.

Figure 6.39: Probabilities after the CRY-gate
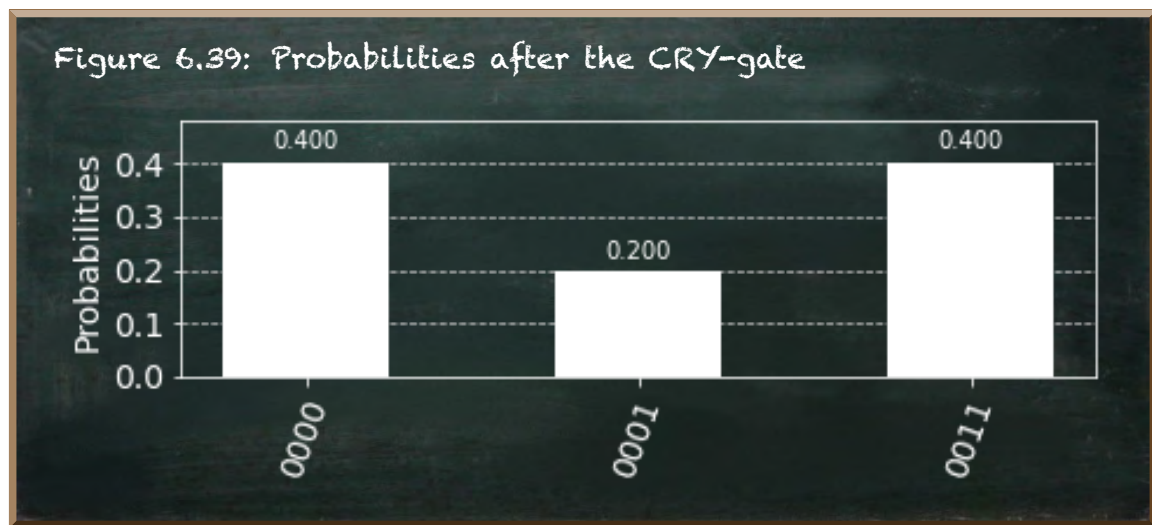
The controlled R_Y-gate splits the remainder into two parts. The one part (state 0011) represents the *prior*. So does the state 0000 we separated in the very first step. There is no more overlap between these two states. To keep things ordered, we apply the NOT-gate on qubit 0 again. The state 0000 becomes 0001 and vice versa, and the state 0011 becomes 0010. It leaves us with the qubits 0 and 1, each representing the *prior* probability without overlap.

Figure 6.40: Probabilities after the second X-gate
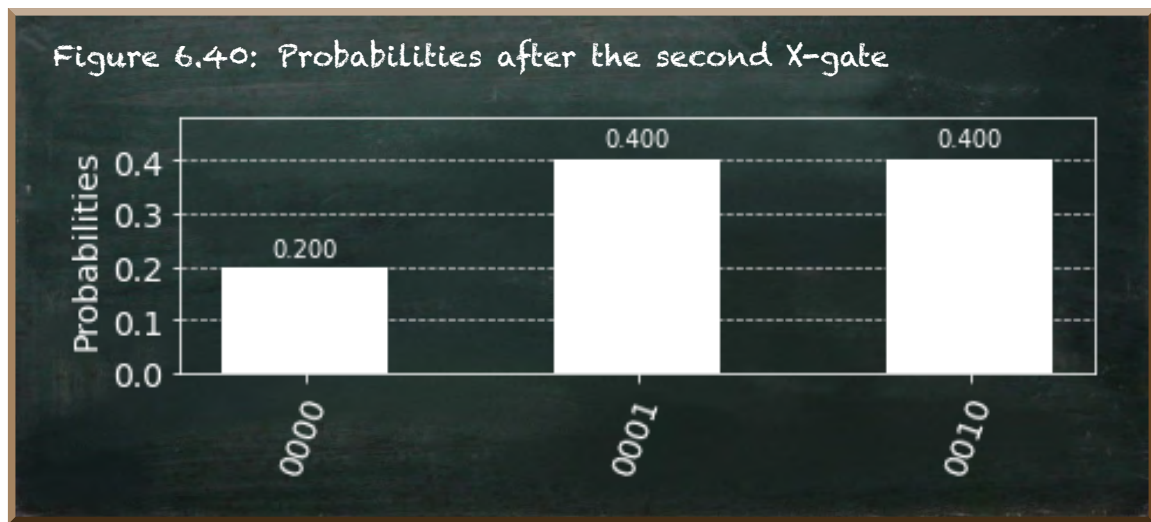
We're now prepared to apply the reduced *modifier* to one of the priors.

We can now cut the part of the modifier out of one of the priors. Again, we choose the lower qubit so that we have the resulting ones at the right-hand side.

Listing 6.22: Apply the modifier on a separated prior

```
1  # Specify the prior probability and the modifier
2  prior = 0.4
3  modifier = 1.2
4
5  qc = QuantumCircuit(4)
6
7  # Apply prior to qubit 0
8  qc.ry(prob_to_angle(prior), 0)
9
10 # Apply prior to qubit 1
11 qc.x(0)
12 qc.cry(prob_to_angle(prior/(1-prior)), 0, 1)
13 qc.x(0)
14
15 # Apply the modifier to qubit 2
16 qc.cry(prob_to_angle(modifier-1), 0,2)
17
18 run_circuit(qc)
```

Figure 6.41: Result of applying the modifier on a separated prior
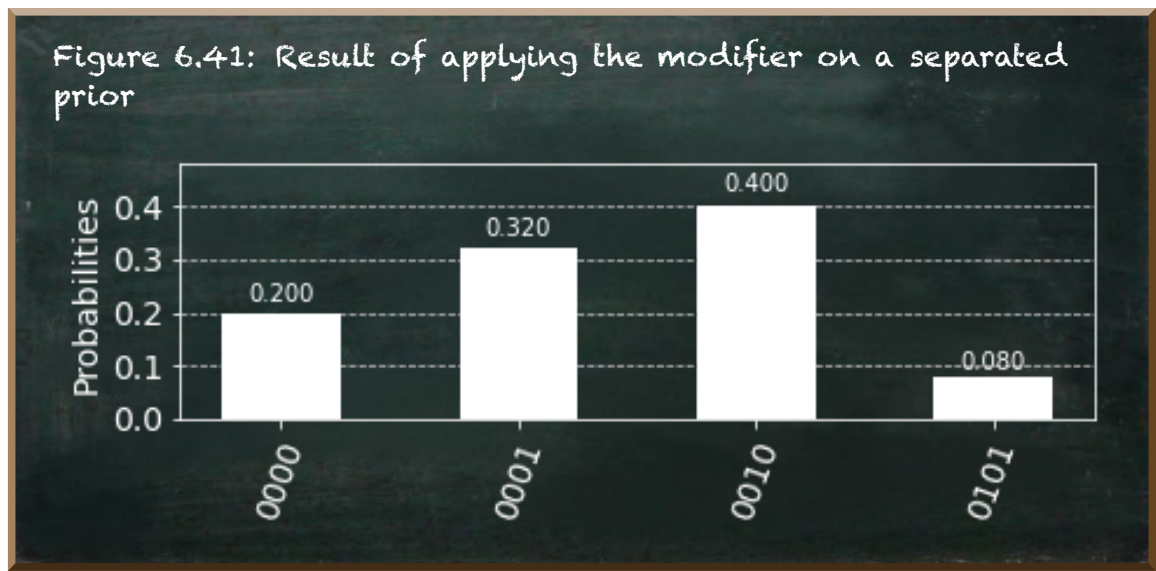
Now, the states 0010 and 0101 add up to the *posterior* probability. Let's clean this up a little more. Wouldn't it be nice to have a single one qubit representing the conditional?

First, we apply the CNOT-gate on qubits 1 and 3 with qubit 1 as the control qubit (`qc.cx(1,3)`). If qubit 1 is 1 it applies the NOT-gate on qubit 3.

The following figure depicts the state after this gate.

Figure 6.42: State after qc.cx(1,3)
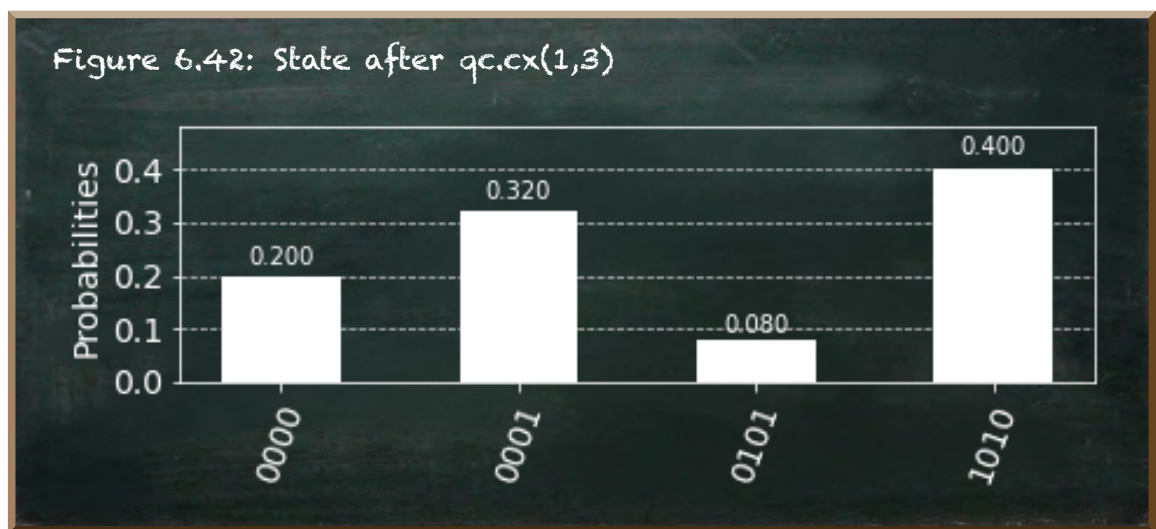
As usual, the CNOT-gate does not change the probabilities we see. It only changes the states representing them. In this case, the state 0010 was the only

state where qubit 1 is 1. This state has now changed to 1010. The only difference is that qubit 3 is 1 in the given case now, too.

Listing 6.23: Qubit 3 represents the posterior

```
1  # Specify the prior probability and the modifier
2  prior = 0.4
3  modifier = 1.2
4
5  qc = QuantumCircuit(4)
6
7  # Apply prior to qubit 0
8  qc.ry(prob_to_angle(prior), 0)
9
10 # Apply prior to qubit 1
11 qc.x(0)
12 qc.cry(prob_to_angle(prior/(1-prior)), 0, 1)
13 qc.x(0)
14
15 # Apply the modifier to qubit 2
16 qc.cry(prob_to_angle(modifier-1), 0,2)
17
18 # Make qubit 3 represent the posterior
19 qc.cx(1,3)
20 qc.cx(2,3)
21
22 run_circuit(qc)
```


Figure 6.43: Qubit 3 represents the posterior
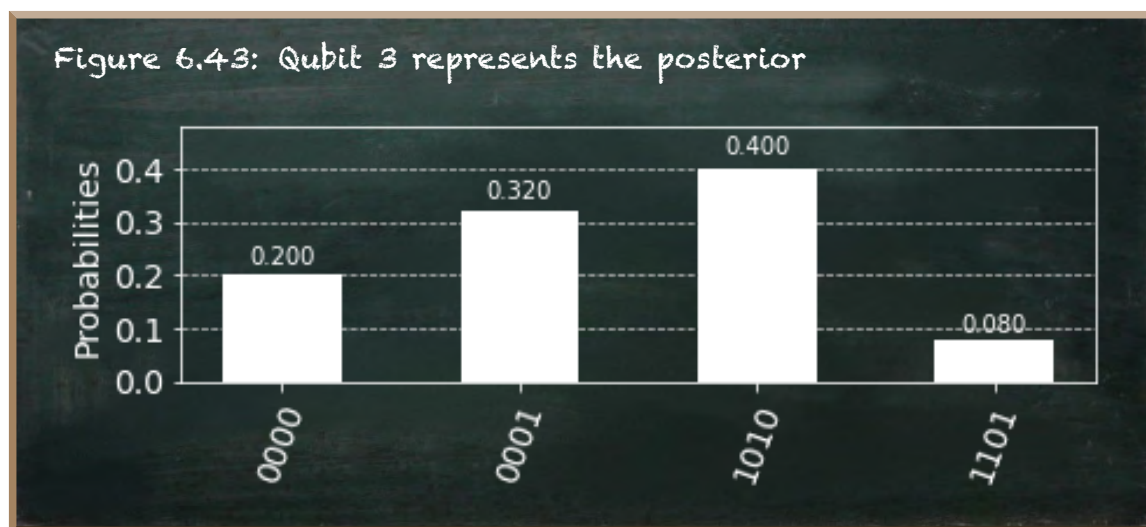
Next, we want to do the same for state 0101. Since this state is the only state

where qubit 2 is 1 we can use the CNOT-gate again to set qubit 3 to 1 if qubit 2 is 1. The following code contains all the steps.

We apply two CNOT-gates (lines 19-20). The size of the bars did not change. But the states representing them did. Then, we measure qubit 3 as 1 with the conditional probability.

So far, so good. There's but one problem. This approach does not work for a prior greater than 0.5 because we only have a total probability of 1.0 to work with. But if the *prior* is greater 0.5, we can't have two independent states representing it.

Have a look at what happens.

Listing 6.24: A prior greater than 0.5 and a modifier greater than 1

```python
1  # Specify the prior probability and the modifier
2  prior = 0.6
3  modifier = 1.2
4
5  qc = QuantumCircuit(4)
6
7  # Apply prior to qubit 0
8  qc.ry(prob_to_angle(prior), 0)
9
10 # Apply prior to qubit 1
11 qc.x(0)
12 qc.cry(prob_to_angle(prior/(1−prior)), 0, 1)
13 qc.x(0)
14
15 # Apply the modifier to qubit 2
16 qc.cry(prob_to_angle(modifier−1), 0,2)
17
18 # Make qubit 3 represent the posterior
19 qc.cx(1,3)
20 qc.cx(2,3)
21
22 run_circuit(qc)
```

```
ValueError: math domain error
```

Again, we get a `math domain error`. Mathematically, it fails when calculating (prior/(1−prior) because the term would be greater than 1, and thus, it is not a

valid input for the `prob_to_angle`-function. For instance:

$$0.6/(1.0 - 0.6) = 0.6/0.4 = 1.5$$

Solving this situation is a little tricky. I'd argue it is even a hack.

If you're a mathematician, I'm quite sure you won't like it. If you're a programmer, you might accept it. Let's have a look, first. Then, it's open for criticism.

When the *prior* is greater than `0.5`, and the *modifier* is greater than `1.0`, the trick with using the *prior* twice does not work because our overall probability must not exceed `1.0`.

Of course, we could use the *prior* to adjusting the remaining probability so that we can precisely apply the *modifier* afterward. But in this case, we would need to know the *prior* when we apply the *modifier*. This would not be different than initializing the qubit with the product of $prior * modifier$ in the first place.

But we aim for a qubit system that represents a given prior, and that lets us apply a modifier without knowing the prior. So, we need to prepare the remainder $(1 - prior)$ in a way that lets us work with it (that means we apply the reduced *modifier*) without knowing the *prior*.

Rather than using the *prior* when we apply the *modifier* to the remainder, we pre-apply the *prior* to the remainder with some auxiliary steps. For instance, we set aside a part that is `0.3` of the `prior`.

We can do this in the same way we set aside the entire *prior* earlier.

Listing 6.25: Setting aside a part of the prior

```python
1  # Specify the prior probability
2  prior = 0.6
3
4  qc = QuantumCircuit(4)
5
6  # Apply prior to qubit 0
7  qc.ry(prob_to_angle(prior), 0)
8
9  # Apply 0.3*prior to qubit 1
10 qc.x(0)
11 qc.cry(prob_to_angle(0.3*prior/(1-prior)), 0, 1)
12
13 run_circuit(qc)
```

Figure 6.44: Result of setting aside a part of the prior
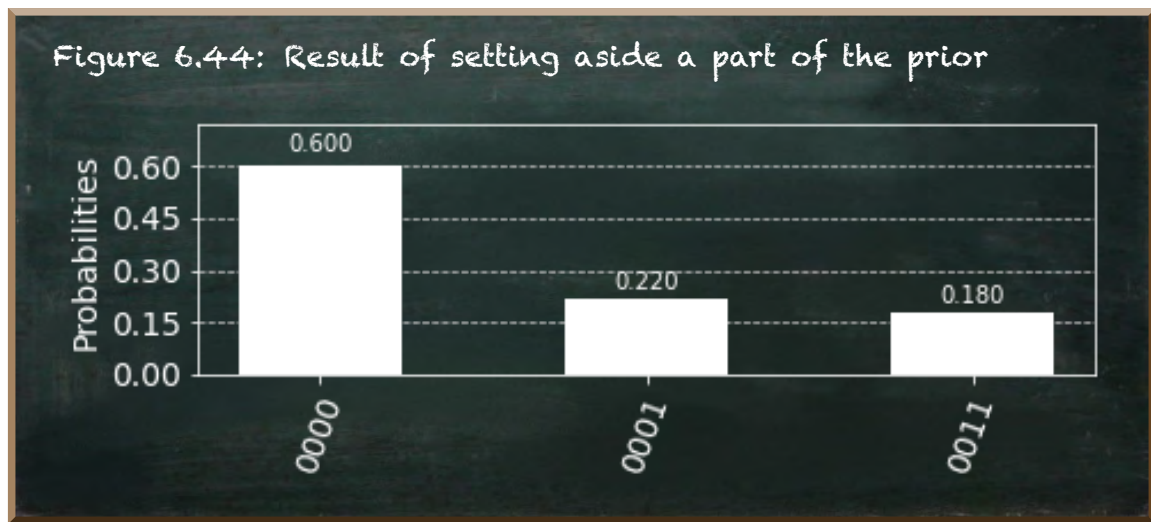
As a result, state 0000 represents the *prior* probability (0.6) and the state 0011 represents $0.3 * prior = 0.18$. We can now apply the reduced *modifier* to this state without knowing the *prior*. Let's have a look.

Listing 6.26: Calculating the posterior for prior > 0.5

```
 1  # Specify the prior probability and the modifier
 2  prior = 0.6
 3  modifier = 1.2
 4
 5  qc = QuantumCircuit(4)
 6
 7  # Apply prior to qubit 0
 8  qc.ry(prob_to_angle(prior), 0)
 9
10  # Apply 0.3*prior to qubit 1
11  qc.x(0)
12  qc.cry(prob_to_angle(0.3*prior/(1-prior)), 0, 1)
13
14  # Apply the modifier to qubit 2
15  qc.cry(prob_to_angle((modifier-1)/0.3), 1,2)
16
17  # Make qubit 3 represent the posterior
18  qc.x(0)
19  qc.cx(0,3)
20  qc.cx(2,3)
21
22  run_circuit(qc)
```

Figure 6.45: Resulting posterior
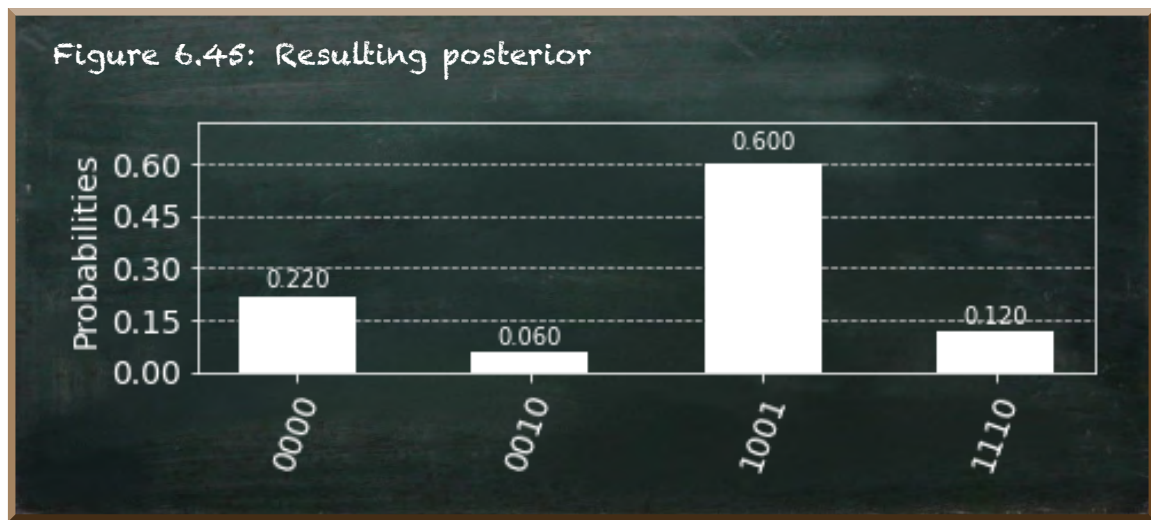
Up until line 12, there's nothing new. The important part is line 15.

Listing 6.27

```
1  qc.cry(prob_to_angle((modifier−1)/0.3), 1,2)
```

We apply a controlled R_Y-gate. Thus, we only change states where qubit 1 is
1. This is the case for the state 0010 that represents 0.3 of the *prior*. The im-
portant part is that we adjust our reduced *modifier* to 0.3 by dividing by it. If
the portion is only 0.3 of the *prior*, we need to separate an accordingly greater
part.

The remaining code (lines 18-20) changes the states to get the resulting condi-
tional probability by measuring qubit 3.

There's a caveat, though. Of course, there is. You may have wondered how I
came up with 0.3. The fraction we choose must be smaller than the remaining
probability $(1 − prior)$. If it weren't, we would exceed the overall probability
of 1.0 again. But it must be greater than the effect the *modifier* has, too. If it is
too small, we can't separate a part of it that accounts for the *modifier*'s effect
on the *prior*.

So, when settling for the best value, we need to know both *prior* and *modi-
fier*. This is where the solution becomes a hack. While we don't want to work
with the *prior* and the *modifier* simultaneously, we do not set aside one spe-
cific fraction of the *prior*. But we set aside many of them. We set aside all the
portions from 0.1 to 1.0. This way, we are prepared for any modifier up to 2.0.

Listing 6.28

```
1  for i in range(1,10):
2      qc.cry(prob_to_angle(min(1, (i*0.1)*prior/(1−prior))), 0,i)
```

To not feed the `prob_to_angle`-function with a value greater than `1.0`, we limit the input with the `min` function. So, whenever the part we want to set aside is bigger than the remainder, we only set aside the remainder. However, this means that this part is useless. It does not represent the corresponding portion of the *prior* anymore.

When we apply the modifier, we need to select the correct part. This is the smallest possible one that is big enough to contain the *modifier*'s effect.

We calculate the maximum of the reduced *modifier* by multiplying it by 10 (the reverse of the step size we chose above). The `ceil` function rounds that up. So, we have the next greater position.

Listing 6.29

```
1  pos = ceil((modifier−1)*10)
2  qc.cry(prob_to_angle((modifier−1)/(pos*0.1)), pos,11)
```

But what if we chose a part that does not correctly represent the corresponding portion of the *prior*? Technically, we get the wrong result. However, this is only the case when the actual result (*prior* ∗ *modifier*) exceeds `1.0`. Such a result would not make any sense in the first place. It would imply that observing a certain event would cause another event to occur with a probability greater than `1`. In that case, we would need to question our input data.

> ⚠ Depending on the step size we choose, there is a little area close to `1` where the resulting probability is not calculated correctly.

So, let's have a look at the final code. Due to the number of qubits we're using for the prepared parts, we exceed the limits of what can be represented in the histogram. Rather than showing all the states, we include a measurement into the circuit. We measure qubit 3 that holds the result (line 31).

A measured qubit is either `0` or `1`. We receive only a single number as output, not the probability. But we can run the circuit several times (here 1000 shots,

line 33) to calculate the resulting probability. Due to the empiric reconstruction of the probability, it is not perfectly accurate, though.

Listing 6.30: Calculating the posterior with a prior greater than 0.5

```
 1  from math import ceil
 2  from qiskit import ClassicalRegister, QuantumRegister
 3
 4  # Specify the prior probability and the modifier
 5  prior = 0.6
 6  modifier = 1.2
 7
 8  # Prepare the circuit with qubits and a classical bit to hold the
        measurement
 9  qr = QuantumRegister(12)
10  cr = ClassicalRegister(1)
11  qc = QuantumCircuit(qr, cr)
12
13  # Apply prior to qubit 0
14  qc.ry(prob_to_angle(prior), 0)
15
16  # Separate parts of the prior
17  qc.x(0)
18  for i in range(1,10):
19    qc.cry(prob_to_angle(min(1, (i*0.1)*prior/(1-prior))), 0,i)
20
21
22  # Apply the modifier
23  pos = ceil((modifier-1)*10)
24  qc.cry(prob_to_angle((modifier-1)/(pos*0.1)), pos,11)
25
26  # Make qubit 11 represent the posterior
27  qc.x(0)
28  qc.cx(0,11)
29
30  # measure the qubit
31  qc.measure(qr[11], cr[0])
32
33  run_circuit(qc, simulator='qasm_simulator', shots=1000 )
```

Figure 6.46: Posterior with a prior greater than 0.5
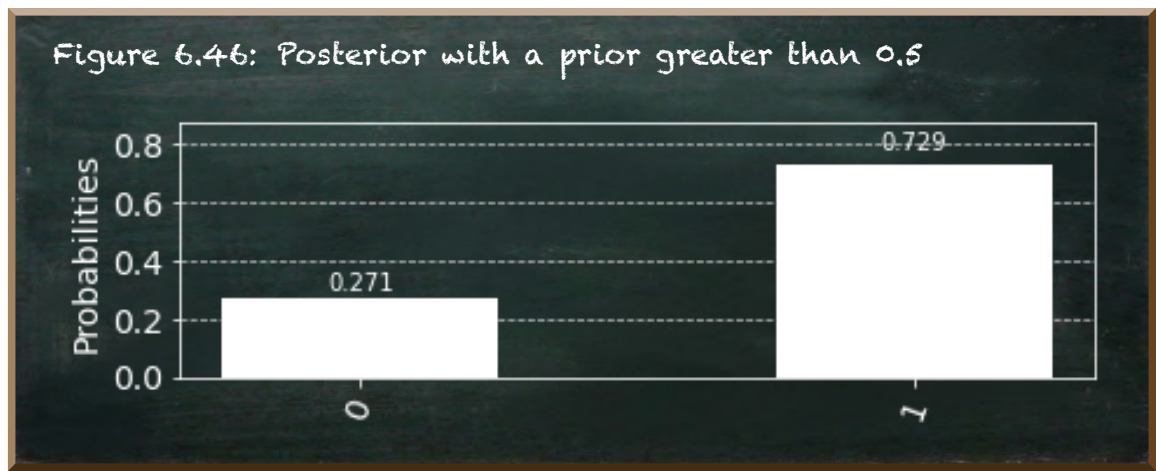
The circuit correctly calculates the conditional probability given a *prior* and a *modifier*. We have seen that it gets quite tricky to calculate the conditional for a *prior* greater than `0.5` and a *modifier* greater `1.0`.

In this example, we prepare for a modifier up to `2.0`. While this is enough to consider all cases for a prior greater than `0.5`, the modifier could be even greater if the *prior* is accordingly smaller. Therefore, to completely separate applying the *prior* from using the *modifier*, we need to consider these cases.

If we considered all possible cases, we would end up with lots of required qubits. Alternatively, we could sacrifice some precision for edge cases when we're close to a probability of `1.0`.

7. Quantum Naïve Bayes

Naïve Bayes is a probabilistic machine learning algorithm based on Bayes' Theorem. Even though it is pretty simple, it has been successfully used in a wide variety of classification tasks.

We tap the theoretical and practical knowledge we gathered in the last few chapters and use it to build a Quantum Naïve Bayes classifier. Similar to our previous quantum classifier we introduced in section 3.6, the Quantum Naïve Bayes is a Variational Hybrid Quantum-Classical Algorithm. It consists of three parts:

- We pre-process the data on a classical computer to determine the modifiers for a set of features.
- We apply the modifiers in a quantum circuit and measure the qubit that represents the posterior probability.
- We post-process the measurement and transform it into a prediction that we evaluate with the labels from our training data set.

In section 3.6, we used the pre-processing to create the final quantum state, and we only used the quantum circuit to measure it. This time, we go beyond creating a simple quantum state to be measured. This time, we make a quantum circuit that includes the calculation of the probabilities.

Figure 7.1 shows the overall architecture of our simple Variational Hybrid Quantum-Classical Algorithm.

If you read this book from the beginning, I suppose you know Bayes' Theo-

Figure 7.1: A Variational Hybrid Quantum-Classical Algorithm
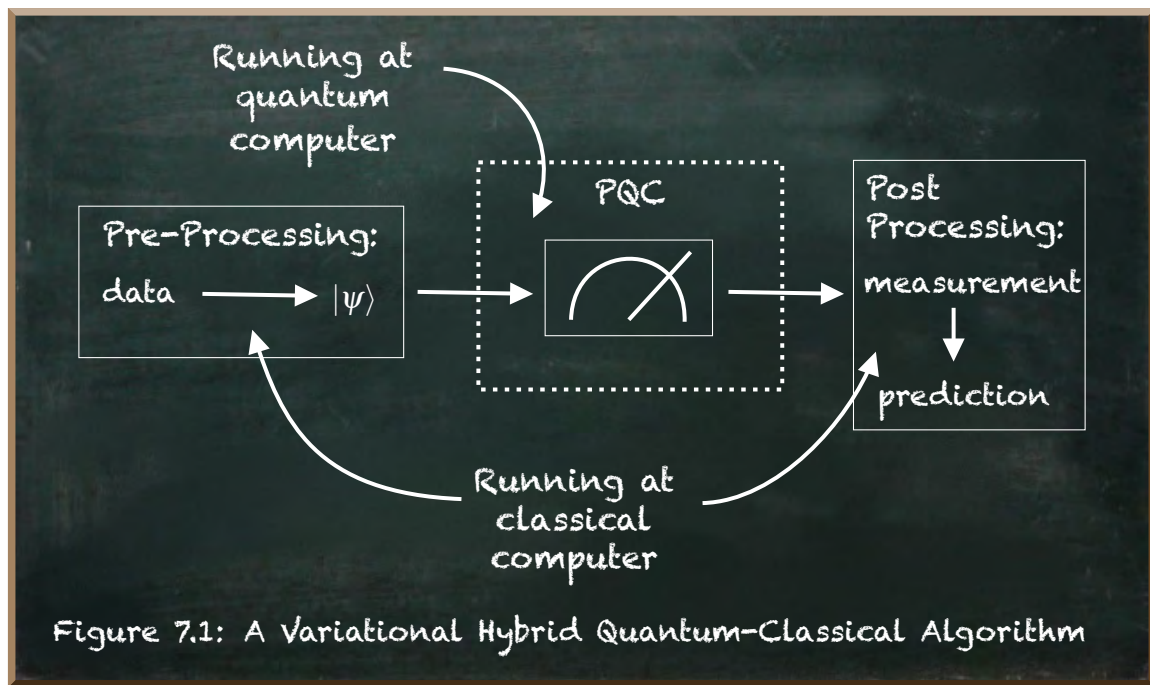
rem by heart already. If not, here's the formula one more time.

$$\underbrace{P(Hypothesis|Evidence)}_{posterior} = \underbrace{P(Hypothesis)}_{prior} \cdot \underbrace{\frac{P(Evidence|Hypothesis)}{P(Evidence)}}_{modifier}$$

If we have one hypothesis H and multiple evidences $E_1, E_2, \ldots, E_n$, then we have n modifiers $M_1, M_2, \ldots, M_n$:

$$\underbrace{P(H|E_1, E_2, \ldots, E_n)}_{posterior} = \underbrace{\frac{P(E_1|H)}{P(E_1)}}_{M_1} \cdot \underbrace{\frac{P(E_2|H)}{P(E_2)}}_{M_2} \cdot \cdots \cdot \underbrace{\frac{P(E_n|H)}{P(E_n)}}_{M_n} \cdot \underbrace{P(H)}_{prior}$$

This formula tells us we can represent each characteristic of a passenger by a modifier that changes her probability to survive the Titanic shipwreck.

In the pre-processing, we calculate these modifiers for two features, the ticket class and the gender of the passenger. We limit our Quantum Naïve Bayes classifier to these two features to keep our quantum circuit as simple as possible so that we can concentrate on the underlying structure of the algorithm.

Listing 7.1: Calculate the prior-probability

```python
1  import pandas as pd
2  train = pd.read_csv('./data/train.csv')
3
4  # total
5  cnt_all = len(train)
6
7  # list of all survivors
8  survivors = train[train.Survived.eq(1)]
9  cnt_survivors = len(survivors)
10
11 # calculate the prior probability
12 prob_survival = len(survivors)/cnt_all
13
14 print("The prior probability to survive is: ", round(prob_survival, 2))
```

```
The prior probability to survive is:  0.38
```

We start easy. We import Pandas (line 1) and load the training data from the raw CSV file (line 2). We use the raw data because we only cope with categorical data (*Pclass* and *Sex*) and these two data do not miss for any of the passengers (see section 2.3).

We calculate the prior (marginal) probability of surviving the Titanic shipwreck (line 12) as the ratio between the number of survivors (line 9) and the total number of passengers (line 5) in our dataset.

We see a prior probability of roughly 0.38.

7.1 Pre-processing

The pre-processing covers the calculation of the modifiers. We start with the ticket class.

Listing 7.2: Calculating the ticket class modifier

```python
# get the modifier given the passenger's pclass
def get_modifier_pclass(pclass):
  # number of passengers with the same pclass
  cnt_surv_pclass = len(survivors[survivors.Pclass.eq(pclass)])

  # backward probability
  p_cl_surv = cnt_surv_pclass/cnt_survivors

  # probability of the evidence
  p_cl = len(train[train.Pclass.eq(pclass)])/cnt_all

  return p_cl_surv/p_cl
```

We define a function that takes the passenger's `pclass` as input. The `Pclass` column in our dataset is the ticket class (1 = 1st, 2 = 2nd, 3 = 3rd).

We calculate the backward probability $P(Pclass|Survived)$ by dividing the passengers who survived having the given ticket class (`cnt_surv_pclass`, line 4) by all survivors (`cnt_survivors`, line 7). Then, we calculate the probability of a passenger owning the given ticket class. It is the number of passengers with the given ticket class divided by the total number of passengers (line 10).

The modifier is the evidence's backward probability divided by the likelihood to see the evidence. For the given ticket class, the modifier is $m_{Pclass} = \frac{P(Pclass|Survived)}{P(Pclass)}$ (line 12).

Listing 7.3: Calculating the gender modifier

```python
# get the modifier given the passenger's pclass
def get_modifier_sex(sex):
  # number of passengers with the same pclass
  cnt_surv_sex = len(survivors[survivors.Sex.eq(sex)])

  # backward probability
  p_sex_surv = cnt_surv_sex/cnt_survivors

  # probability of the evidence
  p_sex = len(train[train.Sex.eq(sex)])/cnt_all

  return p_sex_surv/p_sex
```

The calculation of the modifier for a passenger's gender works accordingly.

We calculate the backward probability `p_sex_surv` as the number of survivors of the given gender divided by the total number of survivors (line 7).

The probability of a passenger having the given gender is the number of passengers with the given gender divided by the total number of passengers (line 10). The function returns the modifier $m_{Sex} = \frac{P(Sex|Survived)}{P(Sex)}$ (line 12).

These two functions serve as helper functions in our `pre_process`-function.

Listing 7.4: Pre-processing

```python
def pre_process(passenger):
  """
  passenger -- the Pandas dataframe-row of the passenger
  returns a list of modifiers, like this [modifier_a, modifier_b, ...]
  """
  return [
    get_modifier_pclass(passenger["Pclass"]),
    get_modifier_sex(passenger["Sex"]),
  ]
```

The actual pre-processing is quite simple. The function `pre_process` takes the passenger data as a row of a *Pandas* Dataframe. It takes the actual passenger's values for `Pclass` and `Sex` and calls the respective helper functions (lines 7-8). It returns the modifiers it gets back from these functions in a list.

Let's have a look at the modifiers of two exemplary passengers–a female passenger with a first-class ticket and a male passenger with a third-class ticket.

Listing 7.5: Two exemplary passengers

```python
# a female passenger with 1st class ticket
print (pre_process(train.iloc[52]))

# a male passenger with 3rd class ticket
print (pre_process(train.iloc[26]))
```

```
[1.6403508771929822, 1.9332048273550118]
[0.6314181584306999, 0.49215543190732464]
```

The modifiers vary between 0.49 and 1.93. The modifiers of the male passenger are below 1.0. No modifier exceeds 2.0. The female passenger with a first-class ticket has two high modifiers above 1.6. When we apply these two modifiers on the prior probability, we get a posterior probability of $0.38 * 1.64 * 1.93 = 1.20$. This value exceeds the maximum probability of 1.0.

In our PQC, we need to consider these things.

7.2 PQC

In the previous section 6.3.3, we learned how to calculate a posterior (conditional) probability given a prior probability and a modifier. We build upon this approach.

Thus, in general, we:

- apply the prior to a qubit,
- set aside an auxiliary qubit to hold a fraction of the prior,
- and apply the modifier. If the modifier is above 1.0, we use the auxiliary to add to the prior.

The implementation of this approach in our Quantum Naïve Bayes classifier has some challenges, though. While the posterior probability calculation is easy when the modifier is below 1.0, it becomes tricky when it is above 1.0. And, it becomes tough when the prior probability is above 0.5 at the same time.

Now, we could sigh of relief our prior is 0.38 - below 0.5. However, we apply two modifiers in a row. Once we applied the first modifier, the result becomes the new prior when we apply the second modifier. For instance, if the modifier represents the first-class ticket (modifier of 1.64) or a female passenger (modifier of 1.93), then the new prior is above 0.5 (0.62 or 0.73). Moreover, there are passengers whose modifiers are both above 1.0, as in the case of a female passenger with a first-class ticket. In this case, we need two auxiliary qubits to add to prior.

So, here's our refined approach.

- We apply the prior to a qubit,
- set aside two auxiliary qubits to hold a fraction of the prior each,
- order the modifiers starting with the lower one,
- and apply the first modifier. If the modifier is above 1.0:
    - We use the first auxiliary to add to the prior.
    - We use the second auxiliary to "refill" and adjust the first auxiliary to reflect the size of the new prior.

We start with the basic structure of our PQC. This `pqc` function takes a reusable `Qiskit` backend, a prior probability, and the modifiers representing a passenger as mandatory parameters. Further, we let the `pqc` function take optional parameters we can use during the development.

Listing 7.6: The basic pqc-function

```python
from functools import reduce
from qiskit import QuantumCircuit, Aer, execute, ClassicalRegister,
    QuantumRegister
from math import asin, sqrt, ceil
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

def prob_to_angle(prob):
    """
    Converts a given P(psi) value into an equivalent theta value.
    """
    return 2*asin(sqrt(prob))

def pqc(backend, prior, modifiers, shots=1, hist=False, measure=False):
    # Prepare the circuit with qubits and a classical bit to hold the
        measurement
    qr = QuantumRegister(7)
    cr = ClassicalRegister(1)
    qc = QuantumCircuit(qr, cr) if measure else QuantumCircuit(qr)

    # INSERT QUANTUM CIRCUIT HERE

    # measure the qubit only if we want the measurement to be included
    if measure:
        qc.measure(qr[0], cr[0])
    results = execute(qc,backend, shots=shots).result().get_counts()
    return plot_histogram(results, figsize=(12,4)) if hist else results
```

The `shots` parameter allows the caller to specify the number of times to run the quantum circuit. Note, this works only in combination with the `qasm_simulator` as the backend.

The `hist` parameter lets us specify whether we want to return a histogram (`hist=True`) or the raw data (`hist=False`).

The `measure` parameter allows us to easily switch between including the measurement of a qubit (`measure=True`) into the circuit or not. It is usually helpful not to include the measurement during development because it allows us to

use the `statevector_simulator` backend. This computes all the states of the individual qubits. But once you have too many qubits in your circuit, things become hard to follow. Then, it may be helpful to use the `qasm_simulator` backend and measure a single one qubit you're interested in. Since you would only get a single number and not a probability distribution anymore, you can use the `shots` parameter to run the circuit multiple times. This way, you get back the approximate probability. Of course, it is only approximate because it is an empiric reconstruction of the probability and not a precise calculation. But it is pretty accurate most of the time.

In this code listing, we also added the `prob_to_angle` function we already used before, and we will use it here, too.

In the `pqc` function, we start with the definition of the `QuantumCircuit`. We will use seven qubits in total in this circuit (line 15). Depending on the `measure` parameter, we add a `ClassicalRegister` (line 16) to receive the measurement of a qubit (lines 22-23).

Once the quantum circuit is wholly specified (with or without a measurement), we execute it with the given `backend` and specify how many times (`shots`) we want the circuit to run (line 24).

Depending on the `hist` parameter, the function returns the plot of a histogram or the raw results (line 25)

Unless indicated otherwise, the following code listings become part of the `pqc` function. To keep them small, we skip the repetition of the code we already discussed.

Whenever we apply transformation gates in Qiskit, we need to specify the qubit index we want to apply the gate on. For once, numbers are not that easy to remember as words. Second, we might want to change the position a qubit with a specific purpose has. Therefore, we define and use constant values to keep the indexes of our qubits. Let's start with the target qubit.

Listing 7.7: Set the target qubit to represent the prior probability

```
1  # target qubit has position 6
2  target = 6
3
4  # Apply prior to qubit to the target qubit
5  qc.ry(prob_to_angle(prior), target)
```

The effect of this step is simple. First, we measure the target qubit as 1 with the prior probability. The following figure depicts the state.

Figure 7.2: The state after applying th eprior to the target
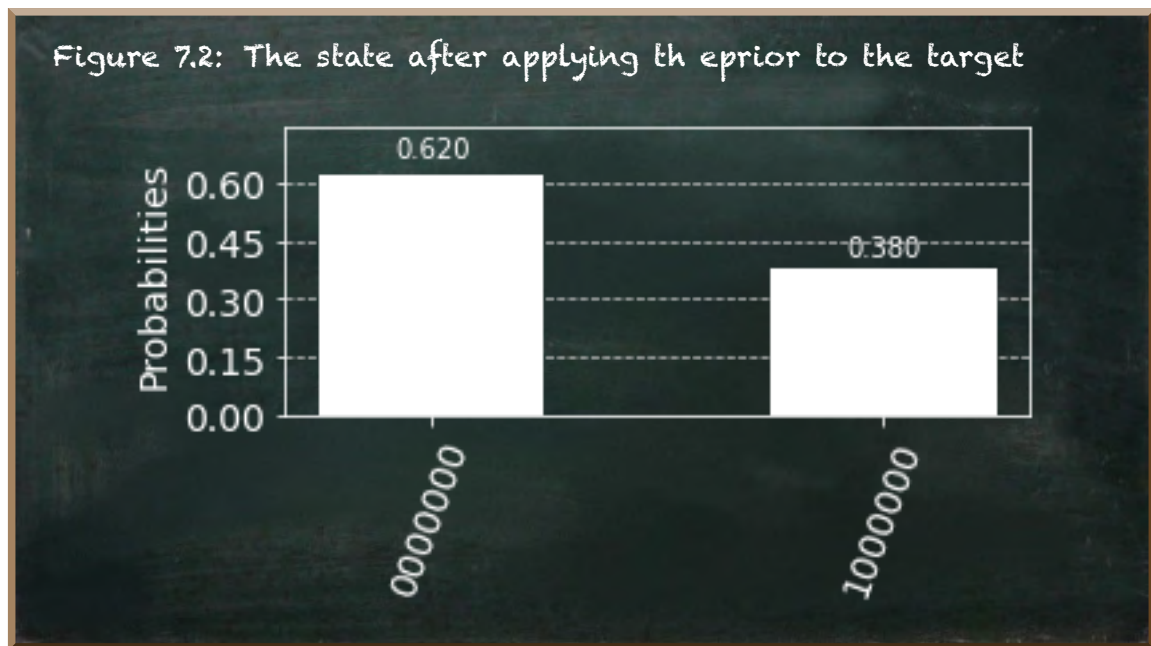
In the next step, we apply the prior to an auxiliary qubit we call `aux_full` because it represents the whole prior.

Listing 7.8: Apply prior to aux-full-qubit

```
1  # auxiliary qubit has position 5
2  aux_full = 5
3
4  # Work with the remainder
5  qc.x(target)
6
7  # Apply prior to full auxiliary qubit
8  qc.cry(prob_to_angle(prior/(1−prior)), target, aux_full)
```

We apply the NOT-gate to change the qubit 0 so that it is 1 for the remaining 0.62 rather than the 0.38 of the prior (line 5). This allows us to apply the following controlled R_Y-gate only on the part that does not overlap with the target qubit (line 8). Since the remainder is smaller than 1.0, we need to adjust the rotation to the smaller part that is now $1 - prior$.

The resulting state shows that two states match the prior probability.

Figure 7.3: The state after applying the prior to the aux-full qubit
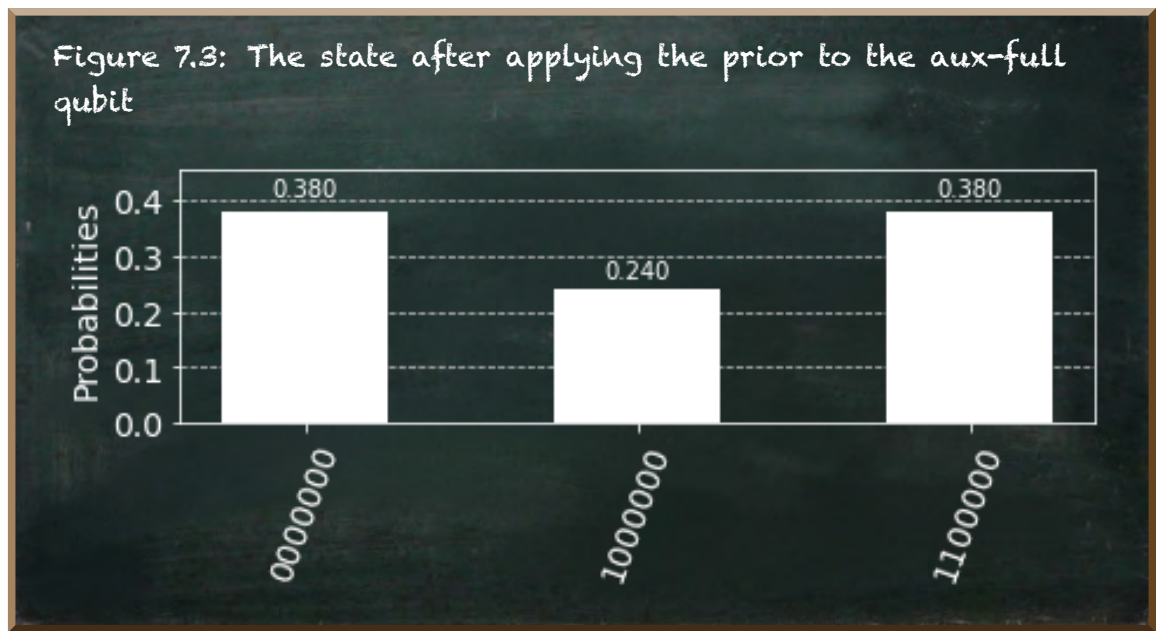
With two states representing the prior of 0.38, we have enough space left to let another qubit represent half of the prior. This qubit has the name `aux_half`.

Listing 7.9: Apply half the prior to aux-half-qubit

```
1  # second auxiliary qubit has position 4
2  aux_half = 4
3
4  # Work with the remainder
5  qc.cx(aux_full,target)
6
7  # Apply 0.5*prior to qubit 1
8  qc.cry(prob_to_angle(0.5*prior/(1-(2*prior))), target,aux_half)
9
10 # Rearrange states to separated qubits
11 qc.x(target)
12 qc.cx(aux_full, target)
```

We need to work with the remainder again. Thus, we need to set `aux_full` qubit to `0` for the state where it represents the prior. This time, however, we can't use a normal NOT-gate. While it would have the desired effect for the `aux_full` qubit, it would mix up the state where the `target`-qubit represents the prior.

Instead, we use a controlled NOT-gate (line 5). This separates the remainder

of the remainder. Finally, we rearrange the states so that each qubit represents the prior (or half the prior) without overlap (lines 11-12).

The result shows that it works. We see two states with the entire prior probability and one state with half of the prior probability, each represented by a single qubit measured as 1.

Figure 7.4: The state after applying half the prior to the aux-half qubit
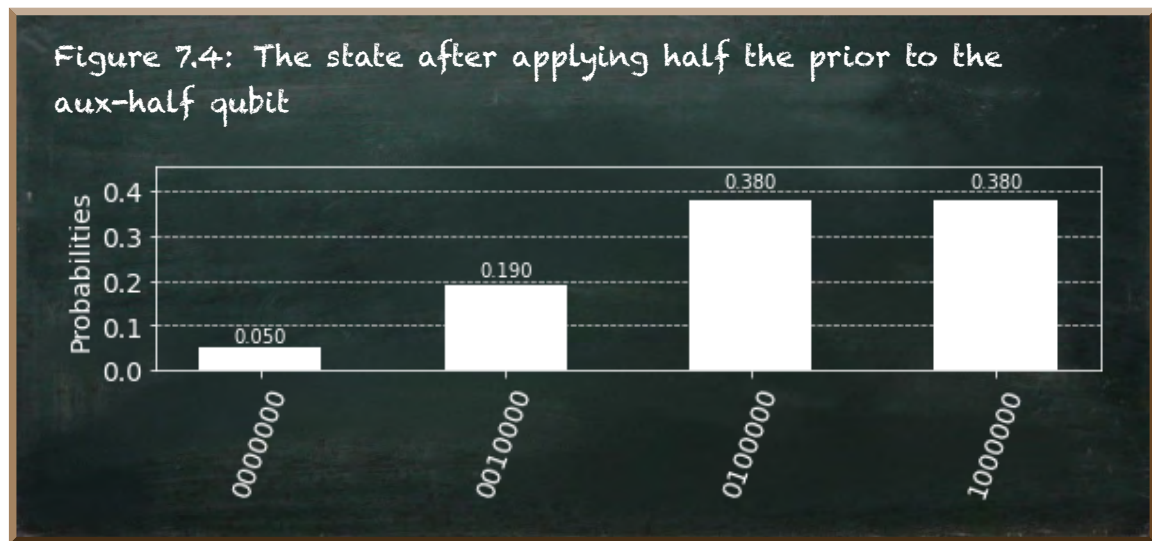
We're now prepared to work with the modifiers. Since the highest modifier we expect is 1.93 for a passenger's ticket class and 1.64 for a passenger's gender, the `aux_full`-qubit is sufficient to apply any modifier (because `prior`+`aux_full` $= 2*$`prior`).

But the `aux_half`-qubit may not suffice. Both modifiers are above 1.5, and thus, they exceed the resulting probability we could represent with the help of `aux_half` (because `prior`+`aux_half`$= 1.5*$`prior`).

Unless both modifiers are above 1.5, we can use the `aux_full`-qubit to represent the larger one and the `aux_half`-qubit to represent the lower one. Therefore, we sort the modifiers, beginning with the larger. If both modifiers are above 1.5, we have a problem. However, the resulting probability would exceed 1.0. In this case, we would need to limit the modifiers to the maximum possible values, anyway.

In any case, starting with the larger modifier is a good idea.

Listing 7.10: Sort the modifiers

```
1  # sort the modifiers
2  sorted_modifiers = sorted(modifiers)
```

In the next step, we iterate through the sorted modifiers. Depending on whether the modifier is above or below 1.0, we need to do different things. Let's start with the easy case, a modifier below 1.0.

In this case, we use the controlled R_Y-gate to calculate the posterior probability.

Listing 7.11: Calculate the posterior probability for a modifier smaller than 1.0

```
1  for step in range(0, len(modifiers)):
2    if sorted_modifiers[step] > 1:
3      # TO BE IMPLEMENTED
4      pass
5
6    else:
7      # apply modifier to the target qubit
8      qc.cry(prob_to_angle(1−sorted_modifiers[step]), target, step*2)
9      qc.cx(step*2,target)
```

If the modifier is below 1.0 (block after line 6), we need to reduce the target probability by a portion of $1 - modifier$ (line 8). The `target`-qubit acts as the control qubit. Thus, in states when the `target` qubit is 1, we separate a part of $1 - modifier$ and set the controlled qubit to 1. The controlled qubit has the index `step*2`. For step is 0, the index of this qubit is 0, too. This qubit acts as a trunk. We do not work with it anymore.

The controlled rotation does not change the value of the control qubit. Thus, the `target`-qubit is 1 in both parts. The following CNOT-gate (line 9) changes this. If our trunk-qubit is 1 (which is only the case we just separated), we reverse the value of the `target`-qubit from 1 to 0.

Figure 7.5: Result of a modifier of 0.8
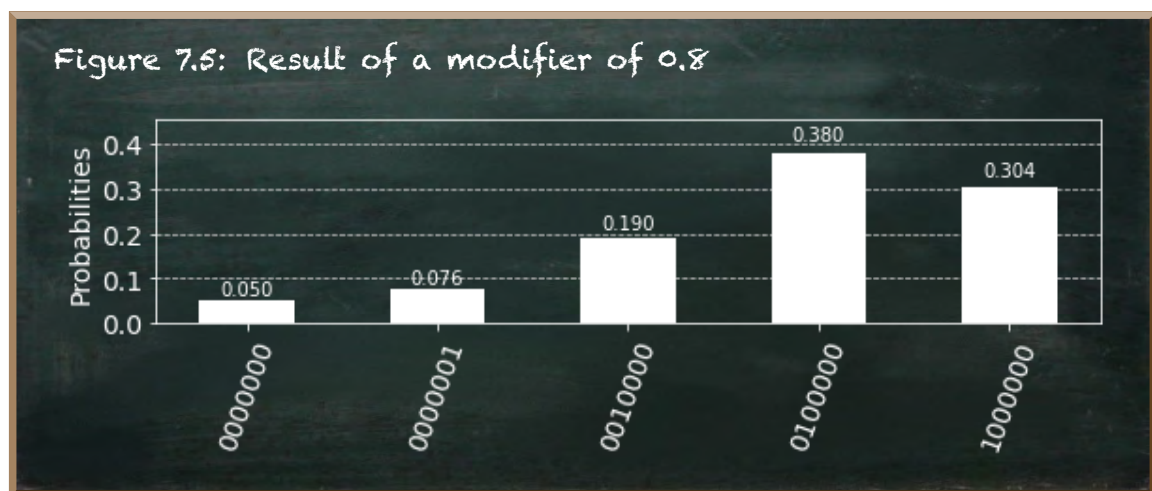
The figure depicts the state if the first modifier is 0.8. The state at the right-hand side is the only state where the `target`-qubit is `1` with a probability of $0.38 * 0.8 = 0.304$.

We also see that the `aux_full`-qubit still represents the prior probability of 0.38. Technically, we might want to apply the modifier on the `aux_full`-qubit because we could need it in the second step. However, since we sorted the modifiers and the greater one is below 1.0, we can be sure the second modifier is below 1.0, too.

In this case, the same step works. We apply a controlled rotation with the `target` qubit as a control qubit and a new trunk-qubit as the target qubit.

Let's move to the more exciting part. The modifier is above 1.0.

Listing 7.12: Calculate the posterior probability for a modifier greater than 1.0

```
1  if sorted_modifiers[step] > 1:
2    qc.cry(prob_to_angle(min(1, sorted_modifiers[step]−1)), aux_full,
         target)
3
4    # separate the aux_full and the target qubit
5    qc.ccx(target, aux_full, 0)
6    qc.ccx(target, 0, aux_full)
```

If the modifier is above 1.0, we apply a controlled rotation with our `target`-qubit as the controlled qubit. Thus, we "add" to it. The `aux_full`-qubit serves as a control qubit (line 2).

The two *CCNOT*-gates (lines 5-6) separate states where the `aux_full` and the `target` qubits are `1`. As a result, the sum of all states where the `target` qubit is `1` represents the posterior probability after applying the first modifier.

The figure depicts the state after applying a modifier of 1.4.

Essentially, we "moved" some probability from the `aux_full`-qubit to the `target`-qubit. But since the first modifier was above 1.0, the second might be too. In this case, the `aux_full`-qubit would not be appropriate anymore. It has a smaller probability, now.

Figure 7.6: Result of a modifier of 1.4

0.380

0.228

0.190

0.152

0.050

Figure 7.6: Result of a modifier of 1.4
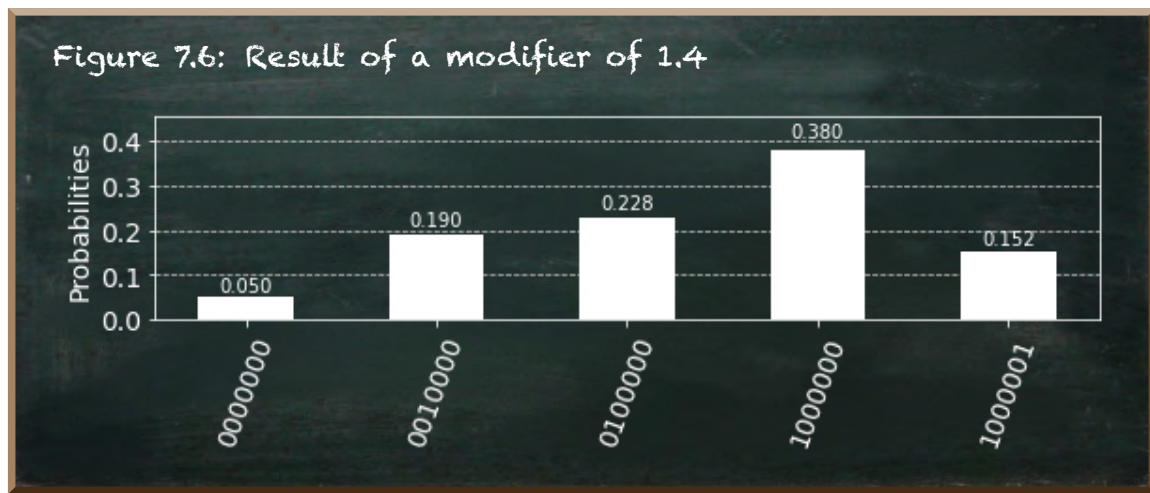
But we have another auxiliary qubit left. So, we use this qubit to "refill" our `aux_full`-qubit.

Listing 7.13: refill the aux-full-qubit

```
1  if step == 0:
2    # equalize what we transferred to the target (*2) and increase the
     aux_full to reflect the modifier (*2)
3    qc.cry(prob_to_angle(min(1,(sorted_modifiers[step]−1)*2*2)), aux_half,
     aux_full)
```

After we applied the first modifier (line 1), we use another controlled R_Y-gate to "move" probability from the `aux_half`-qubit to the `aux_full`-qubit. The interesting question here is: how much?

The answer is the angle that represents four times the modifier. We reduced the probability of measuring the `aux_full`-qubit as 1 when we used the modifier to move the probability to the `target`-qubit. We want the `aux_full`-qubit to represent the posterior probability after applying the first modifier as the new prior probability before applying the second modifier. Thus, we must not only "refill" the `aux_full`-qubit, but we must apply the modifier on it, too. This makes two.

We need to remember the `aux_half`-qubit represents only half the probability of the `aux_full`. Thus, we need to apply twice the rotation we want to "move." This makes four.

The following figure shows the `target`-qubit and the `aux_full`-qubit after we applied the first modifier.

Figure 7.7: Result after preparing the aux-full-qubit
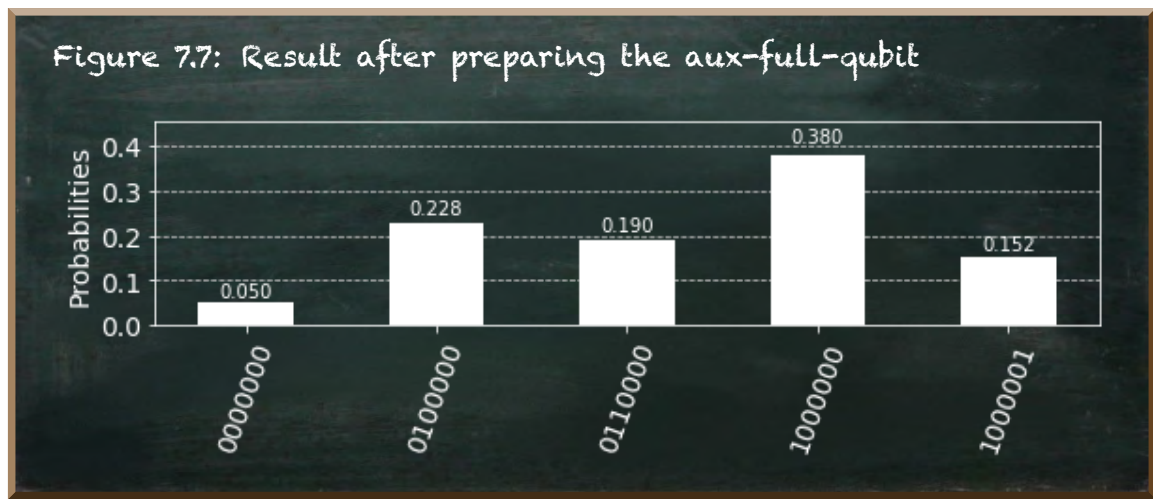
If the second modifier is below 1.0, we don't need the `aux_full`. But when it is greater 1.0, we have our auxiliary qubit well-prepared.

The following figure depicts the state after we applied the second modifier of 1.3.


Figure 7.8: Result of two modifiers of 1.4 and 1.3
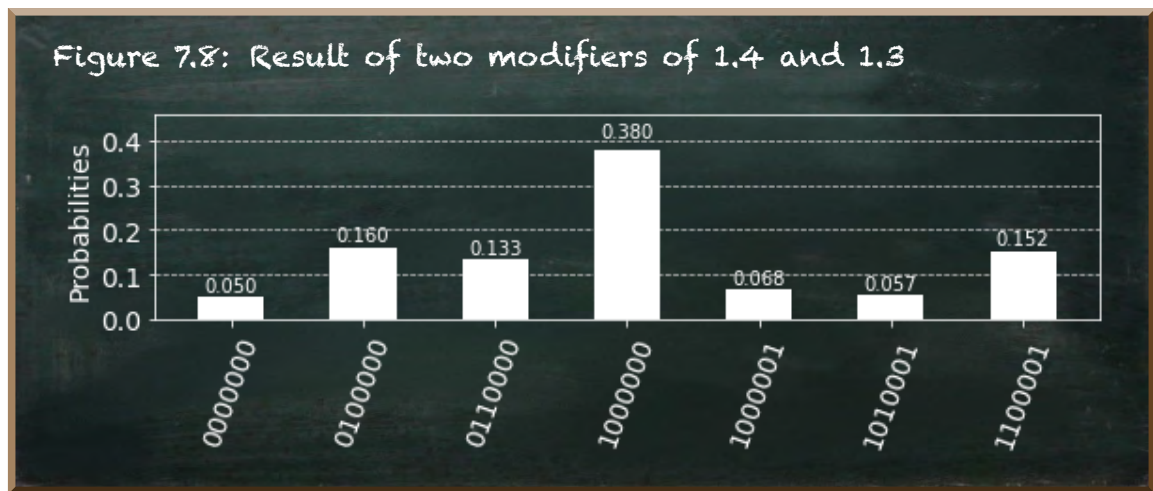
Altogether, there are seven different states. The posterior probability is the sum of all states where the `target`-qubit is 1.

We prepared the `pqc`-function to include measurement. Let's use this ability to look at the resulting probability of measuring the `target`-qubit as 1.

Listing 7.14: Include the measurement into the circuit

```
1 plot_histogram(pqc(Aer.get_backend('qasm_simulator') , 0.38, [1.4, 1.3],
      shots=1000, hist=False, measure=True))
```


Figure 7.9: Result of two modifiers of 1.4 and 1.3, including measurement
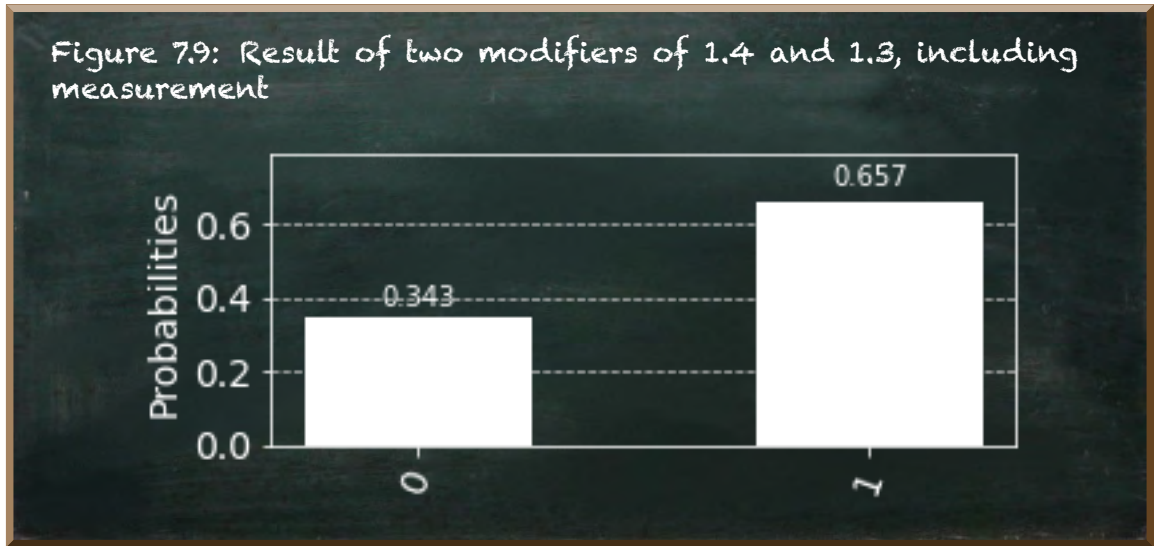
We can see a posterior probability that is very close to the exact probability of $0.38 * 1.4 * 1.3 = 0.69$. However, the empirical nature of the `qasm_simulator` causes the fuzziness of the measurement.

This leaves the question of what happens if both modifiers result in a posterior probability that exceeds 1.0.

In this case, we would "move" the complete probability of the `aux_half`-qubit to the `aux_full`-qubit after the first step. And, we would move most of it to the `target`-qubit in the second step. As a result, we would see a very high `probability` of measuring our `target`-qubit as 1. But it wouldn't be accurate.

However, I'd argue that there is no accurate result in this case at all.

Here's the result after applying the modifiers of a female passenger with a first-class ticket (modifiers: `[1.6403508771929822, 1.9332048273550118]`).

This time, let's use the `qasm_simulator` to see the result of the measured qubit only.

Listing 7.15: Calculate the posterior of a female passenger with a first-class-ticket

```
1  plot_histogram(pqc(Aer.get_backend('qasm_simulator') , 0.38,
       [1.6403508771929822, 1.9332048273550118], shots=1000, hist=False,
       measure=True))
```


Figure 7.10: Result of a female passenger with a first-class-ticket
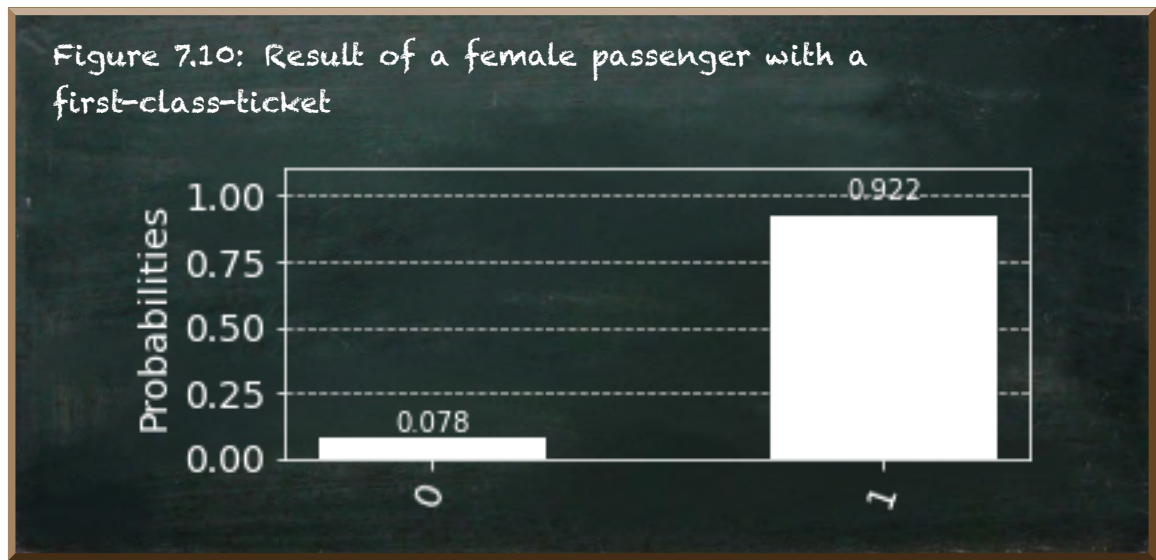
The following listing depicts the complete code of the pqc-function.

Listing 7.16: The parameterized quantum circuit - part 1

```python
def pqc(backend, prior, modifiers, shots=1, hist=False, measure=False):
    # Prepare the circuit with qubits and a classical bit to hold the
      measurement
    qr = QuantumRegister(7)
    cr = ClassicalRegister(1)
    qc = QuantumCircuit(qr, cr) if measure else QuantumCircuit(qr)

    # the qubit positions
    trunks = 3
    aux = trunks+1
    aux_half = trunks+1
    aux_full = trunks+2
    target = trunks+3

    # Apply prior to qubit to the target qubit
    qc.ry(prob_to_angle(prior), target)

    # Work with the remainder
    qc.x(target)

    # Apply prior to full auxiliary qubit
    qc.cry(prob_to_angle(prior/(1−prior)), target, aux_full)

    # Work with the remainder
    qc.cx(aux_full,target)

    # Apply 0.5*prior to qubit 1
    qc.cry(prob_to_angle(0.5*prior/(1−(2*prior))), target,aux_half)

    # Rearrange states to separated qubits
    qc.x(target)
    qc.cx(aux_full, target)

    sorted_modifiers = sorted(modifiers)

    # CONTINUED...
```

Listing 7.17: The parameterized quantum circuit - part 2

```python
# CONTINUE
for step in range(0, len(modifiers)):
  if sorted_modifiers[step] > 1:
    qc.cry(prob_to_angle(min(1, sorted_modifiers[step]−1)), aux_full,
  target)

    # separate the aux_full and the target qubit
    qc.ccx(target, aux_full, 0)
    qc.ccx(target, 0, aux_full)

    if step == 0:
      # equalize what we transferred to the target (*2) and increase
  the aux_full to reflect the modifier (*2)
      qc.cry(prob_to_angle(min(1,(sorted_modifiers[step]−1)*2*2)),
  aux_half, aux_full)

  else:
    # apply modifier to the target qubit
    qc.cry(prob_to_angle(1−sorted_modifiers[step]), target, step*2)
    qc.cx(step*2,target)

    if step == 0:
      # apply modifier to full auxiliary qubit
      qc.cry(prob_to_angle(1−sorted_modifiers[step]), aux_full, step
  *2+1)

      # unentangle the full auxiliary from trunk
      qc.cx(step*2+1,aux_full)


# measure the qubit only if we want the measurement to be included
if measure:
  qc.measure(qr[target], cr[0])
results = execute(qc,backend, shots=shots).result().get_counts()
return plot_histogram(results, figsize=(12,4)) if hist else results
```

7.3   Post-Processing

We need to post-process the results we receive from the `pqc`. If we set the `hist`-parameter to `False`, the `pqc` function returns the counts. These are in the form of a Python dictionary with two keys `'0'` and `'1'`. The values assigned to these keys are the number of measurements that yielded the respective key as a

result. For instance, if we have $1,000$ shots and 691 returned 1, our result is
`{'0':209, '1':691}`.

If we have a single one-shot, we will get either `{'0': 1}` or `{'1': 1}` as counts.
When we run our classifier, we want to get a distinct prediction for each passenger. Thus, a single shot is sufficient.

Listing 7.18: Post-processing

```python
def post_process(counts):
  """
  counts -- the result of the quantum circuit execution
  returns the prediction
  """
  return int(list(map(lambda item: item[0], counts.items()))[0])
```

The `post_process`-function takes the counts the `pqc`-function returns. It looks for the keys (`counts.items()`) and returns the first. The underlying assumption is we have only one measurement and thus, we have only one key representing the prediction.

Finally, we can put it all together.

Listing 7.19: Run the Quantum Naive Bayes Classifier

```python
# redefine the run-function
def run(f_classify, data):
  return [f_classify(data.iloc[i]) for i in range(0,len(data))]

# specify a reusable backend
backend = Aer.get_backend('qasm_simulator')

# evaluate the Quantum Naive Bayes classifier
classifier_report("QuantumNaiveBayes",
  run,
  lambda passenger: post_process(pqc(backend, prob_survival, pre_process(
    passenger), measure=True, hist=False)),
  train,
  train['Survived'])
```

```
The precision score of the QuantumNaiveBayes classifier is 0.63
The recall score of the QuantumNaiveBayes classifier is 0.62
The specificity score of the QuantumNaiveBayes classifier is 0.78
The npv score of the QuantumNaiveBayes classifier is 0.77
The information level is: 0.70
```

First, we define the `run` function (line 2). It takes the `f_classify`-function and the data (that is, the `train`-dataset). It classifies each row in the data and returns the prediction (line 3).

We create the `qasm_simulator`-backend we can reuse for all our predictions (line 6).

We reuse the `classifier_report`-function we introduced in section 2.7 (line 9). Besides an arbitrary name it uses in the output (line 9), it takes as arguments

- the `run`-function (line 10),
- the classifier (line 11),
- the dataset (line 12),
- and the actual results (line 13).

The classifier we provide (line 11) is an anonymous function (a function without a name) that takes the `passenger` as the parameter. From inside to outside, we first pre-process the `passenger`. Then, alongside the reusable backend (line 6) and the prior probability of survival (`prob_survival`), we call the `pqc` with the modifiers we get from the `pre_process`-function. Finally, we call the `post_process`-function that returns the overall prediction.

The overall information level of the Quantum Naïve Bayes classifier is about 0.70. This is almost the same level we achieved before with the first Variational Hybrid Quantum-Classical Classifier we developed in section 3.6.

This Quantum Naïve Bayes classifier has quite a few problems. Foremost, we can compute two modifiers above 1.0 at most. The approach we applied does not scale well. While we use a quantum circuit, we do not exploit the possible advantage quantum computing may provide. We need to think differently.

8. Quantum Computing Is Different

8.1 The No-Cloning Theorem

Quantum computing is fundamentally different from classical computing. To master quantum computing, you must unlearn what you have learned.

Figure 8.1: Quantum Computing is different
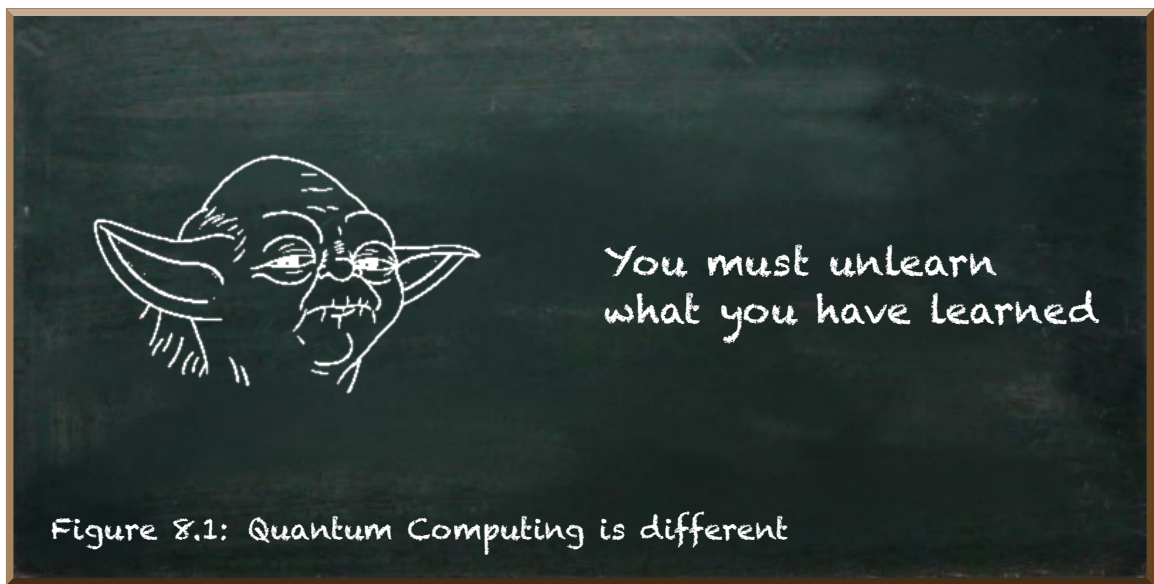
It starts with the quantum superposition. Unlike a classical bit, a quantum bit (qubit) is not 0 or 1. Unless you measure it, the qubit is in a complex linear combination of 0 and 1. But when you measure it, the quantum superposition collapses, and the qubit is either 0 or 1, as a classical bit.

It continues with quantum entanglement. Two qubits can share a state of superposition. Once you measure one qubit, its entangled fellow instantly jumps to a different state of superposition. Even if it is light-years away, it seems to know a measurement has taken place, and it takes on a state that acknowledges the measured value.

When starting with quantum computing, we're tempted to focus on the possibilities that arise from superposition and entanglement. But quantum computing does not simply add new features we can use. Instead, it is a fundamentally different way of computing. And it requires a different kind of programs.

Classically, we think about input, transformational boolean logic, and output. But this thinking will not let us succeed in quantum computing. Classical control structures are a dead-end.

Let's take one of the simplest operators in classical computing, the assignment.

Listing 8.1: Copying a variable

```
my_var = 1
copy_of_my_var = my_var
print (my_var, copy_of_my_var)
```

We can use the assignment operator to create a copy of a variable. The value of the variable doesn't matter. We can create a copy of it.

In a classical program, we rely on this ability to copy data–a lot. In a classical circuit, this is the fan-out operation. In electrical engineering, we have wires. If there is a voltage at a particular time, we interpret it as 1. If there is no voltage, it is 0. We can connect another wire to it. We will receive the same output at both ends.

Figure 8.2: Classical fan-out
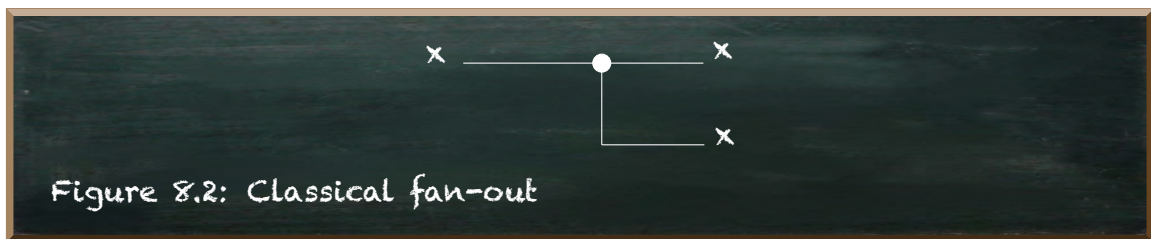
Copying data is helpful in manifold ways. First, we can use copies as inputs to different operators. For instance, in the half-adder, we copy the input to use it in two other operators.

Figure 8.3: The half-adder
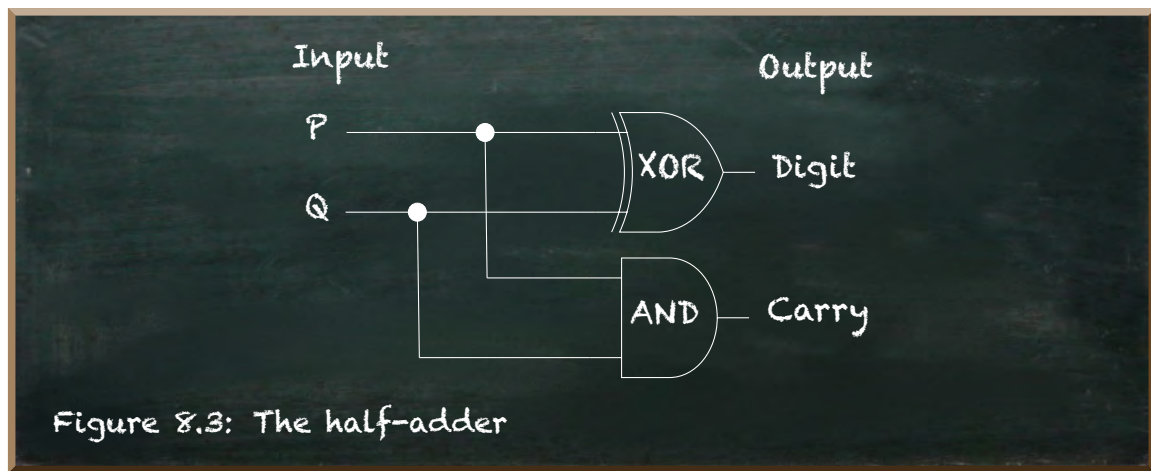
Secondly, we can use the copies to evaluate the state at different parts of the program at other times. This would be particularly useful in quantum computing.

In section 5.3, we learned how to change a qubit's measurement probabilities by rotating it around the y-axis. We got to know the angle θ, which controls the probability of measuring the qubit as 0 or 1. But we struggled with the problem that θ is the angle between the vector $|\psi\rangle$ and the basis state vector $|0\rangle$. But if the qubit is not in the basis state $|0\rangle$, then the same value of θ represents a different probability change. The gradients of trigonometric functions (such as sine and arcsine) are not constant. Thus, the probability an angle represents at the top of the circle (state $|0\rangle$) is another probability that the same angle represents at the horizontal axis such as the state $|+\rangle$. To calculate the correct angle, we need to consider the state the qubit is currently in.

But measuring the qubit state collapses it to either 0 or 1. So, measuring destroys the qubit superposition. But, if you're not allowed to measure the qubit, how could you specify the prior probability?

Wouldn't it be good to create a copy of the qubit before we measure it? Then, we would measure one copy of the qubit while continuing to work with the other copy.

Figure 8.4: An army of clones

So, let's have a look at the respective operator, the fan-out. In a classical circuit, one input wire connects to two output wires. It copies a bit. In quantum computing, we use transformation gates, and we use the word cloning for the analogous idea of copying a qubit.

The following figure depicts the diagram of a cloning transformation gate.

Figure 8.5: The cloning gate

The gate (let's call it G) takes an arbitrary qubit $|\psi\rangle$ and a fixed $|0\rangle$ (an ancilla bit) as input. It outputs two copies of $|\psi\rangle$. Here are three examples of cloning transformations.

1. It clones a qubit in the state $|0\rangle$.

$$G(|0\rangle|0\rangle) = |0\rangle|0\rangle$$

2. It clones a qubit in the state $|1\rangle$.

$$G(|1\rangle|0\rangle) = |1\rangle|1\rangle$$

3. It clones an arbitrary qubit $|\psi\rangle$.

$$G(|\psi\rangle|0\rangle) = |\psi\rangle|\psi\rangle$$

The superposition state of a single-qubit consists of two basis states ($|0\rangle$ and $|1\rangle$) and two corresponding probability amplitudes α and β.

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

The squares of the amplitudes represent the probabilities of measuring the qubit as either 0 (given by α^2) or 1 (given by β^2). The sum of all probabilities is 1:

$$|\alpha|^2 + |\beta|^2 = 1$$

The cloning transformation gate works with two qubits. Let's call them $|a\rangle$ and $|b\rangle$. Each of the two qubits has its own probability amplitudes: $|a\rangle = a_0|0\rangle + a_1|1\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$ and $|b\rangle = b_0|0\rangle + b_1|1\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$.

If we look at these two qubits concurrently, there are four different combinations of the basis states ($|0\rangle|0\rangle$, $|0\rangle|1\rangle$, $|1\rangle|0\rangle$, and $|1\rangle|1\rangle$). And each of these combinations has its probability amplitude that is the product of the probability amplitudes of the two qubits' corresponding probability amplitudes. The following equation depicts our qubit system ($|a\rangle|b\rangle$) that has four possible states and corresponding probability amplitudes.

$$|a\rangle|b\rangle = a_0 b_0 |0\rangle|0\rangle + a_0 b_1 |0\rangle|1\rangle + a_1 b_0 |1\rangle|0\rangle + a_1 b_1 |1\rangle|1\rangle \tag{8.1}$$

This equation lets us represent and reason about a system that consists of two qubits. Let's use them to clone the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

We start with applying the cloning transformation gate to the arbitrary qubit.

$$G(|\psi\rangle|0\rangle) = |\psi\rangle|\psi\rangle$$

It results in two qubits in the same state $|\psi\rangle$.

We use the equation 8.1 to rewrite the state of our two qubits. Since both qubits are equal, we can say that $a_0 = b_0 = \alpha$ and $a_1 = b_1 = \beta$

We represent the qubit state as the sum of each state we could measure with

its respective probability amplitude. This is the result of cloning a qubit.

$$|\psi\rangle|\psi\rangle = \alpha^2|0\rangle|0\rangle + \alpha\beta|0\rangle|1\rangle + \beta\alpha|1\rangle|0\rangle + \beta^2|1\rangle|1\rangle$$

Next, let's first expand our arbitrary qubit $|\psi\rangle$.

$$G(|\psi\rangle|0\rangle) = G((\alpha|0\rangle + \beta|1\rangle)|0\rangle)$$

We multiply out the inner term.

$$G((\alpha|0\rangle + \beta|1\rangle)|0\rangle) = G(\alpha|0\rangle|0\rangle + \beta|1\rangle|0\rangle)$$

Since the application of G is matrix multiplication, we can apply the distributive law of matrix algebra.

$$G(\alpha|0\rangle|0\rangle + \beta|1\rangle|0\rangle) = G(\alpha|0\rangle|0\rangle) + G(\beta|1\rangle|0\rangle)$$

Finally, we apply the initial specifications of how G transforms the basis states $|0\rangle$ and $|1\rangle$

$$G(\alpha|0\rangle|0\rangle) + G(\beta|1\rangle|0\rangle) = \alpha|0\rangle|0\rangle + \beta|1\rangle|1\rangle$$

We get another result of cloning a qubit in an arbitrary state. However, these two results of the cloning transformation gate are not equal.

$$\alpha^2|0\rangle|0\rangle + \alpha\beta|0\rangle|1\rangle + \beta\alpha|1\rangle|0\rangle + \beta^2|1\rangle|1\rangle \neq \alpha|0\rangle|0\rangle + \beta|1\rangle|1\rangle$$

If the cloning transformation gate G exists, then two terms that are not equal must be equal. This is a contradiction. The only logical conclusion is that G can't exist. Therefore, it is impossible to clone a qubit of an arbitrary state.

This is known as the **no-cloning theorem**. It has important consequences.

In classical computing, we rely heavily on the ability to copy. Even the simplest classical operation, the addition, relies on copying bits. But in quantum computing, it is impossible.

In quantum computing, we can't use the information stored in a qubit as many times as we want to. The idea of cloning a qubit in an arbitrary state would contradict the underlying concept of superposition. Measuring a qubit collapses its state of superposition. But when we could clone a qubit, we could measure its state indirectly. We could measure the clones without collapsing the original qubit.

This might look like a severe problem. But, it is only problematic if we continue to think classically. We need to unlearn how to solve a certain type of problem programmatically. When we want to use the unique characteristics of quantum computing, such as superposition, entanglement, and interfer-

ence, we need to learn a new way of solving problems.

8.2 How To Solve A Problem With Quantum Computing

This is what they mean with quantum computing can evaluate different states concurrently.
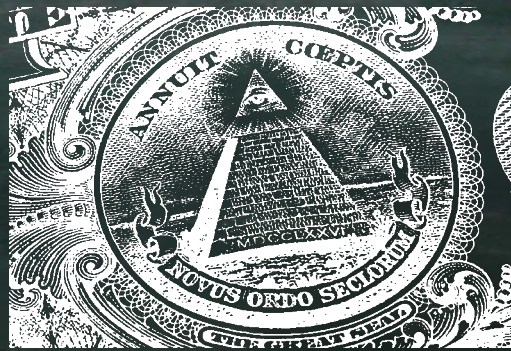

Figure 8.6: Ask the oracle

Quantum computing comes with quite a few caveats.

- When transforming qubits, you have to ensure reversibility.
- You can't copy a qubit in an arbitrary state.
- And foremost, you can't even measure a qubit without collapsing its state of superposition.

But a qubit can do things a classical bit can't. A qubit is not restricted to 0 or 1. It can be a combination of both states. Further, you can entangle two qubits so that they share a state of superposition.

With these characteristics, qubits are a powerful tool if used properly. Of course, you can treat qubits like ordinary bits and solve a problem the same way you solve other computational problems. But you would not benefit from the advantage a quantum computer promises. When solving a problem classically, you won't see any quantum speedup. The algorithm will be much slower because a quantum computer is extremely slow (in terms of clock frequency) and extremely small (in terms of the number of qubits).

In this section, we learn how to solve a problem through an algorithm that could be faster than a classical one. By that, we mean the quantum algorithm

will solve a problem in fewer steps than a classical algorithm requires. If both computers were equal in speed and size, the quantum computer would solve the problem faster. However, a classical computer might compensate for the larger number of steps by its sheer speed and size. But with the increasing complexity of the problem to solve, speed and size will not suffice anymore. There are problems on earth, such as the factorization problem, that are too complex for a classical computer, regardless of its speed and size. But they are not too complex for a quantum computer–theoretically.

The problem we use as our example is **not** one of these problems. A classical computer solves it in a split second. But the example allows us to demonstrate how to solve a problem the quantumic way.

Let's assume we have a function f. It takes a single bit as input–either 0 or 1. And it provides a single bit as its output, too. Again, either 0 or 1.

There are four different possible functions.

- Function f_0 always returns 0.
- Function f_1 returns 0 if the input is 0 and it returns 1 if the input is 1.
- Function f_2 returns 1 if the input is 0 and it returns 0 if the input is 1.
- Function f_3 always returns 1.

The functions f_0 and f_3 provide constant outputs. No matter what their input is, they always return the same result. f_0 returns 0. f_3 returns 1. Always.

The functions f_1 and f_2 provide balanced outputs because for half of the inputs, they return 0 (f_1 if the input is 0 and f_2 if the input is 1), and for the other half, they return 1 (f_1 if the input is 1 and f_2 if the input is 0).

If you're given one of these functions at random, how can you determine whether the function is constant (f_0 or f_3) or balanced (f_1 or f_2)? We don't care about the specific function we got. We only care about whether it is constant or balanced.

Classically, you have to run the function twice. You run it with the input 0. If you get 0 as a result, the function at hand could be the constant function f_0 that always returns 0. And it could be the balanced function f_1 that returns 0 if the input is 0. You have to rerun the function with input 1 to decide whether the function is constant or balanced. The constant function f_0 still returns 0. But the balanced function f_1 returns 1 for input 1.

The same situation applies if you get 1 as a result of running the function for the input 0. You would need to distinguish between the balanced function f_2 and the constant function f_3.

It does not help to run the function with input 1 first, either. With either result, 0 and 1, the function at hand could be constant or balanced. You need to rerun the function with input 0.

In quantum computing, we only need to run the function once.

We start with a new quantum gate–the gate O_i. This gate should represent our four functions f_i. Mathematically, this would be

$$O_i(|x\rangle) = |f_i(x)\rangle$$

It transforms an arbitrary quantum state $|x\rangle$ into the quantum state $|f_i(x)\rangle$ - the output of the function f_i given the input x.

For O_i is a quantum transformation gate, it must be reversible. Therefore, it must have the same size of input and output. And each output must be uniquely identity the input it originates from.

But that's a problem. The constant functions always return the same value regardless of their inputs. Given their output, you can't tell what the input was.

We can deal with this problem with a little trick. We add a second qubit $|y\rangle$, and we use the exclusive or (XOR, $\oplus$) operator to keep track of the result of the function f_i.

Mathematically, the refined gate O_i is:

$$O_i(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus f_i(x)\rangle$$

The following figure depicts the transformation gate O_i.

Figure 8.7: The O-gate
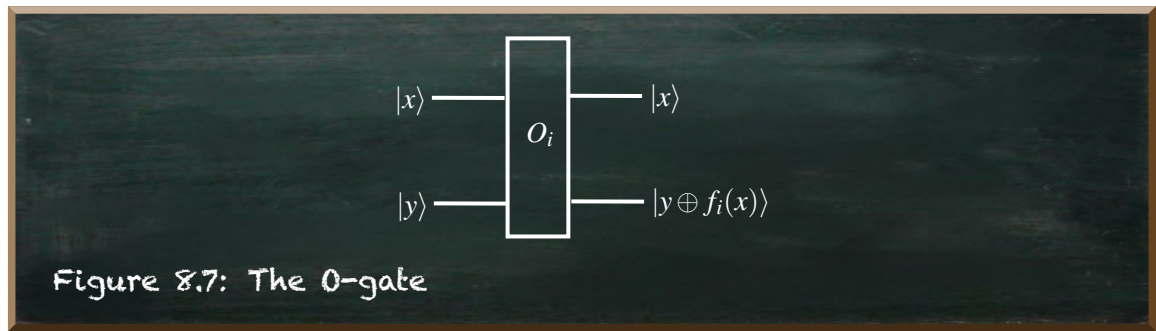
Let's say, $i = 0$. Thus, we apply the function f_0. Per definition, $f_0(x) = 0$. When we insert this into the above equation, we can see the output of O_i is equal to its input:

$$O_0(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus |f_0(x)\rangle = |x\rangle \otimes |y \oplus |0\rangle = |x\rangle \otimes |y\rangle$$

We can safely state that not changing a state is reversible.

When $i = 1$, we apply the function f_1 that returns 0 for $x = 0$ and 1 for $x = 1$. Thus, $f_1(x) = x$.

$$O_1(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus f_1(x)\rangle = |x\rangle \otimes |y \oplus x\rangle$$

The truth table of the term $|x\rangle \otimes |y \oplus x\rangle$ shows that it is reversible.

Figure 8.8: Truth table of x | y XOR x

When we apply f_2 that returns 1 for $x = 0$ and 0 for $x = 1$, we can say $f_2(x) = x \oplus 1$.

$$O_2(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus f_2(x)\rangle = |x\rangle \otimes |y \oplus x \oplus 1\rangle$$

The truth table discloses that the term $|x\rangle \otimes |y \oplus x \oplus 1\rangle$ is reversible, too.

Figure 8.9: Truth table of x | y XOR x XOR 1

Finally, f_3 always returns 1.

$$O_3(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus |f_3(x)\rangle = |x\rangle \otimes |y \oplus 1\rangle)$$

The output is like the input but with a reversed y.

Our refined gate O_i is a valid transformation gate for all our functions f_i.

But, we're not too interested in how the functions f_i work. Rather, we regard O_i as a black box. The two important things are

x	y		x	y ⊕ 1
0	0		0	1
0	1		0	0
1	0		1	1
1	1		1	0

Figure 8.10: Truth table of x | y XOR 1

- O_i is a valid two-qubit gate for all i
- The output of O_i

The following truth table shows how the O_i-gate transforms pairs of qubits in the basis states.

x	y	$\lvert x\rangle \otimes \lvert y \oplus f_i(x)\rangle$
0	0	$\lvert 0\rangle \otimes \lvert f_i(0) \oplus 0\rangle = \lvert 0\rangle \otimes \lvert f_i(0)\rangle$
0	1	$\lvert 0\rangle \otimes \lvert f_i(0) \oplus 1\rangle$
1	0	$\lvert 1\rangle \otimes \lvert f_i(1) \oplus 0\rangle = \lvert 1\rangle \otimes \lvert f_i(1)\rangle$
1	1	$\lvert 1\rangle \otimes \lvert f_i(1) \oplus 1\rangle$

Figure 8.11: Truth table of the O-gate

As usual, when we only look at qubits in the basis states, there is nothing special with a quantum circuit. However, things get interesting when the qubits are in a state of superposition.

Let's input the states $\lvert +\rangle$ (for x) and $\lvert -\rangle$ (for y). We can construct these states if we apply the Hadamard gate to the basis states $\lvert 0\rangle$, respectively $\lvert 1\rangle$.

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

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

Collectively, the two qubits are in the state

$$
\begin{aligned}
H(|0\rangle) + H(|1\rangle) &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) + \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\
&= \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)
\end{aligned}
\tag{8.2}
$$

Now, we apply the gate O_i. Thus, we replace the four basis states with the terms we calculated in the truth table above.

$$
\begin{aligned}
&O\left(H(|0\rangle) + H(|1\rangle)\right) \\
&\qquad = \frac{1}{2}\left(|0\rangle \otimes |f_i(0)\rangle - |0\rangle \otimes |f_i(0) \oplus 1\rangle + |1\rangle \otimes |f_i(1)\rangle - |1\rangle \otimes |f_i(1) \oplus 1\rangle\right)
\end{aligned}
\tag{8.3}
$$

We can rearrange this term by putting the basis states of the first qubit outside the brackets.

$$
= \frac{1}{2}\left(|0\rangle \otimes (|f_i(0)\rangle - |f_i(0) \oplus 1\rangle) + |1\rangle \otimes (|f_i(1)\rangle - |f_i(1) \oplus 1\rangle)\right)
$$

Let's have a closer look at the term $|f_i(0)\rangle - |f_i(0) \oplus 1\rangle$.

- For $f_i(0) = 0$, the term is $|0\rangle - |1\rangle$.
- For $f_i(0) = 1$, the term is $-|0\rangle + |1\rangle$.

We can rewrite it as $(-1)^{f_i(0)}(|0\rangle - |1\rangle)$.

The term $(-1)^{f_i(0)}$ takes care of the signs. For $f_i(0) = 0$, it is 0 because anything with exponent 0 is 1. For $f_i(0) = 1$, it is -1, yielding $-|0\rangle + |1\rangle$.

Therefore:

$$
|f_i(0)\rangle - |f_i(0) \oplus 1\rangle = (-1)^{f_i(0)}(|0\rangle - |1\rangle)
$$

The same logic applies to the term $|f_i(1)\rangle - |f_i(1) \oplus 1\rangle$.

$$
|f_i(1)\rangle - |f_i(1) \oplus 1\rangle = (-1)^{f_i(1)}(|0\rangle - |1\rangle)
$$

We insert these terms into our qubit state equation

$$
= \frac{1}{2}\left(|0\rangle \otimes ((-1)^{f_i(0)}(|0\rangle - |1\rangle)) + |1\rangle \otimes ((-1)^{f_i(1)}(|0\rangle - |1\rangle))\right)
$$

And we rearrange it, by putting the terms $(-1)^{f_i(0)}$ and $(-1)^{f_i(1)}$ outside the brackets.

$$
= \frac{1}{2}\left(((-1)^{f_i(0)}|0\rangle \otimes (|0\rangle - |1\rangle)) + (-1)^{f_i(1)}|1\rangle \otimes (|0\rangle - |1\rangle)\right)
$$

Then, we move anything except the term $|0\rangle - |1\rangle$ outside the brackets, too.

$$= \frac{1}{2}\left(\left((-1)^{f_i(0)}|0\rangle + (-1)^{f_i(1)}|1\rangle\right) \otimes (|0\rangle - |1\rangle)\right)$$

Finally, we apply the common factor $\frac{1}{2}$ to each of the resulting terms (note $\frac{1}{2} = \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}}$).

$$= \frac{1}{\sqrt{2}}\left((-1)^{f_i(0)}|0\rangle + (-1)^{f_i(1)}|1\rangle\right) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

The result is a two-qubit state in the form of $|x\rangle \otimes |y\rangle$. $|x\rangle$ and $|y\rangle$ are the two output qubits with the qubit $|x\rangle$ is the top one.

Let's have a closer look at the state of qubit $|x\rangle$.

$$|x\rangle = \frac{1}{\sqrt{2}}\left((-1)^{f_i(0)}|0\rangle + (-1)^{f_i(1)}|1\rangle\right)$$

Inside the brackets, we see the usual sum of our two basis states, $|0\rangle + |1\rangle$. Yet, the output of the function f_i determines the signs of the basis states. $f_i(0)$ controls the sign of $|0\rangle$. It is $+$ for $f_i(0) = 0$ because $(-1)^0 = 1$ and it is $-$ for $f_i(0) = 1$ because $(-1)^1 = -1$. Accordingly, $f_i(1)$ controls the sign of $|1\rangle$.

The following table depicts the possible values of the qubit $|x\rangle$ in dependence of f_i.

$f_i(0)$	$f_i(1)$	$(-1)^{f_i(0)}|0\rangle + (-1)^{f_i(1)}|1\rangle$
0	0	$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$
0	1	$\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$
1	0	$\frac{1}{\sqrt{2}}(-|0\rangle + |1\rangle) = |-\rangle$
1	1	$\frac{1}{\sqrt{2}}(-|0\rangle - |1\rangle) = |+\rangle$

Figure 8.12: The resulting state of qubit |x>

Do you remember the section 3.2? The graphical representation of the qubit state lets us distinguish between the states $|+\rangle$ and $|-\rangle$. While we usually use

$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ when we write $|+\rangle$ in terms of the basis states, we could as well use $\frac{1}{\sqrt{2}}(-|0\rangle - |1\rangle)$.

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

The same accounts for the state $|-\rangle$, respectively

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

The upfront constant amplitude of $\frac{1}{\sqrt{2}}$ applies to both basis states. Its square $(\frac{1}{\sqrt{2}})^2 = \frac{1}{2}$ is the probability of measuring either one state. So, half of the time, we measure the qubit as 0. And half of the time, we measure the qubit as 1. This applies to both states $|+\rangle$ and $|-\rangle$. The measurement probabilities are the same for all four functions f_i.

Unless we apply another transformation gate to this qubit. If we apply the Hadamard gate to the first qubit, it transforms the state $|+\rangle$ into $|0\rangle$. And it transforms $|-\rangle$ into $|1\rangle$.

Thus, we measure the qubit as 0 with certainty, if the output of our gate O_i is $|+\rangle$. This is the case for f_0 because $f_0(0) = 0$ and $f_0(1) = 0$. And it is the case for f_3 because $f_3(0) = 1$ and $f_3(1) = 1$. These are the two constant functions.

Accordingly, we always measure 1 if O_i outputs $|-\rangle$. This is the case for f_1 and f_2. These are the balanced functions.

So, no matter what function f_i we plug into our circuit, we can tell whether it is constant or balanced by running it only once–something we can't achieve classically.

The following figure depicts the complete quantum circuit.

Figure 8.13: The full quantum circuit
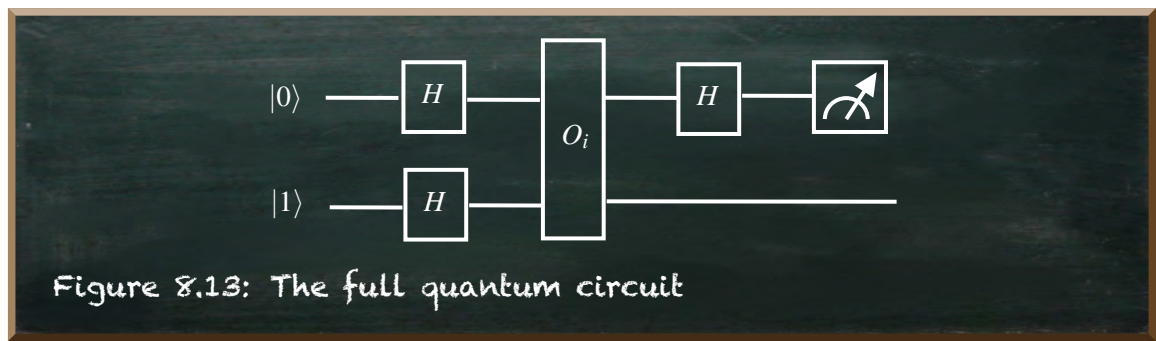
David Deutsch developed this algorithm. He was the first to prove quantum algorithms can reduce the query complexity of solving a particular problem.

The query complexity is the number of times we have to evaluate a function to get an answer.

The underlying idea of this algorithm is to treat solving a problem as the search for a function. We don't know how the function works internally. It is a black box. We don't have to worry about it.

But we know all the possible outcomes of the function. And this knowledge allows us to represent the function by a quantum gate. This gate is called a quantum **oracle**. Therefore, we named it O-gate. Rather than repeatedly calling the function to analyze how it works, we savvily craft a quantum algorithm around the oracle to separate the outcomes that solve the problem from those that don't. In other words, we ask the oracle for an answer.

At this point, it is of utmost importance to understand what we are asking for. The oracle will answer precisely the question we ask. Literally, it does not care about what you intended to ask for.

In our example, we asked the oracle whether the function is constant or balanced. And this is what we get as an answer. Not more, not less.

The following code shows a function that embeds a discrete oracle in a quantum circuit.

This circuit includes a measurement of the first qubit. Therefore, our `QuantumCircuit` contains a `QuantumRegister` with two qubits (line 11) and a `ClassicalRegister` with one bit (line 12) to hold the measurement.

By default, both qubits are initialized in the state $|0\rangle$. We put the second qubit into the state $|1\rangle$ by applying the X-gate (line 18). Then, we apply the Hadamard gate to both qubits (lines 20-21). We send the two qubits through the oracle (line 24) that we take as a parameter of this function (line 5).

We apply another Hadamard gate on the first qubit we receive as an output of the oracle (line 27). We measure it and store it in the `ClassicalRegister` (line 30). We use the `qasm_simulator` (line 33) because it supports multiple executions (in this case 1,000, line 36) of a quantum circuit that includes a measurement (see section 6.1).

Even though our algorithm around the oracle treats it as a black box, when we want to run the oracle for a specific function, we need a corresponding implementation.

Listing 8.2: Deutsch's algorithm

```python
1  from math import sqrt
2  from qiskit import QuantumCircuit, QuantumRegister,ClassicalRegister, Aer,
        execute
3  from qiskit.visualization import plot_histogram
4
5  def solve(oracle):
6    """
7    A reusable function that identifies whether the oracle represents
8    a constant or a balanced function.
9    """
10
11   qu = QuantumRegister(2)
12   cl = ClassicalRegister(1)
13
14   # initialize the circuit
15   qc = QuantumCircuit(qu,cl)
16
17   # Prepare the input state of the oracle
18   qc.x(1)
19
20   qc.h(0)
21   qc.h(1)
22
23   # Apply the Oracle
24   oracle(qc)
25
26   # Prepare the output state
27   qc.h(0)
28
29   # measure qubit-0
30   qc.measure(qu[0], cl[0])
31
32   # Tell Qiskit how to simulate our circuit
33   backend = Aer.get_backend('qasm_simulator')
34
35   # execute the qc
36   results = execute(qc,backend, shots = 1000).result().get_counts()
37
38   # plot the results
39   #return plot_histogram(results)
40   return plot_histogram(results, figsize=(3,2), color=['white'])
```

Let's implement the oracles. It is a different one for each possible function f_i.

We start with $i = 0$ and apply the function f_0 that always returns 0. As the truth table for the oracle O_0 from above shows, the output is the unchanged input

$$O_0(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus |f_0(x)\rangle = |x\rangle \otimes |y \oplus |0\rangle = |x\rangle \otimes |y\rangle$$

Thus, O_0 does nothing. Quite easy to implement.

Listing 8.3: Apply the gate o-0

```
1  def o_0(qc):
2      pass
3
4  solve(o_0)
```


Figure 8.14: Result of the oracle representing the constant function f0
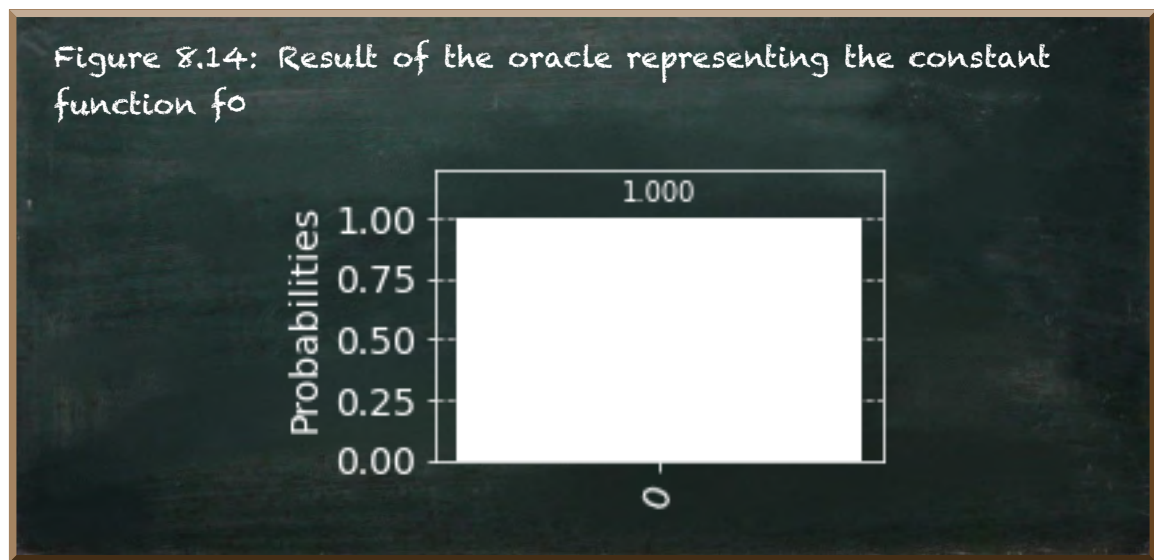
The result of the circuit with O_0 is always 0. The result of our calculations predicted for a constant function.

When $i = 1$, we apply the function f_1 that returns 0 for $x = 0$ and 1 for $x = 1$. Thus, $f_1(x) = x$.

$$O_1(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus f_1(x)\rangle = |x\rangle \otimes |y \oplus x\rangle$$

The gate O_1 returns an unchanged first qubit and $|y \oplus x\rangle$ as the second qubit. This is the behavior of the CNOT gate we got to know in section 6.1.

Listing 8.4: Apply the gate o-1

```python
def o_1(qc):
    qc.cx(0,1)

solve(o_1)
```


Figure 8.15: Result of the oracle representing the balanced function f1
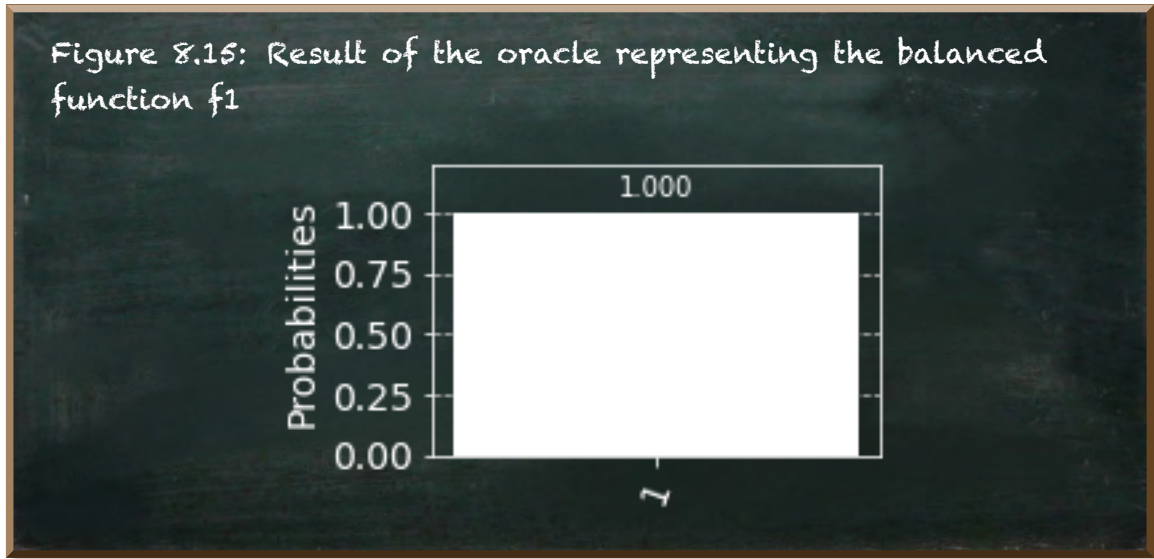
We measure 1 with certainty. This is the expected result of a balanced function.

The other balanced function f_2 returns 1 for $x = 0$ and 0 for $x = 1$. Thus, it behaves like the CNOT-gate but with $|y\rangle$ switching the value if $|x\rangle = 0$. We can construct this gate by first applying the X-gate to $|x\rangle$. If it was in state $|0\rangle$, it is now in state $|1\rangle$. The following CNOT-gate switches the state of $|y\rangle$. Finally, we apply the X-gate on $|x\rangle$ again to put it back into its original state.

Listing 8.5: Apply the gate o-2

```python
def o_2(qc):
    qc.x(0)
    qc.cx(0,1)
    qc.x(0)

solve(o_2)
```

Figure 8.16: Result of the oracle representing the balanced function f2
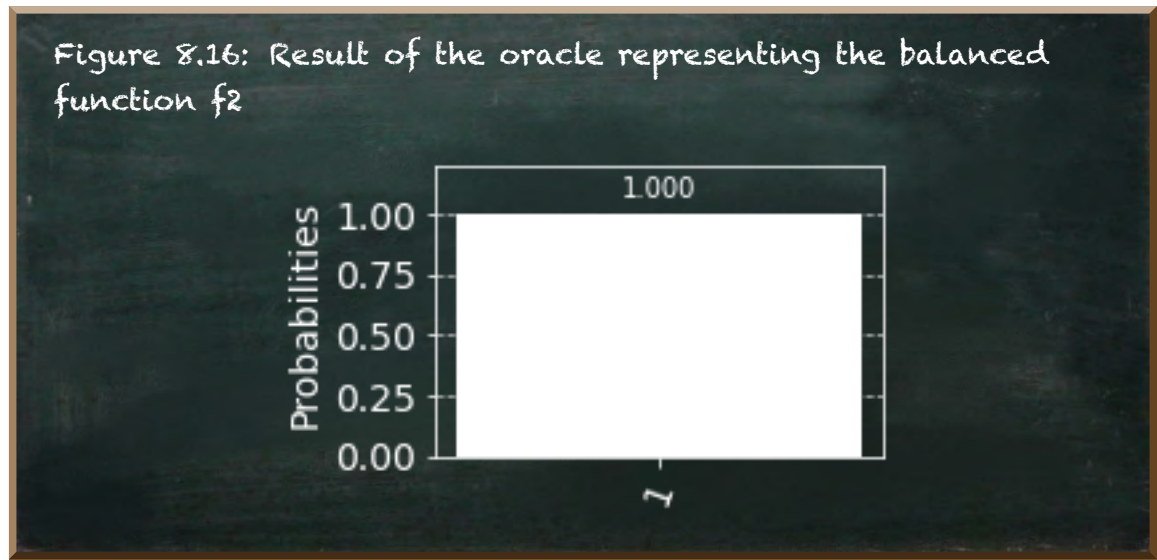
Again, we measure 1 with certainty. This result is correct for f_2 is a balanced function.

Finally, we implement the oracle representing the other constant function f_3. f_3 always returns 1.

$$O_3(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus |f_3(x)\rangle = |x\rangle \otimes |y \oplus 1\rangle)$$

Its output is like the input but with a reversed y. Thus, we apply the X-gate on the second qubit $|y\rangle$.

Listing 8.6: Apply the gate o-3

```python
def o_3(qc):
    qc.x(1)

solve(o_3)
```

Figure 8.17: Result of the oracle representing the balanced function f3
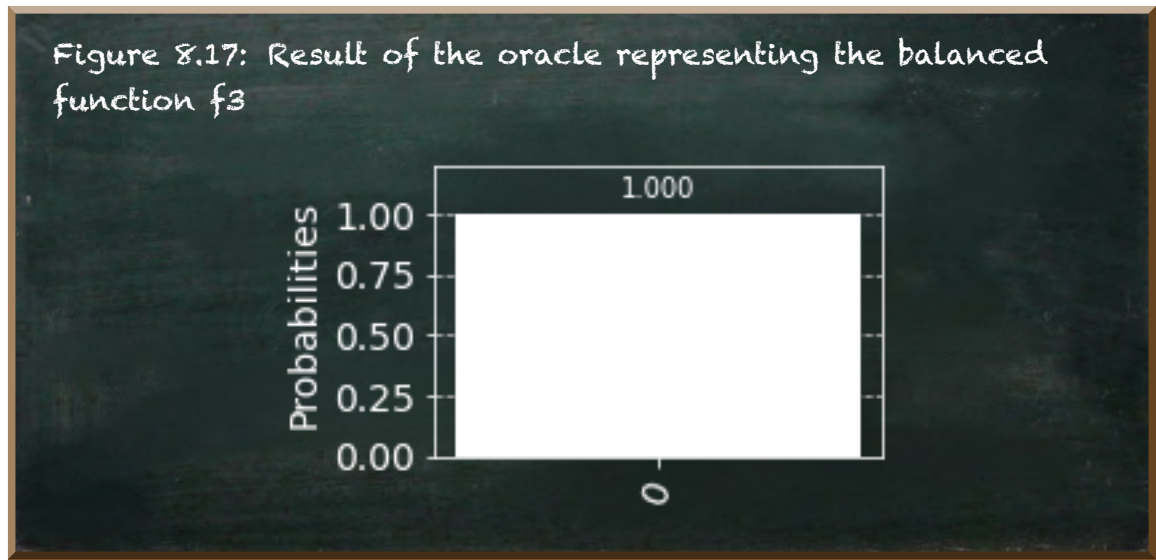
f_3 is a constant function, and we get the expected result of 0 accordingly.

8.3 The Quantum Oracle Demystified

When I first started learning quantum computing, it took me quite a while to understand how it could be faster than classical computing. Something mysterious must be going on.

Of course, the quantum superposition a qubit can be in is astonishing. The qubit is not 0 or 1. It is in a relationship between states 0 and 1.

Notwithstanding, the ability to entangle two qubits is mind-blowing. You measure one qubit, and another one instantly changes its state, no matter how far away it is. It is hard to believe that anyone would not think of teleportation as in Star Trek.

But once I started working with (simulated) qubits, the state of superposition came down as a probability. Entanglement emerged as a way to manipulate these probabilities. All this is pretty cool. But it isn't mysterious at all. I could not see how this kind of computing could be so much faster.

Then, I came across the quantum oracle. The name itself speaks of mystery. Finally, I must have found the magic ingredient I was searching for. Once I understood how it works, it I would be able to solve the problems intractable for a classical computer.
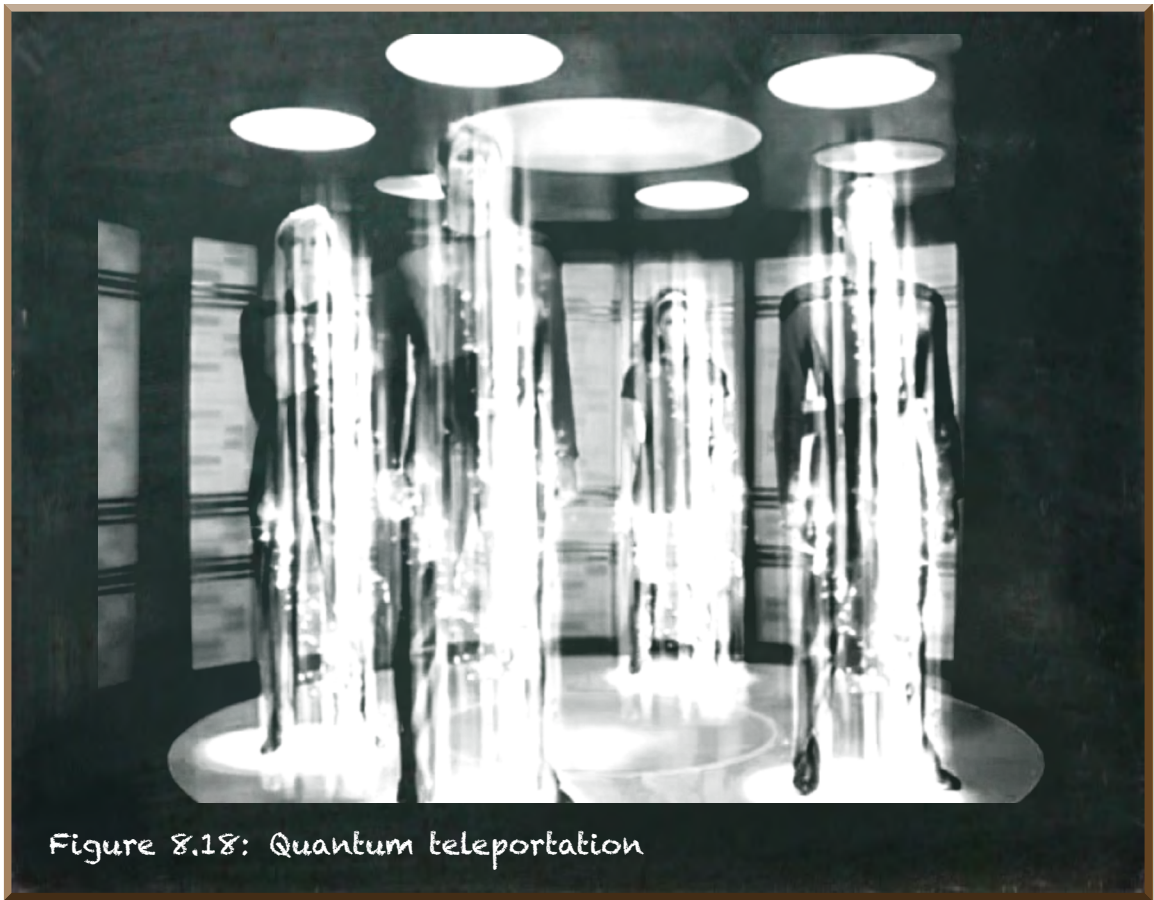
Figure 8.18: Quantum teleportation

In the previous section, we already got to know the quantum oracle. As part of David Deutsch's algorithm, it allowed us to solve a problem in fewer steps than a classical algorithm would need. It identifies the type of a function in a single shot. This is less than the two shots a classical algorithm needs.

Sounds magical. Doesn't it?

So, *how does* it *work*? You ask.

It's like a magic trick. It baffles you when you see it for the first time. You want to know how it works. But once someone tells you the secret, it becomes mundane. It loses its mystery. While you join the community of magicians, a selected group who knows the trick, you can't look at it anymore and think, "wow."

Figure 8.19: The secret of a magician

You still *want* to know? Good. Read on.

The magician presents a coin. While the coin is in the air, the magician predicts the coin to show heads. She catches the coin. Guess what you see. The coin is heads up.

How could she know? You'd bet the coin is manipulated. But she presents it to you. It looks like a regular coin. One side heads. The other side tails.

She tosses it again. She asks you to predict the result. This time, you say tails– the coin lands. Guess what you see? The coin is tails up.

You examine the coin again. It still looks like a normal coin. But it is not a normal coin. It is a quantum coin. When you look at the coin, it is either heads or tails, but once you toss it, it is in a state of superposition. It is unlike a regular coin.

A regular coin is sensitively dependent on the initial conditions. If you knew everything in complete detail, if you knew the applied force when tossing the coin, the rotational force, the air pressure, and even slight air movements, then you could calculate how a normal coin would land. If you knew everything in detail, randomness would disappear.

A quantum coin, by contrast, is truly random. So why then should the quantum coin be the one that allows the magician to predict the outcome?

While a quantum superposition contains randomness, it does not behave arbitrarily. The quantum system abides by strict rules. And these rules can be specified. One of these rules involves a quantum oracle.

Our magician created a quantum system that seems to let her know the outcome of tossing the coin. It sounds like an oracle, doesn't it?

But this is not what she did. Instead, she created a quantum system that would listen to her prediction and behave accordingly.

This doesn't seem plausible?

Then, why don't we create this quantum system programmatically with Python and Qiskit?

A qubit denotes our quantum coin. Once you toss it, it is in a state of superposition of the states $|0\rangle$ and $|1\rangle$. If you look at it, it is either 0 representing the coin lands heads up or 1 representing tails up. Each with a probability of 0.5.

Mathematically, the state of the qubit that we also know as $|+\rangle$ is

$$\psi = |+\rangle = \tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

We have seen this state before. We can create it by applying the Hadamard-gate on a qubit in the state $|0\rangle$.

The magician's prediction is a quantum transformation gate, too.

She crafted two gates and connected them with her prediction. She connected the I-gate with the prediction "heads up." And she connected the $R_Y(\pi)$-gate with the prediction "tails up."

The I-gate is the Identity-gate. Its output is equal to its input. It does not change anything.

$$\psi_{heads} = I(\psi) = \psi = |+\rangle = \tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

The R_Y-gate rotates the qubit state vector around the Y-axis. This is the axis that affects the measurement probabilities. It takes as a parameter the angle by which to rotate the state. The angle π denotes precisely half of a complete circuit.

The following image depicts the rotation of the qubit state vector graphically.

Figure 8.20: 360° 2-dimensional qubit system
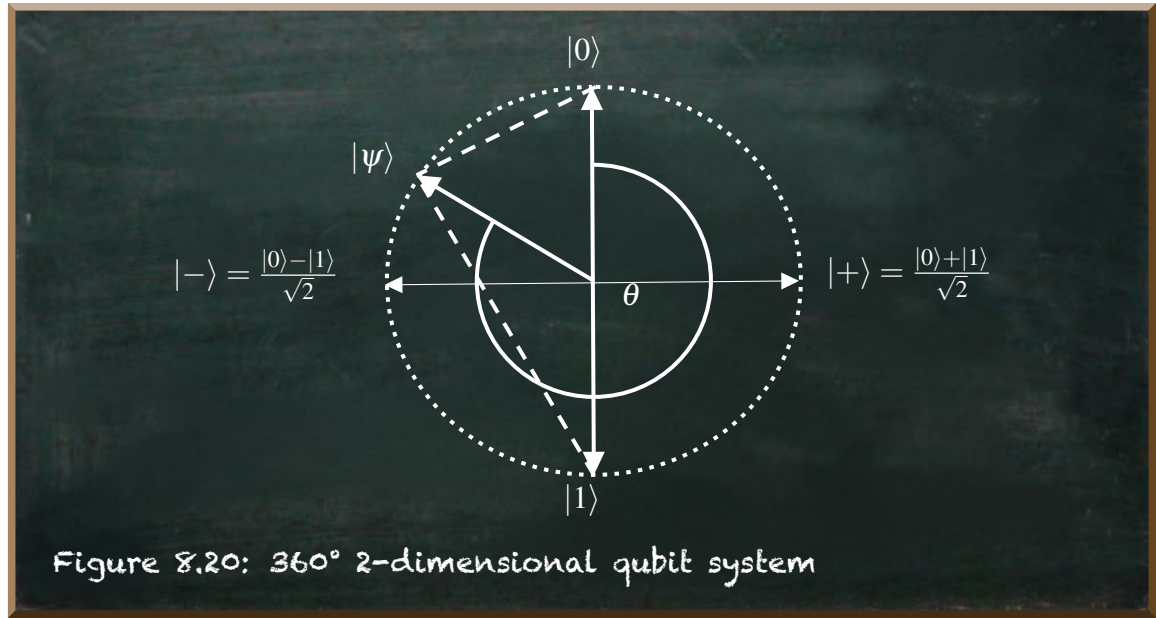
When a state vector ends on the left-hand side of the Y-axis, then one of the two amplitudes becomes negative.

When we start in the state $|+\rangle$ a rotation by π results in the state $|-\rangle$ because $cos\frac{\pi}{2} = 0$ and $sin\frac{\pi}{2} = 1$.

$$\psi_{tails} = R_Y(\pi)(\psi) = \begin{bmatrix} cos\frac{\pi}{2} & -sin\frac{\pi}{2} \\ sin\frac{\pi}{2} & cos\frac{\pi}{2} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = |-\rangle$$

Turning the state $|+\rangle$ into $|-\rangle$ did not change the probabilities of measuring 0 or 1 because the probability is the square of the amplitude. And this is positive. In our case, it is $\left(-\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}$.

In the end, neither one of the two gates changed the measurement probability of the qubit. But the two states differ.

When she prepares the magic trick, the magician does not limit her trick to either one prediction. She wants the flexibility to use a different prediction any time she performs the trick. So, during her preparation, she adds a placeholder into her quantum circuit. She calls it the O-gate. The oracle.

She only knows the oracle can be either the I-gate or the $R_Y(\pi)$-gate.

The I-gate represents her "heads up" prediction and leaves the qubit in the state $|+\rangle$. The $R_Y(\pi)$-gate represents her "tails up" prediction and changes the qubit state to $|-\rangle$.

The savvy magician sees the differences between these two states. This is her chance to make her prediction come true. All she needs to do is transform the oracle's output into the state that corresponds to her prediction. She needs to turn the state $|+\rangle$ into $|0\rangle$ and the state $|-\rangle$ into $|1\rangle$. She applies another Hadamard gate on her quantum coin, ehm qubit. It has the desired effect for both possible outputs of the oracle. Have a look:

$$H(|+\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1+1 \\ 1-1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

$$H(|-\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1-1 \\ 1+1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

Sometimes, it may be harder to transform the differences into a meaningful output. But the principle is the same. Identify the differences between the possible outcomes of the oracle and make sure these outcomes result in different measurement results.

Our magician is well prepared now. She has a quantum circuit with an oracle. This is a placeholder. When she runs the circuit, she must fill this placeholder with a valid qubit transformation gate by speaking out loud her prediction.

The magician created a reusable function. It takes as a parameter a callback function - the oracle (line 5). First, we create the `QuantumCircuit` with a single qubit (line 8). Tossing the coin sets it into superposition. This is what the first Hadamard gate does (line 11). Then, we apply the oracle (line 14). Whatever it is. The magician uses the second Hadamard gate to transform the result of the oracle into the desired state (line 17). Finally, we run the circuit (line 23) and return the results (line 26).

Here's the code the magician created.

Listing 8.7: The code the magician created

```python
1  from qiskit import QuantumCircuit, Aer, execute
2  from qiskit.visualization import plot_histogram
3  import matplotlib.pyplot as plt
4
5  def run_with_oracle(oracle):
6
7    # Create a quantum circuit with one qubit
8    qc = QuantumCircuit(1)
9
10   # toss the coin
11   qc.h(0)
12
13   # apply the oracle
14   oracle(qc)
15
16   # catch the coin
17   qc.h(0)
18
19   # Tell Qiskit how to simulate our circuit
20   backend = Aer.get_backend('statevector_simulator')
21
22   # Do the simulation, returning the result
23   result = execute(qc,backend).result()
24
25   # get the probability distribution
26   return result.get_counts()
```

Let's run the circuit with the heads-up prediction.

Listing 8.8: Run the heads up prediction

```python
1  plot_histogram(run_with_oracle(lambda qc: qc.i(0)))
```

Figure 8.21: Result of the heads up prediction
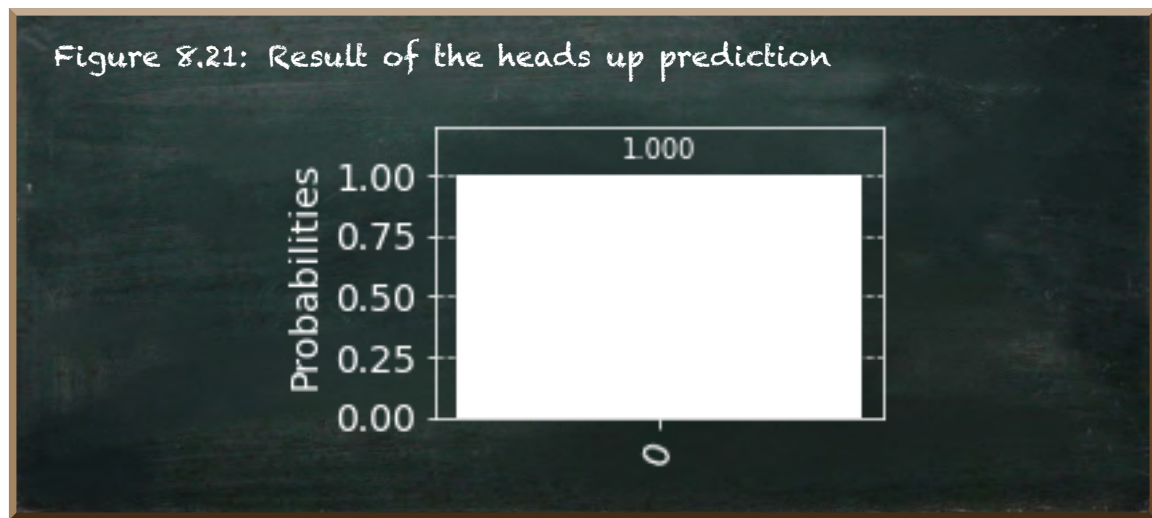
When the magician predicts heads up, we always measure the qubit as 0, representing heads up. So let's do the same for the "tails up" prediction.

Listing 8.9: Run the tails up prediction

```python
from math import pi
plot_histogram(run_with_oracle(lambda qc: qc.ry(pi, 0)))
```

Figure 8.22: Result of the tails up prediction
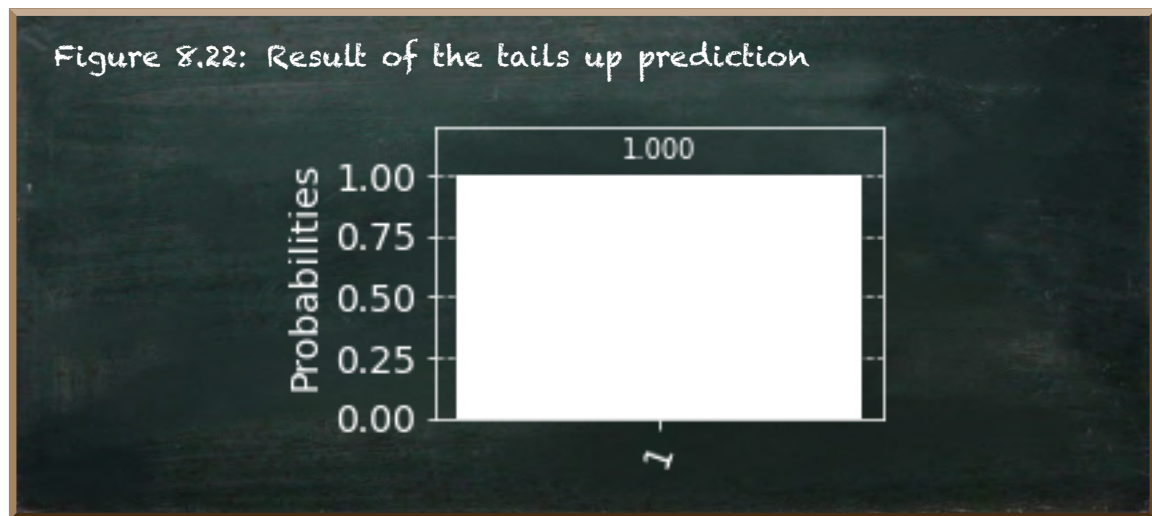
The "tails up" prediction always results in a qubit that we measure as 1 - the representation of "tails up."

The audience is baffled by her ability to see into the future. But, as usual, with

magic tricks, this is an illusion. An intended illusion is thoroughly crafted by arranging the stage–or in this case–the circuit.

The quantum oracle is nothing but a placeholder for a transformation gate. While it changes the system's state, it does not tell the future or answer any question. It is up to you to identify how the different gates may affect the quantum state differently. And, it is up to you to craft a quantum circuit around the oracle to transform these differences into an answer to your question.

A quantum oracle is a tool. It allows you to distinguish different states of the world. During the construction of your circuit, you specify how these different states should affect the outcome. When you run your circuit in a certain state, it produces the outcome you prepared it to result in.

The oracle is like the switch-case control structure you may know from classical programming. You have a variable, the oracle. When you write your program, the circuit, you don't know the specific value the variable will have during runtime. So, you specify the behavior for each possible value it may have. Once you run your program, the variable will have a specific value and your circuit will act the way you specified it to and result in the outcome you wanted it to result in.

The quantum oracle is not a magical ingredient. It is a control structure used in quantum computing.

9. Quantum Bayesian Networks

Bayes' Theorem helps us building a classifier capable of predicting the survival of a passenger on board the Titanic. However, the Quantum Naïve Bayes classifier we created in section 7 includes only two features. Yet, we are already moving on the edge of the possible. While handling modifiers below 1.0 that reduce the prior probability is easy, the modifiers above 1.0 are difficult to handle.

Altogether, our Quantum Naïve Bayes classifier has quite a few shortcomings.

1. Most of the work remains at the classical part. We need to consult the data for each passenger to get the backward probability and the marginal probability of the evidence. For each passenger, we calculate the corresponding modifiers.
2. We calculate the same modifiers over and over again. We do not reuse the results.
3. We construct a completely new quantum circuit for each unique combination of passenger attributes. For example, the circuit of a female passenger with a first-class ticket looks quite different from a male passenger's circuit with a third-class ticket. This is error-prone and hard to debug. We programmed quite a lot of logic in Python.

The first figure depicts the quantum circuit of a male passenger with a third-class ticket. The second figure depicts the circuit of a female passenger with a first-class ticket.

How we specify the prior probability is similar in both circuits. But, we apply

the modifiers in entirely different ways.

Figure 9.1: The circuit of a male passenger with a third-class ticket

Figure 9.2: The circuit of a female passenger with a first-class ticket

The modifiers used in a Naïve Bayes classifier are simple yet powerful tools. But they are not well-suited to be replicated in a quantum algorithm.

In this chapter, we address these shortcomings by using quantum oracles. We create a Quantum Bayesian Network.

9.1 Bayesian Networks

Bayesian networks are probabilistic models that model knowledge about an uncertain domain. Such as the survival of a passenger aboard the Titanic.

Bayesian networks build on the same intuitions as the Naïve Bayes classifier. But in contrast to Naïve Bayes, Bayesian networks are not restricted to represent solely independent features. They allow us to include as many interdependences that appear reasonable in the current setting.

A Bayesian network is represented as a directed acyclic graph with nodes and edges.

Figure 9.3: Example of a Bayesian Network
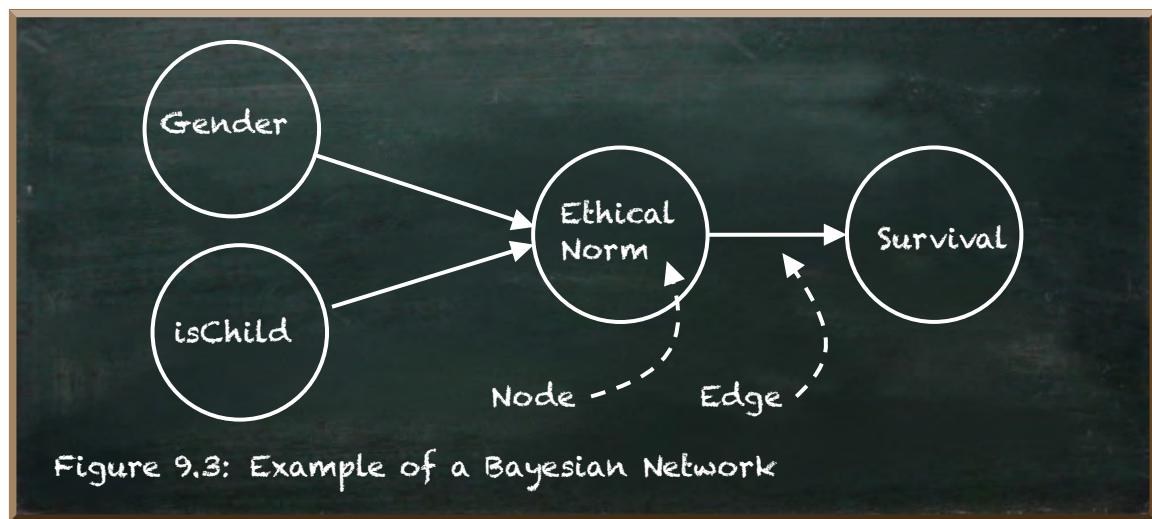
The nodes represent random variables, such as the gender of a passenger or whether s/he was a child.

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.

We quantify this dependence using conditional probability tables (CPT) for

discrete variables and conditional probability distributions (CPD) for continuous variables.

The following table depicts the posterior probabilities of survival given a passenger's gender (Sex).

	Female	Male
P(Survival=True, Sex)	0.74	0.19
P(Survival=False, Sex)	0.26	0.81

Table 9.1: Probabilities to survive given the gender

Female passengers had a much better chance to survive than male passengers.

While there are only two genders (in our dataset), there are many different ages of passengers. Technically, being still a discrete variable, it rather seems appropriate to model age as a continuous variable. The following graph depicts the CPD of the posterior probability of survival given a passenger's Age.

Figure 9.4: CPD of survival given a passenger's age
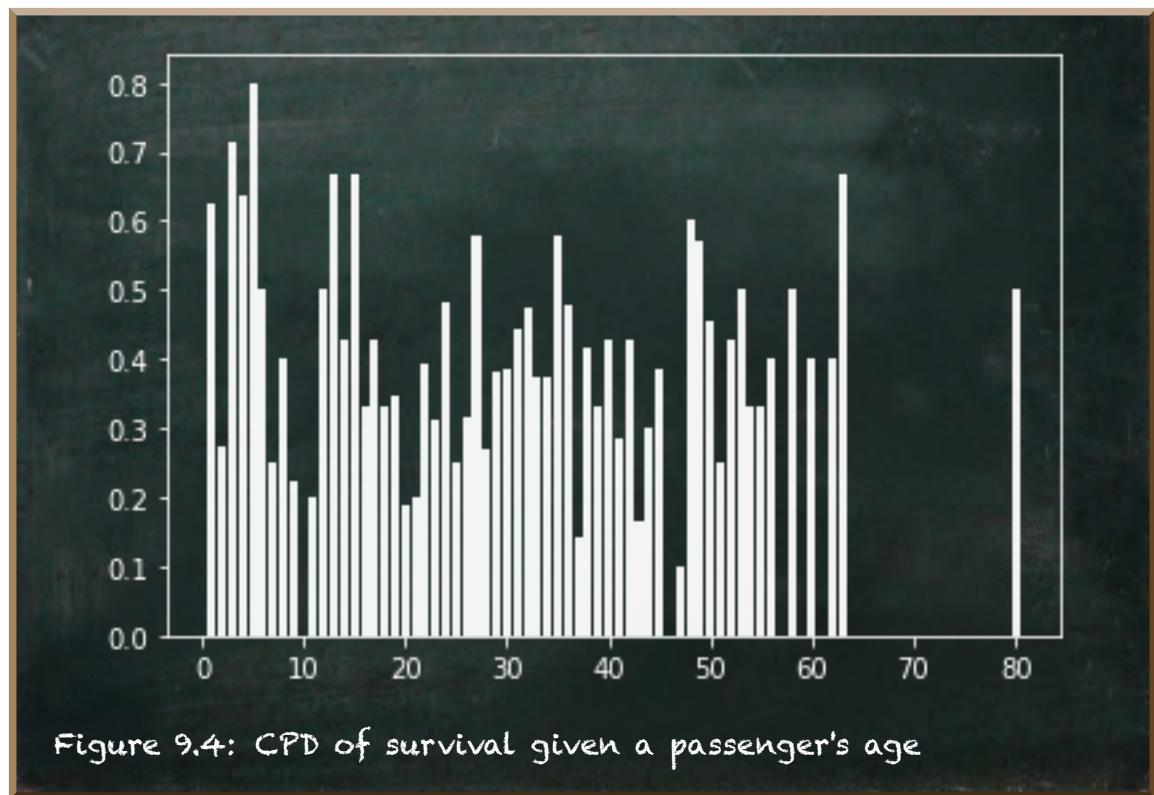
At first sight, it seems as if the age of a passenger does not have an apparent effect on the chances to survive. Even worse, the chance to survive varies a lot

between subsequent ages. For instance, a 47-year old passenger had a chance to survive of 0.1, whereas a 48-year old had a chance of 0.6.

Listing 9.1: The chances of survival

```
1  def chance_to_survive(age):
2    pop = train[train.Age.eq(age)]
3    surv = pop[pop.Survived.eq(1)]
4    prob = len(surv)/(len(pop)+1)
5    return "A {}-year old passenger had a chance to survive of {}".format(
       age, prob)
6
7  print(chance_to_survive(47))
8  print(chance_to_survive(48))
```

```
A 47-year old passenger had a chance to survive of 0.1
A 48-year old passenger had a chance to survive of 0.6
```

Such variations do not seem reasonable.

Instead, if we consider the characteristic of being a child (isChild) instead of the Age of a passenger. Children of the age of 8 or below had a significantly higher chance to survive than adults.
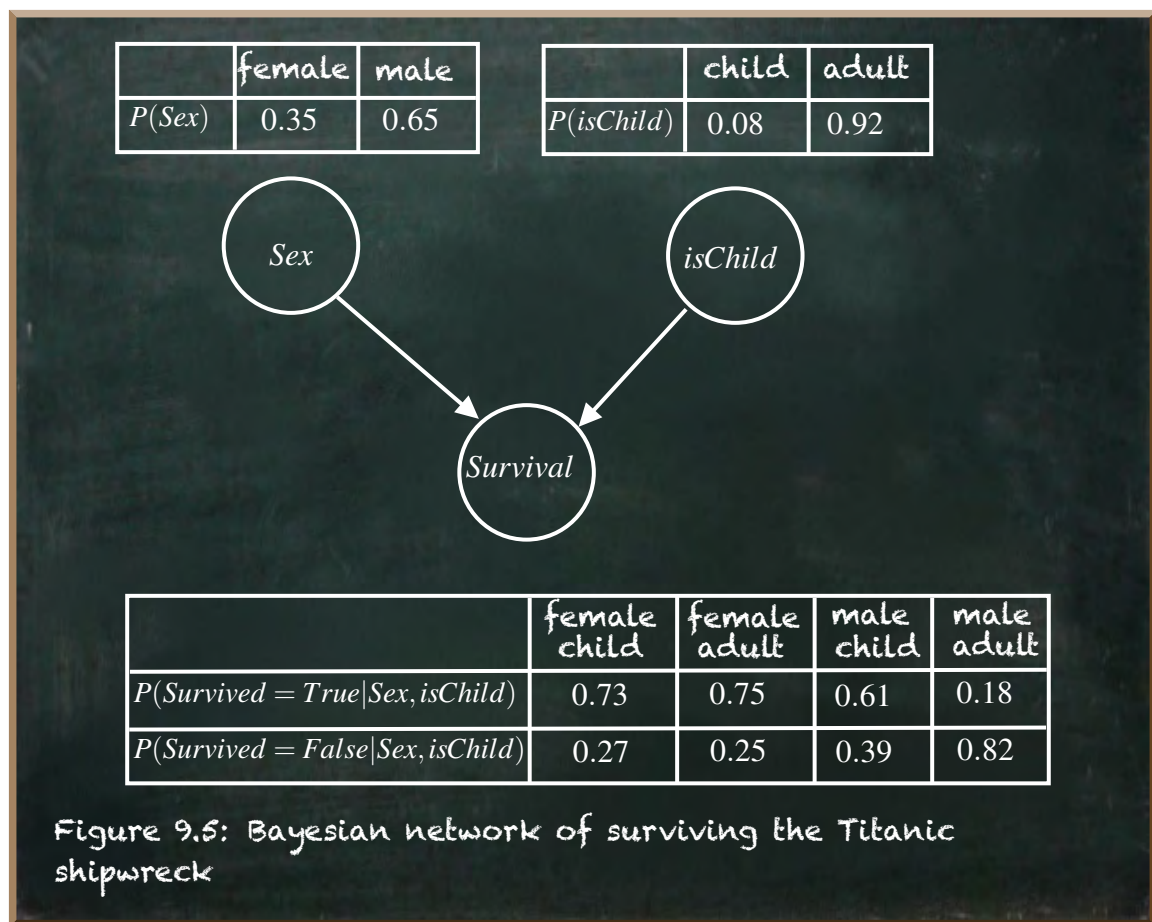
Listing 9.2: Survival of children

```
1  # max age of a child
2  max_child_age = 8
3
4  # probability to survive of children
5  pop_child = train[train.Age.le(max_child_age)]
6  surv_child = pop_child[pop_child.Survived.eq(1)]
7  p_child = len(surv_child)/len(pop_child)
8
9  # probability to survive of adults
10 pop_adult = train[train.Age.gt(max_child_age)]
11 surv_adult = pop_adult[pop_adult.Survived.eq(1)]
12 p_adult = len(surv_adult)/len(pop_adult)
13
14 print("{} children had a chance to survive of {}".format(len(pop_child),
       round(p_child, 2)))
15 print("{} adults had a chance to survive of {}".format(len(pop_adult),
       round(p_adult, 2)))
```

```
54 children had a chance to survive of 0.67
660 adults had a chance to survive of 0.38
```

Let's consider the Bayesian network with three nodes. The variables Sex and being a Child denote the parent nodes. These nodes don't have parents themselves. They are the root nodes. Their CPTs are conditioned on an empty set of variables. Thus, they are equal to the marginal (or prior) probabilities. Note, these are not the probabilities of surviving but the probabilities of the appearance of the respective characteristic.

Survival of the Titanic shipwreck is the child node. This CPT is conditioned on the values of the parent nodes as depicted in the following figure.

	female	male
P(Sex)	0.35	0.65

	child	adult
P(isChild)	0.08	0.92

Sex   isChild

Survival

	female child	female adult	male child	male adult
P(Survived = True|Sex, isChild)	0.73	0.75	0.61	0.18
P(Survived = False|Sex, isChild)	0.27	0.25	0.39	0.82

Figure 9.5: Bayesian network of surviving the Titanic shipwreck

Given such a set of CPTs, we can calculate the marginal probability of survival.

Due to the independence between nodes Sex and isChild (their values are independent because we have not modeled any dependence, but their effect on

Survival is not independent), the joint probability of a passenger having a certain Sex and isChild can be calculated as $P(Sex, isChild) = P(Sex) \cdot P(isChild)$.

Therefore, the conditional probability to survive given a certain Sex and isChild is $P(Survival) = P(Survival|Sex, isChild) \cdot P(Sex) \cdot P(isChild)$.

Foremost, a Bayesian network is a data structure. First, it represents the set of conditional independence assumptions. Any two nodes that are not connected through an edge are assumed independent of each other. Second, a Bayesian network contains probability tables and distributions in a compact and factorized way.

This data structure enables us to deduce the properties of a population. A Bayesian network supports forward and backward inference. For instance, we can calculate the overall chance to survive by integrating over the distribution of the child node (forward inference). And, given knowledge about the survival of a passenger, we can deduce how much certain characteristics contributed to his or her survival. For instance, if we look at the graphs of the child node, we can see the passenger's gender mattered a lot unless the passenger was a child. According to the norm of women and children first, they did not favor girls over boys a lot. This interdependency between Sex and isChild could not be included in a Naïve Bayes classifier.

This data structure, the Bayesian network graph, can be created in two different ways. Given sufficient knowledge of the dependencies, it can be designed a priori by the developer. Alternatively, it can be learned by the machine itself.

On our path to quantum machine learning, we will do both. We start with a small quantum Bayesian network (QBN) that we model ourselves. Then, we let the machine actually learn from the data.

9.2 Composing Quantum Computing Controls

The QBN we are about to build will consist of some advanced transformation gates. Let's have a brief look at how we can create such gates

Quantum transformation gates allow us to work with qubits. The R_Y-gate enables us to specify the qubit state vector angle θ that controls the probability of measuring the qubit as either 0 or 1. We used it to let a qubit represent the marginal probability of surviving the Titanic shipwreck.

Listing 9.3: Specify the marginal probability

```python
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
from math import asin, sqrt

def prob_to_angle(prob):
    """
    Converts a given P(psi) value into an equivalent theta value.
    """
    return 2*asin(sqrt(prob))

qc = QuantumCircuit(1)

# Set qubit to prior
qc.ry(prob_to_angle(0.4), 0)

# execute the qc
results = execute(qc,Aer.get_backend('statevector_simulator')).result().
    get_counts()
plot_histogram(results)
```


Figure 9.6: The marginal probability
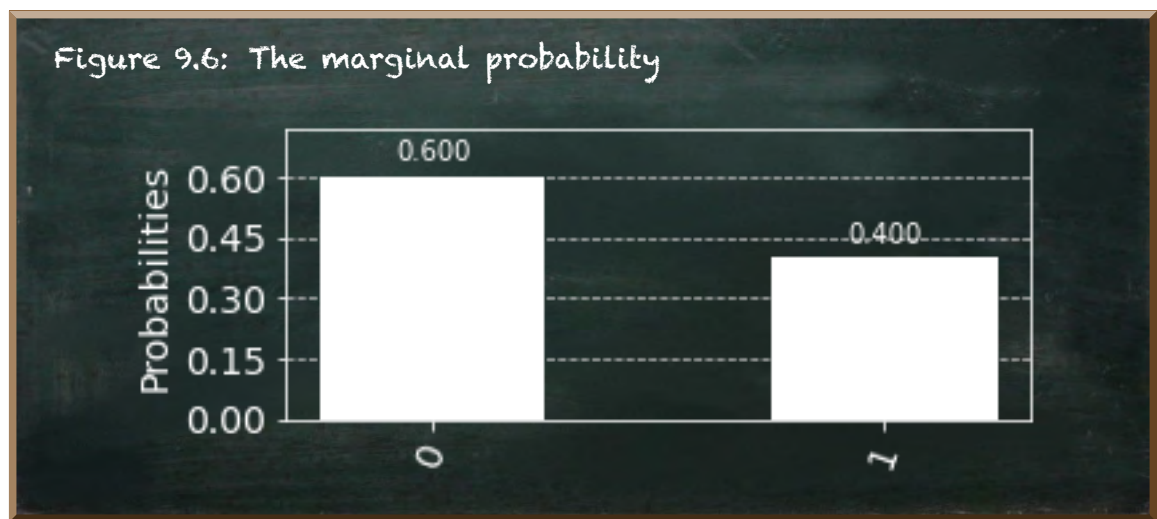
The *X*-gate (NOT-gate) switches the probability amplitudes of a qubit. We used it to set a qubit value to 1 in a specific state. For instance, to work with the remainder after we calculated the prior.

Listing 9.4: Use the X-gate to work with the remainder

```
 1  qc = QuantumCircuit(1)
 2
 3  # Set qubit to prior
 4  qc.ry(prob_to_angle(0.4), 0)
 5
 6  # Switch the qubit's value
 7  qc.x(0)
 8
 9  # execute the qc
10  results = execute(qc,Aer.get_backend('statevector_simulator')).result().
        get_counts()
11  plot_histogram(results)
```


Figure 9.7: The X-gate changes the qubit state
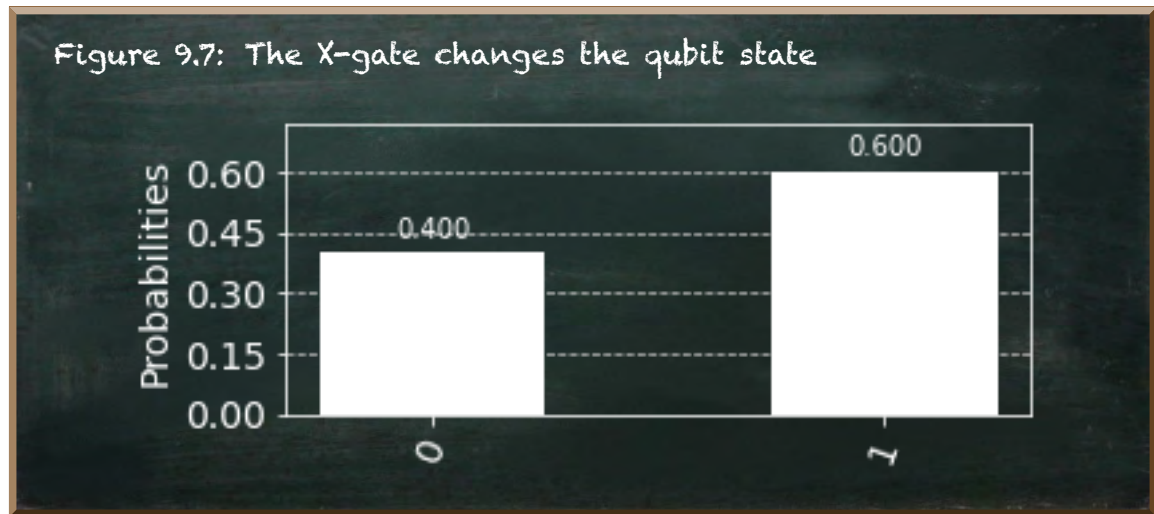
This is useful because some gates only apply a transformation on a qubit (the target qubit) when another qubit (the control qubit) is in the state $|1\rangle$.

For instance, the controlled R_Y-gate (CR_Y-gate) lets us specify a joint probability of the prior's remainder and another probability.

Listing 9.5: Calculate the joint probability

```
1  qc = QuantumCircuit(2)
2
3  # Set qubit to prior
4  qc.ry(prob_to_angle(0.4), 0)
5
6  # Switch the qubit's value
7  qc.x(0)
8
9  # Calculate the joint probability of NOT−prior and an event
10 qc.cry(prob_to_angle(0.8), 0,1)
11
12 # execute the qc
13 results = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_counts()
14 plot_histogram(results)
```


Figure 9.8: The joint probability
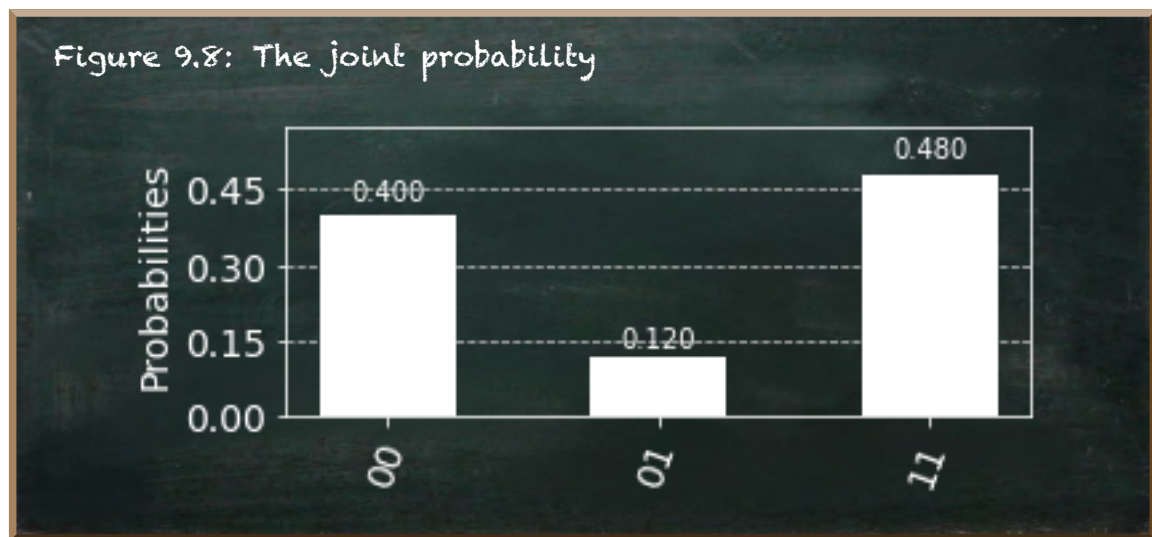
The CR_Y-gate is a composite gate. We learned how to create this gate from more basic gates, in section 6.3. Foremost, we used the $CNOT$-gate.

At first sight, the ability to apply an X-gate on a qubit if another qubit is $|1\rangle$ does not seem necessary. But the $CNOT$-gate takes a central role when creating higher-level qubits because it entangles two qubits. Conceptually, entangled qubits share a state of superposition. Practically, the $CNOT$-gate is the building block of most composite quantum transformation gates.

The following code depicts the decomposition of the CR_Y-gate.

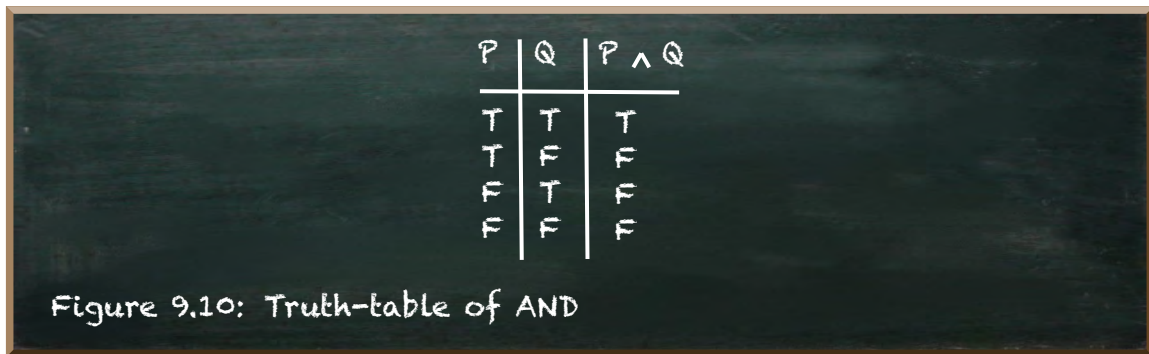Listing 9.6: Decomposition of the CRY-gate

```
1 qc = QuantumCircuit(2)
2
3 # Set qubit to prior
4 qc.ry(prob_to_angle(0.4), 0)
5
6 # Switch the qubit's value
7 qc.x(0)
8
9 # Apply half of the event's probability
10 qc.ry(prob_to_angle(0.8)/2, 1)
11
12 # entangle qubits 0 and 1
13 qc.cx(0,1)
14
15 # Apply the other half of ev_b
16 qc.ry(-prob_to_angle(0.8)/2, 1)
17
18 # unentganle qubits 0 and 1
19 qc.cx(0,1)
20
21 # execute the qc
22 results = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_counts()
23 plot_histogram(results)
```


Figure 9.9: Result of the decomposed CRY-gate
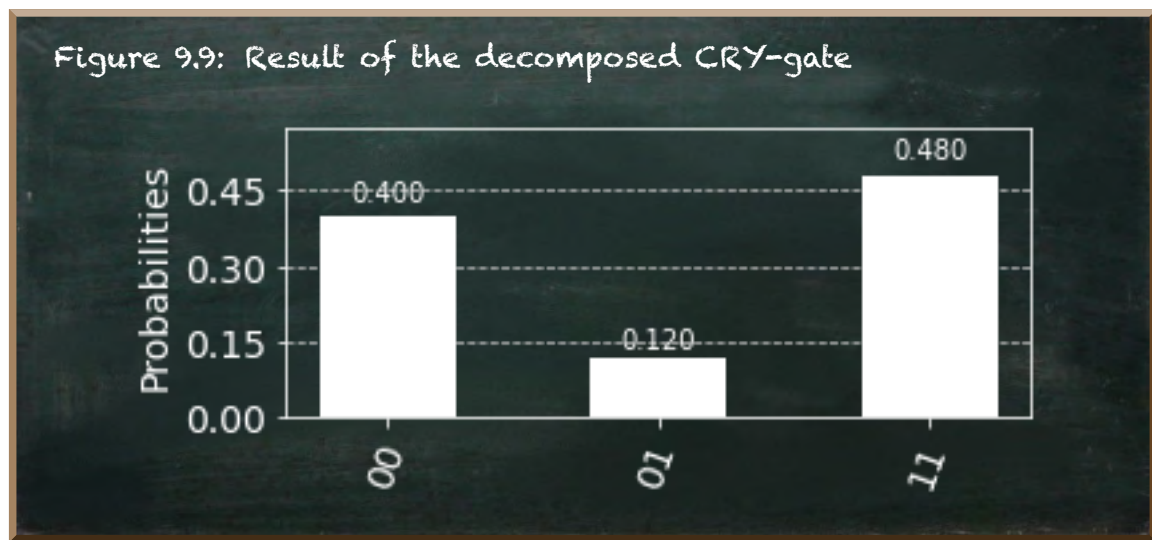
What if you wanted to apply a certain gate if and only if two other qubits are

in state $|1\rangle$? If you read this book carefully thus far, you may object AND is not a valid qubit gate. A brief look at the truth table discloses that the AND-operator is not reversible. If you get false as its output, you can't tell what the input was. It could be one of three different states.


Figure 9.10: Truth-table of AND

But the *CNOT*-gate provides a way out. Remember, when we constructed the *CR$_Y$*-gate, we used the *CNOT*-gate to switch the amplitudes of the controlled qubit in the middle of a rotation about the first half and a backward rotation about the second half of the overall rotation. A similar pattern allows us to create a controlled-controlled gate. Such a gate contains an AND-relationship because it has two control qubits and it only changes the target qubit if both control qubits are in state $|1\rangle$.

The following figure depicts the circuit of the *CCNOT*-gate - a controlled-controlled-NOT-gate.

> The *CCNOT*-gate is also known as the Toffoli-gate. The Toffoli-gate has a different algorithm than this one. The Toffoli-gate uses qubit phases. Phases are concept we cover later in this book. The implementation we presented here is not optimal, but it provides a vivid explanation of the underlying concept.

Figure 9.11: The CCNOT-gate
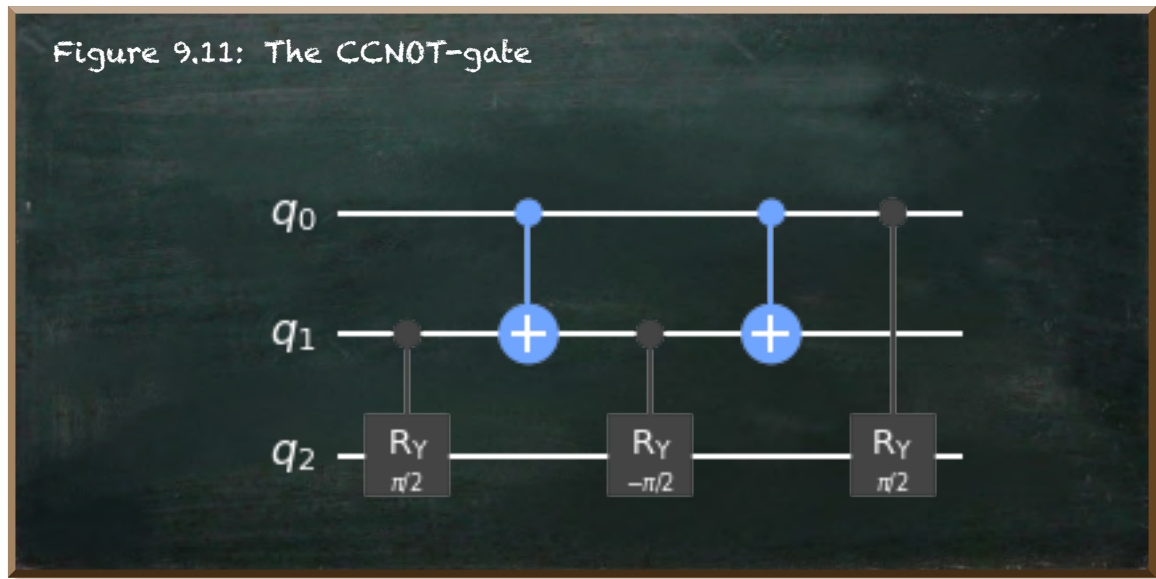
The following listing depicts the code of this *CCNOT*-gate sequence. We define a reusable function `ccnot` (line 4). It starts with the controlled rotation with qubit q_0 as control qubit (line 6). It rotates the controlled qubit about $\theta = \frac{\pi}{2}$, the value we defined earlier (line 2).

Then, we have another controlled rotation with the same qubit as the control qubit (line 11) encapsulated into *CNOT*-gates (lines 10 and 12). It is important to note that this encapsulated CR_Y-gate has $-\theta$ as a parameter. It denotes a rotation in the opposite direction.

Finally, we have another controlled rotation about θ. Here, qubit q_1 is the control qubit.

Let's go through the circuit one by one. First, we define our $\theta = \frac{\pi}{2}$ (line 2). The value $\frac{\pi}{2}$ represents rotation about a quarter of the circle. This is half of the overall rotation we want to apply. The rotation about half of the circle (π) switches the amplitudes from $|0\rangle$ to $|1\rangle$ and vice versa.

In the first step, we rotate the controlled qubit about a quarter circle if qubit q_1 is in state $|1\rangle$ through a $CR_Y(\frac{\pi}{2})$-gate (line).

Listing 9.7: The CCNOT-function

```
1  from math import pi
2  theta = pi/2
3
4  def ccnot(qc):
5      # Apply the first half of the rotatione
6      qc.cry(theta, 1,2)
7
8      # This sequence has no effect if both control qubits
9      # are in state |1>
10     qc.cx(0,1)
11     qc.cry(-theta,1,2)
12     qc.cx(0,1)
13
14     # Apply the second half of the rotation
15     qc.cry(theta, 0,2)
16
17     # execute the qc
18     return execute(qc,Aer.get_backend('statevector_simulator')).result().
         get_counts()
```

If both control qubits are in state $|1\rangle$, the result of this gate is as the following figure depicts.

Figure 9.12: State after half of the rotation
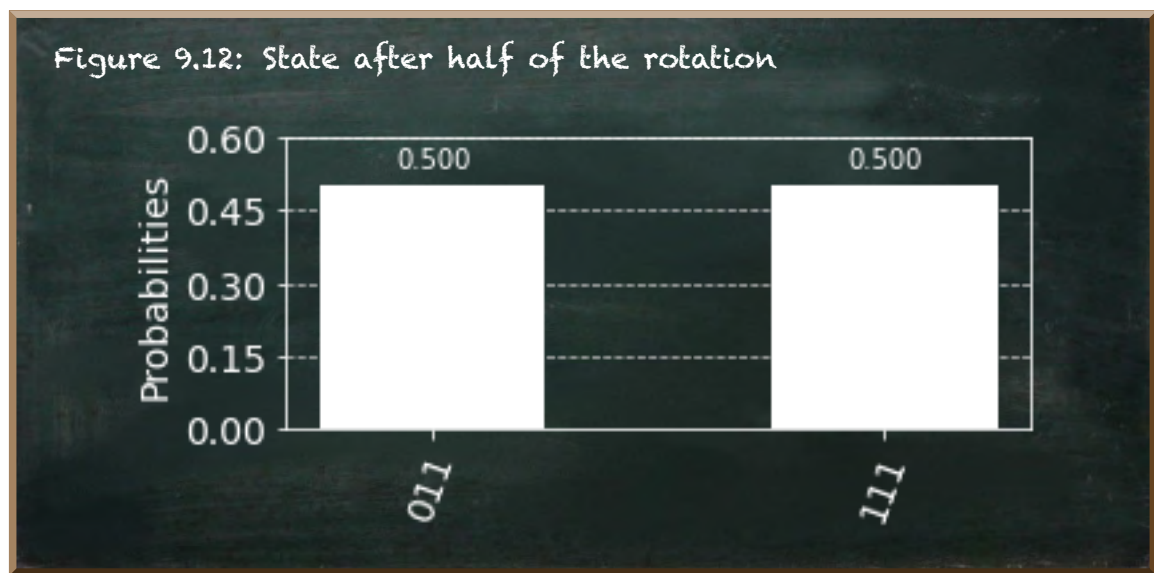
Both control qubits (the upper ones, read right to the left) are in state $|1\rangle$ per our initialization. Then, half the time (as per rotation about $\frac{\pi}{2}$), the con-

trolled qubit is in state $|1\rangle$.

Next, we apply a sequence of a *CNOT*-gate with q_0 as the control qubit and q_1 as the target qubit. For q_0 is in state $|1\rangle$, it changes the state of q_1 from $|1\rangle$ to $|0\rangle$. The following controlled rotation with q_1 as the control qubit has no effect because q_1 is now in state $|0\rangle$ and the *CR_Y*-gate only changes the controlled qubit if the control qubit is in state $|1\rangle$. The next *CNOT*-gate reverts the effect the first *CNOT*-gate had. For the control qubit q_0 is still in state $|1\rangle$, it switches the state of q_1 back from state $|0\rangle$ to $|1\rangle$.

If both control qubits are in state $|1\rangle$, these three gates have no effect at all.

Finally, we apply a controlled rotation about $\frac{\pi}{2}$ with q_0 as the control qubit. This turns the state of the controlled qubit q_2 from being in state $|1\rangle$ half the time to be in state $|1\rangle$ all the time. It rotates the qubit state vector about the other quarter of the circle, adding up to a half rotation. A half rotation about the circle turns the state $|0\rangle$ into $|1\rangle$ as the following figure shows.


Figure 9.13: Angles in a circle
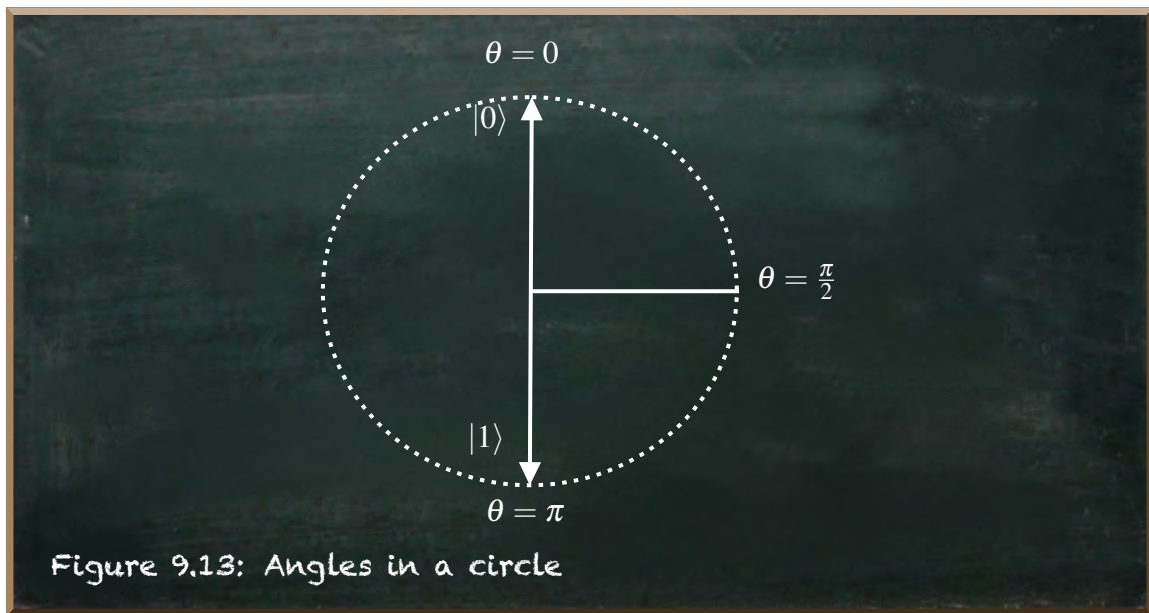
Let's look at the code and the result if both control qubits are in state $|1\rangle$.

Listing 9.8: The CCNOT-gate with both control qubits in state |1>

```
1  qc = QuantumCircuit(3)
2
3  # set both qubits to |1>
4  qc.x(0)
5  qc.x(1)
6
7  # apply the ccnot-gate and execute the qc
8  results = ccnot(qc)
9  plot_histogram(results)
```


Figure 9.14: Result of the CCNOT-gate with both control qubits in state |1>
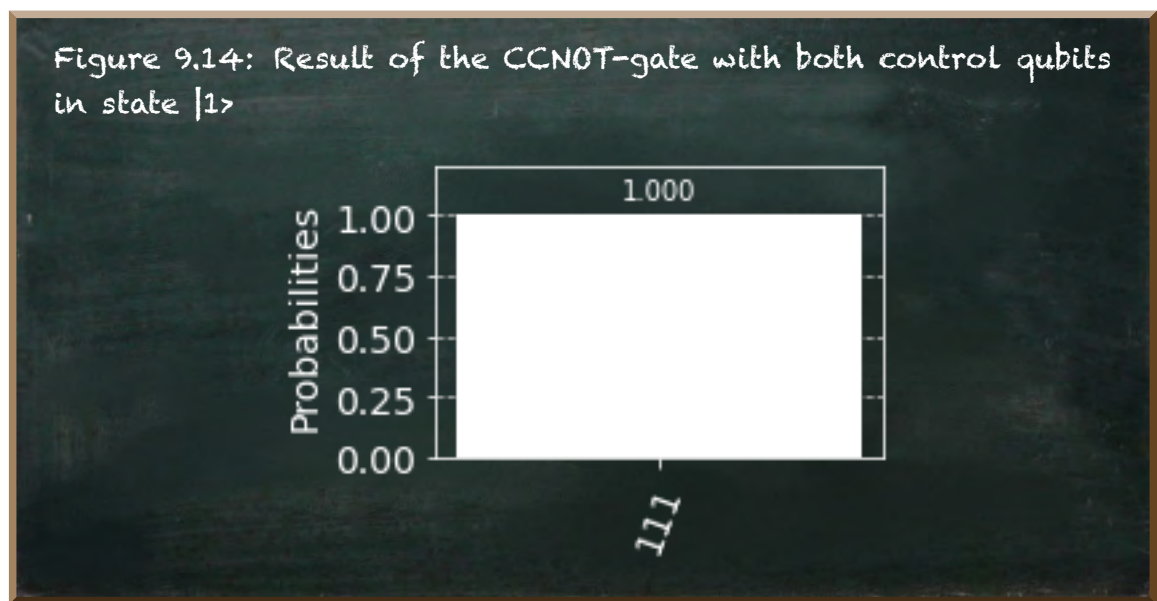
We see qubit q_2 is in state $|1\rangle$ all the time. It completely switched from its initial state $|0\rangle$.

What if one of the control qubits is not in state $|1\rangle$? Let's say qubit q_0 is in state $|0\rangle$.

Again, the first CR_Y-gate rotates the qubit state vector of the controlled qubit by $\frac{\pi}{2}$ - a quarter circle - because the control qubit q_1 is in state $|1\rangle$.

But this time, the following $CNOT$-gate has no effect. For qubit q_0 is in state $|0\rangle$, it does not switch the state of qubit q_1 from $|1\rangle$ to $|0\rangle$. As a result, the following CR_Y-gate with $\theta = -\frac{\pi}{2}$ takes effect. It reverts the effect the first CR_Y-gate had. The following $CNOT$-gate and the final CR_Y-gate have no effect because qubit q_0 is in state $|0\rangle$. Thus, we only applied the first two CR_Y-gates with the second reverting the first. Let's see the code and the result.

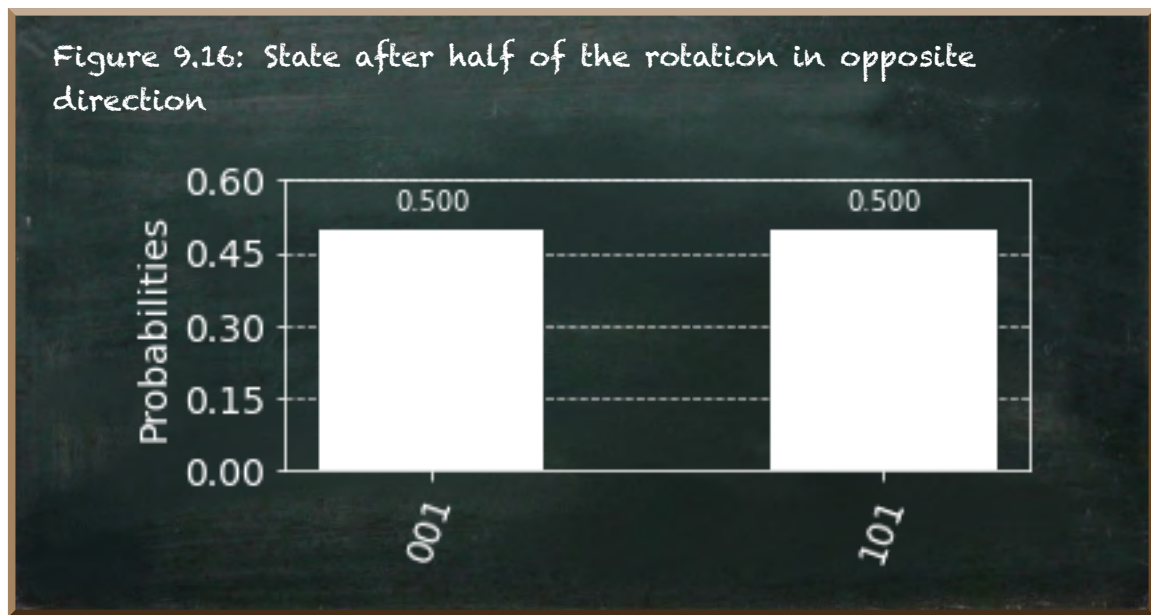Listing 9.9: The CCNOT-gate with only control qubit q1 in state |1>

```
1  qc = QuantumCircuit(3)
2
3  # set only qubit q_1 to |1>
4  qc.x(1)
5
6  # apply the ccnot-gate and execute the qc
7  results = ccnot(qc)
8  plot_histogram(results)
```


Figure 9.15: Result of the CCNOT-gate with only control qubit q1 in state |1>
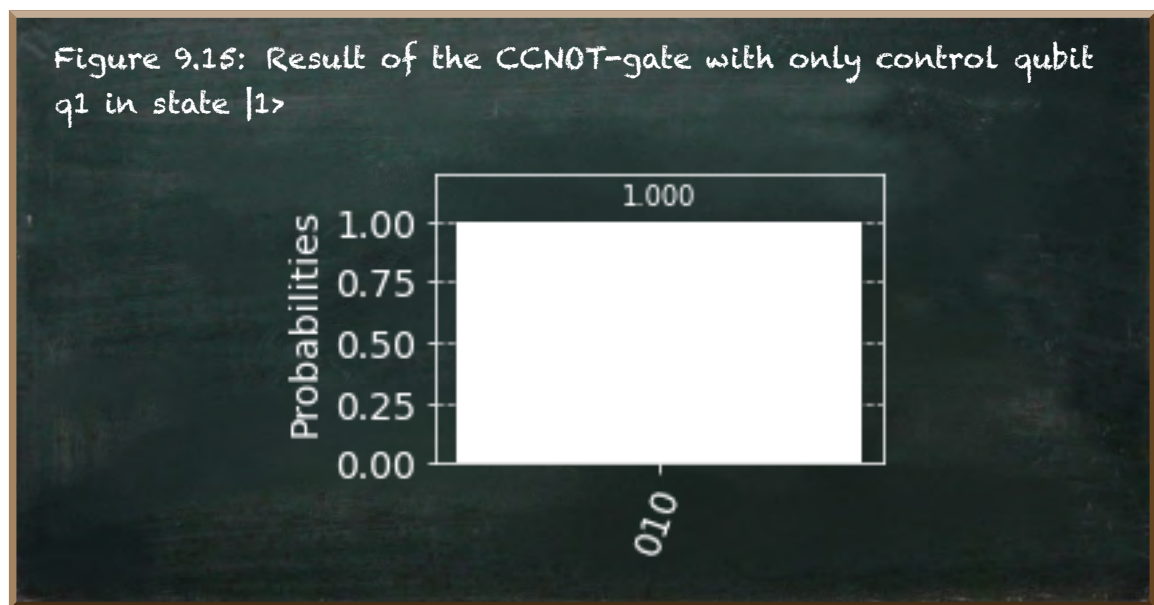
We see the overall state did not change. The target qubit remains in state $|0\rangle$.

Finally, let's see what happens if only control qubit q_0 is in state $|1\rangle$, but qubit q_1 is not. Then, the first CR_Y-gate is passed without effect. The following sequence of the second CR_Y-gate encapsulated into $CNOT$-gates first switches qubit q_1 from $|0\rangle$ to $|1\rangle$, then applies the rotation of the controlled qubit about $-\theta = -\frac{\pi}{2}$, and switches qubit q_1 back from $|1\rangle$ to $|0\rangle$. Now the controlled qubit has been rotated by half of a circuit in the opposite direction. The following figure depicts the result thus far.

Figure 9.16: State after half of the rotation in opposite direction

Half of the time, the controlled qubit is in state $|1\rangle$. Since the probabilities are the squared amplitudes that we change by a rotation, we do not see a negative value here.

Finally, the last CR_Y-gate rotates the controlled qubit back by θ because the control qubit q_0 is in state $|1\rangle$. The result is the original state again, as the following code and result show.

Listing 9.10: The CCNOT-gate with only control qubit q0 in state |1>

```
1  qc = QuantumCircuit(3)
2
3  # set only qubit q_0 to |1>
4  qc.x(0)
5
6  # apply the ccnot-gate and execute the qc
7  results = ccnot(qc)
8  plot_histogram(results)
```

Figure 9.17: Result of the CCNOT-gate with only control qubit q0 in state |1⟩
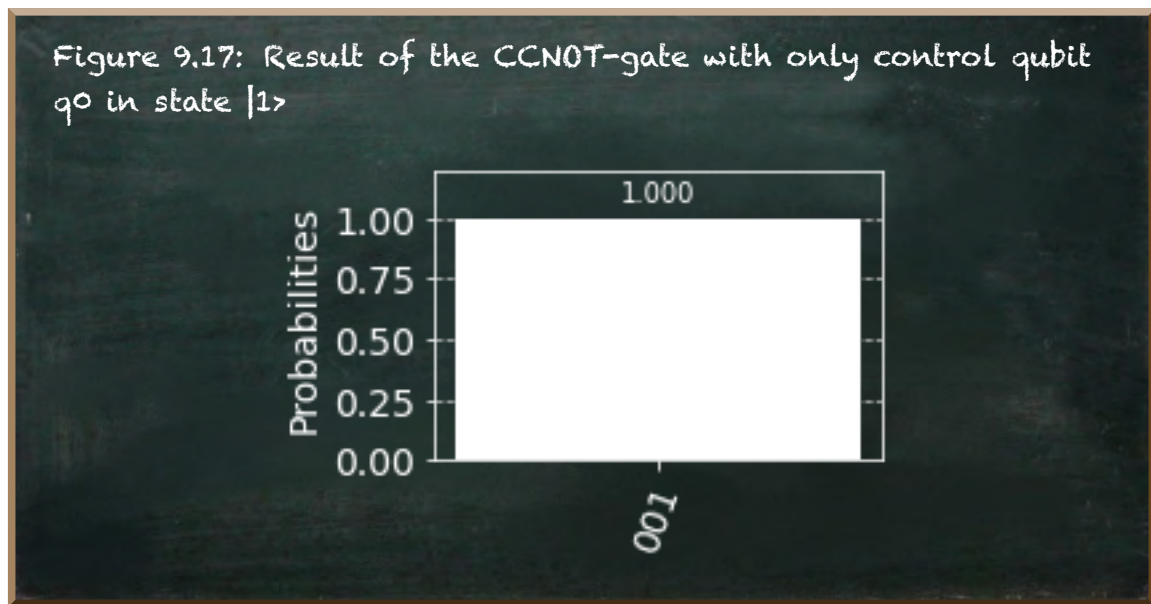
We created a controlled-controlled-NOT-gate through a composition of $CNOT$- and CR_Y-gates. We could even further compose the CR_Y-gates through $CNOT$- and R_Y-gates. This effectively shows the importance of the $CNOT$-gate. The $CNOT$-gate does not only serve as the best example to explain quantum entanglement, but it is also the building block of creating further controlled gates. Or controlled-controlled-gates. And even controlled-controlled-controlled-gates. You may continue this sequence until you run out of qubits.

The pattern we used here can be applied in general. Thus, let's have a more abstract look at what we just did.

In section 6.3, we constructed a controlled R_Y-gate using the CNOT-gate. Let's revisit this approach in general, too.

The $CNOT$-gate lets us easily turn any qubit transformation gate, let's call it U, into a controlled one.

Besides the $CNOT$-gate, there are two more things we need. The first thing is to split the gate U into halves. We must find a gate-let's call it V that, if applied twice, results in the desired overall transformation gate U. We can say $V \cdot V = U$ or $V = \sqrt{U}$. Thus, V is the square root of U.

The second thing is to create a gate that reverts the effect of gate V. Usually, this is the transpose of the gate's transformation matrix. The transpose (V^T) of a matrix is the original matrix V flipped over its diagonal (from top-left to bottom-right).

The following figure depicts how we control an arbitrary gate U.

Figure 9.18: Constructing a controlled gate
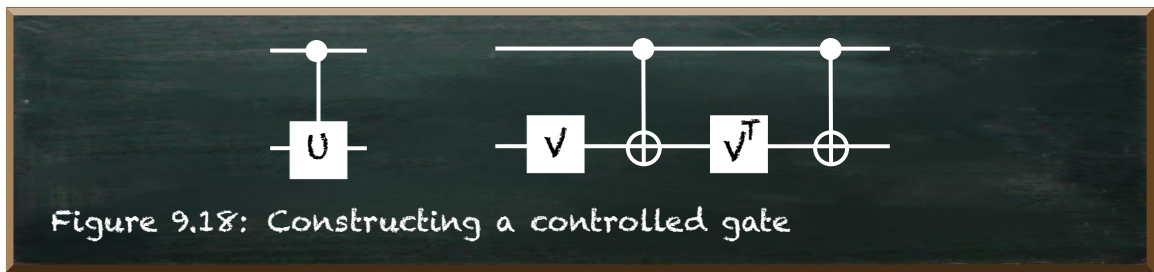
First, we apply the V-gate on the controlled qubit. Thus, we completed half of the U-gate. Then, we entangle the qubits. Thus, the controlled qubit flips its state. But it flips it only if the control qubit is in state $|1\rangle$.

When we now apply the transpose matrix V^T, it reverts the effect of V. But only if the control qubit is $|0\rangle$ because in this case, the $CNOT$-gate does not have any effect.

By contrast, if the control qubit is in state $|1\rangle$, the $CNOT$-gate flipped the state of the controlled qubit. It is in the exact opposite state. When we now apply the transposed V^T-gate, we apply the exact opposite of V once again because V^T is the opposite of V. Essentially, if the control qubit is in state $|1\rangle$ we applied NOT-$V \cdot V$-or NOT-U.

The final $CNOT$-gate turns the state NOT-U into U. But again, only if the control qubit is in state $|1\rangle$.

Now, let's have a look at the controlled-controlled gate. We want to create a quantum transformation gate we apply on a qubit q_c (the target qubit) only if two control qubits, q_0 and q_1 are in state $|1\rangle$.

We use the same tools.

- $CNOT$-gate
- V-gate is the square root of U
- V^T-gate that reverts the effect of V

This time, the gate V needs to be a controlled gate already. If it is not, you can use the $CNOT$-gate to turn any gate into a controlled gate. As we just saw.

The following image shows the construction of a controlled-controlled gate.

This controlled-controlled gate, let's call it U again, applies only if two control qubits, q_0 and q_1 are in state $|1\rangle$.

Again, we start with applying the gate V. For this is a controlled qubit now, we use q_1 as the control qubit and q_c as the target qubit. If q_1 is in state $|1\rangle$, we

Figure 9.19: The structure of a controlled-controlled gate
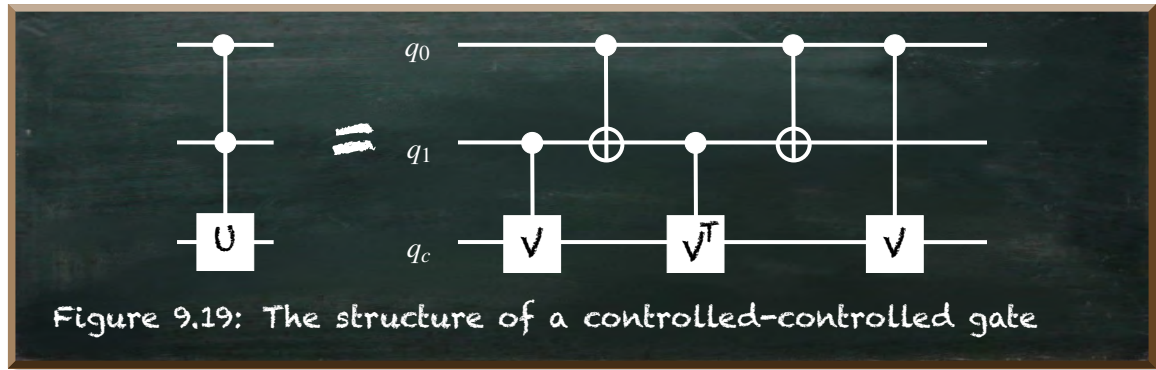
apply the first half of the overall transformation.

Accordingly, we end the circuit by applying gate V with q_0 as the control qubit and q_c as the controlled qubit. If q_0 is in state $|1\rangle$, too, we apply the second half of the overall transformation.

In between, we have a sequence of three gates: $CNOT$, V^T, and $CNOT$. The first $CNOT$-gate puts the then controlled qubit q_1 into the state $|0\rangle$ if both qubits q_0 and q_1 are in state $|1\rangle$ or in state $|0\rangle$. If one qubit is in state $|0\rangle$ and the other qubit is in state $|1\rangle$, it puts the controlled qubit q_1 into the state $|1\rangle$.

The following figure shows the truth-table of applying the $CNOT$-gate with q_1 as the controlled qubit.


Figure 9.20: Truth-table of CNOT

As a result, we apply the following gate V^T only if one qubit is in state $|0\rangle$ and the other is in state $|1\rangle$. In these cases, we applied one of the two V-gates, either the one at the start or the end. Thus, there is no effect on the controlled qubit q_c in total. We applied V on it and V_T to revert it.

If both qubits q_0 and q_1 are in state $|0\rangle$, we do nothing at all leaving the controlled qubit as it was, too. Only if both qubits q_0 and q_1 are in state $|1\rangle$, then we apply both V-gates while not applying the gate V_T.

The $CNOT$-gate after the V^T-gate reverts the effect of the first $CNOT$-gate so

that we leave qubit q_1 unchanged as well.

Finally, if V is not a controlled gate, we can make it one by inserting the first circuit into the second. The following figure depicts the resulting circuit.

Figure 9.21: Complete construction of a controlled-controlled gate
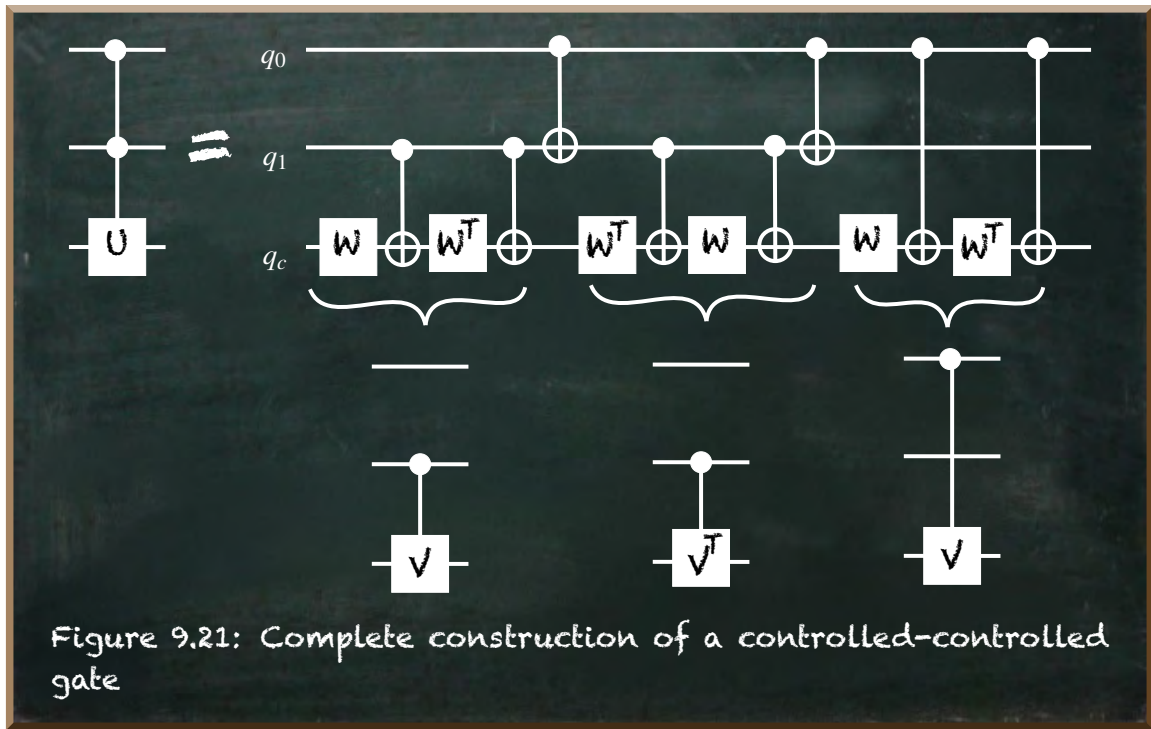
In this case, we need to split the V-gate into halves, too. Thus, W is the square root of V.

In summary, by splitting the overall effect into halves (V), we can implement a gate (U) that depends on two other qubits to be in state $|1\rangle$. If neither one control qubit is $|1\rangle$, nothing happens at all. If only one control qubit is in state $|1\rangle$, we miss one application of V, and we apply the V^T-gate that cancels the effect of the one V-gate we applied. If both control qubits are in state $|1\rangle$, we apply both V-gates but miss the V^T gate. That is the controlled-controlled U-gate.

9.3 Circuit implementation

We start with the implementation of our example thus far, the effect the `Sex` of a passenger and being a child (`isChild`) had on the `Survival` of the Titanic shipwreck.

A qubit represents each node in the Bayesian network. Since all our nodes are binary (`Sex`, `isChild`, `Survival`), a single qubit each is sufficient. If we had more

discrete states or a continuous distribution, we would need more qubits. The qubit states represent the marginal (for root nodes) and the conditional (for `Survival` node) probability amplitudes of the corresponding variables.

The state $|0\rangle$ represents a male passenger or an adult. The state $|1\rangle$ a female or a child. The superposition of the qubit denotes the probability of either state.

- $\psi_{Sex} = \sqrt{P(male)}|0\rangle + \sqrt{P(female)}|1\rangle$
- $\psi_{isChild} = \sqrt{P(adult)}|0\rangle + \sqrt{P(child)}|1\rangle$

We initialize these two qubits through rotations around the Y-axis.

Listing 9.11: Initialize the parent nodes

```
1  # the maximum age of a passenger we consider as a child
2  max_child_age = 8
3
4  # probability of being a child
5  population_child = train[train.Age.le(max_child_age)]
6  p_child = len(population_child)/len(train)
7
8  # probability of being female
9  population_female = train[train.Sex.eq("female")]
10 p_female = len(population_female)/len(train)
11
12 # Initialize the quantum circuit
13 qc = QuantumCircuit(3)
14
15 # Set qubit0 to p_child
16 qc.ry(prob_to_angle(p_child), 0)
17
18 # Set qubit1 to p_female
19 qc.ry(prob_to_angle(p_female), 1)
```

We calculate the probabilities of being a child (line 6) and being female (line 10). We use R_Y-gates to let the qubits q_0 (line 16) and q_1 (line 19) represent these marginal probabilities.

Next, we add the CPT of `Survival` to the circuit. This is a little more work.

There are four different combinations of parent node values, `Sex`, and `isChild`. These are, a male adult ($|00\rangle$), a male child ($|01\rangle$), a female adult ($|10\rangle$), and a female child ($|11\rangle$). Therefore, we have four rotation angles, one for each parent node combination.

For each of these combinations, we use a controlled-controlled R_Y-gate (CCR_Y) to specify the probability of Survival. If there were n parent nodes, then we would implement a $C^n R_Y$-gate.

As the following figure shows, we encapsulate each rotation into X-gates. For a CCR_Y-gate only applies the rotation on the controlled qubit if both control qubits are in state $|1\rangle$, the leading X-gates select the corresponding combination, and the trailing X-gates revert the selection.

For example, to apply the conditional probability of a male adult (state $|00\rangle$), we need to flip both qubits. This is what we do. After applying the CCR_Y-gate with the respective angle, we flip the qubits back into the original state.

Figure 9.22: The quantum bayesian network circuit
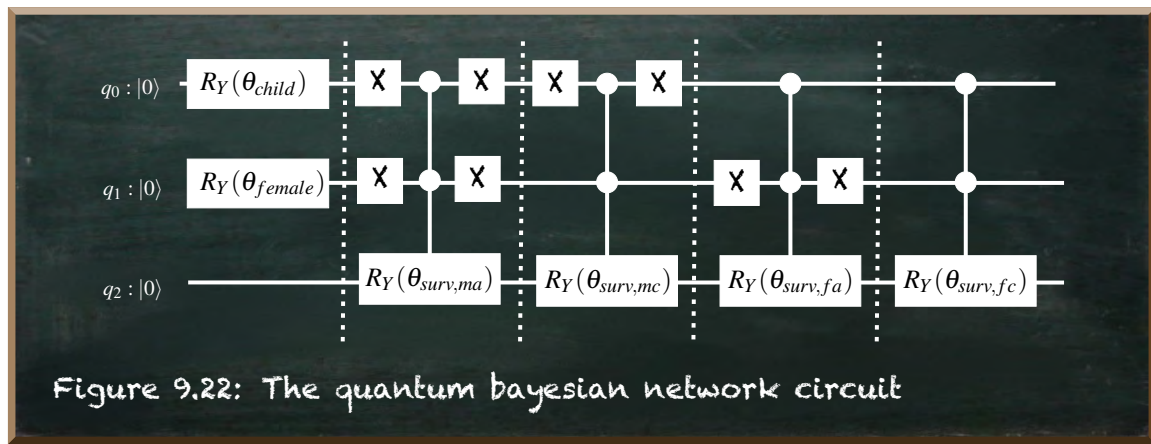
We learned how to create a CCR_Y gate in section 9.2. The function ccry (lines 1-6) adds such a gate to our circuit.

Listing 9.12: Definition of the CCRY-gate

```
1  def ccry(qc, theta, control1, control2, controlled):
2    qc.cry(theta/2, control2, controlled)
3    qc.cx(control1, control2)
4    qc.cry(-theta/2, control2, controlled)
5    qc.cx(control1, control2)
6    qc.cry(theta/2, control1, controlled)
```

In the following code, we calculate the conditional probability of each of our four cases. We separate the population, for example, female children (line 3), separate the survivors among them (line 4), and calculate their probability to survive by dividing the number of survivors by the total number of female children among the passengers (line 5).

We do the same for female adults (lines 8-10), male children (lines 13-16), and male adults (lines 19-21).

Listing 9.13: Calculate the conditional probabilities

```python
 1  # female children
 2  population_female=train[train.Sex.eq("female")]
 3  population_f_c=population_female[population_female.Age.le(max_child_age)]
 4  surv_f_c=population_f_c[population_f_c.Survived.eq(1)]
 5  p_surv_f_c=len(surv_f_c)/len(population_f_c)
 6
 7  # female adults
 8  population_f_a=population_female[population_female.Age.gt(max_child_age)]
 9  surv_f_a=population_f_a[population_f_a.Survived.eq(1)]
10  p_surv_f_a=len(surv_f_a)/len(population_f_a)
11
12  # male children
13  population_male=train[train.Sex.eq("male")]
14  population_m_c=population_male[population_male.Age.le(max_child_age)]
15  surv_m_c=population_m_c[population_m_c.Survived.eq(1)]
16  p_surv_m_c=len(surv_m_c)/len(population_m_c)
17
18  # male adults
19  population_m_a=population_male[population_male.Age.gt(max_child_age)]
20  surv_m_a=population_m_a[population_m_a.Survived.eq(1)]
21  p_surv_m_a=len(surv_m_a)/len(population_m_a)
```

Next, we select the states representing these groups of passengers and apply the CCR_Y-gate with the corresponding probability.

Listing 9.14: Initialize the child node

```
1  # set state |00> to conditional probability of male adults
2  qc.x(0)
3  qc.x(1)
4  ccry(qc,prob_to_angle(p_surv_m_a),0,1,2)
5  qc.x(0)
6  qc.x(1)
7
8  # set state |01> to conditional probability of male children
9  qc.x(0)
10 ccry(qc,prob_to_angle(p_surv_m_c),0,1,2)
11 qc.x(0)
12
13 # set state |10> to conditional probability of female adults
14 qc.x(1)
15 ccry(qc,prob_to_angle(p_surv_f_a),0,1,2)
16 qc.x(1)
17
18 # set state |11> to conditional probability of female children
19 ccry(qc,prob_to_angle(p_surv_f_c),0,1,2)
```

We're now ready to run the circuit. Let's have a look.

Listing 9.15: Execute the circuit

```
1  # execute the qc
2  results = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_counts()
3  plot_histogram(results)
```

Figure 9.23: Results of the qbn
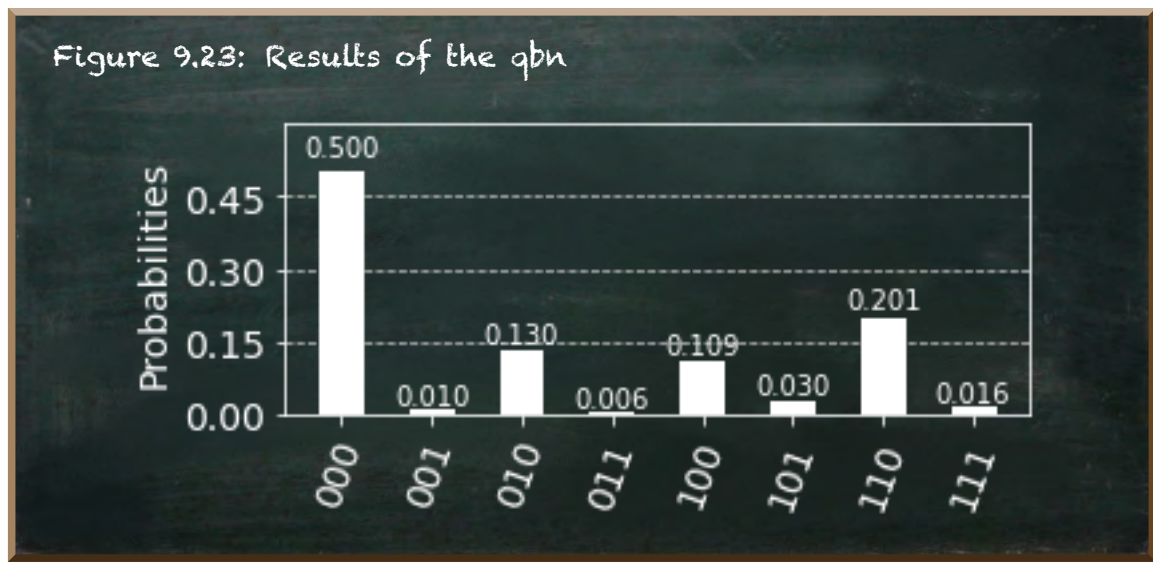
We can see eight different states. These belong to the victims (qubit $q_2 = 0$) and the survivors ($q_2 = 1$) of the four groups. Thus, the overall marginal probability of surviving is the sum of all states where qubit $q_2 = 1$.

To not add them manually, let's include a measurement into our circuit. We include a `ClassicalRegister` into our circuit.

Listing 9.16: A quantum circuit with classical register

```
1 # Quantum circuit with classical register
2 qr = QuantumRegister(3)
3 cr = ClassicalRegister(1)
4 qc = QuantumCircuit(qr, cr)
```

Then, we need to apply all the gates (skipped for brevity). We add a measurement (line 3). Here, we're interested in qubit q_2.

Finally, we select the appropriate backend (`qasm_simulator`) and run the circuit several times (here 1,000 shots) (line 4).

Listing 9.17: Run the circuit including a measurement

```
1 # —— INCLUDE ALL GATES HERE ——
2
3 qc.measure(qr[2], cr[0])
4 results = execute(qc,Aer.get_backend('qasm_simulator'), shots=1000).
      result().get_counts()
5 plot_histogram(results)
```

Figure 9.24: QBN including a measurement
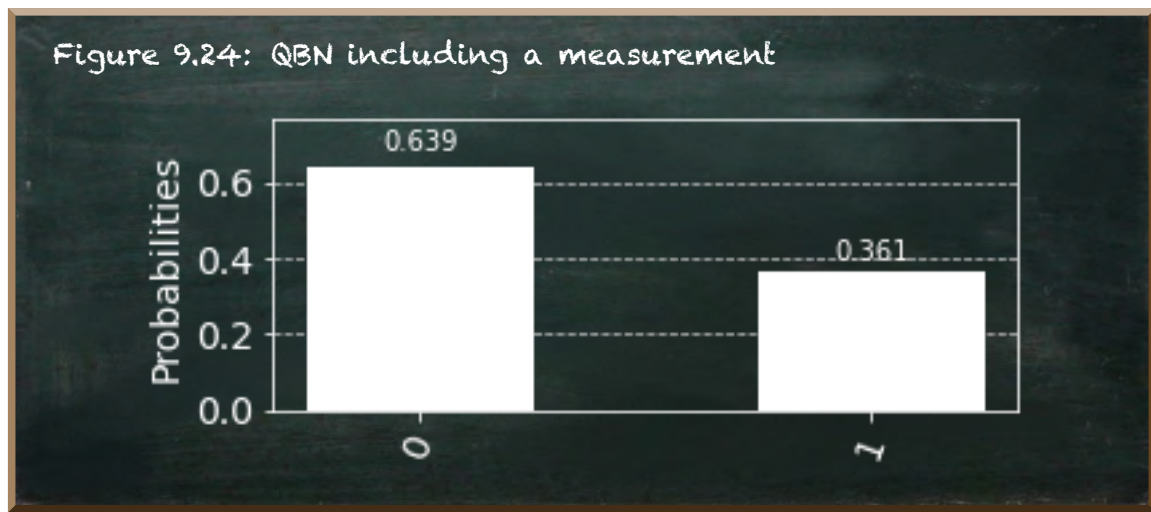
The result shows that we're close to the actual probability of surviving of 0.38. The actual result varies a little since we do not calculate but empirically simulate this result.

Implementing a quantum Bayesian network is straightforward for a set of binary state variables because we can represent each variable by a single qubit. Even if we had variables with more than two states, the structure would not change. We still would activate each state by X-gates and apply the corresponding controlled rotation. But, we would have to cope with more states.

10. Bayesian Inference

We have implemented our quantum Bayesian network. It represents a passenger's overall chance to survive the Titanic shipwreck. It considers two features, the passenger's gender and whether the passenger was a child.

It's time to use this network. We want to infer something we don't already know. We perform inference.

Generally, (statistical) **inference** is the process of deducing properties about a population or probability distribution from data. This is the reason why we build the entire network. We want to be able to make predictions about some new data from the data we already know.

Specifically, **Bayesian inference** is the process of deducing properties about a population or probability distribution from data *using Bayes' theorem*.

There are various questions we can answer with inference. We already performed one type of inference. That is marginal inference. We calculated the overall probability of survival. Given our network with three variables, we tried to find the probability of one variable, Survival.

Posterior inference is the second type of inference. It aims to find the posterior distribution $P(H|E = e)$ for a hidden variable H given some evidence $E = e$. Basically, we infer the posterior probabilities by applying Bayes rule. For example, given that we know the passenger's gender and age, what was her chance to survive? We perform this type of inference when we use our Bayesian network to predict the survival of a single passenger.

Maximum-a-posteriori (MAP) inference is the third type of inference. It is a variational approach for fitting model parameters to training data. We can use it to estimate the distribution of a hidden variable that best explains an observed dataset. In general, variational methods approximate the distribution of a hidden variable analytically. Based on a mathematical expression of the distribution of interest, these methods solve alternate expressions that are known to be close to the original distribution.

Let's do it.

10.1 Learning Hidden Variables

Our quantum Bayesian network is quite simple. It uses only two features of our dataset. Let's add another. Let's add the ticket class (`Pclass`). We already know the ticket class was a pretty important factor determining the `Survival` of the Titanic shipwreck.

If we added `Pclass` as the third condition of `Survival`, the CPT would have twelve cases. One case for each combination. It would be straightforward to create. However, the advantage of a Bayesian network lies in deciding which dependencies to include and which to omit. A network with a more selective structure sometimes reveals interesting patterns in our data. Moreover, a flat Bayesian network is not that interesting.

Let's make a different choice on the design of our network. Thus far, we use the `Age` and the gender (`Sex`) of a passenger because we know about the social norm of "women and children first."

Instead of keeping the direct effect of Age and Sex on Survival, we define another variable. It represents whether a passenger was favored by a norm when the crew decided who may board a lifeboat. It is not limited to women and children. Maybe some men were selected for other aspects we don't know. We name it `Norm`.

Of course, being favored by a `Norm` affects a passenger's chances to survive. Yet, to be favored for a valuable seat in a lifeboat, the passenger must have been close to a lifeboat.

Once no more favored person was in sight of the crew, they might have given the next seat to anyone else in sight. Another decisive factor is the accessibility of the decks where the passengers boarded the lifeboats. Let's assume that the `Pclass` represents this accessibility. To not complicate things too much right away, we say `Pclass` has a direct effect on `Survival`. In contrast the `Age` and `Sex` of a passenger have an indirect impact by determining the `Norm` that

influences `Survival`.

The following figure depicts our updated Bayesian network.

Figure 10.1: Updated Bayesian network
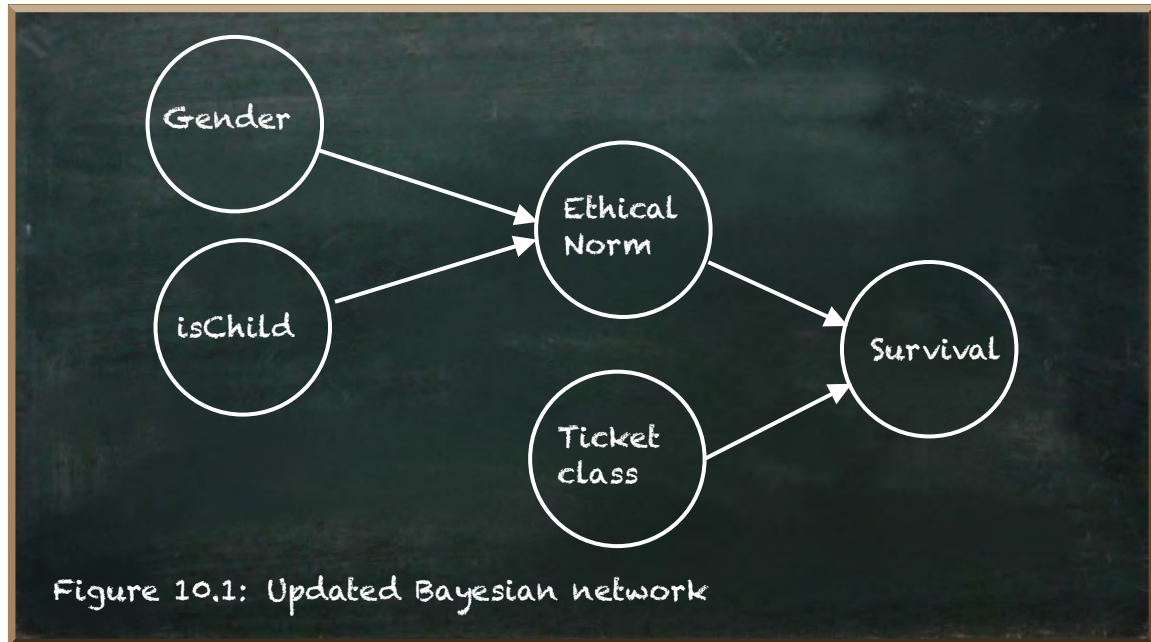
The question is, how do we calculate the CPTs involving the `Norm`. The `Norm` is conditionally dependent on `Age` and `Sex`. And `Survival` is conditionally dependent on `Norm` and `Pclass`. But beyond its relation to other variables, we don't have any data of the `Norm`. It is a hidden variable. Fortunately, given some observed data and the specified relationships, we can infer the CPTs involving the `Norm`.

10.2 Estimating A Single Data Point

Before we calculate `Norm`, let's look at a straightforward case first. We apply a variational method to approximate the distribution of a hidden variable analytically. Let's say we have two binary variables, `A` and `B`. We know they're not independent. `A` is the parent node. `B` is the child node whose conditional probability we try to estimate.

The following figure depicts this Bayesian network.

Usually, we count how many times both variables are true, how many times `A` is true and `B` is false, how many times `A` is false, but `B` is true, and the times both are false. Then, we divide each of these counts by the total number of cases and get the maximum likelihood probabilities.

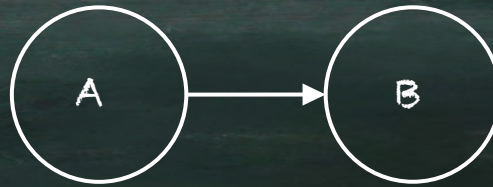Here's the data we have of `A` and `B`. The problem is, we miss one data point.

Figure 10.2: A simple Bayesian network

Figure 10.3: Dataset with missing value

The following listing specifies the data in Python.

Listing 10.1: Dataset with missing value

```
data = [
  (1, 1), (1, 1), (0, 0), (0, 0), (0, 0), (0, None), (0, 1), (1, 0)
]
```

> ⚠ The data have to be missing at random for the methods we apply to be helpful. The data must not be missing for a reason. For instance, if we don't know the Age of passengers who died aboard the Titanic but only know the Age of survivors because we asked them after their rescue, data would not be missing at random. We would have biased data and could not reliably infer the Age of victims from the data. But if we took the Age from a passenger list but we could not read the Age of some due to bad handwriting, we could assume the data to miss at random.

Before we start filling in the missing value, we need an evaluation function. We need some measures to tell us how well we do.

Let's use a likelihood function. Likelihood functions represent the likelihood of a model to result in the observed data. There are two well-known types of

likelihood functions.

The maximum likelihood function is defined as the product of all probability estimations.

$$L(\theta) = \prod_{i=1}^{n} f_i(y_i|\theta)$$

The log-likelihood function takes the natural logarithm of the estimations and sums them.

$$F(\theta) = \sum_{i=1}^{n} \ln f_i(y_i|\theta)$$

In these equations, we calculate the likelihood score (either $L(\theta)$ or $F(\theta)$) based on the data (θ), the variables (y_i, in our example A and B), and the model (f_i). The model is our Bayesian network, including all its parameters.

Both methods produce a single number as output—the likelihood score. More likely events have higher values. Thus, the higher the number, the better the model. However, the likelihood score must be interpreted carefully. The more data you add, the lower the overall score will be. With each probability below 1 (any probability is at maximum 1), you either multiply by a number below 1 (the actual probability if you use maximum likelihood) or you add a number below 0 (the logarithm of a probability below 1 if you use log-likelihood). The result gets smaller and smaller. Consequently, these methods are only meaningful if you compare two (or more) models on the same data.

Compared to the maximum likelihood method, the log-likelihood offers mathematical convenience because it lets us turn multiplication into addition. Therefore, we use the log-likelihood function.

The following function implements the log-likelihood algorithm adapted to our needs.

Listing 10.2: The log-likelihood function adapted for our data

```
 1  from math import log
 2
 3  def log_likelihood(data, prob_a_b, prob_a_nb, prob_na_b, prob_na_nb):
 4    def get_prob(point):
 5      if point[0] == 1 and point[1] == 1:
 6        return log(prob_a_b)
 7      elif point[0] == 1 and point[1] == 0:
 8        return log(prob_a_nb)
 9      elif point[0] == 0 and point[1] == 1:
10        return log(prob_na_b)
11      elif point[0] == 0 and point[1] == 0:
12        return log(prob_na_nb)
13      else:
14        return log(prob_na_b+prob_na_nb)
15
16    return sum(map(get_prob, data))
```

The function expects the `data` to be a list of tuples with two items each. Further, it takes the parameters of our model. These are the probabilities of A and B (`prob_a_b`), A and not B (`prob_a_nb`), not A and B (`prob_na_b`), and not A and not B (`prob_na_nb`) (line 3).

We call the function `get_prob` for each tuple in the list and return the sum of all results. This function `get_prob` takes a data `point` (the tuple) and evaluates the combination of it. It simply returns the logarithm of the corresponding probability. For example, if both values of A and B are 1 it returns `log(prob_a_b` `)`– the probability of A and B.

If we can't identify the combination, we return the logarithm of the sum of `prob_na_b` and `prob_na_nb`. This is the case when we miss the value of B. We have only a single case (`(0, None)`) in our data, and its value of A is `0`. Thus, we know it contains not A. But we're not sure about B.

If our data were different, we would need a different implementation.

We start with importing the Qiskit libraries and the implementation of the `prob_to_angle`-function we introduced earlier.

Listing 10.3: Our known convenience function

```python
from qiskit import QuantumCircuit, Aer, execute
from qiskit import ClassicalRegister, QuantumRegister
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
from math import asin, sqrt

def prob_to_angle(prob):
  return 2*asin(sqrt(prob))
```

Further, we introduce another convenience function. It creates the scaffold of a quantum circuit for us.

Listing 10.4: the as-pqc function

```python
def as_pqc(cnt_quantum, with_qc, cnt_classical=1, shots=1, hist=False,
    measure=False):
  # Prepare the circuit with qubits and a classical bit to hold the
    measurement
  qr = QuantumRegister(cnt_quantum)
  cr = ClassicalRegister(cnt_classical)
  qc = QuantumCircuit(qr, cr) if measure else QuantumCircuit(qr)

  with_qc(qc, qr=qr, cr=cr)

  results = execute(
    qc,
    Aer.get_backend('statevector_simulator') if measure is False else Aer.
    get_backend('qasm_simulator'),
    shots=shots
  ).result().get_counts()

  return plot_histogram(results, figsize=(12,4)) if hist else results
```

The `as_pqc`-function takes as the required parameter the number of qubits (`cnt_quantum`) we use during the initialization of the `QuantumRegister` (line 3). The optional parameter `cnt_classical` takes the number of classical bits we employ in the circuit and initializes the `ClassicalRegister` (line 4). We add the `ClassicalRegister` to the circuit (line 5) only if we include a measurement into our circuit by setting the optional parameter `measure` to `True`. We then measure the qubit at position 0 by default (line 11), and use the `qasm_simulator` (line 15). Further, only in this case, we need to work with multiple `shots` to reproduce

the resulting measurement probability empirically. If we don't measure our qubits, we use the `statevector_simulator` (line 15) that gives us precise probabilities in a single shot. The parameter `hist` specifies whether we want to return a histogram of the results (`True`) or the raw measurement data (`False`).

The parameter `with_qc` is a callback function. We call it with the `QuantumCircuit` (`qc`), the `QuantumRegister` (`qr`), and the `ClassicalRegister` (`cr`) as arguments. This callback function lets us implement the specificities of the PQC.

Next, we implement the quantum Bayesian network. We define the function `qbn`. It takes `data` and `hist` as parameters.

Listing 10.5: The quantum bayesian network

```
 1  def qbn(data, hist=True):
 2    def circuit(qc, qr=None, cr=None):
 3      list_a = list(filter(lambda item: item[0] == 1, data))
 4      list_na = list(filter(lambda item: item[0] == 0, data))
 5
 6      # set the marginal probability of A
 7      qc.ry(prob_to_angle(
 8        len(list_a) / len(data)
 9      ), 0)
10
11      # set the conditional probability of NOT A and (B / not B)
12      qc.x(0)
13      qc.cry(prob_to_angle(
14        sum(list(map(lambda item: item[1], list_na))) / len(list_na)
15      ),0,1)
16      qc.x(0)
17
18      # set the conditional probability of A and (B / not B)
19      qc.cry(prob_to_angle(
20        sum(list(map(lambda item: item[1], list_a))) / len(list_a)
21      ),0,1)
22
23    return as_pqc(2, circuit, hist=hist)
```

We implement the actual quantum circuit in the `circuit` function we pass to `as_pqc` as the callback function (line 23). This callback function starts with the declaration of two lists. The first `list_a` (line 3) contains all the items in our data where the value of A is `1`, representing A is true. The second `list_na` (line 4) includes all items where the value of A is `0` representing not A.

We use these lists to calculate the probabilities of the four combinations ($A \wedge B$,

$A \wedge \neg B$, $\neg A \wedge B$, $\neg A \wedge \neg B$).

We start with the marginal probability of A (lines 7-9). This is the number of items in our data where A is true (1, the length of `list_a`) divided by the total number of items in our data (size of `data`). We let the qubit at position 0 represent this probability.

Next, we separate the cases where A is false into those where B is true and those where B is false (lines 12-16). First, we "activate" the state where A is false by applying the NOT-gate on the qubit at position 0 (line 12). The controlled R_Y-gate sets the qubit at position 1 into state $|1\rangle$ when B is true (line 14). We calculate the probability by dividing the number of items where A is false and B is true (`sum(list(map(lambda item: item[1], list_na)))`) divided by the number of items where A is false (`len(list_na)`). Of course, we need to switch the qubit back to state $|1\rangle$ in the cases where A is true (line 16).

Finally, we separate the cases where A is true into those where B is true, and those where B is false. We apply another controlled R_Y-gate. The rotation angle represents the probability of B is true given A is also true (line 20).

The following figure depicts this quantum circuit graphically.

Figure 10.4: The quantum Bayesian network with two nodes
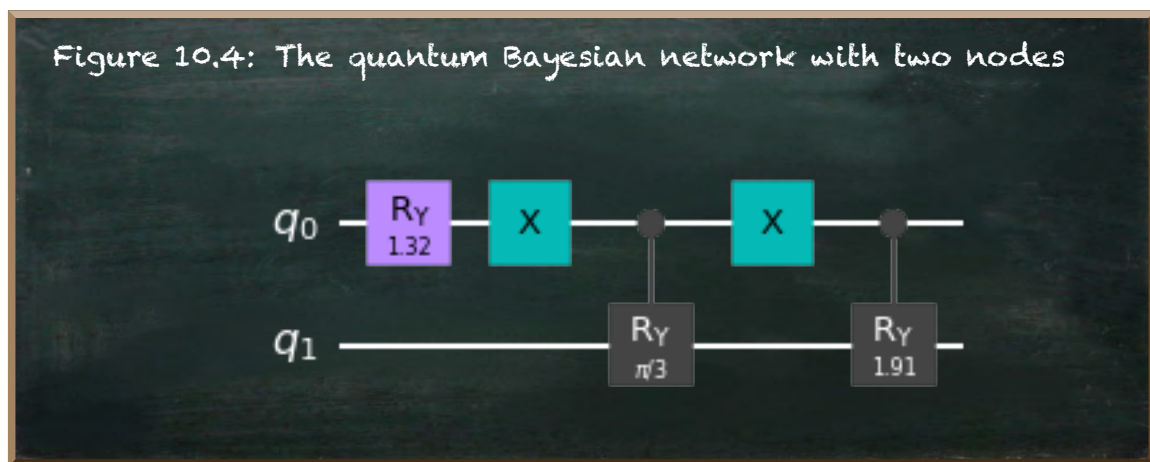
Let's now see it in action. The first and simplest way of coping with missing data is to ignore it. We drop the item with the missing item (`list(filter(lambda item: item[1] is not None ,data))`) before we pass the `data` to the `qbn` function.

Listing 10.6: Ignoring the missing data

```
1 qbn(list(filter(lambda item: item[1] is not None ,data)))
```

Figure 10.5: Resulting probabilities when irgnoring the mssing data
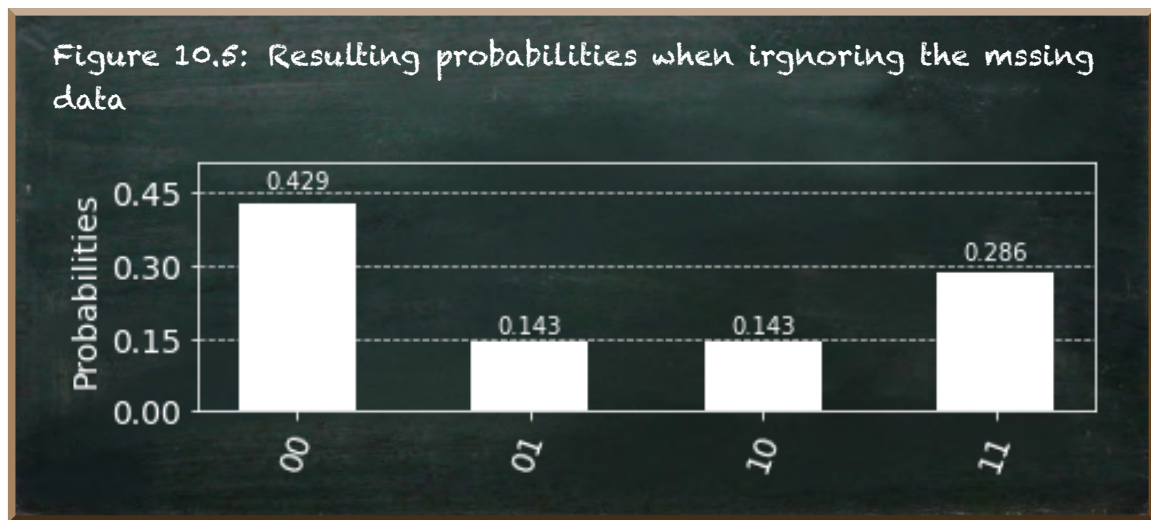
The measurement probabilities of the four different states represent the probabilities of the possible combinations of A or $\neg A$ and B or $\neg B$.

Let's feed the results into the `log_likelihood` function we created earlier. We create another convenience function for that.

Listing 10.7: Calculate the log-likelihood when ignoring the missing data

```
 1  def eval_qbn(model, prepare_data, data):
 2    results = model(prepare_data(data), hist=False)
 3    return round(log_likelihood(data,
 4      results['11'], # prob_a_b
 5      results['01'], # prob_a_nb
 6      results['10'], # prob_na_b
 7      results['00'] # prob_na_nb
 8    ), 3)
 9
10  eval_qbn(qbn, lambda dataset: list(filter(lambda item: item[1] is not
        None ,dataset)), data)
```

```
-9.499
```

This function `eval_qbn` (line 1) takes the `qbn`-function as the model. But we can plug in any other model, too, as long as it takes a dataset of the given format and returns the results we obtain from Qiskit. The second parameter `prepare_data` is a function that takes care of the missing data point. We put in our data and expect the dataset we put into our model (line 2).

The function returns the log-likelihood score of the given model (line 3). Therefore, we provide the probability measures we get from the quantum circuit (lines 4-7). Note that the states we get from the quantum circuit read from the right (qubit at position 0 represents *A*) to the left (qubit at position 1 represents *B*).

In this example, we provide a function that filters out the missing item (`filter (lambda item: item[1] is not None)`) (line 10).

The results show a log-likelihood score of -9.499. As mentioned, the overall value does not say much. We see how good it is when we compare it with other models.

Next, let's try to fill in a value for the missing data. In three of five cases where *A* is `0`, *B* is `0`, too. *B* is `1` in only one of these cases. And, one time, it is missing. Thus, filling in `0` seems to be the better option.

Listing 10.8: Calculate the log-likelihood when filling in 0

```
1  eval_qbn(qbn, lambda dataset: list(map(lambda item: item if item[1] is
       not None else (item[0], 0) ,dataset)), data)
```

```
-9.481
```

When filling in `0` for the missing item, we get a log-likelihood score of -9.481. This is an improvement over -9.499, the value of the previous model.

Let's try to find the value that makes the log-likelihood (of the actual data) the biggest.

In a Bayesian network, we work with probabilities all the time. So why don't we fill in the missing data point with a probability distribution instead of a particular value?

Wait! How could we fill in the value with a probability distribution if this distribution is what we aim to calculate in the first place? Let's do something extraordinary. Spock!

Figure 10.6: A Guess?
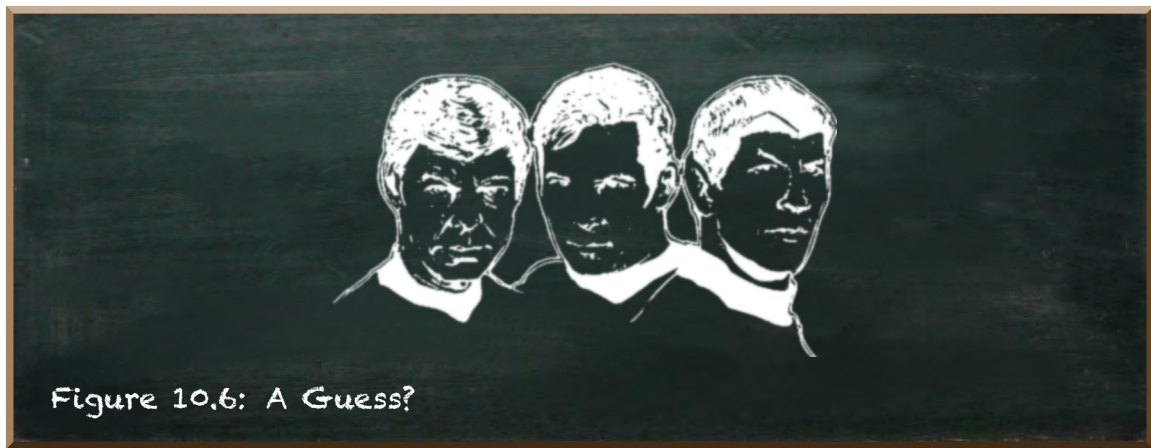
Kirk: "Mr. Spock, *have you accounted for the variable mass of whales and water in your time re-entry program?*"

Spock: "*Mr. Scott cannot give me exact figures, Admiral, so... I will make a guess.*"

Kirk: "*A guess? You, Spock? That's extraordinary.*"

Spock to McCoy: "*I don't think he understands.*"

McCoy: "*No, Spock. He means that he feels safer about your guesses than most other people's facts.*"

Spock: "*Then you're saying... it is a compliment?*"

McCoy: "*It is.*"

Spock: "*Ah. Then I will try to make the best guess I can.*"

Listing 10.9: The updated eval-qbn

```
 1  def eval_qbn(model, prepare_data, data):
 2    results = model(prepare_data(data), hist=False)
 3    return (
 4      round(log_likelihood(data,
 5        results['11'], # prob_a_b
 6        results['01'], # prob_a_nb
 7        results['10'], # prob_na_b
 8        results['00'] # prob_na_nb
 9      ), 3),
10      results['10'] / (results['10'] + results['00'])
11    )
```

We guess a distribution. And we do not only take the log-likelihood score but also the distribution of B given $\neg A$. We need to edit the qbn function to get this distribution.

The updated eval_qbn-function did not change much. It simply adds another number as its returned value. Now, it returns a tuple. At the first position of the tuple, it returns the log-likelihood score. At the second position (line 10), it returns the probability of B given $\neg A$

$$P(B|\neg A) = \frac{P(\neg A \wedge B)}{P(\neg A)} = \frac{P(\neg A \wedge B)}{P(\neg A \wedge B) + P(\neg A \wedge \neg B)}$$

So, let's start by initializing our distribution with $P(B|\neg A) = 0.5$.

Listing 10.10: Evaluation of the guess

```
1 eval_qbn(qbn, lambda dataset: list(map(lambda item: item if item[1] is
    not None else (item[0], 0.5) ,dataset)), data)
```

```
(-9.476, 0.3)
```

It seems as if it was a pretty good guess. The log-likelihood score is at -9.476.

But we don't stop there. The model tells us a new value of $P(B|\neg A) = 0.3$. Let's run our model with this value.

Listing 10.11: Refining the model

```
1 eval_qbn(qbn, lambda dataset: list(map(lambda item: item if item[1] is
    not None else (item[0], 0.3) ,dataset)), data)
```

```
(-9.452, 0.26)
```

Our model improves. We got a log-likelihood score of -9.452 and a new distribution for our missing data point.

We can iterate between filling in the missing data with the distribution and estimating a new probability distribution.

Listing 10.12: Further refining the model

```
1  eval_qbn(qbn, lambda dataset: list(map(lambda item: item if item[1] is
       not None else (item[0], 0.26) ,dataset)), data)
```

```
(-9.451, 0.252)
```

Listing 10.13: Another iteration

```
1  eval_qbn(qbn, lambda dataset: list(map(lambda item: item if item[1] is
       not None else (item[0], 0.252) ,dataset)), data)
```

```
(-9.451, 0.2504)
```

This iterative process is an example of a general procedure called the expectation-maximization (EM) algorithm.

But for how long do we have to iterate?

If you don't get tired first, you iterate until the score converges. In our example, the log-likelihood score did not improve (measurably) between the values of 0.252 and 0.2504.

It can be hard to tell when the EM has converged, though. Sometimes, the models just get a tiny bit better for a long time. Once you think the process is done, there is a sudden increase in the score. There's no way to tell.

Another problem with EM is that it is subject to local maxima. As a result, it might effectively converge to a maximum near to the starting point even though there's a much better model with different values. To prevent from getting stuck in a local maximum, you can either run the algorithm multiple times starting at different initial values or——if available——you can use domain knowledge to initialize the model.

10.3   Estimating A Variable

Let's get back to our quantum Bayesian network consisting of four nodes. The Age and Sex of a passenger determine the Norm. The Norm and the Pclass determine Survival.

	female	male
$P(Sex)$	0.35	0.65

	child	adult
$P(isChild)$	0.08	0.92

Sex

isChild

$P(Norm = Favored|Sex, isChild)$

	female	male
child	?	?
adult	?	?

Norm

	1st	2nd	3rd
$P(isChild)$	0.24	0.21	0.55

Class

$P(Survival = True|Norm, Class)$

	1st	2nd	3rd
favor	?	?	?
unfav,	?	?	?

Survival

Figure 10.7: The CPTs including the norm
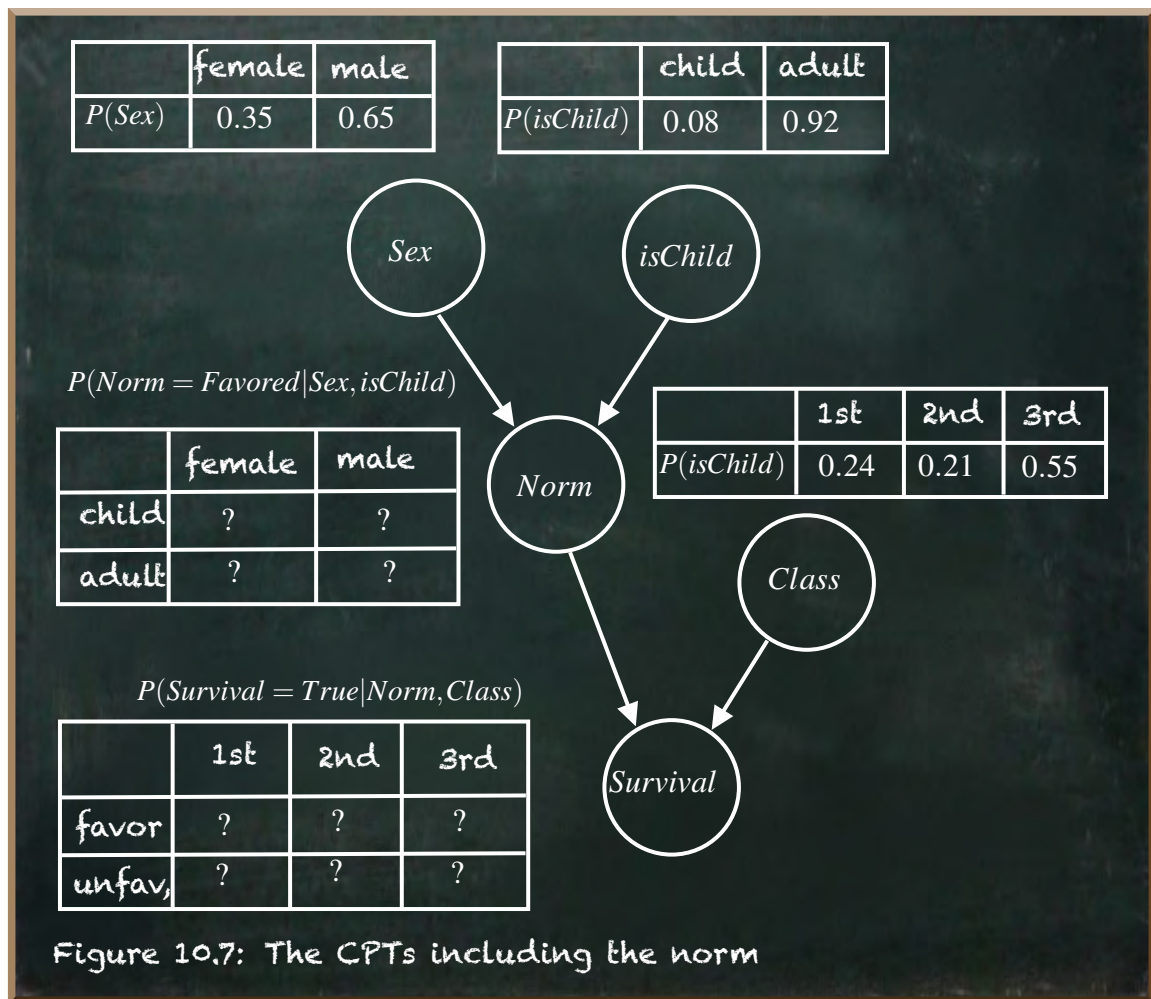
Our data consists of all the cases of passengers onboard the Titanic. The dataset contains observations of Age, Sex, and Survival. These are observable variables. The values of the Norm are missing data. The Norm is a hidden variable.

The image above depicts the missing CPT of our Bayesian network.

We aim to find the CPTs that maximize the probability of the observed data.

Rather than writing a single big function, we split our code into small pieces we can put together at the end. Let's start with the marginal probabilities of being a child (isChild) and a passenger's gender (Sex).

Listing 10.14: Apply the known

```python
1  import pandas as pd
2  train = pd.read_csv('./data/train.csv')
3
4  # the maximum age of a passenger we consider as a child
5  max_child_age = 8
6
7  # probability of being a child
8  population_child = train[train.Age.le(max_child_age)]
9  p_child = len(population_child)/len(train)
10
11 # probability of being female
12 population_female = train[train.Sex.eq("female")]
13 p_female = len(population_female)/len(train)
14
15 # positions of the qubits
16 QPOS_ISCHILD = 0
17 QPOS_SEX = 1
18
19 def apply_ischild_sex(qc):
20     # set the marginal probability of isChild
21     qc.ry(prob_to_angle(p_child), QPOS_ISCHILD)
22
23     # set the marginal probability of Sex
24     qc.ry(prob_to_angle(p_female), QPOS_SEX)
```

We keep the maximum age of 8 years of a passenger we consider as a child (line 5). The probability of being a child is given by the number of children (line 8) divided by the total number of passengers (line 9).

We do the same calculation for the passenger being female (lines 12-13).

We specify two constant values, QPOS_ISCHILD and QPOS_SEX (lines 16-17). These depict the positions of the qubits that represent the respective marginal probabilities.

We use the R_Y-gate and the prob_to_angle-function to put the qubits into the corresponding states. The qubit at position QPOS_ISCHILD has the probability being in state $|0\rangle$ that corresponds to the probability of the passenger being an adult. The probability of being in state $|1\rangle$ is the probability of the passenger being a child (line 21).

Accordingly, the qubit at position QPOS_SEX represents the probabilities of the passenger being male (state $|0\rangle$) and being female (state $|1\rangle$) (line 24).

In the next step, we specify the conditional probabilities of being favored by a norm.

Listing 10.15: Represent the norm

```python
# position of the qubit representing the norm
QPOS_NORM = 2

def apply_norm(qc, norm_params):
  """
  norm_params = {
    'p_norm_am': 0.25,
    'p_norm_af': 0.35,
    'p_norm_cm': 0.45,
    'p_norm_cf': 0.55
  }
  """

  # set the conditional probability of Norm given adult/male
  qc.x(QPOS_ISCHILD)
  qc.x(QPOS_SEX)
  ccry(qc, prob_to_angle(
    norm_params['p_norm_am']
  ),QPOS_ISCHILD, QPOS_SEX, QPOS_NORM)
  qc.x(QPOS_ISCHILD)
  qc.x(QPOS_SEX)

  # set the conditional probability of Norm given adult/female
  qc.x(QPOS_ISCHILD)
  ccry(qc, prob_to_angle(
    norm_params['p_norm_af']
  ),QPOS_ISCHILD, QPOS_SEX, QPOS_NORM)
  qc.x(QPOS_ISCHILD)

  # set the conditional probability of Norm given child/male
  qc.x(QPOS_SEX)
  ccry(qc, prob_to_angle(
    norm_params['p_norm_cm']
  ),QPOS_ISCHILD, QPOS_SEX, QPOS_NORM)
  qc.x(QPOS_SEX)

  # set the conditional probability of Norm given child/female
  ccry(qc, prob_to_angle(
    norm_params['p_norm_cf']
  ),QPOS_ISCHILD, QPOS_SEX, QPOS_NORM)
```

We also define another constant to keep the position of the qubit that represents the `Norm` (line 2).

The function `apply_norm` applies the conditional probability given a set of parameters (`norm_params`). This is a Python dictionary with four key-value pairs. The keys are `p_norm_am`, `p_norm_af`, `p_norm_cm`, and `p_norm_cf`. The values are the conditional probabilities of being favored by a norm given the passenger is a male adult (`p_norm_am`), a female adult (`p_norm_af`), a male child (`p_norm_cm`), and a female child (`p_norm_cf`).

For each of these conditional probabilities, we select the qubit states representing the marginal probabilities of `isChild` and `Sex` using X-gates. For instance, the state $|00\rangle$ of the qubits at the positions `QPOS_ISCHILD` and `QPOS_SEX` represents the probability of being a male adult. Since the CCR_Y-gate only applies a rotation on the controlled qubit if both control qubits are in state $|1\rangle$, we need to switch both qubits first (lines 15-16). Now, the state $|11\rangle$ represents the probability of a male adult. We apply the corresponding conditional probability (`prob_to_angle(norm_params['p_norm_am'])`) (lines 17-18) and switch back the state of the control qubits (lines 20-21).

We do the same for the other three conditional probabilities, too.

Listing 10.16: Calculate the probabilities related to the ticket-class

```
1  pop_first = train[train.Pclass.eq(1)]
2  surv_first = round(len(pop_first[pop_first.Survived.eq(1)])/len(pop_first)
       , 2)
3  p_first = round(len(pop_first)/len(train), 2)
4
5  pop_second = train[train.Pclass.eq(2)]
6  surv_second = round(len(pop_second[pop_second.Survived.eq(1)])/len(
       pop_second), 2)
7  p_second = round(len(pop_second)/len(train), 2)
8
9  pop_third = train[train.Pclass.eq(3)]
10 surv_third = round(len(pop_third[pop_third.Survived.eq(1)])/len(pop_third)
       , 2)
11 p_third = round(len(pop_third)/len(train), 2)
12
13 print("First class: {} of the passengers, survived: {}".format(p_first,
       surv_first))
14 print("Second class: {} of the passengers, survived: {}".format(p_second,
       surv_second))
15 print("Third class: {} of the passengers, survived: {}".format(p_third,
       surv_third))
```

First class: 0.24 of the passengers, survived: 0.63
Second class: 0.21 of the passengers, survived: 0.47
Third class: 0.55 of the passengers, survived: 0.24

Now, let's turn to the marginal probability of having a ticket of a certain class (`Pclass`) and the respective chances to survive.

The calculation of the probabilities is straightforward. The marginal probability of owning a ticket is given by the number of tickets of the respective class divided by the total number of passengers (lines 3, 7, 11). The conditional probability of surviving given a certain ticket class is the quotient of the number of survivors and the total number of passengers with a ticket of that class (lines 2, 6, 10).

Listing 10.17: Represent the ticket-class

```python
# positions of the qubits
QPOS_FIRST = 3
QPOS_SECOND = 4
QPOS_THIRD = 5

def apply_class(qc):
  # set the marginal probability of Pclass=1st
  qc.ry(prob_to_angle(p_first), QPOS_FIRST)

  qc.x(QPOS_FIRST)
  # set the marginal probability of Pclass=2nd
  qc.cry(prob_to_angle(p_second/(1-p_first)), QPOS_FIRST, QPOS_SECOND)

  # set the marginal probability of Pclass=3rd
  qc.x(QPOS_SECOND)
  ccry(qc, prob_to_angle(p_third/(1-p_first-p_second)), QPOS_FIRST,
    QPOS_SECOND, QPOS_THIRD)
  qc.x(QPOS_SECOND)
  qc.x(QPOS_FIRST)
```

Thus far, we only had to cope with boolean variables. These are variables that have only two possible values. The `Pclass` is different because there are three different ticket classes, 1st, 2nd, and 3rd.

Technically, we could represent three values by using two qubits. But we will use three. We represent the probability of having a ticket of a certain class by

a single one qubit being in state $|1\rangle$. We start with applying an R_Y-gate on the qubit at position `QPOS_FIRST`. It lets this qubit be in state $|1\rangle$ with the probability of a passenger having a first-class ticket (line 8).

Now, given that the passenger doesn't have a first-class ticket, we want the qubit at position `QPOS_FIRST` to be in state $|0\rangle$. Therefore, we temporarily switch the amplitudes of this qubit by an X-gate (line 10). This allows us to use this qubit as a control qubit for controlled rotations of the qubits representing the other two classes.

We apply the rotation on the qubit at position `QPOS_SECOND` only if qubit `QPOS_FIRST` is in state $|1\rangle$. Temporarily, this is the case if the passenger does not have a first-class ticket. Since the control qubit is not always in state $|1\rangle$, we have to adjust the rotation angle we apply. We apply the conditional probability of having a second-class ticket given the passenger doesn't have a first-class ticket (`prob_to_angle(p_second/(1−p_first))`) (line 12).

Next, we want the qubit at position `QPOS_SECOND` to be in state $|0\rangle$ if the passenger has a third-class ticket. Thus, we temporarily switch its amplitude, too (line 15). Now, we apply a controlled-controlled rotation of the qubit at position `QPOS_THIRD`. We apply the conditional probability of having a third-class ticket given the passenger doesn't have a first-class or a second-class ticket (`prob_to_angle(p_third/(1−p_first−p_second))`) (line 16).

Finally, we undo the temporary switches of the amplitudes of the qubits at the positions `QPOS_SECOND` (line 17) and `QPOS_FIRST` (line 18). Each of these three qubits has the probability of being in state $|1\rangle$ now that corresponds to the probability of the passenger having a ticket of the corresponding class. And if one of these three qubits is in state $|1\rangle$, the other two qubits are in state $|0\rangle$ because a passenger can only have a ticket of a single class.

In the next listing, we apply the conditional probability of `Survival`.

Listing 10.18: Represent survival

```
 1  # position of the qubit
 2  QPOS_SURV = 6
 3
 4  def apply_survival(qc, surv_params):
 5    """
 6    surv_params = {
 7      'p_surv_f1': 0.3,
 8      'p_surv_f2': 0.4,
 9      'p_surv_f3': 0.5,
10      'p_surv_u1': 0.6,
11      'p_surv_u2': 0.7,
12      'p_surv_u3': 0.8
13    }
14    """
15
16    # set the conditional probability of Survival given unfavored by norm
17    qc.x(QPOS_NORM)
18    ccry(qc, prob_to_angle(
19      surv_params['p_surv_u1']
20    ),QPOS_NORM, QPOS_FIRST, QPOS_SURV)
21
22    ccry(qc, prob_to_angle(
23      surv_params['p_surv_u2']
24    ),QPOS_NORM, QPOS_SECOND, QPOS_SURV)
25
26    ccry(qc, prob_to_angle(
27      surv_params['p_surv_u3']
28    ),QPOS_NORM, QPOS_THIRD, QPOS_SURV)
29    qc.x(QPOS_NORM)
30
31    # set the conditional probability of Survival given favored by norm
32    ccry(qc, prob_to_angle(
33      surv_params['p_surv_f1']
34    ),QPOS_NORM, QPOS_FIRST, QPOS_SURV)
35
36    ccry(qc, prob_to_angle(
37      surv_params['p_surv_f2']
38    ),QPOS_NORM, QPOS_SECOND, QPOS_SURV)
39
40    ccry(qc, prob_to_angle(
41      surv_params['p_surv_f3']
42    ),QPOS_NORM, QPOS_THIRD, QPOS_SURV)
```

Again, we start with the specification of the qubit position (line 1). Like the function `apply_norm`, the function `apply_survival` takes a Python dictionary as a parameter that holds all the probabilities we want to apply on the qubits.

The value at the key `p_surv_f1` represents the probability of surviving given the passenger was favored (`f`) by a norm and had a first-class ticket (`1`). The key `p_surv_u3` represents the probability of surviving given the passenger was unfavored (`u`) by a norm and had a third-class ticket (`3`). The other keys depict all the possible combinations.

We use a temporary X-gate (line 17) to set the qubit representing the `Norm` to the value we want to apply. We start with the unfavored passengers (lines 18-28). Before we continue with the favored passengers, we switch the qubit back (line 29).

Since we prepared three qubits to represent the three ticket classes, we do not need another X-gate to activate the corresponding state. But we can use a CCR_Y-gate with the qubits representing the `Norm` (`QPOS_NORM`) and the respective ticket class (`QPOS_FIRST`, `QPOS_SECOND`, or `QPOS_THIRD`) as the control qubits.

With these few functions, we can create a parameterized quantum circuit easily.

Listing 10.19: The quantum bayesian network

```
1  QUBITS = 7
2
3  def qbn_titanic(norm_params, surv_params, hist=True, measure=False, shots
      =1):
4    def circuit(qc, qr=None, cr=None):
5      apply_ischild_sex(qc)
6      apply_norm(qc, norm_params)
7      apply_class(qc)
8      apply_survival(qc, surv_params)
9
10   return as_pqc(QUBITS, circuit, hist=hist, measure=measure, shots=shots)
```

We define the function `qbn_titanic` (line 3). It takes two Python dictionaries (`norm_params` and `surv_params`) that contain all the parameters we need to construct the circuit. We use the `as_pqc` function we created earlier again (line 11). Finally, we pass the number of qubits our circuit should have (defined in line 1) and a callback function (`circuit`) that constructs the actual circuit (lines 4-8).
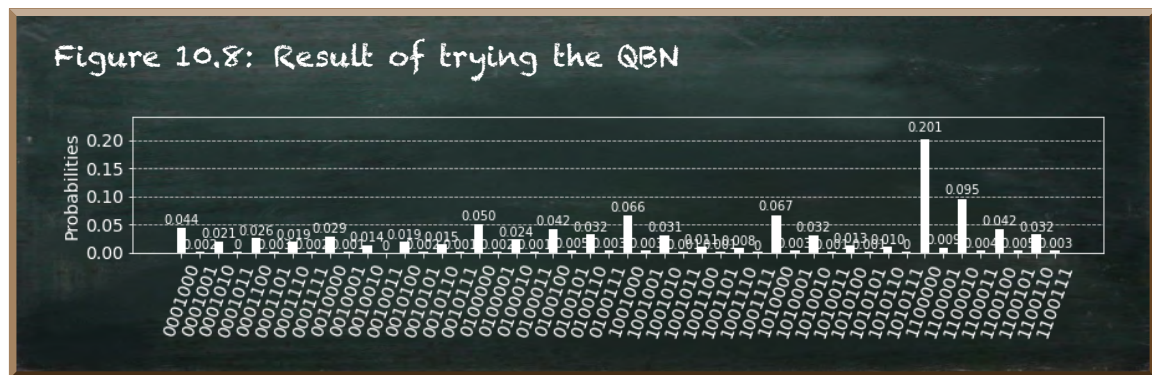
Let's try to run it with some arbitrary parameters.

Listing 10.20: Try the QBN

```python
 1  norm_params = {
 2      'p_norm_am': 0.25,
 3      'p_norm_af': 0.35,
 4      'p_norm_cm': 0.45,
 5      'p_norm_cf': 0.55
 6  }
 7
 8  surv_params = {
 9      'p_surv_f1': 0.3,
10      'p_surv_f2': 0.4,
11      'p_surv_f3': 0.5,
12      'p_surv_u1': 0.6,
13      'p_surv_u2': 0.7,
14      'p_surv_u3': 0.8
15  }
16
17  qbn_titanic(norm_params, surv_params, hist=True)
```


Figure 10.8: Result of trying the QBN

It produces a set of quite a few states with associated probabilities. These resulting probabilities are quite meaningless because we made up the input parameters. We need to derive them from the data.

Let's start with the norm_params.

Listing 10.21: Calculate the parameters of the norm

```python
def calculate_norm_params(passengers):
    # the different populations in our data
    pop_children = passengers[passengers.IsChild.eq(1)]
    pop_adults = passengers[passengers.IsChild.eq(0)]

    # combinations of being a child and gender
    pop_am = pop_adults[pop_adults.Sex.eq('male')]
    pop_af = pop_adults[pop_adults.Sex.eq('female')]
    pop_cm = pop_children[pop_children.Sex.eq('male')]
    pop_cf = pop_children[pop_children.Sex.eq('female')]

    norm_params = {
      'p_norm_am': pop_am.Norm.sum() / len(pop_am),
      'p_norm_af': pop_af.Norm.sum() / len(pop_af),
      'p_norm_cm': pop_cm.Norm.sum() / len(pop_cm),
      'p_norm_cf': pop_cf.Norm.sum() / len(pop_cf),
    }

    return norm_params
```

The function `calculate_norm_params` takes the Pandas `dataframe` of the passengers and returns the `norm_params` dictionary. First, we specify different populations (groups) of passengers (lines 2-10). Then, we calculate the probabilities of a passenger being favored by a Norm (`Norm`) given the passenger belongs to a group (lines 12-17).

We separate the children from the adults in the data by evaluating whether the value of column `IsChild` is `1` (children) or `0` (adults) (lines 3-4). We further split these two groups into four based on the gender(`Sex`) being `female` or `male` (lines 7-10).

Let's pay some attention to how we calculate the probabilities of a passenger being favored by a Norm (lines 13-16). We sum the `Norm` of all passengers of a group and divide by the number of passengers in the group. `Norm` is the hidden variable. Similar to the example of a missing value, we will fill this column with a number between 0 and 1 that represents the probability of the respective passenger to be favored by a norm.

For example, if we have ten passengers and five have a value of 0, and five have a value of 1, we get a resulting probability of $P(Norm) = (5 \cdot 1 + 5 \cdot 0)/10 = 0.5$. Likewise, if we have five passengers with a value of 0.75 and five with a value of 0.25, we get a resulting probability of $P(Norm) = (5 \cdot 0.75 + 5 \cdot 0.25)/10 = 0.5$.

Next, we calculate the `surv_params`.

Listing 10.22: Calculate the parameters of survival

```python
def calculate_surv_params(passengers):
  # all survivors
  survivors = passengers[passengers.Survived.eq(1)]

  # weight the passenger
  def weight_passenger(norm, pclass):
    return lambda passenger: (passenger[0] if norm else 1−passenger[0]) *
     (1 if passenger[1] == pclass else 0)

  # calculate the probability to survive
  def calc_prob(norm, pclass):
    return sum(list(map(
      weight_passenger(norm, pclass),
      list(zip(survivors['Norm'], survivors['Pclass']))
    ))) / sum(list(map(
      weight_passenger(norm, pclass),
      list(zip(passengers['Norm'], passengers['Pclass']))
    )))

  surv_params = {
    'p_surv_f1': calc_prob(True, 1),
    'p_surv_f2': calc_prob(True, 2),
    'p_surv_f3': calc_prob(True, 3),
    'p_surv_u1': calc_prob(False, 1),
    'p_surv_u2': calc_prob(False, 2),
    'p_surv_u3': calc_prob(False, 3)
  }

  return surv_params
```

Let's go through this function backward from the bottom to the top. First, we return the Python dictionary that contains the conditional probabilities of survival given the passenger's ticket class and whether a norm favored him or her.

We use a convenience function `calc_prob` to calculate these probabilities given the specific values of being favored (the first parameter is `True` or `False`) and the ticket class (either `1`, `2`, or `3`) (lines 20-25).

The probability to survive is defined by the number of survivors in a group (numerator) divided by the total number of passengers in a group (denomi-

nator). The problem is that the `Norm` column contains a number between 0 and 1–the probability that a norm favors the passenger. Therefore, we can't count the passengers, but we have to "weigh" each of them.

If a passenger has a `Norm` value of 0.75, she belongs to the group of favored passengers in 75% of the cases and the group of unfavored passengers in 25% cases. If we sum the weights of all survivors and divide it by the sum of the weights of all passengers, it yields the probability (lines 11-17).

Since we do this calculation of the weight twice (lines 12 and 15), we put it into another function (`weight_passenger`) (lines 5-7).

We create tuples of the `Norm` and the `Pclass` values of a passenger (using Python's `zip`-function) and pass them into `weight_passenger` as input (lines 13 and 16).

If we weigh whether a passenger belongs to a group of favored passengers, we use the `Norm`-value. But if we weigh whether a passenger belongs to a group of unfavored passengers, we need to calculate this probability as $1-$`Norm`. This is the first part of our `weight_passenger` function (`passenger[0] if norm else 1−` `passenger[0]`, line 7). But we do not only need to consider the `Norm` but also the `Pclass`. Rather than separating these groups upfront, we set the weight of a passenger to `0` if the `pclass` does not fit the group we are calculating (`(1 if passenger[1] == pclass else 0)`). Then, the returned weight is also `0` for we multiply both values.

Maybe, you wonder where the `IsChild` and `Norm` columns in the data set come from. Do you? We explained their meaning and what kind of values we expect them to have. Yet, they are not part of the original data. Just like the missing value in our first example, we have to fill them ourselves.

Listing 10.23: Prepare the data

```python
def prepare_data(passengers, params):
    """
    params = {
        'p_norm_cms': 0.45,
        'p_norm_cmd': 0.46,
        'p_norm_cfs': 0.47,
        'p_norm_cfd': 0.48,
        'p_norm_ams': 0.49,
        'p_norm_amd': 0.51,
        'p_norm_afs': 0.52,
        'p_norm_afd': 0.53,
    }
    """
    # is the passenger a child?
    passengers['IsChild'] = passengers['Age'].map(lambda age: 0 if age >
      max_child_age else 1)

    # the probability of favored by norm given Age, Sex, and Survival
    passengers['Norm'] = list(map(
        lambda item: params['p_norm_{}{}{}'.format(
            'a' if item[0] == 0 else 'c',
            item[1][0],
            'd' if item[2] == 0 else 's'
        )],
        list(zip(passengers['IsChild'], passengers['Sex'], passengers['
          Survived']))
    ))
    return passengers
```

Filling the values of the IsChild-column is quite simple. We defined all passengers with an age of max_child_age=8 or below to be a child. All older passengers are considered adults. Since we have the Age of the passengers, it is a simple calculation (line 15).

Things are different concerning the Norm. We do not have the value of any passenger. In the previous example, we only had two variables, A and B. And we had the data of both for most cases. It was clear we used the values of A to determine the probability values of B.

In the passenger data set, there are a lot more variables. How do we determine which variables to use to determine the value of Norm? The answer is in our Bayesian network. Per definition, it contains all dependencies that mat-

ter. The value of `Norm` depends on `IsChild` and `Sex`, and it affects `Survival`. There-
fore, we need to condition the probability of `Norm` on these three variables. If
any other variable mattered, we would need to include it in our Bayesian net-
work.

Since `IsChild`, `Sex`, and `Survival` have two possible values each, there are eight
possible combinations ($2^3 = 8$). We expect these probabilities to be provided
as parameters in a Python dictionary `params` (line 1). The three letters at the
end indicate whether the passenger was a child (`c`) or an adult (`a`), a male (`m`) or
a female (`f`), and whether the passenger survived (`s`) or died (`d`).

We create tuples of these three variables using the `zip` function (line 24). We
`map` the tuple for each passenger (line 18) onto the corresponding value of the
`params` dictionary (line 19). We select the right key-value pair by creating the
proper string (lines 19-23). If the value of `IsChild` is 0 we insert an `a`, otherwise
a `c` (line 20). We take the first letter of the column `Sex` that contains either `male`
or `female` (line 21). And we insert a `d` if the value of `Survival` is 0 and an `s` if it is 1
(line 22).

As a result, each passenger has a value of `Norm` between 0 and 1 that represents
the probability of being favored by a norm given the other data of the passen-
ger.

Now, we're ready to train our Bayesian network. We need to find the best set
of values for the parameters.

We start the iterative process by initializing the set of parameters with arbi-
trary values. We should not use zeros unless we are sure that they are correct
in our domain.

Listing 10.24: Initialize the parameters

```
# Step 0: Initialize the parameter values
params = {
    'p_norm_cms': 0.45,
    'p_norm_cmd': 0.46,
    'p_norm_cfs': 0.47,
    'p_norm_cfd': 0.48,
    'p_norm_ams': 0.49,
    'p_norm_amd': 0.51,
    'p_norm_afs': 0.52,
    'p_norm_afd': 0.53,
}
```

Let's put all the components together.

Listing 10.25: Run the qbn

```
1 passengers = prepare_data(train, params)
2 results = qbn_titanic(calculate_norm_params(passengers),
       calculate_surv_params(passengers), hist=False)
```

First, we prepare the passenger data (line 1) that we use to calculate the `norm_params` by `calculate_norm_params(passengers)` and the `surv_params` by `calculate_surv_params(passengers)` (line 2). We set `hist=False` to get a Python dictionary as a result instead of a histogram. This dictionary is quite long. Remember, the qubit positions read from the right (position 0) to left (position 6).

```
{'0001000': 0.026844845468374,
 '0001001': 0.001902440523418,
 '0001010': 0.014135181707589,
 '0001011': 0.001006254687522,

 ...

 '1100100': 0.040445280069615,
 '1100101': 0.002358986860305,
 '1100110': 0.022705559076552,
 '1100111': 0.001326392674849}
```

From this data, we can derive all probabilities by summing the respective state probabilities. For instance, if we wanted to know the chance to survive, we need to look at the qubit at position `QPOS_SURV` = 6 is 1. Let's write a small convenience function to get a list of all the relevant states.

Listing 10.26: Get a list of relevant states

```
1 def filter_states(states, position, value):
2   return list(filter(lambda item: item[0][QUBITS−1−position] == str(
      value), states))
```

We expect as input a list of tuples, like this: `[('0001000', 0.026844845468374),` `('0001001', 0.001902440523418), ... ]`. We can create this list using the `states.` `items()`-function that turns the dictionary with the states into such a list.

We get one tuple for each state in the dictionary with the first value (position 0) of the tuple is the state string and the second value (position 1) is the probability.

This allows us to iterate through the state using the `filter` function. It runs the anonymous (`lambda`) function for each item in the list and returns a list of

those items the lambda function returns `True` for. It returns `True` if the state string (`item[0]`) has the requested `value` at the position of the specified qubit (`[QUBITS−1−position]`).

Here's the list of states where the passenger survived.

Listing 10.27: The states with surviving passengers

```
1 filter_states(results.items(), QPOS_SURV, '1')
```

```
[('1001000', 0.04521802427481),
 ('1001001', 0.003204510969179),
 ('1001010', 0.023809598395179),
 ('1001011', 0.001694956632945),
 ...
 ('1100100', 0.040445280069615),
 ('1100101', 0.002358986860305),
 ('1100110', 0.022705559076552),
 ('1100111', 0.001326392674849)]
```

The sum of all these states depict the marginal probability of survival.

Listing 10.28: Calculate the marginal probability to survive

```
1 def sum_states(states):
2    return sum(map(lambda item: item[1], states))
3
4 sum_states(filter_states(results.items(), QPOS_SURV, '1'))
```

```
0.383652628610444
```

We use the `map` function only to keep the probability of each tuple (`map(lambda item: item[1])`) and return the `sum` of all these values.

As we see, the probability of surviving is the value we expect (0.38). But this is not a measure for the performance of our Bayesian network. Our measure is the log-likelihood score. Let's have a look at it.

Listing 10.29: The log-likelihood function adapted for our data

```python
def log_likelihood_titanic(data, results):
  states = results.items()

  def calc_prob(norm_val, ischild_val, sex_val, surv_val):
    return sum_states(
      filter_states(
        filter_states(
          filter_states(
            filter_states(states, QPOS_SEX, sex_val),
            QPOS_ISCHILD, ischild_val
          ), QPOS_SURV, surv_val
        ), QPOS_NORM, norm_val))

  probs = {
    'p_fcms': calc_prob('1', '1', '0', '1'),
    'p_fcmd': calc_prob('1', '1', '0', '0'),
    'p_fcfs': calc_prob('1', '1', '1', '1'),
    'p_fcfd': calc_prob('1', '1', '1', '0'),
    'p_fams': calc_prob('1', '0', '0', '1'),
    'p_famd': calc_prob('1', '0', '0', '0'),
    'p_fafs': calc_prob('1', '0', '1', '1'),
    'p_fafd': calc_prob('1', '0', '1', '0'),
    'p_ucms': calc_prob('0', '1', '0', '1'),
    'p_ucmd': calc_prob('0', '1', '0', '0'),
    'p_ucfs': calc_prob('0', '1', '1', '1'),
    'p_ucfd': calc_prob('0', '1', '1', '0'),
    'p_uams': calc_prob('0', '0', '0', '1'),
    'p_uamd': calc_prob('0', '0', '0', '0'),
    'p_uafs': calc_prob('0', '0', '1', '1'),
    'p_uafd': calc_prob('0', '0', '1', '0'),
  }

  return round(sum(map(
    lambda item: log(probs['p_{}{}{}{}'.format(
        'u',
        'a' if item[1] == 0 else 'c',
        item[2][0],
        'd' if item[3] == 0 else 's'
      )] + probs['p_{}{}{}{}'.format(
        'f',
        'a' if item[1] == 0 else 'c',
        item[2][0],
        'd' if item[3] == 0 else 's'
      )]
    ),
    list(zip(data['Norm'], data['IsChild'], data['Sex'], data['Survived'])
    )
  )), 3)
```

We start with the definition of the `calc_prob`-function (line 4) that retrieves the probability of a certain combination of values for our variables. We consider the variables `Sex`, `IsChild`, `Survival`, and `Norm`. We use the `filter_states` -function we just created to filter out all the states that have the respective values (lines 6-12) and we return the sum of these states' probabilities (line 5).

We use the `calc_prob`-function to fill a dictionary with all the probabilities we need (lines 14-31). For instance, `'p_fcms': calc_prob('1', '1', '0', '1')` is the probability of the passenger to be favored by a norm, a child, female, and had survived.

Now, we put these columns into a list of tuples (line 46) and calculate for each passenger the logarithm of his or her probabilities. While `Sex`, `IsChild`, and `Survival` are known values, we can select the right probability. But `Norm` contains a number between 0 and 1. We are not sure whether the passenger was favored or not. Thus, we need to take the logarithm of the sum of both possible probabilities–the probability of the passenger being favored (lines 34-38) and the probability of the passenger not being favored (lines 39-44).

The total log-likelihood score is the sum of all the passenger's values (line 33).

Listing 10.30: Calculate the log-likelihood

```
1  log_likelihood_titanic(train, results)
```

```
  -1860.391
```

Our initial parameters result in a log-likelihood score of around -1860.39. As we mentioned earlier, the absolute score does not tell us much. It only tells us which of the two models perform better on the same data.

So, let's improve our model. We need better values for our parameters. Similar to the probabilities we used to calculate the log-likelihood score, we can obtain the values for our parameters from the results of running our Bayesian network.

Listing 10.31: Obtain new parameter values from the results

```python
def to_params(results):
  states = results.items()

  def calc_norm(ischild_val, sex_val, surv_val):
    pop = filter_states(filter_states(filter_states(states, QPOS_SEX,
    sex_val), QPOS_ISCHILD, ischild_val), QPOS_SURV, surv_val)

    p_norm = sum(map(lambda item: item[1], filter_states(pop, QPOS_NORM,
    '1')))
    p_total = sum(map(lambda item: item[1], pop))
    return p_norm / p_total


  return {
    'p_norm_cms': calc_norm('1', '0', '1'),
    'p_norm_cmd': calc_norm('1', '0', '0'),
    'p_norm_cfs': calc_norm('1', '1', '1'),
    'p_norm_cfd': calc_norm('1', '1', '0'),
    'p_norm_ams': calc_norm('0', '0', '1'),
    'p_norm_amd': calc_norm('0', '0', '0'),
    'p_norm_afs': calc_norm('0', '1', '1'),
    'p_norm_afd': calc_norm('0', '1', '0'),
  }
```

The `to_params`-function takes the results as a parameter (line 1). It returns a
dictionary with the probabilities of being favored by a norm given the set of
values for the variables `IsChild`, `Sex`, and `Survival` (lines 12-21). These are condi-
tional probabilities. We first filter all the states that have the specified values
for the variables (line 5). From this set, we filter those states where the `Norm`
has value 1 (`filter_states(pop, QPOS_NORM, '1')`, line 7).

The conditional probability of being favored by a norm is the probability of
those state where the `Norm` has value 1 divided by the probability of all states
with the specified values for the variables (line 9).

Listing 10.32: Calcualte new parameters

```python
to_params(results)
```

```
{'p_norm_cms': 0.45532583440735436,
 'p_norm_cmd': 0.4593883474337892,
 'p_norm_cfs': 0.47052387314654737,
 'p_norm_cfd': 0.47460383062928546,
 'p_norm_ams': 0.5039016630401505,
 'p_norm_amd': 0.5079933634215915,
 'p_norm_afs': 0.5199079166576689,
 'p_norm_afd': 0.5239923091774149}
```

These values differ slightly from the ones we started with. This time, we may need a few more iterations to get a good set of parameters. Let's automate the training.

Listing 10.33: The recursive training automatism

```
 1  def train_qbn_titanic(passengers, params, iterations):
 2    if iterations > 0:
 3      new_params = train_qbn_titanic(passengers, params, iterations − 1)
 4
 5      passengers = prepare_data(passengers, new_params)
 6      results = qbn_titanic(calculate_norm_params(passengers),
        calculate_surv_params(passengers), hist=False)
 7
 8      print ('The log−likelihood after {} iteration(s) is {}'.format(
        iterations, log_likelihood_titanic(passengers, results)))
 9      return to_params(results)
10
11    return params
```

The function `train_qbn_titanic` takes the `passengers` data set, the initial `params`, and the number of `iterations` as parameters (line 1). If the number of `iterations` is `0`, we simply return the `params` (line 11). However, if the number of `iterations` is greater `0` (line 2), `train_qbn_titanic` recursively calls itself–yet, with a reduced number of `iterations`. As a result, we pass the initial `params` through all the calls until `iterations` reach `0` as a value. Then, we get back the same `params` (line 11).

We use the returned value (`new_params`) (line 3) to prepare the data set of passengers (line 5) and derive the `norm_params` and `surv_params` from it (line 6). We run the `qbn_titanic` and obtain the results (line 6). We use the results to calculate and print the log-likelihood-score (line 8) and–most importantly–to calculate and return the new parameters (`to_params(results)`, line 9).

Listing 10.34: Train the QBN

```
1  trained_params = train_qbn_titanic(train, {
2    'p_norm_cms': 0.45,
3    'p_norm_cmd': 0.46,
4    'p_norm_cfs': 0.47,
5    'p_norm_cfd': 0.48,
6    'p_norm_ams': 0.49,
7    'p_norm_amd': 0.51,
8    'p_norm_afs': 0.52,
9    'p_norm_afd': 0.53,
10 }, 25)
```

```
The log-likelihood after 1 iteration(s) is -1860.391
The log-likelihood after 2 iteration(s) is -1860.355
The log-likelihood after 3 iteration(s) is -1860.332
The log-likelihood after 4 iteration(s) is -1860.3
The log-likelihood after 5 iteration(s) is -1860.243
The log-likelihood after 6 iteration(s) is -1860.13
The log-likelihood after 7 iteration(s) is -1859.901
The log-likelihood after 8 iteration(s) is -1859.426
The log-likelihood after 9 iteration(s) is -1858.439
The log-likelihood after 10 iteration(s) is -1856.393
The log-likelihood after 11 iteration(s) is -1852.213
The log-likelihood after 12 iteration(s) is -1843.99
The log-likelihood after 13 iteration(s) is -1829.057
The log-likelihood after 14 iteration(s) is -1805.719
The log-likelihood after 15 iteration(s) is -1777.24
The log-likelihood after 16 iteration(s) is -1752.49
The log-likelihood after 17 iteration(s) is -1737.602
The log-likelihood after 18 iteration(s) is -1730.95
The log-likelihood after 19 iteration(s) is -1728.411
The log-likelihood after 20 iteration(s) is -1727.468
The log-likelihood after 21 iteration(s) is -1727.107
The log-likelihood after 22 iteration(s) is -1726.965
The log-likelihood after 23 iteration(s) is -1726.908
The log-likelihood after 24 iteration(s) is -1726.884
The log-likelihood after 25 iteration(s) is -1726.872
```

The next iteration gets these parameters as `new_params` (line 3) and does the
same. One by one, the recursive calls of `train_qbn_titanic` resolve and return

an improved set of parameters to its predecessor.

We see how the log-likelihood score improves each iteration. It converges after about 22 iterations. Let's have a look at the `trained_params` after the training.

Listing 10.35: The parameters after training

```
1 trained_params
```

```
{'p_norm_cms': 0.6021334301303094,
 'p_norm_cmd': 0.07088902981523437,
 'p_norm_cfs': 0.9904336919724537,
 'p_norm_cfd': 0.8392179490424515,
 'p_norm_ams': 0.49195927424087027,
 'p_norm_amd': 0.04654642501038004,
 'p_norm_afs': 0.9978526500251851,
 'p_norm_afd': 0.9590619707414763}
```

The result is not surprising. We see a high probability (almost 1) of all females to be favored by a norm. However, we see male children (`p_norm_cms` and `p_norm_cmd`) not being favored a lot.

10.4 Predict Survival

Let's see how the performs on predicting the survival of the passengers.

We apply our proven procedure of a Variational Quantum-Classical Algorithm. We start with pre-processing the data to prepare a quantum state, evaluate the quantum state, and post-process the measurement.

The pre-processing is quite simple.

Listing 10.36: Pre-processing

```
1 def pre_process(passenger):
2   return (passenger['IsChild'] == 1, passenger['Sex'] == 'female',
      passenger['Pclass'])
```

We take the entry of a passenger and return a tuple of whether the value of

IsChild is 1, the value of Sex is female, and the ticket-class. We apply these three values in the quantum circuit.

Listing 10.37: Apply the known data on the quantum circuit

```python
def apply_known(qc, is_child, is_female, pclass):
  if is_child:
    qc.x(QPOS_ISCHILD)

  if is_female:
    qc.x(QPOS_SEX)

  qc.x(QPOS_FIRST if pclass == 1 else (QPOS_SECOND if pclass == 2 else
    QPOS_THIRD))
```

If the passenger is a child, we apply an X-gate on the qubit at the position QPOS_ISCHILD (lines 2-3). Since Qiskit initializes the qubits in state $|0\rangle$, we set this qubit to $|1\rangle$ for children and leave it in state $|0\rangle$ for adults.

We apply the same logic to set the qubit at position QPOS_SEX to $|1\rangle$ only if the passenger is female (lines 5-6).

Finally, we set the qubit to $|1\rangle$ that represents the ticket-class of the passenger (line 8).

To apply the known data, we need to rewrite the QBN a little bit. Thus far, the qbn_titanic takes all the probabilities as parameters. But we want it to apply the known values based on the passenger data and take all the other probabilities from the set of trained parameters.

The latter do not change anymore. We only want to do this calculation once. Therefore, we write a function get_trained_qbn that does all the one-time stuff and returns a simple function we can feed with the data of a single passenger.

The function get_trained_qbn takes the passengers and the trained parameters as input (line 1). It prepares the passenger data (filling the values IsChild and Norm) (line 3) and calculates the norm_params and the surv_params. These things are done only once.

We define another function, trained_qbn_titanic, that only takes a tuple of the three values of a passenger (line 7). This returns a prepared PQC (line 17) with the quantum circuit (lines 10-15) that applies the known data of the passenger (line 11), the norm_params (line 12), and the surv_params (line 13).

Listing 10.38: Get the trained QBN

```python
def get_trained_qbn(passengers, params):

  prepared_passengers = prepare_data(passengers, params)
  norm_params = calculate_norm_params(prepared_passengers)
  surv_params = calculate_surv_params(prepared_passengers)

  def trained_qbn_titanic(passenger):
    (is_child, is_female, pclass) = passenger

    def circuit(qc, qr, cr):
      apply_known(qc, is_child, is_female, pclass)
      apply_norm(qc, norm_params)
      apply_survival(qc, surv_params)

      qc.measure(qr[QPOS_SURV], cr[0])

    return as_pqc(QUBITS, circuit, hist=False, measure=True, shots=100)

  return trained_qbn_titanic
```

We return the `trained_qbn_titanic`-function (line 19).

In the post-processing, we transform the `counts` we get back from the executed PQC into the predicted value of the passenger's `Survival`.

Listing 10.39: Post-processing

```python
def post_process(counts):
  """
  counts -- the result of the quantum circuit execution
  returns the prediction
  """
  #print (counts)
  p_surv = counts['1'] if '1' in counts.keys() else 0
  p_died = counts['0'] if '0' in counts.keys() else 0

  #return int(list(map(lambda item: item[0], counts.items())))[0])
  return 1 if p_surv > p_died else 0
```

We use the convenience function `classifier_report` we created in section 2.7. But we extend it a little bit. This time, we run the circuit many times and

evaluate whether we saw more predicted survivals or more predicted deaths in the data.

Listing 10.40: Run the Quantum Naïve Bayes Classifier

```python
# redefine the run-function
def run(f_classify, data):
    return [f_classify(data.iloc[i]) for i in range(0,len(data))]

# get the simple qbn
trained_qbn = get_trained_qbn(train, trained_params)

# evaluate the Quantum Bayesian Network
classifier_report("QBN",
    run,
    lambda passenger: post_process(trained_qbn(pre_process(passenger))),
    passengers,
    train['Survived'])
```

```
The precision score of the QBN classifier is 0.79
The recall score of the QBN classifier is 0.60
The specificity score of the QBN classifier is 0.90
The npv score of the QBN classifier is 0.78
The information level is: 0.77
```

The overall information level of the quantum Bayesian network classifier is about 0.79. It is an improvement of the score of the quantum Naïve Bayes classifier we created in section 7.

11. The World Is Not A Disk

11.1 The Qubit Phase

> **"There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy."**

(Hamlet (1.5.167-8), Hamlet to Horatio)

Figure 11.1: Hamlet, Horatio, Marcellus, and the Ghost
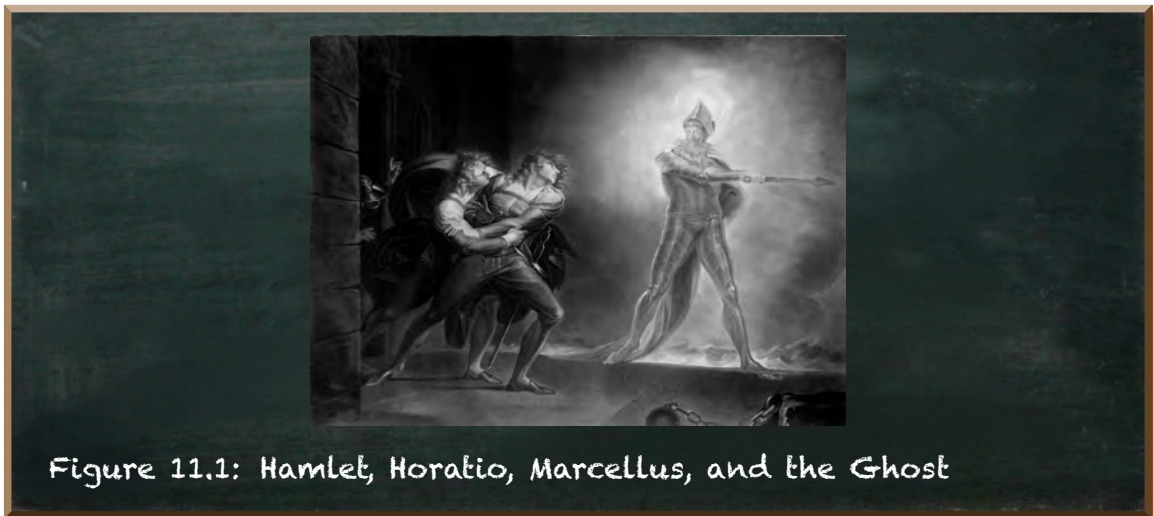
Do you remember our introductory example of the quantum coin? Coins have two states, heads or tails. In the air, our quantum coin is in a state of superposition of both states.

Let's have a closer look at this coin. It continuously flips between heads and tails. Once the coin lands, this rotation determines whether we see heads or tails.

Look even closer. This coin also spins along its edge.
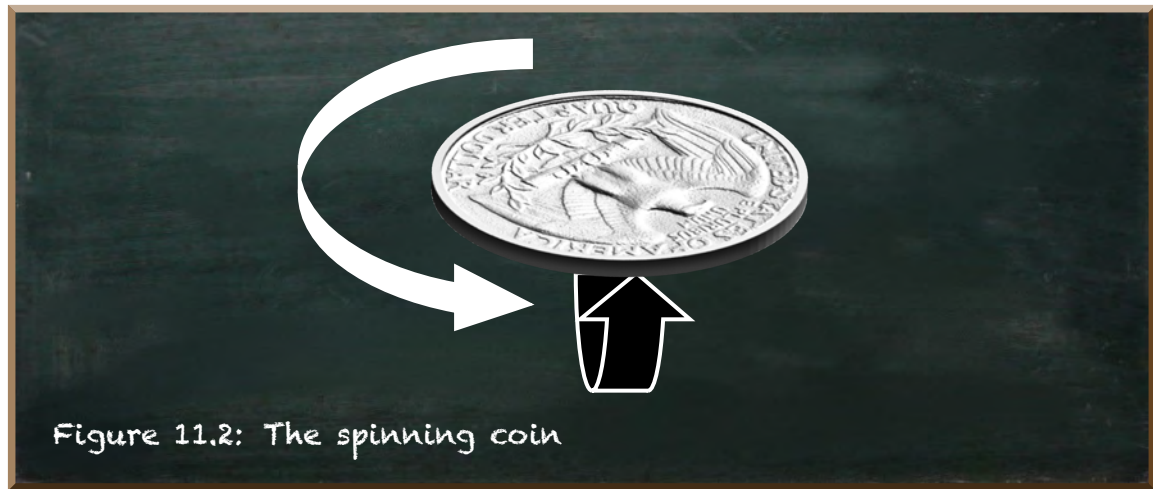
Figure 11.2: The spinning coin

At first sight, the spin doesn't matter for the direction of heads and tails. But it affects the coin's trajectory through the air. And this matters for the orientation once the coin lands, doesn't it?

In quantum mechanics, particles have such a spin, too. It is the phase. In physical terms, the quantum mechanical phase originates from the concept that every quantum entity may be described as a particle and as a wave.

The debate over whether light is particles or waves dates back over three hundred years. In the seventeenth century, Isaac Newton proclaimed that light consists of a stream of particles. About two hundred years later, in the nineteenth century, Thomas Young countered light consists of waves. He devised the double-slit experiment to prove his claim.

In this experiment, a beam of light is aimed at a barrier with two vertical slits. The light passes through the slits, and the resulting pattern is recorded on a photographic plate.

If one slit is covered, there's a single line of light aligned with whichever slit is open. Intuitively, we would expect to see two lines aligned with the slits if both slits are open. But something else happens. The photographic plate is entirely separated into multiple lines of lightness and darkness in varying degrees.

The implications of Young's experiment are counter-intuitive if you regard light as a stream of particles. But it makes perfect sense once you start think-

Figure 11.3: The double-slit experiment
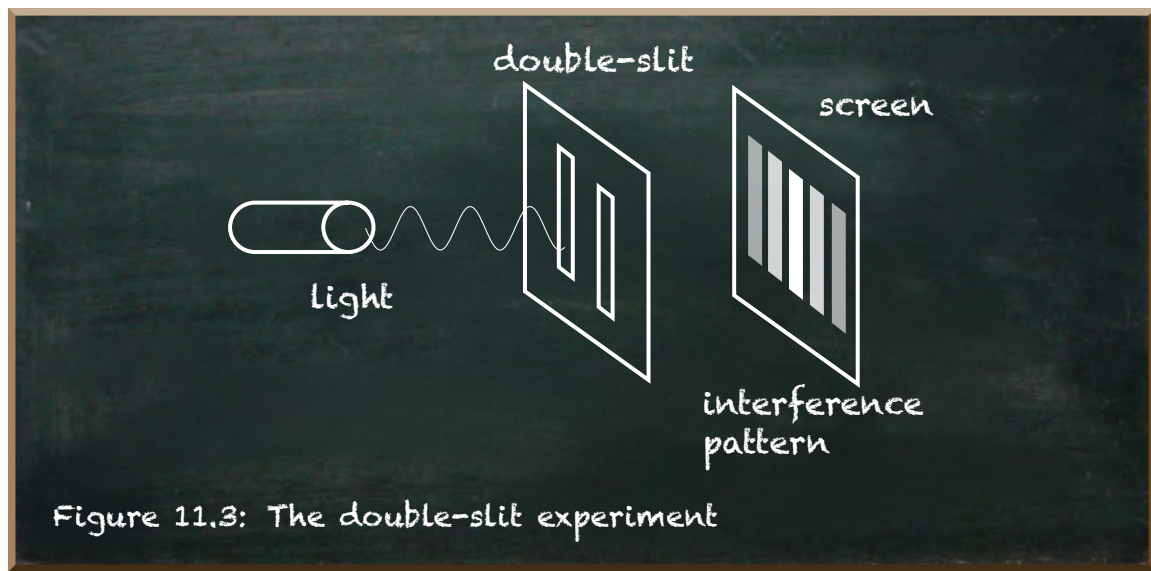
ing of light as a wave.

The main characteristic of a wave is that it goes up and down as it moves. This entails a variety of other characteristics.

- The wavelength is the distance over which the wave's shape repeats.
- The amplitude of a wave is the distance between its center and its crest.

The following figure depicts these characteristics.

Figure 11.4: Properties of a wave
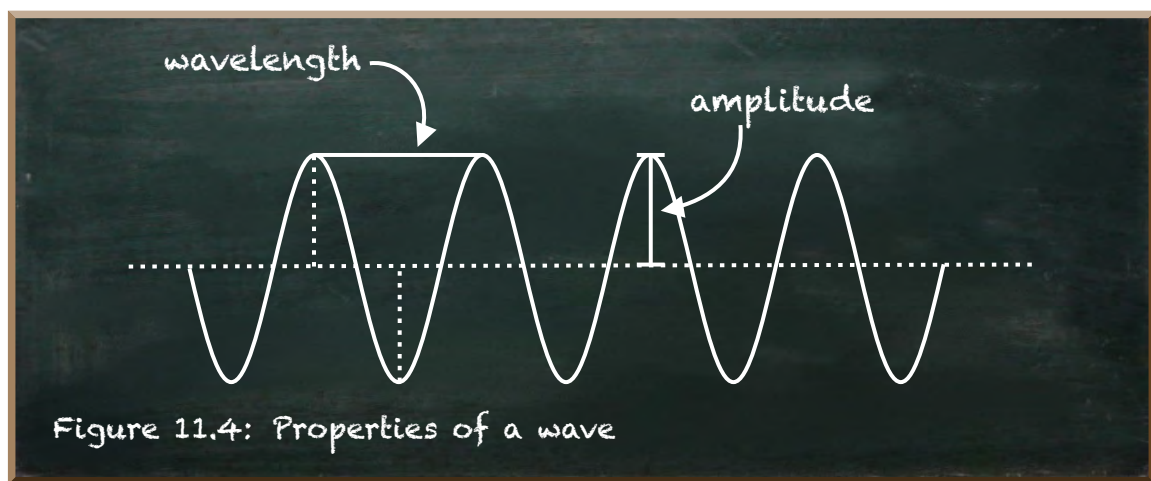
Another distinct property of waves is that they interfere with each other. Simply put, they add up. If you have two waves traveling on the same medium, they form a third wave representing the sum of their amplitudes at each point.

For example, if you have two identical waves that have their crests at the same point, the resulting wave has a greater amplitude.

Figure 11.5: Two interfering waves
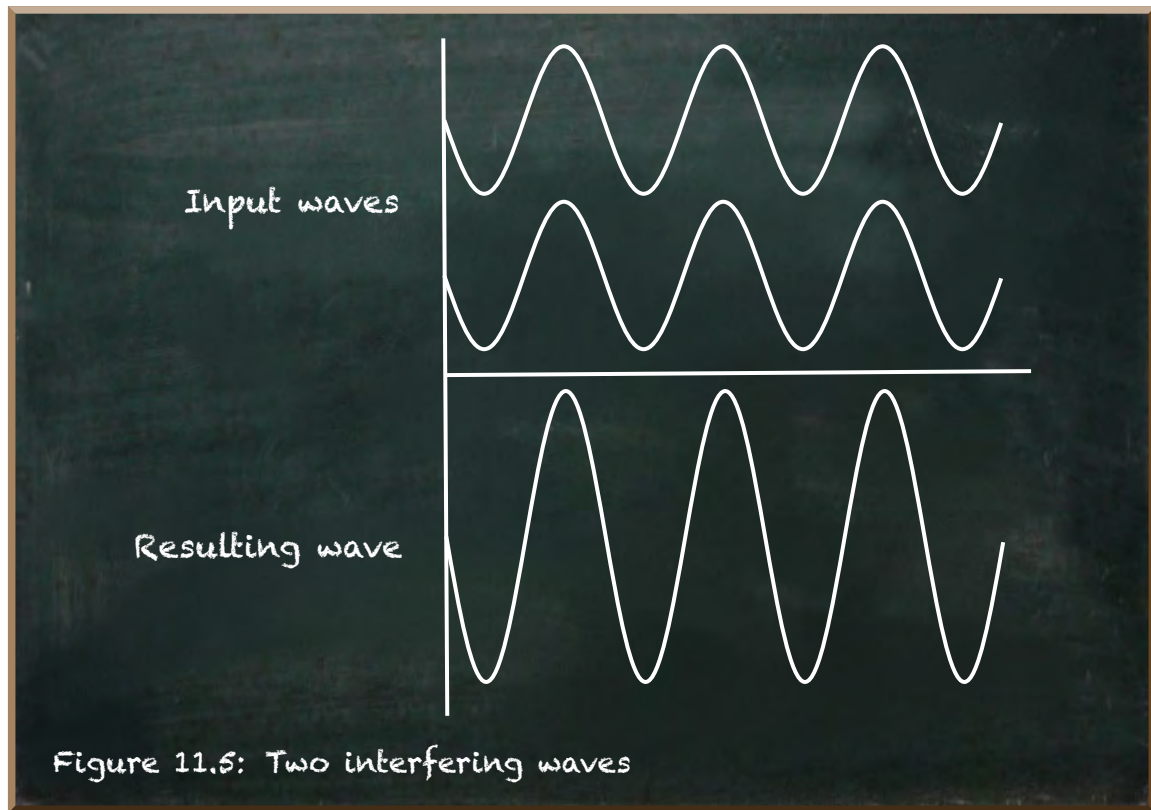
But if the waves are shifted so that the crest of one wave overlaps with the trough of the other wave, they cancel each other.

These two waves have the same amplitude and the same wavelength. But they differ in the relative positions of their crests and troughs. This relative position is the phase of the wave.

For the outside world, the phase of a wave is not observable. Observed individually, the two waves appear identical. But once waves interfere with each other, the phase matters.

Let's go back to the double-slit experiment and look at how two waves travel from the slits to the screen. The two slits have different distances to any given point on the screen. With one wave traveling a longer distance than the other, their relative positions differ for each point on the screen. At some points, they hit the screen being in the same phase. They constructively interfere and lighten the area. At other points, one wave is at its crest, whereas the other is at its trough. They destructively interfere, and the area stays dark.

The qubit is a quantum mechanical entity we can describe as a wave. It has a

Figure 11.6: Two cancelling waves

Figure 11.7: Interference of double-slit
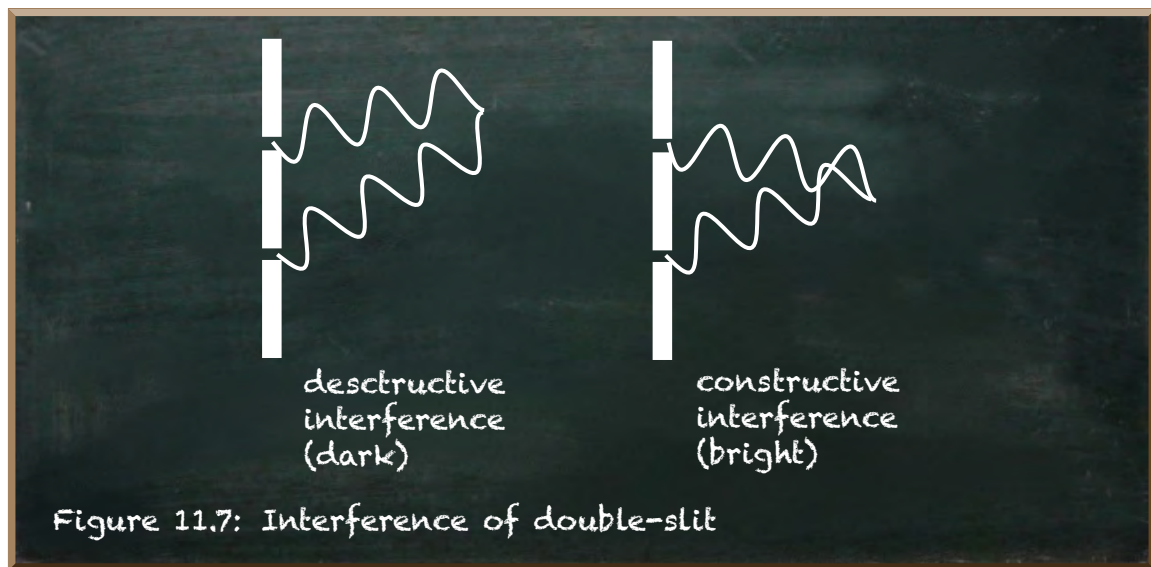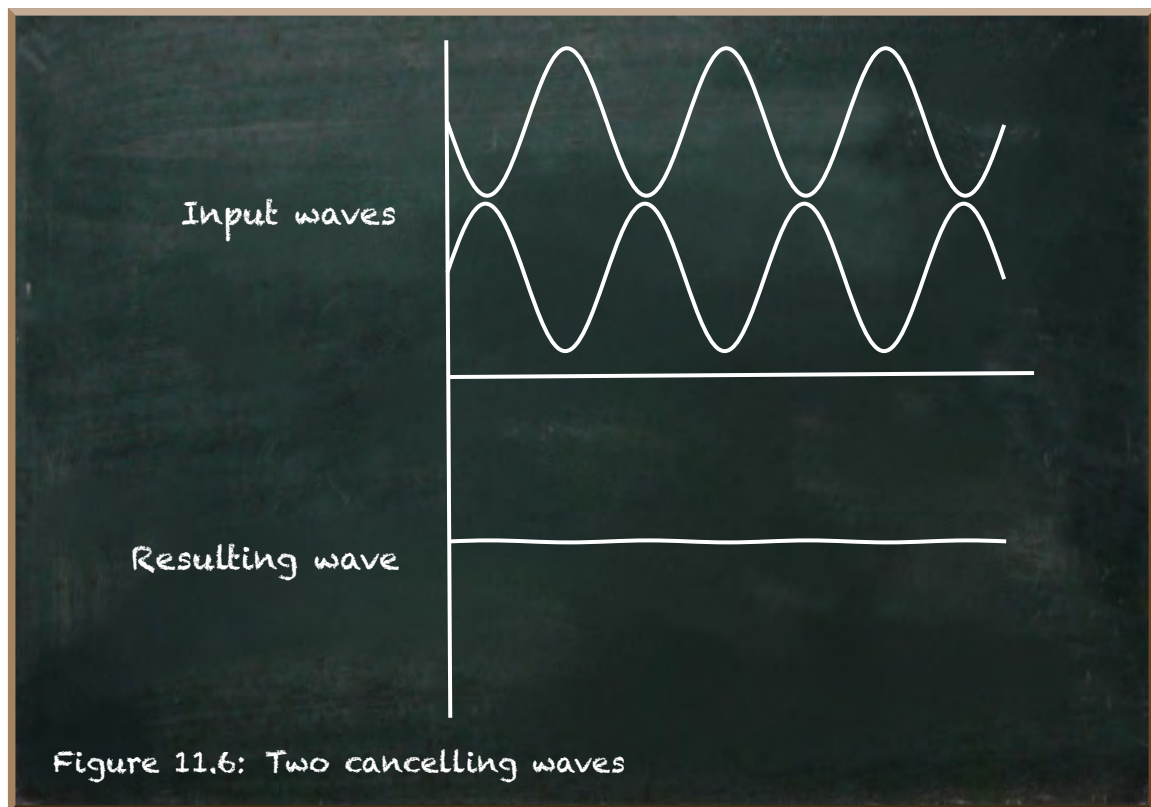
phase, too.

We learned the qubit is in a state of superposition of states $|0\rangle$ and $|1\rangle$ with α and β are the corresponding amplitudes. Mathematically, the superposition is defined as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. When observed, the probability of the qubit to

result in ❶ equals α^2. And, it equals β^2 to result in ❶. Thus, $\alpha^2 + \beta^2 = 1$ because the sum of the probabilities of all possible states must add up to 1 (100%).

The sign of the amplitudes does not matter for the resulting measurement probability. Thus, similar to waves, a qubit with a negative amplitude has the same measurement probability as the original qubit.

Graphically, the qubit state is a vector. And vectors are directed. There is a difference between a vector and a negative vector (opposite direction). But the measurement probabilities are defined as the distance between the head of the qubit state vector and the standard basis vectors $|0\rangle$ and $|1\rangle$. And distances have no direction.

There are two different qubit states for each pair of measurement probabilities of $|0\rangle$ and $|1\rangle$. For instance, the states $\frac{\alpha|0\rangle + \beta|1\rangle}{2}$ and $\frac{\alpha|0\rangle - \beta|1\rangle}{2}$ have the identical measurement probabilities. So does any pair of states $|\psi\rangle$ and $|\psi'\rangle$ whose state vector is mirrored at the Z-axis as depicted in the following image.

Figure 11.8: Two different states with the same measurement probabilities
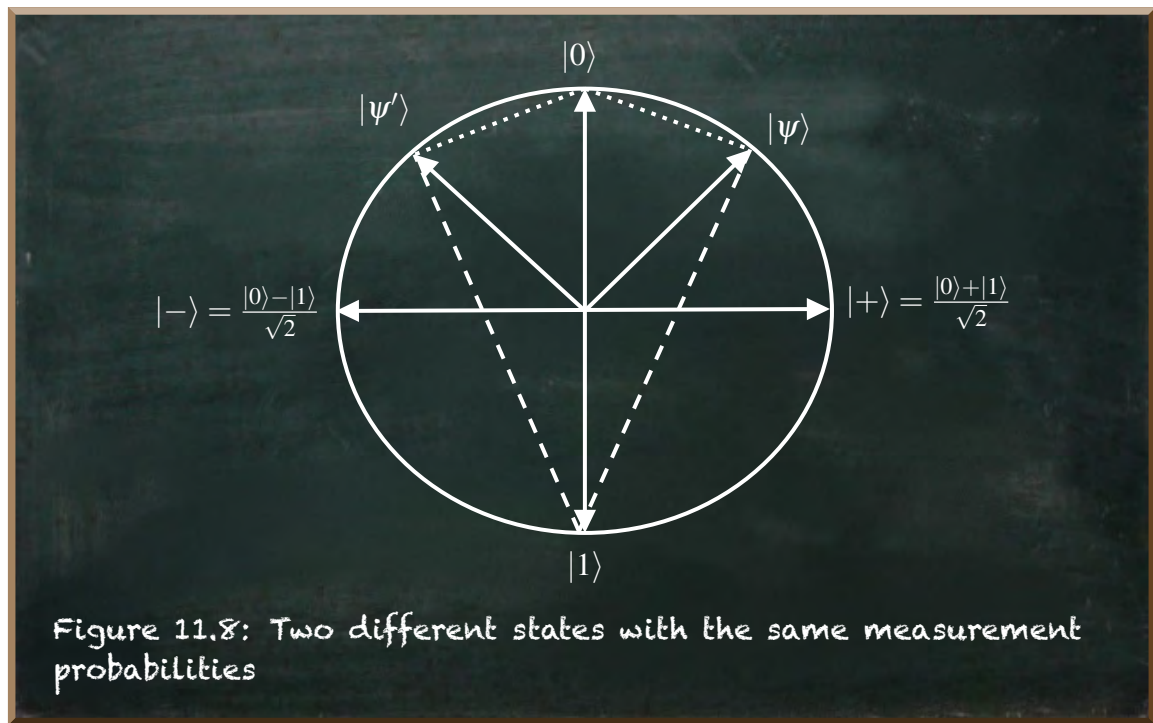
Regarded as waves, the two states $|\psi\rangle$ and $|\psi'\rangle$ denote two waves shifted by half their wavelength. The one's crest is the other's trough.

This notion of a qubit lets us distinguish two opposite phases. But how about all the other possible phases a qubit can be in? Similar to waves, the phase can be any arbitrary value. The only meaningful restriction we can formulate is that the phase repeats once it exceeds the wavelength. It may remind you of

the angle θ we used to rotate the qubit state vector and change its amplitudes.

We can represent the qubit phase as an angle ϕ (the Greek letter "phi") that spans a circle around the center, and that is orthogonal to the circle of the amplitudes. This circle uses another dimension.

In the following figure, the angle θ describes the probability of the qubit to result in $|0\rangle$ or $|1\rangle$ and the angle ϕ describes the phase the qubit is in.

Figure 11.9: The Bloch Sphere
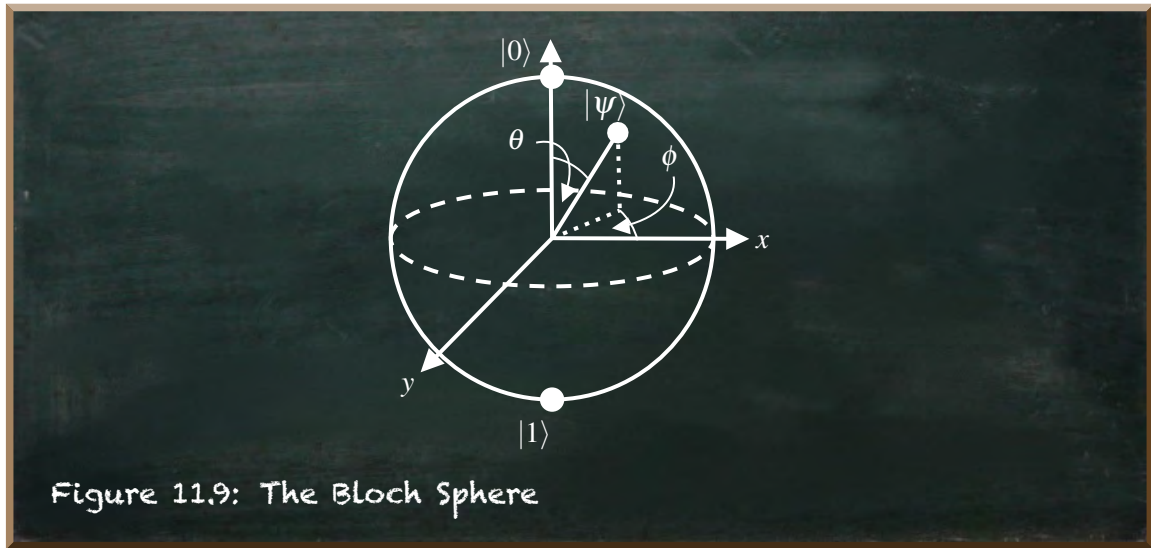
These two circles form a sphere around the center. This sphere is known as the Bloch Sphere.

The Bloch Sphere offers a visual reference of both the phase and the probabilities of measuring a qubit as either of the basis states $|0\rangle$ or $|1\rangle$. In this sphere, the angle θ that determines the measurement amplitudes revolves around the Y-axis. Correspondingly, the R_Y-gate we used thus far rotates the qubit state vector around this axis. It cuts the Z-axis in the basis states $|0\rangle$ and $|1\rangle$. If we don't apply a phase shift, it "lies" flat on the plane the X-axis spans. But once the qubit state has a different phase, it rises from this plane.

The angle φ that determines the phase revolves around the Z-axis. Thus, any change in the phase of a qubit does not affect the proximity to the Z-axis or any point on it, such as the basis states $|0\rangle$ and $|1\rangle$ that denote the top and the bottom of the Z-axis.

Therefore, the phase does not affect the measurement probabilities.

Mathematically, we could create a three-dimensional sphere using three-

dimensional vectors, such as $v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix}$. Then, however, the third dimension would matter for the measurement probability. The qubit phase would be equivalent to the other two dimensions.

Instead, we achieve a three-dimensional sphere with two-dimensional vectors

$$\psi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

for α and β are complex numbers. A complex number is a number that can be expressed in the form $a + b \cdot i$, where a is the real part and $b \cdot i$ is the imaginary part. i represents the imaginary unit satisfying the equation $i^2 = 1$. Because no real number satisfies this equation, the parts a and $b \cdot i$ are independent of each other. Therefore, we don't represent a complex number as a line, as we represent real numbers. But a complex number forms a plane. A complex number is two-dimensional.

Figure 11.10: Complex numbers
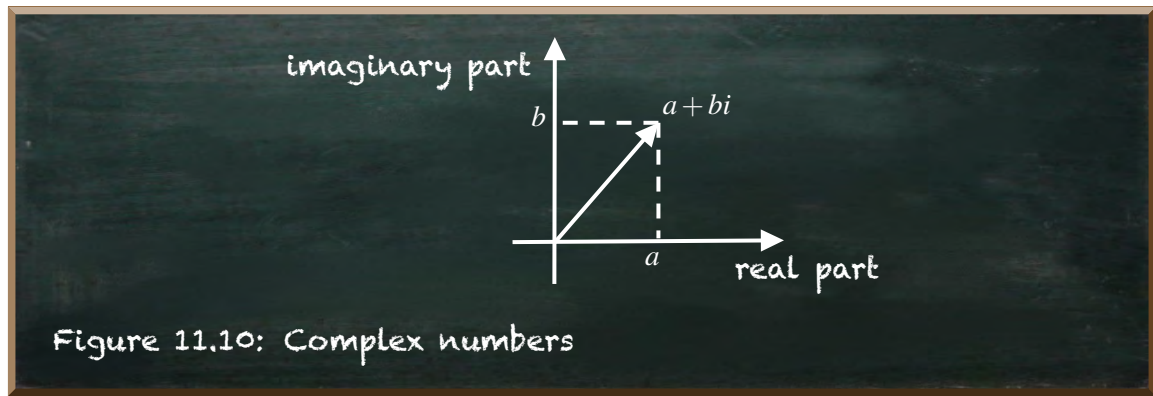
Complex numbers extend the concept of the measurement probabilities α and β. But to be useful for our purposes, we still require the relation between α and β to be normalized by the equation $|\alpha|^2 + |\beta|^2 = 1$.

For α and β are complex numbers ($\alpha, \beta \in \mathbb{C}$), it becomes mandatory to take the absolute before we sum their squares. Simple squares would not suffice anymore because $i^2 = -1$.

In section 3.2, we introduced the polar form of the qubit state in which the angle θ controls the probabilities of measuring the qubit in either state 0 or 1.

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix}$$

We only need to apply the phase to one of the two amplitudes. Consider the

state $|-\rangle$ for instance. We can represent it in two ways.

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

If we added the $-$ to both parameters, we would result in state $|+\rangle$.

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

Per convention, we put the phase to the amplitude of state $|1\rangle$.

But how do we express the phase? We know it is a function of the angle ϕ. Let's insert this into the function.

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + f(\phi) \cdot sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ f(\phi) \cdot sin\frac{\theta}{2} \end{bmatrix}$$

In fact, we know some characteristics of $f(\phi)$. For $\phi = 0$, we expect no change whatsoever. Since we multiply $f(\phi)$ with $\beta = sin\frac{\theta}{2}$, $f(\phi)$ must equal 1 in this case. Therefore, $f(0) = 1$.

Further, with a shift by a full wavelength, we end up in the identical phase. Due to the normalization, the qubit wavelength is 2π – the perimeter of a circle. Thus, $f(2\pi) = 1$. With a shift by half the wavelength, we effectively negate the original wave. Therefore, $f(\pi) = -1$.

The following figure depicts $f(\phi)$ graphically. We can see that it's output is $f(\phi) = cos(\phi) + i \cdot sin(\phi)$.

Figure 11.11: Characterstics of f(phi)
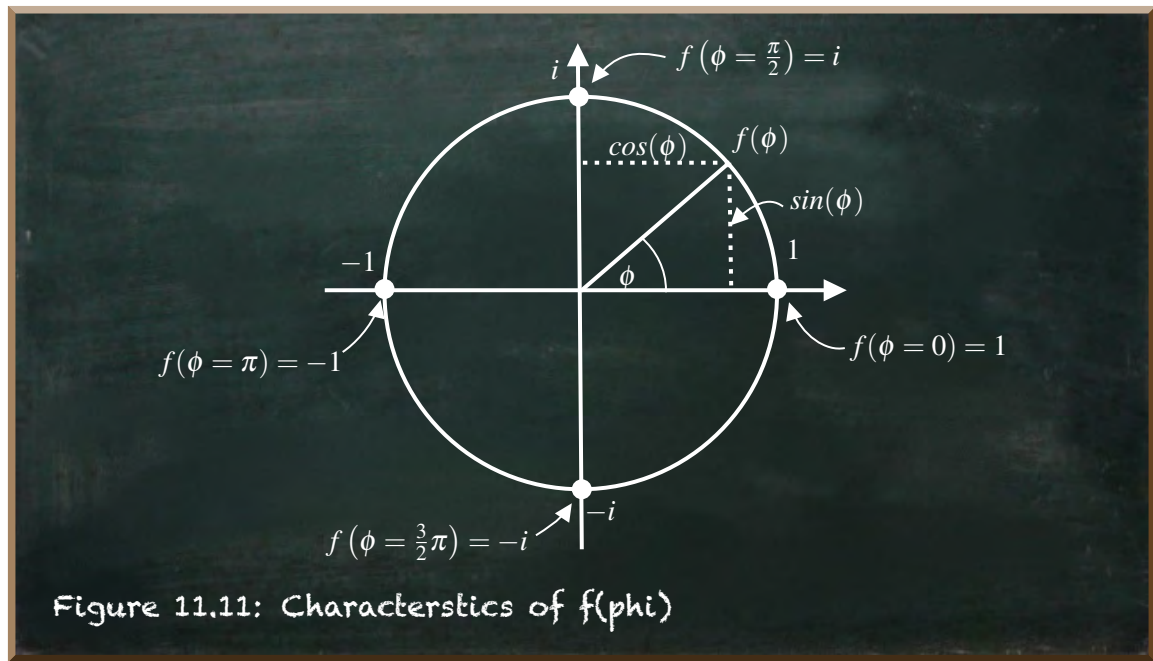
Leonhard Euler discovered that we could describe such a function by $f(\phi) = cos(\phi) + i \cdot sin(\phi) = e^{i\phi}$. This equation is known as Euler's formula.

And this gives us the refined qubit state vector of

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{i\phi}sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ e^{i\phi}sin\frac{\theta}{2} \end{bmatrix}$$

"But doesn't $e^{i\phi}$ matter for the measurement probabilities?"

The part of the amplitude e^{ix} makes it a complex number. It has a real part and an imaginary part. When we calculate the measurement probabilities, we square the absolute of the amplitudes.

So, let's look at what squaring the absolute of a complex number means. Let's say we have a complex number $c = a + bi$

The absolute square of a complex number c, is calculated by multiplying it by its complex conjugate. The conjugate of a complex number (written as c^*) is the number with the sign of the imaginary part reversed. In our case, this is $c^* = a - bi$. This is known as the squared norm. It is defined as $|c|^2 = c^*c$

We can transform this formula as follows by applying the third binomial rule and by inserting the definition of $i^2 = -1$.

$$|c|^2 = c^*c = (a - bi)(a + bi) = a^2 - (bi)^2 = a^2 - (bi)^2 = a^2 + b^2$$

Now, if the complex number is given by $c = e^{ix} = cos(\phi) + i \cdot sin(\phi)$, we can see that $a = cos(\phi)$ and $b = sin(\phi)$.

Per Pythagorean" identities (use the rectangular triangle), the sum of the squares of $sin(\phi)$ and $cos(\phi)$ is 1. Therefore, we can state

$$|e^{ix}|^2 = cos(\phi)^2 + sin(\phi)^2 = 1$$

So whatever the phase is, its absolute square is 1. 1 is the neutral element of multiplication. Whatever we multiply by 1 remains unchanged.

Complex numbers allow us to add the dimension of the qubit phase without touching the measurement probabilities. While this affects the amplitude of the qubit state, it does not affect the measurement probability.

Let's explore different states in the Bloch Sphere. We use the Python library `qutip` for it allows us to easily play around with other visualizations of the Bloch Sphere (see the documentation for details).

The `Bloch`-class of `qutip` works with cartesian coordinates. Let's write a function that allows us to use the angles θ and ϕ. If we interpret θ and ϕ as spherical coordinates (with the radius is 1) we can plot any qubit state on the surface of the Bloch Sphere.

Listing 11.1: Convenience function to plot a Bloch Sphere using spherical coordinates

```python
from math import pi, sin, cos
import matplotlib.pyplot as plt
import matplotlib
from qutip import Bloch

def plot_bloch_vector_spherical(theta, phi):
  b = Bloch()
  b.clear()
  vec = [sin(theta)*cos(phi),sin(theta)*sin(phi),cos(theta)]
  b.add_vectors(vec)
  return b.show()
```

We start with importing all the libraries and functions we use (lines 1-4). The function `plot_bloch_vector_spherical` takes two parameters, `theta` and `phi`, our two angles.

Plotting a Bloch Sphere is quite simple. We initialize the `Bloch` class (line 7), clear any old data (line 8), add the vector we want to show (line 10), and show the plot (line 11).

We calculate the coordinates of the vector we want to show based on the angles (line 9).

> ⚠ The Bloch Sphere usually shows the Y-axis to the right-hand side. This differs from the two-dimensional perspective we used thus far. In the two-dimensional figures, we plotted the X-axis to the right.

Let's start with the state we already know as $|+\rangle$

Listing 11.2: The qubit state |+>

```python
plot_bloch_vector_spherical(pi/2, 0)
```

Figure 11.12: The state |+> on the Bloch Sphere
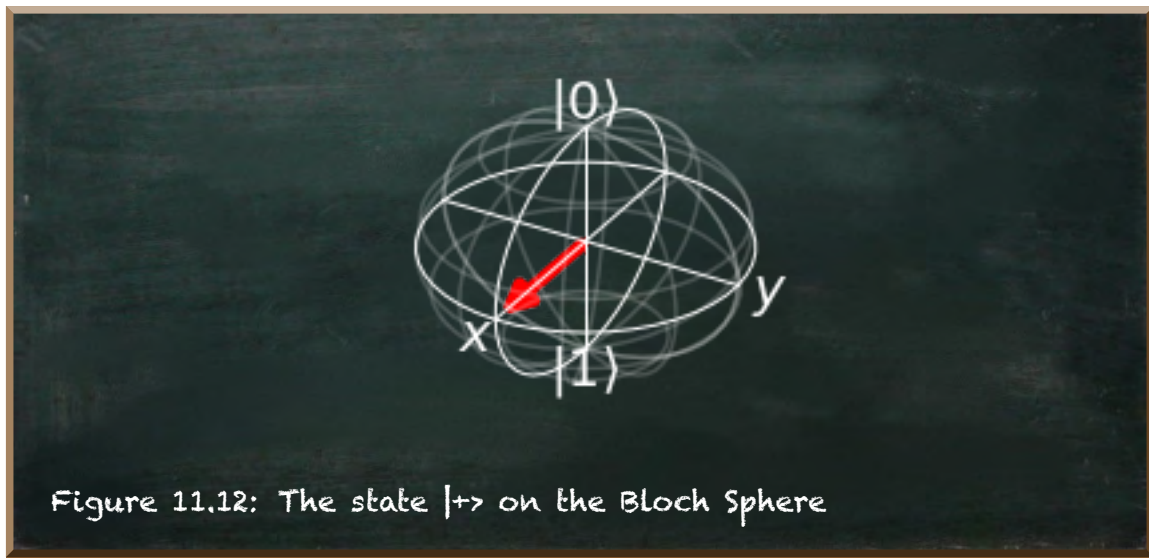
And the state $|-\rangle$:

Listing 11.3: The qubit state |->

```
1  plot_bloch_vector_spherical(−pi/2, 0)
```


Figure 11.13: The state |-> on the Bloch Sphere
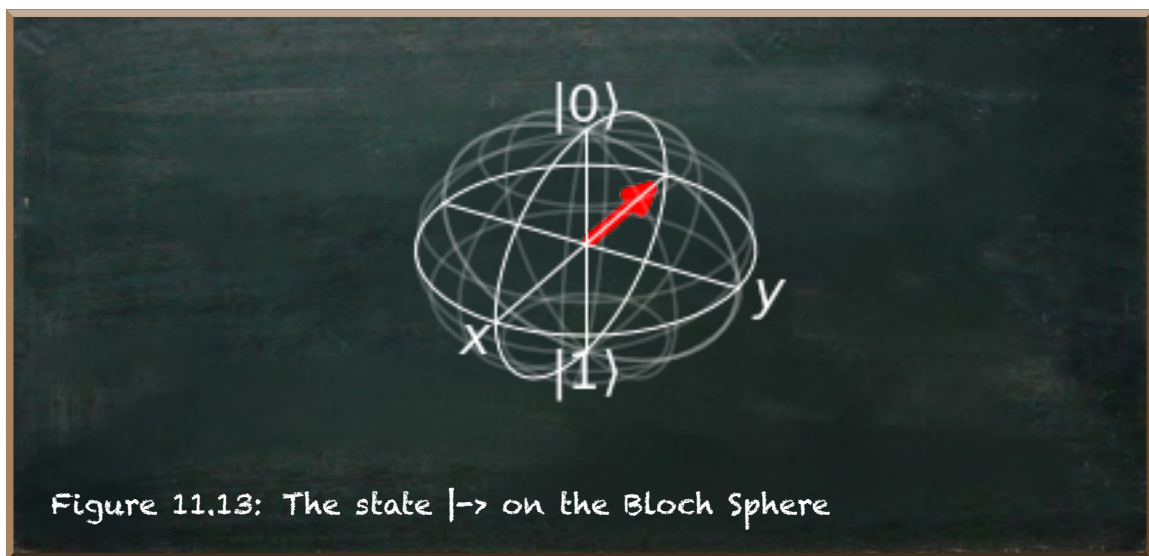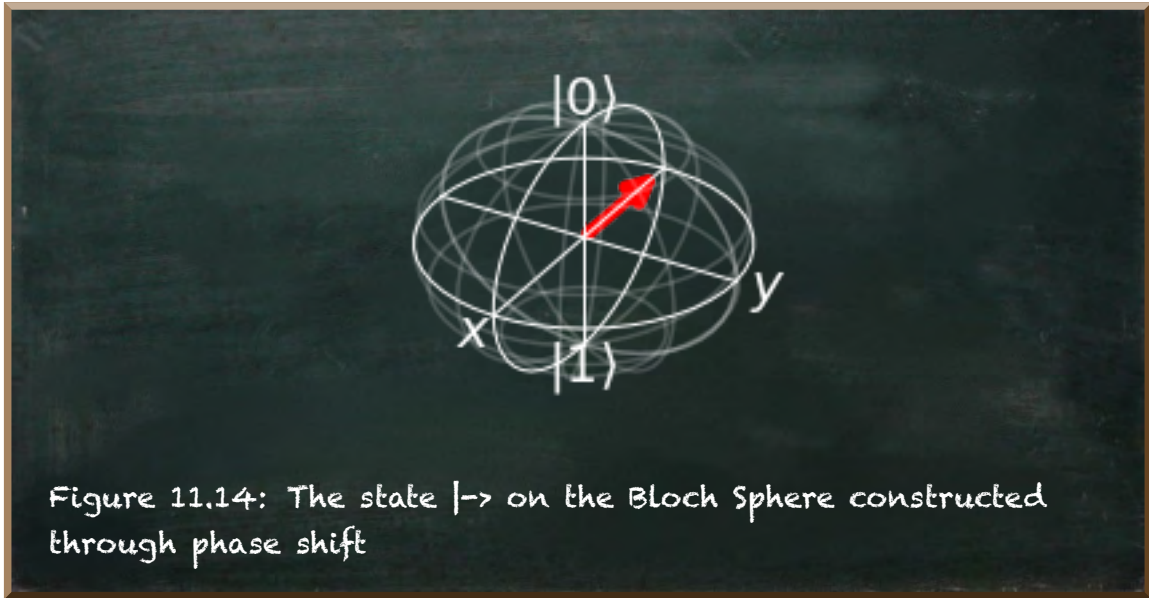
But we can construct the state $|-\rangle$ by applying a phase shift on the state $|+\rangle$ by half a circle π.

Listing 11.4: The constructed qubit state |->

```
1  plot_bloch_vector_spherical(pi/2, pi)
```


Figure 11.14: The state |-> on the Bloch Sphere constructed through phase shift

Phase shifts affect only qubits in one of the basis states. Turns around the Z-axis don't affect a state vector that lies on the Z-axis.

Listing 11.5: Qubit in state |0>

```
1  plot_bloch_vector_spherical(0, pi/2)
```


Figure 11.15: The phase does not affect qubits in a basis state
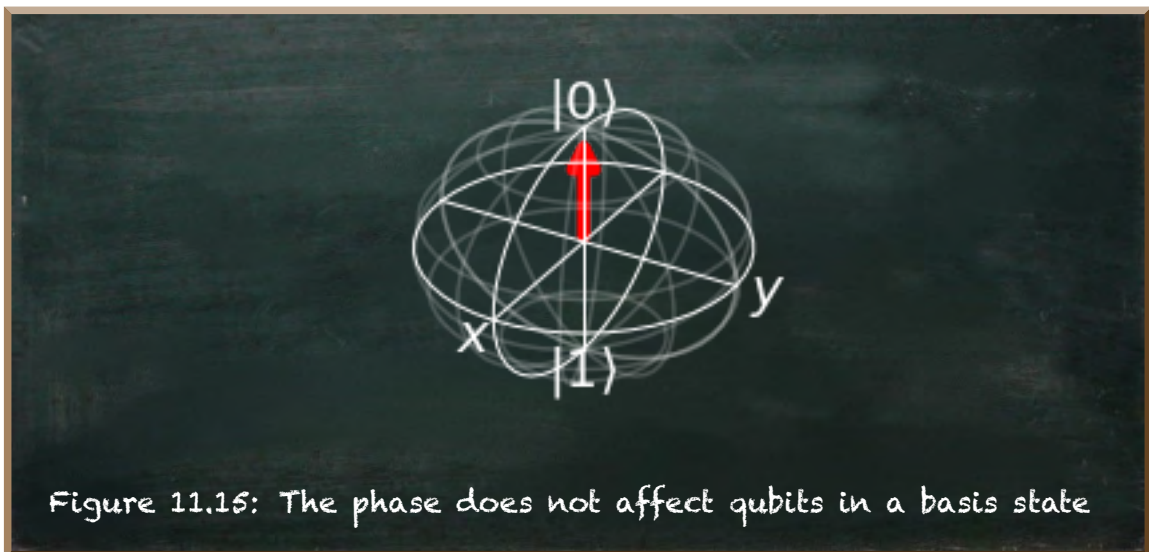
11.2 Visualize The Invisible Qubit Phase

The phase of a qubit is the decisive factor when we work with amplitudes. Even though we can't measure the phase directly, it is not invisible. Once, we can use our simulator to visualize it. Second, we can use math to describe it precisely.

When we look at the Bloch Sphere, states with a positive amplitude reside at the front-side of the X- and Z-axes.

Figure 11.16: States with positive amplitudes
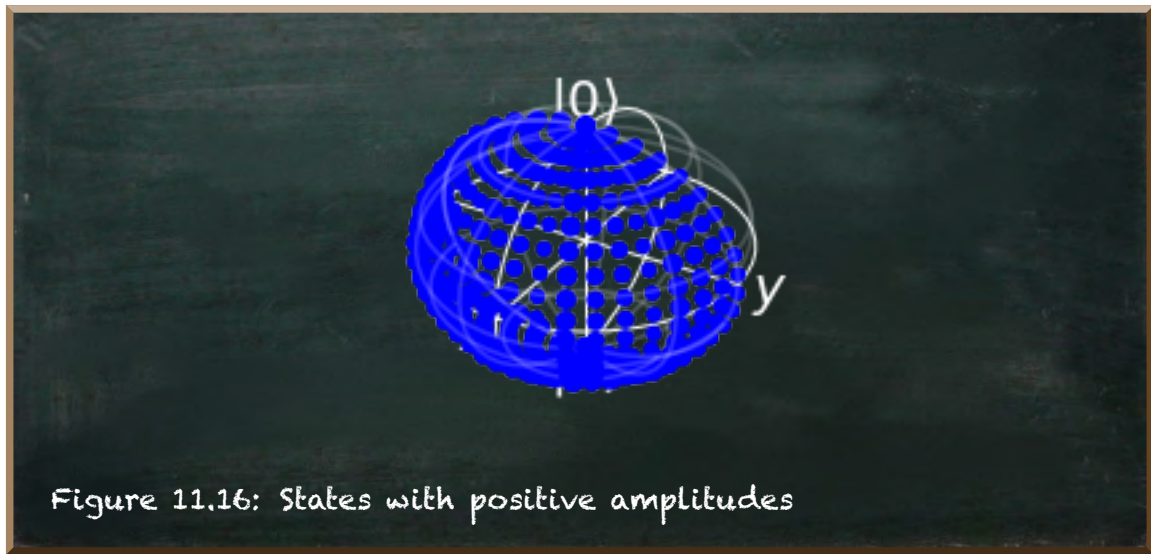
By contrast, states with a negative amplitude reside at the back-side of the Bloch Sphere.

Figure 11.17: States with negative amplitudes
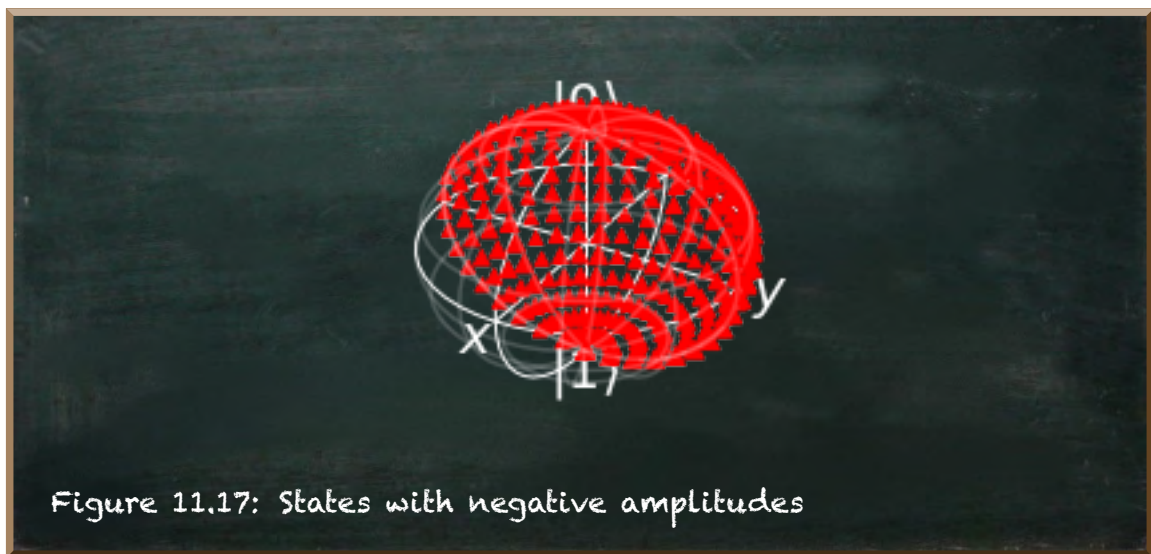
Let's consider some simple states first. If we rotate the qubit state vector around the Y-axis by the angle θ, we get the following states. For $0 < \theta < \pi$, we can say the phase is positive (blue). For $\pi < \theta < 2\pi$, the phase is negative (red).

Figure 11.18: States spanned by theta
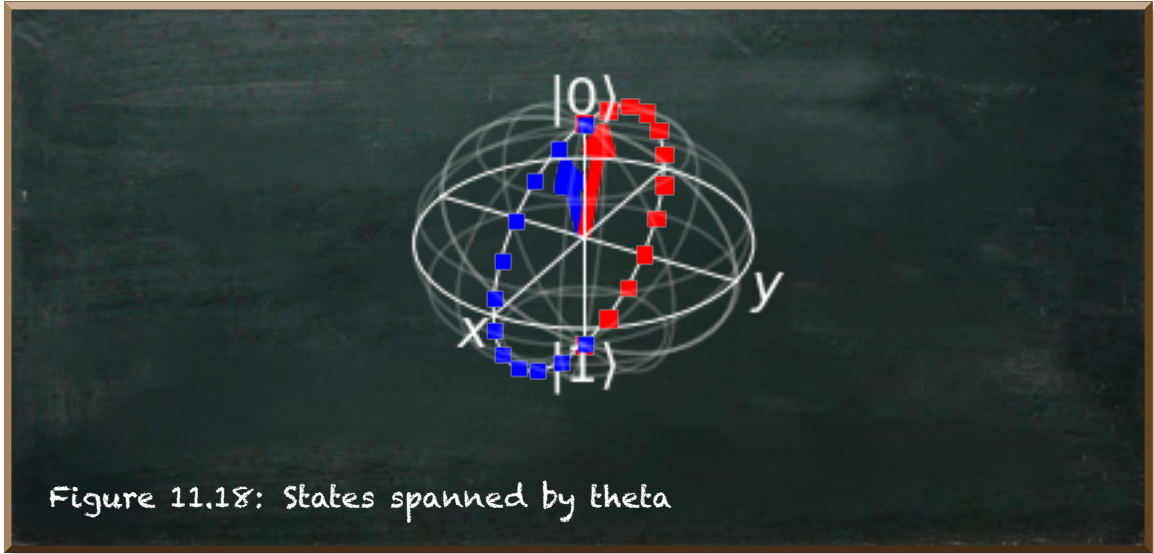
The figure shows that these states lie on the plane the X- and the Z-axes span. The vector θ specifies the rotation around the Y-axis. Therefore, we call the corresponding gate R_Y-gate.

Two vectors mirroring each other on the Z-axis have the same measurement probability, such as the two vectors depicted in the figure. They share the same measurement probability, but their phase differs.

11.2.1 The Z-gate

The Z-gate reflects the state of a qubit on the Z-axis. It has the similar effect of the X-gate that reflects the state on the X-axis. A reflection on the X-axis affects the resulting measurement probabilities because it changes the proximities to the ends of the Z-axis ($|0\rangle$ and $|1\rangle$). But it leaves untouched the phase. Contrariwise, a reflection on the Z-axis flips the phase but leaves the measurement probabilities unchanged.

The following equation denotes the transformation matrix of the Z-gate.

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

The Z-gate turns a qubit in state $|+\rangle$ into state $|-\rangle$. The states $|+\rangle$ and $|-\rangle$ reside

on the *X*-axis. Mathematically, the following equation describes this transformation.

$$HZ|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

Let's have a look at the effect of the *Z*-gate programmatically.

Qiskit lets us effortlessly show the qubit state vectors. First, we define and prepare our quantum circuit as usual (lines 4-11). In this case, we apply a single Hadamard gate on qubit 0 (line 7) and an additional *Z*-gate on qubit 1 (lines 10-11).

We execute the circuit using the `'statevector_simulator'`-backend. But instead of obtaining the counts from the execution results, we call the `get_statevector ()`-function (line 13). The output is an array of the state vectors, like `array([ 0.5+0.j, 0.5+0.j, −0.5+0.j, −0.5+0.j])`.

We can feed this array into the function `plot_bloch_multivector` (line 14) we imported from `qiskit.visualization` (line 2). As a result, we get a Bloch Sphere representation of each qubit.

Listing 11.6: Putting qubit 0 into state |+> and qubit 1 into state |->

```
1  from qiskit import QuantumCircuit, Aer, execute
2  from qiskit.visualization import plot_bloch_multivector
3
4  qc = QuantumCircuit(2)
5
6  # put qubit 0 into state |+>
7  qc.h(0)
8
9  # put qubit 1 into state |−>
10 qc.h(1)
11 qc.z(1)
12
13 out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
14 plot_bloch_multivector(out)
```

Figure 11.19: Putting qubit 0 into state |+> and qubit 1 into state |->
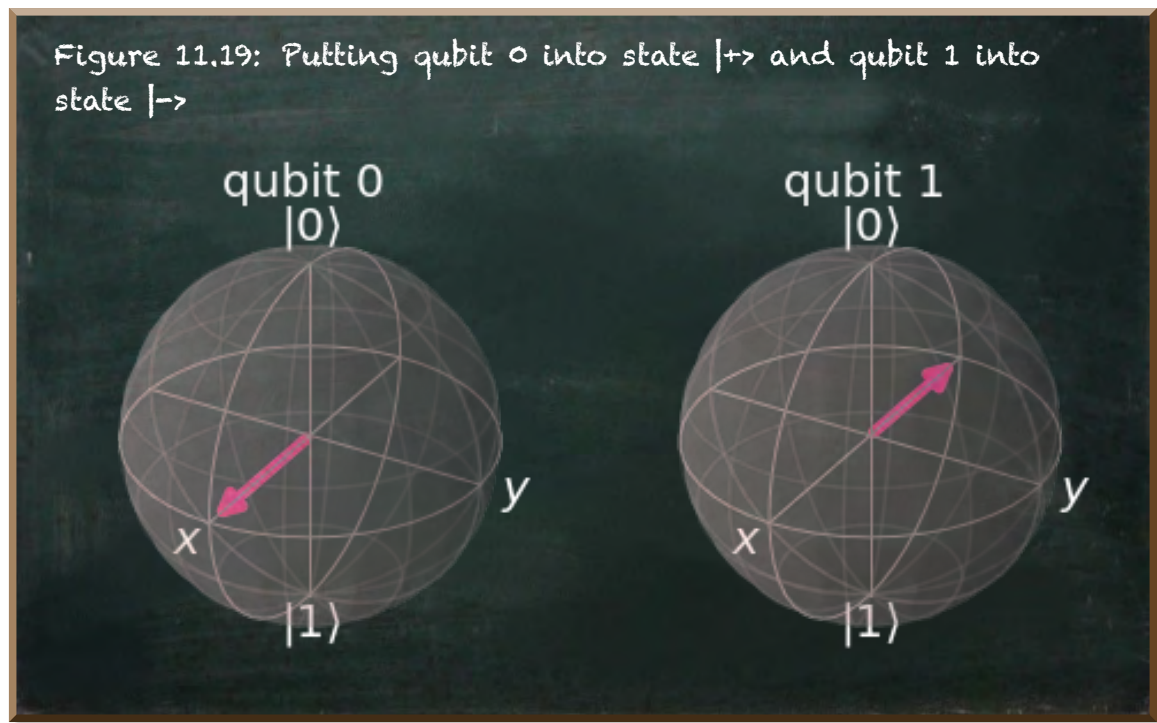
We can see both vectors reside on the Y-axis. Their amplitudes yield the same measurement probability of 0.5 each. The $|+\rangle$ state-vector points to the positive direction (front) of the X-axis whereas the $|-\rangle$ state-vector points to the negative side (back).

Let's look at another example. Let's rotate the state by a small θ around the Y-axis before we apply the Z-gate.

Listing 11.7: Reflection on the Z-axis

```
1  from math import pi
2  qc = QuantumCircuit(2)
3  qc.ry(pi/12, 0)
4  qc.ry(pi/12, 1)
5  qc.z(1)
6
7  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
8  plot_bloch_multivector(out)
```

Figure 11.20: The Z-gate reflects the state on the Z-axis
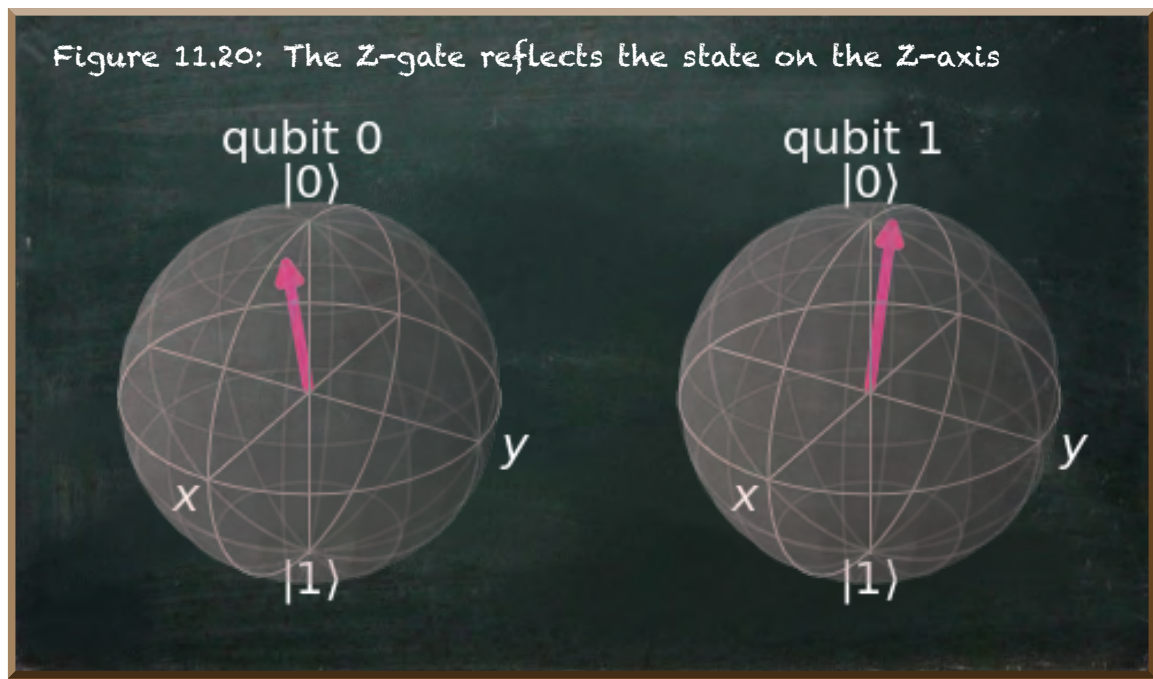
This second example emphasizes the reflection on the axis. It is not a reflection on the center of the coordinate system.

The Z-gate has a notable property. It does not affect qubits in state $|0\rangle$ and $|1\rangle$ because these two states reside on the Z-axis.

Listing 11.8: Apply the Z-gate on a qubit in state |0>

```
1  qc = QuantumCircuit(2)
2
3  # qubit 0 remains in state |0>
4  qc.i(0)
5
6  # Apply the Z—gate on qubit 1
7  qc.z(1)
8
9  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
      get_statevector()
10 plot_bloch_multivector(out)
```

Figure 11.21: Effect of the Z-gate on a qubit in state |0⟩
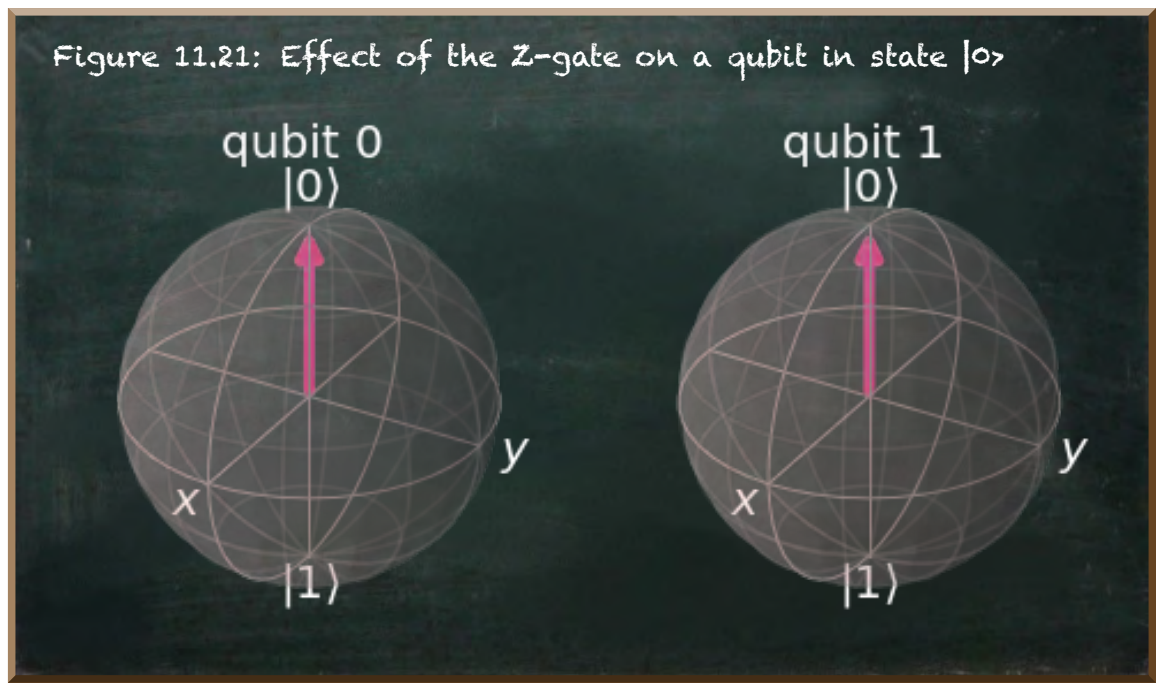
When we look at the math, the situation is straightforward for state $|0\rangle$.

$$Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

We get a confirmation of the visualization. But how about state $|1\rangle$.

$$Z|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

The result of the equation is ambiguous. The resulting state vector $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$ is different from the original vector $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

The phase shift seems to make a difference. But the visualization shows us that the two vectors are identical.

Listing 11.9: Apply the Z-gate on a qubit in state |1>

```
1  qc = QuantumCircuit(2)
2
3  # a qubit in state |1>
4  qc.x(0)
5
6  # The effect of the Z-gate on state |1>
7  qc.x(1)
8  qc.z(1)
9
10 out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
11 plot_bloch_multivector(out)
```


Figure 11.22: Effect of the Z-gate on a qubit in state |1>
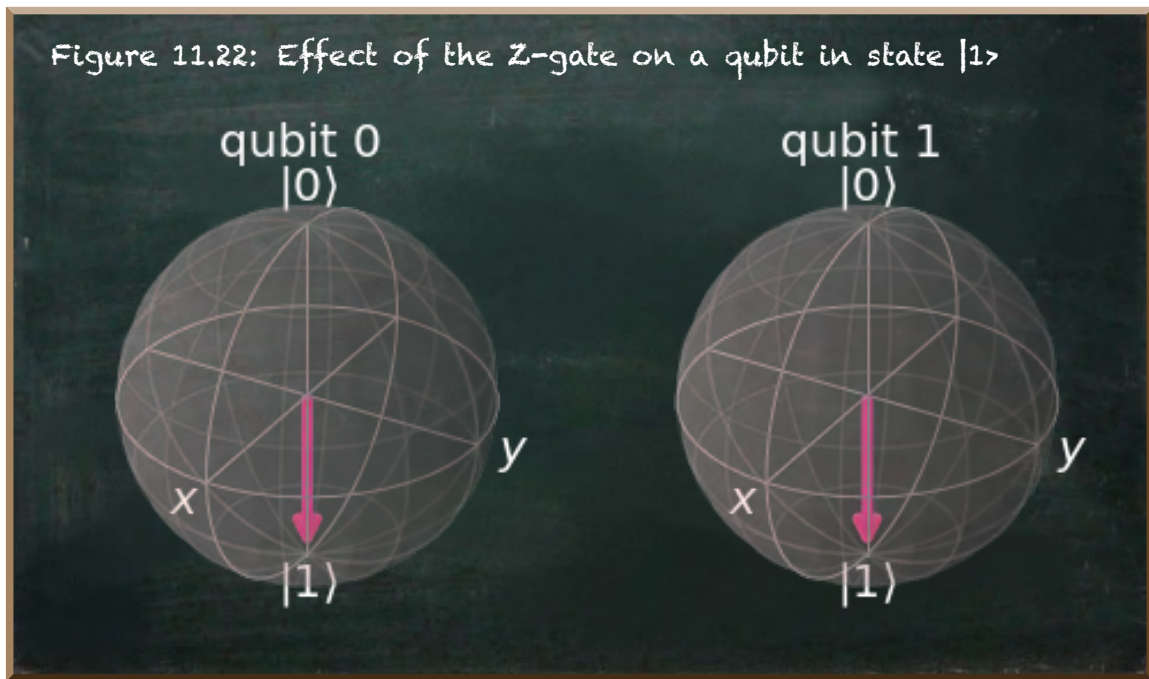
The minus sign does not seem to matter. But what happens if we put this state into superposition?

Listing 11.10: Apply the ZH-gates on a qubit in state |1>

```
 1  qc = QuantumCircuit(2)
 2
 3  # Apply H—gate on a qubit in state |1>
 4  qc.x(0)
 5  qc.h(0)
 6
 7  # Apply ZH—gates on a qubit in state |1>
 8  qc.x(1)
 9  qc.z(1)
10  qc.h(1)
11
12  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
        get_statevector()
13  plot_bloch_multivector(out)
```


Figure 11.23: Effect of the ZH-gates on a qubit in state |1>
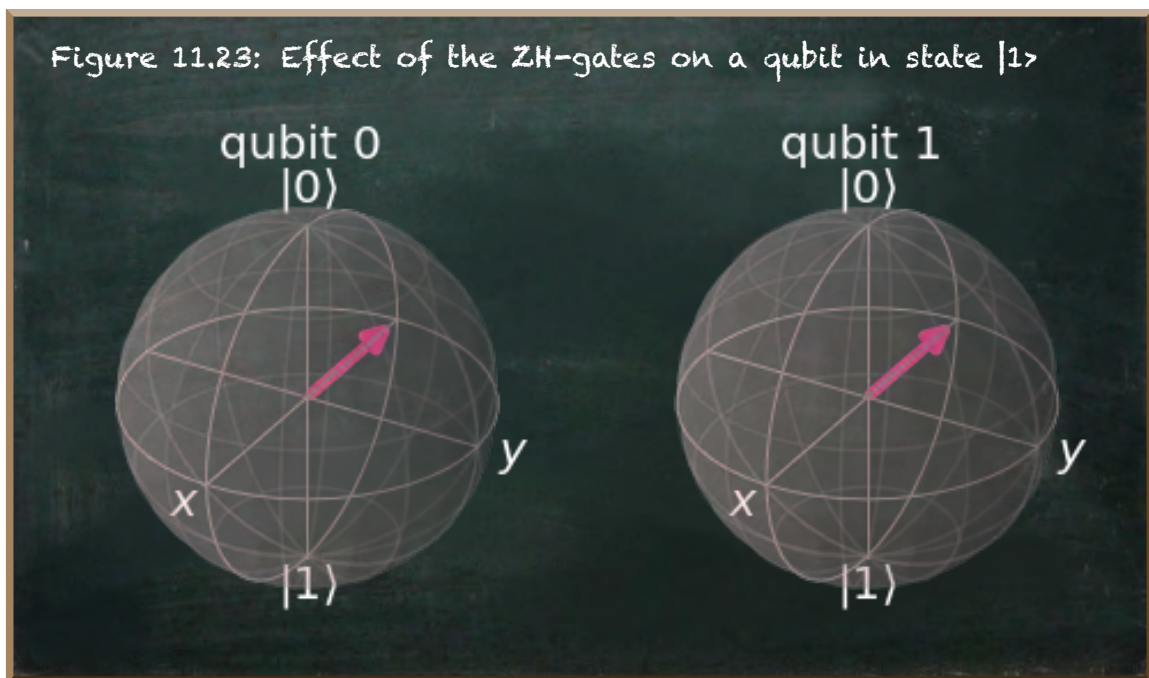
Again, the visualization tells us that there is no difference between these states. So, let's have another look at the math. If we apply the Hadamard gate on state $|1\rangle$ it results in state $|-\rangle$.

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

If we apply the *Z*-gate first, the $-$-sign jumps from the amplitude of $|0\rangle$ to the amplitude of $|1\rangle$.

$$ZH|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \frac{-|0\rangle+|1\rangle}{\sqrt{2}} = |-\rangle$$

Remember section 5.2, where elaborate the inability to distinguish between $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ and $\frac{-|0\rangle+|1\rangle}{\sqrt{2}}$.

As we just saw, there's no difference in the resulting state when we apply other quantum gates on either one state. The states $\alpha|0\rangle$ and $\beta|1\rangle$ form a shared quantum state. For the resulting qubit state vector, it does not matter whether α or β contains the phase.

But we're about to start working with the phase. So, let's be a little more meticulous. Qiskit provides another visualization, the "qsphere" representation of a quantum state. In this representation, the size of the points is proportional to the probability of the corresponding term in the state and the color represents the phase.

Listing 11.11: create state (|0>-|1>)/sqrt(2)

```
from qiskit.visualization import plot_state_qsphere

# Create a quantum circuit with one qubit
qc = QuantumCircuit(1)

# create state (|0>−|1>)/sqrt(2)
qc.h(0)
qc.z(0)

out = execute(qc,Aer.get_backend('statevector_simulator')).result().
    get_statevector()
plot_state_qsphere(out)
```

Figure 11.24: The state (|0>-|1>)/sqrt(2)
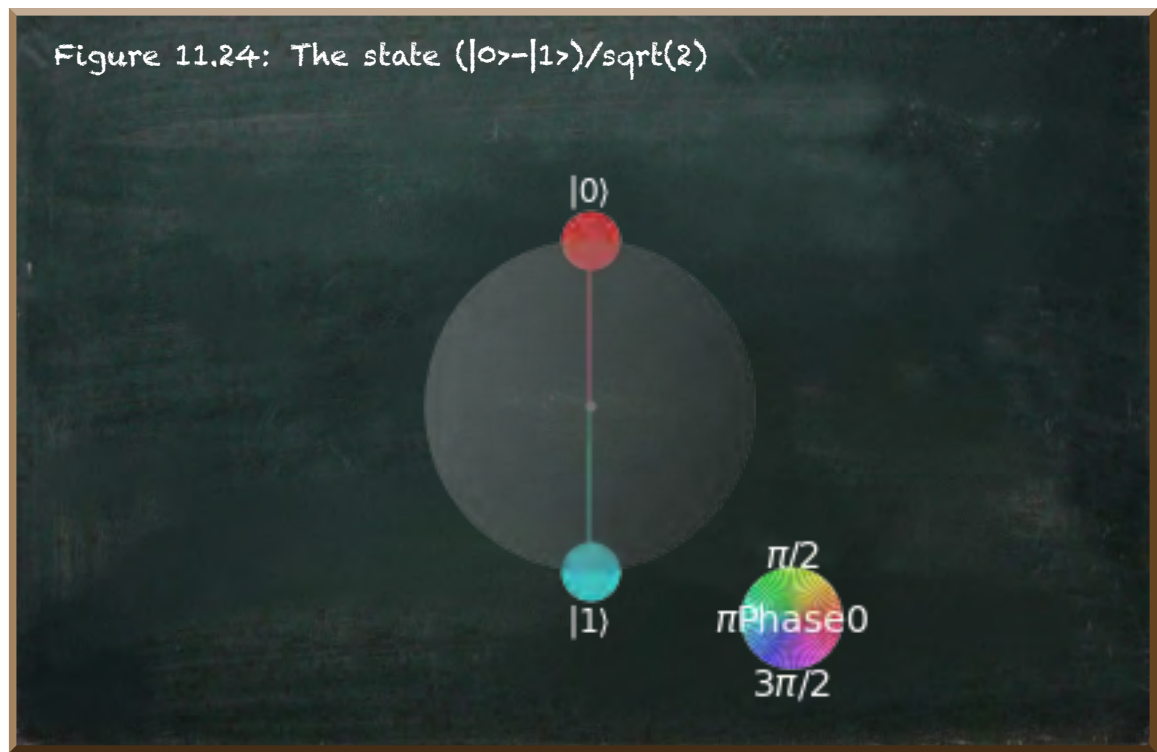
The figure above shows the qubit in state $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$. We first apply the Hadamard gate (line 5) before using the Z-gate (line 6). The phase applies to the amplitude of the state $|1\rangle$. Thus, we only see state $|1\rangle$ shown in turquoise. State $|0\rangle$ remains red. Both circles have the same size, for both states have the same measurement probability.

Now, let's apply the Z- and Hadamard-gates on a qubit in state $|1\rangle$ as we did before.

Listing 11.12: Apply the ZH-gates on a qubit in state |1>

```
1  qc = QuantumCircuit(1)
2
3  # Apply ZH-gates on a qubit in state |1>
4  qc.x(0)
5  qc.z(0)
6  qc.h(0)
7
8  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
9  plot_state_qsphere(out)
```

Figure 11.25: The state (-|0>+|1>)/sqrt(2)
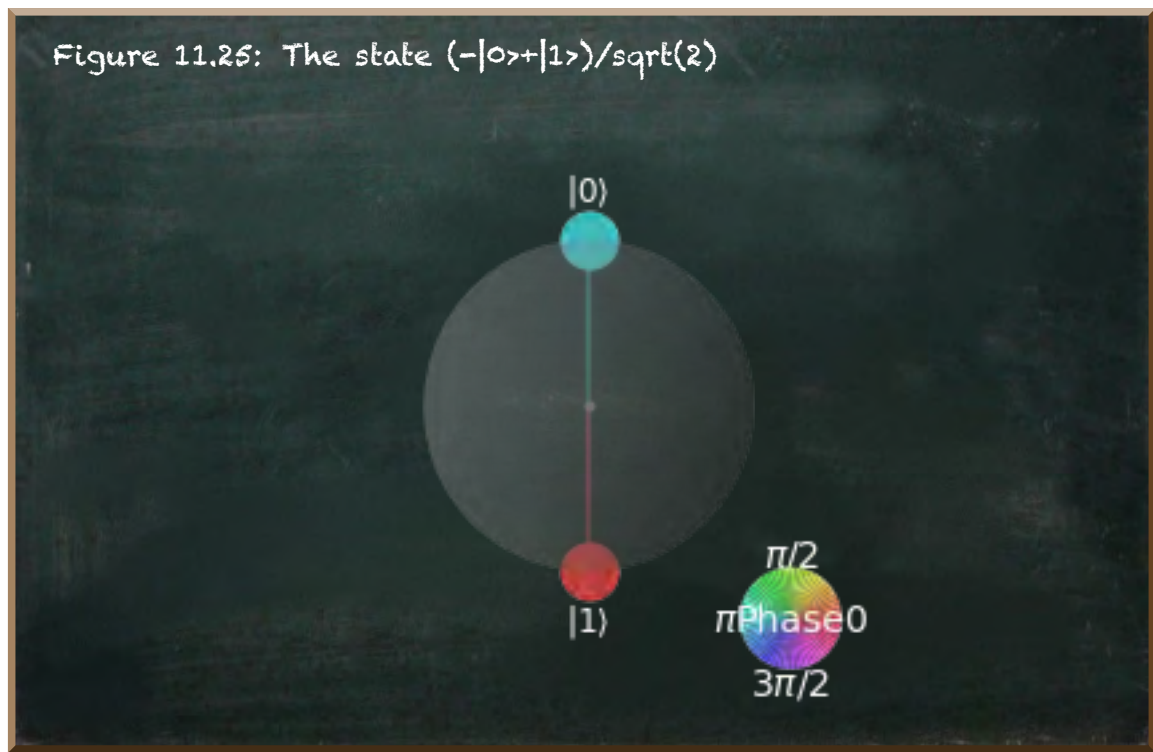
This circuit results in state $\frac{-|0\rangle+|1\rangle}{\sqrt{2}}$. The colors indicate we applied the phase shift on state $|0\rangle$.

So, we can apply the phase to any one of the two amplitudes, α or β. It is important that we don't apply it to both amplitudes because this would effectively revert the overall phase again.

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

11.2.2 Multi-Qubit Phase

What about if we have multiple qubits? The following equation denotes the state of a two-qubit system.

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}$$

The two-qubit system can be in four different states. Each state has an amplitude, too.

We already specified a two-qubit system above to show two Bloch Spheres

side by side. Qubit 0 is in state $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ and qubit 1 is in state $\frac{-|0\rangle + |1\rangle}{\sqrt{2}}$.

Let's have a look at the phases of the four states of this system.

Listing 11.13: Show the phases of a two-qubit system

```
 1 qc = QuantumCircuit(2)
 2
 3 # put qubit 0 into state (|0>−|1>)/sqrt(2)
 4 qc.x(0)
 5 qc.h(0)
 6
 7 # put qubit 1 into state (−|0>+|1>)/sqrt(2)
 8 qc.x(1)
 9 qc.z(1)
10 qc.h(1)
11
12 out = execute(qc,Aer.get_backend('statevector_simulator')).result().
      get_statevector()
13 plot_state_qsphere(out)
```


Figure 11.26: The phases of a two-qubit system
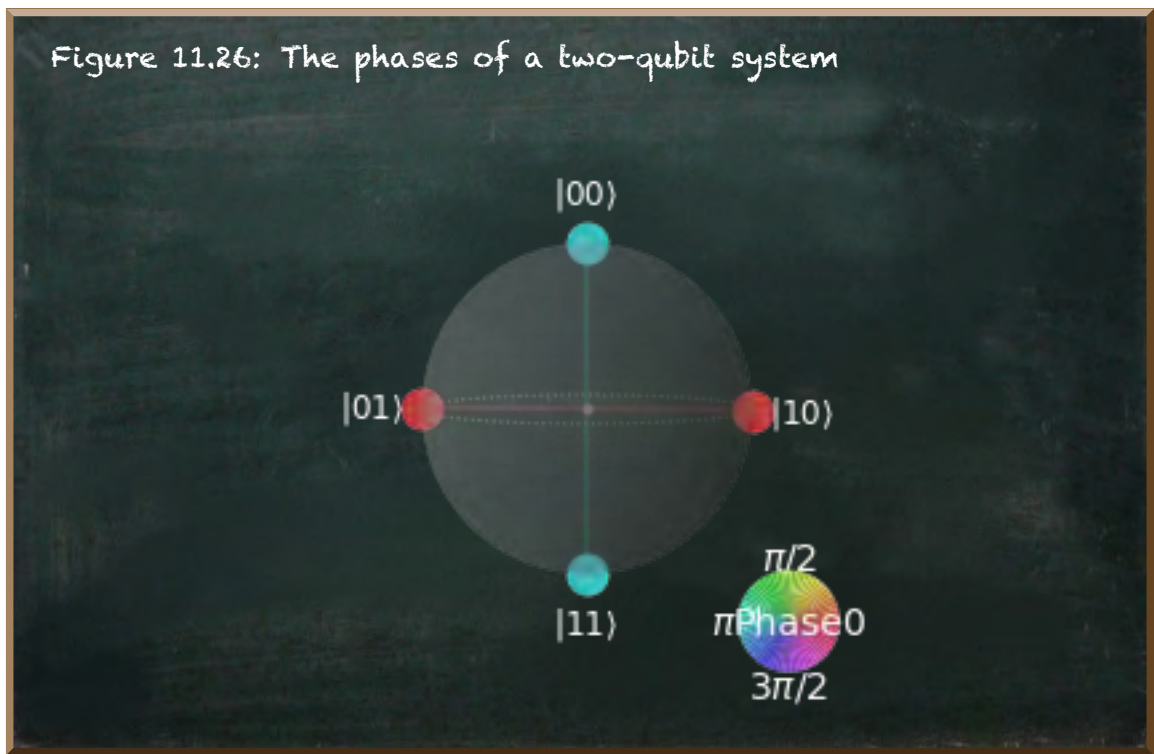
We see two states with a shifted phase, $|00\rangle$ and $|11\rangle$. In this notation, we read

the qubits from the right (qubit at position 0) to the left (qubit at position 1). The state $|11\rangle$ gets the phase shift for the qubit 0 has its phase in the amplitude of $|1\rangle$ (as in $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$). Accordingly, state $|00\rangle$ gets the phase shift for the qubit 1 has its phase in the amplitude of $|0\rangle$ (as in $\frac{-|0\rangle+|1\rangle}{\sqrt{2}}$).

Even though this makes sense, it is not the whole truth. Let's see what happens if both qubits are in state $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$. We would expect to see a shift in the phases $|01\rangle$ and $|10\rangle$.

Listing 11.14: Phases of two qubits in state (|0>-|1>)/sqrt(2)

```
1 qc = QuantumCircuit(2)
2 # put qubit 0 into state (|0>−|1>)/sqrt(2)
3 qc.x(0)
4 qc.h(0)
5 # put qubit 1 into state (|0>−|1>)/sqrt(2)
6 qc.x(1)
7 qc.h(1)
8 out = execute(qc,Aer.get_backend('statevector_simulator')).result().
      get_statevector()
9 plot_state_qsphere(out)
```
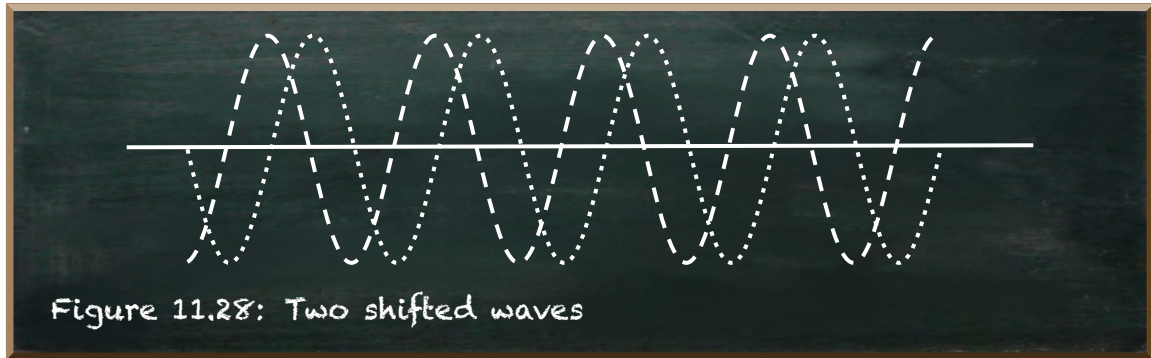

Figure 11.27: Phases of two qubits in state (|0>-|1>)/sqrt(2)
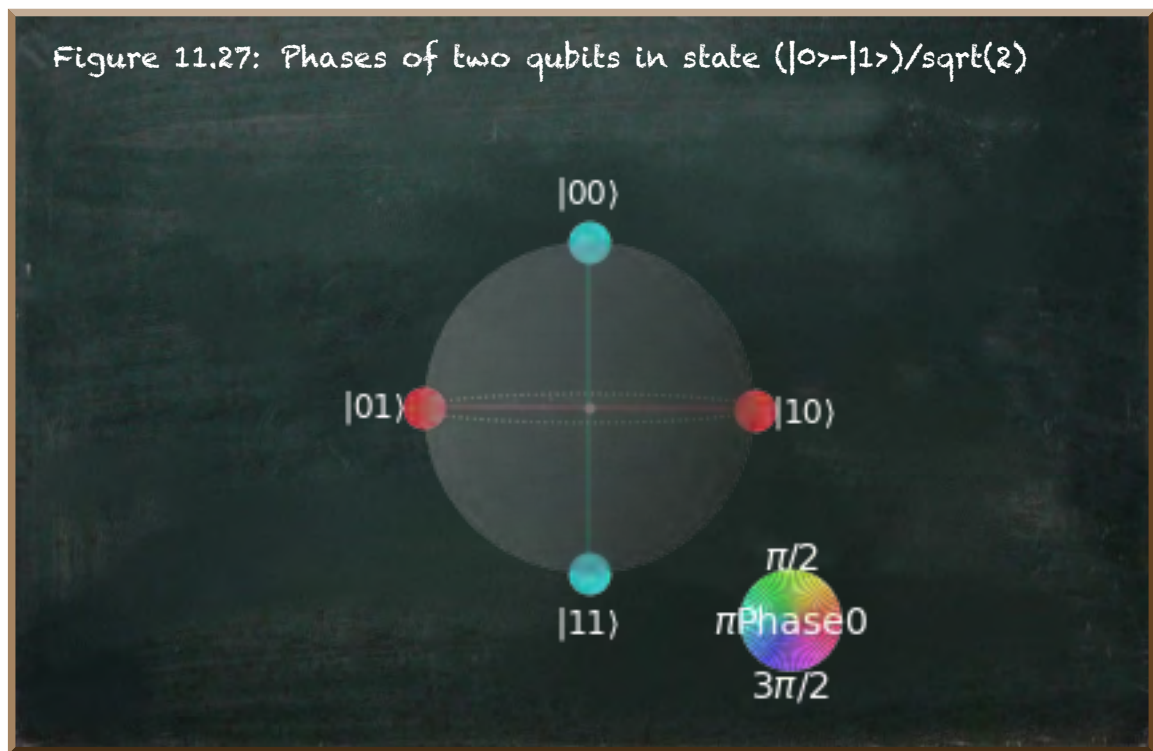
Deviating from our expectation, we see the phase shift in the states $|00\rangle$ and $|11\rangle$. The simple reason is phases are relative. We have two similar yet shifted waves. But how could you tell which of the two waves is the original?

Figure 11.28: Two shifted waves

Therefore, we need to settle with the fact we can't tell the difference between $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ and $\frac{-|0\rangle + |1\rangle}{\sqrt{2}}$. Both states are the same: $|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{-|0\rangle + |1\rangle}{\sqrt{2}}$.

The ability to distinguish between these two states when we use a single qubit is spurious. It already disappears when we add a second qubit.

The following circuit creates the state $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ on a single qubit.

Listing 11.15: create state (|0>-|1>)/sqrt(2) in a single-qubit circuit

```
1  # Create a quantum circuit with one qubit
2  qc = QuantumCircuit(1)
3
4  # create state (|0>-|1>)/sqrt(2)
5  qc.h(0)
6  qc.z(0)
7
8  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
9  plot_state_qsphere(out)
```

Figure 11.29: Phase of state (|0⟩-|1⟩)/sqrt(2) in a single-qubit circuit
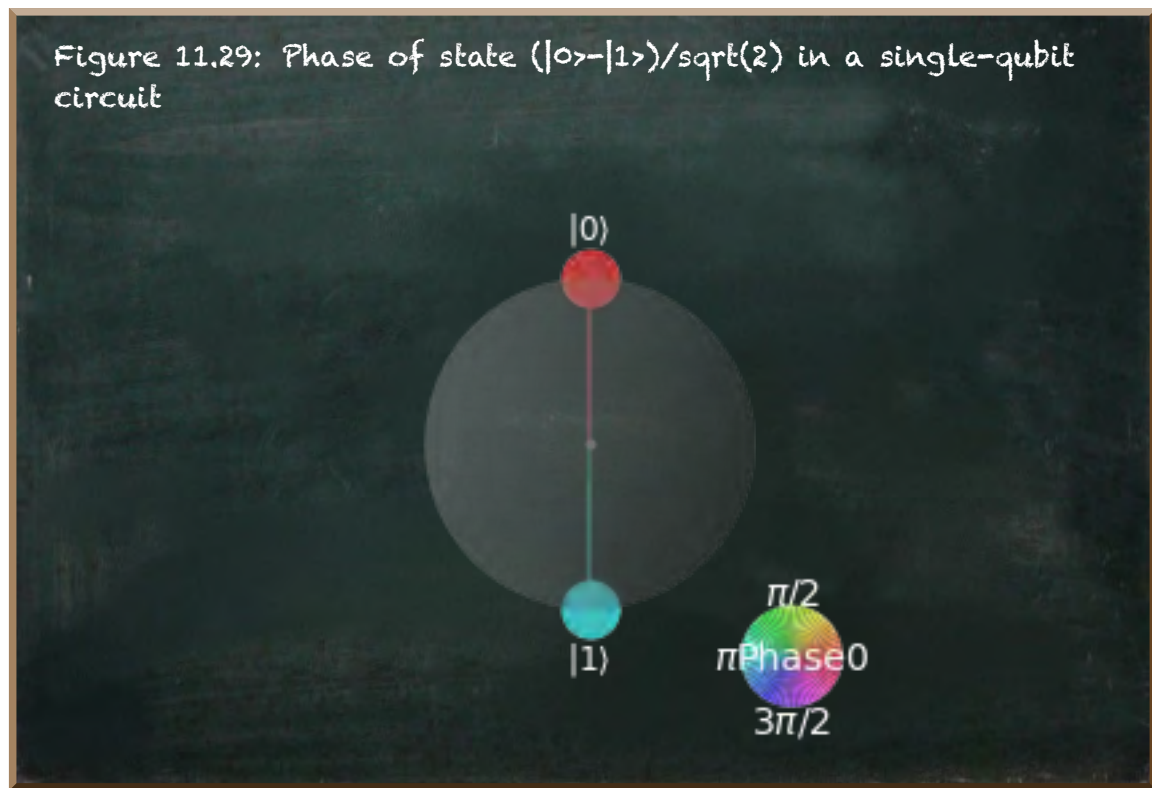
We see the phase in state $|1\rangle$. See what happens when we define the circuit as a two-qubit circuit.

Listing 11.16: create state (|0>-|1>)/sqrt(2) in a two-qubit circuit

```
1  # Create a quantum circuit with one qubit
2  qc = QuantumCircuit(2)
3
4  # create state (|0>−|1>)/sqrt(2)
5  qc.h(0)
6  qc.z(0)
7
8  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
9  plot_state_qsphere(out)
```

Figure 11.30: Phase of state (|0>-|1>)/sqrt(2) in a two-qubit circuit
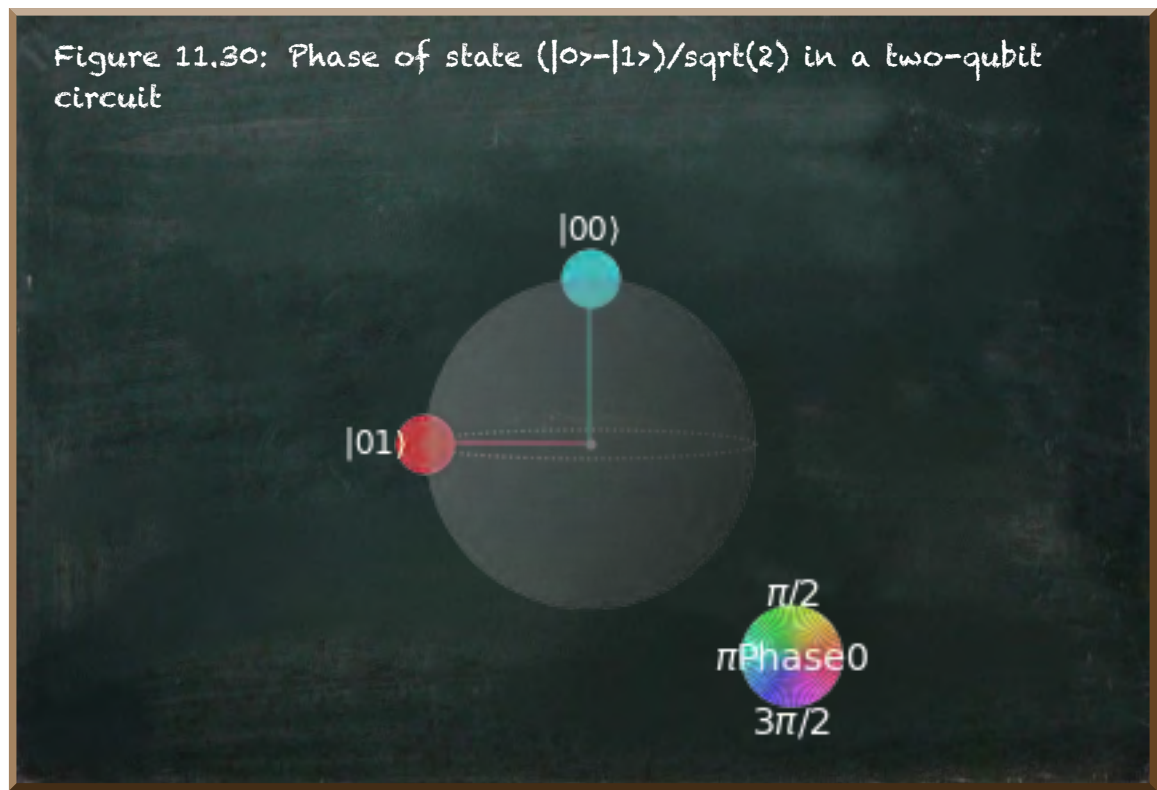
In a two-qubit circuit, we see the phase in state $|00\rangle$. Accordingly, when we apply a phase shift for one of the two qubits, we see it in the states where this qubit is in state $|1\rangle$.

Listing 11.17: Show the phases of a two-qubit system

```
 1  qc = QuantumCircuit(2)
 2
 3  # Apply H—gates on both qubits
 4  qc.h(0)
 5  qc.h(1)
 6
 7  # Shift the phase of qubit 0
 8  qc.z(0)
 9
10  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
        get_statevector()
11  plot_state_qsphere(out)
```

Figure 11.31: One out of two qubits with phase shift
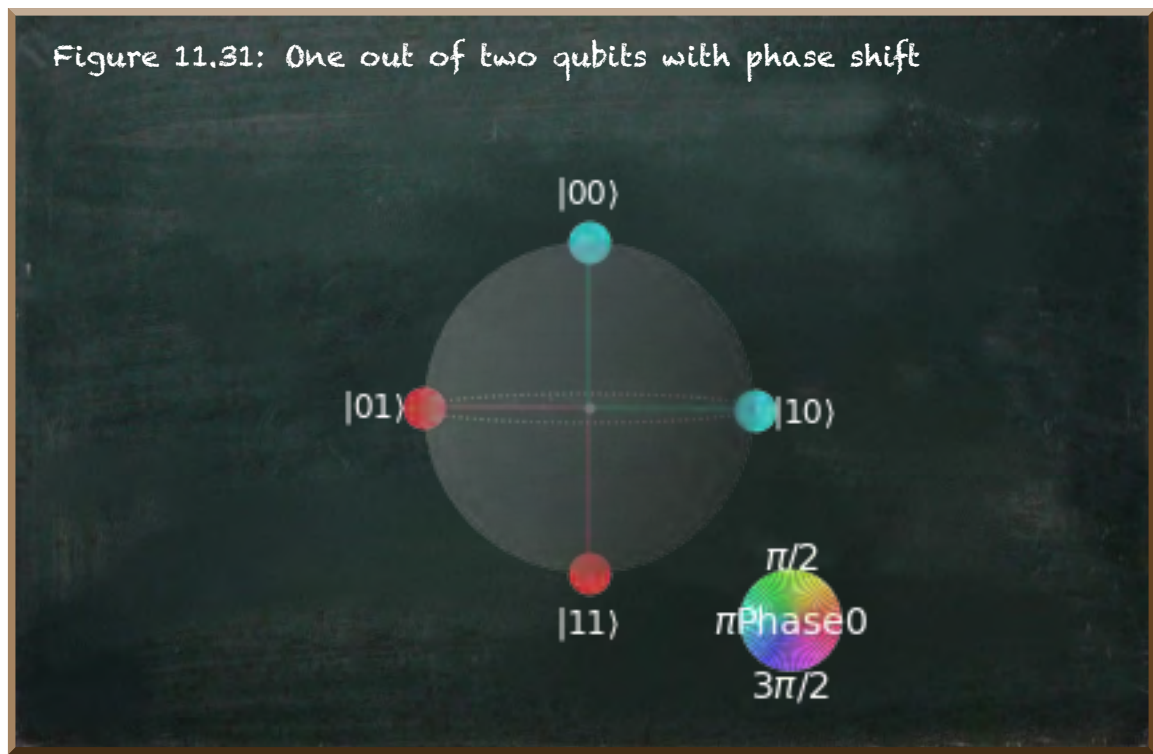
In this case, we put both qubits into a state of superposition by applying Hadamard gates (lines 4-5). We shift the phase of the qubit at position 0. Therefore, we see the states that differ for this qubit (at the right-hand side) having different phases. States with qubit 0 in state $|1\rangle$ have a different phase than the states with qubit 0 in state $|0\rangle$. Again, Qiskit makes only sense of the relative phase and indicates the shift for the states where qubit 0 is in state $|0\rangle$.

The global phase is unobservable. Global phases are the artifacts of the mathematical framework we use. They don't have a physical meaning. If two states differ only by a global phase, they effectively present the same physical system. By contrast, relative phases are the core of quantum mechanics and, therefore, of utmost interest in quantum computing. If two states differ by a relative phase, they are different systems that evolve in different ways. Even though they appear identical if measured separately, they have different effects when interfering with other quantum systems.

11.2.3 Controlled Z-gate

When we apply the Z-gate on separate qubits, we usually see the phase shift for those states where the respective qubit is in state $|1\rangle$. When both qubits are in state $|1\rangle$, the phase shift adds up. Since the Z-gate accounts for a phase

shift by half a wavelength, applying it twice results in the initial phase. Thus, the state $|11\rangle$ has the same phase as state $|00\rangle$.

Sometimes, we don't want this effect. We might want to switch the phase of a single state. Then, the controlled Z-gate comes in handy. Like its controlled peers, the controlled Z-gate applies the Z-gate on the target qubit only if the control qubit is in state $|1\rangle$.

The controlled Z-gate has the following transformation matrix.

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

As a result of the CZ-gate, we see state $|11\rangle$ has a different phase than the other three states. The qubit at position 0 must be in state $|1\rangle$, for it is the control qubit. The qubit at position 1 must be on state $|1\rangle$ because the Z-gate applies the phase to this part of the qubit state.

The CZ-gate induces a phase shift for states where both qubits are in state $|1\rangle$. It does not matter if we use qubit 0 or qubit 1 as the control qubit. Therefore, in Qiskit, we use a drawing that does not indicate which one is the control qubit.

Listing 11.18: The effect of the CZ-gate

```
1  qc = QuantumCircuit(2)
2  # Apply H-gates on both qubits
3  qc.h(0)
4  qc.h(1)
5  # Shift the phase of qubit 0
6  qc.cz(0,1)
7  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
8  plot_state_qsphere(out)
```

Figure 11.32: The effect of the CZ-gate
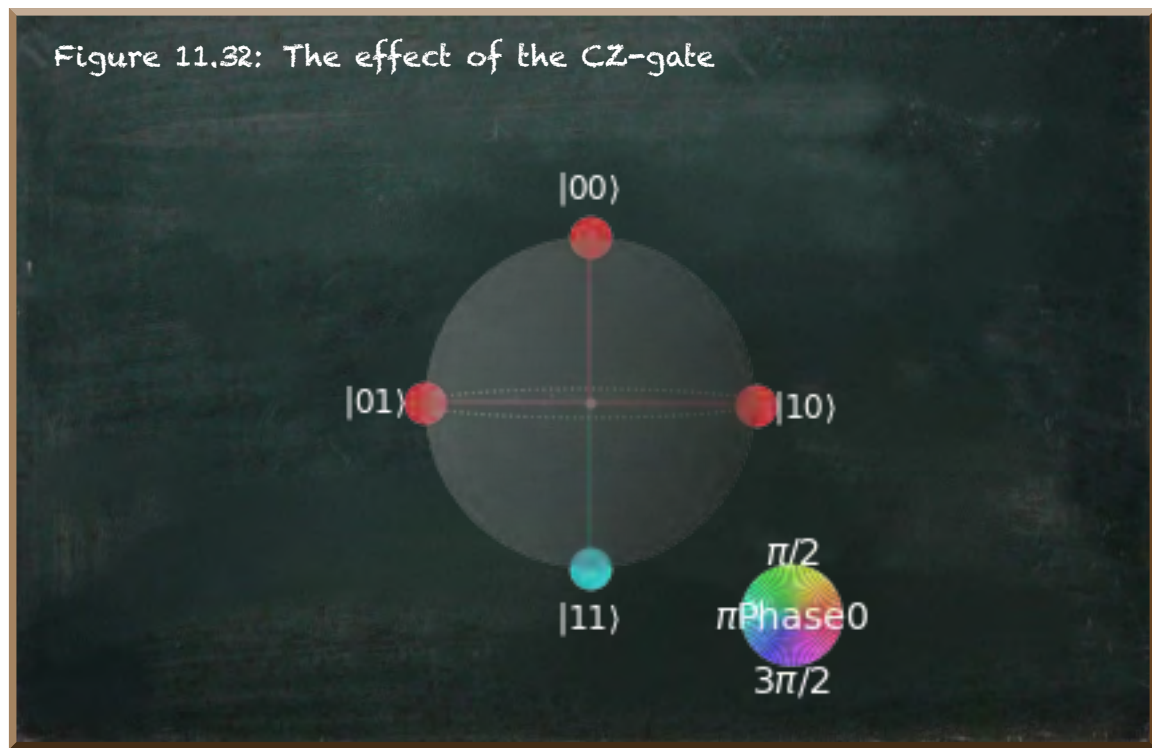
The following figure depicts the quantum circuit with the two Hadamard gates and the controlled Z-gate.

Figure 11.33: Quantum circuit with CZ-gate
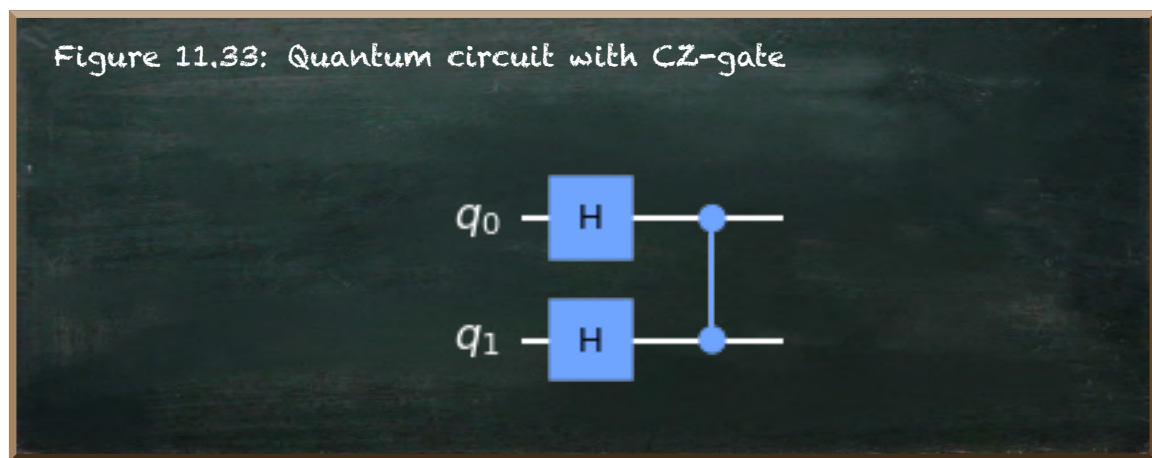
11.3 Phase Kickback

Quantum entanglement is one of the astonishing characteristics of quantum mechanics. Two entangled particles share a state of superposition—no matter how far apart they are.

From a practical perspective, we can use entanglement to let one qubit control the state of another. For instance, the controlled NOT-gate (CNOT- or CX-gate) switches the amplitudes of a target qubit only if the control qubit is in state $|1\rangle$. Nothing happens if the control qubit is in state $|0\rangle$.

Such controlled quantum gates let us precisely manipulate a multi-qubit system. In section 9.2, we let certain states of the quantum system exhibit the measurement probabilities we want them to have. We use entanglement to create a fine-grained probabilistic system.

Another practical characteristic of controlled quantum gates is that they leave the control qubit untouched.

The following figure depicts the truth table of the CNOT-gate.

Figure 11.34: Truth table of the CNOT gate

No matter which combination of qubit values we feed into the operation, the control qubit does not change.

Having a practically applicable notion of quantum transformation gates is paramount when we work with qubits. However, every once in a while, we need to remember that a quantum operation is essentially a physical operation. So is the CNOT-gate.

For every action in physics, there is an opposite reaction. For this reason, we should be suspicious even of the apparent one-sidedness of the CNOT-gate.

Figure 11.35: What I told you was true... from a certain point of view
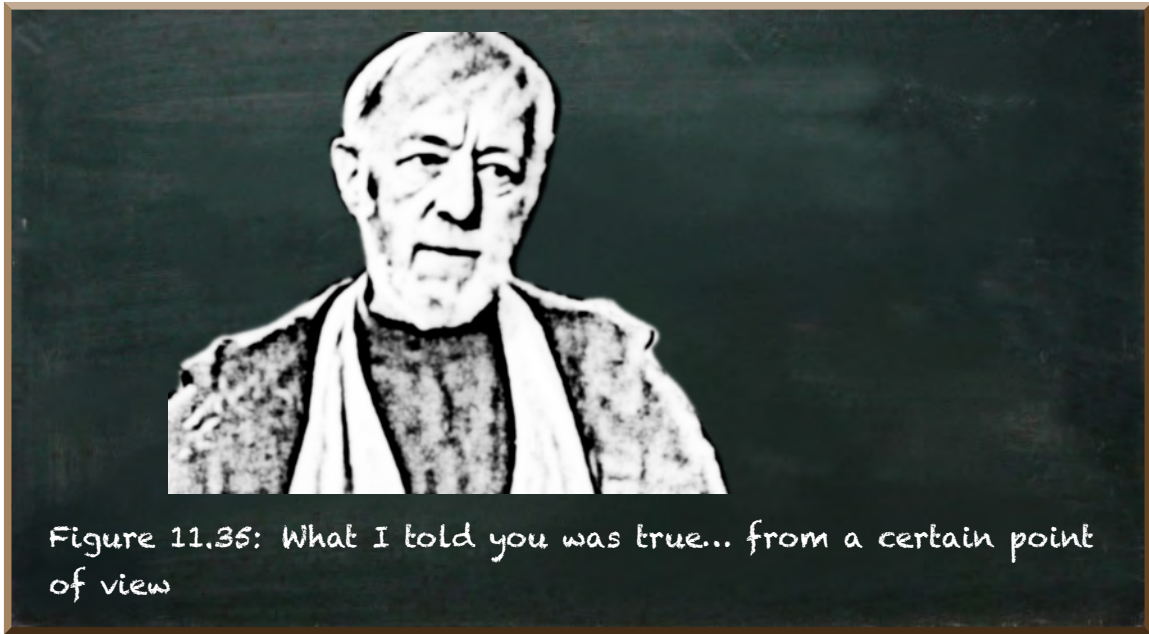
Whenever we challenge our intuition in quantum computing, it is good to consult the underlying math.

The CNOT-gate is a two-qubit gate. Thus, it transforms qubit states whose state we represent by a four-dimensional vector.

$$|\psi\rangle = \alpha|0\rangle|0\rangle + \beta|0\rangle|1\rangle + \gamma|1\rangle|0\rangle + \delta|1\rangle|1\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}$$

Accordingly, the CNOT-gate has a 4x4 transformation matrix.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

There is no effect if the control qubit (at the left-hand position in the Dirac notation) is in state $|0\rangle$.

$$CNOT \cdot |00\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

$$CNOT \cdot |01\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = |01\rangle$$

But if the control qubit is in state $|1\rangle$, then the controlled qubit switches from $|0\rangle$ to $|1\rangle$ and vice versa.

$$CNOT \cdot |10\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle$$

$$CNOT \cdot |11\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = |10\rangle$$

The math confirms our intuition.

When we describe the quantum states and operations in terms of mathematical formulae, we use the vectors $|0\rangle$ and $|1\rangle$ as a basis. $|0\rangle$ and $|1\rangle$ denote the standard or computational basis states. These states correspond to the possible measurements we might obtain when looking at the qubit. We measure a qubit in state $|0\rangle$ as 0 with absolute certainty. And, we measure a qubit in state $|1\rangle$ as 1, accordingly. While the basis $\{|0\rangle, |1\rangle\}$ is convenient to work with mathematically, it is just a representation of the underlying physics.

Just like the very idea that there is a *control* qubit embeds a prejudice about the states of the qubits that invites us to think of the operation as one-sided, the mathematical basis we chose leads to a specific representation of the CNOT-transformation. But this is not the only possible representation. There are infinitely many other possible choices. Our qubits are not limited to these two basis states. Qubits can be in a superposition of both states.

Consider the Bloch sphere. It is, in the end, a sphere — perfectly symmetric, with no one point being more special than any other and no one axis more special than any other. The standard basis is not particularly special, either.

The following figure depicts the two basis states $|0\rangle$ and $|1\rangle$.

Figure 11.36: The two basis states
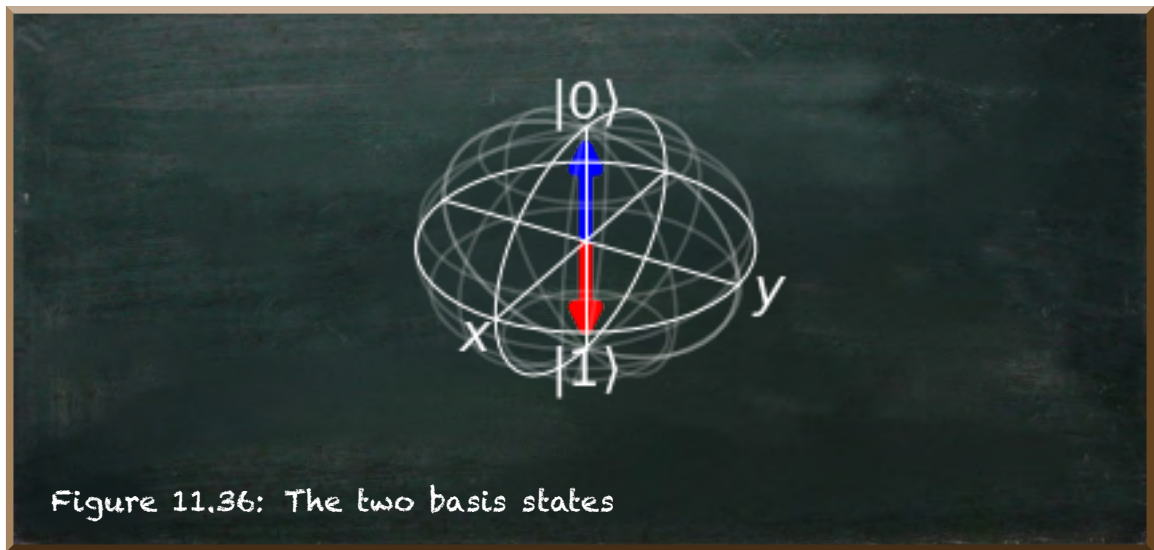
But our two qubits can be in any other state, too.

For instance, there are the states $|+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$ and $|-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$ that result from applying the Hadamard-gate on the basis states. The following figure depicts these two states.

Figure 11.37: The states |+> and |->
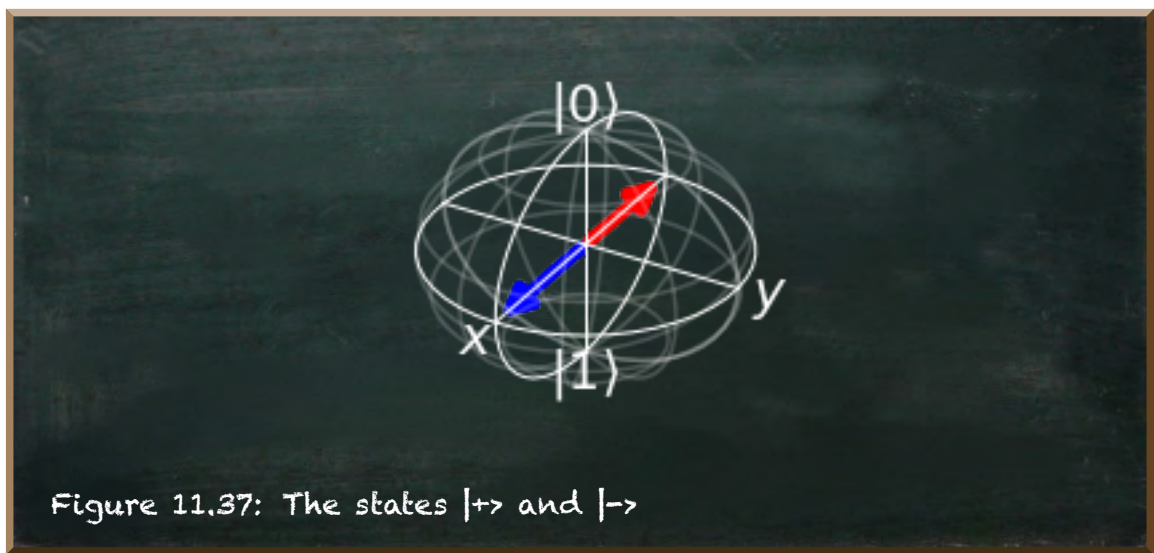
Mathematically, the following matrix represents the application of

Hadamard gates on each of the two qubits.

$$H \otimes H = \tfrac{1}{\sqrt{2}} \begin{bmatrix} H & H \\ H & -H \end{bmatrix} = \tfrac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

So, if we apply this matrix on two qubits in state $|00\rangle$, they end up in state $|++\rangle$.

$$
\begin{aligned}
H \otimes H(|00\rangle) &= \tfrac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \tfrac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \\
&= \tfrac{1}{2} (|0\rangle|0\rangle + |0\rangle|1\rangle + |1\rangle|0\rangle + |1\rangle|1\rangle) \\
&= \tfrac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \otimes \tfrac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \\
&= |++\rangle
\end{aligned}
$$

(11.1)

The input state $|01\rangle$ results in state $|+-\rangle$.

$$
\begin{aligned}
H \otimes H(|01\rangle) &= \tfrac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \tfrac{1}{2} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \\
&= \tfrac{1}{2} (|0\rangle|0\rangle - |0\rangle|1\rangle + |1\rangle|0\rangle - |1\rangle|1\rangle) \\
&= \tfrac{1}{\sqrt{2}} (|0\rangle - |1\rangle) \otimes \tfrac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \\
&= |+-\rangle
\end{aligned}
$$

(11.2)

The input state $|10\rangle$ results in state $|-+\rangle$.

$$
\begin{aligned}
H \otimes H(|10\rangle) &= \tfrac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \tfrac{1}{2} \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \\
&= \tfrac{1}{2} (|0\rangle|0\rangle + |0\rangle|1\rangle - |1\rangle|0\rangle - |1\rangle|1\rangle) \\
&= \tfrac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \otimes \tfrac{1}{\sqrt{2}} (|0\rangle - |1\rangle) \\
&= |-+\rangle
\end{aligned}
$$

(11.3)

Finally, if we apply this transformation on two qubits in state $|11\rangle$, we put

them into state $|--\rangle$.

$$
\begin{aligned}
H \otimes H(|11\rangle) &= \tfrac{1}{2}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1
\end{bmatrix}
\cdot
\begin{bmatrix}
0 \\ 0 \\ 0 \\ 1
\end{bmatrix}
= \tfrac{1}{2}
\begin{bmatrix}
1 \\ -1 \\ -1 \\ 1
\end{bmatrix} \\
&= \tfrac{1}{2}\left(|0\rangle|0\rangle - |0\rangle|1\rangle - |1\rangle|0\rangle + |1\rangle|1\rangle\right) \\
&= \tfrac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) \otimes \tfrac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) \\
&= |--\rangle
\end{aligned}
$$

(11.4)

Let's fresh up matrix-vector multiplication. When we multiply a matrix with a column vector (our quantum state), the result is another column vector, like this:

$$
M \cdot |v\rangle =
\begin{bmatrix}
a & b \\
c & d
\end{bmatrix}
\cdot
\begin{bmatrix}
v_0 \\ v_1
\end{bmatrix}
=
\begin{bmatrix}
a \cdot v_0 + b \cdot v_1 \\
c \cdot v_0 + d \cdot v_1
\end{bmatrix}
$$

For each row of the matrix, We multiply each value (column) in that row with the x-th value of the vector. If all but one values of the vector are 0 and the one value is 1, then the position of the 1 denotes the column of the matrix we end up as a result. So, $|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ends up in the first column, and $|11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ ends up in the fourth column.

Now, let's apply the CNOT-gate on qubits in superposition. We can calculate the overall transformation matrix by multiplying the matrices of the CNOT-gate and the $H \otimes H$ transformation.

$$
CNOT(H \otimes H) =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{bmatrix}
\cdot \tfrac{1}{2}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1
\end{bmatrix}
= \tfrac{1}{2}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & -1 & -1 & 1 \\
1 & 1 & -1 & -1
\end{bmatrix}
$$

The sharp-eyed reader may notice that the CNOT-gate switches the second and fourth columns of the $H \otimes H$-matrix.

When we apply this transformation to the four combinations of basis states,

we can see an interesting pattern.

$$CNOT(H \otimes H(|00\rangle)) = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

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

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

$$= |++\rangle$$

(11.5)

$$CNOT(H \otimes H(|01\rangle)) = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

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

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

$$= |--\rangle$$

(11.6)

$$CNOT(H \otimes H(|10\rangle)) = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

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

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

$$= |-+\rangle$$

(11.7)

$$CNOT(H \otimes H(|11\rangle)) = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

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

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

$$= |+-\rangle$$

(11.8)

Effectively, if the target qubit (at the right-hand side) is in state $|1\rangle$, the state of the control qubit flips from $|+\rangle$ to $|-\rangle$ and vice versa.

In short, we can say:

$$CNOT(|++\rangle) = |++\rangle$$

$$CNOT(|+-\rangle) = |--\rangle$$

$$CNOT(|-+\rangle) = |-+\rangle$$

$$CNOT(|--\rangle) = |+-\rangle$$

The two states $|+\rangle$ and $|-\rangle$ have the identical measurement probabilities of $|0\rangle$ and $|1\rangle$. They result in either value with a probability of 0.5. So, the CNOT-gate does not have any directly measurable implications. However, the control qubit switches its phase. It takes on the phase of the controlled qubit.

Since the phase of the target qubit is kicked up to the control qubit, we call this phenomenon *phase kickback*.

Let's return to practice. The following code plots the Bloch Spheres of the state $|+-\rangle$.

Listing 11.19: Show state |+->

```
1  # Create a quantum circuit with one qubit
2  qc = QuantumCircuit(2)
3
4  # put qubit 0 into |+>
5  qc.h(0)
6
7  # put qubit 1 into |->
8  qc.x(1)
9  qc.h(1)
10
11 out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
12 plot_bloch_multivector(out)
```

Figure 11.38: The Bloch Spheres of state |+->
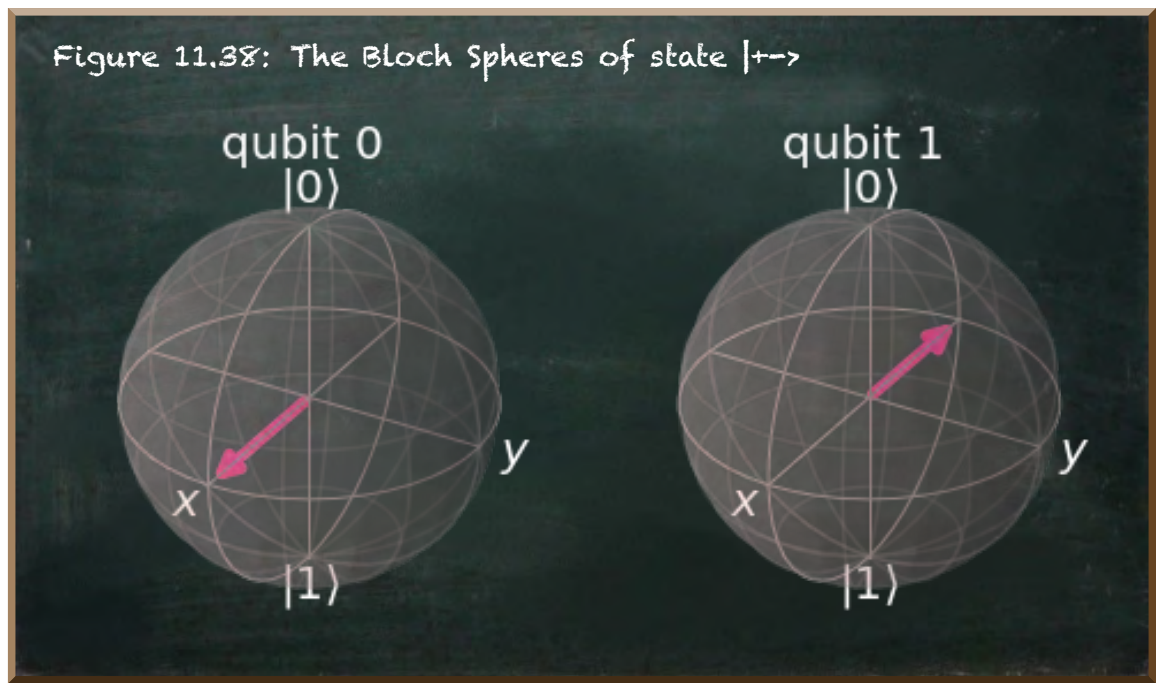
Next, we apply the CNOT-gate on this state with the qubit at position is the control qubit and the qubit at position 1 is the target.

Listing 11.20: Show effect of CNOT-gate on state |+->

```
1  # Create a quantum circuit with one qubit
2  qc = QuantumCircuit(2)
3
4  # put qubit 0 into |+>
5  qc.h(0)
6
7  # put qubit 1 into |->
8  qc.x(1)
9  qc.h(1)
10
11 # apply CNOT gate with qubit 0 as control qubit
12 qc.cx(0,1)
13
14 out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
15 plot_bloch_multivector(out)
```

Figure 11.39: The Bloch Spheres of CNOT(|+->)
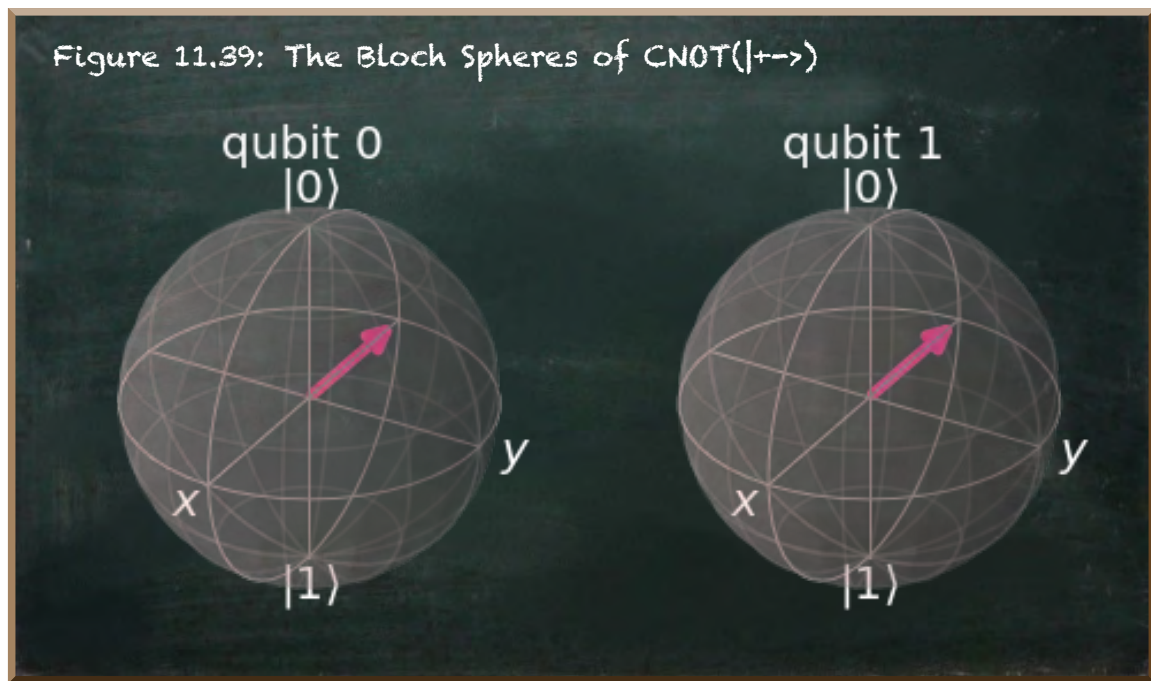
In accordance with the math, it is not the target qubit but the control qubit at position 0 that switches its phase. Let's have a look at another situation.

The following circuit applies the Hadamard gate on both qubits and a phase shift on qubit 1 by using the R_Z-gate. Similar to the R_Y-gate that rotates the qubit state vector around the Y-axis, the R_Z-gate rotates it around the Z-axis and, therefore, applies a phase.
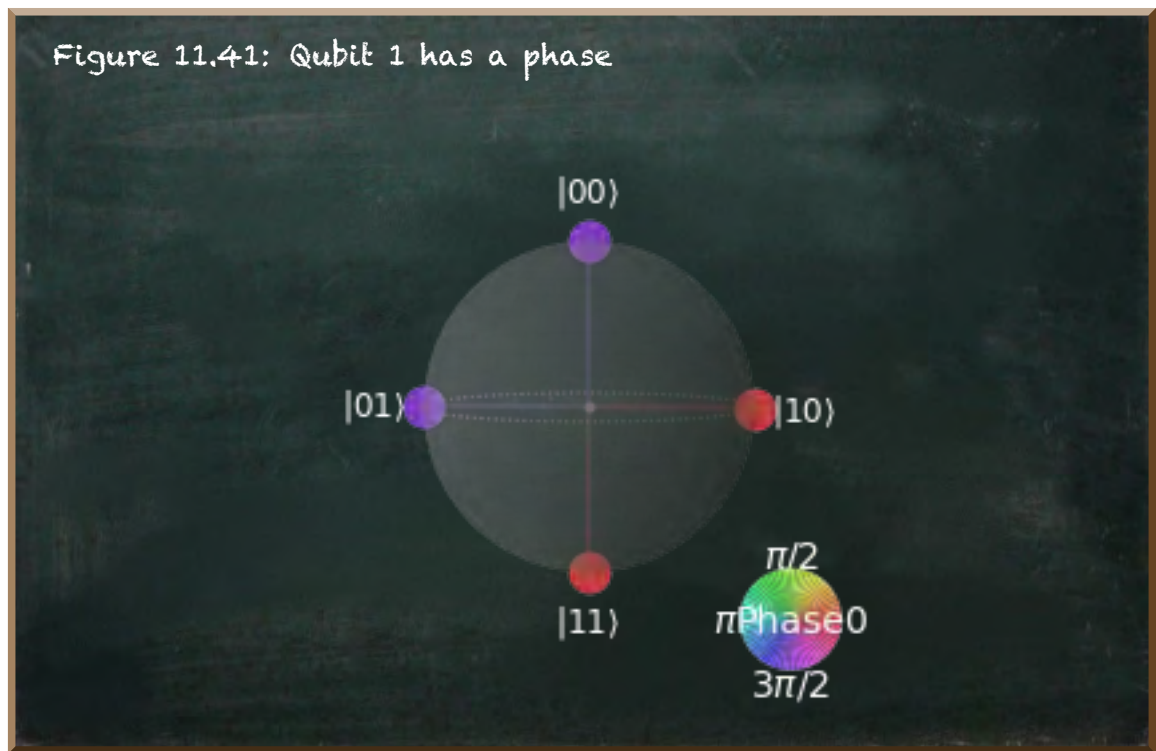
Listing 11.21: Show effect of RZ-gate on state |+->

```
1  # Create a quantum circuit with one qubit
2  qc = QuantumCircuit(2)
3
4  # put qubit 0 into |+>
5  qc.h(0)
6
7  # apply phase to qubit 1
8  qc.h(1)
9  qc.rz(pi/2,1)
10
11 out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
12 plot_bloch_multivector(out)
```

Figure 11.40: The Bloch Spheres of CNOT(|+->)
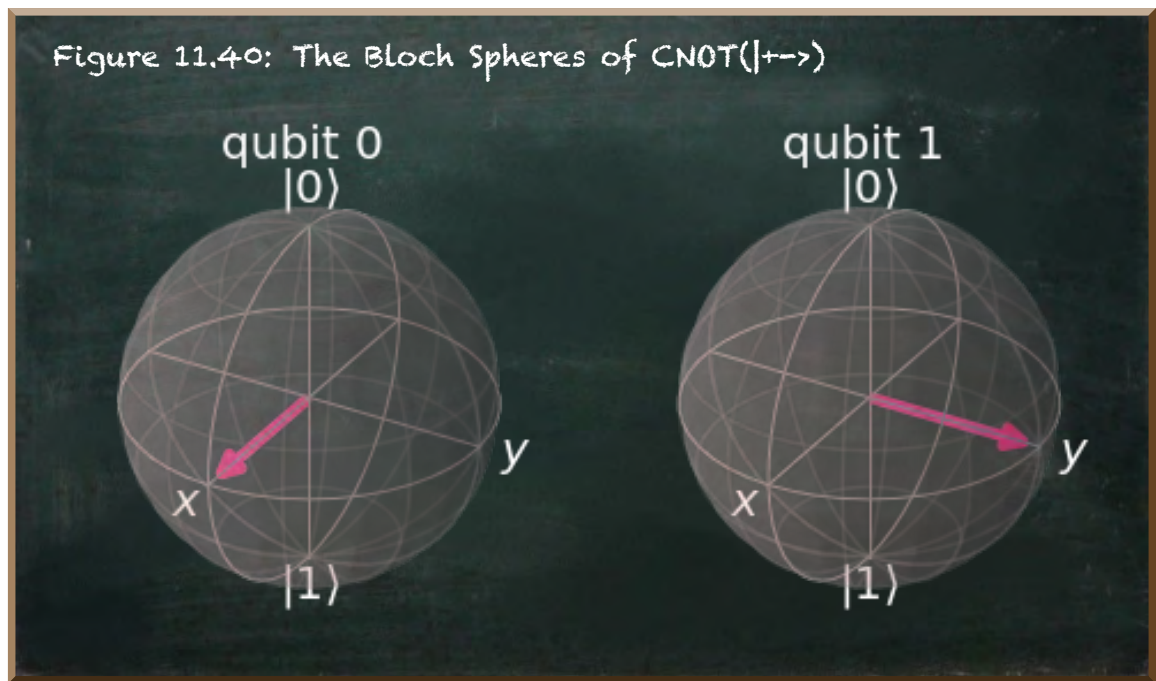
We see the qubit at position 0 resides on the X-axis. The qubit at position 1 resides on the Y-axis. Let's also look at the relative phases of the four states of this two-qubit system.


Figure 11.41: Qubit 1 has a phase

We see a phase shift for those states where qubit 1 (in Qiskit, the qubit positions read from right to left) is in state $|1\rangle$. In the next step, we apply the CNOT-gate with qubit 0 as the control qubit and qubit 1 as the target.

Listing 11.22: The phase kickback

```python
1  # Create a quantum circuit with one qubit
2  qc = QuantumCircuit(2)
3
4  # put qubit 0 into |+>
5  qc.h(0)
6
7  # apply phase to qubit 1
8  qc.h(1)
9  qc.rz(pi/2,1)
10
11  # apply CNOT gate with qubit 0 as control qubit
12  qc.cx(0,1)
13
14  out = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
15  plot_state_qsphere(out)
```


Figure 11.42: Phase kickback
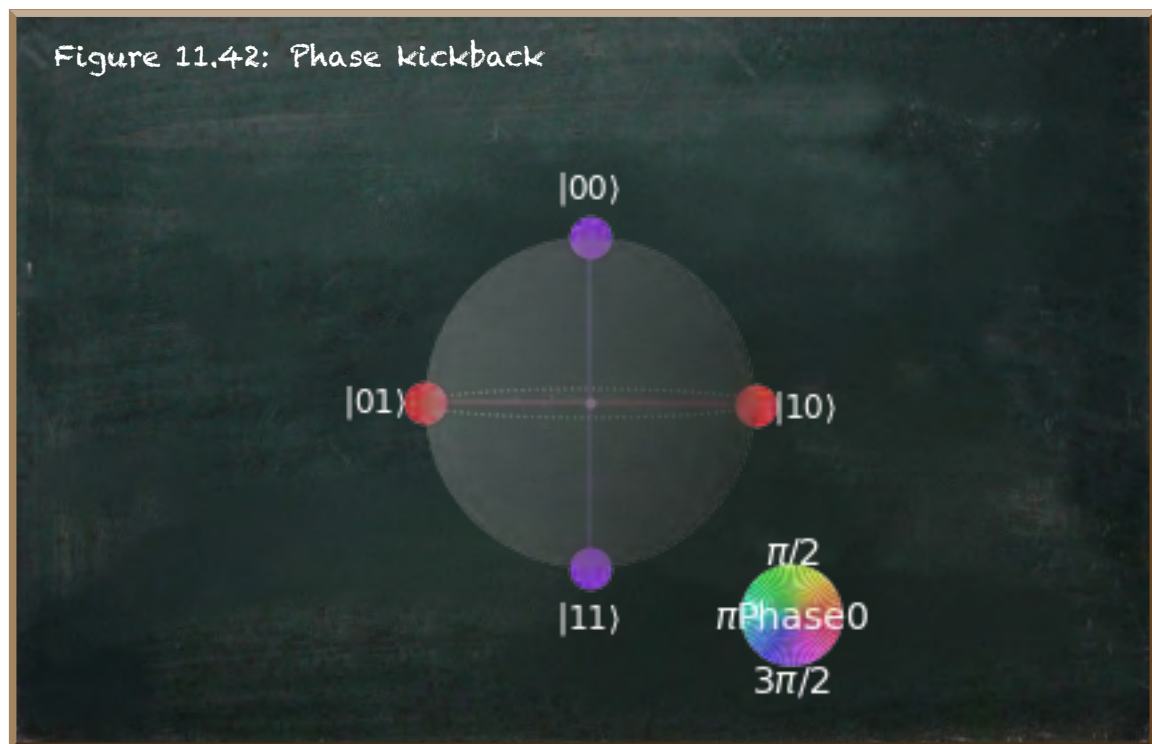
The first thing to note is the relative phase. States $|00\rangle$ and $|11\rangle$ are in the same phase that differs from the phase of the states $|01\rangle$ and $|10\rangle$. While it seems as if the phase flipped for states where the control qubit (right-hand qubit) is $|1\rangle$ it flipped for states where the target qubit is $|1\rangle$. Due to phases are relative, we can't tell which phase is the original and which is the shifted.

But the more important thing to note is the degree of the shift. The colors indicate a relative shift of $\frac{\pi}{2}$. This is the phase we applied on qubit 1. Thus, the CNOT-gate does not always flip phases by half a circuit as it does when the target qubit is either $|+\rangle$ or $|-\rangle$. But the CNOT-gate flips the phases the states $|01\rangle$ and $|11\rangle$ have. Effectively, it applies the phase of the target qubit on the control qubit.

Usually, it is good to develop a non-mathematical intuition of quantum states and operations. Though, we always need to consider that we're coping with a quantum mechanical system. Not too seldom, quantum mechanics are counter-intuitive. We can complement our intuition with math. But math is not free from pitfalls, either.

From classical computing and for mathematical convenience, all too often, we rely on the standard basis vectors $|0\rangle$ and $|1\rangle$. But when working with qubits, we need to remember that they are not limited to these states but can be in a state of superposition.

We learned the CNOT-gate is not a one-sided operation. It clearly has the potential to affect the state of the control qubit. Even though the phase is not directly measurable, there are ways to exploit differences in the phase between states. In fact, prominent algorithms, such as Grover's search algorithm, use this effect. So will we in the future.

11.4 Quantum Amplitudes and Probabilities

More than once in this book, I emphasized that the qubit state vector contains amplitudes rather than measurement probabilities.

The amplitudes belong to waves. Because in quantum mechanics, the behavior of quantum particles is described by wave functions.

Waves have three characteristics.

- The wavelength is the distance over which the wave's shape repeats.
- The phase of a wave is the position on the waveform cycle at a certain point.

• The amplitude of a wave is the distance between its center and its crest.

The following figure depicts these three characteristics.

Figure 11.43: Characteristics of a wave
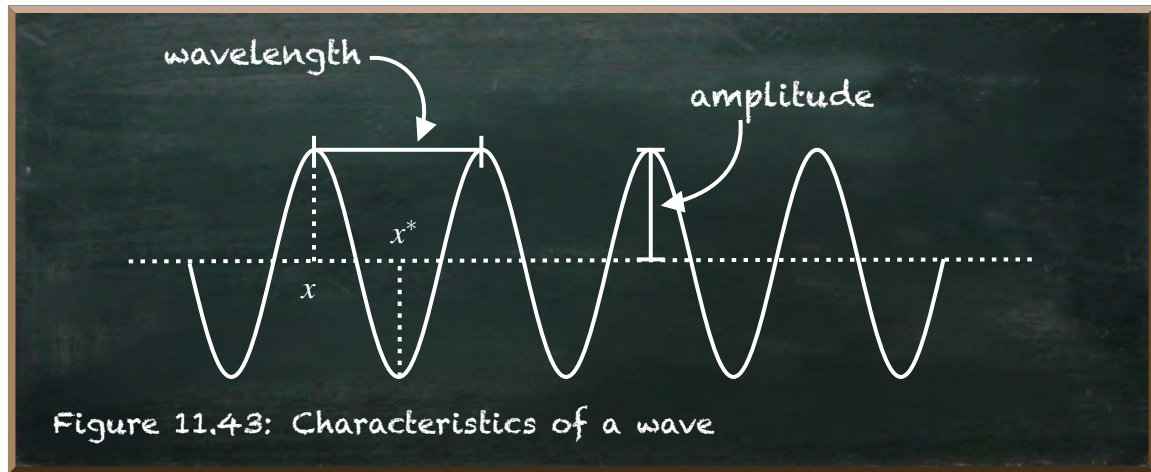
As we can see in the figure, amplitudes can be positive or negative. Whether the amplitude is positive or negative depends on the imaginary point x. If you chose a different point x^*, the same wave would have a negative amplitude.

If you take two identical yet shifted waves, one might have a positive amplitude, whereas the other has a negative at point x. These two waves differ in their phase. But when you measure either one of these two waves, they are alike. You don't see any difference. Their effects on the measurement probabilities are the same.

Mathematically, the probability of measuring the qubit as 0 or 1 is the square of the corresponding amplitude. It does not matter whether the amplitude is positive or negative.

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

with

$$|\alpha|^2 + |\beta|^2 = 1$$

In the last section 11.1, we learned the amplitudes α and β are complex numbers. As such, their imaginary part can be negative when squared ($i^2 = -1$). Thus, we need to take the absolute of the amplitudes in this normalization before we square them to obtain the measurement probabilities.

Though, amplitudes and probabilities are two different things.

Thus far, the measurement probabilities were all we cared about. We only

cared about amplitudes inside the math. We didn't yet work with qubit amplitudes and phases.

But we can work with amplitudes, too. For instance, the Z-gate switches the sign of the amplitude.

Listing 11.23: Negating the amplitude

```
1  from qiskit import QuantumCircuit, Aer, execute
2
3  qc = QuantumCircuit(1)
4  qc.h(0)
5  qc.z(0)
6
7  # execute the qc
8  results = execute(qc,Aer.get_backend('statevector_simulator')).result().
       get_statevector()
```

When we look at the resulting amplitudes of the qubit, we see the qubit has a negative amplitude in state 1. But the measurement probabilities are both positive.


Figure 11.44: Amplitudes and probabilities
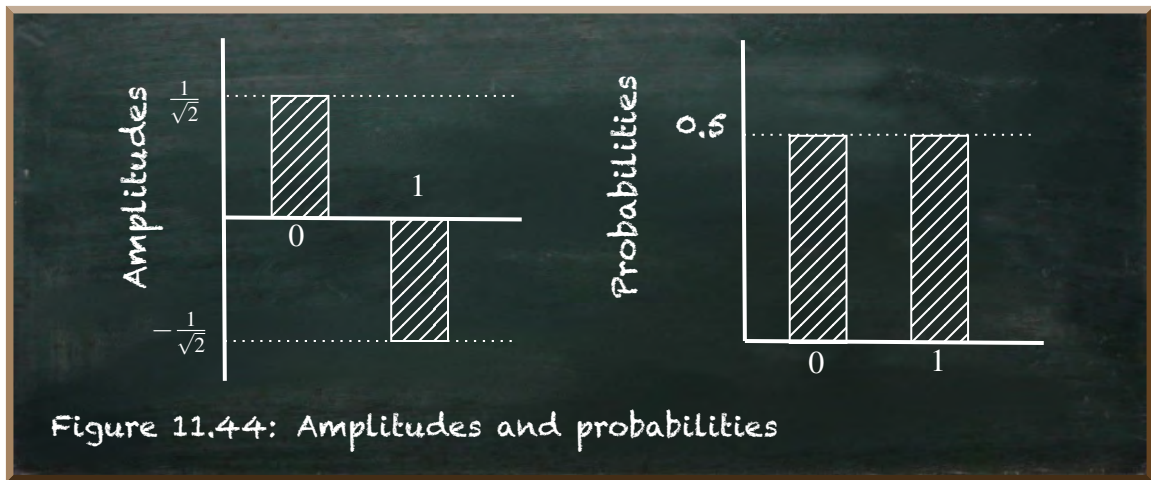
While amplitudes can be negative, probabilities can't. This is one consequence of the qubit state normalization $|\alpha|^2 + |\beta|^2 = 1$.

The normalization formula constrains the values the measurement probabilities can obtain. It does not constrain the amplitudes.

But why should we even care about the sign of a state's amplitude in quantum computing if it doesn't matter for the resulting probability?

The answer is wave interference. Waves have the distinct characteristic that they interfere with each other. Simply put, they add up. If you have two waves traveling on the same medium, they form a third wave representing the sum of their amplitudes at each point.

Figure 11.45: Two interfering waves
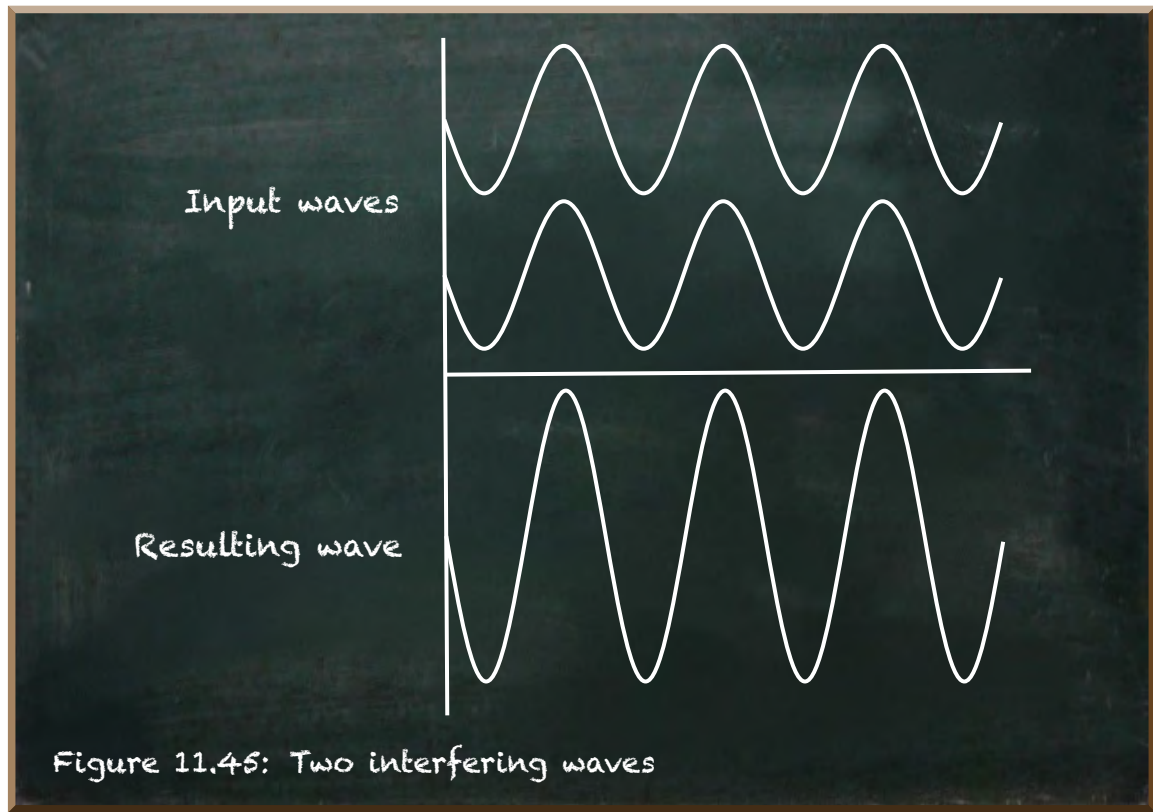
Qubits work accordingly. While the sign of the amplitude and the resulting phase of a qubit do not affect the measurement probabilities, once qubits interfere with each other, their resulting amplitudes may differ and affect the measurement probabilities.

This is known as amplitude amplification. It is an essential tool in our quantum computing toolbox.

Let's have a look at the quantum circuit depicted in the following figure.

Figure 11.46: Working with amplitudes
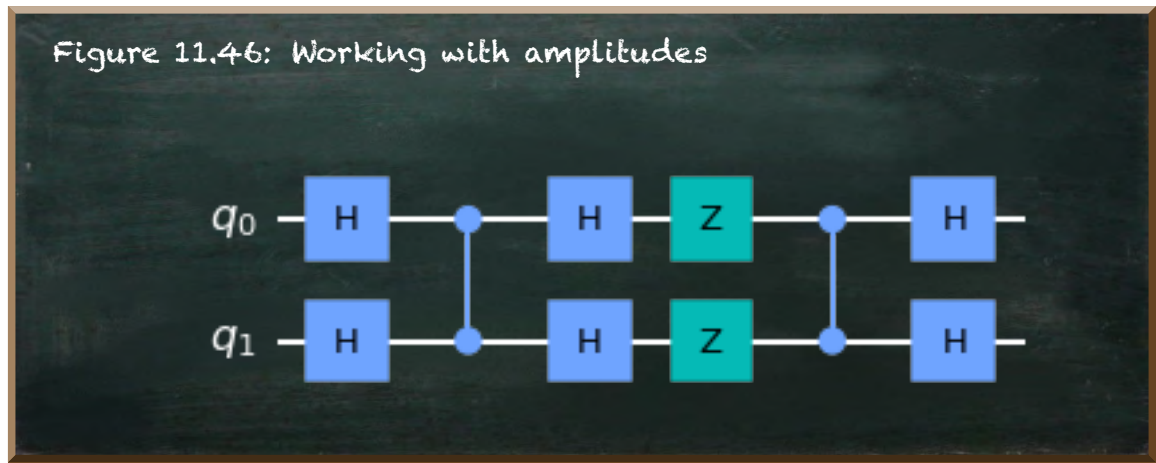
We have two qubits. Both are in a balanced state of superposition after we applied the first Hadamard-gate on each of them.

At this time, we have four possible states with equal amplitudes and equal measurement probabilities.

The following controlled Z-gate switches the sign of the amplitude only of the state $|11\rangle$. It works like the other controlled gates we got to know except for the applied gate. Here, it is the Z-gate.

The following sequence of Z-gates and another controlled Z-gate encapsulated in H-gates (HZH) is known as Grover's iterate. It is a reflection circuit that inverts all states about the mean amplitude (not the mean probability).

Since three out of four states have a positive amplitude (0.5) but only one has a negative amplitude (-0.5), the mean amplitude is 0.25.

Those states with the positive amplitude result at an amplitude of 0 because $0.5 - (2*0.25) = 0$.

The one state with the negative amplitude results in an amplitude of 1 because $-0.5 - (2*(-0.75)) = 1$.

Even though the phase of a qubit state vector (the sign of its amplitude) does not matter for the resulting measurement probability, it does matter inside the quantum circuit because we can work with the phases.

The following code shows the implementation of this circuit and the measurement probabilities.

Listing 11.24: Working with amplitudes

```
 1  from qiskit.visualization import plot_histogram
 2
 3  qc = QuantumCircuit(2)
 4  qc.h(0)
 5  qc.h(1)
 6  qc.cz(0,1)
 7  qc.h(0)
 8  qc.h(1)
 9  qc.z(0)
10  qc.z(1)
11  qc.cz(0,1)
12  qc.h(0)
13  qc.h(1)
14
15  # execute the qc
16  results = execute(qc,Aer.get_backend('statevector_simulator')).result().
        get_counts()
17  plot_histogram(results)
```


Figure 11.47: Result of reflecting amplitudes
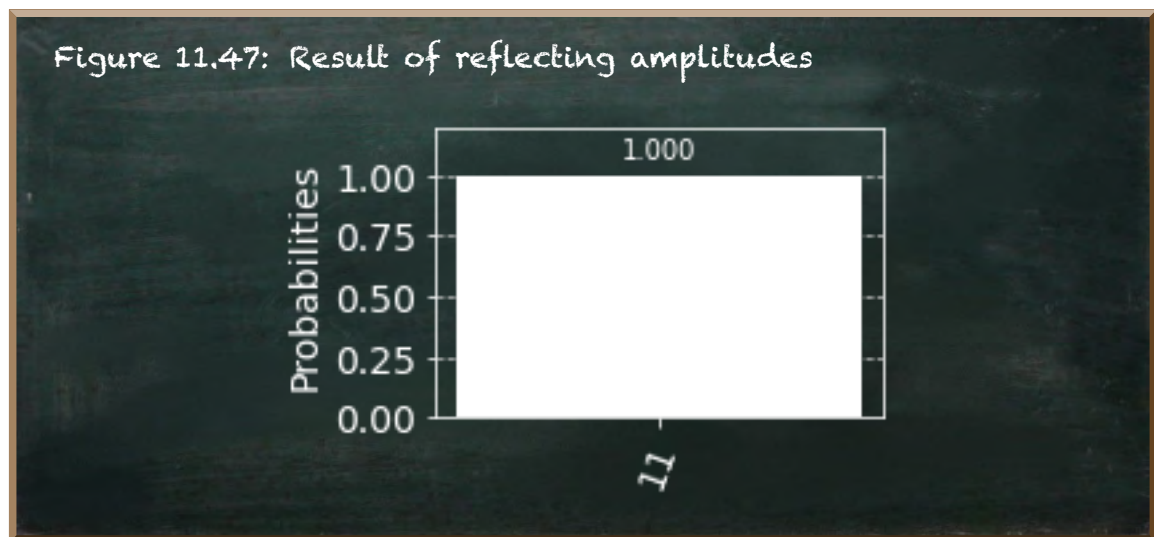
The result shows a chance of 1.0 to measure the system as 11. Unlike before, when we directly worked with the measurement probabilities, we only change the phase of the qubit.

Qubits resemble the behavior of waves. As such, their amplitudes matter when they interact. The circuit we implemented first changes the state $|11\rangle$ (by negating its amplitude). Then, it adds another wave that shares the same

phase as state $|11\rangle$. It doubles the amplitude of this state from $(-)0.5$ to 1. For this phase is the exact opposite of the phase the other three states have. They cancel out.

12. Working With The Qubit Phase

Machine learning can be thought of as a search problem. Given only observations of inputs and related outputs, we can characterize the learning problem as searching for a mapping of inputs to the correct outputs. A good mapping, the solution to the search problem, allows predicting the output for any given input.

Logically, the mapping is a function, such as $y = f(x)$ with x as the input and y as the output. The problem is we don't know anything about the function f. If we did, we wouldn't need to learn it from the data. We would specify it directly.

If we can't specify the function, the next best thing we can do is find an approximation of the function f. But again, since we don't know anything about the mapping function f, we can't tell how a good approximation would look. It could be a mathematical function. It could be a Bayesian system. It could even be a artificial neural network. There are infinitely many possible approximations. We don't know which kind of approximation works best.

You can think of all possible approximations of the mapping function as a huge search space. The learning task is to search this space for a good enough approximation. In general, search involves trial and error. You try a possible solution, you see how it performs, and you adjust your solution accordingly.

12.1 The Intuition Of Grover's Algorithm

In many situations, we need to find one particular item in a set of many items. Unsurprisingly, searching algorithms are among the most prominent and useful algorithms in Computer Science.

Let's imagine you need to call a famous quantum computing pioneer, Mr. Grover. You search a phone book for his number because you don't have it yet. You open up the book in the middle, and you see names with the starting letter L. For G is before L, you take the first half of the book and open it up in the middle again. There you see names with an E. For G is after E, you open up the book in the middle between E and L. There you see Mr. Grover's number.

The name of this search algorithm is binary search. This algorithm repeatedly divides the search interval in half. If the searched item is lower than the item in the middle of the interval, it narrows the interval to the lower half. Otherwise, it narrows it to the upper half. It repeats until the value is found or the interval is empty. The binary search algorithm narrows down the search space pretty fast and converges to a solution. The only problem is that the algorithm relies on the data you search to be sorted. The algorithm doesn't work if the data isn't sorted.

This is a big problem because not only binary search relies on sorted data, but almost all search algorithms do.

If the data is not sorted or structured in any other way, the only valid search algorithm is a linear search. In linear search, we have to evaluate every single item to verify whether it is the searched one or not. While this approach works for small data, it becomes inappropriate for large data sets.

This is where Grover's algorithm comes into play. In section 8.2, we got to know Deutsch's algorithm that could evaluate a function for two input parameters while only running it only once. Grover's algorithm teaches us how to search for an item in an unsorted list without needing to look at each item one by one but by looking at them all at once. It accomplishes that using two techniques. First, it uses a quantum oracle to mark the searched state. Second, it uses a diffuser that amplifies the amplitude of the marked state to increase its measurement probability.

We can describe quantum systems in terms of their states, amplitudes, and measurement probabilities. But these are internal descriptions we can't observe directly. Whenever we measure a quantum system, we get a single

value–the one state this specific system is in.

If we measure two similar systems, we might measure different values. Then, we know that this particular system can be in different states. If we have many similar systems, we might measure the system in some states more often than in other states. How often we measure a quantum system in a certain state depends on probability. Each state of a quantum system has a certain measurement probability. The higher it is, the more likely we will measure the system in this state.

The measurement probability of a state depends on the amplitude of this particular state. Mathematically, the measurement probability is the squared absolute of the amplitude. We will see what this means in a second.

Grover's search algorithm starts from a set of qubits in equal superposition. This means all states have equal amplitudes. Therefore, they all have the same measurement probability. If we measure this system only once, we will find it in any state. Which one it is, is up to chance. If we measured this system myriads of times, we would see it in each of the states equally often.

Figure 12.1: Initial states
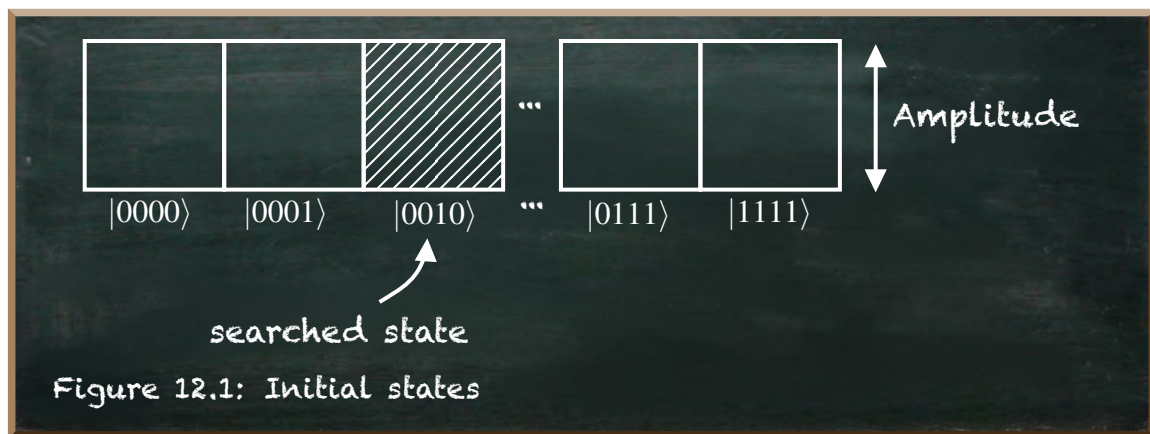
The state we search for is among these states. The goal is to change the system so that if measured, we find the system in this one state. Always.

It is the task of the quantum oracle to identify the searched state. Rather than a magical ingredient, the quantum oracle is a control structure. It is a quantum operator. This operator negates the amplitude of the searched state.

Of course, the vital question is: *"How does the oracle identify the searched state?"*

The oracle uses any characteristic of the searched quantum state. If we start with a set of equal states, then, per definition, the states only differ in their enumeration. If we use four qubits, for example, then there are 2^4 different

states. Starting from $|0000\rangle$, to $|0001\rangle$, ending at $|1111\rangle$.

Each qubit can have a specific meaning. We could interpret it as a letter. A letter that does not have 26 different options but only two, $|0\rangle$ and $|1\rangle$. With a sufficient number of qubits, we could represent all living humans. With 33 qubits, we can represent around 8.5 billion different states. A phonebook of mankind. And we haven't sorted it.

Now, let's say four qubits are enough, and Mr. Grover is known as $|0010\rangle$. The oracle uses the specific characteristic of this state to identifying it. That is the state has a $|1\rangle$ at the third position and $|0\rangle$ otherwise.

Since the quantum oracle takes all qubits as input, it can easily transform this exact state. It doesn't matter whether we use four qubits or 33. The oracle identifies Mr. Grover in a single turn.

The transformation the oracle applies to the searched state is an inversion of the amplitude.


Figure 12.2: State with negative amplitude
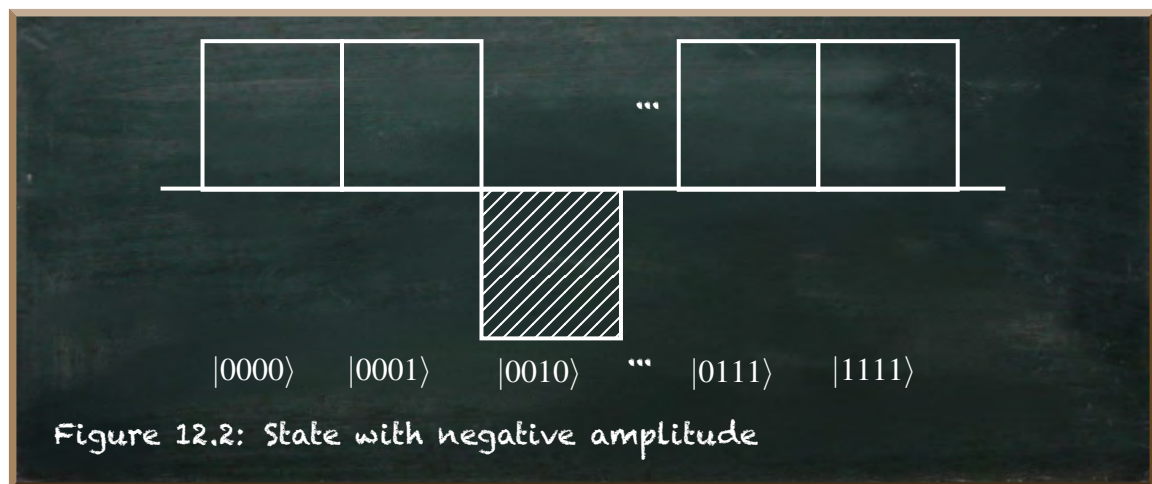
In this representation of the amplitudes, we can see a difference between the searched state and all the other states. We could prematurely declare the search is over.

The only difference is in the sign of the amplitude. For the measurement probability results from the amplitude's absolute square, the sign does not matter at all.

The amplitude originates from the concept that every quantum entity may be described as a particle and as a wave. The main characteristic of a wave is that it goes up and down as it moves. The amplitude is the distance between the center and the crest of the wave.

If we invert the amplitude of a wave at all positions, the result is that the same wave shifted by half of its wavelength.

These two waves differ only in their relative position. This is the phase of the wave. For the outside world, the phase of a wave is not observable. But, observed individually, the two waves appear identical. So, the problem is that we can't tell the difference between these two waves.

Figure 12.3: Wave with a negated amplitude
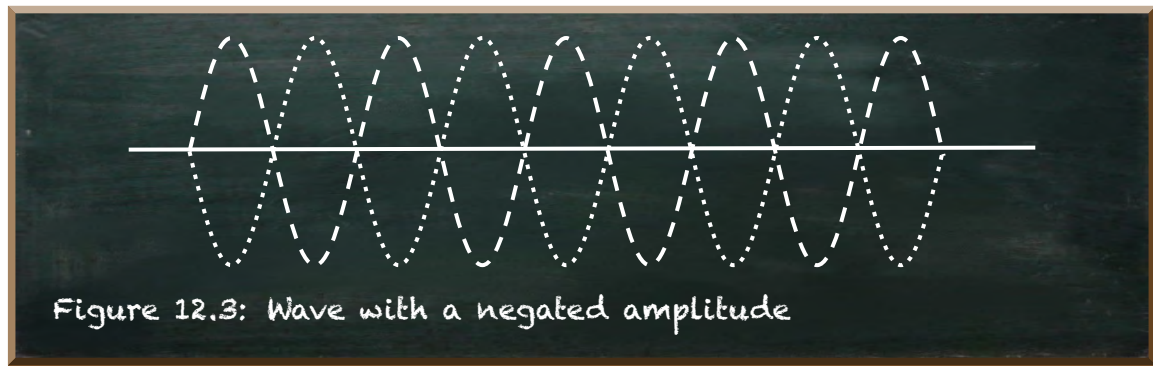
As a consequence, the system does not appear any different from the outside. Even though the oracle marked the searched state and it, therefore, differs from the other states, all states still have the same measurement probability.

We need to turn the difference into something measurable. We need to increase the measurement probability of the marked state. This is the task of the diffuser. The diffuser applies an inversion about the mean amplitude.

Let's have a look at the average amplitude.

Figure 12.4: The average amplitude
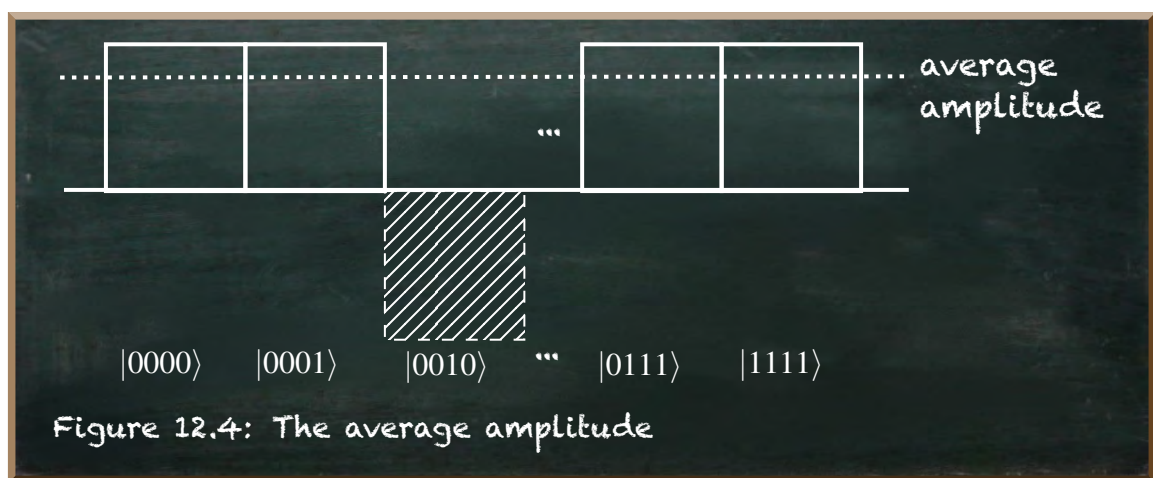
With four qubits, we have 16 different states. Each state has an amplitude of $\frac{1}{\sqrt{16}} = \frac{1}{4}$. Each but one state–the searched state has this amplitude. The

searched state has an amplitude of $-\frac{1}{4}$. Thus, the average is $\frac{15*\frac{1}{4}-\frac{1}{4}}{16} = \frac{\frac{14}{4}}{16} = 0.21875$.

The average is a little less than the amplitude of all states we did not mark. If we invert these amplitudes by this mean, they end up a little lower than the average at 0.1875.

Since the amplitude of the marked state is negative, it is pretty far away from the average. The inversion about the mean has a greater effect. It flips the amplitude from -0.25 by $2*(0.25+0.21875)$ to 0.6875.

Figure 12.5: Inversion about the mean
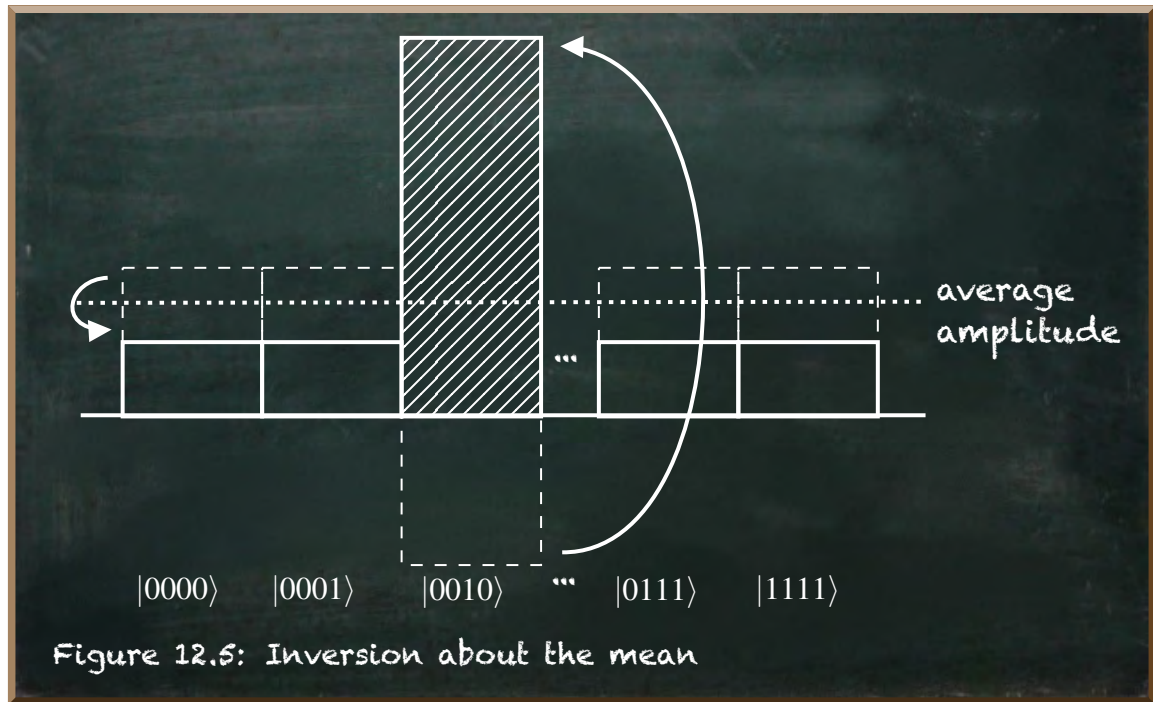
The inversion about the mean works well if we search for a single or a few negative amplitudes among many positive amplitudes. Then, this operation increases the negative amplitudes we know are the correct ones. And this operation decreases the positive amplitudes we know are wrong.

This operation increases the negative amplitude by a considerable amount while decreasing the positive amplitudes by a small amount.

But the more states we have, the lower the overall effect will be. In our example, we calculated the new amplitude of the searched state as 0.6875. The corresponding measurement probability is $0.6875^2 = 0.47265625$. Accordingly, we measure this system only about every other time in the state we are looking for. Otherwise, we measure it in any other case.

Of course, we could now measure the system many times and see our

searched state as the most probable one. But running the algorithm so often would give away any advantage we gained from not searching all the states.

Instead, we repeat the algorithm. We use the same oracle to negate the amplitude of the searched state. Then we invert all the amplitudes around the mean, again.

However, we must not repeat this process too many times. There is an optimal number of times of repeating this process to get the highest chance of measuring the correct answer. The probability of obtaining the right result grows until we reach about $\frac{\pi}{4}\sqrt{N}$ with N is the number of states of the quantum system. Beyond this number, the probability of measuring the correct result decreases again.

In our example with four qubits and $N = 16$ states, the optimum number of iterations is 3.

Grover's algorithm searches unsorted data. It follows a simple procedure. A quantum oracle inverts the amplitude of the searched state. Then, the diffuser reverses all states about the mean amplitude, therefore, magnifying the searched state. The algorithm repeats these steps until the solution has a measurement probability close to 100%.

The number of repetitions depends on the number of states it needs to consider. Since the number of repetitions only increases by around the square root of the number of states, this algorithm provides a quadratic speedup compared to a classical linear search–that is the only classical approach that can search unsorted data.

However, like all quantum computer algorithms, Grover's algorithm is probabilistic. It returns the correct answer with high but not absolute probability. Therefore, we might need to repeat the algorithm to minimize the chance of failing.

12.2 Basic Amplitude Amplification

In the previous section 12.1, we built the conceptual understanding of how Grover's search algorithm works. Building up some intuition is one thing. Translating this intuition into a working quantum circuit is a whole different story. Implementing reflections in a quantum circuit requires us to stretch our minds. Don't worry. We start simple.

Grover's algorithm consists of two major components. First, the oracle identifies and marks a favorable state. Second, the diffuser amplifies the ampli-

tude of good states.

The first stretch of mind involves not thinking in qubits but thinking in states. Of course, qubits are the computational unit we work with. Of course, the possible states of a multi-qubit system depend on the qubits we use. Of course, important visualizations of a quantum system, such as the Bloch Sphere, build upon the qubit.

But we also must keep in mind an essential feature of qubits. We can entangle qubits. And we can't represent two entangled qubits by two separated qubits anymore. Two entangled qubits share their states. We can't represent one without the other because if we measure one of the qubits, the other's state inevitably changes, too.

Moreover, the power of quantum computing lies in the fact that qubits not only form states, but we can work with their states all at once. So, rather than thinking in qubits, we need to think in states.

Let's start with the simplest case of Grover's algorithm. We have a single qubit—two possible states, on and off, $|1\rangle$ and $|0\rangle$.

The first step in Grover's algorithm is always the same. We put all qubits into an equal superposition so that each state has the same amplitude and thus, the same measurement probability. We achieve this through the Hadamard gate.

Now, both possible states, $|0\rangle$ and $|1\rangle$ have a probability of 0.5 each.

Listing 12.1: Equal superposition of two states

```
1  from qiskit import QuantumCircuit, Aer, execute
2  from qiskit.visualization import plot_histogram, plot_bloch_multivector,
       plot_state_qsphere
3
4  qc = QuantumCircuit(1)
5  qc.h(0)
6
7  # execute the qc
8  results = execute(qc,Aer.get_backend('statevector_simulator')).result()
9  plot_histogram(results.get_counts())
```

Figure 12.6: Equal superposition of two states

Further, both states share the same phase.

Figure 12.7: No phase difference

Let's say the state $|1\rangle$ depicts the favorable state we want to find. Then, the oracle consists of the Z-gate that switches the amplitude when the corresponding qubit is in state $|1\rangle$.

Figure 12.8: Circuit with an oracle
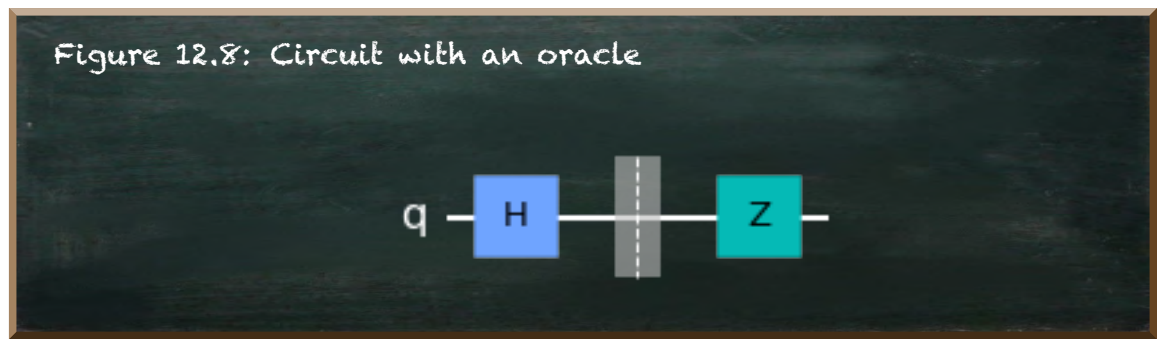
As a result, we see the amplitude changed for state $|1\rangle$. The qubit is now in state $|-\rangle$. Its two states $|0\rangle$ and $|1\rangle$ are in two different phases, now.

Figure 12.9: State |1> has a different phase
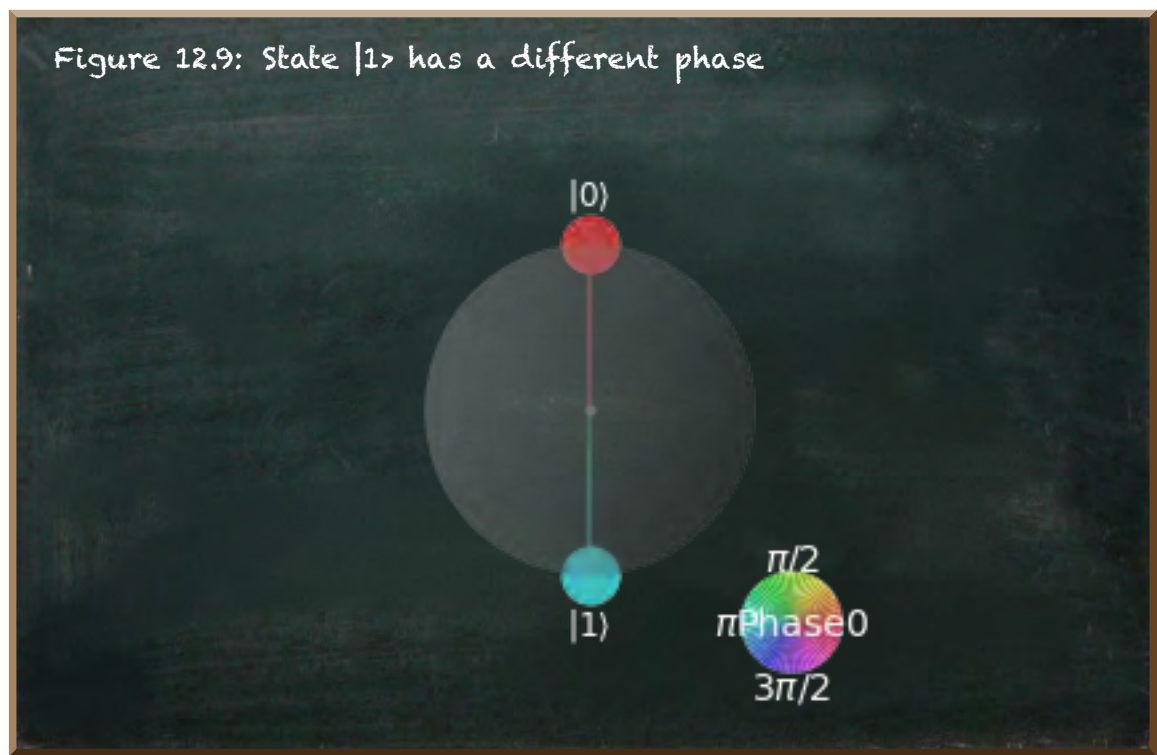
In other words, we flipped the amplitude of state $|1\rangle$ from positive to negative.

Both states still have a measurement probability of 0.5. It is the task of the diffuser to magnify the amplitude to favor the searched state.
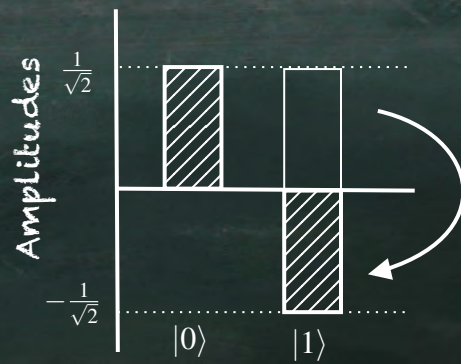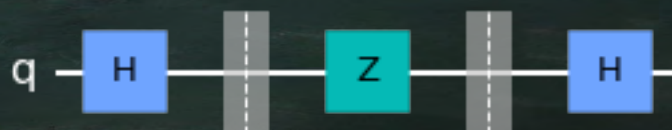
Figure 12.10: Flipping the amplitude

Figure 12.11: The phase does not affect the measurement probabilities

The diffuser in a single-qubit circuit is quite simple. It is another H-gate.

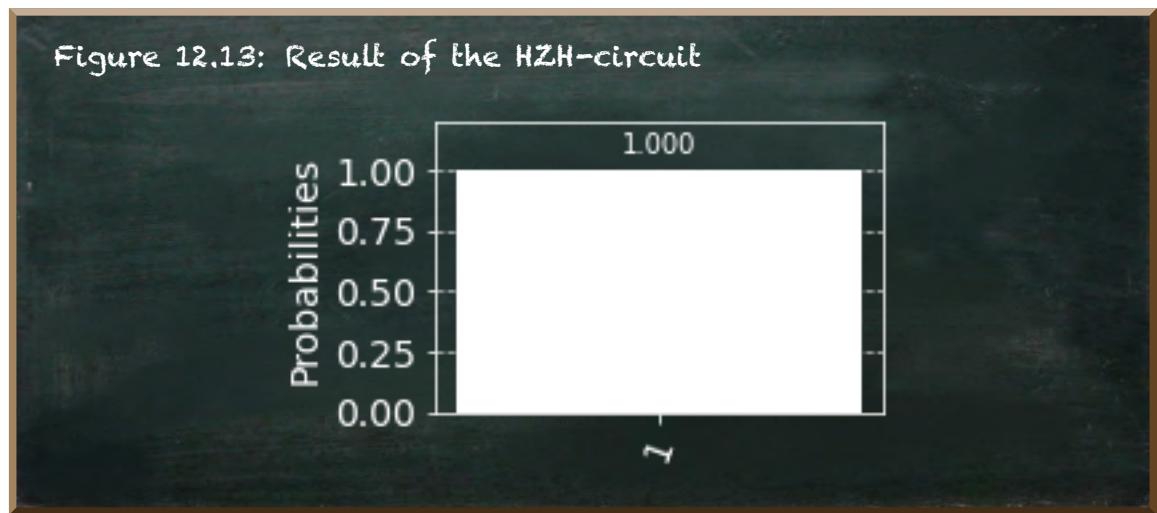Figure 12.12: Circuit with an oracle and a diffuser

This circuit results in state $|1\rangle$ with absolute certainty.

Listing 12.2: HZH-circuit

```
1  # prepare the circuit
2  qc = QuantumCircuit(1)
3  qc.h(0)
4  qc.z(0)
5  qc.h(0)
6
7  # execute the qc
8  results = execute(qc,Aer.get_backend('statevector_simulator')).result()
9  plot_histogram(results.get_counts())
```


Figure 12.13: Result of the HZH-circuit

We end up with our qubit in the desired state $|1\rangle$ because the Hadamard gate turns the state $|-\rangle$ into $|1\rangle$. The following figure depicts the rotations we apply in this circuit.

Figure 12.14: Transformations of the HZH-circuit
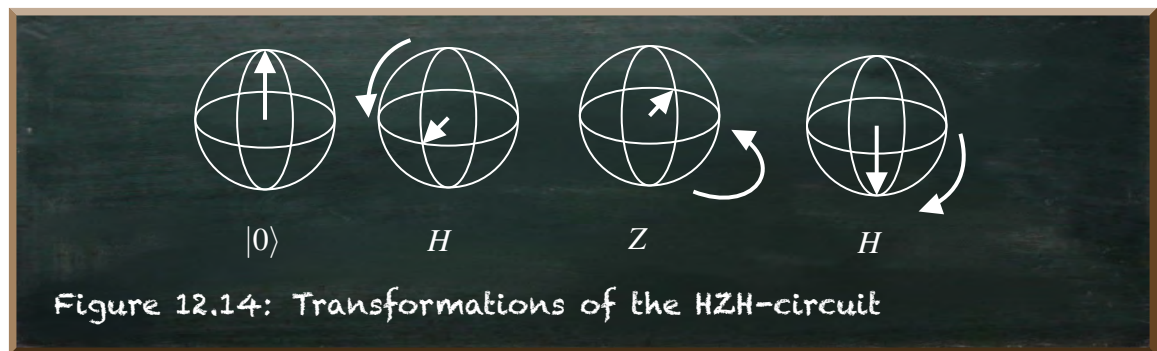
We applied an important sequence on the qubit, the HZH-circuit. This circuit is known as an identity to the NOT-gate (X-gate) that turns state $|0\rangle$ into $|1\rangle$ and vice versa.

The following equation proves this identity.

$$HZH = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = X$$

Then, why would we use the HZH-sequence? If it is similar to the NOT-gate, why don't we use that instead?

Simply put, the HZH-sequence is more flexible. It is the simplest form of Grover's search algorithm. It starts with all states being equal (the first H-gate). It applies an oracle (Z-gate). And, it uses a diffuser that amplifies the amplitude of the selected state $|1\rangle$ (the second H-gate).

To prove this flexibility, let's say we wanted to select state $|0\rangle$ instead. This is the task of the oracle. The starting state and the diffuser remain untouched. The oracle for state $|0\rangle$ is the I-gate. Or, simply doing nothing.

Figure 12.15: Circuit with an oracle and a diffuser

This circuit results in state $|0\rangle$ that we measure as 0 with absolute certainty.
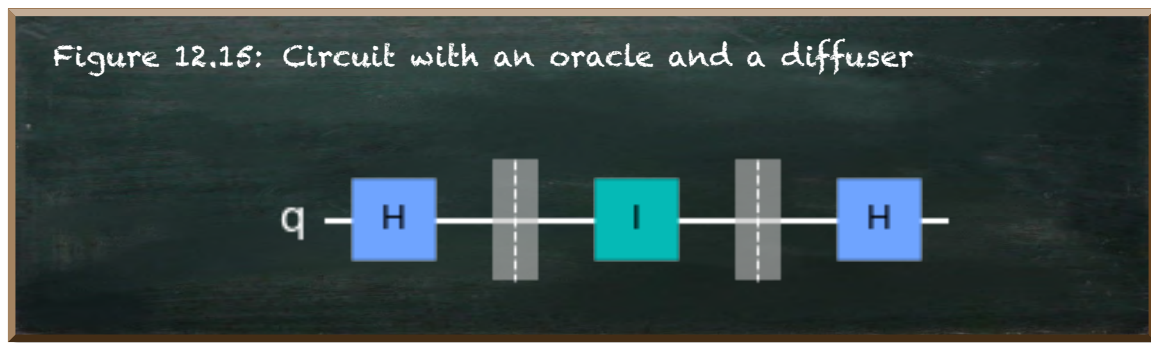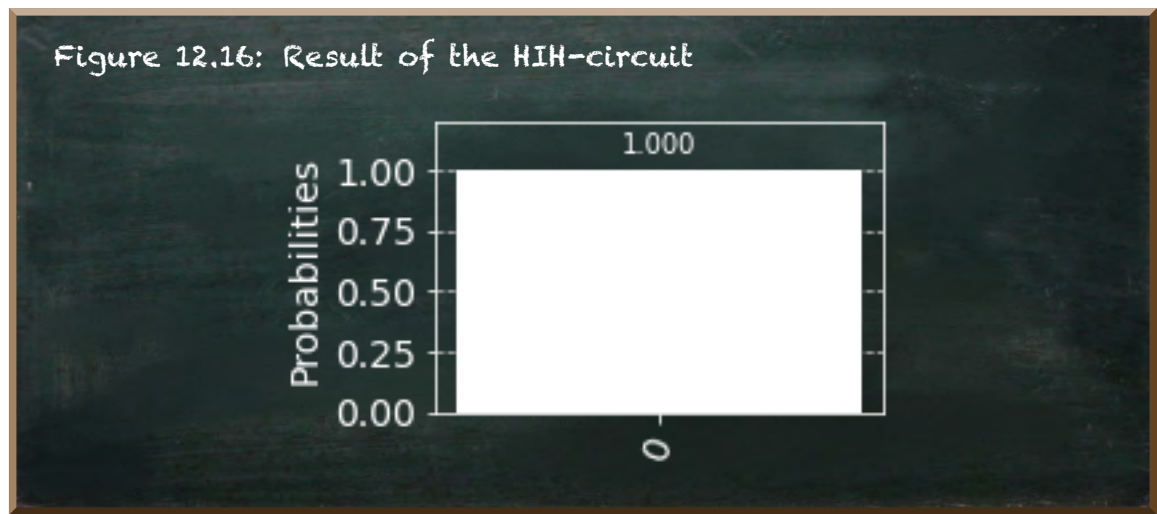
Listing 12.3: HIH-circuit

```
1  # prepare the circuit
2  qc = QuantumCircuit(1)
3  qc.h(0)
4  qc.i(0)
5  qc.h(0)
6
7  # execute the qc
8  results = execute(qc,Aer.get_backend('statevector_simulator')).result()
9  plot_histogram(results.get_counts())
```

Figure 12.16: Result of the HIH-circuit

On closer inspection, we can easily see this circuit does nothing at all. The *I*-gate does nothing, and the Hadamard-gate reverts itself. Thus, we end up in the default initialization state $|0\rangle$.

Figure 12.17: HIH-sequence
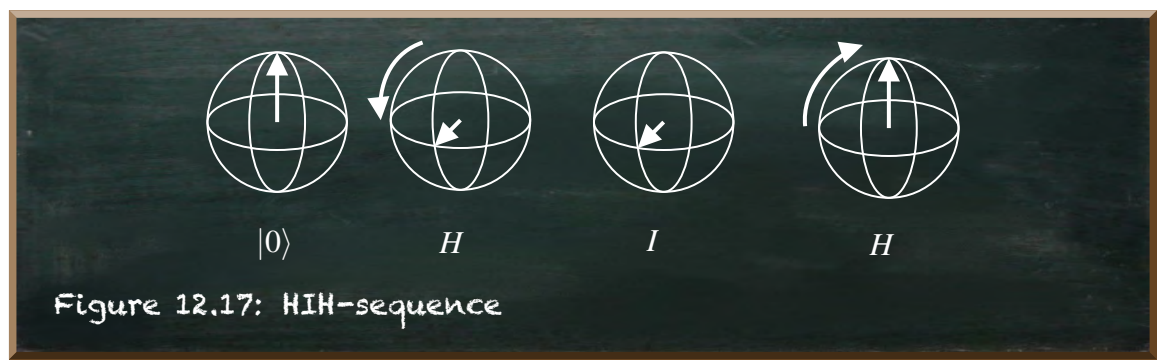
While we could rewrite these two circuits more succinctly, the circuit identities of $HZH = X$ and $HIH = I$ let us use the general structure of Grover's algorithm. Simply by changing the oracle, we can mark and amplify different states. We don't need to develop a new algorithm for each possible state we want to select out of a list. But we only need to find an appropriate oracle.

This ability comes in handy the more states our quantum system has.

The search for one of two possible states does not even deserve to be called a search. But the example of marking state $|1\rangle$ foreshadows what's to come. We use a phase shift to increase the amplitude of the favorable state.

12.3 Two-Qubit Amplification

Let's continue with the more interesting circuit with two qubits and four possible states.

Again, we start with an equal superposition of all states. With four states, each state has an amplitude of 0.5 and a resulting measurement probability of 0.25.

Listing 12.4: Equal superposition of four states

```
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram, plot_bloch_multivector,
    plot_state_qsphere

qc = QuantumCircuit(2)
qc.h(0)
qc.h(1)

# execute the qc
results = execute(qc,Aer.get_backend('statevector_simulator')).result()
plot_histogram(results.get_counts())
```


Figure 12.18: Equal superposition of four states
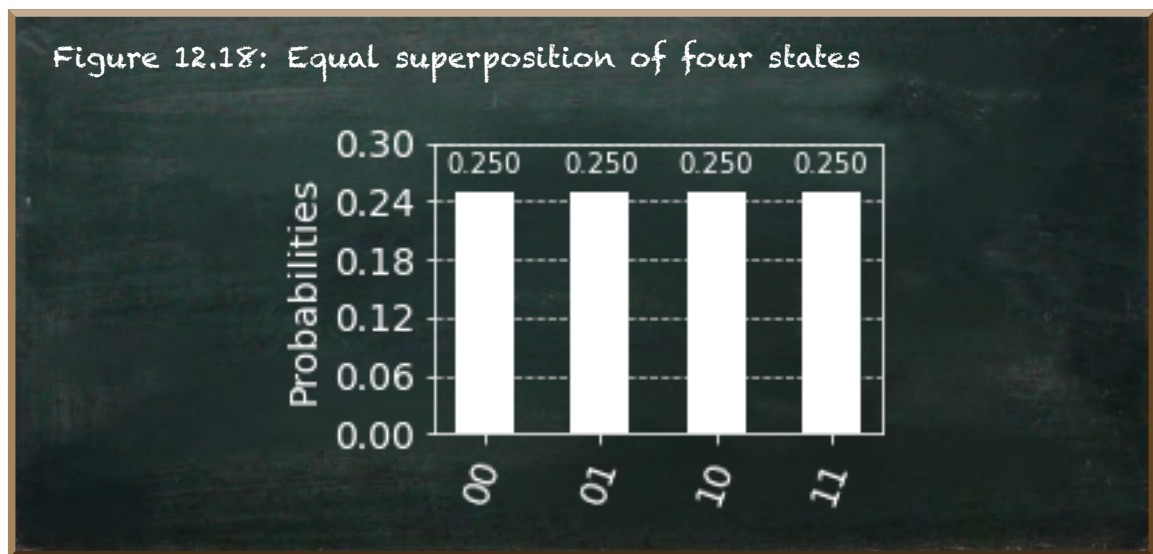
In the next step, we need to implement the oracle. Its purpose is to flip the amplitude of the favorable state. For example, let's say it is in state $|10\rangle$. Remember, we read the qubits from the right (qubit at position 0) to the left (qubit at position 1).

Figure 12.19: Two-qubit circuit with an oracle for |10>
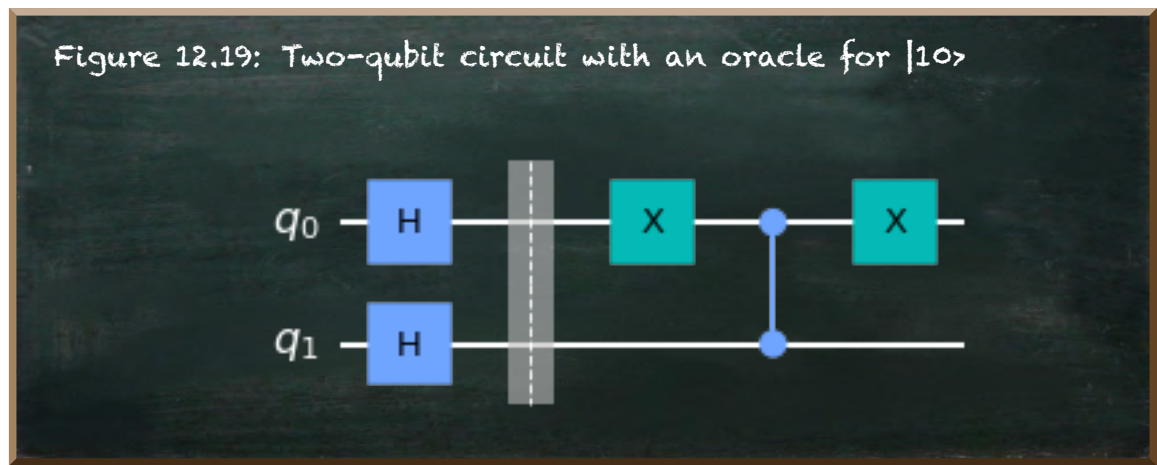
The controlled Z-gate (CZ) applies the Z-gate on the target qubit if the control qubit is in state $|1\rangle$. Thus, this gate applies the phase shift when both qubits are in state $|1\rangle$ as in state $|11\rangle$. By encapsulating the controlled Z-gate in NOT-gates that we apply on the first qubit, we select the state $|10\rangle$ instead of $|11\rangle$.

The following figure of the states and their phases confirms this effect. We flipped the phase of state $|10\rangle$. The CZ-gate entangles both qubits.

Figure 12.20: State |10> has a different phase

Now, we need a diffuser to amplify the amplitude. It starts with applying

Hadamard gates on all qubits. In the figure below, the diffuser begins right of the second vertical separator.

Figure 12.21: The diffuser starts with Hadamard gates
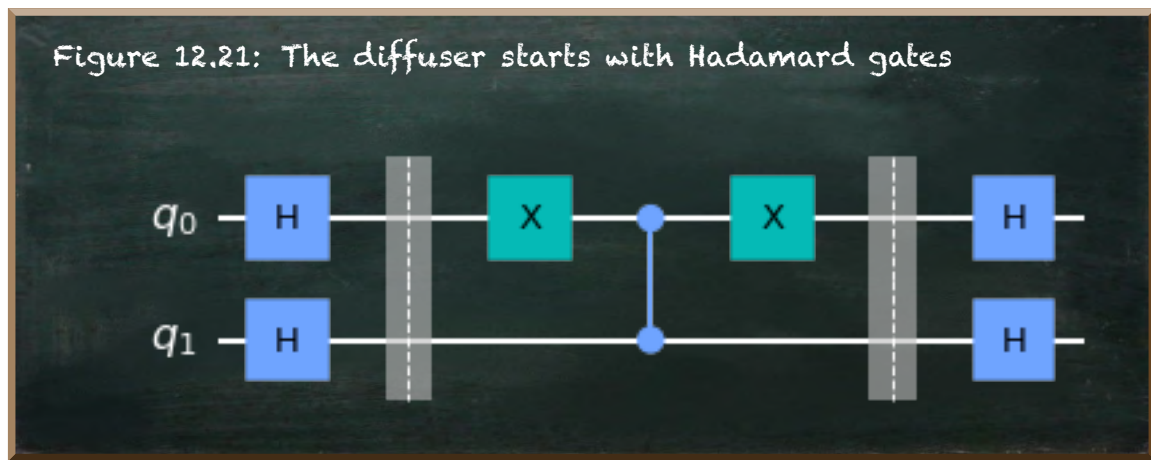
This series of Hadamard gates has quite an interesting effect. It switches the amplitudes of $|01\rangle$ and $|10\rangle$.

Figure 12.22: Switching the amplitudes of $|10\rangle$ and $|01\rangle$
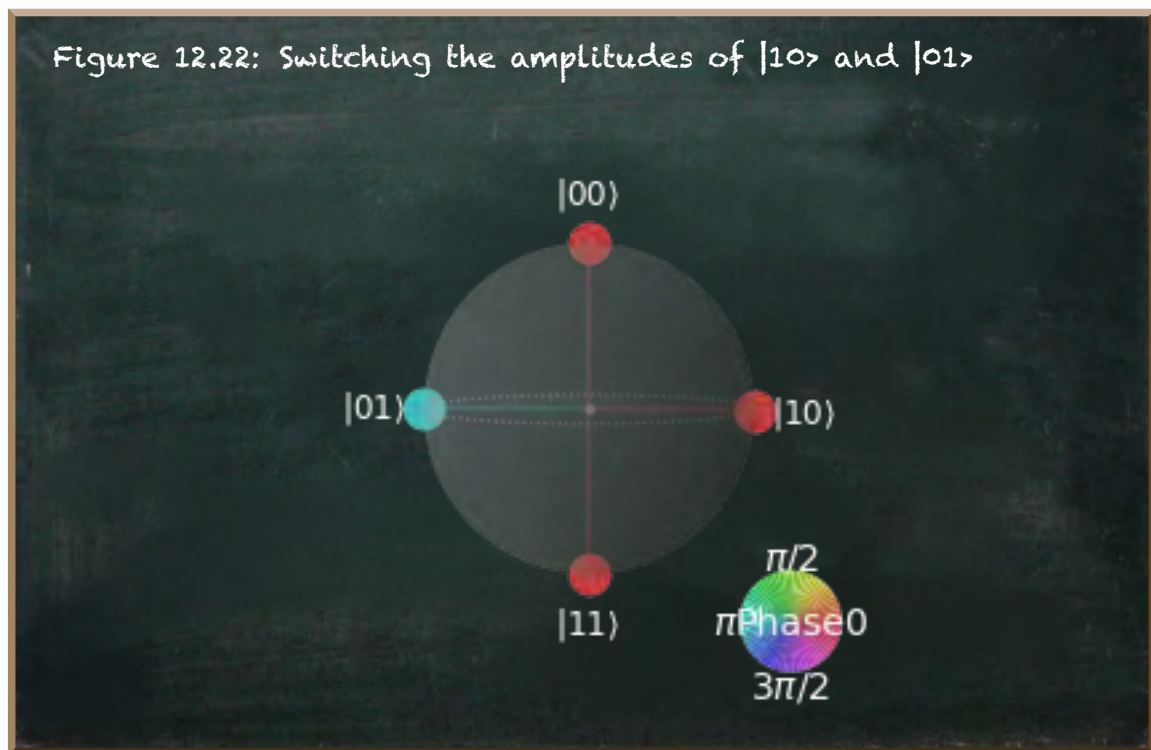
The successive sequence of a controlled Z-gate encapsulated in NOT-gates has a simple effect. It flips the amplitude of state $|00\rangle$. Furthermore, the controlled Z-gate unentangles both qubits again.

Figure 12.23: Flipping |00⟩ and unentangling the qubits

Figure 12.24: Effect of the diffuser
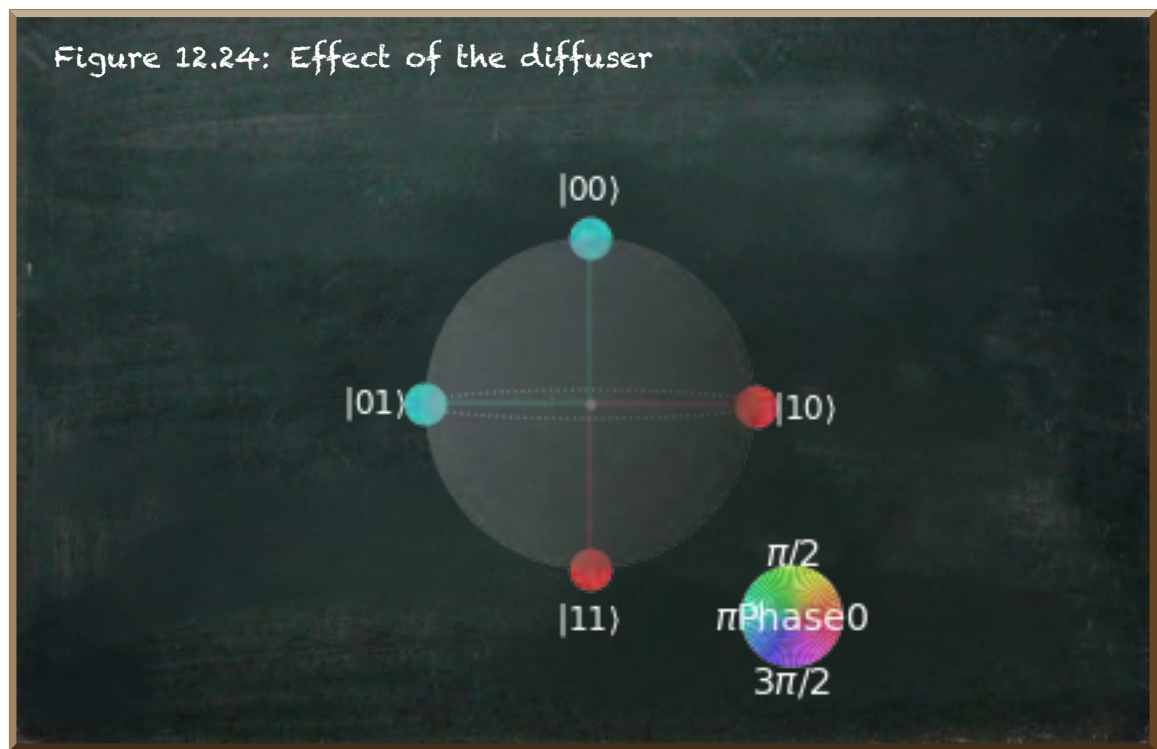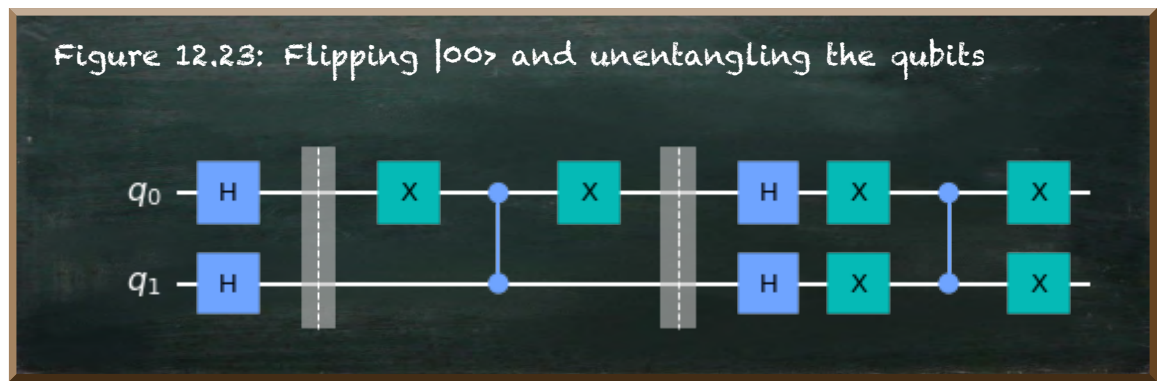
Now, states $|00\rangle$ and $|01\rangle$ share a phase and states $|10\rangle$ and $|11\rangle$ share a phase. Formulated differently, the first qubit (right-hand side of the notation) is in state $|+\rangle$ and the second qubit is in state $|-\rangle$.

When we apply Hadamard gates on both qubits, we turn the first qubit from $|+\rangle$ into $|0\rangle$, and we turn the second qubit from $|-\rangle$ into $|1\rangle$. We always measure the system in state $|10\rangle$–the state we marked through the oracle.

Listing 12.5: Two-qubit Grover searching |10>

```
 1  qc = QuantumCircuit(2)
 2
 3  qc.h(0)
 4  qc.h(1)
 5  qc.barrier()
 6
 7  qc.x(0)
 8  qc.cz(0, 1)
 9  qc.x(0)
10
11  qc.barrier()
12  qc.h(0)
13  qc.h(1)
14
15  qc.x(0)
16  qc.x(1)
17  qc.cz(0,1)
18  qc.x(0)
19  qc.x(1)
20
21  qc.barrier()
22  qc.h(0)
23  qc.h(1)
24
25  # execute the qc
26  results = execute(qc,Aer.get_backend('statevector_simulator')).result()
27  plot_histogram(results.get_counts())
```


Figure 12.25: Result of two-qubit Grover searching |10>

The following figure depicts the complete circuit.
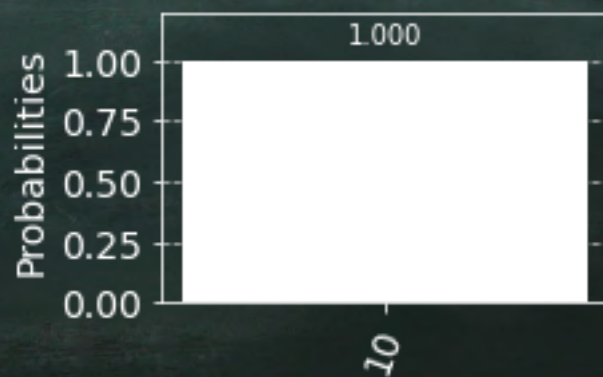
Figure 12.26: The complete circuit
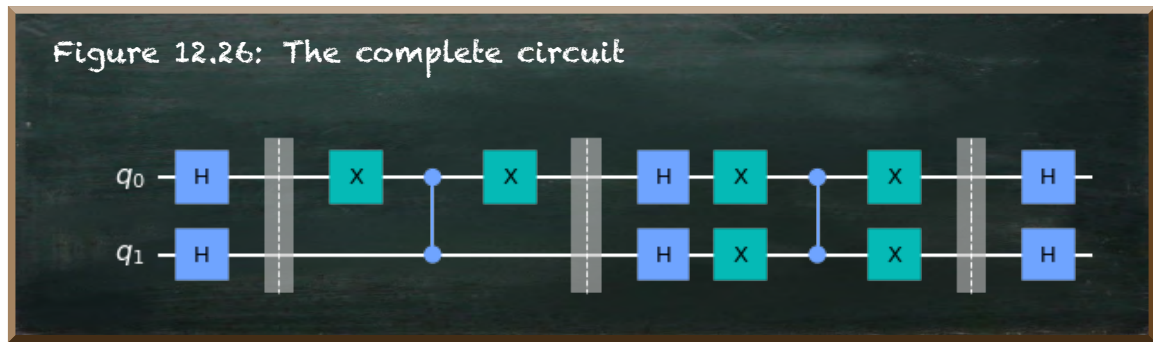
This circuit contains quite a few phase shifts until we apply the Hadamard-gates at the end, and the qubits result in the desired state that the oracle marked.

Most of the gates inside this circuit serve a technical purpose. For instance, the *CZ*-gate allows us to mark one single state in a multi-qubit system.

If we split the overall circuit into pieces and look at these pieces conceptually, we can discover a familiar structure. We see Hadamard gates at the start and the end. The center parts, consisting of the oracle and the diffuser, represent a *Z*-gate that we apply on the first qubit and an *I*-gate to apply on the second qubit.

The overall pattern of this circuit resembles an *HIH*-sequence we apply on the first qubit and an *HZH*-sequence we apply on the second qubit. The *I*-gate does not change a qubit, and the Hadamard gate reverts itself. Thus, the first qubit ends up in the same state it started with. That is $|0\rangle$. The *HZH*-gate is identical to a *NOT*-gate that turns the second qubit from its initial state $|0\rangle$ into $|1\rangle$.

Grover's circuit follows a simple idea. The first set of Hadamard gates bring the qubits to a state where their phases matter. The oracle shifts the phase of the searched state. Finally, the diffuser rearranges the phases of all states so that the latter Hadamard gates bring the qubits into the marked state.

The beauty of this circuit is that the only thing that changes is the oracle that marks the searched state. The diffuser remains the same.

Let's say we want to search a different state, $|01\rangle$. It is nearly similar to the circuit we just examined. The only difference is the oracle. And even this is almost alike. We use *NOT*-gates to shifts the phase of states where the second qubit is in state $|1\rangle$. The controlled *Z*-gate then flips the phase of state $|01\rangle$.

Figure 12.27: The oracle for |01⟩
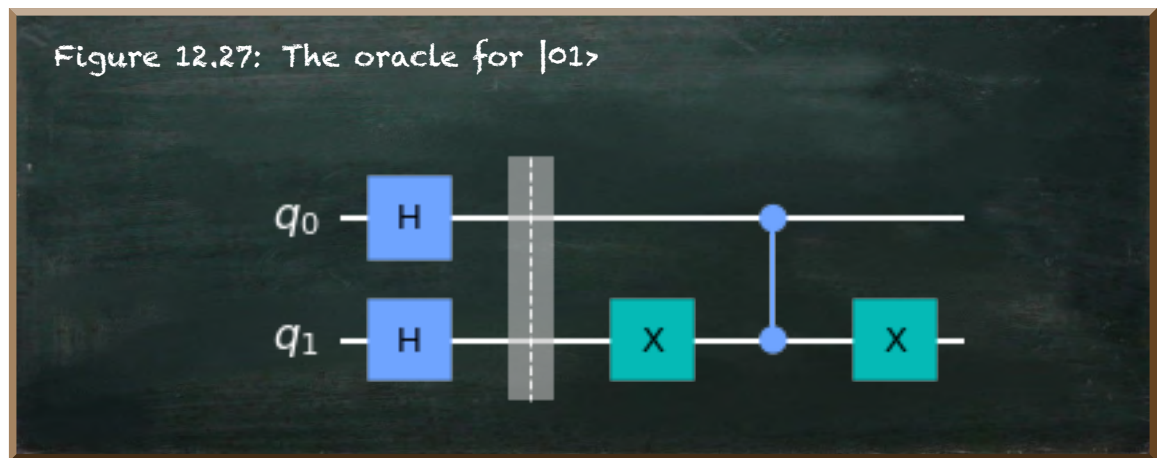
As a result, only state $|01\rangle$ has a shifted amplitude.

Figure 12.28: State |01⟩ has a different phase
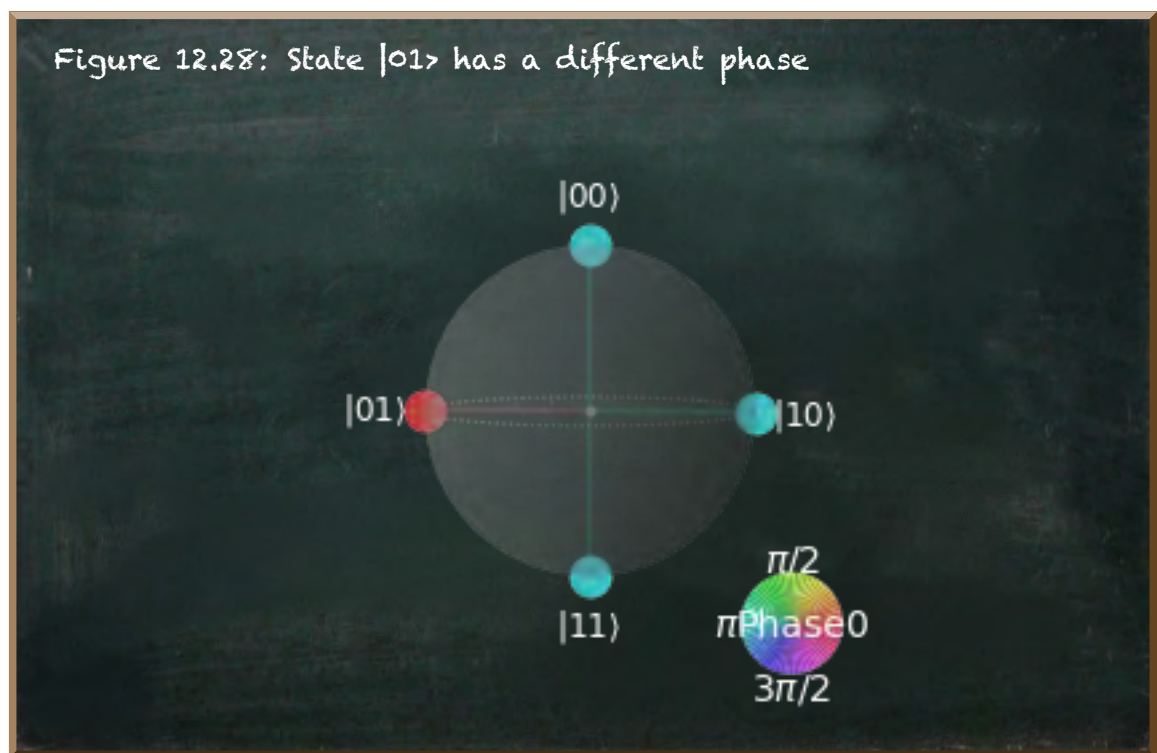
The diffuser is the same as in the previous case. The controlled Z-gate encapsulated in NOT-gates flips the phase of state $|00\rangle$ and unentangles both qubits again.

Figure 12.29: The effect of the Hadamard gates in the diffuser
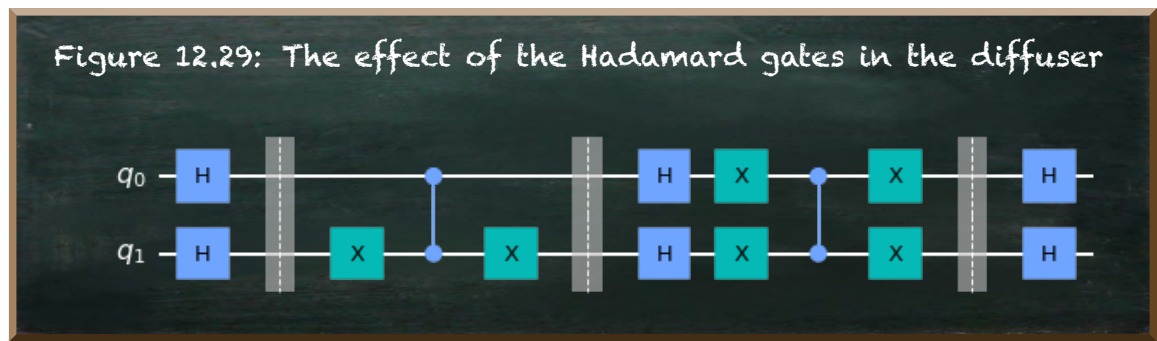
This time, after the diffuser is through, we see states $|00\rangle$ and $|10\rangle$ share a phase.

Figure 12.30: Effect of the diffuser when state $|01\rangle$ is marked
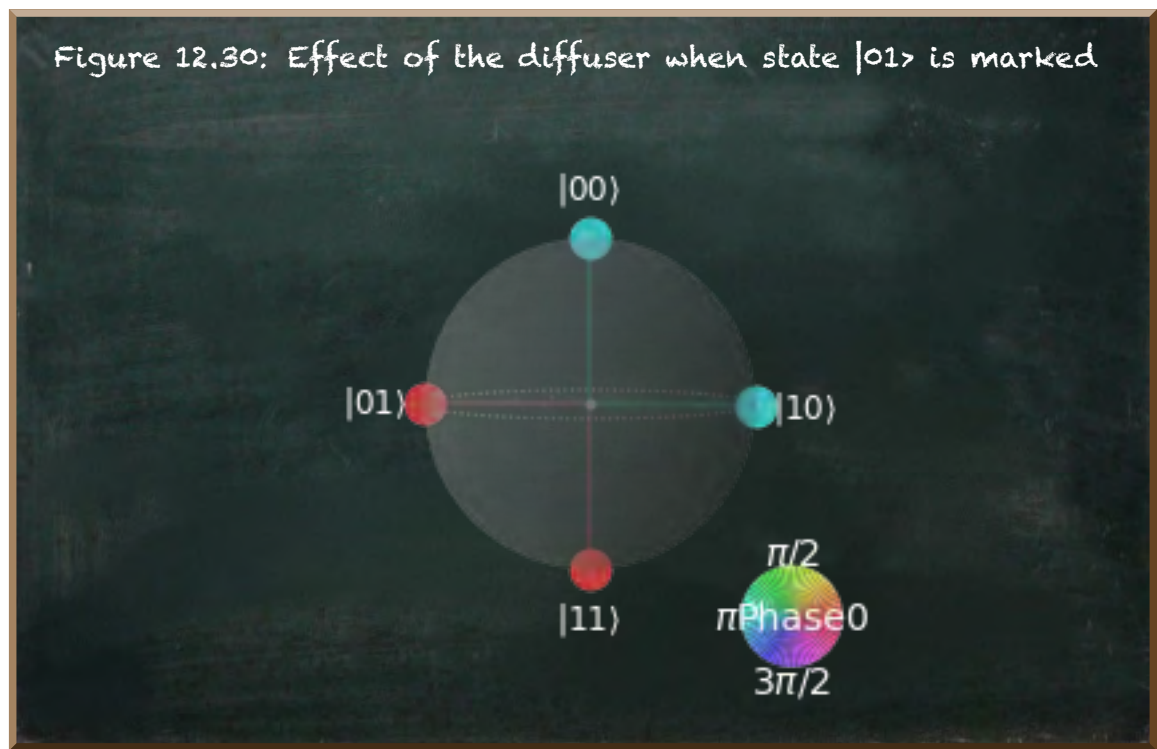
These phases correspond the first qubit in state $|-\rangle$ and the second qubit in state $|+\rangle$. The closing Hadamard gates turn both qubits into the respective basis states, again. We end up in state $|01\rangle$ (remember to read from right to left).

Figure 12.31: Result of two-qubit Grover searching |01>
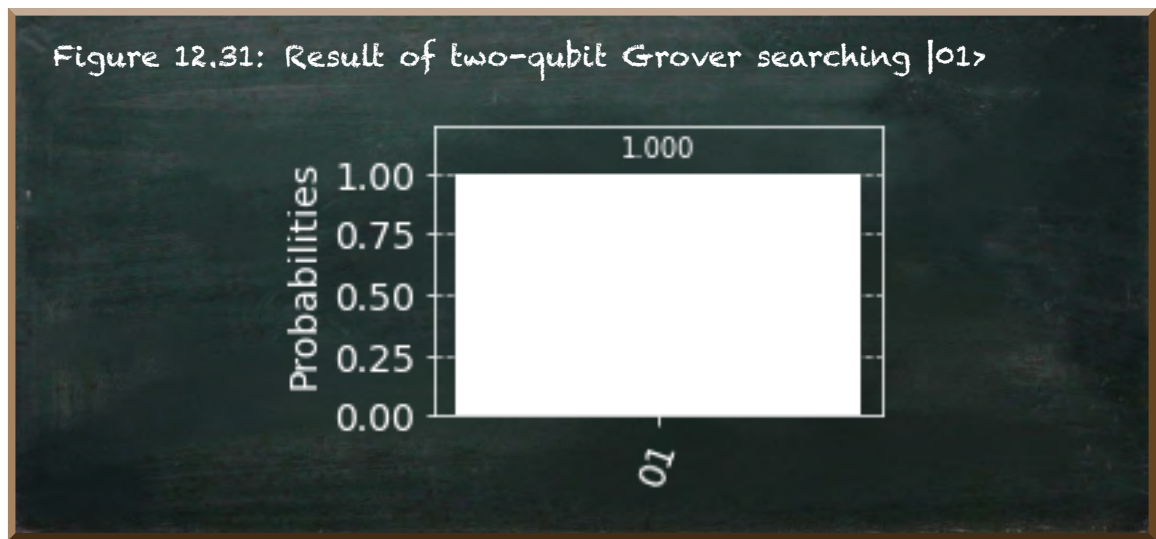
Again, the whole circuit represents an *HZH* sequence and an *HIH* sequence, respectively, that we apply to the two qubits.

In the following case, we look at is the magnification of state $|11\rangle$. Again, the only thing we need to change is the oracle. The oracle that selects state $|11\rangle$ is quite simple. We only apply the *CZ*-gate. It flips the phase of this state.

Figure 12.32: The full circuit to amplify |11>

Figure 12.33: The effect of the oracle for |11>
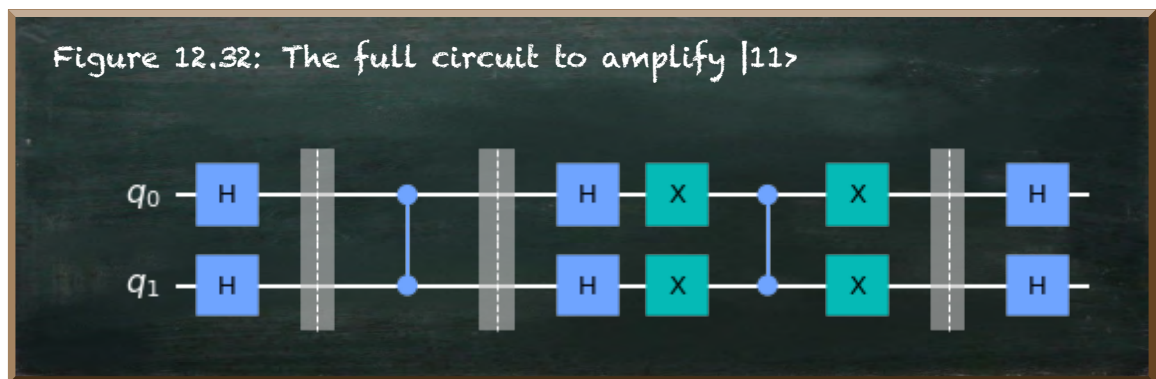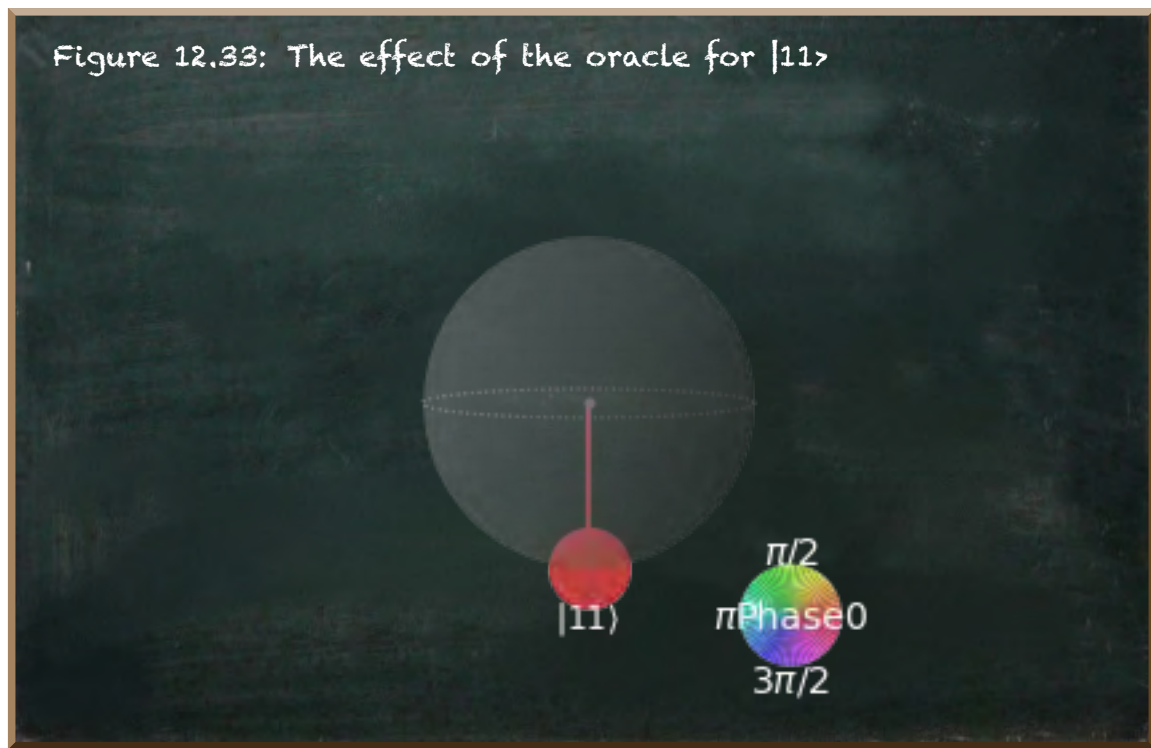
The diffuser is the same as in all other cases. It results in states $|00\rangle$ and $|11\rangle$ sharing a state as well as states $|10\rangle$ and $|01\rangle$.

Figure 12.34: Effect of the diffuser when state |01> is marked
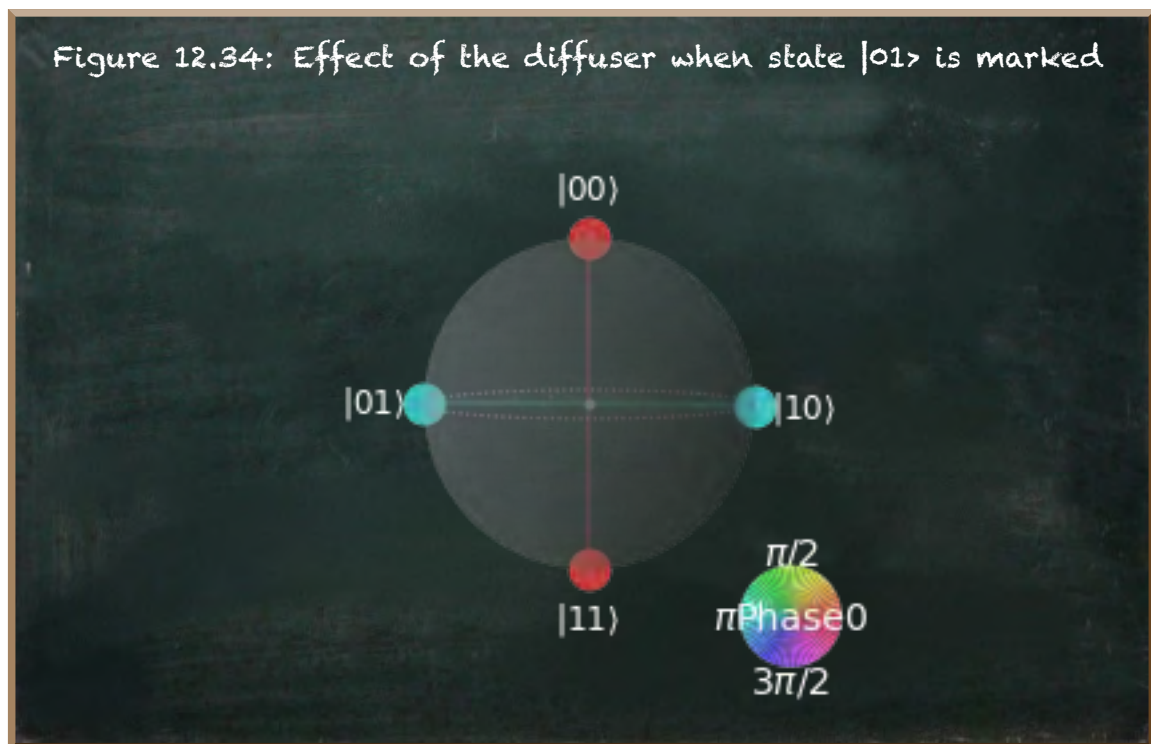
These states correspond to the two qubits in state $|-\rangle$ that the closing Hadamard gates turn into the final state $|11\rangle$.

Finally, there is state $|00\rangle$. This time, we wrap the *CZ*-gate into *NOT*-gates we apply to both qubits.

Figure 12.35: The full circuit to amplify $|00\rangle$
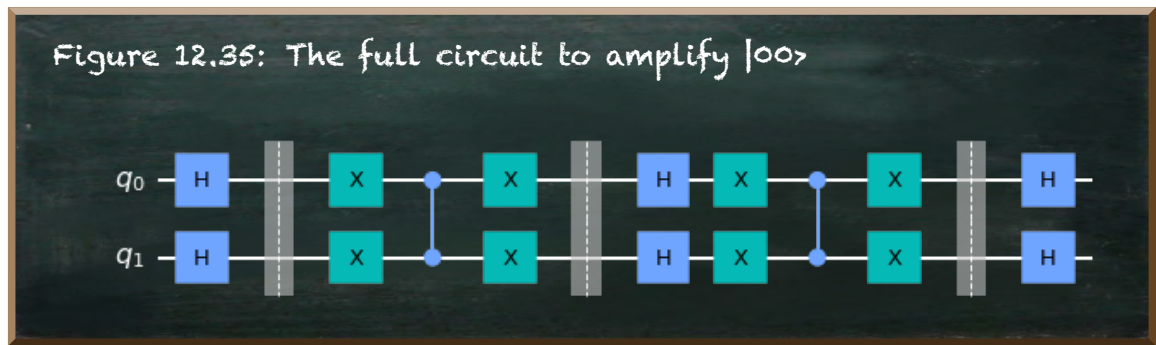
As a result of the oracle, only state $|00\rangle$ has a different phase.

Figure 12.36: Effect of the diffuser when state $|01\rangle$ is marked

As in all the other cases, the diffuser switches the phase of state $|00\rangle$. This time, we remove all phase flips before the final Hadamard gates bring back the qubits into basis states. Thus, we have not applied any *HZH*-circuit but two *HIH*-sequences.

Figure 12.37: Effect of the diffuser when state |00⟩ is marked
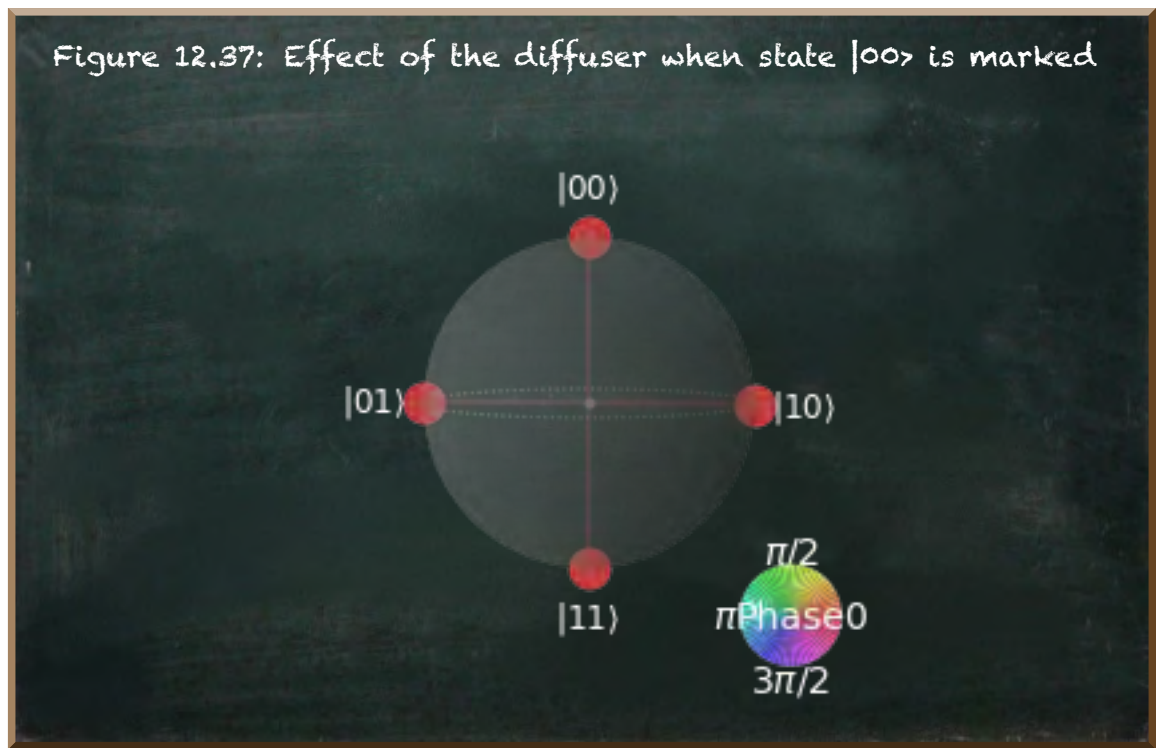
With only four possible states, we need to iterate through the oracle and the diffuser only once and end up in the searched state with absolute certainty.

13. Search For The Relatives

Let's get back to the Titanic. There are still plenty of ways how we could improve our QBN. A promising feature to include is the relations between passengers. Thus far, we have entirely ignored any family relationship. Yet, our dataset contains information about the number of siblings and spouses (`SibSp`) and the number of parents and children (`Parch`) traveling with a passenger.

Both fields, `SibSp` and `Parch` are numeric values denoting the number of a related passenger aboard the Titanic.

The following function lets us evaluate how a certain number of related passengers affects the chance to survive.

Listing 13.1: Convenience function to evaluate the effect of a relationship

```python
def evaluate_relation(relation, value):
    # separate the population
    population = train[train[relation].eq(value)] if value < 2 else train[
        train[relation].ge(value)]
    p = len(population)/len(train)

    # chance to survive
    surv = population[population.Survived.eq(1)]
    p_surv = len(surv)/len(population)
    return (p, p_surv)
```

The function `evaluate_relation` takes two parameters, the name of the `relation`

and the `value`. We start by separating the population from our training dataset (line 3). If the provided `value` is smaller than `2`, we select all passengers with this exact `value` for the given `relation`. We summarize all passengers with a `value` that is greater or equal to `2`. The marginal probability of having a certain number of related passengers is given by the size of the selected population divided by the size of the training dataset (line 4).

Next, we further separate the survivors of the population (line 7) and calculate the posterior probability of survival given that the passenger belongs to the respective population (line 8).

In the following snippet, we call this convenience function for the different groups.

Listing 13.2: The probabilities of the different populations

```
1  print("No Sibling: {:.2f} of the passengers, survival: {:.2f}".format(*
       evaluate_relation("SibSp", 0)))
2  print("One Sibling: {:.2f} of the passengers, survival: {:.2f}".format(*
       evaluate_relation("SibSp", 1)))
3  print("More Siblings: {:.2f} of the passengers, survival: {:.2f}".format(*
       evaluate_relation("SibSp", 2)))
4  print()
5  print("No Parent/Child: {:.2f} of the passengers, survival: {:.2f}".
       format(*evaluate_relation("Parch", 0)))
6  print("One Parent/Child: {:.2f} of the passengers, survival: {:.2f}".
       format(*evaluate_relation("Parch", 1)))
7  print("More Parents/Children: {:.2f} of the passengers, survival: {:.2f}".
       format(*evaluate_relation("Parch", 2)))
```

```
No Sibling: 0.68 of the passengers, survival: 0.35
One Sibling: 0.23 of the passengers, survival: 0.54
More Siblings: 0.08 of the passengers, survival: 0.27

No Parent/Child: 0.76 of the passengers, survival: 0.34
One Parent/Child: 0.13 of the passengers, survival: 0.55
More Parents/Children: 0.11 of the passengers, survival: 0.46
```

The number of relatives onboard the Titanic certainly affected whether a passenger survived. But instead of adding another feature to our QBN, we use what we have learned to find the relatives of a passenger. Let's have a look at a group of passengers.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
249	250	0	2	Carter, Rev. Ernest Courtenay	male	54.0	1	0	244252	26.0	NaN	S
390	391	1	1	Carter, Mr. William Ernest	male	36.0	1	2	113760	120.0	B96 B98	S
435	436	1	1	Carter, Miss. Lucile Polk	female	14.0	1	2	113760	120.0	B96 B98	S
763	764	1	1	Carter, Mrs. William Ernest (Lucile Polk)	female	36.0	1	2	113760	120.0	B96 B98	S
802	803	1	1	Carter, Master. William Thornton II	male	11.0	1	2	113760	120.0	B96 B98	S
854	855	0	2	Carter, Mrs. Ernest Courtenay (Lilian Hughes)	female	44.0	1	0	244252	26.0	NaN	S

Apparently, all these passengers share the same last name, Carter. They all have one sibling or spouse. Four of them have two children or parents. Two have none.

By looking at the `Ticket` number and class (`Pclass`), the `Fare`, and the `Cabin`, we can distinguish two families named Carter. Mr. and Mrs. Ernest Courtenay Carter (`PassengerIds` 249 and 854) are a couple without children.

The other four Carters shared the same `Cabins` and `Ticket`. So, it is pretty safe to assume they are a family, too. When we look at their age and gender, we can conclude that the passengers with the `PassengerIds` 390 and 763 (both aged 36) are married (they are their respective `SibSp`). And, they are the parents of the passengers with the `PassengerIds` 435 (age 14) and 802 (age 11). These two children are their respective siblings (`SibSp`).

A look at the exact names of the passengers confirms this assumption. In 1912, relationships were a lot more conservative. I believe there were no same-sex marriages, and the wife took over her husband's full name (at least in terms of our dataset). Further, the formal title of a female passenger indicated whether she was married. "Miss" had been the traditional title for an unmarried woman. "Mrs.," on the other hand, refers to a married woman. Further, while the title Master indicates a boy, the title "Mr." not necessarily a married man.

Let's look at the Goodwins.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
59	60	0	3	Goodwin, Master. William Frederick	male	11.0	5	2	CA 2144	46.9	NaN	S
71	72	0	3	Goodwin, Miss. Lillian Amy	female	16.0	5	2	CA 2144	46.9	NaN	S
386	387	0	3	Goodwin, Master. Sidney Leonard	male	1.0	5	2	CA 2144	46.9	NaN	S
480	481	0	3	Goodwin, Master. Harold Victor	male	9.0	5	2	CA 2144	46.9	NaN	S
678	679	0	3	Goodwin, Mrs. Frederick (Augusta Tyler)	female	43.0	1	6	CA 2144	46.9	NaN	S
683	684	0	3	Goodwin, Mr. Charles Edward	male	14.0	5	2	CA 2144	46.9	NaN	S

The last one on the list, Mr. Charles Edward Goodwin, has the title "Mr." at the age of 14. He has five siblings and two parents. Since his siblings have five siblings, too, it is safe to assume he is not married.

The data of this family further reveals we are coping with only a partial list of the passengers. We are missing Mr. Frederick Goodwin and a child. However, even the incomplete information we can draw from a passenger's related travelers is clear. If we'd be tasked with predicting the survival of Mr. Frederick Goodwin, the father of five children and the husband of a wife who all died that night, the answer is pretty clear.

Passenger relationships are a means to improve the performance of our classifier. Especially when we consider whether the related passengers survived or not, it confronts us with the challenge to find the correct – or most likely – related passenger first. The conservative family structures and naming conventions facilitate the search. Think about modern cultures. Today, we don't call a woman by the name of her husband anymore. Spouses may keep their surnames. And we see patch-work families where each spouse brings his or her children from an earlier marriage. To find the person that is most likely the related one, we need to consider all passengers and numerous combinations.

This sounds like a good case to apply Grover's search algorithm.

13.1 Turning the Problem into a Circuit

Our dataset contains roughly 900 of the 1,300 passengers of the Titanic. The columns SibSp and Parch indicate the number of the respective relatives a passenger has among the other passengers.

For instance, Rev. Ernest Courtenay Carter has one sibling or spouse but no parents or children traveling with him. But we don't know who the relative is. We don't even know whether s/he is in our part of the data or not.

Listing 13.3: Passenger no 250

```
1 train[train["PassengerId"].eq(250)]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
249	250	0	2	Carter, Rev. Ernest Courtenay	male	54.0	1	0	244252	26.0	NaN	S

To find the respective relative, we need to consider three columns of interest. These are the name, the ticket, and the cabin.

First and foremost, we expect two relatives to have the same last name. Back in 1912, family structures were very conservative. Let's look at all the passengers who share the last name with Mr. Carter.

Listing 13.4: Get potential relatives

```
1 current_passenger = train[train["PassengerId"].eq(250)]
2 last_name = current_passenger.Name.to_string(index=False).split(',')[0]
3 train[train["Name"].str.contains(last_name)]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
249	250	0	2	Carter, Rev. Ernest Courtenay	male	54.0	1	0	244252	26.0	NaN	S
390	391	1	1	Carter, Mr. William Ernest	male	36.0	1	2	113760	120.0	B96 B98	S
435	436	1	1	Carter, Miss. Lucile Polk	female	14.0	1	2	113760	120.0	B96 B98	S
763	764	1	1	Carter, Mrs. William Ernest (Lucile Polk)	female	36.0	1	2	113760	120.0	B96 B98	S
802	803	1	1	Carter, Master. William Thornton II	male	11.0	1	2	113760	120.0	B96 B98	S
854	855	0	2	Carter, Mrs. Ernest Courtenay (Lilian Hughes)	female	44.0	1	0	244252	26.0	NaN	S

We select Mr. Carter by his `PassengerId` (250) (line 1). In our dataset, the `Name` column contains the last name and the first name(s) divided by a comma(,). Thus, we split the column at the comma and use the first part (at position `0`) (line 2). Finally, we extract all passengers whose name contains this last name (line 3).

We get a list of six passengers, including Mr. Carter.

While the full name of passenger 855 (Mrs. Ernest Courtenay Carter) identifies her as the spouse of Mr. Rev. Ernest Courtenay Carter, we don't want to mess around with the name too much. Instead, let's look at the `Ticket` number.

In general, the algorithm we're about to program is a variational hybrid quantum-classical algorithm consisting of a pre-processing, the parameterized quantum circuit, and the post-processing.

To keep our example small and concise, we pre-select the passengers before we let the parameterized quantum circuit select the correct one. We only consider passengers who share the same name or the same ticket.

It could also make sense to evaluate whether the passengers share the same `Cabin`. But when we look at our data, we can see we only have the `Cabin` for 204 passengers. Therefore, we don't consider this feature now.

Listing 13.5: A look at the data

```
1 train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   PassengerId  891 non-null     int64
 1   Survived     891 non-null     int64
 2   Pclass       891 non-null     int64
 3   Name         891 non-null     object
 4   Sex          891 non-null     object
 5   Age          714 non-null     float64
 6   SibSp        891 non-null     int64
 7   Parch        891 non-null     int64
 8   Ticket       891 non-null     object
 9   Fare         891 non-null     float64
 10  Cabin        204 non-null     object
 11  Embarked     889 non-null     object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Since we do not want to find the passenger whose relative we search, we exclude this passenger from our list.

Listing 13.6: The possible relatives of Mr. Rev. Ernest Courtenay Carter

```
1  ticket = current_passenger["Ticket"].to_string(index=False)
2  passengerId = current_passenger["PassengerId"]
3
4  group = train[
5    train["PassengerId"].ne(passengerId) & (
6    train["Name"].str.contains(last_name) |
7    train["Ticket"].eq(ticket)
8  )]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
390	391	1	1	Carter, Mr. William Ernest	male	36.0	1	2	113760	120.0	B96 B98	S
435	436	1	1	Carter, Miss. Lucile Polk	female	14.0	1	2	113760	120.0	B96 B98	S
763	764	1	1	Carter, Mrs. William Ernest (Lucile Polk)	female	36.0	1	2	113760	120.0	B96 B98	S
802	803	1	1	Carter, Master. William Thornton II	male	11.0	1	2	113760	120.0	B96 B98	S
854	855	0	2	Carter, Mrs. Ernest Courtenay (Lilian Hughes)	female	44.0	1	0	244252	26.0	NaN	S

No other passenger shares the ticket with Mr. Rev Ernest Courtenay Carter. Thus, we do not need to add another passenger to the list of five passengers depicted above.

Usually, we will find less than 8 passengers who might be the searched relative. Therefore, three qubits suffice ($2^3 = 8$). We encode each passenger by a binary string that represents the measurements of the qubits.

For instance, we encode the first passenger (Mr. William Ernest Carter) with the binary string `000`, the second with the string `001`, the third with `010`, and so on. We encode the fifth passenger Mrs. Ernest Courtenay (Lilian Hughes) Carter, who is the searched relative, with the string `100`.

Listing 13.7: Encoding the passengers

```
1  # number of qubits to represent considered passengers
2  QUBITS = 3
3
4  def encode(pos):
5    bpos = "{:0{}b}".format(pos, QUBITS)
6    return bpos
7
8
9  number_of_rows = len(group.index)
10 for pos in range(0, 2**QUBITS):
11   if pos >= number_of_rows:
12     break
13
14   passenger = group.iloc[[pos]]
15   print(pos, encode(pos), passenger.Name.to_string(index=False))
```

```
0 000 Carter, Mr. William Ernest
1 001 Carter, Miss. Lucile Polk
2 010 Carter, Mrs. William Ernest (Lucile Polk)
3 011 Carter, Master. William Thornton II
4 100 Carter, Mrs. Ernest Courtenay (Lilian Hughes)
```

The `encode` function takes a passenger as its parameters (line 4). It reformats the position into a binary string (line 5) and returns this binary position (line 6).

We loop through each possible combination of qubit values we have (line 10). If our `group` is smaller (calculated at line 9) than the eight slots we have, we stop (lines 11-12). Otherwise, We select the passenger at the specified position (line 14) and show the data (line 15).

We're ready to create the quantum circuit. The search algorithm consists of three parts. First, we initialize the quantum circuit and put all qubits into an equal superposition using Hadamard gates.

Listing 13.8: Equal superposition of the Hadamard gates

```
 1  RULES=2
 2
 3  q_pass = QuantumRegister(QUBITS, name='passengers')
 4  q_rules = QuantumRegister(RULES, name='rules')
 5
 6  qc = QuantumCircuit(q_pass, q_rules)
 7
 8  # put passenger qubits into superposition
 9  qc.h(q_pass)
10  qc.barrier()
11
12  qc.draw()
```


Figure 13.1: Equal superposition of the Hadamard gates
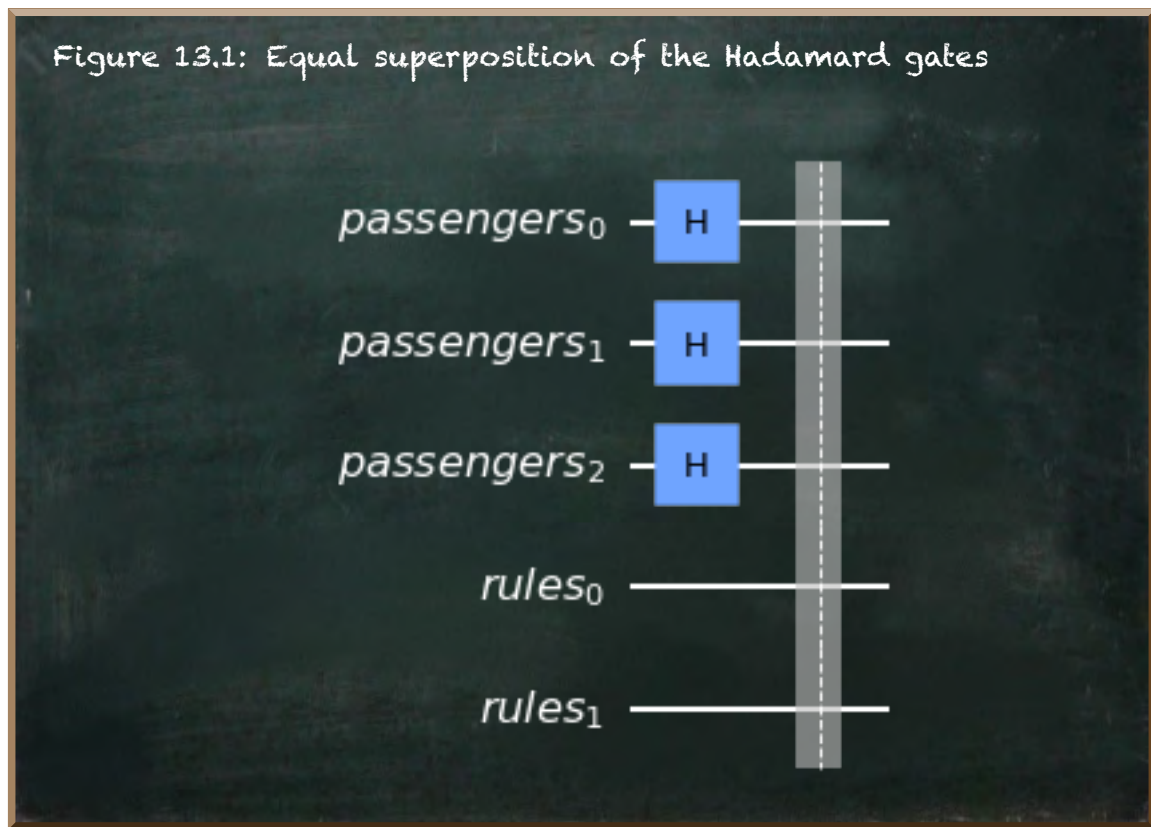
Our quantum circuit will become quite long in comparison to the circuits

thus far. It is time to be a little more concise in our commands. Rather than applying a Hadamard-gate on each qubit separately, we pass the whole `QuantumRegister` as an argument (line 9). It applies Hadamard-gates on all qubits in the register.

Registers are a convenient way to work with qubits and to address them. In our case, we use three qubits (in the `q_pass`-register) to represent the passengers. We also specified another register, the `q_rules`. We can use as many registers as we want in our `QuantumCircuit`.

The qubits in the `q_rules` register serve to represent the rules that govern who is the searched relative. We reserve one qubit for each attribute we want to consider (`Name`, `Ticket`, and `Cabin`).

The second and most crucial part of the algorithm is the oracle. Similar to the oracles in the previous sections, we have to negate the amplitude of the passenger we want to select.

The approach we apply inside our oracle is the following. We loop through all the passengers who might be the relative. For each passenger, we evaluate whether he or she shares the `Name` and the `Ticket` separately. If he or she does, we switch the corresponding qubit in the `q_rules` register.

Since we will use the oracle quite a few times in our circuit, we put it into the function `oracle`.

The `oracle` function takes four mandatory parameters and one optional parameter. The `passenger` denotes the passenger whose relative we search. The `group` is the excerpt of the Pandas `dataframe` with the potential candidates of being the relative. `q_p` is the `QuantumRegister` representing the passengers. `q_r` is the `QuantumRegister` representing the rules. These are all mandatory parameters. Finally, the `draw` is an optional boolean parameter.

We create a separate `QuantumCircuit` (line 3) that includes the two registers we took as parameters in this function. The oracle loops through all possible states we can represent with the specified number of qubits (line 6). With `QUBITS=3`, there are `2**3=8` (2^3) states. We stop once the position of the current state exceeds the number of passengers we consider (lines 7-8).

For each used state that represents a passenger, we calculate the binary position (line 10), select the state that represents this passenger (line 13), apply the rules that govern whether the passenger is the relative (line 16), and unselect the passenger again (line 19).

Listing 13.9: The oracle-function

```python
1  def oracle(passenger, group, q_p, q_r, draw=False):
2    # Create a sub-circuit
3    o_qc = QuantumCircuit(q_p, q_r)
4
5    # loop through all passengers
6    for pos in range(0, 2**QUBITS):
7      if pos >= len(group.index):
8        break
9
10     bpos = encode(pos)
11
12     # select the state representing the passenger
13     select_state(bpos, o_qc, q_p)
14
15     # apply the rules
16     apply_rules(passenger, group.iloc[[pos]], o_qc, q_p, q_r)
17
18     # un-select the state representing the passenger
19     select_state(bpos, o_qc, q_p)
20
21     if draw:
22       o_qc.barrier()
23
24   if draw:
25     return o_qc.draw()
26   else:
27     # We return the oracle as a gate
28     Oracle = o_qc.to_gate()
29     Oracle.name = "oracle"
30     return Oracle
```

The result of the `oracle` function depends on the `draw` parameter. If it is `True`, we call the circuit's `draw` function. If it is `False`, we turn the whole orcale circuit into a custom quantum gate (line 28) and specify a custom name (line 29). Custom gates can't include barriers. Therefore, we only structure the visual representation of the oracle subcircuit, if we draw it (lines 21-22).

The apparent question is how do we select a state and apply the rules. We postponed the respective implementation by calling functions. Let's have a look.

Listing 13.10: The select state function

```python
def select_state(bpos, qc, qubits):
  for i in range(0, QUBITS):
    if bpos[::-1][i] == "0":
      qc.x(qubits[i])
```

The `select_state` function selects or unselects a state. It takes the binary position, the quantum circuit (`qc`), and the `qubits` representing the passengers as parameters. This function loops through all the qubits (line 2) we use for representing a passenger. If the binary string has a `0` at the position the qubit represents, we apply the `x`-gate on this qubit.

For instance, this function applies `x`-gates on all qubits if we select the state `000`. It applies none if we select state `111`. And, it applies an `x`-gate on the first qubit (at position `0`) if we select the state `110`.

At line 3, we reverse the binary string (`bpos[::-1]`). As a result, we read it from right to left. This is the order of the qubits.

The next function depicts the application of the rules that govern whether a passenger is the searched relative.

The `apply_rules` function takes as parameters the `passenger` whose relative we search, the `current` passenger who might be the relative, the `QuantumRegister` representing the passengers, and the `QuantumRegister` representing the rules.

It applies two rules. First, we evaluate whether the `passenger` and the `current` share the same `Ticket` (line 5). The `Ticket` is a `string`. Therefore, we parse the Pandas rows into a simple string (removing the `index`). If we kept the `index`, we would never find a match because the two passengers have different positions in the `dataframe`. If both passengers share the same `Ticket`, we apply the `mcx`-gate. This is a convenience function of the Qiskit `QuantumCircuit` that represents a multi-controlled-X-gate. Thus far, we only used gates with a single or two control qubits (`cx` or `ccx`). Now, we have three qubits we want to control whether to switch a fourth (the target) qubit.

The underlying rationale is the following one. Before we call the `apply_rules`-function, the `oracle` selects the state representing the current passenger. Thus, all the three qubits representing the corresponding state are in state $|1\rangle$. The `mcx`-gate with the `q_pass`-`QuantumRegister` as contol qubits, and the first qubit of the `q_rules`-`QuantumRegister` as the target-qubit flips the state of this target from $|0\rangle$ to $|1\rangle$.

So, if the two passengers share the same `Ticket`, we flip the qubit that repre-

sents this first rule to $|1\rangle$.

We do the same thing for the second rule (lines 9-10). We check whether both passengers share the same last name. Again, we convert the row into a `string`. Further, we split the name into the last name (before the comma) and the first names (after the comma). By using the part at position `[0]`, we only use the last name.

Listing 13.11: Apply the rules

```
1  from qiskit.circuit.library import ZGate
2
3  def apply_rules(passenger, current, qc, q_p, q_r):
4    # apply first rule
5    if passenger.Ticket.to_string(index=False) == current.Ticket.to_string(
       index=False):
6      qc.mcx(q_p, q_r[0])
7
8    # apply second rule
9    if passenger.Name.to_string(index=False).split(',')[0] == current.Name.
       to_string(index=False).split(',')[0]:
10     qc.mcx(q_p, q_r[1])
11
12   # all conditions must be met
13   qc.append(ZGate().control(QUBITS+RULES-1), [*q_p, *q_r])
14
15   # unapply second rule
16   if passenger.Name.to_string(index=False).split(',')[0] == current.Name.
       to_string(index=False).split(',')[0]:
17     qc.mcx(q_p, q_r[1])
18
19   # unapply first rule
20   if passenger.Ticket.to_string(index=False) == current.Ticket.to_string(
       index=False):
21     qc.mcx(q_p, q_r[0])
```

Next, we apply another multi-controlled gate (line 13). This is a z-gate controlled by all except one qubit. The one exception is the target qubit. Remember, for a controlled *Z*-gate, it does not matter which is the control qubit and which is the target qubit because it flips the amplitude of the state where all qubits are in state $|1\rangle$.

Qiskit provides the `ZGate`-function (that we import from qiskit.circuit.library, line 1). We add controls to the gate by calling `control(x)` on it, with `x` is the number of controls we want to add. We want to use all qubits but one as control

qubits. Therefore, we specify QUBITS+RULES−1 as the number of controls. The last parameter we pass is a list of the qubits to apply the gate to. We create a list of all our qubits ([*q_p, *q_r]). The * operator unfolds a list into single items, in our case, the qubits. By surrounding all the qubits of the two registers by brackets ([...]), we create a new list of these qubits.

This multi-controlled Z-gate flips the amplitude of the state $|11111\rangle$. Thus, both qubits representing the rules must be in state $|1\rangle$, too.

Subsequently, we unselect the application of the rules again (lines 16-21). We do this the same way we applied the rules in the first place.

We're ready to have a look at our oracle. We pass the required parameters and set the draw parameter to True to get a drawing of this sub-circuit.

Listing 13.12: Showing the oracle circuit

```
1 oracle(current_passenger, group, q_pass, q_rules, True)
```


Figure 13.2: The oracle circuit
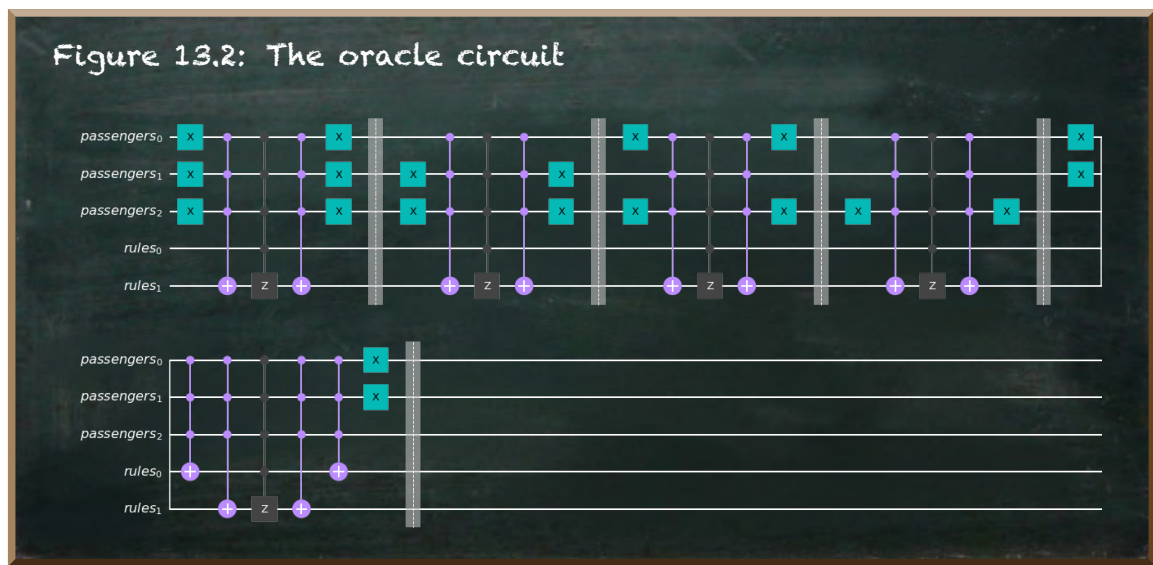
We divided each passenger by barriers (the vertical bars). The first passenger we evaluate has the binary string 000. Therefore, we select the corresponding state by applying X-gates on all three qubits. The first rule (same Ticket) does not apply because we don't see a controlled-X-gate with the target qubit rules_0. But we do see a controlled -X-gate with the target qubit rules_1. It indicates this passenger shares the same last name. The five connected dots across all qubits represent the controlled-Z-gate. Subsequently, we undo the

controlled-X-gate and also undo the selection of the state (single X-gates) before we continue with the next passenger.

The circuit shows the last passenger of the oracle contains controlled-X-gates on both `rules`-qubits. This is the state `100` that represents the fifth passenger in our `group`:

```
4 100 Carter, Mrs. Ernest Courtenay (Lilian Hughes)
```

The third part of our algorithm is the amplifier. Again, we create a convenience function to create it for us.

Listing 13.13: The amplifier-function

```python
def amplifier(passenger, q_p, draw=False):
  # Create a sub-circuit
  a_qc = QuantumCircuit(q_p)

  a_qc.h(q_p)
  a_qc.x(q_p)
  a_qc.append(ZGate().control(QUBITS-1), q_p)
  a_qc.x(q_p)
  a_qc.h(q_p)

  if draw:
    return a_qc.draw()
  else:
    # We return the oracle as a gate
    Amplifier = a_qc.to_gate()
    Amplifier.name = "amplifier"
    return Amplifier
```

We apply the same structure as in the oracle. The `amplifier` function takes three mandatory parameters, the `passenger`, the `QuantumRegister` representing the passengers, and the optional parameter `draw` (line 1).
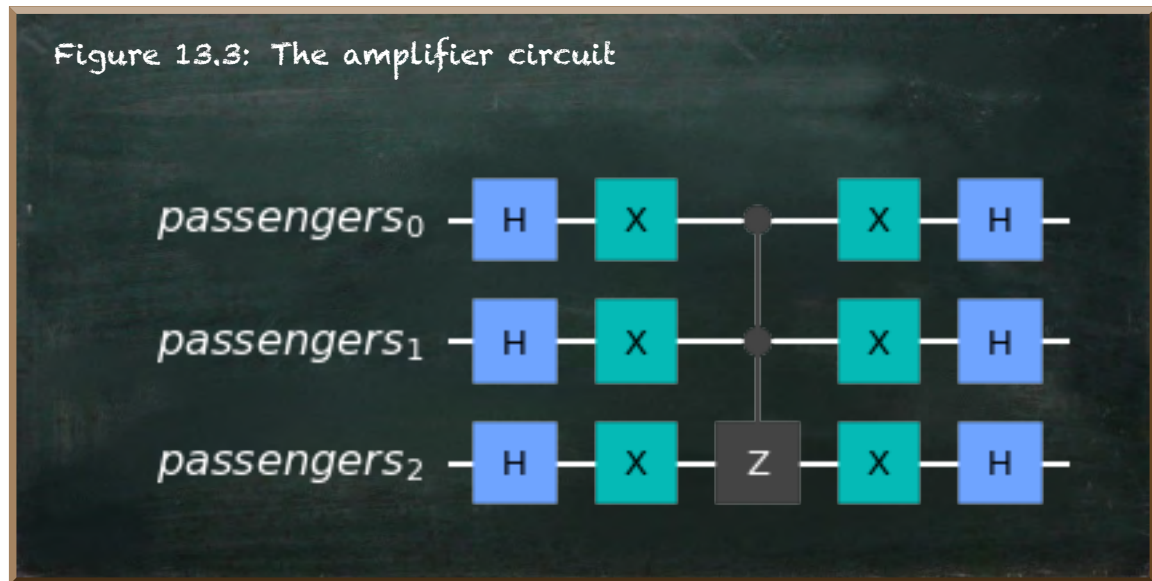
We create another sub-circuit (line 3). The amplifier consists of a multi-controlled-Z-gate encapsulated in X- (lines 6 and 8) and H- gates (lines 5 and 9).

If the `draw` parameter is set to `True`, we draw the circuit (lines 11-12). Otherwise, we turn this sub-circuit into a gate (line 15), specify its name (line 16), and return this gate (line 17).

The following figure depicts the amplifier circuit.

Listing 13.14: Showing the amplifier circuit

```
1 amplifier(passenger, q_pass, draw=True)
```

Figure 13.3: The amplifier circuit

passengers_0 — H — X — ● — X — H —

passengers_1 — H — X — ● — X — H —

passengers_2 — H — X — Z — X — H —

Let's put these pieces together.

Listing 13.15: The search-algorithm

```
 1 qc = QuantumCircuit(q_pass, q_rules)
 2
 3 # put passenger qubits into superposition
 4 qc.h(q_pass)
 5
 6 # Apply the oracle
 7 qc.append(oracle(current_passenger, group, q_pass, q_rules), [*q_pass, *
     q_rules])
 8
 9 # Apply the amplifier
10 qc.append(amplifier(current_passenger, q_pass), q_pass)
11
12 qc.draw()
```

Figure 13.4: The complete circuit
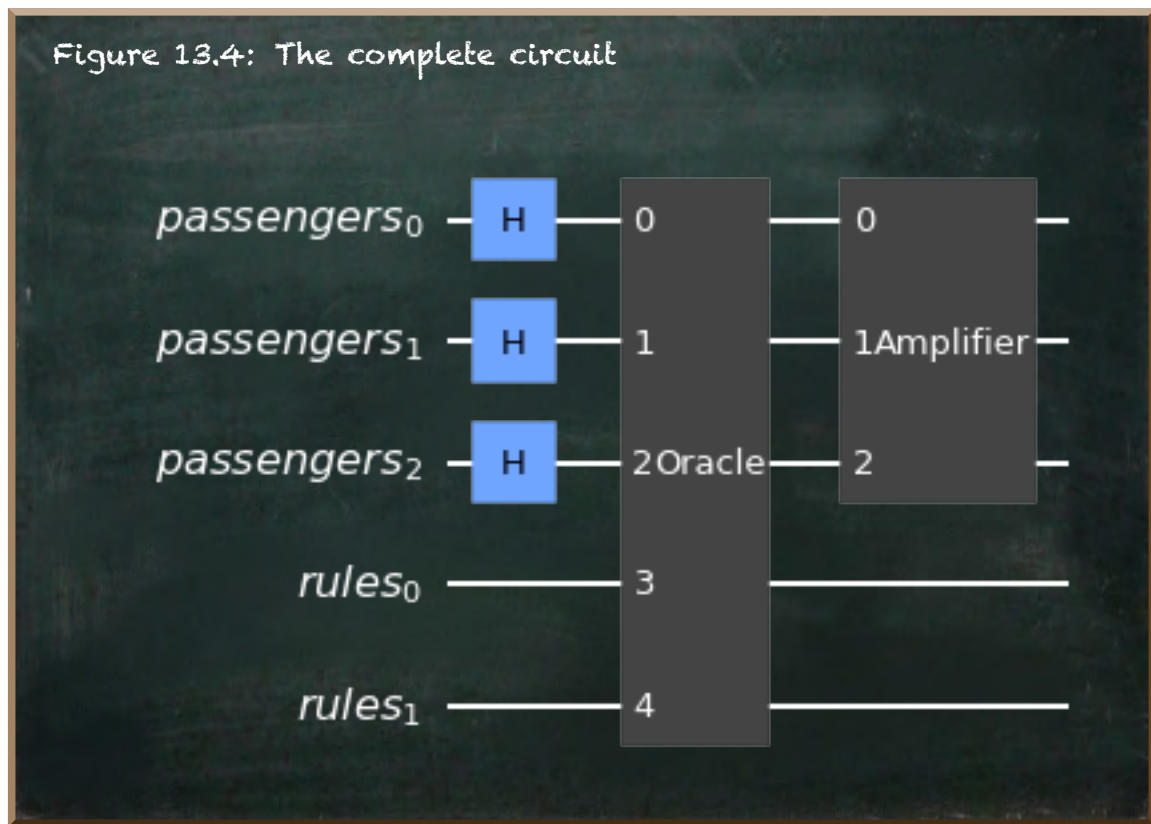
Thanks to the preparation work, our main algorithm is relatively small. We define a `QuantumCircuit` with two `QuantumRegisters` (line 1). We apply Hadamard-gates on all qubits inside the `q_pass`-register (line 4). Then, we apply the oracle (line 7) and the amplifier (line 10) by appending them to our main circuit. The `append` function takes two parameters. The first is the gate that we create in our convenience functions. The second is a list of qubits that we want to apply to this gate to. The oracle uses all qubits. The amplifier only uses the passenger-qubits.

Are you curious? Let's run this circuit.

Listing 13.16: Result of the search algorithm

```
1 results = execute(qc,Aer.get_backend('statevector_simulator')).result()
2 plot_histogram(results.get_counts())
```

Figure 13.5: Result of the search algorithm
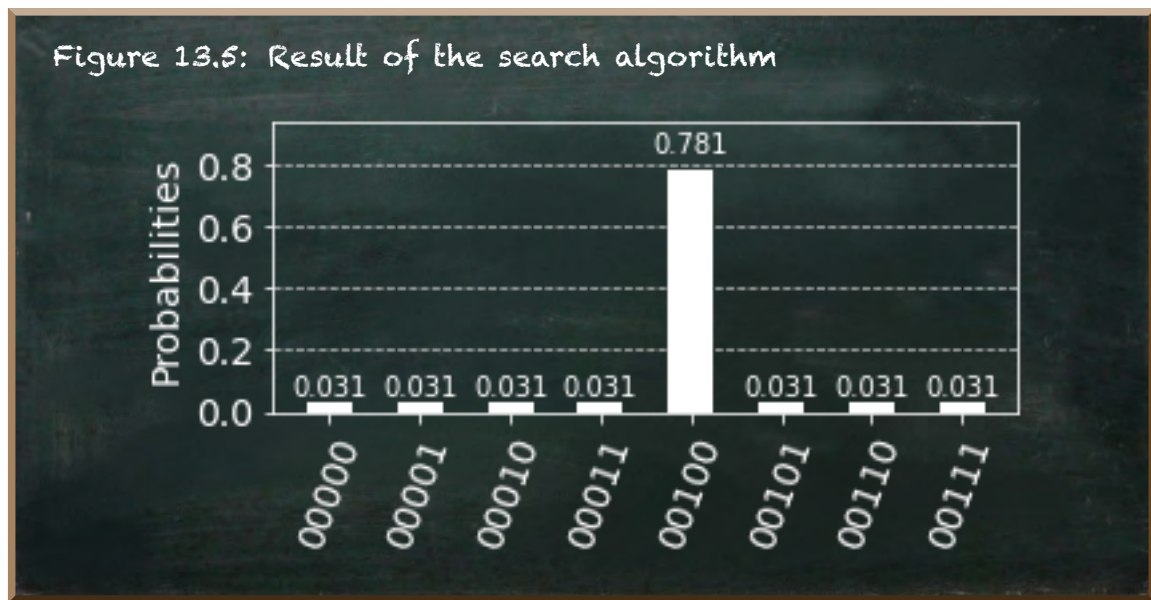
Our algorithm measures state 00100 with the highest probability of 0.781. This is the state representing the searched relative of Mr. Rev. Ernest Courtenay Carter. It is Mrs. Ernest Courtenay (Lilian Hughes) Carter.

Even though the result is quite evident, there's one thing missing. At the end of section 12.1, we mentioned we need to repeat the amplification more than once, depending on the number of states we have.

In our case, we have 2^3 states. Therefore we need to apply the amplifier $\frac{\pi}{4}\sqrt{8}$ times. Let's put this calculation into a separate function.

Listing 13.17: Apply the amplifier multiple times

```
1  def rounds(number_of_rounds, qc, current_passenger, group, q_pass,
       q_rules):
2    print ("{} iterations".format(number_of_rounds))
3    for i in range(0,round(number_of_rounds)):
4      qc.append(oracle(current_passenger, group, q_pass, q_rules), [*q_pass,
         *q_rules])
5      qc.append(amplifier(current_passenger, q_pass), q_pass)
```

Besides the usual parameters, the rounds function takes the number_of_rounds and repeats the oracle followed by the amplifier accordingly. Let's have a look.
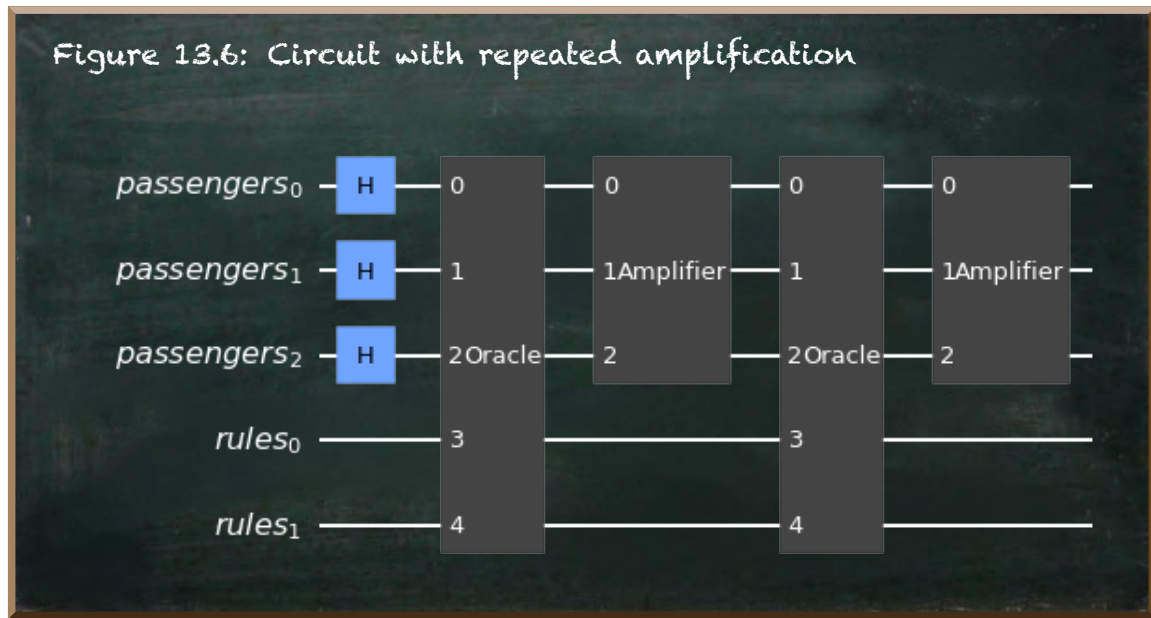
Listing 13.18: Search algorithm with repeated amplification

```
1 from math import pi
2 qc = QuantumCircuit(q_pass, q_rules)
3
4 # put passenger qubits into superposition
5 qc.h(q_pass)
6 rounds(pi*sqrt(2**QUBITS)/4, qc, current_passenger, group, q_pass,
     q_rules)
7 qc.draw()
```

```
 2.221441469079183 iterations
```


Figure 13.6: Circuit with repeated amplification

With eight state or three qubits, we need to iterate through the amplification 2.22 times. We round this to 2. The following figure depicts the results of this circuit.

Listing 13.19: Results of the repeated amplification

```
1 results = execute(qc,Aer.get_backend('statevector_simulator')).result()
2 plot_histogram(results.get_counts())
```

Figure 13.7: Results of the repeated amplification
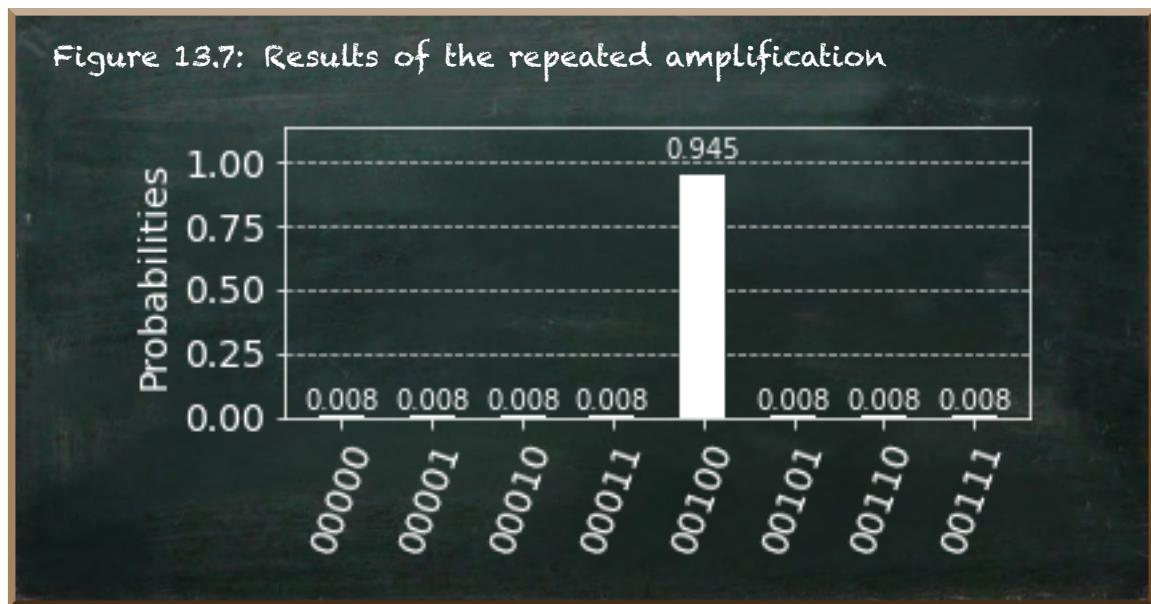
The measures show an improved result. Now, we receive the correct searched relative with a probability of 0.945. Further, we created a pretty flexible algorithm. Let's say we want our circuit to consist of 10 qubits representing the passengers.

> ⚠ This circuit takes quite a while on a classical computer.

Listing 13.20: Running a circuit with 10 passenger-qubits

```
1  QUBITS=10
2
3  q_pass = QuantumRegister(QUBITS, name='passengers')
4  q_rules = QuantumRegister(RULES, name='rules')
5  qc = QuantumCircuit(q_pass, q_rules)
6
7  # put passenger qubits into superposition
8  qc.h(q_pass)
9  rounds(pi*sqrt(2**QUBITS)/4, qc, current_passenger, group, q_pass,
       q_rules)
10 results = execute(qc,Aer.get_backend('statevector_simulator')).result()
11 "Probability of finding '100': {}".format(results.get_counts()['
       000000000100'])
```

In the circuit above, we search the one state out of $2^{10} = 1024$ states. The only thing we need to change is the `QUBITS` value. Then, we renew the `q_pass` -`QuantumRegister` and the `QuantumCircuit`. The output shows we used 25 iterations, and the circuit has a probability of 0.999 to result in the correct state.

When you run this circuit on your local machine, it needs quite some time. Simulating twelve qubits altogether is already a challenge for a classical computer.

13.2   Multiple Results

Passengers may not only have a single sibling, spouse, parent, or child. They can have many. For instance, the Carter family, whose father is Mr. William Ernest Carter, consists of the parental couple and two children.

Before we see what happens, let's even further structure our code. We define the `prepare_group`-function. It takes a `PassengerId` and returns the corresponding passenger and a list of potential relatives.

Listing 13.21: Prepare the search

```
1  def prepare_group(passengerId):
2    current_passenger = train[train["PassengerId"].eq(passengerId)]
3    last_name = current_passenger.Name.to_string(index=False).split(',')[0]
4    train[train["Name"].str.contains(last_name)]
5
6    ticket = current_passenger["Ticket"].to_string(index=False)
7    passengerId = current_passenger["PassengerId"]
8
9    group = train[
10     train["PassengerId"].ne(passengerId) & (
11     train["Name"].str.contains(last_name) |
12     train["Ticket"].eq(ticket)
13   )]
14   return (current_passenger, group)
```

We also define the `find_relatives` function. It takes the passenger and the group and runs the search algorithm.

Listing 13.22: Run the search for a passenger's relatives

```python
1  def find_relatives(current_passenger, group):
2      q_pass = QuantumRegister(QUBITS, name='passengers')
3      q_rules = QuantumRegister(RULES, name='rules')
4      qc = QuantumCircuit(q_pass, q_rules)
5  
6      # put passenger qubits into superposition
7      qc.h(q_pass)
8      rounds(pi*sqrt(2**QUBITS)/4, qc, current_passenger, group, q_pass,
          q_rules)
9      results = execute(qc,Aer.get_backend('statevector_simulator')).result()
10     return plot_histogram(results.get_counts())
```
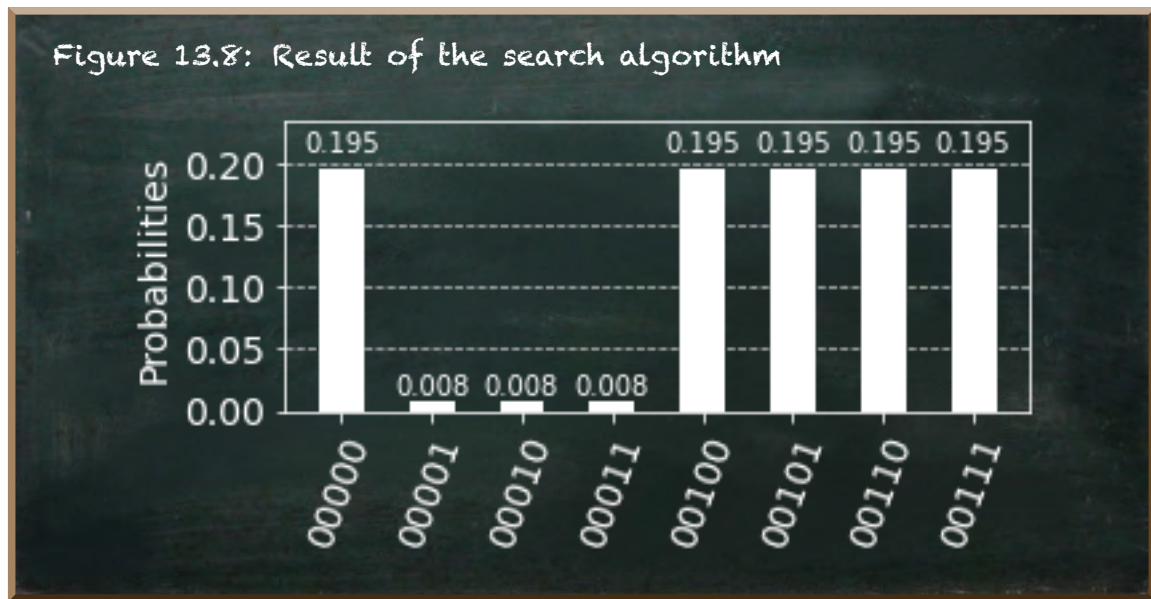
So, let's see what happens if we run our algorithm to search for the relatives of Mr. William Ernest Carter.

Listing 13.23: Search for the relatives of Mr. William Ernest Carter.

```python
1  QUBITS=3
2  (current_passenger, group) = prepare_group(391)
3  find_relatives(current_passenger, group)
```


Figure 13.8: Result of the search algorithm

When we look at the table of potential relatives, we see the algorithm found all but the actual relatives.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
249	250	0	2	Carter, Rev. Ernest Courtenay	male	54.0	1	0	244252	26.0	NaN	S
435	436	1	1	Carter, Miss. Lucile Polk	female	14.0	1	2	113760	120.0	B96 B98	S
763	764	1	1	Carter, Mrs. William Ernest (Lucile Polk)	female	36.0	1	2	113760	120.0	B96 B98	S
802	803	1	1	Carter, Master. William Thornton II	male	11.0	1	2	113760	120.0	B96 B98	S
854	855	0	2	Carter, Mrs. Ernest Courtenay (Lilian Hughes)	female	44.0	1	0	244252	26.0	NaN	S

The problem is the number of iterations. When we have more than one correct relative, calculating the required iterations is a little different. If we search for M results, we need to iterate $\frac{\pi}{4}\sqrt{\frac{N}{M}}$ times.

Listing 13.24: Run the search for mutliple relatives

```python
def find_relatives(current_passenger, group, cnt_searched):
  q_pass = QuantumRegister(QUBITS, name='passengers')
  q_rules = QuantumRegister(RULES, name='rules')
  qc = QuantumCircuit(q_pass, q_rules)

  # put passenger qubits into superposition
  qc.h(q_pass)
  rounds(pi/4*sqrt(2**QUBITS/cnt_searched), qc, current_passenger, group,
     q_pass, q_rules)
  results = execute(qc,Aer.get_backend('statevector_simulator')).result()
  return plot_histogram(results.get_counts())
```

We adjust the calculation of required iterations accordingly (line 8). Fortunately, our dataset indicates the number of relatives we search for. This is the sum of a passenger's SibSp and Parch values.

Listing 13.25: Result of the search with adjusted number of iterations

```python
find_relatives(current_passenger, group, current_passenger["SibSp"]+
    current_passenger["Parch"])
```

Figure 13.9: Result of the search algorithm for mutliple relatives
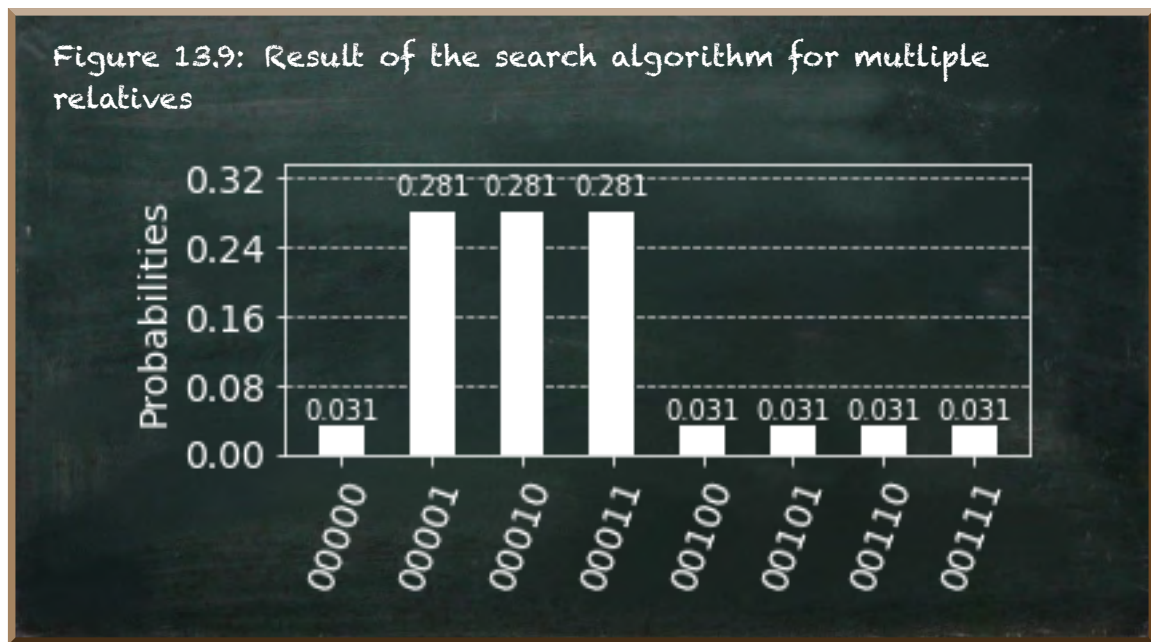
When we run the adjusted circuit, we see a decreased number of iterations. The updated algorithm finds the correct three relatives.

14. Sampling

14.1   Forward Sampling

In the previous chapter 10, we learned how to apply a variational method to learn a hidden variable. Variational methods approximate the distribution of a hidden variable analytically.

Sampling-based methods work differently. Instead of calculating the hidden distribution, they approximate it empirically. The principle is straightforward. These methods repeatedly select an instance and calculate the discrete value of the hidden variable for this instance. Let's say we have a coin, and we want to determine the probability of getting heads. We do not analyze the texture and structure of the coin. We toss it over and over again and count the times it lands heads up. Having tossed it for a hundred times, we got quite a good approximation of the probability of getting heads.

Sampling builds upon simulation. We do not necessarily select real instances from our data. But we generate random instances based on our data.

In our QBN, each variable has a probability table (the CPT). We start with the variables that have no parents. Their CPTs denote the marginal (without conditions) probability of the respective values. We use this probability to assign a value to a new instance.

We continue with the child variables. Their probability tables tell us how likely the variable takes on a particular value given the values of its parents. Since our instance already has a value for the parent variable, we can assign a

value for the child variable. A single instance is not representative of our data. It could be a very unlikely case that we would only observe one in a million times. Therefore, we repeat creating instances many times. The higher the number of instances, the closer our generated population of instances represents the probabilities given by our probability tables. We get a good estimation of the probability a variable takes on. This method is known as forward sampling.

We already did this when we used our QBN to calculate the marginal probability to survive the Titanic shipwreck.

Listing 14.1: Preparing the trained QBN

```
1  def get_trained_qbn(passengers, params, hist=True):
2
3    prepared_passengers = prepare_data(passengers, params)
4    norm_params = calculate_norm_params(prepared_passengers)
5    surv_params = calculate_surv_params(prepared_passengers)
6
7    def qbn_titanic():
8      def circuit(qc, qr=None, cr=None):
9        apply_ischild_sex(qc)
10        apply_norm(qc, norm_params)
11        apply_class(qc)
12        apply_survival(qc, surv_params)
13
14        qc.measure(QPOS_SURV, cr[0])
15
16      return as_pqc(QUBITS, circuit, hist=hist, cnt_classical=1, measure=
         True, shots=1000)
17    return qbn_titanic()
```

A quantum simulator is the perfect example of a sampling algorithm. It allows us to specify the number of instances to draw (shots). Our QBN specifies the probabilities that govern the creation of the instances.

The get_trained_qbn-function creates a PQC based on the passenger data and the parameters we trained previously. As a result, we get the marginal probability that a passenger survived in accordance with our data.

Listing 14.2: Use the QBN for forward sampling

```
1  get_trained_qbn(train, trained_params)
```

Figure 14.1: Result of forward sampling
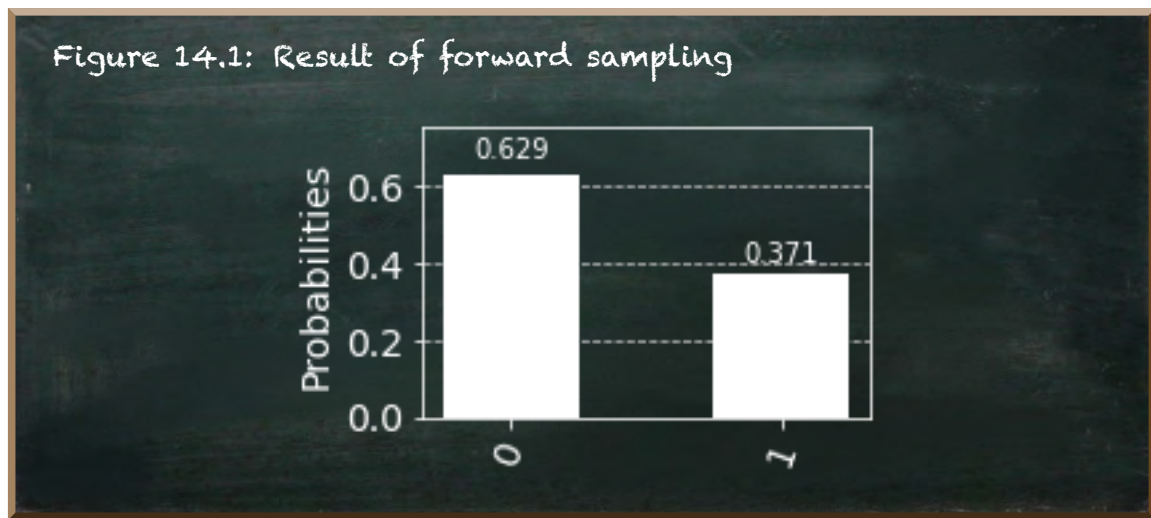
14.2 Bayesian Rejection Sampling

Our QBN is well trained. Yet, when we look back to all the CPT it consists of (see section 10.3), we haven't completed it. We still miss the CPT of being favored by a Norm given the passenger's Sex and Age. And we miss the CPT of Survival given the Norm and the Pclass. So let's catch up on this.

Listing 14.3: Preparing the trained QBN to measure the norm

```
 1  def get_trained_qbn_norm(passengers, params):
 2
 3    prepared_passengers = prepare_data(passengers, params)
 4    norm_params = calculate_norm_params(prepared_passengers)
 5    surv_params = calculate_surv_params(prepared_passengers)
 6
 7    def qbn_titanic():
 8      def circuit(qc, qr=None, cr=None):
 9        apply_ischild_sex(qc)
10        apply_norm(qc, norm_params)
11        apply_class(qc)
12        apply_survival(qc, surv_params)
13
14        qc.measure(QPOS_ISCHILD, cr[0])
15        qc.measure(QPOS_SEX, cr[1])
16        qc.measure(QPOS_NORM, cr[2])
17
18      return as_pqc(QUBITS, circuit, hist=False, cnt_classical=3, measure=
         True, shots=10000)
19    return qbn_titanic()
```

Again, estimating the numbers in the CPT of a Bayesian network builds upon counting how many times that event occurred in our training data.

First, we need to change the variables we measure.

Instead of the qubit at position `QPOS_SURV` that represents whether a passenger survived, we measure the qubits representing whether a passenger was a child (`QPOS_ISCHILD`, line 14), whether she was female (`QPOS_SEX`, line 15), and whether she was favored by a norm (`QPOS_NORM`, line 16).

Since we measure three qubits, we need to specify the corresponding number of classical bits (`cnt_classical=3`). Further, we specify not to draw a histogram (`hist=False`) because we want to get the raw data.

Listing 14.4: Obtain the measurements

```
1  norm_mes = get_trained_qbn_norm(train, trained_params)
2  norm_mes
```

```
{'000': 5392,
 '001': 307,
 '010': 50,
 '011': 17,
 '100': 753,
 '101': 78,
 '110': 3194,
 '111': 209}
```

As a result, we get a Python dictionary whose keys represent the states and whose values denote the number of counts of each state. The state is a binary string read from right to left.

The right-hand bit represents the classical bit at position 0 – whether the passenger was a child (==1) or not (==0). The middle bit represents whether the passenger was female (==1) or male (==0). The left-hand bit shows whether the passenger was favored by a norm (==1) or not (==0).

Listing 14.5: Calculate the conditional probabilities of being favored by a norm

```
1 print("Chance to be favored by norm")
2 print("Female children: {:.2f} ".format(norm_mes['111']/(norm_mes['111']+
      norm_mes['011'])))
3 print("Female adults: {:.2f} ".format(norm_mes['110']/(norm_mes['110']+
      norm_mes['010'])))
4 print("Male children: {:.2f} ".format(norm_mes['101']/(norm_mes['101']+
      norm_mes['001'])))
5 print("Male adults: {:.2f} ".format(norm_mes['100']/(norm_mes['100']+
      norm_mes['000'])))
```

```
Chance to be favored by norm
Female children: 0.92
Female adults: 0.99
Male children: 0.21
Male adults: 0.13
```

This time, we aim to calculate conditional probabilities. We aim to calculate the chance to be favored by a norm given the passenger's gender and whether she was a child. The samples we create may belong to one out of four groups (female children, male children, female adults, and male adults). Accordingly, we calculate the conditional probabilities of being favored by a Norm by counting the number of instances where the passenger of the specific group was favored divided by the size of the whole group.

For instance, the probability of a female child being favored is given by the count of favored female children (norm_mes['111']) divided by the sum of all female children (favored norm_mes['111'] or unfavored norm_mes['011']).

We refer to this kind of sampling as rejection sampling. Let's say we only wanted to calculate the probability of a female child being favored. Then, we would only keep the instances that contain the corresponding evidence and reject (throw away) all other instances.

Since we want to calculate the probabilities of all the groups, we keep the instances, but we regard each group separately.

Next, we calculate the conditional probability of survival.

Listing 14.6: Preparing the trained QBN to measure survival

```
1  def get_trained_qbn_surv(passengers, params):
2
3    prepared_passengers = prepare_data(passengers, params)
4    norm_params = calculate_norm_params(prepared_passengers)
5    surv_params = calculate_surv_params(prepared_passengers)
6
7    def qbn_titanic():
8      def circuit(qc, qr=None, cr=None):
9        apply_ischild_sex(qc)
10       apply_norm(qc, norm_params)
11       apply_class(qc)
12       apply_survival(qc, surv_params)
13
14       qc.measure(QPOS_NORM, cr[0])
15       qc.measure(QPOS_FIRST, cr[1])
16       qc.measure(QPOS_SECOND, cr[2])
17       qc.measure(QPOS_THIRD, cr[3])
18       qc.measure(QPOS_SURV, cr[4])
19
20      return as_pqc(QUBITS, circuit, hist=False, cnt_classical=5, measure=
     True, shots=10000)
21    return qbn_titanic()
```

In the quantum circuit, we measure the corresponding qubits (lines 14-18). The rest remains unchanged.

Listing 14.7: Obtain the measurements

```
1  surv_mes = get_trained_qbn_surv(train, trained_params)
2  surv_mes
```

```
{'10010': 307,
 '10011': 1000,
 '10100': 93,
 '10101': 863,
 '11000': 225,
 '11001': 1285,
 '00010': 1043,
 '00011': 67,
 '00100': 1074,
 '00101': 91,
 '01000': 2836,
 '01001': 1116}
```

As a result, we get a list of states and their counts. The meaning of each bit changed according to the changed measurements. The left qubit denotes `survival`, the second bit whether the passenger had a third-class ticket, the third bit whether the passenger had a second-class ticket, the fourth bit whether the passenger had a first-class ticket, and the fifth bit whether the passenger was favored by a `Norm`.

Again, we calculate the probabilities by dividing the number of survivors of a group by the size of the group.

Listing 14.8: Calculate the conditional probabilities of survival

```
 1  print("Chance to survive")
 2  print("Favored 1st: {:.2f} ".format(
 3    surv_mes['10011']/(surv_mes['10011']+surv_mes['00011'])))
 4  print("Favored 2nd: {:.2f} ".format(
 5    surv_mes['10101']/(surv_mes['10101']+surv_mes['00101'])))
 6  print("Favored 3rd: {:.2f} ".format(
 7    surv_mes['11001']/(surv_mes['11001']+surv_mes['01001'])))
 8  print("Unfavored 1st: {:.2f} ".format(
 9    surv_mes['10010']/(surv_mes['10010']+surv_mes['00010'])))
10  print("Unfavored 2nd: {:.2f} ".format(
11    surv_mes['10100']/(surv_mes['10100']+surv_mes['00100'])))
12  print("Unfavored 3rd: {:.2f} ".format(
13    surv_mes['11000']/(surv_mes['11000']+surv_mes['01000'])))
```

```
Chance to survive
Favored 1st: 0.94
Favored 2nd: 0.90
Favored 3rd: 0.54
Unfavored 1st: 0.23
Unfavored 2nd: 0.08
Unfavored 3rd: 0.07
```

Let's put these numbers into the CPTs of our QBN.

Figure 14.2: The completed CPT
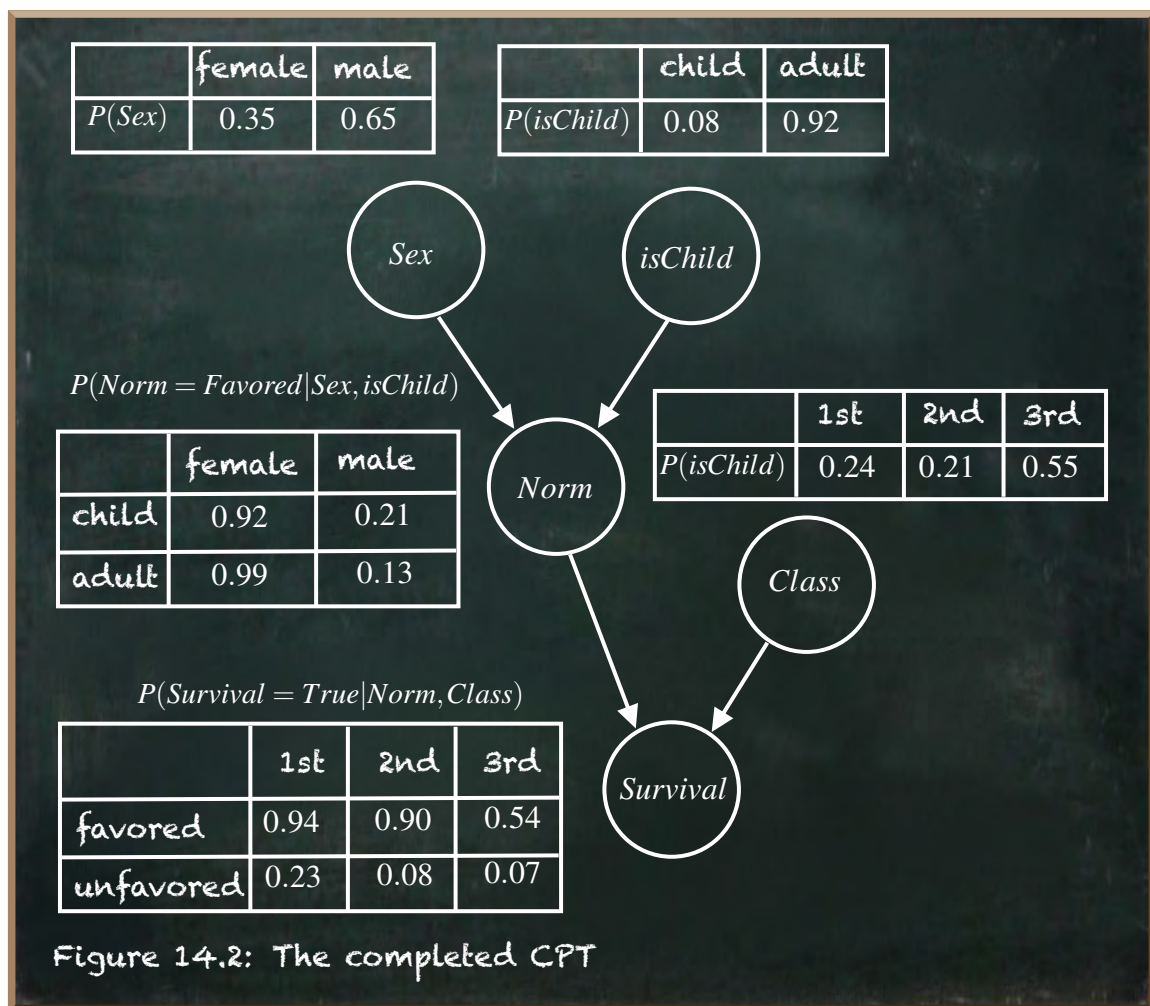
14.3 Quantum Rejection Sampling

Rejection sampling is a fancy name for a straightforward method. We create samples and pick the ones that contain the evidence we're interested in.

The problem with this type of sampling is that we generate many samples we don't use. Let's have a critical look at the following case of calculating the probability of female children being favored by a norm.

Listing 14.9: Evaluate the effectiveness.

```
1  norm_mes = get_trained_qbn_norm(train, trained_params)
2  print("{:.2f} of female children were favored by a norm ".format(
3    norm_mes['111']/(norm_mes['111']+norm_mes['011'])))
4  print("Considered {} samples ".format(
5    norm_mes['111']+norm_mes['011']))
6  print ("Acceptance of {:.1f}% of the samples".format(
7    (norm_mes['111']+norm_mes['011'])/100))
```

```
0.94 of female children were favored by a norm
Considered 210 samples
Acceptance of 2.1% of the samples
```

We repeated our quantum circuit $10,000$ times, but only around 200 instances were female children. Of course, this is not bad per se. It reflects our data that contains only very few female children.

It results in only 2% of the samples being useful. This is quite a waste of resources and can become a problem if the network becomes complex.

Wouldn't it be great if we had a way of increasing the probability of obtaining the proper evidence?

In fact, there is! In a QBN, a quantum state represents a possible configuration of passenger characteristics. In the previous chapter 12 we got to know amplitude amplification – a method that allows us to increase the probability of measuring a certain state.

Let's start with the oracle. It is quite simple. A simple *CCZ*-gate flips the amplitude of the state where the relevant qubits are in state $|1\rangle$ (line 5). We only want to select female children. But these directly affect whether the passenger was favored by a norm. In our circuit, we entangle these qubits. Therefore, include the qubit at the position QPOS_NORM in the controlled *Z*-gate. You can add further qubits (such as the qubit at position QPOS_SURV) if you like. But this is not required.

Further, if we wanted to select male children, female adults, or male adults,

we would need to encapsulate the controlled Z-gate into the respective *NOT*-gates on the qubit we wish to obtain a different value.

Listing 14.10: The oracle-function

```python
1  def oracle(qr, draw=False):
2    # Create a sub−circuit
3    o_qc = QuantumCircuit(qr)
4
5    o_qc.append(ZGate().control(2), [qr[QPOS_ISCHILD], qr[QPOS_SEX], qr[
       QPOS_NORM]])
6
7    if draw:
8      return o_qc.draw()
9    else:
10     # We return the oracle as a gate
11     Oracle = o_qc.to_gate()
12     Oracle.name = "oracle"
13     return Oracle
```

The `oracle` function creates and returns a custom gate named "oracle" (lines 11-13). The following image depicts our oracle.

Figure 14.3: The oracle for female children
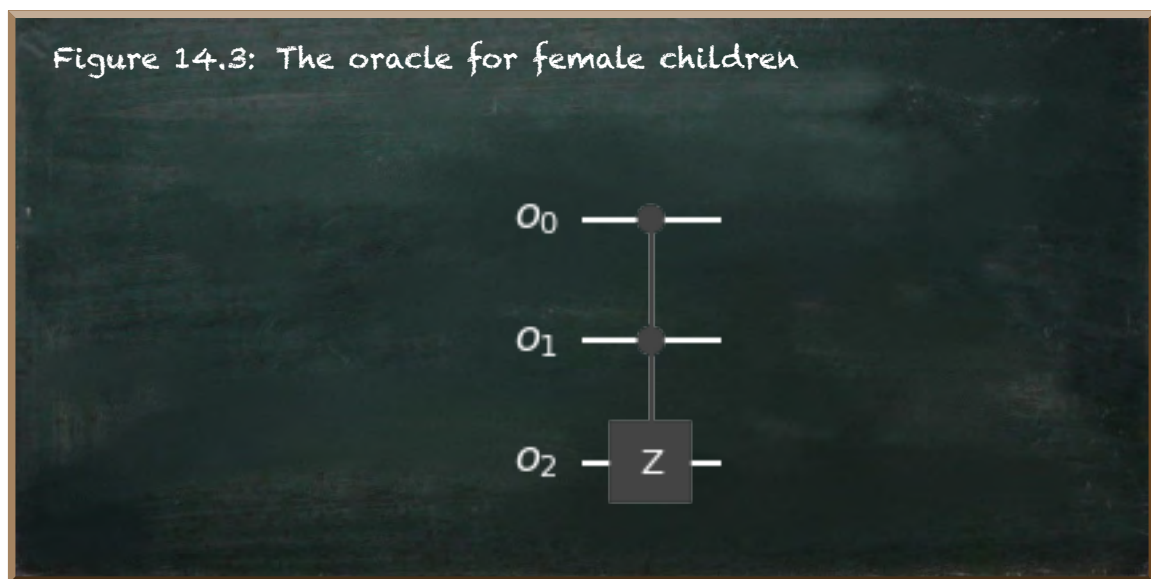
The next gate we create is the amplifier. Thus far, we learned the amplifier to consist of a controlled Z-gate encapsulated into Hadamard- and *NOT*-gates. Our amplifier is almost alike. However, we do not use Hadamard-gates. But

we use a part of the actual implementation of our quantum Bayesian network instead. We use the function `apply_ischild_sex` we created in section 10.3. Basically, this function applies R_Y-gates on the qubits at the positions `QPOS_SEX` and `QPOS_ISCHILD` by the angle representing the corresponding marginal probabilities.

The rest of the amplifier remains the standard amplifier we already know. Again, we apply a controlled Z-gate inside *NOT*-gates. However, this time, we must apply the controlled Z-gate only on the two qubits whose state we want to amplify. These are the qubits at the positions `QPOS_SEX` and `QPOS_ISCHILD` (line 8).

Listing 14.11: The amplifier-function

```python
def amplifier(qr, draw=False):
  # Create a sub-circuit
  a_qc = QuantumCircuit(qr)

  apply_ischild_sex(a_qc) # H
  a_qc.x(qr[QPOS_ISCHILD])
  a_qc.x(qr[QPOS_SEX])
  a_qc.cz(qr[QPOS_ISCHILD], qr[QPOS_SEX])
  a_qc.x(qr[QPOS_ISCHILD])
  a_qc.x(qr[QPOS_SEX])
  apply_ischild_sex(a_qc) # H

  if draw:
    return a_qc.draw()
  else:
    # We return the oracle as a gate
    Amplifier = a_qc.to_gate()
    Amplifier.name = "amplifier"
    return Amplifier
```

The following image depicts the amplifier.

Figure 14.4: The amplifier

$amplifier_0$ — R_Y 0.497 — X — ● — X — R_Y 0.497

$amplifier_1$ — R_Y 1.27 — X — ● — X — R_Y 1.27
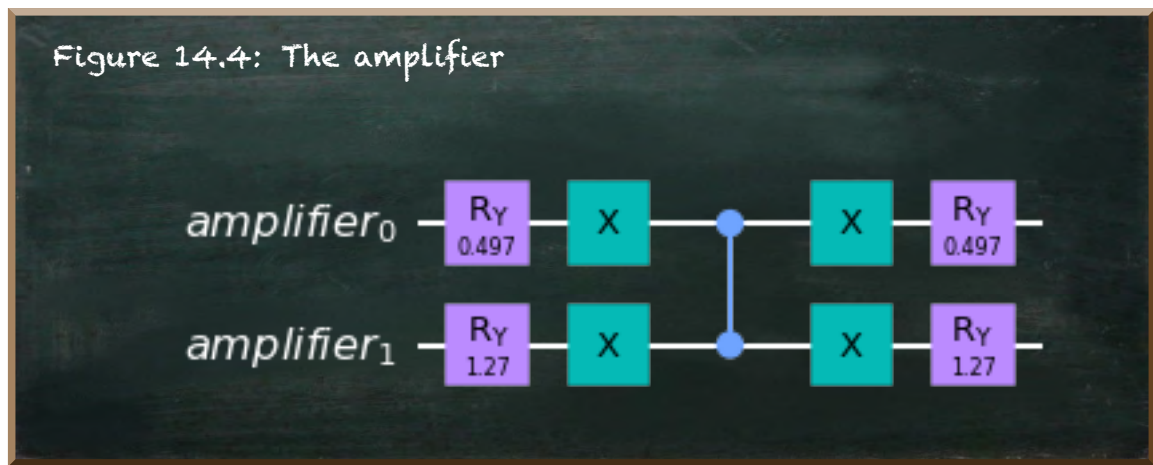
We are ready to put the circuit together. We start with preparing the passengers (line 3) and calculating the parameters (lines 4-5). Usually, the main circuit starts with a series of Hadamard gates. But we don't want all qubits to start in a balanced state of superposition. Instead, we apply our QBN inside the amplifier (line 11)

Then, the amplification consists of two iterations of the oracle and the amplifier (lines 13-17).

Before we apply the rest of our QBN (applying the norm and the survival, lines 21-22), we need to apply *NOT*-gates on the amplified qubits (lines 19-20)

Finally, we measure the qubits we're interested in (lines 24-26)

Listing 14.12: Prepare the amplified QBN

```
1  def get_trained_qbn_amp(passengers, params):
2
3    prepared_passengers = prepare_data(passengers, params)
4    norm_params = calculate_norm_params(prepared_passengers)
5    surv_params = calculate_surv_params(prepared_passengers)
6
7    def qbn_titanic():
8      def circuit(qc, qr=None, cr=None):
9
10       # amplifier replacing the H-gates
11       qc.append(amplifier(qr), qr)
12
13       # Amplification
14       qc.append(oracle(qr), qr)
15       qc.append(amplifier(qr), qr)
16       qc.append(oracle(qr), qr)
17       qc.append(amplifier(qr), qr)
18
19       qc.x(QPOS_ISCHILD)
20       qc.x(QPOS_SEX)
21       apply_norm(qc, norm_params)
22       apply_survival(qc, surv_params)
23
24       qc.measure(QPOS_ISCHILD, cr[0])
25       qc.measure(QPOS_SEX, cr[1])
26       qc.measure(QPOS_NORM, cr[2])
27
28      return as_pqc(QUBITS, circuit, hist=False, cnt_classical=3, measure=
     True, shots=10000)
29    return qbn_titanic()
```

We're ready to run our amplified QBN.

Listing 14.13: Evaluate the effectiveness of the amplified QBN

```
1  amp_mes = get_trained_qbn_amp(train, trained_params)
2  print("{:.2f} of female children were favored by a norm ".format(
3    amp_mes['111']/(amp_mes['111']+amp_mes['011'])))
4  print("Considered {} samples ".format(
5    amp_mes['111']+amp_mes['011']))
6  print ("Acceptance of {:.1f}% of the samples".format(
7    (amp_mes['111']+amp_mes['011'])/100))
```

```
0.94 of female children were favored by a norm
Considered 9809 samples
Acceptance of 98.1% of the samples
```

The result is astonishing. First and most importantly, we obtain the same result of whether female children were favored by a norm. But we increased the probability of female children in the sample to around 98%. Therefore, we could reduce the overall number of generated samples and have enough evidence to conclude from the sample.

Grover's algorithm has become famous for being able to search an unordered list of items in fewer steps than a classical algorithm can. But amplitude amplification is a general technique. It lets us manipulate a quantum system to obtain a state we want it to have. This is something useful beyond just searching for a state.

15. What's Next?

You've reached the end of **Hands-On Quantum Machine Learning with Python–Volume 1**. I hope you enjoyed reading it. And I hope you stay with me in the journey of learning about quantum machine learning.

We learned how to create, train, and use a probabilistic quantum system. We explored how to develop sophisticated quantum circuits. And we learned how to use entanglement and interference to reduce the complexity of a problem at hand. Yet, we have just scratched the very surface of quantum machine learning. There is a lot to discover in quantum machine learning. And you're well-equipped for this journey.

My work on **Quantum Machine Learning** is not finished. You may have noticed the declaration "Volume 1". I already plan the second volume. While I have not settled for the final content, my goal is to concentrate on quantum machine learning algorithms, such as the variational quantum eigensolver (VQE), quantum approximate optimization algorithm (QAOA), quantum Boltzmann machines, and quantum neural networks (QNN).

There's so much to learn. Don't miss the regular updates on Substack, Medium, and www.pyqml.com. If you like, please provide me with feedback at mail@pyqml.com.

You can also mail me if you have any other questions regarding quantum machine learning in general. I'll strive to reply.

Thank you for reading.

Hands-On Quantum Machine Learning With Python
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, Hands-On Quantum Machine Learning With Python is your comprehensive guide to get started with Quantum Machine Learning – the use of quantum computing for the computation of machine learning algorithms.

This book offers a practical, hands-on exploration of quantum machine learning.

PyQML