Picturing Quantum Software

An Introduction to the ZX-Calculus and Quantum Compilation

[Preprint Version 1.0.0 September 2024]

Aleks Kissinger and John van de Wetering

Contents

0.1 About This Version

This is a preprint of:

Picturing Quantum Software:
An Introduction to the ZX-Calculus and Quantum Compilation

to be released by Cambridge University Press. It is released under Creative Commons Attribution-NonCommercial (CC-BY-NC) Licence v4.0, as per the terms of CUP's Green Open Access Policy. It is copyright (C) Aleks Kissinger and John van de Wetering. All rights reserved.

This is the colour version of the book. Colour and black-and white versions are available from the website above.

1

Introduction

"Maybe in order to understand Mankind, we should look at the word itself. Basically, it's made up of two separate words 'mank' and 'ind'. What do these words mean? It's a mystery. And that's why so is mankind."

- Deep Thoughts by Jack Handy. Saturday Night Live 1993

Maybe in order to understand what this book is about, we should break it up into its parts: "Picturing" and "Quantum Software". Let's talk about the second part first. Broadly, quantum software refers to the "code" that runs on a quantum computer. This can mean many different things and many different levels: ranging from *quantum algorithms*, i.e. high-level descriptions of how to solve problems with a quantum computer, all the way down to actual low-level code used to program specialised hardware like microwave emitters that are actually responsible for making things happen to quantum systems and reading out the results. This book is largely focused on what lies between those two levels and also how we can move from higher-level descriptions of a computation to lower-level ones. In classical computing, passing from a high-level programming language to low-level machine code is called *compilation*. Compilers are extremely important to making classical computers work, both because high-level programming is crucial to representing something as complex as an operating system or a web application and because a big reason why modern computers are so fast is because advanced compilers do huge amounts of optimisation to programs in order to squeeze as much performance as possible out of your computer.

As quantum computers are starting to take shape, so is a new field of **quantum compilation**. Instead of just directly focusing on the code that runs on quantum computers (e.g. quantum circuits, which we'll introduce in the next chapter), it is becoming increasingly interesting to focus on the "code that makes that code" or even the "code that makes that code better".

There are a lot of interesting problems in this area, many of which we'll touch on in this book. The problem most analogous to classical compilation is the following: given a high-level description of a quantum computation, can we build a **quantum circuit** to perform that computation?

Quantum circuits are a *de facto* assembly language for quantum computation, and form the most important part in how a quantum computation is described using the *quantum circuit model*. The latter is a straightforward way to describe quantum computations, which is *computationally universal* (in the sense we describe in Section 2.3.5). Quantum circuits give a simple description of a quantum process by means of a sequence of primitive operations, called **gates**, which are applied on a register of quantum memory, e.g.

```
INIT 5
CNOT 1 0
H 2
Z 3
H 0
H 1
CNOT 4 2
...
```
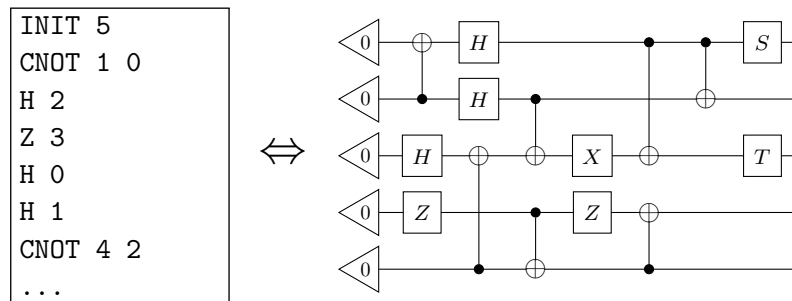
Here, we have deliberately been a bit ambiguous about what a "high-level description" means. This could be a program in a high level (quantum) programming language. Such languages do exist (as we'll survey briefly in the references of Chapter 2), but are still in their infancy. At the time of this writing, it is still not clear which "genuinely quantum" features are useful to have in a quantum programming language, and the majority of what gets called "quantum programming" these days amounts to using some specialised libraries in a familiar classical programming language like Python to build (and sometimes run) quantum circuits. In fact, you'll get to experiment with many of the techniques we discuss in this book using our Python library **PyZX**, but more on that later.

In addition to a program written in a high-level language, a high-level description of a quantum computation could mean various other things, depending on the application. For example, it could simply be a big unitary matrix, which as we will review in Chapter 2, are the way quantum processes are modelled mathematically. However, this method of representing a computation grows exponentially in size with the number of quantum bits, or **qubits**, involved in a computation, so it is not feasible for concretely representing computations on more than a dozen qubits or so. Also, we don't

know about you, but we don't personally find staring at a matrix full of numbers to be the most enlightening way to understand what is actually going on.

This brings us to the other part of the title: "Picturing". This book focuses heavily on representing quantum computations using pictures. Unlike the kinds of pictures you might know, e.g. from a physics textbook, where simple diagrams are used to aid our intuitions, but all the "real math" happens in good old fashioned formulas, in this book, the diagrams *are* the real math. We will begin by introducing **string diagram** notation in the next chapter. This notation, usually attributed to Roger Penrose, gives a rigorous way to describe complex linear operators using diagrams consisting of boxes and wires like this one:
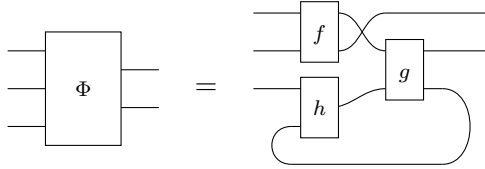
Perhaps the most interesting thing about such diagrams is that we can use them to directly calculate stuff by transforming one diagram into another. In particular, suppose we knew that two different diagrams actually describe the same linear map. Then we might write something like this:

$$(1.1)$$

We'll see how such diagrams actually correspond to linear maps in Section 2.1.6, but even before we know that, it is possible to explain how we could *use* a diagrammatic equation like (1.1): if we see one side of this equation in a bigger diagram, we can "chop it out" and replace it with the other side to get a new diagram:

Thus, from a small equation, we have derived a bigger one. This way of doing calculations directly on diagrams is called **diagrammatic reasoning**.

The "spiritual predecessor" to this book, *Picturing Quantum Processes*, described the basic principles of quantum and classical processes and how they interact using diagrammatic reasoning. Along the way, it introduced a particular set of string-diagram equations called the *ZX-calculus*, which turns out to be extremely useful for doing just about anything you'd want

to do with quantum circuits and related structures. However, that book just gave the slightest inkling of how one might apply ZX to solve concrete problems in quantum software, and didn't say much about how the ZX-calculus picture relates to the whole myriad of techniques developed over the past few decades for designing and reasoning about quantum software.

In this book, we will dive straight into using the language of ZX to work concretely with quantum circuits. This turns out to be a very good language to explain many concepts in quantum software that were not originally introduced using diagrammatic methods, such as stabiliser theory, phase polynomials, measurement-based quantum computation, and quantum error correcting codes. Whether readers are familiar with these concepts or meeting them for the first time, the ZX-calculus offers an intuitive look at the few (relatively simple) structures at play, and how they come together to make quantum computations behave the way they do.

A simple, "Hello World" style example is quantum teleportation, which we'll go through in some detail in Section 3.3.2. This is a simple quantum protocol where Alice and Bob can use a shared entangled state and some classical communication to allow Alice to send an arbitrary quantum state to Bob. In a typical quantum computing textbook, you would probably see teleportation pictured like this:
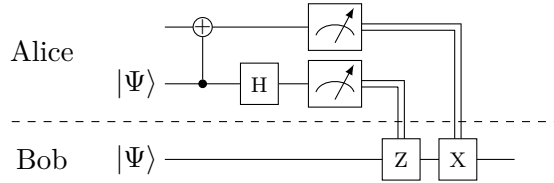
followed by a fairly elaborate linear algebraic calculation to show that it actually works. By the time we get to Chapter 3, we'll see that the ZX-calculus makes these kinds of calculations super easy. After translating the quantum circuit diagram above into ZX language, we can calculate what it does using just a few graphical rules:
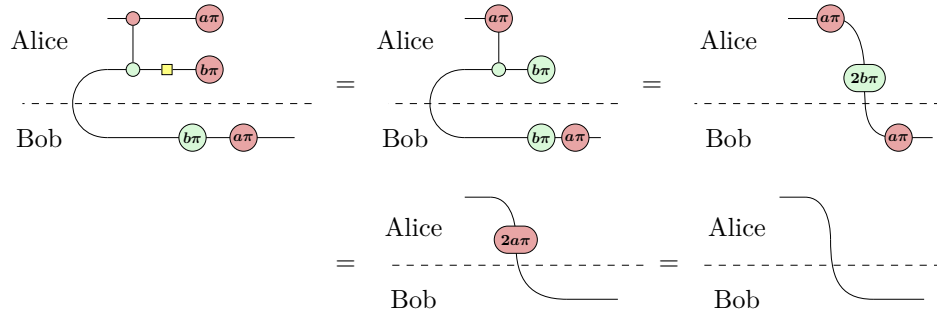
In the end, we get just a wire passing from Alice to Bob, capturing that fact that any quantum state Alice starts with will pop out on Bob's side.

We'll see over the following nine chapters that these techniques can be pushed a lot further than toy examples to explain some of the most important structures, concepts, and algorithms in the design and analysis of quantum software.

1.1 How to read this book

We're glad you made it here! Or, for those coming from *Picturing Quantum Processes*, we're glad you made it back! This book caters to a variety of people with a variety of different backgrounds. As such, you may want to read this book differently depending on what you know already, and what you hope to get out of it. This section is intended to give a roadmap for a few different kinds of readers.

1.1.1 From scratch

This book is designed to function well as a *second* course in quantum computing. However, this book is also designed to be pretty self-contained, which means there is nothing to stop you from diving in with nothing but a bit of linear algebra under your belt. Some basic topics are not covered in as much depth as you'll find in a first text, and also some intuitions may be taken for granted. To adopt this strategy, we suggest you start with the next chapter, then if you get confused, or want to know something more in depth, pick up one of the following books to help you out:

- *Picturing Quantum Processes*. Coecke and Kissinger.
- *Quantum Computation and Quantum Information*. Nielsen and Chuang.
- *An Introduction to Quantum Computing*. Kaye, Lamflamme, and Mosca.
- *Quantum Computer Science*. Mermin.

1.1.2 Coming from *Picturing Quantum Processes*

You will notice a few things if you've landed here straight from the 'spiritual' prequel to this book *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning* (which we'll call PQP in this section).

The first is we don't mention *process theories* at all in this book, even though they were a big deal in PQP. Process theories give us a general

way to talk about processes and computation that includes quantum theory but also many other theories, such as classical probability theory and even "super-quantum" theories which allow processes which (we think) are not physically possible.

While it is fascinating, and foundationally important, to understand quantum theory as just one example of many possible ways the world could have been, we will forgo discussion of generic process theories in this book and assume everything is quantum, right from the beginning! Well, that's actually not quite true. PQP introduced a process theory of **linear maps**, where wires represent Hilbert spaces and boxes represent linear maps, then obtained the theory of **quantum maps** by some diagrammatic trickery. For simplicity, we will pretty much exclusively work in the theory of **linear maps** in this book. As you might recall from PQP, linear maps are pretty much the same thing as **pure quantum maps**, as long as we (1) compute probabilities by taking the absolute value squared of complex numbers, and (2) throw away global phases. It turns out, for all of the concepts in this book, that's good enough for us!

This extra simplicity comes at a price, in that we don't give ourselves access to the elegant graphical techniques for representing mixed quantum states and processes as well as classical-quantum interaction. However, readers with a good handle on these techniques from PQP will probably notice many places they can be applied throughout this book, and we invite them to explore and be creative when filling in these gaps.

One final difference to note, is that in PQP time in diagrams flowed from the bottom to the top, while in this book every diagram should be read from left to right.

1.1.3   Coming from another quantum computing background

If you are already used to looking at $|\text{this}\rangle \, |\text{kind}\rangle \, |\text{of}\rangle \, |\text{stuff}\rangle$, but not necessarily *pictures* of quantum processes, welcome! We have tried to write this book such that readers with little prior experience with the graphical notation for quantum maps will start to feel comfortable with it in no time, and start using diagrams as a natural extension of the quantum circuit notation you undoubtedly already know and love.

You may want to briefly look through Section 2.1, and in particular 2.1.6, as we introduce the graphical notation for linear maps along side the usual bra-ket notation. In those sections, we explain the 3-fold correspondence between bra-ket language, tensor networks, and string diagrams. We use this

correspondence throughout the book, in order to freely switch to whichever tool is best for the job at hand (spoiler alert: it is usually string diagrams!).

After going through Section 2.1, you'll probably not have much problem skipping over the rest of Chapter 2 and diving straight into the ZX-calculus in Chapter 3.

Note that even if you are already familiar with some of the concepts we talk about in this book, you might find that we approach it with quite a different perspective. So even for the quantum computing expert there should be enough to discover in this book!

1.2 Organisation

This book can be split roughly into three parts:
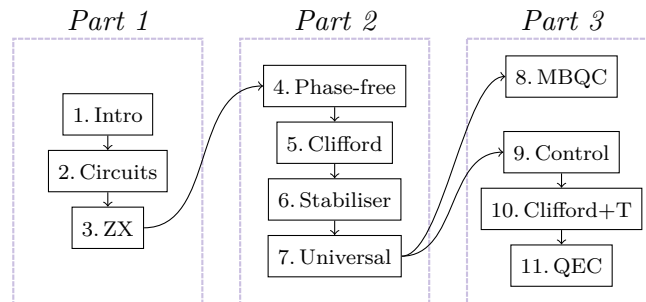
In the first part of the book, Chapters 1–3, we introduce the basics of quantum computation and the main tool we'll be using throughout the book: the ZX-calculus. After this introductory chapter, we move to **Chapter 2: The Quantum Circuit Model**, which briefly reviews the mathematical preliminaries you will need (namely linear algebra over the complex numbers) and gives a short, pragmatic introduction to quantum theory by explaining what we call the **SCUM postulates**.

- **S. States** of a quantum system correspond to vectors in a Hilbert space.
- **C. Compound systems** are represented by the tensor product.
- **U. Unitary maps** evolve states in time.
- **M. Measurements** are computed according to the Born rule.

We then introduce some basic building blocks of quantum circuits, and explain some of the most important facts about circuits, such as universality of gate sets, which we will use throughout the book.

What you will NOT find in this chapter (as opposed to virtually any other introduction to quantum computing) is much discussion about quantum algorithms. These of course are the reason one wants to study quantum

computation in the first place. However, since we will be focusing on quantum compiling in this book, we will mostly abstract over specific quantum algorithms and focus mainly on the sorts of circuit structures that arise from "typical" algorithms and how to work with them effectively. To get a deeper appreciation for how specific algorithms yield the kinds of circuits we will study, we encourage readers to have a look at some of the resources mentioned in Section 1.1.1 and also in the References and Further Reading sections at the end of each chapter.

Chapter 3: The ZX-Calculus introduces ZX-diagrams, the main graphical representation we'll use for all the quantum computations we study in this book, as well as the ZX-calculus, a simple set of rules for transforming and simplifying ZX-diagrams. We also give some examples of "ZX in action", where we show how to perform simple computations using ZX. We focus mainly here on manipulating ZX-diagrams by hand, leaving the more high-powered theorems and automated techniques involving the ZX-calculus for later.

The next part of the book, Chapters 4–7, build up slowly from very restricted families of circuits and ZX-diagrams to full-powered, *universal* quantum computation. In Chapters 4 and 5, we introduce particular, non-universal gate sets, and corresponding **fragments** of the ZX-calculus, whose unitary ZX-diagrams are precisely the circuits constructible in those gate sets.

A natural question is then: Why consider more limited kinds of quantum computations and not just jump straight to maximum power? This is because, like in many areas of science, there is a delicate balancing act between how powerful a system is and how much we can say about it.

The paradigmatic example in classical computation is Turing machines. These are super powerful, so powerful that if a computation is even *possible* to do on a computer (classical or quantum!), then it is possible with a Turing a machine. However, if we pluck a Turing machine out of the air, we can say almost nothing about it. We don't even know if it will halt with an answer or just run forever. At the other extreme are finite state machines. These are much, much weaker than Turing machines, but we can compute (often efficiently) pretty much anything we want to know about them. Does this machine ever answer Yes for some input? How about infinitely many inputs? Do these two machines actually do the same thing? These questions and many others are pretty easy to answer for finite state machines, but extremely hard for Turing machines.

A similar thing can be said for quantum computations. The more powerful the gate set, the closer we are to fully universal quantum computation, and

the more difficult it becomes to simulate, optimise, and answer questions about, the quantum circuits we can build. Generally this should be regarded as A Good Thing. Indeed if it were possible to efficiently simulate universal quantum circuits, you wouldn't be reading this book in the first place.

However, it is nevertheless an interesting and fruitful question to ask what kinds of quantum computations *can* be simulated on a classical computer or otherwise efficiently reasoned about. It turns out this goes much farther than one might initially expect.

In **Chapter 4: CNOT Circuits and Phase-Free ZX-Diagrams**, we look just at those circuits constructible using CNOT gates, and show these are closely connected to the *phase-free* ZX-calculus, i.e. the calculus involving ZX-diagrams whose phase parameters are all restricted to zero. This will give us some important insights into how this single entangling gate interacts with itself, and how linear algebra over the two-element field $\mathbb{F}_2$ plays an important role in the structure of quantum computations. We will also meet the first automated simplification strategy for ZX diagrams, which always efficiently terminates for phase-free diagrams in what we call a **normal form**. These normal forms can tell us a lot about a computation, and also admit strategies for "extracting" circuits back out. Furthermore, these normal forms will come back with a vengeance when we discuss quantum error correcting codes in Chapter 11.

In **Chapter 5: Clifford Circuits and Diagrams**, we add the H and S gates to CNOT circuits to obtain a much richer, much "more quantum" family of circuits, the Clifford circuits, while still retaining lots of structure, and importantly, efficient classical simulability. Clifford circuits, despite leaving the realm of what we would normally call classical computation, are efficiently classically simulable thanks to the *Gottesman-Knill theorem*. We will take an unconventional route to proving this famous theorem by going via the corresponding fragment of the ZX-calculus: the *Clifford* ZX-calculus, whose phase parameters are restricted be integer multiples of $\frac{\pi}{2}$. Like in the phase-free case, we give an automated strategy for efficiently turning any Clifford ZX-diagram into a compact normal form. In fact, we'll look at two different normal forms, which are closely related, and can be applied to solve three important problems for Clifford circuits: efficient classical simulation, equality testing, and circuit optimisation/resynthesis.

In **Chapter 6: Stabiliser Theory**, we introduce an equivalent, and more widespread, technique for studying Clifford circuits, called stabiliser theory. Rather than restricting the class of gates used to build Clifford circuits, we give a more global, group-theoretic characterisation of the Cliffords in terms of certain subgroups of the Pauli group called stabiliser groups. Such a group

can always be represented by listing a small number of generators, and this in turn enables us to perform many tasks involving Clifford circuits efficiently by working not with the (exponentially large) unitary matrix, but with this set of generators of the stabiliser group. Using this formalism, we will see a more traditional approach to proving the Gottesman-Knill theorem, and also how to get the best of both worlds by converting back-and-forth between stabiliser and ZX representations.

Finally, in **Chapter 7: Universal Circuits**, we meet full-powered quantum computation. We will see that it is useful to treat universal circuits as Clifford circuits with a bit of "special sauce" mixed in: namely arbitrary Z-phase gates. We first see what happens when we mix arbitrary phase gates into CNOT circuits, and we find we can reuse some of the tricks from Chapter 5 to efficiently represent the resulting unitaries. A structure that emerges naturally is that of **phase polynomials**, which have a graphical ZX representation using **phase gadgets**. We will also show that there are two ways to extend phase polynomials/gadgets to model universal circuits: using **path sums** and **Pauli gadgets**, each with various benefits and applications. We will show how these structures can be used to implement a simple circuit optimisation technique called **phase folding**. We will find many other applications for these structures in Chapters 9–11.

The third part of the book extends the structures developed in part 2 in various different directions. Until this point, the dependency between chapters is essentially linear, but here it splits.

On one track is **Chapter 8: Measurement-based Quantum Computing**, where we introduce an alternative model of quantum computation called measurement-based quantum computation (MBQC), and show how it is equivalent to the circuit model. In the process, we will learn some new things about the structure of ZX diagrams, which turn out to be closely related to certain kinds of measurement-based quantum computations. Notably, we will see when it is possible to efficiently extract a quantum circuit from a ZX-diagram (or MBQC computation) using a graph-theoretic notion called **generalised flow**. While not strictly necessary to understand the following chapters, the idea of using measurements and corrections to deterministically implement quantum operations will be a running theme in this part of the book.

On the other track, we continue to develop the structure of universal quantum circuits. In **Chapter 9: Controlled Gates and Classical Oracles**, we see how the basic gates and ZX-diagrams we have studied so far can be used to construct higher-level gates, particularly to add control wires and to embed complex classical functions into quantum computations. We will

introduce a new spider-like generator for ZX-diagrams called the **H-box** which is especially convenient for dealing with these higher-level structures and explaining how phase gadgets and H-boxes are related to each other by a type of "graphical Fourier transform".

In **Chapter 10: Clifford+T**, we will specialise from the exactly universal computations we have been studying so far to Clifford+T circuits. Remarkably, if we add just one additional gate, called the T gate, to Clifford circuits, we obtain an approximately universal family of quantum circuits. The graphical analogue to such circuits are Clifford+T ZX-diagrams, i.e. diagrams whose phase parameters are integer multiples of $\frac{\pi}{4}$. It turns out that when we restrict the angles in this way (and more generally to angles of the form $\frac{\pi}{2^\ell}$), lots of interesting new structures start to emerge. We explore how these structures can be used to efficiently approximate any unitary using Clifford+T circuits, perform certain sophisticated circuit optimisations, and prove completeness and universality results for increasingly large fragments of the ZX/ZH calculus.

Finally, in **Chapter 11: Quantum Error Correction**, we introduce quantum error correction and fault-tolerant quantum computation. This brings together almost all of the concepts that have come before. Notably, we use the phase-free normal forms we met back in Chapter 4 to construct a graphical depiction of encoder maps, which enable error correction by embedding a collection of logical qubits into a larger collection of physical qubits. We see how to check for errors in this encoded quantum data using measurements represented by the Pauli projections from Chapter 6, and build toward a universal set of fault-tolerant operations for implementing quantum computations on encoded qubits. An important ingredient is something called **magic-state distillation**, which we can derive using the structure of T gates developed in Chapter 10. Once we distil magic states, we can inject them into our computations using a technique we met way back in Chapter 2! Putting all of these ingredients together, we obtain a way to implement universal quantum computation in a way that can, in principle, be made arbitrarily robust to errors.

So that's the plan. Let's get started!

2

The Quantum Circuit Model

The goal of this chapter is not to provide a comprehensive overview of quantum computation, as there are already plenty of good general-purpose books on that topic (see Section 1.1.1 for some pointers). In this chapter, we will simply lay out some of the basic ingredients from quantum theory needed to get us started working with circuits, then dive straight into their super interesting underlying structures, which will be the topic of the following chapters.

2.1   Preliminaries

In this section, we will lay out the mathematical foundations and notation which will be used throughout the book. We will introduce **Hilbert spaces** and various linear algebraic operations on them, and two different notations:

- **Dirac notation**, which is by far the most common notation appearing in the quantum computing literature. It gives a compact way to express states and transformations for a *single* quantum system, but starts to get clunky if we want to express lots of systems interacting with each other.
- **String diagram notation**: a graphical notation for expressing very general compositions of linear algebraic operations (called *tensor networks*). This is the main notation used in the so-called **quantum picturalism** approach to quantum theory, which was adopted exclusively e.g. by this book's predecessor *Picturing Quantum Processes*.

Readers familiar with the mathematics behind quantum theory, but unaccustomed to string diagram notation may wish to skip ahead to section 2.1.6.

2.1.1   Some things you should know about complex numbers

Before we go into defining Hilbert spaces, we'll need to make a couple of remarks about complex numbers. We will write $\mathbb{C}$ for the set of complex numbers, and we will often write $\lambda \in \mathbb{C}$ for an arbitrary complex number. There are two main ways to write down a complex number. The first is the familiar **cartesian form**: $\lambda = a + ib$ in terms of real numbers $a, b \in \mathbb{R}$ and $i := \sqrt{-1}$. The second, possibly less familiar, is the **polar form**: $\lambda = re^{i\alpha}$ for a positive real number $r \in \mathbb{R}_{\geq 0}$ called the **magnitude** and an angle $\alpha \in [0, 2\pi)$ called the **phase**. Phases play an important role in quantum theory, and they are often where the 'quantum magic' happens, so we'll run into the polar form a lot. These two representations are related by the trigonometric identity:

$$e^{i\alpha} = \cos \alpha + i \sin \alpha \quad \Longrightarrow \quad \left\{ \begin{array}{l} a = r\cos(\alpha) \\ b = r\sin(\alpha) \end{array} \right. \tag{2.1}$$

which can be visualised in the **complex plane** as follows:

$$\tag{2.2}$$

A useful operation on the complex numbers is the **complex conjugation** $\lambda \mapsto \overline{\lambda}$, which flips the sign of b in cartesian form or the sign of α in polar form:

$$a + ib \mapsto a - ib \qquad\qquad re^{i\alpha} \mapsto re^{-i\alpha}$$

It's pretty straightforward to derive from (2.1) that these two operations are the same for a complex number λ. This can be visualised by realising that flipping the sign of b or α amounts to a vertical reflection of the complex plane:
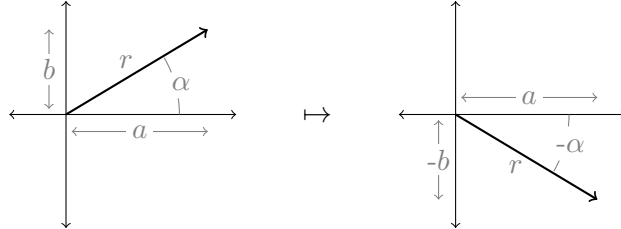
We will slightly overload the term 'phase', and also refer to complex numbers of the form $e^{i\alpha}$ as phases. If there is some ambiguity, we will call

$e^{i\alpha}$ the phase and α the **phase angle**. Because of the behaviour of exponents, when we multiply phases together, the phase angles add:

$$e^{i\alpha}e^{i\beta} = e^{i(\alpha+\beta)}$$

Consequently, multiplying a phase with its conjugate always gives 1:

$$e^{i\alpha}\overline{e^{i\alpha}} = e^{i\alpha}e^{-i\alpha} = e^{i(\alpha-\alpha)} = e^0 = 1.$$

More generally, we can always get the magnitude of a complex number by multiplying with it's conjugate and taking the square root:

$$\sqrt{\lambda\overline{\lambda}} = \sqrt{(re^{i\alpha})(re^{-i\alpha})} = \sqrt{r^2 e^0} = r$$

As a consequence, whenever we have $\lambda = re^{i\alpha}$ and $|\lambda| = 1$, we can conclude $r = 1$ so $\lambda = e^{i\alpha}$. In other words, $|\lambda| = 1$ if and only if λ is a phase.

Getting the angle out is a bit trickier. For this we use a trigonometric function called **arg**, which for $\lambda \neq 0$ is defined (somewhat circularly) as the unique angle α such that $\lambda = re^{i\alpha}$. If $\lambda = a+ib$ for $a > 0$, $\arg(\lambda) = \arctan(\frac{b}{a})$. This obviously won't work when a is zero, and needs a bit of tweaking when a is negative. Hence, the full definition of arg needs a case distinction, which we'll leave as an exercise.

Exercise 2.1. Just to make sure our trig is not too rusty, give a full definition for arg for all non-zero complex numbers, using (2.2) as a guide.

Exercise 2.2. Write $\cos\alpha$ and $\sin\alpha$ in terms of complex phases. Hint: remember that $\cos-\alpha = \cos\alpha$ while $\sin-\alpha = -\sin\alpha$.

2.1.2 Hilbert spaces

A **Hilbert space** is a vector space over the complex numbers, with an inner product defined between vectors, which we'll write this way:

$$\langle \psi, \phi \rangle \in \mathbb{C}$$

where ψ and ϕ are vectors in a Hilbert space H. Hilbert spaces can be finite- or infinite-dimensional. For finite-dimensional Hilbert spaces, we are already done with the definition. For infinite dimensions, we need to say some other stuff about limits and convergence, but since quantum computing is all about finite-dimensional Hilbert spaces, we won't need to go in to that here.

So, for our purposes, the main thing that separates a Hilbert space from a plain ole vector space is the inner product. So let's say a few things about inner products and introduce some handy notation. First, a definition.

Definition 2.1.1. A mapping $\langle -, - \rangle$ from a pair of vectors to the complex numbers is called an **inner product** if it satisfies the following three conditions:

1. *Linearity* in the second argument. For $\psi, \phi_1, \phi_2 \in H$ and $\lambda_1, \lambda_2 \in \mathbb{C}$:

$$\langle \psi | \lambda_1 \phi_1 + \lambda_2 \phi_2 \rangle = \lambda_1 \langle \psi | \phi_1 \rangle + \lambda_2 \langle \psi | \phi_2 \rangle.$$

2. *Conjugate-symmetry.* For $\psi, \phi \in H$: $\overline{\langle \phi | \psi \rangle} = \langle \psi | \phi \rangle$.
3. *Positive-definiteness.* For $\psi \in H$, $\langle \psi | \psi \rangle$ is a real number and $\langle \psi | \psi \rangle > 0$.

It might seem weird that we only ask inner products to be linear in one of the arguments. However, combining conditions 1 and 2, we can show that the inner product is actually **conjugate-linear** in the first argument. Conjugate-linearity is pretty much the same as linearity, but scalar multiples pop out as their conjugates:

$$\langle \lambda_1 \psi_1 + \lambda_2 \psi_2 | \phi \rangle = \overline{\lambda_1} \langle \psi_1 | \phi \rangle + \overline{\lambda_2} \langle \psi_2 | \phi \rangle$$

Example 2.1.2. Our main example of a Hilbert space is $\mathbb{C}^n$, the Hilbert space whose vectors are column vectors with n entries:

$$\psi = \begin{pmatrix} \psi^0 \\ \psi^1 \\ \vdots \\ \psi^{n-1} \end{pmatrix}, \qquad \phi = \begin{pmatrix} \phi^0 \\ \phi^1 \\ \vdots \\ \phi^{n-1} \end{pmatrix}, \qquad \cdots$$

Note we adopt the physicist's convention of allowing subscripts *and* superscripts to index entries in a vector or matrix. On the other hand, we adopt the computer scientists' convention of counting from 0. So for example, ψ^2 means the third entry of the column vector ψ, not 'ψ squared'. This looks a bit weird at first, but it will come in really handy when we get to section 2.1.4.

To take the inner product, we first take the conjugate-transpose ψ, i.e. we transpose the column vector into a row vector then conjugate every entry, then matrix multiply this with ϕ. This amounts to multiplying a $1 \times n$ matrix with an $n \times 1$ matrix, which gives us a 1×1 matrix, i.e. just a number:

$$\langle \psi | \phi \rangle := \begin{pmatrix} \overline{\psi^0} & \overline{\psi^1} & \cdots & \overline{\psi^{n-1}} \end{pmatrix} \begin{pmatrix} \phi^0 \\ \phi^1 \\ \vdots \\ \phi^{n-1} \end{pmatrix} = \sum_{i=0}^{n-1} \overline{\psi^i} \phi^i \qquad (2.3)$$

Since a Hilbert space has an inner product, we know a bunch of things about its vectors that a plain old vector space doesn't tell you. For one thing, the inner product tells us *how long* vectors are, i.e. the **norm** $\|\psi\| := \sqrt{\langle \psi | \psi \rangle}$, and in particular when a vector is **normalised**: $\|\psi\| = 1$. It also tells us when two vectors are **orthogonal**: $\langle \psi | \phi \rangle = 0$. Putting these pieces together we get the following.

Definition 2.1.3. A basis $\{h_i \,|\, 0 \leq i < n\} \subset H$ for an n-dimensional Hilbert space H is called an **orthonormal basis** (ONB) if:

$$\langle h_i | h_j \rangle = \delta_{ij} \qquad \text{where} \qquad \delta_{ij} := \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \text{ is the } \textbf{Kronecker delta}.$$

So, an inner product defines ONBs. In particular, if we were to equip the vector space with a different inner product, then which bases you consider orthonormal will change. This also works the other way: an ONB defines an inner product, in the sense of the following exercise.

Exercise 2.3. Show that any basis $\mathcal{B} = \{h_i \,|\, 0 \leq i < n\} \subset H$ on a vector space H defines a unique inner product that makes H into a Hilbert space and $\mathcal{B}$ into a ONB.

Since the inner product is linear in it's second argument, any vector $\psi \in H$ defines a linear map $\psi^\dagger : H \to \mathbb{C}$ by $\phi \mapsto \langle \psi | \phi \rangle$, which we call the **adjoint** of ψ. As you can see from (2.3), the adjoint of a column vector $\psi \in \mathbb{C}^n$ is the row vector given by its conjugate-transpose:

$$\psi = \begin{pmatrix} \psi^0 \\ \psi^1 \\ \vdots \\ \psi^{n-1} \end{pmatrix} \qquad \implies \qquad \psi^\dagger = \begin{pmatrix} \overline{\psi^0} & \overline{\psi^1} & \cdots & \overline{\psi^{n-1}} \end{pmatrix}$$

If I have a linear map between two (possibly) different Hilbert spaces H and K, the adjoint $A^\dagger$ of A is the unique linear map satisfying the equation $\langle A^\dagger \psi | \phi \rangle = \langle \psi | A \phi \rangle$ for all $\phi \in H$, $\psi \in K$. This definition takes a bit of head-scratching at first, but when $H = \mathbb{C}^m$ and $K = \mathbb{C}^n$, this again just

amounts to the conjugate-transpose of A:

$$A = \begin{pmatrix} a_0^0 & a_1^0 & \cdots & a_{m-1}^0 \\ a_0^1 & a_1^1 & \cdots & a_{m-1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ a_0^{n-1} & a_1^{n-1} & \cdots & a_{m-1}^{n-1} \end{pmatrix} \implies A^\dagger = \begin{pmatrix} \overline{a_0^0} & \overline{a_0^1} & \cdots & \overline{a_0^{m-1}} \\ \overline{a_1^0} & \overline{a_1^1} & \cdots & \overline{a_1^{m-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \overline{a_{n-1}^0} & \overline{a_{n-1}^1} & \cdots & \overline{a_{n-1}^{m-1}} \end{pmatrix}$$

Like with matrix transposes, one can show that adjoints satisfy $(A^\dagger)^\dagger = A$ and $(AB)^\dagger = B^\dagger A^\dagger$.

Inner products and adjoints are so ubiquitous in quantum theory that we typically build them right into the way we write vectors down. For this, we introduce **Dirac bra-ket notation**. Rather than writing a vector as it's usual naked self, we'll wrap it up into a symbol called a **ket**:

$$\psi \qquad \mapsto \qquad |\psi\rangle$$

If we want to write the adjoint of $|\psi\rangle$, we flip the ket into a **bra**:

$$\psi^\dagger \qquad \mapsto \qquad \langle\psi|$$

Now, the ket $|\psi\rangle$ is still secretly just ψ, a vector in the Hilbert space H. However, the bra $\langle\psi|$ is an element of the **dual space** H^*. That is, it is a linear map from H to the complex numbers. In particular, we can try plugging a ket into it, and some magic happens: we get a **bra-ket**, which is just the inner product again.

$$\langle\psi||\phi\rangle := \psi^\dagger\phi = \langle\psi|\phi\rangle$$

Breaking the two parts of an inner product apart gives us some extra flexibility. For example, if we have some linear map $M : H \to H$ that we want to sandwich in the middle of the inner product, we can do that: $\langle\psi|M|\phi\rangle$. This also leads us to more naturally think of bra's and ket's themselves as processes, which are dual to one-another. We can think of $|\phi\rangle$ as a process which starts with nothing and gives us something, namely the vector $\phi \in H$:

$$|\phi\rangle \qquad \rightsquigarrow \qquad \underset{\phi}{\triangleleft}\!\!-\!\!\overset{H}{-}$$

Conversely, we can think of $\langle\psi|$ as a process which 'swallows' a vector and leaves nothing (well, actually nothing but a scalar):

$$\langle\psi| \qquad \rightsquigarrow \qquad \overset{H}{-}\!\!\triangleright_{\psi}$$

The pictures to the right of '$\rightsquigarrow$' are in **string diagram notation**. This

gives us a way to visualise compositions of linear maps as boxes (or other kinds of shapes), connected by wires. We've already seen how to draw bras and kets above. The only thing missing at this point (at least until we meet tensor products in section 2.1.4) is linear maps $M : K \to L$ between two Hilbert spaces. These are pictured as a box with an input wire labelled H and output labelled K:

$$\xrightarrow{\quad H\quad} \boxed{M} \xrightarrow{\quad K\quad}$$

We visualise composition by plugging wires together. For example:

$$\langle\psi|\phi\rangle \qquad \leadsto \qquad$$

$$\langle\psi|M|\phi\rangle \qquad \leadsto \qquad$$

$$\langle\psi|CBA|\phi\rangle \qquad \leadsto \qquad$$

There's a couple of things to note about these two notations. First, the string diagrams give slightly more information, since the wires can be labelled to tell us which Hilbert spaces the maps are defined on. We can drop these labels when it is clear from context, but sometimes it's handy to get an extra visual cue for which maps can be plugged together, and which can't. Second, the Dirac notation is being read from right-to-left, whereas the pictures are being read from left-to-right. This is an artefact of the usual 'backwards' way composition is defined symbolically. For example, function composition $g \circ f(x) := g(f(x))$ means first apply f to an element x, then apply g to the result. In other words: $g \circ f$ means 'g AFTER f'. Similarly, with composition of linear maps (and hence matrix multiplications) $BA|\phi\rangle$ means first apply A to $|\phi\rangle$ then apply B to the result, i.e. BA means 'B AFTER A'.

This is not such a big deal, as long as we remember to flip the order of maps around when we write them in Dirac notation. What is more of a big deal, as we'll see in section 2.1.4, is that most quantum computations really want to be done in 2D. Dirac notation is only really handy for working with 1D chains of matrix multiplications like the ones above, and it gets a bit clunky when we start mixing (sequential) composition of linear maps with (parallel) tensor products. On the other hand, these more general kinds of composition are what string diagrams are meant for, so this is where they really start to shine.

Remark 2.1.4. In some circles, it is popular to symbolically write compositions in 'diagram order', using a notation like $f\,\text{;}\,g$ to mean $g \circ f$. We won't

use this notation in the book, preferring to jump to full-fledged diagrams instead.

Another advantage of writing vectors as kets is it gives us a compact way of writing the standard basis. Namely, we write the standard basis for $\mathbb{C}^n$ as $\{|i\rangle \,|\, 0 \leq i < n\}$, i.e. simply as numbers in a ket, ranging from 0 to $n-1$:

$$
|0\rangle := \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad |1\rangle := \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} \qquad \cdots \qquad |n-1\rangle := \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}
$$

In particular, the two-dimensional Hilbert space $\mathbb{C}^2$ has standard basis elements written as follows:

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

As we'll see in Section 2.2, $\mathbb{C}^2$ represents quantum bits, or **qubits**. In $\mathbb{C}^2$, the basis vectors $|0\rangle$ and $|1\rangle$ play the role of the classical bits 0 and 1. Part of the power of quantum computing is that we cannot just access these classical bit states, but also many more, coming from taking arbitrary linear combinations of them.

2.1.3 Types of linear maps

The adjoint operation $(-)^\dagger$ lets us define a handful of special kinds of linear maps that crop up in quantum theory. This first is isometries, where the adjoint acts as a one-sided inverse.

Definition 2.1.5. A linear map $U : H \to K$ is called an **isometry** if $U^\dagger U = I$.

Next, we have a whole slew of maps from a Hilbert space to itself.

Definition 2.1.6. A linear map $M : H \to H$ is called:

- **normal** if $M^\dagger M = MM^\dagger$
- a **unitary** if $M^\dagger M = I$ and $MM^\dagger = I$
- **self-adjoint** if $M = M^\dagger$
- **positive** if $M = N^\dagger N$ for some N
- a **projector** if $M = M^\dagger = M^2$

There are clearly some containments here: projectors are always positive (since $M = MM = M^\dagger M$), and positive maps are always self-adjoint (since $M^\dagger = (N^\dagger N)^\dagger = N^\dagger N = M$). Finally, everything in Definition 2.1.6 is normal. Unitaries are normal because $M^\dagger M = I = MM^\dagger$. Self-adjoint maps (and hence also positive maps and projectors) are normal because $MM^\dagger = M^2 = M^\dagger M$.

While this is a bit of a definition dump, we'll treat the most important kinds of maps from Definitions 2.1.5 and 2.1.6 – as well as their significance to quantum theory – in their own sections later in this chapter.

A nice feature about normal maps (and hence all the maps in Definition 2.1.6) is that they can always be **diagonalised**. That is, we can find an ONB $\mathcal{M} = \{|\phi_j\rangle\}_j$ such that for all i, $M|\phi_j\rangle = \lambda_j|\phi_j\rangle$. The scalars λ_j and vectors $|\phi_j\rangle$ are called **eigenvalues** and **eigenvectors**, respectively, whereas $\mathcal{M}$ is called an **eigenbasis**.

Bra-ket notation (and hence string diagram notation) gives us a convenient way to write maps in diagonal form.

$$M = \sum_j \lambda_j |\phi_j\rangle\langle\phi_j| = \sum_j \lambda_j \cdot \overset{H}{\underset{}{\phi_j}}\!\!\!\rangle \quad \langle\!\!\overset{H}{\underset{}{\phi_j}}$$

A special case is the identity map, which diagonalises with respect to *any* ONB, with eigenvalues all equal to 1:

$$I = \sum_j |\phi_j\rangle\langle\phi_j| = \sum_j \overset{H}{\underset{}{\phi_j}}\!\!\!\rangle \quad \langle\!\!\overset{H}{\underset{}{\phi_j}}$$

We call such a sum over an ONB a **resolution of the identity** by ONB $\{|\phi_i\rangle\}_i$.

As it turns out, all of the particular kinds of normal maps can be characterised by the types of eigenvalues they have.

Exercise 2.4. Let $M : H \to H$ be a linear map such that $M^\dagger M = MM^\dagger$. Then $M = \sum_j \lambda_j |\phi_j\rangle\langle\phi_j|$ for some sets $\{\lambda_j\}_j, \{|\phi_j\rangle\}_j$. Show that:

a) M is self-adjoint iff all $\lambda_j \in \mathbb{R}$
b) M is positive iff all $\lambda_j \in \mathbb{R}_{\geq 0}$
c) M is a projector iff all $\lambda_j \in \{0, 1\}$
d) M is a unitary iff all $\lambda_j \in U(1)$

where $\mathbb{R}_{\geq 0}$ is the set of all real numbers ≥ 0 and $U(1) := \{e^{i\alpha} \,|\, \alpha \in [0, 2\pi)\}$ is the set of all **phases**.

One thing you might notice is what happens when a map is unitary *and* self-adjoint: it's eigenvalues are all in $U(1) \cap \mathbb{R} = \{-1, 1\}$. This is indeed

what happens with the *Pauli maps*, which as we'll see later, have many nice properties.

2.1.4 Tensors and the tensor product

Quantum computing relies crucially on multiple systems interacting with each other. To build up to how we should represent multiple systems in quantum theory, we can first see what happens if we bring two classical systems together. The simplest classical system that has more than one state is a **bit**, which we can represent as the set $\mathbb{B} := \{0, 1\}$. A single bit has two possible states: 0 and 1. A pair of bits has four possible states: 00, 01, 10, and 11. In other words, the system describing two bits is the **cartesian product** of two one-bit systems:

$$\mathbb{B} \times \mathbb{B} = \{(0,0), (0,1), (1,0), (1,1)\}$$

Since the role of classical bit values 0 and 1 in the qubit system $\mathbb{C}^2$ is played by the basis vectors $|0\rangle$ and $|1\rangle$, it stands to reason that the system of two qubits should have four basis vectors $|00\rangle, |01\rangle, |10\rangle, |11\rangle$, labelled by the four possible bit strings. This is exactly what taking the tensor product does for us.

There are lots of ways to define the tensor product of two vector spaces. For the sake of simplicity, we'll adopt a fairly brutal definition here, which makes explicit use of some fixed bases for a pair of Hilbert spaces.

Definition 2.1.7. For an m-dimensional Hilbert space H and an n-dimensional Hilbert space K with fixed ONBs $\{|i\rangle \,|\, 0 \leq i < m\} \subset H$ and $\{|j\rangle \,|\, 0 \leq j < n\} \subset K$ the **tensor product** $H \otimes K$ is the mn-dimensional Hilbert space consisting of all linear combinations of states in the following **product basis**:

$$\{|i\rangle \otimes |j\rangle \,|\, 1 \leq i \leq m, 1 \leq j \leq n\}$$

The inner product on this tensor product is fully determined by requiring $\{|i\rangle \otimes |j\rangle\}_{i,j}$ to form an ONB. That is, we have: $\langle |i\rangle \otimes |j\rangle, |k\rangle \otimes |l\rangle \rangle = \delta_{ik}\delta_{jl}$.

We call this a 'brutal' definition, because the tensor product doesn't *really* depend on a choice of basis, but choosing a basis will make it easier for us to see concretely what's going on. Note that the symbol $\otimes$ used in defining the product basis doesn't really mean anything in it's own right. The important thing is just that we have one basis vector for each pair of basis vectors from H and K. We could just as well have written basis elements as $|ij\rangle$. We will indeed do this sometimes, e.g. we will write the basis vectors of the

four-dimensional space $\mathbb{C}^2 \otimes \mathbb{C}^2$ as:

$$|00\rangle := |0\rangle \otimes |0\rangle$$
$$|01\rangle := |0\rangle \otimes |1\rangle$$
$$|10\rangle := |1\rangle \otimes |0\rangle$$
$$|11\rangle := |1\rangle \otimes |1\rangle$$

However, this notation is suggestive of how we want to send a pair of states $|\psi\rangle \in H$ and $|\phi\rangle \in K$ into the bigger tensor product space $H \otimes K$. Suppose these states decompose in our chosen bases as follows:

$$|\psi\rangle := \sum_{i=1}^{m} \psi^i |i\rangle \qquad |\phi\rangle := \sum_{j=1}^{n} \phi^j |j\rangle$$

Then, we'll define the **product state** as follows, just by pulling the sums and scalars out:

$$|\psi\rangle \otimes |\phi\rangle = \left(\sum_i \psi^i |i\rangle \right) \otimes \left(\sum_j \phi^j |j\rangle \right) := \sum_{ij} \psi^i \phi^j |i\rangle \otimes |j\rangle$$

This looks a bit abstract, but if we instantiate this to the case of qubit states, it becomes pretty obvious what is going on. We just 'multiply things out':

$$(\psi^0 |0\rangle + \psi^1 |1\rangle) \otimes (\phi^0 |0\rangle + \phi^1 |1\rangle) =$$
$$\psi^0 \phi^0 |00\rangle + \psi^0 \phi^1 |01\rangle + \psi^1 \phi^0 |10\rangle + \psi^1 \phi^1 |11\rangle$$

If we write things out as column vectors, it should be even more clear what's going on:

$$\begin{pmatrix} \psi^0 \\ \psi^1 \end{pmatrix} \otimes \begin{pmatrix} \phi^0 \\ \phi^1 \end{pmatrix} = \begin{pmatrix} \psi^0 \phi^0 \\ \psi^0 \phi^1 \\ \psi^1 \phi^0 \\ \psi^1 \phi^1 \end{pmatrix} \tag{2.4}$$

Note that, by writing $|\psi\rangle \otimes |\phi\rangle$ as a four-dimensional column vector, we are implicitly treating the four basis states as the standard basis for $\mathbb{C}^4$, just by counting them base-2:

$$\{ |0\rangle := |00\rangle, |1\rangle := |01\rangle, |2\rangle := |10\rangle, |3\rangle := |11\rangle \}$$

This is a general phenomenon. When we take the tensor products of Hilbert spaces, the dimensions multiply. Hence, we can treat a basis vector $|i, j\rangle \in \mathbb{C}^n \otimes \mathbb{C}^{n'}$ as a basis vector $|in' + j\rangle \in \mathbb{C}^{nn'}$.

So, taking the tensor product of $\mathbb{C}^m$ with $\mathbb{C}^n$ just multiplies the dimensions. Since every finite-dimensional Hilbert space is isomorphic to $\mathbb{C}^n$ for some n,

it follows that the tensor product is associative and has a unit given by the one-dimensional Hilbert space $\mathbb{C} = \mathbb{C}^1$:

$$(H \otimes K) \otimes L \cong H \otimes (K \otimes L) \qquad H \otimes \mathbb{C} \cong H \cong \mathbb{C} \otimes H$$

The symbol '$\cong$' here means 'is isomorphic to'. Because of this associativity we are justified in dropping the brackets when writing tensor products of many different spaces, e.g. $H_1 \otimes \ldots \otimes H_k$.

The equation (2.4) is actually a special case of a more general kind of operation we can apply to matrices of any size.

Definition 2.1.8. The **Kronecker product** of an $n \times m$ matrix A with an $n' \times m'$ matrix B is a new $nn' \times mm'$ matrix $A \otimes B$ whose entries are all possible products of the entries of A and B, i.e.

$$(A \otimes B)_{im'+j}^{kn'+l} := a_i^k b_j^l$$

While this definition might be a bit of head-scratcher the first time you see it, it's pretty easy to think about in terms of block matrices. To form the matrix $A \otimes B$, we form a big matrix which has a block consisting of the matrix $a_i^j B$ for every element of A. That is, for matrices:

$$A := \begin{pmatrix} a_0^0 & \cdots & a_{m-1}^0 \\ \vdots & \ddots & \vdots \\ a_0^{n-1} & \cdots & a_{m-1}^{n-1} \end{pmatrix} \quad \text{and} \quad B := \begin{pmatrix} b_0^0 & \cdots & b_{m'-1}^0 \\ \vdots & \ddots & \vdots \\ b_0^{n'-1} & \cdots & b_{m'-1}^{n'-1} \end{pmatrix}$$

the Kronecker product is the $nn' \times mm'$ matrix:

$$A \otimes B := \begin{pmatrix} a_0^0 B & \cdots & a_{m-1}^0 B \\ \vdots & \ddots & \vdots \\ a_0^{n-1} B & \cdots & a_{m-1}^{n-1} B \end{pmatrix}$$

This applies to any dimension of matrix, not just matrices with the same numbers of rows or columns. For example, the Kronecker product of the 2×1 matrix of a state $|\psi\rangle$ and the 2×2 matrix of map A is computed as follows:

$$|\psi\rangle \otimes A = \begin{pmatrix} \psi^0 \\ \psi^1 \end{pmatrix} \otimes \begin{pmatrix} a_0^0 & a_1^0 \\ a_0^1 & a_1^1 \end{pmatrix} = \begin{pmatrix} \psi^0 A \\ \psi^1 A \end{pmatrix} = \begin{pmatrix} \psi^0 a_0^0 & \psi^0 a_1^0 \\ \psi^0 a_0^1 & \psi^0 a_1^1 \\ \psi^1 a_0^0 & \psi^1 a_1^0 \\ \psi^1 a_0^1 & \psi^1 a_1^1 \end{pmatrix}$$

The Kronecker product is furthermore *non-commutative*: $B \otimes A \neq A \otimes B$.

The matrix of $B \otimes A$ will be the same size and contains the same elements, but those elements will be in different places. For example:

$$A \otimes |\psi\rangle = \begin{pmatrix} a_0^0 & a_1^0 \\ a_0^1 & a_1^1 \end{pmatrix} \otimes \begin{pmatrix} \psi^0 \\ \psi^1 \end{pmatrix} = \begin{pmatrix} a_0^0|\psi\rangle & a_1^0|\psi\rangle \\ a_0^1|\psi\rangle & a_1^1|\psi\rangle \end{pmatrix} = \begin{pmatrix} a_0^0\psi^0 & a_1^0\psi^0 \\ a_0^0\psi^1 & a_1^0\psi^1 \\ a_0^1\psi^0 & a_1^1\psi^0 \\ a_0^1\psi^1 & a_1^1\psi^1 \end{pmatrix}$$

Often, rather than smooshing the upper and lower indices together into a single index, as we did in the definition of Kronecker product, we can keep them separate and simply allow 'generalised matrices' that have zero or more upper and lower indices:

$$m\left\{ \begin{array}{|c|} \hline \vdots\; T\; \vdots \\ \hline \end{array} \right\}n \quad \rightsquigarrow \quad T = \{t^{j_0...j_{n-1}}_{i_0...i_{m-1}} \in \mathbb{C} \mid\; 0 \le i_\mu < D_\mu, 0 \le j_\nu < D'_\nu\}$$

Note, rather than having a fixed number of 'rows' and 'columns', we have fixed *input dimensions* $D_0, \ldots, D_{m-1}$ and *output dimensions* $D'_0, \ldots, D'_{n-1}$. These generalised matrices are called **tensors**.

In fact, we've already met some tensors: kets are represented by tensors with zero inputs and one output:

$$\begin{array}{c} \underset{\psi}{\triangleleft}\!\!\!-\!\!\!\overset{H}{-} \end{array} \quad \rightsquigarrow \quad \left\{\, \psi^i \in \mathbb{C} \,\middle|\, 0 \le i \le \dim(H) \,\right\}$$

bras by tensors with one input and zero outputs:

$$\begin{array}{c} \overset{H}{-}\!\!\!\underset{\psi}{\triangleright} \end{array} \quad \rightsquigarrow \quad \left\{\, \psi_i \in \mathbb{C} \,\middle|\, 0 \le i \le \dim(H) \,\right\}$$

and plain old numbers are tensors with no inputs and no outputs:

$$\boxed{\lambda} \quad \rightsquigarrow \quad \left\{\, \lambda \in \mathbb{C} \,\right\}$$

We can relate tensors to bra-ket notation by taking a big sum over all the basis elements, with the tensor elements as coefficients:

$$T = \sum_{\substack{i_0...i_{m-1} \\ j_0...j_{n-1}}} t^{j_0...j_{n-1}}_{i_0...i_{m-1}} |j_0...j_{n-1}\rangle\langle i_0...i_{m-1}|$$

A tensor with two indices of dimensions n and n' has the same data as one with a single index of dimension nn', just packaged up in a different way. With that in mind, we can simplify the definition of the Kronecker product to give us a very similar operation, which is often just called the **tensor product**:

$$(A \otimes B)^{k,l}_{i,j} := a_i^k b_j^l$$

This readily generalises to tensors with lots of indices:

$$(S \otimes T)^{k_0\dots,\,l_0\dots}_{i_0\dots,\,j_0\dots} := s^{k_0\dots}_{i_0\dots} t^{l_0\dots}_{j_0\dots}$$

In either case, we represent the tensor product in string diagrams by 'stacking boxes':

$$A \otimes B \qquad \rightsquigarrow \qquad \boxed{A} \atop \boxed{B} \qquad\qquad S \otimes T \qquad \rightsquigarrow \qquad \boxed{S} \atop \boxed{T}$$

We can sequentially compose tensors the same way we would matrices. For matrix composition, we get the components of BA by multiplying components A and B together, and *contracting* (i.e. summing over) the output index of A with the input index of B:

$$BA \qquad \rightsquigarrow \qquad \boxed{A}\boxed{B} \qquad \Longrightarrow \qquad (BA)^k_i := \sum_j a^j_i b^k_j$$

For tensors, the story is much the same, except we contract all the upper indices of one with the lower indices of the other:

$$TS \qquad \rightsquigarrow \qquad \boxed{S}\boxed{T} \qquad \Longrightarrow \qquad (TS)^{k_0\dots}_{i_0\dots} := \sum_{j_0\dots} s^{j_0\dots}_{i_0\dots} t^{k_0\dots}_{j_0\dots}$$

As the pictures indicate, we should think of BA as a **sequential composition** and $A \otimes B$ as **parallel composition**. When we start mixing these two together, the magic happens.

Exercise 2.5. We have seen that we can construct some vectors in the tensor product by combining states of the component Hilbert spaces together to form a *product state*. This raises the question:

Are all the states in $H \otimes K$ product states?

The answer is no. We know that any product state e.g. in $\mathbb{C}^2 \otimes \mathbb{C}^2$ has the form given in equation (2.4). Find a vector $|\psi\rangle$ in $\mathbb{C}^2 \otimes \mathbb{C}^2$ that is not a product state and prove that is the case by by showing the following equation is not solvable:

$$\begin{pmatrix} \psi^0 \phi^0 \\ \psi^0 \phi^1 \\ \psi^1 \phi^0 \\ \psi^1 \phi^1 \end{pmatrix} = |\psi\rangle = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

2.1.5 Sums and diagrams

Since these diagrams describe linear maps, it makes perfect sense to take linear combinations of them: for linear maps f, g and scalars λ, μ, the linear map $\lambda f + \mu g$ sends a vector v to $\lambda f(v) + \mu g(v)$. At the level of matrices, this amounts to scalar multiplication and adding matrices together element-wise.

In fact, we were already doing this back in Section 2.1.3 when we wrote diagonalisation as:

$$M = \sum_j \lambda_j \cdot \overset{H}{\underset{}{\triangleright}}_{\phi_j} \quad \triangleleft_{\phi_j} \overset{H}{\underset{}{}}$$

Conveniently, we can summarise all of the various linearity and bilinearity conditions involving composition and tensor products into one principle:

Sums distribute over string diagrams.

In other words, a summation occurs somewhere inside a diagram, it can always be pulled to the outside. We can see how this works using the following example. Consider the following quantum state, which is often called the (unnormalised) **Bell state**:

$$|\Phi\rangle = \sum_i |ii\rangle = \sum_i \overset{\triangleleft_i}{\underset{\triangleleft_i}{}}$$

This state famously satisfies the following **yanking equation** with its adjoint $\langle\Phi|$. Shown both in bra-ket notation and as a string diagram, this equation is the following:

$$(\langle\Phi| \otimes I)(I \otimes |\Phi\rangle) = I \qquad \overset{\Phi}{\underset{\Phi}{}} = \text{———} \qquad (2.5)$$

Intuitively, we can think of the diagram of the left-hand side as a zig-zag shape that we are "yanking" into a straight piece of wire. We can prove this by substituting the definitions of $|\Phi\rangle$ and $\langle\Phi|$ into the diagram and pulling the sums out to the outside:
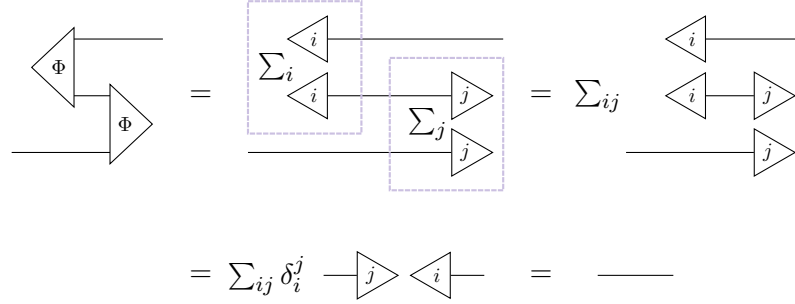
2.1.6   Tensor networks and string diagrams

What happens when we compose two matrices A, B in sequence and two more matrices C, D in sequence, then compose the results in parallel?

$$(BA \otimes DC)^{k,l}_{i,j} = (BA)^{k}_{i}(DC)^{l}_{j} = \sum_{x} a^{x}_{i} b^{k}_{x} \cdot \sum_{y} c^{y}_{j} d^{l}_{y} = \sum_{xy} a^{x}_{i} b^{k}_{x} c^{y}_{j} d^{l}_{y}$$

How about when we do it the other way around?

$$[(B \otimes D)(A \otimes C)]^{k,l}_{i,j} = \sum_{xy} (A \otimes C)^{x,y}_{i,j}(B \otimes D)^{k,l}_{x,y} = \sum_{xy} a^{x}_{i} c^{y}_{j} b^{k}_{x} d^{l}_{y} = \sum_{xy} a^{x}_{i} b^{k}_{x} c^{y}_{j} d^{l}_{y}$$

We get the same thing! The resulting equation is called the **interchange law**:

$$BA \otimes DC = (B \otimes D)(A \otimes C)$$

Let's see what happens if we do the same thing in the string diagram notation. Doing the composition one way around, we plug A and B together in sequence, plug C and D together in sequence, then stack the results in parallel:

The other way around, we stack A and C in parallel, then stack B and D in parallel, then plug the results together in sequence:

We see that the interchange law becomes completely trivial in the language

of string diagrams:

$$\frac{\boxed{A}\ \boxed{B}}{\boxed{C}\ \boxed{D}} \quad = \quad \frac{\boxed{A}\ \boxed{B}}{\boxed{C}\ \boxed{D}}$$

This is a good thing! The interchange law isn't capturing something fundamental about the processes we are studying, but is instead just a bit of unavoidable bureaucracy we have to deal with to fit sequential and parallel composition all in one line. The more our notation can swallow this bureaucracy and let us focus on what's really important, the better!

You might have noticed that, while we have already been using string diagrams in this section, we haven't yet defined them in generality. For our purposes, string diagrams are a graphical notation for a general kind of composition of tensors called a **tensor network**.

A tensor network is a particular way of composing tensors, where the elements of each of the component tensors are multiplied together and some pairs of indices are matched and summed over. We can represent this as a string diagram by depicting each of the component tensors as boxes and indices as wires. Wires that are open at one end represent *free* indices (i.e. the indices corresponding to inputs/outputs of the tensor), whereas wires that are connected at both ends correspond to *bound indices*. The latter are matched pairs of indices that get summed over.

This might sound a little abstract, but if we see an example, it should be pretty clear what is going on. Suppose we have the following string diagram describing a big linear map:

$$(2.6)$$

We can (arbitrarily) assign index names to each of the wires:

Then, this diagram describes the following tensor network:

$$\Phi^{de}_{abc} = \sum_{xyz} f^{xd}_{ab} g^{ez}_{xy} h^{y}_{cz} \tag{2.7}$$

We will typically work purely with string diagrams and omit the indices, which are only relevant inasmuch as they uniquely identify each wire in the diagram. One thing to notice is that wires can freely cross over each other and even form feedback loops. Also, if we deform the string diagram and draw it different on the page, it will still describe the same calculation (2.7). For example:
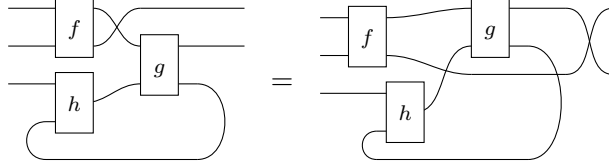
In other words, the only relevant data is which inputs and outputs are connected to which others. This fact can be summarised in the string diagram mantra:

Only connectivity matters!

The final thing to note is we can also build up arbitrary tensor networks from $\circ$ and $\otimes$, provide we add two additional ingredients: **swap** maps and the partial **trace** operation. The former is a linear map which interchanges the two tensor factors of a vector in $A \otimes B$:

$$\mathrm{SWAP} \;=\; \times$$

By linearity, it is totally defined by how it acts on product vectors:

$$\mathrm{SWAP} :: |\psi\rangle \otimes |\phi\rangle \mapsto |\phi\rangle \otimes |\psi\rangle$$

Note that the SWAP for two qubits has the following matrix:

$$\times \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.8}$$

The latter operation, the partial trace, sums the last input index together with the last output index of a tensor. That is, for a linear map $f : A \otimes X \to B \otimes X$, $\mathrm{Tr}_X(f) : A \to B$ is a linear map whose matrix elements are given by:

$$\mathrm{Tr}_X(f)_i^j = \sum_k f_{ik}^{jk}$$

In the special case where a map has only one input and output, this is just the normal trace, i.e. summing over the diagonal elements of a matrix:

for $g : X \to X$ we have $\mathrm{Tr}(g) = \sum_k g_k^k$. Graphically, we depict the partial trace as introducing a feedback loop:

$$\mathrm{Tr}_X \left(\begin{array}{c} A \quad\quad B \\ X \;\boxed{f}\; X \end{array} \right) = \begin{array}{c} A \quad\quad B \\ X \;\boxed{f}\; X \end{array}$$

Exercise 2.6. Write an expression for (2.6) using $\otimes, \circ, \mathrm{SWAP}$, and $\mathrm{Tr}(-)$. Show that it evaluates to the same thing as (2.7).

2.1.7 Cups and caps

If this is your first time seeing this notation for the partial trace it might make you... a little uncomfortable. Don't these feedback loops introduce all kinds of nasty complications like grandfather paradoxes and other problems to do with time travel? One way to see that it is in fact all fine, is to remember that the string diagrams are just notation for particular linear maps, and a wire denotes an index you contract over, where a feedback loop simply means we are contracting an input and an output 'in the wrong order'. So you can just 'shut up and calculate' and treat the loops as a handy piece of notation.

But we can also go one step further and decompose the loop into a couple of smaller pieces, which correspond to some special types of tensors. We have in fact already seen these pieces: they are the Bell state $|\Phi\rangle$ and effect $\langle\Phi|$. We introduce some special notation for these tensors:

$$\subset \; := \; \langle\Phi|\qquad\qquad \supset \; := \; |\Phi\rangle \tag{2.9}$$

The reason this notation works is because it makes the yanking equation of Eq. (2.5) particularly nice:

$$\begin{array}{c} \Large\supset \end{array} = \; \text{——} \qquad\qquad \Large\infty \; = \; \subset \tag{2.10}$$

Here the second equation states the state is symmetric in its two outputs. In the context of string diagrams we call this special state the **cup** and the effect the **cap**. Note that as matrices we have:

$$\subset \; = \; \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \qquad\qquad \supset \; = \; \begin{pmatrix} 1 & 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

These are the matrices when we consider the cup and cap of qubits. But we do in fact have a cup and cap for any dimension. We can also write the cup and cap using indices: $\subset^{ij} = \delta_{ij}$ and $\supset^{ij} = \delta_{ij}$, where δ_{ij} is the standard Kronecker delta. In this notation it becomes especially clear that the cup and cap are like 'bended identities', since $\mathrm{id}_j^i = \delta_{ij}$.

Using the cup and cap we can view a trace not as a feedback loop, but as a composition of a cup, a cap, and an identity wire connecting the two of them:
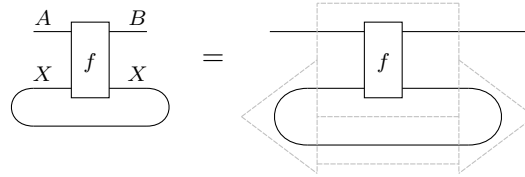
We are then left with a string diagram where every input is connected to an output, and not vice versa.

2.2   A bluffer's intro to quantum theory

Okay, so now we have all the mathematical apparatus to describe quantum theory. But what is quantum theory actually? You will have almost certainly heard of *quantum mechanics*, and you may have heard of a few things starting with the word 'quantum', like *quantum optics*, *quantum electrodynamics*, *quantum chromodynamics*, and the *standard model* (okay, that last one doesn't start with 'quantum', but it's still relevant here). These are all theories of physics, which cover the behaviours of certain kinds of things that exist in the world (electrons, photons, quarks, the Higgs boson, ...).

They are all underpinned by *quantum theory*, which can be seen more as a framework to build theories of physics on top of, rather than a full-fledged theory of physics in its own right. All the theories of physics we might build based on quantum theory have some characteristics in common, which can pretty much be summed up in the following 4 statements, which we'll call the **SCUM postulates**.

  S. **States** of a quantum system correspond to vectors in a Hilbert space.
  C. **Compound systems** are represented by the tensor product.
  U. **Unitary maps** evolve states in time.
  M. **Measurements** are computed according to the Born rule.

These 4 postulates are a (loose) paraphrase of the **Dirac-von Neumann axioms** of quantum theory, where we've put a bit more emphasis on the

bits that are most relevant for quantum computing. Everything in bold above will get defined soon enough. The point here is to show that there really isn't very much at the core of quantum theory. Quantum theory is in fact very simple, and yet it somehow happens to be very good at making predictions about a whole lot of different kinds of physical systems. However, if you have encountered other theories of physics which can be derived from simple postulates (and one in particular about space and time proposed by a dude with crazy hair), you'll notice something a little strange about the von Neumann postulates: they are all about *maths* and not at all about *stuff*. This is pretty odd for a theory of physics, because of course physics is fundamentally about stuff, and mathematics is just the tool.

This has led to almost a century of dissatisfaction about the 'fundamental' nature of quantum theory, and led to a lot of interesting arguments, many books, and the whole field called *quantum foundations*, which asks not just *what* quantum theory is, but *why* it is the way it is. That is to say, if after reading the four postulates and learning what all the words in bold mean, you are still scratching your head about what the words in bold *mean*, you are actually in good company.

2.2.1 Quantum states

There are a handful of equivalent ways we can represent a quantum state. The first is as a normalised vector $|\psi\rangle$ in a Hilbert space H. This representation is the simplest, but also contains a bit of redundancy, because as we will see in Section 2.2.4, quantum theory will make the exact same predictions about a state $|\psi\rangle$ as it will about $e^{i\alpha}|\psi\rangle$ for any angle α. Hence, there is no physical difference between $|\psi\rangle$ and $e^{i\alpha}|\psi\rangle$. We call the scalar factor $e^{i\alpha}$ a **global phase**, and we say the states $|\psi\rangle$ and $e^{i\alpha}|\psi\rangle$ are **equivalent up to global phase**.

So, technically, a quantum state is not just one vector, but a set of vectors $\{e^{i\alpha}|\psi\rangle \,|\, \alpha \in [0, 2\pi)\}$ which are equivalent up to a global phase. But then, this is just all the normalised states in a one-dimensional subspace of H.

Proposition 2.2.1. The set of all normalised states in a one-dimensional subspace S of a Hilbert space H is always of the form $\{e^{i\alpha}|\psi\rangle \,|\, \alpha \in [0, 2\pi)\}$ for some normalised state $|\psi\rangle$.

Proof Every vector in a one-dimensional space is a scalar multiple of some fixed vector, i.e. such spaces are always of the form $S := \{\lambda|\psi\rangle \,|\, \lambda \in \mathbb{C}\}$ for some non-zero vector $|\psi\rangle$. Since S contains every scalar multiple of $|\psi\rangle$, we can also assume without loss of generality that $|\psi\rangle$ is normalised. From this

it follows that any state of the form $e^{i\alpha}|\psi\rangle$ is a normalised state in S:

$$(e^{i\alpha}|\psi\rangle)^\dagger e^{i\alpha}|\psi\rangle = (e^{-i\alpha}\langle\psi|)(e^{i\alpha}|\psi\rangle) = 1 \cdot \langle\psi|\psi\rangle = 1$$

Conversely, if $\lambda|\psi\rangle \in S$ is normalised, we have:

$$1 = (\lambda|\psi\rangle)^\dagger(\lambda|\psi\rangle) = \overline{\lambda}\lambda\langle\psi|\psi\rangle = \overline{\lambda}\lambda = |\lambda|^2$$

But, as we saw in Section 2.1.1, $|\lambda|^2 = 1$ if and only if it is of the form $e^{i\alpha}$ for some α. Hence, all the normalised states in S are of the form $e^{i\alpha}|\psi\rangle$. $\qquad\square$

So, the second equivalent way to represent a quantum state is to write down it's one-dimensional subspace S. Picking any normalised vector in that space will recover $|\psi\rangle$, up to a global phase, whereas looking at the space spanned by $|\psi\rangle$ will recover S.

A third equivalent way to represent a quantum state is by **doubling** $|\psi\rangle$. Somewhat counter-intuitively, we can make the representation of the state *less* redundant by putting two copies of $|\psi\rangle$ together. Or, more precisely, by multiplying the ket $|\psi\rangle$ with its adjoint bra $\langle\psi|$:

$$|\psi\rangle \qquad \rightsquigarrow \qquad |\psi\rangle\langle\psi|$$

The result is a linear map $P_{|\psi\rangle} := |\psi\rangle\langle\psi|$ which goes from H to H. So, why is this representation actually *less* redundant than just taking the ket $|\psi\rangle$? Well, if we think about doubling an equivalent state $|\phi\rangle := e^{i\alpha}|\psi\rangle$, something magic happens:

$$|\phi\rangle\langle\phi| = (e^{i\alpha}|\psi\rangle)^\dagger(e^{i\alpha}|\psi\rangle) = e^{-i\alpha}e^{i\alpha}|\psi\rangle\langle\psi| = |\psi\rangle\langle\psi|$$

The global phase disappears!

So, what's going on here? There's a couple of ways to think about this. The most natural way to see this, is that it has something to do with measurement probabilities, and the actual reason we don't care about phases in the first place. We'll return to this point in Section 2.2.4, once we know how measurement probabilities are computed. For now, perhaps the best way to think about this is that $|\psi\rangle\langle\psi|$ is just another way of representing a one-dimensional subspace S, which we already know is an equivalent way of representing a quantum state.

To see that, first recall from Definition 2.1.6 that a **projector** P is a linear map that is self-adjoint ($P^\dagger = P$) and **idempotent** ($P^2 = P$). Projectors are linear maps that 'squash' vectors down into a subspace, called the **range** of the projector: $S := \{ |\phi\rangle \,|\, \exists|\psi\rangle \in H : |\phi\rangle = P|\psi\rangle \} \subseteq H$. If S is an n-dimensional subspace, we say P is a **rank-n projector** on to S.

Now, it happens to be that $P_{|\psi\rangle} := |\psi\rangle\langle\psi|$ is a rank-1 projector, whose range is exactly $S := \{\, \lambda|\psi\rangle \mid \lambda \in \mathbb{C} \,\}$. If we draw this as:

$$|\psi\rangle\langle\psi| \qquad \rightsquigarrow \qquad \overset{H}{\triangleright}_{\psi} \quad {}_{\psi}\triangleleft^{H}$$

we can imagine vectors coming in the wire on the left, then getting 'squashed' down to nothing (i.e. just a scalar factor), then getting embedded back into the H as a scalar times $|\psi\rangle$:

$$\triangleleft_{\phi}\!\!\overset{H}{—}\!\!\triangleright_{\psi} \quad {}_{\psi}\triangleleft\overset{H}{—} \quad = \quad \lambda \cdot \;\; {}_{\psi}\triangleleft\overset{H}{—} \qquad \text{where} \qquad \lambda := \triangleleft_{\phi}\!\!\overset{H}{—}\!\!\triangleright_{\psi}$$

Exercise 2.7. Show $|\psi\rangle\langle\psi|$ is a projector with image $S := \{\, \lambda|\psi\rangle \mid \lambda \in \mathbb{C} \,\}$. Conversely, show that if P is a projector with 1D range S, $P = |\psi\rangle\langle\psi|$.

So, that completes our three equivalent pictures of the quantum state. A **quantum state** is:

1. a normalised vector $|\psi\rangle \in H$, up to a redundant global phase $e^{i\alpha}$,
2. a one-dimensional subspace $S \subseteq H$, or
3. a rank-1 projector $|\psi\rangle\langle\psi|$.

Of course, this really just tells us how a quantum state is represented mathematically, which is enough for our purposes. What the quantum state it is *actually representing* is a whole other question, which has been debated for more than a century. We'll make some remarks about this and point to some further reading in Section 2.7.

Remark 2.2.2. Note that the quantum states we introduced in this section are sometimes called *pure* quantum states, to distinguish them from more general *mixed* quantum states. The latter allow us to represent having only partial information or access to a quantum system. For the most part, we won't need this extra generality for the topics covered in this book, so we'll focus purely on pure states for now.

2.2.2 Qubits and the Bloch sphere

We now turn our attention to the most interesting system from the point of view of quantum computation: the quantum bit, or **qubit**. We'll see in the next section that the one-dimensional Hilbert space $\mathbb{C}^1 = \mathbb{C}$ is the trivial, 'no system' system. So, to get something non-trivial, we should go one dimension up, to $\mathbb{C}^2$. Hence, a qubit is a system whose state lives in the Hilbert space $\mathbb{C}^2$. Physically, qubits can be implemented with any physical system that

has at least 2 states that can be perfectly distinguished by a single quantum measurement (which we'll cover in Section 2.2.4).

As the name suggests, a qubit is the quantum analogue of a bit. As such, we'll give the two standard basis elements some suggestive names:

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

Graphically, we will depict these states as follows:

These are the quantum analogues of the two possible states 0 and 1 that a classical bit could take. In Section 2.3, we'll see how this embedding of classical bits into two-dimensional vector spaces can be used to lift classical logic gates on bits to quantum logic on qubits.

Because of this connection to classical computation, the basis $\{|0\rangle, |1\rangle\}$ is often called the *computational basis*. We will occasionally use this term, but will instead mostly call it the **Z-basis**, for reasons that will shortly become clear.

We know from the last section that quantum states in $\mathbb{C}^2$ correspond to normalised vectors $|\psi\rangle \in \mathbb{C}^2$, up to a global phase. We'll now use this fact to find a convenient way to parametrise a generic state $|\psi\rangle := a|0\rangle + b|1\rangle$ in $\mathbb{C}^2$. First note that normalisation puts a restriction on the values a and b can take:

$$1 = \langle\psi|\psi\rangle = \overline{a}a + \overline{b}b = |a|^2 + |b|^2.$$

Since $|a|^2 + |b|^2 = 1$, we can draw the following triangle with 1 on the hypotenuse:
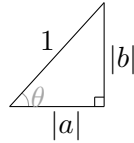
From the sine and cosine laws, we can always express $|a|$ and $|b|$ in terms of a single angle θ: $|a| = \cos\theta$ and $|b| = \sin\theta$. As a matter of convention, it's slightly more convenient to use $\frac{\theta}{2}$ instead of θ, so let $|a| = \cos\frac{\theta}{2}$ and $|b| = \sin\frac{\theta}{2}$.

As we saw in Section 2.1.1, the absolute value of a complex number $\lambda = re^{i\alpha}$ fixes r. Hence, we can conclude that $a = \cos\frac{\theta}{2}e^{i\beta}$ and $b = \sin\frac{\theta}{2}e^{i\gamma}$, for some phase angles β, γ. So, using just normalisation, we can replace 2

complex numbers with 3 angles:

$$|\psi\rangle = \cos\frac{\theta}{2}e^{i\beta}|0\rangle + \sin\frac{\theta}{2}e^{i\gamma}|1\rangle$$

Now we can use the freedom to choose a global phase to get rid of another one of these angles. Since we can freely multiply $|\psi\rangle$ by a global phase without changing the physical state, the angle β is actually redundant. We can cancel it out by multiplying the whole thing by $e^{-i\beta}$.

$$e^{-i\beta}\left(\cos\frac{\theta}{2}e^{i\beta}|0\rangle + \sin\frac{\theta}{2}e^{i\gamma}|1\rangle\right) = \cos\frac{\theta}{2}|0\rangle + \sin\frac{\theta}{2}e^{i(\gamma-\beta)}|1\rangle$$

Letting $\alpha := \gamma - \beta$, we can therefore write a generic qubit state $|\phi\rangle$ conveniently as:

$$|\phi\rangle := \cos\frac{\theta}{2}|0\rangle + \sin\frac{\theta}{2}e^{i\alpha}|1\rangle$$

Since the quantum state is now totally described by two angles, we can plot it on a sphere, called the **Bloch sphere**:
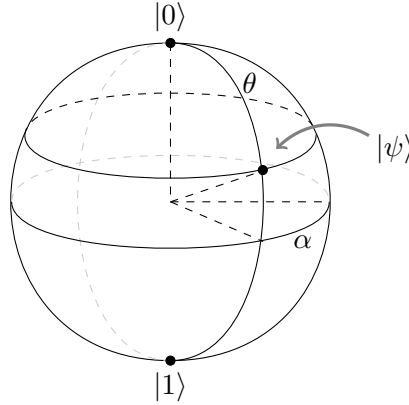
This picture is useful for our intuition. For example, the more 'similar' two states are, that is, the higher the value of their inner product, the closer they are on the Bloch sphere. In particular, antipodes are always orthogonal states.

Exercise 2.8. Show that, for the following antipodal states on the Bloch sphere:

$$|\phi_0\rangle = \cos\frac{\theta}{2}|0\rangle + \sin\frac{\theta}{2}e^{i\alpha}|1\rangle$$

$$|\phi_1\rangle = \cos\frac{\theta+\pi}{2}|0\rangle + \sin\frac{\theta+\pi}{2}e^{i\alpha}|1\rangle$$

we have $\langle\phi_0|\phi_1\rangle = 0$.

For a qubit, picking an orthonormal basis just comes down to picking a pair of orthogonal normalised states, hence, we can always think of orthonormal bases as different axes cutting through the Bloch sphere.

Exercise 2.9. The standard basis $|0\rangle$, $|1\rangle$ corresponds to the Z axis of the Bloch sphere. Show that the following states

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

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

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

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

correspond to the X and Y axes, respectively. That is, show they are located on the Bloch sphere as follows:
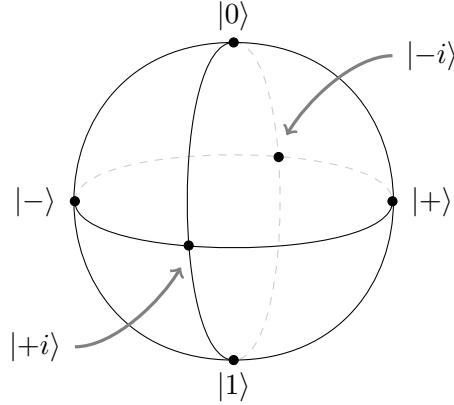
The three pairs of antipodal points each correspond to an orthonormal basis for $\mathbb{C}^2$:

$$X\text{-basis} := \{|+\rangle, |-\rangle\}$$
$$Y\text{-basis} := \{|+i\rangle, |-i\rangle\}$$
$$Z\text{-basis} := \{|0\rangle, |1\rangle\}$$

These bases mark out the X, Y, and Z axes of the Bloch sphere. We'll see in Section 2.3.2 that they are also eigenvectors of the Pauli X, Y, and Z operations, respectively.

Let's now take a look at what unitaries do to qubit quantum states visually. A paradigmatic example is the Z phase gate:

$$Z(\alpha) = |0\rangle\langle 0| + e^{i\alpha}|1\rangle\langle 1|$$

On eigenstates $|0\rangle$ and $|1\rangle$, the $Z(\alpha)$ doesn't do anything: or rather it only changes the state up to a global phase:

$$Z(\alpha)|0\rangle = |0\rangle \qquad Z(\alpha)|1\rangle = e^{i\alpha}|1\rangle \propto |1\rangle$$

However, on a generic state, we end up multiplying the coefficient of $|1\rangle$ by $e^{i\alpha}$, which amounts to adding α to its phase:

$$|\psi\rangle = \cos\tfrac{\theta}{2}|0\rangle + e^{i\beta}\sin\tfrac{\theta}{2}|1\rangle$$
$$\mapsto Z(\alpha)|\psi\rangle = \cos\tfrac{\theta}{2}|0\rangle + e^{i(\alpha+\beta)}\sin\tfrac{\theta}{2}|1\rangle$$

Since the phase of the coefficient of $|1\rangle$ determines how far around the Z-axis to plot the state, we can conclude that $Z(\alpha)$ amounts to a Z-rotation of the Bloch sphere by an angle of α.
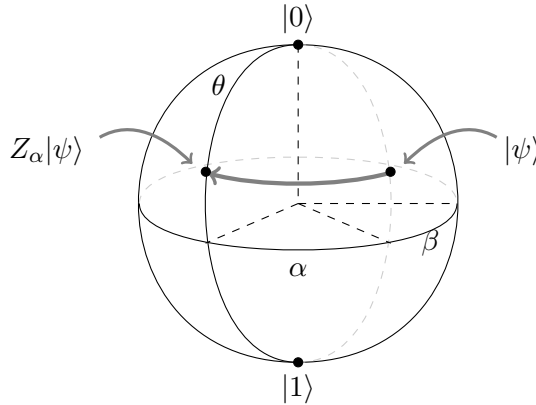
While we said that $Z(\alpha)$ is a paradigmatic example, it is in fact (essentially) the only example of a unitary over $\mathbb{C}^2$, up to a global phase. We can see that by taking some generic unitary and diagonalising it:

$$U = \lambda_0|\phi_0\rangle\langle\phi_0| + \lambda_1|\phi_1\rangle\langle\phi_1|$$

Then, as we saw in Exercise 2.4, the eigenvalues of a unitary are always phases, so $\lambda_j = e^{i\alpha_j}$, hence up to a global phase, we have:

$$U \propto |\phi_0\rangle\langle\phi_0| + e^{i\alpha}|\phi_1\rangle\langle\phi_1| \tag{2.12}$$

So, U is basically $Z(\alpha)$, written with respect to another ONB $\{|\phi_0\rangle, |\phi_1\rangle\}$.

Exercise* 2.10. We have already seen that ONBs correspond to axes on the Bloch sphere. Show that U as defined in (2.12) corresponds to a rotation about the axis defined by $\{|\phi_0\rangle, |\phi_1\rangle\}$ by an angle of α.

Hence, single-qubit unitaries correspond exactly to rotations of the Bloch sphere. A standard fact about rotations of a sphere is that any rotation can be generated by a sequence of three rotations about two orthogonal axes. The standard choice of orthogonal axes you see crop up in the quantum computing literature is a Z-axis rotation, which we've already met, and an X-axis rotation:

$$X(\alpha) = |+\rangle\langle+| + e^{i\alpha}|-\rangle\langle-|$$

Exercise* 2.11. Show that, up to a global phase, we can always write a qubit unitary as a composition of Z and X rotations as follows:

$$\boxed{U} \;\; = \;\; e^{i\theta} \cdot \boxed{Z(\alpha)} \boxed{X(\beta)} \boxed{Z(\gamma)}$$

This is called the **Euler decomposition** of U.

2.2.3   Unitary evolution

We won't talk much about the Schrödinger equation in this book, but no bluffers intro to quantum theory would be complete without showing it at least once. So here it is, the time-dependent **Schrödinger equation**:

$$i\frac{d}{dt}|\psi(t)\rangle = H(t)|\psi(t)\rangle \tag{2.13}$$

where $H(t)$ is a self-adjoint operator, called the *Hamiltonian*, which may depend on the time t (and yes physicists, we are ignoring the constants and units). The actual form this operator takes depends on what sorts of physics is actually going on. If we for instance are describing the internals of a molecule we would have a particular Hamiltonian describing the momentum, the spin, and attraction or repulsion between all the electrons, protons and neutrons. If instead we are describing a lab situation where we are shining a laser on trapped ion, we might only care about the terms having to do with the time-dependent laser interaction.

This differential equation describes how a quantum state evolves in time. When H doesn't depend on t, solutions to (2.13) take a nice form in terms of an initial state $|\psi(0)\rangle$ and the **matrix exponential** e^{itH}. We'll talk about matrix exponentials a lot more in Chapter 7, but for now, suffice to say that matrix exponentials behave a lot like plain ole number exponentials when you take their derivatives. In particular, for matrices we have $\frac{d}{dt}e^{tM} = Me^{tM}$,

so taking the derivative of $e^{-itH}|\psi(0)\rangle$ with respect to t just pops out a factor of $-iH$. Hence:

$$i\frac{d}{dt}e^{-itH}|\psi(0)\rangle = (i)(-iH)e^{itH}|\psi(0)\rangle = He^{-itH}|\psi(0)\rangle$$

So, $e^{-itH}|\psi(0)\rangle$ gives a solution to the Schrödinger equation, and we can let $|\psi(t)\rangle := e^{-itH}|\psi(0)\rangle$. Another handy thing is, whenever H is self-adjoint, e^{itH} is unitary (again we'll see why in Chapter 7). Hence, $U(t) := e^{-itH}$ gives us the unitary **time evolution operator** of a quantum system.

The reason we don't talk much about the Schrödinger equation in this book is, for the purposes of quantum computation, it usually suffices to talk about time evolution abstractly, fully in terms of unitaries. That is, if a system is initially in state $|\psi\rangle := |\psi(0)\rangle$, its time evolution for a fixed chunk of time (say $t = 1$) is described by some unitary map $U := U(1)$. Conversely, there is a powerful theorem that says *any* unitary map arises as the time evolution of some Hamiltonian in this kind of way. For this reason, it is often convenient to just talk directly about the unitaries and leave it to the physicists to worry about Hamiltonians.

2.2.4 Measurements and the Born rule

Most quantum computations consist of preparing a quantum state, doing some unitary evolution of the state, then measuring the result. The only ingredient we are missing so far is measurement. In order to do a quantum measurement, we need to choose *what* we want to measure. A **measurement** is a (non-deterministic) process which extracts some information from a quantum system and usually changes the state of the system when we do it. Mathematically, this is represented as a set of projectors that sum up to the identity:

$$\mathcal{M} = \{M_1, \ldots, M_k\} \qquad \sum_i M_i = I$$

The probability of getting the i-th outcome when we measure a state $|\psi\rangle$ with $\mathcal{M}$ is computed with the following M-sandwich:

$$\text{Prob}(i \,|\, \psi) = \langle\psi|M_i|\psi\rangle \tag{2.14}$$

This quantum sandwich is called the **Born rule**.

Note that $\sum_i M_i = I$ implies that the set of projectors in M_i are mutually orthogonal, i.e. $M_i M_j = 0$ whenever $i \neq j$. So, one can think of measuring roughly as checking whether a state $|\psi\rangle$ is "in" one of a collection of k orthogonal subspaces, given by the images of the projectors $M_1, \ldots, M_k$.

The reason for the scare-quotes around "in" is that a given state doesn't need to be totally inside the image of *any* of the projectors M_i. But if it *is* completely inside the i-th subspace, then $M_i|\psi\rangle = |\psi\rangle$ and:

$$\text{Prob}(i \,|\, \psi) = \langle\psi|M_i|\psi\rangle = \langle\psi|\psi\rangle = 1.$$

So in that case we get outcome i with certainty.

However, in general $|\psi\rangle$ could be a linear combination of vectors in more than one orthogonal subspace given by $\mathcal{M}$. But here's the funny thing: after we measure a system and get outcome i, the state of the system will *always* be entirely in the image of M_i. In other words, if it was only partially in the image of M_i before, it has to move!

Quantum theory tells us that, if we measure $|\psi\rangle$ with $\mathcal{M}$ and get outcome i, the resulting state is given by projecting $|\psi\rangle$ onto the image of M_i and renormalising, i.e.

$$|\psi\rangle \quad \xmapsto{\;project\;} \quad M_i|\psi\rangle \quad \xmapsto{\;renormalise\;} \quad \frac{1}{\sqrt{\text{Prob}(i \,|\, \psi)}} \, M_i|\psi\rangle$$

This rule for updating a state after a measurement is called **Lüder's rule**.

This idea that measuring something changes it is not so strange if you think about it. Even in the classical world, if we want to know something about a system, we need to interact with it somehow. For instance, if we want to know what colour a ball is, we can shine a light on it, and see which light reflects back. In the process, we barrage the ball with a bunch of photons which interact with the particles in the ball, energizing them and even sloughing off a tiny, tiny bit of paint. The only reason we tend to ignore such things in the classical world is that there are many ways to measure something which *essentially* don't change it. For example, you would be hard pressed to figure out if someone shined a light on that ball or not.

The story is different in quantum systems. A fundamental property of quantum theory says that the amount of information we can get out of a system is directly proportional to how much we disturb it. So, if we want *lots* of information, we need *lots* of disturbance. We can see this with the two extremes of measurement.

At one extreme we have **ONB measurements**. These are perhaps the most common kinds of measurements considered, whose components are the rank-1 projectors given by the ONB states. That is, for an ONB $\mathcal{B} = \{|\phi_i\rangle\}_i$, we have a measurement $\mathcal{M}_{\mathcal{B}} = \{|\phi_i\rangle\langle\phi_i|\}_i$. As we saw in Section 2.1.3, summing over these projectors gives a resolution of the identity:

$$\sum_i |\phi_i\rangle\langle\phi_i| = I$$

So this does indeed define a measurement. This kind of measurement gives us as much information as possible, since each of the subspaces corresponding to the measurement outcomes is one-dimensional. However, it also disturbs the system a great deal, since any state will get projected into one of these 1D subspaces after the measurement. That is, $|\psi\rangle$ will **collapse** to a basis state $|\phi_i\rangle$ after the measurement. For the qubit Z ONB measurement $\{|0\rangle\langle 0|, |1\rangle\langle 1|\}$, this can be visualised as sending a state the north or south pole of the Bloch sphere, depending the associated Born rule probabilities:
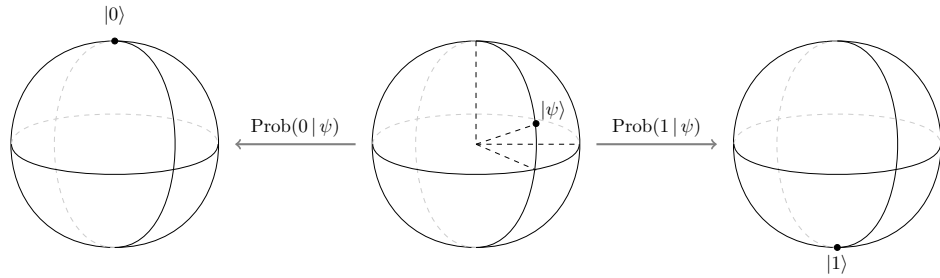
At the other extreme, we have the trivial measurement $\mathcal{I} = \{I\}$. This is indeed a collection of projectors which sum up to I, but we only have one outcome: I. Afterwards, the state is "updated" by $|\psi\rangle \mapsto I|\psi\rangle$. Since there's only one outcome, measuring $\mathcal{I}$ doesn't tell us anything, but at least it doesn't disturb anything either!

Between these two extremes, there are lots of examples which can all be seen as some sort of "coarse-graining" of an ONB measurement. Perhaps the most relevant example for us will be measuring just part of a state. For example, the measurement $\mathcal{M} = \{|\phi_i\rangle\langle\phi_i| \otimes I\}_i$ on a system $A \otimes B$ corresponds to performing the basis measurement $\{|\phi_i\rangle\langle\phi_i|\}_i$ on just the subsystem A.

Exercise 2.12. Define the following 3 measurements on a pair of qubits:

$$\mathcal{A} = \left\{ \; \text{} \; \right\}_i$$

$$\mathcal{B} = \left\{ \; \text{} \; \right\}_j$$

$$\mathcal{C} = \left\{ \; \text{} \; \right\}_{ij}$$

Show that the probability of getting outcome i for $\mathcal{A}$ then getting outcome j when measuring $\mathcal{B}$ on the resulting state is the same as the probability of getting outcome (i, j) for $\mathcal{C}$.

We will often be interested in computing the post-measurement state after measuring just some of the qubits of a quantum state. For example, this plays an essential role in state injection and quantum teleportation in Section 3.3 and is the key ingredient in measurement-based quantum computing, introduced in Chapter 8.

For example, consider measuring the first qubit of a 2-qubit quantum state $|\Psi\rangle$ in the computational basis, i.e. performing measurment $\mathcal{A}$ from Exercise 2.12. If we get outcome 0, the post-measurment state will be the following, up to renormalisation:
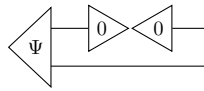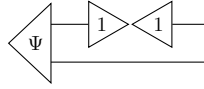

whereas if we get outcome 1, the state will be:

In either case, the state of the second qubit could change, depending on which measurement was performed and the measurement outcome. This sometimes referred to as the **back-action** of a quantum measurement.

It is worth noting that after a basis measurement is performed, the measured qubits are no longer entangled with the rest of the state, so we will

sometimes simply ignore them:

$$(2.15)$$

Depending on how we implement a measurement, we may have no longer have access to the system after we measure it anyway. For example, if a photon hits a detector and makes a click, then for all intents and purposes, that photon is gone. This kind of measurement is sometimes called a **demolition measurement**. We can compute the state of the remaining systems after a demolition ONB measurement simply by applying the basis effect to the measurement qubit (and renormalising), e.g. sending $|\Psi\rangle$ to $(\langle i| \otimes I)|\Psi\rangle$ as in (2.15).

There are also ways to measure a system without physically destroying it. For example, we may apply a unitary interaction between the system of interest and an ancillary system, and measure the latter. Measurements which leave the measured system in place (albeit changed via a projector) are called **non-demolition measurements**. This kind of measurement will be especially important for quantum error correction, which we study in Chapter 11.

We conclude this section with an alternative way for specifying a quantum measurement. Often in quantum theory and quantum computing literature, measurements are defined in terms of self-adjoint maps called **observables**. However, an observable just amounts to packing a bunch of projectors up together into one map. That is, if we fix distinct real numbers $r_1, \ldots, r_k$, we can define a self-adjoint map M from a measurement $\mathcal{M} = \{M_i\}_i$ as follows:

$$M = \sum_{i=1}^{k} r_i M_i \tag{2.16}$$

Conversely, by diagonalising a self-adjoint map M and gathering up terms in the sum with the same eigenvalues, we can see that any self-adjoint map can be written in the form of (2.16) for same unique set of projectors M_i. That is,

$$M = \sum_{j=1}^{d} \lambda_j |\phi_j\rangle\langle\phi_j| = \sum_{i=1}^{k} r_i M_i$$

where M_i is the sum over all the projectors $|\phi_k\rangle\langle\phi_k|$ associated with a (possibly repeated) eigenvalue $\lambda_k = r_i$.

2.3   Gates and circuits

Now that we know about states, unitaries, and measurements, we have all
the basic ingredients we need to start talking about computation in the
quantum circuit model. In this model, computation proceeds in three
steps:

1. Prepare N qubits in a fixed state.
2. Apply a **quantum circuit** to the input state.
3. Measure one or more of the qubits.
4. (Sometimes) perform some classical post-processing on the measurement
   outcome.

A quantum circuit describes a large unitary map in terms of many smaller
unitaries (typically drawn from a fixed set) called **gates**, combined via
composition and tensor product. This is essentially the "code" of a quantum
algorithm. After performing all the gates, we measure some of the qubits.
It is these measurement outcomes that is the actual output of the quantum
circuit. Generally, we will have to run a quantum circuit many times in
order to gather more measurement outcomes and estimate the probability
of getting a certain outcome.

The quantum circuit model abstracts away all the physical details of
how the computation is actually performed. In practice a qubit might be
represented by two chosen energy states of an ion trapped in a magnetic field,
or as the different polarisations of a photon, or in terms of any other physical
system that has two different quantum states that we have arbitrarily chosen
to label as the states $|0\rangle$ or $|1\rangle$. Each individual gate will then physically
correspond to some type of interaction with this chosen model of a qubit. It
might mean we have to shine a laser on some ions for 10 nanoseconds, or let
some photons meet and entangle using a beam splitter. It doesn't matter, as
long as the physical operation corresponds to applying (or approximating)
that unitary operation on the quantum state.

Quantum circuits enable us to construct large and complex unitaries from
known, relatively simple ones. The most interesting sets of quantum gates
are **universal** ones, i.e. sets of gates that allow us to construct any unitary,
or at least approximate it to high precision. We will spend a large part of this
book thinking about circuits and how we can construct interesting behaviour
from simple components.

2.3.1 Classical computation with quantum gates

Quantum computation is a strict generalisation of classical computation. This means any computation involving bits on a classical computer can be lifted to a quantum computation over qubits. We can see this by first noting that any reversible function (a.k.a. bijection) on N bits $\phi : \mathbb{B}^N \to \mathbb{B}^N$ lifts to a unitary permutation over basis vectors:

$$U_\phi |\vec{b}\rangle := |\phi(\vec{b})\rangle \qquad \forall \vec{b} \in \mathbb{B}^N$$

The simplest non-trivial such gate is the NOT gate:

$$\text{NOT}|0\rangle = |1\rangle \qquad \text{NOT}|1\rangle = |0\rangle$$

More interesting is the controlled-NOT, or **CNOT gate**:

$$\text{CNOT} :: \begin{cases} |00\rangle \mapsto |00\rangle \\ |01\rangle \mapsto |01\rangle \\ |10\rangle \mapsto |11\rangle \\ |11\rangle \mapsto |10\rangle \end{cases}$$

One way to think of this gate is that the first bit is controlling whether the second bit gets a NOT applied to it. Constructing the 'controlled' variant of a quantum gate is a common theme in quantum circuits.

Another way to think about the CNOT gate is as an exclusive-OR (XOR) gate, where we've also kept a copy of one of the inputs to keep things reversible. We write XOR (a.k.a. addition modulo 2) as $\oplus$. Using this notation, we have:

$$\text{CNOT}|x, y\rangle = |x, x \oplus y\rangle$$

Another useful classical gate is the controlled-controlled-NOT gate, which is also called the **Toffoli gate**. It is defined as follows:

$$\text{TOF}|x, y, z\rangle = |x, y, (xy) \oplus z\rangle$$

where xy is the product (a.k.a. the AND) of the two bits x and y. This gate flips the third bit if and only if the first two bits are 1.

The NOT, CNOT, and Toffoli gates are all in a family of N-controlled NOT gates, i.e. NOT is a 0-controlled NOT gate, CNOT is 1-controlled, and Toffoli is 2-controlled. We draw them in a similar way, with dots on each of the N **control qubits**, connecting to an $\oplus$ on the **target qubit**, i.e. the

one receiving the NOT:

$$\text{NOT} \;=\; \text{—}\oplus\text{—} \qquad\qquad \text{CNOT} \;=\; \text{[diagram]} \qquad\qquad \text{TOF} \;=\; \text{[diagram]}$$

Note that plugging a $|1\rangle$ into any of the control qubits gives a smaller N-controlled NOT gate on the remaining qubits:

$$\text{[diagram]} \;=\; \text{[diagram]} \qquad\qquad \text{[diagram]} \;=\; \text{[diagram]}$$

whereas plugging a $|0\rangle$ into any of the control qubits leaves an identity map on the remaining qubits.

We can think of the Toffoli gate as a reversible version of the AND gate, where again we keep copies of the inputs and 'store' the output of the AND gate by XOR'ing it with the third input qubit. In particular, if we provide it with a 'fresh' qubit in the $|0\rangle$ state for the third input, we have:

$$\text{TOF}|x, y, 0\rangle = |x, y, xy\rangle$$

This is in fact a special case of a standard technique, sometimes called the **Bennett trick**, for turning any classical function $f : \mathbb{B}^N \to \mathbb{B}^M$ into a unitary:

$$U_f :: |\vec{x}, \vec{y}\rangle \mapsto |\vec{x}, f(\vec{x}) \oplus \vec{y}\rangle$$

where $\vec{x} \in \mathbb{B}^N, \vec{y} \in \mathbb{B}^M$ and $\oplus$ is taking the XOR of the two M-bit strings elemement-wise. Even if f is not itself a bijection, the mapping $(\vec{x}, \vec{y}) \mapsto (\vec{x}, f(\vec{x}) \oplus \vec{y})$ is bijective, hence U_f is a permutation of basis states.

We call this a **quantum oracle** for the function f. This construction plays a central role in many quantum algorithms, since it allows us to query a classical function f using a quantum state. If this is a basis state of the form $|\vec{x}, 0\rangle$, this essentially amounts to evaluating the function and storing its output in the last qubit:

$$U_f|\vec{x}, 0\rangle = |\vec{x}, f(\vec{x})\rangle$$

However, choosing other input states allows us to query f in non-classical ways, which if we are clever, can give us some extra *quantum oomph*.

2.3.2 Pauli and phase gates

The Pauli gates are single-qubit unitaries which will play a major role throughout this book, especially in Chapters 6, 7 and 11. We've seen one of

the Pauli's in the previous section already: the NOT gate, a.k.a. the Pauli X gate. It is called such because it represents a $180°$ rotation around the X-axis on the Bloch sphere:

$$\mathrm{NOT} = X = X(\pi)$$
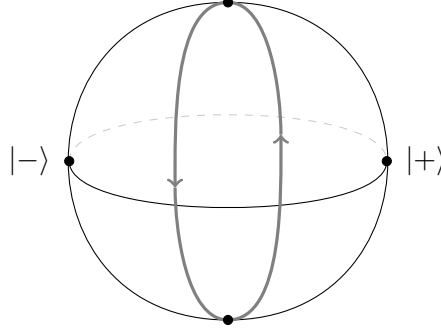
There are two other canonical choices for 'quantum' NOT gates, corresponding to $180°$ rotations around the Z and Y axes, respectively. Together, these three maps form the single-qubit **Pauli matrices**:

$$X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y := \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad (2.17)$$

All three of these matrices are unitary and self-adjoint (hence equal to their own inverse), and multiplying any two of them gives the third one, up to a factor of $\pm i$. As a result, they generate a finite group called the (single-qubit) **Pauli group** $\mathcal{P}_1$. We'll have a lot more to say about Pauli groups in Chapter 6.

As we already saw in Section 2.2.2, we can produce unitary rotations about an axis of the Bloch sphere given by an ONB $\{|\phi_0\rangle, |\phi_1\rangle\}$ as:

$$R_{\mathcal{B}}(\alpha) = |\phi_0\rangle\langle\phi_0| + e^{i\alpha}|\phi_1\rangle\langle\phi_1|$$

Applying this to the X, Y, and Z basis, we obtain the **X-, Y-, and Z-phase gates** as follows.

$$\begin{aligned} X(\alpha) &:= |+\rangle\langle+| + e^{i\alpha}|-\rangle\langle-| \\ Y(\alpha) &:= |+i\rangle\langle+i| + e^{i\alpha}|-i\rangle\langle-i| \\ Z(\alpha) &:= |0\rangle\langle0| + e^{i\alpha}|1\rangle\langle1| \end{aligned} \qquad (2.18)$$

We also noted in Section 2.2.3 that the $X(\alpha)$ and $Z(\alpha)$ gates can generate any single-qubit unitary, up to a global phase. Indeed we have:

$$-\boxed{Y(\alpha)}- \;\propto\; -\boxed{Z\left(-\tfrac{\pi}{2}\right)}-\boxed{X(\alpha)}-\boxed{Z\left(\tfrac{\pi}{2}\right)}- \;\propto\; -\boxed{Z\left(\tfrac{\pi}{2}\right)}-\boxed{X(-\alpha)}-\boxed{Z\left(-\tfrac{\pi}{2}\right)}-$$

$$(2.19)$$

Remark 2.3.1. We write $\propto$ to mean equal up to a non-zero scalar factor. It is important that we say *non-zero*, because $0 \cdot M = 0 \cdot N$ for any M, N, so if we allowed 0 the statement becomes vacuous. In the case of equation (2.19) above, this scalar is a global phase, which is always non-zero since $|e^{i\alpha}| = 1$ for all α.

From (2.18), there is an evident matrix presentation for Z-phase gates:

$$Z(\alpha) \;=\; \begin{pmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{pmatrix}$$

We could also compute matrix presentations for X- and Y-phase gates from (2.18), yielding matrices containing sums of $1, i$, and $e^{i\alpha}$. However, it is often more convenient when working concretely with the matrices to pass to an equivalent (up to global phase) form using sine and cosine functions:

$$X(\alpha) \;\propto\; \begin{pmatrix} \cos\frac{\alpha}{2} & -i\sin\frac{\alpha}{2} \\ -i\sin\frac{\alpha}{2} & \cos\frac{\alpha}{2} \end{pmatrix} \qquad Y(\alpha) \;\propto\; \begin{pmatrix} \cos\frac{\alpha}{2} & -\sin\frac{\alpha}{2} \\ \sin\frac{\alpha}{2} & \cos\frac{\alpha}{2} \end{pmatrix}$$

There are two Z-phase gates that crop up so often in quantum computing that they have gained special names. These are the **S gate** $S := Z(\frac{\pi}{2})$ and the **T gate** $T := Z(\frac{\pi}{4})$. Note that we have $T^2 = S$ and $S^2 = Z$. These names 'S' and 'T' don't really mean anything (apart from that 'T' follows 'S' in the alphabet), they are just names that people happened to decide on. In some (especially older) quantum computing papers you might see people call the S gate the P gate, where in this case 'P' stands for 'Phase'. Some people also refer to the T gate as the $\frac{\pi}{8}$-gate. This is because:

$$T \;\propto\; \begin{pmatrix} e^{-i\frac{\pi}{8}} & 0 \\ 0 & e^{i\frac{\pi}{8}} \end{pmatrix}$$

2.3.3   Hadamard gates

The **Hadamard**, or H gate is a gate that sends the Z basis to the X basis, and vice-versa:

$$H \;=\; |{+}\rangle\langle 0| + |{-}\rangle\langle 1| \;=\; |0\rangle\langle{+}| + |1\rangle\langle{-}| \;=\; \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

It can be visualised, like all self-inverse unitaries on qubits, as a 180° rotation of the Bloch sphere. This time, it is through the diagonal axis that
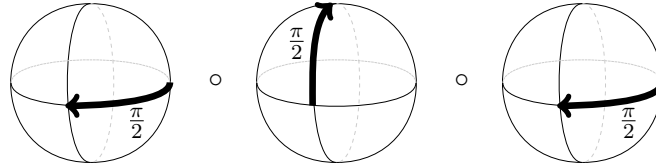
lies halfway between the X-axis and the Z-axis:

$$(2.20)$$

One way to see this is by computing the Euler decomposition of H:

As rotations of the Bloch sphere, this gives:

While it might not be immediately obvious from staring at the pictures above that this indeed gives the rotation depicted in (2.20), you can probably convince yourself this is true if you try it on a real world object. If you are using a coffee cup or a beer can, we suggest draining it first.

We can write the effect of a Hadamard gate on computational basis elements compactly as follows:

$$H :: |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y \in \{0,1\}} (-1)^{xy} |y\rangle \tag{2.21}$$

This behaviour of Hadamard gates, where it introduces a phase of $-1 = e^{i\pi}$ whenever the product of two boolean variables (in this case x and y) is 1, will play a special role in Chapter 7, when we study the **path sum** representation of quantum circuits.

We can generalise equation (2.21) to the action of n H-gates applied in parallel to an n-qubit computational basis element as follows:

$$H^{\otimes n} :: |\vec{x}\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{x_1 y_1 + \ldots + x_n y_n} |\vec{y}\rangle = \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{\vec{x} \cdot \vec{y}} |\vec{y}\rangle \tag{2.22}$$

where $\vec{x} \cdot \vec{y}$ is the dot product of the two vectors $\vec{x}$ and $\vec{y}$, taken modulo 2. This works because, for an integer k, the value of $(-1)^k$ only depends on where k is even or odd. This is sometimes called the **Hadamard transform** of a vector, which is a type of discrete Fourier transform. We will discuss different types of Fourier transforms, particularly in the context of boolean logic, in Chapter 9.

2.3.4   Controlled unitaries

Controlled unitaries are unitaries where the first qubit acts as a **control**, dictating whether the unitary U gets applied to the remaining qubits.

$$\text{CTRL-}U \;::\; \begin{cases} |0\rangle \otimes |\psi\rangle \mapsto |0\rangle \otimes |\psi\rangle \\ |1\rangle \otimes |\psi\rangle \mapsto |1\rangle \otimes U|\psi\rangle \end{cases}$$

This can be seen as a sort of "quantum IF statement":

```
IF q[0] THEN APPLY U TO (q[2] .. q[n-1])
ELSE DO NOTHING
```

Exercise 2.13. The CNOT gate is also often called the CX gate, because it applies an X gate to the target if the control is $|1\rangle$. We similarly also have a CZ gate that applies a Z instead. We can construct this using a CNOT and Hadamard gates as follows:
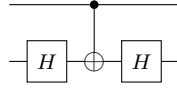
Find the matrix of the CZ by directly calculating the matrices involved in the circuit above.

2.3.5   (Approximate) universality

A set $\mathcal{G}$ of gates is **exactly universal** if the gates in $\mathcal{G}$ can be combined to construct any N-qubit unitary U. A typical example of an exactly universal set of unitaries is $\mathcal{G} = \{\text{CNOT}, H\} \cup \{Z(\alpha) \mid \alpha \in [0, 2\pi)\}$. Using just H and $Z(\alpha)$, we can produce $X(\alpha) = HZ(\alpha)H$. Hence, any single-qubit unitary can be constructed via the Euler decomposition. What is trickier to show is that, by combining single-qubit unitaries and CNOT, we get can any N-qubit unitary. We will leave this as an exercise to the reader. :)

Since there are uncountably many N-qubit unitaries, exactly universal sets of gates are necessarily infinite. As a result, it is often more convenient to consider **approximately universal** (or simply **universal**) sets of gates, which are able to approximate any N-qubit unitary up to arbitrarily high precision. Put a bit more precisely:

Definition 2.3.2. For some $\varepsilon > 0$, a unitary U' ε-approximates U if for all normalised states $|\psi\rangle$, $\|U|\psi\rangle - U'|\psi\rangle\| \leq \varepsilon$.

The more mathematically-inclined literature may say in this case that U and U' are ε-close in the **operator norm**. Note that it is important we take

only the normalised states here (or at least states with some bounded norm). Otherwise, two unitaries will never be ε-close unless they are equal, because we can always just pick some huge number λ where $\|U(\lambda|\psi\rangle) - U'(\lambda|\psi\rangle)\| = |\lambda| \cdot \|U|\psi\rangle - U'|\psi\rangle\| > \varepsilon$.

Definition 2.3.3. A set of gates $\mathcal{G}$ is **approximately universal** if for any unitary U and $\varepsilon > 0$, we can construct a unitary U' using just gates from $\mathcal{G}$ that ε-approximates U.

Unlike exactly universal sets of gates, approximately universal sets can be finite. Perhaps the most commonly considered approximately universal set of gates is the **Clifford+T** set:

$$\text{Clifford+T} := \{\text{CNOT}, H, T := Z\left(\frac{\pi}{4}\right)\}$$

This is called the Clifford+T set, because it is often regarded as adding the T gate to this set:

$$\text{Clifford} := \{\text{CNOT}, H, S := Z\left(\frac{\pi}{2}\right)\}$$

Note that the Clifford circuits are a subset of the Clifford+T circuits, since $S = T^2$.

Clifford+T circuits are approximately universal. While we won't go into the details here (they are covered in many standard quantum computing textbooks), we can sketch a fairly simple argument. First, note that if we can rotate around any axis of the Bloch sphere by an irrational multiple of π that we can approximate *any* rotation around that axis: we just need to keep going around and around until we land somewhere close. Then, defining $R = HTH$, we can observe that TR is an irrational rotation around *some* axis of the Bloch sphere and RT is an irrational rotation around some different axis. Putting these together, we can approximate generic rotations around two distinct axes, so that we can approximate any rotation. Then, just like in the exactly universal case, as soon as CNOT gets involved, we can boost up generic single-qubit unitaries to generic many-qubit unitaries.

The way we defined approximate universality, we are only asking that the gate set *can* approximate any unitary, we are not asking that it can do so with small overhead, or that we can *find* the approximation efficiently. Luckily, we get those features for free. The **Solovay-Kitaev theorem** states that if we have any finite set of single-qubit unitaries that can approximate any single-qubit unitary arbitrarily well, then it can also do so efficiently (to be precise: with a poly-logarithmic overhead in the precision $1/\varepsilon$). The algorithm is constructive, so it also tells you how to find this approximation.

In Chapter 10 we will describe a concrete approximation algorithm using the Clifford+T gate set, which works even better than just using the Solovay-Kitaev algorithm.

A consequence of the Solovay-Kitaev algorithm is that any finite approximately universal gate set is essentially equivalent to any other. If we require N gates for a particular circuit in one gate set, then we require $O(N \log^c(N/\varepsilon))$ gates in the other gate set if we want to approximate it up to an error ε. Here c is a particular (small) constant that depends on the precise gate set being used.

What all of this means is that we are essentially free to choose whatever universal gate set we want, and it won't change which unitaries we can efficiently approximate.

Exercise 2.14. In this exercise we will see that the eigenvectors of the Hadamard can be represented by a Clifford+T circuit. Let $|H\rangle := |0\rangle + (\sqrt{2} - 1)|1\rangle$.

a) Show that $|H\rangle$ is an eigenvector of the Hadamard gate. What is its eigenvalue?
b) Give an eigenvector of the Hadamard with a different eigenvalue.
c) We will now work towards showing that $|H\rangle$ can be constructed (up to non-zero scalar) using a $|+\rangle$ state and Clifford+T gates. First, note that $\tan \frac{\pi}{8} = \sqrt{2} - 1$, and then write $|H'\rangle = 2\cos \frac{\pi}{8}|H\rangle$ as $a|0\rangle + b|1\rangle$ with a and b written in terms of $\pm e^{\pm i\frac{\pi}{8}}$ factors (you will need the result from Exercise 2.2).
d) Now write $e^{-i\frac{\pi}{8}}HS|H'\rangle$ as a superposition of $|0\rangle$ and $|1\rangle$.
e) Give a sequence of Clifford+T gates $G_1, \ldots, G_k$ such that $G_k \cdots G_1|+\rangle \propto |H\rangle$.

2.3.6 Quantum circuit notation

A certain set of conventions for representing quantum circuits has developed throughout the years that is affectionately called **quantum circuit notation**. We have already seen many aspects of this in this section: qubits are represented by lines going from left to right, with multiple qubit lines being drawn from top to bottom. Unitary gates are represented by boxes on these qubits, usually with certain standardised names or symbols:
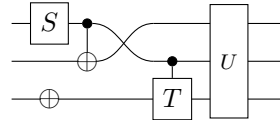
The NOT gate (the Pauli X) has one of these special symbols: $\oplus$. Adding a control-wire to a unitary is represented by a black dot connected to that unitary (a unitary controlled on the control being $|0\rangle$ instead of $|1\rangle$ is often depicted with a white dot instead of a black dot, though we will not need that in this book). A multi-qubit unitary, like U here, is a box connected to multiple input and output wires. A SWAP gate is depicted as literally just a swapping of the qubit wires.

Quantum circuit notation also allows for representing non-unitary components like state preparations, measurements and classically controlled operations. Consider for instance the following implementation of *magic state injection* (see Section 3.3.1):
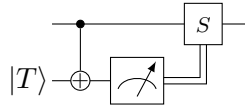
Here the second qubit has a $|T\rangle$ in front to denote that this qubit is prepared in the specific $|T\rangle := T|+\rangle$ state. The box with the arrow that looks a bit like a meter on a dial represents a demolition measurement in the computational basis that returns a classical value of 0 or 1 depending on if the measurement outcome was $|0\rangle$ or $|1\rangle$. The doubled-up wire coming out of it on the right represents a *classical* state, the outcome of the measurement. This classical wire flows into the S gate, meaning this S gate is controlled on this classical outcome: we only apply the S gate it the measurement outcome was 1.

Quantum circuit notation has been very useful in representing operations one would want to run on a quantum computer. It is however not ideal when trying to prove properties about computations, we will see a much more versatile way to do that in the next chapter.

2.4 A dash of quantum algorithms

It might be a bit strange to spend this long talking about quantum circuits and computations without saying much about what those computations are actually being used for. Indeed we imagine all of the circuits we are working with are part of a quantum algorithm. We have deliberately chosen not to talk too much about algorithms in this book because we wanted to focus on many other important aspects of quantum software like compiling, verification/classical simulation, and error correction. However, in the interest of being somewhat self-contained, we will give a brief taster of what a typical "oracle-style" quantum algorithm looks like.

While it cannot be said for *every* quantum algorithm out there, many

algorithms (including Shor's famous factoring/period finding algorithm) fit a very simple template. We start with a classical function f that is easy to describe (e.g. as a small boolean formula or program) and we want to know some kind of global property P about f. Then, the quantum algorithm proceeds as follows:

1. Prepare a superposition of computational basis states.
2. Apply the **quantum oracle** U_f to that state.
3. Measure some or all of the qubits (usually in a non-standard basis).
4. Perform some classical post-processing on measurement outcome(s) to compute P.

The quantum oracle U_f is the unitary map derived from the classical function f using the Bennett trick we explained in Section 2.3.1:

$$U_f|\vec{x}, \vec{y}\rangle = |\vec{x}, \vec{y} \oplus f(\vec{x})\rangle$$

By *global* property, we mean something that would classically require evaluating f on (much) more than one input. The prototypical example is the boolean satisfiability problem SAT, where we want to know if there exists any bit string $\vec{x}$ such that $f(\vec{x}) = 1$. Naïvely, we might need to try lots and lots of bit strings before we found one where $f(\vec{x}) = 1$.

For various reasons, the prospects of solving SAT on a quantum computer are not looking too good (see the discussion on **NP** vs **BQP** in Section 2.5.3), but we can still look to uncover other global properties about some function f. The most exciting such algorithms extract a global property from f exponentially faster than any known classical algorithm. However, these are relatively elaborate to explain in detail, so for the sake of an example, we will settle for something that is simply faster.

The **Bernstein-Vazirani problem** takes as input a function f from n bits to a single bit. This function has a secret bit string $\vec{s}$ "hiding" inside, in the sense that $f(\vec{x}) = \vec{s} \cdot \vec{x}$, where $\vec{s} \cdot \vec{x}$ is the dot product (modulo two) of the vectors $\vec{s}$ and $\vec{x}$. Our task is simply to find $\vec{s}$.

This is a global property of f because there is no way to figure out $\vec{s}$ from querying f on any single input. In fact, it is possible to show that, in order to recover $\vec{s}$ classically, we will need to evaluate f on precisely n different inputs. The easiest choice is just to take all of the bit strings $\vec{e_i}$ that have a 1 in the i-th place and 0's everywhere else. Then $f(\vec{e_i}) = \vec{e_i} \cdot \vec{s} = s_i$, the i-th component of $\vec{s}$.

However, *quantumly*, we can do this with just a single evaluation of the quantum oracle U_f. The quantum solution to this problem is called the

Bernstein-Vazirani algorithm. To solve this problem, we simply perform the following circuit on $n + 1$ qubits:


$$(2.23)$$
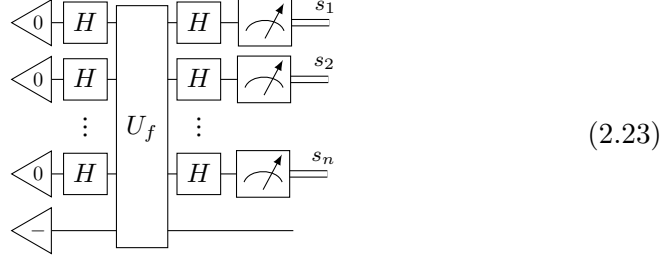
Then, with probability 1 (assuming no noise), the measurement outcome will be precisely the n-bit string $\vec{s}$.

We can see that this works doing a concrete calculation involving bras and kets. The details of this calculation are not too important, since we'll see a much nicer way to do this same derivation in the next chapter. However, this is typical of the kinds of calculations one meets when showing the correctness of a quantum algorithm.

First, we can simplify the presentation of the algorithm a little bit. Rather than using a full Bennett-style oracle U_f on $n + 1$ qubits, we can use the **reduced oracle** V_f.

Exercise 2.15. Show that $U_f(I \otimes |-\rangle) = V_f \otimes |-\rangle$, where $V_f|\vec{x}\rangle := (-1)^{f(\vec{x})}|\vec{x}\rangle$.

This lets the $|-\rangle$ "fall through" U_f in (2.23):
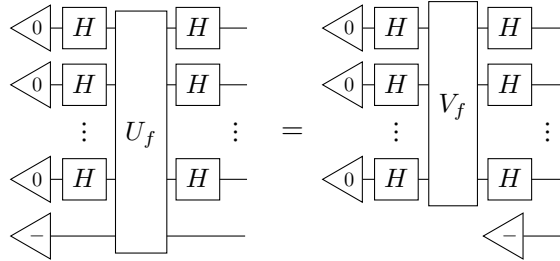
Since the last qubit is not entangled with the first n qubits, we can simply compute the state of the first n qubits to predict the measurement outcome.

Recall from Section 2.3.3 that $H^{\otimes n}$ acts as follows:

$$H^{\otimes n} :: |\vec{x}\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{\vec{x} \cdot \vec{y}} |\vec{y}\rangle$$

Since H is its own inverse, we also have

$$H^{\otimes n} :: \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{\vec{x} \cdot \vec{y}} |\vec{y}\rangle \mapsto |\vec{x}\rangle$$

This is enough to compute the state of the first n qubits. They start in the initial state of $|\vec{0}\rangle = |0\rangle^{\otimes n}$ and evolve through the unitaries as follows:

$$
|\vec{0}\rangle \xmapsto{H^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{\vec{0}\cdot\vec{y}} |\vec{y}\rangle = \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} |\vec{y}\rangle
$$
$$
\xmapsto{V_f} \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{f(\vec{y})} |\vec{y}\rangle = \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \{0,1\}^n} (-1)^{\vec{s}\cdot\vec{y}} |\vec{y}\rangle
$$
$$
\xmapsto{H^{\otimes n}} |\vec{s}\rangle
$$

Since the first n qubits end up in the computational basis state $|\vec{s}\rangle$, if we measure in the computational basis, we'll get outcome $\vec{s}$ with certainty.

Hence, in a single quantum query, we obtain the hidden bit string $\vec{s}$. As long as we are forced to treat f as a black box (i.e. the only thing we can do classically is query values $f(\vec{x})$), this is polynomially better than any classical algorithm. However, there is a variation of this problem, called **Simon's problem** which is in fact exponentially better in terms of query complexity.

In Simon's problem, we are given a function f from n-bit strings to n-bit strings. It also hides a secret bit string $\vec{s}$, but in a more subtle way. We promise that $f(\vec{x}) = f(\vec{y})$ if and only if $\vec{y} = \vec{x} \oplus \vec{s}$ for some fixed bit string $\vec{s}$.

In order to find $\vec{s}$ classically, we need to find a *collision*, i.e. a pair of distinct $\vec{x}$ and $\vec{y}$ such that $f(\vec{x}) = f(\vec{y})$, or prove that no such collision exists. Classically, the best strategy is just querying f at random, as we will expect to find a collision after about $\sqrt{2^n}$ queries. Quantumly, we can solve this using a circuit that looks very similar to the one before, but with U_f now having $2n$ qubits, instead of $n + 1$, since f outputs n qubits:
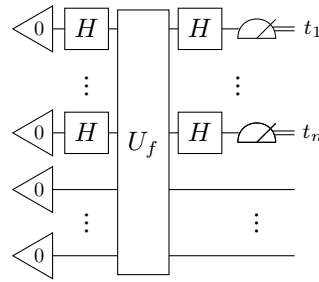
With a bit of work, we can show that the measurement outcome $\vec{t}$ that pops out isn't $\vec{s}$ itself, but it satisfies the property that $\vec{s} \cdot \vec{t} = 0$. Furthermore, we are equally likely to get *any* bit string $\vec{t}$ satisfying this property, so if we run this quantum part over and over again, pretty soon we get a system of n

independent linear equations involving $\vec{s}$, so we can solve for $\vec{s}$. (Note, we'll talk a lot more about doing linear algebra with bits in Chapter 4.)

Remark* 2.4.1. While we won't give the derivation for Simon's problem here, it is part of a whole family of problems called **hidden subgroup problems** which have an efficient quantum solution. You can find a diagrammatic derivation in *Picturing Quantum Processes*, Chapter 12.

As mentioned before, many algorithms follow the broad outline we gave at the beginning of this section. A variation on this theme is the "Grover-like" family of algorithms, which include Grover's quantum search algorithm as a prototypical example, where now U_f is called many times, with a thin layer of quantum gates between each iteration.

2.5   A dash of complexity theory

In Section 2.3.5 above we wrote $O(N \log^c(N/\varepsilon))$ to denote how fast a function would grow. Expressions like these and some other concepts from complexity theory will crop up here and there throughout the book, so we will give a quick primer in this section. We will keep this mostly informal, as we won't be needing any really formal complexity-theoretic definitions in this book.

2.5.1   Asymptotic growth

Let's give a formal definition of what it means for something to be bounded by some polynomial.

Definition 2.5.1. We say a function $f : \mathbb{N} \to \mathbb{N}$ is of **order $O(n^k)$ for some $k \in \mathbb{N}$ when there is some constant $c \in \mathbb{N}$ such that $f(n) \leq cn^k$ for all $n \in \mathbb{N}$.**

The notation $O(n^k)$ is known as **big O notation**. When a function is $O(n^k)$ it means that it grows *at most* as quickly as the function n^k. Note that an equivalent definition is that the quotient $f(n)/n^k$ stays bounded as n increases. For a function, its order only depends on the factors in the function that grow the quickest. For instance, if we had $f(n) = 3n^5 + 400n^2$, then f is $O(n^5)$, as for large values of n, the n^5 term contributes more and more to the value of f. Note that if a function is $O(n^k)$ that it is also $O(n^{k'})$ for any $k' > k$.

Exercise 2.16. Prove that the function $f(n) = 3n^5 + 400n^2$ is order $O(n^5)$ but not $O(n^4)$.

The reason we care about the order of functions is that it allows us to say which functions end up growing the quickest in 'the asymptotic limit', where the input to the function is (very) large. Suppose for instance that f and g are two functions that measure the runtime of two algorithms for solving the same problem and that g is $O(n^2)$ while f is $O(n^3)$ (but not $O(n^2)$). Then we know that there will be some (possibly very large) N such that for any $n > N$ we have $f(n) > g(n)$, and furthermore, that as we increase n that the value of f will speed away from that of g. So because the order of g is lower than that of f we know that *eventually* it will be better to run the algorithm corresponding to g. But note that we could have for instance $f(n) = n^3$ while $g(n) = 1000n^2$ so that the switchover would only come for $n > 1000$.

In the definition above we only talked about $O(n^k)$, but we could really replace n^k with any monotonically increasing function. For instance, we could say a function f is order $O(2^n/\log(n))$ meaning that f is bounded by some constant multiple of the function $2^n/\log(n)$. Additionally, we could consider a function of multiple variables, like $f(n, k)$, and say for instance that it is $O(k2^n)$, meaning that f grows slowly when you increase k, but really fast when you increase n.

In this book we will often say something is 'efficient' to calculate. What we mean by this is that if we let $f(n)$ be an upper bound on the cost of calculating an instance of the problem of size n, that this f is order $O(n^k)$ for *some k*. In words: there is some polynomial that upper-bounds f. What do we mean here by 'size'? This depends on the thing we want to calculate! Generally, there will be some set of size parameters that determine how much data we need to specify the problem instance that we want to calculate something about. For instance, if we wished to calculate something about a quantum circuit, then the relevant size parameters might be the number of qubits and the number of gates in the circuit. When we say something is efficient to calculate, it should be taken to mean 'efficient in all relevant parameters' unless we say otherwise. For an example of something that is efficient in one parameter, but not in another we could look at calculating the matrix of a quantum circuit. This scales linearly with the number of gates, but exponentially with the number of qubits, and so we might say this problem is 'efficient in the number of gates'.

An important note must be made here, and that is that 'efficient' should be taken with a grain of salt. We say something is efficiently calculable when the calculation time is bounded by some polynomial. But what if this polynomial is n^{100}? In that case a computer would quickly need to run for thousands of years for even small values of n. The reason computer scientists

have decided to make the benchmark of 'efficient' be 'polynomial', is because it is the smallest useful class of orders that is closed under composition: if we have an algorithm taking polynomial time that is used as a subroutine in another program which is called a polynomial number of times, then the resulting algorithm still takes polynomial time. With this definition we can hence safely combine efficient programs and be ensured that the result is still 'efficient'. Another caveat is that while it is technically possible for an algorithm to have a runtime that scales with n^{100}, *in practice*, most polynomial time algorithms used in the real world have a complexity where the exponent is small, such as $O(n^3)$.

Ultimately, the only real way to know whether an algorithm is efficient is to implement it on a computer and benchmark it on the problem instances you care about.

2.5.2 Classical complexity classes

We can classify problems in terms of how hard they are to solve in the asymptotic case. We call these classes **complexity classes**. We have for instance the complexity class **P** that contains all the problems that can be solved deterministically in polynomial time. We have to be a bit careful here by what we mean by 'problem'. **P** is a class of **decision problems**. These are problems where the answer is either yes or no, true or false, 0 or 1. An example problem that is in **P** is determining whether a given input bit string returns 1 when we apply a given Boolean function to it. We often care about problems where the answer is not just a yes or no answer, but where we have a more complicated type of output, like an integer, a matrix, or a quantum circuit. We call such problems **function problems**, and the complexity class of function problems that can be solved deterministically in polynomial time is called **FP**. For instance, adding or multiplying two numbers is in **FP**, and so is Gaussian eliminating a matrix. We consider the complexity classes **P** and **FP** as containing the problems that are efficiently solvable.

Probably the most (in)famous complexity class is **NP**, standing for Non-deterministic Polynomial time. **NP** contains the problems you could efficiently solve if you had a computer where every time you had to make a decision, you could actually run both branches in parallel, and as long as one of the branches returns 'yes', the whole computation returns 'yes'. This definition is a bit mind-bending to think about, but luckily there is an easier way to think about **NP**: it contains the problems where we can verify the solutions in polynomial time. Suppose for instance I am given a half-finished

sudoku, and I'm asked whether this sudoku can actually be finished. The only way I might be able to do that is just trying to finish the sudoku and see if I got stuck. This can be quite inefficient (especially if you are bad at sudoku). If someone gives you a completion of the sudoku however, it is easy to check that it was filled in correctly, and hence that the sudoku is indeed solvable.

A canonical problem that is in **NP** is determining whether a given Boolean formula $f : \mathbb{F}_2^n \to \mathbb{F}_2$ can be satisfied, that is whether there exists an $\vec{x} \in \mathbb{F}_2^n$ such that $f(\vec{x}) = 1$. If we are given an $\vec{x}$ such that $f(\vec{x}) = 1$, then we can easily check this is the case by just calculating what f does on $\vec{x}$. This problem of Boolean satisfiability, or SAT for short, is in fact an **NP-complete** problem. That means it is among the 'hardest' problems in **NP**. More specifically, if we have some algorithm for solving SAT, then we can use this to solve any other problem in **NP** with at most polynomial overhead.

Arguably the most famous problem in computer science is whether $\mathbf{P} \neq \mathbf{NP}$. That is, whether there are any problem for which we can efficiently check the solution, but we cannot efficiently find the solution. Most researchers believe this is the case, but through decades of effort we still do not know the answer. In the complexity theory literature you will find many results that are *conditional* on $\mathbf{P} \neq \mathbf{NP}$ (or on similar claims about complexity classes not being equal to each other): "If the problem X is easy to solve, then $\mathbf{P} = \mathbf{NP}$, hence we think that X is *not* easy to solve." In this book we will also sometimes make claims like these, when we want to argue that we don't expect to be able to solve some problem efficiently.

2.5.3 BQP

The classes **P** and **NP** are *classical*, in the sense that they describe problems that a classical computer can (not) efficiently solve. In this book we will obviously also care about which problems a *quantum* computer can (not) efficiently solve. To define the relevant complexity class, we will first define another classical complexity class.

The class **BPP** contains the decision problems that can be solved with Bounded error, Probabilistically, in Polynomial time. That is, we are allowed to solve the problem using an algorithm that makes random probabilistic choices, and that is allowed to make some errors, as long as it gives the right answer in the (vast) majority of cases. We have $\mathbf{P} \subseteq \mathbf{BPP}$, but we don't know whether they are equal.

The class of problems efficiently solvable by a quantum computer is defined similarly to **BPP**. The class **BQP** contains the decision problems that can

be solved with Bounded error, using a Quantum computer, in Polynomial time. Formally, we would construct a particular quantum circuit based on the particular instance of the problem (this translation into a quantum circuit would be given by some classical algorithm running in polynomial time), and then measure a single qubit. Our answer is then whether this qubit gives the outcome 0 or 1. In practice we probably want to post-process the output of a quantum computer, or run the computation many time to gather more statistics, but formally we can make all these steps also 'quantum', so that there is just one big quantum computation taking care of everything, hence why **BQP** is defined this way. Note that we haven't specified what gate set the quantum computer is using. Because all approximately universal gate sets are essentially equivalent (Section 2.3.5), this is fine, and the complexity class is the same regardless of which gate set is being used.

We have of course $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{BQP}$. It is not known whether $\mathbf{NP} \subseteq \mathbf{BQP}$ or $\mathbf{BQP} \subseteq \mathbf{NP}$ or if neither is the case. It is widely believed that $\mathbf{NP} \not\subseteq \mathbf{BQP}$, and hence that NP-complete problems like SAT should not have an exponential speed-up on a quantum computer, but at most a polynomial speed-up.

The final complexity class we will consider is a quantum analogue to **NP**. Quantum Merlin-Arthur (we will let the reader speculate where this name comes from), or **QMA** for short, is the class of decision problems that a quantum computer can verify with high probability. A problem that is **QMA**-complete is for instance determining whether a given quantum circuit is equal to the identity, or is far away from being the identity, promised that one of these is the case: given a circuit not equal to the identity, there will be a state that is mapped to something quite different, and hence when giving the verifying quantum computer this state, it can run the circuit and see that it is indeed mapping this state to something else by doing some appropriate measurement.

2.6   Summary: What to remember

1. A tensor is a generalisation of a matrix that can have any number of inputs and outputs.
2. A tensor network is a linear map built out of tensors by tensor product and contraction.
3. A string diagram is a graphical representation of a tensor network where the tensors are boxes, and wires connecting boxes denote contractions.
4. Deforming a string diagram by moving tensors around or bending wires preserves the meaning the map.

5. We can summarise the basics of quantum theory with the SCUM model:

   - States are represented by normalised states on a complex Hilbert space.
   - Compound systems are made by taking the tensor product of their Hilbert spaces.
   - Unitaries describe the dynamics of a system.
   - Measurements follow the Born rule.

6. The quantum circuit model represents a quantum computation by starting with a simple state, and then applying a sequence of *gates*, unitaries usually acting on a small number of qubits, before finally measuring the qubits in a fixed basis.

7. The state-space of a qubit can be modelled by the Bloch sphere. Single-qubit unitaries then correspond to rotations of this sphere. Using Euler angles any unitary can be decomposed into rotations along the Z- and X-axis.

8. Other useful quantum gates include the CNOT, CZ, Hadamard, and Toffoli.

2.6.1   Advanced Material*

2.6.2   Quantum mixed states and channels*

This first "Advanced Material" section isn't really that advanced, but it is optional, since we will focus on pure quantum states and operations for almost the entirety of this book. However, for the sake of completeness, we will briefly explain how to model more general states and processes, as this can often be more convenient for certain calculations.

Pure quantum states and processes are the correct mathematical objects for representing a quantum system in isolation evolving according to a process that is completely known. Of course, this never happens in the real world. The way we model all physical processes is necessarily a simplification, where we choose to disregard parts of the system which (we hope!) won't have too big of an effect on the outcome. When we do this, we limit the knowledge and degree of control we have over a quantum state. We can model these limitations in a rigorous mathematical way using quantum **mixed states** and **channels**.

A quantum mixed state is a generalisation of the pure states we saw before, and can be used to model one of two seemingly different scenarios which actually turn out to be mathematically identical in quantum theory. A mixed state can capture:

1. an unknown member of a fixed set of pure states, which are drawn from a classical probability distribution, or
2. part of a quantum pure state on two systems where we don't know (or care) what happens to the state on one of the systems.

Situation 1 above is sometimes called an *ensemble*, whereas situation 2 is called the *reduced state*. From a practical point of view, the only utility of a quantum state is to provide just enough data to compute Born rule probabilities of measurements. To do that in either of these situations, we just need the following data.

Definition 2.6.1. A quantum mixed state on a Hilbert space $\mathcal{H}$ is a positive operator $\rho : \mathcal{H} \to \mathcal{H}$ such that $\mathrm{Tr}(\rho) = 1$.

We say that mixed states generalise pure states because, for every pure state $|\psi\rangle$, the associated ket-bra $|\psi\rangle\langle\psi|$ is a positive semi-definite operator where $\mathrm{Tr}(|\psi\rangle\langle\psi|) = \langle\psi|\psi\rangle = 1$. A nice feature of this representation is that the need to consider states "up to" a global phase is now handled automatically:

$$e^{i\alpha}|\psi\rangle \quad \rightsquigarrow \quad (e^{i\alpha}|\psi\rangle)(e^{-i\alpha}\langle\psi|) = e^{i\alpha}e^{-i\alpha}|\psi\rangle\langle\psi| = |\psi\rangle\langle\psi| \quad \leftsquigarrow \quad |\psi\rangle$$

A recurring theme with mixed states is that things we said with inner products for pure states can now be stated in terms of the trace of a linear map. Normally, if we specialise to pure states, we can use the cyclicity of the trace (i.e. that $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$) to recover our original pure state concept.

For example, if we fix a measurement $\{M_1, \ldots, M_m\}$, the **Born rule for mixed states** says that $\mathrm{Prob}(i \mid \rho) = \mathrm{Tr}(M_i\rho)$. Specialising to pure states gives the Born rule from the previous section:

$$\mathrm{Prob}(i \mid |\psi\rangle\langle\psi|) = \mathrm{Tr}(M_i|\psi\rangle\langle\psi|) = \langle\psi|M_i|\psi\rangle$$

If we have situation 1 above of an ensemble of pure states, i.e. a collection of pure states $\{|\psi_1\rangle, \ldots, |\psi_k\rangle\}$ drawn from a probability distribution $\{p_1, \ldots, p_k\}$, we can represent this as:

$$\rho = \sum_j p_j |\psi_j\rangle\langle\psi_j| \tag{2.24}$$

Applying the Born rule gives:

$$\mathrm{Prob}(i \mid \rho) = \mathrm{Tr}(M_i(\sum_j p_j |\psi_j\rangle\langle\psi_j|)) = \sum_j p_j \langle\psi_j|M_i|\psi_j\rangle = \sum_j p_j \mathrm{Prob}(i \mid |\psi_j\rangle)$$

In other words, it is the sum over all of the different ways we could have gotten

outcome i, weighted by the appropriate probabilities. This is exactly how classical probability theory would tell us we should compute this quantity.

For situation 2 of a reduced state, we can represent ρ as

$$\rho = \sum_j (I \otimes \langle j|)|\Psi\rangle\langle\Psi|(I \otimes |j\rangle) \tag{2.25}$$

We saw this operation, the partial trace, at the end of Section 2.1.6. Notably, it doesn't depend on the choice of ONB $\{|j\rangle\}$ and can be graphically depicted as:
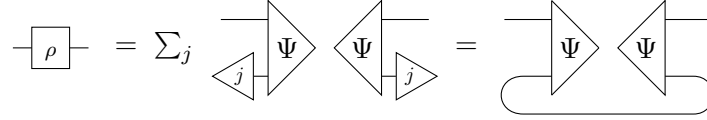
Now, it is easy to check that, for any measurement on the first system, the Born rule for ρ gives:

$$\mathrm{Prob}(i \mid \rho) = \sum_j \mathrm{Prob}(ij \mid |\Psi\rangle\langle\Psi|)$$

That is, we get the same thing as if we measured both systems of the pure state $|\Psi\rangle\langle\Psi|$, and then marginalised over the second measurement outcome. Since Eq. (2.25) doesn't depend on the choice of ONB $\{|j\rangle\}$, it actually doesn't matter which measurement we did on the second system, so this is like not knowing or caring what happened to it.

Notably, both (2.24) and (2.25) can represent a generic trace-1 positive operator ρ, so they are equivalent.

Quantum mixed states are the most general representation of a quantum state in quantum theory. The most general representation of a process is a **quantum channel**. This is a linear map which completely preserves the set of quantum states. That is, a quantum channel Φ should have the property that if ρ is a quantum state in $\mathcal{H}$, then $\Phi(\rho)$ should be a quantum state in $\mathcal{H}'$. However, this is not quite strong enough, because we need to take compound systems into account.

Let $L(\mathcal{H})$ be the vector space of linear operators from $\mathcal{H}$ to itself. Then Φ should be a linear map from $L(\mathcal{H})$ to $L(\mathcal{H})$. This is sometimes called a linear super-operator. We can define the tensor product of linear super-operators as follows:

$$(\Phi \otimes \Psi)(e_{ij} \otimes e_{kl}) := \Phi(e_{ij}) \otimes \Psi(e_{kl})$$

where e_{ij} represents the matrix containing a single 1 in position (i,j) and zeros everywhere else. Since such matrices span the whole space of operators, this fully defines $\Phi \otimes \Psi$ by linearity.

For a map Φ to be a quantum channel, it should not only preserve the set of quantum mixed states when it is applied to the whole state, but it should also preserve the set of states when it is applied to just one subsystem. We can say this in terms of the tensor product of Φ with the identity superoperator $1_{\mathcal{K}}(\rho) := \rho$ on an arbitrary Hilbert space $\mathcal{K}$.

Definition 2.6.2. A linear super-operator $\Phi : L(\mathcal{H}) \to L(\mathcal{H}')$ is called a **quantum channel** if for any Hilbert space $\mathcal{K}$ and any trace-1 positive operator $\rho \in L(\mathcal{H} \otimes \mathcal{K})$, $(\Phi \otimes 1_{\mathcal{K}})(\rho)$ is also positive and trace-1.

Just like in the case of states, pure maps can be interpreted as a special case of quantum channels. For any unitary U, we can form the **unitary channel** $\widehat{U}(\rho) := U\rho U^{\dagger}$. If we apply such a channel to a pure state, it will send the ket-bra of $|\psi\rangle$ to the ket-bra of $U|\psi\rangle$, as expected:

$$\widehat{U}(|\psi\rangle\langle\psi|) = U|\psi\rangle\langle\psi|U^{\dagger} = U|\psi\rangle(U|\psi\rangle)^{\dagger}$$

There is a lot that can, and has, been said about quantum channels. For this, we refer the interested reader to one of the textbooks in the References for this chapter. For the purposes of this book, we will conclude with one of the most useful tools for doing concrete calculations with channels.

Theorem 2.6.3 (Kraus Representation Theorem)**.** A linear super-operator $\Phi : L(\mathcal{H}) \to L(\mathcal{H}')$ is a quantum channel if and only if there exists a set of linear operators $\{B_j : \mathcal{H} \to \mathcal{K}\}_i$ called the **Kraus operators**, such that $\Phi(\rho) = \sum_j B_j \rho B_j^{\dagger}$ and $\sum_j B_j^{\dagger} B_j = I$.

Clearly unitary channels are a special case, but this allows more general maps such as probabilistic mixtures of unitaries like:

$$\Phi(\rho) = \sum_j p_j U_j \rho U_j^{\dagger}$$

and even more general things such as discarding our state all together and preparing a fixed pure state.

Exercise 2.17. Show that the super-operator $\Phi(\rho) := \text{Tr}(\rho)|0\rangle\langle 0|$ is indeed a quantum channel by giving its Kraus operators. Show that furthermore it cannot be written as a probabilistic mixture of unitaries.

2.7 References and further reading

On the origin of quantum computing There is a rich history of connections between physics and computation in the early 20th century, typified by the

works of Landauer (1961) connecting principles of information, computation, and thermodynamics. Inspired by these lines of thought, Paul Benioff (1980) showed that a Turing Machine could be accurately modelled by quantum mechanics, but stops short of using quantum processes to go beyond classical computation. At the same time, on the other side of the Iron Curtain, Yuri Manin (1980) proposed the first thing to look something like a genuine quantum computer in his book *Computable and Uncomputable*, in the form of a "quantum automaton" that can make use of superposition and entanglement for its computations. One year later, and seemingly independently, Richard Feynman (1982) quipped that "Nature isn't classical, dammit" and went on to lay out a proposal for how something like a quantum computer could be used to simulate quantum phenomena, a problem that seems to intractable on a classical computer.

Quantum Turing machines The first universal quantum computer, in the sense that we understand today, was described by David Deutsch (1985), using the formalism of *quantum Turing machines*. Quantum Turing machines are analogous to classical Turing machines, but with the "tape" replaced by Hilbert spaces and the typical read/write operations replaced by quantum operations. It is interesting to note that Deutsch's original paper was motivated largely by answering foundational questions about the nature of quantum physics and computation. Deutsch, a lifelong proponent of Everett's "Many worlds" interpretation of quantum theory, emphasises that the parallelism present in the quantum Turing machine is something that cannot be meaningfully captured by any classical description of computation, and claims in the abstract of his seminal paper: "The intuitive explanation of these properties places an intolerable strain on all interpretations of quantum theory other than Everett's."

Quantum circuits While quantum Turing machines give a universal model of quantum computation, their definition is somewhat unwieldy. Hence, a few years later they were largely superseded by another universal model of quantum computation, also due to Deutsch (1989), which is now called the *quantum circuit model*. Deutsch himself called quantum circuits "quantum networks", and this may have been a more appropriate name. As we noted at the beginning of this chapter, the analogy between classical and quantum circuits is already somewhat strained. However, this name stuck, and it was the name used by Chi-Chih Yao (1993) when he showed the polynomial-time equivalence of the quantum circuit model with quantum Turing machines.

Universal gate sets The notion of universal sets of gates for quantum circuits goes all the way back to the first papers in the field. Deutsch (1985) showed that arbitrary classical reversible computation plus a generating set of single-qubit unitaries is approximately universal for quantum computation. Since any classical reversible computation can be generated from the Toffoli gate, this implies that 3-qubit gates are universal for quantum circuits. This was made slightly stronger in (Deutsch, 1989), where Deutsch showed that a single 3-qubit gate and ancillae could be used to approximate arbitrary n-qubit unitaries. However, the question of universality with 2-qubit gates remained open until DiVincenzo (1995) showed that 2-qubit gates generate the entire family of n-qubit unitaries. That same year, the result was strengthened to show that a controlled-NOT gate and single-qubit unitaries suffice for universal computation Barenco et al. (1995). What is now known as the *Solovay-Kitaev theorem*, that any approximately universal gate set can approximate any other arbitrarily well with just a poly-logarithmic overhead, was first announced by Solovay in 1995 and independently proved in (Kitaev, 1997, Lemma 4.7). A good reference for this result is Dawson and Nielsen (2005).

Quantum algorithms Following the formalisation of the quantum circuit model, many influential quantum algorithms started to appear in the 1990s, such as the Deutsch-Jozsa algorithm (Deutsch and Jozsa, 1992), Shor's factoring algorithm (Shor, 1994, 1997), and Grover's search algorithm (Grover, 1996). The hidden subgroup algorithm, along with its encoding of Shor's factoring algorithm and Simon's problem, was given by Jozsa (1997). Quantum algorithms is now a huge area, which we hardly touch on in this book, but there are many excellent resources out there. A standard starting point is Nielsen and Chuang's canonical textbook *Quantum computation and quantum information* (Nielsen and Chuang, 2010). Some overviews and surveys of quantum algorithms include (Mosca, 2008), (Ambainis, 2010), (Montanaro, 2015), and (Dalzell et al., 2023). Finally, the Quantum Algorithms Zoo (2022) is an excellent online resource which has collected and categorised many quantum algorithms.

Quantum complexity theory The complexity class BQP was defined by Bernstein and Vazirani (1997) and used to give evidence, by means of an oracle separation proof, that bounded-error quantum computation is strictly more powerful (in a complexity-theoretic sense) from bounded-error probabilistic classical computation. The complexity class QMA was defined by Kitaev, apparently in a 1999 lecture at Hebrew University (Aharonov and Naveh,

2002), in order to give quantum analogues of the complexity class NP and the famous Cook-Levin theorem for the NP-completeness of SAT.

Quantum programming There are a number of higher-level quantum programming languages. The first attempt at a practically usable Quantum Programming Language (QPL) that went beyond a theoretical framework like quantum Turing machines or quantum lambda calculus was the imperative language QCL Ömer (2002) originally presented in Ömer's Master thesis in 2000. A more modern and more developed functional language is Quipper Green et al. (2013), which is an embedded language into Haskell. Microsoft released their Q# in 2017, which is based on C# Microsoft (n.d.). The main challenge of a quantum programming language is to deal with the interactions between classical and quantum information, with quantum variables for instances not being able to be cloned or freely read out. Every high-level QPL needs to compile down into low-level instructions that can be more easily understood by physical quantum devices. A useful intermediary language for this is Open Quantum Assembly, known as OpenQASM Cross et al. (2017), which at the time of writing sits at version 3.0 Cross et al. (2022). There are also a number of integrated programming environments and compilers that offer the user a combination of features related to programming high-level algorithms, transpile this down to lower level QASM, and compiling it further down into machine-specific instructions. Some examples include IBM's Qiskit IBM (n.d.), Quantinuum's TKET Quantinuum (n.d.), Xanadu's PennyLane Xanadu (n.d.), or the library written by the authors of this book, PyZX Kissinger and van de Wetering (2020a).

Quantum textbooks In addition to Nielsen and Chuang's aforementioned textbook, there are many sets of lecture notes that provide a good all-around introduction to quantum computing. For a strong emphasis on algebraic and quantum information theoretic aspects, a good resource is the lecture notes of Watrous (2006), which later became a textbook (Watrous, 2018). Another excellent set of lecture notes is that of John Preskill (2015), which contains among many other things a very nice introduction to stabiliser theory and quantum error correction, which we'll cover in Chapters 6 and 11, respectively.

3

The ZX-Calculus

Now that we have covered the basic concepts of quantum computing, it is time to do some interesting stuff. In the previous chapter we discussed string diagrams and how they can be used to represent linear maps and, in particular, quantum circuits. Actually *calculating* the linear map they represent is tedious though, and in some cases also (impossibly) hard, as the dimension of the spaces involved scales exponentially in the number of qubits. It would then certainly be nice if we could work directly with string diagrams instead. In this chapter we will introduce the **ZX-calculus**, a graphical language that works with a particular type of string diagrams we call **ZX-diagrams**. These ZX-diagrams can represent any quantum computation. In addition, the ZX-calculus gives us **rules** for manipulating ZX-diagrams. In this way we can prove interesting properties of states, unitaries, projections, or any other linear map between qubits we want, all the while working with diagrams instead of those pesky matrices.

In this chapter we will see how to represent quantum circuits as ZX-diagrams, and how to use the basic rules of the ZX-calculus to start proving some interesting things about them. We will just barely be scratching the surface however, as there is much more to be said about the things we can do with the ZX-calculus. In fact, we will be spending the rest of the book doing exactly that!

3.1 ZX-diagrams

3.1.1 Spiders

ZX-diagrams arise as compositions and tensor products of the following basic linear maps:

Z-spiders	**X-spiders**
$\vdots \; \alpha \; \vdots \; =$	$\vdots \; \alpha \; \vdots \; =$
$\lvert 0...0 \rangle \langle 0...0 \rvert + e^{i\alpha} \lvert 1...1 \rangle \langle 1...1 \rvert$	$\lvert +...+ \rangle \langle +...+ \rvert + e^{i\alpha} \lvert -...- \rangle \langle -...- \rvert$

These generators are called spiders for no other reason than the fact that they look a bit like spiders when you draw them.

Let's focus on the Z-spider. Most of what we say will also be true of the X-spider, by symmetry. First, note this is actually not just a single map, but a family of linear maps of the form:

$$\left\{ Z_m^n[\alpha] : (\mathbb{C}^2)^{\otimes m} \to (\mathbb{C}^2)^{\otimes n} \;\middle|\; m, n \in \mathbb{N}, \alpha \in [0, 2\pi] \right\}$$

where the notation $(-)^{\otimes n}$ means 'n tensor copies', e.g.

$$(\mathbb{C}^2)^{\otimes n} := \underbrace{\mathbb{C}^2 \otimes \ldots \otimes \mathbb{C}^2}_{n} \cong \mathbb{C}^{2^n}$$

The number m is the number of input 'legs' and the number n is the number of output 'legs'. These two numbers together are called the **arity** of a spider. These correspond to the number of qubits which the map takes as input and output, respectively:

$$Z_m^n[\alpha] \;\; := \;\; m\left\{ \; \vdots \; \alpha \; \vdots \; \right\}n \;\; = \;\; \lvert 0 \rangle^{\otimes n} \langle 0 \rvert^{\otimes m} + e^{i\alpha} \lvert 1 \rangle^{\otimes n} \langle 1 \rvert^{\otimes m} \quad (3.1)$$

where $\langle j \rvert^{\otimes m}$ and $\lvert j \rangle^{\otimes n}$ are the m- and n-fold tensor products of bras and kets, respectively, and we take the convention that $(...)^{\otimes 0} = 1$.

Keep in mind that, just like in circuit notation, the ordering of composition is flipped between the picture and the bra-ket expression. For example, wires coming in the *left* side of the diagram above correspond to bras which actually appear on the *right* side of the bra-ket expression. That's why the m's and n's seem to have flipped around in equation (3.1).

We can also describe a Z-spider in matrix notation. Start with a $2^n \times 2^m$ matrix of all 0's, then add a 1 to the top-left corner and an $e^{i\alpha}$ to the bottom-right corner:

$$\vdots \; \alpha \; \vdots \;\; = \;\; \begin{pmatrix} 1 & 0 & & \\ 0 & 0 & \ddots & \\ & \ddots & 0 & 0 \\ & & 0 & e^{i\alpha} \end{pmatrix}$$

When $m = n = 0$, we'll get a $2^0 \times 2^0 = 1 \times 1$ matrix (i.e. a scalar), so the top-left corner *is* the bottom-right corner. In that case, the resulting scalar is $1 + e^{i\alpha}$:

$$\boxed{\alpha} \;=\; 1 + e^{i\alpha}.$$

Unlike quantum gates, Z-spiders can have a different number of inputs from outputs, and hence do not need not be unitary. In fact, they are often not even invertible.

Exercise 3.1. Show that for $m > 0, n > 0$, Z-spiders always correspond to matrices with rank 2. What are the other possibilities for the rank of a Z-spider if m or n is 0?

However, when $m = n = 1$, Z-spiders *are* unitary and correspond to the familiar Z-phase gates we met in the previous chapter:

$$\vcenter{\hbox{---}}\boxed{\alpha}\vcenter{\hbox{---}} \;=\; \vcenter{\hbox{---}}\boxed{Z(\alpha)}\vcenter{\hbox{---}} \;=\; |0\rangle\langle 0| + e^{i\alpha}|1\rangle\langle 1| \tag{3.2}$$

We don't always write angles on spiders, in which case the 'default' angle is assumed to be 0:

$$\vcenter{\hbox{$\vdots$}}\!\!\bowtie\!\!\vcenter{\hbox{$\vdots$}} \;:=\; \vcenter{\hbox{$\vdots$}}\!\!\boxed{0}\!\!\vcenter{\hbox{$\vdots$}} \;=\; |0\rangle^{\otimes n}\langle 0|^{\otimes m} + |1\rangle^{\otimes n}\langle 1|^{\otimes m}$$

We will call such spiders **phase-free**. Some familiar quantum states arise out of phase-free Z-spiders when $m = 0$. Namely, we obtain (unnormalised) Bell, GHZ, and n-fold GHZ states:

$$\bigcirc\!\!\!< \;=\; |00\rangle + |11\rangle \qquad \bigcirc\!\!\!\!\Leftarrow \;=\; |000\rangle + |111\rangle \qquad \bigcirc\!\!\!\vdots \;=\; |0\rangle^{\otimes n} + |1\rangle^{\otimes n}$$

To normalise these states, we should multiply both sides in these examples by $\frac{1}{\sqrt{2}}$. We will indicate a scalar multiple of a ZX-diagram just like we would for any other kind of linear map. For example:

$$\frac{1}{\sqrt{2}} \cdot \bigcirc\!\!\!\!\Leftarrow \;:=\; \frac{1}{\sqrt{2}}\left(|000\rangle + |111\rangle\right)$$

Most the time we won't bother with normalisation unless it really matters. It usually doesn't.

Finally, looking at Z-spiders with a single output leg and $\alpha \in \{0, \pi\}$, we can construct X-basis (a.k.a. "plus" basis) elements $|+\rangle$ and $|-\rangle$, again up to normalisation:

$$\begin{aligned}
\frac{1}{\sqrt{2}} \cdot \bigcirc\!\!\!- &= \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) =: |+\rangle \\
\frac{1}{\sqrt{2}} \cdot \boxed{\pi}\!\!- &= \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) =: |-\rangle
\end{aligned} \tag{3.3}$$

X-spiders are pretty much the same as Z-spiders, but everything is defined relative to the X-basis rather than the Z-basis:

$$X^n_m[\alpha] \quad := \quad m\left\{ \begin{array}{c} \vdots \\ \alpha \\ \vdots \end{array} \right\} n \quad = \quad |+\rangle^{\otimes n}\langle+|^{\otimes m} + e^{i\alpha}|-\rangle^{\otimes n}\langle-|^{\otimes m} \quad (3.4)$$

Hence, all of the special cases highlighted above apply equally well to X-spiders, after substituting $0 \leftrightarrow +$ and $1 \leftrightarrow -$. In particular, when $m = n = 1$, the X-spider is unitary and represents the X-phase gate:

$$-\!\!\boxed{\alpha}\!\!- \quad = \quad -\!\!\boxed{X(\alpha)}\!\!- \quad = \quad |+\rangle\langle+| + e^{i\alpha}|-\rangle\langle-| \quad\quad (3.5)$$

It will be useful for us to calculate the definition of an X-spider in terms of the Z-basis. This will enable us, for example, to do concrete matrix calculations involving both kinds of spiders, as well as understand some of the interaction properties we'll meet later in this chapter.

We'll start with the basis states. Note that the Z and the X bases have a symmetric presentation with respect to each other:

$$|+\rangle \;=\; \tfrac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$$
$$|-\rangle \;=\; \tfrac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right)$$
$$|0\rangle \;=\; \tfrac{1}{\sqrt{2}}\left(|+\rangle + |-\rangle\right)$$
$$|1\rangle \;=\; \tfrac{1}{\sqrt{2}}\left(|+\rangle - |-\rangle\right)$$

Consequently, we can modify (3.3) to get an expression for the two Z-basis states:

$$\tfrac{1}{\sqrt{2}} \cdot \!\!-\!\!\circ\!\!- \quad = \quad \tfrac{1}{\sqrt{2}}\left(|+\rangle + |-\rangle\right) \;=:\; |0\rangle$$
$$\tfrac{1}{\sqrt{2}} \cdot \!\!\boxed{\pi}\!\!- \quad = \quad \tfrac{1}{\sqrt{2}}\left(|+\rangle - |-\rangle\right) \;=:\; |1\rangle \quad\quad (3.6)$$

Notice that *X-spiders* are used to construct *Z-basis* states and vice-versa. This is a common "gotcha" in ZX-calculation: to get basis elements, make sure you use spiders of the *opposite* colour.

Now, let's calculate the matrix for the special case where $\alpha = 0$, $m = 2$ and $n = 1$:

$$\succ\!\!\circ\!\!- \quad = \quad |+\rangle\langle++| + |-\rangle\langle--| \quad\quad (3.7)$$

If we expand the first term in the expression above, we'll get a sum over all 8 possible combinations of Z-basis elements, times the normalisation factor:

$$|+\rangle\langle++| = \tfrac{1}{2\sqrt{2}} \cdot \big(\, |0\rangle\langle00| + |0\rangle\langle01| + |0\rangle\langle10| + |0\rangle\langle11|$$
$$+ |1\rangle\langle00| + |1\rangle\langle01| + |1\rangle\langle10| + |1\rangle\langle11| \,\big)$$

Expanding the second term in (3.7), we get almost the same thing, except we pick up a -1 sign whenever there is an odd number of 1's:

$$|-\rangle\langle--| = \tfrac{1}{2\sqrt{2}} \cdot (|0\rangle\langle00| - |0\rangle\langle01| - |0\rangle\langle10| + |0\rangle\langle11|$$
$$- |1\rangle\langle00| + |1\rangle\langle01| + |1\rangle\langle10| - |1\rangle\langle11|)$$

In Eq. (3.7) we add these two terms together, so we see that when we expand them that each term with an odd number of 1's cancels, while those with an even number add together. We hence have:

$$\rawhtml >\!\!\!\!\!\bullet\!\!- \ = \ \frac{1}{\sqrt{2}} (|0\rangle\langle00| + |0\rangle\langle11| + |1\rangle\langle01| + |1\rangle\langle10|) \tag{3.8}$$

We see then that this map is 'classical': it sends Z-basis states to Z-basis states. In particular, it maps $|00\rangle$ and $|11\rangle$ to $|0\rangle$, while $|01\rangle$ and $|10\rangle$ are mapped to $|1\rangle$. But this is exactly what the XOR does! So we have $X_2^1[0]|x,y\rangle = |x \oplus y\rangle$ where $x \oplus y$ is the XOR of the two bits $x, y \in \mathbb{F}_2$.

Exercise 3.2. What classical map would we get if we instead took the X-spider with 2 inputs, 1 output and a π phase?

We can generalise this construction to see what any X-spider does on states in the Z-basis. Let's consider a general X-spider as defined in (3.4). Expanding the first term gives a uniform superposition over all Z-basis elements:

$$|+...+\rangle\langle+...+| \ = \ \left(\tfrac{1}{\sqrt{2}}\right)^{n+m} \cdot \sum_{\substack{i_1...i_m \\ j_1...j_n}} |j_1...j_n\rangle\langle i_1...i_m|$$

The normalisation factor comes from the fact that we have m copies of $\langle+|$ and n copies of $|+\rangle$, each of which contribute a $\frac{1}{\sqrt{2}}$, which multiply together.

Expanding the second term in (3.4), we again get a sum over all of the Z-basis elements, but with different coefficients inside the sum:

$$e^{i\alpha}|-...-\rangle\langle-...-| \ = \ \left(\tfrac{1}{\sqrt{2}}\right)^{n+m} \cdot \sum_{\substack{i_1...i_m \\ j_1...j_n}} (-1)^{i_1+...+i_m+j_1+...+j_n} e^{i\alpha} |j_1...j_n\rangle\langle i_1...i_m|$$

Every term in the sum gets an $e^{i\alpha}$, but when any of the i's or j's is 1, it also gets a factor of -1. Putting the two terms together, we get this expression for an X-spider:

$$\rawhtml \vdots \!\!\bowtie\!\! \vdots \ = \ \left(\tfrac{1}{\sqrt{2}}\right)^{n+m} \cdot \sum_{\substack{i_1...i_m \\ j_1...j_n}} X[\alpha]_{i_1...i_m}^{j_1...j_n} |j_1...j_n\rangle\langle i_1...i_m|$$

where the coefficients are given by:

$$X[\alpha]_{i_1 \ldots i_m}^{j_1 \ldots j_n} \ := \ 1 + (-1)^{i_1 + \ldots + i_m + j_1 + \ldots + j_n} e^{i\alpha}$$

Now, that's a pretty big expression, so let's try to make some sense out of it. First, note that, whenever the bitstring $(i_1, \ldots, i_m, j_1, \ldots, j_n)$ contains an *even* number of 1's, this term has a coefficient of $1 + e^{i\alpha}$, whereas if it contains an *odd* number of 1's, it's coefficient is $1 - e^{i\alpha}$. This can be written succintly in terms of an XOR as follows:

$$X[\alpha]_{i_1 \ldots i_m}^{j_1 \ldots j_n} \ = \ \begin{cases} 1 + e^{i\alpha} & \text{if } i_1 \oplus \ldots \oplus i_m \oplus j_1 \oplus \ldots \oplus j_n = 0 \\ 1 - e^{i\alpha} & \text{otherwise} \end{cases}$$

Here, we used the fact that the XOR of all the elements in the bitstring tells us if there were an even number of 1s or an odd number. For that reason, the XOR of a bunch of bits is sometimes called the **parity** (where the word "parity" means the property of something being even vs. odd).

If we specialise to the case where $\alpha = 0$, we get the numbers 2 and 0, respectively, in the two cases above. By swallowing the 2 into the normalisation factor, we can write the coefficients for a phase-free X-spider as follows:

$$X[0]_{i_1 \ldots i_m}^{j_1 \ldots j_n} = \left(\tfrac{1}{\sqrt{2}}\right)^{n+m-2} \cdot \begin{cases} 1 & \text{if } i_1 \oplus \ldots \oplus i_m \oplus j_1 \oplus \ldots \oplus j_n = 0 \\ 0 & \text{otherwise} \end{cases}$$

We can think of such spiders as a family of "generalised XOR" maps.

3.1.2 Defining ZX-diagrams

We will now give two equivalent definitions of ZX-diagrams: one that is closely related to quantum circuit notation and the other to tensor networks.

3.1.2.1 *ZX-diagrams are "circuits" made of spiders*

Just like gates form the building blocks for circuits, Z-spiders and X-spiders form the building blocks for ZX-diagrams, under tensor product and composition. For example:

$$=: \ \left(X_3^2[\beta] \otimes Z_1^1[\tfrac{\pi}{2}] \right) \left(Z_1^2[0] \otimes X_3^2[\alpha] \right)$$

A ZX-diagram can be built iteratively by composing other ZX-diagrams either horizontally, by connecting the output wires of the first to the input wires of the second, or vertically, simply by 'stacking' the diagrams to create the tensor product. The base case of this inductive construction starts with the Z- and X-spiders.

To demonstrate how this works, let's work through an example: we will show how to construct the CNOT gate using a Z- and X-spider.

The first ingredient we need is the phase-free Z-spider with 1 input and 2 outputs:

$$
\begin{array}{c} \text{—<} \end{array} \quad = \quad \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.9}
$$

Its matrix has 2 columns and 4 rows. In general, the matrix of a general ZX-diagram with n inputs and m outputs will have 2^n columns, and 2^m rows.

Suppose now that we wish to vertically compose the spider (3.9) with an identity, which has matrix $\mathrm{diag}(1,1)$. The way we calculate the result is with the Kronecker product:

$$
\begin{array}{c} \text{—<} \\ \text{——} \end{array} \quad = \quad \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

Note that we said above that a ZX-diagram is built by composing spiders. We see here that that is not entirely true: we also need some 'structural' components like the identity wire. We'll say a bit more about this later.

We calculate the matrix of the other ingredient we need similarly:

$$
\begin{array}{c} \text{——} \\ \text{>—} \end{array} \quad = \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \quad = \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}
$$

Now to make a CNOT we need to horizontally compose these two subdia-

grams:

$$(3.10)$$
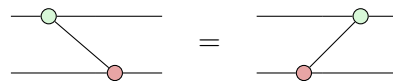
To see that this is indeed a CNOT we calculate its matrix. On the level of the matrix, the horizontal composition of diagrams corresponds to matrix multiplication:

$$
\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}
$$

$$(3.11)$$

Up to a scalar factor of $\sqrt{2}$ this is indeed the matrix of the CNOT gate.

If this matrix calculation business seems tedious: that is precisely *why* we want to use the ZX-calculus! We will see that many identities which would normally be proven by working with matrices can be replaced with graphical reasoning instead.

In Eq. (3.10) we have first the Z-spider and then the X-spider. As we will see later, spiders have many symmetries, and for this reason we can also write the CNOT as first an X-spider and then a Z-spider:

$$(3.12)$$

Exercise 3.3. Prove Eq. (3.12) by directly calculating the matrix the right-hand side represents and showing it agrees with Eq. (3.11).

Because it doesn't matter in which direction the middle wire points, top-left to bottom-right or bottom-left to top-right, we can actually without ambiguity just write the wire vertically:

$$(3.13)$$

If we want to view this diagram as being composed of simple pieces via horizontal and vertical composition then we have to pick an orientation for

this wire, but generally we can just leave it like it is. As a nice bonus this then matches nicely with the quantum circuit notation for a CNOT gate:

$$\vcenter{\hbox{(CNOT diagram)}} \quad = \quad \sqrt{2} \, \vcenter{\hbox{(spider diagram)}} \tag{3.14}$$

We saw that to construct the diagram for a CNOT we needed to use an identity wire. Note that however we could also write the identity as a spider, since:

$$\vcenter{\hbox{(Z-spider wire)}} \quad = \quad \vcenter{\hbox{(plain wire)}} \quad = \quad \vcenter{\hbox{(X-spider wire)}} \tag{3.15}$$

To construct arbitrary ZX-diagrams we also need the ability to swap wires, so that we can connect the spiders up however we want. We can also represent a swap just using spiders, by using the standard trick to build it out of three CNOT gates:

$$\vcenter{\hbox{(swap)}} \quad = \quad \vcenter{\hbox{(three CNOT diagram)}} \tag{3.16}$$

Finally, we also need the cup and cap we saw in Section 2.1.7, but again we can construct these using spiders:

$$|00\rangle + |11\rangle \quad = \quad \vcenter{\hbox{(green cup)}} \quad = \quad \vcenter{\hbox{(plain cup)}} \quad = \quad \vcenter{\hbox{(red cup)}} \quad = \quad |++\rangle + |--\rangle \tag{3.17}$$

For the cap we just make the inputs into outputs instead. Note that Eq. (3.17) is just Eq. (3.15) but with the input bended to be an output.

We see then that the "structural generators" of identity, swap, cup and cap can actually be constructed using spiders, so that we can in fact say that a ZX-diagram *only* consists of spiders. We will however consider these structural pieces as first-class citizens of ZX-diagrams, and not as derived from spiders, since they have useful properties by themselves, which we will see more of in Section 3.1.3.

3.1.2.2 ZX-diagrams are tensor networks

For those readers familiar with tensor networks it might be helpful to note that ZX-diagrams are in fact just tensor networks. Indeed, considering them as tensor networks, our notation matches the standard notion of a graphical tensor network as introduced by Penrose. Considered as a tensor network, each wire in a ZX-diagram corresponds to a 2-dimensional index and a wire between two spiders denotes a tensor contraction. The Z- and X-spiders are

then defined as:

$$
Z[\alpha]^{j_1...j_n}_{i_1...i_m} = \begin{cases} 1 & \text{if } i_1 = ... = i_m = j_1 = ... = j_n = 0 \\ e^{i\alpha} & \text{if } i_1 = ... = i_m = j_1 = ... = j_n = 1 \\ 0 & \text{otherwise} \end{cases}
$$

where i_k, j_l range over $\{0, 1\}$.

Note that in both cases, the definition of the spider doesn't depend on the order of indices nor whether the indices appear in the upper or lower position. Hence, permuting, raising, and lowering indices has no effect on the definition of a spider. For example:

$$
Z[\alpha]^{mn}_{ijk} = Z[\alpha]^{kmn}_{ij} = Z[\alpha]^{nmk}_{ij} = Z[\alpha]^{ijkmn}
$$

So, unless it is important to distinguish inputs and outputs for some other reason (e.g. to do matrix multiplication in the usual way), we might as well take all indices to be upper indices. This allows us to simplify the expression for the Z-spider to:

$$
Z[\alpha]^{j_1...j_n} = \begin{cases} 1 & \text{if } j_1 = ... = j_n = 0 \\ e^{i\alpha} & \text{if } j_1 = ... = j_n = 1 \\ 0 & \text{otherwise} \end{cases}
$$

When $\alpha = 0$, $e^{i\alpha} = 1$ so phase-free Z-spiders further simplify to:

$$
Z[0]^{j_1...j_n} = \begin{cases} 1 & \text{if } j_1 = ... = j_n \\ 0 & \text{otherwise} \end{cases}
$$

This is a generalisation of the Kronecker delta matrix we introduced in Definition 2.1.3 from two indices i, j to any number of indices. The intuition is very much same: a Z-spider 'forces' a collection of indices to take the same value by going to 0 whenever they do not. Consider for example the following tensor network involving a single Z-spider and some generic matrix M:
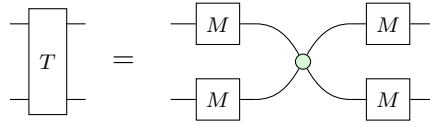
We can compute the elements of the tensor T in terms of M as follows:

$$
T^{kl}_{ij} = \sum_{abcd} M^a_i M^b_j Z[0]^{cd}_{ab} M^k_c M^l_d = \sum_a M^a_i M^a_j M^k_a M^l_a
$$

In the rightmost step, we use a Kronecker-delta-like simplification to replace the sum over multiple indices with a sum over a single index.

In fact, when the Z-spider does have a phase, we can do a similar simplification by collapsing all of its summed-over indices into a single index, but we should keep a 1-index Z-spider around to keep track of the phase. Generalising the example above, we have:
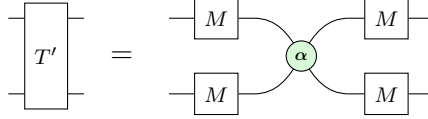
which can be written as a tensor contraction and simplified as follows:

$$(T')^{kl}_{ij} = \sum_{abcd} M^a_i M^b_j Z[\alpha]^{cd}_{ab} M^k_c M^l_d = \sum_a M^a_i M^a_j Z[\alpha]^a M^k_a M^l_a$$

X-spiders are defined identically to Z-spiders, except with respect to the X-basis $\{|+\rangle, |-\rangle\}$. For tensor networks it is more convenient to define everything in terms of a single, fixed basis. Hence, we express an X-spider concretely in terms of the Z-basis. We already computed the coefficients of this expression at the end of Section 3.1.1, so we use them here:

$$X[\alpha]^{j_1 \dots j_n}_{i_1 \dots i_m} = \left(\frac{1}{\sqrt{2}}\right)^{n+m} \cdot \begin{cases} 1 + e^{i\alpha} & \text{if } i_1 \oplus \dots \oplus i_m \oplus j_1 \oplus \dots \oplus j_n = 0 \\ 1 - e^{i\alpha} & \text{otherwise} \end{cases}$$

Just like the Z-spider, the X-spider treats inputs and outputs identically, so we can permute indices or raise/lower them at will:

$$X[\alpha]^{mn}_{ijk} = X[\alpha]^{kmn}_{ij} = X[\alpha]^{nmk}_{ij} = X[\alpha]^{ijkmn}$$

Recall that in the previous section, we interpreted a ZX-diagram as a tensor and composition of some linear maps. This required us to make an arbitrary choice of how to "chop up" the diagram. Equivalently, since we now have expressions for spiders as tensors, we can interpret a ZX-diagram directly as a tensor network. For example the ZX-diagram:
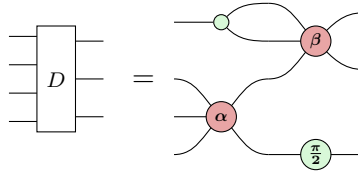
can be interpreted as a tensor network as follows. First assign an arbitrary

index to each wire, e.g.
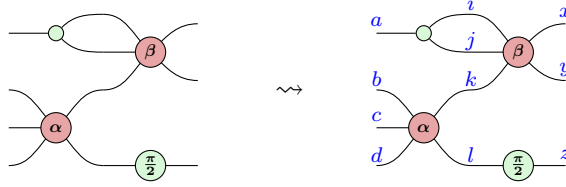
then write D as a contraction involving X- and Z-spiders:

$$D^{xyz}_{abcd} = \sum_{ijkl} Z[0]^{ij}_a X[\alpha]^{kl}_{bcd} X[\beta]^{xy}_{ijk} Z[\frac{\pi}{2}]^z_l$$

Just like for general tensor networks, all of the indices that are not inputs or outputs are summed over.

ZX-diagrams are just tensor networks whose basic components are only X-spider and Z-spider tensors. Certainly most of the ways tensor networks are used in the literature allow the component tensors to be any indexed set of complex numbers. A natural question then is: why focus only on spider tensors and not more general components? There are two reasons. First, we don't lose anything in terms of expressiveness by restricting to spiders. Namely, any linear map on qubits can be constructed from spiders, and hence any tensor network with 2D wires can also be constructed purely out of spider tensors (we will have more to say about this in Section 3.1.7). Second, ZX-diagrams come with an extremely useful set of rewrite rules, the *ZX-calculus*, that lets us transform and simplify the tensor network itself. If our basic tensors were "black box" collections of numbers, we wouldn't be able to do this. We'll first meet these rules in Section 3.2, and we'll be using them throughout the book.
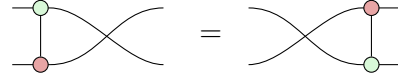
Exercise 3.4. Show that the following equality is true, by writing down the left-hand side as a tensor contraction and simplifying it:

3.1.3 Symmetries

We noted in the previous section that spiders are very symmetric tensors, and that we were allowed to raise, lower or permute their indices however we wanted. In this section we will see what this looks like graphically. Namely, this translates into some nice interactions between cups and caps, swap, and spiders.

The SWAP interacts with spiders the way you would expect. Namely, that
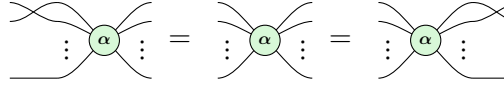
you can 'slide' a spider along the wires, such as in the equation commuting a CNOT gate through a swap below:
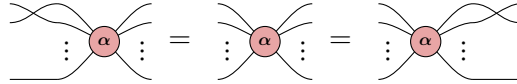
$$\tag{3.18}$$

It is possible to write down a set of equations that formally captures this behaviour of 'sliding through a swap', but we will not bother with that and instead let your intuition do the work. An important point to note though is that we are not working in a 'braided' setting, and hence the swap is self-inverse:

$$\tag{3.19}$$

The fact that we can permute the indices in the tensor of a spider corresponds to being able to swap the wires of their inputs or outputs:

$$\tag{3.20}$$
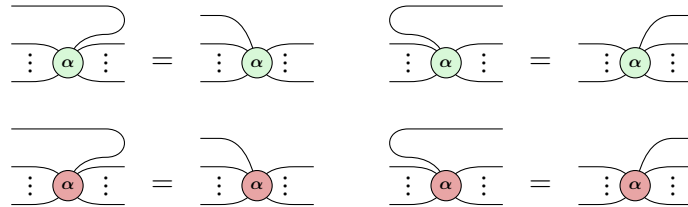
These symmetries hold for all phases α, for spiders with any number of input and output wires, and for the swapping of any of their inputs or outputs.

We can't just swap the wires among the inputs and outputs: we can also swap an input with an output. We do this using the cup and cap.

Just as we can slide spiders along the wires of a swap gate (cf. (3.18)), we can slide a spider along the wires of cups and caps. For instance:

$$\tag{3.21}$$

This again works for Z-spiders and X-spiders, with any phase, and with any number of input or output wires:

$$\tag{3.22}$$

This might just look like some meaningless bending of wires *and that is exactly the point*. Spiders treat inputs and outputs in essentially the same

way, where the distinction between the two is really just bookkeeping, which we represent in the graphical notation as "bending wires".

As a consequence, ZX-diagrams enjoy a particularly strong version of the principle of **Only Connectivity Matters** that was introduced in Section 2.1.6. There, we saw that the only relevant data in a string diagram was which inputs were connected to which outputs. For ZX-diagrams, the order and direction of wires connecting to a single spider are irrelevant. Hence, we can treat a ZX-diagram essentially as a labelled, undirected graph with some inputs and outputs.

To make more clear the type of freedom this entails, *only connectivity matters* means that the following ZX-diagrams all represent the same matrix.

$$(3.23)$$

That is, any way we can interpret the diagrams above as compositions and tensor products of spiders, swaps, identities, cups, and caps will yield the same matrix simply because the spiders and their connectivity are the same.

Note that while we can freely move around the spiders in the diagram, we have to make sure the order of the inputs and outputs of the diagram as a whole stays the same. So for instance, we have:

$$(3.24)$$

This works because for instance the top input stays connected to the same Z-spider in both diagrams, and the same for the other output and inputs. However, the following is not equal:

$$(3.25)$$

Even though the spiders in both diagrams are still connected in the same way, we see that in the second diagram the first input is connected to the X-spider while it was connected to the Z-spider in the first diagram.

We now know enough about ZX-diagrams to address the elephant in the room: why Z and X, but not Y? One reason for this is that Y can be presented as a composition of X and Z, and hence we don't *need* it. In principle however, we could have worked with the 'XY'-calculus instead of

the ZX-calculus, but there is an important reason to prefer Z and X over Y. Namely, the Z and X eigenbases are *self-conjugate* meaning that each of the vectors in the basis is equal to its own componentwise complex conjugate, for instance: $\overline{|0\rangle} = |0\rangle$. This is not the case for the Y eigenbasis $|\pm i\rangle := |0\rangle \pm i|1\rangle$ where we have $\overline{|+i\rangle} = |-i\rangle$. As a result, the 'Y-spider' does not have the nice symmetry between inputs and outputs of (3.22) that the Z- and X-spiders do, which makes it a bit more tricky to work with.

3.1.4 Scalars

In the diagram deformation example given in equation (3.23), the zero-arity X-spider with a $-\frac{\pi}{2}$ phase can be moved completely freely throughout the diagram. This is because a zero-arity spider, or more generally any ZX-diagram with zero inputs and outputs, simply contributes an overall scalar factor. For instance, we have:

$$
\begin{array}{ll}
\bigcirc \;\; = \;\; 2 & \bullet\!\!-\!\!\alpha \;\; = \sqrt{2} \\[2mm]
\pi \;\; = \;\; 0 & \pi\!\!-\!\!\alpha \;\; = \sqrt{2}\,e^{i\alpha} \\[2mm]
\alpha \;\; = 1 + e^{i\alpha} & \bullet\!\!\Longrightarrow\!\!\bigcirc \;\; = \tfrac{1}{\sqrt{2}}
\end{array}
\tag{3.26}
$$

For this reason, we call ZX-diagrams with no inputs and no outputs **scalar ZX-diagrams**. Putting two such diagrams side-by-side corresponds to the tensor product, which for scalars just means multiplication. Note that just by using combinations of the scalar diagrams of (3.26) we can represent any complex number as a scalar ZX-diagram.

Exercise 3.5. By combining the diagrams from (3.26), find a ZX-diagram to represent the following scalar values z:

a) $z = -1$.
b) $z = e^{i\theta}$ for any θ.
c) $z = \frac{1}{2}$.
d) $z = \cos\theta$ for any value θ.
e) Find a general description or algorithm to construct the ZX-diagram for any complex number z.

When working with ZX-diagrams we will often ignore non-zero scalar factors unless they are relevant to the calculation. This is for the same reason that physicists will sometimes work with unnormalised quantum states: it is simply inconvenient and sometimes unnecessary to know the exact scalar values. Recall that we use the notation $M \propto N$ to mean M and N are equal up to a non-zero scalar factor (see Remark 2.3.1).

Because any scalar can be represented by a ZX-diagram, we could work just with ZX-diagrams even if we want to have the correct scalar values. For instance, if we want the scalar-accurate CNOT gate we could write:

$$\tag{3.27}$$

However, we will mostly just write the actual number we need instead of the diagram representing the number. So instead of (3.27) we would just write:

$$\sqrt{2} \tag{3.28}$$

3.1.5   Adjoints, transpose and conjugate

There are some useful global operations on ZX-diagrams that correspond to well-known matrix operations.

First of all, if we take a ZX-diagram and negate all the phases in the spiders, so that for instance $\frac{\pi}{2}$ is mapped to $-\frac{\pi}{2} \equiv \frac{3\pi}{2}$, and π is mapped to $-\pi \equiv \pi$, the matrix of the resulting diagram is the *conjugate* of the matrix of the original diagram (recall that the conjugate of a complex matrix is the pointwise complex conjugate of each the elements of the matrix).

The transpose of a matrix is also easily represented by ZX-diagrams: we use cups and caps to change every input into an output and vice versa.
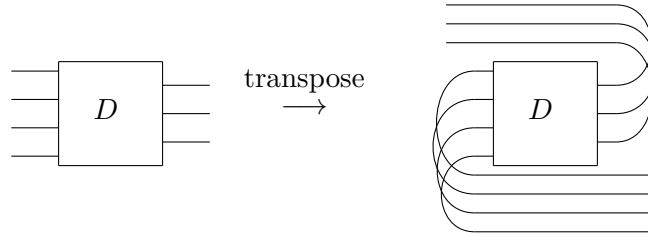
Note that these 'crossovers' of the wires are necessary to ensure that the first input is mapped to the first output, instead of to the last output. If we apply this procedure to the CNOT gate we can verify that the CNOT gate is indeed self-transpose:
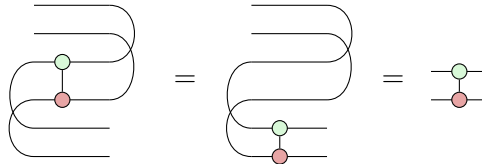
Here in the first step we slid the spiders along the wires to the bottom, and in the second step we applied the first yanking equation (2.10).

The adjoint of a matrix is constructed by taking the transpose and conjugating the matrix. Hence, the adjoint of a ZX-diagram is constructed by interchanging the inputs and outputs using cups and caps as described above, and then negating all the phases of the spiders in the diagram.

3.1.6 Hadamards

We introduce some special notation for the Hadamard gate:

$$
\—\square\— \quad = \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}
$$

This 'Hadamard box' is the final generator of ZX-diagrams that we will need (although note that we can in fact also define it in terms of spiders if we want to; see Section 3.2.5).

By inspecting the matrix of the Hadamard it is easy to check that it is self-transpose:

$$
\vcenter{\hbox{\includegraphics{cap-hadamard}}} \quad = \quad \—\square\— \tag{3.29}
$$

This means that, just like with the spiders, the orientation of the Hadamard is not important, which is why we are allowed to draw it as a simple box. Also just like with the spiders, we can slide Hadamards along wires as intuition would suggest.

3.1.7 Universality

We have now seen that quite a number of quantum gates and constructions can be represented in the ZX-calculus. This raises the following question: exactly which linear maps can be represented in the ZX-calculus. The answer to this is simple: *everything*. Here, 'everything' means any complex matrix of size $2^n \times 2^m$ for some n and m.

To see why this is the case, let's first look at which unitaries we can represent. Eqs. (3.2) and (3.5) show that we can represent Z and X phase gates using spiders. Using the Euler decomposition, any single-qubit unitary can be written as a composition of Z and X phase gates, and hence we can represent any single-qubit unitary as a ZX-diagram. In Eq. (3.14) we additionally saw how to represent the CNOT gate as a ZX-diagram. Note that this equation involves a complex number to get the correct scalar factor, but in Exercise 3.5 we saw how to represent any complex number as a ZX-diagram, so that is no problem. In Section 2.3.5 we recalled that

any many-qubit unitary can be written as a circuit of CNOT gates and single-qubit unitaries. Hence, we then know we can write any unitary as a ZX-diagram:
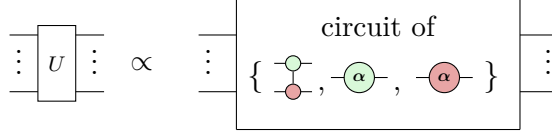
Proposition 3.1.1. Any n-qubit unitary can be written as a ZX-diagram.

Note that for most unitaries, the diagram we would get when we apply this procedure would be exponentially large, so it would not necessarily be *useful* to write it as a ZX-diagram. However, we *can* do it, and that is the important part in this section.

Now let's see which states we can represent as a ZX-diagram. We know we can represent $|0\rangle$ using an X-spider (Eq. (3.6)). Hence, by taking tensor products we can represent $|0\cdots0\rangle$. Now if we have any other normalised n-qubit state $|\psi\rangle$, there will be *some n-qubit unitary that maps $|0\cdots0\rangle$ to $|\psi\rangle$ (think back to your linear algebra classes, why is this the case?). We can then write:

$$\langle\psi | \vdots \quad = \quad \tfrac{1}{\sqrt{2^n}}\; \vdots\; U\; \vdots \tag{3.30}$$

So we can represent any normalised state as a ZX-diagram. Since we also have all complex numbers, we can multiply the normalised state by a number to get an *unnormalised* state. Or, the other way around: if we have an unnormalised state $|\phi\rangle$ we want to represent, we first use Eq. (3.30) to represent the normalised $\frac{1}{\||\phi\rangle\|}|\phi\rangle$ as a ZX-diagram, and then compose this with the number $\||\phi\rangle\|$, also represented as a ZX-diagram.

Proposition 3.1.2. Any n-qubit (unnormalised) state can be written as a ZX-diagram.

Now finally, let's see how to represent arbitrary linear maps as diagrams. Suppose we are given some linear map L from n qubits to m qubits. We can transform this into an $(n+m)$-qubit state by the Choi-Jamiołkowski isomorphism:
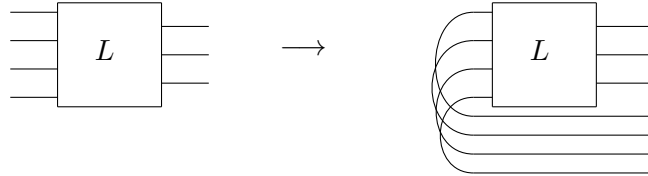
As this is a state, we know we can represent it as a ZX-diagram!

$$
\begin{array}{c} L \end{array} \quad \overset{\substack{3.1.2 \\ (3.30)}}{=} \quad \lambda \quad \begin{array}{c} U \end{array} \tag{3.31}
$$

Here λ is a complex number needed to scale the state to the right normalisation.

We can now bend back the wires in order to get a diagram for L itself:

$$
\begin{array}{c} L \end{array} \quad \overset{(2.10)}{=} \quad \begin{array}{c} L \end{array} \quad \overset{(3.31)}{=} \quad \lambda \begin{array}{c} U \end{array} \tag{3.32}
$$

We see now that we can indeed represent *any* linear map between qubits as a ZX-diagram.

Theorem 3.1.3. Any linear map $L : (\mathbb{C}^2)^{\otimes n} \to (\mathbb{C}^2)^{\otimes m}$ can be represented by a ZX-diagram.

We call this property **universality**. Later on we will also consider various restrictions of ZX-diagrams. We can then also say that this restriction is universal for a given subset of linear maps, if we can represent any linear map from this subset using a ZX-diagram satisfying this restriction.

It is important to emphasise again that just because we *can* represent any linear map using a ZX-diagram, this does not mean that such representations will necessary be 'nice'. It is an ongoing project to find good representations of useful linear maps in the ZX-calculus. For instance, in Chapter 9 we will introduce a new (derived) generator that allows us to more easily represent Toffoli gates in the ZX-calculus, since the 'native' representation as a ZX-diagram is a bit cumbersome.

3.2 The rules of the ZX-calculus

The thing that makes ZX-diagrams so useful, above and beyond plain old tensor networks, is a collection of rewrite rules called the **ZX-calculus**.

These rules enable you to replace a little piece of a ZX-diagram with another, equivalent piece, without changing the linear map represented by the diagram.

Let's look at a simple example, involving a Z-spider from 2 to 1 wires and an X-spider from 1 to 2:

$$\text{—⟨◦} \quad := \quad |00\rangle\langle 0| + |11\rangle\langle 1| \qquad\qquad \text{⟩◦—} \quad := \quad |+\rangle\langle ++| + |-\rangle\langle --|$$

Then, using the fact that:

$$\langle ++|00\rangle = \langle ++|11\rangle = \langle --|00\rangle = \langle --|11\rangle = \tfrac{1}{2}$$

we can do a little calculation:

$$
\begin{aligned}
\text{—◦⟨⟩◦—} \quad &= \quad (|+\rangle\langle ++| + |-\rangle\langle --|)(|00\rangle\langle 0| + |11\rangle\langle 1|) \\
&= \quad \tfrac{1}{2}\,(|+\rangle + |-\rangle)(\langle 0| + \langle 1|) \\
&= \quad \tfrac{1}{2}\,\cdot\, \text{—◦ ◦—}
\end{aligned}
$$

and we see that we have found a pair of ZX-diagrams that are equal, up to a scalar:

$$\text{—◦⟨⟩◦—} \quad = \quad \tfrac{1}{2}\,\cdot\,\text{—◦ ◦—} \tag{3.33}$$

We won't see many concrete calculations with bras and kets in this chapter, because once we know a few simple rules, we will be able to apply them graphically to do all the calculations we need to. For example, now that we have equation (3.33), we can use it to derive equations between bigger ZX-diagrams. Here's one:

$$\text{—◦⟨⟩◦—} \quad = \quad \tfrac{1}{2}\,\cdot\,\text{—◦ ◦—}$$

Note the scalar factor $\tfrac{1}{2}$ pops out in front, by linearity.

The most commonly-used version of the ZX-calculus, and indeed what is often referred to as *the* ZX-calculus consists of the following set of rules in Figure 3.1.

We will use these 7 rules quite a bit in the coming pages, so we've tried to give them some reasonably easy-to-remember names:
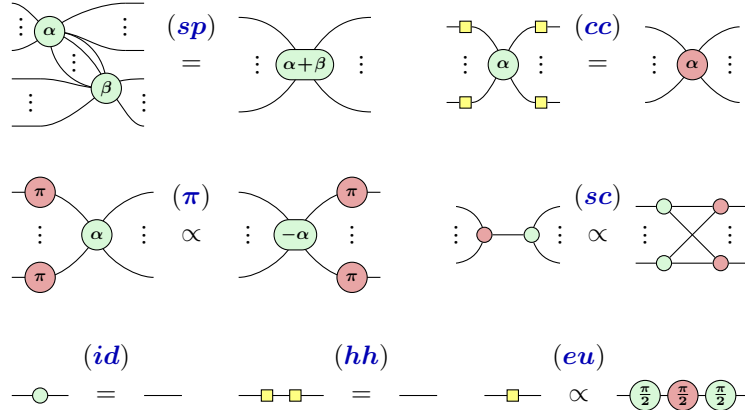
Figure 3.1 The standard rules of the ZX-calculus

(**sp**)	**sp**ider fusion
(**cc**)	**c**olour **c**hange
(**π**)	**π**-commutation rule
(**sc**)	**s**trong **c**omplementarity
(**id**)	**id**entity rule
(**hh**)	**h**adamard-**h**adamard cancel rule
(**eu**)	**eu**ler decomposition of hadamard

This is sometimes also called the **stabiliser** or **Clifford ZX-calculus**, for reasons that will become clear in Section 5.5.

Note that some of the rules of Figure 3.1 only hold up to a non-zero scalar. This is fine for most of the book, since we often don't care about the exact scalar factors. In Section 3.6.2 we discuss rewriting with scalars in more detail, and we present a scalar-accurate version of the rewrites in Figure 3.2.

These rules of the ZX-calculus will be our main tool for reasoning about quantum computation in this book, so we better make sure we are all on the same page about them. In the next few subsections we will discuss where each of the rules comes from, and some examples of how you can use them to prove interesting things.

3.2.1   Spider fusion and identity removal

The most fundamental of all the graphical rewrite rules allowed in the ZX-calculus is **spider fusion**. This says that we can *fuse* two spiders together when they are of the same colour and connected by one or more wires. When

spiders are fused, their phases are added together. In terms of diagrams:

$$(3.34)$$

Note that we interpret the numbers α and β as phases $e^{i\alpha}$ and $e^{i\beta}$, and hence this addition is assumed to be modulo 2π.

Spider fusion can be used to prove many well-known identities of quantum circuits. For instance, that a Z-phase gate commutes through the control of a CNOT gate:

Flipping the colours, we can use spider fusion to show that an X gate commutes through the *target* of a CNOT gate:

That phases add when spiders fuse essentially generalises the observation that two rotations of the Bloch sphere in the same direction add together:

$$(3.35)$$

As we have $|0\rangle \propto$ ●— and $|1\rangle \propto$ (π)— (cf. (3.6)), we also see that spider fusion proves that applying the Pauli X gate to $|0\rangle$ gives $|1\rangle$:

$$(3.36)$$

An equation like $Z|-\rangle = |+\rangle$ is given similarly:

$$(3.37)$$

Note here that we have $\pi + \pi = 2\pi \equiv 0$.

The following rewrite rule we already saw in Eq. (3.15):

$$(3.38)$$

We call this rule **identity removal**. This rule can be interpreted in several ways. Considering a 1-input 1-output spider as a phase gate over either the Z- or X-axis, this rule says that a rotation by 0 degrees does nothing. Combining it with spider fusion it says that the inverse of a rotation by α is a rotation by $-\alpha$:

Composing the diagrams of (3.38) with a cup it says we can make a Bell state by either taking a maximally entangled state over the Z basis or over the X basis:

$$|00\rangle + |11\rangle \;\; = \;\; \text{<image>} \;\; = \;\; \text{<image>} \;\; = \;\; \text{<image>} \;\; = \;\; |{++}\rangle + |{--}\rangle$$

Identity removal also allows us to get rid of self-loops on spiders:

$$\text{<image>} \quad \overset{(3.38)}{=} \quad \text{<image>} \quad \overset{(3.34)}{=} \quad \text{<image>} \tag{3.39}$$

Here in the first step we applied the rule in reverse to *add* an identity. Then we fused the two spiders that are now connected by two wires using the spider fusion rule.

Example 3.2.1. The following quantum circuit implements the GHZ state $|000\rangle + |111\rangle$:

$$\begin{array}{l} |0\rangle \\ |+\rangle \\ |0\rangle \end{array} \text{<image>}$$
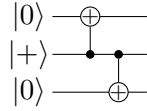
We can verify this by translating it into a ZX-diagram and simplifying it using the rules we have just seen:

$$\text{<image>} \quad \overset{(\boldsymbol{sp})}{=} \quad \text{<image>} \quad \overset{(\boldsymbol{sp})}{=} \quad \text{<image>} \quad \overset{(\boldsymbol{id})}{=} \quad \text{<image>} \tag{3.40}$$

By the definition of the Z-spider, this last diagram is equal to $|000\rangle + |111\rangle$ which is indeed the 3-qubit GHZ state.

3.2.2 The copy rule and π-commutation

The second set of rules of the ZX-calculus we will look at concerns the interaction of the Pauli X and Z gates and their eigenstates $|0\rangle, |1\rangle, |+\rangle, |-\rangle$ with the spiders. Recall from Eqs. (3.3) and (3.6) that:

$$\begin{array}{llll} \text{<image>} & \propto & |0\rangle & \qquad \text{<image>} \;\; \propto \;\; |+\rangle \\ \text{<image>} & \propto & |1\rangle & \qquad \text{<image>} \;\; \propto \;\; |-\rangle \\ \text{<image>} & = & X & \qquad \text{<image>} \;\; = \;\; Z \end{array}$$

The spider fusion rule already covers the interaction of a couple of these

maps and states. Indeed we can use the spider fusion rule to show that a Z gate commutes through a Z-spider:
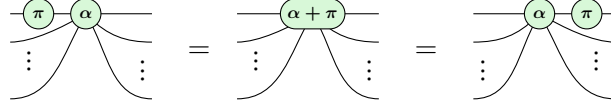
Spider fusion also shows what happens when a $|+\rangle$ or $|-\rangle$ state is applied to a Z-spider:

By flipping all the colours we then also know that an X gate commutes through an X spider and that the $|0\rangle$ and $|1\rangle$ fuse into an X spider.

A more interesting question is what happens to an X gate when it encounters a Z-spider (or vice versa, a Z gate that encounters an X-spider). Consider a Z-spider with just a single input and no phase. Its linear operator is then given by $|0\cdots0\rangle\langle0| + |1\cdots1\rangle\langle1|$. Hence, if we apply an X gate to the input the resulting linear operator is

$$|0\cdots0\rangle\langle0|X + |1\cdots1\rangle\langle1|X \;=\; |0\cdots0\rangle\langle1| + |1\cdots1\rangle\langle0|,$$

as an X gate interchanges $|0\rangle$ and $|1\rangle$. This is easily seen to be equivalent to a Z-spider *followed* by an X gate on each of the outputs:

$$|0\cdots0\rangle\langle1| + |1\cdots1\rangle\langle0| \;=\; (X\otimes\cdots\otimes X)|1\cdots1\rangle\langle1| + (X\otimes\cdots\otimes X)|0\cdots0\rangle\langle0|.$$

Hence, in terms of diagrams, we have:

(3.41)

We will refer to this as the **π-copy rule**. By applying the appropriate cups and caps we see that such a π-copy rule continues to hold regardless of how many inputs and outputs the spider has. But what about when the spider has a non-zero phase? Using spider fusion in reverse to *unfuse* the phase we can reduce this to a simpler question:

(3.42)

Hence, we need to resolve what happens when we apply an X gate to a $|0\rangle + e^{i\alpha}|1\rangle$ state:

$$X(|0\rangle + e^{i\alpha}|1\rangle) = X|0\rangle + e^{i\alpha}X|1\rangle = |1\rangle + e^{i\alpha}|0\rangle = e^{i\alpha}(|0\rangle + e^{-i\alpha}|1\rangle).$$

Ignoring the global phase $e^{i\alpha}$ we can write this in terms of a diagram:

 (3.43)

The most generic case, together with its colour-flipped counterpart is then:

(3.44)

With this rule we can also see what happens if there are already multiple π phases present:

(3.45)

We see then that the position of the π's on the wires of the spider gets toggled.

Similar copy rules hold for the eigenstates of the X and Z operators. Indeed, applying a $|0\rangle$ to the 1-input Z-spider $|0\cdots0\rangle\langle0| + e^{i\alpha}|1\cdots1\rangle\langle1|$ it is clear that we get the output $|0\cdots0\rangle$:

 (3.46)

The same works for $|1\rangle$ (although we then do get a global phase of $e^{i\alpha}$ that we will ignore):

 (3.47)

By introducing a Boolean variable a that is either 0 or 1 we can represent the last two equations as a single parametrised equation:

 (3.48)

Here the $a\pi$ phase is either zero (when $a = 0$), or π (when $a = 1$), and hence the input represents the Z-basis state $|a\rangle$ (being either $|0\rangle$ or $|1\rangle$).

As before, the same equation but with the colours flipped also continues to hold, which gives rules for copying $|+\rangle$ and $|-\rangle$ states through X-spiders. We will refer to all these rules as **state-copy rules**. These state-copy rules only hold when the spider being copied has a phase of 0 or π. For any other phase the analogue of (3.48) does not hold, nor would we expect it to! Quantum theory famously does not allow you to *clone* arbitrary states. Note here the important quantifier of 'arbitrary'. The reason that a Z-spider can act as a cloning process for the $|0\rangle$ and $|1\rangle$ states is because they are orthogonal. It does not clone any other state.

Note that the orientation of the wires in the rewrite rules we have derived is irrelevant, as we can pre-compose and post-compose each rule with 'cups and caps' to change inputs to outputs and vice versa. For instance, for the 'reverse' of the state-copy rule:

$$
\begin{array}{ccccccc}
\alpha - a\pi & = & a\pi - \alpha & \stackrel{(3.48)}{\propto} & a\pi & = & -a\pi \\
 & & & & a\pi & & -a\pi
\end{array}
\tag{3.49}
$$

At this point the reader might wonder why we focus so much on X and Z, and not on Y. The reason for this is that Y can be presented as a composition of X and Z, and hence the rules relating to the Y-eigenbasis can be derived from the equations we have covered here. In principle, we could have worked with the 'XY'-calculus instead of the ZX-calculus, but there is an important reason to prefer Z and X over Y. Namely, the Z and X eigenbases are *self-conjugate* meaning that each of the vectors in the basis is equal to its own componentwise complex conjugate, for instance: $\overline{|0\rangle} = |0\rangle$. This is not the case for the Y eigenbasis $|\pm i\rangle := |0\rangle \pm i|1\rangle$. As a result, the 'Y-spider' does not enjoy the symmetry between inputs and outputs of (3.22) that the Z- and X-spider do.

Example 3.2.2. Knowing how the Pauli operators commute through a circuit (or a more general computation) is important for several areas in quantum computation, such as when verifying the functioning of an error correcting code, implementing a measurement-based quantum computation, or calculating the stabiliser tableau of a Clifford circuit. The ZX-calculus makes it easy to remember the rules for how the Pauli's propagate. Indeed, the only rules required are spider fusion (*sp*), π-copy rule (*sc*), the colour-change rule (*cc*) to be introduced in the next section (and occasionally an identity will need to be removed with (*id*)).

For instance, suppose we wish to know how the Pauli operator $X \otimes Z$ commutes through the following circuit:
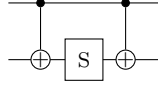
We write it as a ZX-diagram with the Pauli X and Z on the left, and start pushing them to the right:
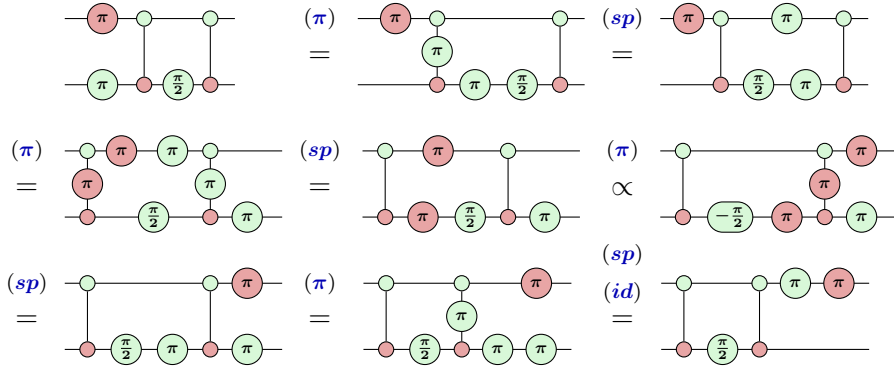
Note that we have used (**sp**) both to commute spiders of the same colour through each other (like the Pauli Z through the control of the CNOTs, or the Pauli Z through the S gate), as well as to combine the phases of the Pauli's in order to cancel out the phases.

We see that we end up with the Pauli $Y \otimes$ id. Of course, this process was quite lengthy, because every rewrite step is shown. Once you get comfortable with this process, many of the steps can be combined and it becomes easy to see where each of the Pauli's ends up.

3.2.3 Colour changing

As has been noted several times in the preceding sections, when we have derived some rule in the ZX-calculus, an analogous rule with the colours interchanged also holds. This follows from the rules concerning the Hadamard gate:

$$\tag{3.50}$$

The first of these rules states that the Hadamard gate is self-inverse, while the second says that commuting a Hadamard through a spider changes its colour (reflecting the fact that the Hadamard interchanges the eigenbasis of

the Z and X gate). Note that the Hadamard commutation rule also holds for spiders that have no inputs, so for instance:

$$\tag{3.51}$$

Let us give an example to demonstrate how the two Hadamard rules (3.50) imply the colour-inverse of any other rule we have. For instance, suppose we want to prove the colour-inverse of Eq. (3.41):

$$\tag{3.52}$$

We start with the left-hand side of the colour-swapped version, and then insert two Hadamards on the input by applying the self-inverse rule in reverse:

We then commute one of the Hadamards all the way to the outputs:

We now see the left-hand side of the original (not colour-swapped) equation (3.52), and hence we can apply it:

Now it remains to commute the Hadamards back to the left, and to cancel the resulting double Hadamard:

We have hence succeeded in proving the colour-inverse of Eq. (3.52):

Using the Hadamard commutation rule we can also prove that surrounding a spider with Hadamards changes its colour:

$$\vdots \; \alpha \; \vdots \;\; \overset{(3.50)}{=} \;\; \vdots \; \alpha \; \vdots \;\; \overset{(3.50)}{=} \;\; \vdots \; \alpha \; \vdots \tag{3.53}$$

In Exercise (2.13) we saw a circuit for building the CZ gate using a CNOT and Hadamards. We can translate this circuit to a ZX-diagram and then simplify it using the Hadamard rules:

$$\overset{(3.50)}{=} \qquad \overset{(3.50)}{=} \tag{3.54}$$

We see that the diagram we get looks symmetrically on the control and target qubit. This is because a CZ gate *is* acting symmetrically on its two qubits. I.e. we have $\text{SWAP} \circ \text{CZ} \circ \text{SWAP} = \text{CZ}$. So here the ZX-diagram directly reflects a useful property the unitary has.

Exercise 3.6. Help the trapped π phase find it's way to the exit.
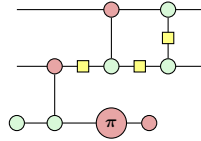
Note that it might multiply itself before the end.

3.2.4 Strong complementarity

The previous sets of rules all have a very distinct topologically intuitive character: spider fusion allows you to fuse adjacent spiders of the same colour; identity removal and the Hadamard self-inverse rule allow you to remove certain vertices; the π-copy, state-copy and colour-change rule allow you to commute certain generators through spiders by copying them to the other side. It is therefore relatively easy (after you gain some practice) to spot where they can be applied and what the effect of their application will be on the structure of the rest of the diagram.

The last major rule of the ZX-calculus takes more time to work with intuitively, although it does correspond to a natural and crucial property of the interaction of the Z- and X-spider.

3.2.4.1 Bialgebras, CNOTs and SWAPs

Before we introduce the rule, let us give some motivation. Treating the $|0\rangle$ and $|1\rangle$ states as Boolean bits, we can view the phase-free 1-input, n-output

Z-spider as the COPY gate that copies the bit 0 to $0 \cdots 0$ and the bit 1 to $1 \cdots 1$. Indeed, this is exactly what Eq. (3.48) states. Analogously, we can view the phase-free 2-input, 1-output X-spider as the XOR gate:

$$
\begin{array}{c}
a\pi \\
\quad \bullet \quad = \quad (a \oplus b)\pi \\
b\pi
\end{array}
$$

We already saw this in matrix form in Eq. (3.8), but we can also prove it using spider fusion and the fact that $2\pi \cong 0$.

The XOR gate and the COPY gate have a natural commutation relation: first XORing bits, and then copying the outcome is the same as first copying each of the bits and then XORing the resulting pairs:

$$
\boxed{XOR} - \boxed{COPY} \quad = \quad \boxed{COPY} \; \boxed{XOR} \atop \boxed{COPY} \; \boxed{XOR} \tag{3.55}
$$

This equation says that the XOR *algebra* and the COPY *coalgebra* together form a **bialgebra**. This is why the analogous equation in the ZX-calculus is often called the **bialgebra rule**:

$$
\succ\!\!\bullet\!\!-\!\!\circ\!\!\prec \quad \propto \quad \diagdown\!\!\diagup \tag{3.56}
$$

However, for reasons we will explain below, we will refer to this rule as (a special case of) **strong complementarity**. This rule is essential to many proofs in the ZX-calculus. As a demonstration of its utility, let us prove a variation on the well-known 'three CNOTs make a swap' circuit identity We will start with a circuit containing two CNOT gates, and deform it to make clear where we can apply the rule:

$$
\begin{array}{c}
\text{(diagram)}
\end{array}
\quad = \quad
\begin{array}{c}
\text{(diagram)}
\end{array}
\tag{3.57}
$$

We then see that we can apply Eq. (3.56) in reverse, and then deform the diagram again to bring it into a simpler form:

$$
\begin{array}{c}
\text{(diagram)}
\end{array}
\quad \overset{(3.56)}{\propto} \quad
\begin{array}{c}
\text{(diagram)}
\end{array}
\quad = \quad
\begin{array}{c}
\text{(diagram)}
\end{array}
\tag{3.58}
$$

We could have done all this without the diagram deformation, as the rewrite rules apply regardless of the orientation of the inputs and outputs.

The circuit equality is then given in a single step by an application of Eq. (3.56):

$$\text{(circuit diagram)} \quad \overset{(3.56)}{\propto} \quad \text{(swap diagram)} \tag{3.59}$$

We can see this as another interpretation of strong complementarity: it relates CNOT gates to the SWAP gate.

3.2.4.2 Complementarity

We called Eq. (3.59) a version of 'three CNOTs make a swap', but it is of course not the real deal. Let's actually try to prove the *real* 'three CNOTs make a swap' rule:

$$\text{(diagram)} \quad \overset{(3.59)}{\propto} \quad \text{(diagram)} \quad = \quad \text{(diagram)} \tag{3.60}$$

Here in the last step we simply pushed the spiders of the first CNOT through the wires of the SWAP. To finish the proof it then remains to show that two CNOTs applied in succession cancel each other:

$$\text{(diagram)} \quad = \quad \text{(diagram)} \tag{3.61}$$

The first step towards doing this might seem clear, namely, we fuse the adjacent spiders of the same colour:

$$\text{(diagram)} \quad \overset{(3.34)}{=} \quad \text{(diagram)} \tag{3.62}$$

But now we are seemingly stuck.

The solution comes from another equation between the Boolean gates XOR and COPY. When we first apply an XOR and then a COPY, we can commute the XOR by essentially 'copying' it through the COPY gate. But what happens when we first apply a COPY *and then* an XOR gate to the two outputs of the COPY:

$$|0\rangle \overset{\text{COPY}}{\mapsto} |00\rangle \overset{\text{XOR}}{\mapsto} |0\rangle$$
$$|1\rangle \overset{\text{COPY}}{\mapsto} |11\rangle \overset{\text{XOR}}{\mapsto} |0\rangle$$

We see that regardless of the input, we output a 0. Generalising the notion of a XOR gate to allow for a 0-input XOR gate that just outputs its unit

(namely 0), and a 0-output COPY gate that copies its output zero times (and hence discards it) we can write this relation diagrammatically as:

$$\text{—}\boxed{\text{COPY}}\boxed{\text{XOR}}\text{—} \quad = \quad \text{—}\boxed{\text{COPY}} \quad \boxed{\text{XOR}}\text{—} \tag{3.63}$$

An algebra (XOR) and a coalgebra (COPY) satisfying this equation are known together as a **Hopf algebra**. Which is why the following analogous equation between the Z- and X-spider is often called the **Hopf rule**:

$$\text{—}\!\!\bigcirc\!\!\bigcirc\!\!\bullet\!\!\text{—} \quad \propto \quad \text{—}\!\!\bigcirc \quad \bullet\!\!\text{—} \tag{3.64}$$

However, we will refer to this rewrite rule as the **complementarity** rule, because this rule can also be seen as a consequence of the *Z*-basis and *X*-basis defining complementary measurements: having maximal information about the outcome of a *Z* measurement on a state $|\psi\rangle$ (namely, we will obtain precisely 1 outcome with probability 1) entails minimal information about the outcome of an *X* measurement (namely, we will obtain any outcome with equal probability). Mathematically, this corresponds to the *Z* and *X* bases being **mutually unbiased**:

$$\forall |x_i\rangle \in \mathcal{X}, |z_j\rangle \in \mathcal{Z} \, . \, |\langle x_i | z_j \rangle|^2 = \frac{1}{2} \tag{3.65}$$

It turns out that complementarity can be derived from strong complementarity using some clever deformation of the diagram and the rules from the previous sections:

$$\text{(3.66)}$$

In this derivation, we first deformed the diagram, then we introduced identities, we unfused some spiders, applied the strong complementarity rule, copied a state twice, and in the last equation we removed the dangling scalar diagram as it just introduces some scalar factor.

The complementarity rule is exceedingly useful. Indeed, combining it with some spider fusion we can use it to show that we can always cancel pairs of

connections between spiders of opposite colours:

$$(3.67)$$

So if two spiders of opposite colour are connected by n wires, then this can be reduced to $n \bmod 2$ wires. In particular, we can finish the proof that 2 CNOTs cancel each other out, and thus that 3 CNOTs make a SWAP:

$$(3.68)$$

Exercise 3.7. We proved Eq. (3.60) by applying strong complementarity in the reverse direction. Show that you can also prove it by using it in the forward direction (for instance by applying Eq. (3.56) to the top-left Z- and X-spiders). You will also need to use spider fusion and complementarity to finish the proof.

3.2.4.3   *General strong complementarity*

We can extend equation (3.56) to spiders of arbitrary arity to obtain the full **strong complementarity** rule:

$$(3.69)$$

That is, for any two connected phase-free spiders of different colours we can apply strong complementarity, resulting in a fully connected bipartite graph as on the right-hand side. For $n = m = 2$ this is exactly Eq. (3.56). For $n = 1$ or $m = 1$ this follows in a trivial manner by adding and removing identities (cf. Eq. (3.38)). When $n = 0$ or $m = 0$ this is just the state-copy rule, i.e. (the colour-swapped) Eq. (3.46). Now suppose $n = 2$ and $m = 3$, we will show how to derive its strong complementarity rule using just spider

fusion and the $n = 2, m = 2$ version of the rule:

$$\tag{3.70}$$

$$\tag{3.70}$$

Exercise 3.8. Prove the remaining cases of bialgebra for $n, m > 2$. Hint: first use induction on m, and then on n.

There are two common mistakes people make when using the strong complementarity rule. The first is that the rule (3.56) only works when the phases on the spiders are zero. When the phases are π a modification is possible by combining it with the π-copy rules (3.44):

$$\tag{3.71}$$

Exercise 3.9. Prove Eq. (3.71) using the rules of the ZX-calculus we have seen so far.

But when one of the spiders contains some arbitrary phase α, the result will be more complicated:

$$\tag{3.72}$$

The additional bit of diagram we get is called a *phase gadget*. We will have a lot more to say about phase gadgets in Chapter 7.

The second common mistake is that people apply the rule (3.56) from right-to-left without paying attention to how many outputs the spiders have, wrongfully equating the following diagrams:

$$\tag{3.73}$$

The correct way to apply strong complementarity here is to first unfuse the spider:

$$\tag{3.74}$$

3.2.4.4 Strong complementarity as spider pushing

Because strong complementarity is one of the most useful rewrite rules, but also the hardest one to see when to use, we want to share one more perspective on what it means topologically. Recall that we can push a π phase of the opposite colour through a spider:

$$\tag{3.75}$$

Note that this is actually two equations, since it holds for both $a = 0$ and $a = 1$. Let's see if we can't compress this into just a single equation. To do this, we will unfuse the $a\pi$ phases and reduce them to just a single instance, by applying the copy rule (3.48) in reverse:

$$\tag{sp} = \qquad \tag{3.48} \propto$$

Now let's compare this again to the left-hand side of Eq. (3.75), where we also unfuse the $a\pi$ phase onto its own spider:

$$\propto \tag{3.76}$$

Now we have an equation which holds for $a = 0$ and $a = 1$, where the phase $a\pi$ occurs in the same position: as an input to the first qubit. We can then see this as an equation of linear maps where we input a specific state into the top input. But these states are respectively $|0\rangle$ and $|1\rangle$ (up to a scalar factor). This means that this equation holds for a basis of the qubit! But then the linear maps themselves need to be equal regardless of the state that we input. We hence get:

$$\propto \tag{3.77}$$

But we recognise this as an instance of strong complementarity! This then gives us another perspective on strong complementarity: it is a 'controlled' version of the π pushing we saw before, where we don't have to specify what the phase is we want to push. We instead just leave the wire open.

Remark 3.2.3. We need to be a bit careful with this reasoning we used here, because this is actually one of the areas where the scalar factors *do* matter. We used the fact that if $M_1|\psi_j\rangle = M_2|\psi_j\rangle$ for a basis $\{|\psi_j\rangle\}_j$ and linear maps M_1 and M_2, that then $M_1 = M_2$. But this only works if we have equality on the nose for all j. If we just had $M_1|\psi_j\rangle \propto M_2|\psi_j\rangle$ so that the scalar factor could be different for every j, then we wouldn't get the equality $M_1 = M_2$, or even $M_1 \propto M_2$. To demonstrate this, let's consider $M_1 = \mathrm{id}$ and $M_2 = S = \mathrm{diag}(1, i)$. Then $M_1|0\rangle = |0\rangle = M_2|0\rangle$, and $M_1|1\rangle = |1\rangle \propto i|1\rangle = M_2|1\rangle$, but of course id is not proportional to S. The reason we were still warranted in making the conclusion we did based on Eq. (3.76), is because here the proportionality constant happens to be the same for both $a = 0$ and $a = 1$.

Using this realisation of 'strong complementarity as controlled π pushing', we can give a nice visual intuition to what happens when you apply strong complementarity:

$$\tag{3.78}$$

Here we added an additional Z-spider the X-spider is connected to at the start to make it look a little more intuitive. We see then that in the same way that for π pushing, the π appears on all the other legs of the spider, with strong complementarity the X-spider also appears on all the other legs of the Z-spider.

Exercise 3.10. Prove the following circuit identity by translating them to ZX-diagrams and rewriting the left-hand side into the right-hand side.

Exercise 3.11. Consider the following parametrised unitary:

a) Using the ZX-calculus show that $G(\beta) \circ G(\alpha) = G(\alpha + \beta)$ and that $G(0) = \mathrm{id}$. Note that this equation in particular implies that $G(\alpha)^\dagger = G(-\alpha)$.

b) Using the ZX-calculus show that:

$$\tag{3.79}$$

c) This previous equation implies that:

$$\boxed{G(\alpha)} \quad = \quad \times\boxed{G(\alpha)}\times \tag{3.80}$$

Show that this indeed holds by calculating the matrix of both sides.

d) Show that Eq. (3.80) holds by using the ZX-calculus to reduce the left-hand side to a diagram that is clearly symmetric under interchanging the two qubits.

3.2.5 Euler decomposition

The final rule of the ZX-calculus we need to introduce is a way to write the Hadamard in terms of spiders. Recall that any single-qubit unitary gate is equal (up to global phase) to a Z-rotation followed by an X-rotation, followed once again by a Z-rotation:

$$\boxed{U} \quad = \quad \alpha-\beta-\gamma$$

This is then of course also true for the Hadamard. In particular for $\alpha = \beta = \gamma = \frac{\pi}{2}$:

$$\text{—}\square\text{—} \quad = \quad e^{-i\frac{\pi}{4}} \; \frac{\pi}{2}-\frac{\pi}{2}-\frac{\pi}{2} \tag{3.81}$$

Here we have included the correct global phase for good measure.

This Euler decomposition is just one possible way to write the Hadamard in terms of spiders. There is in fact an entire family of representations that will also be useful to note:

$$\begin{aligned}
\text{—}\square\text{—} &= e^{-i\frac{\pi}{4}} \; \tfrac{\pi}{2}-\tfrac{\pi}{2}-\tfrac{\pi}{2} \; = \; e^{i\frac{\pi}{4}} \; \text{-}\tfrac{\pi}{2}-\text{-}\tfrac{\pi}{2}-\text{-}\tfrac{\pi}{2} \; = \; \tfrac{\pi}{2}-\bullet-\tfrac{\pi}{2} \;\; {}^{\tfrac{-\pi}{2}} \\
&= e^{-i\frac{\pi}{4}} \; \tfrac{\pi}{2}-\tfrac{\pi}{2}-\tfrac{\pi}{2} \; = \; e^{i\frac{\pi}{4}} \; \text{-}\tfrac{\pi}{2}-\text{-}\tfrac{\pi}{2}-\text{-}\tfrac{\pi}{2} \; = \; \tfrac{\pi}{2}-\bullet-\tfrac{\pi}{2} \;\; {}_{\tfrac{\pi}{2}}
\end{aligned} \tag{3.82}$$

Exercise 3.12. Prove that all the equations of (3.82) hold in the ZX-calculus, by using Eq. (3.81) and the other rewrite rules of the ZX-calculus we have seen so far.

By including the rule (3.81) it looks like we are creating an asymmetry in the phases in the rules of Figure 3.1 with special significance being given to $\frac{\pi}{2}$ over $-\frac{\pi}{2}$. However, (3.82) shows that the situation is still fully symmetric. In particular, all the rules still hold when we flip the value of all phases from α to $-\alpha$. In particular we have the following 'meta-rule'.

Proposition 3.2.4. An equation in the ZX-calculus between diagrams D_1 and D_2 can also be proven when we flip all the phases of spiders in the diagrams of D_1 and D_2.

Such 'flipping of the phases' corresponds to taking the complex conjugate of the linear map the diagram represents.

Exercise 3.13. The rules presented in Figure 3.1 are actually not all necessary. Show that we can prove (*hh*) using Eq. (3.81) and the other rewrite rules of Figure 3.1.

Exercise 3.14. Prove using the ZX-calculus that we can get rid of Hadamard self-loops at the cost of acquiring a phase:

$$\tag{3.83}$$

Exercise 3.15. There are two ways in which we can write the eigenstate $|i\rangle := \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$ of the Pauli operator Y (up to global phase). Namely, as $|i\rangle = R_X(-\frac{\pi}{2})|0\rangle$ or as $|i\rangle = R_Z(\frac{\pi}{2})|+\rangle$. Prove this equivalence using the ZX-calculus:

$$\tag{3.84}$$

An analogous equation holds for $|-i\rangle := \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$, which boils down to flipping the signs of the phases in the spiders.

3.3   ZX in action

Now that we have seen all the rules of the ZX-calculus, let's see what we can do with them! In the previous section we have already seen a couple of small use-cases: simplifying a circuit to verify that it implements the correct state in Example 3.2.1, or pushing π phases through a circuit to see where it maps Pauli gates in Example 3.2.2. Here we will give several more small examples that demonstrate the use of the ZX-calculus in reasoning about quantum computing. This will also demonstrate several best practices when trying to simplify a diagram using the ZX-calculus.

3.3.1   Magic state injection

The following circuit implements the well-known *magic state injection* protocol, where a $|T\rangle := |0\rangle + e^{i\pi/4}|1\rangle$ magic state is pushed onto a qubit using

a measurement and potential correction:

$$\text{(3.85)}$$

Here the double wire represents a classical measurement outcome being fed forward into the S gate, so that the S gate is only applied if the measurement outcome corresponded to $\langle 1|$.

We can represent this in the ZX-calculus by introducing a Boolean variable a to represent the measurement outcome. We can then easily prove its correctness:

$$\text{(3.86)}$$

Most of this proof is spider (un)fusion, but there are a couple of interesting bits. The step labelled (π) actually consists of two branches, because if $a = 0$, then the X-spider can be removed using (*id*) so that the phase on the Z-spider remains $\frac{\pi}{4}$, while if $a = 1$, then (π) is indeed applied, and the phase flips to $-\frac{\pi}{4}$. Combining these two branches we see that the phase becomes $(-1)^a \frac{\pi}{4}$. In the step after that we make the observation that $(-1)^a \frac{\pi}{4} = \frac{\pi}{4} - a\frac{\pi}{2}$.

The usage of a variable to denote a measurement outcome is an easy way to deal with circuits that contain measurements and classically-controlled gates. While it is easy to use, it does have the drawback that it is less clear in which direction the information is 'flowing'. Here $a\pi$ is the result of a measurement outcome that we don't have control over, while $a\frac{\pi}{2}$ denotes a $\frac{\pi}{2}$ phase gate that we only apply when we saw the measurement outcome $a = 1$. For more complicated scenario's where we *do* care about showing this causality more clearly we could adopt a variant of the ZX-calculus where we allow both classical and quantum wires to exist. We will however not need this additional machinery in this book.

3.3.2 Teleportation

The standard state-teleportation protocol consists of two parties, Alice and Bob, that share a maximally entangled state $|00\rangle + |11\rangle$. Alice does some quantum operations, measures her states, sends the classical measurement outcomes to Bob, and Bob does some quantum corrections based on those outcomes. At the end Bob has the state that Alice started out with. We can represent this as a quantum circuit as follows:
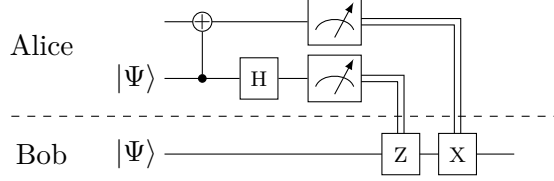
Here the $|\Psi\rangle$ label is to denote that the bottom two qubits start in the maximally entangled Bell state. By representing the measurement outcomes by Boolean variables a and b we can again represent this in the ZX-calculus and prove the correctness of the protocol:
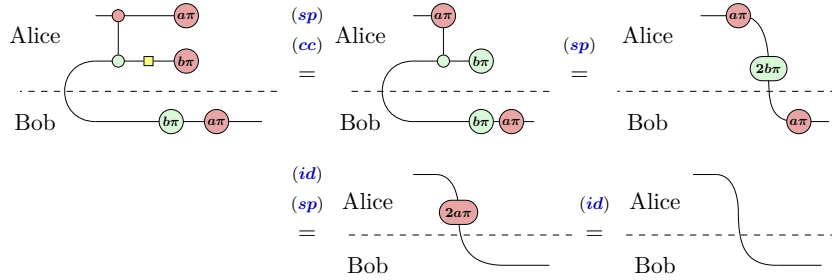
Here we used the fact that $2b\pi$ is either 2π or 0 which are both 0 modulo 2π, and hence the spider can be removed using (**id**).

We see that we end up with a wire going directly from Alice and Bob. Hence, whatever state Alice puts in her qubit wire will end up at Bob.

3.3.3   Detecting entanglement

Let us now do a more involved calculation with the ZX-calculus, demonstrating how one goes about systematically simplifying a diagram.

Suppose you are given the following (somewhat randomly chosen) quantum circuit:

$$(3.87)$$
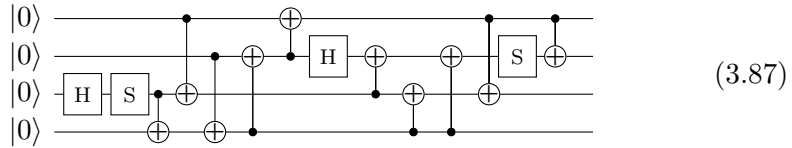
Suppose further that you wish to know which qubits are entangled after applying this circuit. Calculating the state directly gives

$$\frac{1}{2}(1,0,0,0,i,0,0,0,0,0,0,1,0,0,0,i)^T.$$

While a trained eye might be able to spot that the 2nd qubit is unentangled, and that the remainder form a GHZ-state, at first glance this is altogether not obvious. Instead, let's simplify this same circuit in the ZX-calculus and

see where this gets us. In this simplification we have a step marked (*) which will be explained after.
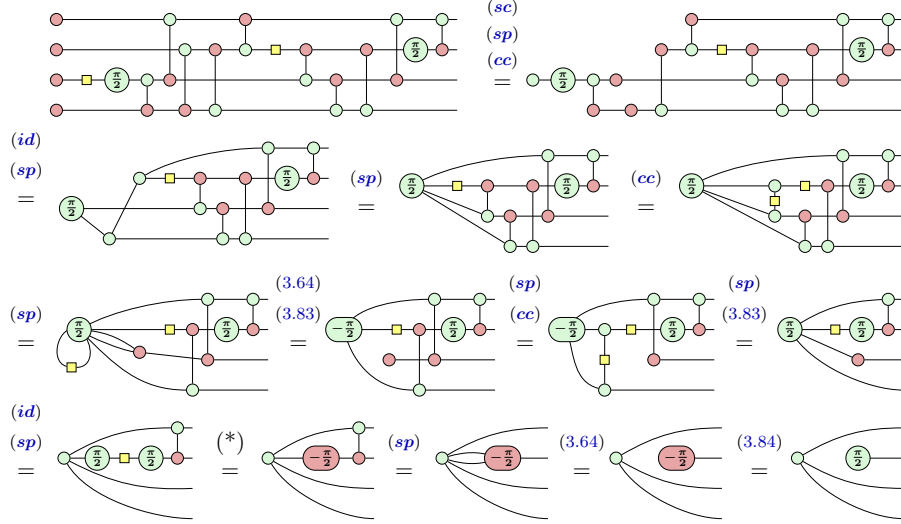
If you don't stare at diagrams for a living like we do, it might not necessarily be clear what is happening here, but we are in fact following a simple algorithm. Most of the steps follow the general pattern that we always try to fuse spiders of the same colours with (**sp**) (though we don't do that here for all spiders at the same time to prevent cluttering), always remove identities with (**id**), always copy states through spiders with (**sc**), and always remove parallel wires between spiders of opposite colours using the Hopf rule (3.64). Hadamards are moved out of the way with (**cc**) when no other obvious move is available. Additionally we have used the rewrites of Eq. (3.83) and (3.84). We have marked one rewrite by (*), this is just an application of the Euler decomposition rule:

This equation is an example of how a series of *Clifford* phase gates can often be combined together into a simpler series of phase gates. We will have a lot more to say about Clifford gates in Chapter 5.

The size of the circuit (3.87) is nearing the limit of what is still comfortable to rewrite manually. In fact, we used some software to verify and help with the rewriting here: the algorithm we just described is easily implemented, so that the entire simplification can be automated. We presented the derivation here in detail to show how one would go about systematically simplifying

a ZX-diagram. We will look at more detailed approaches to simplifying diagrams in other chapters.

Exercise 3.16. Write the following circuit with input state as a ZX-diagram and simplify it to figure out which qubits are entangled to each other.
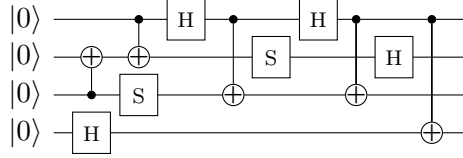
3.3.4   The Bernstein-Vazirani algorithm

Back in Section 2.4, we introduced the **Bernstein-Vazirani algorithm**, as a simple example where a quantum computer gets a modest advantage over a classical computer. Namely, one can extract a secret piece of information from an unknown function f on n bits using just a single quantum query to the function (i.e. one use of the unitary oracle U_f), rather than n classical queries.

Here is the problem, stated formally:

Given: a function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ such that there exists a fixed bitstring $\vec{s}$ such that $f(\vec{x}) = \vec{s} \cdot \vec{x}$.

Find: $\vec{s} \in \mathbb{F}_2^n$.

We saw that the quantum algorithm for solving this problem is very simple. If we simply perform the following circuit:
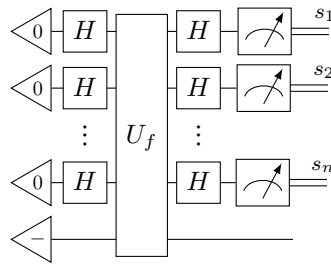
we obtain the bitstring $\vec{s}$ as the measurement outcomes.

We gave a proof using bra-ket calculations in Section 2.4. We'll now give a different proof using the ZX calculus. This follows almost immediately from finding a nice diagram to capture the oracle U_f. We claim that U_f can be

written as follows:

$$U_f \ := \ \ \text{(3.88)}$$

where the labels on the wires mean there is an edge connecting the Z spider on the i-th qubit to the one X spider if and only if $s_i = 1$. This works because, as we saw in Section 3.1.1, the X dot acts like an XOR on the computational basis. Hence, if we take the XOR of all the bits x_i where $s_i = 1$, we get the dot product $\vec{s} \cdot \vec{x}$. If we additionally XOR in some other bit y, we get $(\vec{s} \cdot \vec{x}) \oplus y = f(\vec{x}) \oplus y$.

Since the Z dot copies computational basis elements, (3.88) does exactly what the Bernstein-Vazirani oracle should do:

Now, let's see what happens if we plug $|+\rangle = H|0\rangle$ into the first n qubits and $|-\rangle$ into the last qubit. In Section 3.1.1, we saw that:

$$|+\rangle \ \propto \ \circ\!\!-\!\!- \qquad\qquad |-\rangle \ \propto \ (\pi)\!\!-\!\!-$$

Plugging in, we get:

In the first step, the π-labelled Z spider is copied through the X spider. This will give a π-labelled Z spider on the last output and anywhere there is a wire to one of the other Z spiders, i.e. whereever $s_i = 1$. If we fuse these in, we see that we get a Z gate where $s_i = 1$ and identity where $s_i = 0$. We can capture this succinctly in the middle picture above by writing this as the Z-phase gate $Z[s_i\pi]$.

After the final spider fusion, we can see the state on the first n-qubits is an element of the X basis. Hence, if we measure in the X basis (i.e.

perform Hadamards then measure in the Z basis), we will obtain the outcome $(s_1, \ldots, s_n)$ with certainty. This completes the correctness proof for the Bernstein-Vazirani algorithm.

3.4 Extracting circuits from ZX-diagrams

We have already seen that ZX-diagrams are a strict superset of circuit notation. That is, if we have a quantum circuit, it is straightforward to translate it into a ZX-diagram. But what about the converse? How do we translate a ZX-diagram back into a circuit? The first thing to note is that ZX-diagrams can represent more types of maps, that aren't necessarily unitaries or isometries, and hence in general there is no way to turn an *arbitrary* diagram back into a circuit. But what if you know somehow that the diagram implements a unitary, how do we get a circuit back out? It turns out this is hard enough that we should state it is a proper **Problem**:

Problem 3.4.1 (Circuit Extraction)**.** Given a ZX-diagram D with designated inputs and outputs, which describes a unitary map. Produce a circuit whose gate-count is polynomial in the number of spiders in D.

It turns out that Circuit Extraction is a "Hard with a capital H" problem in the complexity-theory sense, which you will be asked to prove in Exercise* 7.7. Nevertheless, if we impose some restrictions on a ZX diagram, the problem becomes much easier. In this section we will show how diagrams we call *circuit-like* can be made back into circuits.

3.4.1 ZX-diagrams to circuits with post-selection

First, it's worth noting that obtaining a circuit with post-selection is relatively straightforward. A generic ZX-diagram can be constructed using the following 5 things:

We can see this by transforming each X-spider into a Z-spider using (**cc**), and then unfusing spiders until they all have at most 3 legs.

We can interpret each of these generators as circuit fragments:
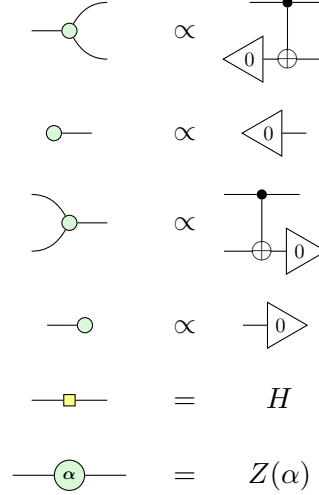
where the first two require ancilla qubits in the state $|0\rangle$ and the second two require post-selection onto $\langle 0|$. With this interpretation, there is an evident translation of any ZX-diagram into a post-selected circuit. First 'unfuse' the spiders into generators, then interpret them as post-selected circuits, e.g.
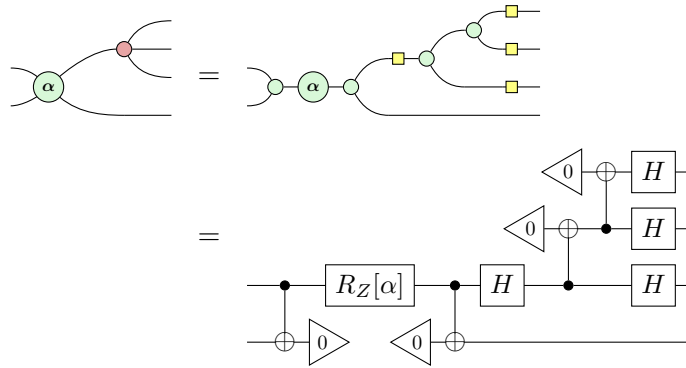
It could be that we are happy with this. Indeed any post-selected circuit corresponds to a particular sequence of measurement outcomes. And, if we start with a ZX-diagram that is not equal to zero, the post-selected circuit will also be non-zero, so it corresponds to a computation that is realisable with non-zero probability.

However! The probability of getting the diagram we wanted will, in general, becomes exponentially small with the number of post-selections. So that kindof sucks. What would be much better is to turn a ZX-diagram into a computation that can be run *deterministically* (or at least with high probability) on a quantum computer.

3.4.2 Circuit-like diagrams and optimisation

One of the simplest kinds of ZX diagrams that are easy to extract are the **circuit-like** diagrams.

Definition 3.4.2. A ZX diagram is called **circuit-like** if it can be decomposed into a circuit consisting only of CZ, H, and $R_Z[\alpha]$ gates just by applying the colour change rule (**cc**) and unfusing spiders, i.e. applying the rule (**sp**) from right-to-left.

While this seems to be pretty much building the property we want directly into the definition, there is (essentially) an equivalent formulation for circuit-like diagrams which is called **ZX diagrams with causal flow**. We will meet these in Chapter 8 (specifically Section 8.2.4), along with several generalisations that are relevant to circuit optimisation and measurement-based quantum computing.

Even though the circuit-like ZX diagrams are a very limited class of diagrams, we can already define a (pretty effective!) quantum circuit optimisation algorithm that never leaves the realm of circuit-like diagrams.

Algorithm 3.4.3. Circuit optimisation using circuit-like ZX diagrams

- **Given:** A circuit consisting of CNOT, CZ, H, $Z(\alpha)$, and $X(\alpha)$ gates.
- **Output:** An optimised circuit.

1. Replace every CNOT and $X(\alpha)$ gate by its decomposition into CZ, H and $Z(\alpha)$, or equivalently use (**cc**) to turn all X spiders into Z spiders.
2. Apply the rules (**sp**), (**id**), (**hh**), and the following derived rule:

$$\vdots \; \alpha \; \text{---} \; \beta \; \vdots \quad \propto \quad \vdots \; \alpha \quad \beta \; \vdots \tag{3.89}$$

   from left-to-right as much as possible.
3. Extract the circuit from the resulting circuit-like ZX diagram by unfusing spiders.

There are a few things that we should show before we are happy with Algorithm 3.4.3. First, we can see that (3.89) is indeed a derived rule. Since this is essentially just an "all-Z" variation on the complementarity rule we met in Section 3.2.4, this is straightforward enough to be an exercise for now.

Exercise 3.17. Show that (3.89) follows from the rules of the ZX calculus.

The second thing we would like to know is that we indeed have a circuit-like ZX diagram by the time we get to step 3. The final thing we need to

know is *which* spiders to unfuse to turn the circuit-like diagram into a circuit. Let's look at these properties in turn.

Proposition 3.4.4. Left-to-right applications of (**sp**), (**id**), (**hh**), and (3.89) preserve the property of being circuit-like.

Proof (**sp**) preserves circuit-like diagrams by definition. Suppose we can apply the rule (**id**) to a circuit-like ZX diagram. That means that there must be a 2-legged Z spider with phase 0. The only way this can arise from spider fusions of a circuit is if the circuit contains a $Z(0) = \text{id}$ gate. Deleting this trivial gate from the circuit results in a new circuit, which can be obtained by unfusing spiders after applying the (**id**) rule.

We can reason similarly about the rules (**hh**) and (3.89). The only way the left-hand sides of these rules can arise from spider fusions of a circuit is if the circuit contains a pair of cancelling H gates or cancelling CZ gates, respectively:

Again, deleting these gates from the circuit will result in a new circuit, which we can obtain by unfusing spiders after applying the (**hh**) and (3.89) rules, respectively. $\square$

Now, let's see how to actually perform the spider-unfusions necessary to extract a circuit from a circuit-like diagram. It turns out we can actually just do this greedily. Imagine a *frontier* between the part of the diagram that has been decomposed into CZ, H, and $Z(\alpha)$ gates. We initially start with the frontier on the outputs, and move the frontier toward the inputs, performing spider-unfusions as necessary.

For this, we consider three cases:

1. If an H gate is on the frontier, move it to the circuit part of the diagram.
2. If a 2-legged Z spider is to the left of the frontier, it can only be an $R_Z[\alpha]$ gate, so move it to the circuit part.
3. If two spiders on the frontier are connected via a Hadamard gate, perform spider-unfusion as necessary to produce a CZ gate, and move it to the circuit part.

For circuit-like diagrams, this will never get stuck, and terminates when there are no spiders or Hadamards to the left of the frontier. Proving this is much easier when we have some extra tools in our tool belt (namely the aforementioned notion of **causal flow**, which we introduce in Chapter 8), so we will postpone this proof until then. However, you should know enough at

this point to code up a full-fledged circuit optimisation algorithm using the ZX calculus. You can see for yourself how well it works!

3.5   Summary: What to remember

1. Spiders are a class of linear maps that can have any number of inputs and outputs. They come in two flavours: Z-spiders and X-spiders.
2. Spiders can be connected together to form ZX-diagrams. Stacking ZX-diagrams correspond to the tensor product of linear maps, and connecting ZX-diagrams corresponds to composition.
3. ZX-diagrams can be arbitrarily deformed as long as the order of inputs and outputs is preserved.
4. ZX-diagrams are universal, meaning that they can represent any linear map between any number of qubits.
5. In particular, any quantum circuit consisting of Z- and X-phase gates and CNOTs can be easily written as a ZX-diagram. Conversely, any ZX-diagram can be written as a post-selected quantum circuit.
6. The ZX-calculus is a set of graphical rewrite rules for transforming ZX-diagrams while preserving the linear map they represent.
7. There are 7 rules of the ZX-calculus (see Figure 3.1): spider fusion, colour change, π-copy, strong complementarity, identity, Hadamard-Hadamard cancellation, and Euler decomposition of the Hadamard.
8. We can code up a pretty simple yet effective circuit simplification routine by converting all X-spiders to Z-spiders, and applying spider fusions, Hadamard-Hadamard cancellations and Hopf rules (Eq. (3.89)) wherever possible, and then *extracting* a circuit from this diagram.

3.6   Advanced Material*

3.6.1   Formal rewriting and soundness*

While many of you might be familiar with mathematical rewriting of algebraic expressions, you might be unfamiliar with doing the same with diagrams, and might even be sceptical that what we are doing is even valid mathematics. To ease those worries, we will say a bit more about what is actually going on formally when we rewrite a diagram.

The rewrite rules we have given in this chapter are of the form $D_1 = D_2$, where D_1 and D_2 are some ZX-diagrams with the same number of inputs and outputs. We then used these rewrite rules on *larger* diagrams that contain either D_1 or D_2 as subdiagrams. This is because when we are asserting

$D_1 = D_2$, we are actually asserting an entire family of equalities. We are for instance also asserting that $D_1 \otimes E = D_2 \otimes E$ for any other ZX-diagram E, and $D \circ D_1 = D \circ D_2$ for any composable ZX-diagram D.

In a sense we are assuming we have the following implication:

$$(3.90)$$

In words: if D_1 appears as a subdiagram of some larger 'ambient' diagram D', then D_1 can be replaced by D_2 inside of D'. We are saying we can apply the rewrite rule regardless of the context.

An important question to settle here is whether this is **sound**. A rewrite rule in a language is called sound when it preserves the underlying *semantics*, i.e. the underlying meaning. In the setting of ZX-diagrams, the language is the set of diagrammatic rewrite rules of the ZX-calculus, and the semantics of a ZX-diagram is the linear map it represents. So a rule in the ZX-calculus is sound when the diagrams on each side of the equation represent the same matrix. For a single equation like $D_1 = D_2$ soundness is easy to check: just calculate the matrix on both sides. But how can we be sure that employing this rewrite rule in a larger context as in (3.90) is still sound when we would have to check an infinite family of diagrams?

To make the following discussion more clear, we will denote the matrix of a ZX-diagram D by $[\![D]\!]$. We can then say a diagrammatic rule $D_1 = D_2$ is sound when $[\![D_1]\!] = [\![D_2]\!]$.

By definition of how we calculate the matrix of a ZX-diagram, we have $[\![D_1 \otimes D_2]\!] = [\![D_1]\!] \otimes [\![D_2]\!]$ and $[\![D_1 \circ D_2]\!] = [\![D_1]\!] \circ [\![D_2]\!]$, where this last composition is just matrix multiplication. Hence, if we have $[\![D_1]\!] = [\![D_2]\!]$, then we also have $[\![D_1 \otimes E]\!] = [\![D_1]\!] \otimes [\![E]\!] = [\![D_2]\!] \otimes [\![E]\!] = [\![D_2 \otimes E]\!]$, and hence $D_1 \otimes E = D_2 \otimes E$ is also sound. Similarly we have $[\![D_1 \circ E]\!] = [\![D_2 \circ E]\!]$. Since we build any ZX-diagram by iteratively composing and tensoring, this shows that the rewrite rule stays sound regardless of the context it appears in, and hence that the implication in (3.90) is indeed true.

Even given this discussion, some of you might still not be happy about this. How can all of this be rigorous mathematics when it is just pictures on a page? Since this is a quantum computing book and not a mathematics book, we won't give the actual proof that all of this is solid here, and instead we will give a Proof by Fancy Words and point the interested reader to the References of this chapter.

So here it comes: the right formal language to think about the ZX-calculus is *category theory*. In a *symmetric monoidal* category we can compose *morphisms* via both horizontal and vertical composition. The *coherence theorem* then tells us that the *coherence isomorphisms* of a symmetric monoidal category are enough to prove we can move morphisms around on a page while preserving equality. Because ZX-diagrams have cups and caps they are actually an example of a *compact closed category*, for which an even stronger coherence theorem is true, so that we can move the generators around the cups and caps as well. The interpretation into linear maps is a *strong monoidal functor* from ZX-diagrams to the category of complex vector spaces. The rewrite rules can be understood as an equivalence relation which respect composition and tensor product, so that it induces a *quotient category* of ZX-diagrams modulo equality by rewrite rule. Soundness of the rewrite rules means that the functor into linear maps restricts to this quotient category.

3.6.2   Dealing with scalars*

We have been ignoring non-zero scalar factors in ZX-diagrams in this chapter, and will continue to do so for most of the rest of the book. This is not however because the ZX-calculus is incapable of reasoning about scalars. The opposite in fact: because it is a straightforward exercise to deal with the scalar factors once you know a couple of tricks, it just isn't worth the effort of including them most of the time.

In this section we will see what some of these tricks are.

So first, the rule set of Figure 3.1 is not taking into account the actual scalar factors. Instead, we should use a rule set where we do have all the correct scalars. See Figure 3.2.

Most of these rules should look familiar, but there are some perhaps weird-looking things going on. Let's go through the differences rule by rule. The spider-fusion rule and the colour-change rule are unchanged. We have split up the strong complementarity rule into two parts, a bialgebra rule and a state-copy rule. This is necessary because the scalar factor for strong complementarity depends on the number of inputs and outputs of the diagram. These scalar subdiagrams included in the rules should make more sense once you recall what they are equal to:

$$
\begin{array}{ll}
\bigcirc \;\; = \;\; 2 & \bullet\!-\!\alpha = \sqrt{2} \\[4pt]
\pi \;\; = \;\; 0 & \pi\!-\!\alpha = \sqrt{2}e^{i\alpha} \\[4pt]
\alpha \;\; = \;\; 1 + e^{i\alpha} & = \tfrac{1}{\sqrt{2}}
\end{array}
$$
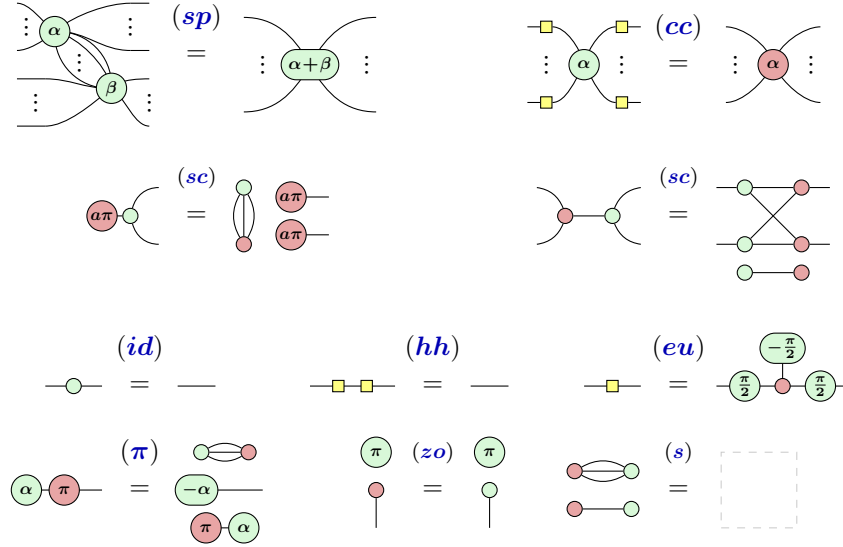
Figure 3.2 A complete and scalar-accurate set of rewrites rules for the Clifford fragment of the ZX-calculus. These rules hold for all $\alpha, \beta \in R$

The identity-removal rule and the Hadamard cancellation rule are still the same, but we now have chosen for a different Euler decomposition of the Hadamard that does not need a global phase to be accurate. We have replaced the pi-commutation rule for a more minimal one that has now acquired a scalar factor (which corresponds to a global phase of $e^{i\alpha}$). Finally, there are two new rewrite rules. The first rewrite rule ($\boldsymbol{zo}$) tells us that in the presence of a zero scalar, essentially anything equals each other. For instance, as we will see later, we can use it to show that the identity wire disconnects. The second rule ($\boldsymbol{s}$) allows us to cancel some scalars. This essentially implements $\sqrt{2} \cdot 1/\sqrt{2} = 1$. Note that the right-hand side is the empty diagram. This rule is in fact necessary: there is no other rewrite rule that can produce an empty diagram, so the equation of ($\boldsymbol{s}$) can not be proven by any combination of the other rewrite rules. It can also be shown that ($\boldsymbol{zo}$) is necessary using a more complicated argument.

Now in this section we will show two things we can do with scalars. First, we will show that when there is a zero scalar π that the entire diagram can be simplified, so that every two diagrams with this scalar in it are the same. Second, we will show that certain scalar diagrams can be 'multiplied' together in exactly the way we would expect for their corresponding scalar values.

So let's prove some things about scalar diagrams. In this section, whenever

we refer to a rewrite rule like (**sc**), we are meaning the rewrite rules from Figure 3.2, not Figure 3.1. Note that due to (**cc**) and (**hh**) that all the rules also hold with the colours interchanged as usual.

Lemma 3.6.1. Twice $\sqrt{2}$ equals 2:

Proof

Now we can show that any two diagrams that contain a zero scalar can be rewritten into each other by reducing them a unique normal form. This will require a couple of lemmas.

Lemma 3.6.2. $\sqrt{2} \cdot 0 = 0$:

Proof

Lemma 3.6.3.

Proof

Lemma 3.6.4.

Proof First we prove the (colour-reversed) version where $\alpha = \pi$:

(3.91)

And then we prove the general case:

$$
\alpha\!\!-\!\!\circ \overset{(\boldsymbol{sp})}{=} \alpha\!-\!\pi\!-\!\pi \overset{(\boldsymbol{\pi})}{=} \begin{matrix} -\alpha\!-\!\pi \\ \alpha\!-\!\pi \end{matrix} \overset{(\boldsymbol{sc})}{=} \begin{matrix} -\alpha \\ \alpha \end{matrix}\!\!-\!\!\circ\!-\!\pi \overset{(\boldsymbol{sp})}{=} \circ\!-\!\pi \overset{(3.91)}{=} \circ\!-\!\circ \qquad \square
$$

Proposition 3.6.5. Any ZX-diagram with n inputs and m outputs containing a zero scalar can be rewritten to the following normal form:

$$
\tag{3.92}
n \left\{ \begin{matrix} \overset{\pi}{} \\ \vdots \ \ \vdots \end{matrix} \right\} m
$$

Proof Apply Lemma 3.6.3 to all the connections between spiders and the inputs and outputs. The resulting diagram is then that of Eq. (3.92) except that there might be some scalar pairs of spiders and some isolated spiders, at most one of which has a phase. This phase can be removed using Lemma 3.6.4. If it is an isolated spider then it can be decomposed into a pair of pairs using Lemma 3.6.1. Now all the scalar pairs of spiders can be removed using Lemma 3.6.2. $\qquad\square$

Now let's see how we can combine some scalar diagrams by 'multiplying' them together.

Lemma 3.6.6. Two complex phases can be added together:

$$
\begin{matrix} \alpha\!-\!\pi \\ \beta\!-\!\pi \end{matrix} = \overset{\circ-\circ}{(\alpha+\beta)\!-\!\pi}
$$

Proof

$$
\begin{matrix} \alpha\!-\!\pi \\ \beta\!-\!\pi \end{matrix} \overset{\underset{(\boldsymbol{sc})}{(\boldsymbol{s})}}{=} \begin{matrix} \alpha \\ \beta \end{matrix}\!\!-\!\!\circ\!-\!\pi \overset{(\boldsymbol{sp})}{=} \overset{\circ-\circ}{(\alpha+\beta)\!-\!\pi} \qquad\square
$$

As a special case, where $\alpha = \beta = \pi$ we can show that $(-\sqrt{2})\cdot(-\sqrt{2}) = 2$:

Lemma 3.6.7.

$$
\begin{matrix} \pi\!-\!\pi \\ \pi\!-\!\pi \end{matrix} = \begin{matrix} \circ\!-\!\circ \\ \circ\!-\!\circ \end{matrix}
$$

Note that when the X-spider does not have a π phase, that the phases simply dissappear as in Lemma 3.6.4.

We can also simplify some expressions involving $\frac{\pi}{2}$. This is because this diagram equals $1 + i$, and when we multiply it with itself, for instance, we get $(1+i)(1+i) = 2i$, which should be equal to a simpler diagram. In order

to prove this, we will need the following scalar-accurate version of Eq. 3.84 in Exercise 3.15.

Lemma 3.6.8.

$$\left(\tfrac{\pi}{2}\right)\!- \;=\; \left(-\tfrac{\pi}{2}\right)\!-$$

Proof   We prove this using the colour-reversed Euler decomposition rule of Figure 3.2:

$$\left(\tfrac{\pi}{2}\right)\!- \;\overset{(cc)}{=}\; \left(\tfrac{\pi}{2}\right)\!-\square\!- \;\overset{(eu)}{=}\; \cdots \;\overset{(sc)}{=}\; \cdots \;\overset{(cc)}{=}\; \cdots \qquad\qquad \square$$

Lemma 3.6.9. All instances of $\tfrac{\pi}{2}$ and $-\tfrac{\pi}{2}$ can be combined together so that only one $\pm\tfrac{\pi}{2}$ phase is necessary.

Proof   First, when we have two $\tfrac{\pi}{2}$ subdiagrams we do the following rewrite:

$$\overset{(cc)}{=} \;\cdots\; \overset{(sp)}{=} \;\cdots\; \overset{3.6.8}{=} \;\cdots \tag{3.93}$$

When we have multiple diagrams resulting in this way, we can combine them using Lemma 3.6.6. We can use this to combine a $\tfrac{\pi}{2}$ and a $-\tfrac{\pi}{2}$ as well:

$$\overset{(sp)}{=} \;\cdots\; \overset{3.6.8}{=} \;\cdots\; \overset{(3.93)}{=} \;\cdots\; \overset{(3.93)}{=} \;\cdots\; \overset{3.6.4}{=} \;\cdots \tag{3.94}$$

We leave it to the reader to find an analogous simplification of Eq. (3.93) to combine $\tfrac{\pi}{2}$.

After applying these rewrites there are then at most two $\tfrac{\pi}{2}$ phases in the diagram, one coming from a possible $\tfrac{\pi}{2}$ or $-\tfrac{\pi}{2}$, and the other coming from a diagram which results from the rewrite (3.93). These can be combined as follows:

$$\overset{(s)}{=} \;\cdots\; \overset{(3.94)}{=} \;\cdots\; \overset{(s)}{=} \;\cdots\; \overset{(s)}{=} \;\cdots$$

The case with $-\tfrac{\pi}{2}$ is proven similarly. $\square$

There is of course more to be said about calculating the values of small scalar ZX-diagrams in this way, but hopefully the reader can see now how this generally can be done. We will use these results on manipulating scalars in Section 5.5 when we prove that the rules of Figure 3.2 are actually enough to prove all equalities concerning ZX-diagrams where the phases are multiples of $\frac{\pi}{2}$.

3.7 References and further reading

Categorical quantum mechanics The ZX-calculus came forth from the field of *categorical quantum mechanics*, an area of research initiated by Abramsky and Coecke (2004) that aimed to study quantum phenomena abstractly, using nothing but the structure of *symmetric monoidal categories*. These are mathematical structures wherein both composition and tensor product make sense Selinger (2010): in a category we have composition Mac Lane (2013), 'monoidal' means that there is a tensor product, and 'symmetric' means that we have a SWAP operation. In a ZX-diagram we can't just swap two inputs, we can also swap an input with an output using cups and caps. A category with this kind of structure is called *compact closed* Kelly and Laplaza (1980). It is in this abstract category theory that it was proven that we can move the generators around on the page while preserving the semantics of the diagram, and hence why it even makes sense to write it as a diagram. The original categorical quantum mechanics paper Abramsky and Coecke (2004) gave a proof of quantum teleportation that is essentially the basis of the ZX proof in Section 3.3.2.

The origin of the ZX-calculus The ZX-calculus was born on a bus in Iran in 2007. Bob Coecke, a professor at Oxford and Ross Duncan, a recently-finished PhD student, were discussing how to capture a certain feature of quantum measurements using category theory. In particular, they were thinking about mutually unbiased bases (mubs), i.e. sets of bases that pairwise satisfy equation (3.65) from Section 3.2.4.2. While mubs, which were first introduced by Schwinger (1960), are very simple to define, they have a suprisingly rich structure that is still only partially understood. A running joke at the time was, if you wanted to ruin a quantum computer scientist's career (and personal life), you should just introduce them to the problem of classifying all mubs in dimension 6. Earlier that year, Coecke and Pavlovic (2007) gave a more "categorically flavoured" notion of fixing a measurement, which they called **classical objects**. These have gone under a variety of names—notably

classical structures or **dagger-special commutative Frobenius algebras**—but these days most people refer to them as **spiders**. It seemed at the time that these abstract algebraic entities gave the correct notion of fixing an ONB without needing to refer to concrete Hilbert space structure, an intuition that was vindicated in 2013 when Coecke, Pavlovic, and Vicary showed that spiders are in 1-to-1 correspondence with ONBs (Coecke et al., 2013).

By studying two families of spiders satisfying some interaction equations, Coecke and Duncan hoped to get some deeper insights in the behaviour of mubs, and hence the ZX-calculus was born (Coecke and Duncan, 2008). In fact, it wasn't called the ZX-calculus until almost three years later (Coecke and Duncan, 2011), with most early talks and papers referring to it as the "calculus of complementary observables", simply "the graphical calculus", or even the "Christmas calculus" due to its convention of using green and red dots for Z and X spiders. As the story goes, this convention was born from the fact that black and blue whiteboard markers in the Oxford Computing Laboratory were constantly running out of ink, so the only working ones were green and red.

Strong complementarity The name of *strong complementarity* for the rule of Section 3.2.4 was introduced in Coecke et al. (2012), but as noted in that section is also called the *bialgebra* equation. In fact, the bialgebra of a Z-spider and X-spider we have in the ZX-calculus is a very special case, as both spiders are Frobenius algebras. Two Frobenius algebras that form a bialgebra are called *interacting Frobenius algebras*. It was shown in Duncan and Dunne (2016) that these also form a Hopf algebra, essentially using the proof of Eq. (3.66), and hence that strong complementarity implies regular complementarity.

Completeness The version of the ZX-calculus from (Coecke and Duncan, 2008) was missing the Euler decomposition rule. This rule was introduced by Duncan and Perdrix (2009), where the authors showed that, without this rule, the ZX-calculus is incomplete for Clifford ZX-diagrams. With this rule, one obtains a version of the ZX-calculus equivalent to the one used in this book, which was shown to be complete for Clifford ZX-diagrams by Backens (2014a). That same year, Schröder de Witt and Zamdzhiev (2014) showed that the Clifford ZX-calculus was incomplete for non-Clifford ZX-diagrams, i.e. if we allow angles that are not multiples of $\pi/2$. The counter-example, suggested by Kissinger, came from considering the two possible Euler decompositions

of a single-qubit unitary:

$$ -\!\!\!\bigcirc\!\!\!\alpha\!\!\!\bigcirc\!\!\!\beta\!\!\!\bigcirc\!\!\!\gamma\!\!\!\bigcirc\!\!\!- \quad \propto \quad -\!\!\!\bigcirc\!\!\!\alpha'\!\!\!\bigcirc\!\!\!\beta'\!\!\!\bigcirc\!\!\!\gamma'\!\!\!\bigcirc\!\!\!- \tag{3.95} $$

It is always possible to compute α', β', γ' as (fairly elaborate) trigonometric functions of α, β, γ. When the angles on the LHS are not $\pi/2$ multiples, the angles on the RHS are typically irrational multiples of π, and it is not possible to prove equation (3.95) using just the Clifford ZX rules.

This remained the status of completeness for several years, and it seemed like it might even be the case that no finite set of rules would suffice to show completeness for a computationally universal fragment of the ZX-calculus such as Clifford+T ZX-diagrams (with $\pi/4$ multiple phases) or all ZX-diagrams (with arbitrary phases). Some smaller results appeared, such as the one-qubit Clifford+T completeness of Backens (2014b) and the 2-qubit completeness of Coecke and Wang (2018), but it remained unclear how to extend these to n qubits. In fact, Aleks would have bet money in 2015 that no finite, complete set of rules for n-qubit Clifford+T ZX-diagrams existed. *Picturing Quantum Processes*, the precursor to this book, which came out in the spring of 2017, had this problem as an "advanced exercise", at least partially as a joke.

However, it's a good thing Aleks didn't bet, because he would have lost his money. In the autumn of 2017, Jeandel, Perdrix, and Vilmart showed a complete ZX-calculus for the Clifford+T fragment (Jeandel et al., 2018), and later that same year Wang and Ng showed completeness for all ZX-diagrams (Ng and Wang, 2017). The key idea in these two completeness theorems was to use an encoding into a related graphical calculus called the ZW-calculus, formulated by Coecke and Kissinger (2010a) and proved complete by Hadzihasanovic (2015a).

While these presentations of the ZX-calculus were complete, they contained some redundancy, in the sense that some rules could be derived from others. A more compact presentation of the Clifford ZX-calculus, consisting just of the spider rules, strong complementarity, Euler, and colour change rules, was given by Backens et al. (2016). Similarly, a small complete set of rules for the full ZX-calculus was given by Vilmart (2019), essentially by adding (3.95) to the basic Clifford rules.

Measurement-based quantum computing Along with understanding the structure of mutually unbiased bases, a lot of the early development into ZX-calculus was motivated by providing a convenient notation for working with the one-way model of measurement-based quantum computation (MBQC), which we'll discuss extensively in Chapter 8. The representation of cluster

states as ZX-diagrams already appeared in the first paper (Coecke and Duncan, 2008) and started to be put to good use in subsequent work relating quantum circuits and measurement patterns (Duncan and Perdrix, 2010) and doing ZX-based optimisation of quantum circuits (Duncan et al., 2020).

Resources To some extent, this book grew out of the survey/tutorial paper *ZX for the Working Quantum Computer Scientist* (van de Wetering, 2020), which presents some of the topics we cover in a more condensed form. A good place to start learning more about ZX is zxcalculus.com, which maintains a tagged list of >100 papers on ZX, related calculi, and applications. It also contains information about various communities, online seminars, and mailing lists. The diagrams in this book were typeset with the help of TikZiT, which acts as a GUI for making LaTeX suitable TikZ diagrams. TikZiT was made by Aleks in an effort to make typesetting the 5000 diagrams in his previous book somewhat less horrific, and his current coauthor John is very thankful for its existence.

4

CNOT circuits and phase-free ZX-diagrams

ZX-diagrams are a universal language for talking about quantum computing. This makes them useful, but it also means that for generic ZX-diagrams, we expect certain problems to be hard to solve with rewriting. For example, if we could efficiently determine if two different ZX-diagrams describe the same matrix, we can also determine if two quantum circuits actually describe the same unitary. However, we have good, complexity-theoretic reasons to believe that that is way harder than anything even a quantum computer can do (much less a classical one).

As a special case, this would let us determine efficiently the complex number described by a diagram with no inputs and outputs, which would let us efficiently simulate quantum computers. This would of course make the whole project of quantum computation (and essentially the jobs of the authors of this book) pointless.

Since we don't believe we can efficiently solve certain problems for *any* ZX-diagram, it makes sense to restrict to classes of diagrams which can be reasoned about efficiently.

The restriction of the ZX-calculus to a family of ZX-diagrams that is closed under composition and tensor product is a **fragment** of the ZX-calculus. In this chapter we will look in detail at one of the simplest possible fragments of the ZX-calculus: the **phase-free ZX-calculus**. This fragment concerns ZX-diagrams where all the phases on the spiders are required to be zero, and where we do not have Hadamard gates. The phase-free diagrams turn out to behave very nicely, and we can rewrite them in various fruitful ways.

Their nice structure comes from a deep connection with linear algebra over the two-element field $\mathbb{F}_2$. Whereas quantum theory primarily concerns itself with (exponentially large) matrices whose elements are in $\mathbb{C}$, we'll see in this chapter that a special class of operations can be described in a different way using (much smaller) matrices whose elements are in $\mathbb{F}_2$.

The reason we care about phase-free ZX-diagrams is because such diagrams that are unitary correspond precisely to quantum circuits consisting solely of CNOT gates, so-called **CNOT circuits**. In this chapter we will explore in detail the relation between CNOT circuits, phase-free ZX-diagrams and linear parity maps.

4.1 CNOT circuits and parity matrices

In this section, we will see that the action of a CNOT circuit on n qubits can be succinctly represented as an $n \times n$ matrix over the field $\mathbb{F}_2$ and how to translate to and from this representation.

4.1.1 The two-element field and the parity of a bit string

First and foremost: what is $\mathbb{F}_2$? Let's give a formal definition.

Definition 4.1.1. The **field with 2 elements** $\mathbb{F}_2$ is defined as the set $\mathbb{F}_2 := \{0, 1\}$ which comes equipped with addition and multiplication operations defined as: $x + y := x \oplus y$ and $x \cdot y = x \wedge y$, where $\oplus$ and $\wedge$ are the XOR and AND operations defined on bits, respectively.

This may not look much like the fields of numbers you are used to—like the real numbers $\mathbb{R}$, complex numbers $\mathbb{C}$, or rational numbers $\mathbb{Q}$—but just by thinking about the behaviour of XOR and AND, we can verify all of the field axioms. First off, we can easily see that XOR and AND are both associative, commutative, and have units 0 and 1 respectively:

$$(x + y) + z = x + (y + z) \qquad x + y = y + x \qquad x + 0 = x$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \qquad x \cdot y = y \cdot x \qquad x \cdot 1 = x$$

But what about inverses? In a field, every number x has to have an additive inverse $-x$ satisfying $-x + x = 0$. A special property of $\mathbb{F}_2$ is that every element is its own additive inverse. We have $0 + 0 = 0 \oplus 0 = 0$, but also $1 + 1 = 1 \oplus 1 = 0$. Furthermore, for $\mathbb{F}_2$ to be a field, every non-zero number has to have a multiplicative inverse. But here there is only one non-zero number: 1, and clearly $1 \cdot 1 = 1$.

We sometimes refer to an element in $\mathbb{F}_2$ as a **parity**. Parity is a property of bit strings: if a bit string contains an even number of 1s, we say it has parity 0, whereas if it contain an odd number of 1s, we say it has parity 1. For a bit string $\vec{b} = (b_1, \ldots, b_n)$, we can compute the parity by taking the

XOR of all of its bits, i.e. we sum up the bits as elements of the field $\mathbb{F}_2$:

$$\mathbf{parity}(\vec{b}) := \sum_i b_i$$

One is often interested not just in the overall parity of a bit string, but also the parity of some subset of bits. This can be computed using simple matrix operations. For example, we can express the operation of computing the parity of the 1st, 3rd, and 4th bits of a 4-bit string as a row vector:

$$\begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = b_1 \oplus b_3 \oplus b_4$$

More generally, if we are interested in computing several parities at once, we can arrange them in the rows of a matrix:

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} b_1 \oplus b_3 \oplus b_4 \\ b_2 \oplus b_3 \\ b_1 \oplus b_4 \\ b_4 \end{pmatrix}$$

By saving or transmitting some parity information about a bit string, we can often detect and correct errors. For example, if the bit b_2 got lost in transmission, but we know the values of b_3 and $b_2 \oplus b_3$, we can recover b_2 since $b_2 = b_2 \oplus b_3 \oplus b_3$.

Such basic operations are the basis of classical linear error correcting codes, and as we'll see in Chapter 11, also play a major role in quantum error correction.

However, before we get there, we'll see a much more immediate connection between parity matrices and quantum circuits: the behaviour of a CNOT circuit can be exactly captured by an invertible parity matrix.

4.1.2 From CNOT circuits to parity maps

The CNOT gate acts on a pair of qubits via $|x, y\rangle \mapsto |x, x \oplus y\rangle$. When we put multiple CNOT gates in a circuit, we can build unitaries that have more complicated actions on the computational basis states. For instance, suppose we have a CNOT from qubit 1 to 2, and then a CNOT from 2 to 1:
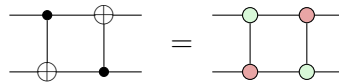
We can calculate its action on the computational basis state $|xy\rangle$ as:

$$|x, y\rangle \mapsto |x, x \oplus y\rangle \mapsto |x \oplus x \oplus y, x \oplus y\rangle = |y, x \oplus y\rangle$$

More generally, we can calculate the action of any CNOT circuit on a computational basis state by labelling each of the input wires with a variable x_i, then pushing those variables through the circuit:

$$
\begin{array}{cc}
 & (4.1)
\end{array}
$$
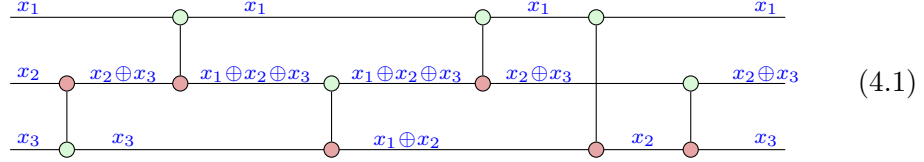
That is, we work from left to right. When we encounter a CNOT gate whose input wires are labelled a on the control qubit and b on the target qubit, we copy the label a onto the output wire of the control qubit and write $a \oplus b$ on the output wire of the target qubit. Once we get to the output of the circuit, we will have calculated the overall action of the unitary on computational basis states. For example, the circuit (4.1) implements the following unitary:

$$U \ :: \ |x_1, x_2, x_3\rangle \mapsto |x_1, x_2 \oplus x_3, x_3\rangle \tag{4.2}$$

Generally we can describe the action of a CNOT circuit by a parity map:

Definition 4.1.2. A **parity map** is a function $f : \{0,1\}^n \to \{0,1\}^m$ where $f = (f_1, \ldots, f_m)$ and each f_i calculates the parity of some of the input bits: $f_i(\vec{x}) = x_{j_{i1}} \oplus x_{j_{i2}} \oplus \cdots \oplus x_{j_{ik_i}}$.

By pushing variables through a circuit as in the example above, we can straightforwardly calculate the parity map for any CNOT circuit. Hence, the following proposition is immediate.

Proposition 4.1.3. Let C be an n-qubit CNOT circuit describing the unitary U_C. Then there is a parity map $f : \{0,1\}^n \to \{0,1\}^n$ such that for every computational basis state $|\vec{x}\rangle$ we have $U_C|\vec{x}\rangle = |f(\vec{x})\rangle$.

Furthermore, the parity map totally defines the unitary associated with the CNOT circuit. In particular, if we find another CNOT circuit with the same parity map, it implements the same unitary. For example, the unitary implemented by the CNOT circuit (4.1) could also be implemented by this much smaller CNOT circuit:

$$
\begin{array}{cc}
 & (4.3)
\end{array}
$$

By starting with a CNOT circuit, computing its parity map, then finding a new circuit that implements that same parity map, we ended up with a circuit that was a lot smaller, with just one CNOT gate rather than six. We call the problem of generating a CNOT circuit that implements a given parity map the **CNOT circuit synthesis** problem.

In the next section, we will show that, as long as a parity map is invertible, we can always synthesise a CNOT circuit that implements it.

4.1.3 CNOT circuit synthesis

We can view a bit string $\vec{x} \in \{0,1\}^n$ as a vector of the vector space $\mathbb{F}_2^n$. With this in mind, it follows that parity maps, which only ever compute XORs of their input bits, are actually linear maps over such vector spaces. Consequently, we can always write them in matrix form.

Definition 4.1.4. A **parity matrix** is a matrix with entries in $\mathbb{F}_2$.

One can show straightforwardly that for any parity map f, we can find a parity matrix A such that $f(\vec{x}) = A\vec{x}$. Indeed we saw an example of this in section 4.1.1 when we represented the parity map:

$$f(b_1, b_2, b_3, b_4) = (b_1 \oplus b_3 \oplus b_4, b_2 \oplus b_3, b_1 \oplus b_4, b_4)$$

with the matrix:

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

In the last section, we saw how to represent the action of a CNOT circuit as a parity map. It will be convenient to write this map as a parity matrix. Since CNOT circuits represent reversible computations on bits, their associated parity matrices are always invertible.

Remark 4.1.5. Note that we now have two different ways to represent CNOT circuits as matrices. On the one hand, treating them as quantum circuits gives us a $2^n \times 2^n$ unitary matrix U whose entries are all complex numbers $\{0,1\} \subseteq \mathbb{C}$. These matrices always correspond to permutations of computational basis states, so there is always a single 1 in each row and column and $U^\dagger = U^T = U^{-1}$. On the other hand, treating them as classical parity maps gives us a $n \times n$ *invertible* matrix P whose entries are bits $\{0,1\} \subseteq \mathbb{F}_2$. In general, these can have multiple 1's in any give row/column and P^T might not equal P^{-1}.

Thinking about the action of a single CNOT gate:

$$\text{CNOT}|x_1, x_2\rangle = |x_1, x_1 \oplus x_2\rangle$$

we get the parity map $f(x_1, x_2) = (x_1, x_1 \oplus x_2)$ and hence the following lower-triangular parity matrix:

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

Notably, since $1 + 1 = 0$ in $\mathbb{F}_2$, this matrix is its own inverse:

$$E^2 = \begin{pmatrix} 1 & 0 \\ 1 \oplus 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

This captures the fact that two successive CNOT gates cancel out.

We can generalise this to a single CNOT gate appearing in a larger circuit as follows.

Example 4.1.6. For an n-qubit circuit, the parity matrix corresponding to a CNOT with the control on qubit i and the target on qubit j is the identity matrix with one additional 1 in the i-th column and the j-th row. We will denote this matrix by E^{ij}. Just like with the matrix E above, we have $(E^{ij})^2 = I$.

You may have met matrices like these before in a linear algebra course. Matrices that look like the identity matrix with the exception of one additional non-zero value correspond to **primitive row and column operations** used for Gaussian elimination.

If I multiply a generic matrix M with E^{ij} on the left, this has the effect of adding the i-th row of M to the j-th row:

$$E^{ij} \begin{pmatrix} R_1 \\ \vdots \\ R_i \\ \vdots \\ R_j \\ \vdots \\ R_n \end{pmatrix} = \begin{pmatrix} R_1 \\ \vdots \\ R_i \\ \vdots \\ R_i + R_j \\ \vdots \\ R_n \end{pmatrix}$$

whereas if we multiply by E^{ij} on the right, it has the effect of adding the j-th column to the i-th column:

$$\begin{pmatrix} C_1 & \cdots & C_i & \cdots & C_j & \cdots & C_n \end{pmatrix} E^{ij} = \begin{pmatrix} C_1 & \cdots & C_i + C_j & \cdots & C_j & \cdots & C_n \end{pmatrix}$$

A general property of invertible square matrices is we can reduce them to the identity matrix by means of primitive row operations or primitive column operations. For a generic field, there are two kinds of primitive row/column operations: multiplying a row/column by a non-zero scalar and adding one row/column to another. This is what happens when we apply the **Gauss-Jordan reduction** procedure, sometimes called simply Gaussian elimination, to an invertible matrix. For $\mathbb{F}_2$, there is only one non-zero scalar, 1, so in fact the second kind is all we need.

There are two, essentially equivalent ways we can do Gauss-Jordan reduction, either working from the left side of the matrix with row operations or the right side of the matrix with column operations. Although it is the (slightly) less typical version, here we'll use column operations. This will make things easier when it comes to generalising to bigger families of circuits in Chapter 7.

Suppose we find any sequence of k primitive column operations that reduce a parity matrix to the identity. Then we have a series of elementary matrices $E^{i_1 j_1}, \ldots, E^{i_k j_k}$ such that:

$$AE^{i_1 j_1} \cdots E^{i_k j_k} = I$$

As we noted in Example 4.1.6, each of the elementary matrices is their own inverse. So, we can move them to the other side of this equation, reversing the order:

$$A = E^{i_k j_k} \cdots E^{i_1 j_1} \tag{4.4}$$

Hence, we can see Gaussian elimination as a way of decomposing invertible matrices as a composition of elementary matrices. Since we know that every parity matrix coming from a CNOT circuit is invertible and that each elementary matrix corresponds to a CNOT gate, we see that synthesising CNOT circuits from parity matrices amounts to Gaussian elimination!

We give the full synthesis procedure in Algorithm 1. Note we write CNOT^{ij} for a CNOT gate with a control on the i-th qubit and a target on the j-th qubit.

Theorem 4.1.7. Algorithm 1 works.

Proof This algorithm produces one CNOT gate corresponding to each elementary column operation in the decomposition of A from equation 4.4. Hence, the overall effect of C on a computation basis input will be the composition of these elementary matrices, which is A. □

Exercise 4.1. Consider the following CNOT circuit:

Algorithm 1: CNOT circuit synthesis by Gauss-Jordan reduction

Input: An $n \times n$ invertible parity matrix A
Output: A CNOT circuit implementing A
Procedure CNOT-SYNTH*(A)*

    let C be an empty circuit on n qubits
    for $i = 1$ **to** n **do** *// forward part*
        if $A_{ii} = 0$ **then**
            // ensure that a 1 is on the diagonal
            find $k > i$ such that $A_{ik} \neq 0$
            add column k of A to column i
            append CNOT^{ik} to C
        end
        for $j = i + 1$ **to** n **do**
            if $A_{ij} \neq 0$ **then**
                add column i of A to column j
                append CNOT^{ji} to C
            end
        end
    end
    for $i = n$ **to** 1 **do** *// backwards part*
        for $j \in (i - 1)...1$ **do**
            if $A_{ij} \neq 0$ **then**
                add column i of A to column j
                append CNOT^{ji} to C
            end
        end
    end
    return C
end

---
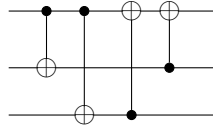
a) Calculate the parity matrix of this circuit.

b) Resynthesise a new equivalent CNOT circuit from this parity matrix by using Algorithm 1.
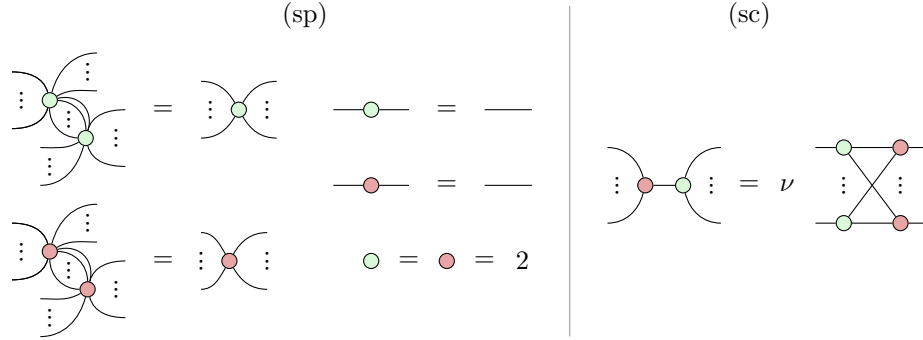
Figure 4.1 The rules of the phase-free ZX calculus: the spider rules (sp) and strong complementarity (sc). Note the righthand-side of the (sc) rule is a complete bipartite graph of m Z spiders and n X spiders, with a normalisation factor $\nu := 2^{(m-1)(n-1)/2}$, which we typically drop when scalar factors are irrelevant.

4.2 The phase-free ZX calculus

Now that we understand CNOT circuits and their relationship to parity matrices a bit better, let's turn our attention to phase-free ZX-diagrams and the phase-free ZX calculus. As we noted at the beginning of this chapter, phase-free ZX-diagrams are those without Hadamard gates and whose spiders all have phase 0. It turns out that once we make this restriction, one can simplify the rules a great deal. As we'll see in Section 4.3.1, the rules in Figure 4.1 suffice to prove any true equation between phase-free ZX-diagrams.

We will see in this section and the next that the phase-free ZX calculus encodes a great deal of information about CNOT circuits, and more generally linear algebra over $\mathbb{F}_2$. To start to get an idea of why this is the case, we'll temporarily introduce some π phases into X spiders to represent basis states, and recap some of the things we already saw in the previous chapter.

Recall that we can represent the two Z-basis states as X-spiders, with a phase of 0 or π, respectively:

$$\bullet\!\!-\!\!- \quad \propto \ |0\rangle \qquad \pi\!\!-\!\!- \quad \propto \ |1\rangle$$

where '$\propto$' here means we are ignoring the scalar factor. Due to this fact, it will be convenient to represent computational basis elements using a boolean variable $a \in \{0, 1\}$:

$$a\pi\!\!-\!\!- \quad \propto \ |a\rangle$$

A Z-spider acts on a computational basis state by copying it through,

while an X-spider calculates the XOR of its inputs:
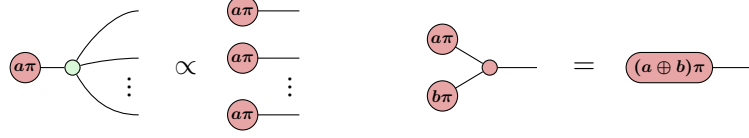
The latter follows from the fact that, for boolean variables a, b, we have $(a + b)\pi = (a \oplus b)\pi$, modulo 2π.

Recall what the CNOT gate looks like as a ZX-diagram:
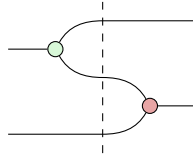
With the interpretation of the Z-spider as copying and the X-spider as XOR we see that we can interpret this diagram as saying that we first copy the first bit, and then XOR it with the second bit, and this is indeed how we understand the functioning of the CNOT:
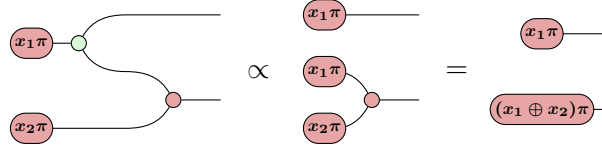
It turns out that this idea generalises to describe the action of an arbitrary parity matrix in the following way: suppose we have an $n \times m$ parity matrix A, then its corresponding ZX-diagram is given by

- one Z-spider connected to each input $1 \leq i \leq n$ and one X-spider connected to each output $1 \leq j \leq m$,
- a connection from the ith Z-spider to the jth X-spider if and only if $A_{ij} = 1$.

Another way to say this is that the **biadjacency matrix** of the connectivity from the Z-spiders to the X-spiders is equal to the parity matrix. This biadjacency matrix has a row for each X-spider, and a column for every Z-spider, and there is a 1 in the matrix when the corresponding Z-spider is connected to the corresponding X-spider, and a 0 if they are not connected.

Just like for CNOT circuits in the previous section, the parity matrix of a ZX-diagram in parity form exactly captures the action of the associated linear map on computational basis states.

Example 4.2.1. We have the following correspondence between this parity matrix and a ZX-diagram:

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \qquad \leftrightarrow \qquad \text{} \tag{4.5}$$

From the parity matrix, we can compute the action of the linear map on basis states (up to some scalar factor):

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1 \oplus x_3 \\ x_1 \oplus x_2 \end{pmatrix}$$

In other words, the linear map depicted above acts as follows on basis states:

$$L :: |x_1, x_2, x_3\rangle \mapsto |x_1 \oplus x_3, x_1 \oplus x_2\rangle$$

We can check this by inputting a computational basis state, expressed in terms of X-spiders and reducing:

It will be helpful to have a name for diagrams such as the one depicted in equation (4.5) above.

Definition 4.2.2. We say a phase-free ZX-diagram is in **parity normal form** (PNF) when

- every Z-spider is connected to exactly one input,
- every X-spider is connected to exactly one output,
- spiders are only connected to spiders of the opposite colour and via at most one wire.

If a diagram is in PNF, the only relevant information is the number of inputs, number of outputs, and which inputs are connected to which outputs. Hence biadjacency matrices are in 1-to-1 correspondence with ZX-diagrams in parity normal form.

4.2.1 Reducing a ZX-diagram to normal form

We have now seen that we can represent a CNOT circuit as a parity matrix and parity matrices as ZX-diagrams in parity normal form. Additionally, using Gaussian elimination we can go back from this parity form to a CNOT circuit. There is just one step missing in this trifecta of representations: how to reduce a unitary phase-free ZX-diagram to parity normal form. This is what we will do in this section. Apart from it being useful to understand how CNOT circuits relate to ZX-diagrams, it will also give us a first taste about how to define simplification strategies in the ZX-calculus.

The first step in this process of simplification will be to bring the ZX-diagram into a more canonical form.

Definition 4.2.3. We say a ZX-diagram is **two-coloured** when

1. no spiders of the same colour are connected to each other,
2. there are no self-loops,
3. there is at most one wire between each pair of spiders,
4. there are no Hadamards.

We call such diagrams two-coloured because we can view them as simple graphs (where the spiders are the vertices) with a two-colouring (corresponding to the colours of the spiders).

Lemma 4.2.4. Any ZX-diagram (not just a phase-free one) can be efficiently simplified to a two-coloured one.

Proof Given a ZX-diagram we do the following rewrites. First, if there are any Hadamards, we decompose them into spiders using (**eu**), so that the diagram will only contain actual spiders. Then we apply (**sp**) wherever we can. As a result, no two spider of the same colours are connected to each other in the resulting diagram (since otherwise we would have fused them). Second, we apply Eq. (3.39) to remove all self-loops on the spiders, so that the resulting diagrams has no self-loops left. Finally, whenever there is more than one connection between a pair of spiders, we apply complementarity (3.64) in order to remove a pair of these connections. This can always be done since the connections must necessarily be between spiders of different colours. □

Given a phase-free ZX-diagram we can apply the above procedure to reduce it a two-coloured diagram. This diagram will of course still be phase-free. Now we will apply rewrites to this diagram that bring it closer to parity normal form. In the parity normal form, all Z-spiders are connected to an input, while all X-spiders are connected to an output. So, 'bringing it closer'

to the normal form means reducing the number of Z-spiders that are *not* connected to an input or reducing the number of X-spiders *not* connected to an output. The way we do this will be to strategically apply the strong complementarity rule (***sc***).

Definition 4.2.5. We say a spider is an **input spider**, respectively **output spider** when it is connected to at least one input, respectively output. If a spider is neither an input spider nor an output spider, it is called an **internal spider**.

Definition 4.2.6. We say a phase-free ZX-diagram is in **generalised parity form** when it is two-coloured and

1. every input is connected to a Z-spider and every output to a X-spider,
2. every spider is connected to at most 1 input or output,
3. there are no zero-arity (i.e. scalar) spiders, and
4. no internal spiders are connected directly to each other.

To phrase the condition above in a different way, it is saying that there are only two kinds of Z-spiders: input Z-spiders and internal Z-spiders directly connected to output X-spiders. A similar categorisation applies to X-spiders, reversing the role of inputs/outputs. Hence, the generalised parity form looks like this, for some $m, n, j, k \geq 0$:

$$(4.6)$$
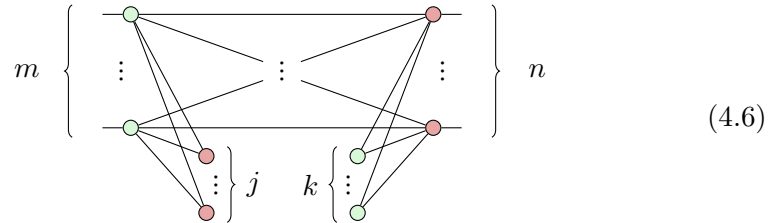
What makes this form "generalised" as opposed to diagrams in parity normal form is the possibility of some internal spiders. When $j = k = 0$, we get exactly the parity normal form.

In Algorithm 2 we show how to efficiently transform a phase-free ZX-diagram into generalised parity form.

Lemma 4.2.7. Algorithm 2 terminates efficiently with a ZX-diagram in generalised parity form.

Proof Note that step 1 always removes spiders from the diagram. Step 2 will remove a pair of spiders, but it will introduce new Z-spiders on all the neighbours of ● and new X-spiders on all the neighbours of ○. Since ● is not an output, the only possibility is that the new Z-spiders will be inputs

---

Algorithm 2: Reducing to generalised parity form

Input: A phase-free ZX-diagram

Output: A phase-free ZX-diagram in generalised parity form

1. Apply (***sp***) as much as possible and remove zero-arity spiders by multiplying the overall scalar by 2.
2. Try to apply (***sc***) where

   - ○ is **not** an input and
   - ● is **not** an output.

3. If (***sc***) was applied at step 2, go to step 1, otherwise go to step 4.
4. Use (***id***) in reverse to ensure inputs are connected to Z-spiders, outputs are connected to X-spiders, and that no spider is connected to multiple inputs/outputs.

---

or they will be adjacent to other Z-spiders. In the latter case, they will get removed by spider fusion when step 1 is repeated. Hence, step 2 removes a non-input Z-spider without introducing any new non-input Z-spiders.

This shows that the algorithm terminates after a number of iterations of steps 1–3 bounded by the number of non-output Z-spiders, where each of steps 1–4 takes polynomial time (in the number of spiders), so the whole algorithm terminates in polynomial time. The main loop in Algorithm 2 will only terminate once condition 4 of Definition 4.2.6 is satisfied, and since it applies step 1 just before exiting the loop, it will be in two-coloured form. Finally, step 4 ensures condition 1 is satisfied while preserving the other conditions. □

Exercise 4.2. Show how Step 4 in Algorithm 2 works. That is, show that we can use (***id***) in reverse to fix (1) cases where inputs/outputs are not connected to the correct type of spider and (2) cases where multiple inputs/outputs connect to the same spider.

Now, why do we care about generalised parity form? Well, it turns out that if the diagram describes a unitary, that such a diagram *must* be in parity normal form.

Proposition 4.2.8. Let D be a diagram in generalised parity form:
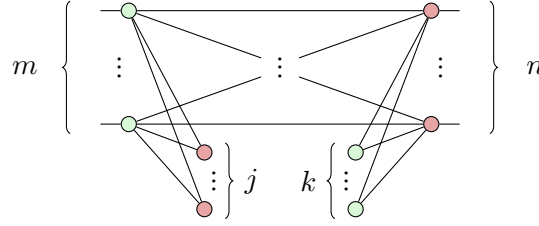
If D is an isometry, then $j = 0$ and if it is unitary $j = k = 0$. In particular, a unitary in generalised parity form is already in parity normal form.

Proof We first need to show that if D is an isometry then $k = 0$. That is, every Z-spider is connected to an input. Suppose there is an X-spider that is not connected to any output. By assumption we don't have any floating scalar spiders, so it must be connected to at least some Z-spiders. Since the diagram is in generalised parity form, these must then all be input Z-spiders. Input a $|1\rangle$ to one of the input Z-spiders that the X-spider is connected to and $|0\rangle$ to all the others. Then after copying states and fusing spiders there is a π phase on the X-spider:
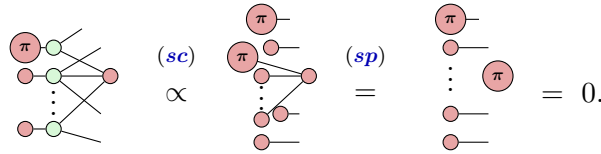
Hence, we have some input state that is mapped to zero, which contradicts D being an isometry. So all X-spiders must be connected to an output, and hence to a unique one by the previous paragraph.

If D is furthermore unitary, then we can similarly show that $k = 0$, i.e. that each Z-spider must be connected to an input. If this were not the case, we can plug $\langle -| \otimes \langle +| \otimes \ldots \langle +|$ into a subset of the output qubits to send the whole diagram to 0, which contradicts unitarity (since unitaries also preserve the norm of bras, and not just kets). $\square$

Theorem 4.2.9. Any unitary phase-free ZX-diagram can be efficiently rewritten in parity normal form.

Proof First use Lemma 4.2.4 to reduce it to two-coloured form. Then apply Algorithm 2 to reduce it further to generalised parity form. Since the diagram is unitary, it must then already be in parity normal form by Proposition 4.2.8. $\square$

Exercise 4.3. Prove that phase-free ZX-diagrams are invertible if and only if they are unitary. *Hint: Look closely at the assumptions that are needed to make the proof of Proposition 4.2.8 work.*

4.2.2 Graphical CNOT circuit extraction

In this section, we will see that we can derive the exact same CNOT circuit synthesis procedure from Section 4.1.3 starting with ZX-diagrams in parity normal form and applying strong complementarity to extract a CNOT circuit via Gaussian elimination. While the ZX reformulation doesn't tell us something new about the specific case of CNOT circuits *per se*, we will see this basic graphical technique appearing in various guises throughout the book, so it will be instructive to work through it explicitly here. For instance, if we have a diagram that is more complicated, but locally has a part that looks like like a parity normal form, then we can still (partially) apply the graphical Gaussian elimination strategy.

The key point to the proof is to realise that the phase-free ZX-calculus already 'knows' how to do $\mathbb{F}_2$-linear algebra. To get started, we'll see that we can prove that CNOT matrices perform elementary column operations on parity matrices using just the ZX rules.

First of all, what does this look like graphically? In a parity normal form, the Z-spiders correspond to columns of the biadjacency (i.e. parity) matrix, and the wires coming out of the Z-spider correspond to 1's in that column. In order to see a CNOT as performing a column operation, we should therefore see that pre-composing two Z-spiders by a CNOT has the effect of "adding" the connections of one spider to the other, modulo 2. That is the content of the following lemma.

Lemma 4.2.10. The following identity holds in the ZX-calculus:
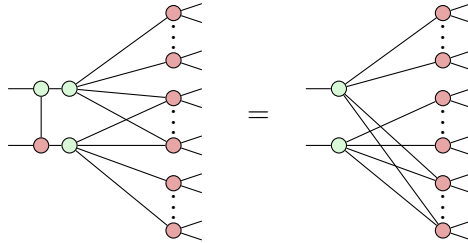
Proof
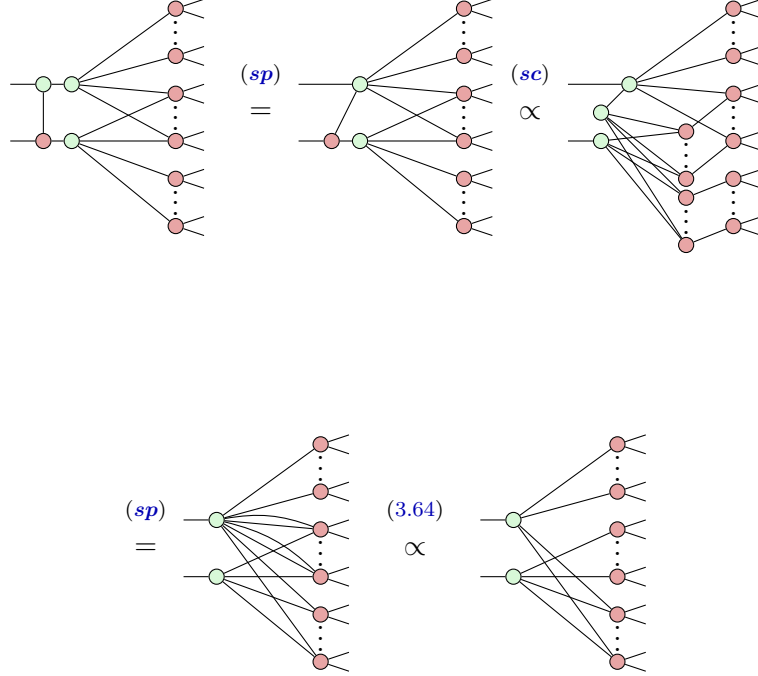
This lemma says that when we apply a CNOT to a pair of inputs in a diagram in parity normal form that we can 'absorb' this CNOT at the cost of changing the connectivity of the diagram. In the case of parity normal form diagrams, this change in connectivity corresponds precisely to an elementary column operation. This is easiest to see by means of an example.

Example 4.2.11. Consider the following ZX-diagram in parity normal form, and its associated biadjacency matrix:

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

For unitaries, this matrix will always be a square and invertible. Since the matrix is invertible, we can always reduce it to the identity using primitive column operations, which in turn correspond to pre-composing with CNOT gates. For example, consider this sequence of column operations:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{c_2 := c_2 + c_1} \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{c_3 := c_3 + c_2} \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{c_1 := c_1 + c_2} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This corresponds to a sequence of applications of Lemma 4.2.10, where each

one introduces a CNOT gate and changes the connectivity of the rest of diagram:
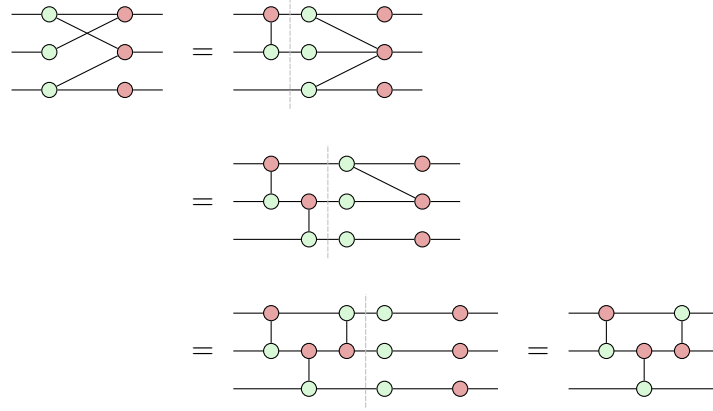
Here, the diagram to the right of the dashed line is always in PNF, and its connectivity corresponds exactly to the intermediate steps of the Gauss-Jordan reduction.

As we noted in Section 4.1.3, whenever a parity matrix is invertible, it is possible to reduce it all the way to the identity using just elementary column operations. Hence, the following proposition follows immediately.

Proposition 4.2.12. A ZX-diagram in parity normal form whose biadjacency matrix is invertible can be diagrammatically rewritten into a CNOT circuit.

Exercise 4.4. We need the assumption that the biadjacency matrix is invertible. Generate the parity normal form diagram of the following parity matrix and try to apply the strategy above. What goes wrong?

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

Exercise 4.5. Show that a ZX diagram in PNF is unitary if and if only its biadjacency matrix is invertible.

Combining Lemma 4.2.7 and Proposition 4.2.8 we see that we have a strategy for simplifying any unitary phase-free ZX-diagram to parity normal form. Additionally, such a diagram is equivalent to a CNOT circuit.

Theorem 4.2.13. Any unitary phase-free ZX-diagram can be efficiently rewritten into a CNOT circuit.

Proof By Theorem 4.2.9 a unitary phase-free diagram can be simplified to parity normal form. By Exercise 4.5, we know the associated biadjacency matrix is invertible, hence we can apply Proposition 4.2.12 to extract a CNOT circuit. ☐

As CNOT circuits themselves are unitary phase-free ZX-diagrams, this theorem gives us an evident technique for a purely ZX-based method for optimising CNOT circuits: from a CNOT circuit compute the PNF, then re-extract using strong complementarity as in Example 4.2.11.

Exercise 4.6. If the diagram V is an isometry, then it can be written as:

where V' is in parity normal form with an injective parity matrix P. Using this fact, show that we can then furthermore write V as follows, in terms of a unitary phase-free diagram U:

4.3 Phase-free states and $\mathbb{F}_2$ linear subspaces

Let's have a look again at the generalised parity form (4.6):
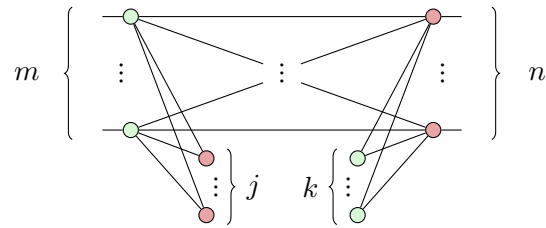
In the previous section, we focussed on the special case where the phase-free ZX-diagram diagram describes a unitary, in which case we must have $j = k = 0$.

We now turn to a *different* special case, where the ZX-diagram depicts a state. In this case, $m = 0$, and hence up to scalar factors, we can also take

$j = 0$. Hence, we are only left with a layer of k internal Z spiders, followed by a layer of n output X spiders:

$$k \left\{ \vcenter{\hbox{\includegraphics{}}} \right\} n \tag{4.7}$$

Note that we could still represent such a diagram by its biadjaceny matrix, but columns correspond to internal Z spiders rather than inputs. In particular the order of the columns no longer matters, so rather than representing such a diagram as a matrix, it makes more sense to represent it simply as a set of $\mathbb{F}_2$-vectors.

Definition 4.3.1. The **Z-X normal form** of a phase-free ZX-diagram state consists of a row of internal Z spiders connected to a row of X spiders each connected to precisely 1 output. It can be described as a set of vectors $\{v_1, \ldots, v_k\}$ over $\mathbb{F}_2$ where $(v_p)_q = 1$ if and only if the p-th internal Z spider is connected to the q-th X spider.
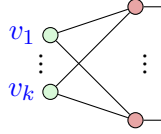
At this point, the careful reader may note that a Z-X normal form might contain more than one Z spider connected to the exact same set of X spiders. From this, one might think we should actually describe the Z-X normal form by a *multi-set* of $\mathbb{F}_2$-vectors, i.e. a set that allows duplicate elements. However, thanks to the rules of the phase-free ZX calculus, we can always remove spiders corresponding to duplicate bit-vectors.

Exercise 4.7. Show, using the phase-free ZX calculus, that:

$$\propto \tag{4.8}$$

Unlike the parity form, the Z-X normal form is not unique. However, we will soon see that it is straightforward to discover when two Z-X normal forms actually describe the same state, and how in that case we can transform one into the other.

The key point is to realise that the important information captured in $\{v_1, \ldots, v_k\}$ is not the set itself, but rather the subspace S of $\mathbb{F}_2^n$ spanned

by it. The usual way to define S is as the set of all of the possible linear combinations of $\{v_1, \ldots, v_k\}$:

$$S = \mathrm{Span}\{v_1, \ldots, v_k\} := \{a_1 v_1 + a_2 v_2 + \ldots + a_k v_k \mid a_1, \ldots, a_k \in \mathbb{F}_2\}$$

Equivalently, since the field is $\mathbb{F}_2$, we can think of S as the set of all vectors obtained by XOR-ing any subset of $\{v_1, \ldots, v_k\}$, including the zero vector (which corresponds to XOR-ing the empty set).

Since $\mathbb{F}_2$ is a finite field, any subspace of $\mathbb{F}_2^n$ (including $\mathbb{F}_2^n$ itself) is a finite set of vectors, so we can compute it explicitly. For example:

$$\mathrm{Span}\left\{ \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \right\} = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right\} \tag{4.9}$$

Let's look at the Z-X normal form corresponding to the pair of vectors on the lefthand-side of (4.9).
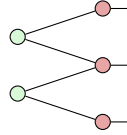
We can compute the state explicitly by expanding each Z spider as a sum over X spiders:

$$\propto \sum_{jk} |j, j \oplus k, k\rangle = |000\rangle + |110\rangle + |011\rangle + |101\rangle$$

The result is precisely the sum over all of the vectors in the subspace (4.9), written as elements of the computational basis.

This is true in general, as stated in the following theorem.

Theorem 4.3.2. For any Z-X normal form described by a set of $\mathbb{F}_2$-vectors $\{v_1, \ldots, v_k\}$, we have:

 $$\propto \sum_{b \in S} |b\rangle \qquad \text{where} \qquad S = \mathrm{Span}\{v_1, \ldots, v_k\} \tag{4.11}$$

This can be seen by direct calculation similar to (4.10), by expanding each

of the Z spiders as a sum $|0...0\rangle + |1...1\rangle$ and using the fact that X spiders act like XOR.

Exercise 4.8. Complete the proof of Theorem 4.3.2.

Theorem 4.3.2 says that the state associated with an n-qubit Z-X normal form is uniquely described by an $\mathbb{F}_2$-linear subspace S of $\mathbb{F}_2^n$. From the Z-X normal form itself, we can read off a spanning set of vectors for S. As a result, we are not too far from showing completeness of the phase-free ZX calculus.

4.3.1   Phase-free completeness

Recall that a **completeness** theorem states that if any two diagrams correspond to the same linear map, one can be transformed to the other just using graphical rules. In this section, we will prove completeness of the phase-free ZX-calculus for the fragment of phase-free ZX-diagrams.

Since we have already shown that phase free states correspond to linear subspaces and their associated Z-X normal forms correspond to sets of vectors spanning those spaces, the only thing left to do is show that we can transform one spanning set into another using just the phase free ZX rules.

Proposition 4.3.3. Let $\mathcal{B}$ and $\mathcal{C}$ be two sets spanning the same $\mathbb{F}_2$-linear subspace S. Then $\mathcal{B}$ can be transformed into $\mathcal{C}$ by

1. adding or removing the zero vector, or
2. replacing a pair of vectors $v, w \in \mathcal{B}$ with the pair $v, v + w$ zero or more times.

We will omit the proof here, as it is a standard result in linear algebra. Note that usually case 2 consists of replacing the pair v, w with $v, \lambda v + w$ for some non-zero scalar λ. But in $\mathbb{F}_2$, the only non-zero scalar is 1, so this simplifies.

We can now prove the following lemma by reproducing the two cases of Proposition 4.3.3 using the ZX calculus.

Lemma 4.3.4. Let $\mathcal{B} = \{v_1, \ldots, v_k\}$ and $\mathcal{C} = \{w_1, \ldots, w_l\}$ describe two Z-X normal forms on n qubit space. If the sets $\mathcal{B}$ and $\mathcal{C}$ span the same subspace of $\mathbb{F}_2^n$, one can be transformed into the other, up to a scalar factor.

Proof   We will show that the Z-X normal form for $\mathcal{B}$ can be transformed into that for $\mathcal{C}$ using the two cases described in Proposition 4.3.3. First, note that adding or removing a zero vector corresponds to adding or removing a zero-legged Z spider, which simply adds or removes a scalar factor of 2.

Hence, it suffices to show that we can transform a pair of Z-spiders described by the vectors v, w into Z spiders described by $v, v + w$. That is, we can "add" the wires of one spider to the other one, modulo 2. We can see this by splitting the X spiders into 3 groups: those connected to v, those connected to w, and those connected to both. We can then perform the vector addition by two applications of the (sc) rule. First, we apply the rule in reverse:
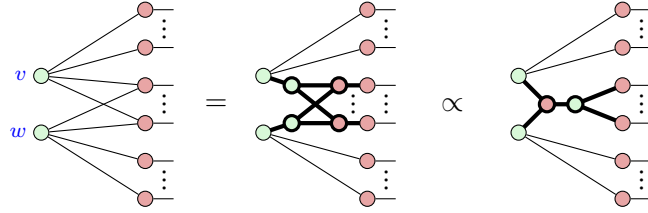
then forward:

□
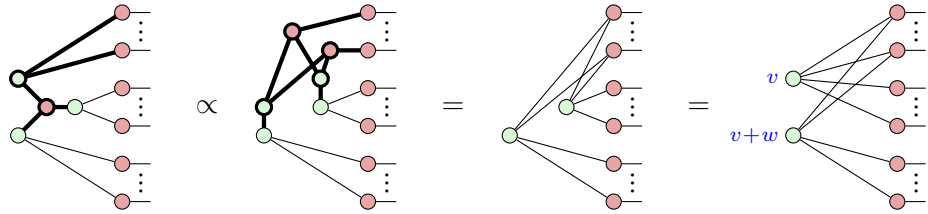
Proposition 4.3.5. If two phase-free ZX-diagram states describe the same linear map, one can be rewritten into the other using only the rules in Figure 4.1.

Proof Suppose two diagrams describe the same quantum state. Then, they can both be brought into Z-X normal form. By equation (4.11), their associated $\mathbb{F}_2$-vectors must span the same linear space S. Hence, by Lemma 4.3.4, one can be transformed into the other. The rules we used were only accurate up to scalar factors, however if the states are exactly equal, then once the non-scalar portion of the diagram is made equal, these ignored scalar factors must then also be exactly equal. □

We now have completeness of the phase-free ZX-calculus for states. To get the full completeness result we need to show how to adapt this to work for arbitrary diagrams. Because we can just bend wires this is not too hard.

Theorem 4.3.6 (Completeness of phase-free ZX)**.** If two phase-free ZX-diagrams describe the same linear map, one can be rewritten into the other using only the rules in Figure 4.1.

Proof Suppose the phase-free diagrams D_1 and D_2 are equal as linear maps. Then we know if we bend all the input wires to be output wires so that they are both states, that we still have equality. Additionally, because of completeness for states, we know that we can then rewrite one into the other. We can use this to show we can rewrite D_1 into D_2 diagrammatically:

Here we marked with a dotted box the states that we rewrite into each other using the completeness for phase-free states (Proposition 4.3.5) $\qquad\square$

In this proof we implicitly rewrote the "state form" of D_1 and D_2 into the Z-X normal form. If we take this normal form, and bend the input wires back we get a diagram that looks like the following:

$$(4.12)$$

We then have a layer of X-spiders connected to both the inputs and outputs, and these are connected via some internal Z-spiders. This is hence an alternative normal form for phase-free diagrams.

4.3.2   X-Z normal forms and orthogonal subspaces

All of the rules of the ZX calculus are colour-symmetric, so we could just as well obtain a normal form for phase-free ZX-diagrams consisting only of a row of internal X-spiders and a row of boundary Z-spiders. One way to see this is to start yet again from the generalised parity form:

and set $k = n = 0$ this time. We'll obtain a colour-reversed, mirror image of the Z-X normal form from (4.7):

$$m \left\{ \begin{array}{c} \vdots \end{array} \right. \left. \begin{array}{c} \vdots \end{array} \right\} j \tag{4.13}$$

Just like before, we can describe this object by a set of bit-vectors $\{w_1, \ldots, w_j\}$.

Definition 4.3.7. The **X-Z normal form** of a phase-free ZX-diagram consists of a row of internal X spiders connected to a row of Z spiders, each connected to precisely 1 input. It can be described as a set of vectors $\{w_1, \ldots, w_j\}$ over $\mathbb{F}_2$, where $(w_p)_q = 1$ if and only if the p-th internal X spider is connected to the q-th Z spider.

Suppose we plug some basis state $|v_1, \ldots, v_n\rangle$ into an X-Z normal form. The result will be a scalar, which depends on the XORs of some subsets of the input bits. For example:

$$\propto \qquad \propto$$
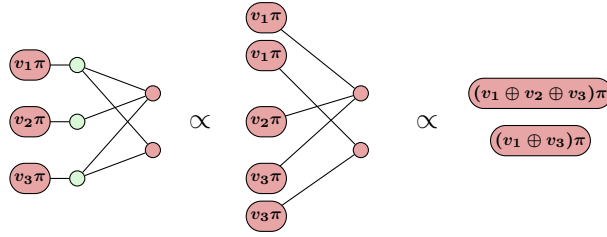
If either of the scalar X spiders has a phase of π, the whole thing goes to zero. If both X spiders have a phase of 0, then the scalar equals a fixed non-zero value N. In other words, the result is non-zero precisely when a collection of parities of input variables all equal zero. Hence, we can write the linear map as follows:

$$N \cdot \sum_{b \in S} \langle b| \qquad \text{where} \qquad S = \left\{ \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \;\middle|\; \begin{array}{c} v_1 \oplus v_2 \oplus v_3 = 0 \\ v_1 \oplus v_3 = 0 \end{array} \right\}$$

Recalling that in $\mathbb{F}_2$, $\oplus = +$ and the only possible coefficients in a linear equation are zero and one, requiring a set of parities to all be zero is the same thing as saying we have a solution to a system of **homogeneous $\mathbb{F}_2$-linear equations**. The term *homogeneous* just means all the righthand-sides are equal to zero, and famously the set of solutions to a homogeneous system of equations always forms a subspace.

A handy perspective on a system of homogenous equations is to think of

them as a spanning set of vectors for the **orthogonal subspace**, or "perp", $S^\perp$ of S, defined this way:
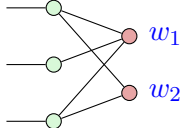
$$S^\perp = \{w \mid \text{ for all } v \in S, \ w^T v = 0\}.$$

Then, note:

$$\begin{pmatrix} w_1 & w_2 & \cdots & w_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = 0 \iff w_1 v_1 \oplus w_2 v_2 \oplus \ldots \oplus w_n v_n = 0.$$
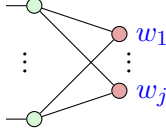
So, giving a spanning vector for $S^\perp$ is really just giving a list of coefficients $w_1, \ldots, w_n$ for a linear equation.

Rewriting the example above in terms of $S^\perp$, we have:

 $\propto \sum_{b \in S} \langle b \mid$ where $S^\perp = \text{Span}\left\{ w_1 := \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, w_2 := \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right\}$

Exercise 4.9. Show that this interpretation for X-Z normal forms is true in general, namely that:

 $\propto \sum_{b \in S} \langle b \mid$ where $S^\perp = \text{Span}\{w_1, \ldots, w_j\}$ (4.14)

As in the case for spanning sets of vectors for S, there are many choices of spanning set for $S^\perp$, which correspond to equivalent systems of linear equations. For example, adding one of the equations in a system to another one doesn't change the set of solutions:

$$\begin{array}{c} v_1 \oplus v_2 \oplus v_3 = 0 \\ v_1 \oplus v_3 = 0 \end{array} \qquad \Longleftrightarrow \qquad \begin{array}{c} v_1 \oplus v_2 \oplus v_3 = 0 \\ v_2 = 0 \end{array}$$

Such a move just corresponds to changing the spanning set for $S^\perp$:

$$S^\perp = \text{Span}\left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right\} = \text{Span}\left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right\}$$

which, like in the Z-X case, can be replicated graphically thanks to strong complementarity:
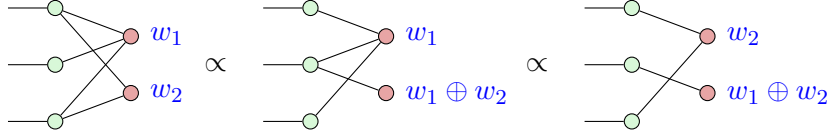
Exercise 4.10. Prove the two equations above using strong complementarity.

Exercise 4.11. Prove the general case of changing a basis for $S^\perp$ using strong complementarity. Use this to give an alternative proof of the phase-free completeness Theorem 4.3.6 that relies on the X-Z normal form.

The fact that we ended up writing X-Z normal forms as bra's rather than ket's is just an artifact of the convention we chose back in Section 4.2.1 to get Z spiders on the left and X spiders on the right in our generalised parity form. Everything in ZX is colour-symmetric, so we could have just as well done this the other way around to obtain an X-Z normal form for states. Hence, we actually get two equivalent ways to represent a state: the Z-X normal form and the X-Z normal form:

$$\propto \sum_{b\in S} |b\rangle \qquad \text{where} \qquad S = \text{Span}\{v_1,\dots,v_k\}$$

$$\propto \sum_{b\in S} |b\rangle \qquad \text{where} \qquad S^\perp = \text{Span}\{w_1,\dots,w_j\}$$

It is important to note that, most of the time, the number of vectors in these two forms will be different. For example, the GHZ state, given by the subspace $S = \text{Span}\{(1,1,1)\} = \{(0,0,0),(1,1,1)\}$ has this Z-X normal form:

$$|\text{GHZ}\rangle \propto |000\rangle + |111\rangle \ \propto$$

whereas $S^\perp = \text{Span}\{(1,1,0),(0,1,1)\}$, so it has this X-Z normal form:

$$|\text{GHZ}\rangle \ \propto$$

But of course, these two are equal, thanks to the rules of the ZX calculus:
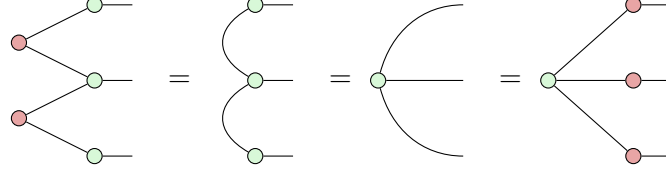
This relationship between the Z-X and X-Z normal form will come in handy when we start thinking of quantum error correcting codes in Chapter 11.

4.3.3 Relating parity matrices and subspaces

We've seen that phase-free unitaries can be represented efficiently as $\mathbb{F}_2$ linear maps and phase-free states can be represented as $\mathbb{F}_2$ linear subspaces. So, a natural question is: what happens when we hit a phase-free state with a phase-free unitary?

You may not be particularly surprised to find out that we can compute the subspace of the resulting state as the **direct image** of the original subspace. That is, for a phase-free unitary with parity map p:

$$U|\vec{x}\rangle = |p(\vec{x})\rangle$$

and a phase-free state corresponding to a subspace $S \subseteq \mathbb{F}_2^n$:

$$|\psi\rangle \propto \sum_{\vec{x} \in S} |\vec{x}\rangle$$

we have that $|\psi\rangle$ gets mapped to a new phase-free state described by the subspace $p(S)$:

$$U|\psi\rangle \propto \sum_{\vec{y} \in p(S)} |\vec{y}\rangle \qquad \text{where} \qquad p(S) := \{p(\vec{x}) \,|\, \vec{x} \in S\} \qquad (4.15)$$

In fact, this just follows from linearity, and the fact that U (which is injective) never takes two bit strings to the same place:

$$U|\psi\rangle \propto U(\sum_{\vec{x} \in S} |\vec{x}\rangle) = \sum_{\vec{x} \in S} U|\vec{x}\rangle = \sum_{\vec{x} \in S} |p(\vec{x})\rangle = \sum_{\vec{y} \in p(S)} |\vec{y}\rangle$$

Exercise 4.12. Consider the following phase-free state, corresponding to the subspace $\mathrm{Span}\{(1,1,0),(0,1,1)\}$:

$$|\psi\rangle = \quad$$

Compute $U|\psi\rangle$ for the following phase-free unitaries:

$$U \in \left\{ \quad \times \quad , \quad \varphi \quad , \quad \mathbb{Z} \quad \right\}$$

using the ZX calculus.

4.4 Summary: What to remember

1. An n-qubit CNOT circuit corresponds to an $n \times n$ *parity* matrix: a matrix over the field with 2 elements $\mathbb{F}_2 = \{0, 1\}$.
2. A CNOT circuit can be synthesised from a parity matrix using Gaussian elimination.
3. Phase-free ZX-diagrams are those diagrams where all the phases on the spiders are zero. They have a restricted rule set presented in Figure 4.1.
4. Phase-free ZX-diagrams can efficiently be rewritten to several different normal forms: generalised parity form, Z-X normal form, or X-Z normal form.
5. Unitary phase-free diagrams correspond to CNOT circuits.
6. The row operations of Gaussian elimination correspond to doing strong complementarity in the ZX-calculus.

4.5 Advanced Material*

4.5.1 Better CNOT circuit extraction*

In the discussion on universal gate sets in Section 2.3.5 we saw that the only multi-qubit gate we need is the CNOT gate. In current physical devices entangling gates are harder to implement than single-qubit gates, incurring more noise and taking a longer time to execute. All this is to say that CNOT gates are important, and we should think hard about how we can optimise their use as much as possible. We saw in this chapter that we can always resynthesise a CNOT circuit using Gaussian elimination. Whereas a starting CNOT circuit can be as long as we want, with this resynthesis we can get a definite bound on how long the circuits can get.

Each row operation in the elimination procedure gives us one CNOT gate. The question then is: how many row operations do we need in the worst case? Let's suppose we have some arbitrary $n \times n$ parity matrix. We will find an upper bound to how many row operations we need to reduce it to the identity matrix. Let's recall the distinct steps of Gaussian elimination.

For every column we need to make sure there is a 1 on the diagonal. Then we eliminate all the elements on the column below the diagonal. Once we have done this with every column, the matrix is in upper-triangular form. We then need to do the same elimination, column-by-column, for all the elements above the diagonal. The first column in the first stage then gives us at most n row operations: 1 for correcting the diagonal, and $n-1$ for correcting all the elements below the diagonal. Similarly, the second column takes $n-1$ operations. Reducing the matrix to upper-triangular form hence takes $\sum_{k=1}^{n} k - 1 = n(n+1)/2 - 1$, where this last -1 comes from the fact the final column will already have a 1 on the diagonal when we are doing elimination and the matrix is full rank. To reduce the rest of the matrix we no longer have to correct the diagonal, and hence fixing the last column only takes $n-1$ operations, and the second $n-2$, and so on. This hence occurs $\sum_{k=1}^{n-1} k = (n-1)n/2$ operations. This gives us a total of $n(n+1)/2 - 1 + (n-1)n/2 = n^2 + 1/2n - 1/2n - 1 = n^2 - 1$ operations exactly.

Proposition 4.5.1. Every CNOT circuit can be resynthesised using at most $n^2 - 1$ CNOT gates.

Requiring just n^2 gates, even though naively these circuits could be arbitrarily long is already pretty good. But is it also the best we can do? To determine this, we need to find some way to figure out how many CNOT gates the worst-case parity matrices require. We will do this by using a **counting argument**: we will count how many different possible invertible parity matrices there are, and then argue that therefore circuits must reach at least a certain length to represent them all. For instance, if we have just a single CNOT gate acting on n qubits, then this can represent at most $n(n-1)$ different parity matrices, as these are all the different ways we can place the CNOT in the circuit. If instead we have d CNOT gates, then there are $(n(n-1))^d$ different ways to place them, so that this is the maximum number of parity matrices we can represent. Since we actually care about how many parity matrices we can write down with circuits *up to D* CNOT gates, we should also allow the identity operation as one possibility, so that there is actually $n(n-1) + 1$ ways to optionally place a CNOT gate.

Now let's count how many invertible parity-matrices there are. Suppose we wish to build an invertible $n \times n$ parity matrix. We will do this column by column. The first column we can pick arbitrarily, as long as it is not all zero. We hence have $2^n - 1$ options for that column. Let's call this vector $\vec{c}_1$. Now for the second column we need to make sure that it is independent of the first column. So it can't be $\vec{c}_1$ or $\vec{0}$, but otherwise we are free to choose it.

So there are $2^n - 2$ options for $\vec{c}_2$. Now to choose $\vec{c}_3$, we need to make sure it is independent of $\vec{c}_1$ and $\vec{c}_2$. The vector space spanned by these vectors has 4 elements, and hence there are $2^n - 4$ options. At this point we can spot a pattern! The kth column has $2^n - 2^{k-1}$ options to choose from. The total number of invertible parity matrices is then

$$(2^n - 2^0)(2^n - 2^1) \cdots (2^n - 2^{n-1}) \ \geq \ \frac{1}{2}2^n \cdot \frac{1}{2}2^n \cdots \frac{1}{2}2^n \ = \ (2^{n-1})^n \ = \ 2^{n(n-1)}.$$

If we then claim that we can represent any invertible parity matrix by a CNOT circuit of at most d CNOT gates, then we must at least have:

$$(n(n-1) + 1)^d \ \geq \ 2^{n(n-1)}.$$

If this wasn't the case, then there would be no way for use to write down all the different parity matrices, since we just don't have enough space in our circuit for that many unique options. Let's take the logarithm in this equation, and rearrange the terms:

$$d \ \geq \ \frac{n(n-1)}{\log n(n-1) + 1} \ = O\left(\frac{n^2}{\log n}\right).$$

We see then that just to be able to represent all the parity matrices, we need circuits that have length at least of order $n^2/\log n$, and hence there will be parity matrices that require at least that many CNOT gates to write down.

Our naive Gaussian elimination strategy gave us n^2 CNOT operations, and we see that this lower bound is $n^2/\log n$. It would certainly be nice if we could do some smarter elimination, just to squeeze out that last bit of logarithmic factor. And we can!

The idea is that we don't want to eliminate the matrix just element per element, but instead we divide each row up into small 'chunks'. We then first try to eliminate whole chunks at a time. For instance, suppose we pick our chunk size to be 3, then we would first consider the first 3 columns at a whole, and we would look for any duplicate 3-bit strings in these columns. If we find any, then we apply a row operation to get rid of this whole bit string. Once we have done that, we proceed to eliminate these 3 columns as usual, and then go on to the second set of 3 columns.

The benefit of doing this, is that when the chunk size is c, there are only 2^c different types of sub-rows, meaning that of all the n elements in a row, at most 2^c can be non-zero after the chunk elimination, and hence the further standard elimination step only requires $c2^c$ row operations to clear out the c columns. The cost of first doing the chunk elimination is n (ignoring some details here regarding chunks being counted twice across the diagonal), and

both these steps need to happen n/c number of times (assuming c divides n), once for each of the chunked columns. We also have a cost of n total to insert the right diagonal elements. The total cost is then

$$(c2^c)\frac{n}{c} + n\frac{n}{c} + n \;=\; \frac{n^2}{c} + n2^c + n.$$

Setting $c = \alpha \log n$ for some $\alpha < 1$ then simplifies this cost to $O(\frac{n^2}{\log n})$. Note that $\alpha < 1$ is needed for the $n2^c = n^{1+\alpha}$ term to be asymptotically smaller then $n^2 \log n$.

This hence gives us an algorithm with an asymptotically optimal CNOT count!

Theorem 4.5.2. Every CNOT circuit can be resynthesised using at most $O(n^2/\log n)$ CNOT gates.

This algorithm isn't just theoretically interesting: in practice it already gives better results than naive Gaussian elimination for very small number of qubits.

4.6   References and further reading

CNOT circuit synthesis An efficient CNOT circuit synthesis procedure based on Gaussian elimination was proposed by Alber et al. (2001). This was later refined by Markov, Patel, and Hayes in (Markov et al., 2008). This algorithm is the one described in Section 4.5.1. This algorithm remains one of the best for unconstrained architectures, obtaining an asymptotically optimal gate count. Recently, there has also been work on synthesising CNOT circuits for architectures with topological constraints, such as superconducting devices. In these approaches, a graph is provided whose nodes are qubits and edges are places where CNOTs may be placed. This can be accomplished by constraining which row operations are allowed in Gaussian elimination using **Steiner trees**, a technique proposed simultaneously by Kissinger and de Griend (2020) and Nash et al. (2020). A different technique, based on $\mathbb{F}_2$-linear decoding, was given by de Brugière et al. (2020).

Interacting bialgebras The phase-free ZX calculus has also been called **IB**, for **interacting bialgebras**, in the categorical algebra literature (Bonchi et al., 2014). This is because this set of generating maps:

$$(4.16)$$

as well as this set:

 (4.17)

each form an algebraic structure called a **bialgebra**. Bialgebras, and the more specific structure of Hopf algebras, have been extensively studied in representation theory, as they generalise group algebras. This system is called *interacting* bialgebras because the two bialgebras (4.16) and (4.17) interact with each other, via the spider fusion laws. In algebraic terms, this means the four Z generators and the four X generators each form **Frobenius algebras**.

Normal forms In (Bonchi et al., 2014), the authors showed that any diagram built out the generators (4.16) and (4.17), i.e. any phase-free ZX diagram, can be put into one of two normal forms, which they called the *span* and *cospan* form. These are closely related to the Z-X and X-Z normal forms we used in this chapter. They also showed that the resulting maps from m wires to n wires are in 1-to-1 correspondence with $\mathbb{F}_2$-**linear relations**, i.e. subspaces $S \subseteq \mathbb{F}_2^{m+n}$. Up to bending wires, this is what we showed in Section 4.3.

PROPs This work lives in the broader field of studying **PROPs**, or PROduct categories with Permutations, a categorical formulation of algebraic structures whose generators can have many inputs or outputs. PROPs were first formulated by Mac Lane MacLane (1965) and an important technique for composing PROPs together was developed by Lack Lack (2004).

The techniques used to get normal forms in (Bonchi et al., 2014) used abstract arguments based on composing PROPs, and did not directly show how to compute normal forms by applying graphical rules. The rewriting strategy used for obtaining normal forms in Section 4.2.1 is new to this book. Similar strategies have been known, but not published, for a while. For example, the graphical proof assistant Quantomatic (Kissinger and Zamdzhiev, 2015) and the ZX library PyZX (Kissinger and van de Wetering, 2020a) both have implementations of such strategies, with the former implementation dating back to the mid-2010s.

5

Clifford circuits and diagrams

In the previous chapter we saw that there were already a lot of interesting things to say about phase-free ZX-diagrams. However, interesting as they are, because there are no phases, these diagrams don't allow us to do many cool quantum computing things. In this chapter we will remedy this problem and introduce some phases back into the picture.

Instead of immediately allowing all possible phases, we will expand our scope to the **Clifford ZX-diagrams**. These are ZX-diagrams whose angles are all integer multiples of $\pi/2$. At first, this might seem somewhat artificial, but we'll see that in this special setting, even huge diagrams can always be simplified all the way down to a compact canonical form, called **GSLC form**. It turns out this reduced diagram only has $O(n)$ spiders in it and $O(n^2)$ wires, where n is the number of qubits. A couple of magical things happen as a consequence. In particular, virtually everything we would want to do with such a diagram is efficient, from computing single matrix elements to comparing two such diagrams for equality. Related to this: any circuit which can be translated into a Clifford ZX-diagram can be efficiently classically simulated. We call such circuits **Clifford circuits**, and this is (a version of) the famous **Gottesman-Knill theorem**. Just because we can efficiently reason about Clifford diagrams does not mean they are not useful, far from it! We will see that Clifford diagrams form the backbone in measurement-based quantum computing (Chapter 8) and quantum error correction (Chapter 11).

In a standard textbook on quantum computing, Clifford circuits and the technique for efficiently classically simulating them would be introduced in the context of *stabiliser theory*, a powerful collection of tools based on group-theoretic properties of the Pauli group. These techniques are important and ubiquitous in the quantum information literature, and we'll go through them in detail in Chapter 6. However, it is interesting to note that we also have a

purely graphical bag of tricks based on the ZX-calculus that are extremely useful for working with Clifford circuits, and in fact already suffice to prove the Gottesman-Knill theorem. Thus, in this chapter, we'll deal with Clifford circuits using just the ZX-calculus.

5.1 Clifford circuits and Clifford ZX-diagrams

In the previous chapter, we established a close correspondence between CNOT circuits and phase-free ZX-diagrams (see Section 4.2). Namely, we saw that any CNOT circuit translates into a phase-free ZX-diagram, and conversely any unitary phase-free ZX-diagram is equal to a CNOT circuit (Theorem 4.2.13).

In this chapter we will see that a similar relationship exists if we generalise from only allowing the phase 0 to allowing phases from the set $\{0, \frac{\pi}{2}, \pi, -\frac{\pi}{2}\}$. On one side of this correspondence, we have the following family of ZX-diagrams.

Definition 5.1.1. A **Clifford ZX-diagram** is a ZX-diagram where all the phases on the spiders are integer multiples of $\frac{\pi}{2}$.

On the other side, we have a certain family of circuits, which generalises CNOT circuits.

Definition 5.1.2. A **Clifford circuit** is a circuit constructed from the following gates:
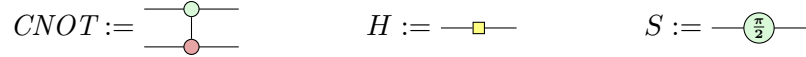
We will also refer to unitaries that can be built by composing these gates as **Clifford unitaries**.

It immediately follows that any Clifford circuit yields a Clifford ZX-diagram (as the Hadamard can be decomposed into a series of $\frac{\pi}{2}$ rotations, see Eq. (3.81)). That any *unitary* Clifford ZX-diagram can be realised by a Clifford circuit is less obvious, and proving this will in fact be one of the primary goals of this chapter.

Exercise 5.1. A single-qubit Clifford circuit is constructed out of just Hadamard and S gates. Show that any single-qubit Clifford circuit can be rewritten to the form
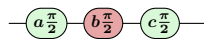
for some integers *a*, *b* and *c. Hint: We know that a single S or Hadamard can be brought to this form. So you just need to show that when you compose this normal form with an additional S or Hadamard gate that the resulting circuit also be brought to this normal form. You probably will want to make a case distinction on the value of b.*

Definition 5.1.3. A **Clifford state** is a state which can be realised as $U|0\cdots0\rangle$ for some Clifford unitary U.

Exercise 5.2. Show that the following states are Clifford states. I.e. construct a Clifford circuit C that when applied to $|0\cdots0\rangle$ gives the desired state.

a) $|1\rangle$.
b) $|+\rangle$.
c) $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.
d) $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$.
e) $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$.

5.1.1 Graph-like diagrams

When dealing with phase-free diagrams it turned out to be useful to simplify our diagrams somewhat in order to work more easily with them: we fused all the spiders and got rid of self-loops and double edges to get diagrams that we called *two-coloured* (Definition 4.2.3). In this chapter it will be useful to introduce a variation on this that allows us to more easily work with Hadamards. This type of diagram only contains Z-spiders. This can be achieved by changing the colour of every X-spider using the Hadamard colour-change rule (**cc**). We can then cancel all double Hadamards that appear by the (**hh**) rule and fuse all connected Z-spiders using (**sp**). In the resulting diagram all connections of spiders will then go via a Hadamard. It will be useful to have a special name and notation for this.

Definition 5.1.4. When two spiders in a ZX-diagram are connected via a Hadamard, we can denote this using a blue dotted line:

$$\vdots \;\bullet\!\!-\!\!-\!\!-\!\!-\!\!\bullet\; \vdots \;\; := \;\; \vdots \;\bullet\!\!-\!\!\square\!\!-\!\!\bullet\; \vdots \tag{5.1}$$

We call such a connection a **Hadamard edge**.

Hadamard edges have a couple of useful properties. First, we only have to deal with at most a single Hadamard edge between a pair of spiders, since any parallel pair of Hadamard edges cancels.

Lemma 5.1.5.
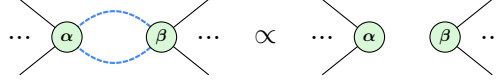
Proof
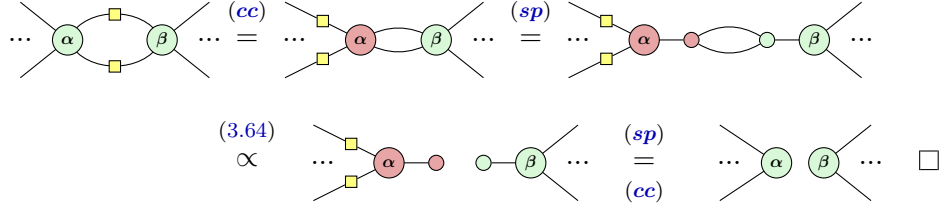
Second, we can always remove a "Hadamard edge self-loop."

Lemma 5.1.6.

Proof   This is just Eq. (3.83). □

Now let's give the definition of this special class of diagrams we are interested in.

Definition 5.1.7. We say a ZX-diagram is **graph-like** when

- Every spider is a Z-spider.
- Spiders are only connected via Hadamard edges.
- There are no self-loops or parallel edges.
- Every Z-spider is connected to at most one input and at most one output.
- Every input and output wire is connected to a Z-spider.

Every ZX-diagram can be efficiently reduced to an equivalent graph-like diagram.

Proposition 5.1.8. Let D be an arbitrary ZX-diagram. Then the following sequence of steps reduces D efficiently to an equivalent graph-like diagram.

1. Convert all X-spiders to Z-spiders using the (**cc**) rule.
2. Cancel all pairs of adjacent Hadamards using the (**hh**) rule.
3. Fuse all spiders by applying the (**sp**) rule until it can no longer be applied.
4. Remove parallel Hadamard edges using Lemma 5.1.5.
5. Remove self-loops using Eq. (3.39), and remove Hadamard self-loops using Lemma 5.1.6.

6. Introduce Z-spiders and Hadamards using (**id**) and (**hh**) in reverse, in order to ensure every input and output is directly connected to a Z-spider and no Z-spiders are connected to multiple inputs/outputs.

Proof   It is clear that after this procedure there are only Z-spiders, as all X-spiders have been converted. Every connection between spiders must be via a Hadamard edge, since if it were a regular connection, then the spiders would have been fused in step 3, and if there were multiple Hadamards on the connection, then pairs of them would be cancelled in step 2. There is at most one Hadamard edge between each pair of spiders because of step 4, and there are no self-loops because of step 5. It could now still be that we have input/output wires that are connected via a Hadamard to a Z-spider, or wires that are not connected to a spider at all. We take care of these cases as follows:

Similarly, if a Z-spider is connected to multiple inputs or outputs, we can introduce an extra Z spider and a pair of Hadamards to fix it up:

Each of the steps of this algorithm touches each vertex or edge at most once, and hence this is all efficient in the size of the diagram. □

Example 5.1.9. We demonstrate how an example ZX-diagram is transformed into a graph-like diagram below. In the last step we write down its underlying graph.
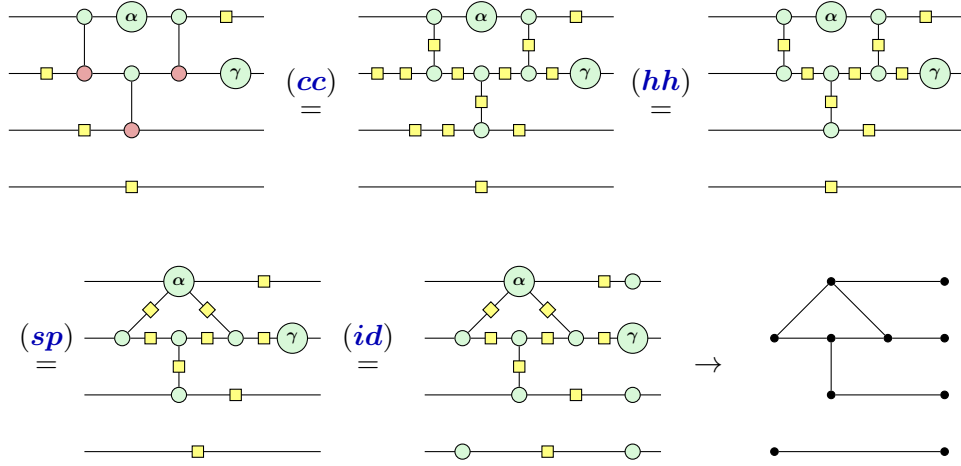
The reason we call these diagrams graph-like is because they are neatly described by a structure that we call an *open graph*, a graph with a specified list of inputs and outputs. We will have more to say about open graphs in Chapter 8. For now, let's just note that we can view each spider in a graph-like diagram as a vertex, and every Hadamard as an edge.

Exercise 5.3.

a) Show that every two-coloured diagram (see Definition 4.2.3) is transformed into a graph-like diagram by changing all the X-spiders into Z-spiders using (**cc**), and then potentially introducing some more identities with (**id**) and (**hh**) to the input and output wires.

b) Show that the converse is not true: find a graph-like ZX-diagram where it is not possible to rewrite it back into a two-coloured diagram using just (**cc**).

c) Find a systematic way in which a graph-like diagram can be transformed into a two-coloured diagram.

5.1.2 Graph states

A particularly useful subset of graph-like diagrams are the *graph states*. We can either describe these diagrammatically as a restricted set of graph-like diagrams, or directly as a type of quantum state. We will give the description as a quantum state first.

For each simple undirected graph $G = (V, E)$, where V denotes the set of vertices of the graph, and E the set of edges, we can define its corresponding

graph state $|G\rangle$ as

$$|G\rangle := \prod_{(v,w)\in E} \mathrm{CZ}_{v,w}|+\rangle^{\otimes|V|}.$$

I.e. we prepare the $|+\cdots+\rangle$ state, where the number of qubits is equal to the number of vertices in the graph, and then for every edge in G we apply a CZ gate between the corresponding qubits (recall the CZ from Exercise 2.13).

A $|+\rangle$ state as a ZX-diagram is just a phase-free Z-spider with a single output. A CZ gate is a pair of Z-spiders connected via a Hadamard gate:

To go from a graph to a graph state represented as a ZX-diagram is then straightforward:

$$(5.2)$$
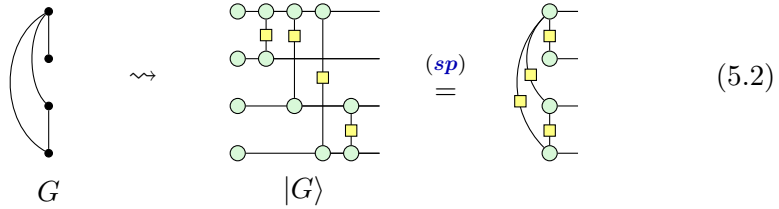
Here we get the diagram on the right by simply fusing all the spiders together. To go from a graph to the graph state represented as a ZX-diagram we see then that every vertex becomes a phase-free Z-spider with an output attached to it and each edge in the graph becomes a Hadamard edge between the corresponding spiders.

With this description in hand we can also define a graph state as a particular type of graph-like diagram (spend some time convincing yourself that this definition is indeed equivalent to the description given above):

Definition 5.1.10. A graph-like diagram is a **graph state** when

- it has no inputs,
- every spider is connected to an output,
- and all phases are zero.

Some useful quantum states are not exactly graph states, but are instead merely *very close* to being a graph state. For instance, the GHZ-state (cf. (3.40)), is not a graph state, but we can construct it by starting with a graph state and then applying some Hadamard gates on a couple of the

qubits:

$$
\text{(5.3)}
$$

In the context of graph states, we refer to the application of single-qubit unitaries on some of its outputs as **local** unitaries. Imagine for instance some quantum protocol where we start with a bunch of qubits in one spot, which are then entangled to make a graph state. We then give each of the qubits of this graph state to a different person, who are then allowed to take their qubit very far away from the others. While each of these people can't change the graph state by applying some operation on multiple qubits at once, they can still modify the qubit they have access to by applying a unitary on just that qubit. Hence why we refer to single-qubit operations as 'local'.

Note furthermore that the operation we had to apply in the case of the GHZ state was not just any unitary but specifically a Clifford unitary. This leads us to our next definition of a useful subclass of states and ZX-diagrams.

Definition 5.1.11. A **graph state with local Cliffords** (GSLC) is a graph state to which some single-qubit Clifford unitaries have been applied on its outputs.

We will often say a ZX-diagram is in **GSLC form** or (at the risk of sounding like the kind of people that say "ATM machine") that a quantum state is a **GSLC state**.

Every single-qubit Clifford unitary can be written as a composition of three phase gates where the phases are multiples of $\frac{\pi}{2}$ (cf. Exercise 5.1). So an example of a GSLC state would be a composition of the graph state of (5.2) with any $Z(\frac{\pi}{2})$ or $X(\frac{\pi}{2})$ phase gates:
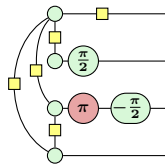
Where here the Hadamard gate on the top qubit is a number of $\frac{\pi}{2}$ phase gates in disguise.

Exercise 5.4. Show that the graph state (5.2) is a Clifford state by finding a Clifford circuit that builds it when applied to $|0 \cdots 0\rangle$.

Exercise 5.5. A graph state has no internal spiders, but a general graph-like diagram does. Show that any graph-like diagram with no inputs (i.e. a

state) can be written as a graph-state where each internal spider becomes a post-selection by adapting the arguments from Section 3.4.1.

5.2 Simplifying Clifford diagrams

Now that we have all the tools we need, we will see how we can systematically simplify Clifford diagrams. Instead of rewriting arbitrary diagrams, we will work with graph-like diagrams. Thanks to Proposition 5.1.8, we can translate any ZX-diagram into a graph-like one, so we can do this without loss of generality.

Because graph-like ZX-diagrams are pretty much just graphs with a bit of extra stuff, we can use some techniques from graph theory to simplify them. Before we do this in the general case, it pays to first look at the special case of GSLC states.

5.2.1 Transforming graph states using local complementation

When looking at a graph state it might seem at first glance that the presence or absence of an edge between two qubits determines whether those qubits are entangled. Considering again the scenario where we have prepared a graph state and sent each of the qubits far away from each other. This might then seem to mean that there is no way in which we can get any more entanglement between these qubits. However, the structure of entanglement in a graph state can be deceiving. It turns out that just applying local Cliffords can greatly affect the connectivity of the underlying graph. For instance, in Eq. (5.3) we saw that we can represent a GHZ state as a graph state with some local Cliffords. Because the GHZ is of course symmetric in all three qubits, that means we can do this in three equivalent ways, and hence the following graph states are all equal up to the application of some local Cliffords:
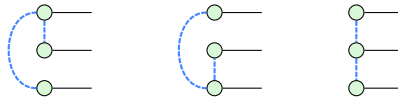
A natural question is then how we can find out when two graph states can be transformed into each other just using local Clifford operations.

One important graph transformation we can do just by using local operations is the *local complementation.*

Definition 5.2.1. Let G be a graph, and u a vertex in G. The **local complementation of G about u**, which we write as $G \star u$, is the graph

which has the same vertices and edges as G, except that the neighbourhood of u is complemented. In other words, two neighbours v and w of u are connected in $G \star u$ if and only if they are *not* connected in G.

Example 5.2.2. Consider the graph G below with vertices a, b, c, d. In $G \star a$ we see that the neighbourhood of a, consisting of the vertices b, c, d is complemented. So because b and c are not connected in G, they are connected in $G \star a$, and because b and d *are* connected in G, they are not in $G \star a$. In $(G \star a) \star b$ we see that the connection between c and d is not affected, as d is not a neighbour of b.
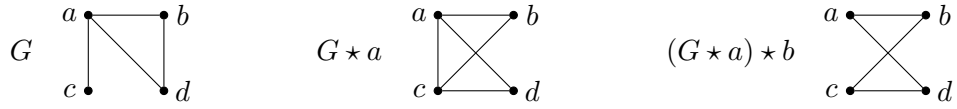
To transform a graph state so that its underlying graph is locally complemented about a vertex (i.e. qubit) u, we only need to apply a $X(-\frac{\pi}{2})$ gate on u, and a $Z(\frac{\pi}{2})$ gate on each of its neighbours:

$$(5.4)$$

Here $N(u)$ denotes the neighbourhood of u.

In the remainder of this section we will prove that (5.4) indeed holds. To do this it will be helpful to introduce the family of fully connected ZX-diagrams.

Definition 5.2.3. We define K_n to be the **fully connected ZX-diagram** on n qubits, defined recursively as:

$$(5.5)$$

When we fuse all the spiders in K_n we see that they indeed give totally-connected graphs of Hadamard edges:
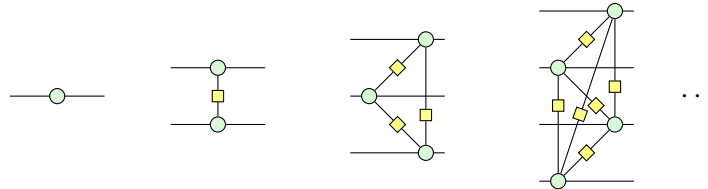
Using this definition of K_n we can state the equality that will allow us to do local complementations:

Lemma 5.2.4. The following holds in the ZX-calculus for all $n \in \mathbb{N}$:

$$\tag{5.6}$$

Before we prove this, see Example 5.2.5 for a demonstration of how this is related to doing local complementations. The crucial point is that the introduction of a fully connected graph by Eq. (5.6) makes a parallel Hadamard edge if there was already a Hadamard edge present, which is then subsequently removed by Lemma 5.1.5.

Example 5.2.5. Let us take the graph G from Example 5.2.2, but now seen as the graph state $|G\rangle$.

We indeed end up with $|G \star a\rangle$ (up to local Cliffords).

For the proof of Lemma 5.2.4 we will need the following base case.

Exercise 5.6. Prove the following using the ZX-calculus.

$$\tag{5.7}$$

Hint: Push the rightmost Hadamards to the right and decompose the middle Hadamard using one of the versions of the Euler decomposition from Exercise 3.12 to reveal a place where you can apply strong complementarity.

And now we can prove the general case.

Proof of Lemma 5.2.4. Note that for $n = 0$ and $n = 1$ this equation becomes:

Exercise 5.6 shows $n = 2$. We prove the other cases by induction. For our induction hypothesis, assume (5.6) holds for some fixed $n \geq 2$, which we indicate as (ih) below. Then, for $n + 1$ we calculate:

Exercise 5.7. We say two n-qubit quantum states $|\psi_1\rangle$ and $|\psi_2\rangle$ are **equivalent under local operations** when $U|\psi_1\rangle = |\psi_2\rangle$ for a local quantum circuit $U = U_1 \otimes U_2 \otimes \cdots \otimes U_n$ consisting of just single-qubit gates. Show that the following two graph states are equivalent under local operations.

Hint: Use the fact that a local complementation can be done using just local unitaries.

5.2.2   Pivoting

It turns out that it is often useful to not just do a single local complementation, but to do a particular sequence on a pair of connected vertices.

Definition 5.2.6. Let G be a graph and let u and v be a pair of connected vertices in G. We define the **pivot of G along uv**, written as $G \wedge uv$, as the graph $G \star u \star v \star u$.

Note that in this definition, the ordering of u and v does not matter.

Exercise 5.8. Show that for any graph G and connected pair of vertices u and v in G we have $G \star u \star v \star u = G \star v \star u \star v$.

On a graph, pivoting consists in exchanging u and v, and complementing the edges between three particular subsets of the vertices: the common neighbourhood of u and v (i.e. $N_G(u) \cap N_G(v)$), the exclusive neighbourhood of u (i.e. $N_G(u) \setminus (N_G(v) \cup \{v\})$), and exclusive neighbourhood of v (i.e. $N_G(v) \setminus (N_G(u) \cup \{u\})$). Schematically:

As a pivot is just a series of local complementations, it can also be performed on a graph state by the application of a particular set of local Cliffords. Indeed, in terms of ZX-diagrams, we have:

$$(5.8)$$

I.e. we can perform a pivot on the graph state by applying a Hadamard to the vertices we pivot along, and applying a Z gate to the vertices in their common neighbourhood.

Exercise 5.9. Prove Eq. (5.8) by decomposing the Hadamards into sequences of $\frac{\pi}{2}$ phase gates and then applying a sequence of local complementations using Eq. (5.4).

It turns out we can prove (5.8) more directly by using strong complementarity.

Lemma 5.2.7. Eq. (5.8) holds in the ZX-calculus.

Proof For clarity, let us first assume that the set of vertices connected to both u and v is empty.

$$(5.9)$$
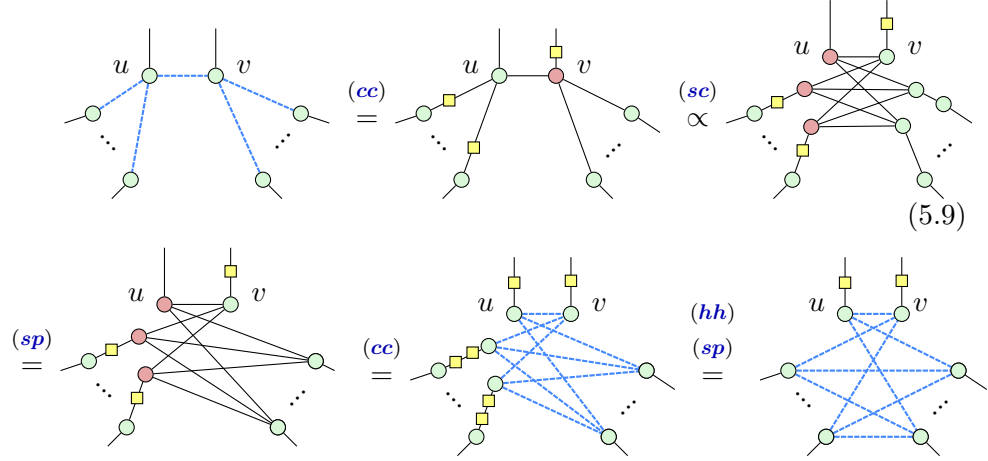
We see that u ends up connected to whatever v was connected to and vice versa, and that the neighbourhoods of u and v are now fully connected, so that if there were connections they will get complemented in the same way as described in Example 5.2.5.

Now, if there are spiders that are connected to both u and v, then we can unfuse spiders in a clever way to reduce it to the previous case:
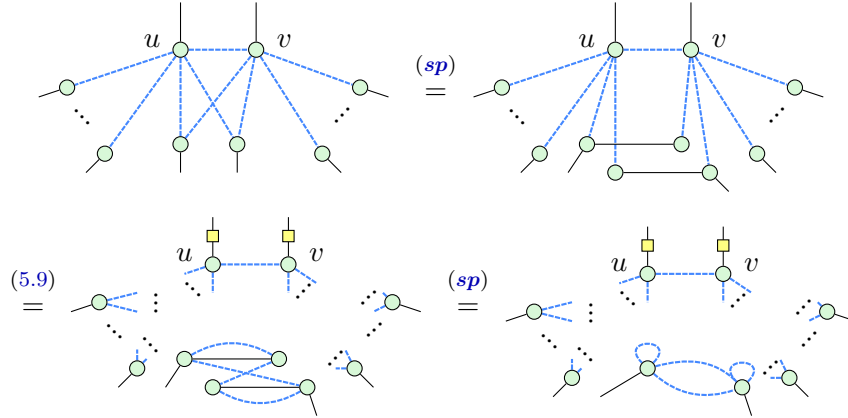
For clarity we have only drawn the Hadamard edges that stay within the joint neighbourhood of u and v in the bottom two diagrams. We see that we end up with parallel Hadamard edges that can be removed using Lemma 5.1.5. The Hadamard self-loops are simplified to $Z(\pi)$ phases using Lemma 5.1.6, which indeed gives the expected result. □

5.2.3 Removing spiders in Clifford diagrams

In Sections 5.2.1 and 5.2.2 we saw that we can apply the graph operations of local complementation and pivoting on graph states by introducing some local Cliffords. It turns out that we can use some variations on these rules to greatly simplify graph-like diagrams. As this chapter deals with Clifford diagrams, we will focus here on the variations that are useful to simplify these diagrams, but later on we will introduce some additional variations that can also simplify generic graph-like diagrams.

   The rules we introduce in this section all serve to remove one or more spiders from a Clifford diagram. By repeatedly applying these rules we then get smaller and smaller diagrams until there are no longer any spiders to remove using these rules. For these rewrite rules it will be useful to introduce a distinction between two classes of spiders in graph-like diagrams (see also Definition 4.2.5).

Definition 5.2.8. Let D be a graph-like diagram. We say a spider in D is **internal** if it is not an input or output spider (i.e. it is not connected to any input or output wire). Conversely, we say a spider is a **boundary** spider when it *is* connected to at least one input or output wire.

   Our first simplification rule is based on the local complementation rule (5.6).

Lemma 5.2.9. The following local complementation simplification hold:

$$(5.10)$$

Proof We pull out all of the phases via (**sp**) then apply the local complementation rule (5.6) (from right to left) on the vertex marked by ($*$):

Using Eq. (3.84), the topmost spider in the right-hand side above becomes an X-spider, with phase $\mp\pi/2$, which combines with the phase below it into

an $a\pi$ phase, where $a = 0$ if we started with $\pi/2$ and $a = 1$ if we had started with $-\pi/2$. The resulting X-spider copies and fuses with the neighbours:
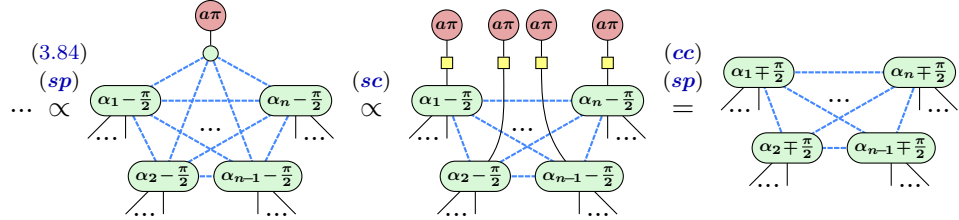
In words we can describe this rule as follows: if we have a spider (here marked on the left-hand side by a $*$) with a $\pm\frac{\pi}{2}$ phase in a graph-like diagram that is internal, i.e. that is not connected to inputs or outputs but only to other spiders, then we can remove it from the diagram by complementing the connectivity on its neighbourhood and changing some phases. The reason we complement the neighbourhood, is because in Lemma 5.2.9 we get a fully connected graph on the right-hand side, but if there were already edges present between some of the spiders, then the resulting double edges would be removed by Lemma 5.1.5, so that we indeed complement the edges in the neighbourhood. For the remainder of this chapter, when we say we 'apply' Lemma 5.2.9 we mean that we apply it from left to right on some chosen vertex, and that we immediately follow it by Lemma 5.1.5 in order to cancel the introduced parallel edges, so that we are still left with a graph-like diagram.

Remark* 5.2.10. In this rule we ignored non-zero scalar factors (like we always do). However, when we applied Eq. (3.84), the actual equation with the correct scalar, Lemma 3.6.8, introduces an additional scalar $\frac{\pi}{2}$ spider. So in this sense, Lemma 5.2.9 is not really removing the $\pm\frac{\pi}{2}$ spider, as it is just interchanging it for a $\pm\frac{\pi}{2}$ spider that is not connected to any other spider, and hence is just a simple scalar. This is important for when we discuss completeness in Section 5.5.

In a Clifford diagram each spider has a phase $k\frac{\pi}{2}$ for some $k \in \mathbb{Z}$. Using the rule above repeatedly on a graph-like Clifford diagram we can remove *all* internal spiders with a $\pm\frac{\pi}{2}$ phase. Hence, the only internal spiders that remain are those that have a 0 or π phase. Most of these internal spiders can also be removed, by using a variation on the pivot rule (5.8).

Lemma 5.2.11. The following pivot simplification holds:
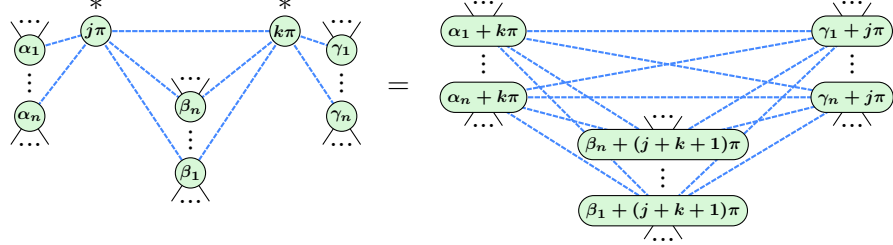
Proof We pull out all of the phases via (**sp**) and apply the pivot rule Lemma 5.2.7:
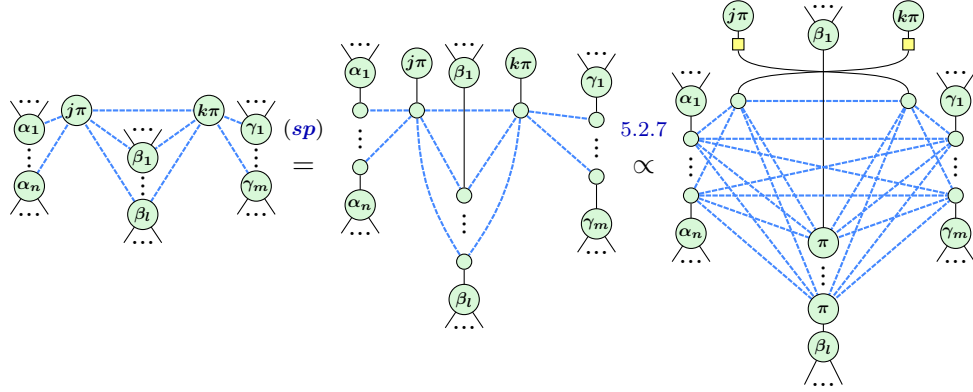
We then apply the colour-change rule to turn the Z-spiders with phases $j\pi$ and $k\pi$ into X-spiders. They can then be copied, colour-changed again and fused with their neighbours:
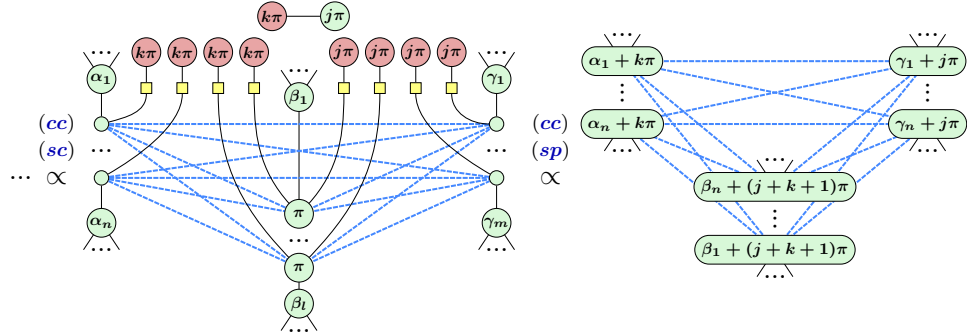
Note that the dangling scalar diagram appears because we copy twice and the vertices are connected. We simply ignore this non-zero scalar. $\qquad\square$

Here, the marked spiders on the left-hand side are internal connected spiders with a 0 or π phase. On the right-hand side, these spiders are removed,

at the cost of complementing their neighbourhood in the manner described by the pivot rule, and introducing some phases (again, the complementation happens because fully connected bipartite connectivity is introduced, and parallel edges are then removed using Lemma 5.1.5).

Exercise 5.10. Simplify the following circuit to a diagram that has no internal spiders with a $\pm\frac{\pi}{2}$ phase or pairs of internal spiders with a 0 or π phase.
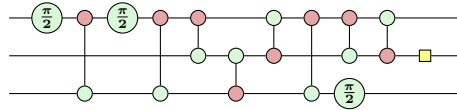
5.3 Clifford normal forms

These simplification lemmas allow us to remove many of the spiders in a Clifford diagram. In fact, so many that the resulting types of diagrams deserve to be called *normal forms* for Clifford diagrams. In this section we will see two types of normal forms. The first is what you get if you just keep applying the local complementation and pivoting simplifications. The second requires an additional type of pivot operation that removes the final internal spiders.

5.3.1 The affine with phases normal form

Lemma 5.2.9 removes those internal spiders with a phase of $\pm\frac{\pi}{2}$ so that if we started with a Clifford diagram, the only phases left on internal spiders are 0 are π. Then Lemma 5.2.11 can apply to any remaining internal spider that is connected to at least one other internal spider. Hence, once we are done applying Lemmas 5.2.9 and 5.2.11 on a Clifford diagram the only remaining internal spiders are then those that carry a 0 or π phase and are connected only to boundary spiders. These diagrams turn out to be rather useful, so let's give them a name.

Definition 5.3.1. We say a graph-like Clifford diagram is in **affine with phases** form (AP form) when:

1. every boundary spider is connected to exactly one input or output,
2. every internal spider has a phase of 0 or π, and
3. no two internal spiders are connected to each other.

Our previous summary of the simplification procedure can then be summarised as follows.

Proposition 5.3.2. We can efficiently rewrite any Clifford diagram into an equivalent AP form.

In an AP form we have two groups of spiders. We have the boundary spiders and we have the internal spiders. The boundary spiders can be connected to any other spider (via a Hadamard edge) and carry any phase, but the internal spiders are *only* connected to the boundary spiders and can only have a phase of 0 or π.

Example 5.3.3. An example AP form:
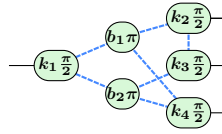
for $b_j \in \{0, 1\}$ and $k_j \in \{0, 1, 2, 3\}$.

Thanks to condition 1 of Definition 5.3.1, AP forms don't treat inputs and outputs differently. Thus, from hence forth we will primarily study *states* in AP form. Here, we implicitly use the fact that we can treat arbitrary AP form maps as states by 'bending wires' to turn inputs into outputs. For example, the map from Example 5.3.3 can be treated as a state as follows:
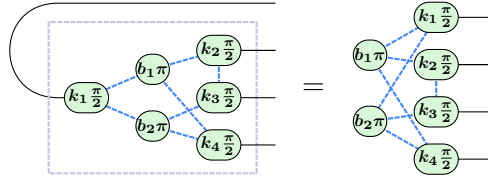
Why do we call these diagrams 'affine with phases'? To answer this we need to look at what types of states they encode. There are a couple of different things going on here, so for simplicity we'll start with just the 'affine' part of AP forms then build up to the general case. Recall from Section 4.3.2 that a phase-free X-Z normal forms give us a state defined by a system of linear equations:

$$\text{} \quad \propto \quad \sum_{\vec{x} \in S} |\vec{x}\rangle \qquad \text{where} \qquad S = \left\{ \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \ \middle| \ \begin{matrix} x_1 \oplus x_2 \oplus x_3 = 0 \\ x_1 \oplus x_3 = 0 \end{matrix} \right\}$$

Importantly, S is always defined by a *homogeneous* system of linear equations, meaning the right-hand side of every equation is 0. Equivalently, it is the set of bit vectors $\vec{x}$ satisfying $M\vec{x} = 0$ for some $\mathbb{F}_2$-matrix M. We can generalise this form by additionally allowing the X spiders to carry 0 or π phases.

This gives us almost the same thing, except now S can be defined by an *inhomogeneous* system of linear equations. Each X spider with a 0 phase gives us an equation with a 0 on the right-hand side, whereas an X spider with a π phase gives us an equation with a 1 on the right-hand side. For example:

$$
\vcenter{\hbox{}} \quad \propto \sum_{\vec{x} \in S} |\vec{x}\rangle \qquad \text{where} \qquad A = \left\{ \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \; \middle| \; \begin{matrix} x_1 \oplus x_2 \oplus x_3 = 1 \\ x_1 \oplus x_3 = 0 \end{matrix} \right\}
$$

(5.11)

That is, we get the set of vectors $\vec{x}$ satisfying $M\vec{x} = \vec{b}$ for some fixed matrix M and vector $\vec{b}$. In the example above, its:

$$
M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \qquad\qquad \vec{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}
$$

Whereas the set of all solutions to a system of homogeneous linear equations always gives us a *linear* subspace of $\mathbb{F}_2^n$, the solutions to an inhomogeneous system will, in general, form an affine subspace. Intuitively, an affine subspace is like a linear subspace that has been shifted away from the origin by some fixed amount.

Definition 5.3.4. An **affine subspace** $A \subseteq \mathbb{F}_2^n$ is either:

- the empty set, or

- a set of the form $\vec{w} + S := \{\vec{w} + \vec{v} \,|\, \vec{v} \in S\}$ for some fixed vector $\vec{w} \in \mathbb{F}_2^n$ and a linear subspace $S \subseteq \mathbb{F}_2^n$.

Just like in the case of linear spaces, we have two equivalent representations for an affine subspace, either in terms of a spanning set of vectors or a system of equations. As before, these correspond to Z-X normal forms and X-Z normal forms, respectively. In both cases, the extra data needed to define an affine space (as opposed to a linear one) is included by introducing π phases

on to some of the X spiders.

$$\propto \quad \sum_{\vec{x}\in A}\ket{\vec{x}} \qquad \text{where} \quad A = \vec{w} + \text{span}\{\vec{v}_1,\dots,\vec{v}_k\}$$

$$\propto \quad \sum_{\vec{x}\in A}\ket{\vec{x}} \qquad \text{where} \quad A = \left\{ \vec{x}\in\mathbb{F}_2^n \ \middle|\ \begin{array}{c} \vec{w}_1^T\vec{x}=b_1 \\ \cdots \\ \vec{w}_k^T\vec{x}=b_k \end{array} \right\}$$

(5.12)

As before, we have decorated spiders with vectors to mean that there is an edge to the j-th spider if the j-th entry of the associated vector is 1. Note the second row is a general form for (5.11), since $\vec{w}_i^T\vec{x}$ gives the XOR of the variables x_j for which $(\vec{w}_i)_j = 1$. Equivalently, the vectors $\vec{w}_i$ correspond to the rows in a matrix M such that $A = \{\vec{x}|M\vec{x}=\vec{b}\}$.

Exercise 5.11. The bitstrings appearing in the superposition in an AP state are described by the solutions to the affine system of equations $M\vec{x}=\vec{b}$. When we do row operations on M and $\vec{b}$ (as in a Gaussian elimination of the linear system) this does not change the solutions, and so this transformed system $(M',\vec{b}')$ should still describe the same state. As the matrix M corresponds to the connectivity of the internal spiders to the boundary spiders, show that these row operations can be implemented diagrammatically:
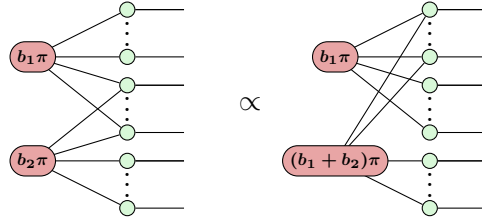
If we colour-change the X spiders, we see that we're most of the way to an AP form:

(5.13)
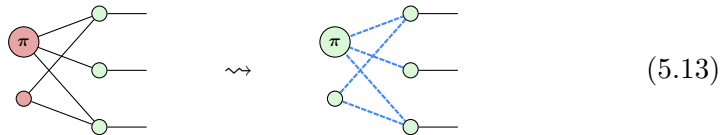
Hence, if we do not have phases on the outputs or Hadamard edges between them, an AP form can always be described by an affine subspace. For a generic AP form, by spider (un)fusion, we can split off the affine part from

the rest, which we'll call the 'phase' part:

$$\underbrace{}_{affine} \quad \overset{phase}{}$$
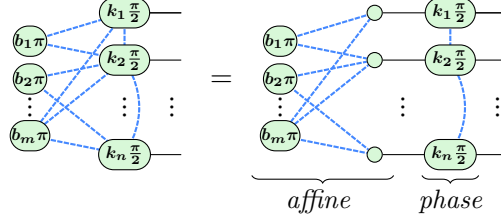
The reason for the name is that the 'phase' part always forms a diagonal unitary, which means all it will do to the state is introduce some phases on the computational basis states $|\vec{v}\rangle$ from (5.13). By unfusing some spiders, we can see that this phase part is generated by S gates, which introduce $\pi/2$ phases to individual outputs, and CZ gates, which introduce Hadamard edges between outputs. We can see that each of these gates affects the phase in a way that depends on the computational basis state:

$$-\boxed{\tfrac{\pi}{2}}- \quad :: \ |x\rangle \mapsto e^{i\frac{\pi}{2}\cdot x}|x\rangle$$

$$\overset{}{\underset{}{\vdots}} \quad :: \ |x_1 x_2\rangle \mapsto e^{i\pi\cdot x_1 x_2}|x_1 x_2\rangle$$

We can describe the action of unitaries built out of these gates using certain polynomials, called **phase polynomials**. For example:

$$:: \ |x_1 x_2 x_3\rangle \mapsto e^{i\frac{\pi}{2}\cdot\phi(x_1,x_2,x_3)}|x_1 x_2 x_3\rangle$$

$$\text{where} \qquad \phi(x_1, x_2, x_3) := x_1 + 2x_2 + 2x_1 x_2 + 2x_1 x_3$$

Here, each single-qubit gate (i.e. the S gate on the first qubit and the $S^2 = Z$ gate on the second) contributes a linear term to ϕ, whereas each two-qubit CZ gate contributes a quandratic term, whose coefficient is always even. Note that, even though these polynomials can contain products of variables, they are always linear in each argument individually. For that reason, we call these phase polynomials in **multilinear form**. In Chapter 7 we will see phase polynomials in **XOR-form**, and in Chapter 9, we will see how these two forms are related.

By applying a diagonal Clifford unitary to an affine state, the phase is applied to each of the terms in the same. So, if we combine the example

above with (5.13) we get:
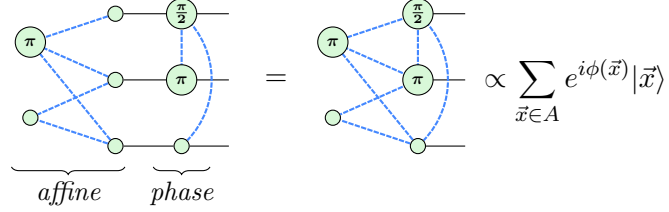
$$\text{where} \quad \begin{cases} A := \{(x_1, x_2, x_3) \mid x_1 \oplus x_2 \oplus x_3 = 1, x_1 \oplus x_3 = 0\} \\ \phi(x_1, x_2, x_3) := \frac{\pi}{2}(x_1 + 2x_2 + 2x_1x_2 + 2x_1x_3) \end{cases}$$

(5.14)

Exercise 5.12. Reduce the following diagram to AP-form:
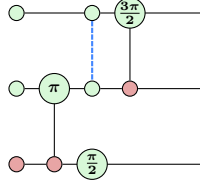
What is its parity matrix and its phase polynomial?

5.3.2 GSLC normal form

In a diagram in AP form there are still some internal spiders. It turns out that we can actually also get rid of these. However this does come at a cost: we must then allow Hadamards on input and output wires, so that the diagram is not what we have defined to be a 'graph-like diagram'. Just as it was useful to define a 'graph state with local Cliffords', it will be useful here to define a diagram that is graph-like up to Hadamards on the boundary wires.

Definition 5.3.5. We say a diagram is **graph-like with Hadamards** when it satisfies all the conditions for being graph-like (Definition 5.1.7), except that inputs and outputs are also allowed to be connected to Z-spiders via a Hadamard.

Note that a graph-like diagram with Hadamards can be easily transformed into a graph-like diagram by introducing some additional Z-spiders using the (*id*) rule.

So how do we get rid of the final internal spiders in a Clifford diagram? Note that each of those spiders has a 0 or π phase and is only connected

to boundary spiders (in particular, it is connected to *at least one* boundary spider, since otherwise it would just be a floating scalar we can ignore). The first step is to introduce some dummy Hadamards and identity spiders to make this boundary spider into an internal spider:

$$\tag{5.15}$$

The diagram is now no longer just graph-like, but graph-like with Hadamards and the spider with the γ phase has become internal. Then we make two case distinctions. If $\gamma = 0$ or π, we have an internal pair of $0/\pi$ spiders, and we can remove them using the pivot simplification Lemma 5.2.11. If $\gamma = \pm\frac{\pi}{2}$, then we can first remove that spider using the local complementation Lemma 5.2.9. As a result of this, the phase of the spider with the $j\pi$ phase becomes $j\pi \mp \frac{\pi}{2}$ so that its phase is also $\pm\frac{\pi}{2}$. We can then also remove this spider using local complementation. In both cases we see that we can remove both spiders, and hence that we get rid of the original internal spider. Note that in the second case, when $\gamma = \pm\frac{\pi}{2}$, the other spiders it is connected to also gain a $\mp\frac{\pi}{2}$ phase, so that this might also give some additional opportunities to remove internal spiders using local complementation.

We can repeat this procedure for any remaining internal spider. This might result in multiple Hadamards appearing on the same input or output wire. In that case we can of course cancel these Hadamards using (**hh**). Combining the simplifications so far we see that we can hence actually remove *all* internal spiders in a Clifford diagram. Let's give a name to such a type of diagram.

Definition 5.3.6. We say a Clifford diagram is in **GSLC form** when it is graph-like with Hadamards and has no internal spiders.

As before, GSLC here stands for *graph state with local Cliffords*. This is a fitting name, because states in GSLC form *are* graph states with local Cliffords as defined in Definition 5.1.11. Indeed, if we have a state, so a diagram with no inputs, that is in GSLC form, then every spider must be connected to an output, possibly via a Hadamard, so after unfusing the phases on the spiders, we see that it is indeed a graph state followed by some single-qubit Clifford unitaries:
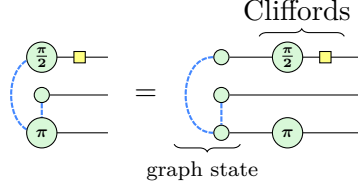
Let's summarise what we have shown in a proposition.

Proposition 5.3.7. Any Clifford diagram can be efficiently rewritten to an equivalent diagram in GSLC form.

As a consequence of this, all *Clifford states*, those states that can be produced from applying a Clifford unitary to $|0\cdots0\rangle$, must be equal to graph states with local Cliffords.

Theorem 5.3.8. Let U a Clifford unitary (i.e. a circuit consisting of CNOTs, Hadamards and S gates). Then $U|0\cdots0\rangle$ is a graph state with local Cliffords.

Proof We can easily represent $U|0\cdots0\rangle$ as a Clifford diagram. Proposition 5.3.7 shows it can be reduced to GSLC form. Unfusing the phases then transforms this into a graph state with local Cliffords. □

In fact, we have something even stronger: we can apply the Clifford circuit to $|0\cdots0\rangle$, and then also *post-select* some outputs to be $\langle0|$ (or $\langle1|$, $\langle+|$ or $\langle-|$ for that matter), and *still* have a graph state with local Cliffords.

Proposition 5.3.9. Let $|\psi\rangle$ be a state produced by applying a Clifford unitary U to $|0\cdots0\rangle$, and then post-selecting some qubits to $\langle0|$. Then $|\psi\rangle$ is (proportional to) a graph state with local Cliffords.

Proof The rewrite strategy to bring the diagram to GSLC form works regardless of how we produced the Clifford ZX-diagram, so we can still apply it in this setting. □

This means that allowing Z-basis measurements on Clifford states doesn't give us something more general or powerful: we get exactly the same class of states.

Finally, let us write down an equivalence between two different notions of Clifford states.

Proposition 5.3.10. Let D be a Clifford diagram without inputs (i.e. a state). Then D is equal to a Clifford state $U|0\cdots0\rangle$ for some Clifford unitary U.

Proof Rewrite D to GSLC form so that it is a graph state with local

Cliffords, and then build U as the circuit that transforms $|0\cdots0\rangle$ into that graph state plus the local Cliffords. □

5.3.3 Normal form for Clifford circuits

In the previous section we saw that if we simplify a Clifford *state*, that we get a graph state with local Cliffords. What happens if we simplify a Clifford *unitary* in the same manner? Again, there are no internal spiders in the diagram, so each spider must be connected to at least one input or output. By introducing some dummy spiders we can ensure that each spider is connected to exactly one input or output. The diagram will then look something like the following:
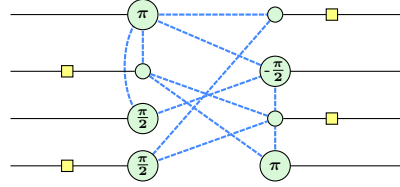
We have possible Hadamards on the inputs and outputs as our reduced diagram is graph-like with Hadamards. Each spider can carry a phase, and each spider is connected to exactly one input or output, so that we have two layers of spiders. There are no further (obvious) restrictions for how the spiders can be connected via Hadamard edges. We will now investigate how we can make diagrams like this look more like circuits.

First, note that we can unfuse the phases on both sides so that the Hadamards and the phases form a layer of local Cliffords:
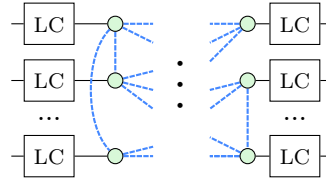
Second, we can unfuse the connections between the spiders in the same layer. This reveals those connections to be CZ gates:
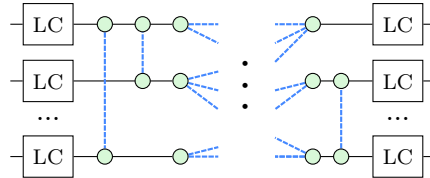
It remains to see what is happening in the middle-part of the diagram. This

should become clear once we change the colour of the layer of spiders on the right:

$$(5.16)$$
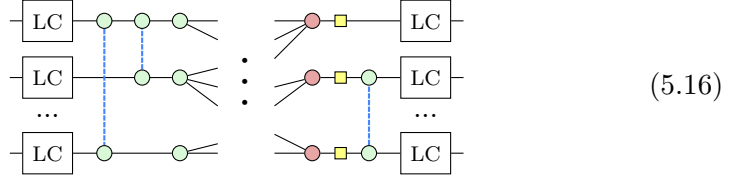
We recognise this middle part of the diagram as a parity normal form (Definition 4.2.2)! So far, all the steps we have done on this GSLC diagram work regardless of whether the diagram is unitary. However, when the diagram is unitary, then this parity normal form must represent a unitary matrix itself, and hence by Proposition 4.2.12, it is equivalent to a CNOT circuit.

Summarising this analysis of GSLC unitary diagrams we see that we can represent any Clifford unitary as a particular sequence of gates.

Theorem 5.3.11. Let U be a Clifford unitary. Then U can be written as a circuit consisting of 8 layers of gates in the following order:

$$\text{Had} - S - \text{CZ} - \text{CNOT} - \text{Had} - \text{CZ} - S - \text{Had}$$

This is really surprising! For one thing, before we started this chapter you might think there are an infinite number of different unitaries you can build from the Hadamard, S and CNOT gate, as you can just make longer and longer circuits, but this result shows that we can always reduce such a long circuit to one consisting of just a few layers of gates. In particular, we need at most a quadratic number of gates in terms of the number of qubits.

Proposition 5.3.12. Any n-qubit Clifford unitary can be represented by an equivalent circuit consisting of at most $4n(n + 3/2) - 1$ Hadamard, S and CNOT gates.

Proof A layer of Hadamard gates contains at most n gates, and there are three of those. Each layer of S gates has at most 3 gates per qubits, corresponding to an S, Z or $S^\dagger$ gate, and there are again two of those layers. Hence, the single-qubit gates contribute at most $9n$ gates.

A circuit of CZ gates contains at most $n(n-1)/2$ CZ gates (corresponding to all possible pairs of qubits there can be a CZ between). A CZ gate can be constructed from a CNOT and two Hadamards, and hence each CZ layer, of which there are two, contributes at most $3 \cdot n(n-1)/2$ gates. The number of CNOTs needed in the CNOT circuit corresponds to how many Gaussian elimination row operations are needed to fully reduce the associated parity

matrix. It is a little exercise to show that you need at most $n^2 - 1$ row operations to fully reduce any matrix.

The total number of needed Hadamard, S and CNOT gates is then:

$$9n + 2 \cdot 3 \cdot n(n-1)/2 + n^2 - 1 = 6n + 4n^2 - 1 = 4n(n + 3/2) - 1. \quad \square$$

Corollary 5.3.13. For any given number of qubits, there are only a finite number of different Clifford unitaries.

Exercise 5.13. Using Theorem 5.3.11, give an upper-bound on the number of different n-qubit Clifford unitaries for any n. *Hint: See Section 4.5.1 for how you can count this.*

We can also conclude another interesting fact from this extraction of a circuit from a GSLC normal form. The simplification procedure to get to this GSLC normal form works for *any* Clifford ZX-diagram, i.e. any ZX-diagram whose phases are multiples of $\pi/2$, not just those coming from a Clifford circuit. If such a diagram happens to be unitary, then we can use the procedure above to transform it into a Clifford circuit. Hence, the following proposition immediately follows.

Proposition 5.3.14. Any unitary Clifford ZX-diagram is equal to a Clifford circuit.

Exercise 5.14. Show that any Clifford ZX-diagram of an isometry V can be transformed into a Clifford circuit with some ancilla qubits in state $|0\rangle$. That is, there exists come Clifford circuit U such that $V = U(I \otimes |0...0\rangle)$, or graphically:
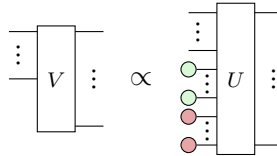
Hint: See Exercise 4.6.

Exercise 5.15. In Theorem 5.3.11 we had three layers of Hadamards. It turns out we only need two, because we can get rid of either the first or last one by doing some additional rewriting before extracting a circuit. To start, let's assume we have a unitary Clifford ZX-diagram in GSLC form. We are going to progressively remove Hadamards on its input wires.

a) Suppose we have an input spider with a Hadamard on it, and that the phase of the spider is $a\pi$. Show that we can remove this Hadamard by doing a regular, non-vertex-removing, pivot (5.8) between this input and

an output it is connected to. Why will it always be connected to an output? Argue that this does not introduce any Hadamards on other inputs.

b) Suppose we have an input spider with a Hadamard on it, and that the phase of the spider is $\pm\frac{\pi}{2}$. Show that you can remove the Hadamard by using a Euler decomposition and removing the resulting $X(\frac{\pi}{2})$ phase using a regular local complementation (5.4). Show that you can rewrite the diagram back into GSLC form, but now with one fewer Hadamard on an input wire.

c) Argue that if you do these steps for all the inputs with Hadamards on them, that you get a GSLC form diagram where extracting a circuit does not give any Hadamards in the first layer.

5.4 Classical simulation of Clifford circuits

In the previous sections we saw that we can use two simple rewrites, local complementation and pivots, to reduce Clifford states to graph states with local Cliffords, and Clifford circuits to a normal form consisting of just a few layers of gates. There is a third class of relevant Clifford diagrams that we can simplify using these rewrite rules: scalars. Recall that a scalar diagram is any ZX-diagram that has no inputs or outputs. The reason we care about scalar diagrams is because they can represent amplitudes of a quantum computation. If we start with the basis state $|0\cdots0\rangle$, then apply a unitary U, and finally wish to know the amplitude of the state $U|0\cdots0\rangle$ with respects to some computational basis effect $\langle x_1\cdots x_n|$ we get the scalar $\langle x_1\cdots x_n|U|0\cdots0\rangle$. In our case we are interested in U's that are Clifford unitaries, so that the resulting scalar ZX-diagram is also Clifford.

So what happens if we feed a scalar Clifford diagram to the simplification procedure described in Section 5.2? Remember that the procedure allowed us to remove any internal Clifford spider. However, in a scalar Clifford diagram, *all* spiders are internal and Clifford, so after we are done simplifying there will be no spiders left! The diagram will have been simplified away completely. What does this mean? The empty diagram evaluates to the scalar 1. However, remember that all our rewrites also introduce some scalar factors, which in the case of Clifford diagrams are always multiples of $1/\sqrt{2}$ and $e^{i\frac{\pi}{4}}$. So the end result is just a number, which is equal to the amplitude we were calculating.

Let's summarise all this in a proposition.

Proposition 5.4.1. Let U be a Clifford circuit, and let $\langle x_1\cdots x_n|$ for $\vec{x}\in\mathbb{F}_2^n$

be any computational basis effect. Then we can efficiently calculate the amplitude $\langle x_1 \cdots x_n | U | 0 \cdots 0 \rangle$.

This is really surprising! Clifford circuits contain essentially all the features we would expect from a quantum computation—entanglement, superpositions, and negative and complex amplitudes—and we can use Clifford circuits to realise many truly quantum protocols such as quantum teleportation or quantum key distribution, *and yet*, such circuits offer no computational benefit over classical computers. This result is known in the literature as the Gottesman-Knill theorem.

Gottesman-Knill theorem: A Clifford computation can be efficiently classically simulated.

Why is this the case? Well, on the surface we see it's because we can just rewrite the corresponding diagrams very well. But why is it that we can do that? A hint is given by the affine with phases normal form of Clifford states. Apparently, Clifford circuits can only produce quantum states that are uniform superpositions of computational basis states that are efficiently described by an affine subspace of bitstrings. This means there is a limit to how much we can actually use the complex amplitudes and entanglement present in Clifford states. There is for instance no way in which we can iteratively repeat a procedure to slowly add more and more amplitude to certain states like is done in Grover's algorithm: the amplitudes are always distributed equally.

5.4.1 Simulating Cliffords efficiently

In Proposition 5.4.1 we said we could simulate a Clifford amplitude efficiently, but how efficient are we talking?

The entire 'simulation' just consists of diagram simplification operations, so the complexity of the method, how expensive it is to actually calculate an amplitude, comes down to how hard it is to find the correct rewrite rule to apply, the number of rewrites we need to do, and how hard these rewrites are to perform.

The first two of these questions are easily answered. First, how hard is it to find the correct rewrite rule to apply? Well, local complementation applies to any spider with a $\pm\frac{\pi}{2}$ phase, so we simply have to loop over the spiders of the diagram until we encounter a spider with the correct phase. When we are done doing local complementations, any spider will have a 0 or π phase left, so all the spiders are suitable for a pivot, and we only need to look at any neighbour, so that finding a place where we can apply a rewrite rule

is quite trivial in this setting: we can do it anywhere. The second question, how many rewrite rules need to be applied, is also easily answered. Each of the rewrite rules, local complementation and pivoting, removes at least one spider, so that the number of rewrite rules applied is bounded by the number of spiders.

Finally, how hard is it actually to change the diagram based on the rewrite rules? We will measure this in terms of the number of **elementary graph operations** we need to perform: vertex and edge additions or removals. When we do a local complementation, we remove a single vertex, and we toggle the connectivity of all its neighbours. In the worst case, the spider we wish to remove is connected to pretty much all other spiders. In this case we end up changing the connectivity of the entire graph. So if there are N spiders, then this could cost N^2 edge additions and removals. A similar cost estimation holds for pivoting: we remove two spiders, and we toggle connectivity between three groups of spiders that could also pretty much encompass the entire graph, so that it also costs N^2 graph operations to implement a pivot.

So a single simplification costs about N^2 elementary graph operations in the worst case, and as we've seen, we will need about N rewrite rules to fully simplify the graph, which means we will need $N \cdot N^2 = N^3$ graph operations in the worst case. In comparison to this, the cost of finding the right rewrite rules to apply is negligible (adding at most an $O(N)$ cost to the application of every rewrite rule). We can now state a more detailed version of Proposition 5.4.1.

Proposition 5.4.2. Let D be a scalar Clifford diagram containing N spiders. Then we can calculate the value of D in time $O(N^3)$. In particular, we can calculate amplitudes of a Clifford circuit containing k gates in time $O(k^3)$.

Proof The first claim follows from the discussion on the complexity of rewriting above. For the second claim we simply note that each gate in a Clifford circuit can be translated into a fixed small number of spiders, so that the ZX-diagram corresponding to the amplitude to be calculated has $O(k)$ spiders. $\square$

Now, $O(N^3)$ is not too bad, but it does mean that once we start dealing with diagrams with, say, *millions*, of spiders, that we run into trouble. Can we do better?

As it turns out: yes! With a more clever simplification strategy, we can actually obtain a significantly better upper bound.

The reason we got this N^3 scaling, is because we weren't telling the

simplifier *which* spiders to target, so that we couldn't limit the number of wires that end up in the diagram. There is a better strategy that we can use to simplify the diagram, which works when we know that the diagram came from a circuit. The idea is that once we have a GSLC diagram that we can very efficiently 'absorb' a Clifford gate and rewrite the whole thing as another GSLC diagram.

So suppose we have a Clifford circuit consisting of CZ, Hadamard and *S* gates (if your circuit contains CNOTs, then these can be converted into CZ gates surrounded by Hadamards). Now when we write this circuit as a ZX-diagram, it will only contain Z-spiders, so that it already looks a bit like a graph-like diagram. *However*, we will not actually reduce it to graph-like form like we did with our previous simplification algorithm. Instead we will keep the circuit structure intact.

Now we will introduce some dummy spiders and Hadamards in order to insert a GSLC form diagram at the start of the circuit:

$$(5.17)$$
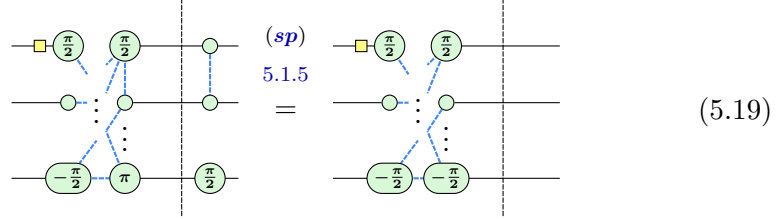
We will now consume gates one by one from the circuit and absorb them into the GSLC part of the diagram, transforming it into a different GSLC diagram, while not affecting the other parts of the diagram. Depending on the gate and on the specific configuration the GLSC part of the diagram is in, we will need to do different things.

The easiest gate to deal with is the Hadamard. This is simply absorbed as part of the GSLC if there is no Hadamard already on that qubit, or cancelled if there is one:

$$(5.18)$$

For *S* gates and CZ gates the situation is more complicated. If there are no Hadamards on the qubits they act on, then we can also absorb them as part

of the GSLC:

$$
\text{(5.19)}
$$

For the CZ gate, as always, if there was already a Hadamard edge present between the spiders, then we simply toggle the connectivity.

Now, if there *is* a Hadamard in the way then we can't just fuse the S or CZ gate into the GSLC. Instead, we will need to resort to our old friends, the local complementation and pivot. The reason we can use these, is because if there is a Hadamard in the way, with an S or CZ gate on the right-hand side, then the spider that is part of the GSLC is 'internal' in the sense that all its connections are to other spiders via a Hadamard edge. Now, if this spider has a $\pm\frac{\pi}{2}$ phase, this is straightforward enough: we just apply a local complementation on it to remove it. This connects the spider of the S or CZ gate to all the neighbours of this internal spider, so that the S or CZ spider takes its place in the GSLC diagram:

$$
\text{(5.20)}
$$

Now, if the spider we wish to remove has a 0 or π phase, then we need to apply a pivot to get rid of it. The spider must be connected to at least one input spider (since the diagram is unitary), so that we can apply the standard non-spider-removing pivot rule of (5.8). The end result is that the output Hadamard disappears, so that the spider of the gate can be fused and become part of the GSLC diagram.

Combining these different options we see that we can always absorb a gate into the GSLC portion of the diagram. We can hence simplify the entire circuit into a GSLC diagram.

Exercise 5.16. Reduce the following Clifford circuit to GSLC form using the algorithm described in this section.
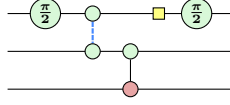
Crucially, the connectivity change resulting from these local complementations and pivots is now restricted to just the other spiders of the GSLC diagram and the spider corresponding to the gate to be absorbed, instead of potentially involving the entire diagram. Letting q denote the number of qubits, there are $2q$ qubits in the GSLC diagram, so that this requires toggling at most $(2q)^2$ edges. We need to do such a rewrite potentially for every gate we absorb, so that the entire simplification costs $O(Nq^2)$ elementary graph operations where N is the number of gates in the circuit. Compare this to our previous method that required $O(N^3)$ elementary graph operations. The number of gates is generally a lot more than the number of qubits, so this is a significant savings. Indeed, from Proposition 5.3.12 it follows that to represent an arbitrary Clifford unitary we need $O(q^2)$ gates, so taking $N = q^2$ we see that we have improved the complexity from $O(q^6)$ to $O(q^4)$.

Proposition 5.4.3. Let U be a q-qubit Clifford circuit with N CNOT, Hadamard and S gates. Then we can reduce it to GSLC form using $O(Nq^2)$ elementary graph operations. Furthermore, assuming that $N \geq n$, we can also write U in the layered normal form of Theorem 5.3.11 in $O(Nq^2)$ time.

Proof The reduction to GSLC form in $O(Nq^2)$ time follows from the algorithm described above. To rewrite a GSLC diagram into the layered Clifford normal form requires a Gaussian elimination of the central parity diagram (5.16). This takes $O(q^3)$ time for a total cost of $O(Nq^2 + q^3)$. As long as $N \geq q$, this reduces to $O(Nq^2)$. □

Proposition 5.4.4. Let U be a q-qubit Clifford circuit with $N \geq q$ CNOT, Hadamard and S gates. Then we can calculate any amplitude $\langle x_1 \cdots x_n | U | 0 \cdots 0 \rangle$ in $O(Nq^2)$ time. If U is given as a GSLC diagram then this reduces to $O(q^3)$.

Proof We first reduce U to GSLC form using the algorithm described above. This takes $O(Nq^2)$ time. Then we compose the diagram with the ZX-diagrams for $|0 \cdots 0\rangle$ and $\langle x_1 \cdots x_n|$. This adds $O(q)$ spiders, so that the total diagram has $O(q)$ spiders. Using the standard simplification algorithm on this diagram then requires $O(q^3)$ graph operations. Assuming $N \geq q$ the total cost is then $O(Nq^2)$ for fully reducing the diagram to a scalar. If U were already given as a GSLC diagram, then only the final set of simplifications is necessary, requiring just $O(q^3)$ graph operations. □

Exercise* 5.17. The $O(N^3)$ bound on calculating amplitudes in Proposi-

tion 5.4.2 is just an upper-bound. In practice it might turn out not to be so bad. Implement a benchmark of random Clifford circuits with an increasing number of gates and/or qubits and measure what the actual exponent is more like. Does the exponent stay the same for different number of qubits? Can you find a different strategy for targeting spiders to remove that leads to better scaling?

5.4.2 Weak vs strong simulation

In the previous sections we discussed how to calculate the amplitude corresponding to observing a particular effect on a Clifford state and we called this 'simulating' a Clifford computation. But this is not entirely correct. In order to truly say we are simulating a quantum computation we should be getting the same types of outcomes that we would get when we would actually run the quantum circuit. These outcomes come in the form of measurement samples. Namely, we would prepare a quantum state, execute quantum gates on it, and finally measure each qubit. The outcome is then a bitstring specifying which measurement outcome we got for each qubit. By running the experiment many times we will see a particular distribution of bitstrings as outcomes. To simulate a quantum computation is then to be able to generate a series of bitstrings that have a similar distribution of outcomes.

Definition 5.4.5. Let U be some unitary, and write

$$P(x_1, \ldots, x_n) = |\langle x_1 \cdots x_n | U | 0 \cdots 0 \rangle|^2$$

for the probabilities of observing the outcome $|x_1 \cdots x_n\rangle$ when applying U to the input state $|0 \cdots 0\rangle$. We then say a probabilistic algorithm **weakly simulates** U when it produces bitstrings $\vec{y} \in \mathbb{F}_2^n$ according to a distribution suitably close to P.

In this definition 'suitably close' can be made exact, but that won't be necessary for us for now. With what we have seen up to now, it is not obvious how we can actually efficiently weakly simulate Clifford unitaries. We are however quite close to being able to *strongly* simulate Clifford circuits.

Definition 5.4.6. Let U be some unitary, and let $P(x_1, \ldots, x_n)$ be its associated probability distribution as above. Then we say an algorithm **strongly simulates** U when it can calculate (or closely approximate) any marginal probability of P.

Let's explain this definition. Recall that a **marginal probability distribution** is one where we don't care about the outcome of one or more

of the variables of the distribution. For instance, if we have a distribution $P(x_1, x_2, x_3)$ of three variables, then we can **marginalise** x_3 to get the distribution $P(x_1, x_2) := \sum_{x_3} P(x_1, x_2, x_3)$. When $P(x_1, x_2, x_3)$ corresponds to the probabilities of observing particular measurement outcomes in a 3-qubit circuit, then this marginal $P(x_1, x_2)$ tells us the probability of observing x_1 on qubit 1 and x_2 on qubit 2, when we don't measure qubit 3 at all.

The reason we require the ability to determine any marginal probability in the definition of strong simulation, is that if we can only calculate the non-marginal probabilities, then it will generally take exponential resources to determine marginal probabilities. Consider for instance an n-variable distribution $P(x_1, \ldots, x_n)$ of Boolean variables. Then to calculate the marginal $P(x_1)$ we have to sum the values of $n-1$ variables $P(x_1) = \sum_{x_2,\ldots,x_n} P(x_1, x_2, \ldots, x_n)$, and this summation contains $O(2^n)$ terms.

Okay, so now that we understand what the definition says, it might be helpful to answer why we call this 'strong' simulation as opposed to 'weak' simulation. First of all, note that being able to weakly simulate doesn't seem to easily imply the ability to calculate the value of the probabilities: it might be that some probability $P(x_1, \ldots, x_n)$ is exponentially small (in n), and hence we would need to sample at least exponentially many bit strings using weak simulation to get an estimate of its value. But the converse is true: if we can *strongly* simulate a unitary, then we can also *weakly* simulate it. To see how to do this, we need to recall the concept of conditional probabilities. A **conditional probability distribution** tells us the probability of observing some outcome *conditional* on some other variables taking particular values. For instance $P(x_1, x_2 \mid x_3 = 1)$, the probability of observing the particular values x_1 and x_2 given that $x_3 = 1$, is equal to $P(x_1, x_2, 1)/P(x_3 = 1)$. Hence, since strong simulation allows us to calculate any marginal probability, we can then also calculate any conditional probability.

So how do we use marginal and conditional probabilities to generate a sample bitstring $\vec{y}$ with the correct distribution? We do this by determining the value of each bit of $\vec{y}$ in turn. First we ask for the probability $p_1 = P(x_1 = 0)$, the marginal probability that we observe qubit 1 to give a value of 0. We then decide with probability p_1 to set $y_1 = 0$ and otherwise we set $y_1 = 1$. We then calculate $p_2 = P(x_2 = 0 \mid x_1 = y_1)$ and set $y_2 = 0$ with probability p_2 and otherwise $y_2 = 1$. This means that we have chosen y_1 and y_2 with probability

$$P(y_2 \mid x_1 = y_1) \cdot P(y_1) = \frac{P(y_1, y_2)}{P(y_1)} \cdot P(y_1) = P(y_1, y_2) \qquad (5.21)$$

which is the correct distribution. We carry on and calculate $p_3 = P(x_3 = 0 \mid x_1 = y_1, x_2 = y_2)$ and repeat the procedure until we have determined the entire bit string $y_1 \cdots y_n$. Repeating the argument of Eq. (5.21) we see that we will have chosen this specific bit string with probability $P(x_1 = y_1, x_2 = y_2, \ldots, x_n = y_n)$ so that we are indeed sampling correctly from this distribution.

To summarise:

- Weak simulation is about sampling correctly from the probability distribution of measurement outcomes we would get if we were to actually run the quantum circuit.

- Strong simulation is about calculating the actual probabilities of observing certain measurement outcomes;

- If we can do strong simulation, then we can do weak simulation.

So how do we actually do strong simulation of Clifford circuits with the ZX-diagram simplification strategies we have discussed in this chapter? Well, we have seen how to calculate an amplitude $\langle x_1 \cdots x_n | U | 0 \cdots 0 \rangle$ which allows us to calculate a non-marginal probability $P(x_1, \ldots, x_n) = |\langle x_1 \cdots x_n | U | 0 \cdots 0 \rangle|^2$. But how do we calculate a marginal probability? To see this it first helps to expand $P(x_1, \ldots, x_n)$ in a way that allows us to more easily represent it as a diagram. Recall that for a complex number z we have $|z|^2 = zz^*$ where z^* is the conjugate. The conjugate of the inner product $\langle x_1 \cdots x_n | U | 0 \cdots 0 \rangle$ is $\langle 0 \cdots 0 | U^\dagger | x_1 \cdots x_n \rangle$. We can represent each of these by ZX-diagrams, and as they are scalar diagrams, their multiplication is implemented just by putting them next to each other:

$$P(x_1, \ldots, x_n) = \left(\tfrac{1}{\sqrt{2}} \right)^{4n} \quad \vcenter{\hbox{}}$$

$$\tag{5.22}$$

We get these scalar factors of $1/\sqrt{2}$ because the X-spiders only represent $|0\rangle$ and $|1\rangle$ up to a scalar. Note that these middle pairs of spiders correspond then (up to scalar) to operators $|x_i\rangle\langle x_i|$. Now if we want to calculate a marginal probability then this corresponds to a sum of such diagrams where we vary the values of some of the x_i. Remember that $\sum_x |x\rangle\langle x| = \text{id}$, hence on the diagram we can implement this summation by replacing the middle spiders

by an identity wire:

$$P(x_1, \cdots, x_k) = \left(\frac{1}{2}\right)^{n+k}$$

(5.23)

Using this diagrammatic representation of a marginal probability allows to calculate the marginals of a Clifford computation.

Proposition 5.4.7. Let U be a q-qubit Clifford circuit with $N \geq q$ gates. Then we can calculate any marginal probability of U using $O(Nq^2)$ graph operations. If U is given in GSLC form then it takes only $O(q^3)$ operations.

Proof First reduce U to GSLC form using $O(Nq^2)$ operations (see Proposition 5.4.3). Then construct the diagram as in (5.23) to represent the desired marginal probability. This diagram has $O(q)$ spiders, so fully simplifying it requires $O(q^3)$ graph operations. As $N \geq q$ we see that the total number of steps taken is $O(Nq^2)$. □

Since we can now calculate any marginal probability, we have also found a way to sample measurement outcomes from the Clifford circuit.

Proposition 5.4.8. Let U be a q-qubit Clifford circuit with N gates. Then we can sample k bit strings from its measurement distribution using $O(Nq^2 + kq^4)$ graph operations.

Proof We first reduce U to GSLC form, which requires $O(Nq^2)$ operations. Now in order to sample a bit string from its distribution (i.e. to perform weak simulation), we can use the method described above, which requires us to calculate q marginals, each of which takes $O(q^3)$ operations to calculate. So sampling a single bitstring requires $O(q^4)$ operations. We need to do this k times, so the total cost is $O(Nq^2 + kq^4)$. □

It is perhaps a bit unsatisfying that the weak simulation seems to be more expensive than the so-called strong simulation. However, it turns out that there is a smarter way to perform weak simulation of Clifford circuits. To do this, we also need to simplify our unitary U using ZX-calculus rewrites, but now instead of reducing to GSLC form, we want to reduce it to the affine with phases form (AP form) of Section 5.3.1. We can do this with the same order of graph operations as it takes to reduce to GSLC form. We can see this

in two ways. Either we first reduce to GSLC form using Proposition 5.4.3, and then we do some final rewrites to reduce it to AP form. Or, we modify the algorithm used in Proposition 5.4.3 so that instead of absorbing gates into a GSLC part of the diagram, we absorb gates into an AP form diagram. We hence have the following.

Proposition 5.4.9. Let U be a q-qubit Clifford unitary with N gates. Then we can write both U and $U|0\cdots0\rangle$ in AP form using $O(Nq^2)$ operations.

Recall from Eq. (5.14) that a state in AP form can naturally be written as

$$U|0\cdots0\rangle \;=\; \frac{1}{\sqrt{2}^N} \sum_{\substack{\vec{x}\in\mathbb{F}_2^q \\ A\vec{x}=\vec{b}}} i^{f(\vec{x})}|\vec{x}\rangle$$

for some phase function f, parity matrix A and bit string $\vec{b}$. This state is an equal superposition of those $|\vec{x}\rangle$ for which $A\vec{x} = \vec{b}$. Hence, if we measure all the qubits then the outcomes will correspond to one of those bit strings $\vec{x}$ for which this parity condition $A\vec{x} = \vec{b}$ is satisfied (the phase function is irrelevant for the outcomes). So we see that sampling from the distribution of U boils down to being able to uniformly randomly select vectors that satisfy such parity conditions.

Proposition 5.4.10. Let U be a q-qubit Clifford unitary with N gates. Then we can sample k bit strings from its measurement distribution using $O(Nq^2 + q^3 + kq^2)$ operations.

Proof First write $U|0\cdots0\rangle$ in AP form using Proposition 5.4.9, which requires $O(Nq^2)$ operations, and let A and $\vec{b}$ denote the parity matrix and bit string determining the AP form. Now we need to uniformly randomly return k bitstrings $\vec{x}$ that satisfy $A\vec{x} = \vec{b}$. First find a single $\vec{z}$ for which $A\vec{z} = \vec{b}$. This takes $O(q^3)$ operations as A is of size $O(q)$, and Gaussian elimination takes cubic time. Now do another Gaussian elimination of A to find a basis $\vec{x}^1, \ldots, \vec{x}^r$ for the kernel of A, i.e. of those $\vec{x}^i$ for which $A\vec{x}^i = \vec{0}$. This also takes $O(q^3)$ time. Now to determine a uniformly random bitstring $\vec{x}$ satisfying $A\vec{x} = \vec{b}$ we simply generate uniformly random bits $c_1, \ldots c_r$ and output $\vec{x} = \vec{z} + c_1\vec{x}^1 + \cdots + c_r\vec{x}^r$ (this is indeed uniformly random as each bitstring in the kernel of A can be written in a unique way as a combination of the basis vectors). Hence, each additional sample just requires generating some random bits and summing together $O(q)$ bitstrings of length $O(q)$ for a total cost of $O(q^2)$ per sample. □

Instead of casting this algorithm in terms of Gaussian elimination, we can

also view it as a diagram rewriting exercise. In Eq. (5.12) we saw that we can relate the affine part of an AP form diagram, a X-Z normal form, to a different Z-X form:

$$\vec{w}_1\;\cdots\;\vec{w}_k \quad \propto \quad \sum_{\vec{x}\in A}|\vec{x}\rangle \qquad \text{where} \quad A = \left\{ \vec{x} \in \mathbb{F}_2^n \;\middle|\; \begin{array}{c} \vec{w}_1^T\vec{x} = b_1 \\ \cdots \\ \vec{w}_k^T\vec{x} = b_k \end{array} \right\}$$

$$\vec{v}_1\;\cdots\;\vec{v}_k \quad \propto \quad \sum_{\vec{x}\in A}|\vec{x}\rangle \qquad \text{where} \quad A = \vec{w} + \text{span}\{\vec{v}_1,\ldots,\vec{v}_k\}$$

From this latter type of diagram we can easily read of which computational basis states $|\vec{x}\rangle$ are part of the superposition making it up: each Z-spider represents a basis vector $\vec{x^j}$ of the kernel of A, where x_i^j is 1 precisely when the Z-spider is connected to output spider i. While the spiders give the solution to $A\vec{x} = \vec{0}$, the π phases on the output transform this into solutions to $A\vec{x} = \vec{b}$.

5.5 Completeness of Clifford ZX-diagrams

We have seen in this chapter that we can prove *a lot* about Clifford circuits using the ZX-calculus, but can we actually prove *everything*? In the previous chapter we saw that the phase-free ZX-calculus is *complete*, meaning we can prove all equations between phase-free ZX-diagrams using the rules of the phase-free ZX-calculus. It turns out the same is true for Clifford diagrams, when we use the Clifford rewrite rules.

In this section we will adopt the scalar-accurate rewrite rules of Section 3.6.2. We will then show that these rewrite rules are complete for the Clifford fragment, meaning that if we can represent a linear map by a Clifford diagram, then any two ways to do so can be rewritten into one another. We show completeness in two steps. First we study scalar Clifford diagrams and show that these can be reduced to a form that *uniquely* encodes the number they are equal to. Hence, any two scalars representing the same number are reduced to the same diagram, which shows that we have completeness for these scalar diagrams. Then we show that we can refine the AP normal form so that every Clifford diagram can be reduced to a unique normal form. This means that if two diagrams represent the same linear map, that they will be reduced to the same normal form, and hence we have a path of rewriting from one to the other.

5.5.1 A normal form for scalars

First let's show that we can correctly deal with scalar Clifford diagrams, so that we are then again free to ignore them. For this we will use the results of Section 3.6.2, and hence we use the rules of Figure 3.2 there. We've seen in Proposition 3.6.5 that when the diagram contains a zero scalar $\overset{\pi}{\bigcirc}$, that the entire diagram can be reduced to a simple unique form. Hence, we may assume our scalar diagram is non-zero.

Our strategy will be to simplify an arbitrarily complicated scalar Clifford diagram to a diagram that consists of small disconnected bits and pieces that can each be dealt with individually. It is straightforward enough to check that with the rules of Figure 3.2 we can prove all the rules of Figure 3.1 up to some scalar diagrams that contain at most two spiders. Hence, everything we have proven so far using the rules of Figure 3.1 remains true using our scalar-accurate set of rules, except that we might acquire some additional small scalar diagrams.

This means that in particular our simplification strategy for Clifford diagrams still works, except for some small modifications. We can still use the local complementation simplification Lemma 5.2.9, but as noted in Remark 5.2.10, now it doesn't actually *remove* the spider we complement on. Instead, the spider remains as an unconnected scalar. Hence, if we were to apply this rewrite to an unconnected spider (so where $n = 0$ in that lemma), it would not remove the spider. In order for the simplification strategy to work we must then only apply this lemma to spiders with a $\pm\frac{\pi}{2}$ phase that are connected to at least one other spider. The same consideration holds for the pivot simplification Lemma 5.2.11, where we should only apply it if at least one of the spiders being pivoted on is connected to some other spider.

In Section 5.4 it was noted that if we apply the simplification strategy to a scalar Clifford diagram, that the entire diagram is simplified away. With our scalar-accurate calculus we see that this now becomes slightly different. We can still remove most connections between spiders, but there will generally be some small scalar subdiagrams left.

Proposition 5.5.1. Any scalar Clifford diagram can be rewritten to be a product of the following scalar subdiagrams:

$$\overset{\frac{\pi}{2}}{\bigcirc}, \overset{\pi}{\bigcirc}, \overset{-\frac{\pi}{2}}{\boxed{}}, \circ\!\!-\!\!\circ\, , \overset{\pi}{\bigcirc}\!\!-\!\!\overset{\pi}{\bigcirc}\, , \circ\!\!\Longleftrightarrow\!\!\circ$$
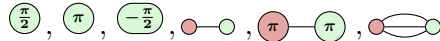
Proof First rewrite the ZX-diagram to graph-like form, except that we can't remove the triple set of wires in $\circ\!\!\Longleftrightarrow\!\!\circ$. We leave those subdiagrams as is. Then apply the scalar-accurate version of the local complementation Lemma 5.2.9 to any spider with a $\pm\frac{\pi}{2}$ phase that is connected to at least

one other spider. After doing this, any connected spider must have a phase of 0 or π. Apply the pivot Lemma 5.2.11 to any such pair where at least one of the spiders has connections to other spiders. Hence, after we are done there can only be connections between pairs of spiders that aren't connected to anything else. By simply enumerating the possibilities (and applying some colour-changes using (**cc**)) we see that the only possible connected subdiagrams are then the ones listed above, ●—(π) , and ○ . The first of these can be rewritten to ●—○ using Lemma 3.6.4, and ○ can be decomposed into a pair of ●—○ 's using Lemma 3.6.1. □

Now, if our scalar diagram contains a (π) then it can already be rewritten to a normal form, so let's assume it does not contain that subdiagram. Then our diagram is a composition of five different types of diagrams. By applying (**s**) we can ensure that the diagram does not contain both a ●—○ and a ○⊂⊃○ (as they are each others inverses). What other relations hold between these scalars? Using Lemma 3.6.7 we see that there can be at most one subdiagram of (π)—(π) and using Lemma 3.6.9 we can make it so that there is at most one spider with a $\pm\frac{\pi}{2}$ phase in our diagram.

Theorem 5.5.2. We can rewrite any non-zero scalar Clifford diagram D to a unique normal form $D_1^k \circ D_2 \circ D_3$ where $k \in \mathbb{N}$, $D_1 \in \{\,$●—○ $,$ ○⊂⊃○ $\}$, $D_2 \in \{\,$(π)—(π) $,$ ⌐⌐⌐ $\}$ and $D_3 \in \{\,$(π/2) $,$ (−π/2) $,$ (π)—(π/2) $,$ ⌐⌐⌐ $\}$.

Proof Apply Proposition 5.5.1 to rewrite the diagram to a product of pairs of spiders. Apply Lemma 3.6.9 so that the diagram contains at most one spider with a $\pm\frac{\pi}{2}$ phase. If this one diagram with a $\frac{\pi}{2}$ is (π)—(−π/2) then apply Lemma 3.6.6 in reverse with $\alpha = \pi$ and $\beta = \frac{\pi}{2}$, so that the $\frac{\pi}{2}$ diagram is then (π)—(π/2) .

Apply Lemma 3.6.7 so that there is at most one instance of (π)—(π) . Finally, cancel all pairs of ●—○ and ○⊂⊃○ using (**s**). It is then straightforward to verify that the diagram must be of the form specified.

To see this is a unique representation of the scalar let s denote the value of the scalar the diagram represents. The type of D_3 tells us whether $\mathrm{Re}(s) = \mathrm{Im}(s)$ ($D_3 = $ (π/2) $= 1 + i$), $\mathrm{Re}(s) = -\mathrm{Im}(s)$ ($D_3 = $ (−π/2) $= 1 - i$), $\mathrm{Re}(s) = 0$ ($D_3 = $ (π)—(π/2) $= \sqrt{2}i$) or $\mathrm{Im}(s) = 0$ ($D_3 = $ ⌐⌐⌐ $= 1$). As $s \neq 0$ by assumption D_3 must then be the same for any diagram representing s in this normal form. Similarly D_2 tells us whether the real part of s is positive (or the imaginary part in case $D_3 = $ (π)—(π/2)), since D_2 either represents $-\sqrt{2}$ or 1, so that this must also be the same diagram for every representation of s. Finally, D_1 being ●—○ $= \sqrt{2}$ or ○⊂⊃○ $= 1/\sqrt{2}$ together with the value

of k tells us the magnitude of $Re(s)$ (or $\text{Im}(s)$), and hence is also uniquely determined by s. □

  This theorem tells us that we can always rewrite any two scalar Clifford ZX-diagrams into one another if they represent the same scalar, since they can both be rewritten into this unique normal form.

5.5.2   A unique normal form for Clifford diagrams

The previous section settled the question of completeness for scalar Clifford diagrams. In this section we will do the same for non-scalar Clifford diagrams. Since the question of scalars is now settled, we will again revert back to not caring about these scalar subdiagrams. As before, we will write $\propto$ when we are denoting an equality up to a non-zero scalar factor. We will just work with states in this section, so diagrams that don't have any inputs. Once we have completeness for states, completeness for all diagrams follows easily just by bending some wires (like we did in Theorem 4.3.6).

  What we will show in this section is that we can refine the AP form, rewriting it into something more canonical. We will then show that this reduced normal form is unique. In particular, we will rewrite our diagrams to the following form.

Definition 5.5.3. A non-zero diagram in AP-form defined by the triple $A, \vec{b}, \phi$ is in **reduced AP-form** when:

  1. A is in reduced row echelon form (RREF) with no zero rows,
  2. ϕ only contains free variables from the system $A\vec{x} = \vec{b}$.
  3. all the coefficients of ϕ are in the interval $(-1, 1]$.

  Recall that the first non-zero element in a row of a matrix A in RREF is called a **pivot** (no relation to the pivot graph rewrite rule). The variable x_i is called a *free variable* if the ith column of A does *not* contain a pivot, otherwise it is called a *bound variable*. The utility of looking at the reduced AP-form is the following theorem.

Theorem 5.5.4. For any non-zero state $|\psi\rangle$, there is at most one triple $(A, \vec{b}, \phi)$ satisfying the conditions of Definition 5.5.3 such that:

$$|\psi\rangle \propto \sum_{\vec{x}, A\vec{x}=\vec{b}} e^{i\pi \cdot \phi}|\vec{x}\rangle$$

Proof   Since $|\psi\rangle \neq 0$, the set $\mathcal{A} = \{\vec{x} | A\vec{x} = \vec{b}\}$ is non-empty. Hence, there is a unique system of equations in RREF that define $\mathcal{A}$ (why?). From this

it follows that A and $\vec{b}$ are uniquely fixed. Now, for any assignment $\{x_{i_1} := c_1, \ldots, x_{i_k} := c_k\}$ of free variables, there exists $|\vec{x}\rangle \in \mathcal{A}$ such that $x_{i_\mu} = c_\mu$ (this is why we call these variables 'free', they can be chosen without restrictions while still satisfying the constraints given by A). Hence:

$$\langle \vec{x} | \psi \rangle = \lambda e^{i\pi\phi(c_1, \ldots, c_k)}$$

for some fixed constant $\lambda \neq 0$. From this it follows that, by inspection of $|\psi\rangle$, we can determine the value of ϕ at all inputs $(c_1, \ldots, c_k) \in \mathbb{F}_2^k$, modulo 2. This is enough to compute each of the coefficients of ϕ, modulo 2. (Q: How?) Since we require the coefficients of ϕ to be in the interval $(-1, 1]$, we can drop the "modulo 2" and conclude that ϕ is uniquely fixed by $|\psi\rangle$. $\quad\square$

Our goal then will be to show that we can rewrite any Clifford diagram into reduced AP-form. We already know that we can rewrite the diagram to AP form using Proposition 5.3.2. Such a diagram is then fully described (up to scalar) by a parity matrix A, a bitstring vector $\vec{b}$ and a phase function ϕ. We fully Gaussian eliminate the pair $(A, \vec{b})$ and do the corresponding transformation on the diagram as in Exercise 5.11, to bring it to RREF. If the Gaussian elimination shows that there is no solution to the affine system, then this corresponds to a scalar X-spider with phase π appearing, which is equal to the scalar 0. In this case, we can bring the diagram to the zero normal form of Proposition 3.6.5. So let's suppose that this is not the case, and the affine system is not inconsistent.

We remove all the zero rows of A (which correspond to scalar spiders). So now each row of A is a linear independent bit string. The affine part of the diagram is then unique. It then remains to show that we can rewrite the phase part so that the phase function ϕ only depends on free variables. To do this it will be useful to introduce some notation. Each of the internal spiders in the diagram defines a *parity set* P_j containing the spiders it is connected to. Since we have fully reduced the matrix A, each P_j contains a *pivot spider* $p_j \in P_j$ that does not appear in any other parity set. We then need to rewrite the diagram so that the pivot spiders p_j does not interact at all with ϕ. This means it must not have any phase, and not be involved with any CZs.

First, if a pivot spider has a phase of π, then this can be pushed to the other spiders of its parity set:
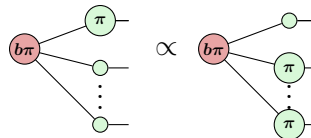
If instead its phase is $\pm\frac{\pi}{2}$ then we can remove it by applying a local complementation type rewrite rule. We start by applying the local complementation Lemma 5.2.9 in reverse:
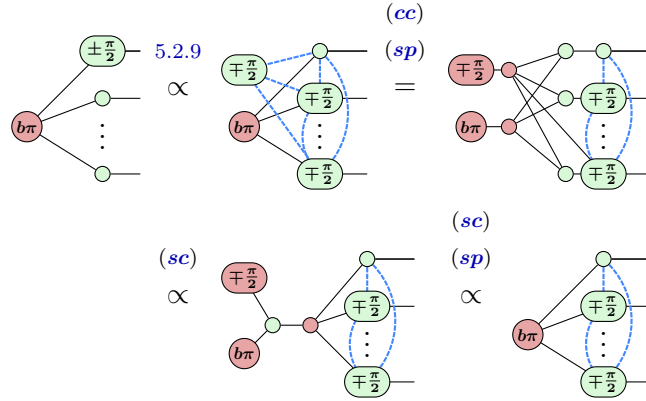
Here in the last step we also implicitly got rid of the scalar spiders that result after applying the copy rule (**sc**).

Now that all pivot spiders have no more phase, we will remove all the Hadamard edges that pivot spiders are involved in. First, we remove all the Hadamard edges from a pivot to a spider within its own parity set. We can do this by repeating the following rewrite:
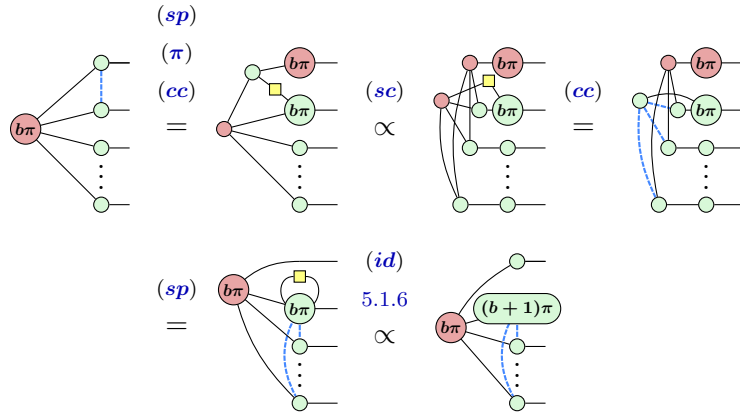
After every such rewrite the number of Hadamard edges from a pivot strictly decreases, so that the procedure terminates.

Finally, we can use a variation of this rewrite rule to remove Hadamard

edges from a pivot to a spider outside its parity set:
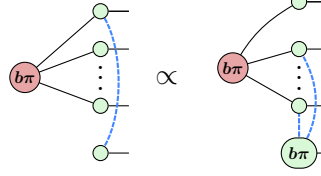
Note that this rewrite can introduce new Hadamard edges to the pivot of the other parity set, namely when we apply the rewrite rule to two pivots that are connected to each other. However, the rewrite rule either decreases the total number of edges that are connected to a pivot *or* it decreases the number of pivots that are connected to each other, so that we can always keep applying it and it will always reduce one of these metrics, so that the procedure does terminate. In the end when we can no longer apply the rewrite rule anywhere, it must mean that there are no Hadamard edges attached to a pivot in the diagram.

Remark 5.5.5. The previous phase-removal and edge-removal rewrites also apply when the pivot spider is the only spider in its parity set, but then these rewrites become quite trivial:

We conclude then that we get a diagram in reduced AP form.

Proposition 5.5.6. Any diagram in AP form can be transformed to reduced AP form.

Theorem 5.5.7. The ZX-calculus is complete for Clifford diagrams.

Proof Suppose we have two diagrams D_1 and D_2 that represent the same linear map. Without loss of generality we may assume that they are states by bending all the wires to the right. We can rewrite the diagrams to reduced AP forms D_1', respectively D_2'. The diagrams represent the same linear map and Theorem 5.5.4 shows that this linear map has a unique reduced AP form associated to it, so that D_1' and D_2' must be the same diagram. Furthermore, by reducing the scalars in the diagrams to a unique normal form (Theorem 5.5.2), we see that the scalar parts of the diagram are also equal. We then indeed have a set of rewrites from D_1 to D_2: first go from D_1 to D_1', which is equal to D_2', and then do rewrites in reverse to go from D_2' to D_2. □

Remark 5.5.8. The reader familiar with stabiliser theory might be wondering how it is that we get a *unique* normal form for Clifford states, as there is no really 'canonical' way to do this. The trick is that we fully Gaussian eliminate the parity matrix. This procedure makes a distinction between the qubits: the first qubit is much more 'likely' to be a pivot than the last qubit. A state that is symmetric in the qubits hence would not necessarily be reduced to a symmetric unique normal form (try to construct for instance the reduced AP form of the GHZ state).

You might wonder how these reduced AP forms relate to graph states. We have already seen that we can transform a state in AP form to one in GSLC form by doing some boundary pivots. If the diagram is in *reduced* AP form, the translation is even simpler: since for every internal spider we have a pivot spider that does not have a phase, it's Z-spider is actually an identity, and we can do some identity removal and colour change to turn it into a GSLC state:
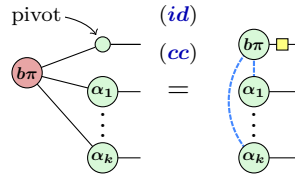
We see that that every internal spider in AP form translates into a Hadamard on the output of its corresponding pivot in the resulting graph state.

5.6   Summary: What to remember

1. Clifford *circuits* are circuits consisting of CNOT, Hadamard and S gates. Clifford *states* are the quantum states that can be produced by starting with $|0 \cdots 0\rangle$ and then applying a Clifford circuit to it.
2. Clifford circuits are represented by ZX-diagrams where all the phases are multiples of $\frac{\pi}{2}$. We call these diagrams *Clifford diagrams*.
3. Any Clifford diagram that is unitary can be written as a Clifford circuit.
4. We can efficiently simplify Clifford diagrams by the graph-theoretic operations of local complementation and pivoting.
5. This allows us to
   a) reduce each Clifford circuit to a normal-form consisting of just a few layers of Hadamard, S, CNOT and CZ gates,
   b) write each Clifford state in the *affine with phases* form,
   c) write each Clifford state as a *graph state with local Cliffords*,
   d) efficiently calculate amplitudes of Clifford states,

e) efficiently calculate marginal probabilities of Clifford states (*strong* simulation),

f) efficiently sample measurement outcomes from a Clifford state (*weak* simulation).

6. This scalar-accurate set of rewrites of the ZX-calculus is *complete* for Clifford ZX-diagrams, meaning that any two Clifford diagrams representing the same linear map can be rewritten into each other using these rewrite rules.

7. This follows from the existence of a *unique normal form* for Clifford states that we call the reduced AP form.

5.7 References and further reading

Classical simulation Clifford circuits were first shown to be efficiently classically simulable by Gottesman (1998), using what he called the *Heisenberg representation*, which is now more commonly called the stabiliser representation or stabiliser tableau. We will go into detail on this structure in Chapter 6. Notably, the classical simulation theorem itself appears in this single-author paper by Gottesman as "Knill's theorem", crediting Knill with the idea. These days it is called the **Gottesman-Knill theorem**.

Circuit normal forms The first Clifford circuit synthesis algorithm was given by Aaronson and Gottesman (2004), which produced a circuit with 11 layers of distinct gate types. This was subsequently improved to more compact forms Dehaene and De Moor (2003); Van Den Nest (2010); Maslov and Roetteler (2018). The shallowest decompositions, at the time of this writing, are due to Maslov and Zindorf (2022) have a 2-qubit gate depth of $2n + O(\log^2 n)$.

Graph states with local Cliffords The GSLC form for Clifford ZX diagrams is based on a result by Van den Nest et al. (2004b), who showed that any stabiliser state, i.e. any state prepared from $|0 \ldots 0\rangle$ using a Clifford circuit, can be expressed as a graph state with local Cliffords. The definition of GSLC form for ZX diagrams is due to Backens (2014a), who used it to prove completeness of the Clifford ZX calculus. The reduction of an arbitrary diagram to GSLC form is from Duncan et al. (2020).

Local complementation Local complementation, as a purely graph-theoretic concept, goes back to Kotzig (1968). It's relevance to quantum computing

originates with Van den Nest et al. (2004a), who showed that two graph states are equal up to local Cliffords if and only if one graph can be transformed into the other via local complementations. Elliott et al. (2008) showed that, furthermore, this sequence of local complementations can be found efficiently.

The local complementation rule was proved in the ZX-calculus by Duncan and Perdrix (2009). A more accessible proof is provided in Ref. (Coecke and Kissinger, 2017, Prop. 9.125). The pivoting rule was introduced for ZX diagrams by Duncan and Perdrix (2013), where it was used to show completeness of the "real stabiliser" fragment of the ZX-calculus, i.e. for ZX diagrams generated by phase-free spiders and the Hadamard gate.

The simplification-versions of local complementation and pivot rules (i.e. the ones that delete spiders) were introduced by Duncan et al. (2020) to simplify non-Clifford circuits, and with some additional variations on the pivoting rule, they were used in Backens et al. (2020) to simplify MBQC patterns (though note that the idea that 'a pivot deletes vertices' was also present in (Mhalla and Perdrix, 2013)).

AP normal form The AP form and reduced AP form were introduced in a preprint of this book in 2022 to give, among other things, a simplified proof of Clifford completeness. It first appears in published form in (Poór, 2022) for the case of odd-prime-dimensional qudits. A circuit decomposition for Clifford state preparations, which is closely related to the AP normal form, appears more than ten years earlier, in a classical simulation paper by Van Den Nest (2010).

6

Stabiliser theory

In Chapter 5, we already built up a good collection of practical tools for working efficiently with Clifford diagrams and circuits using rewriting. However, this is quite different from the toolkit one more typically encounters in the literature. An alternative way to work with Clifford maps is to focus not on the maps themselves, but rather families of Pauli operators that *stabilise* them. This approach is sometimes referred to as *stabiliser theory* or the *stabiliser formalism*. In this section, we will see that this formalism gives a fully equivalent way to represent and efficiently simulate Clifford maps.

In order to do this, we will develop a theory of *Pauli projections*, which are maps that project onto the $+1$ or -1 eigenspace of an n-qubit Pauli operator. Interestingly, these always chop a space precisely in half: the 2^n-dimensional space of n qubits can be regarded as the direct sum of two 2^{n-1}-dimensional subspaces in the range of a Pauli projection and its orthocomplement. As we'll see, subsequent commuting Pauli projections will further chop the space in half, until we get all the way down to a $2^0 = 1$ dimensional space.

This leads us to what we will call the **Fundamental Theorem of Stabiliser Theory** in Section 6.2.1, which states that k independent, commuting Pauli operators on n qubits uniquely fix a 2^{n-k} dimensional subspace of $(\mathbb{C}^2)^{\otimes n}$. In particular, when $n = k$, this gives us a 1D space, i.e. it uniquely fixes a state, up to a scalar factor. We will see that these states, often called **stabiliser states** in the literature, are precisely the Clifford states we saw in Chapter 5. We will also see that all of the interesting things we might want to compute about a stabiliser state, such as its evolution through a Clifford circuit or measurement probabilities, can be computed in terms of its associated Pauli operators.

This gives us a powerful second perspective on Clifford maps, which we will apply to quantum error correction and fault-tolerant quantum computation in Chapter 11. In Chapter 7, we will see that Pauli projections induce a

related type of map, called Pauli exponentials, which are closely related to phase gadgets, and give us a new way to speak about universal quantum computations and to synthesise circuits that simulate physical processes on a quantum computer.

6.1 Paulis and stabilisers
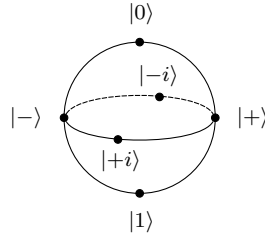
We met the **Pauli matrices** back in Chapter 2. Here they are again:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

These are self-adjoint matrices, so they each have a basis of eigenstates. These are respectively:

$$Z|0\rangle = |0\rangle \qquad X|+\rangle = |+\rangle \qquad Y|+i\rangle = |+i\rangle$$
$$Z|1\rangle = -|1\rangle \qquad X|-\rangle = -|-\rangle \qquad Y|-i\rangle = -|-i\rangle$$

When viewed on the Bloch sphere these six states lie in the corners corresponding to the three principal axes:

$$(6.1)$$

Each of these states is uniquely defined (up to global phase) as being the $+1$ eigenvector of one of the Pauli matrices. In particular, $|0\rangle, |+\rangle$ and $|+i\rangle$ are the $+1$ eigenvectors of Z, X, Y, respectively, while $|1\rangle, |-\rangle$ and $|-i\rangle$ are the $+1$ eigenvectors of $-Z, -X, -Y$. So while we don't care about the global phase of the vectors, for this definition of being the $+1$ eigenstate, we *do* care about the global phase of the Pauli operators, since the $+1$ eigenvector of Z is $|0\rangle$ while the $+1$ eigenvector of $-Z$ is $|1\rangle$.

Definition 6.1.1. We say a linear map A is a **stabiliser** of the state $|\psi\rangle$ when $A|\psi\rangle = |\psi\rangle$, i.e. when $|\psi\rangle$ is a $+1$ eigenvector of A.

An important fact about stabilisers is that the product of stabilisers is also a stabiliser. Suppose $A|\psi\rangle = |\psi\rangle$ and $B|\psi\rangle = |\psi\rangle$. Then, $AB|\psi\rangle = A|\psi\rangle = |\psi\rangle$. For that reason, we'll soon be interested not in just single operators stabilising a state, but whole groups of them.

Each of the Pauli matrices is its own inverse: $X^2 = Y^2 = Z^2 = I$, and if we multiply two different Pauli matrices, we get the third one, up to a factor of i:

$$XY = iZ \qquad ZX = iY \qquad YX = -iZ \qquad XZ = -iY \qquad (6.2)$$

One way to remember this is to think of the Pauli matrices arranged in a cycle:
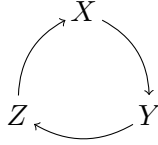
If we multiply two Paulis going forward in the cycle, we get the third one with a factor of $+i$, and if we multiply backwards, we get it with a factor of $-i$. Note that from the multiplications of (6.2) we also see that if we have two different Paulis P and Q that then $PQ = -QP$. We say then that these Paulis **anti-commute**.

Accounting for scalar multiples of i, we get a 16-element group, called the single-qubit **Pauli group**:

$$\mathcal{P}_1 \;=\; \{\, i^k P \mid k \in \{0, 1, 2, 3\}, P \in \{I, X, Y, Z\} \,\}$$

We can then ask which subgroups of this group stabilise some state $|\psi\rangle$. The trivial subgroup $\{I\} \subseteq \mathcal{P}_1$ stabilises everything, so that's not very interesting. Also, $-I|\psi\rangle = -|\psi\rangle$, so any group containing $-I$ doesn't stabilise anything except the zero state, which is also not very interesting. This rules out the whole group $\mathcal{P}_1 \subseteq \mathcal{P}_1$, and any group containing $\pm iP$, because $(\pm iP)^2 = (\pm i)^2 I = -I$.

That leaves just 6 subgroups, corresponding exactly to the 6 points on the Bloch sphere of (6.1):

$$\langle X \rangle, \langle Y \rangle, \langle Z \rangle, \langle -X \rangle, \langle -Y \rangle, \langle -Z \rangle \subseteq \mathcal{P}_1$$

Note we write $\langle \ldots \rangle$ to mean the subgroup generated by the given operators, so for example $\langle X \rangle = \{I, X\}$.

Exercise 6.1. Show that $\langle X, Y, Z \rangle = \mathcal{P}_1$, but $\langle X, Z \rangle$ generates a strict subset of $\mathcal{P}_1$.

In this section, we will generalise this idea of groups stabilising a state from one qubit to n qubits, by looking at subgroups of the following group.

Definition 6.1.2. The n-qubit **Pauli group** is defined as follows:

$$\mathcal{P}_n = \{\, i^k P_1 \otimes \ldots \otimes P_n \mid k \in \{0, 1, 2, 3\}, P_j \in \{I, X, Y, Z\} \,\}$$

We will refer to the elements of the n-qubit Pauli group as **Pauli strings**, and introduce special notation for them:

$$\vec{P} := i^k P_1 \otimes \ldots \otimes P_n$$

Concretely, fixing a Pauli string consists of choosing a global phase from the set $\{1, +i, -1, -i\}$ and for each P_j, a Pauli from the set $\{I, X, Y, Z\}$. Since each of these choices yields a distinct element of $\mathcal{P}_n$, it follows that $|\mathcal{P}_n| = 4^{n+1}$.

Exercise 6.2. Fix an arbitrary pair of Pauli strings $\vec{P}, \vec{Q} \in \mathcal{P}_n$, where:

$$\vec{P} := i^k P_1 \otimes \ldots \otimes P_n \qquad \vec{Q} := i^l Q_1 \otimes \ldots \otimes Q_n$$

Prove the following properties:

a) $\vec{P}$ is unitary.
b) $\vec{P}^\dagger = \vec{P}$ if k is even, otherwise $\vec{P}^\dagger = -\vec{P}$.
c) $\vec{P}^2 = I$ if k is even, otherwise $\vec{P}^2 = -I$.
d) $\vec{P}$ and $\vec{Q}$ either commute ($\vec{P}\vec{Q} = \vec{Q}\vec{P}$) or anti-commute ($\vec{P}\vec{Q} = -\vec{Q}\vec{P}$).

For the majority of this chapter and the next one, we will mostly be interested in **self-adjoint Pauli strings**, i.e. those whose global phase is ± 1. As the name suggests, this implies that $\vec{P}^\dagger = \vec{P}$. Since Pauli strings are unitaries, being self-adjoint implies that they are also self-inverse, i.e. $\vec{P}^2 = I$.

6.1.1 Clifford conjugation, a.k.a. pushin' Paulis

As we first pointed out way back in Example 3.2.2, whenever there are Z or X gates on the input of a Clifford circuit, we can always 'push them through' to the outputs using ZX rules. We now know enough about Clifford unitaries to prove this straightforwardly. We saw in Proposition 5.3.14 that any Clifford unitary U can be written as a circuit built out of CNOT, Hadamard, and S gates. Hence, we only need to check that we can push arbitrary single Pauli gates through each of these three. That is, for any single Z or X on any input, we can push it through to Z and X gates on the outputs.

For the Hadamard gate, Z pushes through and becomes X and X pushes through to become Z:

For the S gate, Z commutes straight through, whereas X turns into Y:

Note that we picked up the global phase of $e^{i\pi/2} = i$ when we used the π-commutation rule ($\boldsymbol{\pi}$). Finally, for the CNOT gate, there are 4 cases to consider, corresponding to a Z or X gate on either of the two inputs:
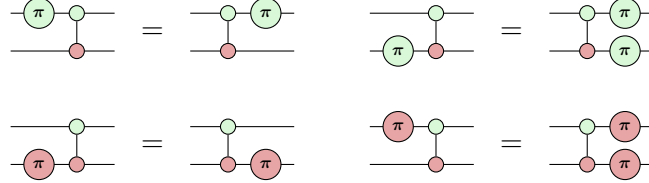
Since we can push any Z or X gate through any Clifford gate, we can push any $Y = iXZ$ gate through as well. Putting these rules together, we can see that, for any Pauli string $\vec{P}$ and Clifford unitary U, we can push $\vec{P}$ through U gate-by-gate, getting a (probably different) Pauli string $\vec{Q}$ on the outputs.

Proposition 6.1.3. For any Clifford unitary U and Pauli string $\vec{P}$, there exists a Pauli string $\vec{Q}$ such that:

or equivalently:

The second equation above, written symbolically as $\vec{Q} = U\vec{P}U^\dagger$, says that **conjugating** a Pauli string by a Clifford unitary yields another Pauli string. We will see in Section 6.4 that the converse of Proposition 6.1.3 also holds: any unitary that sends Pauli strings to Pauli strings under conjugation is Clifford.

A variation on this result, which will prove to be very useful in this chapter, is that we can use Clifford unitaries to simplify a multi-qubit Pauli string to one that is non-trivial on just one qubit (say Z_1).

Proposition 6.1.4. Let $\vec{P} = \pm P_1 \otimes \cdots \otimes P_n$ be a non-trivial self-inverse Pauli string. Then there exists a Clifford unitary U such that $U\vec{P}U^\dagger = Z_j$ for any choice of j.

Proof The first step will be to find a Clifford unitary that conjugates $\vec{P}$ to a Pauli string $\vec{P}'$ where $P_i' \in \{I, Z\}$ for all i. Set $V_i = I$ if $P_i = I$ or $P_i = Z$, set

$V_i = H$ if $P_i = X$, and set $V_i = X(-\frac{\pi}{2})$ if $P_i = Y$. Then $V := V_1 \otimes \cdots \otimes V_n$ gives $V\vec{P}V^\dagger = \vec{P}'$ with the desired property.

We will now find a CNOT circuit W such that $W\vec{P}'W^\dagger = \pm Z_j$. The desired unitary is then $U := W \circ V$, potentially composed with an additional X_j in order to get of the unwanted minus sign in $\pm Z_j$. We will build W step by step as follows.

By assumption $\vec{P}$ is non-trivial, and hence so is $\vec{P}'$, which means at least one P_k' is non-trivial. If $P_j' = I$, then we add a CNOT from j to k to W (meaning the control is on j and the target is on k). This ensures that there is a Z on qubit j. Now it remains to 'clean up' the Z's on the other qubits. To do this, for every P_k' where $k \neq j$ and $P_k' = Z$ we apply a CNOT from k to j. $\qquad\square$

Since any Clifford isometry can be written as a Clifford unitary with some ancilla qubits, the 'Pauli pushing' property holds for isometries as well.

Proposition 6.1.5. For any Clifford isometry $V : (\mathbb{C}^2)^{\otimes k} \to (\mathbb{C}^2)^{\otimes n}$ and any Pauli string $\vec{P}$, there exists a Pauli string $\vec{Q}$ such that:

Proof By Exercise 5.14, for any Clifford isometry V, there exists a Clifford unitary U such that:

Since $\vec{P} \otimes I \otimes \ldots \otimes I$ is a Pauli string on n qubits, we can use Proposition 6.1.3 to push it through U:

6.1.2 Stabiliser subspaces

We already noted that, if unitaries A and B both stabilise the same state $|\psi\rangle$, then their product AB also stabilises that state, as $AB|\psi\rangle = A|\psi\rangle = |\psi\rangle$. Hence, the set of stabilisers of a given state forms a group.

In the other direction, if two states $|\psi\rangle$ and $|\phi\rangle$ are stabilised by a map A, then:

$$A(\lambda|\psi\rangle + \mu|\phi\rangle) = \lambda A|\psi\rangle + \mu A|\phi\rangle = \lambda|\psi\rangle + \mu|\phi\rangle.$$

This means the set of states stabilised by a group of linear operators always forms a subspace of $(\mathbb{C}^2)^{\otimes n}$.

Definition 6.1.6. For a group $\mathcal{S} \subseteq \mathcal{P}_n$, we define the **stabiliser subspace** of $\mathcal{S}$ as:

$$\text{Stab}(\mathcal{S}) := \{ |\psi\rangle \mid \vec{P}|\psi\rangle = |\psi\rangle, \forall \vec{P} \in \mathcal{S}\}$$

If $\text{Stab}(\mathcal{S}) \neq \{0\}$, we call $\mathcal{S}$ a **stabiliser group**.

Note that in our definition of stabiliser group, we have explicitly ruled out those groups which only stabilise the zero state $|\psi\rangle = 0$. This is very convenient because it immediately puts several constraints on which subgroups form stabiliser groups. Most importantly, stabiliser groups are always commutative (a.k.a. Abelian), meaning all of their elements commute.

Proposition 6.1.7. If a subgroup $\mathcal{S} \subseteq \mathcal{P}_n$ is a stabiliser subgroup, then:

1. $-I \notin \mathcal{S}$,
2. $\vec{P}^2 = I$ for all $\vec{P} \in \mathcal{S}$, and
3. $\mathcal{S}$ is commutative.

Proof If $-I \in \mathcal{S}$, then all $|\psi\rangle \in \text{Stab}(\mathcal{S})$ satisfy $|\psi\rangle = -I|\psi\rangle = -|\psi\rangle$, so that $\text{Stab}(\mathcal{S}) = \{0\}$. If, for some $\vec{P}$, we have $\vec{P}^2 \neq I$, then by Exercise 6.2, $\vec{P}^2 = -I$. But then $-I \in \mathcal{S}$, so again $\text{Stab}(\mathcal{S}) = \{0\}$.

Finally, since every element in $\mathcal{S}$ is self-adjoint (or equivalently for unitaries, self-adjoint), $\mathcal{S}$ must be commutative: $\vec{P}\vec{Q} = \vec{P}^\dagger\vec{Q}^\dagger = (\vec{Q}\vec{P})^\dagger = \vec{Q}\vec{P}$. □

Remark 6.1.8. From this proof we in fact see that for *any* subgroup $\mathcal{S} \subseteq \mathcal{P}_n$, as long as $-I \notin \mathcal{S}$, that then $\mathcal{S}$ is commutative and it only contains self-adjoint Pauli strings.

Thus we have shown that, in order to stabilise a non-zero subspace, a subgroup of the Pauli group must satisfy several conditions. But are those conditions also *sufficient*? That is, does any subgroup of the Pauli group satisfying the conditions in Proposition 6.1.7, or by Remark 6.1.8 just the property that $-I \notin \mathcal{S}$, form a stabiliser group?

As it turns out, the answer is *yes*, and we can even figure out the dimension of the subspace stabilised by such a group. The key point to figuring this out has to do with number of independent generators of $\mathcal{S}$.

Definition 6.1.9. We say a collection of Paulis $\vec{P}_1, \ldots, \vec{P}_k \in \mathcal{P}_n$ is **independent** when none of the Paulis can be written as a product of some of the others (even up to global phase), or equivalently, when the Paulis can't be multiplied together to give $i^k I$ for any k, when we only allow each Pauli to appear once in the product.

Exercise 6.3. Let $\mathcal{S} \subseteq \mathcal{P}_n$ be a group satisfying the properties of Proposition 6.1.7.

a) Show that there exists a set of independent Paulis $\vec{P}_1, \ldots, \vec{P}_k$ that generates $\mathcal{S}$.
b) Show that any $\vec{Q} \in \mathcal{S}$ can be written uniquely as some product of these Paulis (taking the convention that the "empty product" is equal to $I \in \mathcal{S}$).
c) Show that such a product gives $\vec{Q}$ exactly, and not up to phase.

Hence, we can describe a stabiliser group by a set of independent Paulis that generate it. While n-qubit stabiliser groups may in general be very large, we will see that we can work with them efficiently by representing them as sets of generators.

6.2 Stabiliser measurements

In Section 2.2.4, we defined measurements as sets of projectors that sum up to the identity:

$$\mathcal{M} = \{M_1, \ldots, M_k\} \qquad \sum_i M_i = I$$

We also noted in that section that a self-adjoint operator O always defines such a set of projections, on to each of the subspaces associated with each distinct eigenvalue of O. Any self-adjoint Pauli string $\vec{P}$ is, in particular, a self-adjoint operator, so we can find such a set of projectors. In fact, these projectors always take a particularly simple form.

Definition 6.2.1. For a self-adjoint Pauli string, we define the associated **Pauli projectors** as follows:

$$\Pi_{\vec{P}}^{(0)} = \frac{1}{2}(I + \vec{P}) \qquad \Pi_{\vec{P}}^{(1)} = \frac{1}{2}(I - \vec{P})$$

Proposition 6.2.2. The maps $\Pi_{\vec{P}}^{(k)}$ for $k = 0, 1$ are projectors and furthermore:

$$\vec{P}|\psi\rangle = (-1)^k |\psi\rangle \iff \Pi_{\vec{P}}^{(k)}|\psi\rangle = |\psi\rangle \tag{6.3}$$

Proof We can show that $(\Pi_{\vec{P}}^{(k)})^\dagger = \Pi_{\vec{P}}^{(k)}$ and $\Pi_{\vec{P}}^{(k)}\Pi_{\vec{P}}^{(k)} = \Pi_{\vec{P}}^{(k)}$ by concrete calculation, using the fact that $\vec{P}^\dagger = \vec{P}$ and $\vec{P}^2 = I$. For (6.3), first assume $\vec{P}|\psi\rangle = (-1)^k|\psi\rangle$ for $k = 0, 1$. Then:

$$\Pi_{\vec{P}}^{(k)}|\psi\rangle = \tfrac{1}{2}(I + (-1)^k\vec{P})|\psi\rangle = \tfrac{1}{2}(|\psi\rangle + (-1)^{2k}|\psi\rangle) = |\psi\rangle$$

Conversely, if $\Pi_{\vec{P}}^{(k)}|\psi\rangle = |\psi\rangle$, then $\tfrac{1}{2}(I + (-1)^k\vec{P})|\psi\rangle = |\psi\rangle$. Multiplying both sides by 2 gives:

$$|\psi\rangle + (-1)^k\vec{P}|\psi\rangle = 2|\psi\rangle$$

Subtracting a $|\psi\rangle$ from both sides gives $(-1)^k\vec{P}|\psi\rangle = |\psi\rangle$. Multiplying both sides by $(-1)^k$ then gives $\vec{P}|\psi\rangle = (-1)^k|\psi\rangle$. $\square$

In the next exercise, you will prove some facts about Pauli projectors that will come in handy later.

Exercise 6.4. For $\vec{P}, \vec{Q}$ self-adjoint Pauli strings, show that Pauli projectors satisfy the following properties:

a) $\Pi_{\vec{P}}^{(0)} + \Pi_{\vec{P}}^{(1)} = I$ and $\Pi_{\vec{P}}^{(0)} - \Pi_{\vec{P}}^{(1)} = \vec{P}$.
b) $U\Pi_{\vec{P}}^{(k)}U^\dagger = \Pi_{U\vec{P}U^\dagger}^{(k)}$ for any unitary U.
c) $\vec{P}\vec{Q} = \vec{Q}\vec{P} \implies \Pi_{\vec{P}}^{(j)}\Pi_{\vec{Q}}^{(k)} = \Pi_{\vec{Q}}^{(k)}\Pi_{\vec{P}}^{(j)}$.
d) $\Pi_{\vec{P}}^{(0)}\Pi_{\vec{Q}}^{(0)} = \Pi_{\vec{P}}^{(0)}\Pi_{\vec{P}\vec{Q}}^{(0)}$.
e) $\Pi_I^{(0)} = I$ and $\Pi_I^{(1)} = 0$.

In this section, we will derive a useful graphical representation for the projectors $\Pi_{\vec{P}}^{(0)}$ and $\Pi_{\vec{P}}^{(1)}$. This representation will help us prove the Fundamental Theorem of Stabiliser Theory in the next section, as well as leading naturally to Pauli exponentials, which we'll introduce in Chapter 7.

Rather than directly building the Pauli projectors $\Pi_{\vec{P}}^{(k)}$, we will construct a "controlled" version of the projector, namely a ZX-diagram with the property that, when we plug in the computational basis effect $\langle k|$ to the first output, we'll be left with the projector $\Pi_{\vec{P}}^{(k)}$. This will give us a handy way to build up projectors for arbitrary Pauli strings from the basic Pauli projectors for X, Y, and Z.

We'll start with Z. The Pauli projectors are simply the projections onto the two Z eigenstates, i.e. the computational basis elements:

$$\Pi_Z^{(0)} = \tfrac{1}{2}(I + Z) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = |0\rangle\langle 0| \qquad \Pi_Z^{(1)} = \tfrac{1}{2}(I - Z) = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = |1\rangle\langle 1|$$

So, want to find a "box" that satisfies this property:

Thanks to the copy law, a single Z-spider will do the job:

More generally, we want to construct a new kind of generator, called a **Pauli box** that produces the projector $\Pi_{\vec{P}}^{(k)}$ when we plug in a computational basis effect $\langle k|$.

We can get the other two basic Pauli boxes, corresponding to projectors $\Pi_X^{(k)}$ and $\Pi_Y^{(k)}$, by applying unitaries to send the Z eigenstates to X and Y eigenstates, respectively. To do that, let's ZX-ify the Bloch sphere picture from (6.1).

$$\text{(6.4)}$$

Note that, while there is one obvious choice for the Z and X eigenstates, using spiders of the opposite colours, we actually have two choices for Y: either using Z phases or (negative) X phases. For our purposes, it will be more convenient to use the X phases, so:

$$|{+}i\rangle\langle{+}i| \;\propto\; \text{—}\tfrac{\pi}{2}\,\tfrac{\pi}{2}\text{—} \qquad\qquad |{-}i\rangle\langle{-}i| \;\propto\; \text{—}{-}\tfrac{\pi}{2}\,\tfrac{\pi}{2}\text{—}$$

Recall that because the $\langle{+}i|$ is defined to be the adjoint of $|{+}i\rangle$, we need to flip the phase to represent the $\langle{+}i|$. By conjugating the Z-spider with H or $X(-\tfrac{\pi}{2})$, we obtain Pauli boxes for X and Y. That is, for $P \in \{X, Y, Z\}$, we have:

    where $U \in \{\ \text{———}\ ,\ \text{—}\square\text{—}\ ,\ \text{—}{-}\tfrac{\pi}{2}\text{—}\ \}$ (6.5)

For building up generic Pauli strings $\vec{P}$, we should also have a trivial Pauli box corresponding to $\Pi_I^{(k)}$. We can obtain this simply as $|0\rangle \otimes I$. In summary, the four basic Pauli boxes are given as:

$$-\boxed{I}- \; := \; \tfrac{1}{\sqrt{2}} \; \multimap \qquad\qquad -\boxed{X}- \; := \; \multimap$$

$$-\boxed{Y}- \; := \; \multimap \qquad\qquad -\boxed{Z}- \; := \; \multimap$$

Exercise 6.5. Show that:

$$\tfrac{1}{\sqrt{2}} \cdot \overset{\overset{k\pi}{\bullet}}{-\boxed{P}-} \;\; = \;\; \Pi_P^{(k)}$$

for $P \in \{I, X, Y, Z\}$.

Curiously, we can also use Pauli boxes to 'switch on' the Pauli unitary by plugging in basis elements of the other colour.

Exercise 6.6. Show that:

$$\overset{\overset{k\pi}{\circ}}{-\boxed{P}-} \;\; = \;\; \begin{cases} I & \text{if } k = 0 \\ P & \text{otherwise} \end{cases}$$

for $P \in \{I, X, Y, Z\}$.

There is a good reason for this, which we'll discuss in more detail when we introduce Pauli exponentials in Chapter 7. For now, we'll see that this gives us everything we need to construct the projectors $\Pi_{\vec{P}}^{(k)}$ for longer Pauli strings.

Definition 6.2.3. For a self-inverse Pauli string $\vec{P} = P_1 \otimes \ldots \otimes P_n$ with $P_i \in \{I, X, Y, Z\}$, the associated **Pauli box** is defined as follows:

$$\vdots \; \boxed{\vec{P}} \; \vdots \;\; := \;\; \sqrt{2}^{\,n-1} \; \begin{array}{c} -\boxed{P_1}- \\ -\boxed{P_2}- \\ \vdots \\ -\boxed{P_n}- \end{array}$$

Using this definition, we can prove the generalisation of the equation in Exercise 6.5 to any self-adjoint Pauli string $\vec{P}$.

Proposition 6.2.4. Pauli projectors can be defined by plugging (normalised)

Z-basis effects into Pauli boxes as follows:

$$\Pi^{(k)}_{\vec{P}} \ :=\ \vdots \ \boxed{\vec{P}} \ \vdots$$

Proof This follows from decomposing the X spider as a number of Z spiders (i.e. X basis elements):

$$\left(k\pi\right) \ =\ \tfrac{1}{\sqrt{2}}\left(\ \bigcirc \ +(-1)^k \ \left(\pi\right)\ \right) \tag{6.6}$$

Applying (6.6), we have:

$$\tfrac{1}{\sqrt{2}}\ \vdots\ \boxed{\vec{P}}\ \vdots\ =\sqrt{2}^{\,n-2}\ \begin{matrix}P_1\\P_2\\\vdots\\P_n\end{matrix}\ =\ \sqrt{2}^{\,n-3}\left(\begin{matrix}P_1\\P_2\\\vdots\\P_n\end{matrix}\ +(-1)^k\begin{matrix}P_1\\P_2\\\vdots\\P_n\end{matrix}\right)$$

$$=\ \sqrt{2}^{\,-2}\left(\begin{matrix}P_1\\P_2\\\vdots\\P_n\end{matrix}\ +(-1)^k\begin{matrix}P_1\\P_2\\\vdots\\P_n\end{matrix}\right)$$

Then, by Exercise (6.6), the diagram above equals $\tfrac{1}{2}(I+(-1)^k\vec{P})=\Pi^{(k)}_{\vec{P}}$. $\square$

By unrolling the definition of the Pauli box, we can see that, for any Pauli string $\vec{P}=P_1\otimes\ldots\otimes P_n$ with $P_i\in\{X,Y,Z\}$, we have:

$$\vdots\ \boxed{\vec{P}}\ \vdots\ \propto\ \begin{matrix}\boxed{U_1^\dagger}&&\boxed{U_1}\\\boxed{U_2^\dagger}&&\boxed{U_2}\\\vdots&&\\\boxed{U_n^\dagger}&&\boxed{U_n}\end{matrix}\qquad\text{where}\quad -\boxed{U_i}-\ =\begin{cases}-\ \square\ -&\text{if }P_i=X\\-\left(\tfrac{\pi}{2}\right)-&\text{if }P_i=Y\\-\!-\!-&\text{if }P_i=Z\end{cases}$$

$$\tag{6.7}$$

The general case can be obtained by additionally representing any $P_i = I$ by simply not connecting to the i-th qubit, as we will see in the following example.

Example 6.2.5. The single-qubit and two-qubit Pauli projectors correspond to some simple ZX-diagrams. In particular, for a single qubit it always disconnects:

$$\Pi_Z^{(k)} = \quad \boxed{Z} \quad \propto \quad \overset{(sc)}{\longrightarrow} \quad \propto \quad -\!\!\underbrace{k\pi}\;\;\underbrace{k\pi}\!\!- \tag{6.8}$$

While $\Pi_{\vec{P}}^{(0)}$ for $\vec{P} = X \otimes X$ (abbreviated XX) becomes a 2-to-2 X spider:

$$\Pi_{XX}^{(0)} = \quad \boxed{X}\boxed{X} \quad = \quad \overset{(id)}{\underset{(sp)}{\quad}} \quad = \quad \overset{(cc)}{\bowtie} \quad = \quad \times \tag{6.9}$$

Note that for a Pauli string $\vec{P}$, if some $P_i = I$, then the projector does not interact with the qubit i:

$$\Pi_{IPI}^{(k)} \quad = \quad \boxed{I}\boxed{P}\boxed{I} \quad \propto \quad \boxed{P} \quad \overset{(sp)}{=} \quad \boxed{P} \tag{6.10}$$

Exercise 6.7. Show that the $Z \otimes \ldots \otimes Z$ and $X \otimes \ldots \otimes X$ measurements:

$$\mathcal{M}_{Z\ldots Z} := \{\Pi_{Z\ldots Z}^{(k)}\}_k \qquad \mathcal{M}_{X\ldots X} := \{\Pi_{X\ldots X}^{(k)}\}_k$$

are defined by the following Pauli projectors:

$$\Pi_{Z\ldots Z}^{(k)} \quad = \quad \vdots \qquad\qquad \Pi_{X\ldots X}^{(k)} \quad = \quad \vdots$$

We conclude this section by noting that Pauli boxes are an example of a more general object, which we call a *measure box*, that can be used to represent a generic 2-outcome measurement.

Definition 6.2.6. A map $M : \mathcal{H} \to \mathbb{C}^2 \otimes \mathcal{H}$ is called a **measure box** when it satisfies the following identities:

$$-\boxed{M}\!\!-\!\circ \; = \; -\!\!-\!\!- \qquad -\boxed{M}\boxed{M} \; = \; -\boxed{M}\!\!\prec \qquad \left(-\boxed{M}\right)^\dagger = \; \boxed{M} \tag{6.11}$$

Exercise 6.8. Let $M : \mathbb{C}^2 \otimes \mathcal{H} \to \mathcal{H}$ be a measure box.

a) Show that the following is a projector for $k \in \{0, 1\}$:

$$M_k := \tfrac{1}{\sqrt{2}} \cdot \; \overset{\boxed{k\pi}}{\boxed{M}}$$

b) Show that $M_0 + M_1 = I$.

c) Show that each of the 4 Pauli boxes:
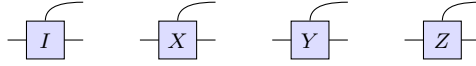
are measure boxes.

d) Let $N : \mathbb{C}^2 \otimes \mathcal{K} \to \mathcal{K}$ be a second measure box. Show that M and N can be combined as follows to form a new measure box satisfying (6.11):
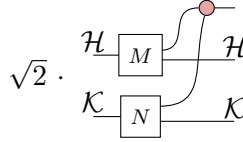
and hence that Pauli boxes for an arbitrary self-adjoint Pauli string $\vec{P}$ are measure boxes.

Since we'll use it later, we'll write down the conclusion of the above exercise explicitly in the following proposition.

Proposition 6.2.7. For any self-adjoint Pauli $\vec{P}$, the associated Pauli box satisfies the following equations:

$$\tag{6.12}$$

Exercise 6.9. Show diagrammatically that Proposition 6.2.7 implies that Pauli projectors on to different measurement outcomes are orthogonal:

6.2.1 The Fundamental Theorem of Stabiliser Theory

Any self-adjoint Pauli string (except the trivial one I) chops a space into two orthogonal parts: the $+1$ eigenspace, a.k.a. the stabiliser subspace, and the -1 eigenspace, a.k.a. the "anti-stabiliser" subspace. It turns out that these spaces have equal dimension, so we can see a single Pauli string $\vec{P}$ as chopping n-qubit space in half. Hence, the image of the projector $\Pi_{\vec{P}}^{(0)}$ (or $\Pi_{\vec{P}}^{(1)}$) is $2^n/2 = 2^{n-1}$ dimensional.

We can see this explicitly by **splitting** the projector $\Pi_{\vec{P}}^{(0)}$, i.e. finding an isometry $V : (\mathbb{C}^2)^{\otimes(n-1)} \to (\mathbb{C}^2)^{\otimes n}$ such that:

Proposition 6.2.8. Let $\vec{P}$ be a non-trivial self-inverse Pauli $\vec{P}$. Then there exists a Clifford unitary U such that $V = U(I \otimes \cdots \otimes I \otimes |0\rangle)$ splits $\Pi_{(-1)^k\vec{P}}^{(0)}$.

Proof By Proposition 6.1.4, there exists a Clifford unitary U such that $U^\dagger(-1)^k\vec{P}U = Z_n$. Then, by Exercise 6.4, we have $U^\dagger\Pi_{(-1)^k\vec{P}}^{(0)}U = \Pi_{U^\dagger(-1)^k\vec{P}U}^{(0)} = \Pi_{Z_n}^{(0)}$. As $V = U(I \otimes \cdots \otimes I \otimes |0\rangle)$ is then a normalised state followed by a unitary, it is an isometry. We can show directly that it splits $\Pi_{(-1)^k\vec{P}}^{(0)}$:

As this proposition shows that a potential minus sign on the Pauli can be incorporate into the splitting isometry, we will just ignore these minus signs in this section and without loss of generality assume that every Pauli string has a constant of $+1$.

If we split $\Pi_{\vec{P}}^{(0)}$ as $VV^\dagger$, then $\Pi_{\vec{P}}^{(0)}V = VV^\dagger V = V$. Hence, for any state $|\phi\rangle$ on $n-1$ qubits we have $\Pi_{\vec{P}}^{(0)}(V|\phi\rangle) = V|\phi\rangle$, so that $V|\phi\rangle$ is stabilised by $\vec{P}$ for any choice of $|\phi\rangle$. Conversely, when $\Pi_{\vec{P}}^{(0)}|\psi\rangle = |\psi\rangle$ then $|\psi\rangle = V(V^\dagger|\psi\rangle)$ so

that taking $|\phi\rangle = V^\dagger|\psi\rangle$ shows that any stabilised state is of this form. Then writing $\langle\vec{P}\rangle$ for the stabiliser group generated by $\vec{P}$ we have $\mathrm{Stab}(\langle\vec{P}\rangle) = \{V|\phi\rangle \,|\, |\psi\rangle \in \mathbb{C}^{2^{n-1}}\}$ so that we have shown the following.

Proposition 6.2.9. Let $\vec{P} \in \mathcal{P}_n$ be a non-trivial self-inverse Pauli. Then $\dim\mathrm{Stab}(\langle\vec{P}\rangle) = 2^{n-1}$.

Now we claim that this 'splitting' of the Hilbert space continues on if we have an additional independent commuting Pauli. In particular, if we have two independent commuting self-inverse Paulis $\vec{P}$ and $\vec{Q}$, then we claim that $\dim\mathrm{Stab}(\langle\vec{P},\vec{Q}\rangle\}) = 2^{n-2}$. This follows from the next lemma.

Lemma 6.2.10. Let $\vec{P},\vec{Q} \in \mathcal{P}_n$ be commuting self-inverse Pauli strings. Then there exists a Clifford unitary U and Pauli string $\vec{Q}' \in \mathcal{P}_{n-1}$ such that:

Proof The U from Proposition 6.2.8 is such that $\vec{P} = U^\dagger Z_n U$, and hence:

$$\Pi_{\vec{Q}}^{(0)}\Pi_{\vec{P}}^{(0)} \;=\; U^\dagger U\Pi_{\vec{Q}}^{(0)}U^\dagger U\Pi_{\vec{P}}^{(0)}U^\dagger U \;=\; U^\dagger\Pi_{\vec{Q}'}^{(0)}\Pi_{Z_n}^{(0)}U, \qquad (6.13)$$

where $Q' = U\vec{Q}U^\dagger$. Now, as $\vec{Q}$ and $\vec{P}$ commute, the Pauli strings that result from conjugating by the same unitary will also commute. Hence, $\vec{Q}'$ commutes with Z_n. This can only be the case when $Q'_n = I$ or $Q'_n = Z$. In both cases we then have:

$$(6.14)$$

Note that we still write $\vec{Q}'$ to denote $\vec{Q}'$ where the nth Pauli is removed to get a string of length $n-1$. We then calculate:

Note that in the lemma statement, we interchanged the role of U and $U^\dagger$ for convenience. $\qquad\square$

If $\vec{Q}$ is independent from $\vec{P}$, then the $\vec{Q}'$ we get here in this lemma is non-trivial, so that we can then also split $\Pi^{(0)}_{\vec{Q}'}$ using Proposition 6.2.8. We then end up with $n-2$ wires in the middle, showing that indeed $\dim \operatorname{Stab}(\langle \vec{P}, \vec{Q} \rangle) = 2^{n-2}$.

We can now iterate this procedure, so that if we have k independent commuting Pauli strings, that the states they mutually stabilise has dimension 2^{n-k}. In fact, this result is so crucial to this chapter, that we will give it a grand name: the **Fundamental Theorem of Stabiliser Theory**.

Theorem 6.2.11. Let $\vec{P}_1, \ldots, \vec{P}_m$ be independent commuting self-adjoint Pauli strings on n qubits. Then there exists a Clifford unitary U such that:

$$(6.15)$$

Consequently, letting $\mathcal{S}$ denote the group generated by the $\vec{P}_j$, we see that $\dim \operatorname{Stab}(\mathcal{S}) = 2^{n-m}$.

Proof We prove this by induction on m. The case of $m = 1$ is just Proposition 6.2.8. So suppose we know Eq. (6.15) holds when we have $m-1$ independent generators. Then let U_1 be a Clifford unitary such that $U_1 \vec{P}_1 U_1^\dagger = Z_n$. We can then use an argument similar to that of Lemma 6.2.10 to write:

Here $\vec{Q}_j = U_1 \vec{P}_j U_1^\dagger$. In order to use the induction hypothesis on $\vec{Q}_2, \ldots, \vec{Q}_m$ we need to show that they are still commuting and independent when we are ignoring the last qubit. First note that because we conjugated them all with the same unitary U_1 that $Z_n, \vec{Q}_2, \ldots, \vec{Q}_m$ are all still commuting and independent (not ignoring the n-th qubit). As a result each of the $\vec{Q}_j$ has either an I or Z in the nth position. This means that they all commute on the n-th position, and hence the $\vec{Q}_j$ must also be commuting when we ignore this last qubit. Additionally, no product of $\vec{Q}_2, \ldots, \vec{Q}_m$ can result in Z_n due to the them being independent of Z_n, so that no product of the $\vec{Q}_j$ can result in $\vec{I}$ when we ignore the last qubit. Hence, now ignoring the nth qubit,

the $\vec{Q}_j$ are independent and commuting, so that we can use the induction hypothesis. We then simply combine the resulting unitary U with U_1, and we are done. $\qquad\square$

There is a bunch of important results that follow from this theorem. First, it allows us to bound how many independent commuting Pauli strings there can be.

Corollary 6.2.12. If $\vec{P}_1, \ldots, \vec{P}_k$ is a set of commuting independent Pauli strings on n qubits, then $k \leq n$.

Proof Suppose k is at least $n+1$. Then applying the proof of Theorem 6.2.11 to the first $n + 1$ generators, we end up in the following situation once we have repeated the procedure $n - 1$ times:
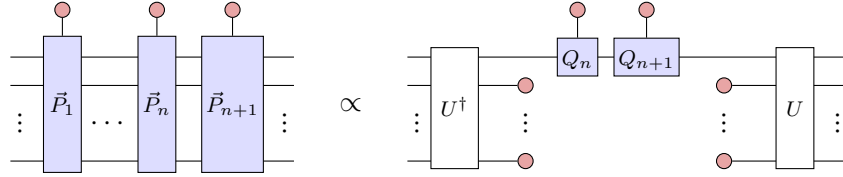
We see that we end up with single-qubit Pauli projectors $\Pi_{Q_n}^{(0)}$ and $\Pi_{Q_{n+1}}^{(0)}$ that must be commuting, but also independent. But such single-qubit Paulis don't exist, so we have reached a contradiction. $\qquad\square$

This corollary allows us to answer the question we raised before: any subgroup of the Paulis satisfying the properties of Proposition 6.1.7 does in fact stabilise at least some non-zero state.

Corollary 6.2.13. Let $\mathcal{S} \subseteq \mathcal{P}_n$ be a subgroup satisfying the properties of Proposition 6.1.7. That is, it is commutative, consists only of self-adjoint Pauli strings, and does not contain $-I$. Then $\mathcal{S}$ stabilises at least one non-zero state, and hence is a stabiliser group.

Proof By Exercise 6.3, $\mathcal{S}$ is generated by independent Paulis $\vec{P}_1, \ldots, \vec{P}_k$. Since $\mathcal{S}$ is commutative, all these $\vec{P}_j$ must also commute. Hence, the previous corollary applies and we must have $k \leq n$. Then Theorem 6.2.11 tells us that $\dim \mathrm{Stab}(\mathcal{S}) = 2^{n-k} \geq 2^0 = 1$, so that there is at least a 1-dimensional space of states that are stabilised by $\mathcal{S}$. $\qquad\square$

In fact, recall from Remark 6.1.8 that as long as $-I \notin \mathcal{S}$, the other properties of Proposition 6.1.7 are also satisfied, so any subgroup of $\mathcal{P}_n$ that does not contain $-I$ stabilises some non-zero state! We then also get the following corollary.

Corollary 6.2.14. Any set of commuting independent self-adjoint Pauli strings generates a stabiliser group.

Proof By definition of independence there is no non-trivial product of independent Pauli strings that produces $-I$, so that $-I$ is not in the group. Hence, the previous corollary applies so that we see that it is a stabiliser group. ☐

Some other major corollaries we will study in detail in the next section.

Exercise* 6.10. Generalise Equation (6.15) to account for other Pauli measurement outcomes. That is, show that:

for any $\vec{s} = (s_1, \ldots, s_m)$.

6.3 Stabiliser states and the Clifford group

In this section we will look at states that are uniquely determined by the Pauli strings that stabilise them. That such states exist follows from Theorem 6.2.11, as we will show next. We will then look at the unitaries that map Pauli strings into Pauli strings. These unitaries form the *Clifford group*, and as the name suggests these are related to the Clifford unitaries we studied in Chapter 5 (in fact, they are the same thing).

6.3.1 Maximal stabiliser groups

Theorem 6.2.11 shows that if the number of generators of $\mathcal{S} \subseteq \mathcal{P}_n$ is k, that its stabiliser subspace has dimension 2^{n-k}. This means that if the number of generators is exactly n, that its isometry V goes from 0 wires to n wires. But then V is just an n-qubit state! So let's write $V = |\psi\rangle$. Then the projector onto the stabiliser subspace of $\mathcal{S}$ is $|\psi\rangle\langle\psi|$. In other words: the only normalised state (up to global phase) that is stabilised by $\mathcal{S}$ is $|\psi\rangle$.

Corollary 6.3.1. Let $\mathcal{S} \subseteq \mathcal{P}_n$ be a stabiliser group with n generators. Then it stabilises a unique non-zero state (up to scalar).

From this theorem it also turns out that no stabiliser group can have more than n generators. Before we prove this it will be helpful to define a new class of stabiliser group.

Definition 6.3.2. We say a stabiliser group $\mathcal{S}$ is **maximal** when it is not strictly included in another stabiliser group. Equivalently, for any self-inverse $\vec{Q} \notin \mathcal{S}$ that commutes with all of $\mathcal{S}$ there must be a $\vec{P} \in \mathcal{S}$ such that $\vec{Q}\vec{P} = -I$.

Proposition 6.3.3. Any maximal stabiliser group has exactly n generators.

Proof Suppose $\mathcal{S}$ is a stabiliser group with $k < n$ generators. We will show that it is not maximal, which will prove our claim. Note that the right-hand side of Eq. (6.15) says that we can write the projection to $\mathrm{Stab}(\mathcal{S})$ as $U\Pi_{Z_{n-k+1}}^{(a_{n-k+1})} \cdots \Pi_{Z_n}^{(a_n)} U^\dagger$, and hence that we can take $\vec{P}_j := (-1)^{a_j} U Z_{n-j} U^\dagger$ for $j = 0, \ldots, k-1$ to be a set of generators for $\mathcal{S}$. Now, let $\vec{Q}$ be any non-trivial Pauli string on $n - k$ qubits. Then $\vec{Q}' = \vec{Q} \otimes I_{2^k}$ commutes with all the $Z_{n-k+1}, \ldots, Z_n$, and is also independent of this set. Hence $U\vec{Q}'U^\dagger$ is also independent from all the $UZ_{n-j}U^\dagger = (-1)^{a_j}\vec{P}^j$, and also still commutes with them. Hence, the stabiliser group generated by $\vec{Q}', \vec{P}^1, \ldots, \vec{P}^k$ strictly includes $\mathcal{S}$, so that $\mathcal{S}$ is not maximal. $\square$

Corollary 6.3.4. The following are equivalent for a stabiliser group $\mathcal{S}$ on n qubits.

- It is maximal.
- It has n generators.
- It stabilises a unique non-zero state (up to scalar).

6.3.2   Stabiliser states

A maximal stabiliser group stabilises a unique state. Such states turn out to be very special, so we will give them a special name.

Definition 6.3.5. We say a non-zero state is a **stabiliser state** if there is a maximal stabiliser group that stabilises it.

It might seem hard to determine when a state is stabilised by a maximal stabiliser group. Luckily, Corollary 6.3.4 offers an easier way to figure out when a state is stabiliser.

Proposition 6.3.6. Let $|\psi\rangle$ be a non-zero n-qubit state. Then $|\psi\rangle$ is a stabiliser state if and only if it is stabilised by n independent Pauli strings.

Proof   Suppose $|\psi\rangle$ is a stabiliser state, so that it is stabilised by a maximal stabiliser group $\mathcal{S}$. Then Corollary 6.3.4 shows that $\mathcal{S}$ has n generators. These generators are independent Pauli strings that stabilise $|\psi\rangle$. Conversely, suppose $|\psi\rangle$ is stabilised by n independent Pauli strings. Then these Pauli strings generate a stabiliser group, that again by Corollary 6.3.4 must be a maximal group, so that $|\psi\rangle$ is indeed a stabiliser state. $\square$

Example 6.3.7. Let $|\Psi\rangle := \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ be the maximally entangled Bell state. This state is stabilised by ZZ and XX. Indeed

$$
\begin{aligned}
(Z \otimes Z)(|00\rangle + |11\rangle) &= (Z|0\rangle) \otimes (Z|0\rangle) + (Z|1\rangle) \otimes (Z|1\rangle) \\
&= |0\rangle \otimes |0\rangle + (-|1\rangle) \otimes (-|1\rangle) \\
&= |00\rangle + |11\rangle,
\end{aligned}
$$

and $(X \otimes X)(|00\rangle + |11\rangle) = |11\rangle + |00\rangle = |00\rangle + |11\rangle$. As $|\Psi\rangle$ is a two-qubit state that is stabilised by two independent Pauli strings, it is hence a stabiliser state.

Example 6.3.8. Any graph state is a stabiliser state: recall that in a graph state $|G\rangle$ we have a one-to-one correspondence between qubits and the vertices of G. We claim that the Pauli string $\vec{P}^v := X_v \prod_{w \in N(v)} Z_w$ is a stabiliser of $|G\rangle$, where $N(v)$ denotes the **neighbourhood** of the vertex v, which is the set of vertices of G that are connected to v. As a Pauli string this stabiliser has an X on the qubit corresponding to v and a Z on every qubit that is connected to X. We can easily check this is a stabiliser of $|G\rangle$ by writing the graph state as a ZX-diagram, and pushing the X into the graph state using (π) and (cc):
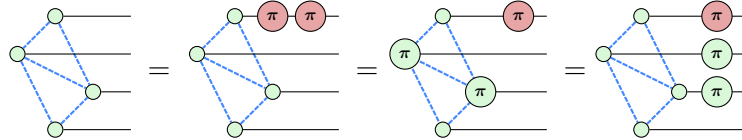
Exercise 6.11. Suppose $|\psi\rangle$ is a stabiliser state, which is stabilised by maximal stabiliser groups $\mathcal{S}$ and $\mathcal{S}'$. Prove that $\mathcal{S} = \mathcal{S}'$.

Recall from Chapter 5 that we defined a Clifford state to be any state of the form $U|0\cdots0\rangle$ for some Clifford unitary U, and that these correspond to Clifford ZX-diagrams with no inputs. Observe that the isometries of Theorem 6.2.11 that split the projector onto a stabiliser subspace are Clifford diagrams. In particular, for a maximal stabiliser group, the isometry is a state, and hence is a Clifford state. We then have the following.

Proposition 6.3.9. Any stabiliser state is a Clifford state (up to potentially

some non-zero scalar). That is, up to a scalar, we can represent any stabiliser state by a Clifford ZX-diagram.

The converse is also true: any Clifford state is a stabiliser state. Recall from Theorem 5.3.8 that we can rewrite any Clifford state to a graph state with local Cliffords. Example 6.3.8 shows that all graph states are stabiliser states, so to prove that any Clifford state is a stabiliser state it remains to show that (local) Cliffords send stabiliser states to stabiliser states. This turns out to follow from the property of Clifford unitaries we proved in Section 6.1.1, namely that conjugating a Pauli string by a Clifford results in another Pauli string.

Proposition 6.3.10. Let $|\psi\rangle$ be a stabiliser state and let U be a Clifford unitary. Then $U|\psi\rangle$ is also a stabiliser state.

Proof   Let $|\psi\rangle$ be an n-qubit stabiliser state and let $\vec{P}_1, \ldots \vec{P}_n$ be a maximal set of independent stabilisers of $|\psi\rangle$. Then we claim that the $\vec{Q}_k := U\vec{P}_k U^\dagger$ are independent stabilisers of $U|\psi\rangle$ so that $U|\psi\rangle$ is indeed a stabiliser state. First note that the $\vec{Q}_k$ are all Pauli strings because of Proposition 6.1.3. Second, they are indeed stabilisers of $U|\psi\rangle$ as $\vec{Q}_k U|\psi\rangle = U\vec{P}_k U^\dagger U|\psi\rangle = U\vec{P}_k|\psi\rangle = U|\psi\rangle$. Finally, note that they are independent since if we had $I = \prod_{k_j} \vec{Q}_{k_j}$ then $I = U^\dagger U = \prod_{k_j} U^\dagger \vec{Q}_{k_j} U = \prod_{k_j} \vec{P}_{k_j}$, which contradicts the independence of the P_k. $\qquad\square$

So stabiliser states and Clifford states are exactly the same thing!

Proposition 6.3.11. Any Clifford state is a stabiliser state, and vice versa any stabiliser state is proportional to a Clifford state.

6.3.3   The Clifford group

The property we used to prove Proposition 6.3.10 above was that conjugations of Pauli strings by Cliffords produces Pauli strings. This is in fact quite a useful property that we have used many times throughout this chapter and the previous, so it is worthwhile to try and understand this a bit better. In particular, can we find *more* unitaries than just the Cliffords that have this property?

For our next definition we need to know a little more group theory. Consider a group G with a subgroup H. An element $g \in G$ **normalises** H when $g^{-1}hg \in H$ for every H. For instance, every $h' \in H$ normalises H, and if G is abelian, then every element of G normalises H. We write $N_G(H)$ for the set of elements of G that normalises H.

Exercise 6.12. Show that $N_G(H)$ is a subgroup of G and that $H \subseteq N_G(H)$.

We can now define the Clifford group.

Definition 6.3.12. We say an n-qubit unitary U is **Pauli normalising** when it normalises $\mathcal{P}_n$. That is, when for every $\vec{P} \in \mathcal{P}_n$ we have $U\vec{P}U^\dagger \in \mathcal{P}_n$. We write $N(\mathcal{P}_n)$ for the group of Pauli normalising unitaries.

The group of Pauli-normalising unitaries is usually just called the **Clifford group**. However, as we already defined the term 'Clifford unitaries', we will not use the term Clifford group for now. Rest assured though that there is in fact a reason for the conflict in the naming scheme, as it turns out that Clifford unitaries and Pauli-normalising unitaries are the exact same thing. One direction of this equivalence we already have:

Example 6.3.13. Proposition 6.1.3 shows that every Clifford unitary is Pauli normalising.

The remainder of this section will work towards proving the converse.

Exercise 6.13. Show that the conjugation of a self-adjoint Pauli string by a Pauli-normalising unitary is also self-adjoint.

The $Z(\frac{\pi}{2})$ phase gate is Clifford and hence Pauli normalising. What other Z phase gates are Pauli normalising? Consider the phase gate $Z(\alpha) = \text{diag}(1, e^{i\alpha})$. Conjugating Z by this gate of course preserves Z as they commute, so it remains to check the case for X:

The only way for this to be equal to a Pauli is for the phase 2α to be equal to 0 or π. Hence, we get $\alpha = 0$ (the identity), $\alpha = \pi$ (the Z Pauli), $\alpha = \frac{\pi}{2}$ (the S gate), or $\alpha = -\frac{\pi}{2}$ (the adjoint of S). There are no other Z phase gates that are Clifford! By symmetry (i.e. colour-changing the spiders) the same situation of course also holds for X phase gates: only $X(\alpha)$ gates with α a multiple of $\frac{\pi}{2}$ are Pauli normalising. This seems to point towards ZX-diagrams only being Pauli normalising if all the phases involved are multiples of $\frac{\pi}{2}$, and this is in fact exactly right.

Definition 6.3.14. Let U be an n-qubit unitary. We denote by $|U\rangle$ the $2n$-qubit **Choi state** that we get by applying U to one side of n nested cups (i.e. Bell states):
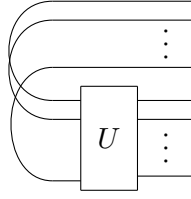
Lemma 6.3.15. Let U be a Pauli-normalising unitary. Then $|U\rangle$ is a stabiliser state.

Proof Let $\vec{P}_k$ be the Pauli string such that $Z_k U = U\vec{P}_k$ and similarly let $\vec{Q}_k$ be such that $X_k U = U\vec{Q}_k$. Then we claim that $\vec{P}_k^T \otimes Z_k$ and $\vec{Q}_k^T \otimes X_k$ are stabilisers for $|U\rangle$. Here $\vec{P}_k^T$ denotes the componentwise transpose of the Paulis in $\vec{P}_k$, such that $Z^T = Z$, $X^T = X$, and $Y^T = -Y$. That these are stabilisers is best demonstrated diagrammatically:
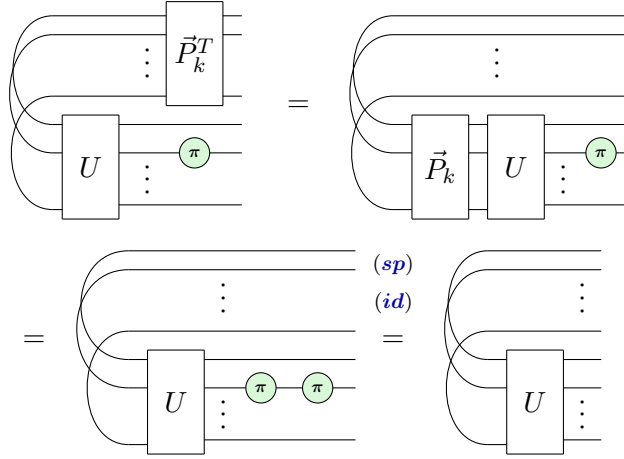
Here, the Z is taken to appear on the $(n+k)$th qubit. The same computation works for $\vec{Q}_k^T \otimes X_k$. We then have $2n$ stabilisers for a $2n$-qubit state. It is easily seen that these are all independent as they all have a unique non-I component on the bottom n qubits. Hence, by Proposition 6.3.6, $|U\rangle$ is a stabiliser state. $\qquad\square$

Proposition 6.3.16. A Pauli-normalising unitary is Clifford.

Proof Let U be Pauli normalising. Then $|U\rangle$ is a stabiliser state, so that by Proposition 6.3.9 we can represent $|U\rangle$ by a Clifford ZX-diagram. Then by bending the top wires in $|U\rangle$ back to be inputs, we can also represent U as a Clifford ZX-diagram. Proposition 5.3.14 then shows that U must be equivalent to a circuit of CNOT, Hadamard and S gates. $\qquad\square$

6.3.4 Putting it all together

We have now seen a lot of different perspectives, definitions and equivalences between them, so let's summarise what we have learned.

Theorem 6.3.17. Let U be a unitary. The following are equivalent:

a) U is Pauli normalising.
b) The Choi state $|U\rangle$ of U is a stabiliser state.
c) U can be presented as a Clifford diagram.
d) U is equivalent to a circuit consisting of Hadamard, S and CNOT gates.

Proof a) to b) is Lemma 6.3.15, b) to c) follows from Proposition 6.3.9, c) to d) is Proposition 5.3.14 and d) to a) is Proposition 6.1.3. $\square$

Because of these equivalences we don't have to make a distinction between Pauli-normalising unitaries, Clifford unitaries (those built from CNOT, Hadamard and S), and Clifford unitary diagrams (those whose diagram only contains $\frac{\pi}{2}$ phases), and just refer to any of these as a Clifford unitary.

We can write down a similar set of equivalences for stabiliser states.

Theorem 6.3.18. Let $|\psi\rangle$ be a state. The following are equivalent:

a) $|\psi\rangle$ is a stabiliser state.
b) $|\psi\rangle$ can be presented (up to non-zero scalar) as a Clifford diagram (that has no inputs).
c) $|\psi\rangle$ is equal (up to non-zero scalar) to a circuit of Hadamard, S and CNOT gates applied to the $|0\cdots0\rangle$ state.
d) $|\psi\rangle$ can be written as a graph state with local Cliffords.

Proof a) to b) is Proposition 6.3.9, b) to c) is Proposition 5.3.10, c) to d) is Theorem 5.3.8 and d) to a) follows from Example 6.3.8 and Proposition 6.3.10. $\square$

6.4 Stabiliser tableaux

We've seen that an n-qubit unitary is Clifford if and only if it maps every n-qubit Pauli string to a Pauli string under conjugation. There are exponentially many Pauli strings as n increases, but luckily we only have to check a few of them to verify the unitary is Clifford. In particular, it suffices to check that the Z_i and X_i are mapped to Pauli strings under conjugation. This gives us $2n$ conditions to check for an n-qubit unitary. We can present the resulting information in the form of a table that we call a *stabiliser tableau*. This is what we will look at in Section 6.4.2. Another way to present this

information is as a special type of matrix that is called a *symplectic* matrix. This is what we will look at in Section 6.4.4. Before we do so however, let's look a bit closer at the information encoded by knowing the result of these Pauli conjugations.

6.4.1 Cliffords are determined by Pauli conjugations

We will show that a Clifford unitary is completely determined by its action on Pauli strings.

Proposition 6.4.1. Let U_1 and U_2 be Cliffords such that $U_1 \vec{P} U_1^\dagger = U_2 \vec{P} U_2^\dagger$ for all Pauli strings $\vec{P} \in \mathcal{P}_n$. Then $U_1 = e^{i\alpha} U_2$ for some global phase α.

Proof By Lemma 6.3.15, $|U_1\rangle$ and $|U_2\rangle$ are stabiliser states. Fix $\vec{P}$ and let $\vec{Q}$ be such that $U_1 \vec{P} = \vec{Q} U_1$. Then we can check that $\vec{P}^T \otimes \vec{Q}$ is a stabiliser of $|U_1\rangle$ in the same way as in Lemma 6.3.15. Since $U_1 \vec{P} U_1^\dagger = U_2 \vec{P} U_2^\dagger$ we also have $U_2 \vec{P} = \vec{Q} U_2$, and hence $\vec{P}^T \otimes \vec{Q}$ is also a stabiliser of $|U_2\rangle$. As $\vec{P}$ was arbitrary, $|U_1\rangle$ and $|U_2\rangle$ have the same stabilisers and hence are equal up to a global phase α. Bending the wires back then shows that $U_1 = e^{i\alpha} U_2$. $\square$

We have been using two similar properties interchangeably: that a Pauli is mapped to a Pauli under conjugation, and that we can push a Pauli through a Clifford to get another Pauli. However, the equivalence between these relies on the map being unitary. It turns out that we can recover unitarity just by looking at whether we can push the Z_i and X_j through the map.

Lemma 6.4.2. Let $A : \mathbb{C}^{2^n} \to \mathbb{C}^{2^m}$ be a non-zero linear map such that $A Z_i = \vec{P}^i A$ and $A X_j = \vec{Q}^j A$ for all i and j, and some Pauli strings $\vec{P}^i$ and $\vec{Q}^j$. Then A is proportional to an isometry. In particular, if $n = m$, then A is proportional to a unitary.

Proof Taking adjoints in the equation $A Z_i = \vec{P}^i A$ we get $Z_i A^\dagger = A^\dagger \vec{P}^i$. We then calculate: $A^\dagger A Z_i = A^\dagger \vec{P}^i A = Z_i A^\dagger A$. Analogously, $A^\dagger A X_j = X_j A^\dagger A$ for all j. We can then check that the $2n$-qubit state $|A^\dagger A\rangle$ has the $2n$ stabilisers $Z_i \otimes Z_i$ and $X_i \otimes X_i$ for all $1 \leq i \leq n$ so that it is a stabiliser state. Furthermore, these stabilisers match those of $|\mathrm{id}_n\rangle$, where id_n is the n-qubit identity ($|\mathrm{id}_n\rangle$ is just n overlapping cups, connecting qubit i to $i + n$). Hence $A^\dagger A = e^{i\alpha} \mathrm{id}_n$ for some α. However, as $A^\dagger A$ is a positive map, we must have $\alpha = 0$ so that A is indeed an isometry. Now if $n = m$ then we have an isometry from the space to itself, and these maps are necessarily unitary. $\square$

This lemma will prove useful later in this section.

6.4.2 Stabiliser tableaux

To check whether an n-qubit unitary is Clifford there are $2n$ conditions we need to check, corresponding to the conjugation of each of the Z_i and X_i Pauli strings by the unitary for $1 \leq i \leq n$. We can write this information succinctly in a table known as a **stabiliser tableau**. To demonstrate this principle let's construct the stabiliser tableau for the CNOT gate. As it is a 2-qubit gate there are 4 conditions we need to check, corresponding to the conjugation of X_1, Z_1, X_2 and Z_2 by the CNOT. As demonstrated in Section 6.1.1, conjugating X_1 leads to the Pauli string XX and similarly Z_1 is mapped to ZI, X_2 to IX and Z_2 to ZZ. Each of these values corresponds to a column in the stabiliser tableau, which we can now write down:

$$
\begin{array}{c|cc|cc}
 & Z_1 & Z_2 & X_1 & X_2 \\
\hline
\pm & + & + & + & + \\
1 & Z & Z & X & I \\
2 & I & Z & X & X
\end{array}
\tag{6.16}
$$

The bottom two rows indicate the Pauli that each of the P_i is sent to on each of the qubits 1 and 2. The row labelled $\pm$ denotes whether the Pauli is sent to the Pauli listed, or minus that. For instance, conjugating Z by X gives $XZX = -ZXX = -Z$, and hence would have a minus sign in the respective column. This phase can indeed only be a $+1$ or -1: the Z_i and X_i are self-adjoint, and hence so are $UZ_iU^\dagger$ and $UX_iU^\dagger$.

These minus signs are important as they allow us to distinguish the Paulis just from their tableaux:

$$
I: \begin{array}{c|cc}
 & Z_1 & X_1 \\
\hline
\pm & + & + \\
1 & Z & X
\end{array}
\quad
X: \begin{array}{c|cc}
 & Z_1 & X_1 \\
\hline
\pm & - & + \\
1 & Z & X
\end{array}
\quad
Z: \begin{array}{c|cc}
 & Z_1 & X_1 \\
\hline
\pm & + & - \\
1 & Z & X
\end{array}
\quad
Y: \begin{array}{c|cc}
 & Z_1 & X_1 \\
\hline
\pm & - & - \\
1 & Z & X
\end{array}
\tag{6.17}
$$

We can use a stabiliser tableau to understand how the unitary acts on other Pauli strings than just those present in the tableau. For instance, suppose we wish to know how the Pauli string XY is changed when we conjugate it by a CNOT. We realise that XY is just the product $iX_1X_2Z_2$ and hence, writing $U = \text{CNOT}$, we get:

$$
U(XY)U^\dagger = iU(X_1X_2Z_2)U^\dagger = iUX_1U^\dagger UX_2U^\dagger UZ_2U^\dagger.
$$

But this latter expression is just the product of the three columns of the stabiliser tableau of the CNOT corresponding to X_1, X_2 and Z_2 (multiplied by a global factor of i). So we look at those columns in (6.16) and multiply

them together:

$$i \begin{pmatrix} + \\ X \\ X \end{pmatrix} * \begin{pmatrix} + \\ I \\ X \end{pmatrix} * \begin{pmatrix} + \\ Z \\ Z \end{pmatrix} = i \begin{pmatrix} + + + \\ XZ \\ XXZ \end{pmatrix} = \begin{pmatrix} + \\ Y \\ Z \end{pmatrix}$$

Hence, XY is mapped to YZ. If some of the $+$'s in the columns where $-$, then the standard rules for multiplying phase applies, e.g. "negative times negative is positive". For instance, if we take the stabiliser tableau of Y in (6.17), and wish to find the action of $Y_1 = iX_1Z_1$, we multiply the columns together as follows:

$$i \begin{pmatrix} - \\ X \end{pmatrix} * \begin{pmatrix} - \\ Z \end{pmatrix} i = \begin{pmatrix} + \\ XZ \end{pmatrix} = \begin{pmatrix} + \\ Y \end{pmatrix}$$

We then see that when Y is conjugated by Y we get Y, as we would expect.

The information presented in a stabiliser tableau is enough to fully characterise a Clifford.

Proposition 6.4.3. Let U_1 and U_2 be two n-qubit Clifford unitaries. Then U_1 and U_2 are equal up to global phase iff they have equal stabiliser tableaux.

Proof Of course, if $U_1 = e^{i\alpha}U_2$ for some global phase α, then they have equal stabiliser tableaux, so let's prove the converse. We have $U_1 Z_i U_1^\dagger = U_2 Z_i U_2^\dagger$ and $U_1 X_i U_1^\dagger = U_2 X_i U_2^\dagger$ for all $1 \leq i \leq n$. These Z_i and X_i generate all n-qubit Pauli strings and hence we have $U_1 \vec{P} U_1^\dagger = U_2 \vec{P} U_2^\dagger$ for all $\vec{P} \in \mathcal{P}_n$. The result then follows from Proposition 6.4.1. $\qquad\square$

Suppose we have a stabiliser tableau for a unitary U_1 and another one for a unitary U_2. Can we then easily construct the tableau for the composition of the unitaries $U_2 \circ U_1$? Suppose we wish to know the column in the stabiliser tableau of $U_2 \circ U_1$ corresponding to a Z_i. This is the Pauli string $U_2 U_1 Z_i U_1^\dagger U_2^\dagger$. Here the inner expression $U_1 Z_i U_1^\dagger$ corresponds to the column of Z_i in the tableau of U_1. This will be some Pauli string $P_1 P_2 \cdots P_n$, and so we need to know the values $U_2 P_j U_2^\dagger$ and multiply these together. But the expressions $U_2 P_j U_2^\dagger$ can be easily read from the tableau of U_2. This means that in order to know the Pauli on the jth qubit of the Z_i column in the tableau of $U_2 U_1$, we need to look at the column of Z_i in U_1, and at the row corresponding to the jth qubit of the tableau of U_2. This feels quite like matrix multiplication. In the next sections we will see that this is in fact, *exactly* like matrix multiplication.

6.4.3 Paulis as bit strings

We can represent elements of the Pauli group as certain bit strings. For instance, we will write:

$$X = (0 \,|\, 1) \, \text{\color{blue}+} \qquad Y = (1 \,|\, 1) \, \text{\color{blue}+} \qquad Z = (1 \,|\, 0) \, \text{\color{blue}+} \tag{6.18}$$

Generally, an element of the n-qubit Pauli group $\mathcal{P}_n$ will be represented as a vector of $2n$ bits, followed by an integer that is taken modulo 4:

$$sP_1 \otimes \ldots \otimes P_n = (\vec{z} \,|\, \vec{x}) \, \text{\color{blue}s} \,. \tag{6.19}$$

Here, the bits in the length-n bit strings $\vec{z}, \vec{x}$ are given by the Paulis P_i via the relation $P_i = i^{-z_i x_i} Z^{z_i} X^{x_i}$ where z_i, x_i are the ith bits of $\vec{z}$ and $\vec{x}$ and s is a complex number in the set $\{1, i, -1, -i\}$. As a shorthand, we write $\text{\color{blue}+}$ and $\text{\color{blue}-}$ for 1 and -1.

This mapping between Pauli operations and bit strings is just a convention, and there are probably other ones that work just as well. The useful feature of this representation is that multiplication of Pauli group elements corresponds to addition of bit strings modulo-2, followed by some book-keeping to get the correct scalar: if we have Pauli strings $\vec{P}$ and $\vec{Q}$ represented as bit strings $\vec{P} = (\vec{z}_1 \,|\, \vec{x}_1) \, \text{\color{blue}a}$ and $\vec{Q} = (\vec{z}_2 \,|\, \vec{x}_2) \, \text{\color{blue}b}$, then:

$$\vec{P}\vec{Q} = (\vec{z}_1 \oplus \vec{z}_2 \,|\, \vec{x}_1 \oplus \vec{x}_2) \, \text{\color{blue}c} \qquad \text{where} \qquad c := i^{\vec{z}_1 \cdot \vec{x}_2 - \vec{z}_2 \cdot \vec{x}_1} ab \tag{6.20}$$

We will use the notation $\star$ to represent combining bit-representations of Pauli operators in this way. That is:

$$(\vec{z}_1 \,|\, \vec{x}_1) \, \text{\color{blue}a} \,\star\, (\vec{z}_2 \,|\, \vec{x}_2) \, \text{\color{blue}b} := (\vec{z}_1 \oplus \vec{z}_2 \,|\, \vec{x}_1 \oplus \vec{x}_2) \, i^{\vec{z}_1 \cdot \vec{x}_2 - \vec{z}_2 \cdot \vec{x}_1} \, \text{\color{blue}ab}$$

A little "gotcha" to watch out for here is that addition of the bit strings is performed modulo-2, but the dot products that appear in the exponent of i use normal addition of integers (or addition modulo 4, since $i^4 = 1$). Instead of talking about 'bit strings' and 'addition modulo-2' it will be helpful to use a more mathematical way of talking about this, just like we had done in Chapter 4. We take the bits to be elements of field with two elements $\mathbb{F}_2 := \{0, 1\}$. Recall from Definition 4.1.1 that the addition operation in this field is defined by $0 + 0 = 0, 0 + 1 = 1 + 0 = 1, 1 + 1 = 0$, and the multiplication is the standard multiplication of the numbers 0 and 1. As this is a field, we can view the bit strings as being part of a vector space over $\mathbb{F}_2$. In particular, ignoring the global phases for now, we can represent n-qubit Pauli strings as elements of the vector space $\mathbb{F}_2^{2n}$.

We can use use these bit-representations to read of whether two Paulis

commute or not. For Paulis $\vec{P} = (\vec{z}_1 \mid \vec{x}_1)\,\boldsymbol{a}$ and $\vec{Q} = (\vec{z}_2 \mid \vec{x}_2)\,\boldsymbol{b}$ we have

$$\vec{P}\vec{Q} = \vec{Q}\vec{P} \iff \vec{z}_1 \cdot \vec{x}_2 \oplus \vec{z}_2 \cdot x_1 = 0 \ (\text{mod } 2).$$

The reason for this formula is that we are counting the number of places where there is a Z on a qubit i in $\vec{P}$ that matches to an X on a qubit i in $\vec{Q}$ (and vice versa for X's on $\vec{P}$ and Z's on $\vec{Q}$). Each of these matches results in an anti-commutation, but as long as there are an even amount, the Pauli strings commute.

It turns out that the expression $\vec{z}_1 \cdot \vec{x}_2 \oplus \vec{z}_2 \cdot x_1$ fits into the language of a well-studied subject in mathematics: symplectic forms.

Definition 6.4.4. Let $\mathbb{F}$ be a field (such as $\mathbb{F}_2$), and let $V = \mathbb{F}^{2n}$ be an even-dimensional vector space over this field. Then the **symplectic inner product** $\omega : V \times V \to \mathbb{F}$ on V is defined as

$$\omega((\vec{a}, \vec{b}), (\vec{c}, \vec{d})) = \vec{a} \cdot \vec{d} - \vec{b} \cdot \vec{c},$$

where $\vec{a}, \vec{b}, \vec{c}, \vec{d} \in \mathbb{F}^n$ are arbitrary vectors and $\cdot$ denotes the standard dot-product of elements of $\mathbb{F}^n$: $\vec{a} \cdot \vec{d} = \sum_i a_i d_i$.

In words: we take the dot product of the top half of the first vector with the bottom half of the second vector and subtract from this the dot product of the bottom half of the first vector with the top half of the second vector.

Another way we could have written this symplectic product is to realise that it is essentially the standard dot product, except that we have interchanged the top and bottom half of the second vector and introduced a negation. The matrix that does this operation is the following:

$$\Omega := \begin{pmatrix} 0_n & I_n \\ -I_n & 0_n \end{pmatrix} \tag{6.21}$$

Here, 0_n and I_n are respectively the $n \times n$ zero matrix and the $n \times n$ identity matrix. We can then write the symplectic inner product as $\langle v, w \rangle = v^T \Omega w$.

Note that if $\mathbb{F} = \mathbb{F}_2$ then $\vec{a} \cdot \vec{d} - \vec{b} \cdot \vec{c} = \vec{a} \cdot \vec{d} + \vec{b} \cdot \vec{c}$ (because $-1 = 1$ modulo 2). So combining what we've seen so far we see that we can represent length n Pauli strings as elements of the symplectic vector space $\mathbb{F}_2^{2n}$ (ignoring global phase), and that Pauli strings commute precisely when their symplectic inner product is zero. Additionally, the multiplication of Paulis corresponds (when we ignore phases) to addition of the vectors in this symplectic vector space. Let's capture this in the following proposition.

Proposition 6.4.5. Let $\mathcal{P}_n$ be the n-qubit Pauli group. Then there is a group homomorphism $S : \mathcal{P}_n \to \mathbb{F}_2^{2n}$ given by mapping a Pauli to its bit

string representation. The kernel of this representation is given by the global phases: $S(\vec{P}) = 0$ iff $\vec{P} = i^k I$. Furthermore, we have $\langle S(\vec{P}), S(\vec{Q}) \rangle = 0$ iff $\vec{P}$ and $\vec{Q}$ commute, where $\langle \cdot, \cdot \rangle$ is the symplectic inner product on $\mathbb{F}_2^{2n}$.

6.4.4 Cliffords as symplectic matrices

We have now seen that we can just treat Pauli strings as vectors in a symplectic vector space, and that, up to a phase, multiplying Pauli strings corresponds to adding their bit strings. If we have a Clifford unitary U, then this maps a Pauli string $\vec{P}$ to another Pauli string $U\vec{P}U^\dagger$, so we should be able to write this as a map on the symplectic vector space as well, right? Let's write $\hat{U} : \mathcal{P}_n \to \mathcal{P}_n$ for the group homomorphism on the Pauli group given by conjugating by U. That is, $\hat{U}(\vec{P}) = U\vec{P}U^\dagger$ (this is indeed a group homomorphism as $U\vec{P}\vec{Q}U^\dagger = U\vec{P}U^\dagger U\vec{Q}U^\dagger$). Note that $\hat{U}$ is in fact an isomorphism, as it has an inverse $\hat{U}^\dagger$. Let $S : \mathcal{P}_n \to \mathbb{F}_2^{2n}$ be the map from Proposition 6.4.5 that maps a Pauli string to its corresponding bit string in the symplectic vector space. Ignoring the phases in $\mathcal{P}_n$, this map is an isomorphism of (abelian) groups. We can then make a 'choice of inverse' $T : \mathbb{F}_2^{2n} \to \mathcal{P}_n$ by setting $T(\vec{z}, \vec{x}) := i^{\vec{z} \cdot \vec{x}} Z_{z_1} \cdots Z_{z_n} X_{x_1} X_{x_n}$. Note that this map produces self-adjoint Pauli strings, but that the map is not a group homomorphism (you can check that $T(1,1) \neq T(1,0)T(0,1)$). We see then that $S \circ T = \mathrm{id}_{\mathbb{F}_2^{2n}}$, but in the other direction we only have $TS(\vec{P}) \propto \vec{P}$. Now let's write $S(U) : \mathbb{F}_2^{2n} \to \mathbb{F}_2^{2n}$ for the map $S(U) := S \circ \hat{U} \circ T$. What properties does this map have?

Even though T is not a group homomorphism, it *is* the case that $S(U)$ is a group homomorphism of $\mathbb{F}_2^{2n}$, i.e. a linear map: $S(U)((\vec{z}_1, \vec{x}_1) + (\vec{z}_2, \vec{x}_2)) = S(U)(\vec{z}_1, \vec{x}_1) + S(U)(\vec{z}_2, \vec{x}_2)$. This is because T is a group homomorphism 'up to phase', and both $\hat{U}$ and S don't care about this phase, so that it doesn't change the outcome. Additionally, $S(U)$ has an inverse $S(U^\dagger)$ so that it is in fact a linear isomorphism. But it is not just any isomorphism. If the Pauli strings $\vec{P}$ and $\vec{Q}$ commute, then $\hat{U}(\vec{P})$ and $\hat{U}(\vec{Q})$ also commute, and similarly, if they didn't commute, then $\hat{U}$ preserves this property as well. We saw in the previous section that commuting Paulis are mapped by S to vectors whose symplectic inner product is zero. So putting these two facts together, we see that $S(U)$ must be mapping vectors in such a way to preserve the symplectic inner product. There is a special name for such maps.

Definition 6.4.6. Let $\mathbb{F}$ be a field, and let $\mathbb{F}^{2n}$ be a symplectic vector space. We call a linear map $A : \mathbb{F}^{2n} \to \mathbb{F}^{2n}$ **symplectic** when it preserves the sym-

plectic inner product: $\langle Av, Aw \rangle = \langle v, w \rangle$. Symplectic maps are automatically invertible, and hence form a group that we will call $\mathrm{Sp}(\mathbb{F}^{2n})$.

For Hilbert spaces we care a great deal about unitaries, which are maps that preserve the inner product. Similarly, for symplectic vector spaces, we care about the maps that preserve the symplectic inner product.

As $\hat{U}$ preserves commuting Paulis, $S(U)$ must be a symplectic map. Writing $\mathcal{C}_n$ for the n-qubit Clifford group, S then gives a group homomorphism $S : \mathcal{C}_n \to \mathrm{Sp}(\mathbb{F}_2^{2n})$. This map is not injective, and that's because $\mathbb{F}_2^{2n}$ captures the Pauli group only up to phase. A Clifford U is in the kernel of this homomorphism, i.e. $S(U) = \mathrm{id}_{\mathbb{F}_2^{2n}}$ when $\hat{U}$ maps every Pauli to itself up to a phase. This phase can only be ± 1 for each Pauli, as a Clifford maps self-adjoint Paulis to self-adjoint Paulis under conjugation. The Pauli strings themselves are examples of Cliffords that are sent to the kernel. When $U = \vec{P}$ is a self-adjoint Pauli string, we see that indeed $U\vec{Q}U^\dagger = \vec{P}\vec{Q}\vec{P} = \pm\vec{P}\vec{P}\vec{Q} = \pm\vec{Q}$, so that $S(U) = \mathrm{id}_{\mathbb{F}_2^{2n}}$. The converse turns out to also be true.

Proposition 6.4.7. Let U be a Clifford such that for every Pauli string $\vec{P}$ we have $U\vec{P}U^\dagger = \pm\vec{P}$. Then U is itself a Pauli string (up to global phase).

Proof Write $\hat{U}$ for the conjugation map. We will construct a Pauli string $\vec{Q}$ that acts the same on all Paulis by conjugation as $\hat{U}$, and hence by Proposition 6.4.1, U must be equal to $\vec{Q}$ up to global phase. First we determine Q_1. Write $\hat{U}(X_1) = aX_1$, $\hat{U}(Z_1) = bZ_1$ for $a, b \in \{1, -1\}$. Then if $a = b = 1$, we set $Q_1 = I$. If $a = -1$ and $b = 1$, we set $Q_1 = Z$. If $a = 1$ and $b = -1$ we set $Q_1 = X$, and finally if $a = b = -1$ we set $Q_1 = Y$. It is then easy to check that $\hat{U}(P_1) = \hat{\vec{Q}}(P_1)$ for any Pauli $P \in \{I, X, Y, Z\}$. We similarly set $Q_2, \ldots, Q_n$, so that $\hat{U}$ and $\hat{\vec{Q}}$ agree on all Pauli strings P_i. As these generate all Pauli strings, we must then have $\hat{U} = \hat{\vec{Q}}$ and we are done. $\qquad\square$

With this proposition we see that $S(U) = S(V)$ iff $U = e^{i\alpha}V\vec{Q}$ for some global phase α and Pauli string $\vec{Q}$. So just like how the symplectic representation of the Pauli strings ignores the global phase, the symplectic representation of the Cliffords ignores Paulis.

Okay, so when thinking about the symplectic space, and ignoring Paulis, the Cliffords become symplectic maps. What about the converse? If we are given a symplectic map on $\mathbb{F}_2^{2n}$, can we find a Clifford that is mapped to it? In other words: is S surjective?

To answer this questions it will be useful to define a 'standard basis' of $\mathbb{F}_2^{2n}$. Write $z_i = S(Z_i)$ and $x_j = S(X_j)$ for the bit strings corresponding to the Pauli strings Z_i and X_j. These z_i and x_j of course span the space and

are linearly independent so that they are indeed a basis. Note furthermore that the symplectic inner products of these are $\langle z_i, z_j \rangle = \langle x_i, x_j \rangle = 0$, while $\langle z_i, x_j \rangle = \delta_{ij}$, which encodes the fact that the Z_i all mutually commute (and the same for the X_j), while Z_i and X_j anti-commute when $i = j$.

Proposition 6.4.8. Every symplectic map $M : \mathbb{F}_2^{2n} \to \mathbb{F}_2^{2n}$ comes from a Clifford unitary U via $S(U) = M$.

Proof Define $v_i := Mz_i$ and $w_j := Mx_j$, the images of the standard basis after applying M. As M is symplectic and hence invertible, the vectors v_i and w_j must then also be independent. Furthermore, as M preserves the symplectic inner product we still have $\langle v_i, v_j \rangle = \langle w_i, w_j \rangle = 0$ and $\langle v_i, w_j \rangle = \delta_{ij}$. Let $\vec{P}^i$ be a self-adjoint Pauli string for which $S(\vec{P}_i) = v_i$ and similarly let $\vec{Q}_j$ be such that $S(\vec{Q}_j) = w_j$ (these Pauli strings are unique up to phases of ± 1). Then the Paulis $\vec{P}_i$ are commuting, since $\langle v_i, v_j \rangle = 0$, and similarly all the $\vec{Q}_j$ are commuting. The only pairs that don't commute are $\vec{P}_i$ and $\vec{Q}_j$ when $i = j$.

We can 'fix' this lack of commutation by considering longer Pauli strings. Consider the following $2n$ Pauli strings acting on $2n$ qubits:

$$Z_1 \otimes \vec{P}_1, \ldots Z_n \otimes \vec{P}_n, X_1 \otimes \vec{Q}_1, \ldots, X_n \otimes \vec{Q}_n.$$

These Pauli strings all commute. Let's check the only non-trivial commutation: $Z_i \otimes \vec{P}_i$ and $X_j \otimes \vec{Q}_j$. When $i \neq j$, we have that Z_i and X_j commute, and $\vec{P}_i$ and $\vec{Q}_j$ commute, so that the full strings obviously commute as well. When $i = j$, then Z_i and X_i anti-commute, but so do $\vec{P}_i$ and $\vec{Q}_i$, so that the complete strings still commute.

We then have built $2n$ independent, commuting, self-inverse Pauli strings acting on $2n$ qubits. These then uniquely define a $2n$-qubit stabiliser state $|\psi\rangle$, which is unique up to scalar. Bending back the top n wires to be inputs, we then get a linear map A going from n qubits to n qubits. We want to show it is unitary. As $Z_i \otimes \vec{P}_i$ is a stabiliser of $|\psi\rangle$, we then see that $AZ_i = \vec{P}_i A$:
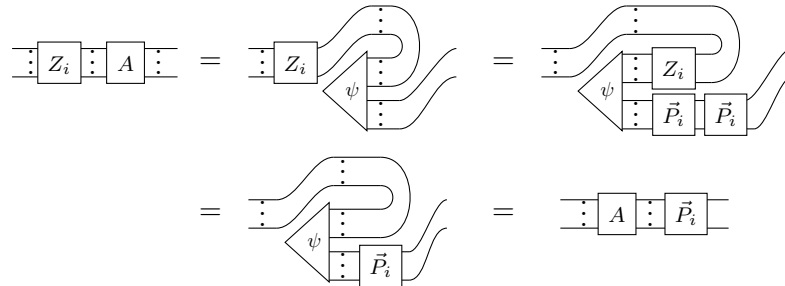
Similarly, $AX_j = \vec{Q}_j A$. Lemma 6.4.2 then shows that A is (proportional to a)

unitary, and as it comes from a stabiliser diagram it is Clifford. Furthermore $AZ_iA^\dagger = \vec{P_i}AA^\dagger = \vec{P_i}$, so that $S(A)z_i = v_i$. We defined v_i to be such that $Mz_i = v_i$, and hence $S(A)z_i = Mz_i$. Completely analogously we have $S(A)x_j = Mx_j$, so that $S(A)$ and M agree on a basis of $\mathbb{F}_2^{2n}$, so that indeed $S(A) = M$. $\square$

6.4.5 Adding back in the phases

So far we have ignored the phases in the symplectic representation. What happens when we try to add them back in? We get stabiliser tableaux!

Let's look again at the stabiliser tableau for the CNOT gate:

	Z_1	Z_2	X_1	X_2
$\pm$	$+$	$+$	$+$	$+$
1	Z	Z	X	I
2	I	Z	X	X

Now for each of the elements in the table we make the substitutions, corresponding to how we were encoding the Paulis as bitstrings:

$$I \rightsquigarrow \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad Z \rightsquigarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad X \rightsquigarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad Y \rightsquigarrow \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

We then get the following matrix of values:

	Z_1	Z_2	X_1	X_2
$\pm$	$+$	$+$	$+$	$+$
$1Z$	1	1	0	0
$1X$	0	0	1	0
$2Z$	0	1	0	0
$2X$	0	0	1	1

Now let's group the Z and X rows together:

$$
\begin{array}{c|cc|cc}
 & Z_1 & Z_2 & X_1 & X_2 \\
\hline
\pm & + & + & + & + \\
1Z & 1 & 1 & 0 & 0 \\
2Z & 0 & 1 & 0 & 0 \\
\hline
1X & 0 & 0 & 1 & 0 \\
2X & 0 & 0 & 1 & 1 \\
\end{array}
\tag{6.22}
$$

If we squint a bit we can see that this is in fact a symplectic matrix, but with some additional information to record the phases. If we remove those

phases and the label for the columns and rows we in fact get the symplectic matrix representation for the CNOT:

$$S(\text{CNOT}) \;=\; \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Then if we want to know for instance what happens when we conjugate $Y \otimes X$ by CNOTs, we first write $Y \otimes X$ as the vector $S(Y \otimes X) = (1, 0, 1, 1)^T$, and then we just multiply them:

$$S(\text{CNOT})S(Y \otimes X) \;=\; \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \;=\; \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

This symplectic representation doesn't contain the phase information though. We can fix this by just keeping the top row of the tableau 6.22 containing the phases, and also adding another top row to the Paulis to represent their global phase. If we do this, we however also have to modify how we multiply together tableaux and vectors, as the phase gets updated in quite a complicated way. This is not really relevant for us, so we leave it as an exercise.

Exercise* 6.14. Find out how to incorporate phases as an additional row on the symplectic matrix, so that matrix-vector multiplication works out the way it should.

6.4.6   Putting it all together

Let's summarise what we have learned. We have seen that we can write a Pauli string as a bit string, where the first half of the bits tells you where the Z's are, and the second half tells you where the X's are (and so the overlap tells you where the Y's are). We can then consider the Pauli group as a vector space of bit strings $\mathbb{F}_2^{2n}$.

Paulis either commute or anti-commute. This information is captured in $\mathbb{F}_2^{2n}$ by whether the associated vectors have a symplectic inner product of zero (for commuting) or one (for anti-commuting).

As Cliffords map Paulis to Paulis under conjugation, and are completely determined by this action, we can represent Cliffords as matrices on the symplectic vector space $\mathbb{F}_2^{2n}$. Conjugating by a Clifford preserves commutation, so that it preserves the symplectic inner product on this space. This means

that the Cliffords correspond to *symplectic maps*. Conversely, any symplectic map on $\mathbb{F}_2^{2n}$ comes from a Clifford unitary.

These constructions with the symplectic phases ignore the phases of the Paulis, and hence capture the Cliffords only up to local Paulis. We can fix this by adding some additional phase information to our symplectic space. This representation is then equivalent to the stabiliser tableau of the Clifford.

6.5   The Clifford hierarchy

In Chapter 5 we introduced Clifford unitaries in a somewhat ad-hoc way as those unitaries built from CNOT, Hadamard and S gates. In this chapter we have seen that we can define them in a mathematically nicer way as precisely those unitaries that send Pauli strings to Pauli strings under conjugation. It turns out we can extend this idea to a whole hierarchy of gates.

First, let's denote the set of Pauli gates by $\mathcal{C}_1$.

Definition 6.5.1. The kth level of the **Clifford hierarchy** for $k > 1$ is defined as
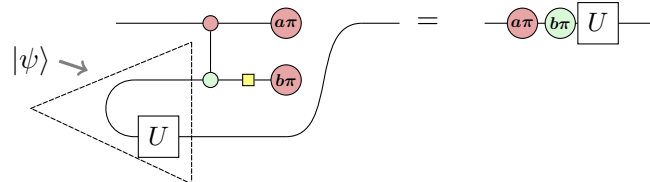
$$\mathcal{C}_k \; := \; \big\{\, U \,|\, \forall \vec{P} : U\vec{P}U^\dagger \in \mathcal{C}_{k-1} \,\big\}$$

The first level of the hierarchy consists of the Paulis by definition, and the second $\mathcal{C}_2$ contains unitaries that map Pauli strings to unitaries in $\mathcal{C}_1$, i.e. the Paulis. Hence, $\mathcal{C}_2$ consists precisely of the Cliffords. The third level $\mathcal{C}_3$ then contains unitaries that map Pauli strings to Clifford unitaries. An example of a third-level unitary is the T gate $-\!\boxed{\tfrac{\pi}{4}}\!-$:

$$TZT^\dagger = Z \qquad TXT^\dagger \propto SX$$

Similarly, the $Z(\tfrac{\pi}{8})$ gate is in the 4th level, and more generally $Z(\tfrac{2\pi}{2^k})$ is in the kth level.

The Clifford hierarchy is useful for thinking about **gate teleportation** protocols. As the name implies, gate teleportation generalises the idea behind quantum teleportation of Section 3.3.2. In quantum teleportation we move information from one qubit to another without changing the state. In gate teleportation we instead try to apply some specific unitary U to the state, without directly applying U to that qubit:

$$\tag{6.23}$$

In this protocol we entangle our state with a specific maximally entangled state $|\psi\rangle$ with the goal of executing U. However, we see that instead we end up with UZ^bX^a. In order to actually end up with U, we hence need to apply a correction of $UX^aZ^bU^\dagger$. For a general U this doesn't help us, since this requires us to execute U on the qubit after all. However, if $U \in \mathcal{C}_k$, then $UX^aZ^bU^\dagger \in \mathcal{C}_{k-1}$, so that instead of applying a $\mathcal{C}_k$ unitary, we can execute a $\mathcal{C}_{k-1}$ unitary. In Eq. (6.23) U is a single-qubit unitary, but the same idea works for an n-qubit unitary, in which case we essentially do the same operations on each of the n qubits in parallel, with the correction depending on a Pauli string of possible measurement outcomes.

These gate teleportation protocols are especially useful when it is hard to correctly implement U. We can then first try to build the correct $|\psi\rangle$ "offline", until we get the result we require, and then apply it to the state we want by gate teleportation. If the correction operation is then easy, we can just apply that and we are done. For instance, we will see in Chapter 11 that in many fault-tolerant architectures it is difficult to execute a T gate, while it is easy to implement Clifford gates. As a T gate is in the third level of the Clifford hierarchy, we can implement it with gate teleportation, and its correction will be a Clifford. If instead we want to implement a 4th-level unitary, like $\sqrt{T} = Z(\frac{\pi}{8})$, its correction will be third-level, which we then also implement with gate teleportation, so that the final correction is a Clifford.

Note that we in fact have already encountered a version of T gate teleportation in the form of state injection in Section 3.3.1:
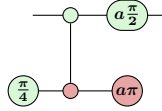
This can be rewritten into a more standard gate teleportation protocol:
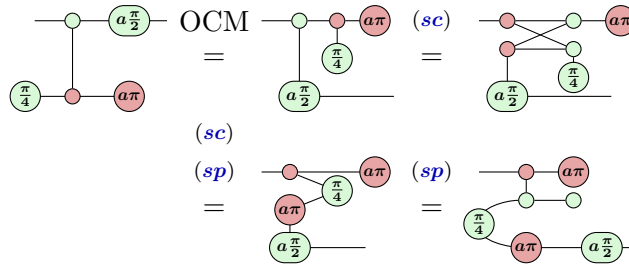
Note though that while gate teleportation requires a two-qubit state, magic state injection requires just a single-qubit state, and hence is more efficient. In general, we can do the more efficient 'magic state' protocol for a unitary

U when U **semi-Clifford**, which means that $U = C_1 D C_2$ where C_1 and C_2 are Clifford and D is diagonal.

While there is a nice connection between the Clifford hierarchy and how hard it is to implement a unitary through gate teleportation, in pretty much any other regard the Clifford hierarchy is not very nice at all. In particular, for $k > 2$, $\mathcal{C}_k$ is not even closed under composition. It is hence not possible to describe the elements of $\mathcal{C}_k$ just by enumerating some small number of generators. In fact, a full characterisation of $\mathcal{C}_k$ for arbitrary number of qubits and k is not known.

Exercise 6.15. Let C be any Clifford and $U \in \mathcal{C}_k$. Show that $CU \in \mathcal{C}_k$ and $UC \in \mathcal{C}_k$.

Exercise 6.16. Show that $U = THT$ is not anywhere in the Clifford hierarchy. *Hint: Show that $UXU^\dagger$ is within a Clifford of U itself, so that $UXU^\dagger$ being in $\mathcal{C}_{k-1}$ implies U is as well.*

When we restrict to the *diagonal* unitaries in $\mathcal{C}_k$ the structure is a lot clearer. In particular, let $\mathcal{D}_k \subset \mathcal{C}_k$ denote the diagonal unitaries in the kth level. Then $\mathcal{D}_k$ is in fact closed under composition and forms a group, and is generated by unitaries of the form:
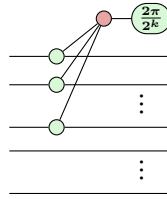
up to permutations of the qubits. These diagrams implement the unitary $|x_1, \ldots, x_n\rangle \mapsto e^{2i\frac{\pi}{2^k}(x_1 \oplus \cdots \oplus x_n)}|x_1, \ldots, x_n\rangle$ and are the focal point of a lot of discussion in the next chapter.

6.6   Summary: What to remember

1. The n-qubit Pauli strings form a group $\mathcal{P}_n$.
2. We can represent the projection onto the $+1$-eigenspace of a Pauli using a *Pauli box*. This allows us to treat all Paulis on the same footing diagrammatically.
3. We call a subgroup $\mathcal{S} \subseteq \mathcal{P}_n$ a stabiliser group if it stabilises at least one non-zero state. This is the case exactly when $-I \notin \mathcal{S}$. Stabiliser groups are necessarily commutative.

4. The Fundamental Theorem of Stabiliser Theory: If an n-qubit stabiliser group has k generators, then its stabiliser subspace has dimension 2^{n-k}.
5. Maximal stabiliser groups have precisely n generators, and stabilise a unique state up to scalar factor. We call states stabilised by a maximal stabiliser group a *stabiliser state.*
6. Stabiliser states correspond precisely to the Clifford states of Chapter 5.
7. The Clifford unitaries of Chapter 5 send elements of $\mathcal{P}_n$ to $\mathcal{P}_n$ under conjugation. Any unitary with this property is in fact Clifford.
8. We can fully describe a Clifford unitary by a stabiliser tableau, which lists its action on the Z_i and X_i Pauli strings under conjugation.
9. Additionally, we can describe Pauli strings, up to global phase, as elements of a symplectic vector space.
10. We can then describe Clifford unitaries, up to multiplication with Paulis, by a symplectic matrix: a matrix preserving the symplectic inner product.

6.7 References and further reading

Cliffords as normalising the Pauli group The stabiliser formalism was introduced by Gottesman (1996) and further developed, with the help of some amusing examples, in Gottesman (1998). In the former paper, it was treated mainly as a tool for defining quantum error correcting codes, whereas in the latter as a more general-purpose set of tools for dealing with quantum operations.

In Chapter 5 we introduced Clifford unitaries as those built out of Hadamard, S and CNOT gates, and now in this chapter we see that they can also be understood to be those unitaries that send Paulis to Paulis under conjugation. Originally, the latter representation came first. That stabiliser unitaries (those that map Paulis to Paulis) can be implemented just using Hadamard, S and CNOT gates was noted in the context of error correcting codes in Bennett et al. (1996), and a proof was given using symplectic matrices in Calderbank et al. (1997). A more concrete proof by induction was presented in Gottesman (1997). The modern form of using a stabiliser tableau to simulate a computation or to produce a normal form for the circuit was presented in Aaronson and Gottesman (2004). Note that in different papers on this subject the rows and columns of a tableau might be interchanged.

The Clifford hierarchy The notion of a Clifford hierarchy is from Gottesman and Chuang (1999) where they also introduce the technique of gate teleportation with these gates. A characterisation of diagonal gates in the Clifford

hierarchy is given in Cui et al. (2017). That you can do gate teleportation
with a smaller ancilla state is the unitary is semi-Clifford is from Zhou et al.
(2000), though the term 'semi-Clifford' was only introduced in Gross and
Van den Nest (2008).

7

Universal circuits

We will now turn from studying Clifford circuits to universal ones. Whereas the Clifford circuits were generated from CNOT, H, and S gates, we can obtain an exactly universal set of gates by replacing the $S := Z[\frac{\pi}{2}]$ gate with the family of $Z[\alpha]$ gates for arbitrary angles α. Perhaps more miraculously, we can obtain an approximately universal set of gates by replacing S with some fixed phase gate $Z[\theta]$ for *any* θ that isn't a multiple of $\pi/2$, the most popular choice being $T := \sqrt{S} = Z[\frac{\pi}{4}]$.

This tiny tweak takes us from the safe, efficient world of Clifford circuits to the wild and wonderful world of full-powered quantum computation. Given we can no longer efficiently compute the outputs of a universal quantum circuit with a classical computer, one might wonder how much we *can* still reason about these circuits. In this chapter we'll see that the answer is, perhaps surprisingly, quite a lot!

In this chapter we will see two different ways to think of universal circuits: **path sums** and **Pauli exponentials**.

Path sums are an extension of the idea of phase polynomials. We already briefly encountered phase polynomials in Chapter 5 when discussing the AP normal form. In the same way that we could fully characterise a CNOT circuit as a parity matrix in Chapter 4, it turns out we can fully characterise a circuit consisting of CNOT gates and phase gates as a phase polynomial together with a parity matrix. However, this representation only deals with Hadamard-free circuits. To also work with Hadamards we need path sums. We can write the action of a Hadamard as $|x\rangle \sum_y (-1)^{x \cdot y} |y\rangle$. In a path sum we interpret this as introducing a new variable y on your qubit, and adding a term $(-1)^{x \cdot y}$ to your phase polynomial. The final expression for your circuit then contains a sum $\sum_y$ over all the different 'paths' a computational basis state can take trough your circuit.

On the other hand, Pauli exponentials offer quite a different perspective on universal circuits. We have seen in the previous chapters that there is a whole lot to be said about the Pauli gates. They allow us to define stabiliser theory and Pauli projections, which lead to the idea of stabiliser states and Cliffords, which have a rich rewrite theory and will allow us to define quantum error correction in Chapter 11. But on the other hand the Clifford computation we have seen is efficiently classically simulable, so if we want to get our money's worth with a real quantum computer, we had better do something more than just Cliffords. Luckily we don't have to look far, we can still work with Paulis, but just in a slightly modified form.

It turns out that for any Pauli $\vec{P}$ we can construct a family of unitaries out of it by considering the matrix exponential $e^{i\theta\vec{P}}$ for phases $\theta \in \mathbb{R}$. The resulting **Pauli exponentials** form a universal gate set for quantum computing. Certain Pauli exponentials also correspond to the native gate set of several types of quantum computers, like ion traps and superconducting quantum computers.

In fact, Pauli exponentials form arguably *the* native gate set to understand quantum computation. They have a natural relation to Clifford computation; the number of them in a circuit directly corresponds to the cost of classically simulating a quantum computation; Hamiltonian simulation can be directly understood in terms of Pauli exponentials; Pauli exponentials can be readily understood in certain quantum error correcting codes like the surface code; and finally, which is important for us, Pauli exponentials have a very natural representation in the ZX-calculus.

7.1 Path sums

7.1.1 Phase polynomials

First, let us recap what we know about phase polynomials. Lets begin with a neat trick that works for any circuit built only out of CNOT and Z-phase gates. We already learned back in Chapter 4 that CNOT circuits correspond to parity maps, i.e. linear maps which send basis states to other basis states that can be described as the parities of input variables. We also saw that a convenient way to compute the parity map is to label each input wire with

a distinct variable and simply push those variables through the circuit:

$$(7.1)$$
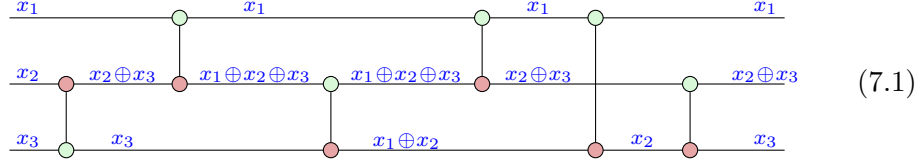
That is, we start with only the input wires labelled and work from left to right. When a CNOT is encountered, we label the output of the control qubit with the same label as the input of the control qubit and label the output of the target qubit with the XOR of the labels on the two input qubits.

In fact, this is the same as computing the overall parity matrix by doing one primitive row operation (i.e. CNOT gate) at a time. Indeed if we look at the action of the unitary U described in (4.1), it maps basis states to an XOR of basis states given by the expression on the output wires:

$$U :: |x_1, x_2, x_3\rangle \mapsto |x_1, x_2 \oplus x_3, x_3\rangle \tag{7.2}$$

We also saw in Chapter 4 that any two CNOT circuits computing the same parity function are equal. By re-synthesising a circuit according to its parity function, we can often find a much more efficient implementation. In the example above, we can implement the unitary U with just one CNOT gate:

$$(7.3)$$

Now, lets sprinkle some Z-phase gates into the circuit (4.1) above:

$$(7.4)$$
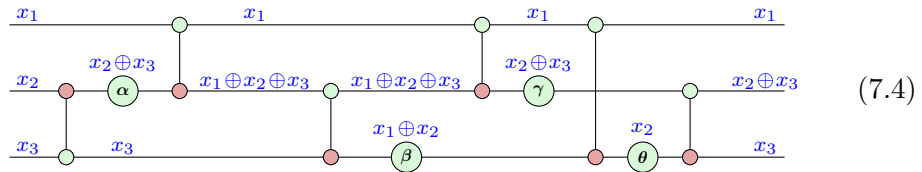
Note that we left the parity labels on the wires. Why are we justified in doing that? Z-phase gates are diagonal in the Z basis, so they preserve basis elements, up to a phase which depends on the basis element $|x\rangle$:

$$Z[\alpha] :: |x\rangle \mapsto e^{i\alpha \cdot x}|x\rangle$$

We can now plug a single computational basis element $|x_1, x_2, x_3\rangle$ into (7.4) and track its progress through the circuit. Applying the first two gates,

we see the CNOT first updates the variables in the ket, then the Z-phase gate introduces a new phase, which depends on whether $x_2 \oplus x_3$ is 0 or 1:

$$|x_1, x_2, x_3\rangle \mapsto |x_1, x_2 \oplus x_3, x_3\rangle$$
$$\mapsto e^{i\alpha \cdot (x_2 \oplus x_3)}|x_1, x_2 \oplus x_3, x_3\rangle$$

Continuing this through the rest of the circuit, the overall expression we get for the unitary V described by circuit (7.4) is:

$$V :: |x_1, x_2, x_3\rangle \mapsto e^{i[(\alpha+\gamma) \cdot (x_2 \oplus x_3) + \beta \cdot (x_1 \oplus x_2) + \theta \cdot x_2]}|x_1, x_2 \oplus x_3, x_3\rangle \qquad (7.5)$$

More generally, *any* CNOT+phase circuit yields a unitary of the form:

$$U :: |\vec{x}\rangle \mapsto e^{i\phi(\vec{x})}|L\vec{x}\rangle$$

for some $\mathbb{F}_2$ matrix L and function $\phi : \mathbb{F}_2^n \to \mathbb{R}$. We already met the parity matrix L, which describes U's action on basis vectors as an $\mathbb{F}_2$-linear map. To this we add ϕ, which is called the **phase polynomial**. This describes the phase associated with each input as a polynomial over XOR's of the input bits.

Just as we could read the parity map off the labelled CNOT circuit (4.1), we can read both the parity map and the phase polynomial off of the labelled circuit (7.4). As before, we can read the parity map off the output wires. For the phase polynomial, we introduce a term of $\alpha \cdot (x_{j_1} \oplus \ldots \oplus x_{j_k})$ for each Z-phase gate $Z[\alpha]$ occurring on a wire labelled $x_{j_1} \oplus \ldots \oplus x_{j_k}$.

We can now come up with a new circuit that implements the same parity map and phase polynomial. We will give a reasonably efficient, general-purpose circuit synthesis algorithm for this in Section 7.2, but for now lets just focus on our example. We already saw that, by ignoring the phase gates, we could produce a very efficient circuit (4.3) to implement the parity map $|x_1, x_2, x_3\rangle \mapsto |x_1, x_2 \oplus x_3, x_3\rangle$ of the unitary V. In order to get the full unitary, we should inspect the phase polynomial of V:

$$\phi(x_1, x_2, x_3) = (\alpha + \gamma) \cdot (x_2 \oplus x_3) + \beta \cdot (x_1 \oplus x_2) + \theta \cdot x_2$$

Going term-by-term, it looks like we need to place Z-phase gates on wires labelled with the expressions $x_2 \oplus x_3$, $x_1 \oplus x_2$, and x_2. Inspecting the circuit in (4.3), we see that we already have wires labelled by $x_2 \oplus x_3$ and x_2. Unfortunately, $x_1 \oplus x_2$ is missing. However, we can temporarily make this

parity available by introducing a redundant pair of CNOT gates:
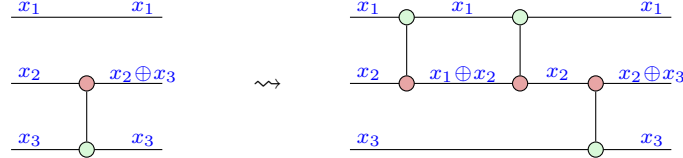
We can then recover the full circuit for V by placing Z-phase gates $\alpha + \gamma$ on any wire labelled $x_2 \oplus x_3$, β on $x_1 \oplus x_2$ and θ on x_2:

(7.6)
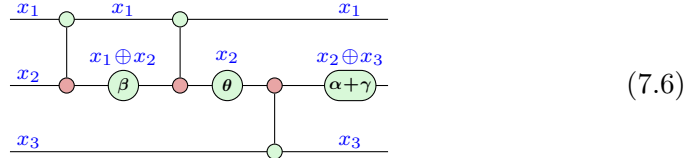
By computing the phase polynomial and re-synthesising, we reduced the circuit (7.4), which had 6 CNOT gates and 4 Z-phase gates to (7.6), which has 3 CNOT gates and 3 Z-phase gates, giving a significant savings. Notably, if two phases occur at the *same* parity, even in totally different parts of the circuit, their angles add in the phase polynomial. Hence they can always be re-synthesised into the same phase gate. This phenomenon is called **phase-folding**, and is one of the major advantages of the phase polynomial approach. For example, two phase gates in different parts of circuit (7.4) contributed a single term $(\alpha + \gamma) \cdot (x_2 \oplus x_3)$ to the phase polynomial in (7.5). When we re-synthesised the circuit in (4.3), they became a single phase gate.

Exercise 7.1. Show that the phase polynomial representation also works for circuits made of CNOT, Z-phase, and X gates, using the fact that an X gate updates wire labels as follows:

More specifically, give an example circuit, compute its parity map and phase polynomial by wire-labelling, and re-synthesise a smaller circuit.

Finally, show that further simplifications are possible: namely expressions of the form $\alpha \cdot y + \beta \cdot (y \oplus 1)$, where y is some XOR of input variables, can be simplified in the phase polynomial, up to global phases.

7.1.2 Phase gadgets

Phase polynomials give a very powerful, efficient way to manipulate CNOT+phase circuits. Some might even (blasphemously!) suggest that they are more effective than graphical techniques for many applications. However, it is nevertheless useful to see how these structures are reflected in the ZX-calculus. Along

the way, we will meet certain little ZX diagrams called **phase gadgets**, which correspond exactly to the terms in a phase polynomial. We'll meet phase gadgets a lot in the coming chapters, as they are really useful tools for lots of different applications, so it is worth spending some time now getting a handle on how and why they work.

In this section, we'll see how phase polynomials and in particular phase-folding show up graphically. Note that in some cases phase folding follows trivially from an application of spider fusion. For example, this circuit:
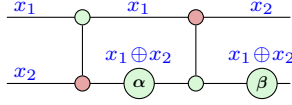
gives the following unitary:

$$U :: |x_1, x_2\rangle \mapsto e^{i(\alpha+\beta)\cdot(x_1\oplus x_2)}|x_2, x_1 \oplus x_2\rangle$$

We see that the phases α and β combine in the phase polynomial. This can also be seen by commuting the α phase gate through the control wire of the last CNOT gate via spider fusion:
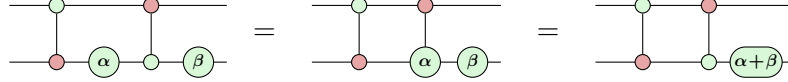
However, as we saw in the last section, phases can merge together even in completely different parts of the circuit. In this case, there is no obvious way to apply spider fusion to merge them. Hence, we'll need to develop a generalisation of spider fusion, which allows distant phases in a CNOT+phase circuit to find each other.

Let's start with a slight variation on the example from the previous section:

 (7.7)
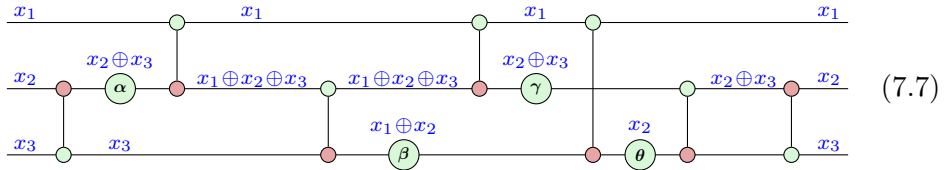
All we've done is add one more CNOT gate at the end to make the overall parity map L trivial. This unitary is now diagonal in the computational basis. That is, if we wrote down its matrix, it would only have a bunch of phases down the diagonal, as described by the phase polynomial. Explicitly, the circuit above implements the following unitary:

$$U :: |x_1, x_2, x_3\rangle \mapsto e^{i[(\alpha+\gamma)\cdot(x_2\oplus x_3)+\beta\cdot(x_1\oplus x_2)+\theta\cdot x_2]}|x_1, x_2, x_3\rangle \qquad (7.8)$$

This circuit has 4 unknown, possibly non-Clifford phases in it, so none of

the strategies we've studied so far let us reduce the whole circuit to any kind of normal form. However, what we can do is unfuse the unknown phases:
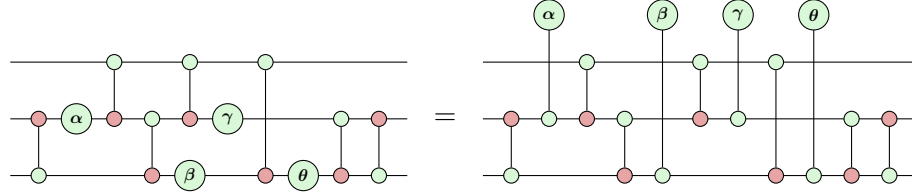
Now, we can treat the 4 wires connecting to unknown phases as outputs of the diagram and apply the simplification strategy we used back in Chapter 4 for CNOT circuits to the rest of the (phase-free) ZX-diagram. This diagram where we treat the phases as additional outputs is an isometry, and so after simplifying, we'll get a parity normal form which only has Z spiders adjacent to inputs and only has X spiders adjacent to outputs and our 4 phases. Here's what this gives:

$$(7.9)$$
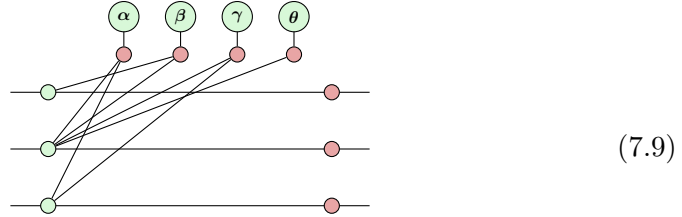
Here, the actual output wires have all ended up with a 2-legged X spider, i.e. an identity map. This corresponds to the fact that the parity map L is trivial. We will refer to this reduced form as the **phase gadget form** of a ZX diagram. These are so named after **phase gadgets**, which are little compound creatures consisting of a Z phase spider connected to a phase-free X spider with k additional legs:

$$(7.10)$$
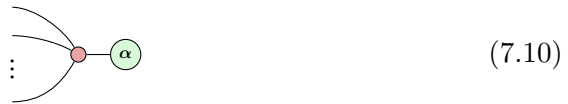
We will often treat these two spiders together as a single node in our diagram. Seen as a map with k inputs and 0 outputs, a phase gadget takes a computational basis state to its XOR, then depending on whether the XOR is 0 or 1, this produces a scalar of either 1 or $e^{i\alpha}$. To see this, lets first think about a 1-legged Z spider as a map from 1 qubit to 0 qubits, i.e. it sends a single-qubit state to a number. On the computational basis, it acts as follows:

$$\underline{\qquad \alpha}\ ::\ |x\rangle \mapsto \begin{cases} 1 & \text{if } x = 0 \\ e^{i\alpha} & \text{if } x = 1 \end{cases}$$

We can summarise this using a very simple phase polynomial:

$$—\!\!\alpha \quad :: |x\rangle \mapsto e^{i\alpha \cdot x}$$

Recall from Section 3.1.1 that X spiders act like XORs on computational basis states, up to a scalar factor:

$$\sqrt{2}^{(k-1)} \cdot \quad \vdots\!\!\!\!\!\!\!\!\!\bigtriangleup\!\!\!\!— \quad :: |x_1, \ldots, x_k\rangle \mapsto |x_1 \oplus \ldots \oplus x_k\rangle$$

Putting these two together, we see that the phase gadget itself does this:

$$\sqrt{2}^{(k-1)} \cdot \quad \vdots\!\!\!\!\!\!\!\!\!\bigtriangleup\!\!\!\!\alpha \quad :: |x_1, \ldots, x_k\rangle \mapsto e^{i\alpha \cdot (x_1 \oplus \ldots \oplus x_k)}$$

In order to get a unitary map from a phase gadget, we can use a bunch of Z spiders to make a copy of the basis state. One copy is sent to the phase gadget and one copy is output. This results in the following diagonal unitary:

$$:: |x_1, \ldots, x_k\rangle \mapsto e^{i\alpha \cdot (x_1 \oplus \ldots \oplus x_k)}|x_1, \ldots, x_k\rangle \qquad (7.11)$$

Note that we have suppressed a $\sqrt{2}^{(k-1)}$ scalar factor. Since we'll be working with unitaries in the rest of this section, we'll just assume the overall scalar is chosen to make everything unitary.

Returning to (7.9), we can see that this diagram can be seen as a composition of unitaries of the form of (7.11):

Each of these phase gadget unitaries contributes one term to the phase

polynomial:

$$|x_1, x_2, x_3\rangle \xmapsto{U_1} e^{i[\alpha \cdot (x_2 \oplus x_3)]} |x_1, x_2, x_3\rangle$$

$$\xmapsto{U_2} e^{i[\alpha \cdot (x_2 \oplus x_3) + \beta \cdot (x_1 \oplus x_2)]} |x_1, x_2, x_3\rangle$$

$$\xmapsto{U_3} e^{i[\alpha \cdot (x_2 \oplus x_3) + \beta \cdot (x_1 \oplus x_2) + \gamma \cdot (x_2 \oplus x_3)]} |x_1, x_2, x_3\rangle$$

$$\xmapsto{U_4} e^{i[\alpha \cdot (x_2 \oplus x_3) + \beta \cdot (x_1 \oplus x_2) + \gamma \cdot (x_2 \oplus x_3) + \theta \cdot x_2]} |x_1, x_2, x_3\rangle$$

But there's some redundancy here. As we noted in the previous section, α and γ are applied to the same parity of input variables $x_2 \oplus x_3$. By inspecting the phase polynomial, we can see that the unitary can be more compactly represented by just 3 phase gadgets:

$$(7.12)$$
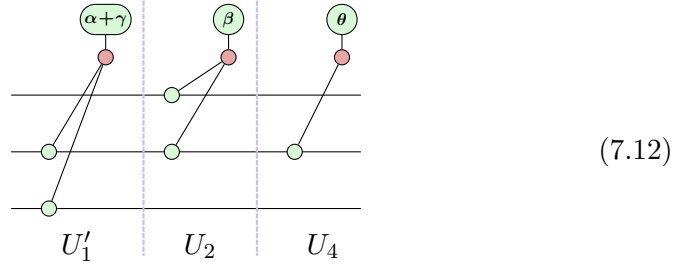
To see this graphically, we would need a ZX rule that lets us "fuse" the phases on phase gadgets connected to an indentical set of wires. We indeed have such a rule, which we call the **gadget fusion** rule:

$$(\boldsymbol{gf})$$
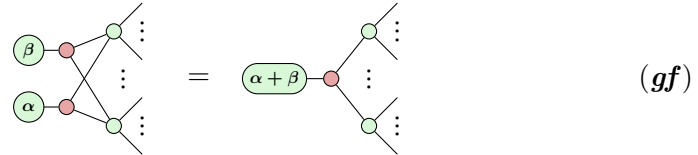
It is in fact a derived rule, which follows from the usual spider fusion rule and strong complementarity:
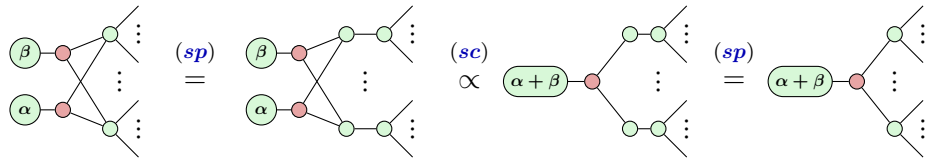
Since the α and γ phase gadgets connect to the exact same spiders in (7.9), we can apply ($\boldsymbol{gf}$) directly to this diagram to simplify it:
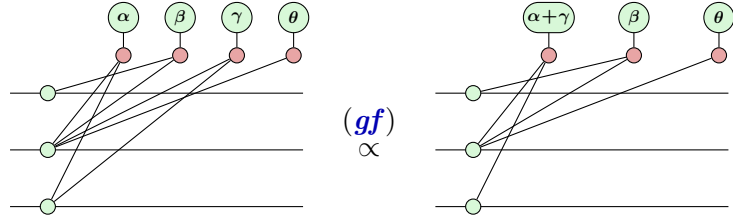
Now, by unfusing spiders, we can see exactly the decomposition into phase gadget unitaries from (7.12). We can also directly read the phase polynomial from the diagram:

$$\rightsquigarrow \quad \underbrace{(\alpha + \gamma) \cdot (x_2 \oplus x_3)}_{g_1} + \underbrace{\beta \cdot (x_1 \oplus x_2)}_{g_2} + \underbrace{\theta \cdot x_2}_{g_3}$$

We computed the phase gadget form (7.9) by appealing to the simplification strategy for CNOT circuits from Chapter 4. However, there is a more direct way we can translate CNOT+phase circuits into circuits of phase gadgets, by studying the way that phase gadgets commute with CNOT gates. There are a few cases to think about here. First, is the trivial case where a phase gadget and CNOT gate share no qubits. Obviously these commute. Only slightly harder to see is if a phase gadget appears on the control qubit (i.e. the Z spider) of a CNOT gate. Then, commutation follows from spider fusion:
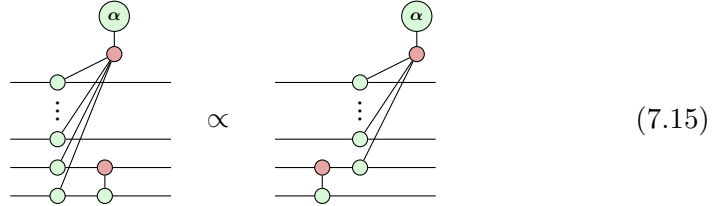
$$(7.13)$$

If the phase gadget overlaps with the target qubit (i.e. the X spider) of a CNOT gate, we can still push a phase gadget through, but it picks up an extra leg:
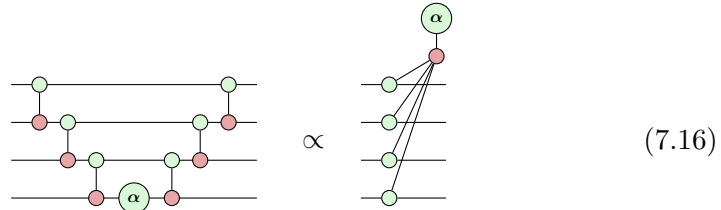
$$(7.14)$$

Just by reading the equation above in reverse, we can also see what happens when a phase gadget overlaps with both qubits of the CNOT gate. It loses

a leg:

$$\propto \tag{7.15}$$

Exercise 7.2. Prove equations (7.14) and (7.15). Use the three "phase-gadget walking" equations (7.13), (7.14), and (7.15), as well as the fact that Z-phase gates are equivalent to 1-legged phase gadgets, to show that an n-legged phase gadget has the following decomposition:

$$\propto \tag{7.16}$$

7.1.3 Universal circuits with path sums
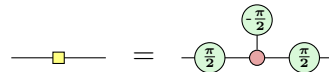
In the previous section, we saw that CNOT+phase circuits can be expressed compactly as phase polynomials, or equivalently using ZX diagrams with phase gadgets. Both of these gates send Z basis states to Z basis states (up to a phase), so they can't possibly be universal. In order to get a universal family of circuits, we need to include a gate which can produce superpositions of Z basis states, like the Hadamard gate.

Unfortunately, Hadamard gates break our nice phase polynomial representation. However, we can salvage things somewhat if we are willing to introduce some extra variables that don't appear in the input state.

Let's start by re-visiting the Euler decomposition of a Hadamard gate.

Back in Section 3.2.5, we saw that there are in fact equivalent ways to write the Hadamard. One way is to unfuse the middle $\pi/2$ from the X spider and change its colour:

This gives us a phase gadget connecting the input and the output spiders. However, unlike the phase gadgets we've met so far, this one is not appearing

as part of a diagonal unitary gate like the one depicted in (7.11). That is, it doesn't seem to be connected to different qubit wires, which can then be labelled by the variables in our phase polynomials. We can fix this by unfusing some spiders:
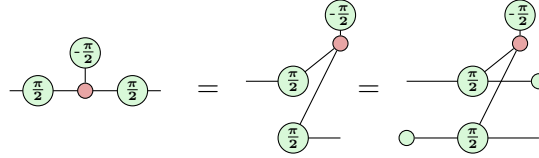
The result is a representation of the Hadamard gate where we first prepare an ancilla in the $|+\rangle$ state, then do a diagonal 2-qubit unitary, and then post-select the input qubit onto $\langle+|$.

We already know how to write the middle part in terms of a phase polynomial:

$$:: \ |xy\rangle \mapsto \tfrac{1}{\sqrt{2}} e^{i[\frac{\pi}{2}\cdot x + \frac{\pi}{2}\cdot y - \frac{\pi}{2}\cdot(x\oplus y)]}|xy\rangle$$
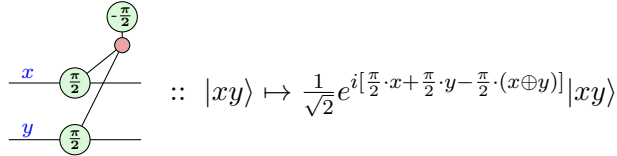
Note that we didn't supress the $\frac{1}{\sqrt{2}}$ factor coming from the phase gadget (as we did before), so technically this is only unitary up to a scalar. This will help us get the Hadamard gate on-the-nose.

The Z-spider plugged into the input means we should sum over y:

$$\circ\!\!-\!\! \ = \ \sum_y |y\rangle$$

whereas the Z-spider plugging into the output means that we should send the computational basis state $|x\rangle$ to 1 (i.e. delete it), regardless of whether $x = 0$ or $x = 1$:

$$-\!\!\circ \ :: \ |x\rangle \mapsto 1$$

Putting these pieces together, we see that we can write the action of a Hadamard gate as something much like the phase polynomial form from before, but now with an extra variable y getting summed over:

$$:: \ |x\rangle \mapsto \tfrac{1}{\sqrt{2}} \sum_y e^{i\pi[\frac{1}{2}\cdot x + \frac{1}{2}\cdot y - \frac{1}{2}\cdot(x\oplus y)]}|y\rangle \qquad (7.17)$$

The extra variable y is called a **path variable** and the overall expression is called a *sum-over-paths* or **path sum** representation of the unitary.

The expression we got in (7.17) may not look much like the definition of a Hadamard gate as we know it, so let's check that this indeed gives us the right thing. First, recall that a Hadamard acts like this on computational basis states:

$$\begin{cases} |0\rangle \mapsto \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \\ |1\rangle \mapsto \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) \end{cases}$$

We can summarise this as something that looks like a path sum as follows:

$$:: \; |x\rangle \mapsto \frac{1}{\sqrt{2}}\sum_{y} e^{i\pi \cdot xy}|y\rangle \tag{7.18}$$

That is, we pick up a $-1 = e^{i\pi}$ coefficient when the input variable $x = 1$ AND the summed variable $y = 1$. However, we don't (yet) know how to introduce the AND of two variables into a phase polynomial. We only know how to introduce XORs using phase gadgets.

Thankfully, for **pseudo-boolean functions**, i.e. functions from bitstrings to the real numbers, these two logical operations are related by the following equation:

$$xy = \tfrac{1}{2}\cdot x + \tfrac{1}{2}\cdot y + \tfrac{1}{2}\cdot(x \oplus y) \tag{7.19}$$

This equation is easy to check just by trying all of values of $x, y \in \{0, 1\}$. There is a deeper reason why this (and many related) equations hold, related to the **Fourier transform** of a pseudo-boolean function. We'll see more about that in Chapter 9. For now, we can see that this is exactly what we need to show that the two expressions for the Hadamard gate, equations (7.17) and (7.18), agree.

This trick of replacing a Hadamard gate with phase gadgets will work for any number of Hadamard gates in a circuit. Hence, for any circuit written in the Clifford+phase gate set, we can transform it into a phase polynomial circuit, at the cost of introducing some ancillae and post-selections:

$$\tag{7.20}$$

The procedure for computing the path sum expression off of a circuit with CNOT, phase, and Hadamard gates can therefore be derived from the simpler one used for CNOT+phase circuits. We start labelling wires with parities as before, but as soon as a Hadamard gate is encountered, we replace

the label with a new, fresh path variable. We then add 3 new terms to the phase polynomial and sum over the path variable.

Example 7.1.1. Doing this procedure with the circuit of Eq. (7.20) we see that we get the path sum

$$|x, y, z\rangle \mapsto \sum_{h_1, h_2} e^{i(\alpha x + \beta h_1 + \gamma h_1 \oplus z + \phi_H(x \oplus y, h_1) + \phi_H(h_1, h_2))} |x, h_2, h_1 \oplus z\rangle$$

where $\phi_H(a, b) = \frac{\pi}{2}(a + b - a \oplus b)$ is the phase polynomial that is added by a Hadamard.

Remark 7.1.2. The physicists among you might be thinking: "hmm, a sum over paths you say, that sounds familiar." And indeed they would be right. The path sums we are using here are exactly a discretised version of the Feynman path integrals that feature so prominently in quantum mechanics. Just like how we can view a quantum particle as evolving over a superposition of paths that interfere together, so we can view a quantum computation as evolving a computational basis state $|\vec{x}\rangle$ through a superposition of paths towards the final outcome.

Exercise 7.3. We know of course that two Hadamards in a row form the identity.

a) Write down the path sum of a single-qubit circuit containing two Hadamards.
b) Figure out how you could prove from this expression that this is equal to the identity.

7.2 Circuit synthesis and path sums

In the previous section we've seen how to convert a circuit into a phase polynomial or path sum expression. Now we will see how we can go back and get a circuit back out again.

First of all, why would we want to do this? The obvious first application for this is circuit optimisation. Namely, we can start with a circuit, compute its path sum expression, then re-synthesise a (hopefully smaller!) circuit that implements the same unitary. We have already met this simplify-and-extract methodology with CNOT circuits in Section 4.2, and we'll meet it again a couple more times in this book. We saw an instance of this already in passing from circuit (7.4) to the smaller circuit (7.6). This was a simple example, so we were able to find a smaller circuit implementing the same phase polynomial in an *ad hoc* way before. In Section 7.2.1 we will explore some algorithms for this which work for any phase polynomial.

For general path sum expressions there is no known efficient algorithm for turning it back in a circuit. We will however in Section 7.2.2 consider a specific example where we can make it work. This actually brings us to the second application for constructing circuits from these expressions: it allows you to build *new* circuits from a higher-level description of the circuit behaviour. In Chapter 9, we'll look at this in more detail and see how, thanks to a bit of boolean Fourier theory, we can build circuits in the Clifford+phase gate set implementing arbitrary classical functions on quantum data using phase polynomials and the circuit synthesis techniques from this section.

7.2.1   Synthesis from phase polynomials

Just like how we were able to commandeer the phase-free simplification algorithm from Chapter 4 to simplify CNOT+phase circuits, we can (essentially) repurpose the phase-free extraction algorithm for the parity form of a ZX-diagram to extract CNOT+phase circuits from the phase gadget form.

Recall that a ZX-diagram in parity form looks like a row of Z-spiders connected to a row of X-spiders, e.g.
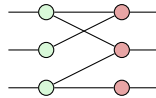
From this form, we can associate a biadjacency matrix:

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

and performing Gauss-Jordan reduction of this matrix gives us a CNOT decomposition, where each CNOT operations corresponds to a primitive column operation:
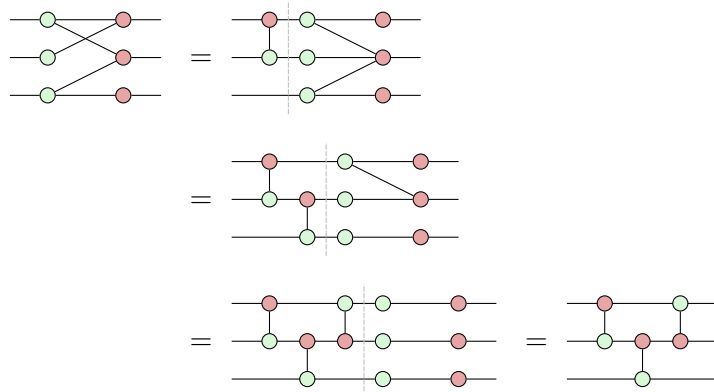
Now, suppose we generalise from the parity form to the phase gadget form, i.e. we add some phase gadgets connected to the input Z-spiders:

$$(7.21)$$

This describes a unitary with phase polynomial expression:

$$U :: |x_1 x_2 x_3\rangle \mapsto e^{i\alpha \cdot (x_1 \oplus x_2) + \beta \cdot (x_2 \oplus x_3)} |x_1 \oplus x_2, x_1 \oplus x_3, x_3\rangle \qquad (7.22)$$

The first thing we realise from inspecting (7.21) is that most of the diagram is already in parity form, except for the two phases themselves:
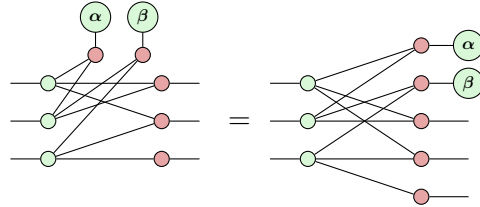
Writing down the biadjacency matrix, we see that we now get a tall, skinny matrix, with some extra rows at the beginning corresponding to phase gadgets:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ \hline 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad (7.23)$$

Just like before, precomposing with CNOTs performs primitive column operations. For example, this:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ \hline 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{c_2 := c_2 + c_1} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ \hline 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

corresponds to this equation of ZX-diagrams:
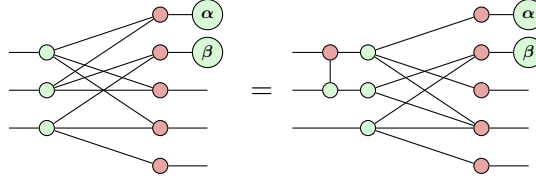
But note that the α phase is now attached to a 1-legged phase gadget, i.e. an identity X-spider. So, we can use spider fusion to pull it out:
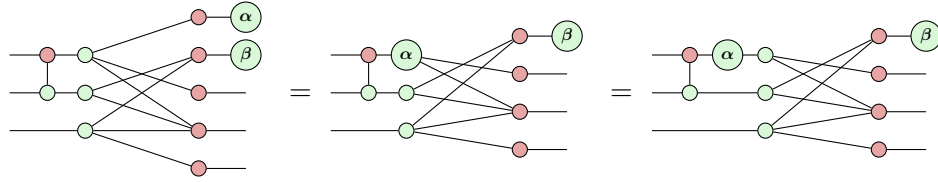
The reason we could do that, is because our primitive column operation turned one of the rows above the line in matrix (7.23) into a unit vector. Since this means that phase gadget is now 1-legged, we can always pull it out to become a Z phase gate. The remaining bit of the diagram in phase gadget form then has a biadjancency matrix obtained by deleting the first row:

$$
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 1 \\
\hline
1 & 0 & 0 \\
1 & 1 & 1 \\
0 & 0 & 1
\end{pmatrix}
\xrightarrow{\text{del}(r_1)}
\begin{pmatrix}
0 & 1 & 1 \\
\hline
1 & 0 & 0 \\
1 & 1 & 1 \\
0 & 0 & 1
\end{pmatrix}
$$

We can now create a unit vector in the first row by adding the second column to the third column:

$$
\begin{pmatrix}
0 & 1 & 1 \\
\hline
1 & 0 & 0 \\
1 & 1 & 1 \\
0 & 0 & 1
\end{pmatrix}
\xrightarrow{c_3 := c_3 + c_2}
\begin{pmatrix}
0 & 1 & 0 \\
\hline
1 & 0 & 0 \\
1 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
$$

Again, the phase gadget turns into a simple Z phase gate, which we can pull

out to the left:
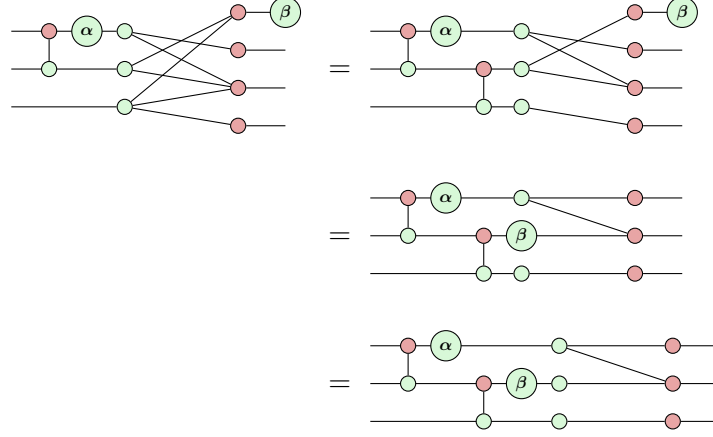
Now that we have no phase gadgets left, there is nothing above the line in our biadjacency matrix, so we finish off by doing normal Gaussian elimination:

$$\left(\begin{array}{ccc} \overline{0} & \overline{1} & \overline{0} \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right) \xrightarrow{\text{del}(r_1)} \left(\begin{array}{ccc} \overline{1} & \overline{0} & \overline{0} \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right) \xrightarrow{c_1 := c_1 + c_2} \left(\begin{array}{ccc} \overline{1} & \overline{0} & \overline{0} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right)$$

This then yields as a final circuit:
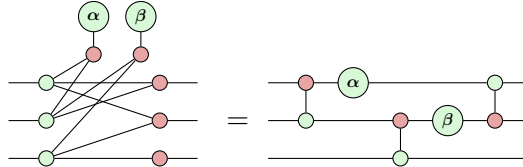
We can check our work by labelling wires with parities:
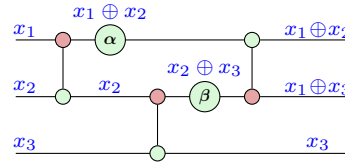
Comparing this labelled circuit to the phase polynomial expression in equation (7.22), we see that we have indeed synthesised a correct circuit for this unitary.

Hence, the general algorithm for synthesising a circuit from a phase polynomial expression for a unitary U is the following:

1. Form a block diagonal matrix over $\mathbb{F}_2$ whose columns are labelled by input variables $x_1, \ldots, x_n$, where:

   - the first k rows correspond to the k terms in the phase polynomial, with a 1 in column i iff x_i appears in that term, and
   - the last n rows correspond to the parity map describing U's action on basis states.

2. Perform primitive column operations (i.e. CNOT gates) until one of the first k rows is reduced to a unit vector with a 1 in column j

3. Apply a phase gate to the j-th qubit and delete the unit vector.

4. Repeat from step 2 until the top block of the parity matrix is empty.

5. Synthesise a CNOT circuit corresponding to the remaining n rows.

Note that step 2 will always succeed: we can reduce any non-zero row to a unit vector using primitive column operations, and since steps 2-4 always remove a row, the algorithm will terminate. However, just as we saw in the CNOT case, the size of the resulting circuit is very sensitive to which column operations are performed and in which order. In fact, here we have two kinds of decisions to make: which rows to eliminate first, and which column operations should be used to eliminate a given row?

The answer to these questions will depend on what our goals are. Namely, do we want to choose operations to minimise the depth of the circuit or just the CNOT count? Or maybe it could be beneficial on our architecture to do as many phase gates in parallel as possible.

For the remainder of this section, we will discuss some heuristics for achieving these goals.

It may also be the case that we cannot apply CNOT gates between arbitrary pairs of qubits, but only nearest neighbours in our architecture. We discuss how to deal with this in the References of this chapter.

7.2.2 Quantum Fourier transform

While it is possible to efficiently extract a circuit from a phase polynomial description, doing the same from a path-sum description is still an open problem. For certain special cases we can try to make it work just by using raw brainpower.

In this section we will find a circuit for the **Quantum Fourier transform** based on its description as a path sum. Recall that this unitary is for instance the magic trick that makes Shor's algorithm work. In general, for a d-dimensional space $\mathbb{C}^d$ we define its Fourier transform as follows. Let

$|0\rangle, \ldots, |d-1\rangle$ be the standard basis states. Then

$$QFT \; :: \; |x\rangle \mapsto \frac{1}{\sqrt{d}} \sum_{y=0}^{d-1} e^{\frac{2\pi i}{d}xy}|y\rangle.$$

For instance, when $d = 2$ we get $|x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_y e^{i\pi xy}|y\rangle$, which is exactly the Hadamard. We will be interested in the case where $d = 2^n$, so that the space corresponds to n qubits. We should read the expression xy then as multiplying the numbers $x, y \in \{0, \ldots, 2^n - 1\}$, *not* as taking the inner product of two bit strings. But since we do like thinking about bit strings, let's define a translation. For a bit string $\vec{x} \in \mathbb{F}_2^n$, define $b(\vec{x}) := 2^{n-1}x_1 + 2^{n-2}x_2 + \cdots + 2^0 x_n$. Then we can write the n-qubit QFT as follows:

$$|\vec{x}\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{\vec{y} \in \mathbb{F}_2^n} e^{\frac{2\pi i}{2^n}b(\vec{x})b(\vec{y})}|\vec{y}\rangle. \tag{7.24}$$

Exercise 7.4. In this exercise we will use the path-sum formalism to find a circuit implementing the n-qubit QFT.

1. To compile the QFT unitary to a quantum circuit, we will need to know first how to compile *controlled phase* gates. These are unitaries that act like $|x_1, x_2\rangle \mapsto e^{i\alpha x_1 \cdot x_2}|x_1, x_2\rangle$. Using phase polynomials and the identity $x \cdot y = \frac{1}{2}(x + y - x \oplus y)$ show that the following circuit implements a controlled phase gate:

 $\tag{7.25}$

2. For $n = 2$, expand the definition of b in Eq. (7.24) to write the phase polynomial as a product of $e^{i\alpha x_k y_l}$ terms. Argue that the $x_1 y_1$ term can be removed without changing the resulting map.

3. Using the path-sum approach, show that the following circuit implements the 2-qubit QFT, by verifying that it is equal to the expression you derived above:

 $\tag{7.26}$

   Note that you might need to 'rename' the variables y you are getting out of an application of a Hadamard gate to make sure it matches the expression you had above.

4. Now for $n = 3$, again expand the definition of b in Eq. (7.24) to write the phase polynomial as a product of $e^{i\alpha x_k y_l}$ terms and remove the terms that you can remove. Construct a circuit to implement the 3-qubit QFT.

5. Give a recipe for constructing the n-qubit QFT for any n.

7.3 Pauli exponentials

Now we will switch to a very different way of looking at universal quantum circuits. Not as a series of logic gates that create branches of computational basis states and evolve these with phase polynomials, but as a series of applications of rotations around Pauli gates.

7.3.1 Unitaries from Pauli boxes

In Chapter 6, we introduced **Pauli boxes**, which gave us a way to wrap up the pair of projectors coming from a Pauli measurement into a single object. We also saw in Exercise 6.8 that these are special case of **measure boxes**, which can represent an arbitrary (2-outcome) measurement using a single map satisfying the following properties:

$$-\boxed{M}-\hspace-3pt\circ \;=\; \text{——} \qquad -\boxed{M}\;\boxed{M}\;=\;-\boxed{M}\hspace-2pt\circ\hspace-3pt< \qquad \left(-\boxed{M}-\right)^{\dagger}=\; \overset{\frown}{-\boxed{M}}$$

These three conditions are equivalent to the condition that the following forms a measurement:

$$\mathcal{M} := \left\{ -\boxed{M}\hspace-3pt\fbox{$k\pi$} \right\}_k$$

So, we get a measurement when plugging in X spiders into the control wire. Here's a (seemingly) random question: what happens if we plug in Z spiders? To find out, lets see what happens when we form a linear map L as follows:

$$L \;:=\; -\boxed{M}\hspace-3pt\bigcirc\hspace-3pt{\alpha}$$

Thanks to the third property of measure boxes, we have:

$$L^{\dagger} \;=\; \overset{-\alpha}{}\boxed{M^{\dagger}}- \;=\; \overset{-\alpha}{-\boxed{M}} \;=\; -\boxed{M}-\hspace-3pt{-\alpha}$$

Using the other two properties:

$$L^{\dagger}L \;=\; -\boxed{M}\hspace-2pt{\alpha}\;-\boxed{M}\hspace-2pt{-\alpha} \;=\; -\boxed{M}\hspace-3pt\overset{\alpha}{<}_{-\alpha} \;=\; -\boxed{M}-\circ \;=\; \text{——} \;=\; I$$

and similarly, $LL^{\dagger} = I$. So L is a unitary!

Proposition 7.3.1. For any self-adjoint Pauli string $\vec{P}$, the following map:
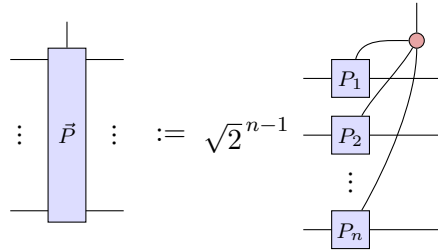
$$\tag{7.27}$$
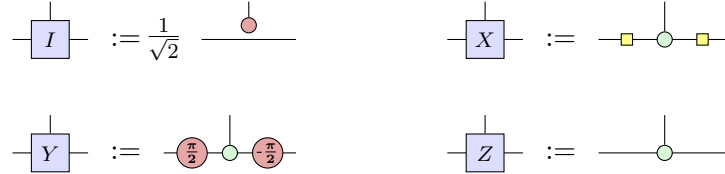
is a unitary.

Proof   Follows immediately from the fact that Pauli boxes are measurement boxes, as shown in Exercise 6.8. $\square$
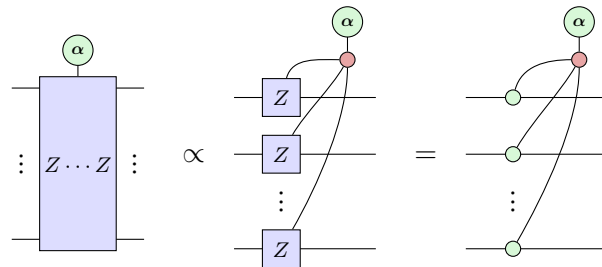
Let's expand out what is in the Pauli box to see what these unitaries from Proposition 7.3.1 actually look like. Recall from Definition 6.2.3 that

$$\vdots \;\; \vec{P} \;\; \vdots \;\; := \;\; \sqrt{2}^{\,n-1}$$

where

$$\boxed{I} := \frac{1}{\sqrt{2}} \quad\quad\quad \boxed{X} :=$$

$$\boxed{Y} := \quad\quad\quad\quad \boxed{Z} :=$$

Hence, for the special case of $\vec{P} = Z \otimes \cdots \otimes Z$ we get:

$$\vdots \;\; \boxed{Z \cdots Z} \;\; \vdots \;\; \propto \quad\quad = \quad\quad \tag{7.28}$$

We get phase gadgets! For a general unitary built from Pauli boxes, we can adapt Eq. (6.7): for any Pauli string $\vec{P} = P_1 \otimes \ldots \otimes P_n$ with $P_i \in \{X, Y, Z\}$,

we have:

$$(7.29)$$

The general case can be obtained by additionally representing any $P_i = I$ by simply not connecting to the i-th qubit.

The unitaries built from Pauli boxes in this way hence generalise phase gadgets: they are just phase gadgets surrounded by some particular single-qubit Cliffords. We will call such unitaries **Pauli exponentials**. As the name suggests, we can view these unitaries as special cases of matrix exponentials, which we'll introduce in the next section.

7.3.2 Matrix exponentials

We first mentioned matrix exponentials back in Section 2.2.3 as a means of constructing solutions to the Schrödinger equation, but avoided giving a formal definition. We'll do that now.

At this point, we should be pretty used to raising e to the power of a complex number z. For instance, when $z = i\alpha$ is imaginary, this is how to obtain phases. If we crack open a textbook on complex analysis, we'll see that e^z for any complex number is defined as the following power series:

$$e^z := \sum_{k=0}^{\infty} \frac{z^k}{k!}$$

Swapping out the complex number z for a square matrix A gives us a definition for the **matrix exponential**:

$$e^A := \sum_{k=0}^{\infty} \frac{A^k}{k!}. \tag{7.30}$$

Here the powers A^k are just multiplying A with itself k times, and in particular $A^0 = I$ is the identity matrix.

There are a few things we should know about matrix exponentials. The first is that if we are working with a diagonal matrix, then its matrix exponential

is just the exponential of each of its components:

$$e^{\text{diag}(a_1,\ldots,a_n)} = \text{diag}(e^{a_1},\ldots,e^{a_n}). \tag{7.31}$$

This works because, when we raise a diagonal matrix to a power k, it just goes inside:

$$e^{\text{diag}(a_1,\ldots,a_n)} = \sum_{k=0}^{\infty} \frac{\text{diag}(a_1,\ldots,a_n)^k}{k!} = \sum_{k=0}^{\infty} \frac{\text{diag}(a_1^k,\ldots,a_n^k)}{k!}$$
$$= \text{diag}\left(\sum_k \frac{a_1^k}{k!},\ldots,\sum_k \frac{a_n^k}{k!}\right)$$
$$= \text{diag}(e^{a_1},\ldots,e^{a_n})$$

The second thing we need to know is that conjugation by a unitary commutes with exponentiation:

$$e^{UAU^\dagger} = \sum_{k=0}^{\infty} \frac{(UAU^\dagger)^k}{k!} = \sum_{k=0}^{\infty} \frac{UA^kU^\dagger}{k!} = Ue^AU^\dagger. \tag{7.32}$$

With these facts we can easily compute the matrix exponential of any diagonalisable matrix. Let A be a matrix with diagonalisation $A = \sum_j \lambda_j |\phi_j\rangle\langle\phi_j|$ for some ONB $\{|\phi_j\rangle\}$. Write $U = \sum_j |\phi_j\rangle\langle j|$ for the unitary that maps the computational basis $\{|j\rangle\}$ into the eigenbasis $\{|\phi_j\rangle\}$ of A, and write $D = \sum_j \lambda_j |j\rangle\langle j| = \text{diag}(\lambda_1,\ldots,\lambda_n)$ for the diagonal matrix of eigenvalues of A. Then we have $A = UDU^\dagger$. Hence, we calculate $e^A = e^{UDU^\dagger} = Ue^DU^\dagger$. As e^D is the exponential of a diagonal matrix, it is easily calculated and so we get an easy expression for e^A, simply by multiplying out the matrices of its diagonalisation.

There is one final property of the matrix exponential we will be interested in. Remember that for regular complex numbers z_1 and z_2 we have $e^{z_1+z_2} = e^{z_1}e^{z_2}$. The same holds for the matrix exponential *as long as* the matrices commute. That is, if we have matrices A and B that commute, then $e^{A+B} = e^Ae^B$. When A and B allow diagonalisations, this is easily proven, as we can then find a joint diagonalisation, and reduce the problem to diagonal matrices.

Proposition 7.3.2. Let A and B be commuting square matrices. Then $e^{A+B} = e^Ae^B = e^Be^A$. In particular $I = e^0 = e^{A-A} = e^Ae^{-A}$ so that $(e^A)^{-1} = e^{-A}$ for any matrix A.

When A and B do not commute, we generally have $e^{A+B} \neq e^Ae^B$. In fact, studying ways to approximate the value of e^{A+B} using just e^A and e^B is incredibly important for simulating quantum mechanical systems using a quantum computer. We'll study this problem in more detail in Section 7.5.

7.3.3 Building unitaries as exponentials

With this mathematical machinery of matrix exponentials in hand, let's look at how it can help us understand unitary maps. Suppose we have some n-dimensional unitary U. As discussed in Section 2.1.3, a unitary is a normal linear map and hence can be diagonalised: $U = \sum_j \lambda_j |\phi_j\rangle\langle\phi_j|$. Using the description above we can then also write this as $U = VDV^\dagger$ where V is some unitary and D is some diagonal matrix consisting of the eigenvalues of U. Now remember that all the eigenvalues of a unitary are phases so that $\lambda_j = e^{i\alpha_j}$ for some phases α_j. This means that D is a matrix of exponentials. In the previous section we saw that the exponential of a diagonal matrix is a matrix of exponentials. The converse is however also true. We can hence write $D = e^{iD'}$ where $D' = \text{diag}(\alpha_1, \ldots, \alpha_n)$. We have here taken out the factor of i out of the matrix for reasons that will become clear soon enough. So now we have $U = Ve^{iD'}V^\dagger$. We can take this conjugation by V inside of the matrix exponential to get $U = e^{iH}$ where $H = VD'V^\dagger$. Now the eigenvalues of H are precisely the values in D', which are real numbers. As we saw in Exercise 2.4 a matrix whose eigenvalues are real numbers is self-adjoint. We hence have proven the following result.

Proposition 7.3.3. Any unitary U is the complex exponential of some self-adjoint matrix H via $U = e^{iH}$.

Note that H is not unique: we can add $2\pi I$ to it and still get the same result: $e^{i(H+2\pi I)} = e^{iH}e^{i2\pi I} = e^{iH}$. Here in the last step we used $e^{2i\pi} = 1$. In fact, adding any multiple of the identity to H just changes the global phase of U, and so we don't care about it.

The converse of Proposition 7.3.3 is also true: if we take some self-adjoint matrix H, then e^{iH} is unitary (This follows from Exercise 2.4, as the eigenvalues of e^{iH} will all be complex phases). This gives us a way to construct unitaries. In fact, given a self-adjoint H, it will be useful to consider the entire family of unitaries constructed by rescaling H: for any $t \in \mathbb{R}$ we have that e^{itH} is unitary. We use the letter t here, as this gives the elapsed time in the Schrödinger equation.

Example 7.3.4. Let $Z = \text{diag}(1, -1)$ be the standard Pauli Z. Then for any $t \in \mathbb{R}$ we calculate $e^{itZ} = \text{diag}(e^{it}, e^{-it}) = e^{it}\,\text{diag}(1, e^{-2it})$. So we see that

$$e^{itZ} \propto Z(-2t). \tag{7.33}$$

In particular, the S and T gates can be written (up to global phase) as

$S \propto e^{-i\frac{\pi}{4}Z}$ and $T \propto e^{-i\frac{\pi}{8}Z}$. It is for this reason that in some works the T gate is also called the $\frac{\pi}{8}$ phase gate.

Example 7.3.5. For the Pauli X gate we can easily calculate its matrix exponential by realising that $X = HZH^\dagger$ (of course $H^\dagger = H$, but it will be useful to write it like this) and using Eq. (7.32): $e^{itX} = e^{itHZH^\dagger} = He^{itZ}H^\dagger \propto HZ(-2t)H^\dagger = X(-2t)$, where we used Eq. (7.33) to convert the Z exponential into a Z phase gate. Hence, the exponentiated X matrices correspond to X phase gates. It is interesting however to see how you would calculate this directly using the definition of the matrix exponential (7.30). The trick is to observe that $X^2 = I$, which allows us to split the infinite sum into two parts:

$$
\begin{aligned}
e^{itX} &\overset{(7.30)}{=} \sum_{k=0}^{\infty} \frac{(itX)^k}{k!} = \sum_{n=0}^{\infty} \frac{i^{2n}t^{2n}X^{2n}}{(2n)!} + \sum_{n=0}^{\infty} \frac{i^{2n+1}t^{2n+1}X^{2n+1}}{(2n+1)!} \\
&= \sum_{n=0}^{\infty} (-1)^n \frac{t^{2n}}{(2n)!} I + i \sum_{n=0}^{\infty} (-1)^n \frac{t^{2n+1}}{(2n+1)!} X \\
&= \cos(t)I + i\sin(t)X
\end{aligned}
$$

Here in the last step we used the standard Taylor series equality for sin and cos.

In this direct calculation of the matrix exponential of the Pauli X we actually only used that $X^2 = I$, and nothing else, so this calculation works for any matrix with this property. As this is actually quite useful, let's record this for later.

Proposition 7.3.6. Let M be any matrix for which $M^2 = I$. Then $e^{itM} = \cos(t)I + i\sin(t)M$.

7.3.4 Pauli exponentials

Now that we can understand any unitary as a matrix exponential, let's zoom in a bit and take a look at Pauli exponentials. The Examples 7.3.4 and 7.30 show that we can view the Z and X phase gates as exponentials of Pauli matrices. It turns out that this holds for essentially *all* unitaries if we do the same with multi-qubit Pauli strings. A Pauli string $\vec{P}$ is just a tensor product of single-qubit Paulis $\vec{P} = e^{i\alpha}P_1 \otimes \cdots \otimes P_n$ where $P_i \in \{I, X, Y, Z\}$ and $\alpha \in \mathbb{R}$ is some global phase. Since we want to take exponentials of these to construct unitaries, we want $\vec{P}$ to be self-adjoint, which requires $e^{i\alpha} \in \{1, -1\}$.

Definition 7.3.7. A **Pauli exponential** is any unitary of the form:

$$e^{\pm i\theta P_1 \otimes \cdots P_n}$$

where the $P_j \in \{I, X, Y, Z\}$ are Paulis and $\theta \in \mathbb{R}$ is a phase.

Here we write θ instead of t, because we will be thinking of this as an angle in a rotation, instead of an elapsed time.

Using Proposition 7.3.6 we calculate:

$$e^{\pm i\theta P_1 \otimes \cdots P_n} = \cos(\theta)I \otimes \cdots \otimes I \pm i\sin(\theta)P_1 \otimes \cdots P_n. \qquad (7.34)$$

Using this, we see that an identity in a Pauli string makes the Pauli exponential act as an identity on that qubit:

$$e^{\pm i\theta I \otimes P} = \cos(\theta)I \otimes I \pm i\sin(\theta)I \otimes P = I \otimes (\cos(\theta)I \pm i\sin(\theta)P) = I \otimes e^{i\theta P}. \tag{7.35}$$

We can use this to calculate a circuit that implements any Pauli exponential using gates we have already encountered. Before we do that in full generality it will be useful to consider a specific example.

Example 7.3.8. Let's calculate what unitary $e^{i\theta Z \otimes Z}$ is. First, let's do it directly. Note that $Z \otimes Z = \mathrm{diag}(1, -1, -1, 1)$. Then using Proposition 7.3.6 we get

$$
\begin{aligned}
e^{i\theta Z \otimes Z} &= \cos(\theta)I \otimes I + i\sin(\theta)Z \otimes Z \\
&= \cos(\theta)\,\mathrm{diag}(1,1,1,1) + i\sin(\theta)\,\mathrm{diag}(1,-1,-1,1) \\
&= \mathrm{diag}(e^{i\theta}, e^{-i\theta}, e^{-i\theta}, e^{i\theta}).
\end{aligned}
$$

Here in the last step we used $e^{\pm i\theta} = \cos(\theta) \pm i\sin(\theta)$.

Now, let's calculate this in a more systematic way. Note that $Z \otimes Z = \mathrm{CNOT}(I \otimes Z)\mathrm{CNOT}^\dagger$ (see Section 6.1.1). Hence:

$$e^{\pm i\theta Z \otimes Z} = e^{\pm i\theta\,\mathrm{CNOT}(I \otimes Z)\mathrm{CNOT}^\dagger} \overset{(7.32)}{=} \mathrm{CNOT}\, e^{\pm i\theta I \otimes Z}\, \mathrm{CNOT}^\dagger$$

$$\overset{(7.35)}{=} \mathrm{CNOT}(I \otimes e^{\pm i\theta Z})\mathrm{CNOT}^\dagger \overset{(7.33)}{\propto} \mathrm{CNOT}(I \otimes Z(\mp 2\theta))\mathrm{CNOT}^\dagger$$

$$= \mathrm{diag}(1, e^{\mp 2\theta}e^{\mp 2\theta}, 1).$$

This last matrix is indeed equal to what we got before, up to global phase.

The expression we got in this example, $\mathrm{CNOT}(I \otimes Z(\mp 2\theta))\mathrm{CNOT}$ consists only of gates we have already encountered, and that we can in particular write as a ZX-diagram:

$$e^{\pm i\theta Z \otimes Z} \quad \propto \quad$$

It is also a phase gadget! This in fact remains true if we add more qubits, by realising that a CNOT ladder sends $Z \otimes \cdots \otimes Z$ to Z_n:

$$(7.36)$$

By Exercise 7.2 such CNOT ladders with a phase in the middle are phase gadgets. As the unitaries we constructed from Pauli boxes *also* were phase gadgets, we see that we can now relate them:

$$(7.37)$$

For a more general Pauli string $\vec{P} = P_1 \otimes \ldots \otimes P_n$ with $P_i \in \{X, Y, Z\}$ we can use the fact that the U_i from Eq. (7.29) satisfy $P_i = U_i Z U_i^\dagger$ to then write any Pauli exponential as a unitary from a Pauli box: setting $U = U_1 \otimes \cdots \otimes U_n$ we have $\vec{P} = U Z \cdots Z U^\dagger$, and hence

Hence:

$$(7.38)$$

It will be helpful to introduce some shorthand for $e^{i\theta \vec{P}}$. For Z and X phase gates we have been writing $Z(\alpha)$ for a phase rotation over α. We have also seen that $e^{i\theta Z} \propto Z(-2\theta)$, or in other words that $Z(\alpha) \propto e^{-i\alpha/2 Z}$. This suggests the following notation for any Pauli string $\vec{P}$:

$$\vec{P}(\alpha) := e^{i\frac{\alpha}{2}} e^{-i\frac{\alpha}{2}\vec{P}}. \tag{7.39}$$

Here we have included a global phase for good measure, so that $Z(\alpha)$ and

$X(\alpha)$ match exactly to the definition of those phase gates that we were using before. For a multi-qubit Pauli string we will again adopt the shorthand of not writing the tensor product symbol $\otimes$. So we will for instance write $ZZ(\alpha)$ for the unitary $e^{i\alpha/2}e^{-i\alpha/2Z\otimes Z}$.

Note that replacing $\vec{P}$ by $-\vec{P}$ in a Pauli exponential is equivalent to flipping the phase of the exponential: $e^{i\theta(-\vec{P})} = e^{-i\theta\vec{P}}$. Hence, in the definition Eq. (7.39) we get $[(-1)^k\vec{P}](\alpha) \propto \vec{P}((-1)^k\alpha)$.

For the future let's record some useful facts now that we have this correspondence between Pauli exponentials and Pauli boxes. First, we can represent any Pauli exponential using just a small number of standard gates.

Proposition 7.3.9. Let $\vec{P}$ be any n-qubit Pauli string. Then we can represent $e^{i\theta\vec{P}}$ as a circuit of at most $4n-1$ gates of type CNOT, $H, X(\frac{\pi}{2}), X(-\frac{\pi}{2})$, and $Z(-2\theta)$.

Second, we see a Pauli exponential is Clifford iff its phase is Clifford.

Proposition 7.3.10. A Pauli exponential $\vec{P}(\alpha)$ for a non-trivial $\vec{P}$ is Clifford iff $\alpha = k\frac{\pi}{2}$ for some $k \in \mathbb{Z}$.

The condition on 'non-trivial' here is necessary since the trivial Pauli exponential $e^{i\theta I}$ is of course just the identity gate up to global phase for any θ.

7.4 Pauli exponential compilation

Now that we know a bit about Pauli exponentials, let's see how they help us think about quantum circuits and compilation.

7.4.1 Pauli exponentials are a universal gate set

We have already seen that Z and X phase gates can be represented as Pauli exponentials. This means in particular that any single-qubit unitary can be represented as a composition of Pauli exponentials. We know that single-qubit unitaries together with the CNOT gate form a universal gate set, so if we can show that the CNOT gate can be built out of Pauli exponentials, then we know that *all* unitaries are just compositions of Pauli exponentials.

It will in fact be easier to look at the CZ gate. Remember that the CNOT is just a CZ conjugated by Hadamards on the target qubit:

As a Hadamard is just a series of Z and X phase gates, which are Pauli exponentials, it then indeed remains to look at the CZ.

Lemma 7.4.1. The CZ gate is a composition of Pauli exponentials:

$$
\propto \quad \propto \quad Z_1\left(\frac{\pi}{2}\right) Z_2\left(\frac{\pi}{2}\right) ZZ\left(-\frac{\pi}{2}\right)
$$

Proof For the ZX-diagram equality use Eq. (3.84) of Exercise 3.15, spider fusion (**sp**) and then the Euler decomposition (**eu**) of the Hadamard. That this diagram is indeed the sequence of Pauli exponentials follows because a phase gadget is just a ZZ Pauli exponential. □

Theorem 7.4.2. Pauli exponentials form a universal gate set. Or in other words, any n-qubit unitary can be written as a composition of Pauli exponentials.

Proof Z and X phase gates are Pauli exponentials (see Examples 7.3.4 and 7.3.5). These gates generate all single-qubit unitaries. Combining this with the CZ gate gives a universal gate set, and this CZ can also be written as a Pauli exponential by the previous lemma. □

Remark* 7.4.3. There is a more 'abstract nonsense' argument for why you should be able to construct any unitary by composing Pauli exponentials. The n-qubit unitaries form a **Lie group**, a group that is also a differentiable manifold in an appropriate way. When we have a continuous family of unitaries we can take its derivative to get an associated matrix, which in this case will be a self-adjoint one. This is in fact the matrix exponentiation we have been looking at: given a self-adjoint matrix H, we get a family of unitaries via e^{itH}, and taking the derivative of the function $t \mapsto e^{itH}$ gives you back H (or well, actually iH, but this is just the physicist convention; mathematicians please don't get angry at us). All of this is to say that the self-adjoint matrices form the **Lie algebra** of the group of unitaries. Now, there is a result (see the References of the chapter) that says that if we have a basis of the Lie algebra $H_1, \ldots, H_k$, that then the associated families of Lie group elements $e^{itH_1}, \ldots, e^{itH_k}$ *generate* the entire (connected part of the) Lie group, meaning that we can write any Lie group element as a finite combination of elements from these families. It just so happens that the Paulis form a basis of the n-qubit self-adjoint matrices, and hence the Pauli exponentials generate the n-qubit unitaries!

7.4.2 Compiling to Pauli exponentials

Theorem 7.4.2 gives one way to write a quantum circuit as a series of Pauli exponentials, but if we are interested in doing this with a *small* number of these, it doesn't do a very good job. There is in fact a more interesting way to transform a circuit into Pauli exponentials, which also has practical relevance in quantum compilation (see the References for this chapter).

To do this we need to use what we learned in the previous chapter. There we saw that pushing a Clifford unitary U through a Pauli $\vec{P}$ gives you another Pauli: $\vec{P}U = U\vec{Q}$ for some $\vec{Q}$. Now, in Eq. (7.32) we saw that conjugating a matrix exponential is the same thing as exponentiating the conjugated matrix. This can be recast as a way to commute a unitary through a matrix exponential:

$$\overset{(7.32)}{e^A U = UU^\dagger e^A U} \;\; = \;\; Ue^{U^\dagger AU}. \tag{7.40}$$

So if our matrix A is actually a Pauli $\vec{P}$, and our unitary U is a Clifford, then we can push the Clifford through the Pauli exponential to get another Pauli exponential:

$$e^{i\theta\vec{P}}U = Ue^{i\theta U^\dagger \vec{P}U} = Ue^{i\theta\vec{Q}} \tag{7.41}$$

Example 7.4.4. Suppose our Pauli exponential is just $e^{-i\theta Z_2}$ and that we want to push a CNOT through it. In terms of ZX-diagrams this becomes:

How does Eq. (7.41) help us? Well, it helps us if we put on a different pair of glasses and view our circuit not as being given by X and Z phase gates and CZ/CNOT gates, but instead view it as consisting of *Clifford* gates and *non-Clifford* gates. With this perspective we consider CZ, CNOT, Hadamard, S and $X(\frac{\pi}{2})$ gates to all be of the same sort, namely Clifford, while the other class of gates contains any $Z(\alpha)$ and $X(\alpha)$ where α is not a multiple of $\frac{\pi}{2}$. These non-Clifford phase gates we can then view as Pauli exponentials (which are only acting on a single qubit). So wearing these glasses we see the circuit as a series of Clifford gates and Pauli exponentials, where each Pauli exponential corresponds to a non-Clifford phase gate:

$$U = U_1 e^{i\theta_1\vec{P}_1} U_2 e^{i\theta_2\vec{P}_2} \cdots U_k e^{i\theta_k\vec{P}_k} U_{k+1}. \tag{7.42}$$

Here each of the Clifford unitaries U_j can consist of multiple basic Clifford gates like CNOT, S and H.

Now let's pick the first non-Clifford Pauli exponential in this circuit $e^{i\theta_1 \vec{P}^1}$ and push it through U_1 using Eq. (7.41) to get:

$$e^{i\theta_1 \vec{Q}^1} U_1 U_2 e^{i\theta_2 \vec{P}^2} \cdots U_k e^{i\theta_k \vec{P}^k} U_{k+1}.$$

Here $\vec{Q}^1 = U_1^\dagger \vec{P}^1 U$. We then push $e^{i\theta_2 \vec{P}^2}$ through both U_2 and U_1 and so on, until finally we get the circuit:

$$e^{i\theta_1 \vec{Q}^1} e^{i\theta_2 \vec{Q}^2} \cdots e^{i\theta_k \vec{Q}^k} U_1 U_2 \cdots U_{k+1}. \tag{7.43}$$

The circuit now has a very different structure! We see that each non-Clifford phase gate in the original circuit, which we saw as single-qubit Pauli exponentials, is transformed into some other Pauli exponential, which can now act on any number of qubits. All the Clifford gates of the original circuit are still present here, but are now pushed all the way to the end. In Chapter 5 we saw how to optimise Clifford circuits, and in particular that those circuits can be reduced to a normal form whose maximal size only depends on the number of qubits (Proposition 5.3.12).

This structure is useful for a variety of reasons that we'll explore in this chapter, so let's give a name to it.

Definition 7.4.5. We say a circuit is in **Pauli exponential form** when it consists of a series of Pauli exponentials followed by a Clifford circuit.

We'll often shorten this to PE form or say it is a PE circuit.

Proposition 7.4.6. A circuit consisting of Clifford gates and k non-Clifford phase gates can be efficiently transformed into a Pauli exponential form containing k Pauli exponentials.

As a first result, this form gives us a bound on the number of gates we need to write down an arbitrary circuit based on the number of non-Clifford gates it contains.

Proposition 7.4.7. Let U be an n-qubit circuit consisting of Clifford gates and k non-Clifford phase gates. Then it can be written in the $\{\text{CNOT}, H, S, S^\dagger, Z(\alpha)\}$ gate set using at most $k(4n-1) + 4n(n+1) - 1 = O(kn + n^2)$ gates.

Proof By Proposition 7.4.6 our circuit has k Pauli exponentials, each of which requires at most $4n - 1$ gates to write down in our gate set by Proposition 7.3.9 for a total of $k(4n - 1)$. The final Clifford circuit requires an additional $4n(n + 1) - 1$ gates (Proposition 5.3.12). □

Note that in most useful quantum computations (those that do calculations that we can't efficiently do classically) we will have $k \gg n$, and hence the number of gates in Pauli exponential form scales as $O(kn)$.

Let's zoom out a bit. Why did it make sense to view our circuit as consisting of Clifford and non-Clifford gates, instead of using a more fine-grained difference between gates? In Chapter 5 we saw that computation involving just Clifford gates is efficiently classically simulable. The power of computation must then necessarily come from the inclusion of non-Clifford gates. We can then interpret a circuit in Pauli exponential form as first doing the 'actual' quantum computation using the Pauli exponentials, which are all non-Clifford, and then doing some final Clifford operations to 'post-process' the data in the right way. This analogy of Clifford operations being a type of post-processing becomes explicit when we are working with quantum error correcting codes. As we saw in Chapter 6, Clifford operations can often be done natively in a quantum error correcting code and so are 'free' operations. This is in contrast to non-Clifford operations which often require more complicated techniques to do fault-tolerantly.

7.4.3 Phase folding

Okay, so we can limit the number of Pauli exponentials we need to write an arbitrary circuit. But we can do even better!

Remember from Proposition 7.3.2 that if we have two commuting matrices A and B that then $e^{A+B} = e^A e^B$. It will be helpful to think of this going in the other direction as well: if we have commuting matrices, then we can combine adjacent matrix exponentials e^A and e^B into a single matrix exponential.

In particular, if we have a circuit in PE form then there will be many Pauli exponentials $e^{i\theta_j \vec{P}^k}$ in a row. When we then have two Pauli exponentials $e^{i\theta_k \vec{P}^k}$ and $e^{i\theta_l \vec{P}^l}$ right next to each other such that $\vec{P}^k$ and $\vec{P}^l$ commute we can combine them into a single exponential. This is especially helpful if $\vec{P}^k = \vec{P}^l =: \vec{P}$. Then we get:

$$ e^{i\theta_k \vec{P}^k} e^{i\theta_l \vec{P}^l} \;\; = \;\; e^{i\theta_k \vec{P}^k + i\theta_l \vec{P}^l} \;\; = \;\; e^{i(\theta_k + \theta_l)\vec{P}} $$

With the notation of Eq. (7.39) we can write this as $\vec{P}(\alpha)\vec{P}(\beta) = \vec{P}(\alpha + \beta)$.

Using Pauli boxes we can write this as:

$$(7.44)$$

However, the Pauli exponentials we want to combine into one might not be next to each other, as there might be other exponentials in the way. But remember also from Proposition 7.3.2 that if A and B commute, that then e^A and e^B will commute as well. So if two Pauli strings commute, then the associated exponentiated Paulis will commute as well:

$$\vec{P}\vec{Q} = \vec{Q}\vec{P} \iff \vec{P}(\alpha)\vec{Q}(\beta) = \vec{Q}(\beta)\vec{P}(\alpha)$$

So using this we can move exponentials out of the way in order to combine the ones we want.

Example 7.4.8. Consider the following circuit:
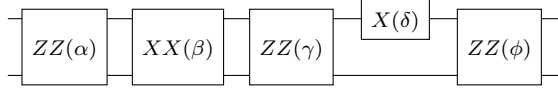
XX commutes with ZZ, and hence we can combine the $ZZ(\alpha)$ and $ZZ(\gamma)$, by moving the $XX(\beta)$ out of the way. However, we can't combine these with $ZZ(\phi)$ as there is an $X_1(\delta)$ in the way that does not commute with the ZZ phases (in general of course: if $\delta = 0$ then we can ignore that gate, and if $\delta = \pi$, then it is a Pauli X that we can push through the Pauli exponentials).

All of this suggests the following algorithm for reducing the number of Pauli exponentials needed to write a circuit, which we will call **phase folding**.

1. Start with a circuit in PE form (Definition 7.4.5)
2. Consider the first (leftmost) Pauli exponential $\vec{P}(\alpha)$.
3. If there is another $\vec{P}(\beta)$ in the circuit, look at all the Pauli exponentials in between $\vec{P}(\alpha)$ and $\vec{P}(\beta)$. If $\vec{P}$ commutes with all of these, then combine $\vec{P}(\alpha)$ and $\vec{P}(\beta)$ into one $\vec{P}(\alpha + \beta)$. Do this check and combination for all Pauli exponentials of $\vec{P}$ in the circuit.
4. Repeat the procedure for the next Pauli exponential in the circuit.
5. Check if any of the resulting Pauli exponentials is Clifford. If so, push these Clifford ones past the Pauli exponentials as described in Section 7.4.2, resulting in a new circuit in PE form. Repeat the algorithm.

If no Clifford Pauli exponentials were created in the previous steps, we are done.

7.5 Hamiltonian simulation

Back in Section 2.2.3, we mentioned briefly that the time evolution of a quantum system comes from taking the matrix exponent of a certain operator, called the Hamiltonian, which encodes all of the physical interactions going on. Now, suppose a physicist or an engineer hands us a Hamiltonian, and asks us to *simulate* the behaviour of a quantum system it represents. Is there a way we can do this on a quantum computer?

That is, if we start in a known quantum state $|\psi\rangle$, can we use a quantum computer to efficiently transform this state into it's time-evolved state $e^{-itH}|\psi\rangle$, then perform some measurements to learn something about it?

It turns out, if we just want to approximate $e^{-itH}|\psi\rangle$, the answer is yes! One way to do this is to employ the Suzuki-Trotter method, a.k.a. **Trotterization**. In our case, this will let us synthesise a circuit the approximates e^{-itH} using Pauli gadgets.

H is a self-adjoint operator on n qubits, so a generic H could take an exponential amount of data to describe. Of course, in that case, we don't have any hope of simulating it efficiently, because even reading H will take exponential time. Thankfully, lots of useful, physically meaningful H's can be written as a linear combination of Pauli strings:

$$H = \frac{1}{2}\sum_{j=1}^{m}\alpha_m \vec{P}_j$$

where m is polynomial in n. Then, if all the $\vec{P}_j$ commute, we're laughing because thanks to Proposition 7.3.2, then $U(t) := e^{-itH}$ splits up as a bunch of Pauli gadgets:

$$e^{-itH} = e^{-i\frac{\alpha_m t}{2}\vec{P}_m + \dots + -i\frac{\alpha_1 t}{2}\vec{P}_1} = e^{-i\frac{t\alpha_m}{2}\vec{P}_m}\dots e^{-i\frac{t\alpha_1}{2}\vec{P}_1}$$

i.e.

$$(7.45)$$

We already know how to synthesise Pauli gadgets, so we can build a nice Clifford+phase circuit, parametrised by t, that will perform time evolution for a generic input state. Life is good.

Unfortunately, if the Pauli strings in H *don't* commute, we can't use Proposition 7.3.2 to turn a sum in the exponent into a product of matrix exponents. Nevertheless, we can power on and try to decompose H as a product of Pauli exponentials, but we might introduce some errors along the way.

We'll start with some Hamiltonian that is the sum of just two other Hamiltonians: $H = H_1 + H_2$. In order to decompose its exponential into a composition of exponentials, we would like it to be the case that:

$$\vdots \boxed{e^{-itH}} \vdots \quad \overset{\epsilon}{\approx} \quad \vdots \boxed{e^{-itH_2}}\boxed{e^{-itH_1}} \vdots \tag{7.46}$$

for some ϵ. By definition, this means:

$$\|e^{-it(H_1+H_2)} - e^{-itH_1}e^{-itH_2}\| \le \epsilon$$

To show this, we'll need to roll up our sleeves and, look again at the definition of the matrix exponential:

$$e^{-itH} := \sum_{j=0}^{\infty} \frac{(-itH)^j}{j!} \tag{7.47}$$

We'll start by applying this to the LHS on (7.46). It will be handy to split this infinite sum into just 3 terms: one that doesn't depend on t, one that is linear in t, and all the rest, which are (at least) quadratric in t. In other words, we can find some linear map M such that:

$$e^{-i(H_1+H_2)t} = I - i(H_1+H_2)t + Mt^2 \tag{7.48}$$

Operating similarly on the RHS of (7.46), we get:

$$\begin{aligned} e^{-iH_1t}e^{-iH_2t} &= (I - iH_1t + \ldots)(I - iH_2t + \ldots) \\ &= I - iH_1t - iH_2t + H_1H_2t^2 + \ldots \\ &= I - i(H_1+H_2)t + Lt^2 \end{aligned} \tag{7.49}$$

for some L. Subtracting these two expressions gives:

$$\|e^{-it(H_1+H_2)} - e^{-itH_1}e^{-itH_2}\| = \|t^2 M - t^2 L\| = t^2\|M - L\| \tag{7.50}$$

It turns out this is the first step to getting a pretty good approximation.

Exercise* 7.5. Show that there is a positive number κ, that does not depend on t, such that $\|M - L\| \le \kappa$.

This gives us $t^2 \|M - L\| \leq t^2 \kappa$ and hence by (7.50) we have:

This basic fact lets us take a Hamiltonian written in terms of Pauli strings, and "peel off" the Pauli exponentials, one-by-one. That is, for $H = \frac{1}{2} \sum_k \alpha_k \vec{P}_k$, there exist some constants $\kappa_1, \ldots, \kappa_m$ such that:

By triangle identity, we conclude:

                           (7.51)

where $\kappa = \sum_k \kappa_k$.

So, we have managed to put a bound on our approximation, and hence the error we'll give if we try to simulate H. But is this bound any good? There is no reason κ needs to be small, and in fact usually it won't be. So, our best bet is taking t to be very small. But what if we don't want to take t to be small, because we want to simulate H for a larger amount of time? It turns out there is a nice trick for this: we perform the decomposition (7.51) many times and compose the results.

That is, we can take advantage of the fact that

then approximate each of the maps $e^{-i\frac{t}{d}H}$ individually using (7.51).

Initially, we might think this is a stupid thing to do, since now we'll be making even more approximations, so maybe things will get worse. The

crucial point, however, is that the error in (7.51) depends not on t, but in t^2. So, even though we have to make this approximation d times, we can still bound the error by $d \cdot (\frac{t}{d})^2 \kappa = \frac{t^2}{d} \kappa$. Hence, we get:

$$
\begin{array}{ccc}
\vdots\; \boxed{e^{-itH}}\; \vdots & \overset{\frac{t^2}{d}\kappa}{\approx} & \vdots \underbrace{\boxed{\vec{P}_1 \cdots \vec{P}_m}}_{} \vdots \cdots \vdots \underbrace{\boxed{\vec{P}_1 \cdots \vec{P}_m}}_{d} \vdots
\end{array}
\tag{7.52}
$$

Supposing we wish to approximate within some error ϵ, we should choose d such that $\frac{t^2}{d}\kappa \leq \epsilon$. When $t = 1$, we should therefore choose some $d \geq \frac{\kappa}{\epsilon}$.

So, we now know how to build a quantum circuit C that approximates e^{-itH} up to some parameter ϵ. We can use C to do a pretty good job at predicting how the real system behaves. That is, for a fixed input state $|\psi\rangle$ and any measurement outcome $\vec{b}$, the probabilities of seeing $\vec{b}$ on the "real" system described by H and the "simulated" system described by C are pretty close. To see this, we'll compare the two probabilities:

$$
\text{Prob}_{real}(\vec{b}) = |r|^2 \qquad \text{Prob}_{sim}(\vec{b}) = |s|^2
$$

for amplitudes $r := \langle\vec{b}|e^{-itH}|\psi\rangle$ and $s := \langle\vec{b}|C|\psi\rangle$. First, since $|\psi\rangle$ and $|\vec{b}\rangle$ are normalised, we have:

$$
e^{-itH} \overset{\epsilon}{\approx} C \implies \langle\vec{b}|e^{-itH}|\psi\rangle \overset{\epsilon}{\approx} \langle\vec{b}|C|\psi\rangle
$$

so $r \overset{\epsilon}{\approx} s$. Since taking the adjoint preserves norms, we also have $\bar{r} \overset{\epsilon}{\approx} \bar{s}$. Since $|r|$ and $|s|$ must both be less than 1, we have:

$$
|r|^2 = \bar{r}r \overset{\epsilon}{\approx} \bar{r}s \overset{\epsilon}{\approx} \bar{s}s = |s|^2
$$

Applying the triangle inequality, we have $|r|^2 \overset{2\epsilon}{\approx} |s|^2$. Hence:

$$
\text{Prob}_{real}(\vec{b}) \overset{2\epsilon}{\approx} \text{Prob}_{sim}(\vec{b})
$$

7.6 Simplifying universal diagrams

In this chapter our focus has been on universal *circuits*. But we of course shouldn't be neglecting our *diagrams*. A universal ZX-diagram is one where we put no restriction on the phases the spiders may contain. It turns out that we can apply some of the tricks we've been developing for reasoning about universal circuits to universal ZX-diagrams. In particular, phase gadgets

are also really useful in this broader context. For one, we can use them to power-up the simplification strategy of Clifford diagrams of the previous chapter.

Let's recap this strategy. Recall from Sections 5.2 and 5.3 that we have two simplification rules, local complementation and pivoting, that allow us to remove certain spiders with a Clifford phase. We then do the following:

1. First we rewrite the diagram to a graph-like diagram, so that all spiders are *Z*-spiders and they are only connected via Hadamard edges.

2. Then using local complementation we remove all internal spiders with a $\pm\frac{\pi}{2}$ phase.

3. Similarly, using pivoting we remove any pair of connected spiders that are both internal and have a 0 or π phase.

Since we'll need to refer to spiders with a phase of 0 or π a lot in this section, we'll give them a name. We will call such a spider a **Pauli spider**.

Now, using this strategy, if the diagram is Clifford, the only remaining internal spiders will be Pauli spiders that are only connected to boundary spiders. In Section 5.3.2 it is described how we can also get rid of these spiders, so that *all* internal spiders are removed, and the diagram will be in GSLC form.

Now, let's look at what happens if we do the same strategy for a *non-Clifford* diagram, i.e. where our spiders are allowed to be labelled by arbitrary phases. Such a diagram we can still rewrite to graph-like form using Proposition 5.1.8, but now the spiders might carry a Clifford phase (a multiple of $\frac{\pi}{2}$), or a non-Clifford phase (any other value). We can apply the local complementation and pivoting rewrites as before to remove any internal spider with a $\pm\frac{\pi}{2}$ phase and a connected pair of internal Pauli spiders.

When we do this we have several types of internal spiders that can remain. First, none of these rewrites remove spiders that have a non-Clifford phase, so if an internal spider has a phase that is not a multiple of $\frac{\pi}{2}$, then it will still be in the diagram. Second, we can only remove the Pauli spiders that are connected to other Clifford spiders (either an internal Pauli spider, or an arbitrary boundary Clifford spider). Hence, we can also have internal Pauli spiders remaining that are connected only to non-Clifford spiders.

We cannot actually remove these Pauli spiders, but we can do something interesting with them: transform them into phase gadgets. As we are working here with graph-like ZX-diagrams, the phase gadgets will look slightly

different:

$$
 \tag{7.53}
$$

Lemma 7.6.1. We can transform an internal Pauli spider connected to another internal spider into a phase gadget using pivoting:

Proof First push the $j\pi$ phase to the right, and then unfuse the γ phase onto its own spider:

Now just apply the regular pivot Lemma 5.2.11 to the two spiders marked by *. □

We see that we end up with the same number of spiders, but that they are connected differently. The γ phase is now isolated on a spider that only has arity 1, i.e. it has become a phase gadget. This turns out to have several benefits.

But first, note that if the internal Pauli spider is only connected to boundaries that we can do a similar rewrite.

Lemma 7.6.2. The following boundary pivot simplification holds:

Here we take the spider with the γ phase to be a boundary spider with the top wires being inputs and outputs to the diagram.

Proof First add spiders to the outputs of the γ spider as in Eq. (5.15) and then use Lemma 7.6.1. □

As in Section 5.3.2, doing operations on the boundary introduces Hadamards on the input and output wires of the diagram so that we are working with graph-like diagrams with Hadamards.

After we've applied the rewrites of Lemmas 7.6.1 and 7.6.2 to all the internal Pauli spiders we see that these internal Pauli spiders are now all part of a phase gadget. As a side-note, in order to not get stuck in an infinite loop in doing these rewrites, we should only be applying these lemmas to Pauli spiders that are not already part of a phase gadget (i.e. we should check if they do not have any single-arity neighbours).

Example 7.6.3. After we apply the rewrite strategy we described above, we might be left with a diagram that looks something like the following:
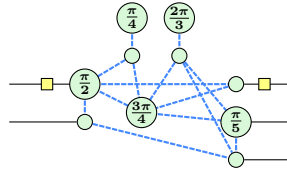
Every internal spider is either part of a phase gadget or has a non-Clifford phase.

Remark 7.6.4. In the diagrams we get from this rewrite strategy, none of the phase gadgets will be connected to each other. To see why this is, note that in Lemmas 7.6.1 and 7.6.2 the phase gadget we create is connected to precisely those spiders that the original Pauli spider was connected to. Our starting assumption was that no Pauli spider is connected to any other internal Pauli spider, since if that was the case, then we could have just applied a regular pivot to them to remove them. As each phase gadget is hence connected to what a Pauli spider was connected to, none of them will be connected to each other. Conversely, if we were given a diagram where two phase gadgets are connected to each other, then we could apply a regular pivot to them, and this would remove the 'base' of these gadgets, and change the phases into 'regular' internal spiders.

While these 'gadgetisation' rewrites don't get rid of more spiders, it is still very useful to do them. First, it turns out that the diagrams we can bound the size of the diagrams we get in terms of the number of non-Clifford phases, and second, there are interesting new rewrites that we can apply to the phase gadgets.

For the first point, note first that if the phase γ in the phase gadget is Clifford that we can remove the phase gadget.
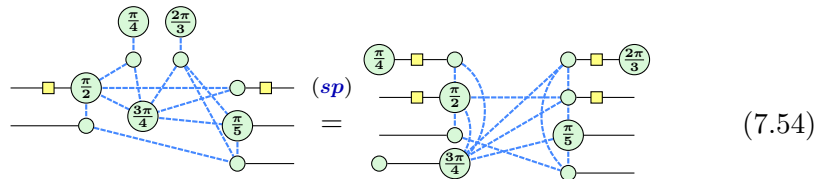
Exercise 7.6. Show that if we have a phase gadget in a graph-like diagram with a phase that is a multiple of $\frac{\pi}{2}$, that we can remove it using a series of local complementation or pivots.

Hence, the only remaining internal spiders are either non-Clifford or are Pauli spiders part of a non-Clifford phase gadget. As none of our rewrites introduce new non-Clifford phases, this means that the number of internal spiders we end up with is bounded by the number of non-Clifford phases we started out with.

Proposition 7.6.5. Let D be a ZX-diagram with n inputs, m outputs and k non-Clifford phases. Then we can efficiently rewrite D to a diagram D' which has at most $n + m + 2k$ spiders and $n + m$ Hadamards on the inputs and outputs.

Proof Transform D into graph-like form and apply the rewrites described above. When we are done with the rewrites we can have a spider for each of the inputs and outputs, contributing at most $n + m$ spiders. Each of the remaining internal spiders is either non-Clifford, or is a Pauli spider that is part of a non-Clifford phase gadget, so that there are at most $2k$ internal spiders. Since the resulting diagram is graph-like with Hadamards we can potentially also have Hadamards on the input and output wires. □

This result is in a sense a more fine-grained version of Proposition 5.3.7 that shows that we can rewrite any Clifford diagram to GSLC form: if $k = 0$ in the proposition above, then the result is a GSLC form diagram. We can in fact view all the internal spiders and phase gadgets as additional states and effects plugged into a GSLC diagram (see also Exercise 5.5):

$$\tag{7.54}$$

This is pointing in the direction of how we can do arbitrary quantum computations by preparing the right graph state and doing the right measurements. We will explore this type of measurement-based quantum computing in detail in Chapter 8.

The fact that the size of our diagram is bounded by the number of non-Clifford gates, and not by the overall number of spiders in the starting

diagram turns out to be useful when we want to simulate universal quantum circuits (see Section 7.8.1).

Remark* 7.6.6. So if we can rewrite our diagram to a form that is bounded in size by the number of non-Clifford spiders, and if the phase gadget rewrites we need to do to get there don't remove any spiders, does that mean that we could have already bounded the size of the diagram *before* doing these rewrites? Well, no. This is because Lemmas 7.6.1 and 7.6.2 *can* actually result in spiders being removed from the diagram. This happens when we have two Pauli spiders that are connected to exactly the same set of non-Clifford spiders. In this case, when we apply Lemma 7.6.1 to one of the Pauli spiders and one of its neighbours, the other Pauli spider gets completely disconnected and becomes a scalar, which we can then ignore. In general, Lemma 7.6.1 will change the connectivity of the Pauli spiders in the diagram, and hence through a sequence of rewrites we could end up with Pauli spiders that are connected to the same set of non-Clifford spiders, resulting in more spiders getting removed by these rewrites.

7.6.1 Removing non-Clifford spiders

Once we have simplified our diagram to a point where all internal spiders are either non-Clifford or phase gadgets, there are a couple more useful rewrite rules we can apply that remove additional spiders.

The first is very simple, as it is just a case of removing an identity spider.

Lemma 7.6.7. We can fuse a one-legged phase gadget with its neighbour:
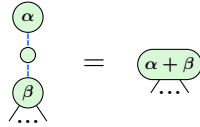
Proof
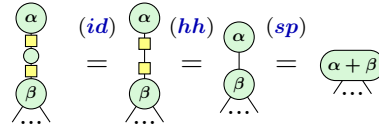
So whenever we have a phase gadget in our diagram that is connected to exactly one spider, we can fuse it with its neighbour. There is really nothing special about phase gadgets here: as noted in Remark 7.6.4, the phase gadget is connected to what the original Pauli spider was connected to. So if we have a one-legged phase gadget in our diagram, then the original Pauli spider must

have also already have been an identity spider, and we could have removed it then and there.

The other rewrite rule we can apply to phase gadgets is one we have already seen: gadget fusion.

Lemma 7.6.8. We can fuse two phase gadgets connected to the same set of spiders:
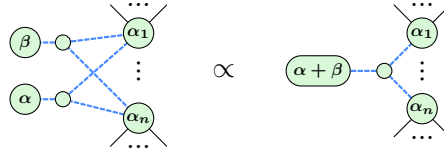
Proof

$\square$
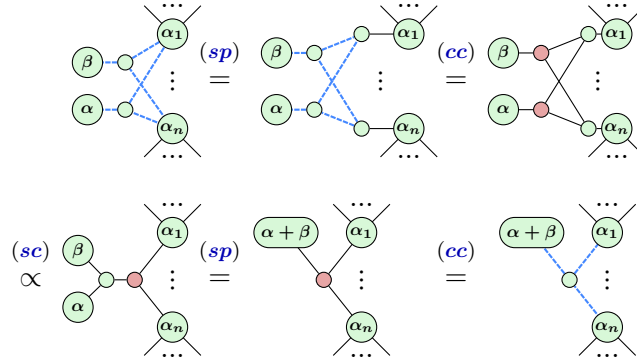
So why do we point out these two rewrite rules for getting rid of phase gadgets? Two reasons: first, note that this phase gadget fusion rule is a generalisation of the phase-folding we can do in the phase polynomial framework. Second, the rewrites of Lemmas 7.6.7 and 7.6.8 are essentially the *only* rewrite rules we can do to get rid of additional non-Clifford phases in a diagram. Or that is, this is the case when we treat the non-Clifford phases as 'black boxes', where we treat the phase as some arbitrary angle that we don't know anything else about. See the References of this chapter for more details. If we know more about the phases, such as that they are all multiples of $\frac{\pi}{4}$, then there are other potential rewrites we could apply that simplify the diagram further. We look at this in detail in Chapter 10.

7.6.2   Circuits from universal diagrams

In previous chapters we have seen that certain gate sets have a natural diagrammatic counterpart, and that for unitary diagrams we can always transform them back into circuits over this gate set. In Chapter 4 we saw

that any unitary phase-free diagram can be rewritten into a CNOT circuit, and in Chapter 5 we saw that any unitary Clifford diagram can be turned back into a Clifford circuit.

This then raises the question of how we can transform these universal diagrams back into universal circuits when they are unitary. Perhaps disappointingly, but not too surprisingly, this does not turn out to be efficiently doable *in general*.

Exercise* 7.7. In this exercise we will see that if we had some function **CircuitExtraction** that takes in a poly-size unitary ZX-diagram and spits out a poly-size circuit implementing it, that then we could solve NP-complete problems with it. Hence, there is probably no implementation of **CircuitExtraction** that is efficient. In particular, what we will show is how to determine whether a Boolean formula $f : \mathbb{F}_2^n \to \mathbb{F}_2$ has a solution using **CircuitExtraction**.

1. Let L_f be the linear map defined by $L_f|\vec{x}\rangle = |f(\vec{x})\rangle$. We will see in Chapter 9 how we can construct maps like L_f efficiently as a ZX-diagram. Argue that

$$\vdots\ \boxed{L_f}\ -\ =\ \frac{N_0}{2^n}|0\rangle + \frac{N_1}{2^n}|1\rangle \tag{7.55}$$

   where N_a is the number of values $\vec{x}$ for which $f(\vec{x}) = a$.

2. If we could calculate the value of Eq. (7.55) then we could determine whether f has a solution. However, **CircuitExtraction** expects a unitary diagram as input, which Eq. (7.55) is not. So we need to construct a unitary diagram containing Eq. (7.55) that allows us to determine N_1. Using the fact that a superposition $\lambda_0|0\rangle + \lambda_1|1\rangle$ can be written as $Y(\alpha)|0\rangle$ show that

$$\vdots\ \boxed{L_f}\ \bigl(-\tfrac{\pi}{2}\bigr) \quad \propto \quad -\!\!\alpha\!\!- \tag{7.56}$$

   and determine α in terms of N_1.

3. The left-hand side of Eq. (7.56) is proportional to a unitary diagram, and hence when given to **CircuitExtraction** will spit out a 1-qubit circuit equal to the right-hand side of Eq. (7.56). Argue that with such a circuit we can efficiently determine α and hence N_1 (you may ignore issues about numerical precision and the precise gate set that **CircuitExtraction** is using).

4. Give the full algorithm that takes in a Boolean function f and with a single call to **CircuitExtraction** tells you whether f has a solution or not.

In this procedure we described we actually know more than just whether f has *a* solution: we get N_1, the *total number* of solutions. **CircuitExtraction** hence doesn't just allow us to solve NP-complete problems, it allows us to solve #P-complete problems, which are generally considered to be much harder.

Universal diagrams can just become too complicated, making it impossible to efficiently see what circuit it is equal to. Luckily for us though, for many diagrams we care about we can tame these complications and efficiently extract out a circuit. This is made possible by a strong connection between graph-like ZX-diagrams and *measurement patterns*. The ability to extract a circuit from the diagram then corresponds to whether the measurement pattern can be made deterministic. We will explain all of this and more in detail in the next chapter on measurement-based quantum computing.

7.7   Summary: What to remember

1. A circuit consisting of CNOT and Z phase gates can be represented by a *phase polynomial* followed by a linear Boolean map acting on the computational basis states.
2. In the ZX-calculus a phase polynomial looks like a collection of *phase gadgets*.
3. The phase-polynomial representation can be synthesised back into a CNOT+Phases circuit using a variation on the Gaussian elimination algorithm of Chapter 4.
4. The *pathsum* technique allows us to incorporate Hadamards in the phase-polynomial representation. Each Hadamard gate introduces a new *path variable* that replaces the current variable on that qubit. By thinking about pathsums we can synthesise complex unitaries like the Quantum Fourier Transform.
5. A different way to think about universal circuits is through *Pauli exponentials*. Any quantum circuit can be written as a series of (non-Clifford) Pauli exponentials, followed by a Clifford circuit.
6. The Pauli exponential representation allows us to bound the size of an n-qubit circuit with k non-Clifford gates by $O(kn + n^2)$.
7. We can construct a quantum circuit for a Hamiltonian evolution e^{-itH}

by splitting it up into small time-slices $e^{-i\frac{t}{n}H}$ and approximating each of these by a series of Pauli exponentials.

8. We can use the Clifford simplification strategy of Chapter 5 to optimise universal diagrams. By using a variation on pivoting we can introduce phase gadgets in such diagrams, and these can then be merged together.

9. In these optimised graph-like diagrams, the only internal spiders are those with non-Clifford phases, or they form the 'base' of a phase gadget.

10. In general, extracting a circuit out of a unitary ZX-diagram is a hard problem (though in certain special cases we can do it efficiently).

7.8 Advanced material*

7.8.1 Simulating universal circuits*

Each of the gates we have considered in this chapter—Z phase gates, CNOTs, and Hadamards—has a different interaction with the path-sum expression of the circuit.

- A Z phase gate looks at the expression that is currently in the qubit it acts on, and puts that expression into a phase.
- A CNOT takes the expression in its control qubit, and XORs it with that of its target qubit.
- A Hadamard introduces a new path-variable, adds a phase term using it and the expression in the qubit it acts on, and finally replaces the expression with the new variable.

What is interesting about these three types of gates is that we truly need *all* of them in order to do interesting quantum computations. If we only had phase gates and CNOTs, then as we saw in Section 7.1.1 we can efficiently evaluate what it does on a computational basis state, as there is just one path involved. If we instead only had phase gates and Hadamards, then all the qubits would be acting on their own, and so there wouldn't be any complex behaviour and we could directly calculate the matrices involved. And finally, if we only had CNOTs and Hadamards, then the circuit would be Clifford, so that we can also efficiently calculate the outcomes.

The interesting behaviour then comes from a multitude of paths (because of Hadamards) which are non-trivially using all parts of the available Hilbert space (because of CNOTs), and each of which can have a different phase so that when we do a measurement the paths interfere in non-trivial ways.

While these arguments only show that you need at least *some* of each type of gates, it turns out that we can prove a bit more. Suppose for instance that

we have a n-qubit circuit with a polynomial number of CNOT, Hadamard and Z phase gates, but that we only have $k \log n$ CNOT gates for some k. Then this means that most qubits don't have *any* CNOT on them: at most $2k \log n$ qubits can be connected by CNOTs. The size of the matrix associated to these qubits is then $O(2^{2k \log n}) = O(n^{2k})$. So calculating what this circuit is doing can be done in polynomial time just by directly representing the matrix and calculating the action of all the gates on it one-by-one.

Analogously, if we have only $k \log n$ Hadamard gates, then using the path-sum representation of the circuit we only have to keep track of $O(2^{k \log n}) = O(n^k)$ different paths, so that we can efficiently simulate the action of the circuit as well, just by sampling from this polynomial number of paths.

These two simulation techniques, based on a cost that scales based on the number of Hadamards or CNOTs are not seriously considered (though lots of tricks for tensor network contractions are based on understanding where entanglement is created and how, and so counting where the CNOTs are is definitely implicitly part of that), but one that is based on a limited number of non-Clifford phases turns out to work quite nicely.

As it turns out, if we only have $k \log n$ non-Clifford Z phase gates, then we can also efficiently strongly simulate the circuit. To see this, let's assume that we have some polynomial size circuit of CNOT, Hadamard and Z phase gates of which t are non-Clifford phases, and that we wish to calculate a specific amplitude of this circuit (let's say the amplitude of observing $\langle 0 \cdots 0 |$ on input of $| 0 \cdots 0 \rangle$). Then we can write this as a scalar ZX-diagram. We can rewrite this diagram into graph-like form, and simplify it further to remove all the Clifford spiders. As described in Section 7.6.1, the number of spiders in the resulting diagram is then $O(2t)$ as each spider either carries a non-Clifford phase, or is part of a phase gadget which has a non-Clifford phase.

In such a diagram, we can't remove any more Clifford spiders, since there aren't any. However, it turns out we can decompose a non-Clifford spider into a sum of Cliffords, so that then the whole diagram becomes a sum of 'slightly more Clifford' diagrams. To do this in a nice generalisable way we first unfuse the non-Clifford phase onto its own spider:

The remaining phase is then Clifford

Now, this state with the α phase is equal to $|0\rangle + e^{i\alpha}|1\rangle$ and hence can be decomposed into a sum of two Clifford spiders corresponding to these terms

$|0\rangle$ and $|1\rangle$:

$$\alpha\!\!-\!\!- \;=\; \frac{1}{\sqrt{2}}\left(\circ\text{-}\text{-}\circ\!\!-\!\!- \;+\; e^{i\alpha}\,\pi\text{-}\text{-}\circ\!\!-\!\!-\right)$$

When we do this unfuse-and-decompose to a single α phase in a bigger diagram, we are then left with two diagrams that each have one fewer non-Clifford phase. We can however do this decomposition to *all* of the non-Clifford phases in the larger diagram at the same time, giving 2^t different diagrams that are now completely Clifford. Each of these diagrams had $O(t)$ spiders, and hence to completely 'simplify away' these diagrams into a single scalar number using the Clifford simplification strategy of Chapter 5, requires $O(t^3)$ graph operations. As we have to do this for each of the 2^t diagrams, the total cost is then $O(t^3 2^t)$. Hence, if we have a logarithmic number of non-Clifford phases $t = k\log n$, then the simulation cost is $O(k^3 n^k \log^3 n)$, which is polynomial in n for a fixed k.

We could also decide not to decompose all phases simultaneously, but instead try to simplify the diagram further after every decomposition. Since a decomposition makes the diagram look a little more Clifford, the Clifford rewrites could make the diagram a bit simpler. More importantly, they might reveal places where the non-Clifford removing rewrites of Section 7.6.1 might apply, so that there are fewer terms we need to decompose in the first place.

If all the non-Clifford phases are arbitrary, with all the non-Clifford phases not being equal to each other then this decompose-optimise-repeat strategy is pretty much the best we can do with this method. However, if the non-Clifford phases are some specific angles, then there is more interesting stuff we can do! In particular, if all the non-Clifford phases are multiples of $\frac{\pi}{4}$, for instance when the circuit we start out with is Clifford+T, then it turns out that we can decompose an entire *group* of $\frac{\pi}{4}$ phases simultaneously, and this requires fewer terms. For instance, if we have a pair of $|T\rangle := \sqrt{2}T|+\rangle = |0\rangle + e^{i\frac{\pi}{4}}|1\rangle$ states, then:

$$
\begin{aligned}
|T\rangle \otimes |T\rangle \;&=\; |00\rangle + e^{i\frac{\pi}{4}}|01\rangle + e^{i\frac{\pi}{4}}|10\rangle + e^{i\frac{\pi}{2}}|11\rangle \\
&=\; (|00\rangle + e^{i\frac{\pi}{2}}|11\rangle) + e^{i\frac{\pi}{4}}(|01\rangle + |10\rangle).
\end{aligned}
$$

Here we've grouped the terms together into two Bell-like maximally entangled states. Written in diagrams we hence have:

$$
\begin{matrix}\frac{\pi}{4}\!\!-\!\!-\\ \frac{\pi}{4}\!\!-\!\!-\end{matrix} \;=\; \frac{\pi}{2}\!\!\prec \;+\; e^{i\frac{\pi}{4}}\,\pi\!\!\prec\!\!\circ
$$

Naively decomposing these two non-Clifford phases would give us 4 terms, but by grouping them into these Clifford states we require only 2 terms.

Hence, if we have t of these $\frac{\pi}{4}$ phases, then the amount of terms we get when we decompose all of them in terms of pairs like this is $2^{\frac{1}{2}t}$ (assuming t is even for simplicity). This hence scales quadratically better than decomposing them all separately!

This is an example of a **stabiliser decomposition** and there are many interesting things to say about them. A stabiliser decomposition is any decomposition of a quantum state into a sum of Clifford states. This is always possible to do, but for an n-qubit state will require in general 2^n stabiliser terms. Finding more efficient decompositions is generally a hard problem. However, for very special states, like a tensor product $|T\rangle \otimes \cdots \otimes |T\rangle$, much better decompositions are known. For instance, it turns out we can decompose six copies of $|T\rangle$ into just 6 terms, meaning we only require $6^{t/6} = 2^{t\frac{1}{6}\log_2 6} \approx 2^{0.43t}$. There are also specific configurations of $\frac{\pi}{4}$ phases that allow for more efficient decompositions.

Note that this stabiliser decomposition simulation technique allows us to calculate an amplitude. If we want to calculate a marginal probability (cf. Section 5.4.2), then we need to 'double' the diagram. This doubles the number of non-Clifford phases, which is a big problem since this method scales exponentially in the number of non-Cliffords. In practice, the interleaved optimisation steps do get rid of some of the redundancy in this doubled representation, but the problem does persist. Usually we are doing strong simulation, and hence calculation of marginal probabilities, because we are actually trying to do weak simulation (i.e. sampling). It turns out that in this case there is a way around having to double the diagram by using a **gate-by-gate simulation** method. This allows us to do sampling by just calculating amplitudes, no marginal probabilities necessary, at the cost of having to calculate a number of amplitudes that scales with the number of *gates* instead of in the number of qubits. See the References of this chapter for more about this.

Another way around this issue of doubling is to not even try to calculate amplitudes, but instead directly do sampling by writing our circuit as a stochastic combination of Clifford computations. We will say a bit more about this when we are talking about *computational universality* in Section 10.5.2. When doing this, we no longer care about the number of terms in the decomposition, but instead we care about the total *weight* of the terms, i.e. the scalar factors in front of them. This weight is known as the **robustness of magic** or the **stabiliser extent** of the state (which is used depends on *which* weights exactly we are talking about).

7.8.2 Higher-order Trotterisation*

The approximation of e^{-itH} we found in Section 7.5 required a number of decompositions n that scaled as $O(\varepsilon^{-1})$. This is fine. But it does raise the question of whether we could do better.

Let's look back at the case with just two terms H_1 and H_2. The reason we didn't get better scaling than $O(\varepsilon^{-1})$ is because $e^{-itH_1}e^{-itH_2}$ only approximates $e^{-it(H_1+H_2)}$ up to a $O(t^2)$ term. If we could somehow boost their agreement up to some $O(t^k)$ for a $k > 2$, then we would get better scaling. Looking at the expansions of these two expressions in Eqs. (7.48) and (7.49) we see that the t^2 terms are respectively $\frac{1}{2}(H_1^2 + H_2^2 + H_1H_2 + H_2H_1)$ and $\frac{1}{2}(H_1^2 + H_2^2 + 2H_1H_2)$. So the only problem is that we get two copies of the term H_1H_2 instead of $H_1H_2 + H_2H_1$. If we had decomposed $e^{-itH_2}e^{-itH_1}$ instead of $e^{-itH_1}e^{-itH_2}$, then we would have lacked the $H1H_2$ term instead of the H_2H_1 term. This is pointing us towards the solution: we need to have both $e^{-itH_1}e^{-itH_2}$ *and* $e^{-itH_2}e^{-itH_1}$ in our decomposition.

So let's see what we get when we decompose the product of these decompositions:

$$\left(e^{-itH_1}e^{-itH_2}\right)\left(e^{-itH_2}e^{-itH_1}\right) = I - it(2H_1 + 2H_2) - \frac{t^2}{2}(2H_1^2 + 2H_2^2)$$
$$- t^2(2H_1H_2 + H_1^2 + H_2^2 + 2H_2H_1) + O(t^3)$$
$$= I - i(2t)(H_1 + H_2) - \frac{(2t)^2}{2}(H_1^2 + H_2^2 + H_1H_2 + H_2H_1) + O(t^3).$$

Here in the last equality we suggestively grouped together the t^2 term in terms of $(2t)^2$. We see that in this case we get the correct quadratic term. So replacing t by $\frac{1}{2}t$ to ensure we get the correct constants, we see that:

$$e^{-it(H_1+H_2)} = e^{-\frac{1}{2}itH_1}e^{-itH_2}e^{-\frac{1}{2}it} + O(t^3). \tag{7.57}$$

Exercise* 7.8. In this exercise we will find a way to lift Eq. (7.57) to a full-fledged procedure for Hamiltonian simulation.

a) Calculate the difference $e^{-it(H_1+H_2)} - e^{-\frac{1}{2}itH_1}e^{-itH_2}e^{-\frac{1}{2}it}$ up to $O(t^4)$ terms. I.e. calculate the t^3 term of this difference.

b) Express this difference in terms of (nested) commutators like $[H_1, [H_1, H_2]]$ and give an inequality of the difference in norm.

c) Let $\mathcal{S}_2(\{H_1, \ldots, H_l\}, t) := e^{-\frac{1}{2}itH_1} \cdots e^{-\frac{1}{2}itH_l}e^{-\frac{1}{2}itH_l} \cdots e^{-\frac{1}{2}itH_1}$. Give an upper bound on

$$\|e^{-it\sum_j^l H_j} - \mathcal{S}_2(\{H_j\}, t)\|$$

which depends on t^3.

d) Give an upper bound to the difference in norm of $\|e^{-it\sum_j^l H_j} - \mathcal{S}_2(\{H_j\}, \frac{t}{n})^n\|$
    and use this to show that if we want to make this difference smaller than
    ε, that we can then choose $n = O(t^{3/2} l^{3/2} / \varepsilon^{1/2})$.

This approach is known as **second-order Trotterization**. It is called
second order because it recovers $e^{-it(H_1+H_2)}$ correctly up to the second
order. It is possible to generalise this technique to kth order. This involves
a map $\mathcal{S}_k(\{H_j\}, t)$ that approximates $e^{-it\sum_j H_j}$ up to a $O(t^{k+1})$ error. To
approximate it up to an error ε we then split t up into n pieces where
$n = O((tl)^{1+1/k} \varepsilon^{1/k})$. While this looks like it is then better to use higher
and higher-order Trotterisations, there are some hidden constants here: the
number of terms in $\mathcal{S}_k(\{H_j\}, t)$ scales exponentially with k. As a result it is
often not beneficial to go beyond $k = 2$, and essentially never to go beyond
$k = 8$.

7.8.3   Randomised compiling*

It turns out that if we want to approximate a unitary, there are multiple
benefits to compiling this to a family of different, somewhat randomly chosen,
unitaries that together 'approximate' the unitary.

Let's see how this could work. Let's again consider a Hamiltonian $H = \sum_j^l \lambda_j H_j$. Here we choose the decomposition of H such that $\lambda_j \geq 0$ and
$\|H_j\| = 1$ for all j. Then the unitary channel corresponding to a $1/n$ fraction
of the total time evolution is

$$\mathcal{U}_n(\rho) \;=\; e^{i\frac{t}{n}H}\rho e^{-i\frac{t}{n}H} \;=\; \rho + i\frac{t}{n}(H\rho - \rho H) + O\left(\frac{t^2}{n^2}\right)$$

$$=\; \rho + i\sum_j \frac{th_j}{n}(H_j\rho - \rho H_j) + O\left(\frac{t^2}{n^2}\right). \tag{7.58}$$

We see that up to some error term, this unitary channel can be written
as a sum over some expression involving just a single H_j. This suggests
we should be able to find a probabilistic channel that can approximate
$\mathcal{U}_n$, just by sampling from H_j. Let's give this a try. Let's define $\mathcal{E}(\rho) = \sum_j p_j e^{i\tau H_j}\rho e^{-i\tau H_j}$. Here p_j is some probability distribution over the terms
$H_1, \ldots, H_l$, and τ is some fixed number that we will try to determine later.
Now let's see what happens when we expand $\mathcal{E}$ in terms of τ:

$$\mathcal{E}(\rho) \;=\; \rho + i\sum_j p_j \tau (H_j\rho - \rho H) + O(\tau^2) \tag{7.59}$$

So we see that this matches the expansion in Eq. (7.58) when we have

$p_j \tau = h_j \frac{t}{n}$. Since the p_j must form a probability distribution, we have $\sum_j p_j = 1$, and hence

$$\tau = \sum_j p_j \tau = \sum_j h_j \frac{t}{n} = \Lambda \frac{t}{n},$$

where we have set $\Lambda = \sum_j h_j$. Then solving for p_j we get $p_j = \frac{h_j}{\Lambda}$.

This means we can approximate the unitary channel $\mathcal{U}_n$ up to a second order $O(\frac{t^2}{n^2})$ error using a channel that is a probabilistic combination of unitaries, each of which is just a single simple conjugation with $e^{-i\tau H_j}$, as long as we choose $\tau = \Lambda \frac{t}{n}$ and $p_j = \frac{h_j}{\Lambda}$. Here Λ is the sum of all the weights h_j of the sub-Hamiltonians H_j. But of course, we don't want to approximate $\mathcal{U}_n$, but $\mathcal{U}_n^n$. Using a similar type of analysis as we have done before (but adapted to work with channels instead of unitaries), we can show that we can approximate $\mathcal{U}_n^n$ with $\mathcal{E}^n$ up to an error ε, by choosing $n \geq 2\Lambda^2 t^2/\varepsilon$. This is interesting because the amount of terms we need does not depend on l, the amount of terms in H, but rather on Λ, the sum of the weights of the terms. This method hence works better than the non-randomised technique when l is large, but Λ is not so large. But even in the worst case, when we have $\Lambda = l$ (this is when $\lambda_j = 1$ for all j), the scaling is $n \geq 2l^2 t^2/\varepsilon$, which is still better than the non-randomised version: recall that we got $n \geq \frac{1}{2} l^2 t^2/\varepsilon$. But each iteration required every $e^{-it\lambda_j H_j}$ to be placed, so that each iteration consists of l term, giving a total gate count scaling of $O(l^3 t^2/\varepsilon)$.

Remark 7.8.1. You might find it very weird or counter-intuitive that we could approximate a specific unitary by creating an ensemble of random 'bad' approximations. However, remember that at the end of a quantum circuit we need to do measurements, and that our output, the only thing we can really 'see', is just a probability distribution over measurement outcomes. So the only thing we need for something to approximate a quantum circuit well, is for all the measurement outcomes to follow the correct probability distribution. So even though every single run of this protocol might not be a good approximation to the unitary, because we use a different one for every run of measurement, the errors can cancel out. We will see a similar idea explored in Section 10.5.2.

This technique here is a variation on Trotterisation. There are however also other techniques that can approximate $e^{-it \sum_j H_j}$ in quite a different way that also result in very favourable scaling. See the References of this chapter for more on this.

7.9 References and further reading

Path-sums It is hard to pinpoint the earliest appearance of phase polynomials and/or path-sum techniques in the literature, since essentially any computation involving Dirac notation and summations is, in some sense, a path-sum calculation. A notable feature of such a calculation is that it makes it clear that one doesn't need exponential space to compute a single amplitude of a state vector prepared using a polynomial-sized quantum circuit. This insight plays an important role in proving some of the first complexity bounds for quantum computation, as in e.g. (Bernstein and Vazirani, 1997), (Adleman et al., 1997), and (Fortnow and Rogers, 1999).

The "wire labelling" trick that we used for computing phase polynomials in Section 7.1.1 seems to first appear in the work of Dawson et al. (2005), under the name **annotated circuits**. The authors use this technique to show that sum-over-path expressions can be efficiently computed from circuits over a universal gate set (in their case Toffoli+Hadamard), yielding a simple way to relate quantum circuit simulation to counting problems involving boolean polynomials. This was used to give a simplified proof of the inclusion $\mathbf{BQP} \subseteq \mathbf{P^{P\sharp}} \subseteq \mathbf{PSPACE}$ of Bernstein and Vazirani (1997).

Optimisation with phase polynomials Dawson *et al.* only consider phase-polynomial expressions of the form $(-1)^f$ for $f : \mathbb{F}_2^m \to \mathbb{F}_2$ a boolean polynomial. More general expressions that can capture T-like phases (i.e. $e^{i\frac{\pi}{4} \cdot f}$) were used by Amy et al. (2013b) to represent CNOT+T circuits and synthesise exact depth-optimal circuits for low numbers of qubits. The first algorithm to use phase polynomials to efficiently optimise large circuits was T-par (Amy et al., 2014), which used the phase polynomial representation and matroid partitioning to reduce T-depth. The circuit synthesis algorithm we gave in Section 7.2.1 is essentially a simplified version of this technique. Subsequently, phase polynomials were used extensively in circuit optimisation. These essentially fall into two categories: those that are agnostic to the particular values of non-Clifford phases, such as 'phase folding' and parity network optimisation, and those that rely on phases taking particular values such as multiples of $\pi/4$ (or more generally $\pi/2^k$ for $k \geq 2$). We will discuss the latter kinds of optimisations in more detail in Chapter 10. Good overviews of phase-polynomial-based synthesis and optimisation methods can be found in the PhD theses of Amy (2019) and Goubault de Brugière (2020). There are also techniques for constructing a circuit from a phase-polynomial that takes into account hardware constraints on two-qubit interactions, see for instance Meijer-van de Griend and Duncan (2023).

Phase gadgets and Pauli gadgets A unitary $\exp(-i\frac{\alpha}{2}Z \otimes Z)$ is sometimes called an *Ising-type interaction* and is (up to a global basis change) the unitary that is implemented by the natural 2-qubit interaction in ion trap quantum computers, the *Mølmer-Sørensen interaction* Sørensen and Mølmer (1999). The diagrammatic form of this expression was first given in Kissinger and van de Wetering (2019), and it was called a *phase gadget* for the first time in Kissinger and van de Wetering (2020b), where the optimisation routine of Section 7.6 is from. The compilation of an arbitrary circuit to a series of Pauli exponentials followed by a Clifford circuit is described in Litinski (2019), where it is used to argue that a simple set of fault-tolerant operations on the surface code that correspond to Pauli exponentials is sufficient for implementing any computation. For more about how this works, see Chapter 11 The original form of Pauli exponentials as ZX-diagrams was given in Cowtan et al. (2020). The observation that we get a unitary out of a measurement box if we plug in a Z-spider, but get a projector if we plug in an X-spider, can be explained in an abstract way as a duality between measurements and observables that occurs for any pair of strongly complementary observables Gogioso (2019).

Trotter decompositions Trotterisation is named after H. F. Trotter, for the techniques he found in Trotter (1959). Masuo Suzuki studied these decompositions in a series of papers, culminating in the definition of higher-order Trotterisations that are now also called Suzuki-Trotter decompositions Suzuki (1991). The result from Lie theory that any basis of the Lie algebra generates the connected part of its Lie group can for instance be found in (Hall and Hall, 2013, Corollary 3.47). The randomised Trotterisation technique is known as QDRIFT and was developed by Earl Campbell Campbell (2019). The current state-of-the-art higher-order Trotter decompositions are given in Morales et al. (2022), where they find some settings wherein an 8th order decomposition is the best possible.

Other techniques for Hamiltonian simulation Instead of using Suzuki-Trotter decompositions, there are a couple of other techniques that can be used to do Hamiltonian simulation. There is the technique of **linear combination of unitaries** (LCU). This gives an approach to approximately and with some probability implement a linear map M where M is given as a sum of unitaries $M = \sum_j^m \alpha_j V_j$ Berry et al. (2015). This technique works as long as we can implement a controlled version of the V_j, and is efficient when these implementations are efficient and m is not too large. As long as M is close to being unitary itself, this technique succeeds with high probability. To use

this for Hamiltonian simulation we realise that by cutting off the Taylor expansion of $e^{it\sum_j H_j}$ at a certain order, that we get a linear combination of products $H_{j_1} \cdots H_{j_k}$. As long as each of the H_{j_i} is unitary itself (for instance, when it is a Pauli string), we can use the LCU method. This requires a circuit consisting of $O(lt \log(lt/\varepsilon))$ components. While this is better than any of the Trotter techniques asymptotically, the circuits themselves are more complex and require ancillae, so that in practice it might not always be better to use this technique.

Stabiliser decompositions Using the efficiency of simulating Clifford operations in order to boost this to an exponential-time universal quantum circuit simulation scheme was first proposed by Aaronson and Gottesman (2004). However, the idea of grouping together magic states in order to decompose them into fewer terms and get better simulation time is from Bravyi et al. (2016) where they used a simulated annealing algorithm to find a decomposition of 6 $|T\rangle$ states into 7 terms. This was later improved to a 6 term decomposition in Qassim et al. (2021), where they also showed that more complicated arrangements of magic states into the shape of a 'cat' state allow for even better decompositions. Combining stabiliser decomposition methods with ZX-based optimising was introduced by Kissinger and van de Wetering (2022). In the follow-up work (Kissinger et al., 2022) 'cat' states were shown to be related to phase gadgets, giving a nice way to incorporate the better decompositions for these states into the ZX pipeline. This paper also introduced the idea of a 'partial magic state decomposition', where the terms in the decomposition don't necessarily have to be stabiliser themselves, they only have to have fewer non-Clifford resources then in the 'mother' state. Using this idea, they find a decomposition of $|T\rangle^{\otimes 5}$ into 3 terms, each of which contain a single $|T\rangle$, this hence 'effectively' removes 4 $|T\rangle$'s at the cost of 3 terms. This is currently the best known generic decomposition.

Stabiliser extent and weak simulation A stabiliser decomposition allows for exact strong simulation by just enumerating all the different terms of the computation. However, we don't need this exactness when wanting to do sampling, i.e. weak simulation. In Bravyi and Gosset (2016) they introduce the concept of an *approximate* stabiliser decomposition, which only gives the desired state up to some small error. This already greatly improves the simulation time. The approximate stabiliser rank of a state $|\psi\rangle$ can be bounded using the *stabiliser extent*, which was introduced by Bravyi et al. (2019), and is equal to $\sum_i |\lambda_i|$ minimised over all stabiliser decompositions $|\psi\rangle = \sum_i \lambda_i |\phi_i\rangle$. A closely related concept is the *robustness of magic* (Howard
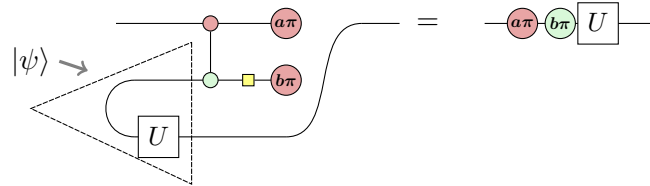
and Campbell, 2017) which is defined for a mixed state ρ as the minimum of $\sum_i |\lambda_i|$ over all decompositions $\rho = \sum_i \lambda_i |\phi_i\rangle\langle\phi_i|$. Robustness of magic directly upper bounds the weak simulation cost of a computation (Howard and Campbell, 2017). Note that these techniques based on approximate stabiliser rank and robustness of magic are instances of **quasiprobabilistic simulation techniques**. Here the 'quasi' means that the probabilities are allowed to be negative, and there also other techniques that belong to this family (Pashayan et al., 2015). In these methods you have a set of states that are 'free' (like Clifford states), and then you write your non-free states as a quasiprobabilistic combination over the free states, with the cost of simulation scaling with the 1-norm of the quasiprobability distribution (the sum of the absolute weights, also called the *negativity* of the distribution).

<h1 style="text-align: center">8</h1>

<h1 style="text-align: center">Measurement-based quantum computation</h1>

Up to this point, whenever we talked about a quantum computation, we were talking about the **circuit model**. In this model of computation you start with a simple state, usually $|0\cdots0\rangle$, which is then inserted into a quantum circuit, i.e. a collection of simple unitaries that combine to make something more complex. At the end we measure the qubits in some fixed basis, and the outcome of these measurements is the outcome of our computation. This model of quantum computation is inspired by classic logic circuits. It is a universal model, meaning that we can in fact present arbitrary quantum computations in this way. This however does not mean that this is always the best way to think of a quantum computation, or the one that matches the most closely to how a real-world quantum computer would function.

There are more ways to implement a computation on a quantum state than just by directly performing a unitary on it. If we have some entangled state, then measurement of one part of the state induces an action on the other part. Depending on the entanglement and the measurement this action could be unitary or non-unitary. Let's take for example a slight modification of the teleportation example of Section 3.3.2:

$$\tag{8.1}$$

In this example of **gate teleportation** we have some maximally entangled state $|\psi\rangle$, which we entangle with the qubit we want to do a computation on using a CNOT. By then measuring some qubits, the resulting state has a unitary U and a Pauli applied to it. The Pauli we apply however is determined by the measurement outcome. This is an inevitable feature of

using measurements to do computations: **measurement non-determinism**. Since we don't control which Pauli gets applied, and it is 'stuck' behind the U, this seems like it would prevent us to control which computations get executed. However, it turns out that for smart choices of measurements and smart choices of U, we can **feed-forward** the Pauli errors. In this way we can make the non-determinism 'deterministic enough' to work for us.

There are various degrees of how much we can use measurements to do computations for us. On one end of the spectrum we don't use measurements at all, performing all computations with unitary gates. Moving a bit further along the spectrum we can decide to implement only certain types of gates using gate teleportation (Eq. (8.1)), or using its closely related cousin of *state injection*. Even further along the spectrum we can let all multi-qubit dynamics be taken care of by measurements, which is the case in *lattice surgery* or *fusion-based computation*. We will have a lot more to say about lattice surgery in Chapter 11. On the extreme end of the measurement-based scale is the **one-way model**, where arguably *all* computation is done by measurements. In the one-way model we start with a large entangled graph state, and then we perform single-qubit measurements in different rotated bases. The choice of which qubit to measure and in what basis it is measured is informed by the computation you want to do and what the previous measurement outcomes were, while the starting graph state (apart from its given size) usually doesn't depend on the particular computation. In this model *all* computation is done by measurements. Such a computation might for instance look like the following:

$$
\begin{array}{c}
\boxed{\frac{\pi}{2} + a\pi} \\
\boxed{b\pi} \\
\boxed{(-1)^a \frac{\pi}{4} + c\pi} \\
\boxed{\pi + d\pi}
\end{array}
\tag{8.2}
$$

Here we start with a four qubit graph state, and then we measure each of these in different bases. The Boolean variables a, b, c and d represent each of the measurement outcomes and are the output of this computation. In this example we have pre-determined **measurement planes**, i.e. whether the measurement takes place in a (rotated) Z or X basis, but some *adaptive* measurement angles. I.e. while the top qubit is always measured in a basis with angle $\frac{\pi}{2}$, the third qubit has a measurement angle $(-1)^a \frac{\pi}{4}$, meaning that whether we measure it in the angle $\frac{\pi}{4}$ or $-\frac{\pi}{4}$ depends on the measurement outcome of the qubit a. This means there is an order fixed on the

measurements of these qubits, with qubit a having to be measured before qubit c.

Collectively, we call models of quantum computation using measurements **measurement-based quantum computing** (MBQC). While the circuit model is inspired by classical circuits, MBQC has no classical counterpart. There are various reasons why we might want to consider MBQC as opposed to quantum circuits:

- Certain unitaries might be tricky to implement, requiring trial and error. By using a technique like gate injection we can prepare resource states 'offline', and have them ready for when they are needed.
- Replacing unitaries for measurements can sometimes help us trade 'time' for 'space', allowing us to parallelise more computation at the cost of requiring more qubits.
- For certain types of hardware, it makes less sense to implement operations using a series of simple unitaries, instead preferring to do more measurements so that your qubits are shorter-lived. For instance, if the qubits are photons, we can entangle them by doing a fusion measurement and then doing simple rotations and measurements. This is preferable since it is hard to store photons for a longer period of time, so here the trade-off between time and space needs to be more in the space direction.
- When we consider fault-tolerant models of computation like in Chapter 11, we will be doing constant measurements anyway in order to protect our data against errors, so why not use measurements to enact operations on our data at the same time.

In this chapter we will look in detail at the one-way model. It turns out that graph-like diagrams can naturally be understood as measurement patterns. In this way we can see that we can indeed do universal quantum computation in the one-way model. We will also see more broadly how we can determine when a computation can be made deterministic. We do this by studying *generalised flow*, or **gflow** for short, which tells us how to feedforward Pauli errors. It turns out that any deterministic measurement pattern with gflow can be efficiently transformed back into a quantum circuit. This gives us a triangle of translations, from quantum circuits, to ZX-diagrams, to measurement patterns, and back to quantum circuits. Because we can rewrite ZX-diagrams, this gives us a unified way to think about circuit optimisation and measurement-pattern optimisation.

8.1 Measurement fragments

If we were to formally specify what a computation in the quantum circuit model is, it would be something like: prepare n qubits in the $|0\rangle$ state, apply a series of unitary gates $G_1, \ldots, G_k$, then measure all n qubits in the computational basis, giving some bit string $\vec{x}$, and finally post-process this with some output function $p(\vec{x})$ giving the final outcome.

In contrast, we might formally specify a one-way computation as follows.

Definition 8.1.1. A one-way model computation consists of the following data:

- A simple undirected graph $G = (V, E)$ specifying the graph state $|G\rangle$ we start the computation with which has m vertices,
- A *measurement order* $f : \{1, \ldots, m\} \to V$ where f is bijective,
- A *measurement-plane function* $\lambda : V \to \{XY, XZ, YZ\}$ that tells us the measurement plane of every qubit,
- for every vertex $f(k)$ a *measurement-angle function* $\alpha_k : 2^{\{1, \ldots, k-1\}} \to \mathbb{R}$ that determines the measurement angle of $f(k)$, which may depend on all the previous measurement outcomes,
- An *output function* $p : 2^m \to S$ that classically post-processes all the m measurement outcomes into a final outcome in some set S (for instance $S = \mathbb{F}_2$ if we care about a binary true/false output).

To execute it, we prepare the graph state $|G\rangle$, and measure the qubits one by one in the order $f(1), f(2), \ldots$ giving outcomes $x_1, x_2, \ldots$. They are measured in the plane $\lambda(f(k))$ with angle $\alpha_k(x_1, \ldots, x_{k-1})$. The final outcome of the computation is $p(\vec{x})$.

Note that we will talk about what these 'measurement planes' are in more detail later, for now you can think of the XY plane as corresponding to Z-spiders and the YZ plane as corresponding to X spiders.

Remark 8.1.2. In Definition 8.1.1 we are not allowing the measurement-plane to depend on previous measurement outcomes. This is just for our convenience, and because this model is already universal.

Definition 8.1.1 specifies a complete calculation from beginning to end where all qubits end up measured. If we represent it as a ZX-diagram it hence corresponds to a scalar ZX-diagram, i.e. a diagram with no inputs or outputs. This is no different from the circuit model, where for a full computation we also have to fix an input state and end by measuring all qubits. However, just as it is useful in the circuit model to think of just a

part of a computation which has some input and output qubits, it is also useful to think of 'fragments' of a measurement-based computation that can be composed together to create more complicated computations. These fragments have inputs and outputs just like a quantum circuit would. For instance, a **measurement fragment** that implements a CNOT gate is given by the following diagram:

$$(8.3)$$

Here the top qubit is both an input and an output and hence isn't measured. The third qubit is an input, but is not an output and hence is measured. The fourth qubit is an output and hence also isn't measured. The second qubit is 'internal' and is hence prepared and measured as part of this fragment. The fragment (8.3) implements a CNOT gate up to a known Pauli error that depends on the measurement outcomes a and b. We can calculate this Pauli error by simplifying the diagram:

$$(8.4)$$
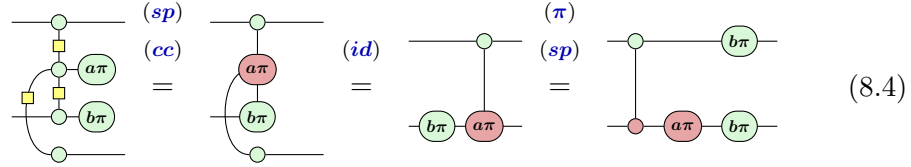
Hence we end up with the circuit $(Z^b \otimes (Z^b X^a)) \circ \mathrm{CNOT}$. Since the error is a known Pauli, later measurements can be adapted to absorb these errors so that the overall effect of this measurement fragment is the application of a CNOT gate. We call this process of pushing Pauli errors outside of the pattern into the future **feed-forward**.

Let's give a bit more formal definition of a measurement fragment. For that we need the concept of an open graph.

Definition 8.1.3. An **open graph** (G, I, O) is a graph $G = (V, E)$ together with a list of **inputs** I and **outputs** O. These lists consists of vertices of G, where repetition of vertices is allowed, and the order is relevant.

Definition 8.1.4. A **stand-alone measurement fragment** consists of

- an open graph (G, I, O) (recall Definition 8.1.3) with $G = (V, E)$ and where m is the number of non-output vertices $|V \backslash O|$,
- a measurement order $f : \{1, \ldots, m\} \to V \backslash O$ where f is bijective,

- A measurement-plane function $\lambda : V \backslash O \to \{XY, XZ, YZ\}$ that tells us the measurement plane of every non-output vertex, and

- for every non-output vertex $f(k)$ a measurement-angle function $\alpha_k : 2^{\{1,\dots,k-1\}} \to \mathbb{R}$ that determines the measurement angle of $f(k)$ depending on all the previous measurement outcomes.

The reason we call this a 'stand-alone' fragment is because this definition doesn't deal with fed-forward Pauli errors. To prevent making this too complicated for now, we postpone that discussion until later.

This definition is maybe a lot to take in, but the corresponding ZX-diagrammatic picture should make things more clear. We construct the ZX-diagram corresponding to a measurement fragment as follows. For every vertex in V we add a Z-spider, and for every edge in E we add a Hadamard edge between them. Then we add an input wire on a spider if it is in I and an output wire if it is in O. This so far gives us a graph-like diagram and an open graph (where the graph-like diagram is phase-free). However, now we also add an *additional* output wire to every non-output spider, to get a diagram that looks something like this:

$$\tag{8.5}$$

Now on each of those additional output wires, we are going to plug in a measurement effect, corresponding to the measurement plane and angle of that vertex:

$$XY \rightsquigarrow \quad \boxed{\alpha_k + b_k\pi} \qquad XZ \rightsquigarrow \boxed{\tfrac{\pi}{2}}\!\!\boxed{\alpha_k + b_k\pi} \qquad YZ \rightsquigarrow \boxed{\alpha_k + b_k\pi} \tag{8.6}$$

Here the α_k denotes the measurement angle of the pattern, and the b_k is a Boolean variable denoting the outcome of the measurement. Note that α_k is actually a function of the variables $b_1, \dots, b_{k-1}$ representing the outcomes of the previous measurements. These names XY, XZ and YZ come from the fact that these effects lie on those corresponding planes of the Bloch sphere.
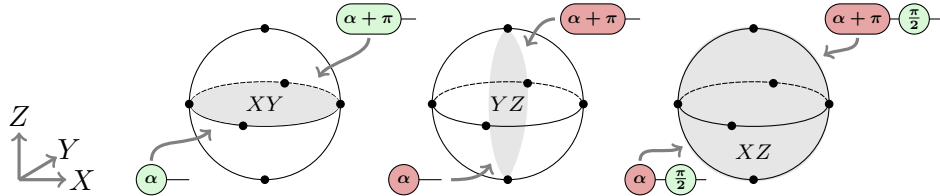
We see then that if in the fragment (8.5) we choose XY for the measurement-plane of both qubits and we simply set the measurement angles to 0 that we get precisely the measurement fragment (8.3) implementing the CNOT gate.

We can also construct a fragment that implements an arbitrary single-qubit unitary. Recall that we can use the Euler decomposition of such a unitary to write it as a sequence of a Z phase gate, and X phase gate, and then another Z phase gate: $-\boxed{\text{U}}- \; = \; -(\alpha)-(\beta)-(\gamma)-$. Let's try to write down a measurement fragment that implements this, by putting each of the phases α, β and γ into the measurement angle of a qubit:

$$\tag{8.7}$$

This is not exactly correct. There are a couple of things we need to fix. For one thing, there is a Pauli error at the end: an X gate depending on $b + d$ and a Z error depending on $a + c$. We will learn how to deal with such errors later. But there is also a problem with the phases. We want the phases to be α, β and γ, but we see that depending on the measurement outcomes we might have actually gotten $-\beta$ or $-\gamma$. This is where the condition of Definition 8.1.4 comes in that measurement angles are allowed to depend on previous measurement outcomes. To make this fragment do what we want it to do, we need to make it **adaptive**. We first measure the qubit labelled by the measurement outcome a. If $a = 0$, we measure the qubit b in the angle β, which gives what we want. Otherwise we measure it in the angle $-\beta$. Hence, the measurement angle is $(-1)^a \beta$. Rerunning the calculation (8.7) with β replaced by $(-1)^a \beta$, we see that we get the final phase of $(-1)^a (-1)^a \beta = \beta$, so that now the final angle no longer depends on the measurement outcome! We can do the same with γ, letting the measurement angle instead be $(-1)^b \gamma$. With these changes the measurement fragment becomes:

$$\tag{8.8}$$

Apart from the Pauli error at the end that we still need to deal with, this

implements the unitary we want. Note that in the fragment implementing the CNOT (8.3), the measurement angles didn't depend on each other, so that the order of measuring the qubits wasn't important. In (8.8) however, the measurement angles of some of the qubits depend on the measurement outcomes of other qubits, so that those qubits have to be measured first. This gives us a 'direction of time' on the pattern. It makes sense that we need to have some kind of adaptivity or dependency in our calculation. If we could do universal computation without such dependencies, then we could measure all qubits simultaneously, and do computations in constant time!

So now we have a measurement fragment implementing a CNOT and another one implementing arbitrary single-qubit unitaries. Hence, if we can combine these then we can implement arbitrary unitaries. However, now we get to why we called these measurement fragments 'stand-alone': there is the problem of the Pauli errors. Let's see what we can do with them.

If a fragment with a Pauli error occurs at the end of our computation this is easy enough to deal with. At the end, we measure all the qubits in the computational basis anyway. This means that these errors just change how we should interpret the measurement outcomes:
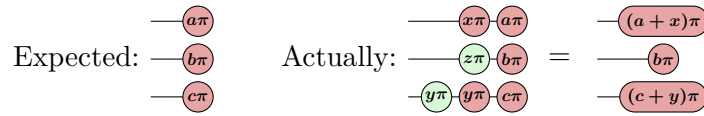
So if we were feedforwarding an X error, i.e. $x = 1$ above, then this changes how we should interpret the measurement outcome we got. If we observe the measurement outcome a' then this is actually $a' = a \oplus x$. So to get the 'actual' measurement outcome a, we just take $a' \oplus x = a \oplus x \oplus x = a$. A Z error ($z = 1$) doesn't change the measurement outcome when we measure in the computational basis, so we can ignore those in this setting. A Y error, i.e. where both a Z error and an X error occurred, we can treat the same as an X error here.

In general though, the measurement fragment will happen somewhere in the middle of the computation so that the errors will not change the final measurement outcomes, but change how we should run the next fragments in the sequence. For instance, let's see what would have happened if a Pauli error had happened on the input qubit prior to executing the measurement

fragment (8.8):

$$(8.9)$$

We see that we can push these errors to the output of the fragment, at the cost of changing the angles in the unitary to potentially have minus signs, and additionally changing our output Pauli error. But these Pauli errors we have on the input arise from measurements we have done before, so we *know* what this error is. This means we can decide the measurement angle of our measured qubits based on this input Pauli error, in the same way we also change it based on the measurement outcomes of the qubits internal to the fragment. The full pattern that can also deal with incoming errors then becomes:

$$(8.10)$$

This still gives us a Pauli error at the end, but this is fine since we now know how to deal with this.

In the definition of a stand-alone measurement fragment Definition 8.1.4 we only allowed the measurement angles to depend on previous measurement outcomes, and not on Pauli errors being fed-forward into it. Now that we know that we need this kind of adaptivity, we can give a definition of the kind of fragment we need.

Definition 8.1.5. A **measurement fragment** consists of

- an open graph (G, I, O) (recall Definition 8.1.3) with $G = (V, E)$ and where m is the number of non-output vertices $|V \backslash O|$, and $n_i = |I|$ is the number of input qubits and $n_o = |O|$ is the number of output qubits,
- a measurement order $f : \{1, \ldots, m\} \to V \backslash O$ where f is bijective,
- A measurement-plane function $\lambda : V \backslash O \to \{XY, XZ, YZ\}$ that tells us the measurement plane of every non-output vertex,

- for every non-output vertex $f(k)$ a measurement-angle function $\alpha_k :$ $2^{n_i} \times 2^{n_i} \times 2^{\{1,\dots,k-1\}} \to \mathbb{R}$ that determines the measurement angle $\alpha_k(\vec{z}, \vec{x}, \vec{a})$ of $f(k)$ depending on all the fed-forward Z errors $\vec{z}$, X errors $\vec{x}$ and previous measurement outcomes $\vec{a}$, and
- Boolean functions $f_Z, f_X : 2^{n_i} \times 2^{n_i} \times 2^m \to 2^{n_o}$ that determine what the Pauli Z errors $f_Z(\vec{z}, \vec{x}, \vec{a})$ and X errors $f_X(\vec{z}, \vec{x}, \vec{a})$ are on each output qubit that we will feed forward, based on the Z errors $\vec{z}$ and X errors $\vec{x}$ that were fed-forward into the pattern and the measurement outcomes $\vec{a}$ of the qubits in the pattern.

Note that we take it as part of the definition of a measurement fragment to specify what the output errors are. By specifying what we consider 'errors' we are at the same time specifying what our desired outcome is, and hence what the linear map we are trying to implement is. We also need this error information as it needs to be fed-forward into the fragments following it.

Example 8.1.6. Translating the diagram (8.10) into the language of Definition 8.1.5, we see that G is a chain of five vertices, let's call them a, b, c, d, o to match with the variables in (8.10). Then $I = \{a\}$ and $O = \{o\}$. We have $m = 4$, $n_i = n_o = 1$. The measurement order f just goes down the chain: $f(1) = a, f(2) = b, f(3) = c, f(4) = d$ (although actually qubit d can be measured whenever as nothing else in the fragment depends on this outcome). The measurement-plane λ is XY for every measured qubit. The measurement-angle functions are then

$$f_1(z, x) = (-1)^x \alpha,$$
$$f_2(z, x, a) = (-1)^{a+z} \beta,$$
$$f_3(z, x, a, b) = (-1)^{b+x} \gamma,$$
$$f_4(z, x, a, b, c) = 0$$

where here α, β and γ are pre-determined phases corresponding to the unitary we wish to implement. Finally, the Pauli error functions are $f_Z(z, x, a, b, c, d) = a + c + z$ and $f_X(z, x, a, b, c, d) = b + d + x$.

Exercise 8.1. Do the same type of analysis of the Pauli errors as was done for (8.9), but for the CNOT measurement fragment (8.3). Do the measurement angles in this pattern have to depend on the incoming Pauli errors? What are the resulting Pauli errors on the output? Translate all this information into the form of a measurement fragment as in Definition 8.1.5.

Now that our definition of a measurement fragment can specify how incoming Pauli errors should change the measurement angles, and how the resulting

errors should be fed-forward, we can actually compose these fragments together. We will not formally define how to do so, as this is quite tedious, but one can visualise it as just connecting the associated ZX-diagrams together and pushing the internal Pauli errors to the end.

Remark 8.1.7. The example fragments we have considered so far, (8.10) and (8.3), implement a linear map that is independent of the measurement outcomes in the pattern (up to a known Pauli error that is being fed-forward). We call such a fragment **deterministic**, because if we zoom out and consider the fragment as a single unit, it just implements a fixed unitary. Our definition of measurement fragment doesn't require the fragments to be deterministic, but in practice we will restrict ourselves to deterministic ones. We will have a lot more to say about determinism in the one-way model in Section 8.2.

8.1.1 Universal resources

We have described a measurement fragment to implement a CNOT gate and one that can implement an arbitrary single-qubit unitary. As CNOT gates and single-qubit unitaries form a universal gate set, we see then that we can construct a measurement fragment for any unitary we would want to implement. Hurray! Note however that the resulting graph state we need will depend on the specific unitary we want to implement. There is then still some 'information' about the computation that is present in the starting resource state. This then raises an interesting question: can we implement arbitrary unitaries using a measurement fragment that has a *fixed* graph state? We need to be a bit careful about how to formally state this question, because there are unitaries that are really hard to implement, and so we would expect them to require a large resource state to prepare (put simply: if we only have k measurements, we could represent at most a k-dimensional subset of all unitaries, but the space of all unitaries scales exponentially with the number of qubits). We hence shouldn't consider a fixed resource, but rather a family of resource state that is allowed to scale with the cost of the unitary. A more formal version of the question would hence be something like:

Does there exist a family of open graphs G_k where G_k consists of $\mathrm{poly}(k)$ vertices, such that for any n-qubit unitary U with m gates, we can find a measurement pattern implementing U which uses the open graph G_k where $k = \mathrm{poly}(n, m)$?

Maybe perhaps surprisingly, the answer to this is yes! There exist **universal resource states** which allow you to implement arbitrary quantum computations up to some size. A particularly nice example of a family of universal

resource states are the 2D **cluster states**:
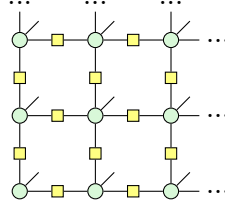
I.e. the graph is just a big square grid. The trick to seeing why this is universal, is that we can use measurements in the YZ plane to strategically cut holes into the grid:
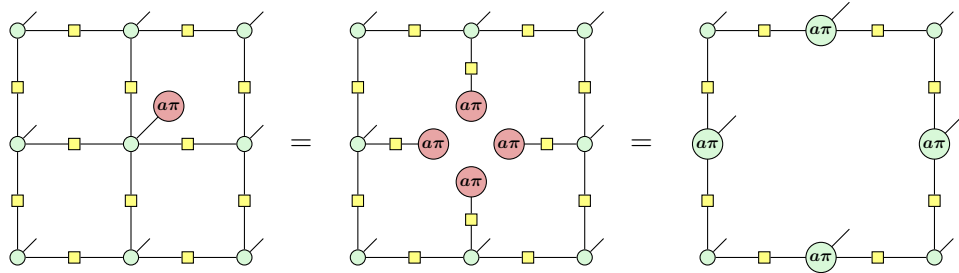
This allows us to shape the grid into whatever pattern we need it to be to perform the computation we want.

8.2 Determinism and gflow

We have now seen that we can do universal computation using the one-way model. An important part in getting there was knowing how to deal with the wrong measurement outcomes by finding ways to feed them forward. So far we have done this essentially by trial and error: we construct a simple fragment, observe that we get the wrong phases sometimes, and then change them to correct for this. In this section we will find a more systematic way to think about correcting errors and when we can see that a measurement fragment can be made deterministic.

But first: let's look at the connection between measurement fragments and graph-like diagrams.

8.2.1 Graph-like ZX-diagrams as measurement fragments

It turns out that if we take the ZX representation of a measurement pattern, ignore the adaptivity needed to correct for 'wrong' measurement outcomes, and fuse some spiders, that we get a graph-like diagram. For instance, doing

this for the fragment (8.7), we get:

$$(8.11)$$

This graph-like diagram is then a representation of the 'post-selected' measurement fragment, where all the measurement outcomes where what we wanted them to be.

It turns out the converse is also true: any graph-like diagram can be seen as a post-selected measurement fragment, just by unfusing some phases and introducing some identities:

$$(8.12)$$

If we were to actually try to run this as a measurement fragment, we would of course usually get some of the 'wrong' measurement outcomes. So this fragment is only useful if we know how to deal with all measurement outcomes, so that we can make the pattern deterministic. Can we do this somehow?

This is not just a niche question we are asking because we like diagrams. We can write any quantum circuit as a graph-like diagram, so if we could do this then this would give us an easy way to translate a circuit into a deterministic measurement fragment.

It turns out that it is *in general* not possible to turn graph-like diagrams into deterministic measurement fragments. This is for the simple reason that we can write down ZX-diagrams that are not isometries. For instance, consider the following graph-like diagram:

$$(8.13)$$

By doing some colour changing and identity removals, we see that this is just a 2-input, 2-output Z-spider, which is a projection and is not unitary. If we could deterministically perform such a projection, we could do non-unitary quantum computation! Or even more simply, consider the graph-like

diagram ──○ . If we could implement this ZX-diagram deterministically, we would be doing post-selection.
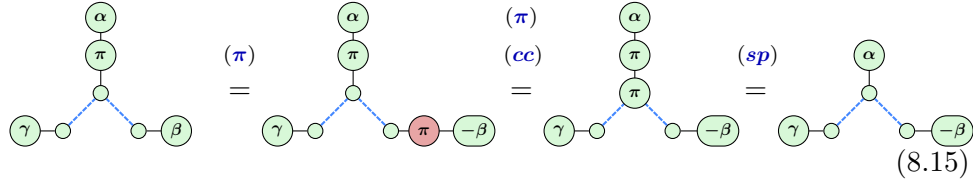
So it is not always possible to turn a graph-like diagram into a deterministic measurement fragment. But can we at least somehow determine *when* we can do so? As it turns out: *yes*. For a particularly strong type of deterministic measurement pattern there is a one-to-one correspondence to graph-like ZX-diagrams with a property called *gflow*.

8.2.2 The measurement correction game

To understand when we can implement a correction strategy for measurement errors on a graph-like ZX-diagram, let's consider a small toy diagram to see what we can do:

$$(8.14)$$

This is of course not a realistic diagram that we would expect to encounter when doing MBQC, but it will serve to demonstrate the principle of correction. Here our goal is to get rid of the π phase. We can do this by changing the spider with the β phase:

$$(8.15)$$

Hence, as long as we measure the qubit with phase β after the one with α, an error at α can be corrected by changing the measurement angle of the β qubit to be $-\beta$. Here we could have also chosen to change γ in order to correct the error at α.

Finding the right correction was easy here, because the β spider was only connected to α and nothing else. Let's consider a slightly more complicated diagram:

$$(8.16)$$

In this case, changing the phase of β does still remove the error at α, but it

also changes the phases at the other spiders:

$$(8.17)$$

Now if we can run the pattern such that the qubits with the γ and δ phase can be measured after the α qubit, then this is fine, since we can still decide to measure them with a different phase to correct for the additional π phase they have. But if we suppose that we have already measured the qubit with the γ phase, then this would not be possible. In that case however, we see that we can apply the same 'π-pushing' trick with δ to remove the additional phase γ got:

$$(8.18)$$

This gave us an additional π phase at β as well, but that is fine as we are already considering changing the phase at β. Hence, to correct the error at α, we can decide to change the phase at β and δ, and this leaves the phase at γ invariant.

Let's try to formalise what is happening here. We see that the trick of pushing out an $X(\pi)$ phase affects the phases on all the neighbours of this vertex, and that if we affect the phase of a vertex twice, that this then cancels out the effect. The qubits will be measured in some order, so we must make sure that whatever qubits we affect the phase of must be measured *after* the qubit we are trying to correct an error at.

We can capture this behaviour in a little game: you are given an open graph with vertices V where every vertex is labelled with either the presence or absence of an error (a 1 or 0). You are allowed to do the operation $\mathbf{flip}_v$ for a vertex $v \in V$, which flips the value of the error on all the neighbours of v (but not at v itself). The goal is to move all the 1's (the errors) to the output vertices. For example:

$$(8.19)$$

For some open graphs and configurations of errors, this task might be impossible. For example, there is no solution for the following graph:
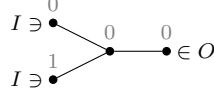
Which makes sense, because if we were to try to build a one-way model computation on this open graph, it would map two qubits to one qubit, so there is no way we could do such a computation deterministically.

The question then is, for which graphs can you always win this game, regardless of where the errors occur?

We can see that a winning strategy is the following: first, find an ordering $\prec$ of vertices which give a direction of 'time' going from inputs to outputs, and find for each non-output vertex u, a **correction set** $g(u) \subset V$ of vertices in the future of u (with respect to the ordering $\prec$) such that applying $\mathbf{flip}_v$ for all $v \in g(u)$ flips the bit on u, and potentially also flips some other bits, as long as those bits all belong to vertices that are in the future of u (again, with respect to $\prec$). Now, for any instance of the game we can apply the following strategy: find a minimal vertex u (w.r.t. $\prec$) which has an error. Then apply $\mathbf{flip}_v$ to all $v \in g(u)$. This removes the error on u, and only introduces new errors on vertices in the future of u. Then find a new minimal vertex with an error and repeat until all the errors are moved all the way to the end at the outputs.

This game describes the property of $g(u)$ in terms of the **flip** operation, but we can actually make the description a bit simpler and more natural to graph-theory language. Namely, we see that a vertex w get its bit flipped by applying $\mathbf{flip}_v$ for all $v \in g(u)$ precisely when it is connected to an *odd* amount of vertices in $g(u)$. This is because if it is connected an even amount of times to vertices in $g(u)$, then it gets its bit flipped twice, so that nothing happens. Let's capture this in a definition.

Definition 8.2.1. Let $G = (V, E)$ be an (open) graph, and $A \subseteq V$ some subset of its vertices. We define the **odd neighbourhood** of A in G as $\mathrm{Odd}_G(A) := \{\, w \in V \mid |N_G(w) \cap A| \bmod 2 = 1 \,\}$. Here $N_G(w)$ denotes the regular neighbourhood of a vertex w, i.e. its set of neighbours.

Using odd neighbourhoods we can rephrase the condition we needed on the correction sets $g(u)$: we need $u \in \mathrm{Odd}_G(g(u))$ (applying **flip** for all vertices in $g(u)$ results in a flip at u) and if $v \in \mathrm{Odd}_G(g(u))$ then $u \prec v$ (if v gets flipped by the corrections for u, then v needs to be in the future of u).

This is in fact exactly what we require, and allows us to give the definition of the central concept of this chapter: gflow.

Definition 8.2.2. Let $G = (V, E, I, O)$ be an open graph. A **gflow** $(\prec, g)$ on G consists of a partial order $\prec$ on the vertices of G, and a **correction set function** $g : V \backslash O \to \wp(V \backslash I)$ that associates to every non-output vertex $v \in V \backslash O$ a subset of non-input vertices $g(v) \subseteq V \backslash I$, such that:

1. The inputs are minimal in the partial order $\prec$ (meaning that if $v \prec i$ for some input i, then $v = i$) and conversely the outputs are maximal;
2. For every $u \in V \backslash O$, we have that $u \prec v$ and $u \neq v$ for every $v \in g(u)$, i.e. all the elements in the correction set are in the future of the vertex being corrected;
3. For every $u \in V \backslash O$ we have $u \in \mathrm{Odd}_G(g(u))$, i.e. applying a correction at all the vertices in the correction set of u, results in a correction at u;
4. For every $u \in V \backslash O$, if $v \in \mathrm{Odd}_G(g(u))$, then $u \prec v$. That is, if v is a vertex that ends up with a correction from the correction set of u, then v is in the future of u.

We say an open graph **has gflow** when at least one gflow can be defined on it.

With what we discussed above we see that if an open graph has gflow, that this gives us a winning strategy for the error correction game.

Remark 8.2.3. We have here only shown that the existence of gflow gives us *a* winning strategy for the error correction game, but not that there might also be other strategies. It turns out that the open graph of a deterministic measurement pattern has gflow if and only if it has a particularly well-behaved kind of determinism. See the References of this chapter for more.

Exercise 8.2. In the definition above we defined $\mathrm{Odd}_G(A)$ by checking all vertices in G. However, there is also an 'intrinsic' definition, given by $\mathrm{Odd}_G(A) = \Delta_{v \in A} N_G(v)$. Here Δ denotes the **symmetric difference** operation of sets, where for example $\{1, 2, 3\} \ \Delta \ \{2, 3, 4\} = \{1, 4\}$. Prove that this definition is equivalent to the one in Definition 8.2.1.

Exercise 8.3. Show that for any two sets of vertices $A, B \subseteq V$ of $G = (V, E)$ that $\mathrm{Odd}_G(A \ \Delta \ B) = \mathrm{Odd}_G(A) \ \Delta \ \mathrm{Odd}_g(B)$.

8.2.3 Diagrams with gflow are deterministic measurement patterns

We can use gflow to describe how to transform a graph-like ZX-diagram into a deterministic measurement pattern.

Let's assume we have a graph-like ZX-diagram D such that its corresponding open graph G has a gflow $(\prec, g)$. We see from Definition 8.1.5 that to describe a measurement fragment, we need to supply a number of data to fully define it. The first parts are easy enough to specify:

- We need to specify an open graph. For this we will just take the open graph we already have: G.
- A measurement order. For this we will take any order that is compatible with the partial order $\prec$. That is: we take a minimal vertex v in the order (which does not have any vertices earlier in the order) and assign it to be measured first. We then find a new minimal vertex in the set $V \backslash \{v\}$ and assign it to be measured second. We repeat this until all vertices in $V \backslash O$ have been assigned a measurement order.
- A measurement plane function. Here we just assign every measured vertex to be measured in the XY plane (we will discuss this choice in more detail in Section 8.5).

But now comes the hard part: we need to describe the measurement angles, and more importantly, how they are affected by the measurement outcomes. In practice, what this means is that we have a standard measurement angle α that comes from the angle we had in the ZX-diagram D. But then this α gains either a $+\pi$ phase or a negation depending on whether it appears in any correction sets, or odd neighbourhoods of correction sets of measured qubits that require a correction.

Let's write α_u for the phase that the spider u has in the original diagram D and a_u for the measurement outcome when the spider u is measured. Hence, if all the measurement outcomes a_u are 0, then the measurement angles we want to measure every qubit in is just α_u. We will change these α_u to functions α'_u that may depend on the a_v of qubits v that have been measured before u.

Let's first consider an input $u \in I$, which has to deal with a fed-forward Pauli X error, but no other corrections from inside the pattern:
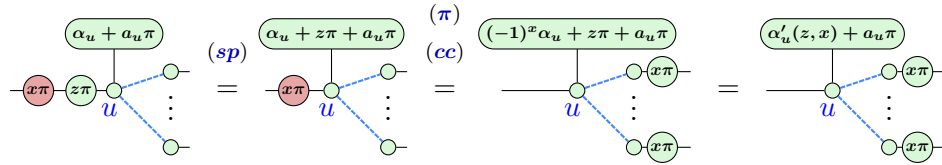
We see that we should change our initial choice of measurement α_u to a function $\alpha'_u(z, x) = (-1)^x \alpha + z\pi$ in order to account for the error being fed-forward into the spider. This doesn't fully capture the incoming Pauli error though, as we see that the X error has to be further fed-forward into

the neighbours of u. All this does is change the measurement angle of that neighbour by $x\pi$, so we will ignore this detail.

While this deals with the incoming Pauli error, it does not deal with the measurement error a_v that it produces itself. To correct this we need the gflow condition. This procedure is the same regardless of whether the spider is an input or not, so let's just take $u \in V \backslash O$ to be any measured qubit now. We then have a set of correction qubits $g(u) = \{v_1, \ldots, v_k\}$ and a set of 'corrected' qubits $\mathrm{Odd}_G(g(u)) = \{u, w_1, \ldots, w_l\}$. We then see that by changing the measurement angles at the correction and corrected qubits, we can remove the measurement error at u:
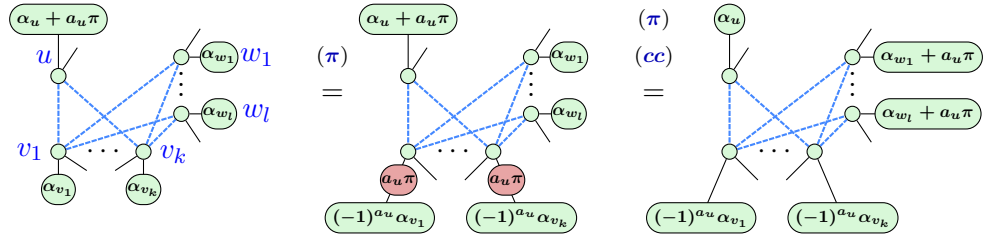
Here for simplicity we have assumed that each v_i and w_j is not an output, as we will deal with that case later. We have given every qubit in this diagram an additional output wire to denote that it can also be connected to some other spiders. The last application of (π) does not affect these additional qubits, precisely because they are not in the odd neighbourhood of the correction set. So for every $v \in g(u)$ we set $\alpha'_v(a_u) = (-1)^{a_u}\alpha_v$ and for every $w \in \mathrm{Odd}_G(g(u))$ with $w \neq u$ we set $\alpha'_w(a_u) = \alpha_w + a_u\pi$.

In general, a spider could of course be part of multiple correction sets and odd neighbourhoods of correction sets. It will be useful to give these a name:

$$S_1(v) := \{u \in V \backslash O \,|\, v \in g(u)\}$$
$$S_2(v) := \{u \in V \backslash O \,|\, v \in \mathrm{Odd}_G(g(u))\backslash\{u\}\}$$

Then the actual measurement angle, as a function of the previous measurement outcomes, is

$$\alpha'_v(\{\, a_u \,|\, u \prec v \,\}) \;=\; (-1)^{\sum_{u \in S_1(v)} a_u}\alpha_v + \left(\sum_{u \in S_2(v)} a_u\right)\pi.$$

We are now almost done! The above tells us how to change measurement angles to correct errors when the correction sets $g(u)$ and corrected sets $\mathrm{Odd}_G(g(u))$ only contain non-outputs. When instead an output qubit o is in such a set, then we don't change the measurement angle of o (which we can't do since o is not measured), but we change the Pauli error that is

being fed-forward. If $o \in g(u)$, then we get a Pauli X error of $a_u\pi$, and if $o \in \mathrm{Odd}_G(g(u))$, then we get a Pauli Z error $a_u\pi$. In general then, the output Pauli that gets forwarded at output o is $(z, x) = (\sum_{u \in S_2(o)} a_u, \sum_{u \in S_1(v)} a_u)$.

We have now specified what all the measurement angles of the qubits should be, based on the fed-forward errors and the internal measurement outcomes, and we have specified what the outgoing fed-forward errors will be based on the measurement outcomes. This then indeed gives us a measurement pattern. By how we constructed the pattern, we see that it implements the ZX-diagram we started out with regardless of the internal measurement outcomes (up to the fed-forward Pauli error at the end). We have then proved the following result.

Theorem 8.2.4. Let D be a graph-like diagram, and $G = (V, E, I, O)$ its corresponding open graph consisting of a vertex for every spider and edge for every Hadamard edge, and suppose that G has gflow. Then we can efficiently construct a deterministic measurement fragment from D and the gflow which has the same underlying open graph and implements the same linear map as D.

Remark 8.2.5. The trick we applied here, introducing Pauli X's to cancel a Pauli Z at a different location, might have seemed a bit familiar. This is because it is actually based on the *stabilisers* of the underlying graph state (see Example 6.3.8). Applying the **flip**$_v$ operation is like introducing the graph-state stabiliser $\mathcal{S}_v$ consisting of an X at v and a Z at all its neighbours. The correction set $g(u)$ then says that to correct the error at u, we need to introduce the stabiliser $\prod_{v \in g(u)} \mathcal{S}_v$.

8.2.4 From circuits to measurement patterns

We started out this section with the goal of understanding how we can systematically create measurement fragments that are deterministic. We have seen that we have one powerful tool for ensuring determinism: the existence of a gflow. What we have however not yet done is finding a good *source* of measurement fragments that have gflow.

As it turns out, if we take a unitary circuit consisting of Clifford and phase gates, and then turn it into a graph-like diagram by fusing all spiders and colour changing to Z-spiders as described in Proposition 5.1.8, the resulting diagram has gflow.

Proposition 8.2.6. Let D be the diagram of some quantum circuit, and

D' the graph-like diagram produced from D using Proposition 5.1.8. Then D' has a gflow.

Proof   The idea will be to set $g(u) = \{v\}$ where v is the spider directly to the right of u on the same qubit line, and for the partial order to encode 'is that spider to the left of me?'

For every spider v of the circuit D we associate a number q_v specifying on which 'qubit-line' it appears. We also associate a 'column-number' $c_v \geq 0$ specifying how 'deep' in the circuit it appears. Hence, each spider has 'xy coordinates' (c_v, q_v):
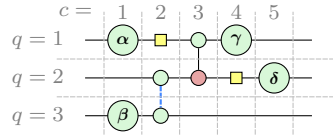
Suppose that v and w are connected in D, $v \sim w$. If they are on the same qubit, so $q_v = q_w$, then necessarily $r_v \neq r_w$, since they can't share the same space. Conversely, if they are on different qubits, $q_v \neq q_w$, then they must be part of a CZ or CNOT gate, and hence $r_v = r_w$.

In D', every spider arises from fusing together adjacent spiders on the same qubit line from the original diagram (apart from the additional identity spiders introduced in the last step, which we will ignore for now):
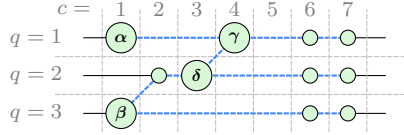
For a spider v in D' we can thus associate two numbers s_v, and t_v, where s_v is the lowest column-number of a spider fused into v, and t_v the highest. For instance, for the second spider on the second qubit, we have $s = 3$ and $t = 5$. Spider fusion in D only happens for spiders on the same qubit-line, and hence v also inherits a unique q_v from all the spiders that got fused into it.

An identity spider v created by the last step of the translation, must be attached to either an input or an output. If it is attached to an input, we set $s_v = t_v = 0$. If it is attached to an output we set $s_v = t_v = +\infty$ (or any value bigger than the largest value of t_v in the diagram). For this proof we can treat these spiders the same way as the others.

We define a partial order on D' as follows: $v \prec w$ if and only if $v = w$ or $t_v < t_w$. It is straightforward to check that this is indeed a partial order. Now for any u in D' that is not an output, set $g(u) = \{v\}$ where v is the

unique neighbour to the right of u on the same qubit-line. We claim $(g, \prec)$ is a gflow.

By construction $u \in \mathrm{Odd}(g(u)) = N(v)$. We need $u \prec v$. The spiders of D that got fused into the same spider in D' must have all been adjacent. Hence, for the distinct spiders u and v in D' on the same qubit we must have $t_v < s_w$, as they couldn't have overlap in the spiders that got merged into them. Hence, $t_u < s_v < t_v$ so that $u \prec v$. Suppose $w \in \mathrm{Odd}(g(u)) = N(v)$ so that $w \sim v$. We need to show that $u \prec w$. If $u = w$ this is trivial so suppose $v \neq w$. First suppose that $q_w = q_v$ (which is also equal to q_u). We know v has a maximum of two neighbours on the same qubit-line (one to the left, and one to the right), and since the one to the left is u, w must be to the right, so $u \prec v \prec w$. If instead $q_v \neq q_w$ then their connection must have arisen from some CNOT or CZ gate in D, and hence the intervals $[s_v, t_v]$ and $[s_w, t_w]$ must have overlap, so that necessarily $s_w \leq t_v$ and $s_v \leq t_w$. Since we also have $t_u < s_v$ we get $t_u < s_v \leq t_w$ so that indeed $u \prec w$. □

Remark 8.2.7. A gflow like we get here, where each correction set only contains a single element, is known as a **causal flow**, or just a *flow*. This is the kind of flow that gflow is generalising.

There are also some simple modifications we can make to an open graph that preserve the existence of a gflow. Like adding new vertices after an output or adding and removing connections between output vertices.

Exercise 8.4. Let $G = (V, E, I, O)$ be an open graph with a gflow $(\prec, g)$ and suppose that outputs $o_i, o_j \in O$ are connected. Show then that $(\prec, g)$ is also a gflow for the open graph $G' = (V, E', I, O)$ where $E' = E \backslash \{(o_i, o_j)\}$, i.e. the graph where we removed an edge between two outputs. Similarly, show that adding an edge between two outputs also preserves gflow.

Exercise 8.5. Let $G = (V, E, I, O)$ be an open graph with a gflow $(\prec, g)$ and let $o \in O$ be some chosen output. Define $G' = (V', E', I, O')$ by $V' = V \cup \{u\}$, $O' = \{u\} \cup O \backslash \{o\}$ and $E' = E \cup \{(o, u)\}$. That is: G' is just G but with another vertex u after o that becomes the new output. Show that $(\prec', g')$ is a gflow for G' where $\prec'$ is just the transitive closure of $\prec$ with the additional relation $o \prec' u$, and $g'(o) = \{u\}$, and $g'(v) = v$ for every other vertex.

Note that these results only talk about creating a new pattern that also has a gflow. We are not defining an actual measurement pattern, and hence we are not saying anything about whether this new pattern implements the same linear map or not.

In Section 8.3 we will prove that some more elaborate ways in which we can change graphs also preserve the existence of a gflow.

8.2.5 Focussed gflow

The definition of a gflow allows the correction set $g(u)$ to also induce corrections on other vertices, as long as those other vertices are in the future of u. That is, $\mathrm{Odd}_G(g(u))$ can contain more than one vertex. It turns out that we can modify a gflow to just allow u to be corrected and no other measured vertex (this turns out to be useful later).

Definition 8.2.8. We say a gflow $(\prec, g)$ on an open graph $G = (V, E, I, O)$ is **focussed** when for every $u \in V \backslash O$, the corrected vertices $v \in \mathrm{Odd}_G(g(u))$ are either $v = u$ or $v \in O$. That is, $\mathrm{Odd}_G(g(u)) \backslash O = \{u\}$.

Exercise 8.6. Let $G = (V, E, I, O)$ be an open graph with a gflow $(\prec, g)$. Pick some vertex $u \in V \backslash O$. Let $A \subseteq V$ be some set of vertices such that for any $v \in A$ and $v \in \mathrm{Odd}_G(A)$ we have $u \prec v$ and $u \neq v$. Show then that the function g' defined by $g'(u) = g(u) \,\Delta\, A$ and $g'(w) = g(w)$ for all $w \neq u$ *also* defines a gflow $(\prec, g')$ on G.
Hint: Use Exercise 8.3.

Exercise* 8.7. Show that if an open graph has a gflow, then it also has a focussed gflow.
Hint: if you have some u for which $Odd_G(g(u))$ contains some vertex v it shouldn't, then v has to be in the future of u. Argue that $A = g(v)$ satisfies the conditions of the previous exercise. Proceed to modify $g(u)$ with A. What has now changed about $Odd_G(g'(u))$?

8.3 Optimising deterministic measurement patterns

In Chapter 5 we saw two rewrites that proved very useful for optimising Clifford diagrams: local complementation and pivoting. These rewrites allowed us to remove internal spiders from a graph-like diagram. As we've seen in this chapter, we can view graph-like diagrams as measurement patterns, and then internal spiders become measured qubits. Hence, these rewrites now offer ways to optimise the number of qubits we need to measure in the pattern. Of course, this is only useful if it preserves our ability to deterministically implement the measurement pattern, i.e. if it preserves the existence of a gflow. That is precisely what we will prove in this section.

While we tried to present the proofs here in as simple a way as possible, it still requires some stamina to process all the odd-neighbourhood acrobatics. So perhaps on a first read-through you might want to just read the statements of Proposition 8.3.2 and 8.3.5 and then move on to the next section to see what we will use these results for.

Recall that for a graph G, we defined $G \star u$ to be the graph where we locally complemented about u. In this graph the neighbourhoods have been updated as follows for any $v \in V$:

$$N_{G\star u}(v) \;=\; \begin{cases} N_G(v) \,\Delta\, N_G(u) \,\Delta\, \{v\} & \text{if } v \in N_G(u) \\ N_G(v) & \text{otherwise} \end{cases} \qquad (8.20)$$

We will also need to know how the odd neighbourhood of a set changes when you apply a local complementation.

Lemma 8.3.1. Given a graph $G = (V, E)$, $A \subseteq V$ and $u \in V$,

$$\mathrm{Odd}_{G\star u}(A) \;=\; \begin{cases} \mathrm{Odd}_G(A) \,\Delta\, (N_G(u) \cap A) & \text{if } u \notin \mathrm{Odd}_G(A) \\ \mathrm{Odd}_G(A) \,\Delta\, (N_G(u) \setminus A) & \text{if } u \in \mathrm{Odd}_G(A) \end{cases}$$

Proof Using the fact that $\mathrm{Odd}_G(A) = \underset{v \in A}{\Delta}\, N_G(v)$, we calculate:

$$\begin{aligned} \mathrm{Odd}_{G\star u}(A) \;&=\; \underset{v \in A}{\Delta}\, N_{G\star u}(v) \\ &\overset{(8.20)}{=}\; \left(\underset{v \in A \cap N_G(u)}{\Delta}\, N_G(v) \,\Delta\, N_G(u) \,\Delta\, \{v\} \right) \Delta \left(\underset{v \in A \setminus N_G(u)}{\Delta}\, N_G(v) \right) \\ &=\; \left(\underset{v \in A}{\Delta}\, N_G(v) \right) \Delta \left(\underset{v \in A \cap N_G(u)}{\Delta}\, N_G(u) \right) \Delta \left(\underset{v \in A \cap N_G(u)}{\Delta}\, \{v\} \right) \end{aligned}$$

Here in the last step we simply regrouped the terms: we used the fact that the $N_G(v)$ term appears in both the terms above it so that we can combine them into $\underset{v \in A}{\Delta}\, N_G(v)$, and we separated out the other terms. We then see that the first term is precisely $\mathrm{Odd}_G(A)$, while the last term is just a union of all the $\{v\}$ in $A \cap N_G(u)$. This expression hence simplifies to

$$\mathrm{Odd}_{G\star u}(A) \;=\; \mathrm{Odd}_G(A) \,\Delta\, \left(\underset{v \in A \cap N_G(u)}{\Delta}\, N_G(u) \right) \Delta \, (A \cap N_G(u))$$

It remains to simplify the middle expression. This is just a symmetric difference of $N_G(u)$ with itself a number of times. This term hence only appears when $|A \cap N_G(u)| \equiv 1 \bmod 2$. This is precisely the case when $u \in \mathrm{Odd}_G(A)$. Hence, if $u \notin \mathrm{Odd}_G(A)$, $\mathrm{Odd}_{G\star u}(A) = \mathrm{Odd}_G(A) \,\Delta\, (A \cap N_G(u))$. Otherwise, if $u \in \mathrm{Odd}_G(A)$, we get $\mathrm{Odd}_{G\star u}(A) = \mathrm{Odd}_G(A) \,\Delta\, N_G(u) \,\Delta\, (A \cap N_G(u))$. We can simplify this expression by using the property $B \,\Delta\, (A \cap B) = B \setminus A$ (convince yourself that this is true). The expression for $\mathrm{Odd}_{G\star u}(A)$ then becomes exactly what we wanted to prove. $\qquad\square$

Proposition 8.3.2. Let $G = (V, E, I, O)$ be an open graph with a gflow $(\prec, g)$ and let $u \in V \setminus (I \cup O)$ be an internal vertex. Then $G' := (G \star u) \setminus \{u\}$, the open graph we get by doing a local complementation about u and then removing u, also has gflow.

Proof By Exercise 8.7 we may assume that the gflow g of G is focussed, i.e. that $\mathrm{Odd}_G(g(w))$ only contains $\{w\}$ and outputs of G for any w.

We then claim that the function $g' : V \setminus (O \cup \{u\}) \to \wp(V \setminus (I \cup \{u\}))$ defined by

$$g'(w) := \begin{cases} g(w) & \text{if } u \notin g(w) \\ g(w) \,\Delta\, \{u\} \,\Delta\, g(u) & \text{otherwise} \end{cases} \qquad (8.21)$$

gives a gflow $(\prec, g')$ on $(G \star u) \setminus \{u\}$. We haven't changed the partial order, so we just need to check the different conditions that g' has to satisfy. First, note that it is well-defined: none of the $g'(w)$ contain u. In the first case this is true by construction, and in the second case we know that $u \in g(w)$, so that $g(w) \,\Delta\, \{u\} = g(w) \setminus \{u\}$.

Now we have to check three things:

1. All the elements of $g'(w)$ are in the future of w.
2. All the elements of $\mathrm{Odd}_{G \star u}(g'(w))$, except for w itself are in the future of w.
3. $\mathrm{Odd}_{G \star u}(g'(w))$ contains w.

For the first point, note that if $u \notin g(w)$, that then $g'(w) = g(w)$, in which case all of $g'(w)$ is definitely in the future of w (since we haven't changed the partial order $\prec$). Otherwise $u \in g(w)$, and hence $w \prec u$, so that all the elements of $g(u)$ are in the future of w. Hence $g'(w) = g(w) \,\Delta\, \{u\} \,\Delta\, g(u)$ only contains elements in the future of w.

Then for the second and third point. Because g is focussed and u is not an output, we know that $u \notin \mathrm{Odd}_G(g(w))$ if $w \neq u$. This means that when we use Lemma 8.3.1 to calculate $\mathrm{Odd}_{G \star u}(g(w))$ the first case applies and we get $\mathrm{Odd}_G(g(w)) \,\Delta\, (N_G(u) \cap g(w))$.

So now let's again make the case distinction on whether $u \in g(w)$. If this is not the case, then $g'(w) = g(w)$, and hence

$$\mathrm{Odd}_{G \star u}(g'(w)) = \mathrm{Odd}_{G \star u}(g(w)) = \mathrm{Odd}_G(g(w)) \,\Delta\, (N_G(u) \cap g(w)).$$

Both the elements of $\mathrm{Odd}_G(g(w))$ and $(N_G(u) \cap g(w)) \subseteq g(w)$ are certainly in the future of w, so that this odd neighbourhood lies in the future of w. Additionally, $\mathrm{Odd}_G(g(w))$ contains w, while $g(w)$ does not, so that $w \in \mathrm{Odd}_{G \star u}(g(w))$. Hence, points 2 and 3 are satisfied for those w where $u \notin g(w)$.

It then remains to check $\text{Odd}_{G\star u}(g'(w))$ for the w that satisfy $u \in g(w)$. To calculate this set it will be useful to first calculate $\text{Odd}_{G\star u}(g(u))$. As $u \in \text{Odd}_G(g(u))$, the second case of Lemma 8.3.1 applies. We hence get

$$\text{Odd}_{G\star u}(g(u)) = \text{Odd}_G(g(u)) \, \Delta \, (N_G(u) \setminus g(u)).$$

Then using the definition of $g'(w)$ in Eq. (8.21) and the linearity property of odd neighbourhoods, $\text{Odd}_G(A \, \Delta \, B) = \text{Odd}_G(A) \, \Delta \, \text{Odd}_G(B)$, we calculate:

$$
\begin{aligned}
\text{Odd}_{G\star u}(g'(w)) &= \text{Odd}_{G\star u}(g(w)) \, \Delta \, \text{Odd}_{G\star u}(\{u\}) \, \Delta \, \text{Odd}_{G\star u}(g(u)) \\
&= \text{Odd}_{G\star u}(g(w)) \, \Delta \, N_G(u) \, \Delta \, \text{Odd}_G(g(u)) \, \Delta \, (N_G(u) \setminus g(u)) \\
&= \text{Odd}_{G\star u}(g(w)) \, \Delta \, \text{Odd}_G(g(u)) \, \Delta \, (g(u) \cap N_G(u))
\end{aligned}
$$

Here in the last step we used the property $A \, \Delta \, (A \setminus B) = B \cap A$ where $A = N_G(u)$ and $B = g(u)$. We have already seen that every element of $\text{Odd}_{G\star u}(g(w))$ is in the future of w. Since we are assuming $u \in g(w)$, we have $w \prec u$, and hence both $g(u) \cap N_G(u) \subseteq g(u)$ and $\text{Odd}_G(g(u))$ are in the future of w. It remains to verify that $w \in \text{Odd}_{G\star u}(g'(w))$, but this follows because $w \in \text{Odd}_{G\star u}(g(w))$.

We hence see that g' is indeed a gflow. Note that while we started with a focussed gflow g, the new gflow g' is not necessarily focussed. $\qquad\square$

Proving that pivoting preserves gflow is done quite similarly. Just like how we needed a lemma that told us how odd neighbourhoods evolved under application of a local complementation, we need one that tells us what happens when a pivot happens. For this it will be helpful to introduce a little bit of extra notation.

Definition 8.3.3. We define the **closed neighbourhood** of a vertex u as $N_G[u] := N_G(u) \, \Delta \, \{u\} = N_G(u) \cup \{u\}$. Similarly we define the **closed odd neighbourhood** of a set A as $\text{Odd}_G[A] := \Delta_{u \in A} N_G[u] = \text{Odd}_G(A) \, \Delta \, A$.

Let u and v be some vertices of G, and write $G \wedge uv$ for the graph where we pivoted along uv. Then for any w we have:

$$
N_{G\wedge uv}(w) = \begin{cases}
N_G(w) \, \Delta \, N_G[u] \, \Delta \, N_G[v] & \text{if } w \in N_G[u] \cap N_G[v] \\
N_G(w) \, \Delta \, N_G[v] & \text{if } w \in N_G[u] \setminus N_G[v] \\
N_G(w) \, \Delta \, N_G[u] & \text{if } w \in N_G[v] \setminus N_G[u] \\
N_G(w) & \text{otherwise}
\end{cases}
\tag{8.22}
$$

Exercise 8.8. Calculate using this equation, that when $w = u$ we get $N_{G\wedge uv}(u) = (N_G(v) \cup \{v\}) \setminus \{u\}$ as we expect (since u and v essentially just swap places in the graph).

Lemma 8.3.4. Given a graph $G = (V, E)$, $A \subseteq V$, $u \in V$, and $v \in N_G(u)$,

$$\text{Odd}_{G \wedge uv}(A) = \begin{cases} \text{Odd}_G(A) & \text{if } u, v \notin \text{Odd}_G[A] \\ \text{Odd}_G(A) \, \Delta \, N_G[v] & \text{if } u \in \text{Odd}_G[A], v \notin \text{Odd}_G[A] \\ \text{Odd}_G(A) \, \Delta \, N_G[u] & \text{if } u \notin \text{Odd}_G[A], v \in \text{Odd}_G[A] \\ \text{Odd}_G(A) \, \Delta \, N_G[u] \, \Delta \, N_G[v] & \text{if } u, v \in \text{Odd}_G[A] \end{cases}$$

$$(8.23)$$

Exercise* 8.9. Prove Lemma 8.3.4.
Hint: You will need the same tricks that were used to prove Lemma 8.3.1.

Proposition 8.3.5. Let $G = (V, E, I, O)$ be an open graph with a gflow $(\prec, g)$ and let $u, v \in V \setminus (I \cup O)$ be connected internal vertices. Then $G' := (G \wedge uv) \setminus \{u, v\}$, the open graph we get by doing a pivot along uv and then removing u and v, also has gflow.

Proof Again we will assume that g is focussed. We then define a gflow g' for $G \wedge uv \setminus \{u, v\}$ as follows: for every $w \in V \setminus (O \cup \{u, v\})$ set $g'(w) := g(w) \setminus \{u, v\}$. Every condition needed for g' to be a focused gflow then follows immediately except for checking that $\text{Odd}_{G \wedge uv \setminus \{u,v\}}(g'(w))$ contains w and lies in the future of w.

Note that $\text{Odd}_G[g(w)] = \text{Odd}_G(g(w)) \, \Delta \, g(w) \subseteq \{w\} \cup O \cup g(w)$ by the focussed property, so that $u \in \text{Odd}_G[g(w)] \iff u \in g(w)$ and similarly for v. Now, to look at the odd neighbourhoods of the correction sets we will make a case distinction on w based on whether $u \in g(w)$ or $v \in g(w)$.

- If $u, v \notin g(w)$, then also $u, v \notin \text{Odd}_G[g(w)]$ and hence by Lemma 8.3.4 $\text{Odd}_{G \wedge uv}(g(w)) = \text{Odd}_G(g(w))$. Since $g'(w) = g(w) \setminus \{u, v\} = g(w)$ we are then done.

- If $u, v \in g(w)$, then also $u, v \in \text{Odd}_G[g(w)]$ and $g'(w) = g(w) \setminus \{u, v\} = g(w) \, \Delta \, \{u, v\}$. Hence:

$$\begin{aligned} \text{Odd}_{G \wedge uv}(g'(w)) &= \text{Odd}_{G \wedge uv}(g(w) \, \Delta \, \{u, v\}) \\ &= \text{Odd}_{G \wedge uv}(g(w)) \, \Delta \, \text{Odd}_{G \wedge uv}(\{u, v\}) \\ (\text{Lem. 8.3.4}) \quad &= \text{Odd}_G(g(w)) \, \Delta \, (N_G[u] \, \Delta \, N_G[v]) \, \Delta \, \text{Odd}_G(\{u, v\}) \\ &= \text{Odd}_G(g(w)) \, \Delta \, \text{Odd}_G(\{u, v\}) \, \Delta \, \text{Odd}_G(\{u, v\}) \\ &= \text{Odd}_G(g(w)) \end{aligned}$$

Note that we used here the fact that $u \in N_G(v)$ and $v \in N_G(u)$, so that $N_G[u] \, \Delta \, N_G[v] = N_G(u) \, \Delta \, N_G(v) = \text{Odd}_G(\{u, v\})$.

- If $u \in g(w)$ and $v \notin g(w)$,

$$
\begin{aligned}
\mathrm{Odd}_{G \wedge uv}(g'(w)) &= \mathrm{Odd}_{G \wedge uv}(g(w) \,\Delta\, \{u\}) \\
&= \mathrm{Odd}_{G \wedge uv}(g(w)) \,\Delta\, \mathrm{Odd}_{G \wedge uv}(\{u\}) \\
(\text{Lem. } 8.3.4) \quad &= \mathrm{Odd}_G(g(w)) \,\Delta\, N_G[v] \,\Delta\, \mathrm{Odd}_G(\{v\}) \\
&= \mathrm{Odd}_G(g(w)) \,\Delta\, \{v\}
\end{aligned}
$$

  Here we have used that $\mathrm{Odd}_{G \wedge uv}(\{u\}) = N_G(v)$. As we are deleting u and v from the final graph, we can ignore the fact that we get $\{v\}$ in the odd neighbourhood.

- If $u \notin g(w)$ and $v \in g(w)$ we prove it similarly to the previous case.

$\square$

We hence see that we can indeed do the spider-removing version of local complementation and pivoting while preserving the existence of gflow. You might at this point wonder whether we couldn't just do the local complementation *without* removing the spider, and whether this preserves gflow. It in fact does not, but that is because we aren't using the *right* notion of gflow for this: the spider-preserving version of local complementation (Lemma 5.2.4) introduces a $X(\frac{\pi}{2})$ phase on the spider being pivoted on. This makes it so that we can no longer interpret this spider as being measured in the XY plane. The solution to this is to treat it as a qubit measured in a different type of plane. We will see how to do this in Section 8.5.

Remark 8.3.6. Propositions 8.3.2 and 8.3.5 are results about open *graphs*, not measurement patterns. Hence, we can in general apply these to make new kinds of measurement patterns, which might not implement the same linear map as before. If we want to apply these in a way that preserves the linear map the pattern is implementing, we also need to consider the specific measurement angles the pattern must have and how these get updated by the rewrite. For this we would need to reference the ZX local complementation and pivot simplifications of Lemma 5.2.9 and 5.2.11.

8.4   From measurement patterns to circuits

Starting with a quantum circuit, we can transform it into a graph-like diagram which has gflow (Proposition 8.2.6), and hence we can turn this circuit into a deterministic measurement pattern. We have also seen several ways in which we can change a measurement pattern while preserving gflow. These patterns we get don't necessarily look very circuit-like any more, so it is not

obvious any more how the connections and measurements in this pattern correspond to unitaries acting on an input state.

In this section we will see that nevertheless we can transform an arbitrary deterministic measurement pattern with gflow back into a circuit that implements the same linear map. Since these patterns are deterministic, we can represent the linear map they implement just as a graph-like ZX-diagram, like we did in Section 8.2.1. The input of our algorithm will be a graph-like diagram whose underlying open graph has a gflow, and the output should be an equivalent quantum circuit. We call this process **circuit extraction**. The basic idea will be similar to the CNOT circuit extraction algorithm we saw way back in Chapter 4, but instead of doing one round of Gaussian elimination to get a circuit out, we will be doing 'a little bit' of row elimination in order to extract a single internal spider from the pattern. We repeat this until the diagram contains no more internal spiders. We then have a diagram looking very similar to the Clifford GSLC normal form of Section 5.3.3, which we know how to turn into a circuit.

Since our algorithm doesn't work with measurement patterns directly, but with ZX-diagrams, we can let the input be *any* graph-like diagram whose underlying open graph has gflow. Such a diagram will then look something like this:

$$(8.24)$$
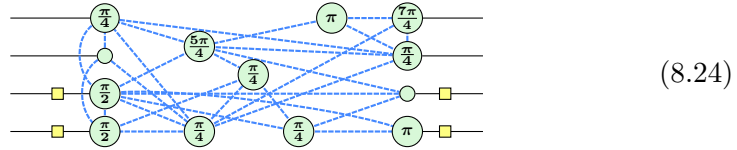
We put some phases on the spiders for concreteness, but as our only consideration is whether the diagram has gflow, these are pretty much irrelevant to the steps we need to do to extract a circuit out of a diagram.

So let's start trying to make this diagram more circuit like. As in Section 5.3.3, we first observe that the spiders on the boundary already directly correspond to quantum gates: either single-qubit gates or two-qubit CZ gates:


$$(8.25)$$
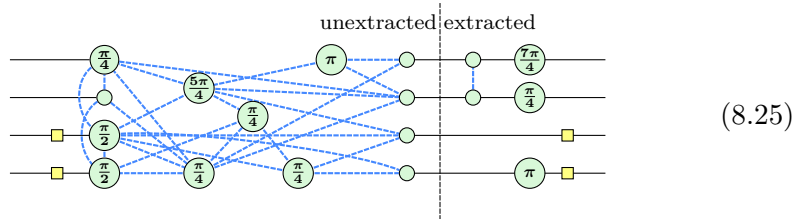
Here we mark the two parts of the diagram by 'unextracted' and 'extracted' to denote that we are done with the part on the right. This part already looks like a circuit now, so we don't have to do anything with that anymore. We

will refer to the spiders on the boundary of the extracted and unextracted part as the **frontier**. We will be applying operations to the frontier in such a way that we can move frontier spiders to the extracted part and internal spiders onto the frontier. The reason we start at the end of the diagram instead of at the beginning is because it will be easier to show that such operations preserve the existence of a gflow on the remaining diagram. The extraction algorithm is only guaranteed to work if the diagram has gflow, so it is important that all the steps change the diagram in ways which preserve the existence of a gflow. For instance, in Eq. (8.25) the underlying graph of the unextracted part only changed in having no more connections between the output vertices. This is exactly the transformation that we showed in Exercise 8.4 preserves gflow.

In the case of the GSLC normal form of Section 5.3.3, we realised that the internal connectivity of the diagram corresponds to a CNOT circuit. Here we can do something similar, but instead of rewriting the entire diagram to a CNOT circuit directly, we can 'extract' a single CNOT in order to change the connectivity of the diagram:

**Proposition 8.4.1.** For any graph-like diagram D, the following equation holds:
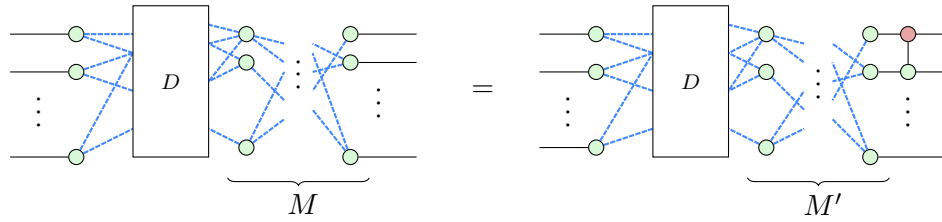
where M describes the biadjacency matrix of the relevant vertices, and M' is the matrix produced by starting with M and then adding row 1 to row 2, taking sums modulo 2. Furthermore, if the diagram on the left has gflow, then the graph-like part on the right also has gflow.

Proof For clarity we will not draw the entire diagram, but instead we focus on the relevant part. First of all we note that we can add CNOTs in the following way while preserving equality:
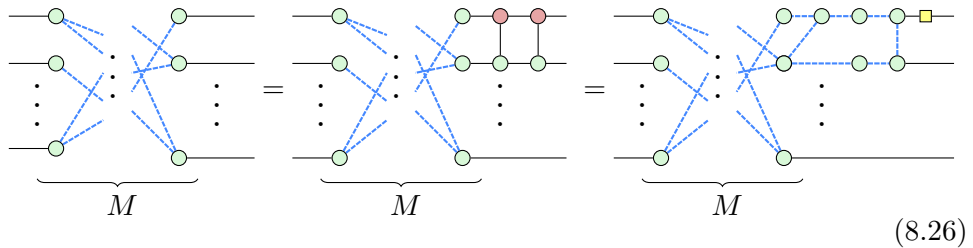
$$(8.26)$$

Now let A denote the set of vertices connected to the top vertex, but not to the vertex beneath it, B the set of vertices connected to both, and C the vertices connected only to the bottom one. Further restricting our view of the diagram to just these two lines, we see that we can apply a pivot:
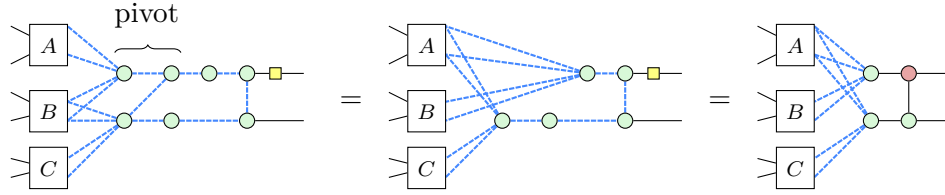
Looking at the connectivity matrix, it is straightforward to see that the matrix M has now been changed in exactly the way described.

To see that these operations preserve the existence of a gflow, note first that we can view the rewrite of Eq. (8.26) as adding some identity spiders to the outputs of a diagram, and adding connections between these outputs. Hence, by Exercises 8.4 and 8.5 the resulting diagram still has gflow. We then do a pivot, which also preserves gflow by Proposition 8.3.5, and finally we consider the last CNOT as no longer being part of the graph-like diagram, which on the level of the underlying open graph looks like removing some output spiders, which can also be shown to preserve gflow. $\qquad\square$

To demonstrate how we can use this, let us write down the biadjacency matrix from the frontier to the other spiders in our example diagram:

$$\leftrightsquigarrow \begin{pmatrix} & a & b & c & d & e & f \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 1 & 0 & 1 & 0 \\ 3 & 0 & 0 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

$$(8.27)$$

We see that if we add the fourth row to the third row that there will be only a single 1 on the third row. This means that the third frontier spider is only connected to a single internal spider, and hence corresponds to an identity

spider that can be removed:

$$(8.28)$$

We now have one less spider in the unextracted part. We can again unfuse the phase of the frontier spider into the extracted part, and the same for the connections between the frontier spiders. We then rinse and repeat, extracting the spiders one by one. But *why* can we always do this? This is where gflow comes into play.

Using gflow we can show that the biadjacency matrix always has this property that we can reduce at least one of the rows to contain a single 1 using row operations: Suppose we have a graph-like diagram D and that its underlying open graph G has a gflow $(\prec, g)$. Pick now the vertex u that is maximal according to the partial order $\prec$ in the set $V \backslash O$. That is, u corresponds to the *last* measured qubit. Now we know that $g(u)$ and $\text{Odd}_G(g(u)) \backslash \{u\}$ only contain vertices in the future of u. So in this case they must only contain outputs. So let $o_1, \ldots, o_k$ be the outputs that form $g(u)$. Their sets of neighbours $N(o_j)$ then correspond to rows of the biadjacency matrix we wish to simplify. We are assuming we have already gotten rid of any connections between outputs, so that there are no outputs in the set $N(o_j)$. We know that $\text{Odd}_G(g(u)) \subseteq O \cup \{u\}$ and furthermore $\text{Odd}_G(g(u)) = N(o_1) \Delta \cdots \Delta N(o_k)$ (Exercise 8.2). So in fact $\text{Odd}_G(g(u)) = \{u\}$.

Now here comes the kicker: Taking the symmetric difference of sets follows exactly the logic of addition in $\mathbb{F}_2$! If we 'add' two sets $A \Delta B$, then if there is an s with $s \in A$ and $s \in B$, then $s \notin A \Delta B$, exactly how $1 \oplus 1 = 0$. All the other cases are checked similarly. So when we were adding together rows in the biadjacency matrix, what we were really doing was calculating the symmetric difference of the neighbourhoods. So since $N(o_1) \Delta \cdots \Delta N(o_k) = \{u\}$, that means if we add together the rows of the biadjacency matrix corresponding to

the outputs o_j, then the resulting row has just a single 1, which corresponds to u. Hence, gflow tells us exactly which rows we can add together, and hence which CNOTs we need to place, in order to make progress. As we can do row operations on our biadjacency matrix using Proposition 8.4.1 and this preserves gflow, we will have after extracting a vertex *still* a diagram which has gflow. And hence when we repeat this procedure we will always be able to find some rows in the biadjacency matrix that combine together to form a row with a single 1, whose spider can then be extracted.

Note that this is a similar procedure to how we extracted a circuit from a CNOT+Phase diagram in Section 7.2: there we did column operations in order to reduce a column to contain just a single 1. Because we are extracting from the right here, we are doing row operations instead. At some point we will have removed all internal spiders, in which the case the remaining diagram is like the GSLC normal form of the previous chapter, except that the phases on the spiders can be arbitrary. This final bit can then also be transformed into a circuit using a variation on the procedure described in Section 5.3.3 and then we are done.

We can then conclude with the following theorem.

Theorem 8.4.2. Let D be a graph-like diagram whose underlying open graph has a gflow and which has the same number of inputs as outputs. Then we can efficiently extract a unitary circuit that implements the same linear map as D.

In Exercise 7.7 we saw that extracting a circuit from a ZX-diagram is in general a very hard problem. But now we see that if the diagram has gflow, that this 'smooths' it out enough so that it becomes much easier to extract a circuit from it.

Since any measurement pattern corresponds to a graph-like ZX-diagram with some additional information on how the measurement outcomes should be propagated, we can also use this result to conclude that any measurement pattern with gflow must be unitary.

Theorem 8.4.3. A deterministic measurement fragment with gflow and the same number of inputs as outputs implements a unitary.

8.5 Measurements in three planes

So far we have restricted all our thinking on determinism and gflow to measurements all occurring in the same plane, namely the XY plane. With some small modifications we can also make it work for measurement patterns

where the measurements occur in all three planes. Before we see how that works, let's answer an important question first: why bother? We've seen that we can do universal computation with measurement patterns where all measurements are in the XY plane, and that any circuit can be transformed into a single-plane measurement pattern. So why should we even talk about measurements in other planes.

There are a couple of reasons. First, in our discussion on universal resource states in Section 8.1.1 we saw that cluster states can be used to perform arbitrary computations. This however crucially depends on 'carving out' paths by doing strategic measurements in the YZ plane. It is possible to do it without this feature, but it makes it harder and less intuitive. Second, in the previous chapter we've seen how useful phase gadgets are. Phase gadgets are however not compatible with single-plane gflow. Consider the following simple diagram containing a phase gadget connected to just one spider:

$$\text{gadget}\Big\{ \quad \tag{8.29}$$

This diagram is of course unitary, and if we were to simplify the gadget to a phase gate, the diagram would definitely have gflow. So what about this diagram itself? Consider the following two facts about gflow that we know: the phases of spiders are not important for having gflow or not; and if you have gflow, the diagram must be unitary. Combining these facts we see that if the diagram (8.29) were to have gflow, then the following diagram would also have gflow, and hence would need to be unitary for all α and β:

$$\tag{8.30}$$

But if we pick $\alpha = \beta = \frac{\pi}{2}$, we can show that the diagram disconnects, and so it is not unitary.

So what is the issue here? We see that the problem with unitarity arises when we put a non-zero phase on the 'base' of the gadget. Doing this stops it from being a phase gadget, and makes it just a stack of two spiders with some phase. Instead, we should be treating the two spiders of the gadget as a single 'unit' where we can't change the phase of the base. We can do this by treating a phase gadget as a qubit measured in a different basis than a regular spider. This then necessitates using multiple measurement axes.

The final reason for considering multiple measurement planes, closely related to allowing for phase gadgets, is that it allows us to be more efficient

about our computations. It turns out we can do the same computations with fewer measured qubits.

So with all this in mind, let's see how we should think about measurements happening in all three principal planes of the Bloch sphere. Recall first that these measurement bases have a nice representation in the ZX-calculus:

$$XY \;\rightsquigarrow\; -\boxed{\alpha_k + b_k\pi} \qquad XZ \;\rightsquigarrow\; -\boxed{\tfrac{\pi}{2}}-\boxed{\alpha_k + b_k\pi} \qquad YZ \;\rightsquigarrow\; -\boxed{\alpha_k + b_k\pi} \tag{8.31}$$

Here, as before, α_k denotes the measurement angle of this specific effect, and the b_k is a Boolean variable denoting the outcome of the measurement. The XY and YZ plane measurements correspond to natural structures in a graph-like diagram: 'regular' internal spiders, and phase gadgets. We can view the 'regular' internal spiders of a graph-like diagram as qubits measured in the XY plane:


$$\tag{8.32}$$

If we instead have a phase gadget, then we can view this as a qubit measured in the YZ plane:

$$\tag{8.33}$$

Measurements in the XZ plane don't naturally occur in the graph-like diagrams we consider, because the $\frac{\pi}{2}$ phase on the spider would have meant it would've been removed by applying a local complementation. Another 'explanation' to why we don't need XZ-plane measurements, is that we see that XY measurements correspond to Z-spiders, while YZ measurements correspond to X-spiders. Hence, XZ-plane measurements correspond to 'Y-spiders', which we also don't need to use.

Remark 8.5.1. It might be a bit confusing to remember that XY measurements correspond to Z spiders and so on for the other types of effects. The trick is to remember that the type of spider is the letter that does *not* appear in the name. So for XY, the corresponding spider is Z, and for YZ it is X.

We see then that to view a graph-like diagram as a measurement pattern, we need to interpret phase gadgets as qubits measured in the YZ plane,

while every other non-output spider is treated as a qubit measured in the *XY* plane.

Now, let's consider what the corresponding notion of gflow should be for a 3-plane measurement pattern. To do so, let's go back to the examples we looked at at the start of Section 8.2. There we considered an *XY* vertex that had an error:
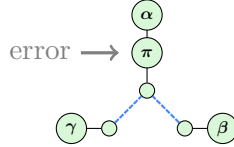

We could then correct this by pushing out an $X(\pi)$ phase from the *XY* vertex with phase β. When we push this $X(\pi)$ phase, it becomes a $Z(\pi)$ at the location we want it:
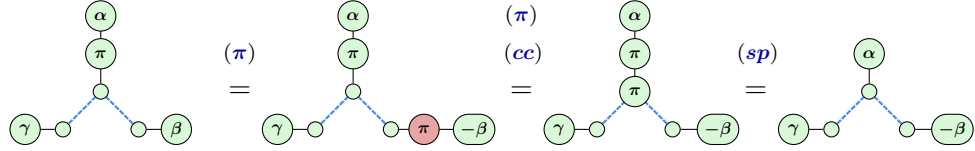
We captured the more general version of this idea with the gflow correction sets $g(u)$. Here the set $g(u)$ tells us where we have to extract out $X(\pi)$ phases in order to get a $Z(\pi)$ at the desired location. In particular, we get $Z(\pi)$ phases at all the vertices in $\text{Odd}_G(g(u))$. The conditions on $g(u)$ for u an *XY* vertex are hence that $u \in \text{Odd}_G(g(u))$ and $u \notin g(u)$ (since we can't extract out an $X(\pi)$ phase from u itself). We see then that we interpret $g(u)$ as the vertices where we need to push out an $X(\pi)$ and $\text{Odd}_G(g(u))$ where we need to push out a $Z(\pi)$. The properties of an *XY*-plane measured vertex u tell us what the relations to $g(u)$ and $\text{Odd}_G(g(u))$ need to be.

Let's now consider the same example, but with a *YZ* vertex instead:
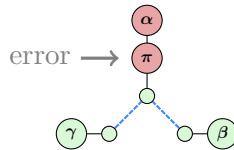
We see then that now we need to correct an $X(\pi)$ error instead of a $Z(\pi)$ error. Still treating $g(u)$ as the 'set of vertices where we push out an $X(\pi)$' we see then that now we *do* need $u \in g(u)$ to correct this error, as pushing out another $X(\pi)$ from u corrects the error present. Analogously, we now do *not* want a $Z(\pi)$ to appear at u, so we must have $u \notin \text{Odd}_G(g(u))$.

Putting it succinctly: $g(u)$ are the *X*-corrections, while $\text{Odd}_G(g(u))$ are the *Z*-corrections. The measurement plane of the vertex determines which

correction its measurement outcome needs. Note that there is no restriction on what types of vertices can appear in $g(u)$ or $\mathrm{Odd}_G(g(u))$, as it is possible to extract both $Z(\pi)$ phases and $X(\pi)$ phases from a vertex used in a correction set, regardless of which measurement plane that vertex has.

In order to give a formal definition of a gflow on multiple measurement planes we need to work with a *labelled* open graph, where each non-output vertex u is labelled by its type $t(u)$. We take this type to be a set that is either $\{X, Y\}, \{Y, Z\}$ or $\{X, Z\}$ (defining it as a set likes this makes the following definition nicer).

Definition 8.5.2. Let $G = (V, E, I, O, t)$ be a labelled open graph with types t. A **3-plane gflow** $(\prec, g)$ on G consists of a partial order $\prec$ on the vertices of G, and a **correction set function** $g : V \backslash O \to \wp(V \backslash I)$ that associates to every non-output vertex $u \in V \backslash O$ a subset of non-input vertices $g(u) \subseteq V \backslash I$, such that for every non-output vertex u the following conditions hold:

1. The inputs are minimal in the partial order $\prec$ (meaning that if $v \prec i$ for some input i, then $v = i$) and conversely the outputs are maximal;
2. We have $u \prec v$ for every $v \in g(u)$ and $v \in \mathrm{Odd}_G(g(u))$, i.e. corrections lie in the future of u.
3. $u \in \mathrm{Odd}_G(g(u))$ if and only if $X \in t(u)$,
4. $u \in g(u)$ if and only if $Z \in t(u)$.

We say an open graph *has 3-plane gflow* when at least one 3-plane gflow can be defined on it.

This definition might look a bit weird, but we can check that this gives the correct behaviour for all three types of measurement plane. If $t(u) = \{X, Y\}$, then we see that condition 3 gives $u \in \mathrm{Odd}_G(g(u))$, while condition 4 gives $u \notin g(u)$. This is because condition 4 is an 'if and only if'. Hence, as we do *not* have $X \in t(u)$, we also do *not* have $u \in g(u)$. In the same way we can check that for $t(u) = \{Y, Z\}$ we get $u \notin \mathrm{Odd}_G(g(u))$ and $u \in g(u)$ as we expect.

For the final type $t(u) = \{X, Z\}$ we see that we get the conditions $u \in \mathrm{Odd}_G(g(u))$ and $u \in g(u)$. This means that to correct an error at u, we should get both an $X(\pi)$ and a $Z(\pi)$ there. We can check that this is indeed what is necessary:

$$-\boxed{\tfrac{\pi}{2}}-\boxed{\alpha+\pi}\quad \propto \quad -\boxed{\pi}-\boxed{-\tfrac{\pi}{2}}-\boxed{\alpha}\quad \overset{(sp)}{=}\quad -\boxed{\pi}-\boxed{\pi}-\boxed{\tfrac{\pi}{2}}-\boxed{\alpha} \qquad (8.34)$$

We indeed see that the wrong measurement outcome in a XZ-plane measured vertex requires both a Pauli X and Z to correct it.

With this discussion we see then that 3-plane gflow allows us to correct the wrong measurement outcome in every measured vertex. We can then write down a 3-plane version of Theorem 8.2.4. As the fully-correct adaptation is a bit technical to write down, we will instead give a more informal description.

Theorem 8.5.3. Let D be a graph-like ZX-diagram and let (G, I, O, t) be its underlying labelled open graph, where t is a choice of measurement planes 'compatible' with D (e.g. the plane is XY for a 'regular' spider and YZ for a phase gadget). Then if its associated labelled open graph has a 3-plane gflow, we can efficiently construct a deterministic measurement pattern from D that implements the same linear map as D.

Remark 8.5.4. The reason we write 'compatible with' above, is because in general there are multiple measurement planes that we can associate to a spider. This happens when the spider has a Clifford phase, or more generally when the diagram is an isometry instead of a unitary.

We can see something nice in Definition 8.5.2 that simplifies our life somewhat: each 3-plane gflow is compatible with exactly one choice of types. This is because conditions 3 and 4 are if-and-only-if. Just by inspecting whether $u \in \mathrm{Odd}_G(g(u))$ and $u \in g(u)$ we can exactly determine which of the three types $t(u)$ is. Hence, if we are given a labelled open graph with a 3-plane gflow, we don't actually need to know the labels as we can recover them from the gflow itself. With slight abuse of notation we will hence just say that an (unlabelled) open graph has a 3-plane gflow, and that the label of a vertex is determined by the properties of its correction set.

8.5.1 Rewriting 3-plane gflow

Measurement patterns with 3-plane gflow allow some rewrites that have no counterpart in the single measurement-plane model. For instance, we can actually remove any vertex that is not measured in the XY plane.

Exercise 8.10. Let G be an open graph with 3-plane gflow $(g, \prec)$. Let u be a non-output, with $u \in g(u)$, i.e. $t(u) \neq \{X, Y\}$. Show that then u can be removed from G while preserving the existence of a 3-plane gflow and that furthermore, this gflow can be chosen in such a way to preserve the measurement plane of all vertices. *Hint: On the new graph with u removed, define a new gflow g' as $g'(v) = g(v)$ if $u \notin g(v)$ and $g'(v) = g(v)\Delta g(u)$ otherwise.*

While it might be surprising at first that we can just remove qubits while preserving gflow, note that these YZ and XZ measurements we can remove are essentially phase gadgets. When we set the phase of a phase gadget to zero, they do not contribute to the linear map. Since determinism must hold regardless of the measurement angle, the determinism must hence not 'depend' on the phase gadget's existence.

In Proposition 8.3.2 we showed that we can do the spider-removing version of local complementation while preserving single-plane gflow. It turns out that we can split this up into its component parts: local complementation, and spider removing. Exercise 8.10 already shows how to remove a spider while preserving existence of gflow.

Proposition 8.5.5. Let $(g, \prec)$ be a 3-plane gflow for an open graph (G, I, O), and let u be an internal vertex. Then the local complementation of G about u, written $G \star u$ has a 3-plane gflow $(g', \prec)$ where g' is defined by

$$g'(u) \; := \; \begin{cases} g(u) & \text{if } t(u) = \{Y, Z\} \\ g(u)\Delta\{u\} & \text{else} \end{cases}$$

$$g'(v) \; := \; \begin{cases} g(v) & \text{if } u \notin \text{Odd}_G(g(v)) \\ g(v)\Delta g'(u)\Delta\{u\} & \text{if } u \in \text{Odd}_G(g(v)) \end{cases}$$

where v is any non-output vertex not equal to u.

The proof of this proposition is similar to that of Proposition 8.3.2, though with more case distinctions for all the different measurement planes u and v can have. We leave the proof to the particularly motivated reader.

Exercise* 8.11. Prove Proposition 8.5.5.

In Proposition 8.5.5 we didn't write out the measurement planes of the of the vertices in $G \star u$. But it is important to keep in mind that these *do* change. Way back in Section 5.2.1 we saw that we can perform a local complementation on a graph state by adding some local Cliffords. In particular, on u we add an $X(\pm\frac{\pi}{2})$ phase, while all of its neighbours get a $Z(\mp\frac{\pi}{2})$ phase. These $\frac{\pi}{2}$ phases change the measurement planes of u and its neighbours. For instance, suppose that $t(u) = \{X, Y\}$. Then when its spider gains an $X(-\frac{\pi}{2})$ phase, we see that:

$$(8.35)$$

Hence, it becomes a spider measured in the XZ plane. This gives us a new

perspective on the spider-removing version of local complementation: the local complementation changes the measurement plane from XY to XZ, and then Exercise 8.10 shows us that we can remove a spider measured in the XZ plane without losing the existence of gflow.

A pivot along the edge between vertices u and v is just a triple of local complementations on u, then v, and then u again. Hence, we see that a (regular, non-spider-removing) pivot also preserves gflow! As we saw in Section 5.2.2, the spiders being pivoted on gain (or lose) a Hadamard on their output wire:
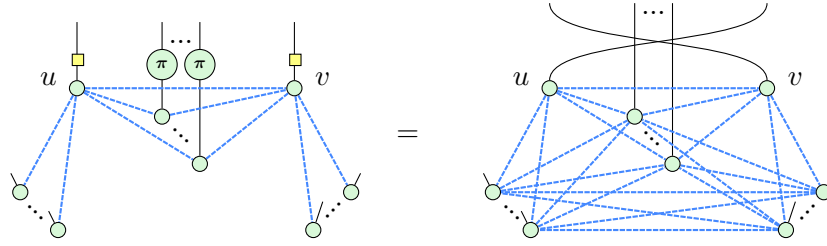
In our setting of measurement planes, this again means that their measurement planes will change. In particular, the Z-spider of an XY measurement, becomes the X-spider of an YZ measurement. This again allows us to decompose the preservation of gflow under the spider-removing pivot as a sequence of two operations: first we pivot to change the XY vertices into YZ vertices, and then we remove these vertices while preserving gflow using Exercise 8.10.

This also immediately gets us the gflow preservation of variations on the pivot we've seen. In particular, the gadget pivot of Lemma 7.6.1 we can see as doing a regular pivot on a pair of XY vertices, where we then only remove the resulting YZ vertex corresponding to the Pauli spider:
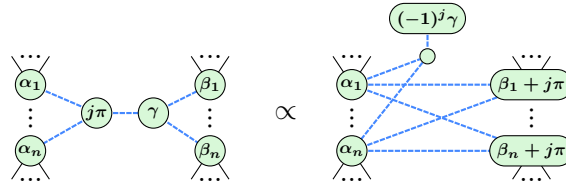
The new phase gadget we get is the second spider of the pivot, which transforms from XY to YZ. As pivoting preserves gflow, we see that the gadget pivot (which we now see is just a regular pivot followed by a vertex removal) also preserves gflow. Hence, we can introduce phase gadgets while preserving the existence of a 3-plane gflow!

Exercise 8.12. Prove that local complementation and pivoting have the following effects on measurement planes:

- For local complementation target: XY goes to XZ, and vice versa, while YZ stays fixed.
- For local complementation neighbours: XY stays fixed, while XZ and YZ switch.
- For pivoting targets: XY switches with YZ, XZ stays fixed.
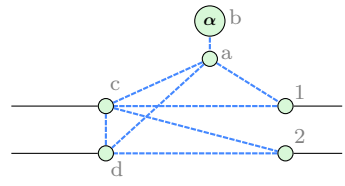- For pivoting neighbours: everything stays fixed.

We said at the start of this section that 3-plane patterns can be more efficient in the number of qubits measured. We can now see why. In Section 7.6 we observed that local complementation and pivoting do not necessarily remove all internal Pauli spiders. In order to do so, we had to turn them into the base of a phase gadget. While this does not decrease the total number of spiders in the diagram, when viewing the diagram as a measurement pattern, the two spiders of a phase gadget correspond to a single qubit measured in the YZ plane. If the phase gadget has a Clifford phase, we can remove it, so we observed that every internal spider either has a non-Clifford phase, or is part of a phase gadget with a non-Clifford phase. Phrasing this in terms of measurement patterns, we see that these rewrite rules allow us to remove all non-input qubits that are measured in a Clifford angle. Since all steps preserve gflow, the resulting compact pattern can still be deterministically implemented.

Theorem 8.5.6. For every 3-plane measurement fragment containing k non-Clifford non-input measured qubits, we can efficiently find an equivalent measurement fragment that contains at most k non-input measured qubits.

8.5.2 Circuit extraction, now phase gadget compatible

It turns out we can also extract circuits from 3-plane patterns. To do this we need to slightly modify the algorithm we described in Section 8.4.

First, let's see how that algorithm will go wrong when we have measurements not in the XY plane. Let's consider the following example where we have a phase gadget (i.e. a YZ plane measurement) connected to the frontier:

$$
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
1 & 1 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 & 1
\end{array}
\qquad (8.36)
$$

Looking at the connectivity matrix we see there is no way to do row operations to reduce any row to just have a single one. So our strategy for

extracting spiders one-by-one fails here. But remember that we introduced
phase gadgets into our diagrams by doing pivots (Lemmas 7.6.1 and 7.6.2).
So if a pivot got us into this mess, perhaps another pivot can get us out of
it:

$$
\begin{pmatrix} & a & b & c \\ 1 & 1 & 1 & 1 \\ 2 & 0 & 1 & 1 \end{pmatrix} \tag{8.37}
$$
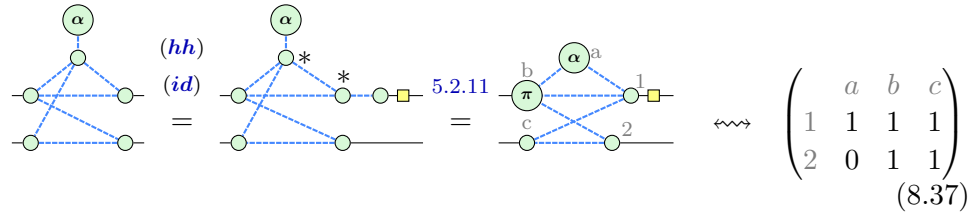
Success! Looking at the connectivity matrix we see that a single row operation
from qubit 2 to qubit 1 gives us the right connectivity to extract the spider
with the α phase.

In general, the way we do circuit extraction is as follows. First, we need
to make sure our pattern only contains XY and YZ measurements, no XZ
measurements allowed (we describe how to do this later). Then, as long as
we have XY vertices we can extract, we extract them. When we get stuck,
it must be because there are YZ vertices in the way. We do a pivot on them
to change them into XY vertices, and then we continue the process. Since
all steps preserve gflow, this procedure works and allows us to extract the
full diagram.

So how do we get rid of the XZ measurements? We can do this by doing a
series of smart pivots and local complementations to change the measurement
planes in useful ways (see Exercise 8.12). First, if a XZ is connected to a
YZ, we do a pivot on them. This transforms the YZ to XY, but keeps the
XZ as is. Once this is done to all XZ vertices, they are only connected to
XY vertices and other XZ vertices. Then for each XZ vertex, we do a local
complementation on them. This changes it into an XY vertex. Neighbouring
XY vertices stay the same, but neighbouring XZ turn into YZ. Hence, each
application reduces the number of XZ vertices, so that we can get rid of all
of them. All of these steps preserve gflow and don't increase or decrease the
number of measurements (although they do change the connectivity of the
pattern).

Theorem 8.5.7. We can efficiently turn a graph-like diagram with 3-plane
gflow into an equivalent quantum circuit.

8.6 There and back again

Let's summarise some of the things we have seen in this chapter. We introduced a new way to perform quantum computations: using measurements. This lead us to the notion of a measurement pattern, where we describe a set of inputs, outputs, measurements and corrections. This model is also universal for quantum computing, as we can implement arbitrary single-qubit gates and CNOTs and compose these together. If we consider a single branch of a measurement fragment (where we fix all the measurement outcomes), this can be represented directly as a graph-like ZX-diagram. Conversely, a graph-like ZX-diagram can be easily understood as a branch of a measurement fragment. In this sense, graph-like diagrams and measurement fragments are 'the same thing'. This misses an important detail though: determinism.

A measurement fragment is deterministic when we can perform the desired computation regardless of the observed measurement outcomes, by applying the appropriate corrections. If our measurement fragment is not deterministic it is considerably less useful, as it doesn't allow us to always calculate what we want. In that case we would need to post-select, i.e. keep running the computation until we get the desired measurement outcomes, to perform the correct calculation. We introduced a useful tool for thinking of determinism: gflow. Gflow is not a property of a specific measurement fragment, but rather of the underlying graph structure, and is agnostic to the specific measurement angles in the pattern.

To specify a full deterministic measurement pattern we have to describe how all the measurement angles change depending on the observed measurement outcomes. If instead we know the pattern has gflow, then this is already sufficient, as this allows us to construct a correction strategy. Instead of deterministic measurement patterns we can hence consider the weaker notion of graph-like diagram with gflow: the graph-like diagram tells us what the map should be when we get the ideal outcomes, and the gflow tells us how to correct when we get the wrong outcomes. This notion is weaker, because there might also be other correction strategies that do not correspond to a gflow strategy, or even measurement patterns that do not have gflow but are still deterministic for ad-hoc reasons.

When we have such a graph-like diagram with gflow, we can efficiently convert this back into a measurement-free (i.e. ancilla-free) quantum circuit. Conversely, when we have a unitary quantum circuit, we can write this as a graph-like diagram with gflow.

By using ZX-diagrams as an intermediary, we can translate back-and-forth between representing a computation as a circuit vs a deterministic

measurement fragment:
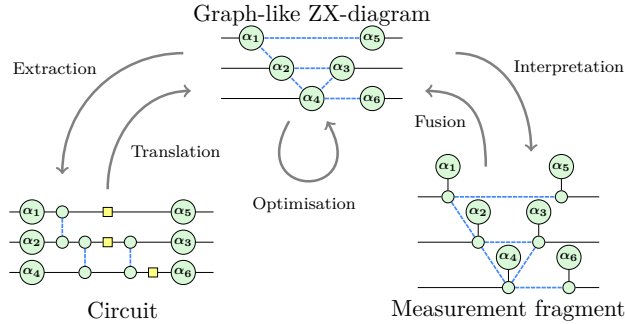
In Chapters 5 and 7 we have strategies to optimise ZX-diagrams, and in particular to remove spiders from the diagram. In this chapter we have seen that these rewrites based on local complementation and pivoting preserve the existence of gflow on the underlying open graph. This allows us to think of these rewrites not just as spider-removing simplifications on ZX-diagrams, but as 'qubit-removing' (or 'measurement-removing') simplifications on deterministic measurement patterns. Essentially, these rewrites, even a variant like the gadget pivot, can be seen as ways to get rid of qubits measured in a Clifford angle. This is yet another way in which we can think of the Clifford operations as being 'easy': when we measure a qubit in a Clifford angle, we can find an equivalent pattern where that qubit didn't exist in the first place.

8.7 Summary: What to remember

1. In measurement-based quantum computing we use quantum measurements as an integral part of the computation, instead of just using them at the end to get the result out of the final computation.

2. There are many different variants on measurement-based quantum computing, but one particularly well-studied model that lies on the extreme end of doing all computations via measurements is the *one-way model*.

3. In the one-way model you start by preparing a graph-state. Then single-qubit measurements are performed in rotated bases in order to do computations. Later measurement angles are allowed to depend on previous measurement outcomes.

4. Just like how a quantum circuit is built out of gates, we can view a one-way model computation as being built out of *measurement fragment*.

5. An important consideration for a measurement fragment is whether it is *deterministic*, meaning that regardless of all the intermediate mea-

surement outcomes, the overall linear map that is performed is the same.

6. One particular strategy for ensuring a fragment is deterministic is for the underlying open graph to have a property called *gflow*.

7. We can view measurement fragments as graph-like ZX-diagrams, and vice versa we can interpret a graph-like ZX-diagram as a measurement fragment without a specified order of measurement.

8. There is an efficient strategy to translate a circuit into a measurement fragment with gflow.

9. There is an efficient strategy to translate a measurement pattern with gflow, or equivalently a ZX-diagram with gflow, back into a circuit.

10. While the one-way model with just measurements in the XY plane is already universal, allowing measurements in all three principal planes allows for more compact patterns. There is a notion of gflow that works for measurements in all three planes, and the circuit extraction algorithm can be adapted to work for that.

11. The ZX rewrite strategy of Chapter 7 preserves the existence of 3-plane gflow.

8.8 References and further reading

The one-way model The one-way model was originally proposed by Raussendorf and Briegel (2001), where they also immediately showed how the cluster state can be used as a universal resource for quantum computation. The cluster state itself was introduced by the same authors in a paper earlier that year (Briegel and Raussendorf, 2001). In a paper the following year they formalised the model underlying the one-way model more, and they showed that a Clifford computation can be performed in a single time-step (Raussendorf and Briegel, 2002). These authors revisit computing on cluster states in more detail in Raussendorf et al. (2003) which also coins the term 'measurement-based quantum computing. The authors together with several others review all the progress made on the one-way model some years later (Briegel et al., 2009).

Measurement patterns and fragments The notion of a measurement pattern was formalised in Danos et al. (2007), and how to compose these together as fragments and 'normalise' them into a standard format where we first do all state preparations and only then do measurements in Danos et al. (2009). How to represent these patterns in ZX was first described by Duncan and Perdrix (2010).

Flow and determinism The notion of *causal flow* was introduced by Danos and Kashefi (2006). This was later extended to *generalised flow*, or gflow, by Browne et al. (2007), where they show that certain graph states do not allow causal flow, but do allow gflow. This paper defined both the single-plane gflow and three-plane gflow, though our presentation of three-plane gflow is a bit simplified. Furthermore, Browne et al. (2007) showed that a measurement pattern has a gflow if and only if the pattern is *uniformly*, *strongly* and *stepwise* deterministic. Uniformity means that the determinism holds for any choice of measurement angle. Strong determinism means that every measurement outcome happens with equal probability. The stepwise determinism means that we retrieve a deterministic pattern after every single measurement. Or in other words, that the necessary corrections depend only on a single previous outcome. Hence, while there are deterministic patterns that do not have gflow, when they are sufficiently 'regular' and the reason for their determinism 'not too complex', it will have a gflow. Finally, Browne et al. (2007) also introduced *Pauli flow*, which can deal with patterns where some qubits are designated to be measured in a Clifford angle. Hence, the uniformity condition doesn't apply to these qubits. Because measurements in a Clifford angle essentially belong to two principal planes of the Bloch sphere at the same time, they can be corrected, and be part of correction sets, in more ways than general measurements, so that their gflow conditions can be less strict.

Circuit extraction An algorithm for extracting a circuit without ancilla from a measurement pattern with single-plane gflow was given by Miyazaki et al. (2015). This was based on finding a *path-cover* of the open graph, and then resolving 'acausal' CZ gates by pushing them through other gates until they were acting at the same point in time on the two qubits. Independently, a different algorithm phrased in the language of ZX was given in Duncan et al. (2020). This was then extended to work for three-plane gflow in Backens et al. (2020). The procedure we show in this chapter is based on that article.

Rewriting measurement patterns How a local complementation changes a graph state was noted in Elliott et al. (2008), and specifically for pivots in Mhalla and Perdrix (2013). That these operations preserve gflow was shown in Duncan et al. (2020), that they preserve 3-plane gflow in Backens et al. (2020) and that they preserve Pauli flow in Simmons (2021). This chapter is based on Duncan et al. (2020) and Backens et al. (2020).

Other measurement-based models The one-way model is not the only way to do measurement based quantum computing. We can also do universal computation using AKLT states (Wei et al., 2011), a state based on phase gadgets (Kissinger and van de Wetering, 2019), or generalisations of graph states known as *hypergraph states*, that are built using multi-qubit controlled-Z operations (Miller and Miyake, 2016; Takeuchi et al., 2019). For a ZX perspective on hypergraph states, see Lemonnier et al. (2020). We can also view the *lattice surgery* operations we will explore in Chapter 11 as a measurement-based model, see de Beaudrap et al. (2020b).

<div align="center">

9

Controlled gates and classical oracles

</div>

In Chapter 7 we looked at universal quantum circuits built from low-level gates: CNOTs, phase gates, and Hadamards. These are the types of gates you might be able to implement on the hardware level of a quantum computer. However, many useful quantum algorithms will contain higher-level primitives: classical oracles, controlled unitaries, adders, etc. In this chapter we will see how we can translate these higher-level constructions into the lower-level ones, and how we can reason with the higher-level constructions directly. To do this we will introduce a new derived generator for ZX-diagrams: the **H-box**. This is a spider-like contraption that allows us to easily represent AND-like gates, just like how the X-spider allows us to represent XOR-like gates.

9.1   Controlled unitaries

Let's take a look first at controlled unitaries. First, what is a controlled unitary? Given some n-qubit unitary U, the k-controlled U is an $(n+k)$-qubit unitary that applies U to the bottom n qubits if and only if the top k qubits are all in the $|1\rangle$ state. In terms of quantum circuit notation we write the following:

$$
k\left\{\begin{matrix}\vdots\\\vdots\end{matrix}\right.\quad n\left\{\begin{matrix}\vdots\,U\end{matrix}\right.\quad::\quad|x_1,\ldots,x_k\rangle\otimes|\psi\rangle\mapsto\begin{cases}|x_1\ldots,x_k\rangle\otimes U|\psi\rangle & \text{if } x_1\cdots x_k=1\\|x_1,\ldots,x_k\rangle\otimes|\psi\rangle & \text{otherwise}\end{cases}
$$

$$(9.1)$$

We refer to the top k qubits here as the 'control wires', with the bottom n qubits being the 'target wires'.

   We use quantum circuit notation here, instead of ZX-diagrams, because

there is no easy way to represent controlled unitaries in ZX. We'll see later in this chapter how to remedy this.

Example 9.1.1. We have already encountered a couple of simple examples of controlled unitaries: the CNOT and CZ gates. In the CNOT gate we have added one control to the single-qubit X gate, while for the CZ gate we have added one control to the single-qubit Z gate.

When dealing with controlled unitaries, we will often talk about the unitary 'firing' or 'not firing'. By this we mean whether the qubits on the control wires are in the state that makes U get applied on the target wires, or whether the identity happens instead. For instance, in the CNOT gate, the NOT gate fires when the top qubit is in the $|1\rangle$ state. When analysing the logic of a circuit containing controlled gates, we can often reduce the analysis to a case distinction where we consider the situation where the gate fires, and where it does not fire (see the next proposition for an example of this). As an example of this kind of logic, let's prove the following proposition.

Proposition 9.1.2. Let U and V be n-qubit unitaries. Then conjugating the controlled U gate by V is the same thing as controlling the $VUV^\dagger$ gate:
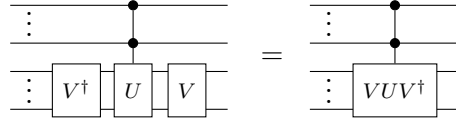
Proof Note that the controlled unitary on the left-hand side fires iff that on the right-hand side fires. Let's check the two cases of firing and not firing to see that they agree on both cases. If the gate does not fire, then the right-hand side is the identity. On the left-hand side, if U doesn't fire, then the $V^\dagger$ and V cancel so that it also gives the identity. If instead the gates do fire then both sides implement the $VUV^\dagger$ gate on the bottom qubits. $\square$

Example 9.1.3. We have seen that we can construct the CZ gate by conjugating the bottom qubit of a CNOT by Hadamards. This works because $HXH = HXH^\dagger = Z$ so that the previous proposition applies.

9.1.1 The Toffoli gate

A controlled unitary of particular importance is the *Toffoli* gate. This is the controlled CNOT gate, or equivalently, the two-qubit-controlled X gate.

Definition 9.1.4. The **Toffoli gate** is the three-qubit unitary defined by $|x, y, z\rangle \mapsto |x, y, (x \cdot y) \oplus z\rangle$. That is, it XORs the value of the third qubit

with the AND of the first two qubits. In quantum circuit notation we write the Toffoli gate as:

$$\text{TOF} \quad = \quad \text{[circuit diagram]}$$
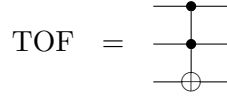
The Toffoli gate is important for a number of reasons. First, it is *universal for classical reversible logic*. Let's unpack that statement a bit. What do we mean by classical reversible logic? A classical function is any map $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$. Such a map only corresponds to a unitary via $U_f |\vec{x}\rangle = |f(\vec{x})\rangle$ iff f is a bijection, and hence has an inverse. When a classical function has an inverse, we say it is **reversible**. As it turns out, any classical reversible function can be decomposed into a sequence of *generalised* Toffoli gates. These are the n-qubit generalisations of the Toffoli gate that control a NOT gate on $n-1$ wires (hence, the NOT and CNOT gates also count as such generalised Toffoli gates). As we will see later, if we allow for clean ancillae initialised in the 0 state that also have to be brought back to the 0 state at the end of the circuit, then we can implement any reversible function using just (the three-bit) Toffoli and the NOT gate. It is in this sense that the Toffoli gate is universal for classical reversible logic.

Second, the Toffoli gate is essentially the 'quantum AND gate': if we input a $|0\rangle$ on the input of the target, and post-select the controls to $|+\rangle$ we can easily verify that:

$$
\begin{array}{l}
|x\rangle \;\text{---}\!\bullet\!\text{---}\; \langle+| \\
|y\rangle \;\text{---}\!\bullet\!\text{---}\; \langle+| \quad \propto \quad |x \cdot y\rangle \\
|0\rangle \;\text{---}\!\oplus\!\text{---}
\end{array}
$$

Since we also have a NOT gate, we can combine these two gates to perform arbitrary classical logic in a quantum circuit (although because the above construction uses post-selection, we will have to be a bit smarter about this).

Third, which is related to the previous point, we can use Toffoli gates to add control wires to other unitaries:

$$\text{[circuit diagram]} \qquad = \qquad \text{[circuit diagram]}$$
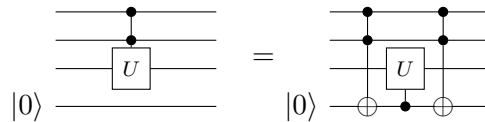
If it is not clear why this works, don't worry, we will analyse constructions like this in more detail in Section 9.4.2. The point we want to make here is that Toffoli gates are very useful, and that it is worthwhile understanding how to construct them.

9.1.2 Diagonal controlled gates and phase polynomials

Before we will look at how to construct Toffoli gates, it will first be helpful to take a bit of a detour. As the Toffoli is just an X gate with two controls, we can conjugate its target qubit by Hadamards to reduce this to a Z gate with two controls; see Proposition 9.1.2. The resulting **CCZ gate** is a diagonal unitary, which makes it easier to think about in some ways. In this section we will see how we can construct the CCZ gate, and related unitaries, using phase gadgets.

But first, let's start by looking at a simpler gate: the CZ.

This gate applies a Z gate—i.e. $Z|y\rangle = (-1)^y|y\rangle$—to the second qubit, if the first qubit is in the $|1\rangle$ state, and applies the identity otherwise. It turns out we can efficiently write down this operation as $\mathrm{CZ}|x,y\rangle = (-1)^{x\cdot y}|x,y\rangle$ (convince yourself of this by plugging in different values of $x, y \in \{0,1\}$). Interestingly, $(-1)^{x\cdot y}$ is not the only way in which we can represent the phase function of CZ.

We've seen two ways to write down a CZ gate as a ZX-diagram:

$$\tag{9.2}$$

We recognise the right-hand side as a circuit consisting of two phase gates and one two-qubit phase gadget. We can hence represent the action of this circuit as $|x,y\rangle \mapsto e^{i\frac{\pi}{2}(x+y-x\oplus y)}|x,y\rangle$.

As we already saw in Section 7.1.3, the reason these two diagrams, corresponding to the expressions $(-1)^{x\cdot y}$ and $e^{i\frac{\pi}{2}(x+y-x\oplus y)}$ are equal is because we have:

$$x \cdot y = \frac{1}{2}(x + y - x \oplus y) \qquad \forall x, y \in \{0, 1\}. \tag{9.3}$$

This formula allows us to construct other diagonal controlled gates. For instance, if we want to construct the controlled $Z(\alpha)$ gate, we first realise that this gate has a phase function of $e^{i\alpha(x\cdot y)}$, which we can transform using Eq. (9.3) to $e^{i\frac{\alpha}{2}(x+y-x\oplus y)}$. Then we can simply write this as a phase polynomial circuit:

$$\mathrm{CZ}(\alpha) \quad = \quad \tag{9.4}$$

But what if we want to construct multiply-controlled gates? For the CCZ gate, the controlled CZ gate, the phase function is $e^{i\pi(x\cdot y\cdot z)}$, so we need some way to decompose $x \cdot y \cdot z$ into a phase polynomial. We can do this by

generalising Eq. (9.3) to more bits:

$$(x \cdot y) \cdot z \overset{(9.3)}{=} \frac{1}{2}(x \cdot z + y \cdot z - (x \oplus y) \cdot z)$$

$$\overset{(9.3)}{=} \frac{1}{4}(x + z - x \oplus z + y + z - y \oplus z - (x \oplus y + z - x \oplus y \oplus z))$$

$$= \frac{1}{4}(x + y + z - x \oplus y - x \oplus z - y \oplus z + x \oplus y \oplus z) \qquad (9.5)$$

So the phase polynomial of the CCZ gate is $e^{i\frac{\pi}{4}(x+y+z-x\oplus y-x\oplus z-y\oplus z+x\oplus y\oplus z)}$. We can hence represent it using the following ZX-diagram:

$$\text{CCZ} \quad = \quad \qquad \qquad \qquad \qquad \qquad (9.6)$$

We can generalise Eq. (9.5) to work for any number of k bits. This will result in an expression with $2^k - 1$ XOR terms and a constant of $1/2^{k-1}$. Hence, the phase polynomial of an n-controlled $Z(\alpha)$ gate consists of $2^{n-1} - 1$ phase gadgets with a phase of $\pm\alpha/2^{n-1}$. On the one hand this is great, as it means we can represent arbitrary controlled phase gates using a phase polynomial circuit. On the other hand, we need exponentially many exponentially small phase gates if the number of controls is high. Luckily there are ways around this exponential cost, as we will explore in this chapter.

For now though, let's see how we can use these multiply-controlled phase gates to construct arbitrary diagonal unitaries. Note that the matrix of an $(n-1)$-controlled $Z(\alpha)$ gate is $\text{diag}(1, \ldots, 1, e^{i\alpha})$. This matrix is diagonal, and it only applies an $e^{i\alpha}$ phase if the input is in the $|1\cdots1\rangle$ state, and applies a trivial phase of 1 otherwise. So the matrix consists of all 1's on the diagonal except for an $e^{i\alpha}$ in the bottom corner. The reason it appears in the bottom corner is because we are controlling the phase on the all-1 state. By conjugating some of the qubits with a NOT gate we can instead make it controlled on some other specific bit-string. This moves the position of the $e^{i\alpha}$ on the diagonal. For instance, for a $CZ(\alpha)$ gate, the matrix is $\text{diag}(1, 1, 1, e^{i\alpha})$. If we conjugate the second qubit by NOT gates, the $e^{i\alpha}$ fires on the $|10\rangle$ state instead, and the matrix looks like $\text{diag}(1, 1, e^{i\alpha}, 1)$. Another way of saying this, is that an n-controlled $Z(\alpha)$ gate on an $n+1$ qubit circuit which is conjugated by NOT gates implements a 'Dirac delta' diagonal unitary $U_{\vec{y}}|\vec{x}\rangle = e^{i\alpha\delta_{\vec{y}}(\vec{x})}|\vec{x}\rangle$. Here $\delta_{\vec{y}}(\vec{x}) = 1$ iff $\vec{y} = \vec{x}$ and is 0

otherwise. But we can write an arbitrary phase function in terms of these delta functions!

Let $f : \mathbb{F}_2^n \to \mathbb{R}$ be some phase function. The delta functions $\delta_{\vec{y}} : \mathbb{F}_2^n \to \mathbb{R}$ form a basis for the space of phase functions, and hence we can write $f = \sum_{\vec{y} \in \mathbb{F}_2^n} \alpha_{\vec{y}} \delta_{\vec{y}}$. Hence, to implement the diagonal unitary U_f that acts as $U_f |\vec{x}\rangle = e^{if(\vec{x})}|\vec{x}\rangle$, we can multiply together unitaries implementing the delta phase functions $\alpha_{\vec{y}} \delta_{\vec{y}}$. This means we can implement an arbitrary n-qubit diagonal unitary using 2^n $(n-1)$-controlled phase gates.

Proposition 9.1.5. Any n-qubit diagonal unitary can be constructed using $O(2^n)$ $(n-1)$-controlled phase gates and NOT gates.

Each of the controlled-phase gates can be decomposed into $O(2^n)$ phase gadgets. The NOT gates conjugating some of the qubits of these controlled-phase gates can then be absorbed into the phase gadgets, so that any circuit of such NOT-conjugated controlled-phase gates can be reduced to a circuit of just phase gadgets. While we are decomposing each of the $O(2^n)$ controlled-phase gates into $O(2^n)$ phase gadgets, the phase gates acting on the same qubits combine, so that in the end the circuit only requires $O(2^n)$ of them (and not $O(2^n \cdot 2^n)$).

Proposition 9.1.6. Any n-qubit diagonal unitary can be constructed using $O(2^n)$ phase gadgets.

9.1.3 Fourier transforming diagonal unitaries

In the previous section we converted controlled phase gates to phase gadgets, but this translation goes both ways. In fact, the relationship between these two types of phase gates is closely related to the **pseudo-Boolean Fourier transform**.

A **pseudo-Boolean function** is any function $f : \mathbb{F}_2^n \to \mathbb{R}$, where $\mathbb{F}_2 = \{0, 1\}$ is the Booleans. We can decompose a pseudo-Boolean function into primitive terms in a number of ways. We already saw that we can write it in terms of delta functions, which we can treat as a set of 'maximally controlled' expressions. These delta functions could be translated into XOR terms, and hence that gives us a different decomposition:

$$f(\vec{x}) = \sum_{\vec{y} \in \mathbb{F}_2^n} \lambda_{\vec{y}} \vec{y} \cdot \vec{x}. \tag{9.7}$$

Here the $\lambda_{\vec{y}}$ are real coefficients that determine f and $\cdot$ is the dot product of bit strings, which for a fixed $\vec{y}$ represents an XOR of bits of $\vec{x}$. For instance,

if $\vec{y} = 101$, then $\vec{y} \cdot \vec{x} = x_1 \oplus x_3$. Note that in Eq. (9.7) we are treating the Booleans 1 and 0 both as Booleans and as real numbers. This decomposition contains 2^n independent parameters $\lambda_{\vec{y}}$. As f has 2^n possible inputs, we see that each pseudo-Boolean function can indeed uniquely be written in this way. The phase polynomials of Chapter 7 are examples of pseudo-Boolean functions written as XOR terms.

Instead of using XOR as the primitive function to decompose it to, we can also use AND:

$$f(\vec{x}) = \sum_{\vec{y} \in \mathbb{F}_2^n} \hat{\lambda}_{\vec{y}} \prod \vec{x}^{\vec{y}}.$$

Here $\vec{x}^{\vec{y}}$ is the bit string $x_1^{y_1} x_2^{y_2} \cdots x_n^{y_n}$ where we set $0^0 = 1$ and $1^0 = 1$. Hence, if $\vec{y} = 101$ then $\vec{x}^{\vec{y}} = x_1 1 x_3$, so that $\prod \vec{x}^{\vec{y}} = x_1 \wedge x_3$. Again, as there are 2^n independent terms in this decomposition, any pseudo-Boolean function can be written in this way.

The transformation of a pseudo-Boolean function written as sums of XOR terms to one written as sums of AND terms and back is what we call the **Fourier transform** of such a function.

This Fourier transform essentially boils down to Eq. (9.5) and its n-bit generalisation. In particular, to transform back from XOR to AND, we use its 'inverse':

$$
\begin{aligned}
(x \oplus y) \oplus z &= x \oplus z + y \oplus z - 2(x \cdot y) \oplus z \\
&= x + z - 2(x \cdot z) + y + z - 2(y \cdot z) - 2(x \cdot y) - z + 4(x \cdot y \cdot z) \\
&= x + y + z - 2(x \cdot y + x \cdot z + y \cdot z) + 4(x \cdot y \cdot z)
\end{aligned}
$$

By using this translation we can hence also write an arbitrary diagonal unitary as a circuit of controlled-phase gates where now we do not need any NOT gates and we use controlled gates acting on different numbers of qubits.

9.2 H-boxes

The representation we found of the CCZ gate in Eq. (9.6) looks a bit messy. The reason for this is that we had to translate the phase function $(-1)^{x \cdot y \cdot z}$ into a sum of XOR phase functions. This is because the Z- and X-spiders *can* directly represent these XOR phases, while they *cannot* directly represent these 'multiplicative' phases. We can solve this issue by introducing a new generator for ZX-diagrams: **H-boxes**.

We define H-boxes as follows (why we call these things H-boxes will become

clear soon enough):

$$m \left\{ \begin{matrix} \vdots \\ \vdots \end{matrix} \boxed{a} \begin{matrix} \vdots \\ \vdots \end{matrix} \right\} n \quad := \quad \frac{1}{\sqrt{2}} \sum a^{i_1 \ldots i_m j_1 \ldots j_n} |j_1 \ldots j_n\rangle\langle i_1 \ldots i_m| \qquad (9.8)$$

The sum in this equation is over all $i_1, \ldots, i_m, j_1, \ldots, j_n \in \{0, 1\}$ and a is an arbitrary complex number. Hence, an H-box represents a matrix with, up to a global factor of $\sqrt{2}$, all entries are equal to 1, except the bottom right element, which is equal to a. We have for instance

$$\boxed{a} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & a \end{pmatrix} \quad \text{and} \quad \boxed{a} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & a \end{pmatrix}. \quad (9.9)$$

Hence, in particular, when $a = -1$, the 1-input 1-output H-box is just the Hadamard:

$$\boxed{} = \boxed{-1} \qquad (9.10)$$

We can then view H-boxes as a generalisation of Hadamard gates to an arbitrary number of inputs and outputs (hence, the letter 'H'). Just as spiders with a zero phase are depicted without a phase label, we will depict H-boxes with a label of -1 without any label:

$$m \left\{ \begin{matrix} \vdots \\ \end{matrix} \boxed{} \begin{matrix} \vdots \\ \end{matrix} \right\} n \quad := \quad m \left\{ \begin{matrix} \vdots \\ \end{matrix} \boxed{-1} \begin{matrix} \vdots \\ \end{matrix} \right\} n \qquad (9.11)$$

We will call such H-boxes **phase-free**. Note that this convention means that the 1-input 1-output phase-free H-box is denoted exactly the same as the Hadamard gate in the ZX-calculus, which is fine since they represent the same matrix.

The linear maps that H-boxes represent have all the symmetries that spiders have:

$$\boxed{a} = \boxed{a} = \boxed{a} = \boxed{a} = \boxed{a} \qquad (9.12)$$

We can hence bend the wires of an H-box however we want.

We will introduce some rewrite rules for H-boxes, but for now let's check that they indeed help us accomplish our goal of having a nicer representation of a CCZ gate. First note that:

$$\boxed{}\hspace{-0.5em}\begin{matrix}<\\\end{matrix} = \frac{1}{\sqrt{2}} \sum_{x,y,z} (-1)^{x \cdot y \cdot z} |x, y, z\rangle \qquad (9.13)$$

We can use this state to represent a CCZ gate:

$$\text{CCZ} \quad = \quad \sqrt{2} \quad \text{} \tag{9.14}$$

Exercise 9.1. Verify Eq. (9.14) by plugging in computational basis states and checking that it gives the correct phase.

Exercise 9.2. The state of Eq. (9.13) turns out to be the **CCZ magic state**, meaning we can use it to construct a CCZ gate by doing some Clifford unitaries and measurements. Show that the following post-selected unitary that uses a CCZ magic state indeed implements a CCZ gate, by rewriting it to Eq. (9.14):
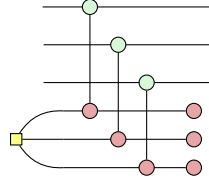
We will later introduce some rewrite rules for H-boxes that allow us to prove that this construction works regardless of the measurement outcome, by doing some Clifford corrections.

Now, we introduced H-boxes as a new 'generator' of ZX-diagrams. But we also know that ZX-diagrams were already *universal*. This means there must be some way to represent the H-boxes just using spiders. In fact, that is actually what the relations Eqs. (9.3), (9.5) and their generalisations allow us to do:

$$\tag{9.15}$$

In general, for an arity n H-box this construction requires $2^n - 1$ phase gadgets. If the parameter of the H-box is not a complex phase $e^{i\alpha}$ we can still represent the H-box using spiders, but it is a bit more complicated: see Exercise 9.7.

9.2.1 AND gates

Using an H-box we can easily represent the CCZ gate. But of course the CCZ gate is related to the Toffoli by conjugating the target by a Hadamard, so with some rewriting we can get an interesting representation of the Toffoli:

$$
\text{TOF} \quad = \quad \overset{(cc)}{=} \tag{9.16}
$$

The reason this is interesting is because it is showing directly the two components that make up the Toffoli: calculating the AND of the first two qubits, and then XORing it with the third qubit. Indeed, we can calculate:

$$
= \quad \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9.17}
$$

Hence:

$$
\propto \quad (x \cdot y)\pi \tag{9.18}
$$

We can then directly verify that Eq. (9.16) implements a Toffoli:

$$
\overset{(sc)}{\propto} \quad \overset{(9.18)}{\propto} \quad \overset{(sp)}{=} \tag{9.19}
$$

Remark 9.2.1. Since a 2-to-1 H-box followed by a Hadamard implements an 'AND gate', it is reasonable to wonder why we didn't just define an 'AND-box' as a new element of ZX-diagrams. This would make it a nice counterpart to the X-spider that implements an XOR. However, one of the main symmetries present in the Z- and X-spiders does not hold for this hypothetical AND-box (which can be verified by calculating the associated matrices):

$$
= \quad \text{but} \quad \boxed{\text{AND}} \quad \neq \quad \boxed{\text{AND}} \tag{9.20}
$$

This symmetry, known as **flexsymmetry**, *does* hold for the H-box (see Eq. (9.12)). Hence, by splitting up the AND gate into an H-box and Hadamard, we still get the benefit of having a compact representation of the AND, while

also only dealing with components that have all the symmetries we want, meaning we can still think of ZX-diagrams as undirected graphs.

9.2.2 Rules for the H-box

Of course we wouldn't be introducing a new graphical part to the ZX-calculus, if it didn't allow us to do some more rewriting! There are some H-box specific rewrite rules that we can use to reason about, for instance, controlled unitaries, and Toffoli gates.

First, let us recall that arity-1 H-boxes labelled by a complex phase are just Z-spiders:

$$\boxed{e^{i\alpha}}\!\!- \quad \propto \quad \alpha\!\!- \tag{9.21}$$

In particular, taking respectively $\alpha = 0$ and $\alpha = \pi$, we get:

$$\boxed{1}\!\!- \quad \propto \quad \circ\!\!- \qquad \boxed{}\!\!- \quad \propto \quad \pi\!\!- \tag{9.22}$$

Most of the other H-box rewrite rules we will use can be motivated by the relation between an H-box and the AND gate. To understand these it will be helpful to use multi-input AND gates:

$$\vdots\,\boxed{\text{AND}}\!\!- \quad = \quad \vdots\,\!\!\!\!\!\!\!\!\!\!\!\Rightarrow\!\!-\square\!\!-\square\!\!- \tag{9.23}$$

A rule on H-boxes we have already seen is that two Hadamard gates cancel: $-\square\!\!-\square\!\!- = -\!\!\!\!-$. Using our interpretation of multi-input AND gates (9.23) as H-boxes we can get a different view on this equation. Using Eq. (9.23) we see that two Hadamard gates in a row correspond to an AND gate with a single input, and this gate is of course the identity.

Our first new rule expresses how a sequence of ANDs can be combined into a single multi-input AND:

$$\vdots\,\boxed{\text{AND}}\!\!\!\!\!\!\!\!\!\!\!\searrow \atop \,\,\,\vdots\,\boxed{\text{AND}}\!\!- \;\; = \;\; \vdots\,\boxed{\text{AND}}\!\!- \;\; \leftrightarrow \;\; = \tag{9.24}$$

This rule can be presented a bit more generally as an *H-box fusion* rule:

$$\vdots\,\!\!\!\!\!\Rightarrow\!\!-\square\!\!-\square\!\!-\boxed{a}\!\!\!\Leftarrow\,\vdots \quad = \quad \vdots\,\!\!\!\!\!\Rightarrow\!\!\boxed{a}\!\!\!\Leftarrow\,\vdots \tag{9.25}$$

Note that whereas two spiders fuse together when they are connected by a leg, for an H-box, this connection needs to be a Hadamard edge.

An important consequence of this rule is that H-boxes absorb $|1\rangle$ states:

$$\text{(9.26)}$$

Using this we can show for instance that inputting a $|1\rangle$ on one of the controls of a Toffoli reduces it to a CNOT:

$$\text{(9.27)}$$

We will see later in (9.31) that, in contrast, a $|0\rangle$ 'explodes' an H-box into Z-spiders.

In Section 3.2.4 we saw how the interpretation of the Z- and X-spider as respectively COPY and XOR lead us to the strong complementarity rule that allowed us to push (phaseless) Z- and X-spiders through each other. This equation (3.55) involving COPY and XOR holds in exactly the same way when XOR is replaced by AND:

$$\text{(9.28)}$$

We can directly translate this into a rule involving Z-spiders and H-boxes:

$$\text{(9.29)}$$

By pushing the Hadamard through the Z-spider and cancelling some Hadamards we can also present this in a format that is often more convenient:

$$\text{(9.30)}$$

As in (3.69), the right-hand side of both of these equations is a fully connected bipartite graph. Note that as a special case of the second equation (taking $n = 0$) we get the following useful state-copy rule, which is a counterpart of (9.26):

$$\text{(9.31)}$$

Here in the last step we dropped the scalar subdiagram, as it only contributes

a (usually irrelevant) non-zero scalar. Using this rule we can show that inputting a $|0\rangle$ on a control wire of a Toffoli reduces it to an identity:

$$(9.32)$$

Exercise 9.3. Using Eqs. (9.30) and (9.31) (and the standard ZX rules), prove that two CCZs in a row equal the identity:

Exercise 9.4. Prove that we can commute a NOT gate through an H-box, resulting in a CZ on the other side:

Hint: Unfuse the π phase onto its own spider, and then apply Eq. (9.30).

Exercise 9.5. In Exercise 9.2 we saw that with post-selection, a CCZ gate can be implemented by using a magic state and post-selection. However, this post-selection is not necessary, as the other measurement outcomes can be corrected by applying the right gates in the future. For instance, if we get the $\langle 1|$ outcome on the first measured qubit instead, we can correct this with a CZ gate on the second and third output qubits. Show this by proving that:

Bonus exercise: figure out what the correction operator is when both the first and second measured qubits get the $\langle 1|$ outcome.

Another consequence of Eq. (9.30) is that the identification of a 1-labelled H-box with a Z-spider of (9.22) can be generalised to higher arity as follows:

$$(9.33)$$

Figure 9.1 The basic rules for H-boxes

Let us now introduce the last pair of AND-inspired rewrite rules for H-boxes. These are based on the following identities:

 (9.34)

The first is quite self-evident: if we copy a value and then AND those values together, it is the same thing as doing nothing to the value. The second requires a bit more explanation. It expresses a fact about the possible ways that AND can return $|1\rangle$. Indeed, as a linear map, we can write AND as $|0\rangle\langle00| + |0\rangle\langle01| + |0\rangle\langle01| + |1\rangle\langle11|$, and hence post-selecting the output of AND with $\langle1|$ we calculate $\langle1|\,\mathrm{AND} = \langle11|$.

Writing the ANDs as H-boxes and simplifying the expressions a bit we get the following rewrite rules:

 (9.35)

Note that using (9.22) we could also have written the second equation of (9.35) as:

 (9.36)

The rules introduced so far are summarised in Figure 9.1.

We have now covered all the 'AND inspired' H-box rules. In fact, these rules, together with the phase-free ZX-calculus rules we have been using throughout the book (that is, those of Figure 4.1), are already complete for a useful fragment of quantum computing. Namely, if we restrict ourselves to phase-free H-boxes, and spiders that only have 0 or π phases, then we can represent precisely those linear maps that can be built by post-selected quantum circuits consisting of Toffoli and Hadamard gates. It turns out that Toffoli-Hadamard circuits are already enough to perform arbitrary quantum computations (see Section* 10.7.2 for more details on that), and hence this fragment of diagrams can represent many interesting maps. Proving that the phase-free rules of Figures 9.1 and 3.1 are complete for this fragment

is quite difficult (see the References of this chapter for some notes), but let's note that this completeness does say something interesting on how to reason about quantum computations: the ZX rules we have been using are complete for Clifford diagrams (Theorem 5.5.7), while the new rules for H-boxes of Figure 9.1 are all directly related to Boolean identities. Hence, somehow classical logic plus reasoning with Cliffords gives us the full power of quantum computing.

9.2.3 More rules for the H-box

The rewrite rules we have now covered are the ones that come up the most often. But there is another set of rewrite rules that deal specifically with H-boxes that are not labelled by -1.

The first two of these rules allow us to perform arithmetic with H-boxes:

$$(9.37)$$

We call these the **multiply rule** and the **average rule**. When a and b are complex phases, the multiply rule is just an instance of the adding of phases when spiders fuse, cf. (9.21) and (3.35). The average rule has no counterpart in the standard ZX-calculus. In practice, the average rule doesn't turn out to be useful very often. We include it here merely for completeness' sake.

The multiply rule can be generalised to H-boxes of arbitrary arity:

$$(9.38)$$

I.e. when two H-boxes are connected to exactly the same set of Z-spiders, then we can fuse the H-boxes together. With the rules we have seen before, the proof of this generalisation is straightforward:

$$(9.39)$$

Using this rule we can prove that two controlled-phase gates combine together:

This should all look quite familiar: Eq. (9.38) is like the phase gadget fusion

rule of Section 7.1.2. While a phase gadget adds a phase depending on the XOR of the inputs, a controlled-phase gate build using an H-box adds a phase based on the AND of the inputs.

Then there is only one more rule: the **introduction rule**:

$$\tag{9.40}$$

We call it the introduction rule, because it allows you to introduce additional edges to an H-box (at the cost of copying the H-box). As do many of the previously introduced rules, it has a generalisation to H-boxes of arbitrary arity:

$$\tag{9.41}$$

Exercise 9.6. Prove Eq. (9.41), using Eq. (9.40) and the previously introduced H-box and ZX rules.

Most of the use-cases of this rule are when it is applied from right-to-left. Indeed, it is a close cousin of the multiply rule (9.38). Both rules target pairs of H-boxes connected to the same set of Z-spiders, although in the case of the introduction rule, they must also differ by a NOT gate on one of the connections, and have the same label. As an example, we can use the introduction rule to prove that if we apply both a controlled-phase gate, and a NOT-conjugated controlled-phase gate that this reduces to just a simple phase gate:

$$\tag{9.42}$$

As noted above, the 'AND inspired' rules together with the ZX-calculus rules are complete for diagrams generated by Toffoli and Hadamard gates. When we add these three additional rules, multiply, average and introduction, we get a rule set that is complete for *all* diagrams. The ZH-calculus, being a superset of the ZX-calculus, is also universal. Hence, we can, in principle, replace all reasoning about qubit linear maps with diagrammatic reasoning. Whether it is beneficial to do so of course depends on the situation.

Exercise* 9.7. We allowed H-boxes to be labelled by an arbitrary complex number, but it turns out that we can represent all of these using just H-boxes

which are labelled by a complex phase. You may ignore scalar factors in this exercise.

a) Show that $\boxed{0}\!\!-\;=\;\circ\!\!-$. Hint: Use the average rule with $a = 1$, $b = -1$.

b) Show that for any $0 \leq r \leq 1$ we can find an α such that we can represent the r-labelled H-box using an $e^{i\alpha}$-labelled H-box and an $e^{-i\alpha}$-labelled H-box.

c) Show that $\boxed{a}\;\boxed{1/a}\!-\;=\;\boxed{a}\!\!-\!\!(\pi)\!\!-$. Hint: use the multiply rule and the 'zero wire version' of Eq. (9.41).

d) Show that we can represent (up to non-zero scalar) an arbitrary a-labelled H-box using spiders and complex-phase labelled H-boxes. Hint: first write a in the polar decomposition $a = re^{i\theta}$ for some $r \geq 0$ and $\theta \in \mathbb{R}$. Then make a case distinction based on whether $r > 1$ or not.

9.2.4 Constructing controlled unitaries using H-boxes

A useful feature of the ZH-calculus is that it allows us to quite easily see how to make a controlled-unitary out of a unitary given as a ZH-diagram. This is perhaps most easily demonstrated by the difference between a CZ and a CCZ gate in the ZH-calculus:

$$CZ \;\; \propto \;\; \quad\quad\quad CCZ \;\; \propto \quad\quad\quad\quad (9.43)$$

This suggests a general procedure for adding a control qubit: identify which H-box 'activates' the application of your gate, and add another wire to it which connects to a Z-spider on your control qubit. Sometimes, one has to work a bit to uncover the correct H-box. For instance, to see how a Z gate relates to a CZ, we unfuse its phase:

$$-(\pi)-\;\;\overset{(sp)}{=}\;\;\overset{(\pi)}{-\!\circ\!-}\;\;\overset{(9.22)}{\propto}\;\;\quad\quad\rightsquigarrow\quad\quad (9.44)$$

This procedure also works for making controlled-phase gates if the phase is something other than π:

$$\mathrm{CZ}(\alpha) \;\; \propto \;\; \boxed{e^{i\alpha}} \quad\quad\quad\quad (9.45)$$

For diagrams containing X-spiders we will usually have to convert these to Z-spiders using (cc) in order to see where we should add the control wire.

For instance, to go from a CNOT to a CCNOT (Toffoli):

$$\text{(9.46)}$$

Note that we here added a control wire to the 'middle' H-box, but left the Hadamards on the qubit wire alone. This is a general rule for constructing a controlled diagram. For instance, it might be tempting to define a controlled-Hadamard as follows:

$$\text{(9.47)}$$

While this does indeed implement a Hadamard gate when the control qubit is in the $|1\rangle$ state, it does not reduce to the identity when the control qubit is $|0\rangle$:

$$\text{(9.48)}$$

To construct the actual controlled-Hadamard we need to find the 'hidden' H-boxes in the Hadamard gate. The way we do this is by using its decomposition into Euler angles:

$$\text{—}\square\text{—} \;=\; e^{-i\frac{\pi}{4}}\;\text{—}\frac{\pi}{2}\text{—}\frac{\pi}{2}\text{—}\frac{\pi}{2}\text{—} \tag{9.49}$$

We can now make each of these phase gates into controlled phase gate using (9.45). When transforming this Euler decomposition into its controlled version, the ignorable global phase $e^{-i\frac{\pi}{4}}$ becomes a local phase that must be taken into account. This is in fact another instance of finding the hidden H-boxes of the diagram, as a scalar is just an H-box with zero wires. We hence get the following transformation:

$$\text{(9.50)}$$

Where in the last step we used the identity $e^{i\frac{\pi}{2}} = i$.

While this procedure works and gives the correct diagram for a controlled-Hadamard, it is not the most efficient implementation of a controlled-Hadamard. A better version is realised by making the observation that if we only control the middle phase-gate and the global phase of (9.49) that we get a diagram

that implements a Hadamard when the control is $|1\rangle$, and implements an X gate otherwise:

$$
\text{(9.51)}
$$

Hence, to make this a controlled-Hadamard, we need to add an X gate on the target wire to cancel the already existing X gate, but doing this will result in the wrong unitary being implemented when the control is $|1\rangle$. To remedy this error we add another gate to the circuit: a CNOT (i.e. a controlled-X gate). We hence arrive at the final controlled-Hadamard circuit:

$$
\text{(9.52)}
$$

Note that we get the $-\frac{\pi}{2}$ X-phase by combining the first $\frac{\pi}{2}$ phase of (9.51) with the added π phase coming from the *X* gate. The gate (9.52) is indeed what one would find for a controlled-Hadamard in a standard textbook (although if one starts with a different Euler decomposition of the Hadamard gate, one might get a CZ gate instead of a CX gate, along with some other permutations of the gates). We can further decompose the 'controlled-i' gate using what we have seen in Section 9.1.2.

Exercise 9.8. By decomposing and simplifying Eq. (9.52) even further, find an implementation of the controlled-Hadamard gate that requires just two T gates, and one CNOT gate.

Exercise 9.9. Construct an implementation of the controlled-Hadamard gate using just one controlled-phase gate, but starting with the Euler Decomposition $\longrightarrow\!\square\!\longrightarrow \;=\; e^{i\frac{\pi}{4}} \; -\boxed{-\frac{\pi}{2}}\!-\boxed{-\frac{\pi}{2}}\!-\boxed{-\frac{\pi}{2}}-$, instead of with Eq. (9.49).

It is currently not clear how one would relate (9.52) and the more complicated (9.50) via an intuitive diagrammatic transformation (as the calculus is complete, there is a set of graphical rewrites that transforms one into the other, but this is likely to be a complicated affair). So how would one find (9.52)? The crucial observation is that only controlling a single phase in the diagram, instead of all three, already resulted in a gate close to the one we desired. The remainder of the construction was then to keep adding simple gates until we get the exact gate we wanted. Experience shows that this method of experimentation and trial-and-error is often successful.

Let's demonstrate this with one more often-encountered controlled unitary: the **controlled swap** (also sometimes called the **Fredkin gate**). Our starting point is the implementation of a swap using three CNOTs:

(9.53)

We could make this controlled by transforming each of the CNOTs into a Toffoli. However, just as with the controlled-Hadamard, we realise that if we 'deactivate' the middle CNOT, that the outer CNOTs cancel each other, and hence it suffices to add a control to the middle CNOT:
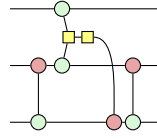
9.3 Reversible Logic synthesis

An important part of many quantum algorithms are **classical oracles**. These are classical functions that are performed on a quantum state (a state that is often in a superposition of many computational basis states). For instance, Shor's algorithm consists of two components: a classical oracle performing modular exponentiation followed by a quantum Fourier transform. In terms of gate cost, the classical oracle is by far the most expensive part (the quantum Fourier transform can be implemented quite efficiently). In Grover's algorithm it is again the classical oracle that pinpoints which elements we are interested in that is the most expensive to implement. As classical oracles form such an important part of these algorithms (and many others), we better understand very well how to actually implement these on quantum computers.

The first step is to realise that usually the function we want to implement is not reversible, so that we can't implement it directly as a unitary. We can however make it reversible by adding some additional scratch space.

Definition 9.3.1. Let $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be some classical function. Its **reversibilisation** $f_r : \mathbb{F}_2^{n+m} \to \mathbb{F}_2^{n+m}$ is defined as $f_r(\vec{x}, \vec{y}) = (\vec{x}, \vec{y} \oplus f(\vec{x}))$. Here the XOR $\oplus$ acts componentwise on the bit string.

It is clear that f_r is always reversible, as it is its own inverse.

Definition 9.3.2. A **classical oracle** for $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ is an $(n+m)$-qubit unitary U_f given by $U_f |\vec{x}, \vec{y}\rangle = |f_r(\vec{x}, \vec{y})\rangle = |\vec{x}, \vec{y} \oplus f(\vec{x})\rangle$.

Example 9.3.3. The classical oracle for the NOT operation is the CNOT gate, and the classical oracle for the AND operation is the Toffoli gate.

So the question is: *how do we efficiently construct reversible functions using simple gates?* It turns out that this question has many different answers depending on what your requirements are. The field of reversible circuit synthesis is vast, and we will only be scratching the surface in this section.

To start, we will consider two different approaches to reversible circuit synthesis. First, in Section 9.3.1 we will suppose that we are given a Boolean function as a concrete collection of simple Boolean operations like AND, XOR and NOT. We then find how we can calculate this function in a quantum circuit by doing each of these simple operations in turn. In Section 9.3.2 we will study reversible functions in a more abstract setting, where we don't know how it is constructed, we just know its truth table. We will see that we can decompose a reversible function into multiple-controlled Toffoli gates (we study how we can efficiently implement these in Section 9.4).

9.3.1 Synthesis of Boolean formulae

So let's suppose we have some classical function $f : \{0,1\}^n \to \{0,1\}^m$ that we want to implement. To simplify our life we will assume that we want the output of f to appear on some additional bits. That is, we have our register of bits $\vec{x}$ that are our inputs, and then we also have a supply of bits given to us in the 0 state. Some of these bits will be used to store intermediate computations, while others will be used as the final output. So, in total, we are looking for a unitary U that implements $U|\vec{x}, \vec{0}, \vec{0}\rangle = |\vec{x}, f(\vec{x}), \vec{0}\rangle$. Here the first register of bits stores the output, while the other register was just used during the computation. Note that it is important that we reset these 'helper bits' to 0 when we are done with them. As long as we stick to classical computations their state doesn't matter, but as soon as superpositions of states are involved, they will cause interferences that we don't want.

Let's look at a small example to make this a bit more concrete. Suppose we want to calculate the three-bit function $x_1 \wedge x_2 \wedge x_3$. We can split this up into two operations acting on fewer bits, by first calculating $z_1 := x_1 \wedge x_2$, and then calculating $z_2 := z_1 \wedge x_3$. This final bit z_2 then carries our output. So this results in the following circuit:
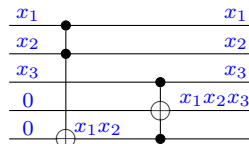
Here we write $x_1 x_2$ for $x_1 \wedge x_2$ as a shorthand. This indeed calculates the function we want, but we also have the outcome $z_1 = x_1 x_2$ still floating around. We get rid of this by 'undoing' the operations done to it. As a Toffoli is its own inverse, this is easy enough:

$$(9.54)$$
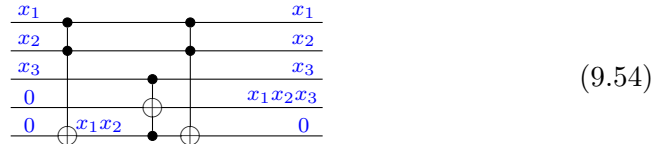
In general, let's assume that f is given to us as a sequence of AND, OR and NOT operations since of course any Boolean function can be decomposed into these operations. For simplicity we will assume that $m = 1$, i.e. that f only has a single output we care about, although the constructions we will talk about can be easily generalised to multiple outputs.

We can then interpret f as a DAG: a **directed acyclic graph**. In this graph, the input variables are the vertices at the start, and all the other vertices correspond to operations done to these variables or intermediate results. There is a directed edge from vertex v to w when the operation w uses the outcome v. For instance, suppose that f is the MAJ function on 3 bits that calculates whether at least two of the bits (i.e. the majority) are 1. One possible way we can decompose MAJ into more fundamental operations is $\mathrm{MAJ}(x_1, x_2, x_3) = (x_1 \wedge (x_2 \vee x_3)) \vee (x_2 \wedge (x_1 \vee x_3))$. The DAG corresponding to this decomposition is:
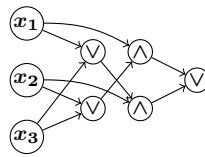
When we have this DAG, translating it to an implementation on a circuit is straightforward. We allocate for each internal vertex a bit prepared in the 0 state, we apply the operations in an order compatible with the DAG (that is, we only apply an operation once we have done the operations associated to its parents first), and at the end once we have calculated what we wanted to calculate, we undo all the operations in the reverse order on all the extra bits we used.

For the DAG above this could for instance result in the following circuit:

$$\begin{array}{c}\text{(circuit diagram)}\end{array} \tag{9.55}$$
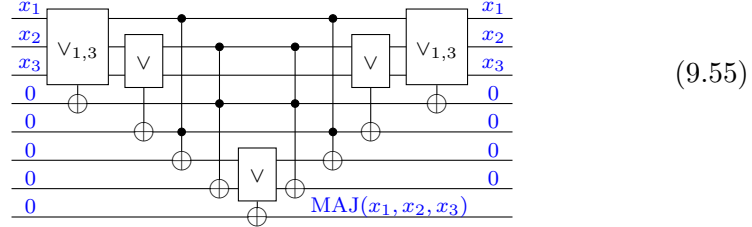
Note that the box with a $\wedge$ can be implemented using a Toffoli and some NOT gates using de Morgan's rule.

We see that each operation is applied to a fresh 0 bit, before finally getting the calculation we want in the final bit. We then repeat all the operations we have done in reverse order to undo our temporary calculations.

We don't actually have to wait until the end to undo operations. We can do this as soon as an intermediate calculation is no longer needed. After this uncomputation this bit is then put back into the 0 state, so that we can reuse it for additional computations. This hence results in needing fewer additional bits. For instance:

$$\begin{array}{c}\text{(circuit diagram)}\end{array} \tag{9.56}$$
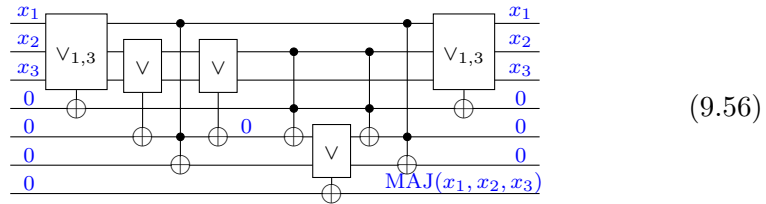
Here we could uncompute one intermediate calculation early in order to save one bit in comparison to Eq. (9.55).

The number of additional bits we need to calculate a function corresponding to a DAG hence doesn't depend on the number of vertices in the DAG, but rather on the amount of computations we have to 'keep in memory'. But suppose we want to reduce the number of additional bits as much as possible, could we do even better? In order to do so, we would need to free up memory that contains computations that we will need later. This means that we will have to recompute these when needed. Finding optimal trade-offs in uncomputing the right things and allocating the bits you have smartly is an interesting problem, but also a bit beyond what we can discuss in this book. See the References of this chapter for some more pointers.

But there is another trick we can exploit to reduce the cost of implementation. This is based on the observation that calculating XORs is much cheaper than calculating ANDs, since calculating an AND requires a Toffoli, while

calculating an XOR requires a CNOT. So instead of allocating a new bit for every operation, we can decide to only allocate a bit for every AND, and do all the XORs 'in place', uncomputing these immediately after we are done using the outcome. This means we will need to do more XOR operations, but will require less additional bits.

For example, another way we could write the MAJ function is as

$$\text{MAJ}(x_1, x_2, x_3) = x_2 \oplus ((x_1 \oplus x_2) \wedge (x_2 \oplus x_3)) \tag{9.57}$$

(to see why this works, do a case distinction on x_2). We can then store $x_1 \oplus x_2$ in the x_1 bit, store $x_2 \oplus x_3$ in the x_2 bit, and apply a Toffoli to calculate their AND. Then undoing the XORs, we have the x_2 value available to do the final XOR:

$$\tag{9.58}$$
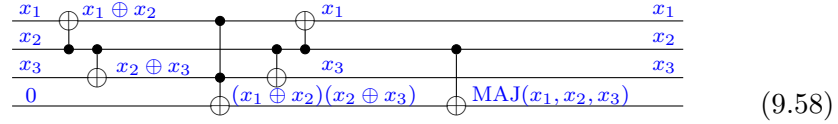
Well this circuit is certainly a lot smaller! It only requires one Toffoli gate and one additional bit. In fact, using this trick, the cost of implementing a classical function depends on its **multiplicative complexity**, the number of AND operations needed to write it down.

But what if we *really* don't want to use additional bits? What can we do in just the space of the inputs? As it turns out, quite a lot, as we will see in the next section.

9.3.2   From arbitrary reversible functions to Toffolis

Instead of being given a concrete specification of a classical function as a collection of simple operations performed in sequence, we can also consider it as just a truth table that tells us where every bit string is mapped to. In this section we will see how we can decompose such functions into *cycles* and how each cycle can be implemented using Toffoli gates.

So let us suppose we are given a reversible function $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$. As f is a bijection, we can see this as a *permutation* on the set of bitstrings $\{0, 1\}^n$. The set of n-bit reversible functions hence forms the **permutation group** on 2^n elements. Now, if you know a little bit of group theory, then you will know that a permutation group is generated by *cycles*, or more specifically, *2-cycles*.

Definition 9.3.4. Let G be a permutation group on some set S (like the set length-n bit strings). A k-**cycle** $\sigma \in G$ is a permutation where there are k distinct elements $x_1, \ldots, x_k \in S$ such that

- $\sigma(x_i) = x_{i+1}$ (and we set $x_{k+1} = x_1$),
- $\sigma(y) = y$ for any $y \in S$ not equal to one of the x_i,
- and there is no smaller k with the above two properties.

We say two cycles are *disjoint* when they don't have any elements they act non-trivially on in common (for σ and σ' that are not equal, this is equivalent to them commuting). We denote a k-cycle on $x_1, \ldots, x_k$ by $(x_1 \ x_2 \ \cdots \ x_k)$.

So a k-cycle is a permutation that *cycles* the value of x_1 to x_2 to x_3, and so on, to x_k and then back to x_1. It acts as the identity on all the other elements. A standard result from group theory is that any permutation can be written as a composition of disjoint cycles.

Lemma 9.3.5. Let G be a permutation group on some set S and let $\sigma \in G$ be any permutation. Then $\sigma = \sigma_1 \cdots \sigma_l$ for some disjoint cycles σ_j.

Proof Pick some element $x_1 \in S$ that is not a fixed point of σ (i.e. $\sigma(x_1) \neq x_1$). If σ only has fixed points, then it is the identity and we are done. Then set $x_2 = \sigma(x_1)$. Set $x_3 = \sigma(x_1)$. Note that we can't have $x_3 = x_2$, since then both x_1 and x_2 would map to x_2, which is not possible in a permutation. If $x_3 \neq x_1$, then we set $x_4 = \sigma(x_3)$. Again, x_4 cannot be equal to x_3 or x_2. We keep repeating this procedure as long as we get new elements out, until we get to some k such that $x_{k+1} = x_1$. Let $\sigma_1 = (x_1 \ \cdots \ x_k)$ and define $\sigma' = \sigma_1^{-1} \circ \sigma$. Note that all the x_i are fixed points of σ', as σ maps x_i to x_{i+1}, and then σ_1^{-1} maps x_{i+1} back to x_i. So σ' has more fixed points then σ. We can now repeat the procedure for σ'. It is guaranteed to terminate as the number of fixed points keeps increasing until we have reduced it to the identity. Then we have some sequence of cycles $\sigma_1, \ldots, \sigma_l$ such that $\sigma_l^{-1} \circ \cdots \circ \sigma_1^{-1} \circ \sigma = \mathrm{id}$. Hence $\sigma = \sigma_1 \cdots \sigma_l$. It is easy to see that all the cycles are disjoint. $\square$

Lemma 9.3.6. Let G be a permutation group on some set S. Then G is generated by 2-cycles.

Proof Lemma 9.3.5 shows that G is generated by cycles, so it suffices to show that each cycle can be built out of 2-cycles. This is easily done: $(x_1 \ \ldots \ x_k) = (x_1 \ x_2) \circ (x_2 \ x_3) \circ \ldots \circ (x_{k-1} \ x_k)$. For each x_i with $i < k$ we can easily check that only exactly one of these 2-cycles does something non-trivial to it, and maps it to x_{i+1}. For x_k, instead all the 2-cycles apply, mapping it first to x_{k-1}, then to x_{k-2}, and so on, until it is finally mapped to x_1. $\square$

Okay, so any permutation can be built out of 2-cycles. So if we want to know how we can construct an arbitrary reversible function, it suffices to

show how we can construct an arbitrary 2-cycle on bit strings. That is, we need to construct for any choice of bit strings $\vec{x}$ and $\vec{y}$, the function $P_{\vec{x},\vec{y}}$ that maps $\vec{x}$ to $\vec{y}$ and vice versa, and acts as the identity on all other bit strings.

So let's suppose $\vec{x}$ and $\vec{y}$ given. We will first simplify our lives somewhat by taking $\vec{x}$ to be equal to the all 1 bit string. We can do this by using the following identity for permutation groups (convince yourself that this works):

$$\sigma \circ (x\ y) \circ \sigma^{-1} \;=\; (\sigma(x)\ \sigma(y)) \tag{9.59}$$

In this case we will take σ to be the bit string function

$$\vec{z} \mapsto (z_1 \oplus x_1 \oplus 1, \ldots, z_n \oplus x_n \oplus 1),$$

which indeed maps $\vec{x}$ to $\vec{1}$. Note that this function is implemented by applying a NOT gate on the indices i where $x_i = 0$. Hence, $P_{\vec{x},\vec{y}}$ is equivalent to $P_{\vec{1},\vec{y}'}$ up to some NOT gates. Note that because we started with $\vec{x} \neq \vec{y}$, that we now have $\vec{y}' \neq \vec{1}$. Furthermore, by rearranging bits (for instance by applying some swap gates before and after the desired operations), we may assume that $\vec{y}' = 1\cdots10\cdots0$. That is, $\vec{y}'$ is a series of 1's followed by a series of 0's So let's suppose that $\vec{y}'$ consists of k 0's and $n-k$ 1's. Note that $k > 0$ as otherwise $\vec{y}' = \vec{1}$. We can now easily write down the required circuit:

$$P_{\vec{1},\vec{y}'} \;=\; \left. \begin{array}{c} n-k \\ k-1 \end{array} \right\{ \quad \text{(9.60)}$$

We can check that is correct by verifying the three possibilities for the input: $\vec{1}$, $\vec{y}'$, or some other bit string. First, the case where all the inputs are $\vec{1}$:
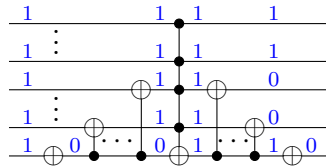
This output indeed matches $\vec{y}' = 1\cdots10\cdots0$. Let's check what happens when we input $\vec{y}'$ instead:
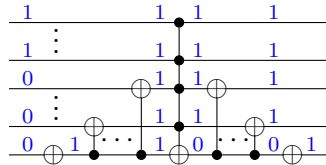
We also get the correct output of $\vec{1}$! We just need to check that this circuit does not do anything when we input any other state.

Exercise 9.10. Prove, by doing smart case distinctions, that the circuit of Eq. (9.60) acts as the identity when the input bit string is not $\vec{1}$ or $\vec{y'} = 1 \cdots 1 0 \cdots 0$.

Hint: if the input is such that the Toffoli does not fire, then the CNOT and NOT gates cancel each other out. In which situations can the Toffoli gate fire?

We hence have the following.

Proposition 9.3.7. We can implement any 2-cycle on n bits using a single $(n-1)$-controlled Toffoli and $O(n)$ CNOT and NOT gates.

Theorem 9.3.8. Any n-bit reversible function can be implemented using $O(n2^n)$ $(n-1)$-controlled Toffoli, CNOT, and NOT gates.

Proof A reversible function can be decomposed into 2^n disjoint 2-cycles, and each 2-cycle can be implemented using $O(n)$ gates, requiring a total of $O(n2^n)$ gates. $\qquad\square$

Now, it might seem like this procedure is very inefficient. After all, we first decompose the permutation down into 2-cycles, and then painstakingly construct each of these 2-cycles. Surely there must be a more efficient way, using k-cycles or some other trick, in order to use less than $O(n2^n)$ gates. While yes, there are clever ways to reduce the constants and to do better, on an asymptotic level, this construction is already close to optimal: we could only improve it up to a logarithmic factor.

Proposition 9.3.9. There exist reversible functions on $n \geq 2$ bits that require at least $cn2^n/\log n$ Toffoli, CNOT and NOT gates for $c = \frac{1}{6}$ (assuming that $\log = \log_2$).

Proof Counting the placement of gates on different bits as distinct, there are n different Toffoli gates with $n-1$ controls, n different NOT gates, and $n(n-1)$ different CNOTs. Hence, $n(n-1)+n+n = n^2+n$ different 1 gate circuits. Using N gates we can hence construct at most $(n^2 + n)^N$ different maps. There are $(2^n)!$ different reversible functions on n bits (where $k!$ represents the factorial function $k! = k(k-1)\cdots 2 \cdot 1$). In order to write down all reversible functions we hence need a number of gates N such that $(n^2 + n)^N \geq (2^n)!$. Stirling's formula for the factorial gives us $\log(k!) \geq \frac{1}{2}k \log k$. So by taking logarithms on both sides we get the inequality $N \log(n^2 + n) \geq \frac{1}{2}2^n \log 2^n =$

$\frac{1}{2}n2^n \log 2$. Assuming $n \geq 2$ and using $n^3 \geq n^2 + n$ we get $3N \log n \geq \frac{\log 2}{2}n2^n$ and hence $N \geq cn2^n/\log n$ for $c = \frac{\log 2}{6} = \frac{1}{6}$ (using $\log 2 = \log_2 2 = 1$). $\qquad \square$

At this point you might wonder if we really need a Toffoli gate with this many controls. Couldn't we make do with just a regular Toffoli with two control wires? The answer is *no*. We cannot decompose such a Toffoli into gates acting on fewer bits, at least in the current setting.

To understand this limitation, we need to know the concept of the **parity** of a permutation. We write the parity of a permutation σ as $\mathrm{sgn}(\sigma)$ and we define this inductively by setting the parity of each 2-cycle to be -1: $\mathrm{sgn}((x\ y)) = -1$, and making it respect composition: $\mathrm{sgn}(\sigma_1\sigma_2) = \mathrm{sgn}(\sigma_1)\mathrm{sgn}(\sigma_2)$. Hence the parity of a permutation captures whether we need an even or odd number of 2-cycles to write it down (it is a bit non-trivial to see that this is actually well-defined). We call a permutation σ *even* when $\mathrm{sgn}(\sigma) = 1$ and *odd* when $\mathrm{sgn}(\sigma) = -1$.

Now comes the catch: from the definition of parity we immediately see that when we compose even permutations, we get another even permutation. It just so happens to be that an $(n-1)$-controlled Toffoli acting on n bits is an odd permutation, while any gate acting on fewer bits is even. This means there is no way we can combine these gates to construct the $(n-1)$-controlled Toffoli. To see these gates indeed have these parities, first note that a $(n-1)$-controlled Toffoli acting on n bits is a 2-cycle that maps the bit string $1\cdots11$ to $1\cdots10$ and vice versa, so that it is indeed an odd permutation. Suppose instead we have a gate that does not act on at least one bit. Denote its corresponding permutation by σ, and let σ' denote the permutation where we have chopped of the last bit (the one it doesn't act on). Let $\sigma' = (\vec{x}^1\ \vec{y}^1)\cdots(\vec{x}^k\ \vec{y}^k)$ be a decomposition of σ' into 2-cycles. Then in the decomposition of σ, each of these 2-cycles must occur twice, one for each possible value of the last bit. That is, a decomposition of σ is given by $\sigma = (\vec{x}^1 0\ \vec{y}^1 0)(\vec{x}^1 1\ \vec{y}^1 1)\cdots(\vec{x}^k 0\ \vec{y}^k 0)(\vec{x}^k 1\ \vec{y}^k 1)$. Since the permutation then consists of an even number of 2-cycles, it is an even permutation.

There are two ways around this issue: use non-classical, i.e. quantum, gates, or use additional bits as 'scratch space'. This first solution won't be very useful to us, so we'll stick with the second one. We already saw in the previous section that we can use scratch space to do arbitrary computations. But it turns out that in the case of Toffoli gates, we can make do with just a tiny bit of additional available bits, as we will see in the next section.

9.4 Constructing Toffoli gates with many controls

Toffoli gates with many controls form a core part of many algorithms, and as we saw in the previous section, they are also essential in constructing arbitrary classical reversible circuits. In this section we will study several ways in which we can decompose Toffoli gates with many controls into Toffoli's with fewer controls. We saw above that we can't do this when we have an n-bit Toffoli acting on an n bit circuit. However, we *can* do it on an $n + 1$ bit circuit.

So let's assume we have an additional bit available. For concreteness, suppose we wish to construct the Toffoli gate with 3 controls:
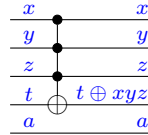
This actually calculates a function very similar to Eq. (9.54), but let's think through it again how to construct this.

First, suppose for simplicity that we knew the extra bit was supplied to us in the 0 state. Then we can use it to store an intermediate result, which we can then use in a later operation:

$$(9.61)$$

We indeed get the correct result on the target bit! Unfortunately, we have now polluted the state of our extra bit, so we wouldn't be able to apply this trick again. We have 'burned' this resource. We can fix this issue by cleaning up after ourselves. Luckily, a Toffoli is self-inverse and we haven't changed the state of the x and y bits, so this clean-up is easy:

$$(9.62)$$
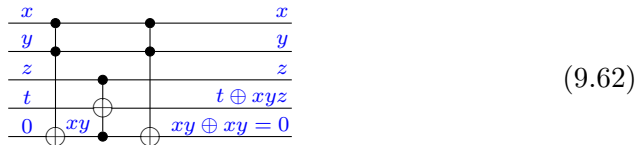
But suppose we didn't know that the bit was supplied to us in the 0 state (maybe because we aren't sure the previous person cleaned up after themselves...), how do we implement the gate we want? In this case, a picture is

worth more than a thousand words:

$$
\begin{array}{c}
\text{(circuit diagram)}
\end{array}
\tag{9.63}
$$

Here $t' = t \oplus zb \oplus z(b \oplus xy) = t \oplus zb \oplus zb \oplus zxy = t \oplus xyz$ is exactly what we want. The reason this works is because we apply the operation to the target t twice, so that the dependency on b disappears: the first Toffoli puts the information about b into t, the second Toffoli changes the information in b, and then the final Toffoli cancels the value of b in t, leaving only the information we wanted to put into it.

This construction didn't clean up after itself though, as it left the value in b changed, so lets add an additional Toffoli to get the construction we want:

$$
\begin{array}{c}
\text{(circuit diagram)}
\end{array}
\tag{9.64}
$$

So we have constructed a 3-controlled Toffoli using 4 regular Toffoli gates and one additional bit. This additional bit was provided to us in an unknown state, and was left at the end in that same unknown state, so on the right-hand side of Eq. (9.64) it looks like we haven't even touched this bit. Its presence was however crucial to the success of this procedure as the argument of the previous section on the impossibility of realising many-controlled Toffolis from regular ones showed. We will call such a bit a **borrowed bit**. While the state of borrowed bits is not changed, their presence can serve as an important catalyst for certain constructions (as Eq. (9.64) shows).

The constructions above are not reserved to just regular 2-controlled Toffoli gates. They in fact work for Toffoli gates with an arbitrary number of controls:

$$
\begin{array}{c}
\text{(circuit diagram)}
\end{array}
\tag{9.65}
$$

We can iterate this procedure. For instance, starting with a Toffoli with 5 controls, we use Eq. (9.65) to decompose it into four Toffoli gates with 3 controls each (pick $n = 3$ and $m = 2$). Then each of those we decompose

into four standard Toffoli's each:

$$(9.66)$$
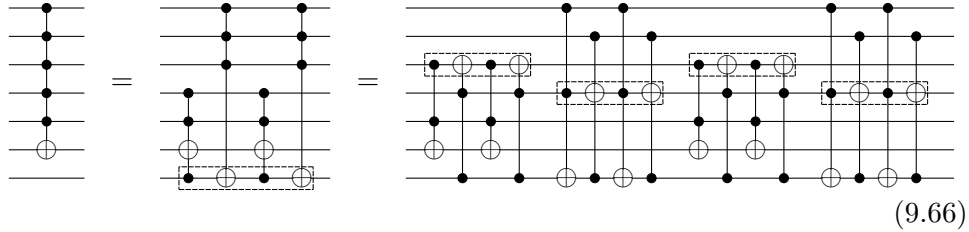
We have marked the places were a bit has been borrowed with a dashed box on the right-hand side. Note that we constantly switch which bit is borrowed. That's the beauty of this system: since the borrowed bit can be in any state and is returned to the same state, we can pick any bit to be the borrowed one.

You might have noticed that the number of Toffoli's required blows up quite a lot as the number of controls increases. In fact, if we have k controls (assuming k is odd for simplicity) then this splits into four gates with $(k+1)/2$ controls. So if we take $k = 2^n + 1$, then after the first step we have Toffoli's with $2^{n-1} + 1$ controls. So doing this n times we are left with 4^n regular Toffoli's. As $4^n = (2^n)^2 \leq (2^n + 1)^2 = k^2$, we see that in general we require $O(k^2)$ Toffoli's.

It turns out we can do better than an $O(k^2)$ scaling in the number of controls. To do this we need the observation that after the first split in Eq. (9.65) we have many more borrowed bits available. So let's try to use them!

To see how this works we will again first look at a construction where the additional bits are supplied in the 0 state and we don't care in which state we leave them. It turns out that to apply this trick to decompose a Toffoli with k controls, we need $k - 2$ additional bits. Let's look at the simplest example: $k = 4$.

$$(9.67)$$
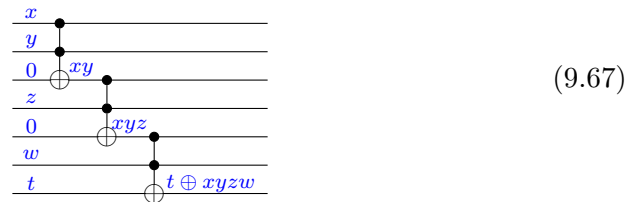
We have interspersed the 0 bits throughout the circuit, to make it look a bit nicer. We see that we can simply build a larger and larger product of bits by storing the intermediate results in these additional bits we have lying around.

If we wanted to return the bits to their zero state, we can just add another

staircase of Toffoli gates to undo the action done to them:

$$
\text{(9.68)}
$$
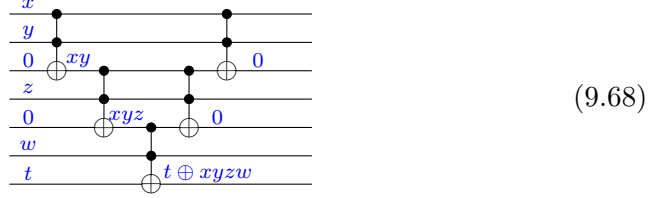
But what if we didn't know that the bits were 0. Then we can apply a similar trick to what we did in Eq. (9.63): we reverse the direction of the staircase (compare Eq. (9.63) with Eq. (9.62) where the order of the Toffoli gates was also reversed):

$$
\text{(9.69)}
$$
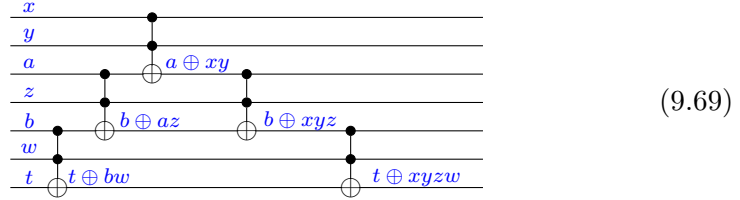
Note that the last two Toffoli gates cancel out respectively the az and bw term. Finally, if we want to make these bits borrowed, then we need to undo the Toffoli gates that affect them, which means we need to add another staircase:

$$
\text{(9.70)}
$$
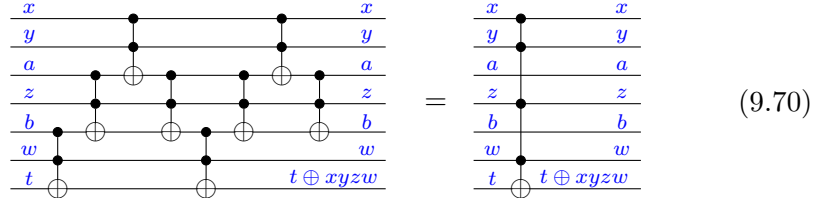
If instead of 4 controls and 2 borrowed bits we had $k \geq 4$ controls and $k - 2$ bits, then we could simply make the staircases longer. The first staircase 'going up' has $k - 1$ gates, then the one going down has $k - 2$ gates, the second going up also has $k - 2$ gates, and the final going down has $k - 3$ gates, for a total of $4k - 8$ Toffoli gates. If we had this many borrowed bits lying around we can hence decompose a k-controlled Toffoli in $O(k)$ regular Toffoli gates!

So let's get back to the case where we start out with a single borrowed bit. Then we can apply the trick of Eq. (9.65) once to decompose our k-controlled Toffoli into smaller Toffoli gates. If k is odd, we split it into four $\frac{k+1}{2}$-controlled Toffoli gates. We then have $\frac{k+1}{2}$ borrowed bits available for each of these Toffoli gates. If instead k is even, then we split into two Toffoli

gates with $\frac{k}{2}$ controls and two with $\frac{k}{2} + 1$ controls. In this case we will have at least $\frac{k}{2}$ borrowed bits available. In both cases this is enough space to apply the construction of Eq. (9.70). Each of those Toffoli gates can then be decomposed into $\approx 4 \cdot (k/2) = 2k$ gates. As we have four of them, the final cost is then $8k$ Toffoli gates.

Proposition 9.4.1. A single k-controlled Toffoli can be decomposed into a circuit of fewer than $8k$ regular Toffoli gates as long as we have a single borrowed bit available.

While it might be possible to improve the constants, this is asymptotically optimal, as we certainly need to at least touch every qubit involved with a Toffoli, and this requires $O(n)$ gates.

Exercise 9.11. In Eq. (9.68) each of the Toffoli gates used a qubit that the previous gate also used so that its circuit depth is also linear in the number of controls of the Toffoli we are constructing. But it is possible to do it more efficiently. Show that we can implement the k-controlled Toffoli in logarithmic depth using $O(k)$ regular Toffoli gates if we are supplied $k - 2$ bits in the 0 state, and make sure the ancilla bits are returned to the 0 state at the end. You may restrict to $k = 2^m$ for simplicity.

9.4.1 Quantum tricks for optimising Toffoli gates

So far we have only studied classical functions using classical means. But this is of course a book about quantum computing, so let's see what we can do once we're allowed to use quantum gates and techniques to construct these classical functions. In this section we will find better ways to decompose certain combinations of Toffoli gates, so that we can implement these more cheaply as quantum circuits.

First, recall that we could decompose the Toffoli gate into a combination of seven T gates and phase gadgets:

$$(9.71)$$
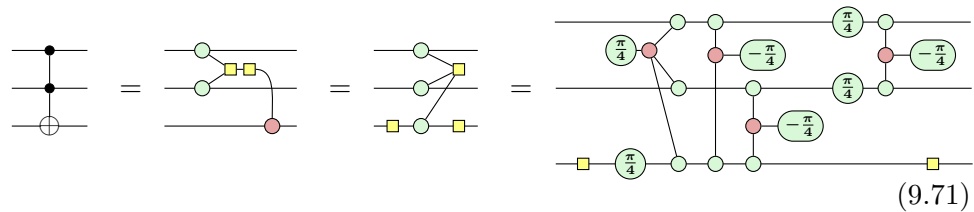
We have here grouped the gates in a suggestive way, with all the gates on the control qubits together. We can then recognise this as the shape of a $CZ(\frac{\pi}{2}) = CS$ gate; see Eq. (9.4). Hence, while a Toffoli gate requires 7 T

gates, if we can somehow combine this with a $CS^\dagger = CZ(-\frac{\pi}{4})$ gate, then 3 of those T gates cancel and we only require four of them:

$$ \tag{9.72} $$

Now, usually we don't have spare CS gates lying around to cancel T gates with, but we can introduce them in pairs at the cost of a Clifford: $I = CS^\dagger \circ CZ \circ CS^\dagger$. This means that whenever we have a pair of Toffoli gates that have the same two control wires and nothing acting in between them on those wires, that we can use this trick to reduce the T-count. In the previous section we saw many examples of such a pair of Toffoli gates. For instance, we can use it to reduce the cost of Eq. (9.62):

$$ \tag{9.73} $$

Now instead of the construction costing $3*7 = 21$ T gates, it costs $2*4+7 = 15$ T gates! Additionally, since we have to synthesise fewer phase gadgets, the construction will also require less two-qubit gates.

Note that this trick is not reserved to just the Toffoli gates that share two controls. Sharing a control and a target also works. For instance, starting with Eq. (9.63):

$$ \tag{9.74} $$

Now we have a pair of Toffoli's sharing two controls and we can apply the trick as before.

But what if we don't have a pair of Toffoli gates with matching controls or targets, what can we do then? Is there any way we can reduce the number of T gates we need? Well, there is some good news and some bad news.

The bad news is that all possible three-qubit Clifford+T circuits with up to six T gates have been enumerated by brute force methods, and none of those circuits were equal to a Toffoli. So there is no circuit with fewer than seven T gates that implements a Toffoli.

So what is the good news? Well, this enumeration only looked at *unitary* circuits. It doesn't say anything about circuits involving ancillae and measurements. It turns out that if we do allow non-unitary constructions that we can do better. To see how we can do this, let's take another look at Eq. (9.72), but now using H-boxes. To simplify the presentation a little, we will be working with a CCZ gate instead of a Toffoli, and a CS gate instead of a $CS^\dagger$ gate. So let's see how we could rewrite this construction:

$$\text{(9.75)}$$

Okay, this first step was obvious: there are spiders of the same colour connected to each other, so we should fuse them. But now it is a little less obvious. However, note that we now have two H-boxes that share two Z-spiders. This looks a lot like the right-hand side of Eq. (9.30). In fact, by doing some clever unfusion, we can actually apply this rule:

$$\text{(9.76)}$$

Here in the last step we used the the fact that an H-box with a single wire and a complex phase is just a spider (Eq. (9.21)): $\boxed{e^{i\alpha}}\!\!-\!\!- \ = \ \textcircled{\alpha}\!\!-\!\!-$ (and of course $i = e^{i\pi/2}$).

Okay, this looks promising! A CCZ followed by a CS, which is cheaper than just a CCZ, corresponds to a CCZ with this $X(\frac{\pi}{2})$ on one of its legs. So how do we transform this into something that looks more like a circuit? The answer is that we have to view the $X(\frac{\pi}{2})$-phase as happening on its own ancilla qubit. We do this by introducing some identity spiders and unfusing:

$$\text{(9.77)}$$

This is now a post-selected circuit (we'll get to how to deal with the 'wrong' measurement outcome later), where the only non-Clifford gate is the CCZ. So we have managed to get rid of the CS gate!

But of course we want to go the other way: instead of removing a CS gate, we want to introduce one. We can however do this procedure in the opposite direction quite easily. The crucial step happened in Eq. (9.76) where we transformed the CS gate into an X-phase on the other side of the H-box. It turns out that it is often useful to apply this rewrite rule in the opposite direction, so let's write it down explicitly:

$$
\begin{array}{c}
(sp)\\
(cc)
\end{array}
\quad\cdots\quad
\overset{(9.21)}{=}\quad
\overset{(9.30)}{\cdots}\quad
=\quad
\overset{(9.25)}{\cdots}\quad
=\quad\cdots\,e^{i\alpha}
\tag{9.78}
$$

So now, starting with a CCZ, let's introduce some X-phases, so that we can push one of them through the H-box to make a CS gate appear:

$$
\begin{array}{c}
(id)\\
(sp)
\end{array}
\;=\;
\overset{(9.78)}{\underset{(sp)}{=}}\;
\overset{(id)}{\underset{(sp)}{=}}\;
\tag{9.79}
$$

Now we could decompose the combined CCZ and CS gate using just 4 T gates. So by introducing an ancilla, we can make a post-selected circuit that implements a CCZ gate using fewer T gates than is possible with any unitary circuit. It turns out we can get rid of the post-selection as well. If we get the wrong measurement outcome, then we can push the resulting NOT gate back through the CCZ using Eq. (9.78):

$$
\begin{array}{c}
(sp)\\
(\pi)
\end{array}
\;=\;
\overset{(\pi)}{\underset{(9.78)}{=}}\;
\overset{(sp)}{=}\;
\tag{9.80}
$$

So the wrong measurement outcome leads to an additional CZ gate applied after the circuit. Since we know the measurement outcome, we can correct for this by applying the inverse of a CZ gate. This inverse is of course also a CZ gate. As this is Clifford, this does not increase the number of T gates we need. So we can indeed deterministically implement a CCZ gate using four T gates.

It turns out we can do something similar, when we have a 'compute-uncompute' pair of Toffoli gates. That is, a pair of Toffoli gates that undo each others action, such that the target is not changed in the mean time. Let's use Eq. (9.62) again as an example, but with the qubits rearranged to

make the presentation a bit nicer:

$$
\tag{9.81}
$$
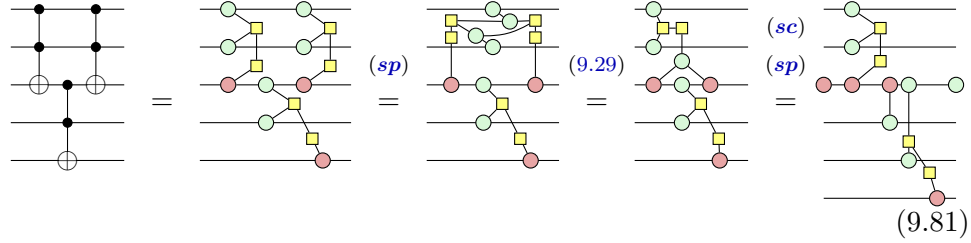
We see that whereas we had three Toffoli gates in the start, we ended up with just two of them in the end, as the pair that computed and uncomputed the AND of two bits was 'fused' together. This unitary is again post-selected, but on the wrong outcome we can push out the π phase outwards to become a CZ and Z correction.

Exercise 9.12. Prove that the correction operator of Eq. (9.81) for the $\langle -|$ measurement outcome is indeed $CZ \otimes Z$.

So while Eq. (9.73) allowed us to reduce the cost of a matching pair of Toffoli gates from 14 to 8, with Eq. (9.81) we can reduce it even further to just 4! But even if we don't care about decomposing into T gates, we see that this construction requires just a single Toffoli per compute-uncompute pair, as the uncomputation can instead be done by a measurement and a correction.

Note that this form of an optimised compute-uncompute pair of Toffolis is often used in the context where they are targetting a zeroed ancilla which is 'cleaned up' at the end. In that case we can simplify the expression a bit more, and for concreteness we will add in the $X(\frac{\pi}{2})$ phases to make it clear that this construction indeed only requires 4 T gates:

$$
\tag{9.82}
$$
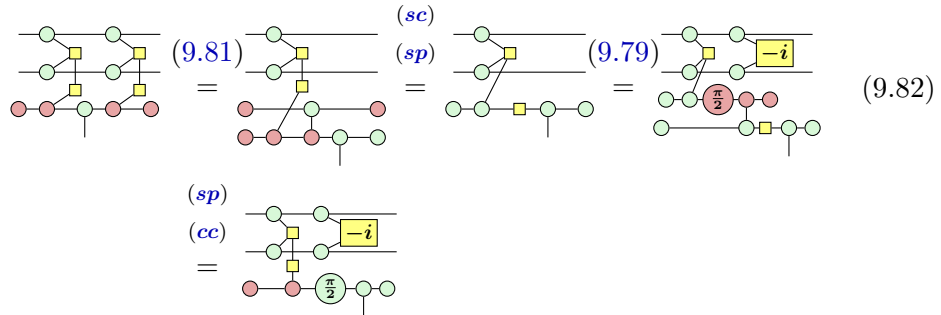
Exercise 9.13. Prove a version of Eq. (9.82), but where there are is an

additional pair of CNOTs involved in the computation-uncomputation:

$$(9.83)$$

What is the correction operation for the post-selected ancilla?

Exercise 9.14. Show that we can implement a CS gate using a single Toffoli and S gate, where we measure an ancilla and perform a CZ correction:
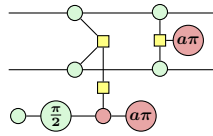
Does this construction also work for any other controlled-phase gates apart from CS?

Exercise 9.15. In this exercise we will show that we can construct the CCCZ gate (i.e. the 3-controlled Z gate) using 6 T gates.

a) Prove using the ZH-calculus that the following post-selected circuit implements a CCCZ gate:
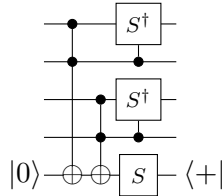
Hint: Use Eq. (9.78) to bring the CS gates into the H-box, and then combine the $\pi/2$ phases using an appropriate Euler decomposition of the Hadamard.

b) Find the correction operator for if the measurement got the wrong outcome instead.

c) Conclude that we can hence deterministically implement the CCCZ gate using 6 T gates. Hint: use Eq. (9.72).

Note: Another way to see that this construction works is because we can decompose the phase function of the CCCZ $(-1)^{xyzw}$ as $e^{i\pi(xy)(zw)} = e^{i\frac{\pi}{2}(xy\oplus zw - xy - zw)}$ and these three phase terms correspond to the two CS gates and the S gate on the ancilla (the two Toffoli gates precisely prepare the $xy \oplus zw$ state on the ancilla).

9.4.2 Adding controls to other quantum gates

Using what we've learned about Toffoli gates, we can also start to construct other unitaries with many controls.

The easiest construction for a many-controlled unitary U, which requires one clean ancilla, and the ability to construct a singly-controlled U, is the following:

$$\vdots \quad U \quad |0\rangle \qquad = \qquad \vdots \quad U \quad |0\rangle \tag{9.84}$$

We have drawn U here as a single-qubit unitary, but this of course works when U targets multiple qubits as well.

This is nice and all, but it still means we need to know how to construct U with a single control. This might be problematic if we want to restrict our gates to be from a particular gate set. For instance, it is not possible to construct a controlled-T gate without ancillae and using only Clifford+T gates. Luckily, in this case, we can adapt Eq. (9.84) for the special case of $U = Z(\alpha)$:

$$\vdots \quad Z(\alpha) \quad |0\rangle \qquad = \qquad \vdots \quad Z(\alpha) \quad |0\rangle \tag{9.85}$$

The reason this works is because $Z(\alpha)$ acts as the identity when the input is $|0\rangle$, so that it only fires when the first Toffoli puts the ancilla into the $|1\rangle$ state. Another way to look at it, is that a $Z(\alpha)$ phase gate is like a 'controlled global $e^{i\alpha}$ phase' gate, which applies a global phase of $e^{i\alpha}$ iff its control wire (the qubit it acts on) is in the $|1\rangle$ state. Hence, Eq. (9.85) is just a special case of Eq. (9.84) where the control wire *is* the target wire.

Exercise 9.16. Prove Eq. (9.85) using the ZH-calculus.

If we want to implement a many-controlled $X(\alpha)$ gate, we can realise that $X(\alpha) = HZ(\alpha)H$, and that these conjugations by a unitary (like H) commute with controls:

$$\vdots \quad X(\alpha) \quad |0\rangle \qquad = \qquad \vdots \quad H \quad Z(\alpha) \quad H \quad |0\rangle \quad \overset{(9.85)}{=} \quad \vdots \quad H \quad Z(\alpha) \quad H \quad |0\rangle \tag{9.86}$$

So now we know how to implement Z and X rotations with an arbitrary number of controls. By taking the Euler decomposition of a unitary, we can hence implement arbitrary many-controlled single-qubit unitaries.

Proposition 9.4.2. Let U be an n-qubit unitary implemented by a circuit of m CNOT, $Z(\alpha)$ and $X(\alpha)$ phase gates. Then we can construct a circuit for a U with k controls using $O(km)$ gates and one additional zeroed ancilla.

9.4.3   2-level operators

Let U be a single-qubit unitary. If we consider an n-qubit circuit containing just a $(n-1)$-controlled U gate, its matrix has a very particular shape:

$$
\begin{array}{c}\vdots\\U\end{array} \quad = \quad \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & & u_{11} & u_{12} \\ & & & u_{21} & u_{22} \end{pmatrix} \tag{9.87}
$$

Here all the empty spots in the matrix are zeroes.

This shape comes from the fact that this gate only does something non-trivial on the basis states $|1\cdots10\rangle$ and $|1\cdots11\rangle$, since the first $n-1$ wires have to be in the $|1\rangle$ state for it to fire. This unitary is a special case of a **2-level operator**, a unitary that acts non-trivially on just 2 basis states.

To make this more clear, instead of labelling all our basis states as bit strings, we will label them as numbers $1,2,3,\ldots,2^n$. Denoting $i = 2^n - 1$ and $j = 2^n$, we will call the above controlled gate $U_{[ij]}$, and it acts as follows:

$$
U_{[ij]} \begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{2^n} \end{pmatrix} = \begin{pmatrix} \psi'_0 \\ \vdots \\ \psi'_{2^n} \end{pmatrix} \text{ where } \begin{pmatrix} \psi'_i \\ \psi'_j \end{pmatrix} = U \begin{pmatrix} \psi_i \\ \psi_j \end{pmatrix} \text{ and for } k \neq i,j : \psi'_k = \psi_k.
$$
$$\tag{9.88}$$

A general 2-level operator is defined the same, but then i and j are allowed to be arbitrary values (as long as $i \neq j$ of course). We have in fact already seen other examples of 2-level operators: the 2-cycles that swap just two basis states and leave every other one invariant. A 2-cycle that swaps the basis states $|k\rangle$ and $|l\rangle$ is just the 2-level $X_{[kl]}$ gate. Because we know how to construct these 2-cycles and the $(n-1)$-controlled U, we have in fact all we need to construct arbitrary 2-level operators. This is because we have the relation $X_{[kj]}U_{[ij]}X_{[kj]} = U_{[ik]}$ as long as $k \neq i,j$: The first $X_{[kj]}$ moves $|k\rangle$

to $|j\rangle$, so that the $U_{[ij]}$ can apply to it, and then we move it back to the $|k\rangle$ spot by another application of $X_{[kj]}$. So with 2-cycle gates we can move the places where a 2-level operator acts non-trivially.

A 2-level operator acts non-trivially on two different basis states, but we also have **1-level operators** that act just on a single basis state. An example of this is the $(n-1)$-controlled $Z(\alpha)$ gate. The $Z(\alpha)$ gate only fires when all the controls are in the $|1\rangle$ state, but additionally the $Z(\alpha)$ only does something non-trivial when its target is also in the $|1\rangle$ state; the $|0\rangle$ state is left alone. We could hence call this gate $Z(\alpha)_{[j]}$ where $j = 2^n$. By conjugating by $X_{[kj]}$ we can change this gate to $Z(\alpha)_{[k]}$, which acts as $Z(\alpha)_{[k]}|j\rangle = e^{i\alpha\delta_{jk}}|j\rangle$, where δ_{jk} is the Kronecker delta.

There is a little bit of a subtlety around constructing these 1-level and 2-level operators: we saw in the previous section that in general we need to have a zeroed ancilla in order to construct unitaries with many controls. We hence can't easily make $(n-1)$-controlled single-qubit unitaries on a n-qubit circuit, since we need to have the additional space for an ancilla available to us. But then the gate is no longer controlled on all the values, and it stops being a 2-level operator (it will instead be a 4-level operator, since it acts non-trivially on its states regardless of the state of the ancilla). This will usually not be a problem however if we are assuming that the ancilla is zeroed, since it ends up back in the zeroed state. Since the $|1\rangle$ state of the ancilla doesn't come into play, the unitary will 'effectively' be a 2-level operator.

We will need these 2-level and 1-level operators when we talk about which unitaries we can exactly write down using Clifford+T gates in Chapter 10.

9.5 Adders

Let's put all we've learned to the test and build an efficient quantum circuit for a certain primitive that is an important component of many quantum algorithms: addition.

Recall that we can interpret an n-bit number $a \in \mathbb{N}$ as a n-qubit quantum state $|a\rangle = |a_{n-1}a_{n-2}\cdots a_0\rangle$ where the $|a_j\rangle$ are just computational basis states and $a = 2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + \cdots + 2^0 a_0$. Our goal then now is to build a $(2n+1)$-qubit quantum circuit we will call Add that acts as $\text{Add}|a, b\rangle = |a, a+b\rangle$. Note that here $a+b$ is addition of natural numbers, and not componentwise addition of bit strings! This circuit requires $2n+1$ qubits because the sum of two n-bit numbers requires $n+1$ bits to write down, and hence the register containing $a+b$ consists of $n+1$ qubits.

We will build the circuit for Add by mimicking how you would add together

numbers by hand: by starting at the least significant digit and proceeding upwards while keeping track of the carry. Let's call the outcome of the addition $s = a + b$, for sum. Calculating the least significant bit s_0 is very simple: it is just $a_0 \oplus b_0$. But now for the second bit s_1 we care about the carry value of the first bit. This carry bit is $c_0 := a_0 \cdot b_0$, since it is only non-zero if both a_0 and b_0 are 1. The value of s_1 is then the sum of a_1, b_1 and the carry c_0 modulo 2: $s_1 = a_1 \oplus b_1 \oplus c_0$. That still isn't too bad, but now we need to calculate the carry of this second bit s_1, and this is a bit more complicated, because now there are multiple ways in which the carry can be 1: we can either have both a_1 and b_1 be 1, or one of these values and the carry c_0, or all three of these values. We can however capture this in a nice symmetric function that we have already seen: the MAJ function that calculates whether the majority of values (in this case 2 out of 3) are 1. Recall from Eq. (9.57) that we have $\mathrm{MAJ}(x_1, x_2, x_3) = x_2 \oplus ((x_1 \oplus x_2) \wedge (x_2 \oplus x_3))$. Hence, setting $x_1 = a_1$, $x_2 = s_0$ and $x_3 = b_1$, we see that we can calculate the carry bit using the circuit:

$$
\begin{array}{c}
\text{(circuit)}
\end{array}
\tag{9.89}
$$
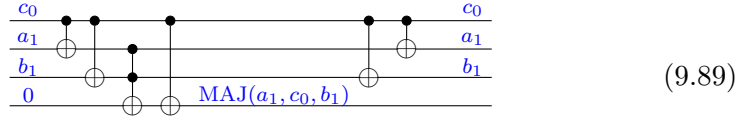
Calculating all the other bits now follows similarly: we set $s_2 = a_2 \oplus b_2 \oplus c_1$ and $c_2 = \mathrm{MAJ}(a_2, b_2, c_1)$. Or in general, for the kth bit we have $s_k = a_k \oplus b_k \oplus c_{k-1}$ where $c_k = \mathrm{MAJ}(a_k, b_k, c_{k-1})$.

Putting this all together, we can then write down a circuit for the adder, for instance for $n = 3$ we have:

$$\tag{9.90}$$
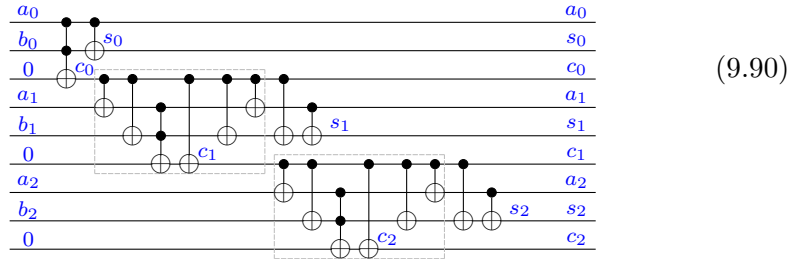
Here we have drawn a dashed box around the calculation of the MAJ function for the second and third carry bits. There are a couple of issues with this construction of the adder right now. First, it obviously contains some CNOT gates that can be cancelled against each other, and hence can be made a bit more efficient. But more importantly: we are not yet uncomputing the carry bits, which is important if we wish to use this Adder in superposition, so

let's do that:

$$(9.91)$$
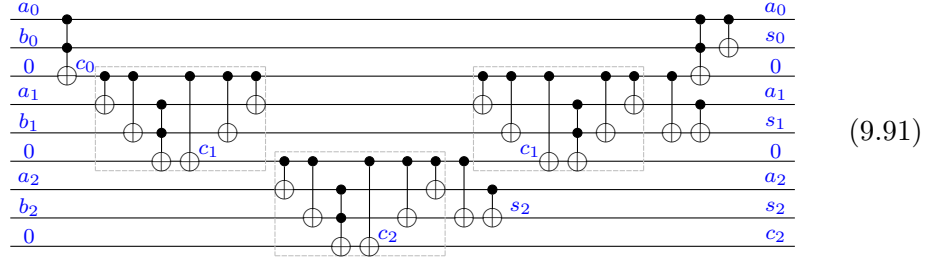
Let's cancel some matching CNOTs:

$$(9.92)$$
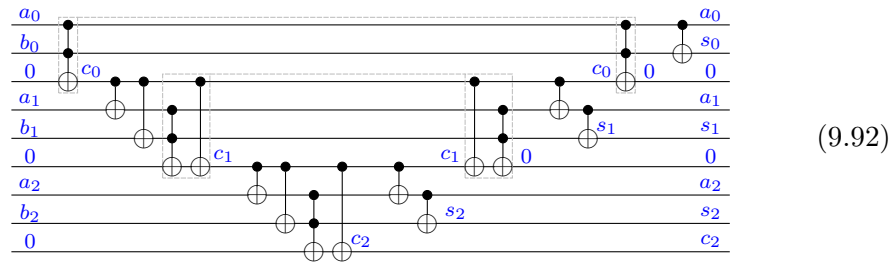
Here we have grouped together the different components that make up the computation: the calculation of first c_0, c_1, c_2 and s_2, followed by the uncomputation of c_1, the computation of s_1, the uncomputation of c_0 and finally the computation of s_0. This pattern extends to the addition of an arbitrary number of bits: first calculate all the carries, and then alternate calculating the sum of a bit and uncomputing the matching carry. We see then that the total cost consists of n zeroed ancillae and $2n - 1$ Toffoli gates: 1 each for the computation of a carry and 1 each for the uncomputation of all the carries except for the last one. Using the quantum tricks we have seen, we can however halve the cost of this addition circuit.

In the circuit (9.92) we have connected together the matching compute and uncompute pairs by dashed lines. These compute-uncompute pair Toffolis can be replaced by just a single Toffoli using the constructions of Eq. (9.82) (in the case of c_0 which is computing with just a Toffoli) and Exercise 9.13 (for c_1, as this involves both a CNOT and a Toffoli). The final carry does not have to be uncomputed and hence uses a single Toffoli regardless. The total number of Toffoli gates is then just n instead of $2n - 1$, and furthermore, each Toffoli can be implemented using just 4 T gates instead of 7.

Putting these optimisations together we see that we have reduced the original T count of $7 \cdot (2n - 1) = 14n - 7$ to $4n$.

Exercise 9.17. The construction of the adder described above is very efficient in the number of (Toffoli) gates, but it does require n ancillae, one

for each of the carry bits. We can also construct an adder without using any ancillae, but using more gates.

1. Argue that the following circuit implements a 'decrement by 1' operation:

$$
\begin{array}{c} \text{(circuit diagram)} \end{array}
= \begin{array}{c} \text{(circuit diagram)} \end{array}
\tag{9.93}
$$

That is, given a computational basis input $|\vec{b}\rangle$ encoding a number $b \in \mathbb{Z}_{2^n}$, it produces $b - 1$ modulo 2^n.

2. Argue that the following circuit of cascading 'controlled decrementers' implements the subtract operation $|a, b\rangle \mapsto |a, b - a\rangle$:

$$
\begin{array}{c} \text{(circuit diagram)} \end{array}
\tag{9.94}
$$

3. Argue that the adjoint of the above circuit hence implements an adder.
4. What is the cost in the number of standard 2-controlled Toffoli gates of this construction? You can use any decomposition of the many-controlled Toffoli gates that we have seen in this chapter (that fits in the available number of qubits).

9.6 Summary: what to remember

1. There is a Boolean Fourier transform from an XOR of bits, to an AND of bits. This allows us to relate phase gadgets (which are based on XOR), to diagonal controlled-unitaries (which are based on AND).
2. The H-box is a spider-like linear map that allows us to more compactly represent controlled unitaries. There are a variety of rewrite rules involving H-boxes that correspond to useful identities involving Toffoli gates and the Boolean AND. See Figure 9.1.
3. If we are given a Boolean formula, then we can implement it as a quantum circuit by storing intermediate computations on additional
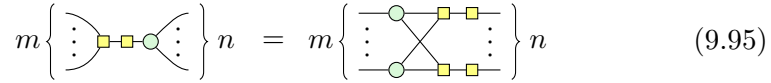
bits. By representing a classical reversible function directly as a truth table we can instead decompose it in terms of 2-cycles, which correspond to many-controlled Toffoli gates.

4. We can implement a k-controlled Toffoli gate with $O(k)$ regular Toffoli gates with the help of a single borrowed ancilla.

5. By introducing, measuring and correcting an ancilla, we can implement a Toffoli using 4 instead of 7 T gates. We can even implement a compute-uncompute pair of a Toffoli using a single ancilla and just 4 T gates.

6. Putting these tricks together allows us to create an n-qubit adder circuit that uses just n Toffoli gates, or $4n$ T gates.

9.7 Advanced Material*

9.7.1 W-spiders*

In this chapter we introduced the H-box to help us reason about Toffoli-like gates. The H-box is nice to work with, because it acts like a spider (Eq. (9.25)), and it interacts via a bialgebra rule with the Z-spider, as we saw in (9.29):

$$m\left\{ \begin{matrix} \vdots \end{matrix} \right. \!\!\!\!\!\!\!\! \left. \begin{matrix} \vdots \end{matrix} \right\} n \;=\; m\left\{ \begin{matrix} \vdots \end{matrix} \right. \!\!\!\!\!\!\!\! \left. \begin{matrix} \vdots \end{matrix} \right\} n \qquad (9.95)$$

This works because an H-box followed by a Hadamard is equal to the classical AND operation acting on computational basis states. In the same way, the X-spider has a bialgebra rule with the Z-spider, because the X-spider is equal to the classical XOR.

It turns out that up to some trivial modifications, there is exactly one other spider-like map that interacts with the Z-spider via a bialgebra rule, and that is the partial map Add:

$$\text{Add}|x, y\rangle \;=\; \begin{cases} |x + y\rangle & \text{if } x \cdot y \neq 1 \\ 0 & \text{otherwise} \end{cases}$$

That is: it adds together the value of the two basis states, *as long as* their sum is not greater than 1. If both x and y are 1, then their sum should be 2, but this doesn't 'fit' into a single qubit, and so it is sent to the scalar zero. It is hence quite similar to the XOR, except that the $|11\rangle$ input is sent to zero. As matrices:

$$\text{XOR} \;=\; \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \qquad \text{Add} \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

We saw in Section 9.2.1 that the reason we work with H-boxes instead of AND gates directly, is because H-boxes have **flexsymmetry**, meaning we can treat inputs and outputs on the same footing and bend wires as we wish. To make the AND flexsymmetric, we had to compose it with a Hadamard. In an analogous way, the Add map, and its n-qubit input generalisation, is not flexsymmetric, but we can make it flexsymmetric by composing it with a NOT. We can easily see this when we write Add in terms of kets and bras:

$$\text{Add} \;=\; |0\rangle\langle 00| + |1\rangle\langle 01| + |1\rangle\langle 10| \qquad \text{NOT}\circ\text{Add} \;=\; |1\rangle\langle 00| + |0\rangle\langle 01| + |0\rangle\langle 10|$$

This 'flexsymmetrised' Add consists of all ket-bra pairs that have exactly one $|1\rangle$, with the same role being played by inputs and outputs. Hence, when we bend the wires so that it is a state, we get the following:

$$\begin{array}{c}\boxed{\text{Add}}\;\tikz\draw(0,0)circle(2pt);\end{array} \;\;=\;\; |001\rangle + |010\rangle + |100\rangle \tag{9.96}$$

where $|\vec{x}|$ is the **Hamming weight** of the bitstring $\vec{x}$, i.e. the number of 1s that appear in $\vec{x}$. This state is known as the **W-state**. The W-state is important in entanglement theory as it is a nice representative of a certain type of genuine three-party entanglement, where one party can do a measurement, and the other two parties are still left with a maximally entangled state (compare this to the GHZ state $|000\rangle + |111\rangle$ where when someone does a measurement, the state completely disconnects). But for us that is all not important, except that it motivates the name for the **W-spider**:

$$\tikz\draw(0,0)circle(2pt); \;\;=\;\; \sum_{\substack{\vec{x},\vec{y} \\ |\vec{x}|+|\vec{y}|=1}} |\vec{y}\rangle\langle \vec{x}| \tag{9.97}$$

The 0-input 3-output W-spider is the W-state, while with a single input and output we get the NOT gate:

$$\tikz\draw(0,0)circle(2pt); \;\;=\;\; |1\rangle\langle 0| + |0\rangle\langle 1| \;\;=\;\; \tikz\draw(0,0)circle(2pt); \tag{9.98}$$

The W-spider (9.97) has all the same symmetries that Z- and X-spiders have: we can permute inputs and outputs freely, and we can interchange inputs with outputs using cups and caps.

However, just as with H-boxes, it has a modified spider-fusion rule, which requires a 2-ary spider to be in the middle (cf. (9.25)):

$$\tikz\draw(0,0)circle(2pt); \;=\; \tikz\draw(0,0)circle(2pt); \tag{9.99}$$

For the H-box we needed something in the middle, as the H-box followed

by a Hadamard was the AND map, which could fuse together due to its associativity. We have the same situation with the W-spider, but now using the associativity of Add:

$$\rightarrowtail\!\!\bullet\!\!\bullet\!\!- \;=\; \boxed{\text{Add}}\!\!- \tag{9.100}$$

As promised, this relation to Add means that the W-spider interacts with the Z-spider via a bialgebra rule:

$$\rightarrowtail\!\!\bullet\!\!\bullet\!\!\circ\!\!\prec \;=\; \text{(bialgebra diagram)} \tag{9.101}$$

There are a number of other rules governing the interaction between the Z- and W-spider:

$$\tag{9.102}$$

In fact, we could build a whole calculus to rival the ZX-calculus using just the Z- and W-spider, which is called the **ZW-calculus**. We didn't really give it a name before, but just thinking about Z-spiders and H-boxes, and viewing the X-spiders as derived from their interactions, we can call this the **ZH-calculus**. Note however that these calculi—ZX, ZW, ZH—can represent the same linear maps, and hence have the same expressive power. However, certain constructions will look more natural using one type of generator versus another. Using Z- and X-spiders we can easily reason about Cliffords and phase gates, with XOR-like phases captured with phase gadgets. Instead using Z-spider and H-boxes we can easily represent controlled gates and multiplicative phases. Using W-spiders we can easily represent additive structure, which is not very prominent in quantum circuits, but is useful when thinking about more general linear maps, and for proving completeness. For instance, we can use a W-spider to represent the 'addition' of two H-boxes:

$$\boxed{a}\!\!\boxed{b}\!\!\rightarrowtail\!\!\bullet\!\!\bullet\!\!- \;=\; \boxed{a+b}\!\!- \tag{9.103}$$

Note that a 3-ary W-spider can be decomposed into the ZH-calculus quite easily:

$$\rightarrowtail\!\!\bullet\!\!- \;\propto\; \text{(ZH diagram)} \tag{9.104}$$

This works because the W-spider is almost the Add, which is just the XOR

with the $|11\rangle$ output projected away. That projection is done by the H-box gadget in front of the X-spider:

$$
\begin{array}{ccc}
\text{[diagram]} & \propto \ \text{[diagram]} & = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}
\end{array}
\tag{9.105}
$$

We won't be using the W-spider in this book, but it would be remiss of us to not mention it in a book that is all about graphical reasoning. The W-spider and the ZW-calculus has played a crucial role in the history of the ZX-calculus and its completeness. See the References for more pointers on this.

9.8 References and further reading

Reversible circuit synthesis The relation between Boolean functions and DAGs was taken from Meuli et al. (2019a). They also introduce a 'pebbling' strategy that we alluded to, where values are uncomputed and recomputed in order to stay under a circuit budget of additional memory bits. They do this by encoding the synthesis as a SAT instance and then SAT solving it, increasing the number of allowed gates and memory bits until they find a satisfying solution.

The trick to reduce the number of bits needed by always computing and uncomputing XORs was taken from Meuli et al. (2019b). See also Meuli et al. (2020). Both these papers also use a pebbling strategy, but one that takes into account the fact that XORs are cheap.

H-boxes The H-box was introduced by Backens and Kissinger (2019), which proved completeness of the calculus in the universal fragment. This was followed up by Backens et al. (2023) which proved that the phase-free fragment could also be made complete. The Fourier decomposition of an H-box is from Kuijpers et al. (2019). The term 'flexsymmetry' was coined in Carette (2021).

Decomposing Toffoli gates Section 9.4 on how to decompose many-controlled Toffoli's into smaller Toffoli's was heavily inspired by a blogpost by Craig Gidney Gidney (2015), from which we have also taken the term 'borrowed bit'. That at least 7 T gates are necessary to implement a Toffoli or CCZ when restricting to unitary circuits was shown in Amy et al. (2013a); Di Matteo and Mosca (2016). That a Toffoli gate combined with a CS gate is cheaper

was first realised by Selinger Selinger (2013). Jones came up with the trick to use an ancilla to reduce the cost Jones (2013) to four T gates. The 'compute-uncompute' construction that reduces the cost of a pair of Toffoli gates to just four T gates was found by Gidney Gidney (2018). This paper also gives the description of the adder circuit we use in Section 9.5. The graphical approach to deriving these identities was developed in Kuijpers et al. (2019). The 6 T construction of the CCCZ in Exercise 9.15 is from Gidney and Jones (2021). A structured approach to finding these types of decompositions is given in Amy and Ross (2021).

W-spiders The W-spider was introduced in Coecke and Kissinger (2010b). This was extended to a complete ZW-calculus by Hadzihasanovic (2015b). It was this ZW-calculus that formed the basis for the first completeness results of a universal fragment of the ZX-calculus Hadzihasanovic et al. (2018); Jeandel et al. (2018). That ZX, ZW, and ZH are essentially the only three possible graphical calculi for qubits was shown in Carette and Jeandel (2020). Using W-spiders to represent arithmetic was first done in Coecke et al. (2010), and then developed much further in Wang et al. (2022), where they show how to represent the sum of two diagrams as a single diagram. This was done independently in Jeandel et al. (2024), although there the W-spiders are a bit more hidden by the use of the triangle generator. For more details on how these different generators of H-boxes, W-spiders and triangles relate to each other, we refer the reader to (van de Wetering, 2020, Section 9).

10

Clifford+T

In the previous chapters, we have often seen a dichotomy between two classes of quantum computations. On the one hand, we have Clifford computations, which have a great deal of structure we can exploit to efficiently solve problems like circuit synthesis, deciding equality of computations, and strong classical simulation. On the other hand, we have looked at universal quantum computation, typically arising from adding $Z[\alpha]$ gates for arbitrary angles $\alpha \in [0, 2\pi]$. While Clifford angles, i.e. integer multiples of $\frac{\pi}{2}$, satisfy many identities, we have so far treated non-Clifford angles as "black boxes", which don't seem to satisfy *any* extra rules, except for trivial ones coming from the spider fusion rule like

and variations thereof. It turns out that for generic angles, this is pretty much all we can do. In fact, as we'll explain in the References, there is a way to make this statement precise.

However, for **dyadic angles**, i.e. angles of the form $\frac{\pi}{2^k}$ for some integer k, there is a great deal more structure at play. As we'll see in this chapter, we can take advantage of this dyadic structure in a variety of ways.

First, we will see that it simplifies the problem of synthesising generic unitary maps using just Clifford gates and the first non-Clifford dyadic phase gate $T := Z[\frac{\pi}{4}]$. We can characterise the set of unitary matrices that we can synthesise exactly using Clifford+T gates as those whose entries are all within a certain subset, called $\mathbb{D}[\omega]$, of the complex numbers. Using some special properties of this set, and a little (light) ring theory, we can figure out precisely how many gates we need to synthesise such a matrix exactly and do this synthesis efficiently. Note, here "efficient" means efficient in the size of the matrix, not the number of qubits. For generic unitaries, this is the best we can hope for. We'll also see that for any tolerance $\epsilon > 0$, we can

approximate any unitary matrix within ϵ with a $\mathbb{D}[\omega]$-matrix, which we can in turn synthesise exactly using Clifford+T gates.

Second, we will see that for dyadic angles, new rules start to hold that wouldn't for generic angles. In particular, certain complex configurations of phase gadgets can all cancel each other out in ways that are not implied by the gadget-fusion law we met back in Chapter 7. These so-called **spider nest identities** come from a particular interaction between the parity (i.e. mod-2) structure of the phase gadget itself and the mod-2^k structure of its angle.

Similar to the representation from Chapter 7 of CNOT+phase circuits, we can represent phase gadgets as collections of binary vectors representing parities of qubits where phases get applied, and as before we can stick these vectors together into a matrix. We introduce a new **Scalable ZX notation** so that we can directly represent these matrices in an efficient way in our diagrams. Using these matrices representing collections of phase gadgets we can also recognise which configurations of $\frac{\pi}{4}$ phase gadgets will cancel out. Namely, it will be those whose "gadget matrix" satisfies a condition called **strongly 3-even**. In this chapter, we will work out some properties of strongly 3-even matrices and a closely related $\mathbb{F}_2$-linear structure called **Reed-Muller codes**. Finally, we will put the pieces together by classifying all the spider nest identities and explaining how they can be used for T count optimisation.

Clifford+T circuits are especially important for fault-tolerant quantum computation, so in this chapter we will also be (secretly) laying the ground-work for Chapter 11, where we explain techniques for implementing universal quantum computation within a quantum error correcting code. In particular, in Section 10.5 we will see how we can relate circuits over different types of gate sets together using a technique called **catalysis** that allows you to perform certain 'hard' operations in an easy way as long as you have a suitable catalyst state lying around. Together with the classification of spider nest identities this will prove very handy for when we want to implement non-Clifford gates in fault-tolerant architectures.

10.1 Universality of Clifford+T circuits

Back in Chapter 2, we noted that CNOT gates plus arbitrary single-qubit unitaries are universal for quantum computation, in the sense that they can be used to build arbitrary unitaries over n qubits. In this section, we will show that, in fact Clifford+T circuits are approximately universal, in the sense that they can get arbitrarily close to any n-qubit unitary.

One way to show this, is to show how for any single-qubit unitary U there exists some U' arbitrary close to U expressed totally in terms of H and T gates. There are several ways to prove this. The "classic" way is to show that $V := HT$ corresponds to a rotation about some axis of the Bloch sphere by an irrational multiple of π. Then, by raising V to larger and larger powers, we will eventually land close to *any* possible rotation around that axis. We can do the same around some other axis, e.g. with $V' = TH$, to obtain a pair of rotation gates that suffice to build any single-qubit unitary.

It takes quite some work to spell out the details of this argument, and this has been done in several standard textbooks on quantum computing. There are also many variations one can use to obtain more or less efficient decompositions of single-qubit unitaries. We'll give some pointers to where you can find all the gory details of this approach and variations at end of this chapter.

However, in this section, we will start with a totally different approach to synthesis of unitaries in Clifford+T, which is based on number theory. This approach starts from the realisation that the numbers appearing in a unitary matrix built from CNOT, H, and T are not just any old complex numbers, but are actually quite special. First, lets have a look at the matrices again:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

Clearly any U that we can construct from these gates will have as its entries sums and products of integers, $\frac{1}{\sqrt{2}}$, and the complex phase $\omega := e^{i\pi/4}$. If we think about where ω lies on the complex plane:

$$(10.1)$$
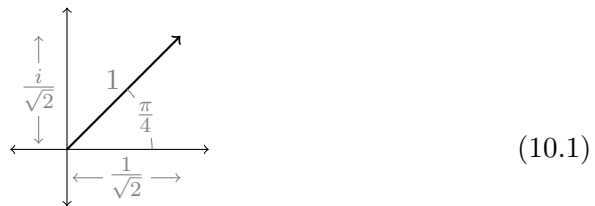
we see that $\omega = \frac{1+i}{\sqrt{2}}$ and $\overline{\omega} = -\omega^3 = \frac{1-i}{\sqrt{2}}$. Using these facts, it's not hard to see that we can already build $\frac{1}{\sqrt{2}}$ using just integers, $\frac{1}{2}$, and ω:

$$\tfrac{1}{2}(\omega - \omega^3) = \tfrac{1}{\sqrt{2}}$$

If we look at just the numbers we can build with integers and $\frac{1}{2}$, this

consists of precisely the rational numbers whose denominator is a power of two. We give this set of numbers a special name.

Definition 10.1.1. The **dyadic rational numbers** $\mathbb{D}$ consist of all rational numbers of the form $\frac{z}{2^k}$ for $z, k \in \mathbb{Z}$.

Clearly $0, 1 \in \mathbb{D}$, and for $q, r \in \mathbb{D}$, $-q, q + r$ and qr are also in $\mathbb{D}$. Hence, $\mathbb{D}$ forms a ring. However, unlike the full set of rational numbers, not every $q \in \mathbb{D}$ has a multiplicative inverse in $\mathbb{D}$, so it is not a field. It turns out that, for the purposes of circuit synthesis, this is not a bug, but a feature, since rings can have very interesting properties owing to the fact that we cannot arbitrarily divide numbers by each other.

To discuss the single-qubit synthesis algorithm, there are two relevant rings based on $\mathbb{D}$ we need to study. The first is $\mathbb{D}[\omega]$, the ring obtained by allowing arbitrary sums and products of dyadic rationals with the complex number ω. The second is $\mathbb{Z}[\omega] \subsetneq \mathbb{D}[\omega]$, which is restricted just to integer multiples of powers of ω.

Such rings are called **ring extensions**, i.e. rings obtained by adding one or more elements outside of the original ring and closing under sums and products. Since $\omega^4 = e^{i\pi} = -1$, we can concretely represent elements of $\mathbb{Z}[\omega]$ and elements of $\mathbb{D}[\omega]$ using quadruples of numbers taken respectively from $\mathbb{Z}$ or $\mathbb{D}$:

$$\mathbb{Z}[\omega] := \{\, a + b\omega + c\omega^2 + d\omega^3 \,|\, a, b, c, d \in \mathbb{Z} \,\}$$
$$\mathbb{D}[\omega] := \{\, a + b\omega + c\omega^2 + d\omega^3 \,|\, a, b, c, d \in \mathbb{D} \,\}$$

Exercise 10.1. Define ring addition and multiplication for $\mathbb{Z}[\omega]$ and $\mathbb{D}[\omega]$ as operations acting on quadruples (a, b, c, d) of numbers respectively taken from $\mathbb{Z}$ and $\mathbb{D}$.

Clearly any unitary matrix built out of CNOT, H, and T will consist of elements from $\mathbb{D}[\omega]$. Perhaps surprisingly, the converse is also true: any unitary matrix whose elements are in $\mathbb{D}[\omega]$ can be constructed *exactly* as a composition of CNOT, H, and T gates. We will show this by giving a concrete algorithm for synthesising unitaries over $\mathbb{D}[\omega]$ in the following sections, then conclude with some remarks on how this lifts to an approximate synthesis algorithm for unitaries over all complex numbers.

10.1.1 Exact synthesis of one-qubit gates

We'll begin by considering the simplest non-trivial synthesis problem we can have: namely preparation of an arbitrary single-qubit state $|\psi\rangle$. That is, we

want to find a sequence of H and T to transform $|0\rangle$ into $|\psi\rangle$. Equivalently, we can find a sequence of H and T transforming $|\psi\rangle$ into $|0\rangle$, then take the adjoint.

One way to do this is starting with a state whose entries are in $\mathbb{D}[\omega]$, and then applying gates to it until all of the entries are in $\mathbb{Z}[\omega]$. The following lemma should make clear why this helps us.

Lemma 10.1.2. If $|\psi\rangle$ is normalised and all of its entries are in $\mathbb{Z}[\omega]$, then it must be a computational basis vector, up to a global phase.

Proof   For any element $z = a + b\omega + c\omega^2 + d\omega^3$ in $\mathbb{D}[\omega]$, we can get an explicit form for $|z|^2 = \overline{z}z$ in terms of $\sqrt{2} = \omega - \omega^3$:

$$|z|^2 = \overline{z}z = (a^2 + b^2 + c^2 + d^2) + (ab + bc + cd - da)\sqrt{2}. \qquad (10.2)$$

If we suppose that all the a, b, c, d are integers, then the only way to have $0 \le |z|^2 \le 1$ is when the $\sqrt{2}$ part here is negative.

Let $|\psi\rangle = \sum_i \psi_i |i\rangle$ be a normalised state with entries in $\mathbb{Z}[\omega]$. Writing $|\psi_i|^2 = n_i + m_i\sqrt{2}$ where $n_i, m_i \in \mathbb{Z}$ and $n_i \ge 0$ and $m_i \le 0$, we have $1 = \langle\psi|\psi\rangle = \sum_i |\psi_i|^2 = \sum_i (n_i + m_i\sqrt{2})$. Since we know that the total sum is 1, the $\sqrt{2}$ components should cancel: $\sum_i m_i = 0$. But all the m_i are negative, so the only way this can happen is if all $m_i = 0$. Similarly, since all the n_i are positive integers and they sum to 1, they must all be zero except for one n_j which is equal to 1. But then exactly one of a_j, b_j, c_j and d_j is ± 1, and the rest are zero. Hence $|\psi\rangle = \omega^k|j\rangle$ for some i, k. $\square$

If we get to $\omega^j|0\rangle$, mission accomplished, up to a global phase. If we get to $\omega^j|1\rangle$, we simply apply an X gate (i.e. HT^4H) and again we are done. So, the name of the game is turning the coefficients of $|\psi\rangle$ into elements of $\mathbb{Z}[\omega]$. One potential strategy is to factor $|\psi\rangle$ as:

$$|\psi\rangle = \frac{1}{2^k}\left(x|0\rangle + y|1\rangle\right) \qquad \text{for } x, y \in \mathbb{Z}[\omega]$$

then apply gates to $|\psi\rangle$ to try and make the coefficients x and y into even numbers. Then, we can factor out a 2 from x and y and the leading scalar becomes $\frac{1}{2^{k-1}}$ and we've made some progress toward getting a state whose coefficients are in $\mathbb{Z}[\omega]$. This does work, but it is a bit tricky to find the gates we need to apply to $|\psi\rangle$ to "make progress".

We can make life a bit easier if we express $|\psi\rangle$ differently, as:

$$|\psi\rangle = \frac{1}{\delta^k}\left(x|0\rangle + y|1\rangle\right) \qquad \text{for } x, y \in \mathbb{Z}[\omega] \qquad (10.3)$$

where $\delta := 1 + \omega$ is a "special" number that has some nice number-theoretic

properties that will help with our synthesis algorithm (for more details on the number theory side of things, check out the advanced section 10.7.1).

First, we should be able to check that $|\psi\rangle$ is indeed a vector over $\mathbb{D}[\omega]$. For this, we should check that $\frac{1}{\delta} \in \mathbb{D}[\omega]$, which is not immediate because not every element in $\mathbb{D}[\omega]$ has a multiplicative inverse. The elements of a ring that do have multiplicative inverses are called **units**. We can see that δ is a unit in $\mathbb{D}[\omega]$ by letting $\delta^{-1} := \frac{1}{2}(1 - \omega + \omega^2 - \omega^3)$ and calculating $\delta\delta^{-1} = 1$.

Conversely, we would like to know that any $|\psi\rangle$ with coefficients in $\mathbb{D}[\omega]$ can be written in the form of (10.3). This is true if any $q \in \mathbb{D}[\omega]$ can be written as $\frac{x}{\delta^k}$ for some $x \in \mathbb{Z}[\omega]$ and a high enough power k of δ. Put another way, we need to prove the following property of δ.

Exercise 10.2. Show that, for any $q \in \mathbb{D}[\omega]$, there exists k such that $\delta^k q \in \mathbb{Z}[\omega]$. *Hint: Using the fact that $\sqrt{2} = \omega - \omega^3$, first show that $\delta^2 = \lambda\sqrt{2}$, where $\lambda := 1 + \omega + \omega^2$ is in $\mathbb{Z}[\omega]$.*

The smallest k needed to express $|\psi\rangle$ in the form of equation (10.3) is called the **least denominator exponent** (lde). If the lde is 0, then $|\psi\rangle$ is already has coefficients in $\mathbb{Z}[\omega]$, so by Lemma 10.1.2, it must be a basis state, up to a phase. It is easy to see that k is minimal precisely when x and y are not both divisible by δ, i.e. when there exists no pair $a, b \in \mathbb{Z}[\omega]$ such that $a\delta = x$ and $b\delta = y$.

We can easily check divisibility by δ using δ^{-1}. Since $\delta^{-1} \notin \mathbb{Z}[\omega]$, then it is not necessarily the case that $z\delta^{-1} \in \mathbb{Z}[\omega]$ for some $x \in \mathbb{Z}[\omega]$. In fact, it will be in $\mathbb{Z}[\omega]$ precisely when δ divides x.

The final piece of the puzzle is the following lemma, which lets us decrease the lde by applying H and T gates.

Lemma 10.1.3. Let $|\psi\rangle = \frac{1}{\delta^k}(x|0\rangle + y|1\rangle)$ be a state where $x, y \in \mathbb{D}[\omega]$ are non-zero. Then there exists some $l \in \{0, \ldots, 7\}$ such that $HT^l|\psi\rangle = \frac{1}{\delta^k}(x'|0\rangle + y'|1\rangle)$ for x', y' divisible by δ.

Since x', y' become divisible by δ, we can factor a δ out and reduce the least denominator exponent by 1. If we repeat this process over and over, eventually the lde will be 0, so by Lemma 10.1.2 it will be a basis vector.

The reason why Lemma 10.1.3 works has to do with the special behaviour of δ within the ring $\mathbb{Z}[\omega]$. Namely, if we start computing numbers modulo δ (or powers of δ) in $\mathbb{Z}[\omega]$, we will see that x and y can always be broken down into a particular form that tells us how many T gates to apply to make progress. To understand this, we will need to define the notion of a **residue class**, which lets us formalise what it means to work "modulo" some element

of an arbitrary ring. We will do this and provide a proof for Lemma 10.1.3 in Section 10.7.1.

However, if we believe the lemma, we now have a synthesis algorithm for qubit states. In fact, this also already gives a synthesis algorithm for single-qubit unitaries. This is because the columns of a unitary matrix must be orthogonal, so if we send the first column to a basis vector, the second column will also get sent to a basis vector, up to a phase. So after transforming the basis vector to the correct positions and phases, we will have synthesised the entire single-qubit unitary. We summarise this procedure in Algorithm 3.

Algorithm 3: Exact synthesis for qubit Clifford+T gates

Input: A unitary U with elements in $\mathbb{D}[\omega]$
Output: A list of H and T gates implementing U

1. Let $|\psi\rangle$ and $|\phi\rangle$ be the columns of U. Since U is unitary, $|\psi\rangle$ and $|\phi\rangle$ must be orthogonal.
2. Express $|\psi\rangle$ as:

$$|\psi\rangle = \frac{1}{\delta^k}\left(x|0\rangle + y|1\rangle\right) \qquad \text{for } x, y \in \mathbb{Z}[\omega]$$

   where k is the least denominator exponent.
3. Try to apply HT^l for all $l \in \{0, \ldots, 7\}$ to $x|0\rangle + y|1\rangle$ to obtain $x'|0\rangle + y'|1\rangle$ where δ divides x' and y'.
4. Factor out a δ to obtain:

$$|\psi'\rangle = HT^l|\psi\rangle = \frac{1}{\delta^{k-1}}\left(x'|0\rangle + y'|1\rangle\right)$$

5. Repeat until the lde is 0 to obtain a sequence of gates sending $|\psi\rangle$ to $\omega^j|i\rangle$. Optionally, apply a final X gate to send $|\psi\rangle$ to $\omega^j|0\rangle$.
6. Since unitaries preserve orthogonality, the same sequence of gates will send $|\phi\rangle$ to $\omega^{j'}|1\rangle$. Perform a final $T^{j-j'}$ to remove the relative phase between $|0\rangle$ and $|1\rangle$. Then $G_s \ldots G_1 U = \omega^j I$.
7. Return $G_1^\dagger \ldots G_s^\dagger$, which implements U up to a global phase.

Example 10.1.4. Consider the following unitary over $\mathbb{D}[\omega]$:

$$U = \frac{1}{2}\begin{pmatrix} 1 - \omega^3 & -1 + \omega \\ -\omega^2 + \omega^3 & -1 + \omega^3 \end{pmatrix}$$

We need to first express U in terms of a $\mathbb{Z}[\omega]$ matrix multiplied by $\frac{1}{\delta^k}$, where k is the lde of U. This can be accomplished by multiplying U by δ repeatedly

until we get a matrix U' whose entries are in $\mathbb{Z}[\omega]$. For our chosen U, we needed to multiply by δ three times to get a $\mathbb{Z}[\omega]$ matrix, giving us the following expression:

$$U = \frac{1}{\delta^3} \begin{pmatrix} 2 + 3\omega + 2\omega^2 & -1 - \omega + \omega^3 \\ -\omega - \omega^2 - \omega^3 & -2 - 3\omega - 2\omega^2 \end{pmatrix}$$

So, we have an initial lde of 3. We then try to apply T^l for some $l \in \{0, \ldots, 7\}$ followed by an H gate to reduce the lde. Picking $l = 0$ (i.e. just applying an H gate) does not reduce the lde, but $l = 1$ reduces the lde from 3 to 2:

$$\vcenter{\hbox{\includegraphics{circuit1}}} \;=\; \frac{1}{\delta^2} \begin{pmatrix} 1 + \omega - \omega^3 & -1 - \omega - \omega^2 \\ 1 + \omega + \omega^2 & \omega + \omega^2 + \omega^3 \end{pmatrix}$$

The lde is not zero yet, so we do another iteration, this time finding we can reduce the lde at $l = 3$. Namely, if we apply T^3 followed by H, we reduce the lde from 2 to 0:

$$\vcenter{\hbox{\includegraphics{circuit2}}} \;=\; \begin{pmatrix} 0 & -1 \\ -\omega^3 & 0 \end{pmatrix}$$

Our matrix now contains only elements of $\mathbb{Z}[\omega]$, so we know the first column must be a basis vector, up to a phase. We find it is $-\omega^3 |1\rangle$. We can change it to $-\omega^3 |0\rangle$ by applying an X gate:

$$\vcenter{\hbox{\includegraphics{circuit3}}} \;=\; \begin{pmatrix} -\omega^3 & 0 \\ 0 & -1 \end{pmatrix}$$

Finally, we finish by correcting the relative phase with a final application of T^3, obtaining identity, up to a global phase:

$$\vcenter{\hbox{\includegraphics{circuit4}}} \;=\; -\omega^3 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Moving everything but U to the RHS, we conclude that:

$$U \;\propto\; \vcenter{\hbox{\includegraphics{circuit5}}}$$

10.1.2 Approximating arbitrary single-qubit gates

Suppose we want to use the exact synthesis algorithm from the previous section to approximate arbitrary, single-qubit unitaries. We now know that we can build any 2×2 unitary matrix over the ring $\mathbb{D}[\omega]$, and it is not hard to see that any complex number can be approximated to arbitrarily high precision by some element of $\mathbb{D}[\omega]$. Indeed, if we take $\frac{a}{2^j} + \frac{b}{2^k}\omega^2 = \frac{a}{2^j} + \frac{b}{2^k}i$

for $a, b \in \mathbb{Z}$ and some suitably high values of $j, k \in \mathbb{N}$, we can get as close as we like to an arbitrary complex number.

We seem to be most of the way there on coming up with an approximate synthesis algorithm. The problem is, if we go through a complex-valued unitary matrix U element-by-element and approximate each complex number with an element of $\mathbb{D}[\omega]$, odds are we won't get something unitary but just very nearly unitary.

So we need a better idea. Like in the exact synthesis case, inspiration comes from looking at the least denominator exponent. The idea is, for a target unitary matrix U and a fixed error bound $\epsilon > 0$, we progressively raise the denominator exponent k until we can find some $V = \frac{1}{\delta^k} V'$ where

1. V' is a matrix over $\mathbb{Z}[\omega]$,
2. V is unitary, and
3. for some global phase α, $\|V - e^{i\alpha}U\| \le \epsilon$.

It turns out condition 2 is already quite restrictive. Since V is a unitary, its columns need to be normalised. Hence, the norm-squared of the columns of V' must be $|\delta|^{2k}$. Using this fact, we can prove that for fixed k, there are only finitely many V' to choose from.

Exercise 10.3. Show that, for a fixed $k \in \mathbb{N}$, there are only finitely many $x, y \in \mathbb{Z}[\omega]$ such that $|x|^2 + |y|^2 = |K|^2$ for any constant $K \in \mathbb{Z}[\omega]$. Show that this implies there are only finitely many matrices V' over $\mathbb{Z}[\omega]$ such that $V = \frac{1}{\delta^k} V'$ is unitary for any fixed k. *Hint: Use* (10.2) *to get an explicit form for* $|x|^2$ *and* $|y|^2$ *in terms of their integer coefficients.*

Since we already know that there are finitely many V satisfying conditions 1 and 2 above for fixed k, we know that there must also be finitely many V satisfying all 3 conditions. Hence, we can enumerate them for a fixed k. If we find a solution, we accept it. Otherwise, we increase k and try again.

It turns out that there is a way to do this enumeration of candidates satisfying conditions 1–3 efficiently, and that it terminates for $k = O(\log \frac{1}{\epsilon})$. The way this works has to do with properties of certain discrete subsets of the complex plane, which are a bit advanced for our purposes. Hence, we'll finish our story on synthesis here, and make a few more remarks about this in the advanced Section 10.7.1.

10.2   Scalable ZX notation

So far in this book we have used ZX-diagrams where each wire represents a single qubit. As we have seen, this already allows us to do a lot of interesting

stuff. However, sometimes there is repetitive structure in the diagram that we should really try to abstract away, so that we don't miss the forest for the trees. The way we do that will be with **scalable ZX notation**. This notation allows us to compactly represent operations on registers of many qubits, while still maintaining much of the flavour of calculations with standard ZX-diagrams.

We will represent a register of qubits as a single thick wire:

$$\overset{n}{\rule[0.5ex]{3em}{2pt}} \quad := \quad \overline{\overline{\underset{\vdots}{\rule{2em}{0.4pt}}}} \Big\} n \tag{10.4}$$

Sometimes we will need to split a register into two registers, peal one qubit off to do something with, or merge registers together again. For that we introduce a little bit of extra notation:

$$\tag{10.5}$$

We call these operations **divide** and **gather**. They don't actually *do* anything as linear maps (they are just the identity), but they help us compactly write down more complicated maps. They come with some simple rewrites:

$$\tag{10.6}$$

Note that in general we will only label the thick wires with the number of qubits they represent when it is necessary for clarity, and leave it implicit otherwise.

Just being able to represent a qubit register as a single wire is not that useful, so let's introduce some things we can do with them. We will use bold spiders (with a thick border) to represent a product of (unconnected) copies of a Z or X spider as a bold spider:

$$\tag{10.7}$$

Note that in this notation each thick wire connected to the same spider needs to represent the same number of qubits, otherwise it is not a valid diagram. These bolded spiders can be labelled by a phase, and this phase then gets applied to each qubit separately. As the bolded Z- and X-spiders represent non-interacting parallel spiders, all the standard ZX rewrites still apply to them, and hence we can still do spider fusion, strong complementarity, etc. We can push spiders through a dividers and this in fact gives us a type of bialgebra between the Z/X spider and the dividers and gatherers (cf. Section 3.2.4):

$$
\tag{10.8}
$$

Note that this also works when the spider has no additional outputs:

$$
\tag{10.9}
$$

The most important component of the scalable notation, and what makes all of this worthwhile, is a new piece of notation called the **matrix arrow**, or just *arrow* for short, which allows us to represent arbitrary connectivity from m Z-spiders to n X-spiders using an $n \times m$ biadjacency matrix $A \in \mathbb{F}_2^{n \times m}$:

$$
\xrightarrow{\;n\;\overset{A}{\Longrightarrow}\;m\;} \;:=\; \tag{10.10}
$$

Taking the convention that A_i^j represents the entry in the i-th column and j-th row of the matrix A, we have in Eq. (10.10) that $A_i^j = 1$ if and only if the i-th Z-spider on the left is connected to the j-th X-spider on the right. Just using a single matrix arrow we can hence write down an arbitrary phase-free ZX-diagram in parity normal form (Definition 4.2.2), by directly writing the biadjacency matrix A on the matrix arrow.

Concretely, Eq. (10.10) corresponds, up to scalar factors, to a linear map that acts as A on computational basis vectors:

$$
\overset{A}{\Longrightarrow} \quad :: \quad |\vec{b}\rangle \mapsto |A\vec{b}\rangle \tag{10.11}
$$

Note that we treat the bit string $\vec{b}$ as a column vector for the purposes of matrix multiplication. Let's consider some special cases for the matrix arrow, an all zero matrix and the identity:

$$
\overset{0}{\Longrightarrow} \;=\; \quad\quad\quad \overset{I}{\Longrightarrow} \;=\; \quad\quad\quad \tag{10.12}
$$

Using Eq. (10.11), we can see that when we compose matrix arrows, we just calculate the matrix product:

$$\xrightarrow{A}\xrightarrow{B} \quad \propto \quad \xrightarrow{BA} \tag{10.13}$$

We can also prove this diagrammatically by unrolling the definitions and using strong complementarity on all the internal spiders (this is essentially doing the reduction to parity normal form from Chapter 4).

Spiders and arrows interact in nice ways. First, we have two "copy" laws allowing us to push arrows through spiders:

$$\tag{10.14}$$

These can be proven by unrolling the definitions and using strong complementarity and spider fusion.

Example 10.2.1. Let $A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Then:

Second, we can express block matrices in terms of spiders:

$$\tag{10.15}$$

We can combine these operations to get a decomposition of a matrix into

smaller matrices:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad = \quad$$

 (10.16)

We could view this as the definition of the matrix arrow, because this allows us to define it inductively, starting from trivial components and combining these into larger matrices.

The block matrix and composition rules also imply a graphical rule for the sum of two matrices.

Proposition 10.2.2. For any $\mathbb{F}_2$ matrices A, B, we have:

Proof   Starting from the LHS, we can decompose $A = AI$ and $B = BI$ and apply (10.13) and (10.15):

$\square$

10.2.1   Scalable phase gadgets

Collections of phase gadgets turn out to look quite simple when using scalable ZX notation. First, let's see how we can represent a single phase gadget:

 (10.17)

Here $\vec{1}^T$ is the row vector $(1 \; \cdots \; 1)$, and hence the matrix arrow represents a collection of Z-spiders connected to a single X-spider, as needed. If instead

the phase gadget is only connected to a subset of the wires, then we can replace $\vec{1}$ by a vector $\vec{v}$ where $v_i = 1$ iff the gadget is connected to the ith qubit. We can then see how we can represent a composition of phase gadgets in scalable notation:

$$\tag{10.18}$$

This construction generalises to any number of phase gadgets, for which we then get:

$$\tag{10.19}$$

In this $k \times n$ matrix M each of the k rows describes a phase gadget, and each of the n columns corresponds to a qubit. Hence, $M_i^j = 1$ iff the jth phase gadget is connected to the ith qubit.

Note that we can prove the gadget fusion rule of Section 7.1.2 in the scalable setting using what we have seen. This applies when we have two phase gadgets with the same connectivity, and hence the same row appears twice in the matrix:

$$\tag{10.20}$$

Conversely, if we have gadgets with phases that are not $\frac{\pi}{4}$, but multiples of that, we can also represent them like (10.19), by unfusing them into additional rows of the matrix (doing (10.20) in reverse).
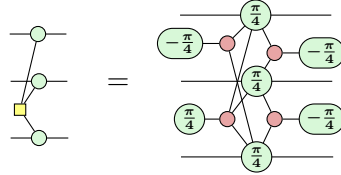
10.3 Rewriting Clifford+T diagrams

In Chapter 9 in order to construct the Toffoli and CCZ gate using more low-level gates, in particular T gates, we used a Boolean Fourier transform to switch from the multilinear phase polynomial $(-1)^{xyz}$ used in the CCZ gate to a phase polynomial built out of XOR terms that can be constructed

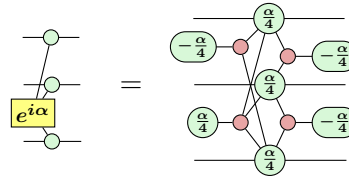using phase gadgets. In essence this all boiled down to the equation:

$$x \cdot y \cdot z = \frac{1}{4}(x + y + z - x \oplus y - x \oplus z - y \oplus z + x \oplus y \oplus z) \qquad (10.21)$$

We used this equation to argue for the following diagrammatic equality:
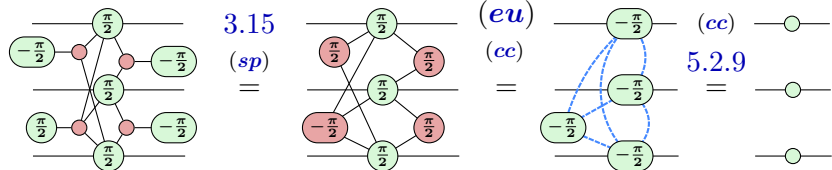

$$(10.22)$$

But there is nothing special about the phase of the CCZ gate here, and in fact we can write a similar equation for a $CCZ(\alpha)$ gate:

$$(10.23)$$

Now when $\alpha = 0$ both sides of this equation are obviously equal to the identity (just copy some spiders and cancel some identity spiders). But this should then also hold for $\alpha = 2\pi$, and then this fact becomes less obvious. On the left-hand side we then have $e^{i2\pi} = 1$ so that it is still the identity, but on the right-hand side we get a bunch of $e^{\pm i2\pi/4} = e^{\pm i\pi/2}$ phases. In that case we can show this by using the Y eigenstate identity of Exercise 3.15, the Euler decomposition of the Hadamard and local complementation:

$$(10.24)$$

In this case we can hence still prove this identity with the tools we have already seen. But when we generalise Eqs. (10.21) and (10.23) to work with more than 3 wires we start getting new and very useful identities. As the number of XOR terms blows up exponentially it will be helpful to introduce a slightly more compact way to talk about them. Note that we can represent a parity like $x_1 \oplus x_3 \oplus x_4$ with a bit string $\vec{y} = 1011$ since $\vec{y} \cdot \vec{x} = y_1 x_1 \oplus y_2 x_2 \oplus y_3 x_3 \oplus y_4 x_4 = x_1 \oplus x_3 \oplus x_4$. We can hence write Eq. (10.21) more compactly as

$$x_1 \cdot x_2 \cdot x_3 = -\frac{1}{4}\sum_{\vec{y} \neq \vec{0}}(-1)^{|\vec{y}|}\vec{y} \cdot \vec{x}.$$

Here $\vec{x} = x_1 x_2 x_3$ and the sum over the $\vec{y}$ goes over all the bit strings $\mathbb{F}_2^3$ except for $\vec{0} = 000$. We can then easily write down the generalisation of this equation to n variables $x_1, \ldots, x_n$ as follows:

$$x_1 \cdot \ldots \cdot x_n \;=\; -\frac{1}{2^{n-1}} \sum_{\vec{y} \neq \vec{0}} (-1)^{|\vec{y}|} \vec{y} \cdot \vec{x}. \tag{10.25}$$

Now if we take $n = 4$, and we consider the CCCZ gate, which applies the phase polynomial $e^{i\pi x_1 x_2 x_3 x_4}$, then applying Eq. (10.25) would result in a bunch of phase gadgets with a phase of $\pm\frac{\pi}{8}$. But this is the Clifford+T chapter, so we want $\pm\frac{\pi}{4}$ phases. We can get those by instead considering the trivial phase polynomial $e^{2\pi i x_1 x_2 x_3 x_4}$. This then results in a constellation of $2^4 - 1 = 15 \pm\frac{\pi}{4}$ phase gadgets:

$$\tag{10.26}$$

While this might look like some sort of confusing alien spacecraft, there is some order to the picture above: it contains all the possible phase gadgets on four qubits: all those with one leg (the $\frac{\pi}{4}$ phases directly on the qubit wires), two legs, three legs, and the single four-legged one. All the gadgets with an odd number of legs have a phase of $\frac{\pi}{4}$, and all the gadgets with an even number of legs have a phase of $-\frac{\pi}{4}$.

Eq. (10.26) is a genuinely new diagrammatic equation, a type of equation we call a **spider-nest identity**. As we will see in this chapter and the next, there are many uses of such equations.

As a first application, note that if we bring the four-legged phase gadget to the other side of the equation that this says that whenever we have a four-legged gadget with a $\pm\frac{\pi}{4}$ phase, we can replace it by a collection of 14 three-, two- and one-legged phase gadgets involving all of the four-legged phase gadget's legs. In fact, we can decompose any n-legged phase gadget with a phase of $\pm\frac{\pi}{4}$ into a collection of phase gadgets with a most 3 legs. For

instance, when $n = 5$:

$$(10.27)$$

So while we would a priori think that we could need $O(2^n)$ different phase gadgets, one for each possible parity, we see that we actually only need $O(n^3)$, only those with at most three legs.

A different use-case for Eq. (10.26) is that we can use it to optimise the number of T gates needed for a circuit. When we have a collection of phase gadgets with $\frac{\pi}{4}$ phases we can look for any subset of four qubits that has many phase gadgets and then use a version of Eq. (10.26) where we have those phase gadgets on the left and all the other parities on the right. Then by applying this equation we essentially 'toggle' which phase gadgets on these four qubits were present. As long as we started out with at least half of all the possible phase gadgets present, so at least 8, we end up with fewer phase gadgets. If we had 8 phase gadgets, then we get $15 - 8 = 7$ phase gadgets at the end. If we had 10, then we end up with $15 - 10 = 5$ of them. The exact phases, whether $+\frac{\pi}{4}$ or $-\frac{\pi}{4}$ is not important for this, since if the signs don't match, this just introduces a $\frac{\pi}{2}$ Clifford phase gadget.

Exercise 10.4. In Eq. (10.26) the two-legged and four-legged phase gadgets have a $-\frac{\pi}{4}$ phase. Show that by unfusing a $-\frac{\pi}{2}$ phase gadget from the four-legged one and applying a set of rewrites similar to those in Eq. (10.24) that we can rewrite it to a collection of phase gadgets all having a $+\frac{\pi}{4}$ phase.

10.3.1 Spider nests as strongly 3-even matrices

Using scalable ZX notation, we can represent a collection of phase gadgets using a parity matrix to describe the connectivity of the gadgets as in (10.19).

The 4-qubit spider nest identity of Eq. (10.26) with 15 gadgets (more specifically, the modified version of Exercise 10.4 where all the phases are $+\frac{\pi}{4}$ so that we can get rid of the repeated rows) then corresponds to the following 15×4 matrix:

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

$$(10.28)$$

Note that we write here the transpose of the matrix to use the space on the page a bit better.

Any spider nest identity will be an identity of the form:
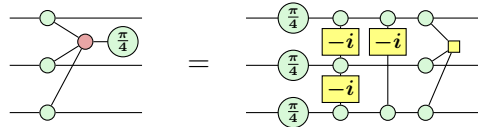
$$
\begin{array}{c}
\text{(10.29)}
\end{array}
$$

This means that when hunting for such identities, we are really looking for a particular type of boolean matrix. To find out what kind of matrices we need, we need to look at the pseudo-Boolean Fourier transform again. This time however, we want to translate phase gadgets back into controlled phase gates. The 'inverse' of the equation (10.25) which relates an AND to a sum of XORs is given by:

$$
x_1 \oplus \cdots \oplus x_n = \sum_{S \subseteq [n]} (-2)^{|S|-1} \prod_{i \in S} x_i. \tag{10.30}
$$

Here $[n] := \{1, \ldots, n\}$, and $|S|$ is the number of elements of S. Hence, each term $\prod_{i \in S} x_i = x_{i_1} \wedge \cdots \wedge x_{i_k}$ has a weight that is a power of 2 in the decomposition. Now, if we are considering this in a phase polynomial, where our phase gadget has a coefficient of $\frac{\pi}{4}$, then in the decomposition the different terms will have a weight of $\pm 2^{|S|} \frac{\pi}{8}$. Since these are phases, we are working modulo 2π so that when $|S| > 3$ each term is a multiple of 2π and disappears:

$$
\begin{aligned}
e^{i \frac{\pi}{4} x_1 \oplus \cdots \oplus x_n} &= \exp\left(i \left(\frac{\pi}{4} \sum_j x_j - \frac{\pi}{2} \sum_{i<j} x_i x_j + \pi \sum_{i<j<k} x_i x_j x_k - 2\pi \cdots \right) \right) \\
&= \exp\left(i \left(\frac{\pi}{4} \sum_j x_j - \frac{\pi}{2} \sum_{i<j} x_i x_j + \pi \sum_{i<j<k} x_i x_j x_k \right) \right) \quad \text{(10.31)}
\end{aligned}
$$

Hence, each phase gadget corresponds to a collection of T gates (the linear terms), CS gates (quadratic terms) and CCZ terms (cubic terms).

$$
\begin{array}{c}
\text{(10.32)}
\end{array}
$$

If instead of a single gadget we have a collection of phase gadgets, then we can add together the respective T, CS and CCZ gates of their decompositions. It turns out that such a circuit can only be equal to the identity if all the gates cancel in the trivial way. That is, each qubit should have a number of

T gates that is multiple of 8 since $T^8 = \text{id}$, each pair of qubits should have a multiple of 4 CS gates and each triple should have an even number of CCZ gates.

Exercise 10.5. Let C be a circuit consisting of T, CS and CCZ gates and suppose we have done all trivial cancellations as described above. Show that if C implements the identity, that then C must be the empty circuit. *Hint: First consider input states $|\vec{x}\rangle$ with Hamming weight 1 to show using $C|\vec{x}\rangle = |\vec{x}\rangle$ that there can't be any T gates, then consider inputs with Hamming weight 2 to show there can't be CS gates, and finally consider input states with Hamming weight 3 to rule out CCZ gates.*

From Eq. (10.31) we see that if a phase gadget involves the qubit j, that then a T gate appears on that qubit. If it connects to both qubits j_1 and j_2 then there will be a CS gate on those qubits, and similarly a CCZ appears when the qubits j_1, j_2 and j_3 are in the phase gadget. Hence, a collection of phase gadget implements the identity when each qubit is part of 0 mod 8 gadgets ($T^8 = I$), each pair of qubits is part of 0 mod 4 gadgets ($\text{CS}^4 = I$), and each triple is part of 0 mod 2 gadgets ($\text{CCZ}^2 = I$).

We can formalise this cancellation property as follows.

Definition 10.3.1. We say a matrix M is **strongly 3-even** when

$$\forall i : \sum_l M_i^l = 0 \ (\text{mod } 8)$$

$$\forall i < j : \sum_l M_i^l M_j^l = 0 \ (\text{mod } 4)$$

$$\forall i < j < k : \sum_l M_i^l M_j^l M_k^l = 0 \ (\text{mod } 2).$$

That is, when each column, product of pairs of columns and product of triples of columns has a Hamming weight that is a multiple of respectively 8, 4 and 2. We say M is **3-even** when all three conditions only hold modulo 2.

When we represent a collection of $\pi/4$ phase gadgets as a set of bit strings $\vec{y}^1, \ldots, \vec{y}^m$, where $y_i^l = 1$ means the lth phase gadget is connected to the ith qubit and set the matrix $M_i^j = y_i^j$ we see that M is strongly 3-even precisely when the gadgets make a spider nest.

Proposition 10.3.2. Let M be a $k \times n$ boolean matrix. Then
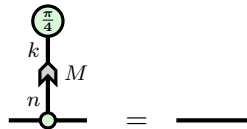
if and only if M is strongly 3-even.

Being just 3-even, as opposed to strongly 3-even, is a weaker condition that means the collection of gadgets is not exactly equal to the identity, but instead are equal to some Clifford unitary: instead of the T gates exactly cancelling due to them appearing a multiple of 8 times, they only appear a multiple of 2 times on each qubit and hence combine into $T^2 = S$ gates, which are Clifford. Similarly, the CS gates appear an even number of times to create CZ gates, and the CCZ gates still completely cancel. For convenience we will also call a set of gadgets that is equal to a Clifford (and hence corresponds to a 3-even matrix) a *spider nest identity*. We can find which Clifford it implements by rewriting all the gadgets using Eq. (10.31). However, if we have gadgets with many legs this can involve exponentially many terms. There is also an efficient way to do it.

Exercise 10.6. Describe an efficient procedure that determines the stabiliser tableau for a circuit consisting of $\frac{\pi}{4}$ phase gadgets with the promise that it implements a Clifford unitary. *Hint: It is diagonal, so all the Pauli Z's trivially commute through. Pushing an X through the circuit however results in some additional $\frac{\pi}{2}$ phase gadgets that can be commuted to the end. If the unitary is Clifford these should all be decomposable into Paulis.*

10.3.2 Proving all spider nest identities

We can now see when a collection of phase gadgets is a spider nest identity: check whether its associated parity matrix is strongly 3-even. Diagrammatically this is not very satisfying however, as it doesn't tell us how to find strongly 3-even matrices or what is required to diagrammatically prove all these identities. In this section we will see that we can build all spider nest identities from one particular one.

To do so, we need to go back to the 4-qubit spider nest Eq. (10.26) which we arrived at by decomposing the 'trivial' phase polynomial $e^{2\pi i x_1 \cdots x_4}$ into phase gadgets using Eq. (10.25). We can similarly get a 5-qubit spider nest by decomposing $e^{4\pi i x_1 \cdots x_5}$. This then will have $2^5 - 1 = 31$ different phase gadgets. Note the factor of 4π instead of 2π which is needed to get a collection of $\pm\frac{\pi}{4}$ phases instead of $\pm\frac{\pi}{8}$ phases. We can in the same way build an n-qubit spider nest by decomposing $e^{2^{n-3}\pi i x_1 \cdots x_n}$, which will result in a spider nest with $2^n - 1$ phase gadgets with a $\pm\frac{\pi}{4}$ phase.

Now, let's take the 5-qubit spider nest, and then 'subtract' the 4-qubit

spider nest from it:

$$
\begin{array}{c}
\boxed{\begin{array}{c}\text{all 1–5}\\\text{legged}\\\text{phase}\\\text{gadgets}\end{array}}\ \left(\boxed{\begin{array}{c}\text{all 1–4}\\\text{legged}\\\text{phase}\\\text{gadgets}\end{array}}\right)^{\dagger}
\quad=\quad
\boxed{\begin{array}{c}\text{all phase}\\\text{gadgets}\\\text{involving}\\\text{qubit 1}\end{array}}
\end{array}
\tag{10.33}
$$

Here on the left-hand side we first have all the phase gadgets on five qubits, but then we subtract from those all the phase gadgets that don't act on the first qubit. On the right-hand side we are then left with precisely those phase gadgets that *do* involve the first qubit. These are hence the parities like x_1, $x_1 \oplus x_2$, ..., $x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$. Since these parities can involve any combination of the other four qubits, there are $2^4 = 16$ such parities (or alternatively you can see that we subtract 15 parities from 31 parities in Eq. (10.33) to arrive at 16 parities).

Exercise 10.7. Generalize Exercise 10.4 by showing that for any $n \geq 4$, the collection of all phase gadgets with phase $+\frac{\pi}{4}$ is the identity. Use this to argue that in the identities described above where we subtract two fully connected spider nests from each other we also get an identity that only has positive $+\frac{\pi}{4}$ phases.

It is this 16-gadget identity (10.33) that turns out to generate all the other ones. We need to do some work to see that though. The first step is to construct such collections of phase gadgets in a more systematic way.

Definition 10.3.3. The *spider-nest maps* $s_n : 1 \to n$ are constructed inductively as follows:

$$\tag{10.34}$$

Intuitively, this inductive definition results in a phase gadget connecting the single input wire to every subset of the output wires. For example:

where the last step follows from applying strong complementarity to the marked spider pair, and then applying spider fusion (**sp**) as much as possible.

Let's formalise this intuitive explanation of s_n using scalable notation. Let B_n be the $n \times 2^n$ matrix whose 2^n rows consist of all n-bit strings. That is, the matrix defined inductively as follows:

$$B_0 = () \qquad B_n = \begin{pmatrix} B_{n-1} & \vec{0} \\ B_{n-1} & \vec{1} \end{pmatrix}$$

where $\vec{0}$ and $\vec{1}$ are respectively the column vectors of all 0's and all 1's. For example, we have:

$$B_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad B_2 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{10.35}$$

Lemma 10.3.4. For all n, we have:

$$\tag{10.36}$$

Proof First, note that:

$$\tag{10.37}$$

Using this equation and the scalable rules, we can prove (10.36) from (10.34) by induction on n:

Now, we can put s_4 in a circuit and it can represent exactly the 16-gadget 5-qubit identity (10.33):

$$(10.38)$$

By plugging ⊙— into all the inputs and yanking the first output to be an input we can present this in a slightly simpler way:

$$(\text{S4})$$

We call this the **(S4) rule**, and it is enough to prove all the spider nest identities. First, let's note that because s_4 disconnects, all the s_n for $n \geq 4$ also disconnect.

Lemma 10.3.5. For $n \geq 4$, the Clifford ZX-calculus augmented with the S4 rule implies:

$$\vcenter{\hbox{\includegraphics{sn}}} \quad = \quad \vcenter{\hbox{\includegraphics{rhs}}} \tag{10.39}$$

Exercise 10.8. Prove Lemma 10.3.5 by induction on n with the base case $n = 4$ being (S4).

In Eq. (10.38) we used s_4 to represent the spider nest consisting of all gadgets connected to the first qubit. By using s_n for $n \geq 4$ and using Lemma 10.3.5 we see then that the collection of all gadgets on $n + 1$ qubits that are connected to the first qubit is also an identity.

We can generalise this to the set of all gadgets connected to the first k qubits. To do this, note that if we connect s_n to an X-spider on the left that we obtain the following:

$$\vcenter{\hbox{\includegraphics{eq1040}}} \tag{10.40}$$

where $\mathbf{1}$ is the $k \times 2^n$ matrix where every entry is 1. Hence, to represent a connection to the first k qubits, we can just compose s_n with an X-spider on its inputs. This then also leads to an identity:

$$\vcenter{\hbox{\includegraphics{eq1041}}} \tag{10.41}$$

Note that the special case of $k = 0$ gives us the set of *all* phase gadgets on a set of wires, like the 15-gadget identity (10.26) (in that case B_n has 16 rows, but the all-zero row of B_n corresponds to an unconnected phase that can be removed as a scalar).

As it turns out, every spider nest identity can be decomposed into a composition of the identities of the form (10.41), and so (S4) indeed suffice to prove all of them. To show this we will need some more machinery however.

10.3.3   Spider nests as Boolean polynomials

In the previous sections we saw that we can reduce a collection of phase gadgets to a series of bit strings denoting the connectivity of the gadgets:

 $\quad \rightsquigarrow \quad x_4, x_1 \oplus x_2 \oplus x_4, x_3 \oplus x_4 \quad \rightsquigarrow \quad 0001, 1101, 0011$
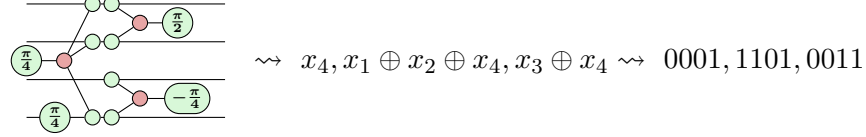
Here we are ignoring the phase gadget with a $\frac{\pi}{2}$ phase, as it is Clifford.

   We saw in Eq (10.19) that these bit strings can be stored in one big matrix and in this way we can efficiently write down a collection of gadgets in scalable notation. Storing them all in a matrix is however not the only way in which we could capture the information of a set of bit strings. We could instead represent them using its Boolean indicator function. That is, to a set of bit strings $S \subseteq \mathbb{F}_2^n$ we associate the function $f_S : \mathbb{F}_2^n \to \mathbb{F}_2$ defined by $f_S(\vec{y}) = 1$ iff $\vec{y} \in S$.

   We should note two important details about representing a collection of gadgets by its indicator function. First, this representation cannot deal with repeated gadgets / bit strings, and so this does not capture the exact phase of the gadgets (whether it is $+\frac{\pi}{4}$ or $-\frac{\pi}{4}$ for instance). This means that when representing a collection of gadgets by its indicator function that we only represent it up to some Clifford information. Second, because $\vec{0}$ corresponds to a phase gadget not interacting with any qubit, we don't care about the value of the function at this value. It would hence be more accurate to write the functions as $f : \mathbb{F}_2^n \setminus \{\vec{0}\} \to \mathbb{F}_2$, but we will ignore this detail for now.

   Any Boolean indicator function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ corresponds to a collection of phase gadgets: a gadget with connectivity $\vec{y}$ is in the collection if $f(\vec{y}) = 1$. Some collections of gadgets are spider nests, so let's call f a **spider-nest function** when its corresponding collection of gadgets forms a spider nest (up to a possible Clifford unitary). I.e. if a collection of bit strings S forms the rows of a 3-even matrix, then f_S is a spider-nest function.

   The indicator function of the 4-qubit spider nest of Eq. (10.26) is the constant function $1_4 : \mathbb{F}_2^4 \to \mathbb{F}_2$ that always returns 1, since every phase gadget is part of the spider nest. The 5-qubit spider nest of Eq. (10.38) that contains all the gadgets using the first qubit has as indicator function $X_1 : \mathbb{F}_2^5 \to \mathbb{F}_2$ that maps $X_1(\vec{y}) = y_1$, since any bit string $\vec{y}$ is part of the set of gadgets if $y_1 = 1$. The spider nest of Eq. (10.41) that uses all gadgets touching the first k qubits has indicator function $X_1 \cdots X_k : \mathbb{F}_2^{k+n} \to \mathbb{F}_2$, which acts as $X_1 \cdots X_k(\vec{y}) = y_1 \wedge \cdots \wedge y_k$.

   These spider-nest functions are examples of Boolean **monomials**. A mono-

mial is a function constructed by multiplying simple bit-indicator functions X_j together. For instance $X_1 X_3 X_4(\vec{y}) = X_1(\vec{y}) \cdot X_3(\vec{y}) \cdot X_4(\vec{y}) = y_1 \wedge y_3 \wedge y_4$. We call the number of bit-indicator functions in such an expression the **degree** of the monomial. For instance the spider-nest function $X_1 \cdots X_k$ has degree k. It turns out that in general, any monomial of degree at most $n - 4$ corresponds to a spider-nest identity. Indeed, $X_1 \cdots X_k : \mathbb{F}_2^{k+n} \to \mathbb{F}_2$ for $n \geq 4$ is a spider-nest function. By permuting the qubits these give us all monomials of degree at most $n - 4$. The number 4 in $n - 4$ comes from the fact that the smallest spider-nest identity, corresponding to the Boolean function 1_4 acts on four qubits.

Note that the spider-nest functions form a linear space: suppose that both $f_{S_1}, f_{S_2} : \mathbb{F}_2^n \to \mathbb{F}_2$ correspond to spider nests. Then if we take the composition of all the phase gadgets with parities in S_1 and that of S_2 we end up with a new set of phase gadgets covering all the parities in S_1 and of S_2. However, the gadgets that are in both S_1 and S_2 will fuse and hence get a Clifford phase. We are ignoring the Clifford unitaries, so we see then that the collection of *non-Clifford* phase gadgets corresponds to the symmetric difference $S_1 \Delta S_2$. The indicator function is then $f_{S_1 \Delta S_2} = f_{S_1} \oplus f_{S_2}$. As this XOR of functions is just the sum in $\mathbb{F}_2$, we see that any sum of spider-nest functions is again a spider-nest function, so that the spider-nests form a linear subspace of all Boolean functions. In particular, we can take a sum of monomials that are all spider-nest functions and create a **Boolean polynomial** that is a spider-nest function.

Any Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ can be written in a unique way as a polynomial $f = \bigoplus_{\vec{a}} \lambda_{\vec{a}} X_1^{a_1} \cdots X_n^{a_n}$ where $\lambda_{\vec{a}} \in \mathbb{F}_2$ are the coefficients that determine which monomials are in the decomposition of f. The degree of a polynomial is then the maximum degree of its monomials. As a sum of spider-nest functions is still a spider-nest function, we then see that any Boolean polynomial of degree at most $n - 4$ is a spider-nest function. What about the converse? Does any spider-nest function correspond to a degree at most $n - 4$ polynomial?

Lemma 10.3.6. A matrix M with n columns is 3-even if and only if its indicator polynomial f_M is of degree at most $n - 4$.

Proof Let M' be a matrix obtained from M by removing all repeated pairs of rows. M is 3-even if and only if M' is, and both matrices have the same indicator polynomial. Hence, we can assume without loss of generality that M has no repeated rows. Write $f = f_M$ for the indicator function of M.

Let $P(r, n)$ denote the set of n-bit polynomials of degree at most r. We need to show that $f \in P(n - 4, n)$. Define an inner product on Boolean

functions by $\langle g_1, g_2 \rangle = \bigoplus_{\vec{y}} g_1(\vec{y}) \wedge g_2(\vec{y}) = \sum_{\vec{y}} g_1(\vec{y}) g_2(\vec{y}) \pmod 2$. We call g_1 and g_2 *orthogonal* when $\langle g_1, g_2 \rangle = 0$ and we define $P(r, n)^{\perp}$ as the space of functions that are orthogonal to all functions in $P(r, n)$. We claim that $P(r, n)^{\perp} = P(n - r - 1, n)$. With this claim it then remains for us to show that $f \in P(3, n)^{\perp}$. Let's do that first.

Let $g = X_j f$. This is then a polynomial with $g(\vec{b}) = 1$ iff $b_j = 1$ and $f(\vec{b}) = 1$. Hence $\sum_{\vec{b}} g(\vec{b})$ is equal to the Hamming weight of the jth column of M. This also works for products of columns: for $h = X_i X_j X_k f$, we have $\sum_{\vec{b}} h(\vec{b})$ equal to the Hamming weight of the element-wise product of the i, j and kth rows, which is hence zero mod 2 because of 3-eveness. Now $\sum_{\vec{b}} h(\vec{b}) \pmod 2 = \langle X_i X_j X_k, f \rangle$ so that f is orthogonal to all degree-3 monomials $X_i X_j X_k$. These span $P(3, n)$, and hence $f \in P(3, n)^{\perp}$ as desired.

Now to prove the claim $P(r, n)^{\perp} = P(n - r - 1, n)$ first note that if a polynomial f of n variables has degree less than n, then $\sum_{\vec{b}} f(\vec{b}) = 0 \pmod 2$. This is easy to check for monomials, as any monomial of degree $< n$ must omit some variable x_j, so that

$$\sum_{\vec{b}} f(\vec{b}) = \sum_{\vec{b}, b_j = 0} f(\vec{b}) + \sum_{\vec{b}, b_j = 1} f(\vec{b}) = 0 \pmod 2$$

By $\mathbb{F}_2$-linearity of the map $f \mapsto \sum_{\vec{b}} f(\vec{b}) \pmod 2$ this then holds for all polynomials. Now, for any polynomial f of degree at most r and g of degree at most $n - r - 1$, $f \cdot g$ has degree at most $n - 1$. Hence $\langle f, g \rangle = \sum_{\vec{b}} (f \cdot g)(\vec{b}) = 0 \pmod 2$. This implies $P(n - r - 1, n) \subseteq P(r, n)^{\perp}$. To show this is actually an equality, we will do dimension counting. Note that for a $\mathbb{F}_2$-vector space V and $A \subseteq V$ we have $\dim(A^{\perp}) = \dim(V) - \dim(A)$ because the dimension of $A^{\perp}$ is restricted by independent linear equations specified by a basis of A. Since $P(r, n)$ has the monomials of degree $\leq r$ as its basis, $\dim(P(r, n)) = \sum_{d=0}^{r} \binom{n}{d}$. By manipulating binomial coefficients, we can then see that:

$$\dim(P(r, n)) + \dim(P(n - r - 1, n))$$

$$= \sum_{d=0}^{r} \binom{n}{d} + \sum_{d=0}^{n-r-1} \binom{n}{d} = \sum_{d=0}^{r} \binom{n}{d} + \sum_{d=r+1}^{n} \binom{n}{d} = 2^n = \dim(\mathbb{F}_2^{2^n}),$$

so that $\dim(P(n - r - 1, m)) = \dim(P(r, n)^{\perp})$ and these spaces must be equal. $\square$

We then see that we have proven the following.

Theorem 10.3.7. Let $\vec{y}^1, \ldots, \vec{y}^k$ describe the connectivities of k gadgets with a $\frac{\pi}{4}$ phase acting on n qubits. Then the following are equivalent.

- The gadgets form a spider-nest identity (i.e. the circuit is equal to a Clifford).
- The matrix with entries $M_i^j = y_i^j$ is 3-even.
- The indicator polynomial of the set $\{\vec{y}^1, \ldots, \vec{y}^k\}$ is of degree at most $n - 4$.

Theorem 10.3.8. The Clifford ZX rules plus the (S4) rule suffice to prove all spider-nest identities. That is, given any collection of $\frac{\pi}{4}$ gadgets that implements a Clifford unitary, we can rewrite this into this Clifford unitary using just the regular Clifford ZX rewrite rules together with (S4).

Proof Let M be the $n \times k$ 3-even matrix describing a spider-nest identity. Then its corresponding indicator polynomial f_M is a sum of monomials $f_M = \sum_j^l m_j$ of degree at most $n - 4$. Let the corresponding matrices of these monomials be $M_1, \ldots, M_l$. We have already shown how to prove the spider-nest identities corresponding to the M_j in Eq. (10.41), hence we can freely introduce them into the circuit of gadgets described by M:

$$
\underset{M}{\overset{\frac{\pi}{4}}{\vcenter}} \overset{(10.41)}{=} \quad \underset{M}{\overset{\frac{\pi}{4}}{\vcenter}} \underset{M_1}{\overset{\frac{\pi}{4}}{\vcenter}} \cdots \underset{M_l}{\overset{\frac{\pi}{4}}{\vcenter}} \quad \overset{(\boldsymbol{sp})}{\underset{(10.16)}{=}} \quad \underset{L :=}{\overset{\frac{\pi}{4}}{\vcenter}} \begin{pmatrix} M \\ M_1 \\ \vdots \\ M_l \end{pmatrix}
$$

Then, the indicator polynomial of L is $f_M + \sum_j m_j = 0$. Hence, every row in L appears an even number of times. Using gadget-fusion, we can therefore reduce all angles to integer multiples of $\pi/2$. Hence, the entire diagram is then Clifford and can be rewritten into a Clifford circuit. $\qquad\square$

We see then that when we restrict to just thinking about what we can do with diagrams, all the complexity of strongly 3-even matrices and degree $n - 4$ Boolean polynomials reduces to just adding (S4) to the Clifford rules. Do note though that in the proof above we needed to know about Boolean polynomials and its decomposition into low-degree monomials in order to find which rewrites we should be applying to prove the spider-nest identity. In addition, the matrices corresponding to the monomials M_j might contain exponentially many rows, and hence this rewriting is not efficient. In fact, it seems very likely that an efficient rewrite strategy for spider nests should not exist (see Exercise 10.9).

Forgetting about all these details again, we can see that we can rephrase this result into a completeness result, which very neatly ties in some of the earlier completeness results we have seen.

Theorem 10.3.9. The Clifford rules plus (S4) are complete for CNOT+T

circuits. That is, given two CNOT+T circuits U and V written as ZX-diagrams, we can rewrite U into V using just the Clifford rules and (S4).

Proof Note that we can trivially rewrite U to $UV^\dagger V$ by consecutively introducing pairs of cancelling gates from $V^\dagger$ and V. Hence, if we can show that $U^\dagger V$ can be rewritten to the identity we are done. In Section 7.1 we saw how we can write any CNOT+Phases circuit into a layer of phase gadgets followed by a CNOT circuit. Now since $U^\dagger V = \mathrm{id}$, it must be the case that both the phase gadget part and the CNOT circuit implement the identity. Hence, we can use the completeness of phase-free ZX (Theorem 4.3.6) to rewrite the CNOT circuit into the identity and Theorem 10.3.8 to rewrite the collection of phase gadgets, which necessarily forms a spider-nest identity, into a Clifford. This Clifford still must be equal to the identity, and hence by Clifford completeness (Theorem 5.5.7) it can be rewritten into the identity. □

10.4 Advanced T-count optimisation

We have now seen that there is a large number of configurations of T gates that actually correspond to Clifford circuits. Getting rid of non-Clifford parts of a circuit is usually a good thing, as we've seen that we can do a lot of rewriting and simplification with the Clifford parts of a circuit. In addition, as we will see in Chapter 11, the T gate in particular is quite costly to implement in many fault-tolerant architectures, and so we want to include as few of them as possible.

The spider-nest identities suggest a simple rewrite strategy to optimise the number of T gates in a Clifford+T circuit. First, since spider nest identities apply to a collection of phase gadgets, and hence to CNOT+T circuits, we need to split up our Clifford+T circuit into CNOT+T subcircuits. We can view the Clifford+T gate set as consisting of CNOT, Hadamard, S and T. Of these, only the Hadamard is not in the CNOT+T set, and so we need to 'split our circuit on Hadamards'. Then, pick a number of identities, preferably not containing too many gadgets and not acting on too many qubits. We want to see where in the CNOT+T circuit we can apply an identity so that it reduces the T-count. This is done as described in the beginning of Section 10.3: for each identity in our list, we check whether more than half of the gadgets are present in the circuit. If this is the case, then we add all the gadgets from the identity to the circuit, which by gadget fusion makes the matching gadgets already present in the circuit into Cliffords, and adds the other gadgets as non-Cliffords. This is repeated greedily until none of

the identities has an overlap of more than half with the gadgets present in the circuit.

Note that if we have n qubits in our circuit, then to match an m-qubit spider-nest identity, we need to check $\binom{n}{m}$ different groupings of m qubits to see whether the identity 'matches' there. Hence, as long as m is bounded, the complexity of this algorithm is polynomial: $O(n^m)$. In practice, only checking spider nests with up to 5 qubits, and using a simple heuristic based on the 'density' of the number of gadgets on a set of qubits, the run-time can be made quite reasonable.

Of course such an algorithm is only a heuristic, and is heavily dependent on the type of identities we include in our search, and since we are applying the identities greedily, you might get stuck in a local optimum. The problem of optimisation of CNOT$+T$ circuits using spider nests is actually related to two well-known problems in computer science, which offer interesting and useful perspectives on this problem.

10.4.1 Reed-Muller decoding

We have seen that n-qubit spider nests correspond to Boolean polynomials of degree at most $n - 4$. In addition, we have a correspondence between Boolean functions f and collections of spider nests specified by their parities as $\{\vec{y} \mid f(\vec{y}) = 1\}$. Let's call the set of all n-bit Boolean functions B_n, and the set of n-bit spider-nest functions $S_n \subseteq B_n$.

Now, if we naively implement the set of gadgets corresponding to f by just implementing each of the gadgets in turn, then this will require $|f| := \sum_{\vec{y} \neq \vec{0}} f(\vec{y})$ number of T gates. We call $|f|$ the **Hamming weight** of f. It is the number of 1s in f's truth table. Note that we do not include $f(\vec{0})$ in this sum, as this corresponds to the trivial phase gadget not connected to any qubit.

However, we don't have to naively implement f, because we know that all $g \in S_n$ are actually *free* to implement: these correspond to Cliffords and hence do not require any T gates. Instead of directly implementing f, we can implement $f \oplus g$, which corresponds to applying the spider nest identity of g to the collection of gadgets of f. The cost of this implementation will then be $|f \oplus g|$. To implement f with as few T gates as possible we are hence looking for a $g \in S_n$ such that $|f \oplus g|$ is minimal.

Let's state this problem again in a slightly different way, and a bit more abstractly. We have a vector space V with a specified subspace $S \subseteq V$. Given a vector $v \in V$, we want to find the *closest* $s \in S$ to v, i.e. such that $v - s$ is as small a vector as possible (in some norm). This is known as a **linear**
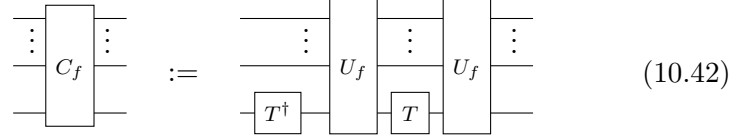
decoding problem, where S is our code space consisting of code words $s \in S$, and v is our 'noisy message' we are trying to decode. We will spare you the details for now, as we will have a lot more to say about linear codes in Chapter 11.

In our case the subspace S_n consists of the spider-nest functions, which we know to be all the degree $n - 4$ Boolean polynomials. This code space actually has a name: it is the degree $n - 4$ **Reed-Muller code**, and hence optimising the T-count of a CNOT+T circuit corresponds to decoding the Reed-Muller code. To summarise this optimisation approach now a bit more concretely: We start with a unitary U that is built out of phase gadgets with phases that are multiples of $\frac{\pi}{4}$. We take its corresponding Boolean function $f \in B_n$. We then find the 'closest' degree $n - 4$ polynomial $g \in S_n$ such that $|\vec{f} \oplus g|$ is as low as possible (corresponding to decoding the Reed-Muller code). We then implement the circuit U' corresponding to $f \oplus g$ by composing its phase gadgets. We know that U is equal to U' up to some Clifford. We find the Clifford C such that $U = U'C$ (see Exercise 10.6). Then $U'C$ is our new circuit, and this has T-count $|f \oplus g|$.

Reed-Muller codes are used a lot in practice and their decoding problem has been extensively studied. So in principle we could use such a decoder to optimise the T-count of a circuit. However, the codes that are used in practice mostly have a size, corresponding in our case to the number of qubits n, that is not too large. However, we don't want to restrict to just small n, so that is a problem. Decoding Reed-Muller codes for large n is believed to be a hard problem, so we don't expect there to be an efficient algorithm to optimally minimize the T-count of a CNOT+T circuit. For optimising the T-count of general Clifford+T circuits (that are allowed to contain Hadamards), there is a simple argument to see that T-count optimisation is NP-hard.

Exercise* 10.9. In this exercise we will work towards a simple proof that determining whether a Boolean function is satisfiable reduces to T-count optimisation of general Clifford+T circuits, and hence is T-count optimisation is NP-hard. Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be some Boolean function, which is described as some poly-size Boolean expression consisting of AND, XOR and NOT terms. We we want to determine whether there is a $\vec{x} \in \mathbb{F}_2^n$ such that $f(\vec{x}) = 1$. We have seen in Chapter 9 how we can construct U_f, the $(n + 1)$-qubit unitary acting as $U_f|\vec{x}, y\rangle = |\vec{x}, y \oplus f(\vec{x})\rangle$, using Clifford+$T$ gates.

1. Let the circuit C_f be defined as follows:

$$ (10.42) $$

Show that C_f is a diagonal unitary which can be described by the following path-sum expression:

$$ C_f|\vec{x}, y\rangle \;=\; e^{i\frac{\pi}{4}(1-2y)f(\vec{x})}|\vec{x}, y\rangle. \qquad (10.43) $$

2. Show that if f is not satisfiable or **everywhere** satisfiable (meaning $f(\vec{x}) = 1$ for all $\vec{x}$) that then C_f is a Clifford unitary (up to global phase) and hence can be implemented with T-count zero.

3. If f is satisfiable but not everywhere satisfiable, then by definition there exist $\vec{z}_1$ and $\vec{z}_2$ such that $f(\vec{z}_1) = 1$ and $f(\vec{z}_2) = 0$. Then it is easy to see from Eq. (10.43) that

$$ C_f|\vec{z}_1, 0\rangle \;=\; e^{i\frac{\pi}{4}}|\vec{z}_1, 0\rangle \qquad \text{and} \qquad C_f|\vec{z}_2, 0\rangle = |\vec{z}_2, 0\rangle. $$

Show that in this case C_f is not Clifford and hence it's T-count non-zero, by finding a Pauli string $\vec{P}$ such that $C_f^\dagger \vec{P} C_f$ is not in the Pauli group. *Hint: You don't have to calculate the full operator $C_f^\dagger \vec{P} C_f$. For the right choice of $\vec{P}$ it is enough to observe that $C_f^\dagger \vec{P} C_f |\vec{z}_1, 0\rangle$ maps $|\vec{z}_1, 0\rangle$ to something that a member of the Pauli group could not.*

Now, note that if we could efficiently determine the optimal T-count of any circuit, then for a given f we could construct C_f and ask whether it's T-count is zero: if it is not then we know it has to be satisfiable. If it is zero, then either the circuit is not satisfiable or everywhere satisfiable. We then just check the value $f(0 \cdots 0)$ to see which is the case. We can hence in either case efficiently determine whether f is satisfiable, an NP-complete task.

10.4.2 Symmetric 3-tensor factorisation

There is another way we can formalise the optimisation of the number of T gates in a diagonal CNOT+T circuit. To do this, we again need to consider the multilinear decomposition of a phase gadget as in Eq. (10.31):

$$ e^{i\frac{\pi}{4}x_1 \oplus \cdots \oplus x_n} \;=\; \exp\left(i\left(\frac{\pi}{4}\sum_j x_j - \frac{\pi}{2}\sum_{i<j} x_i x_j + \pi \sum_{i<j<k} x_i x_j x_k \right) \right) \qquad (10.44) $$

In particular, the action of an arbitrary collection of $\frac{\pi}{4}$ phase gadgets can be represented by some degree-3 multilinear polynomial

$$\frac{\pi}{4} \sum_j a_j x_j + \frac{\pi}{2} \sum_{i<j} b_{ij} x_i x_j + \pi \sum_{i<j<k} c_{ijk} x_i x_j x_k.$$

Here the coefficients a_j can be taken modulo 8, b_{ij} modulo 4 and c_{ijk} modulo 2. By Exercise 10.5 two collections of phase gadgets correspond to the same polynomial if and only if they implement the same linear map. Similarly, two phase gadget circuits are equal up to a *Clifford* when the coefficients of their polynomials have the same parities $a_j \pmod 2, b_{ij} \pmod 2, c_{ijk} \pmod 2$. Hence, if we don't care about the Clifford part of a computation, we can forget that the coefficient a_j should be taken modulo 8, and instead take it modulo 2. Similarly we can take b_{ij} modulo 2 instead of 4.

This is nice, because this information about the coefficients modulo 2 can be captured in a single object.

Definition 10.4.1. An n-dimensional binary 3-tensor is an element $S \in \mathbb{F}_2^{n^3}$ where we label the components of the vector by S_{ijk} for indices $1 \le i, j, k \le n$. We say S is **symmetric** when $S_{ijk} = S_{jik} = S_{ikj} = S_{kji} = S_{kij} = S_{jki}$ for all indices i, j, k, i.e. when S is invariant under permuting its indices.

We define the symmetric 3-tensor S_{ijk} corresponding to a degree-3 multilinear polynomial by setting

$$\begin{aligned}
S_{iii} &= a_i \\
S_{ijj} &= b_{ij} \\
S_{ijk} &= c_{ijk}
\end{aligned}$$

for $i < j < k$. All other coefficients of S are then completely determined by symmetry. It is clear that from any symmetric 3-tensor we can also read of the coefficients of a degree-3 multilinear polynomial, which then corresponds to a phase gadget circuit. Note though that if we start with a phase gadget circuit, find its 3-tensor, and then translate it back into a phase gadget circuit, that we do lose some Clifford information in the process, and the resulting circuit is only equal to the original one up to some Clifford.

Let's look at what the 3-tensor of a single phase gadget looks like. Let $\vec{y} \in \mathbb{F}_2^n$ describe the connectivity of the phase gadget. Then by Eq. (10.44) the corresponding multilinear polynomial has coefficients $a_i = y_i$, $b_{ij} = -y_i y_j$ and $c_{ijk} = y_i y_j y_k$. That is: there is a T gate on all the qubits the gadget is connected to, a $\mathrm{CS}^\dagger$ on all pairs of qubits it is connected to, and a CCZ on all triples of qubits it is connected to. This means the 3-tensor corresponding to the gadget $S^{\vec{y}}$ has a particularly simple form: $S_{ijk}^{\vec{y}} = y_i y_j y_k$. 3-tensors

that have this form are said to have *rank 1*. When we have a set of gadgets $\vec{y}^1, \ldots, \vec{y}^k$, the circuit has the corresponding tensor $S = S^{\vec{y}^1} + \cdots + S^{\vec{y}^k}$, and hence it is a sum of rank 1 tensors. Conversely, any way to write S as a sum of rank 1 tensors corresponds to a way to implement it as a series of phase gadgets.

Definition 10.4.2. A symmetric 3-tensor is **rank 1** when it is of the form $S_{ijk} = y_i y_j y_k$ for some vector $\vec{y}$. For a general symmetric 3-tensor S its **rank** is the minimal number of terms needed to write S as a sum of rank 1 tensors, and we call such a sum a **decomposition** of S.

We see then that we have the following strategy for optimising the T-count of a phase gadget circuit: first find its corresponding 3-tensor. Then find a decomposition of this tensor into as few rank 1 tensors as possible. These rank 1 tensors directly correspond to an optimised set of phase gadgets which implements the same linear map as the original circuit up to a Clifford. Find what Clifford this is using the procedure of Exercise 10.6. The resulting T-count of the circuit is exactly equal to the rank of the decomposition we found. In particular, determining the optimal T-count of a given phase gadget circuit is equivalent to determining the rank of a symmetric 3-tensor. Unfortunately, determining the rank of a (symmetric) 3-tensor is believed to be a hard problem. Fortunately, there are some good heuristics that try to find low rank decompositions that work well in practice. We will say a bit more about these in the References of this chapter.

10.5 Catalysis

We saw way back in Section 3.3.1 that we can implement a T gate by injecting the $|T\rangle := T|+\rangle$ magic state into the circuit:
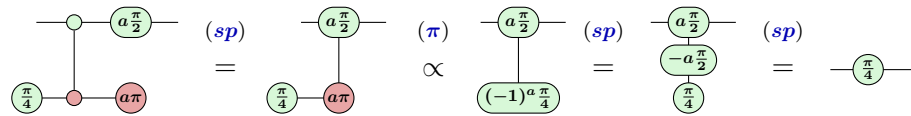
This is useful, because it is sometimes difficult to directly implement the T gate (as we will see in the next chapter), and instead having the ability to prepare magic states 'offline' and inject them when needed is preferable.

However, in this circuit we *consume* the magic state when we inject the T gate. Wouldn't it be nice if we could preserve this state instead? This does turn out to be possible, if we use a different injection circuit that contains some other non-Clifford gates.

In particular, using a CS gate and a single $|T\rangle$ state, we can apply a T gate and get the starting $|T\rangle$ state back:

$$\tag{10.45}$$

Here we used the decomposition of the CS gate written using an H-box (Section 9.2) into elementary gates:

$$\tag{10.46}$$

Just using Clifford operations and CS gates, it is not possible to construct a T gate. We can see this, because the matrices produced by the Clifford+CS gate set have entries in a ring that does not include $e^{i\frac{\pi}{4}}$. However, with Eq. (10.45) we see that as soon as we have just *one* $|T\rangle$ state available to us, we can use CS gates to apply as many T gates as we want:

This is an example of what we call **catalysis**: a process that needs some resource to be present, but doesn't consume that resource. In this case the resource is $|T\rangle$ and the process is the implementation of a T gate using a CS gate.

Another example of catalysis is using a $|CCZ\rangle := CCZ|+++\rangle$ magic state, Clifford operations and a single T gate in order to get 3 $|T\rangle$ states out:

$$\tag{10.47}$$

So again, if we can perform CCZ and Clifford gates and have just a single $|T\rangle$ available, then we can inject as many T gates as we want.

There are three things we can do with catalysis of $|T\rangle$ magic states that we will explore in this section. First, in some cases it turns out to be easier or cheaper to perform a CS or CCZ gate than a T gate, and then these methods allow us to save resources. Second, they allow us to prove that some gate sets are already *computationally universal*, even if they are not (obviously) approximately universal for unitary synthesis. And third, catalysis gives us a nice way to extend complete graphical calculi to larger domains.

10.5.1   Catalysis as a resource for compilation

In this section we will see how catalysis can be used to derive an efficient way to implement small angle rotations.

To do that we first need to generalise Eq. (10.45) to allow us to implement controlled-phase gates. To see how this works it will be helpful to first write Eq. (10.45) in circuit notation:

$$
\begin{array}{c}
\begin{array}{c}
\underline{Z(\alpha)} \\
|Z(\alpha)\rangle \text{———}
\end{array}
\quad = \quad
\begin{array}{c}
\bullet \quad \bullet \\
|Z(\alpha)\rangle \oplus \underline{Z(2\alpha)}
\end{array}
\end{array}
\tag{10.48}
$$

Here we wrote a slightly more general circuit where we replace the T and controlled-S gates with $Z(\alpha)$ and controlled-$Z(2\alpha)$ gates. As a shorthand we write $|Z(\alpha)\rangle := Z(\alpha)|+\rangle$ as a generalisation of $|T\rangle = T|+\rangle$. Since this is a circuit equality that holds on the nose (with a correct global phase), it should continue to hold when we add additional control wires:

$$
\begin{array}{c}
\begin{array}{c}
\vdots \; \bullet \\
\underline{Z(\alpha)} \\
|Z(\alpha)\rangle \text{———}
\end{array}
\quad = \quad
\begin{array}{c}
\bullet \; \vdots \; \bullet \\
\bullet \quad \bullet \\
|Z(\alpha)\rangle \oplus \underline{Z(2\alpha)}
\end{array}
\end{array}
\tag{10.49}
$$

We can prove this is correct using some H-box rules (see Figure 9.1):
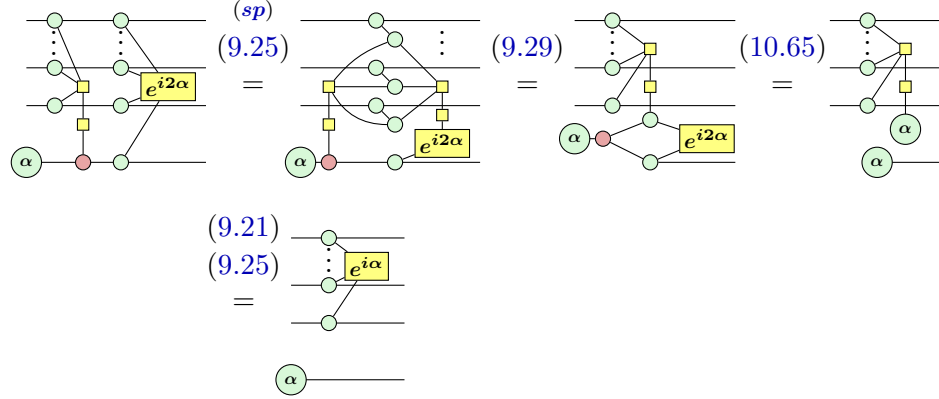
Exercise 10.10. Applying Eq. (10.49) to implement a CS gate requires using both a Toffoli and a CCZ. It turns out there is a different construction that only requires a single CCZ and some *real* Clifford operations (those that do not contain numbers with an imaginary part like CNOT, Hadamard, CZ, X). Prove the following identity:
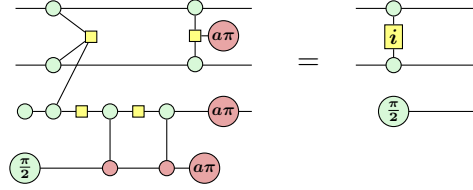
Hint: First prove the identity for $a = 0$, and then show for $a = 1$ how the $a\pi$ phase can be pushed through the circuit to cancel

Because we can apply catalysis equally well to controlled phases, we can start iterating the procedure producing bigger and bigger controlled-phase gates, where the phase being controlled is also increasingly large. For instance, if we want to implement a T gate, we can do the following:
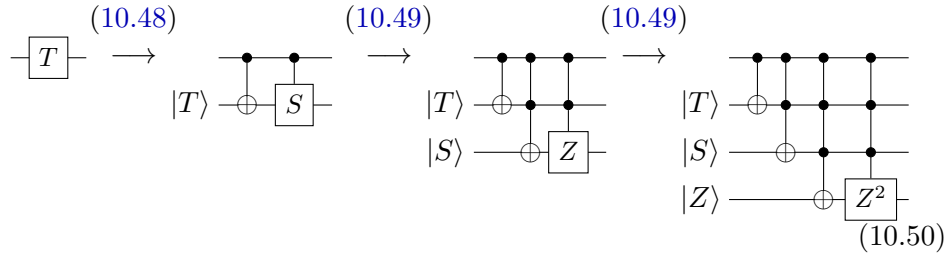
Here in the last step we are left with a controlled Z^2 operation. But since $Z^2 = $ id this does not do anything and we can remove it. So at this point

we can stop the iteration of the catalysis. We see then that we can implement a T gate just using multiple-controlled Toffoli gates, if we have the right catalysis states lying around. This procedure works to implement any $Z(2\pi/2^k)$ gate: we then get a ladder of k Toffoli gates. We have actually seen such a Toffoli ladder before: in Exercise 9.17 we saw that this is actually a controlled-decrementer circuit that decreases the value of an n-bit number by 1, controlled on the top wire. In that same exercise we saw that we can build a subtraction circuit if we make a ladder of these controlled-decrementers. For this reason, when we apply a subtraction circuit to a collection of catalysis states, this implements phase gates on on the top qubits:

$$(10.51)$$

$$(10.52)$$

The reason this is nice, is because in Section 9.5 we found a very efficient construction of the adjoint of the subtraction circuit: the adder. So if we can transform Eq. (10.51) slightly, so that it uses an adder instead of a subtracter, this gives us a way to efficiently implement a whole collection of phase gates at once. The way we do this is by taking Eq. (10.51) and composing both sides on the right by $\text{Add} = \text{Sub}^\dagger$, and on the left by $(T \otimes S \otimes Z)^\dagger$. After cancelling with the adjoints we are then left with the following equation:

We showed the construction here for 3 bits, but this works for any number of bits n, in which case the smallest phase we implement is $Z(2\pi/2^n)$. While this is nice and all, this might not seem immediately useful: we have this pattern of phase gates appearing in parallel, where we have a small-angle phase gate, a slightly-large-angle one, and so on. You might wonder, surely it will not happen often that we can use this exact pattern of phases in a real quantum circuit, and you would be wondering right. However, with the magic of ancillae we can pick and choose exactly which phases we want to appear and where. We can transfer the application of a phase gate to a zeroed ancilla:

 (10.53)

Now when we have a complicated phase, we can decompose it into simple components, and put each of these onto its own ancilla. Suppose for instance we want to implement the phase $Z(\frac{11}{8}\pi)$. We can then write 11 bitwise as 1011 so that $Z(\frac{11}{8}\pi) = Z(2\pi/2^4(2^3 + 2^1 + 2^0))$. We can then put each of these component phases onto their own ancilla to get:

 (10.54)

We have here also sneakily added a zeroed ancilla that gets a $Z(\frac{\pi}{2})$ applied that does nothing. We need this qubit though to complete the pattern: we see then that we get the right shape needed to use Eq. (10.52). However, note that Eq. (10.52) has adjoint phases, instead of the actual phases we need. There are multiple ways we can deal with this. One way is to realise that for phase gates, the adjoint is the conjugate: $T^\dagger = \overline{T}$. Hence, if we take the conjugate of both sides of Eq. (10.52) we do get the right phases. Since the Adder is a real matrix, this stays the same, but the states needed for the catalysis also flip: $\overline{|T\rangle} = |T^\dagger\rangle$. We then have everything we need to produce

the circuit we are after:

$$
\underset{\frac{11}{8}\pi}{\boxed{}} = \qquad (10.55)
$$
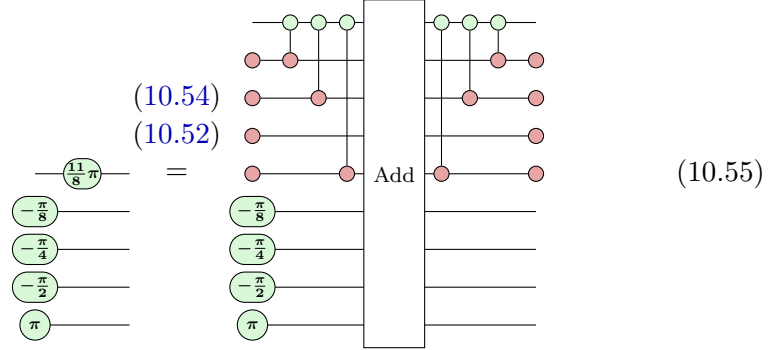
Well, that certainly seems like overkill. Why would we go through all this trouble just to prepare a single phase rotation. Well, it turns out that in a fault-tolerant setting we can't just go and do whatever gate we want to do. We are restricted to just a small set of gates we can (cheaply) implement. So if our computation requires us to do some phase rotation on a weird angle, we have to find a way to this with the gates that we have access too. One way to do this would be to approximate the phase rotation by combining together unitary Clifford+T gates like in Section 10.1.2. But as we have now seen, another way to do it is to prepare just a single copy of each $|Z(2\pi/2^k)\rangle$ state to serve as a catalyst which can be reused, and then apply some CNOTs and an adder. Because the catalysts can be reused, the asymptotic cost of this procedure is just the cost of the adder and the CNOTs. Let's calculate more accurately what the cost then is.

Suppose we want to implement a phase gate with angle α up to a precision ε. We then find the smallest n such that $2^{-n} < \varepsilon$. We can then approximate α by a phase $\hat{\alpha} = a 2\pi/2^n$ where $a < 2^n$ is an integer such that $|\alpha - \hat{\alpha}| < 2^{-n} < \varepsilon$. It hence suffices to implement $\hat{\alpha}$ instead. Since a is an n-bit number, we can implement the $Z(\hat{\alpha})$ gate using a generalisation of Eq. (10.55) to n bits. We saw in Section 9.5 that we can implement an n-qubit quantum adder using n Toffoli gates. In a fault-tolerant architecture the implementation of these Toffoli gates is what dominates the cost. As $2^{-n} < \varepsilon$ we have $\log_2 1/\epsilon < n$, and hence we can express the cost also in terms of the error budget, and say that we require $\log_2 1/\epsilon$ Toffoli gates. Decomposing each Toffoli with 4 T gates we see that we can equivalently say that the cost is $4 \log_2 1/\epsilon$ T gates.

10.5.2 Computational universality via catalysis

Using catalysis we can replace any occurrence of some gate (like$T$) with some other gate (like CS), as long as we have access to some special catalyst

ancilla state. We can use this idea to prove that certain gate sets are also universal for quantum computing. In this section we will demonstrate this idea by showing the Clifford+CS gate set is universal.

This notion of universality we will use is however not the approximate universality like that of the Clifford+T gate set we demonstrated in Section 10.1.2. Instead it is what we will call **computational universality**. Approximate universality requires that we can approximate *any* unitary and hence quantum circuit arbitrarily well. But such a strong condition is actually not needed for a gate set to be useful. It is sufficient if we can just *simulate* the run of an arbitrary quantum circuit using some runs of a quantum computer using the restricted gate set.

Let's work through an example to make this more clear. For this section we will say that the purpose of a quantum computer is to estimate the expectation value of some observable $\mathcal{O}$. We start with some state $|\psi\rangle$, apply some unitary U to it, and then do measurements and post-process these measurements to get an estimate of $\mathcal{O}$. After many such runs we will get a close approximation of $\mathcal{O}$. Mathematically we can represent the exact expectation value as:

$$\langle\mathcal{O}\rangle \;=\; \langle\psi|\!\vdots\!\boxed{U}\,\boxed{\mathcal{O}}\,\boxed{U^\dagger}\!\vdots\!|\psi\rangle \tag{10.56}$$

However, when we are trying to estimate this observable, we don't have to do this with just a single quantum circuit we run over and over again. Instead we could have a collection of different quantum circuits V_j (potentially acting on a different number of qubits), input states $|\psi_j\rangle$, and observables $\mathcal{O}_j$, such that taking a particular weighted average gets us the outcome we are after:

$$\langle\mathcal{O}\rangle \;=\; \sum_j \lambda_j\, \langle\psi_j|\!\vdots\!\boxed{U_j}\,\boxed{\mathcal{O}_j}\,\boxed{U_j^\dagger}\!\vdots\!|\psi_j\rangle \;=\; \sum_j \lambda_j \langle\mathcal{O}\rangle_j, \tag{10.57}$$

where here we define $\langle\mathcal{O}\rangle_j$ to be the expectation value of $\mathcal{O}_j$ with respect to V_j and $|\psi_j\rangle$. We see then that if we can estimate each of the $\langle\mathcal{O}\rangle_j$, that we can also estimate $\langle\mathcal{O}\rangle$ itself, by just summing our estimates like $\langle\mathcal{O}\rangle = \sum_j \lambda_j \langle\mathcal{O}\rangle_j$.

This might seem like quite a hypothetical situation, so let's give a concrete example. Suppose we have a Clifford+T circuit C applied to the input state $|\psi\rangle$. Then we can transform C into a circuit C' containing just Clifford gates and CS gates using catalysis, so that $C'|\psi\rangle|T\rangle = C|\psi\rangle|T\rangle$. If we were trying

to estimate the observable $\mathcal{O}$ we can then check that:

$$\frac{1}{\sqrt{2}}\left\langle\psi\left|C'\,\mathcal{O}\,(C')^\dagger\right|\psi\right\rangle \overset{(10.45)}{=} \frac{1}{2}\left\langle\psi\left|C\,\mathcal{O}\,(C)^\dagger\right|\psi\right\rangle \tag{10.58}$$

$$\overset{(\boldsymbol{sp})}{=} \left\langle\psi\left|C\,\mathcal{O}\,(C)^\dagger\right|\psi\right\rangle$$

So instead of just running the circuit C, we can run C', which doesn't contain any T gates. This is then an example of Eq. (10.57) where the sum is over just one term and we have $\lambda_1 := 1$, $U_1 := C'$, $|\psi_1\rangle := |\psi\rangle \otimes |T\rangle$ and $\mathcal{O}_1 := \mathcal{O} \otimes I$.

But now suppose we don't even want to use that single $|T\rangle$ we need for the catalysis. What we can do then is decompose this magic state into a sum of Clifford states. Because each term in the sum needs to retain the form of an expectation value like (10.56), we can't just decompose $|T\rangle$ into pure states $|\phi_j\rangle$, instead we need to decompose $|T\rangle\langle T|$ into a sum of $|\phi_j\rangle\langle\phi_j|$ density matrices. One way to do this is the following:

$$|T\rangle\langle T| \;=\; \frac{1}{2}\begin{pmatrix} 1 & e^{i\pi/4} \\ e^{-i\pi/4} & 1 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 & \frac{1+i}{\sqrt{2}} \\ \frac{1-i}{\sqrt{2}} & 1 \end{pmatrix}$$

$$=\; \frac{1}{\sqrt{2}}|+\rangle\langle+| + \frac{1}{\sqrt{2}}|-i\rangle\langle-i| - \frac{\sqrt{2}-1}{2}\left(|0\rangle\langle0| + |1\rangle\langle1|\right). \tag{10.59}$$

Hence, we can decompose $|T\rangle\langle T|$ into four Clifford states $|\phi_1\rangle = |+\rangle$, $|\phi_2\rangle = |-i\rangle$, $|\phi_3\rangle = |0\rangle$ and $|\phi_4\rangle = |1\rangle$ with weights $\lambda_1 = \lambda_2 = \frac{1}{\sqrt{2}}$ and $\lambda_3 = \lambda_4 = -\frac{\sqrt{2}-1}{2}$. Starting with the left-hand side of Eq. (10.58) we then have:

$$\tag{10.60}$$

We see then that this is a case of Eq. (10.57) with $|\psi_j\rangle := |\psi\rangle \otimes |\phi_j\rangle$ and $\mathcal{O}_j := \mathcal{O} \otimes I$ and $U_j = C'$ for all $j \in \{1, 2, 3, 4\}$.

While we can reduce the calculation of an expectation value to the calculation of a sum of (potentially simpler to calculate) expectation values in

this way, there is an important issue here that we have however glossed over. We can only ever *estimate* the expectation value, not get an exact value. Generally, we want to determine an error budget for how close we want the estimate to be, and then that determines how many times we need to sample from the quantum computation. Since we are summing together different expectation values, we need to be careful that we aren't blowing up the error in the estimates. Suppose for instance that some $\lambda_k = 100$. Then a small error in our estimate of $\langle\mathcal{O}\rangle_j$ will blow up by a factor of a 100. On the other hand, if $\lambda_k = 1/100$, then any error will also be decreased by a factor of a 100, so that even a large error is not that important. In you want to be efficient and not over-sample a given expectation value, so that we get its estimate at just the right target precision we then need to sample $\langle\mathcal{O}\rangle_j$ a number of times proportional to $|\lambda_j|$. We can calculate that this summing approach gives a total overhead in the number of samples of $|\lambda_j|$ compared to just determining the desired expectation value $\langle\mathcal{O}\rangle$ with the original circuit.

For instance, in the above example where we decomposed $|T\rangle\langle T|$ into four terms, we have $\sum_j |\lambda_j| = 2\sqrt{2} - 1 \approx 1.83$. Hence, if we decompose the magic state in this way we need to collect 1.83 samples more than we would have needed to if we did use the magic $|T\rangle$ state directly.

Summarising the full procedure we see then that we need to do the following:

1. Start with the Clifford+T computation you want to calculate.
2. Replace all T gates by a CS gate catalysis circuit using a $|T\rangle$.
3. Replace the $|T\rangle$ state needed for all the catalysis by the Clifford states $|\psi_j\rangle$.
4. Run each of the resulting four circuits a number of times proportional to $|\lambda_j|$.
5. Combine the resulting estimates of the observable by scaling by λ_j to get the final outcome.

When we have Clifford gates and CS gates, the gate set is generated by CNOT, Hadamard, S and CS. Of course, CNOT can be constructed using CS and Hadamard, and if we allow states to be prepared into $|0\rangle$ and $|1\rangle$, then we can also prepare an S using a CS. Hence, this gate set is equivalent to just the CS and Hadamard gate. We see then that we have proven the following.

Theorem 10.5.1. The CS+Hadamard gate set is computationally universal. In particular, a Clifford+T computation can be simulated by a

CS+Hadamard computation with a linear overhead in the number of samples, qubits and gates needed.

Remark 10.5.2. Our decomposition (10.59) of $|T\rangle\langle T|$ is a **stabiliser decomposition**, a concept we also looked at in Section* 7.8.1. But as we saw here, the simulation overhead was not based on the number of terms in the decomposition, but rather on the weight $\sum_j |\lambda_j| = 1.83$. This value is known as the **stabiliser extent**, or equivalently, the **robustness of magic** of the decomposition. Without using any catalysis, we could have chosen to write each of the T gates as a magic state injection, and then replace each of the $|T\rangle\langle T|$ states by its stabiliser decomposition. When we do this however, the stabiliser extent scales as 1.83^t where t is the number of T gates, so that the simulation overhead becomes exponential in the number of non-Clifford gates. We expect such an exponential dependence, since replacing all the T gates gives us a Clifford circuit, and we don't expect this gate set to be computationally universal. Note that this however does give us a classical simulation technique: write a Clifford+T circuit as a Clifford circuit where each T gate is replaced by a magic state injection using the $|\phi_j\rangle$ Clifford states, and then efficiently classically simulate each of these Clifford circuits. The cost of this method is then roughly $O(k(n + t)^3 1.83^t)$ where n is the number of qubits, and k is the total number of gates in the circuit.

Remark* 10.5.3. We haven't actually given a formal definition of what 'computational universality' really is. There are multiple ways we could define it that all differ in the details. One particular way we could define it, which we could also call '**BQP**-completeness' is as follows: for a given gate set G define the complexity class $\mathbf{BQP}_G$ as the types of decision problems that can be solved with high probability by a quantum computer just using gates from G. Then G is $\mathbf{BQP}$-complete if $\mathbf{P}^{\mathbf{BQP}_G} = \mathbf{BQP}$. That is, if a classical computer that can query a quantum computer using gates from G can solve the same problems in polynomial time as a universal quantum computer.

Because we can also catalyse T gates using a CCZ gate, we can also prove a version of Proposition 10.5.1 for the Clifford+CCZ gate set, showing that Clifford+CCZ is also computationally universal. In fact, we can restrict the gate set a bit more, as Toffoli+Hadamard is itself already computationally universal. This however requires a different argument then we have been using here, and hence we leave this for the advanced section 10.7.2.

10.5.3   Catalysing completeness

We saw in Theorem 10.3.9 that the (S4) spider nest rule combined with the standard ZX rules we have been using throughout the book is enough to get a complete set of rules for CNOT+T circuits. However, this extended rule set is *not* enough to prove all identities of ZX-diagrams in the $\frac{\pi}{4}$-fragment. For instance, a simple identity that cannot be proven is the following rule known as **supplementarity**:

$$\tag{10.61}$$

Formally showing that this cannot be proven using the ZX+(S4) rules is difficult, but intuitively we can see this because none of the standard ZX rules have any special behaviour for $\frac{\pi}{4}$ phases, while all the spider nest identities only deal with at least 15 $\frac{\pi}{4}$ phases, so there is nothing that says anything about the pair of $\frac{\pi}{4}$ phases here.

Finding a rule set and then proving it is complete is usually a difficult challenge. However, using catalysis it turns out we can make our lives much simpler, and simply extend an existing complete calculus.

Recall that in Chapter 9 we introduced a new type of generator for ZX-diagrams we called H-boxes. In particular we found a number of rewrite rules for phase-free H-boxes in Section 9.2.2, summarised in Figure 9.1. We remarked there that when we restrict to spiders with 0 and π phases, together with phase-free H-boxes, that these rules give us a complete calculus for postselected Toffoli-Hadamard circuits. Such diagrams correspond to a specific type of matrices. Namely, all such matrices are of the form $(\frac{1}{\sqrt{2}})^k M$ where M is an integer matrix. Hence, the matrix entries are from a subset of the ring $\mathbb{Z}[1/\sqrt{2}]$. As these matrices are just integer matrices up to a global scalar of $\sqrt{2}$ we will call this the $\mathbb{Z}$ **fragment**. Note that in particular there are no complex numbers in this calculus. Using catalysis we can make the set of matrices this calculus can represent larger in a 'controlled way' where we can also see which rules we need to add to preserve completeness.

To see how this works, we want to first generalise the T gate catalysis of Eq. (10.45). First, as our goal will just be to produce states, we can plug $|+\rangle$ into the top wire of Eq. (10.45). We can then simplify the expression to a more symmetric form:

$$\tag{10.62}$$

We can then identify the underlying reason this catalysis works. It is because:

$$\tag{10.63}$$

Here we suggestively used Eq. (9.15) to write the phases as H-boxes. We do this because such a rule doesn't just hold for an H-box with a label that is a complex phase like $e^{i\alpha}$, it in fact holds for any complex $a \neq 0$:

$$\tag{10.64}$$

This then allows us to write down a generalisation of Eq. (10.62) to arbitrary H-boxes:

$$\tag{10.65}$$

Here we wrote a^2 in the 2-ary H-box instead of a so that we don't have to work with square roots. When we take $a = e^{i\frac{\pi}{4}}$ we get Eq. (10.62), but this works for any value. A particularly simple, but still interesting case is when $a = e^{i\frac{\pi}{2}} = i$. Translating this back into circuit form gives us a catalysis of $|i\rangle := |0\rangle + i|1\rangle$ states using a CZ. While this might seem trivial (it is after all provable using Clifford rewrite rules) in the context of the phase-free H-boxes it allows us to add complex numbers to the fragment.

Namely, we can just add as a generator $\boxed{i}\!-$, a single-ary H-box with label $a = i$ to the calculus, and this in fact turns out to be sufficient to then represent any matrix $\frac{1}{\sqrt{2}^k}M$ where M now has entries in $\mathbb{Z}[i]$. We will call this the $\mathbb{Z}[i]$ fragment. However, just getting this *universality* was the easy part. How do we know what new equations to add to this calculus to make it complete again? Generally, proving completeness is very difficult, as you first need to search for new equations, and then show that those equations are sufficient to prove all true equalities. However, as it turns out, adding the rule Eq. (10.65) for $a = i$ to the already existing rules for the label-free

fragment is already *almost* enough to get a complete calculus for this bigger fragment $\mathbb{Z}[i]$ which includes $\boxed{i}$— .

To see this, let's first consider what a generic diagram in the $\mathbb{Z}[i]$ fragment looks like. We added the generator $\boxed{i}$— , so now a diagram consists of generators from the old phase-free fragment plus this new generator. These generators could all combine to give us really complicated rewrites, so we want to prevent them from combining. Using Eq. (10.65) we can reduce all these separate instances of $\boxed{i}$— into just one of them, reducing the complexity of the diagram. That is, given some diagram D in the $\mathbb{Z}[i]$ fragment, we can rewrite it to a diagram D' containing just generators from the $\mathbb{Z}$ fragment such that:

$$\vdots\ \boxed{D}\ \vdots \quad \overset{(10.65)}{=} \quad \vdots\ \boxed{D'}\ \vdots \atop \boxed{i} \tag{10.66}$$

Or, as it turns out, this is possible for *most* diagrams in the $\mathbb{Z}[i]$ fragment (Can you see for which ones it doesn't work? If not, don't worry, the authors of this book also originally forgot about this case. As we said: completeness is hard). We will talk about the failing case later, but for now let's assume that our diagram satisfies Eq. (10.66).

As a shorthand, we will write $D'[|\psi\rangle]$ for the diagram we get when we plug $|\psi\rangle$ into the bottom input in Eq. (10.66). So here we have $D = D'[\boxed{i}$— $]$. Note that $\boxed{i}$— $= |0\rangle + i|1\rangle$. Hence, if we expand it like this we see that D is equal to a sum of two diagrams: D' where we plugged in $|0\rangle$ into the bottom wire, and iD' where we plugged $|1\rangle$ into the bottom wire: $D = D'[|0\rangle] + iD'[|1\rangle]$.

Now suppose we have two diagrams D_1 and D_2 in the $\mathbb{Z}[i]$ fragment and that they implement the same linear map: $D_1 = D_2$. We can both decompose them as described above to get $D_1'[|0\rangle] + iD_1'[|1\rangle] = D_2'[|0\rangle] + iD_2'[|1\rangle]$. Each of these $D_j'[|x\rangle]$ diagrams represents a matrix that is entirely real, so the only way for this equation of complex matrices to hold, is if it holds for the real part and for the complex part separately:

$$D_1'[|0\rangle] \;=\; D_2'[|0\rangle] \qquad\qquad D_1'[|1\rangle] \;=\; D_2'[|1\rangle] \tag{10.67}$$

We then conclude that D_1' and D_2' are equal when we input either $|0\rangle$ or $|1\rangle$ into the bottom wire. As these states form a basis, this must then hold for any input. We can then leave this wire open and still have an equality:

$$\vdots\ \boxed{D_1'}\ \vdots \quad = \quad \vdots\ \boxed{D_2'}\ \vdots \tag{10.68}$$

Now we are in business! We have this equality as linear maps, but both

diagrams are in the $\mathbb{Z}$ fragment for which we have completeness. We hence know how to rewrite one into the other. This gives us then a path to rewrite the original D_1 into D_2:

$$\vdots \; D_1 \; \vdots \;\; \overset{(10.65)}{=} \;\; \vdots \; D_1' \; \vdots \;\; \overset{(*)}{=} \;\; \vdots \; D_2' \; \vdots \;\; \overset{(10.65)}{=} \;\; \vdots \; D_2 \; \vdots \qquad (10.69)$$

Here each equality is now a diagrammatic equality, and with (*) we denote we are using rewrites from the original complete calculus for the $\mathbb{Z}$ fragment. We have then very easily proven completeness for this larger fragment, all made possible using a single rule about catalysis.

Well..., we would have proven completeness, *if* it were true we could always rewrite a diagram in the $\mathbb{Z}[i]$ fragment as in Eq. (10.66). We however made a hidden assumption: that there is at least one generator $\boxed{i}$— present in the diagram. When that is the case we can use Eq. (10.65) to reduce all these instances of the generator to just a single copy. But if the diagram didn't contain any $\boxed{i}$— to start with, then this rule does not apply. In fact, we currently have no rewrite rules that relate a diagram containing a $\boxed{i}$— to one that does not contain any $\boxed{i}$— . This means in particular that our current rule set cannot prove the following true equation:

$$\bullet\!\!-\!\!\boxed{i} \;\; = \;\; \bullet\!\!-\!\!\circ \qquad (10.70)$$

However, when we also add Eq. (10.70) as an additional rule, then this problem is solved and it *is* true that we can then always rewrite a diagram in the $\mathbb{Z}[i]$ fragment as in Eq. (10.66): if the diagram contains at least one $\boxed{i}$— we can already use Eq. (10.65) to transform to the form of Eq. (10.66), and if it doesn't we use Eq. (10.70) once to introduce one $\boxed{i}$— , in which case it is also in the form of Eq. (10.66).

Proposition 10.5.4. The $\mathbb{Z}$ fragment of Z- and X-spiders with 0 and π phases and phase-free H-boxes, augmented with the $\boxed{i}$— generator, the catalysis rule Eq. (10.65) for $a = i$, and the rule $\bullet\!\!-\!\!\boxed{i} \; = \; \bullet\!\!-\!\!\circ$ is complete for matrices of the form $\frac{1}{\sqrt{2}^k} M$ where M has entries in $\mathbb{Z}[i]$.

This trick for extending the calculus doesn't just work for i: it works for any complex number $a \neq 0$ such that $a^2 \in \mathbb{Z}$. Let's for example take $a = \sqrt{3}$. We can then do all the steps as before, translating a diagram D containing an arbitrary number of the H-box with label $\sqrt{3}$ into a diagram D' in the $\mathbb{Z}$ fragment which just requires a single input of the $\sqrt{3}$ H-box: $D = D'[|0\rangle + \sqrt{3}|1\rangle] = D'[|0\rangle] + \sqrt{3}D'[|1\rangle]$. If we then have an equality between

two diagrams D_1 and D_2 in the $\mathbb{Z}[\sqrt{3}]$ fragment, we get $D_1'[|0\rangle]+\sqrt{3}D_1'[|1\rangle] = D_2'[|0\rangle] + \sqrt{3}D_2'[|1\rangle]$. Because each of the component diagrams only contains integers, this equation can only hold if the integer part and the $\sqrt{3}$ part hold separately. We hence again get two equalities $D_1'[|0\rangle] = D_2'[|0\rangle]$ and $D_1'[|1\rangle] = D_2'[|1\rangle]$, which allows us to conclude that $D_1' = D_2'$ with the wire left open. We can then use a modified version of Eq. (10.69) to conclude that we have completeness. We also have a modified version of Eq. (10.70) that continues to be true: ○—$\boxed{\sqrt{3}}$ = ○—○ . Adding the catalysis rule and this scalar rule then gives us a complete calculus for the ring $\mathbb{Z}[\sqrt{3}]$.

Although this covers many possible extensions, it does not cover one we care about: extending it with a T gate. This is because taking $a = e^{i\frac{\pi}{4}}$ we see that $a^2 = i \notin \mathbb{Z}$. However, it turns out we can just iterate the catalysis procedure to get larger and larger calculi. Starting now with the calculus for the $\mathbb{Z}[i]$ fragment, we can add the H-box with the label $e^{i\frac{\pi}{4}}$ and add its catalysis rule. When we then go through the motions of the completeness proof again we will end up at the equation $D_1'[|0\rangle]+e^{i\frac{\pi}{4}}D_1'[|1\rangle] = D_2'[|0\rangle] + e^{i\frac{\pi}{4}}D_2'[|1\rangle]$, where now each of the diagrams $D_j'[|x\rangle]$ has entries in $\mathbb{Z}[i]$ instead of $\mathbb{Z}$. Luckily for us, $e^{i\frac{\pi}{4}}$ is still 'independent' of the entries of $\mathbb{Z}[i]$ so that again the only way for this equation to hold is if it holds for each component separately, so that the proof goes through without change. Since $e^{i\frac{\pi}{4}} = (1 + i)/\sqrt{2}$, this calculus can represent arbitrary matrices with entries in the ring $\mathbb{Z}[i, \frac{1}{\sqrt{2}}]$.

Theorem 10.5.5. The $\mathbb{Z}$ fragment of Z- and X-spiders with 0 and π phases and phase-free H-boxes, augmented with H-boxes with a label of i and $e^{i\frac{\pi}{4}}$ and the catalysis rule Eq. (10.65) and scalar rule ○—$\boxed{a}$ = ○—○ for $a = i$ and $a = e^{i\frac{\pi}{4}}$ is complete for matrices with entries in the ring $\mathbb{Z}[i, \frac{1}{\sqrt{2}}]$.

Exercise* 10.11. Two copies of a $e^{i\frac{\pi}{4}}$ H-box can be used to represent an H-box with an i label. So instead of adding the i generator, we could only add the $e^{i\frac{\pi}{4}}$ generator. We then have to modify the catalysis rules: instead of having two separate ones, we need to stack them together into a single one. Find a modified catalysis rule that works in the label-free ZH-calculus augmented with just an H-box with a label of $e^{i\frac{\pi}{4}}$ and find which other rules you need to get completeness.

Exercise* 10.12. Prove supplementarity (10.61) using the phase-free H-box rules and the catalysis rules.

10.6 Summary: What to remember

1. The Clifford+T gate set is approximately universal.
2. In particular, it can exactly represent unitary $2^n \times 2^n$ matrices with entries in the ring $\mathbb{D}[\omega] = \mathbb{Z}[\frac{1}{2}, e^{i\frac{\pi}{4}}]$.
3. We can efficiently approximately synthesise single-qubit unitaries over the Clifford+T gate set. To achieve a precision of ε requires $O(\log 1/\varepsilon)$ number of gates.
4. The scalable ZX notation allows us to represent large collections of parities as a single diagram. This is especially useful in representing large collections of phase gadgets.
5. Certain collections of phase gadgets with phases that are multiples of $\frac{\pi}{4}$ correspond to Clifford unitaries or the identity. We call such collections of phase gadgets *spider nest identities*.
6. A collection of gadgets represents an identity if its corresponding parity matrix is *strongly 3-even*, meaning that the Hamming weight of every column is a multiple of 8, of the product of every pair of columns is a multiple of 4, and the product of every triple of columns is a multiple of 2. The collection of gadgets is a Clifford if it's parity matrix is *3-even*, meaning that the three previous conditions only hold modulo 2.
7. We can instead represent a collection of n-qubit gadgets by its indicator function. If this is a polynomial of degree at most $n-4$, then it is equal to a Clifford.
8. Using this representation we can show that the standard ZX rewrite rules plus one additional rule (S4) suffice for completeness of CNOT+T circuits.
9. Optimising the number of T gates in a CNOT+T circuit is equivalent to decoding a Reed-Muller code, or equivalently to finding a rank decomposition of a symmetric 3-tensor.
10. We can relate gate sets involving the CCZ, CS or T gate together using the framework of *catalysis*, where we can interchange the role of one gate with another using a resource state that we call a *catalyst*. This catalyst is not consumed in the process and hence can be reused.
11. Using catalysis we can find an efficient way to implement small-angle rotations, prove the computational universality of a gate set, and prove completeness by extending other complete rule sets.

10.7 Advanced Material*

10.7.1 Exact and approximate synthesis

10.7.1.1 Exact synthesis of Clifford+T states

In this section we will take another look at the exact synthesis algorithm for Clifford+T unitaries described in Section 10.1.1, but now we will consider multi-qubit unitaries and fill in the number theory details. As we saw in that section, when we understand how to synthesise a *state*, an algorithm for synthesising a unitary follows easily, so let's look at synthesising states first.

So let's suppose we have a normalised vector $|\psi\rangle \in \mathbb{D}[\omega]^{2^n}$. Our task is to find a Clifford+T unitary U such that $U|\psi\rangle = |0\cdots0\rangle$.

Writing $|\psi\rangle = (\psi_1, \cdots, \psi_{2^n})$ we can represent the vector components ψ_i as $\psi_i = a_i\omega^3 + b_i\omega^2 + c_i\omega + d_i$ where $a_i, b_i, c_i, d_i \in \mathbb{D}$. In Lemma 10.1.2 we saw that if these coefficients are integers, that then all the entries except one must be zero and hence $|\psi\rangle$ is a unit vector. For a single-qubit state, the only possible unit vectors are $|0\rangle$ and $|1\rangle$, so if we got $|1\rangle$ we just apply an X gate to get it to be the $|0\rangle$ we want. However, now the state can be any $|\vec{x}\rangle$ up to global phase. We can map this to $|\vec{0}\rangle$ by applying X gates wherever $x_i = 1$. While this is fine if we are synthesising a state, this messes things up when we are synthesising this state as part of a bigger unitary synthesis routine where we care about many columns being sent to the right location. In that case we need to apply the appropriate *2-cycle* classical gate (cf. Section 9.4.3) (decomposed into Clifford+T gates of course), to transform $|\vec{x}\rangle$ into $|\vec{0}\rangle$, or whatever basis state we need it to be. We can also get rid of its phase ω^k by applying the 1-level $T_{[1]}^{-k}$ gate that adds a ω^{-k} phase just to the $|00\cdots0\rangle$ state. Because these 2-level and 1-level operations only change the basis states we are interested in, they do not mess up any of the other columns of the unitary we are synthesising.

So as in the single-qubit case, if all the components in the vector are integers we are essentially done. We then just need to find a strategy to make the vectors be 'closer to being integers', i.e. elements of $\mathbb{Z}[\omega]$. The obvious metric for how far an element in $\mathbb{D}[\omega]$ is from being an element in $\mathbb{Z}[\omega]$ is the smallest power of 2 we have to multiply the element with to get an integer. However, this turns out not to be the best choice. This is because 2 is not a *prime number* in $\mathbb{Z}[\omega]$. The 'magic number' $\delta = 1 + \omega$ we saw in Section 10.1.1 *is* a prime in $\mathbb{Z}[\omega]$.

Definition 10.7.1. Let R be a ring and $a \in R$. We say a is a **unit** if there exists $b \in R$ such that $ab = 1$. We instead say a is **prime** if a is not a unit or 0, and if for any decomposition $a = bc$ with $b, c \in R$ we have that either b or c is a unit.

Example 10.7.2. In $\mathbb{Z}$ the only units are 1 and -1, while in any field, like $\mathbb{C}$, every non-zero element is a unit. The primes of $\mathbb{Z}$ are precisely the prime numbers and their negations (since if a is prime, then multiplying a by any unit gives you another prime). In $\mathbb{Z}[\omega]$ examples of units are ω, because $\omega \omega^7 = 1$, and $1 + \sqrt{2}$, because $(1 + \sqrt{2})(\sqrt{2} - 1) = 1$.

We can prove δ is prime in $\mathbb{Z}[\omega]$ by defining a new kind of norm on $\mathbb{Z}[\omega]$.

Exercise 10.13. On the ring $\mathbb{Z}[\omega]$ we have a norm given by $N_\omega(z) = z\bar{z}$. This norm has some nice properties, namely that it is multiplicative, $N_\omega(z_1 z_2) = N_\omega(z_1) N_\omega(z_2)$, and that it sends elements to positive elements of $\mathbb{Z}[\sqrt{2}]$. We can define a different norm with similar properties on $\mathbb{Z}[\sqrt{2}]$. For $a + b\sqrt{2} \in \mathbb{Z}[\sqrt{2}]$ define the conjugate to be $(a + b\sqrt{2})' = a - b\sqrt{2}$, and then define the new norm by $N_{\sqrt{2}}(z) := zz'$.

a) Show that the conjugate on $\mathbb{Z}[\sqrt{2}]$ is multiplicative: $(z_1 z_2)' = z_1' z_2'$. Use this to show that the norm $N_{\sqrt{2}}$ is multiplicative.
b) For $z \in \mathbb{Z}[\omega]$ define $N(z) := N_{\sqrt{2}}(N_\omega(z)) = (z\bar{z})(z\bar{z})'$. Argue that N is also multiplicative, and that it maps all z to positive integers.
c) Show that z is a unit of $\mathbb{Z}[\omega]$ if and only if $N(z) = 1$. *Hint: For the if direction the definition of the norm already gives you the inverse of z.*
d) Show that z is prime if $N(z)$ is prime in $\mathbb{Z}$.
e) Calculate $N(\delta)$, $N(\sqrt{2})$ and $N(2)$, and conclude that $N(\delta)$ is prime, but the others are not.

So now we know that δ is prime while $\sqrt{2}$ (and hence 2) is not. But it turns out that δ is also a prime factor of $\sqrt{2}$ (and hence 2).

Exercise 10.14. Let $\delta = 1 + \omega$.

a) Write δ^2 and δ^3 as $a + b\omega + c\omega^2 + d\omega^3$ for some integers a, b, c, d.
b) Using the fact that $\omega + \omega^{-1} = \sqrt{2}$, write $\delta^2 \omega^{-1}$ as $x + y\sqrt{2}$ for some integers x and y.
c) Define the unit $\lambda := 1 - \sqrt{2}$. Show that $\delta^2 \omega^{-1} \lambda = \sqrt{2}$.

So we see that $\sqrt{2}$ can be decomposed up to units into two copies of δ, and hence 2 can be decomposed into four copies. Hence, instead of considering the smallest power of 2 we have to multiply a number in $\mathbb{D}[\omega]$ with to get something in our integer ring $\mathbb{Z}[\omega]$, we instead consider the smallest power of δ, as this is a more finegrained metric.

For a $z \in \mathbb{D}[\omega]$, we call the smallest k such that $\delta^k z \in \mathbb{Z}[\omega]$ the **least denominator exponent** (lde) of z. For a vector of values $|\psi\rangle \in \mathbb{D}[\omega]^N$, we call its least denominator exponent the smallest k such that $\delta^k |\psi\rangle \in \mathbb{Z}[\omega]^N$.

Of course if the lde of a vector is 0, then it already consists of elements in $\mathbb{Z}[\omega]$, and we know that such a normalised vector must be very simple. So if we can just find some procedure to iteratively reduce the lde to 0, then we are happy. The goal then is to find, for a given $|\psi\rangle$ with lde k, a set of unitaries $G_1, \ldots, G_l$ such that $|\psi\rangle' = G_l \cdots G_1 v$ has lde at most $k-1$. Then we could just repeat this procedure until we get to denominator exponent 0.

Given a $|\psi\rangle$ with lde k, we can define the vector $|u\rangle := \delta^k |\psi\rangle \in \mathbb{Z}[\omega]^N$. After making some modifications to $|u\rangle$ by applying gates to get a $|u'\rangle \in \mathbb{Z}[\omega]^N$, we are interested in whether this modification has reduced the lde. In order to see when this is the case, we hence need to know when we can divide $|u'\rangle$ by δ, and still get a vector in $\mathbb{Z}[\omega]^N$. Of course, when we start caring about divisibility by some number, we will need to talk about calculating *modulo* this number. So in the same way as we have been talking about *parities*, which are elements of $\mathbb{Z}$ modulo 2, now we are going to work with **residues**, which are elements of $\mathbb{Z}[\omega]$ modulo δ.

For elements $x, y \in \mathbb{Z}[\omega]$ we will write $x \equiv_\delta y$ to denote that $x - y = a\delta$ for some $a \in \mathbb{Z}[\omega]$. For instance, in the exercise above we saw that $\sqrt{2} = (\delta \omega^{-1} \lambda)\delta$, and hence $\sqrt{2} \equiv_\delta 0$. It is not hard to see that $\equiv_\delta$ is an equivalence relation, and that it is preserved by addition and multiplication: if $a \equiv_\delta b$ and $c \equiv_\delta d$, then $a + c \equiv_\delta b + d$ and $ac \equiv_\delta bd$. We then also have $2 \equiv_\delta \sqrt{2}\sqrt{2} \equiv_\delta 0$.

Lemma 10.7.3. For any $z \in \mathbb{Z}[\omega]$ we have $z \equiv_\delta 0$ or $z \equiv_\delta 1$.

Proof We have $\delta \cong_\delta 0$, and as $\delta = 1 + \omega$, we calculate then that

$$\omega \ \equiv_\delta \ \omega + 0 \ \equiv_\delta \ \omega + 2 \ \equiv_\delta \ \delta + 1 \ \equiv_\delta \ 1.$$

Hence for any j we have $\omega^j \equiv_\delta 1$, so that $a + b\omega + c\omega^2 + d\omega^3 \equiv_\delta a + b + c + d$. Since furthermore $2 \equiv_\delta 0$, we see that hence the residue of an element modulo δ is either 0 or 1. $\qquad\square$

Given some $|u\rangle = \delta^k |\psi\rangle \in \mathbb{Z}[\omega]^N$ our goal is to apply operations to $|u\rangle$ to make all the components divisible by δ, and hence have zero residue. The components with residue 1 are then the 'obstacles' we want to get rid of.

Lemma 10.7.4. Let $|\psi\rangle \in \mathbb{D}[\omega]^N$ be a normalised vector with lde $k > 0$, so that $|u\rangle = \delta^k |\psi\rangle \in \mathbb{Z}[\omega]^N$. Then there are at least 2 components u_i and u_j of $|u\rangle$ that have residue 1.

Proof $|u\rangle$ is divisible by δ iff $u_j \equiv_\delta 0$ for all j. But assuming that k was the lde of $|\psi\rangle$, then by definition it won't be divisible, and so there will be at least one u_j with non-zero residue. By normalisation of $|\psi\rangle$ we have $\langle \psi | \psi \rangle = 1$ and hence $\sum_j u_j \overline{u_j} = \langle u | u \rangle = \delta^k \overline{\delta}^k \equiv_\delta 0$. Since residues are either

0 or 1, we then know that there are an even number of cases where $u_j \overline{u_j} \equiv_\delta 1$. The residue is multiplicative, so for these j we must then also have $u_j \equiv_\delta 1$. Hence, if $|u\rangle$ is not divisible by δ there must be at least a pair of elements u_i and u_j that each have non-zero residue. $\qquad\square$

The fact that non-zero residues come in pairs is good, because it turns out we can only reduce the residue of elements of $|u\rangle$ in pairs.

We are working with Clifford+T gates. The CNOT, S and T only contain non-zero elements that are units in $\mathbb{Z}[\omega]$ and have at most one non-zero entry per row, and hence applying these gates does not affect the residues of the state. The only gate then that can affect residues is the Hadamard $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. We see that the Hadamard creates sums $u_i + u_j$ and differences $u_i - u_j$ of elements of the vector, which can lead to lower lde, but then it also divides the elements by $\sqrt{2}$, which can *increase* the lde. Because $\sqrt{2}$ contains two powers of δ, we need to look at the vector elements modulo δ^3 (the next power up), to see if applying a Hadamard will result in lower lde.

Exercise 10.15. We have $\delta^3 = 1 + 3\omega + 3\omega^2 + \omega^3$. Show that any element in $\mathbb{Z}[\omega]$ is equivalent modulo δ^3 to an element in the set

$$\{0, 1, \omega, \omega^2, \omega^3, 1 + \omega, 1 + \omega^2, 1 + \omega^3\}.$$

Hint: First note that $2 \equiv_{\delta^3} 0$, so that we only have to deal with elements $a + b\omega + c\omega^2 + d\omega^3$ where $a, b, c, d \in \{0, 1\}$. Then argue that when $a = b = c = d = 1$, the residue is zero, so that you only have to consider the cases where at most two of a, b, c or d are 1.

Note that from this above exercise we immediately get the following, just by checking all the possible cases:

Lemma 10.7.5. If $z \in \mathbb{Z}[\omega]$ has $z \equiv_\delta 1$, then $z \equiv_{\delta^3} \omega^j$ for some $j \in \{0, 1, 2, 3\}$.

Now we have all the tools we need to solve the problem at hand. For simplicity, let's again first assume we are dealing with a single-qubit vector $|u\rangle = (u_1, u_2) \in \mathbb{Z}[\omega]^2$. If it is not already divisible by δ then we must have $u_1 \equiv_\delta u_2 \equiv_\delta 1$, since the non-zero residues come in pairs. Then by the previous lemma we have $u_1 \equiv_{\delta^3} \omega^l$ and $u_2 \equiv_{\delta^3} \omega^k$. In other words: $u_1 = \omega^l + x\delta^3$ and $u_2 = \omega^k + y\delta^3$ for some $x, y \in \mathbb{Z}[\omega]$. Then we see that if

we apply a T^{l-k} gate to this vector that we get:

$$T^{l-k}|u\rangle \;=\; T^{l-k}\begin{pmatrix}\omega^l + x\delta^3\\ \omega^k + y\delta^3\end{pmatrix} \;=\; \begin{pmatrix}\omega^l + x\delta^3\\ \omega^{k+l-k} + \omega^{l-k}y\delta^3\end{pmatrix} \;=\; \begin{pmatrix}\omega^l + x\delta^3\\ \omega^l + y'\delta^3\end{pmatrix},$$

where we have defined $y' := \omega^{l-k}y$.

Now comes the magic trick: we apply a Hadamard, and we use the fact that $\sqrt{2} = \delta^2\omega^{-1}\lambda$, and hence $\delta^2 = \sqrt{2}\omega\lambda^{-1}$ where λ is the unit from Exercise 10.14:

$$HT^{l-k}|u\rangle \;=\; \frac{1}{\sqrt{2}}\begin{pmatrix}2\omega^l + (x+y')\delta^3\\ (x-y')\delta^3\end{pmatrix} \;=\; \begin{pmatrix}\delta^2\omega^{-1}\lambda\omega^l + (x+y')\omega\lambda^{-1}\delta\\ (x-y')\omega\lambda^{-1}\delta\end{pmatrix}. \tag{10.71}$$

We see now that every term has at least one factor of δ, so we can factor it out:

$$HT^{l-k}|u\rangle \;=\; \delta\begin{pmatrix}\delta\omega^{-1}\lambda + (x+y')\omega\lambda^{-1}\\ (x-y')\omega\lambda^{-1}\end{pmatrix}. \tag{10.72}$$

Success! Because this means that $HT^{l-k}|u\rangle$ is divisible by δ. As $|u\rangle = \delta^k|\psi\rangle$, this means that $HT^{l-k}|\psi\rangle$ now has lde smaller than k. We can now repeat this procedure until we get to lde 0, in which case we know the vector we have is a basis vector, and we are done.

This just covers the single-qubit case, but reducing the lde of a multi-qubit normalised vector $|\psi\rangle \in \mathbb{D}[\omega]^{2^n}$ is done very similarly. Defining $|u\rangle = \delta^k|\psi\rangle$, where k is the lde of $|\psi\rangle$, we saw that there must be an even number of elements of $|u\rangle$ with non-zero residue. We can hence pick a pair (u_i, u_j) that both have non-zero residue, and then apply the above technique, just 'targeting' this pair to zero out their residues. We can do this targetting by replacing the Hadamard and T gates above with the 2-level and 1-level operators $H_{[ij]}$ and $T_{[j]}$ that hence only change the residues of the elements u_i and u_j. The constructions in Section 9.4.3 show how to implement these gates using just Clifford+T gates. We do this reduction of lde with every pair that needs it, until all the residues are zero, in which case the modified $|u\rangle$ is divisible by δ. This then means that the modified $|\psi\rangle$ has lower lde. We then just repeat until the lde is zero and we are left with a basis vector.

Pfew, that was a lot, so let's summarise what we have actually done to get to the solution:

1. We started out with a $2^n \times 2^n$ unitary U where all the entries are in the ring $\mathbb{Z}[\frac{1}{\sqrt{2}}, i]$.
2. Then we realised in Eq. (10.73) that we wanted to find a unitary G that reduces the first column of U to a standard basis vector.

3. Finding such a unitary G is equivalent to finding a way to reduce an arbitrary normalised vector $|\psi\rangle$ to $|0\cdots0\rangle$ using G: $G|\psi\rangle = |0\cdots0\rangle$.

4. Instead of writing $|\psi\rangle$ as a vector over $\mathbb{Z}[\frac{1}{\sqrt{2}}, i]$, we write it as a vector over $\mathbb{D}[\omega]$. We find its least denominator exponent k: the smallest number such that $\delta^k|\psi\rangle \in \mathbb{Z}[\omega]$, where $\delta = 1 + \omega$. We picked δ as the base, since it is prime in $\mathbb{Z}[\omega]$.

5. We look for two components u_i and u_j of $|u\rangle = \delta^k|\psi\rangle$ such that the residues $u_i \equiv_\delta u_j \equiv_\delta 1$. If there is such a pair we apply 2-level Hadamard and 1-level T gates to zero out their residues.

6. If there isn't such a pair of components left, then we have transformed $|u\rangle$ to be divisible by δ, so that the new $|\psi\rangle$ we found must have lower least denominator exponent.

7. We then repeat this procedure until $|\psi\rangle$ has lde 0, in which case it is a standard basis vector up to a phase, which is easily permuted into the desired basis vector, and its phase removed by applying the appropriate 1-level gate.

8. We have now found our desired unitary G_1 that we can apply to U to simplify its first column. Because U is unitary, this means its first row must now also be simplified.

9. Now just rinse and repeat for all the other columns of U, resulting in a series of Clifford+T unitaries $G_1, \ldots, G_{2^n}$ such that $G_{2^n} \cdots G_1 U = \mathrm{id}$.

10. We hence have $U = G_{2^n}^{-1} \cdots G_1^{-1}$.

10.7.1.2 Exact unitary synthesis

Let's fill in the details on how to exactly synthesise an entire unitary and not just a single state. We have a $2^n \times 2^n$ unitary U with matrix entries in $\mathbb{D}[\omega]$. Let $|u_1\rangle$ be the first column of U, and V_1 the Clifford+T unitary satisfying $V_1|u_1\rangle = |\vec{0}\rangle$ that we can find using the procedure described in the previous section. Then we have:

$$
G_1 U \;\; = \;\; \begin{pmatrix} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & U' & \\ 0 & & & \end{pmatrix}.
\tag{10.73}
$$

Note that here the first row also becomes a unit vector, because of the orthogonality conditions between the columns of the unitary $G_1 U$. Now we can take the first column of the smaller unitary U' and synthesise it as a state again. We have to be careful to not undo the work we did with reducing the first column of U, and hence we need to use 2-level and 1-level operators to only touch the elements of the matrix we want to.

Repeating this procedure, we see that we get Clifford+T unitaries $V_1, \ldots, V_{2^n}$ satisfying $V_{2^n} \cdots V_1 U = \text{id}$. Hence, we have synthesised U as $V_1^\dagger \cdots V_{2^n}^\dagger$. All of this is a bit reminiscent of the CNOT synthesis algorithm using Gaussian elimination of Chapter 4. However, there we could encode the entire function of the CNOT circuit into an $n \times n$ matrix, while here we are working with a $2^n \times 2^n$ matrix. Hence, even if each V_i is a small circuit, the overall circuit synthesising U might still be exponentially large. We would not expect to do any better as we are now dealing with an approximately universal gate set, and hence there are simply too many possible unitaries we can synthesise for all circuits implementing them to be small.

Let's record what we have now seen in a Theorem.

Theorem 10.7.6. Let U be an n-qubit unitary with entries in $\mathbb{D}[\omega]$. Then U can be realised by a Clifford+T circuit using at most one zeroed ancilla.

Proof In these sections we have found a method to write U using 2-level Hadamard and X operators and 1-level T operators. As described in Section 9.4.3, these can be built using gates with $n-1$ controls, which require a single zeroed ancilla to be implemented over the Clifford+T gate set (cf. Section 9.4.2). $\qquad\square$

These techniques we have seen for exact synthesis are not unique to Clifford+T. They work for many gate sets that can at least express the 2-level operators necessary to move elements of the vector to the place where they are needed.

One particularly simple example of this is the correspondence between circuits of Toffoli, CNOT, NOT and Z gates, and unitaries over the ring $\mathbb{Z}$. Since all the entries in such a unitary U are integers, the normalisation of the column means that there is at most one non-zero element and that this element is ± 1. Hence, ignoring the possible -1 phases, such a unitary is just a big permutation of the basis vectors, which we know we can realise using a Toffoli circuit V (Section 9.3.2). The resulting unitary VU is then diagonal and only has ± 1 phase. The -1 phases can be realised by 1-level Z operators, and then we are done!

Proposition 10.7.7. Let U be an n-qubit unitary with entries in the ring of integers $\mathbb{Z}$. Then U can be realised by a quantum circuit consisting of Toffoli, CNOT, NOT, and Z gates, using at most one zeroed ancilla.

There are several other results like this that make a correspondence between a certain quantum gate set and the set of matrices over a given ring.

Theorem 10.7.8. Let U be an n-qubit unitary and let R be a ring such

that all the matrix entries of U are in R. Then U can be synthesised as a quantum circuit over the gate set $\mathcal{G}$ using at most one zeroed ancilla when:

- $R = \mathbb{Z}$ and $\mathcal{G} = \{\mathrm{TOF}, \mathrm{CNOT}, X, Z\}$.
- $R = \mathbb{Z}[i]$ and $\mathcal{G} = \{\mathrm{TOF}, \mathrm{CNOT}, X, S\}$.
- $R = \mathbb{Z}[\frac{1}{2}]$ and $\mathcal{G} = \{\mathrm{TOF}, \mathrm{CNOT}, X, H \otimes H\}$.
- $R = \mathbb{Z}[\frac{1}{\sqrt{2}}]$ and $\mathcal{G} = \{\mathrm{TOF}, \mathrm{CNOT}, X, H, CH\}$.
- $R = \mathbb{Z}[\frac{1}{\sqrt{2}}, i] = \mathbb{D}[\omega]$ and $\mathcal{G} = \{\mathrm{CNOT}, H, T\}$.

Hence, we see that the exact synthesis of Clifford+T circuits doesn't exist in a vacuum, but is in fact part of a ladder of increasingly more powerful gate sets corresponding to larger rings.

This ladder can be continued, by replacing the $T = Z(\frac{\pi}{4})$ gate by $Z(\frac{2\pi}{2^k})$ for some $k > 3$. The ring that corresponds to the resulting Clifford+$Z(\frac{2\pi}{2^k})$ gate set is $\mathbb{D}[e^{i\frac{2\pi}{2^k}}]$.

10.7.1.3   Optimality of single-qubit Clifford+T unitary synthesis

Using the exact synthesis algorithm for Clifford+T unitaries generally results in very large circuits. The exception is when we apply it to single-qubit unitaries, for which it is in fact *optimal* in the number of gates needed.

As we already saw in Section 10.7.1.3, for a single-qubit unitary, the Clifford+T exact synthesis takes a particular nice form. We start with some unitary

$$U \;=\; \begin{pmatrix} \psi_1 & \overline{\psi_2}e^{i\alpha} \\ \psi_2 & -\overline{\phi_1}e^{i\alpha} \end{pmatrix}. \tag{10.74}$$

Then we find a unitary V built out of H and T gates that synthesises $|\psi\rangle = (\psi_1, \psi_2)$ so that $V|\psi\rangle = |0\rangle$. Then

$$VU \;=\; \begin{pmatrix} 1 & 0 \\ 0 & e^{i\alpha'} \end{pmatrix} \tag{10.75}$$

for some α' which is a multiple of $\frac{\pi}{4}$. Hence we can further reduce VU to the identity by applying the appropriate power of T. Up to some small constant, synthesising a single qubit unitary hence costs just as much as synthesising a single-qubit state $|\psi\rangle$. The cost of synthesising $|\psi\rangle$ is directly related to its lde k. To reduce it to lde 0 we need to apply for each reduction a Hadamard gate and some power T^m of the T gate where $m = 1, 2$ or 3. If $m \neq 1$ we can see this as applying a T gate and/or an S gate. We hence get a sequence of H, T and S gates, where each application of the Hadamard reduces the lde by at least 1. The total number of gates is hence at most $3k$, and the number

of Hadamard gates and T gates is each at most k. You could wonder whether there is any way we could do better, but it turns out that this number of gates is actually optimal!

Suppose we start from the state $|0\rangle$ and we apply Hadamard, S and T gates to it. Then we end up with some state that we can write as

$$|\psi\rangle \;=\; \frac{1}{\delta^k}\begin{pmatrix} x + y\delta \\ z + w\delta \end{pmatrix}, \tag{10.76}$$

where k is its least denominator exponent. Assuming that $k > 0$ we see that $x + y\delta \equiv_\delta z + w\delta$, as $|\psi\rangle$ is normalised. Hence $x \equiv_\delta z$. Furthermore, we necessarily have $x \equiv_\delta z \equiv_\delta 1$ since otherwise both expressions would be further divisible by δ contradicting k being the lde. Applying an S and T gate to this state doesn't change the denominator exponent, but a Hadamard can change the lde. We calculate then:

$$H|\psi\rangle \;=\; \frac{1}{\delta^k}\frac{1}{\sqrt{2}}\begin{pmatrix} x + z + (y + w)\delta \\ x - z + (y - w)\delta \end{pmatrix}. \tag{10.77}$$

Since $x \equiv_\delta z$, we see then that each of the components of the vector $x + z + (y + w)\delta$ and $x - z + (y - w)\delta$ are divisible by δ. As furthermore $\sqrt{2}$ is δ^2 up to some unit $\omega^{-1}\lambda$, we see then that

$$H|\psi\rangle \;=\; \frac{1}{\delta^k}\frac{1}{\delta^2}\begin{pmatrix} a'\delta \\ b'\delta \end{pmatrix} \;=\; \frac{1}{\delta^{k+1}}\begin{pmatrix} a' \\ b' \end{pmatrix} \tag{10.78}$$

for some numbers a' and b'. Applying a Hadamard can hence only increase the lde by at most 1. So if we got some state $|\psi\rangle$ with lde k, then we know the circuit building it must contain at least k Hadamards. But our synthesis algorithm requires *at most* k Hadamards to synthesise it. Hence the lde is *exactly* equal to the optimal number of Hadamards needed to synthesise it. Or, well, this is almost true, because our argument above only holds if $k > 0$. If we have $k = 0$, so that $|\psi\rangle$ is a unit vector up to some phase, applying a Hadamard increases the lde by 2. So the number of Hadamards is $k - 1$. This is actually what our synthesis algorithm above also finds if we were to analyse it a bit more carefully, because it turns out that there are no normalised vectors that have lde 1, so that in Eq. (10.72) our lde would actually reduce by 2 if we had $k = 2$ to start with.

Exercise 10.16. Prove that there are no normalised vectors that have lde 1.
Hint: This corresponds to showing that there are no vectors $|u\rangle \in \mathbb{Z}[\omega]^N$ for which $\langle u|u\rangle = \delta\bar{\delta} = 2 + \sqrt{2}$. Now use Eq. (10.2) for the component norms

$|u_i|^2$, *and argue that the only possible solution to $\sum_i |u_i|^2 = 2 + \sqrt{2}$ is the one where there is a single non-zero component which is in fact divisible by δ.*

To conclude we hence see that if we have a vector with lde k, then our synthesis algorithm requires an optimal number of $k - 1$ Hadamards to synthesise it, and up to k T gates. This last part is because we can have a T gate after every Hadamard, but we can also have one appear before the first one.

10.7.1.4   *Approximate single-qubit Clifford+T synthesis**

In Section 10.1.2 we hinted at how we can do approximate synthesis of arbitrary single-qubit unitaries using a Clifford+T circuit. In this section we will fill in some more details on how this works, though some parts are a bit too technical even for this advanced section, and so we will just refer to the References for details on how to do that.

So we have some arbitrary single-qubit unitary we want to approximate using Clifford+T gates. First we recall that any single-qubit unitary can be written as $Z(\alpha)X(\beta)Z(\gamma)$ for some phases α, β, γ. Additionally, $X(\beta) = HZ(\beta)H$, so if we know how to synthesise $Z(\alpha)$ gates, then we can synthesise arbitrary single-qubit unitaries.

Let's assume then that our goal is to approximate a $Z(\alpha)$ gate for some arbitrary α using just single-qubit Clifford+T gates. First, it will be useful to work with matrices that have determinant 1, so we write our $Z(\alpha)$ and our approximating unitary U as follows:

$$Z(\alpha) \;=\; \begin{pmatrix} e^{-i\alpha/2} & 0 \\ 0 & e^{i\alpha/2} \end{pmatrix} \quad U \;=\; \frac{1}{\sqrt{2}^k} \begin{pmatrix} u & -\bar{t} \\ t & \bar{u} \end{pmatrix} \qquad (10.79)$$

Here $u, t \in \mathbb{Z}[\omega]$. Note this this means our definition of $Z(\alpha)$ is different from the one we have been using in this book by a global phase. Note also that we wrote our approximation unitary with a factor of $\frac{1}{\sqrt{2}^k}$ in front of it, instead of a power of δ. This turns out to be nicer for this algorithm.

We will fix an error budget $\epsilon \in \mathbb{R}$, and require $\|U - Z(\alpha)\| < \epsilon$. If we found a U with this property, then we know how to synthesise it using the algorithm we described in the previous section, and this synthesis will have an optimal number of Hadamards and T gates, given the lde k. So our goal then is to find a U within ϵ of $Z(\alpha)$, with as low a k as possible. Given that we know how to determine whether a solution exists for a given k, we can find the optimal value of k by just starting low and increasing it until we find a solution. Since k won't be too big (it will be $O(\log 1/\epsilon)$), this will still

be efficient. We can hence assume that k and ϵ are fixed, and then we need to determine whether a solution exists, and if it does, what this solution is. In practice the finding of the solution will also tell us whether there is a solution, so we will focus on that.

Our goal then is to find a U as in Eq. (10.79) for some given k and α, such that $\|U - Z(\alpha)\| < \epsilon$ for some given ϵ. In order to help us do that, we need to have a more concrete description of the norm $\|U - Z(\alpha)\|$ in terms of the matrix elements.

Lemma 10.7.9. Let $u_n = \frac{1}{\sqrt{2}^k}u$ and $t_n = \frac{1}{\sqrt{2}^k}t$ be the normalised versions of u and t, and define $z = e^{-i\alpha/2}$. Then:

$$\|U - Z(\alpha)\|^2 \;=\; |u_n - z|^2 + |t_n|^2$$

Proof Note first that $\|U - Z(\alpha)\| = \|\mathrm{id} - U^\dagger Z(\alpha)\|$ by the unitary invariance of the operator norm. Now, $U^\dagger Z(\alpha)$ is unitary, and hence has two eigenvectors $|\phi_j\rangle$ with eigenvalues $e^{i\alpha_j}$. Because U and $Z(\alpha)$ both have determinant 1 we have $1 = \det(U^\dagger Z(\alpha)) = e^{i\alpha_1}e^{i\alpha_2}$, and hence setting $\alpha := \alpha_1$ we necessarily have $\alpha_2 = -\alpha$. The eigenvectors of $U^\dagger Z(\alpha)$ are also eigenvectors of $\mathrm{id} - U^\dagger Z(\alpha)$ and have eigenvalue $1 - e^{\pm i\alpha}$, so that $\|\mathrm{id} - U^\dagger Z(\alpha)\| = |1 - e^{i\alpha}|$.

Recall that the Hilbert-Schmidt norm of a matrix A is given by $\|A\|_{\mathrm{HS}}^2 := \sum_j \||A|\psi_j\rangle\|^2 = \sum_j \langle\psi_j|A^\dagger A|\psi_j\rangle$ where the $\{|\psi_j\rangle\}$ form an orthonormal basis. It is a standard exercise in linear algebra to show that the Hilbert-Schmidt norm is independent of choice of orthonormal basis. Choosing the eigenbasis of $U^\dagger Z(\alpha)$ we see that $\|\mathrm{id} - U^\dagger Z(\alpha)\|_{\mathrm{HS}}^2 = 2|1 - e^{i\alpha}|^2 = 2\|\mathrm{id} - U^\dagger Z(\alpha)\|^2$, as both eigenvalues have equal magnitude.

Instead picking $|0\rangle$, $|1\rangle$ for our orthonormal basis, we calculate that $\|U - Z(\alpha)\|_{\mathrm{HS}}^2 = 2|u_n - z|^2 + 2|t_n|^2$ by just evaluating the matrices.

Putting these different expressions for the norm together, we calculate:

$$\begin{aligned}
\|U - Z(\alpha)\|^2 &= \|\mathrm{id} - U^\dagger Z(\alpha)\|^2 = \frac{1}{2}\|\mathrm{id} - U^\dagger Z(\alpha)\|_{\mathrm{HS}}^2 \\
&= \frac{1}{2}\|U - Z(\alpha)\|_{\mathrm{HS}}^2 = |u_n - z|^2 + |t_n|^2 \qquad \square
\end{aligned}$$

Now, using the fact that $u_n\overline{u_n} + t_n\overline{t_n} = 1$ and $z\overline{z} = 1$, we can expand $|u_n - z|^2 + |t_n|^2$ further and simplify some more:

$$\begin{aligned}
|u_n - z|^2 + |t_n|^2 &= (u_n - z)\overline{(u_n - z)} + t_n\overline{t_n} \\
&= u_n\overline{u_n} - u_n\overline{z} - z\overline{u_n} + z\overline{z} + t_n\overline{t_n} \\
&= 2 - 2\Re(\overline{z}u_n).
\end{aligned}$$

Here $\Re(\overline{z}u_n)$ denotes the real part of the complex number $\overline{z}u_n$. So if U

is a solution to our approximation problem, then $2 - 2\Re(\overline{z}u_n) \leq \epsilon^2$, or equivalently $\Re(\overline{z}u_n) \geq 1 - \frac{\epsilon^2}{2}$. Interestingly, this does not depend on t_n, but only u_n. Furthermore, if we write $u_n = a + bi$ and $z = x + yi$, then we can interpret them as 2-dimensional real vectors $\vec{u}_n = (a, b)$ and $\vec{z} = (x, y)$, and then $\Re(\overline{z}u_n) = \vec{u}_n \cdot \vec{z}$ is just a dot-product.

We hence want to solve the following problem: given a 2-dimensional real unit-vector $\vec{z}$, find a subnormalised vector $\vec{u}_n = (a, b)$, such that $\vec{u}_n \cdot \vec{z} \geq 1 - \frac{\epsilon^2}{2}$, where $a, b \in \mathbb{Z}[\frac{1}{\sqrt{2}}]$ and we allow some maximal power of $\frac{1}{\sqrt{2}}$ depending on k. We call this a **grid problem**, because we have some target region, the vectors whose inner product with $\vec{z}$ is very close to 1, and a grid of points, given by $\mathbb{Z}[\frac{1}{\sqrt{2}}]^2$, and we want to find a point on this grid that is in the target region. It turns out that this problem is efficiently solvable, although describing in full how to do so is quite lengthy (see the References of this chapter).

So let's suppose that we can solve the grid problem, and hence that we get a $\vec{u}_n$ satisfying $\vec{u}_n \cdot \vec{z} \geq 1 - \frac{\epsilon^2}{2}$. This then gives us the candidate u we are looking for. But we then still need to find a t, so that the matrix U defined as in Eq. (10.79) is in fact unitary. The only equation we need to satisfy for this matrix to be unitary is $|u|^2 + |t|^2 = 2^k$, as this means that the columns of the matrix correspond to normalised vectors. We know the expression for a norm for an element in $\mathbb{Z}[\omega]$, this is namely given by Eq. (10.2): for $z = a\omega^3 + b\omega^2 + c\omega + d$ we have

$$|z|^2 = \overline{z}z = (a^2 + b^2 + c^2 + d^2) + (cd + bc + ab - da)\sqrt{2}.$$

Rewriting the equation $|u|^2 + |t|^2 = 2^k$ to isolate our unknown t we get $|t|^2 = 2^k - |u|^2$. But this right-hand side will now be some $\zeta = x + y\sqrt{2}$ for integers x, y. Writing $t = a\omega^3 + b\omega^2 + c\omega + d$, we can also expand the left-hand side. We then have a part of the equation that results in an integer, and a different part that results in an integer multiple of $\sqrt{2}$. These need to be independently satisfied, so that we can split the equation up into two parts:

$$a^2 + b^2 + c^2 + d^2 = x$$
$$ab + bc + cd - da = y$$

This is a pair of quadratic equations over the integers, and is hence a special case of a **Diophantine equation**. While these are in general hard to solve, in this particular case it turns out that a large proportion of them are fast to solve. So in practice, we can try a bunch of candidates until we find one for which our solution strategy works.

Once the details about solving the grid problem and this Diophantine

equation are filled in, this gives an algorithm that gives an approximation of the phase gate $Z(\alpha)$ using Clifford+T gates that uses an optimal number of Hadamard and T gates. The length of the circuit scales quite well with the desired precision. For instance, we can approximate $Z(\pi/128)$ to within $\epsilon = 10^{-10}$ with a circuit containing 102 T gates.

This algorithm turns out to be both efficient and *optimal*. However, this optimality guarantee only applies when we restrict to unitary ancilla-free circuits. If we allow ancillae, measurements or classical control it is possible to do better by some constant factors. The optimal number in the unitary ancilla-free case scales as $3\log_2(1/\epsilon) + C$ for some small constant C, while the best known protocol when we allow measurements and classical control scales as $\frac{1}{2}\log_2(1/\epsilon) + C'$ (see the References of this chapter).

10.7.2   Computational universality of Toffoli-Hadamard*

In this section we will show that the set of *real-valued* unitaries, i.e. where all matrix entries are real numbers, is computationally universal. Then we will show that in fact the restriction to just Toffoli and Hadamard is already computationally universal.

First, obviously real-valued unitaries are not *approximately* universal as we can never approximate any complex-valued unitary (like the S gate). But as we saw in Section 10.5.2 for a different gate set, it turns out we can 'simulate' complex-valued unitaries using a real-valued one on a larger set of qubits.

For a (complex-valued) n-qubit unitary U, let $\Re(U)$ be the real part of U. That is: $\Re(U)_{ij} = \Re(U_{ij})$. Similarly define the complex part $\Im(U)$. Then $U = \Re(U) + i\Im(U)$. Now define the $(n+1)$-qubit unitary $\tilde{U}$ via

$$\tilde{U}(|0\rangle \otimes |\psi\rangle) := |0\rangle \otimes (\Re(U)|\psi\rangle) + |1\rangle \otimes (\Im(U)|\psi\rangle)$$
$$\tilde{U}(|1\rangle \otimes |\psi\rangle) := -|0\rangle \otimes (\Im(U)|\psi\rangle) + |1\rangle \otimes (\Re(U)|\psi\rangle)$$

While it is clear that $\tilde{U}$ is real-valued, it is not immediately obvious that it is unitary.

Exercise 10.17. In this exercise we will show that $\tilde{U}$ is indeed unitary for any choice of U.

a) Express $\Re(U^\dagger)$ and $\Im(U^\dagger)$ in terms of $\Re(U)$ and $\Im(U)$.
b) Express $\Re(UV)$ and $\Im(UV)$ in terms of $\Re(U)$, $\Re(V)$, $\Im(U)$ and $\Im(V)$.
c) Show that $((\langle 0| \otimes \langle\psi|)\tilde{U}^\dagger)(\tilde{U}(|0\rangle \otimes |\psi'\rangle)) = \langle\psi|\psi'\rangle$. Hint: You will need $\Re(U^\dagger U) = \Re(I) = I$.
d) Show that $((\langle 0|\otimes\langle\psi|)\tilde{U}^\dagger)(\tilde{U}(|1\rangle\otimes|\psi'\rangle)) = 0$. Hint: You will need $\Im(U^\dagger U) = \Im(I) = 0$.

e) Conclude that $\tilde{U}$ is indeed unitary.

The encoding into $\tilde{U}$ is also compositional, meaning we can apply it iteratively to a sequence of unitaries.

Exercise 10.18. Show that $\widetilde{UV} = \tilde{U}\tilde{V}$. *Hint: Use a case distinction on input states $|0\rangle \otimes |\psi\rangle\otimes$ and $|1\rangle \otimes |\psi\rangle$.*

Note that this construction is in fact an example of catalysis, as:

$$(10.80)$$

Exercise 10.19. Prove Eq. (10.80).

We can however not use the argument of Section 10.5.2 to use this to show real-valued unitaries are computationally universal as there is no way to write the $|-S\rangle\langle -S|$ catalyst state in terms of states that can be prepared by real-valued unitaries. We can however use a different argument to prove computational universality.

Proposition 10.7.10. Real-valued unitaries are computationally universal.

Proof Suppose C is an n-qubit circuit built out of unitaries as $C = U_1 \cdots U_k$. We then build the real-valued $(n+1)$-qubit circuit $\tilde{C}$ by $\tilde{C} = \tilde{U}_1 \cdots \tilde{U}_k$. Then note that:

$$(\langle 0| \otimes \langle\psi|)\tilde{C}(|0\rangle \otimes |\psi'\rangle) = \langle\psi|\Re(C)|\psi'\rangle$$
$$(\langle 1| \otimes \langle\psi|)\tilde{C}(|0\rangle \otimes |\psi'\rangle) = \langle\psi|\Im(C)|\psi'\rangle.$$

Hence, if we input $|0\rangle \otimes |\psi'\rangle$ into $\tilde{C}$ and do a measurement marginalising over the first qubit we also get the probabilities:

$$\sum_{x=0,1} |\langle x, \psi|\tilde{C}|0, \psi'\rangle|^2 = |\langle\psi|\Re(C)|\psi'\rangle|^2 + |\langle\psi|\Im(C)|\psi'\rangle|^2.$$

Now, we can also, with some effort, calculate that $|\langle\psi|C|\psi'\rangle|^2 = |\langle\psi|\Re(C)|\psi'\rangle|^2 + |\langle\psi|\Im(C)|\psi'\rangle|^2$. Hence, the probability distribution we get for C is the same as that for $\tilde{C}$ when we prepare the first qubit in the $|0\rangle$ state and ignore its measurement outcome. $\square$

Because we can simulate complex-valued quantum circuits using real-valued unitaries in this direct manner, we don't need *all* the real-valued unitaries. Given some approximately universal gate set G we only need to be able to represent $\tilde{U}$ for $U \in G$.

For instance, let's take the Clifford+T gate set $G = \{T, H, \mathrm{CNOT}\}$. Now,

H and CNOT are already real-valued, and it is easy to see that $\tilde{U} = I \otimes U$ if U is real-valued, so that it remains to see what gates we need for $\tilde{T}$. Calculating its matrix, we see that it is

$$
\tilde{T} \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} .
$$

It is straightforward to verify that this is equal to:

$$
\boxed{\tilde{T}} \quad = \quad \boxed{H}\boxed{Z} \tag{10.81}
$$

We can construct the CZ gate using H and CNOT so that we only additionally need the controlled-Hadamard gate.

Proposition 10.7.11. The gate set $\{\mathrm{CNOT}, \mathrm{CH}\}$ is computationally universal.

Proof We can use an ancilla in the $|1\rangle$ state and a CH gate to create the Hadamard. Using Hadamard and CNOT we can construct the CZ gate. We can hence represent $\tilde{T}$, $\tilde{H} = I \otimes H$ and $\widetilde{\mathrm{CNOT}} = I \otimes \mathrm{CNOT}$. Since $\{T, H, \mathrm{CNOT}\}$ is approximately universal, this means that we can use the $\{X, \mathrm{CNOT}, \mathrm{CH}\}$ gate set to arbitrarily closely simulate any unitary. $\qquad\square$

The controlled-Hadamard is a bit of an arbitrary choice of gate. It turns out that the gate set of Hadamard and Toffoli is also computationally universal. This is a very pleasing result, because the Toffoli gate is universal for reversible classical computing, while the Hadamard gate is the single-qubit Fourier transform. So this in a way shows that the extra power of quantum computing comes from having access to this Fourier transform. The way we prove its computational universality, is by showing that Toffoli-Hadamard 'simulates' the CS-Hadamard gate set, which we know is computationally universal by Proposition 10.5.1.

The real-valued encoding $\widetilde{\mathrm{CS}}$ of the CS gate is equivalent to a Toffoli up to some swaps. With the Hadamard we just get $\tilde{H} = I \otimes H$. Hence, when we encode the CS+Hadamard gate set, we get the Toffoli+Hadamard gate set. We can hence do the following: starting with a Clifford+T computation, we write it as an ensemble of CS+Hadamard circuits. We then encode each of these circuits into a real-valued Toffoli+Hadamard circuit. By doing this we can efficiently simulate the original Clifford+T circuit. We see then that Toffoli+Hadamard circuits are also computationally universal.

Theorem 10.7.12. The Toffoli+Hadamard gate set is computationally universal.

So just using Hadamard gates and Toffoli gates we can simulate any quantum computation to any desired precision.

The computational universality of real-valued quantum computing has an interesting philosophical consequence. One could wonder why quantum mechanics uses complex numbers. Why not just stick to regular old real numbers? The results of this section show that one plausible answer is in any case *not* a solution: we don't need the complex numbers to reach a certain computational complexity, as we could have done the same with real numbers.

10.8 References and further reading

Exact Clifford+T synthesis That unitaries over the ring $\mathbb{D}[\omega]$ can be exactly synthesised over the Clifford+T gate set was proved in Giles and Selinger (2013). The presentation we use here was originally given in Seth Greylyn's Master thesis (Greylyn, 2014).

Approximating unitaries with Clifford+T gates The first paper to give an efficient algorithm for approximating single-qubit phase gates with Clifford+T gates was given in Selinger (2015), which found a T-count complexity of $4 \log_2(1/\epsilon)$. This was improved to the optimal $3 \log_2(1/\epsilon)$ in Ross and Selinger (2016). A good reference for reading about the improvements made to approximate Clifford+T synthesis by incorporating ancillae, measurements and classical control is Kliuchnikov et al. (2023).

Spider nests That a 4-qubit phase gadget can be decomposed into the collection of all phase gadgets with at most 3 legs first appeared in Wang (2018), which hence gives the first appearance of a spider nest identity. The name 'spider nest' was coined in de Beaudrap et al. (2020c) which also described the optimisation technique based on toggling gadgets based on small spider nest identities. The notion of a strongly 3-even matrix was introduced in Bravyi and Haah (2012). The relation between strongly 3-even matrices and T-count optimisation seems to have been folklore for a number of years as the strongly 3-even matrices were mostly used in the context of quantum error correcting codes to understand when a code has a transversal T gate (we will have a lot more to say about all that in the next chapter). The relation between strongly 3-even matrices and spider-nest identities was formally spelled out

in Kissinger and van de Wetering (2024). In Amy and Mosca (2019) it was shown that phase gadgets with dyadic angles are the only gadgets that allow non-trivial identities. If we for instance have gadgets with phases that are multiples of $\frac{\pi}{3}$, then the only way they can cancel out is if they do so in the trivial way where the gadgets fuse together. See also van de Wetering et al. (2024) where they show that in certain settings the only optimisation we can do to 'black-box' non-Clifford phases is to fuse them.

T-count optimisation The idea to use small spider nests to optimise *T*-count was introduced in de Beaudrap et al. (2020c) and later improved upon in followup work by the same authors (de Beaudrap et al., 2020a). The relationship between T-count optimisation and Reed-Muller codes was established in Amy and Mosca (2019) and the relation to symmetric 3-tensor factorisation in Heyfron and Campbell (2018). It is the representation as a symmetric 3-tensor that is currently the leading approach in T-count optimisation, with the current best methods being Ruiz et al. (2024); Vandaele (2024). The NP-hardness of T-count optimisation was established in van de Wetering and Amy (2024).

Scalable ZX The scalable ZX-calculus was formally introduced in Carette et al. (2019), though many of the new features appeared in an informal way in an earlier preprint Chancellor et al. (2016). The CNOT+*T* completeness in scalable ZX of the Clifford rules plus the (S4) rule appeared in Kissinger and van de Wetering (2024). Note that Clifford+(S4) is not complete for the full fragment of ZX-diagrams where the phases are multiples of $\frac{\pi}{4}$. This is because (S4) preserves the same invariant that was used in Perdrix and Wang (2016) to argue that the Clifford rules are incomplete for the universal fragment of ZX-diagrams. In Jeandel et al. (2018) a complete rule set for ZX-diagrams with $\frac{\pi}{4}$ phases was given.

Catalysis The CCZ to 2 $|T\rangle$ catalysis was first described in Gidney and Fowler (2019), while the *T*-CS catalysis seems to be more of a folklore result. The idea of using catalysis to relate different gate sets was introduced in Amy et al. (2023). The synthesis of small phase gates using an adder and measurement-based uncomputation was described in Gidney (2018). The relation between catalysis, computational universality and completeness was studied by the authors of this book in Kissinger et al. (2024). This paper serves as the basis for Section 10.5.

Toffoli-Hadamard universality The original proof that Toffoli+Hadamard is computational universal is given in Shi (2002) (this also seems to be the paper that coined the term 'computational universality'). This actually shows this property for a wider set of gates. Namely, if you have a Toffoli gate and any non-computational-basis-preserving gate, then this will lie dense in the group of special orthogonal matrices. A simpler proof of Toffoli+H universality, which is what we use, is given in Aharonov (2003), which also used the term 'computational universality'. They show this easily by establishing that the gate set consisting of the controlled-S gate and the Hadamard gate maps to the Toffoli+Hadamard gate set. The approximate universality of the CS+H gate set was shown in (Kitaev, 1997, Lemma 4.6 on p. 1213). Kitaev proves this using a 'geometric lemma' that adding a gate to a set of gates that stabilises a given state creates a larger-dimensional space of gates. This proof is not constructive. He then proves his Solovay-Kitaev algorithm to show how you would do it constructively (that proof only requires that the set of gates does in fact lie dense in the group). Note that what he calls S is the Hadamard gate, and K is the S gate. In this book we instead show that CS+H is 'just' computationally universal, as this follows much more directly using catalysis. This proof originally appeared in Kissinger et al. (2024).

11

Quantum error correction

Classical error correction is ubiquitous in modern computing. The basic idea is to pad out a message we want to send with some extra redundant data, which give us enough info to recover the original message as long as no (catastrophically bad) errors occurred. In the 1990s, people started to notice that many of the same concepts could be applied to quantum data. Namely, by encoding some qubits into a higher-dimensional system and making careful choices of measurements, we can often detect whether errors occurred and even correct them (again if they are not too bad).

One of the key ideas behind quantum error correction is identifying analogous notions between classical, linear error correction and stabiliser theory. We have already seen in previous chapters that there are several deep connections between stabiliser/Clifford states and maps and $\mathbb{F}_2$-linear structures, which are the bread and butter of classical error correction. In this chapter, we will start to cash out these connections and show how quantum error correction can be done in a way that is, in many ways, similar to its classical counterpart.

For example, if I want to send you three bits, (x, y, z) and actually I send you six bits $(x, y, z, x \oplus y, y \oplus z, x \oplus z)$, then I can always recover the original message even if one of these six bits gets flipped. In this simple example, we encode our message (an element of $\mathbb{F}_2^3$) into a 3D subspace $S = \{(x, y, z, x \oplus y, y \oplus z, x \oplus z) \,|\, x, y, z, \in \mathbb{F}_2\}$ of $\mathbb{F}_2^6$ called the **codespace**. If a single error occurs in such an encoded bitstring, it leaves the codespace, so we can detect it just by checking if the last three bits are indeed the correct parities of the first three. Furthermore, there is at most 1 element in the codespace that is within 1 bit flip of any string of six bits, so we can actually *correct* a single error. Hence, a major aspect of designing good error correcting codes is finding particular subspaces of $\mathbb{F}_2^n$ that have nice properties like this, as we'll discuss in Section 11.1.

Quantum error correcting codes can be defined as subspaces of $(\mathbb{C}^2)^{\otimes n}$, the space of n qubits, which have analogous nice properties. Unlike the classical case, these are going to be huge $\mathbb{C}$-linear subspaces, namely 2^k-dimensional subspaces of the 2^n-dimensional space $(\mathbb{C}^2)^{\otimes n}$. As a result, it is not practical to work with these spaces explicitly (e.g by enumerating a spanning set of vectors). However, as we have already seen in Chapter 6, stabiliser theory gives us a powerful tool for efficiently representing complicated subspaces of $(\mathbb{C}^2)^{\otimes n}$. Thanks to the **Fundamental Theorem of Stabiliser Theory**, we can fix a 2^k-dimensional stabiliser subspace of $(\mathbb{C}^2)^{\otimes n}$ by giving $n - k$ indepedent generators of the stabiliser group.

11.1   Classical codes and parameters

Let's look again at the codespace

$$S := \{(x, y, z, x \oplus y, y \oplus z, x \oplus z) \,|\, x, y, z, \in \mathbb{F}_2\} \subseteq \mathbb{F}_2^6 \qquad (11.1)$$

We claimed before that S can correct a single error. That is, if I start with a vector in S and flip 1 bit, this vector will no longer be in S, and furthermore, there is a unique vector in S that is one bit-flip way. For example, if I receive the string: $(1, 1, 0, 0, 0, 0)$, I can tell there is an error somewhere, since $x = y = 1$ and $z = 0$, but $x \oplus z = y \oplus z = 0$. It could be that either my first 3 data bits are wrong, or it could be that the parity bits are wrong. However, there is only one way to satisfy all the parities by flipping a *single* bit, namely setting $z := 1$. Similarly, if I look at a string $(0, 0, 1, 0, 1, 0)$, the only possibility for a single-bit error is that the final parity bit $x \oplus z = 0$ is wrong.

Since S is a linear subspace, it turns out we can succinctly capture the property of being able to detect any of some family of errors (e.g. single bit-flips) and to correct those errors in a single concept called code distance.

Definition 11.1.1. For a linear subspace $S \subseteq \mathbb{F}_2^n$, the **code distance** of S is the smallest non-zero Hamming weight of a vector in S.

Suppose we could flip a single bit of a vector $\vec{v} \in S$ and obtain another vector $\vec{w} \in S$. Then, their sum $\vec{v} \oplus \vec{w}$ is also in S, and it would just be a unit vector consisting of a single 1 in the location of the flipped bit, e.g.

$$(1, 0, 1, 1) \oplus (1, 1, 1, 1) = (0, 1, 0, 0)$$

The same applies if we flip d bits: the sum will have 1's in precisely the locations of the flipped bits. So, saying that flipping d bits leaves S is the same thing as saying there is no vector in S with Hamming weight d. In

other words, a space S with code distance d can detect any amount of errors strictly less than d.

It is also the case that being able to *correct* a given number of errors can be stated in terms of the code distance. We noted before that the code S in (11.1) has the property that, if we take any vector $\vec{v} \in S$ and flip a single bit, there is no other vector $\vec{w} \in S$ that is a single bit-flip away. In other words, there are no two vectors $\vec{v}, \vec{w} \in S$ that are two bit-flips apart, or equivalently, S has a code distance of (at least) 3. In fact, the code distance is exactly 3, which we can see by enumerating all 8 vectors in this 3D subspace:

$$
S = \left\{
\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},
\begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix},
\begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix},
\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix},
\begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix},
\begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix},
\begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix},
\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}
\right\}
$$

and noting that all the non-zero vectors have Hamming weight 3 or more. This is a general feature: any code with distance $d = 2e + 1$ can always correct an error consisting of at most e bit flips.

To move toward the quantum analogue of classical linear codes, we can proceed by thinking of stabiliser measurements as analogous to classical parity checks. As we already mentioned, for classical codes, we can very quickly check whether $\vec{v} \in S$ by taking XORs of certain bits. For the code S in (11.1), then $(v_1, v_2, v_3, v_4, v_5, v_6)$ is in S precisely when the following linear equations are satisfied:

$$
\begin{aligned}
v_1 \oplus v_2 \oplus v_4 &= 0 \\
v_2 \oplus v_3 \oplus v_5 &= 0 \\
v_1 \oplus v_3 \oplus v_6 &= 0
\end{aligned}
$$

As we saw in Chapter 4, given a system of homogeneous $\mathbb{F}_2$-linear equations fixing a linear subspace S essentially the same as giving a spanning set for $S^\perp$. In this case, this is:

$$
S^\perp = \mathrm{Span} \left\{
\begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix},
\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix},
\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}
\right\}
$$

To check $\vec{v} \in S$, it suffices to check $\vec{w}_i^T \vec{v} = 0$ for each of the basis vectors

$\vec{w}_i$ of $S^\perp$. If this holds, then we can be sure no error of weight $< d$ has occurred. If $\vec{w}_i^T \vec{v} = 1$, this is called a **syndrome**, because it indicates that there is an error somewhere in $\vec{v}$.

11.2 Quantum stabiliser codes

In this section, we'll expand on the analogy between parity checks and stabiliser measurements, and show how to define the notion of code distance for a stabiliser subspace. Fix a stabiliser group $\mathcal{S} = \langle \vec{P}_1, \ldots, \vec{P}_m \rangle$ with associated stabiliser subspace $S \subseteq (\mathbb{C}^2)^{\otimes n}$. Since we can use stabiliser subspaces to correct errors, we will from henceforth use the terms stabiliser subspace and **stabiliser code** interchangeably. This is analogous to the classical case, where we referred to subspaces of $\mathbb{F}_2^n$ as codes.

Thanks to the Fundamental Theorem of Stabiliser Theory from Chapter 6, we know that S is 2^k-dimensional, for $k := n - m$. Hence, we can think of the stabiliser subspace as encoding k **logical qubits** in n physical qubits. This gives us quantum analogues to the parameters n and k from classical codes. To build up to the notion of code distance, let's have a look at how we can use $\mathcal{S}$ to detect and correct errors.

We can treat the generators of $\mathcal{S}$ as "quantum parity checks" in a very specific way. If we measure an arbitrary state $|\psi\rangle \in (\mathbb{C}^2)^{\otimes n}$ with one of stabiliser generators, we can compute the Born rule probability as:

$$\text{Prob}(s_j \,|\, |\psi\rangle) \;=\; \langle\psi|\Pi^{(s_j)}_{\vec{P}_j}|\psi\rangle \;=:\; $$

If we restrict to the case where $|\psi\rangle \in S$, then $\vec{P}_j|\psi\rangle = |\psi\rangle$, then by Proposition 6.2.2 we know $\Pi^{(0)}_{\vec{P}_j}|\psi\rangle = |\psi\rangle$. Hence:

$$\text{Prob}(0 \,|\, |\psi\rangle) \;=\; $$

$$= \; 1$$

So, we will always get outcome $s_j = 0$ if no error occurred on $|\psi\rangle$. From this,

we can also conclude that we'll never get outcome $s_j = 1$:

$$\text{Prob}(1 \,|\, |\psi\rangle) \;=\; \left\langle\, \psi \;\vdots\; \vec{P} \;\vdots\; \psi \,\right\rangle \;=\; 0$$

Now, suppose an error *does* occur on $|\psi\rangle$. For simplicity, we'll initially assume that this error takes the form of a self-adjoint Pauli string $\vec{Q}$. In Section 11.2.2, we'll show that arbitrary errors can be reduced to this case. Namely, if we can detect/correct Pauli errors, then we can detect/correct arbitrary errors.

There are two possibilities, either the error commutes with $\vec{P}$ or it anti-commutes with $\vec{P}$. These two possibilities translate into the following commutation rule with respect to the Pauli box $\vec{P}$.

Exercise 11.1. Show that:

$$\vdots \; \vec{P} \;\vdots\; \vec{Q} \;\vdots \;\;=\;\; \vdots \; \vec{Q} \;\vdots\; \vec{P} \;\vdots$$

where $k = 0$ if $\vec{P}\vec{Q} = \vec{Q}\vec{P}$ and $k = 1$ if $\vec{P}\vec{Q} = -\vec{Q}\vec{P}$.

As a consequence of Exercise 11.1, if a commuting error $\vec{Q}$ happens to $|\psi\rangle$, the outcome of measuring $\vec{P}_j$ is unaffected:

$$\left\langle\, \psi \;\vdots\; \vec{Q} \;\vdots\; \vec{P} \;\vdots\; \vec{Q} \;\vdots\; \psi \,\right\rangle \;=\; \left\langle\, \psi \;\vdots\; \vec{P} \;\vdots\; \vec{Q} \;\vdots\; \vec{Q} \;\vdots\; \psi \,\right\rangle \;=\; \left\langle\, \psi \;\vdots\; \vec{P} \;\vdots\; \psi \,\right\rangle$$

So, $\text{Prob}(s_j \,|\, \vec{Q}|\psi\rangle) = \text{Prob}(s_j \,|\, \vec{Q}|\psi\rangle)$. Hence, we'll again get outcome $s_j = 0$ with certainty. However, if $\vec{Q}$ anti-commutes with $\vec{P}_j$, it kicks a π up into the measurement outcome, so the probabilities flip:

$$\left\langle\, \psi \;\vdots\; \vec{Q} \;\vdots\; \vec{P} \;\vdots\; \vec{Q} \;\vdots\; \psi \,\right\rangle \;=\; \left\langle\, \psi \;\vdots\; \vec{P} \;\vdots\; \vec{Q} \;\vdots\; \vec{Q} \;\vdots\; \psi \,\right\rangle \;=\; \left\langle\, \psi \;\vdots\; \vec{P} \;\vdots\; \psi \,\right\rangle$$

Hence, we'll get outcome $s_j = 1$ with certainty.

We can measure all m stabilisers generating $\mathcal{S}$. Since all of the stabilisers commute, we can either think of doing each of these measurements in sequence (where the order doesn't matter), or equivalently, doing one big measurement whose outcome is given by:

$$\text{Prob}(s_1, \ldots, s_m \,|\, |\psi\rangle) \;=\; \left\langle \psi \;\vdots\; \boxed{\vec{P}_1} \;\vdots\; \boxed{\vec{P}_2} \;\cdots\; \boxed{\vec{P}_m} \;\vdots\; \psi \right\rangle$$

If $|\psi\rangle$ is of the form $\vec{Q}|\phi\rangle$ for some state $|\phi\rangle$ in the stabiliser subspace and some (possibly trivial) Pauli error $\vec{Q}$, this will yield some particular bitstring of outcomes with certainty. If $\vec{Q} = I$, the outcome will always be $(s_1, \ldots, s_m) = (0, \ldots, 0)$. Hence, each time $s_j = 1$, this indicates that (1) some non-trivial error $\vec{Q}$ has occurred, and (2) that error anti-commutes with the j-th stabiliser $\vec{P}_j$. By analogy to the classical case, this vector of measurement outcomes is called the **syndrome**.

If we are lucky, this syndrome will provide us with enough information to send $|\psi\rangle$ back to the error free state $|\phi\rangle$. For example, if it uniquely fixes the Pauli string $\vec{Q}$, that is definitely good enough, because we can then just apply $\vec{Q}$ again to get $\vec{Q}|\psi\rangle = \vec{Q}^2|\phi\rangle = |\phi\rangle$. However, we don't even need to hit $\vec{Q}$ on-the-nose, because if we correct $|\psi\rangle$ with some $\vec{Q}'$ that is the product of $\vec{Q}$ and some stabiliser $\vec{S} \in \mathcal{S}$, we get:

$$\vec{Q}'|\psi\rangle = \vec{Q}\vec{S}\vec{Q}|\phi\rangle \propto \vec{Q}^2\vec{S}|\phi\rangle = \vec{S}|\phi\rangle = |\phi\rangle$$

So, we again recover the state $|\phi\rangle$, possibly up to an irrelevant global phase.

Lets see how this works by means of a simple example. Consider the following stabiliser group on three qubits:

$$\mathcal{S} = \langle \vec{Z}_1 = Z \otimes Z \otimes I, \vec{Z}_2 = I \otimes Z \otimes Z \rangle$$

$\mathcal{S}$ has two independent stabilisers, so $\dim(\text{Stab}(\mathcal{S})) = 2^{3-2} = 2$. Hence, this code, called the **GHZ code** encodes one qubit into three. Fixing a basis for $\text{Stab}(\mathcal{S})$, we have

$$\text{Stab}(\mathcal{S}) = \text{Span}\{|000\rangle, |111\rangle\}$$

For the following discussion, it will be convenient to write $\text{Stab}(\mathcal{S})$ as the image of a map from one qubit into three qubits, called the **encoder** of $\mathcal{S}$. We will say more about encoders in the following sections, including how to

build them from stabiliser codes. But for now, note that:

$$\text{---}\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\text{\<\!\!\!} \quad :: \quad |i\rangle \mapsto |iii\rangle \tag{11.2}$$

It thus follows that:

$$\text{Stab}(\mathcal{S}) \;=\; \text{Im}\left(\text{---}\!\!\!\!\!\!\!\!\!\!\text{\<\!\!\!} \right)$$

Hence, we can write a generic state $|\Psi\rangle \in \text{Stab}(\mathcal{S})$ as:

As before, we can look at the Born rule probabilities associated with measuring the stabiliser generators $\vec{Z}_1$ and $\vec{Z}_2$. As was shown in Exercise 6.7, all-Z and all-X Pauli projectors take a particularly simple form:

$$\Pi^{(k)}_{Z...Z} \;=\; \qquad\qquad\qquad \Pi^{(k)}_{X...X} \;=\;$$

This generalises to Pauli strings formed just from Z and I (or just X and I) by connecting the projector to just the subset of qubits were a Z (or X) appears. For example, we have:

$$\Pi^{(k)}_{ZZI} \;=\; \qquad\qquad\qquad \Pi^{(k)}_{IZZ} \;=\;$$

Applying $\Pi^{(k)}_{ZZI}$ to an encoded state:

$$\qquad = \qquad = \qquad = \delta_{0,k}$$

we see that, as expected:

$$\text{Prob}_{ZZI}\left(k \;\Big|\; \text{\<\!\!\!}\psi\text{---}\!\!\!\!\!\text{\<\!\!\!} \right) = \delta_{0,k}$$

By symmetry, we see the same probabilities for measuring $I \otimes Z \otimes Z$. This is consistent with what we saw before: measuring states in $\text{Stab}(\mathcal{S})$ with the generators of $\mathcal{S}$ will leave the states unchanged, and yield outcome 0 with probability 1.

However, suppose we introduce an error to our encoded state, such as a 'bit

flip', i.e. a Pauli X applied to the first qubit. Since $X \otimes I \otimes I$ anti-commutes we $Z \otimes Z \otimes I$, we get:

Hence the Born rule probabilities are flipped:

so we obtain $s_1 = 1$ for out first syndrome bit. On the other hand, if we measure $I \otimes Z \otimes Z$ instead, this commutes with $X \otimes I \otimes I$, so:

so we obtain outcome $s_2 = 0$ for the second syndrome bit. By symmetry, it is easy to check that each of the single bit-flip errors can be associated with a unique syndrome:

$$
\begin{array}{lcl}
X \otimes I \otimes I & \leftrightarrow & (s_1, s_2) = (1, 0) \\
I \otimes X \otimes I & \leftrightarrow & (s_1, s_2) = (1, 1) \\
X \otimes I \otimes I & \leftrightarrow & (s_1, s_2) = (0, 1)
\end{array}
$$

Hence, if any single bit-flip error occurs, we always know how to fix it. We just need to measure the stabiliser generators and apply a Pauli X on the correct qubit to cancel out the error.

Unfortunately, the GHZ code is not good enough to correct arbitrary single qubit errors. Notably, if a Pauli Z error occurs on one of the qubits (which is sometimes called a 'phase-flip' error), it commutes with all of the generators of $\mathcal{S}$. Hence, the syndrome will always be $(0, 0)$, and we'll have no idea that the error even happened.

In this sense, the GHZ code is not very good at detecting or correcting errors. By analogy to the classical case, this a distance-one code, meaning that there are even single-qubit errors that can go undetected. In order to understand what this means (and ultimately to find codes which *can* detect and correct errors), we will now formalise the notion of code distance for quantum stabiliser codes.

11.2.1 Code distance for stabiliser codes

We saw in Section 11.1 that we could quantify the number of detectable errors using the classical code distance. We can do a similar thing for stabiliser codes, where the following notion plays the role of the Hamming weight.

Definition 11.2.1. The **weight** $|\vec{P}|$ of a Pauli string $\vec{P}$ is the number of non-identity Pauli operators that appear in $\vec{P}$.

For example, the Pauli string $X \otimes Z \otimes I$ has weight 2 and the identity string always has weight 0.

Exercise 11.2. Show that we can bound the weight of a product of two Pauli strings as follows:

$$|\vec{P}\vec{Q}| \leq |\vec{P}| + |\vec{Q}|$$

Using this notion, we can quantify the number of detectable errors. Note that an error produces a non-zero syndrome if and only if it anti-commutes with at least one of the generators of the stabiliser group. So, to be undetectable, it must commute with everything in $\mathcal{S}$. Furthermore, we only care about errors that actually mess up the states in our stabiliser subspace, so we shouldn't count elements of the stabiliser group itself as errors.

Definition 11.2.2. For a stabiliser group $\mathcal{S}$, we define the **distance** of the associated stabiliser code as the minimum weight $d := |\vec{P}|$ of a Pauli string $\vec{P} \notin \mathcal{S}$ that commutes with every element in $\mathcal{S}$.

Note that, since $\mathcal{S}$ is a subgroup, $I \in \mathcal{S}$, so $\vec{P}$ must have non-zero weight. Also, note that $\vec{P}$ commutes with everything in $\mathcal{S}$ if and only if it commutes with each of the generators, so we can efficiently decide whether $\vec{P}$ commutes with everything in $\mathcal{S}$. However, like in the classical case, we cannot in general compute the code distance efficiently, since we need to show that *all* Pauli strings with $|\vec{P}| < d$ are either in $\mathcal{S}$ or anti-commute with something in $\mathcal{S}$.

If a stabiliser code has distance d, it can detect any error with $|\vec{Q}| < d$. In other words, it can detect up to $d - 1$ single-qubit errors. Similarly to the classical case, it can also correct up to $\lfloor (d - 1)/2 \rfloor$ errors. More precisely, errors $\vec{Q}$ with $|\vec{Q}| \leq \lfloor (d - 1)/2 \rfloor$ are uniquely fixed, up to stabilisers.

Theorem 11.2.3. Suppose $\mathcal{S} = \langle \vec{P}_1, \ldots, \vec{P}_m \rangle$ defines a stabiliser code with distance d. Then if two errors $\vec{Q}_1, \vec{Q}_2$ with $|\vec{Q}_i| \leq \lfloor (d-1)/2 \rfloor$ yield the same syndrome $(s_1, \ldots, s_m)$, then $\vec{Q}_1 = \vec{S}\vec{Q}_2$ for some $\vec{S} \in \mathcal{S}$.

Proof If $s_j = 0$, then $\vec{Q}_1$ and $\vec{Q}_2$ both commute with the j-th stabiliser generator $\vec{P}_j$, whereas if $s_j = 1$, then they both anti-commute with $\vec{P}_j$. In

either case, the product $\vec{Q}_1 \vec{Q}_2$ commutes with all of the generators of $\mathcal{S}$. By Exercise 11.2, we know $|\vec{Q}_1 \vec{Q}_2| \leq |\vec{Q}_1| + |\vec{Q}_2| \leq 2\lfloor (d-1)/2 \rfloor \leq d-1$. Because $\mathcal{S}$ has distance d, $\vec{Q}_1 \vec{Q}_2$ equals some $\vec{S} \in \mathcal{S}$. Hence $\vec{S}\vec{Q}_2 = \vec{Q}_1 \vec{Q}_2^2 = \vec{Q}_1$. $\quad\square$

So, as in the classical case, a quantum code can correct e errors if $2e+1 \leq d$. Putting the three parameters together, we will write $[[n, k, d]]$ to indicate a quantum error correcting code that encodes k logical qubits in n physical qubits, with code distance d.

Example 11.2.4. The following 7-qubit code is called the *Steane code*:

$$\vec{X}_1 := X \otimes I \otimes I \otimes I \otimes X \otimes X \otimes X$$
$$\vec{X}_2 := I \otimes X \otimes I \otimes X \otimes I \otimes X \otimes X$$
$$\vec{X}_3 := I \otimes I \otimes X \otimes X \otimes X \otimes I \otimes X$$
$$\vec{Z}_1 := Z \otimes I \otimes I \otimes I \otimes Z \otimes Z \otimes Z$$
$$\vec{Z}_2 := I \otimes Z \otimes I \otimes Z \otimes I \otimes Z \otimes Z$$
$$\vec{Z}_3 := I \otimes I \otimes Z \otimes Z \otimes Z \otimes I \otimes Z$$

Since it consists of 7 physical qubits and 6 stabiliser generators, it encodes $7 - 6 = 1$ logical qubit. Any Pauli consisting of a Z on one or two qubits anti-commutes with at least one of $\vec{X}_i$ generators, whereas any Pauli consisting of X on one or two qubits anti-commutes with at least one of the $\vec{Z}_i$ generators. Any Pauli string $\vec{Q}$ of weight ≤ 2 can be written as a product of an all-X string of weight ≤ 2 and an all-Z string of weight ≤ 2. Hence $\vec{Q}$ must anti-commute with some generator. On the other hand, $I \otimes I \otimes I \otimes Z \otimes Z \otimes Z \otimes I$ commutes with all of the generators above and can't be obtained as a product of them. Hence, this code has a distance of 3, i.e. it is a $[[7, 1, 3]]$ code.

If we don't care about error correction and just want to be able to detect an arbitrary weight-1 error, the following code will work.

Example 11.2.5. The following is a $[\![4, 2, 2]\!]$ error detecting code:

$$\vec{X}_1 := X \otimes X \otimes X \otimes X$$
$$\vec{Z}_1 := Z \otimes Z \otimes Z \otimes Z$$

Clearly any weight-1 Pauli will anti-commute with one of the two stabilisers above, hence it detects a single error. Furthermore, since it has only 2 stabilisers for 4 physical qubits, it encodes 2 logical qubits.

In the last two examples, the codes were split into all-X and all-Z generators, which as we'll see in Section 11.3 has some nice consequences which make them easier to work with. However, if we drop this restriction, we can find even smaller distance 3 codes.

Example 11.2.6. The following is a $[[5, 1, 3]]$ code:

$$\vec{S}_1 := X \otimes Z \otimes Z \otimes X \otimes I$$
$$\vec{S}_2 := I \otimes X \otimes Z \otimes Z \otimes X$$
$$\vec{S}_3 := X \otimes I \otimes X \otimes Z \otimes Z$$
$$\vec{S}_4 := Z \otimes X \otimes I \otimes X \otimes Z$$

In the next section, we will see how the code distance enables us to detect and correct not just Pauli errors, but a large class of errors.

Exercise 11.3. Consider the following encoder map, which is a "doubled up" version of the GHZ encoder from the previous section:

(11.3)

It embeds 1 logical qubit into 9 physical qubits, so its image should have 8 stabilisers. What are they? What is the code distance?

11.2.2 Detecting and correcting quantum errors

A natural question arises when one first encounters quantum error correcting codes: why the fixation on Pauli errors? In the classical case, it is quite natural to focus on bit-flips (and sometimes bit-loss) as the only basic kinds of errors we care about. Intuitively, these are the only kinds of errors that can occur for classical data. However, there are many ways for a qubit to experience an error, so why is it we can get away with just Pauli X, Y, and Z?

Indeed, quantum theory says that the physical processes that cause quantum errors could act in infinitely many ways on our qubits, and even interact our qubits with some external systems outside our control, introducing unwanted entanglement. To capture all such possibilities, a generic error can be modelled as a unitary like this:

where the first n qubits represent the system we are actually doing the computation on, and the extra system $\mathcal{H}$ represents some environment that those qubits might interact with when the error occurs.

The key trick at this point is to realise that the Pauli matrices I, X, Y, Z span the whole 4D space of 2×2 matrices.

Exercise 11.4. For a matrix:

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

find complex numbers λ_i such that $M = \lambda_0 I + \lambda_1 X + \lambda_2 Y + \lambda_3 Z$.

Consequently, Pauli strings of length n span the whole space of $2^n \times 2^n$ matrices. Using this fact, we can decompose $\mathcal{E}$ as follows:

where $D_{\vec{P}} : \mathcal{H} \to \mathcal{H}$ is some linear map on the rest of the space that can vary with $\vec{P}$. We don't expect to be able to correct *all* errors of this form. For example, if $\mathcal{E}$ simply swapped all our qubits out into $\mathcal{H}$ and replaced them with garbage from the environment, there is no way to recover. So, we should put some kind of reasonable restriction on $\mathcal{E}$. First, let's split $\mathcal{E}$ into two parts based on the weight of the Pauli strings:

For reasonably well-behaved error processes, the second part of the sum will get exponentially small as we increase d. So, in true physicist style, we will just ignore it! This gives us the following pretty good approximation for $\mathcal{E}$:

                                                    (11.4)
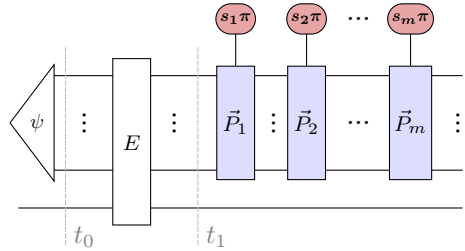
This is a non-trivial assumption, which comes from the fact that we assume the interference from the environment is relatively localised (cf. Remark 11.2.7). An important consequence is that as d gets larger, the form (11.4) gives us a better approximation of an arbitrary error process.

Remark 11.2.7. We can motivate the form of a "reasonably well-behaved" error process (11.4), by considering only those that come from local interactions with the environment. Recall from Section 7.5 that unitary evolutions
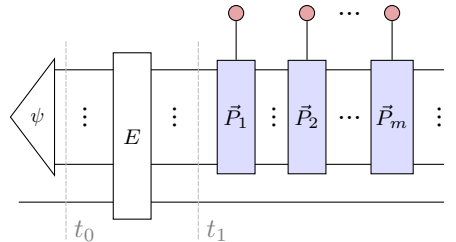
in quantum theory are generated by Hamiltonians, so we can write $\mathcal{E} = e^{itH}$ for some Hamiltonian H. Many physical interactions are well-approximated by so-called **K-local Hamiltonians**, which are those H that can be written as linear combinations of operators with weight at most K. Usually K is small (e.g. 2), so terms of high weight in $\mathcal{E}$ all come from higher-order terms in the Taylor expansion (i.e. terms with high j in equation (7.47) from Section 7.5). When t is small, these terms get exponentially close to zero as we increase j.

Let's use this form of an error to talk about a single round of error detection or correction. We'll start by assuming that we can perform stabiliser measurements perfectly (i.e. without introducing more errors). This is of course not the case, as we'll see when we discuss fault-tolerant measurements in Section 11.4.2.

Fix a stabiliser code of distance d given by the stabiliser group $\mathcal{S} = \langle \vec{P}_1, \ldots, \vec{P}_m \rangle$. Suppose we start with a state $|\psi\rangle \in \text{Stab}(\mathcal{S})$ at t_0, then measure the generators of $\mathcal{S}$ at t_1. If we assume an arbitrary error of the form of (11.4) could happen between t_0 and t_1, the resulting state, which depends on the measurement outcomes $s_1, \ldots, s_m$ looks like this, up to normalisation:

$$(11.5)$$

If we did not detect an error, this means we get outcome $(s_1, \ldots, s_m) = (0, \ldots, 0)$. In that case, the resulting state is the following:

$$(11.6)$$

From (11.4), we can expand $\mathcal{E}$ as a sum over Pauli strings $\vec{P}$ with $|\vec{P}| < d$:

$$\sum_{\vec{P},|\vec{P}|<d} \tag{11.7}$$

Then, since $\mathcal{S}$ has distance d, all $\vec{P}$ are either elements of $\mathcal{S}$ or anti-commute with at least one $\vec{P}_j \in \mathcal{S}$. However, terms in (11.4) corresponding to anti-commuting Pauli strings vanish. To see this, note that stabiliser generators all commute, so we can always move the projector corresponding to $\vec{P}_j$ to the front of the list of projectors in (11.4). Then:

where $(*)$ is using the fact that $\Pi^{(0)}_{\vec{P}_j}|\psi\rangle = |\psi\rangle$ since $|\psi\rangle \in \mathrm{Stab}(\mathcal{S})$.

The only remaining terms are those where $\vec{P} \in \mathcal{S}$. But then, $\vec{P}|\psi\rangle = |\psi\rangle$. Combining this with the fact that $|\psi\rangle$ is also invariant under the projection on to $\mathrm{Stab}(\mathcal{S})$, (11.6) reduces to:

where $D := \sum_{\vec{P}\in\mathcal{S},|\vec{P}|<d} D_{\vec{P}}$ is some (irrelevant) process acting independently on the environment, without disturbing $|\psi\rangle$. Hence, even though E might not be a Pauli string, if we perform a round of stabiliser measurements and detect no errors, we will nevertheless project back on to the error-free state $|\psi\rangle$.

But what happens if we *do* detect errors? Then, we can attempt to **decode** the error syndrome by finding some Pauli string $\vec{Q} \notin \mathcal{S}$ that could have produced that error and undo it. In fact, for reasons that will soon become clear, we often want to find such a $\vec{Q}$ with weight as small as possible.

Definition 11.2.8. For a stabiliser code defined by $\mathcal{S} = \langle \vec{P}_1, \ldots, \vec{P}_m \rangle$ and syndrome $(s_1, \ldots, s_m)$, a **minimum weight decoding** is a Pauli string

$\vec{Q}$ of minimal weight that anti-commutes with $\vec{P}_j$ if and only if $s_j = 1$. Diagrammatically, we have for all j:

Clearly if $(s_1, \ldots, s_m) = (0, \ldots, 0)$, the minimum weight decoding is $\vec{1}$. Otherwise, $\vec{Q}$ will always be a non-trivial Pauli string. If we post-compose the result of a round of stabiliser measurements (11.5) with $\vec{Q}$, we can push $\vec{Q}$ inside, which cancels out the s_j:

Hence, we obtain the form of (11.6) for a new error process E'. If E is a sum over Paulis with weight $\leq \lfloor (d-1)/2 \rfloor$, and $|\vec{Q}| \leq \lfloor (d-1)/2 \rfloor$, then E' is a sum over Paulis with weight at most $d - 1$. Hence, if $\mathcal{S}$ has distance d, we can use the same argument as before to reduce the expression above to $|\psi\rangle \otimes D$, so we have successfully corrected the error E.

Note that the weight of the terms in E' depends on the weight of $\vec{Q}$. This is why we try to find $\vec{Q}$ with weight as small as possible, to give the greatest chance to correcting the error. Naïvely, we can find $\vec{Q}$ by enumerating all of the Pauli strings of weight $1, 2, 3, \ldots$ until we find one with the correct syndrome. Clearly this will take exponential time, so it becomes infeasible for large codes.

Minimum weight decoding for stabiliser codes is (at least) as hard as for classical codes, which is already known to be NP-hard for some families of error correcting codes. Hence, for a stabiliser code to be useful, it needs to

not only have good parameters $[[n, k, d]]$, but also an efficient way to find minimum weight (or at least low weight) decodings.

11.2.3 Encoders and logical operators

An $[[n, k, d]]$ stabiliser code is defined by a stabiliser group $\mathcal{S}$ with $m = n - k$ generators and fixes a 2^k-dimensional subspace $\text{Stab}(\mathcal{S}) \subseteq (\mathbb{C}^2)^{\otimes n}$. Hence, we can think of this as k logical qubits embedded in a space of n physical qubits. $\text{Stab}(\mathcal{S})$ tells us *where* those qubits lie within the big space, but it does not tell how exactly *how* those qubits are embedded. For example, if we are encoding 2 qubits, which part of $\text{Stab}(\mathcal{S})$ corresponds to the first qubit and which to the second qubit? How do unitaries applied to the physical qubits map on to transformations of the logical qubits?

If we only intend to use quantum error correction for quantum memory, the answers to these questions are not too important. However, as we will see in Section 11.4, they are of central importance when we start wanting to perform fault-tolerant computations on encoded qubits.

For that reason, it is useful to define an explicit isometry $E : (\mathbb{C}^2)^{\otimes k} \to (\mathbb{C}^2)^{\otimes n}$ such that the image of E is $\text{Stab}(\mathcal{S})$. This isometry is called an **encoder** associated with code $\mathcal{S}$:

$$k \left\{ \; \vdots \; \boxed{E} \; \vdots \; \right\} n$$

We already saw some explicit examples of encoder maps, built as ZX diagrams in Equations (11.2) and (11.3).

On some occasions, we may actually want to implement the encoder physically, e.g. as a unitary circuit with ancillae. However, this need not be the case, and in fact most of the time, it suffices to treat this as a purely mathematical object, tracking how our k logical qubits are embedded into the space of n physical qubits.

Most importantly, it tracks the relationship between logical maps performed on our k logical qubits and the associated physical maps performed on the n physical qubits.

Definition 11.2.9. A map $F : (\mathbb{C}^2)^{\otimes n} \to (\mathbb{C}^2)^{\otimes n}$ is said to **implement** a map $f : (\mathbb{C}^2)^{\otimes k} \to (\mathbb{C}^2)^{\otimes k}$ in a stabiliser code with encoder E if:

$$\vdots \; \boxed{f} \; \vdots \; \boxed{E} \; \vdots \quad = \quad \vdots \; \boxed{E} \; \vdots \; \boxed{F} \; \vdots \tag{11.8}$$

Intuitively, F acts like f on the codespace of $\mathcal{S}$, which is isomorphic to the space of k qubits. To formalise exactly what "acts like f" means, we

need to choose an isomorphism between this subspace and $(\mathbb{C}^2)^{\otimes k}$, which is exactly what E does for us. This is a very important concept, since in order for quantum error correction to work, our quantum data needs to be continuously protected from errors. Hence, we cannot decode and re-encode qubits every time we want to apply a gate. As we will see in Section 11.4, finding an F satisfying equation (11.8) will help us to implement **fault tolerant** operations on encoded qubits.

Given a generating set of stabilisers for $\mathcal{S}$, we can always derive an encoder by following the procedure we used to prove the FTST in Section 6.2.1. Namely, we can on to $\mathrm{Stab}(\mathcal{S})$ by finding an isometry E such that $\Pi = E^\dagger E$:

$$\tag{11.9}$$

Furthermore, an encoder derived this way will always be a Clifford isometry, so it is easy to reason about the propagation of Pauli errors.
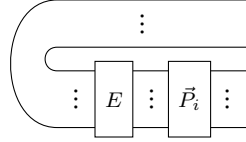
Note that the stabiliser group $\mathcal{S}$ only fixes the image of E (or equivalently, the projector Π). There are many different ways to split a projector, so following the procedure from the proof of the FTST gives us just one possible choice. In particular, for any unitary $U : (\mathbb{C}^2)^{\otimes k} \to (\mathbb{C}^2)^{\otimes k}$, the encoders E and $E' := EU$ have the same image in $(\mathbb{C}^2)^{\otimes n}$, hence they correspond to the same stabiliser code $\mathcal{S}$.

However, whenever E is Clifford, we can perform a nice trick to describe E fully in terms of Pauli strings: we bend the wires! If we bend the input wires around using cups, we obtain an $n + k$ qubit state $|E\rangle$:
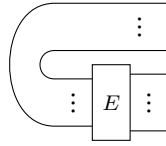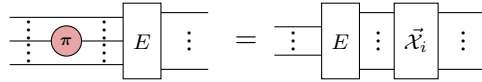
Now, we know from the Fundamental Theorem of Stabiliser Theory that $n + k$ independent stabilisers will uniquely fix $|E\rangle$, and hence will uniquely fix E. We already have $n - k$ stabilisers given by the generators of $\mathcal{S} =$
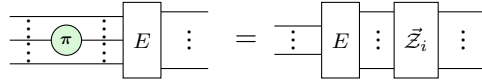
$\langle \vec{P}_1, \ldots, \vec{P}_{n-k} \rangle$:

$$(11.10)$$

So, the question is: where can we find $2k$ more stabilisers? The first thing to note is, assuming E is a Clifford isometry, we can always push arbitrary Pauli strings from its inputs to its outputs, thanks to Proposition 6.1.5. In particular, we can always push single-qubit Pauli X and Z gates through E. As a consequence, we can always find a Pauli string $\vec{\mathcal{X}}_i$ that implements (in the sense of Definition 11.2.9) a Pauli X gate applied in the i-th logical qubit, i.e.

$$(11.11)$$
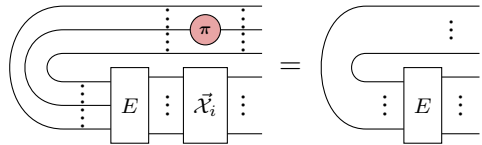
Similarly, for a Z gate on the i-th qubit, there exists a different Pauli string $\vec{\mathcal{Z}}_i$ such that:

$$(11.12)$$
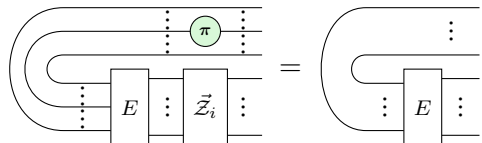
This gives us $2k$ Pauli strings $\{\vec{\mathcal{X}}_1, \ldots, \vec{\mathcal{X}}_k, \vec{\mathcal{Z}}_1, \ldots, \vec{\mathcal{Z}}_k\}$ called the **logical operators** associated with E.

Exercise 11.5. Show that the logical operators $\{\vec{\mathcal{X}}_1, \ldots, \vec{\mathcal{X}}_k, \vec{\mathcal{Z}}_1, \ldots, \vec{\mathcal{Z}}_k\}$ are self-adjoint and commute with every element of the stabiliser group $\mathcal{S}$. Furthermore, show that the pair of logical operators $\vec{\mathcal{X}}_i$ and $\vec{\mathcal{Z}}_i$ anti-commute for all i and all other pairs of logical operators commute.

By moving the X gate to the righthand-side of (11.11) and bending the wires, we see that each logical operator $\vec{\mathcal{X}}_i$ gives us a stabiliser for the Choi state $|E\rangle$:

$$(11.13)$$

Similarly, each $\vec{\mathcal{Z}}_i$ gives us a stabiliser for $|E\rangle$:

$$(11.14)$$

Note that the operators of the form (11.13) and (11.14) all pairwise commute. This is because the -1 coming from anti-commuting an X and Z on the i-th qubit is always cancelled out by anti-commuting $\vec{\mathcal{X}}_i$ with $\vec{\mathcal{Z}}_i$ on the last n qubits (cf. Exercise 11.5).

If we combine the stabilisers coming from $\mathcal{S}$ in equation (11.10) with those coming from the logical operators, we obtain $n - k + 2k = n + k$ stabilisers for $|E\rangle$, which is by construction an $n + k$ qubit state. Hence, E is totally fixed up to a scalar factor.

Hence, if we perform the method from Section 6.2.1 for the full set of $n+k$ stabilisers for $|E\rangle$, then bend the wires, we'll get the encoder associated with a stabiliser group and a set of logical operators. In general, this will be a relatively deep ZX-diagram, with approximately one layer for each stabiliser and logical operator. We could then try and simplify things to get a more compact form, e.g. by reducing to GSLC or AP form, but in general this picture of the encoder might be a bit awkward to work with.

However, as we'll see in Section 11.3, for a certain family of codes, called CSS codes, things get a lot simpler!

11.2.4   The decoder

The encoder has a dual operation, which we call the *ideal decoder*, or simply the **decoder**. This is slightly more difficult to work with than the encoder, so whenever possible we will state things in terms of the encoder rather than the decoder. However, having a way to at least in principle decode n physical qubits back into k logical qubits while detecting/correcting errors is still a useful thing to have in our back pocket.

The encoder is an isometry, so we know how to implement it straight away as a unitary circuit with ancillae. In fact, this definition of E is already clear from the definition of the encoder in equation (11.9) in terms of the splitting of the projectors onto the stabiliser subspace.

However, we can't deterministically implement $E^\dagger$ because it would require performing the unitary $U^\dagger$ then post-selecting each of the ancilla qubits on to $\langle 0|$. However, we can *measure* those ancillae instead:
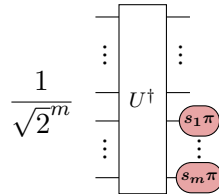
But what are these measurement outcomes $(s_1, \ldots, s_m)$? Recall the following equation from Exercise* 6.10:

This implies that for an encoded state $|\Psi\rangle$, performing $U^\dagger$ and measuring the ancillae will give us the same outcome $\vec{s}$ as measuring all of the stabilisers.

Hence, we can use the measurement outcomes from the ancillae to do error detection or correction. If we get outcome $\vec{s} = (0, \ldots, 0)$, we can conclude that no logical error has occurred with weight $< d$. If we get any other syndrome, we can do least-distance decoding to find the smallest weight physical error $\vec{Q}$ with error syndrome $\vec{s}$, and then figure out the appropriate correction to do on the k logical qubits we have left. Since $U^\dagger$ is Clifford, we can do this efficiently.

Exercise 11.6. Fix a stabiliser code with generators $\{\vec{P}_1, \ldots, \vec{P}_m\}$ and distance d. For a syndrome $\vec{s}$, let $\vec{Q}_{\vec{s}}$ be a Pauli of minimal weight with that syndrome, and let $\vec{P}_{\vec{s}}$ be the first k qubits of the Pauli string $U^\dagger \vec{Q}_{\vec{s}} U$. Show that for any Pauli string $\vec{Q}$ with syndrome $\vec{s}$ and weight t where $2t + 1 \leq d$, we have:

Given this, we therefore define the **decoder** as a non-deterministic process consisting of a measurement yielding syndrome $\vec{s}$ followed by the associated Pauli correction $\vec{P}_{\vec{s}}$ defined as in Exercise 11.6:

(11.15)

When $\vec{s} = \vec{0}$, $\vec{P}_{\vec{s}} = I$, so $\vec{D}_{\vec{0}} = E^\dagger$. Hence this process acts like $E^\dagger$ in the

error-free case, but it additionally also "swallows" any correctable errors when $\vec{s} \neq \vec{0}$.

Remark* 11.2.10. If we don't care about the outcome of the syndrome measurement and just want the error-corrected state out, we can represent the decoder as a single quantum channel (cf. Section* 2.6.2) with Kraus operators $\{D_{\vec{s}}\}_{\vec{s}\in\mathbb{F}_2^m}$, i.e.

$$\mathcal{D}(\rho) := \sum_{\vec{s}\in\mathbb{F}_2^m} D_{\vec{s}}\rho D_{\vec{s}}^{\dagger}$$

One can check this is trace-preserving and for any correctable Pauli error $\vec{Q}$, we have $\mathcal{D}(\vec{Q}\rho\vec{Q}^{\dagger}) = \mathcal{D}(\rho)$.

11.3 CSS codes

In this section, we'll define a family of stabiliser codes called **Calderbank-Shor-Steane codes**, or **CSS codes** for short. As you may have already noticed from Examples 11.2.4 and 11.2.5, some stabiliser codes have generators that split nicely into an all-X part and an all-Z part. In a sense, such codes behave like two classical error correcting codes mushed together, where the parity checks of one code become the Z stabilisers (which detect bit errors) and those of the other code become the X stabilisers (which detect phase errors).

That is pretty much all there is to defining a CSS code, once you additionally account for the fact that all of the stabilisers need to commute. It turns out there is a very natural way to impose this in terms of bitstrings.

Exercise 11.7. Suppose we define two Pauli strings from the bitstrings $\vec{v}$ and $\vec{w}$ as follows:

$$\vec{X} := \bigotimes_{j=1}^n X^{v_j} \qquad \vec{Z} := \bigotimes_{j=1}^n Z^{w_j}$$

Show that $\vec{X}$ and $\vec{Z}$ commute if and only if $\vec{v}$ and $\vec{w}$ are orthogonal, i.e. $\vec{v}^T\vec{w} = 0 \pmod 2$.

Hence, if we fix two classical codespaces, one for the X-stabilisers and one for the Z-stabilisers, all the stabilisers will commute precisely when those subspaces are mutually orthogonal. We say two subspaces S, T are orthogonal if $\vec{v}^T\vec{w} = 0$ for all $v \in S, w \in T$, or equivalently $T \subseteq S^{\perp}$.

Definition 11.3.1. The *CSS code* generated by orthogonal subspaces $S, T \subseteq \mathbb{F}_2^n$ is a stabiliser code whose generators are of the following form:

$$\vec{X}_i := \bigotimes_{k=1}^{n} X^{(v_i)_k} \qquad\qquad \vec{Z}_j := \bigotimes_{k=1}^{n} Z^{(w_j)_k}$$

where $\{\vec{v}_1, \ldots, \vec{v}_p\}$ is a basis spanning S and $\{\vec{w}_1, \ldots, \vec{w}_q\}$ a basis spanning T. A CSS code is called *maximal* if $T = S^\perp$.

That is, we let the basis vectors of S define the X generators and we let the basis vectors of T define the Z generators. Since addition in $\mathbb{F}_2$ is taken modulo 2, orthogonality guarantees that each X generator overlaps with a Z generator in an even number of places, which makes the group commutative. Using this fact, it is easy to verify the resulting group is a stabiliser group.

Example 11.3.2. Let S be a 3D subspace of $\mathbb{F}_2^7$ spanned by:

$$\{(1,0,0,0,1,1,1), (0,1,0,1,0,1,1), (0,0,1,1,1,0,1)\}$$

This particular subspace is known as a *Hamming code.* It's classical code distance is 3, and it has the nice property that S is orthogonal to itself. Hence, we can use S to derive both the X and Z generators of a CSS code. In fact, we already saw this code: it is the *Steane code* from Example 11.2.4.

$$\vec{X}_1 := X \otimes I \otimes I \otimes I \otimes X \otimes X \otimes X$$
$$\vec{X}_2 := I \otimes X \otimes I \otimes X \otimes I \otimes X \otimes X$$
$$\vec{X}_3 := I \otimes I \otimes X \otimes X \otimes X \otimes I \otimes X$$
$$\vec{Z}_1 := Z \otimes I \otimes I \otimes I \otimes Z \otimes Z \otimes Z$$
$$\vec{Z}_2 := I \otimes Z \otimes I \otimes Z \otimes I \otimes Z \otimes Z$$
$$\vec{Z}_3 := I \otimes I \otimes Z \otimes Z \otimes Z \otimes I \otimes Z$$

11.3.1 Stabilisers and Pauli ZX diagrams

Recall from Section 4.3 that phase-free ZX-diagrams can always be put into Z-X or X-Z normal form, which correspond respectively to representations of an $\mathbb{F}_2$-linear subspace S using a basis for S or a basis for $S^\perp$:

$$\propto \sum_{b \in S} |b\rangle \qquad \text{where} \qquad S = \mathrm{Span}\{v_1, \ldots, v_k\}$$

$$\propto \sum_{b \in S} |b\rangle \qquad \text{where} \qquad S^\perp = \text{Span}\{w_1, \ldots, w_j\}$$

If we combine this knowledge with these equations, which follow from (π):

$$\tag{11.16}$$
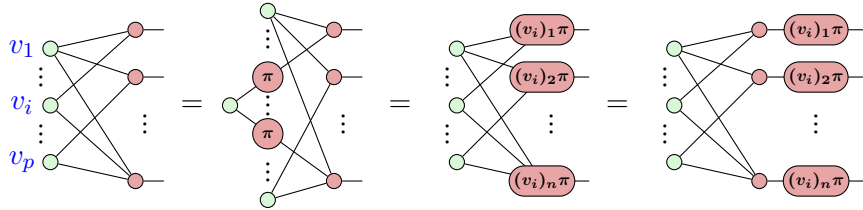
we can see immediately how to derive a generating set of stabilisers for a phase-free ZX-diagram. As we'll see in this section, those stabilisers always generate a CSS code, and conversely any CSS code can be presented using a phase-free ZX-diagram.

11.3.2 Maximal CSS codes as ZX diagrams

Eq. (11.16), plus the two normal forms from Section 4.3.1, will give us everything we need to prove our main theorem.

Theorem 11.3.3. For any $\mathbb{F}_2$-linear subspace $S \subseteq \mathbb{F}_2^n$, $|\psi\rangle$ is stabilised by the maximal CSS code $(S, S^\perp)$ if and only if it is equivalent, up to a scalar factor, to a phase-free ZX diagram with an X-Z normal form given by S (or equivalently with a Z-X normal form given by $S^\perp$).

Proof Suppose $|\psi\rangle$ is described by a phase-free ZX-diagram. Then it can be translated into X-Z normal form, for some basis $\{v_1, \ldots, v_p\}$ of an $\mathbb{F}_2$-linear space S. Then, for each v_i, we can apply Eq. (11.16) to introduce an X phase of π on every wire adjacent to the Z spider labelled by v_i and commute it to the output using (sp'):

This shows that $|\psi\rangle$ is invariant under the action of the Pauli operator $\mathcal{X}_i := \bigotimes_{k=1}^n X^{(v_i)_k}$. Hence, $|\psi\rangle$ is the $+1$ eigenstate of all of the p independent X stabilisers of the maximal CSS code $(S, S^\perp)$. Similarly, we can compute the Z-X normal form of $|\psi\rangle$ and show that it is the $+1$ eigenstate of all q independent Z stabilisers $\mathcal{Z}_j := \bigotimes_{k=1}^n Z^{(w_j)_k}$. This gives $p + q = n$ independent stabilisers

for $|\psi\rangle$, hence it is uniquely fixed by the FTST Theorem 6.2.11. Conversely, any maximal CSS code fixes a state whose stabilisers are given by $\mathcal{X}_i$ and $\mathcal{Z}_j$ as before, so they will be equal to a phase-free ZX diagram with X-Z normal form given by S, or equivalently, with Z-X normal form given by $S^\perp$. $\quad\square$

This proof gives us an evident way of translating the stabiliser generators of a maximal CSS code into a ZX diagram. In fact, it gives us two equivalent ways, using the Z-X normal form, which gives us a generating set of X stabilisers, or the X-Z normal form, which gives us the Z stabilisers. Interestingly, we only ever need to represent one kind of stabilisers for a maximal CSS code diagrammatically, because $S^\perp$ is uniquely fixed by S and vice-versa.

11.3.3 Non-maximal CSS codes as ZX encoder maps

For a non-maximal CSS code, we should end up with an encoder map from k logical qubits to n physical qubits. Since we already know how to turn CSS stabilisers into phase-free diagrams, we can use the wire-bending trick from Section 11.2.3 to treat logical operators as stabilisers on a bigger $n + k$ qubit state and therefore add them to the diagram as well.

As an example, lets return to the 7-qubit Steane code from Example 11.3.2. We will switch to a more compact notation for writing its stabilisers, where X_i (resp. Z_i) corresponds to an n-qubit operator acting non-trivially on the i-th qubit with a Pauli X (resp. Z):

$$\vec{X}_1 := X_1 X_5 X_6 X_7 \quad \vec{X}_2 := X_2 X_4 X_6 X_7 \quad \vec{X}_3 := X_3 X_4 X_5 X_7$$
$$\vec{Z}_1 := Z_1 Z_5 Z_6 Z_7 \quad \vec{Z}_2 := Z_2 Z_4 Z_6 Z_7 \quad \vec{Z}_3 := Z_3 Z_4 Z_5 Z_7$$

This CSS code is non-maximal, and encodes $7 - 6 = 1$ logical qubits. Hence we should fix 2 additional logical operators $\vec{\mathcal{X}} := X_4 X_5 X_6$, $\vec{\mathcal{Z}} := Z_4 Z_5 Z_6$. As we noted before, we only need one kind of stabiliser to build the ZX diagram, so applying the recipe from the previous section to the X stabilisers and logical operator, we obtain the following picture, where we label the logical qubit 0 and the physical qubits 1-7. We can then put the logical qubit on

the left and rearrange some of the physical qubits to obtain the following:
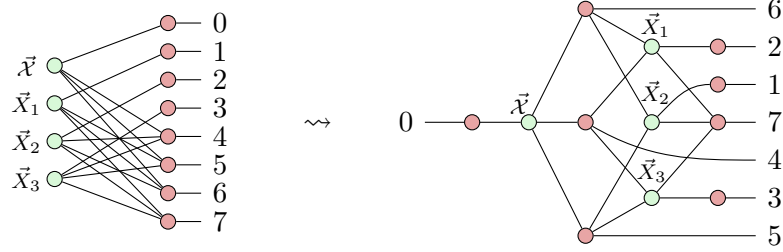
Note that, if we follow this recipe, we will always get identity spiders on the inputs, which are redundant. If we simplify them away, we'll always end up with something in generalised parity form (for the X form of the encoder) or the colour-reverse of generalised parity form (for the Z form of the encoder).

$$(11.17)$$
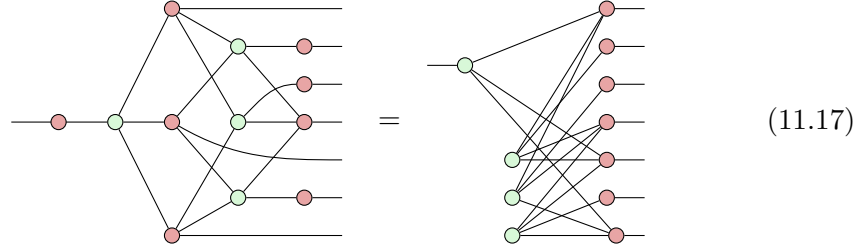
As a result, we can always write a CSS encoder map in generalised parity form using just its X-stabilisers and X-logical operators as follows:

1. Place an output X-spider on all n outputs.
2. For each of the k X-logical operator, add an input Z-spider connected to the output X-spiders of its support.
3. For each X-stabiliser generator, add an internal Z-spider connected to the output X-spiders of its support.

Equation (11.17) gives the X-form of the Steane code encoder. If we used the Z-stabilisers and logicals instead, we'll obtain the Z-form. In the case of the Steane code, this is the exact same picture, but with the colours reversed:

$$(11.18)$$
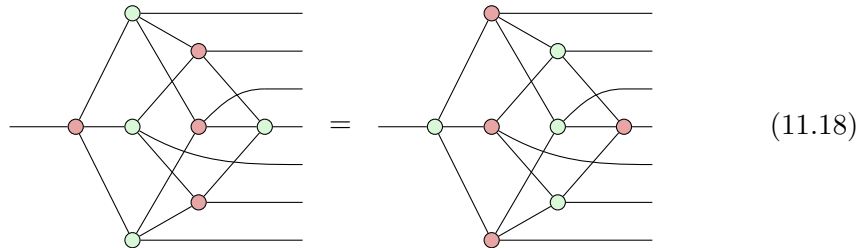
This will not always be the case, and it comes from the fact that the Steane code is **self-dual**. We'll discuss self-dual codes more in Section 11.4.1.1.
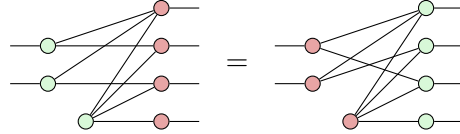
Example 11.3.4. Recall the $[\![4,2,2]\!]$ error detecting code from Example 11.2.5:

$$\vec{X}_1 := X \otimes X \otimes X \otimes X$$
$$\vec{Z}_1 := Z \otimes Z \otimes Z \otimes Z$$
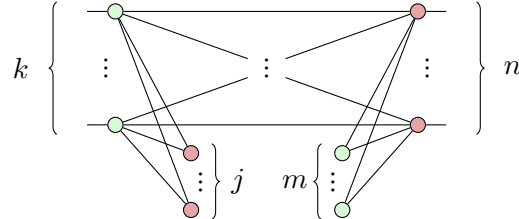
This has two stabilisers on 4 qubits, so it encodes 2 logical qubits. Hence, in order to fully specify the encoder map, we should fix two pairs of anti-commuting logical operators which commute with the stabilisers and each other. There are multiple solutions to this problem, so we will just choose one:

$$\vec{\mathcal{X}}_1 := X \otimes X \otimes I \otimes I$$
$$\vec{\mathcal{Z}}_1 := Z \otimes I \otimes Z \otimes I$$
$$\vec{\mathcal{X}}_2 := X \otimes I \otimes X \otimes I$$
$$\vec{\mathcal{Z}}_2 := Z \otimes Z \otimes I \otimes I$$

These all have weight 2, so they will have even overlap with both stabilisers (and hence commute with them). It is also easy to check that $\vec{\mathcal{X}}_i \vec{\mathcal{Z}}_i = -\vec{\mathcal{Z}}_i \vec{\mathcal{X}}_i$ and all other pairs of logicals commute. We can now use this data to construct the X-form and Z-form of the encoder for the $[\![4,2,2]\!]$ code:

$$(11.19)$$

So, we have seen how to turn any CSS code described by stabilisers and logical operators into a phase-free ZX-diagram of its encoder. Conversely, we can treat *any* isometry described by a phase-free ZX diagram as a CSS code just by computing its generalised parity form. Recall from Chapter 4 that the generalised parity form looks like this:

Furthermore if E is an isometry, we can conclude from Proposition 4.2.8 that $j = 0$, giving us:
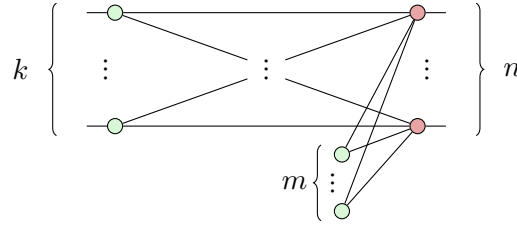
From this picture, we can immediately read off the m X-stabilisers and k X-logical operators associated with E. If do everything colour-reversed, we'll get a different normal form for the encoder:
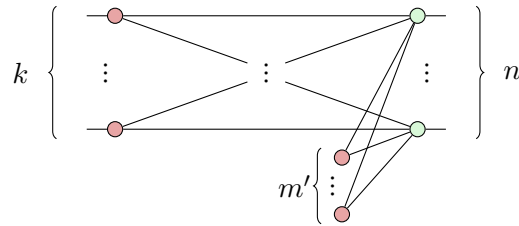
from which we can read off the Z-stabilisers and Z-logical operators.

In summary, we have given an efficient procedure for writing a phase-free isometry from a CSS code with logical operators and for turning any phase-free isometry into its associated CSS code and logical operators.

11.3.4   The surface code

One particular family of CSS codes has been so well-studied in recent years that it deserves some special attention. The **surface code** is a family of CSS codes that encode a single logical qubit in a square (or rectangular) lattice of physical qubits. Larger lattices define codes with a higher code distance.

While they aren't the most memory-efficient codes we know about, surface codes have a number of nice properties coming from this regular geometric structure. For one thing, they can be implemented using only nearest-neighbour gates on a 2D architecture, so there is never any need to perform swap gates or otherwise break planarity. A second nice feature is that even for surface codes with very high code distances, we have efficient algorithms for decoding error syndromes, as we'll see in Section 11.3.4.1 below. Finally, they serve as a useful baseline for fault-tolerant computation, as we know (at least in principle) how to implement all the ingredients needed to implement universal quantum computation on surface-code-encoded qubits in a fault-tolerant manner. In fact, fault-tolerant computation in the surface code has one of the best **thresholds** we know about. That is, they can tolerate quite

a bit of noise at the hardware level while still being able to supress errors. More on that later.

In this section, we will use the slightly more compact, "rotated" version of the surface code. Stabilisers are defined as follows. We start with a rectangular lattice with $d \times e$ vertices, corresponding to qubits. We aim to encode a single logical qubit, so we need to fix $de - 1$ stabilisers. To do so, we first colour in the lattice in a checkerboard pattern, where each darkerarea corresponds to an X stabiliser on all of its adjacent qubits, whereas each lighterarea corresponds to a Z stabiliser. Colouring the inside of the lattice in this way gives $(d-1)(e-1)$ stabilisers. To get all $de - 1$ stabilisers, we still need to fix $d + e - 2$ additional stabilisers. To get these, we introduce weight-2 stabilisers along the boundaries at every other edge, which we depict as "blobs". We colour these blobs in the oppose colour to the nearest tile, obtaining the following picture:

$$
\begin{aligned}
&\vec{X}_1 := X_2 X_3 X_5 X_6 && \vec{X}_2 := X_4 X_5 X_7 X_8 \\
&\vec{X}_3 := X_1 X_4 && \vec{X}_4 := X_6 X_9 \\
&\vec{Z}_1 := Z_1 Z_2 Z_4 Z_5 && \vec{Z}_2 := Z_5 Z_6 Z_8 Z_9 \\
&\vec{Z}_3 := Z_2 Z_3 && \vec{Z}_4 := Z_7 Z_8
\end{aligned}
$$

(11.20)

There are $2(d-1) + 2(e-1)$ edges around the whole boundary, so adding a "blob" to every other edge gives us $(2(d-1) + 2(e-1))/2 = d + e - 2$ more stabilisers as required. By design, all stabilisers of different types overlap on two qubits, so they commute. Since we alternate edges, one pair of opposite boundaries (in this case the left and right) will end up with X-blobs and one pair (top and bottom) with Z-blobs. In the literature, these are called *X-boundaries* and *Z-boundaries*, respectively.
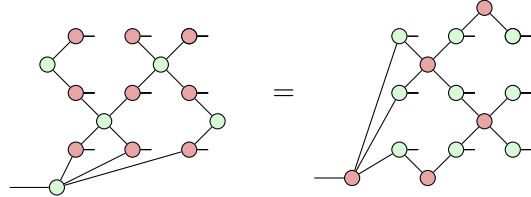
The logical $\vec{\mathcal{X}}$ operator consists of a line of Pauli X operators connecting the two X-boundaries, whereas the logical $\vec{\mathcal{Z}}$ operator consists of a line connecting the two Z-boundaries:

$$\rightsquigarrow \quad \vec{\mathcal{X}} := X_7 X_8 X_9 \qquad\qquad \rightsquigarrow \quad \vec{\mathcal{Z}} := Z_1 Z_4 Z_7$$
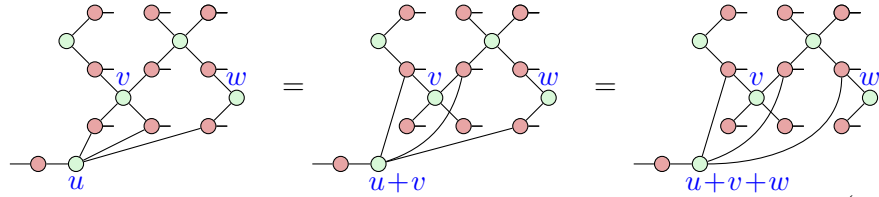
Note that the specific choice of path between the boundaries is not important and we are even allowed to cross tiles diagonally. However, it is important that the path touches each area of the opposite colour an even number of times. This ensures that the logical operator commutes with all of the stabilisers. In the example above we could have equivalently chosen $X_4 X_5 X_6$ or $X_5 X_6 X_7$ for $\vec{\mathcal{X}}$, but not $X_7 X_8 X_5 X_6$, because the latter touches a green tile 3 times.

Using the stabilisers and the logical operators for the surface code, we can apply the recipe from the previous section to construct its encoder. In fact, we can represent the encoder using two equivalent ZX diagrams, one based on the X-stabilisers and one on the Z-stabilisers:

$$\tag{11.21}$$

Note how the diagrams of the encoders have a direct visual relationship to the picture (11.20): to draw the X-stabilisers, we put an X spider on every vertex, place a Z spider in the centre of each red region in (11.20), and connect it to all of the adjacent vertices. Finally, we "embed" the logical X operator by placing a Z spider on the input and connecting it to each of the vertices where $\vec{\mathcal{X}}$ has support. For the Z-stabilisers, we apply the same routine, reversing the roles of X and Z.

In the surface code, we can topologically deform a logical operator by multiplying it by any stabiliser. We can perform the same calculation graphically using strong complementarity. As we saw in the proof of Lemma 4.3.4, we can treat spiders as bit-vectors, and by applying strong complementarity, we can "add" the neighbourhood of one spider to that of another, modulo 2. Applying this concept to the surface code, we obtain for example:

$$\tag{11.22}$$

This just amounts to changing the basis for the linear space S represented by this normal form, which has the same effect as changing the generators for the associated stabiliser group.

11.3.4.1   Decoding errors in the surface code

Aside from the fact that they can be implemented with nearest-neighbour operations in a 2D architectures, surface codes are also popular because error syndromes can be efficiently decoded by taking advantage of their geometric structure. Recall from Section 11.2.2 that the **decoding problem** for stabiliser codes refers to the problem of mapping the outcome of a sydrome measurement to a Pauli correction $\vec{Q}$ that is most likely to recover our error-free state.

There are many proposals for fast decoders for the surface code. Here, we will sketch how one of the simplest ones, based on **minimum weight perfect matching** works. First note that since surface codes are CSS codes, we can identify Z-errors in $\vec{Q}$ using the outcomes of X stabiliser measurements, and vice-versa. Hence, we can treat these two types of errors independently.

Suppose a certain configuration of Z errors occurs on our physical qubits. Much as we did in Section 11.2.2, we can figure out where the error syndrome will be by "pushing" the error through the Pauli projectors corresponding to X measurements, using the π-copy rule, and seeing which projectors end up with a π:
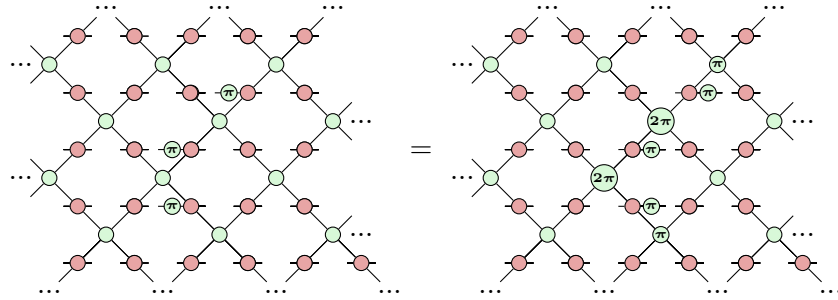
This is the same thing as figuring out which stabilisers anti-commute with the given error. If an even number of Z errors lie in the support of an X projector, a pair of π phases will cancel out, yielding a 0 measurement outcome. Hence, the syndrome only reveals the places where an odd number of errors overlap with a stabiliser. As a consequence, error syndromes will always appear as

the endpoints of a "path of errors" through the lattice:

$$(11.23)$$
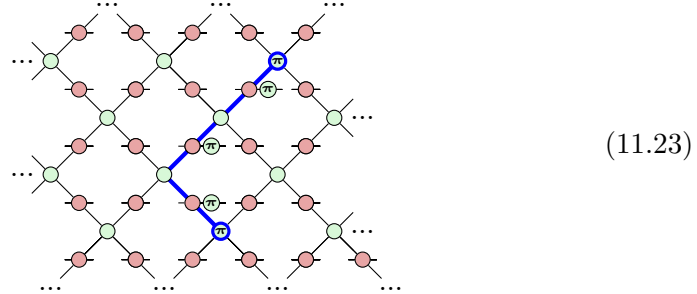
These paths will always either connect a pair of locations where the error syndrome is 1, as we see above, or they will connect a single syndrome-1 location to a boundary of the surface, as in this example:
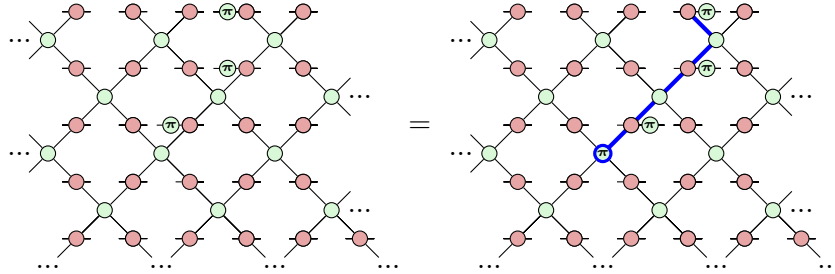
Since Z boundaries don't have any X stabiliser measurements, this also works.

Hence, if we get just two syndrome-1 outcomes far from the boundary, the most likely place errors occured were along the shortest path between those two spots (assuming as we have been that errors occur independently and with the same probability on every qubit). If we get just one syndrome-1 outcome, the most likely place errors occurred was along the shortest path between that syndrome and the boundary without non-commuting stabilisers.

"Hang on!" you might say, "there can be multiple shortest paths between points!" And you'd be right. However, all of these paths are equivalent, up to stabilisers. For example, there are 3 shortest paths between the syndromes in (11.23). The 2 other ones can be obtained from the one shown by multiplying by Z stabilisers. Since stabilisers don't change the encoded state, correcting errors along any of these 3 paths (or actually *any* path between these endpoints) will have the same effect. The reason this works is, as we showed for the logical embedding at the end of the previous section, the stabilisers of the surface code relate topologically-equivalent paths through the surface, provided they have the same end points.

Of course, this only tells us what to do if we see zero, one, or two syndrome-1 measurements. However, we can extend this to an algorithm that does a pretty good job decoding any syndrome. First, we make a graph whose nodes correspond to places we observe a syndrome-1 outcome (and an extra dummy node for the boundary). We label the edges with a weight indicating the length of the shortest path between those locations, e.g.

$$\begin{array}{c}
\text{(graph with nodes } v_1, v_2, v_3, b \text{ and edge weights } 5, 9, 8, 8, 7, 4)
\end{array} \tag{11.24}$$

A **perfect matching** is then a subset of the edges of this graph, where every node (except the "dummy" node for the boundary) is adjacent to exactly 1 edge. A **minimum weight perfect matching** (MWPM) is a perfect matching where the sums of the weights on the chosen edges is minimal. An example of a MWPM for the graph above is:

$$\begin{array}{c}
\text{(graph with nodes } v_1, v_2, v_3, b \text{ and edge weights } 5, 9, 8, 8, 7, 4 \text{ with matching highlighted)}
\end{array}$$
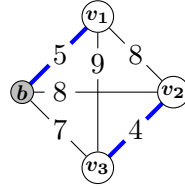
The simplest minimum weight perfect matching decoder therefore proceeds as follows. For the X stabiliser measurements, make a syndrome graph as in equation (11.24) and compute its MWPM. Then, correct Z errors along the paths indicated by the MWPM. After that, rinse and repeat for Z stabiliser measurements and X errors.

It is worth noting that this algorithm does not always find the minimum weight error associated with a syndrome, but it often gets pretty close, especially in the case where there aren't too many syndrome-1 outcomes. It is also very fast, particularly if we are allowed to just approximate the MWPM and/or compute in parallel. We will briefly discuss various implementations of this algorithm and their performance in the References of this chapter.

11.3.5 Scalable ZX notation for CSS codes

As might be clear from diagrams like (11.17), writing CSS encoder maps explicitly can start to get unwieldy for large numbers of stabilisers or stabilisers with relatively high weight. Later on, we will also want to prove some

generalities about all CSS codes, so it would be useful to come up with some notation for writing a *generic* CSS code as a ZX-diagram.

Thankfully, the scalable ZX-calculus, which was introduced in Section 10.2, can solve both of these problems. To see how this works, we can start from the recipe given in the previous section for building an encoder for a CSS code from its X-logical operators and X-stabilisers. Suppose we represent this data as a pair of boolean matrices L_X and S_X, whose columns correspond to each of the operators and whose rows correspond to qubits, where a 1 indicates the presence of an X. For example, the matrices (L_X, S_X) associated with the Steane code are the following:

$$\vec{\mathcal{X}} := I \otimes I \otimes I \otimes X \otimes X \otimes X \otimes I \Big\} \quad \rightsquigarrow \quad L_X := \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}^T$$

$$\left. \begin{aligned} \vec{X}_1 &:= X \otimes I \otimes I \otimes I \otimes X \otimes X \otimes X \\ \vec{X}_2 &:= I \otimes X \otimes I \otimes X \otimes I \otimes X \otimes X \\ \vec{X}_3 &:= I \otimes I \otimes X \otimes X \otimes X \otimes I \otimes X \end{aligned} \right\} \quad \rightsquigarrow \quad S_X := \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}^T$$

Whereas in the previous section, we needed to say in words how to build the encoder out of X operators, we can now do it succinctly as a single diagram, which works for any CSS code described by the pair (L_X, S_X):

$$(11.25)$$

Inspecting this picture, we can indeed see that it is saying to introduce k input spiders for the X-logicals, and connect them according to where X's appear, then introduce m internal spiders for the X-stabilisers and again connect them to outputs according to where X's appear.

This gives us the X form for the encoder, but of course we should also be able to construct a Z form by reversing the role of the two colours of spider. To get the colour reverse of a matrix arrow, we can just reverse the direction and take the transpose of the matrix:

$$(11.26)$$

Hence, if we equivalently present a CSS code as a pair of matrices (L_Z, S_Z) describing the Z-logical operators and Z-stabilisers, respectively, we can write

the associated encoder map as:

$$
\begin{array}{c}
\includegraphics[]{eq1127left} = \includegraphics[]{eq1127right}
\end{array}
\tag{11.27}
$$

This not only gives us convenient notation for the encoder, but also for the stabiliser measurements themselves. It was shown in Exercise 6.7 that the Pauli projectors associated to all-X or all-Z Pauli stabiliser measurements take a particularly simple form:

$$
\Pi^{(s)}_{Z\ldots Z} \;=\; \includegraphics[]{pizz} \qquad\qquad \Pi^{(s)}_{X\ldots X} \;=\; \includegraphics[]{pixx}
$$

If $s = 0$, the projector on to a stabiliser $\vec{Z}_j$ consists of an isolated X-spider connected to Z-spiders on the support of the $\vec{Z}_j$. If we compose all of the projectors on to Z-stabilisers together and fuse spiders, we'll obtain this picture:

$$
\includegraphics[]{eq1128} \tag{11.28}
$$

Similarly, if we project on to the 0 outcome with all of the X stabilisers, we'll obtain:

$$
\includegraphics[]{eq1129} \tag{11.29}
$$

Exercise 11.8. Show that for any A, the following maps are always projectors, up to a scalar:

$$
\includegraphics[]{ex118}
$$

If we simply compose maps (11.28) and (11.29), we will get the projection onto the stabiliser subspace $\mathrm{Stab}(\mathcal{S})$. This is what happens if we measure all of the stabilisers and get outcome 0. To represent other outcomes, we should place a 0 or a π on each of the internal spiders of the Pauli projections. To do this for generic outcomes, we need to extend the scalable notation a bit.

Back when we introduced the scalable notation, we interpreted a scalable spider labelled by a phase α as n spiders, all with the same phase α on them.

However, we could as well label a spider with a vector of different phases $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$ and define scalable spiders as:
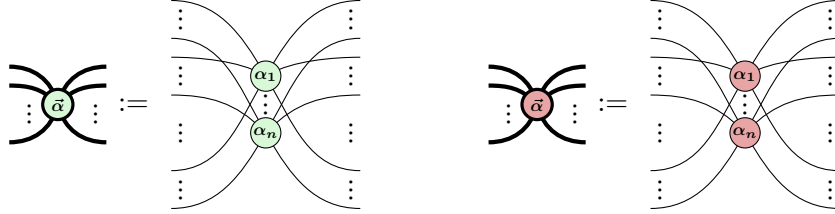
Of particular interest for us are vectors of 0s and πs, i.e. vectors of the form $\vec{b} \cdot \pi := (b_1\pi, \ldots, b_n\pi)$ for some boolean vector $\vec{b} \in \mathbb{F}_2^n$.

Using this extended notation, a full stabiliser measurement for a CSS code with outcome syndrome $\vec{s} = (\vec{x}, \vec{z})$ can be written compactly as:

$$(11.30)$$

For a CSS code, we can assume without loss of generality that all of the X-logical operators and X-stabilisers are linearly independent. In other words, we can assume the block matrix $(L_X \ \ S_X)$ is injective. Consequently, there exists some parity matrix J such that:

$$(11.31)$$

Since injective parity maps are always isometries, we have:

$$(11.32)$$

This implies that the encoder (11.27) is an isometry, and is in fact a strictly stronger condition.

Exercise* 11.9. In Exercise 11.5, you were asked to show that for logical operators, $\vec{\mathcal{X}}_i$ anti-commutes with $\vec{\mathcal{Z}}_j$ if and only if $i = j$. We can express this condition in terms of boolean matrices by stating that $L_Z^T L_X = I$. In terms of scalable notation, this implies that:

$$(11.33)$$

We also know that stabilisers should commute with logical operators and with each other. In terms of matrices, we can express this as $S_Z^T L_X = S_Z^T S_X = L_Z^T S_X = 0$. Pictorially:

$$
\underset{k \quad\quad\quad\quad k}{\xrightarrow{L_X\ S_Z^T}} \quad \propto \quad \underset{k \quad\quad\quad\quad k}{\xrightarrow{S_X\ S_Z^T}} \quad \propto \quad \underset{k \quad\quad\quad\quad k}{\xrightarrow{S_X\ L_Z^T}} \quad \propto \quad {}^k\!\!\!-\!\!\circ \quad \circ\!\!-\!\!{}^k
$$
(11.34)

Give a graphical derivation of (11.33) and (11.34) using just (11.27), the scalable ZX rules, and the fact that the encoder is an isometry. *Hint: Start by composing the Z-form of the encoder with its adjoint and applying idempotence to remove one projector from the middle before applying (11.27).*

This gives us some nice tools for working with CSS codes generically, which we'll use several times when it comes to deriving fault-tolerant operations in Section 11.4.

11.4 Fault-tolerance

If we want to perform quantum computations in a way that can withstand errors, coming up with a good quantum error correcting code is only half the story.

We also need to figure out how to perform state preparations, gates, and measurements on encoded qubits. Suppose that every time we performed a gate we had to decode our n physical qubits into k logical qubits, perform the gate, then re-encode. This seems like a lot of work just to do one gate, but what is actually worse is that our logical state will be completely unprotected between when we decode and re-encode our qubits. We will run into similar problems with state preparations and measurements.

We saw half of a solution to this problem already in Section 11.2.3 with Definition 11.2.9. If we want to perform a unitary gate U on our logical qubits, we can find some other $\widetilde{U}$ acting on the physical qubits, satisfying:

$$
\vdots\ \boxed{U}\ \vdots\ \boxed{E}\ \vdots \quad = \quad \vdots\ \boxed{E}\ \vdots\ \boxed{\widetilde{U}}\ \vdots
$$

Essentially the same idea applies to state preparations, except there are no input wires, so we don't need an encoder on the RHS. That is, we can implement the preparation of a logical state $|\psi\rangle$ by preparing some physical state $|\widetilde{\psi}\rangle$ satisfying:

$$
\langle\psi|\ \vdots\ \boxed{E}\ \vdots \quad = \quad \langle\widetilde{\psi}|\ \vdots
$$

Of course, one could prepare $|\widetilde{\psi}\rangle$ by preparing $|\psi\rangle$ then performing the encoder isometry as an actual quantum operation, which we could implement using unitaries and ancillae. However, as we'll see later, there are various reasons we might want to prepare $|\widetilde{\psi}\rangle$ directly in a different way.

Finally, implementing measurements on encoded qubits is also a similar idea, but with a little twist accounting for the fact that our physical measurement might have more outcomes than our logical one. While we might in general want to implement arbitrary measurements on encoded qubits, lets focus just on ONB measurements to make things a bit simpler.

Let $[m] := \{1, 2, \ldots, n\}$ be a finite set of indices for any $m \in \mathbb{N}$. We say a physical ONB measurement $\widetilde{\mathcal{M}} = \{|\widetilde{\phi}_j\rangle\}_j$ on n qubits implements a logical ONB measurement $\mathcal{M} = \{|\phi_i\rangle\}_i$ on k qubits if there exists a function $\ell : [2^n] \to [2^k]$ where, for all states $|\psi\rangle$, we have:

$$\left| \langle\psi| \vdots |\phi_i\rangle \right|^2 = \sum_{j, \ell(j)=i} \left| \langle\psi| \boxed{E} \vdots |\widetilde{\phi}_j\rangle \right|^2 \tag{11.35}$$

This condition says that, for any logical state, the Born rule probabilities obtained from measuring $\mathcal{M}$ can be computed in terms of the probabilities of $\widetilde{\mathcal{M}}$ measurements. Normally the function ℓ is very simple, e.g. if $k = 1$, it could be simply computing the parity of boolean outcomes on the n physical qubits.

In Section 11.2.3, we mentioned that we usually don't actually perform the encoder map on our quantum computer, but use it purely as a mathematical object to relate the logical qubits to the physical ones. Now we can see why that is the case. If we can find encoded states, gates, and measurements, then we can simulate any logical computation on k qubits using a physical one on n qubits without ever explicitly performing E.

Suppose we are interested in the results of a logical computation involving applying a sequence of gates $U_1, \ldots, U_t$ to a state $|\psi\rangle$ and measuring in the logical ONB $\{|\phi_i\rangle\}_i$. Then we would like to sample from this distribution:

$$\mathrm{Prob}(i \mid |\psi\rangle) = \left| \langle\psi| \vdots \boxed{U_1} \vdots \boxed{U_2} \vdots \cdots \boxed{U_t} \vdots |\phi_i\rangle \right|^2$$

Then, we can translate the logical measurement into a physical measurement using (11.35):

$$\ldots = \sum_{j, \ell(j)=i} \left| \langle\psi| \vdots \boxed{U_1} \vdots \boxed{U_2} \vdots \cdots \boxed{U_t} \vdots \boxed{E} \vdots |\widetilde{\phi}_j\rangle \right|^2$$

We can keep pushing logical gates through the encoder to get physical ones:

$$... = \sum_{j,\ell(j)=i} \left| \left\langle \psi \left| \vdots \left| E \right| \vdots \left| \widetilde{U}_1 \right| \vdots \left| \widetilde{U}_2 \right| \vdots \left| \widetilde{U}_t \right| \vdots \widetilde{\phi}_j \right\rangle \right|^2$$

Finally, when we push the logical state through, the encoder is gone, and we're left with a fully encoded computation that simulates our logical computation:

$$... = \sum_{j,\ell(j)=i} \left| \left\langle \widetilde{\psi} \left| \vdots \left| \widetilde{U}_1 \right| \vdots \left| \widetilde{U}_2 \right| \vdots \left| \widetilde{U}_t \right| \vdots \widetilde{\phi}_j \right\rangle \right|^2$$

Great! We're doing everything on physical qubits encoded in our error correcting code the whole time. We only forgot one thing: we need to do error correction! That was of course the point of this whole exercise. If we are being particularly conservative, we can do one or more rounds of error correction, i.e. measuring all of the stabilisers and applying a Pauli correction, at each step of the computation:
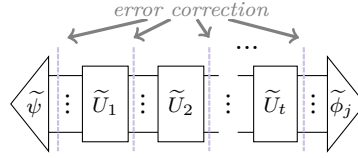
We should also try to engineer our encoded operations, and the error correction itself, in such a way that they avoid spreading errors uncontrollably. For example, if $\widetilde{U}_i$ involves multi-qubit gates, this turn a single error into multiple errors. In this section, we will see how to implement (almost) everything we need for universal computation in such a way that we can keep errors under control. This property of encoded gates is what is known as *fault tolerance*.

There are several definitions of fault tolerance in the literature, with various levels of mathematical rigour. Here's the one we will use. Note that we refer to a single collection of n physical qubits encoded in an error correcting code as a **code block**.

Definition 11.4.1 (Fault tolerance)**.** An operation $\widetilde{O}$ on a single code block is called **fault tolerant** if, whenever there are errors on at most u qubits before applying $\widetilde{O}$, and then v faults occur while performing $\widetilde{O}$, there are errors on at most $u + v$ qubits afterwards.

An example of an operation that doesn't spread errors is a **transversal gate**. For a single code block, transversal gates are simply tensor products of single-qubit operations. An example of something that does spread errors

is a multi-qubit gate involving physical qubits in the same code block. For example, a CNOT gate can spread Pauli X errors from its control to its target qubit:
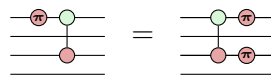
In this case, there was originally 1 error on the physical qubits, but afterwards there are 2.

Note that we were not too specific about what it means for a fault to occur during the performance of $\widetilde{O}$. What counts as a single fault depends on how we represent the operation $\widetilde{O}$ and how we will choose to model potential errors that could happen while attempting to perform $\widetilde{O}$, i.e. the **error model** we are using. By their nature, error models are always a simplification of what happens in reality. One of the simplest error models is the **phenomenological error model** with Pauli errors. In this model, we decompose $\widetilde{O}$ into basic gates and assume at each time step, a Pauli error could occur on each qubit with some fixed (hopefully small) probability p.

Consider implementing a swap gate on our physical qubits with three CNOT gates:
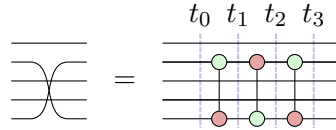
The swap gate never multiplies Pauli errors. So, if u Pauli errors occur at t_0, then only u errors will come out at t_3. However, if an error happens in the process of implementing $\widetilde{O}$, e.g. an X error on qubit 2 at t_2, then it will propagate to X errors on qubits 2 and 5 at t_3.

In some circumstances, this model is somewhat overly optimistic. For example, one would expect that most physical implementations of 2-qubit gates can actually lead to correlated errors on the qubits involved, while we just assume that an error happens on each of the qubits independently with some fixed probability. However, this will be good enough for our purposes of illustrating the basic principles of constructing fault-tolerant implementations. For more elaborate and realistic error models, we point to the References of this chapter.

We'll conclude this section by noting that Definition 11.4.1 extends to a slightly more flexible notion of "not spreading errors" when multiple code blocks are involved.

Definition 11.4.2 (Multi-block fault tolerance)**.** An operation $\widetilde{O}$ on b code

blocks is called **fault tolerant** if, whenever $u_1, \ldots, u_b$ errors only occur within each code block, and v faults occur with performing $\widetilde{O}$, then afterwards at most $\sum_i u_i + v$ errors occur in each code block.

This definition allows errors to propagate between the code blocks, as long the number of errors appearing in a *single* block after the operation don't exceed the total number of errors from before the operation. Consider for example applying a CNOT gate between the i-th qubit of one codeblock and the i-th qubit of another one:

$$\tag{11.36}$$

Then errors could indeed multiply:
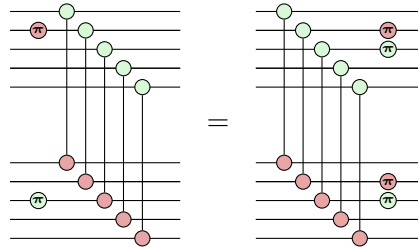
Even though there are a total of 4 errors after performing this tower of CNOT gates, there are only 2 errors in each code block, which is the same as the number of errors we started with. This is still okay for fault tolerance, as we can then perform error correction independently on each code block.

The map (11.36) is an example of a transversal, multi-block operation. We will look at these, and transversal gates in general, in the next section. After that, we will turn to the somewhat trickier issue of fault-tolerant stabiliser measurements.

We call a recipe for constructing a universal set of fault-tolerant operations in a given code or code family a **fault-tolerant scheme**. It makes intuitive sense that having a fault-tolerant scheme is A Good Thing if we want to get to the end of a computation with a relatively low probability of suffering an uncorrectable error. But we can make it more precise why this is the case by discussing the **threshold** of a fault-tolerant scheme.

While we can never push the probability of a logical error occurring all the way to zero, one could imagine having an infinite series of fault-tolerant

simulations $\{\mathrm{FT}_l(C)\}_{l \in \mathbb{N}}$ of a given circuit C, each producing a better and better approximation of a logical circuit C on physical hardware, at the cost of extra resources needed to do more and more error correction. The main example of such a series would be an infinite family of quantum error correcting codes with increasing distance, such as the surface code, and recipes for implementing a universal family of fault-tolerant operations $\widetilde{O}$ on an any code in that family.

Definition 11.4.3. The **threshold** of a fault-tolerant scheme is a probability p_{th} such that for any circuit C, if all physical operations produce a fault with probability $p < p_{th}$, the family of fault-tolerant simulations $\{\mathrm{FT}_l(C)\}_{l \in \mathbb{N}}$ has the property that for any $\epsilon > 0$, there exists some l such that the output probabilities of $\mathrm{FT}_l(C)$ are ϵ-close the those of the ideal circuit C.

The notion of a threshold is an important one, and perhaps the most important in fault-tolerant quantum computation. Performing encoded operations and error correction involves more basic quantum operations, which of course will produce more errors. However, under some critical hardware error rate, a fault-tolerant scheme will start to suppress more errors than it generates. This means that, if we can demonstrate a threshold, it is possible (at least in principle), to perform arbitrarily large quantum computations on noisy hardware.

The **threshold theorem** states that, under certain assumptions, *any* universal set of fault-tolerant operations can be turned into a family of fault-tolerant simulations with a threshold $p_{th} > 0$, and furthermore the overhead scales reasonably in the size of the circuit C and error parameter ϵ. Proving this theorem is complicated, so we will leave it to other resources that have done this in detail (see the Reference), and we will proceed now to the *practical* construction of fault-tolerant operations in a given error-correcting code.

11.4.1   Fault-tolerant computation with transversal gates

Transversal unitary gates are the easiest kind of fault-tolerant operation to understand, although for a given error correcting code, it can be highly non-trivial to characterise the set of transversal gates that can be implemented in that code.

Definition 11.4.4. A **transversal implementation** $\widetilde{U}$ of a logical unitary U on a single code block is a gate of the form $\widetilde{U} = U_1 \otimes \ldots \otimes U_n$ where the U_i are all single-qubit unitaries. A transversal implementation of a logical

unitary acting on multiple code blocks consists of n unitaries each acting only on the i-th qubit of each code block, for $i \in 1, \dots, n$.

An example of a transversal unitary on multiple code blocks is the transversal CNOT we saw in (11.36). It is easy to check that transversal gates satisfy the fault-tolerance conditions laid out in Definitions 11.4.1 and 11.4.2. So, if we can find a universal set of transversal gates, life is good! Unfortunately, there is this little nugget:

Theorem 11.4.5 (**Eastin-Knill**)**.** For any error-correcting code capable of detecting an arbitrary single-qubit error, the set of logical unitaries U with a transversal implementation $\widetilde{U}$ is finite. In particular, no non-trivial error-correcting code has a universal set of transversal gates.

The proof of this theorem uses some tricks from Lie theory which are a bit technical for our purposes, so we won't reproduce it here (see the References). Very roughly, the proof shows that the error-detection property of the code forces the group of distinct unitaries U with transversal implementations to be discrete: there is no way to "nudge" U a small distance in any direction, while keeping $\widetilde{U}$ transversal. But then the group of all logical unitaries must also be compact, and any discrete subgroup of a compact group is finite.

If that explanation didn't do much for you, don't worry. The main thing you should get is that Theorem 11.4.5 is Bad News if we were hoping to do all of our fault-tolerant computation with transversal gates. However, many codes still have lots of useful transversal gates. Also, as we'll see in Section 11.5, we can still achieve universal computation fault-tolerantly, but we'll need to go beyond transversal unitaries to find a bit of extra magic.

11.4.1.1 Transversal Clifford gates

Transversal Clifford gates are usually the easiest fault-tolerant operations to understand for stabiliser codes. In the general case, we can use stabiliser theory to find the transversal gates of a stabiliser code and compute how they act on logical qubits.

Exercise 11.10. Let E be the encoder of a stabiliser code $\mathcal{S} = \langle \vec{S}_1, \dots \vec{S}_m \rangle$ and $\widetilde{U}$ a Clifford unitary. Show that we have:

$$\vdots \boxed{U} \vdots \boxed{E} \vdots \quad = \quad \vdots \boxed{E} \vdots \boxed{\widetilde{U}} \vdots$$

for some Clifford unitary U if and only if we can push stabilisers through $\widetilde{U}$. That is, for all $\vec{S} \in \mathcal{S}$ there exists $\vec{T} \in \mathcal{S}$ such that $\vec{S}\widetilde{U} = \widetilde{U}\vec{T}$.

In particular, as soon as we have a Clifford unitary $\widetilde{U}$ that has the "stabiliser pushing" property, we can compute U just by looking at how $\widetilde{U}$ acts on the logical operators $\vec{\mathcal{X}}_i, \vec{\mathcal{Z}}_i$. We can perform all the computations involving stabiliser groups efficiently, e.g. using the symplectic matrix representation from Section 6.4.4. Hence, a naïve way to find all the transversal Clifford gates in a code is to just enumerate every local Clifford $\widetilde{U}$ and check if it sends all the generators of the stabiliser group to stabilisers.

Of course, there are exponentially many local Clifford operations, so it may not be practical to enumerate them all for large n. However, there are many results that show that some families of codes always have certain transversal Clifford gates.

One of the most well-known results of this kind says that CNOT gates are transversal for CSS codes. In its simplest version, this means that applying a CNOT gate transversally across all physical qubits of two identical codeblocks has the effect of applying a CNOT across all logical qubits. We can formalise this using the scalable notation as follows.

Theorem 11.4.6. CNOT gates are transversal for CSS codes. That is:

$$(11.37)$$

Proof   We begin with the right-hand side of (11.37) and rewrite the encoder of the second code block into Z-form:

We can then push the CNOT through the encoders using strong complementarity and the scalable ZX rules:
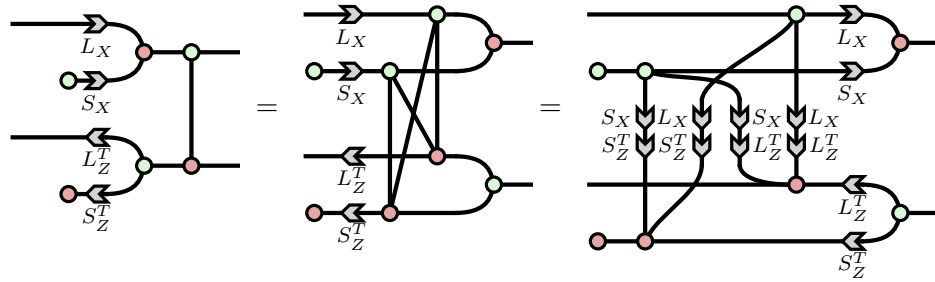
then apply equations (11.33) and (11.34):
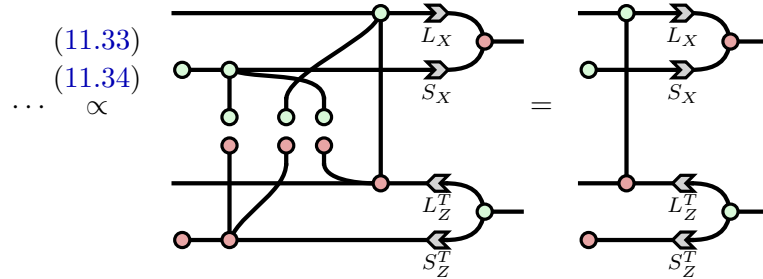
Rewriting the second encoder back into X-form yields the left-hand side of (11.37). □

All CSS codes have transversal CNOT gates. However, it might not be so convenient to implement a CNOT between each qubit in the code block. For example, as the surface code is a CSS code, we know it has a transversal CNOT, which looks like this:
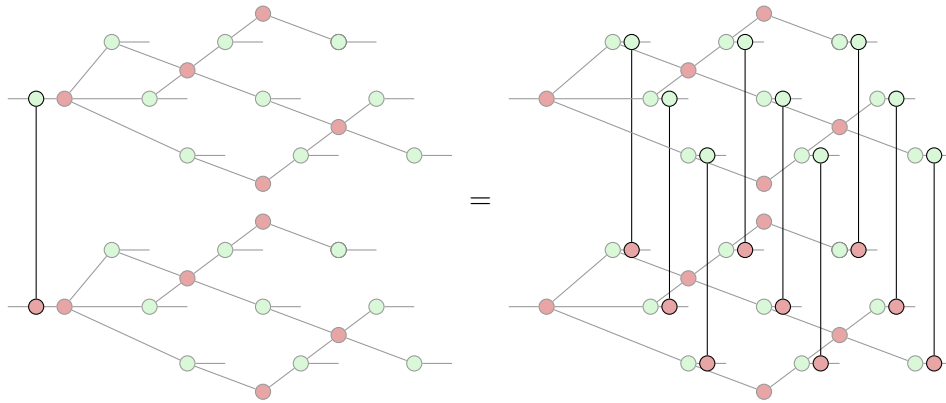
Note however, that implementing it would ruin the nice 2D structure which makes the surface code so attractive for some hardware platforms. We'll see in Section 11.4.3 that there is another way to implement 2-qubit gates in the surface code and friends without destroying planarity.

Not all CSS codes admit transversal H gates, but quite a few do. To see what conditions we need, let's start with a transversal H and see what happens when we push it into the encoder. First, we can introduce a scalable H gate to mean n copies of the H gate:

Then the usual colour change rules work just like they would for the normal H gate, but we also get an additional rule. Using (11.26), we can push a scalable H gate through an arrow as follows:

$$
\text{—□⇒—}_{A} \quad = \quad \text{—⇐□—}_{A^T} \tag{11.38}
$$

Using this new rule, we can introduce H gates on all of the physical qubits of the encoder in Z-form and push it to the left:

$$\tag{11.39}$$

The RHS almost looks like a layer of Hadamard gates followed by the X-form of the encoder, but not quite. The X-form should be labelled by L_X and S_X, not L_Z and S_Z. We could therefore consider codes where $L_X = L_Z$ and $S_X = S_Z$ and we'd be done. However, it turns out that is stricter than we need, so we'll consider codes where just the X-stabilisers and Z-stabilisers are the same.

Definition 11.4.7. A CSS code is **self-dual** if $S_X = S_Z$.

Examples of self-dual CSS codes are the $[\![4, 2, 2]\!]$ code from Example 11.2.5 and the Steane code.

Continuing from (11.39) with a self-dual code, we can see that we almost get to the form we need:

$$\tag{11.40}$$

but the logical operators on the RHS are still the wrong type to get an encoder in X-form.

The trick now is to realise that even though the X-logical and Z-logical operators might still be different in a self-dual CSS code, they always generate the same subspace together with the stabilisers. Hence, we can prove the following proposition.

Proposition 11.4.8. For any self-dual CSS code, there exist matrices M, N such that $L_Z = L_X M + S_X N$.

Proof Suppose $S_Z = S_X$ consists of m independent stabilisers, then the whole CSS code has $2m$ stabilisers. It must then have $n - 2m$ X-logical operators. Hence, the columns of L_X and S_X together consist of $n - 2m + m = n - m$ independent vectors in $\text{cols}(S_Z)^\perp$. Since $\text{cols}(S_Z)$ is m-dimensional, that means they span the whole space. But then, $S_Z = S_X$, so each column of L_Z is in $\text{cols}(S_X)^\perp = \text{cols}(S_Z)^\perp$. Hence, we can write each column of L_Z as a linear combination of columns from L_X and S_X, i.e. we can find matrices M, N such that $L_Z = L_X M + S_X N$.   □

Exercise 11.11. Show using the scalable rules from Section 10.2 and Proposition 11.4.8 that, for a self-dual code, we have for some matrix M:

Theorem 11.4.9. H gates are transversal for self-dual CSS codes. That is, for any self-dual CSS code, there exists a CNOT circuit with parity matrix M such that:

                                                   (11.41)

Proof Starting from the RHS of (11.41), we can switch to the Z-form of the encoder and apply the derivation in (11.40) and Exercise 11.11:

M is the parity matrix of a CNOT circuit if and only if it is invertible. This follows immediately from the fact that the map above is an isometry, and hence in particular is non-singular.   □

This means applying a transversal Hadamard at the physical level is not necessarily the same thing as applying a transversal Hadamard at the logical level. However, it always does something non-trivial to our logical qubits, consisting of a layer of Hadamard gates followed by some (possibly trivial) CNOT circuit. In the case the Steane code, the logical operators are identical, so applying H to every physical qubit is the same as applying H to the single logical qubit:
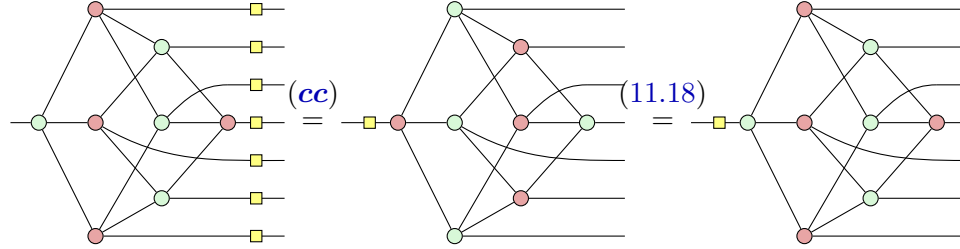
However, the $[\![4, 2, 2]\!]$ code from Example 11.3.4 has X-logical operators and Z-logical operators that differ by a logical swap:
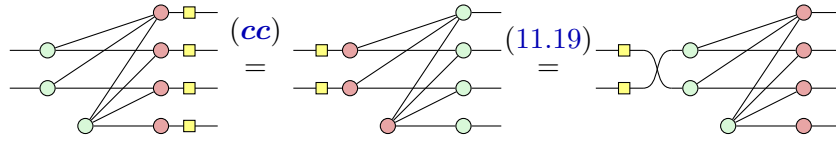
In this case, the parity matrix from Theorem 11.4.9 is

$$M := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

corresponding to a swap gate (or equivalently, 3 CNOTs).

Note that the surface code *almost* has a transversal H gate. When we apply H everywhere on the X-form of the encoder, we get the following:
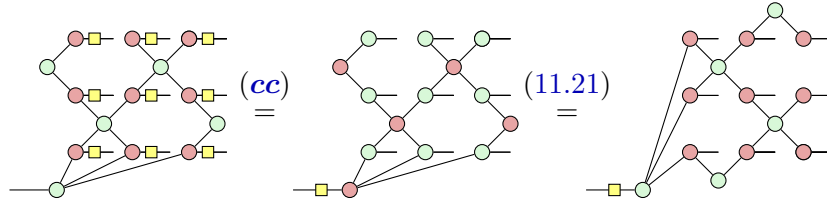
We almost get back to where we started, except the encoder has "rotated 90°", which follows from applying the colour-reverse of equation (11.21). Technically this is a different error correcting code. But of course this new code has all the same properties as before, since it is just a permutation of the physical qubits. As long as we keep track of where the qubits are, this is "close enough" to a transversal H for many purposes.

We could at this point complete the Clifford story and give the conditions for a CSS code to have transversal S gates. Rather than working this out explicitly, we will go first to the harder case of transversal T gates, which as we will explain in Remark 11.4.13, generalises readily to a characterisation of transversal $Z[\frac{\pi}{2^{\ell-1}}]$ gates for all $\ell \geq 1$.

11.4.1.2 Transversal T gates

We now turn to the somewhat thornier question of when stabiliser codes admit transversal *non-Clifford* gates. For the sake of simplicity, we will start with T gates, but in fact the story will be much the same in the next section when we talk about all diagonal gates on the third level of the Clifford hierarchy. Once we pass to non-Clifford gates, we can't expect to be able to just conjugate each of the generators of the stabiliser group and get Paulis again. Hence, we no longer have such an easy way to check if a given operation on physical qubits preserves the codespace.

Anyway, we still have the ZX-calculus at our disposal, so let's be brave. Just like we did with transversal Hadamards, we can start with a transversal application of T gates on the physical qubits and try to push it through the encoder of a CSS code. If it comes out as some non-trivial unitary on the k logical qubits, we're golden. Since we've left Clifford-land, we don't expect life to be that easy, but we will just start and see where we get stuck. We begin by applying strong complementarity and the scalable rules:

$$(11.42)$$

This gets us pretty far, but we still have that pesky S_X connecting our phase gadgets to some wires that don't correspond to logical qubits. In order to preserve the codespace, we want to end up with some phase gadgets just supported on the logical qubits. Just like in the case of Hadamards, this could act differently on the logical space from the physical space, so we don't know *a priori* what this action is.

We will start with the simplest case, where $T^{\otimes n}$ on the physical qubits acts as $(T^\dagger)^{\otimes k}$ on the logical qubits. That is, we want to satisfy this equation:

$$(11.43)$$

The reason it is easier to use $T^\dagger$ and not T is we can now move everything

to one side and we'll only have $\frac{\pi}{4}$ phase gadgets:
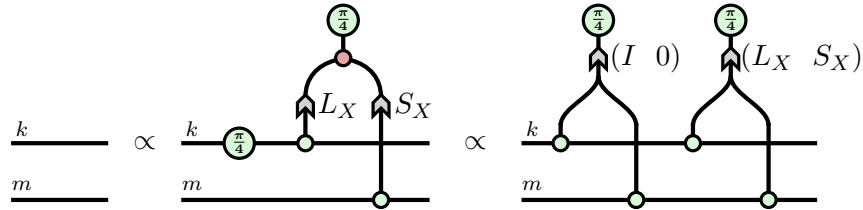
Now, we can "zip up" this equation into a single matrix over a register of $k + m$ qubits:

$$\tag{11.44}$$

Thanks to Section 10.3.1, we know that a configuration of $\frac{\pi}{4}$ phase gadgets equals the identity if and only if its associated matrix is strongly 3-even. Hence, we are ready to characterise when CSS codes have transversal T gates.

Definition 11.4.10. A CSS code (L_X, S_X) is called **triorthogonal** if the following matrix is 3-even:

$$M = \begin{pmatrix} I & 0 \\ L_X & S_X \end{pmatrix} \tag{11.45}$$

and a code is called **strongly triorthogonal** if the matrix above is strongly 3-even.

Theorem 11.4.11. A CSS code (L_X, S_X) admits a transversal T in the sense that:

$$\tag{11.46}$$

if and only if it is strongly triorthogonal.

Proof First assume the CSS code is strongly triorthogonal. From the calculations above, this is equivalent to equation (11.43). Hence, following calculation (11.42), we have:
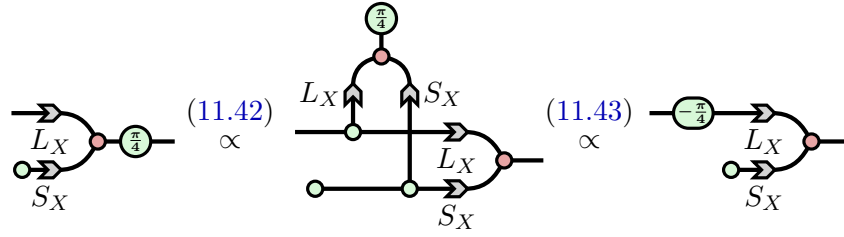
Conversely, if (L_X, S_X) satisfies Eq. (11.46), then:
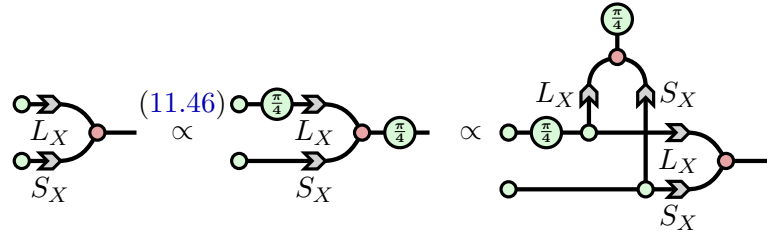
We can then apply Eq. (11.32) to cancel most of the encoder from both sides:
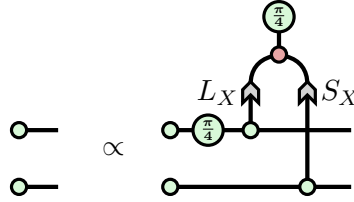
Note that the RHS consist of a diagonal unitary applied to $|+\dots+\rangle$, but diagonal unitaries send the state $|+\dots+\rangle$ to itself if and only if they are the identity. Hence, we can conclude M from Definition 11.4.10 is strongly 3-even and hence (L_X, S_X) is strongly triorthogonal. $\square$

Exercise 11.12. Show that triorthogonal codes implement a transversal T gate, possibly up to a diagonal Clifford map on the outputs. That is, for some A:
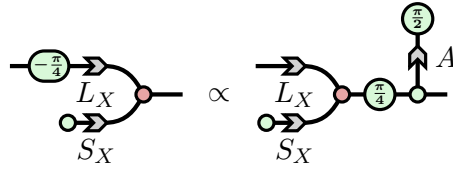
Hint: Use equation (11.31) *to move a diagonal Clifford map on the logical side to the physical side.*

Example 11.4.12. The degree-1 monomial $x_1 \in \mathrm{RM}(1, 5)$ gives the follow-

ing strongly 3-even matrix:

$$
\begin{pmatrix}
\begin{array}{c|cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\end{pmatrix}^{T}
$$

The matrices (L_X, S_X) on the right define a $[\![15, 1, 3]\!]$ quantum Reed-Muller code. This is, by definition, a strongly triorthogonal code, so applying $T^{\otimes 15}$ on the physical qubits results in a logical $T^\dagger$. This property will be used to perform a protocol called **magic state distillation** in Section 11.5.

Remark 11.4.13. Note that almost the exact same argument as above can be given for transversal $Z(\frac{\pi}{2^{\ell-1}})$ gates for any $\ell > 1$. We can define strongly ℓ-even matrices as those whose columns sum to $0 \mod 2^\ell$, pairs of columns sum to $0 \mod 2^{\ell-1}$, and so on up to groups of ℓ columns summing to $0 \mod 2$. Following essentially the same reasoning as Section 10.3.1, we can conclude that

for any strongly ℓ-even matrix M. Hence, any CSS code whose matrix

$$
M = \begin{pmatrix} I & 0 \\ L_X & S_X \end{pmatrix}
$$

is strongly ℓ-even has a transversal $Z(\frac{\pi}{2^{\ell-1}})$ gate. Furthermore, we can generalise Exercise 11.12 to show that if M is only ℓ-even rather than strongly ℓ-even, it is transversal up to a $2^{\ell-2}$ phase gadget. In particular, if the matrix M is 2-even, the code has transversal S gates up to Paulis, and hence it can implement a logical S transversally.

11.4.1.3 Transversal Clifford hierarchy gates

We can generalise from characterising transversal T or $T^\dagger$ gates to transversal implementations of more general unitaries in the third level of the Clifford hierarchy. Recall from Section 6.5 that the third level of the Clifford hierarchy $\mathcal{C}_3$ consists of all the unitary maps U with the property that, for all Pauli strings $\vec{P}$, $U\vec{P}U^\dagger$ is Clifford.

While it seems to be a difficult problem to characterise *all* transversal gates in $\mathcal{C}_3$, it is much easier if we just focus on the diagonal unitaries $\mathcal{D}_3 \subset \mathcal{C}_3$.

As noted also in Section 6.5, $\mathcal{D}_3$ forms a group which is generated by the $\frac{\pi}{4}$ phase gadgets. Consequently, an arbitrary unitary in $\mathcal{D}_3$ can be written as

$$D_M \;:=\; \underset{}{\overset{\frac{\pi}{4}}{\big\uparrow M}}$$

(11.47)

for some boolean matrix M.

We can also capture what it means to be a transversal unitary on n qubits in $\mathcal{D}_3$. These consist precisely of some power T^p on each qubit. Hence, a transversal gate in $\mathcal{D}_3$ is of the form D_P for some matrix P where each row contains exactly one 1. Such a matrix represents a T gate on the j-th qubit by containing the j-th unit vector as a row. We can then represent higher powers T^p as repeated rows.

Example 11.4.14. The following is a transversal application of powers of T and its associated P-matrix:

$$D_P \;=\; \begin{array}{c} \frac{\pi}{4} \\ \hline \\ \frac{3\pi}{4} \\ \frac{\pi}{2} \end{array} \qquad \text{for} \qquad P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

From here, the story goes much like in the previous section. Rather than focusing just on $T^{\otimes n}$, we can start with any transversal $\mathcal{D}_3$ unitary and try to push it past the encoder:

(11.48)

Now, in order to preserve the codespace, the resulting diagonal unitary should be supported only on the logical qubits. That is, for some H, we

should have:
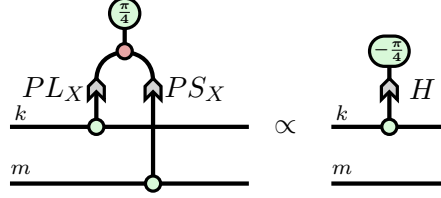
As in the previous section, it is convenient to use $-\frac{\pi}{4}$ phase gadgets on the right-hand side, but note that, since H can now be arbitrary, this generates the same set of unitaries as $\frac{\pi}{4}$ phase gadgets. Moving everything to one side, we get:
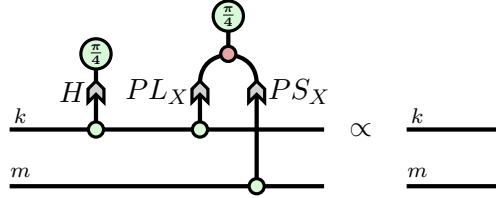
...or equivalently, in "zipped up" form:

$$\underline{\quad k+m \quad} \quad \propto \quad \overset{\frac{\pi}{4}}{\underset{k+m}{\Big\Uparrow}} \begin{pmatrix} H & 0 \\ PL_X & PS_X \end{pmatrix} \tag{11.49}$$

Hence, we can now state a generalised version of Theorem 11.4.11.

Theorem 11.4.15. A CSS code with X-logical operators and X-stabilisers L_X and S_X admits a transversal implementation of a logical gate

$$D_H^\dagger \;=\; \overset{-\frac{\pi}{4}}{\underset{}{\Big\Uparrow}} H$$

if and only if there exists a matrix P whose rows are unit vectors, such that the matrix

$$M = \left(\begin{array}{c|c} H & 0 \\ \hline PL & PS \end{array} \right) \tag{11.50}$$

is strongly 3-even.

Exercise 11.13. Generalise the proof of Theorem 11.4.11 to a proof of Theorem 11.4.15.

Example 11.4.16. The constant polynomial $1 \in \mathrm{RM}(0,4)$ gives us a

strongly 3-even matrix whose columns are all the 4-bitstrings. If we partition the matrix as follows:

$$\left(\begin{array}{cccccccc|cccccccc} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} \right)^T \tag{11.51}$$

then this gives a matrix of the form (11.50) with $P = \mathrm{id}$, with (L_X, S_X) defining the X-logical operators and X-stabilisers of an $[\![8,3,2]\!]$ CSS code, which is sometimes called the "smallest interesting colour code" (see the References for why). It has three X-logical operators and 1 X-stabiliser:

$$\vec{\mathcal{X}_1} := I \otimes X \otimes I \otimes X \otimes I \otimes X \otimes I \otimes X$$
$$\vec{\mathcal{X}_2} := I \otimes I \otimes X \otimes X \otimes I \otimes I \otimes X \otimes X$$
$$\vec{\mathcal{X}_3} := I \otimes I \otimes I \otimes I \otimes X \otimes X \otimes X \otimes X$$
$$\vec{X_1} := X \otimes X \otimes X \otimes X \otimes X \otimes X \otimes X \otimes X$$

We can draw this encoder as a cube, with the X-logical operators connecting to 3 adjacent faces and the X-stabiliser connecting to everything:
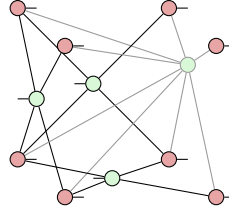
In the upper left corner of the matrix (11.51), we see the phase gadgets of a CCZ gate as in equation (9.6), hence this code admits a logical $\mathrm{CCZ}^\dagger = \mathrm{CCZ}$, implemented via 8 physical T gates:
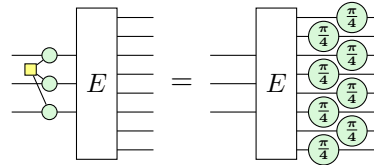
Remark 11.4.17. While Theorem 11.4.15 gives a complete characterisation for the transversal $\mathcal{D}_3$ gates in a CSS code, it is not obvious from the statement whether it can be used to efficiently find such gates. However, it turns out that this is possible. In fact, there are three related problems: (1) for fixed logical $D_H^\dagger$ find transversal gates D_P, (2) for fixed D_P find $D_H^\dagger$, and (3) compute a generating set of all logical gates and their associated

transversal implementations. All three of these problems can be posed as a system of linear equations over the ring $\mathbb{Z}_8$, and can be solved in polynomial time using a generalisation of Gaussian elimination that works over more general kinds of rings. We will say a bit more about this in the References.

11.4.2   Fault-tolerant Pauli measurements

In order to do error correction, we should be able to do stabiliser measurements on our physical qubits. Since any non-trivial stabiliser generator will have support on multiple qubits, these will in general be entangling operations involving multiple physical qubits, so it is not immediately obvious we can perform such measurements fault-tolerantly.

Before we think about a fault-tolerant implementation of a Pauli measurement, we should think about how we would even implement a generic Pauli measurement in the first place. For the first part of this section, we will focus on measuring $Z \otimes \ldots \otimes Z$, but everything we say below will apply to measuring any Pauli string just by conjugating by the appropriate local Cliffords:
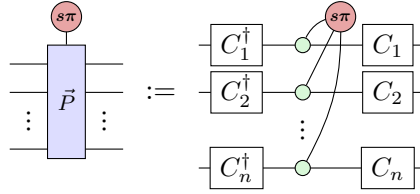
It could be that our hardware natively implements multi-qubit Z measurements, however at the time of writing, this is not possible (or at least quite challenging) on most quantum hardware platforms. Even if it were possible to implement high-weight Pauli measurements on physical qubits, we might not want to, as it could introduce errors on many qubits at once. Hence, the first thing we should do is decompose this measurement into more basic operations like basic gates and single-qubit demolition measurements.

This is particularly nice in ZX, because it just amounts to coming up with a good way to unfuse the big spider in the projector. One such unfusion is:

$$(11.52)$$

We can now interpret the right-hand side as preparing an ancilla in the $|0\rangle$

state, applying a series of CNOT gates, then measuring the ancilla in the Z basis.

In terms of not spreading errors on the physical qubits, this is actually pretty good. Z-errors will just commute through, whereas X-errors will copy on to the ancilla and flip the measurement outcome:
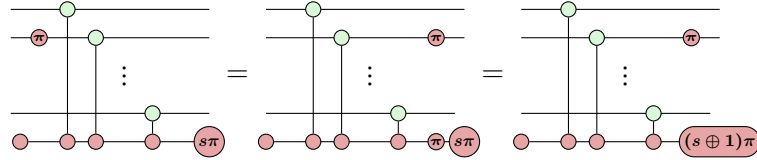
...which is what they are supposed to do!

The problem comes from errors that might occur on the ancilla qubit while implementing the measurement. If an X error occurs on the ancilla, the whole measurement will report the wrong outcome:
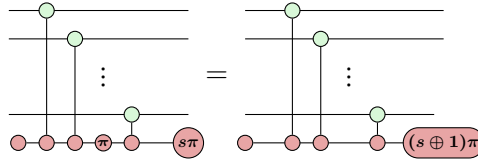
This is bad of course, because doing error correction depends on being able to report the results of syndrome measurements reliably. However, this kind of error is not fatal. As long as the error rate is not too high, we can increase the reliability of our stabiliser measurements just by repeating the measurement multiple times and reporting whatever outcome we got the majority of the time.

The worse thing that can happen is Z-errors, i.e. "phase flips" occurring on the ancilla, because these can propagate out to errors on data qubits:
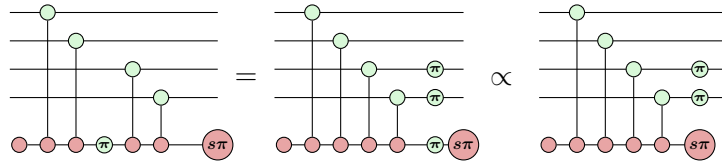
Worse still, it doesn't change the measurement outcome, so this measurement has introduced a new, undetected multi-qubit error.

We hence see that in the naïve implementation of a multi-qubit Z measurement, a single fault causes multiple errors on the data qubits. Hence, it doesn't satisfy the fault-tolerant criterion from Definition 11.4.1. To solve this problem, we need to come up with a "better unfusion" of this Pauli measurement. One pretty good solution is, rather than unfusing the X-spider

sequentially, we unfuse it in parallel:

$$(11.53)$$

where $s_1 \oplus s_2 \oplus s_3 \oplus s_4 = s$.

Now, rather than having a single ancilla prepared in the $|0\rangle$ state, we have w ancillae for a Pauli measurement of weight w prepared in a generalised GHZ state, with respect to the X basis: $|+\rangle^{\otimes w} + |-\rangle^{\otimes w}$. Especially in the context of fault-tolerance, this is often called a **cat state**. Then, we perform a CNOT between the i-th data qubit and the i-th qubit of the cat state and measure each of the ancilla qubits in the Z basis. The XOR of all the measurement outcomes then gives the result of the syndome measurement. This implementation of a multi-qubit measurement is called a **Shor fault-tolerant measurement**. For the sake of concreteness, we have shown the 4-qubit Shor measurement, but this generalises in the obvious way to Pauli measurements of higher weight.
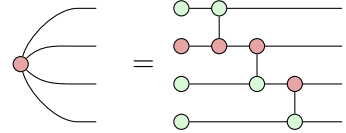
There are a few things to say about Shor measurements. First, notice how in the process of unfusing spiders, we can turn one measurement into multiple measurements. Second, we have now fixed the problem that our naïve implementation had: single Z errors in the RHS of (11.53) now result only in single errors on the data qubits, e.g.
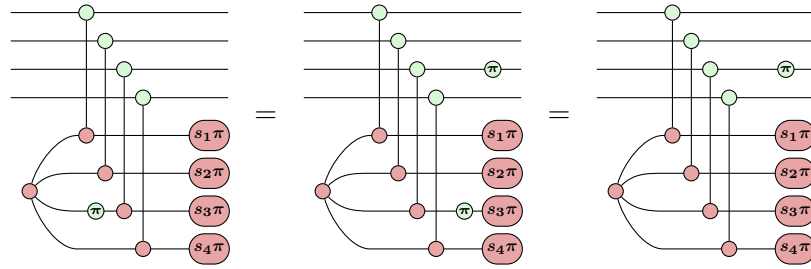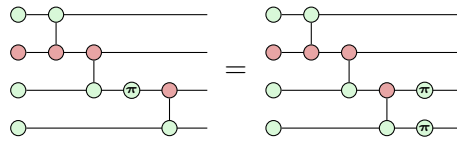
However, this doesn't fully solve the problem yet: it has shifted the difficulty of fault-tolerantly measuring $Z \otimes \ldots \otimes Z$ to fault-tolerantly preparing cat states.

If we assume that our basic operations are single-qubit preparations and measurements, as well as basic 2-qubit gates like CNOT, then we can prepare

cat states using some ladder of CNOTs like this:
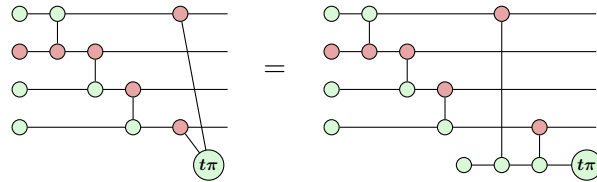
(11.54)

There are many basic circuits that can prepare a cat state, but they all have one thing in common: there is always some location where a single error in the preparation can propagate out to multiple errors on the cat state, and hence the measurement (11.53). For example:

To solve this issue, we can perform some rudimentary error detection or correction on the cat state. The first thing to note is that, because of the orientation of CNOT gates, X errors will only possibly mess up the measurement outcome, but never otherwise effect the data qubits. As mentioned before, we can mitigate this by simply repeating the measurement and taking the outcome we get the majority of the time. Hence, we can focus just on Z errors.

For a cat state prepared as in (11.54), the only place a single Z error during preparation can result in 2 errors on the cat state is the one shown above. We can therefore detect this error by measuring any stabiliser of the cat state that anti-commutes with this error. $X \otimes I \otimes I \otimes X$ will work:

If we get outcome $t = 1$, we can just throw out the cat state and try again. This is called a **repeat-until-success** strategy for state preparation.

But then, what if this nested error-detection measurement propagates errors? Do we need to do error-detection on it? Is it Turtles All the Way Down? Thankfully no. Due to the direction of the CNOT gates, there is no way for this nested measurement to propagate additional Z errors into the cat state. This nested measurement could indeed cause more bit errors into the cat state, but as in the case of our naïve measurement, bit errors on the ancillae are not a bit deal. They can lead to an erroneous syndrome, which

can be mitigated by repeated measurements, but crucially they won't cause any additional errors on the data qubits.

There exist several refinements to the idea behind Shor measurements, which consist of preparing (possibly elaborate) ancilla states, performing some transversal CNOTs and doing single-qubit measurements. One that is particularly nice to analyse in the scalable ZX-calculus is the **Steane fault-tolerant measurement** protocol. This protocol works for any CSS code, and allows one to extract syndrome information for all the stabilisers of a single kind (X or Z) at once.

First, it relies on being able to reliably prepare the encoded $|0\ldots0\rangle$ and $|+\ldots+\rangle$ states associated with a CSS code. We can see what these look like by plugging the all-0 and all-plus logical states into the CSS encoder:

$$\tag{11.55}$$

$$\tag{11.56}$$

Now, suppose we start with an ancilla system prepared in the encoded $|0\ldots0\rangle$ state, perform a transversal CNOT between the ancilla and a block of our CSS code, then measure all the qubits of the ancilla system. Here is what we'll get:

$$\tag{11.57}$$

We can reduce this further once we know how the arrow acts on the $\vec{x} \cdot \pi$ effect corresponding to our measurement outcome $\vec{x}$ arising from doing a Z-measurement on all the physical qubits. First, note that an X-spider labelled by $\vec{x} \cdot \pi$ corresponds to the computational basis state $|\vec{x}\rangle$. We already know that map described by parity matrix A sends a state $|\vec{x}\rangle$ to the state $|A\vec{x}\rangle$. Hence:

Applying the colour-change rule for the arrow (11.38), we get:

Continuing from (11.57), we have:
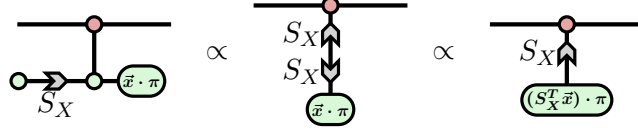
We now have a protocol for measuring all of the X-stabilisers at once. We apply the fault-tolerant circuit on the left-hand side above, obtain an outcome $\vec{x}$, then compute the X part of the error syndrome as $\vec{s} = S_X^T \vec{x}$.

Similarly, we can measure the Z part of the syndrome, by reversing all the colours:
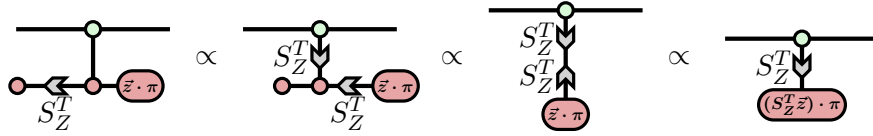
Thus we obtain the Z part of the error syndrome as $\vec{t} = S_Z^T \vec{z}$. Performing these two protocols in sequence therefore gives us a full round of syndrome extraction:
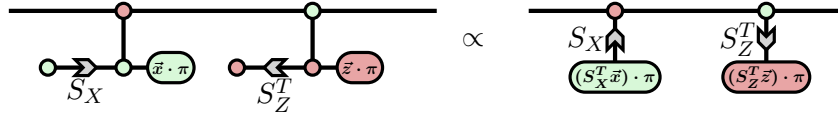
Assuming we have fault-tolerant protocols to prepare the two ancilla states, everything else in sight is fault-tolerant. There are a handful of ways to do this. For example, to prepare the encoded all-0 state, one can start with $|0\rangle^{\otimes n}$ and do several rounds of S_X measurements using some other fault-tolerant protocol, such as the Shor method. One could also attempt to detect errors that occured during preparation with an extra layer of error detection or do some sort of "distillation" procedure akin to the magic state distillation we will discuss in Section 11.5.

One limitation of the Steane method is it only works for CSS codes. The following exercise describes a procedure that works for all stabiliser codes.

Exercise* 11.14. Consider the following "logical Bell state", prepared on two blocks of an $[\![n, k, d]\!]$ stabiliser code with encoder E:
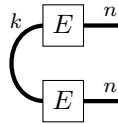
The **Knill fault-tolerant measurement** protocol measures the full error syndrome by performing a multi-qubit Bell meausurement between our data and one code block of this state:
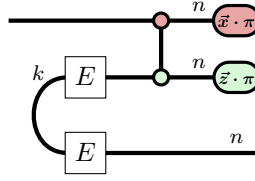
Show that this acts the same on the quantum state as measuring all of the stabilisers, and show how the syndrome can be computed from the measurement outcome $(\vec{x}, \vec{z})$.

11.4.3   Lattice surgery

As we mentioned in Section 11.4.1.1, the transversal CNOT gate might not be the most convenient way to implement multi-qubit operations in the surface code, because it breaks the 2D planar structure. It would be nice if there was a way to implement multi-qubit operations between neighbouring patches of surface code that only touch the boundaries of the code patches, hence not requiring lots of non-local gates, which might be hard to implement on platforms where qubits are embedded in the plane.

This is where lattice surgery comes in. This is a particular technique for implementing multi-qubit operations in the surface code (or CSS codes with similar structure) fault-tolerantly. Unlike the transversal gates we considered before, these operations are not unitaries.

The first class of operation splits a logical qubit into two, and comes in two varieties:

As these are both isometries, they can be performed deterministically. However, the dual operation that merges two logical qubits into one is non-deterministic:

In other words, each of these operations is a degenerate measurement that projects the 4D space of two logical qubits onto a 2D space, which we can then treat as a single logical qubit.

These operations can be combined to produce multi-qubit operations such as a CNOT on the logical level:
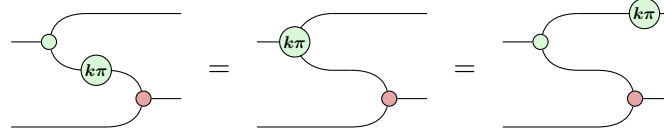
up to a possible Pauli error, which can be corrected for in subsequent operations. If one is additionally able to prepare single logical qubits in a handful of fixed states, and can use merge operations to obtain arbitrary single-qubit gates, and hence a universal model of computation.

This explains how lattice surgery works on the logical level. To explain what is happening at the physical level, we can push these operations through the encoder. However, unlike previous examples, these operations can actually change the code we are using to encode our logical qubits. The surface code works for any $d \times e$ grid of qubits, and has distance $\min(d, e)$.

To explain the split and merge operations, we will use not one encoder but a whole family of encoders of the form $E_{d \times e}$ which embed a single logical qubit into a grid of the appropriate size. Then, the physical operations operations SPLIT and $\{\text{MERGE}_k\}_{k=0,1}$ should commute with this family of encoders as follows:
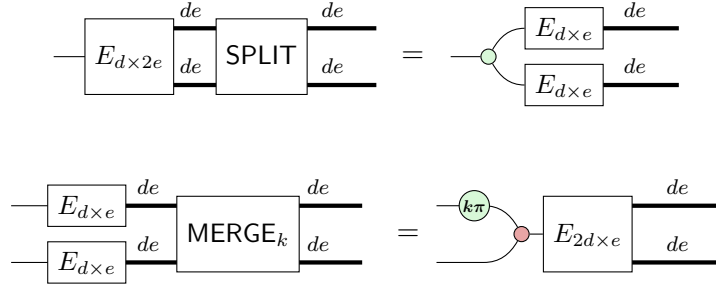
We'll demonstrate these operations concretely on 3×3 surface code patches, but in fact the same derivation will work for surface code patches of any size. Let's start with the Z-split, which is performed on a $d \times 2e$ patch of surface code by performing $X \otimes X$ measurements down the e-th column as if this were the rightmost X boundary of a $d \times e$ surface code patch. This will have the overall effect of splitting the patch in twain.

For this derivation, it will be most convenient to use the X-form of the encoder. To perform the split itself, we do $X \otimes X$ measurements down the 3rd column as if this were the right boundary of a 3×3 surface code patch. In this case, there is only one XX measurement to do. We can then use the π-copy rule to push the $j\pi$ phase coming from the measurement outcome on to the outputs:
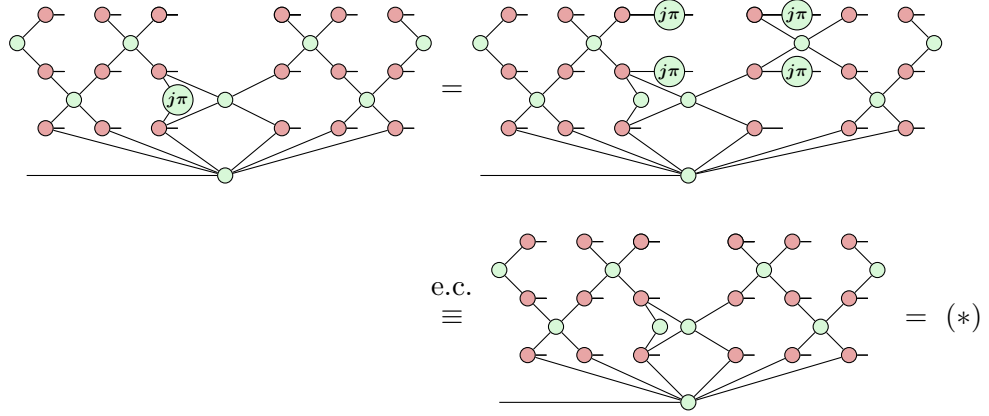
Note we write $\overset{\text{e.c.}}{\equiv}$ to mean the two diagrams are equal up to "error correction", i.e. Pauli operators applied just on the outputs. These can be treated as errors on the physical qubits and corrected later, so we will disregard them in our calculation.

Using the complementarity rule (c), we see that the existing $X^{\otimes 4}$ stabiliser becomes a pair of $X^{\otimes 2}$ stabilisers. This can then be translated into a Z-copy followed by two encoders by unfusing the bottom spider:
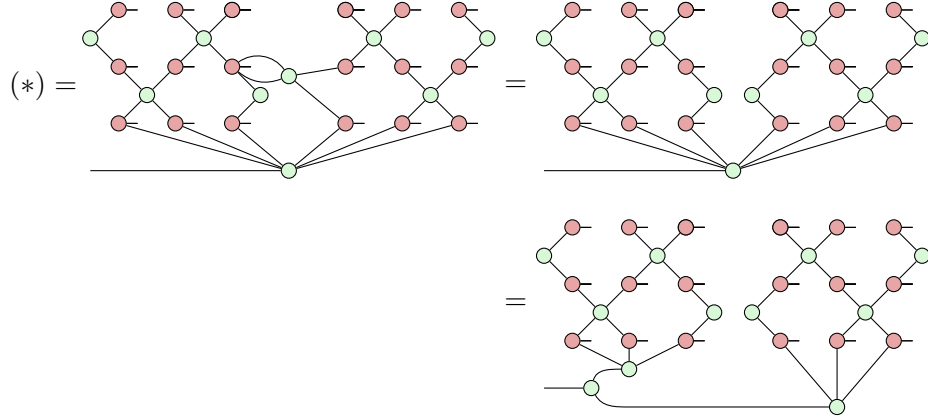
Next we do an X-merge by performing X stabiliser measurements along the boundary between two vertically-stacked surface code patches, as if these were stabilisers of one big 6×3 patch. We can eliminate the π phases from the encoder by error correction, but this time we pick up a phase of $k\pi$ where $k = j_1 \oplus j_2$ on the input (or more generally $k = j_1 \oplus \ldots \oplus j_q$ for bigger patches). We can then use the "deformation" trick from equation (11.22) to move the two logical operators on top of each other and apply strong

complementarity:
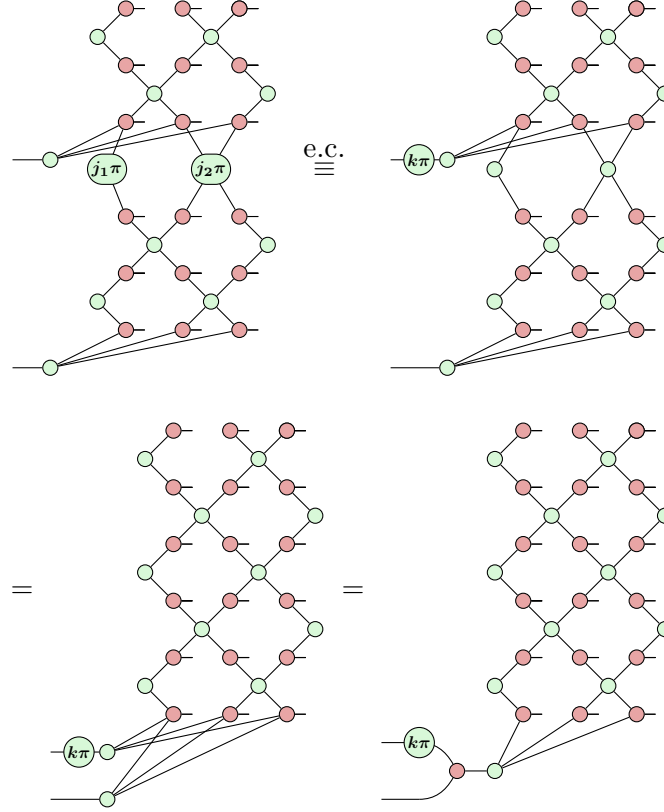
Note that the applications of the spider law, complementarity, strong complementarity, and π-copy rules in these two derivations extend naturally to larger surface code sizes. Also note that reversing the colours of these two derivations and rotating 90° gives recipes for remaining two operations of X-split and Z-merge, using the Z-representation of the surface code embedding rather than the X-representation.

This gives us a nice 2D way to build CNOT gates, as well as more general multi-qubit operations described by phase-free ZX diagrams, using just local measurements in the surface code. We already noted in Section 11.4.1.1 that we nearly have transversal Hadamards in the surface code, as long as we are willing to account for the fact that our surface might get rotated 90°. If we only had T gates, we would have a universal set of gates which can be conveniently performed on surface-code-encoded qubits.

We can check that the surface code is not a triorthogonal code, so it definitely doesn't have a transversal T gate. More generally, due to the Eastin-Knill theorem 11.4.5, we know it doesn't have any universal set of

transversal one-qubit gates. So, we need a different idea. This is where magic
state distillation comes in.

11.5   Magic state distillation

We saw all the way back in Section 3.3.1 that if we have access to states of
the form:

$$|T\rangle := \frac{\pi}{4}\!\!-\!\!\text{—}$$

then we can do magic state injection using just CNOT, S, and a Pauli
measurement:

$$\cdots = \cdots \quad (sp) \qquad \cdots \quad (\pi) \qquad \cdots \quad (sp) \qquad \cdots \quad (sp) \qquad -\frac{\pi}{4}-$$

In fact, with lattice surgery in the surface code, we can do this even more
directly, since we can just MERGE the magic state right in:

$$\cdots \quad \propto \quad -\boxed{(-1)^a\frac{\pi}{4}}-\boxed{a\frac{\pi}{2}}- \quad = \quad -\boxed{\frac{\pi}{4}}-$$

In the case of the surface code, we don't have a transversal S gate to perform
the classical correction. However, if we do state injection for S we only need
to do a Pauli correction:

$$\cdots \quad \propto \quad -\boxed{(-1)^a\frac{\pi}{2}}-\boxed{a\pi}- \quad = \quad -\boxed{\frac{\pi}{2}}-$$

Paulis are just logical operators, so they are always transversal in any error
correcting code. Even better, if we have an $|S\rangle$ state and access to H and
CNOT, we can use it to catalyse as many S gates as we want, following
Section 10.5:

$$\cdots = \cdots \quad (10.48) \quad -\boxed{\frac{\pi}{2}}- \\ \propto \qquad -\boxed{\frac{\pi}{2}}-$$

Thus, all we need to boost the surface code (and actually quite a wide
family of stabiliser codes) to computational universality is access to one $|S\rangle$
state and lots of $|T\rangle$ states. In fact, as soon as we have $|T\rangle$ states, we can
build $|S\rangle$ states, at least probabilistically, so all we really need is enough $|T\rangle$
states. That is what makes these states so magic.

We found a code with a transversal T gate (Example 11.4.12) and one
with a transversal CCZ (Example 11.4.16), but these codes are actually not
that good. They are not high distance, and they can't perform some other

useful gates transversally. However, even though these codes are not any good for doing *all* of our quantum computation, they can help us prepare magic states $|T\rangle$ (and later $|CCZ\rangle$) using a technique called **magic state distillation**.

Magic state distillation works by taking many noisy copies of a state and turning them into one less noisy one. To do this, we first form the encoded $|+\rangle$ state, then apply a noisy T gate to each of our qubits using a bunch of noisy $|T\rangle$ states and magic state injection, then decode back to a 1-qubit state to produce a less noisy $|T\rangle$ magic state.

To convince ourselves this could work, let's start with an idealised case where we want to produce a $|T\rangle$ magic state on a single qubit and we have access to two things:

1. perfect, noiseless Clifford operations, and
2. a procedure that produces a "good" magic state $|T\rangle$ with probability $1 - p_e$ and an erroneous magic state $Z|T\rangle$ with probability p_e.

We'll see how to get something approximating these assumptions on our actual hardware later, but this simple case will be enough to get the main idea.

Now, let's start with an $S|+\rangle$ state, which we prepare using our "perfect" Clifford operations, and encode it with some error correcting code with a transversal T gate, such as the $[\![15, 1, 3]\!]$ code from Example 11.4.12. We can then do magic state injections on all 15 of the physical qubits using 15 (possibly erroneous) magic states. We can then decode back to a single qubit using the decoder described in Section 11.2.4, post-selected onto syndrome $\vec{0}$:

$$\tag{11.58}$$

If no errors occurred in this process, we will indeed always get a syndrome $\vec{0}$ which is what we want. If instead we get a non-zero syndrome $\vec{s} \neq \vec{0}$, then we throw away the resulting state away and try again. Note that since we are assuming that all Clifford operations are perfect and noiseless, that the only way we could get a non-zero error is because of some fault in the $|T\rangle$ states. Hence if we indeed got a syndrome of $\vec{0}$, then the resulting 'distilled' $|T\rangle$ state only contains an error if the noisy $|T\rangle$ introduced some undetected error. Since the code has distance three, at least three magic states needed to

get an error. If we assume the errors occur independently, this probability is at most $\binom{15}{3}p_e^3$ (ignoring the much less likely scenario of getting more than 3 errors). In fact, even distance-3 codes can detect *some* 3-qubit errors, so the probability can actually be a bit lower than that. For the case of the $[\![15, 1, 3]\!]$, there are just 35 undetectable triplets of errors, and so the probability is $35p_e^3$. Hence, while we started with 15 $|T\rangle$ states that have probability p_e of getting an error, we ended up with 1 $|T\rangle$ state with probability $35p_e^3$ of having an error. Hence, as long as $p_e > 35p_e^3$, we have made progress. In fact, for reasonably small p_e, we have made a LOT of progress.

Let's do some calculations to see how much we have accomplished. First, note that $p_e > 35p_e{}^3$ when $p_e < 1/\sqrt{35} \approx 0.17$. So even if we start with $|T\rangle$ states that have an error rate of up to 17% we can use this protocol to improve them. But let's suppose we start with $p_{e,0} = 10^{-2}$, a 1% error rate. There are already many quantum devices that are below this error rate. We see that after a single successful round distillation, we have $p_{e,1} = 3.5 \cdot 10^{-5}$. If this isn't good enough, we just take 15 of these better states and use them in another round of distillation. After a second round we have $p_{e,2} \approx 1.5 \cdot 10^{-12}$. A third round gives $p_{e,3} \approx 1.18 \cdot 10^{-34}$. This is already a much better error rate then we would ever need to do any realistic computation.

So, doing successive rounds of magic state distillation exponentially suppresses the probability of an error. However, it also has an exponential cost in noisy $|T\rangle$ states. For example, doing three rounds of this 15-to-1 protocol requires at least $15 \cdot 15 \cdot 15 = 3375$ noisy $|T\rangle$ states. However, we haven't yet accounted for the states we have to throw away when the protocol fails.

Our protocol aborts when we get a non-trivial syndrome. By far the most likely reason to get a non-trivial syndrome is when we see 1 error. The probability of this happening is $15p_e$. So in the first layer of distillation, when we have $p_{e,0} = 0.01$, the probability the protocol fails is 15%. To get the actual cost estimate, we hence need to multiply the cost of running the protocol once by the estimated number of runs we need to get a successful run. For the first layer this is $15/(1 - 0.15) \approx 17.6$. Hence, the expected $|T\rangle$ cost to produce one $|T\rangle$ state of error $p_{e,1} = 3.5 \cdot 10^{-5}$ is 17.6 instead of 15. By the time we get to the second and third layers, the probability of getting an error is already so small that the expected cost is very close to 15. Multiplying $17.6 \cdot 15 \cdot 15$, we see we can distill a very, very good $|T\rangle$ state for an expected cost of around 4000 noisy ones. Inserting some magic into our computation sure seems to come at a cost.

But while this is expensive, it is not totally crazy. This was one of the first protocols for doing magic state distillation, and its possible to bring this cost down quite a bit by finding better codes and protocols. In the next

section, we will show a state-of-the-art distillation protocol using almost all of the ingredients we've covered in this book so far, but before we get there, let's revisit the two simplifying assumptions we made at the start. The first was that we can do perfect, noiseless Clifford operations. Of course, nothing in quantum-computing land is noiseless, but if we have a family of error correcting codes of increasing distance with fault-tolerant implementations of all Clifford operations, we can perform Cliffords with arbitrary low levels of noise.

To benefit from this, we can do magic state distillation *inside* of another code E_c that has high-distance and fault-tolerant implementations of Clifford gates. Rather than starting with 15 physical $|T\rangle$ states and performing a round of distillation using physical gates and measurements, we prepare 15 noisy *logical* $|T\rangle$ states:

$$|\widetilde{T}\rangle \ := \ \boxed{\tfrac{\pi}{4}}\!-\!\boxed{E_c}\ \vdots$$

and then perform the whole protocol (11.58) encoded within E_c. Since E_c doesn't have transversal T gates, we can't prepare $|\widetilde{T}\rangle$ fault-tolerantly, but we can still prepare possibly faulty $|\widetilde{T}\rangle$ states, which we then distill to get better ones.

Generally to make this procedure as efficient as possible, we want to pick an 'ambient' code E_c whose probability of producing a logical error during the distillation protocol is just a bit lower than the logical error of the produced distilled states. This makes the Clifford operations 'just good enough' so that we can treat them as perfect, while not being overkill by using an overly expensive code.

Remark 11.5.1. The surface code doesn't have transversal S gates, so we don't know yet that it implements the full Clifford group. Nevertheless, we can still perform the analogous protocol to (11.58) using only CNOT, H, and Paulis in order to distill an $|S\rangle$ state. As we already remarked at the end of Section 11.4.3, once we have one good $|S\rangle$ state, we can use it to catalyse as many fault-tolerant S gates as we need, and hence the full Clifford group.

The second assumption was that the only kind of error we get is getting $Z|T\rangle$ instead of $|T\rangle$ with some probability p_e. Of course, many kinds of errors could happen when trying to prepare $|\widetilde{T}\rangle$, not just those that make a logical error of $\widetilde{Z}|\widetilde{T}\rangle = \vec{\mathcal{Z}}|\widetilde{T}\rangle$.

However, there is a nice trick, called **twirling**, which we can use to project all the other kinds of errors that might happen down to either "no error" or a Z error. Start with a $|\widetilde{T}\rangle$ state subject to arbitrary noise, and then with

50% probability, apply a logical unitary $\widetilde{S}\widetilde{X}$. After we do this, we can treat the resulting state as either staying the same ($|T\rangle$ is an eigenstate of SX after all) or flipping to $Z|T\rangle$ with some small probability. It probably seems rather counter-intuitive that this works. The best way to understand this is in the language of quantum mixed states and channels from Section* 2.6.2. We leave the details as a starred exercise:

Exercise* 11.15. In this exercise we will find out why we may assume that the error in preparing a $|T\rangle$ is only a Z, and nothing else.

a) Show that $|T\rangle$ and $Z|T\rangle$ are eigenstates of the operator SX.
b) Let Φ be the quantum channel acting on a qubit density matrix ρ via $\Phi(\rho) = \frac{1}{2}\rho + \frac{1}{2}(SX)\rho(SX)^\dagger$. Show that $|T\rangle\langle T|$ and $Z|T\rangle\langle T|Z$ are fixed points of Φ and that it sends $|T\rangle\langle T|Z$ and $Z|T\rangle\langle T|$ to zero.
c) Conclude that Φ is a projector to the space spanned by $|T\rangle\langle T|$ and $Z|T\rangle\langle T|Z$.
d) Argue that we hence have $\Phi(\rho) = (1 - p_e)|T\rangle\langle T| + p_e Z|T\rangle\langle T|Z$ for some probability p_e.

We see then that the resulting mixed state is a convex combination of the pure states $|T\rangle$ and $Z|T\rangle$, which behaves identically to a perfect $|T\rangle$ state that gets a Z-flip with probability p_e.

In summary, we have managed to convince ourselves that magic state distillation works, even starting from fairly realistic assumptions about what we can do on a real quantum computer. However, 4000-to-1 is still pretty expensive, so let's see if we can bring that cost down.

11.5.1 CCZ distillation and catalysis

We can use magic state distillation to build other kinds of non-Clifford states as well. A particularly handy one is the CCZ magic state:

$$|\text{CCZ}\rangle := \text{CCZ}|{+}{+}{+}\rangle = \ \text{⟨image⟩}$$

In this section, we'll build a distillation protocol for it based on the $[\![8, 3, 2]\!]$ code given in Example 11.4.16. In this code we can use 8 T gates to implement a CCZ gate:

Using this property, we can start with an encoded $|{+}{+}{+}\rangle$ state and apply transversal T gates to end up with a $|\text{CCZ}\rangle$ magic state. This in turn can be used to inject a CCZ gate at will using the procedure described in Exercise 9.5 using just Clifford operations.

We can use this 3-to-8 code to distil a $|\text{CCZ}\rangle$ with a lower error probability than the input $|T\rangle$ states using the repeat-until-success approach we described above. Because the code is distance 2, it can detect up to one error. As there is just one X stabiliser with support on all 8 qubits, we know that any combination of two errors leads to a trivial syndrome and hence will not be detectable, so that any pair of two errors will lead to the protocol failing. An input error probability of $p_{e,0}$ for the T gates hence gets boosted to $\binom{8}{2}p_{e,0}{}^2 = 28p_{e,0}{}^2$. This procedure then has a threshold of $p_e < 1/28 \approx 0.036$, or 3.6% for improving the error in the CCZ.

Note that this protocol is not directly stackable: it takes in $|T\rangle$ magic states, but outputs a $|\text{CCZ}\rangle$ magic state.

For this reason (and another one we will see soon), this distillation protocol is mostly used as a final stage, where we already have relatively high-quality $|T\rangle$ states, and we wish to convert them into a still higher quality $|\text{CCZ}\rangle$.

Suppose for instance that we start with $|T\rangle$ states with an error rate of $p_{e,0} = 10^{-3}$ (current machines get pretty close to being able to prepare states with this precision). Then one round of the 15-to-1 distillation described above gives us $p_{e,1} = 35p_{e,0}{}^3 = 3.5 \cdot 10^{-8}$. Using these states in our CCZ distillation protocol gives us a CCZ gate which has an error probability of $p_{e,2} = 28 \cdot p_{e,1}{}^2 = 3.43 \cdot 10^{-14}$. That is: we can expect to do about 30 trillion CCZ gates before we encounter one that is faulty. The cost for this is only about $15 \cdot 8 = 120$ noisy $|T\rangle$ states per distilled CCZ.

Instead of using the $|\text{CCZ}\rangle$ magic states to implement a CCZ gate, we can also use them to implement T gates. As we saw in Section 10.5, a CCZ gate can be converted back into two separate $|T\rangle$ states using *catalysis*.

If we do that with the CCZs produced by this protocol, this then gives us a pipeline to convert 8 T gates into one better CCZ, which is then converted into 2 T gates. This is then effectively an 8-to-2 protocol with an error improvement of $p_e \to 28p_e{}^2$. But there is an important detail here we shouldn't forget: when the protocol fails, i.e. more than one error happens so that we get a CCZ with an error, then when we convert this CCZ into two $|T\rangle$ states, both of these states will potentially inherit this error. Hence, when we do this we will get *correlated* errors. If these T gates are directly used in the final computation we wish to execute, this is not a problem since there we would consider any single error already catastrophic, so the probability of a double error being higher doesn't affect the overall success probability of the

computation. However, if we were to reuse these potentially "damaged" $|T\rangle$ states in another round of distillation, we might get a much higher chance of the protocol failing, and hence not get as good a quality of $|T\rangle$ states out of it as we would expect.

Despite these problems, the fact that we can distil at a rate of 8-to-2 is clearly a lot higher than 15-to-1, and as such this catalysis-based protocol, or variations of it, form part of some of the best-performing proposals for magic state distillation out there. In particular, as it takes 120 $|T\rangle$ for one CCZ, catalysing this back into 2 $|T\rangle$'s gives an effective rate of 60 $|T\rangle$ states with error 10^{-3} getting converted into 1 $|T\rangle$ state with error $3.43 \cdot 10^{-14}$.

11.6   Summary: What to remember

1. A quantum error correcting code is a subspace of n-qubit space. It lets us detect and correct errors by performing measurements to see if we have left the subspace.

2. Stabiliser codes are a family of quantum error correcting codes we can efficiently represent and manipulate using stabiliser theory and/or the Clifford ZX-calculus. They are subspaces of the form:

$$\mathrm{Stab}(\mathcal{S}) \ := \ \{ |\psi\rangle \mid \vec{P}|\psi\rangle = |\psi\rangle, \forall \vec{P} \in \mathcal{S} \}$$

for a stabiliser group $\mathcal{S} = \langle \vec{P}_1, \ldots, \vec{P}_m \rangle$.

3. An $[\![n, k, d]\!]$ stabiliser code encodes k logical qubits in n physical qubits and can detect any Pauli error of weight $< d$ and correct any error of weight t where $2t + 1 < d$.

4. We can relate logical qubits to physical qubits via an isometry called the encoder map:

5. For CSS codes, the encoder can always be written in one of two simple equivalent forms: the X-form and the Z-form. Using the scalable ZX calculus, these are:
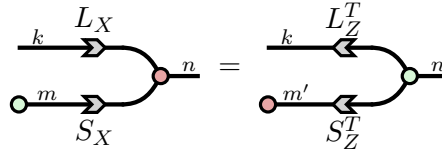
where the columns of L_X and S_X represent X-logical operators and

X-stabilisers, respectively, and similarly the columns of L_Z and S_Z represent Z-logical operators and Z-stabilisers.

6. The surface code is a well-studied family of codes, with many nice properties such as high levels of noise resistance, efficient decoding, and the ability to implement it using just 2D nearest-neighbour gates.
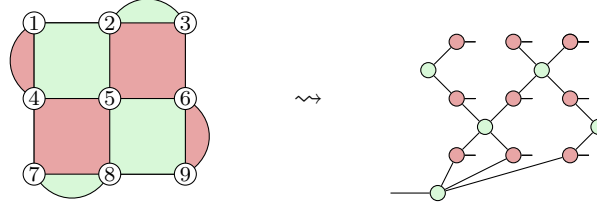
7. In order to achieve fault-tolerance, we must find ways to implement error correction, as well as the building blocks of universal quantum computation (e.g. state preparation, unitaries, and measurements) on encoded qubits without spreading errors.

8. Transversal unitary operations are fault tolerant because they only involve tensor products of operations on one qubit from each code block. However, no single code admits a universal set of transversal gates, due to the Eastin-Knill theorem.

9. CNOT gates are transversal for CSS codes and H gates are transversal for self-dual CSS codes:
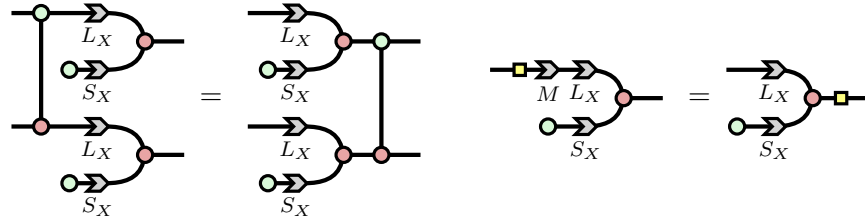
10. Strongly triorthogonal codes have transversal T gates:
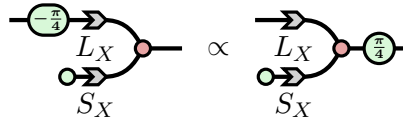
and triorthogonal codes have transversal T gates up to a (possibly non-transversal) Clifford unitary on the physical qubits.

11. Fault-tolerant syndrome-extraction protocols perform stabiliser measurements without spreading errors on the data qubits. Many of these protocols, like Shor's, can be seen as "unfusing" the Pauli projector:
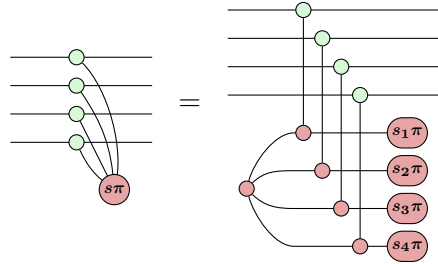
12. Magic state distillation lets us turn many noisy copies of a magic state into fewer, less-noisy copies:
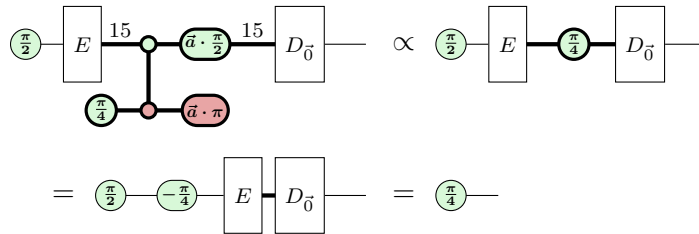
This can be very costly, requiring thousands of noisy $|T\rangle$ states to distil a very clean one.

13. Using a high-distance code with transversal Clifford operations, such as the surface code, along with magic state distillation, is a promising method for implementing universal fault-tolerant quantum computation.

11.7 References and Further Reading

Origins of quantum error correction All the core ideas of quantum error correction—quantum codes, fault-tolerant computations, the treshold theorem, stabiliser theory—originated in a flurry of activity between 1995 and 1999, with especially significant contributions by Shor, Steane, Gottesman and Kitaev. Many ideas were discovered independently by several people at the same. The original idea of quantum error correction was introduced independently by Shor (1995) and Steane (1996a). General conditions that any (not necessarily stabiliser) code must satisfy to correct errors were independently found in (Bennett et al., 1996) and Knill and Laflamme (1997) and are now sometimes called the *Knill-LaFlamme conditions*. The "History and further reading" section of Chapter 10 of Nielsen and Chuang (2010) provides many more references to the early developments in quantum error correction.

Origin of some quantum codes The 9-qubit Shor code was discovered by, well, Shor (Shor, 1995) and the 7-qubit Steane code by, you guessed it, Steane (Steane, 1996a). The 5-qubit code was discovered independently (again) by both Bennett et al. (1996) and Laflamme et al. (1996). Stabiliser theory and the idea of stabiliser codes was introduced by Gottesman (1997). The idea to combine a pair of orthogonal classical codes into a single quantum code was discovered independently by Calderbank and Shor (1996) and Steane (1996b) and hence these are named Calderbank-Shor-Steane (CSS) codes in their honour. The $[\![8, 3, 2]\!]$ 'smallest interesting colour code' was described as such by Campbell (n.d.) (although, as that article notes, the 7-qubit Steane code is the *smallest* non-trivial colour code, but it was judged not interesting enough).

Surface codes The surface code was introduced by Bravyi and Kitaev (1998), based on the slightly older *toric code* of Kitaev (Kitaev, 1997, 2003) which considers a 2d lattice defined on a *torus*, i.e. a donut. So whereas the surface code has an actual boundary where the lattice ends, in the toric code the surface loops around to create the surface of a donut. An in-depth study of correcting errors on the surface code by identifying connecting lines, and doing universal computation using transversal CNOT gates and magic state injection was done in Dennis et al. (2002). There they also found a first estimate of a treshold for the surface code. Transversal CNOTs are of course not practical for surface codes. In (Raussendorf and Harrington, 2007; Raussendorf et al., 2007) they use the method of introducing *punctures*, i.e. holes, into a surface code in order to encode multiple qubits into a single surface. The distance of the code is then the distance between two holes and the boundary. We can then perform two-qubit gates by deforming the code and 'rotating the holes around each other'. They find that this way of performing computations gives a threshold of 0.75% (later improved to $> 1\%$ in Wang et al. (2011)). A more accessible description of these results is given in Fowler et al. (2009), and an extensive review of this topic in Fowler et al. (2012) where they also give an estimate that running Shor's algorithm to factor a 2000-bit number would take about 200 million qubits and a full day of computation. The more compact, 'rotated' version of the surface code we use was first introduced in (Horsman et al., 2012), Section 7.1.

Lattice surgery As might be clear from those latter numbers, performing CNOTs by rotating qubits around each other tends to be expensive. The idea of merging and splitting rectangular patches of surface codes by *lattice surgery* was introduced by Horsman et al. (2012). That this indeed seems

to be much more efficient than braiding was argued in Fowler and Gidney (2018). An experimental demonstration of lattice surgery on real hardware was presented in Erhard et al. (2021).

Fault tolerance An early statement and proof of a fault-tolerant threshold theorem goes back to Shor (1996). This was improved upon by many other groups, including Aharonov and Ben-Or (1997), Knill et al. (1998), Kitaev (2003), and Aliferis et al. (2005) under various assumptions and error models. Shor's protocol for fault-tolerant stabiliser measurements is from (Shor, 1996), Steane's from (Steane, 1997), and Knill's from (Knill, 2005).

Error correcting codes and ZX The surface code was first presented in the ZX-calculus by Horsman (2011), who also found that the logical function of the merging and splitting operation is actually just Z- and X-spiders de Beaudrap and Horsman (2020). Duncan and Lucas (2014) was the first paper to use ZX-calculus to verify the correctness of an error correcting code (the Steane code), which was followed up by a verification of 'the smallest interesting colour code' Garvie and Duncan (2018), which we gave in Example 11.4.16. In Chancellor et al. (2016) they used a proto version of scalable ZX notation to find a new class of quantum codes. The correspondence between phase-free ZX-diagrams and CSS codes was established by Kissinger (2022), where he also proved the correctness of lattice surgery. The ZX description of transversal non-Clifford gates in triorthogonal codes is from Kissinger and van de Wetering (2024).

Magic state distillation The concept of distilling a noisy non-Clifford state by encoding it in a code with a transversal non-Clifford gate was introduced by Bravyi and Kitaev (2005), where they found the 15-to-1 protocol. This was generalised to an entire family of protocols based on triorthogonal codes in Bravyi and Haah (2012). A protocol based on using the transversal Hadamard in the $[\![4, 2, 2]\!]$ code was given in Meier et al. (2013). This works a bit differently, as we use the transversal Hadamard to perform a logical Hadamard eigenbasis measurement, which distills the Hadamard eigenstate, which is Clifford equivalent to $|T\rangle$. The CCZ distillation used to implement T gates via catalysis was introduced in Gidney and Fowler (2019). A comprehensive analysis of running Shor's algorithm to factor a 2048-bit number using surface code lattice surgery with this improved catalysed CCZ distillation scheme was given in Gidney and Ekerå (2021), where they find you require 20 million qubits and 8 hours of computation time, a large improvement over the older scheme using braiding and iterated 15-to-1 distillation.

Even further reading An excellent overview of quantum error correction and fault-tolerance can be found in the book of Gottesman (2024), which at the time of this writing was available as a preprint freely online. A standard, comprehensive, and approachable text on classical error correction is (MacWilliams and Sloane, 1977). An accessible and quite comprehensive fault-tolerant quantum computing scheme based on surface code lattice surgery and magic state distillation is *A Game of Surface Codes* by Litinski (2019).

References

Aaronson, Scott, and Gottesman, Daniel. 2004. Improved simulation of stabilizer circuits. *Physical Review A*, **70**(5), 052328.

Abramsky, S., and Coecke, B. 2004. A categorical semantics of quantum protocols. Pages 415–425 of: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS)*. arXiv:quant-ph/0402130.

Adleman, Leonard M, DeMarrais, Jonathan, and Huang, Ming-Deh A. 1997. Quantum computability. *SIAM Journal on Computing*, **26**(5), 1524–1540.

Aharonov, Dorit. 2003. A simple proof that Toffoli and Hadamard are quantum universal. *arXiv preprint quant-ph/0301040*.

Aharonov, Dorit, and Ben-Or, Michael. 1997. Fault-tolerant quantum computation with constant error. Pages 176–188 of: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*.

Aharonov, Dorit, and Naveh, Tomer. 2002. Quantum NP-a survey. *arXiv preprint quant-ph/0210077*.

Alber, Gernot, Beth, Thomas, Horodecki, Michał, Horodecki, Paweł, Horodecki, Ryszard, Rötteler, Martin, Weinfurter, Harald, Werner, Reinhard, Zeilinger, Anton, Beth, Thomas, et al. 2001. Quantum algorithms: Applicable algebra and quantum physics. *Quantum information: an introduction to basic theoretical concepts and experiments*, 96–150.

Aliferis, Panos, Gottesman, Daniel, and Preskill, John. 2005. Quantum accuracy threshold for concatenated distance-3 codes. *arXiv preprint quant-ph/0504218*.

Ambainis, A. 2010. *New developments in quantum algorithms.* arXiv:1006.4014.

Amy, M., Maslov, D., Mosca, M., and Roetteler, M. 2013a. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **32**(6), 818–830.

Amy, Matthew. 2019. *Formal methods in quantum circuit design.* Ph.D. thesis, University of Waterloo.

Amy, Matthew, and Mosca, Michele. 2019. T-count optimization and Reed-Muller codes. *Transactions on Information Theory*.

Amy, Matthew, and Ross, Neil J. 2021. Phase-state duality in reversible circuit design. *Phys. Rev. A*, **104**(Nov), 052602.

Amy, Matthew, Maslov, Dmitri, Mosca, Michele, and Roetteler, Martin. 2013b. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum

circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **32**(6), 818–830.

Amy, Matthew, Maslov, Dmitri, and Mosca, Michele. 2014. Polynomial-time T-depth optimization of Clifford+ T circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **33**(10), 1476–1489.

Amy, Matthew, Crawford, Matthew, Glaudell, Andrew N, Macasieb, Melissa L, Mendelson, Samuel S, and Ross, Neil J. 2023. Catalytic embeddings of quantum circuits. *arXiv preprint arXiv:2305.07720*.

Backens, Miriam. 2014a. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, **16**(9), 093021.

Backens, Miriam. 2014b. The ZX-calculus is complete for the single-qubit Clifford+T group. Pages 293–303 of: Coecke, Bob, Hasuo, Ichiro, and Panangaden, Prakash (eds), *Proceedings of the 11th workshop on Quantum Physics and Logic*. Electronic Proceedings in Theoretical Computer Science, vol. 172. Open Publishing Association.

Backens, Miriam, and Kissinger, Aleks. 2019. ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity. Pages 18–34 of: Selinger, Peter, and Chiribella, Giulio (eds), *Proceedings of the 15th International Conference on Quantum Physics and Logic, Halifax, Canada, 3-7th June 2018*. Electronic Proceedings in Theoretical Computer Science, vol. 287. Open Publishing Association.

Backens, Miriam, Perdrix, Simon, and Wang, Quanlong. 2016. A Simplified Stabilizer zx-calculus. In: *Proceedings of the 13th International Conference on Quantum Physics and Logic*. arXiv:1602.04744.

Backens, Miriam, Miller-Bakewell, Hector, de Felice, Giovanni, Lobski, Leo, and van de Wetering, John. 2020. *There and back again: A circuit extraction tale*. Preprint.

Backens, Miriam, Kissinger, Aleks, Miller-Bakewell, Hector, van de Wetering, John, and Wolffs, Sal. 2023. Completeness of the ZH-calculus. *Compositionality*, **5**(7).

Barenco, A., Bennett, C. H., Cleve, R., DiVincenzo, D. P., Margolus, N., Shor, P. W., Sleator, T., Smolin, J. A., and Weinfurter, H. 1995. Elementary gates for quantum computation. *Physical Review A*, **52**, 3457–3467.

Benioff, P. 1980. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, **22**, 563–591.

Bennett, Charles H., DiVincenzo, David P., Smolin, John A., and Wootters, William K. 1996. Mixed-state entanglement and quantum error correction. *Phys. Rev. A*, **54**(Nov), 3824–3851.

Bernstein, Ethan, and Vazirani, Umesh. 1997. Quantum Complexity Theory. *SIAM Journal on Computing*, **26**(5), 1411–1473.

Berry, Dominic W., Childs, Andrew M., Cleve, Richard, Kothari, Robin, and Somma, Rolando D. 2015. Simulating Hamiltonian Dynamics with a Truncated Taylor Series. *Phys. Rev. Lett.*, **114**(Mar), 090502.

Bonchi, Filippo, Sobociński, Paweł, and Zanasi, Fabio. 2014. Interacting bialgebras are Frobenius. Pages 351–365 of: *International Conference on Foundations of Software Science and Computation Structures*. Springer.

Bravyi, Sergey, and Gosset, David. 2016. Improved Classical Simulation of Quantum Circuits Dominated by Clifford Gates. *Phys. Rev. Lett.*, **116**(Jun), 250501.

Bravyi, Sergey, and Haah, Jeongwan. 2012. Magic-state distillation with low overhead. *Phys. Rev. A*, **86**(Nov), 052329.

Bravyi, Sergey, and Kitaev, Alexei. 2005. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A*, **71**(2), 022316.

Bravyi, Sergey, Smith, Graeme, and Smolin, John A. 2016. Trading classical and quantum computational resources. *Physical Review X*, **6**(2), 021043.

Bravyi, Sergey, Browne, Dan, Calpin, Padraic, Campbell, Earl, Gosset, David, and Howard, Mark. 2019. Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum*, **3**(9), 181.

Bravyi, Sergey B, and Kitaev, A Yu. 1998. Quantum codes on a lattice with boundary. *arXiv preprint quant-ph/9811052*.

Briegel, Hans J, and Raussendorf, Robert. 2001. Persistent entanglement in arrays of interacting particles. *Physical Review Letters*, **86**(5), 910.

Briegel, Hans J, Browne, David E, Dür, Wolfgang, Raussendorf, Robert, and Van den Nest, Maarten. 2009. Measurement-based quantum computation. *Nature Physics*, **5**(1), 19–26.

Browne, Daniel E., Kashefi, Elham, Mhalla, Mehdi, and Perdrix, Simon. 2007. Generalized flow and determinism in measurement-based quantum computation. *New Journal of Physics*, **9**(8), 250.

Calderbank, A. R., and Shor, Peter W. 1996. Good quantum error-correcting codes exist. *Phys. Rev. A*, **54**(Aug), 1098–1105.

Calderbank, A. R., Rains, E. M., Shor, P. W., and Sloane, N. J. A. 1997. Quantum Error Correction and Orthogonal Geometry. *Phys. Rev. Lett.*, **78**(Jan), 405–408.

Campbell, Earl. *The Smallest Interesting Colour Code.* https://earltcampbell.com/2016/09/26/the-smallest-interesting-colour-code/.

Campbell, Earl. 2019. Random Compiler for Fast Hamiltonian Simulation. *Phys. Rev. Lett.*, **123**(Aug), 070503.

Carette, Titouan. 2021. When Only Topology Matters. *arXiv preprint arXiv:2102.03178*.

Carette, Titouan, and Jeandel, Emmanuel. 2020. A Recipe for Quantum Graphical Languages. Pages 118:1–118:17 of: Czumaj, Artur, Dawar, Anuj, and Merelli, Emanuela (eds), *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 168. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Carette, Titouan, Horsman, Dominic, and Perdrix, Simon. 2019. SZX-Calculus: Scalable Graphical Quantum Reasoning. Pages 55:1–55:15 of: Rossmanith, Peter, Heggernes, Pinar, and Katoen, Joost-Pieter (eds), *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 138. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Chancellor, Nicholas, Kissinger, Aleks, Roffe, Joschka, Zohren, Stefan, and Horsman, Dominic. 2016. Graphical Structures for Design and Verification of Quantum Error Correction. *arXiv preprint arXiv:1611.08012*.

Chi-Chih Yao, A. 1993 (Nov.). Quantum circuit complexity. Pages 352–361 of: *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*.

Coecke, B., and Kissinger, A. 2010a. The compositional structure of multipartite quantum entanglement. Pages 297–308 of: *Automata, Languages and Programming*. Lecture Notes in Computer Science. Springer. arXiv:1002.2540.

Coecke, B., and Pavlovic, D. 2007. Quantum measurements without sums. Pages 567–604 of: Chen, G., Kauffman, L., and Lamonaco, S. (eds), *Mathematics of Quantum Computing and Technology*. Taylor and Francis. arXiv:quant-ph/0608035.

Coecke, B., Kissinger, A., Merry, A., and Roy, S. 2010. The GHZ/W-calculus contains rational arithmetic. *Electronic Proceedings in Theoretical Computer Science*, **52**, 34–48.

Coecke, B., Pavlović, D., and Vicary, J. 2013. A new description of orthogonal bases. *Mathematical Structures in Computer Science, to appear*, **23**, 555–567. arXiv:quant-ph/0810.1037.

Coecke, Bob, and Duncan, Ross. 2008. Interacting quantum observables. In: *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP)*. Lecture Notes in Computer Science.

Coecke, Bob, and Duncan, Ross. 2011. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, **13**, 043016.

Coecke, Bob, and Kissinger, Aleks. 2010b. The compositional structure of multipartite quantum entanglement. Pages 297–308 of: *Automata, Languages and Programming*. Lecture Notes in Computer Science. Springer.

Coecke, Bob, and Kissinger, Aleks. 2017. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press.

Coecke, Bob, and Wang, Quanlong. 2018. ZX-rules for 2-qubit Clifford+ T quantum circuits. Pages 144–161 of: *International Conference on Reversible Computation*. Springer.

Coecke, Bob, Duncan, Ross, Kissinger, Aleks, and Wang, Quanlong. 2012. Strong Complementarity and Non-locality in Categorical Quantum Mechanics. Pages 245–254 of: *2012 27th Annual IEEE Symposium on Logic in Computer Science*.

Cowtan, Alexander, Dilkes, Silas, Duncan, Ross, Simmons, Will, and Sivarajah, Seyon. 2020. Phase Gadget Synthesis for Shallow Circuits. Pages 213–228 of: Coecke, Bob, and Leifer, Matthew (eds), *Proceedings 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*. Electronic Proceedings in Theoretical Computer Science, vol. 318. Open Publishing Association.

Cross, Andrew, Javadi-Abhari, Ali, Alexander, Thomas, De Beaudrap, Niel, Bishop, Lev S, Heidel, Steven, Ryan, Colm A, Sivarajah, Prasahnt, Smolin, John, Gambetta, Jay M, et al. 2022. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, **3**(3), 1–50.

Cross, Andrew W, Bishop, Lev S, Smolin, John A, and Gambetta, Jay M. 2017. *Open quantum assembly language*. Preprint.

Cui, Shawn X., Gottesman, Daniel, and Krishna, Anirudh. 2017. Diagonal gates in the Clifford hierarchy. *Physical Review A*, **95**(1), 012329.

Dalzell, Alexander M, McArdle, Sam, Berta, Mario, Bienias, Przemyslaw, Chen, Chi-Fang, Gilyén, András, Hann, Connor T, Kastoryano, Michael J, Khabiboulline, Emil T, Kubica, Aleksander, et al. 2023. Quantum algorithms: A survey of applications and end-to-end complexities. *arXiv preprint arXiv:2310.03011*.

Danos, V., and Kashefi, E. 2006. Determinism in the one-way model. *Physical Review A*, **74**(052310).

Danos, Vincent, Kashefi, Elham, and Panangaden, Prakash. 2007. The measurement calculus. *Journal of the ACM (JACM)*, **54**(2), 8–es.

Danos, Vincent, Kashefi, Elham, Panangaden, Prakash, and Perdrix, Simon. 2009. Extended measurement calculus. *Semantic techniques in quantum computation*, 235–310.

Dawson, Christopher M, and Nielsen, Michael A. 2005. *The solovay-kitaev algorithm*. Preprint.

Dawson, Christopher M, Hines, Andrew P, Mortimer, Duncan, Haselgrove, Henry L, Nielsen, Michael A, and Osborne, Tobias J. 2005. Quantum computing and polynomial equations over the finite field Z2. *Quantum Information & Computation*, **5**(2), 102–112.

de Beaudrap, Niel, and Horsman, Dominic. 2020. The ZX calculus is a language for surface code lattice surgery. *Quantum*, **4**.

de Beaudrap, Niel, Bian, Xiaoning, and Wang, Quanlong. 2020a. Fast and Effective Techniques for T-Count Reduction via Spider Nest Identities. Pages 11:1–11:23 of: Flammia, Steven T. (ed), *15th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2020)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 158. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

de Beaudrap, Niel, Duncan, Ross, Horsman, Dominic, and Perdrix, Simon. 2020b. Pauli Fusion: a Computational Model to Realise Quantum Transformations from ZX Terms. Pages 85–105 of: Coecke, Bob, and Leifer, Matthew (eds), *Proceedings 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*. Electronic Proceedings in Theoretical Computer Science, vol. 318. Open Publishing Association.

de Beaudrap, Niel, Bian, Xiaoning, and Wang, Quanlong. 2020c. Techniques to Reduce $\pi/4$-Parity-Phase Circuits, Motivated by the ZX Calculus. Pages 131–149 of: Coecke, Bob, and Leifer, Matthew (eds), *Proceedings 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*. Electronic Proceedings in Theoretical Computer Science, vol. 318. Open Publishing Association.

de Brugière, Timothée Goubault, Baboulin, Marc, Valiron, Benoît, Martiel, Simon, and Allouche, Cyril. 2020. Quantum CNOT circuits synthesis for NISQ architectures using the syndrome decoding problem. Pages 189–205 of: *Reversible Computation: 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings 12*. Springer.

Dehaene, Jeroen, and De Moor, Bart. 2003. Clifford group, stabilizer states, and linear and quadratic operations over GF(2). *Physical Review A*, **68**(4), 042318.

Dennis, Eric, Kitaev, Alexei, Landahl, Andrew, and Preskill, John. 2002. Topological quantum memory. *Journal of Mathematical Physics*, **43**(9), 4452–4505.

Deutsch, D. 1989. Quantum computational networks. *Proceedings of the Royal Society of London*, **425**.

Deutsch, D., and Jozsa, R. 1992. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, **439**(1907), 553–558.

Deutsch, David. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, **400**(1818), 97–117.

Di Matteo, Olivia, and Mosca, Michele. 2016. Parallelizing quantum circuit synthesis. *Quantum Science and Technology*, **1**(1), 015003.

DiVincenzo, David P. 1995. Two-bit gates are universal for quantum computation. *Physical Review A*, **51**(2), 1015.

Duncan, Ross, and Dunne, Kevin. 2016. Interacting Frobenius Algebras are Hopf. Pages 1–10 of: *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE.

Duncan, Ross, and Lucas, Maxime. 2014. Verifying the Steane code with Quantomatic. Pages 33–49 of: Coecke, Bob, and Hoban, Matty (eds), Proceedings of the 10th International Workshop on *Quantum Physics and Logic,* Castelldefels (Barcelona), Spain, 17th to 19th July 2013. Electronic Proceedings in Theoretical Computer Science, vol. 171. Open Publishing Association.

Duncan, Ross, and Perdrix, Simon. 2009. Graph states and the necessity of Euler decomposition. *Mathematical Theory and Computational Practice*, 167–177.

Duncan, Ross, and Perdrix, Simon. 2010. Rewriting measurement-based quantum computations with generalised flow. Pages 285–296 of: *International Colloquium on Automata, Languages, and Programming*. Springer.

Duncan, Ross, and Perdrix, Simon. 2013. Pivoting Makes the ZX-Calculus Complete for Real Stabilizers. In: *QPL 2013-10th Workshop on Quantum Physics and Logic*.

Duncan, Ross, Kissinger, Aleks, Perdrix, Simon, and van de Wetering, John. 2020. Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus. *Quantum*, **4**(6), 279.

Elliott, Matthew B, Eastin, Bryan, and Caves, Carlton M. 2008. Graphical description of the action of Clifford operators on stabilizer states. *Physical Review A*, **77**(4), 042307.

Erhard, Alexander, Poulsen Nautrup, Hendrik, Meth, Michael, Postler, Lukas, Stricker, Roman, Stadler, Martin, Negnevitsky, Vlad, Ringbauer, Martin, Schindler, Philipp, Briegel, Hans J, et al. 2021. Entangling logical qubits with lattice surgery. *Nature*, **589**(7841), 220–224.

Feynman, R. P. 1982. Simulating physics with computers. *International journal of theoretical physics*, **21**, 467–488.

Fortnow, Lance, and Rogers, John. 1999. Complexity limitations on quantum computation. *Journal of Computer and System Sciences*, **59**(2), 240–252.

Fowler, Austin G, and Gidney, Craig. 2018. Low overhead quantum computation using lattice surgery. *arXiv preprint arXiv:1808.06709.*

Fowler, Austin G., Stephens, Ashley M., and Groszkowski, Peter. 2009. High-threshold universal quantum computation on the surface code. *Phys. Rev. A*, **80**(Nov), 052312.

Fowler, Austin G., Mariantoni, Matteo, Martinis, John M., and Cleland, Andrew N. 2012. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A*, **86**(Sep), 032324.

Garvie, Liam, and Duncan, Ross. 2018. Verifying the Smallest Interesting Colour Code with Quantomatic. Pages 147–163 of: Coecke, Bob, and Kissinger, Aleks (eds), *Proceedings 14th International Conference on Quantum Physics and Logic, Nijmegen, The Netherlands, 3-7 July 2017*. Electronic Proceedings in Theoretical Computer Science, vol. 266. Open Publishing Association.

Gidney, Craig. 2015 (June). *Constructing Large Controlled Nots*. Blogpost. https://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Nots.html.

Gidney, Craig. 2018. Halving the cost of quantum addition. *Quantum*, **2**(6), 74.

Gidney, Craig, and Ekerå, Martin. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, **5**(4), 433.

Gidney, Craig, and Fowler, Austin G. 2019. Efficient magic state factories with a catalyzed $|CCZ\rangle$ to $2|T\rangle$ transformation. *Quantum*, **3**(4), 135.

Gidney, Craig, and Jones, N Cody. 2021. A CCCZ gate performed with 6 T gates. *arXiv preprint arXiv:2106.11513*.

Giles, Brett, and Selinger, Peter. 2013. Exact synthesis of multiqubit Clifford+T circuits. *Physical Review A*, **87**(3), 032332.

Gogioso, Stefano. 2019. A Diagrammatic Approach to Quantum Dynamics. Pages 19:1–19:23 of: Roggenbach, Markus, and Sokolova, Ana (eds), *8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 139. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Gottesman, Daniel. 1996. Class of quantum error-correcting codes saturating the quantum Hamming bound. *Physical Review A*, **54**(3), 1862.

Gottesman, Daniel. 1997. Stabilizer codes and quantum error correction. *arXiv preprint quant-ph / 9705052*.

Gottesman, Daniel. 1998. The Heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*.

Gottesman, Daniel. 2024. Surviving as a quantum computer in a classical world. *Textbook manuscript preprint*.

Gottesman, Daniel, and Chuang, Isaac L. 1999. Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature*, **402**(6760), 390–393.

Goubault de Brugière, Timothèe. 2020. *Methods for optimizing the synthesis of quantum circuits*. Ph.D. thesis, Université Paris-Saclay.

Green, Alexander S, Lumsdaine, Peter LeFanu, Ross, Neil J, Selinger, Peter, and Valiron, Benoît. 2013. Quipper: a scalable quantum programming language. Pages 333–342 of: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*.

Greylyn, Seth. 2014. *Generators and relations for the group $U_(\mathbb{Z}[1/\sqrt{2}, i])$*. M.Phil. thesis, Dalhousie University.

Gross, David, and Van den Nest, Maarten. 2008. The LU-LC conjecture, diagonal local operations and quadratic forms over GF(2). *Quantum Info. Comput.*, **8**(3), 263–281.

Grover, Lov K. 1996. A Fast Quantum Mechanical Algorithm for Database Search. Pages 212–219 of: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. New York, NY, USA: ACM.

Hadzihasanovic, A. 2015a. A diagrammatic axiomatisation for qubit entanglement. In: *Proceedings of the 30th Annual IEEE Symposium on Logic in Computer Science (LICS)*. arXiv:1501.07082.

Hadzihasanovic, Amar. 2015b. A diagrammatic axiomatisation for qubit entanglement. Pages 573–584 of: *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE.

Hadzihasanovic, Amar, Ng, Kang Feng, and Wang, Quanlong. 2018. Two Complete Axiomatisations of Pure-state Qubit Quantum Computing. Pages 502–511 of: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. New York, NY, USA: ACM.

Hall, Brian C, and Hall, Brian C. 2013. *Lie groups, Lie algebras, and representations*. Springer.

Heyfron, Luke E, and Campbell, Earl T. 2018. An efficient quantum compiler that reduces T count. *Quantum Science and Technology*, **4**(015004).

Horsman, Clare. 2011. Quantum picturalism for topological cluster-state computing. *New Journal of Physics*, **13**(9), 095011.

Horsman, Dominic, Fowler, Austin G, Devitt, Simon, and Van Meter, Rodney. 2012. Surface code quantum computing by lattice surgery. *New Journal of Physics*, **14**(12), 123011.

Howard, Mark, and Campbell, Earl. 2017. Application of a Resource Theory for Magic States to Fault-Tolerant Quantum Computing. *Phys. Rev. Lett.*, **118**(Mar), 090501.

IBM. *Qiskit.* https://www.ibm.com/quantum/qiskit.

Jeandel, Emmanuel, Perdrix, Simon, and Vilmart, Renaud. 2018. A Complete Axiomatisation of the ZX-calculus for Clifford+T Quantum Mechanics. Pages 559–568 of: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM.

Jeandel, Emmanuel, Perdrix, Simon, and Veshchezerova, Margarita. 2024. Addition and Differentiation of ZX-diagrams. *Logical Methods in Computer Science*, **Volume 20, Issue 2**(5).

Jones, Cody. 2013. Low-overhead constructions for the fault-tolerant Toffoli gate. *Physical Review A*, **87**(2), 022328.

Jordan, Stephen. 2022. *The Quantum Algorithms Zoo.* https://quantumalgorithmzoo.org.

Jozsa, R. 1997. Quantum algorithms and the Fourier transform. In: *Proceedings of the Santa Barbarba Conference on Coherence and Decoherence*. Proceedings of the Royal Society of London.

Kelly, Gregory M, and Laplaza, Miguel L. 1980. Coherence for compact closed categories. *Journal of pure and applied algebra*, **19**, 193–213.

Kissinger, Aleks. 2022. Phase-free ZX diagrams are CSS codes (...or how to graphically grok the surface code). *arXiv preprint arXiv:2204.14038*.

Kissinger, Aleks, and de Griend, Arianne Meijer-van. 2020. CNOT circuit extraction for topologically-constrained quantum memories. *Quant. Inf. Comput.*, **20**(arXiv: 1904.00633), 581–596.

Kissinger, Aleks, and van de Wetering, John. 2019. Universal MBQC with generalised parity-phase interactions and Pauli measurements. *Quantum*, **3**(4), 134.

Kissinger, Aleks, and van de Wetering, John. 2020a. PyZX: Large Scale Automated Diagrammatic Reasoning. Pages 229–241 of: Coecke, Bob, and Leifer, Matthew (eds), *Proceedings 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*. Electronic Proceedings in Theoretical Computer Science, vol. 318. Open Publishing Association.

Kissinger, Aleks, and van de Wetering, John. 2020b. Reducing the number of non-Clifford gates in quantum circuits. *Physical Review A*, **102**(8), 022406.

Kissinger, Aleks, and van de Wetering, John. 2022. Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. *Quantum Science and Technology*, **7**(4), 044001.

Kissinger, Aleks, and van de Wetering, John. 2024. Scalable Spider Nests (...Or How to Graphically Grok Transversal Non-Clifford Gates). Pages 79–95 of: Díaz-Caro, Alejandro, and Zamdzhiev, Vladimir (eds), *Proceedings of the 21st International Conference on Quantum Physics and Logic, Buenos Aires, Argentina, July 15-19, 2024*. Electronic Proceedings in Theoretical Computer Science, vol. 406. Open Publishing Association.

Kissinger, Aleks, and Zamdzhiev, Vladimir. 2015. Quantomatic: A proof assistant for diagrammatic reasoning. Pages 326–336 of: *International Conference on Automated Deduction.* Springer.

Kissinger, Aleks, van de Wetering, John, and Vilmart, Renaud. 2022. Classical Simulation of Quantum Circuits with Partial and Graphical Stabiliser Decompositions. Pages 5:1–5:13 of: Le Gall, François, and Morimae, Tomoyuki (eds), *17th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2022).* Leibniz International Proceedings in Informatics (LIPIcs), vol. 232. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Kissinger, Aleks, J. Ross, Neil, and van de Wetering, John. 2024. Catalysing Completeness and Universality. *arXiv preprint arXiv:2404.09915.*

Kitaev, A Yu. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, **52**(6), 1191.

Kitaev, A Yu. 2003. Fault-tolerant quantum computation by anyons. *Annals of physics*, **303**(1), 2–30.

Kliuchnikov, Vadym, Lauter, Kristin, Minko, Romy, Paetznick, Adam, and Petit, Christophe. 2023. Shorter quantum circuits via single-qubit gate approximation. *Quantum*, **7**(12), 1208.

Knill, Emanuel. 2005. Quantum computing with realistically noisy devices. *Nature*, **434**(7029), 39–44.

Knill, Emanuel, and Laflamme, Raymond. 1997. Theory of quantum error-correcting codes. *Phys. Rev. A*, **55**(Feb), 900–911.

Knill, Emanuel, Laflamme, Raymond, and Zurek, Wojciech H. 1998. Resilient quantum computation. *Science*, **279**(5349), 342–345.

Kotzig, Anton. 1968. Eulerian lines in finite 4-valent graphs and their transformations. Pages 219–230 of: *Colloqium on Graph Theory Tihany 1966.* Academic Press.

Kuijpers, Stach, van de Wetering, John, and Kissinger, Aleks. 2019. *Graphical Fourier Theory and the Cost of Quantum Addition.* Preprint.

Lack, S. 2004. Composing PROPs. *Theory and Applications of Categories*, **13**, 147–163.

Laflamme, Raymond, Miquel, Cesar, Paz, Juan Pablo, and Zurek, Wojciech Hubert. 1996. Perfect Quantum Error Correcting Code. *Phys. Rev. Lett.*, **77**(Jul), 198–201.

Landauer, R. 1961. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, **5**(3), 183–191.

Lemonnier, Louis, van de Wetering, John, and Kissinger, Aleks. 2020. Hypergraph simplification: Linking the path-sum approach to the ZH-calculus. *arXiv preprint arXiv:2003.13564.*

Litinski, Daniel. 2019. A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery. *Quantum*, **3**(3), 128.

Mac Lane, Saunders. 2013. *Categories for the working mathematician.* Vol. 5. Springer Science & Business Media.

MacLane, Saunders. 1965. Categorical algebra. *Bulletin of the American Mathematical Society*, **71**(1), 40–106.

MacWilliams, F.J., and Sloane, N.J.A. 1977. *The Theory of Error-Correcting Codes.* North-Holland Mathematical Library, vol. 16. North Holland Publishing Co.

Manin, Y. I. 1980. Vychislimoe i Nevychislimoe. *Sovetskoye Radio.*

Markov, Ketan, Patel, Igor, and Hayes, John. 2008. Optimal synthesis of linear reversible circuits. *Quantum Information and Computation*, **8**(3&4), 0282–0294.

Maslov, Dmitri, and Roetteler, Martin. 2018. Shorter stabilizer circuits via Bruhat decomposition and quantum circuit transformations. *IEEE Transactions on Information Theory*, **64**(7), 4729–4738.

Maslov, Dmitri, and Zindorf, Ben. 2022. Depth optimization of CZ, CNOT, and Clifford circuits. *IEEE Transactions on Quantum Engineering*, **3**, 1–8.

Meier, Adam M., Eastin, Bryan, and Knill, Emanuel. 2013. Magic-state distillation with the four-qubit code. *Quantum Info. Comput.*, **13**(3–4), 195–209.

Meijer-van de Griend, Arianne, and Duncan, Ross. 2023. Architecture-Aware Synthesis of Phase Polynomials for NISQ Devices. Pages 116–140 of: Gogioso, Stefano, and Hoban, Matty (eds), *Proceedings 19th International Conference on Quantum Physics and Logic, Wolfson College, Oxford, UK, 27 June - 1 July 2022*. Electronic Proceedings in Theoretical Computer Science, vol. 394. Open Publishing Association.

Meuli, Giulia, Soeken, Mathias, Roetteler, Martin, Bjorner, Nikolaj, and De Micheli, Giovanni. 2019a. Reversible pebbling game for quantum memory management. Pages 288–291 of: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.

Meuli, Giulia, Soeken, Mathias, Campbell, Earl, Roetteler, Martin, and de Micheli, Giovanni. 2019b. The Role of Multiplicative Complexity in Compiling Low *T*-count Oracle Circuits. Pages 1–8 of: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

Meuli, Giulia, Soeken, Mathias, Roetteler, Martin, and De Micheli, Giovanni. 2020. ROS: Resource-constrained Oracle Synthesis for Quantum Computers. Pages 119–130 of: Coecke, Bob, and Leifer, Matthew (eds), Proceedings 16th International Conference on *Quantum Physics and Logic,* Chapman University, Orange, CA, USA., 10-14 June 2019. Electronic Proceedings in Theoretical Computer Science, vol. 318. Open Publishing Association.

Mhalla, Mehdi, and Perdrix, Simon. 2013. Graph States, Pivot Minor, and Universality of (X,Z)-measurements. *International Journal of Unconventional Computing*, **9**(1-2), 153–171.

Microsoft. *Announcing the Microsoft Quantum Development Kit.* https://azure.microsoft.com/en-us/blog/quantum/2017/12/11/announcing-microsoft-quantum-development-kit/.

Miller, Jacob, and Miyake, Akimasa. 2016. Hierarchy of universal entanglement in 2D measurement-based quantum computation. *npj Quantum Information*, **2**, 16036.

Miyazaki, Jisho, Hajdušek, Michal, and Murao, Mio. 2015. Analysis of the trade-off between spatial and temporal resources for measurement-based quantum computation. *Physical Review A*, **91**(5), 052302.

Montanaro, A. 2015. *Quantum algorithms: an overview.* arXiv:1511.04206.

Morales, Mauro E. S., Costa, Pedro C. S., Burgarth, Daniel K., Sanders, Yuval R., and Berry, Dominic W. 2022 (Oct.). *Greatly improved higher-order product formulae for quantum simulation.* arXiv:2210.15817 [quant-ph].

Mosca, Michele. 2008. Quantum algorithms. *arXiv preprint arXiv:0808.0369.*

Nash, Beatrice, Gheorghiu, Vlad, and Mosca, Michele. 2020. Quantum circuit optimizations for NISQ architectures. *Quantum Science and Technology*, **5**(2), 025010.

Ng, Kang Feng, and Wang, Quanlong. 2017. *A universal completion of the ZX-calculus.* Preprint.

Nielsen, M. A., and Chuang, Isaac L. 2010. *Quantum computation and quantum information.* Cambridge university press.

Ömer, Bernhard. 2002. Procedural quantum programming. Pages 276–285 of: *AIP Conference Proceedings*, vol. 627. American Institute of Physics.

Pashayan, Hakop, Wallman, Joel J., and Bartlett, Stephen D. 2015. Estimating Outcome Probabilities of Quantum Circuits Using Quasiprobabilities. *Phys. Rev. Lett.*, **115**(Aug), 070501.

Perdrix, Simon, and Wang, Quanlong. 2016. Supplementarity is Necessary for Quantum Diagram Reasoning. Pages 76:1–76:14 of: *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 58.

Poór, Boldizsár. 2022. *A unique normal form for prime-dimensional qudit Clifford ZX-calculus.* M.Phil. thesis, University of Oxford.

Preskill, John. 2015. *Quantum computation lecture notes.* http://theory.caltech.edu/~preskill/ph229/.

Qassim, Hammam, Pashayan, Hakop, and Gosset, David. 2021. Improved upper bounds on the stabilizer rank of magic states. *Quantum*, **5**(12), 606.

Quantinuum. *TKET.* https://www.quantinuum.com/developers/tket.

Raussendorf, R., and Briegel, H. J. 2001. A one-way quantum computer. *Physical Review Letters*, **86**, 5188.

Raussendorf, R., Harrington, J., and Goyal, K. 2007. Topological fault-tolerance in cluster state quantum computation. *New Journal of Physics*, **9**, 199.

Raussendorf, Robert, and Briegel, Hans J. 2002. Computational model underlying the one-way quantum computer. *Quantum Info. Comput.*, **2**(6), 443–486.

Raussendorf, Robert, and Harrington, Jim. 2007. Fault-Tolerant Quantum Computation with High Threshold in Two Dimensions. *Phys. Rev. Lett.*, **98**(May), 190504.

Raussendorf, Robert, Browne, Dan E., and Briegel, Hans J. 2003. Measurement-based quantum computation on cluster states. *Physical Review A*, **68**(2), 22312.

Ross, Neil J., and Selinger, Peter. 2016. Optimal Ancilla-Free Clifford+T Approximation of z-Rotations. *Quantum Information and Computation*, **16**(11–12), 901–953.

Ruiz, Francisco JR, Laakkonen, Tuomas, Bausch, Johannes, Balog, Matej, Barekatain, Mohammadamin, Heras, Francisco JH, Novikov, Alexander, Fitzpatrick, Nathan, Romera-Paredes, Bernardino, van de Wetering, John, Fawzi, Alhussein, Meichanetzidis, Konstantinos, and Kohli, Pushmeet. 2024. Quantum Circuit Optimization with AlphaTensor. *arXiv preprint arXiv:2402.14396.*

Schröder de Witt, Christian, and Zamdzhiev, Vladimir. 2014. The ZX-calculus is incomplete for quantum mechanics. Pages 285–292 of: Coecke, Bob, Hasuo, Ichiro, and Panangaden, Prakash (eds), Proceedings of the 11th workshop on *Quantum Physics and Logic,* Kyoto, Japan, 4-6th June 2014. Electronic Proceedings in Theoretical Computer Science, vol. 172. Open Publishing Association.

Schwinger, J. 1960. Unitary operator bases. *Proceedings of the National Academy of Sciences of the U.S.A.*, **46**, 570–579.

Selinger, Peter. 2010. A survey of graphical languages for monoidal categories. Pages 289–355 of: *New structures for physics.* Springer.

Selinger, Peter. 2013. Quantum circuits of T-depth one. *Physical Review A*, **87**(4), 042302.

Selinger, Peter. 2015. Efficient Clifford+T Approximation of Single-Qubit Operators. *Quantum Info. Comput.*, **15**(1–2), 159–180.

Shi, Yaoyun. 2002. *Both Toffoli and controlled-NOT need little help to do universal quantum computation*. Preprint.

Shor, P. W. 1994. Algorithms for quantum computation: discrete logarithms and factoring. Pages 124–134 of: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. IEEE.

Shor, P. W. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, **26**(5), 1484–1509.

Shor, Peter W. 1995. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, **52**(Oct), R2493–R2496.

Shor, Peter W. 1996. Fault-tolerant quantum computation. Pages 56–65 of: *Proceedings of 37th conference on foundations of computer science*. IEEE.

Simmons, Will. 2021. Relating Measurement Patterns to Circuits via Pauli Flow. Pages 50–101 of: Heunen, Chris, and Backens, Miriam (eds), *Proceedings 18th International Conference on Quantum Physics and Logic, Gdansk, Poland, and online, 7-11 June 2021*. Electronic Proceedings in Theoretical Computer Science, vol. 343. Open Publishing Association.

Sørensen, Anders, and Mølmer, Klaus. 1999. Quantum computation with ions in thermal motion. *Physical review letters*, **82**(9), 1971.

Steane, A. M. 1996a. Error Correcting Codes in Quantum Theory. *Phys. Rev. Lett.*, **77**(Jul), 793–797.

Steane, A. M. 1996b. Simple quantum error-correcting codes. *Phys. Rev. A*, **54**(Dec), 4741–4751.

Steane, Andrew M. 1997. Active stabilization, quantum computation, and quantum state synthesis. *Physical Review Letters*, **78**(11), 2252.

Suzuki, Masuo. 1991. General theory of fractal path integrals with applications to many-body theories and statistical physics. *Journal of Mathematical Physics*, **32**(2), 400–407.

Takeuchi, Yuki, Morimae, Tomoyuki, and Hayashi, Masahito. 2019. Quantum computational universality of hypergraph states with Pauli-X and Z basis measurements. *Scientific reports*, **9**(1), 1–14.

Trotter, Hale F. 1959. On the product of semi-groups of operators. *Proceedings of the American Mathematical Society*, **10**(4), 545–551.

van de Wetering, John. 2020. ZX-calculus for the working quantum computer scientist. *arXiv preprint arXiv:2012.13966*.

van de Wetering, John, and Amy, Matthew. 2024. Optimising quantum circuits is generally hard. *arXiv preprint arXiv:2310.05958*.

van de Wetering, John, Yeung, Richie, Laakkonen, Tuomas, and Kissinger, Aleks. 2024. Optimal compilation of parametrised quantum circuits. *arXiv preprint arXiv:2401.12877*.

Van den Nest, M., Dehaene, J., and De Moor, B. 2004a. Graphical description of the action of local Clifford transformations on graph states. *Physical Review A*, **69**(2), 9422.

Van Den Nest, Maarten. 2010. Classical Simulation of Quantum Computation, the Gottesman-Knill Theorem, and Slightly Beyond. *Quantum Info. Comput.*, **10**(3), 258–271.

Van den Nest, Maarten, Dehaene, Jeroen, and De Moor, Bart. 2004b. Graphical description of the action of local Clifford transformations on graph states. *Physical Review A*, **69**(2), 022316.

Vandaele, Vivien. 2024. Lower *T*-count with faster algorithms. *arXiv preprint arXiv:2407.08695*.

Vilmart, Renaud. 2019. A Near-Minimal Axiomatisation of ZX-Calculus for Pure Qubit Quantum Mechanics. Pages 1–10 of: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*.

Wang, David S., Fowler, Austin G., and Hollenberg, Lloyd C. L. 2011. Surface code quantum computing with error rates over 1 *Phys. Rev. A*, **83**(Feb), 020302.

Wang, Quanlong. 2018. A finite presentation of CNOT-dihedral operators. In: Coecke, Bob, and Kissinger, Aleks (eds), *Proceedings 14th International Conference on Quantum Physics and Logic, Nijmegen, The Netherlands, 3-7 July 2017*. Electronic Proceedings in Theoretical Computer Science, vol. 266. Open Publishing Association.

Wang, Quanlong, Yeung, Richie, and Koch, Mark. 2022. Differentiating and Integrating ZX Diagrams with Applications to Quantum Machine Learning. *arXiv preprint arXiv:2201.13250*.

Watrous, John. 2006. *Quantum computation lecture notes*. https://cs.uwaterloo.ca/~watrous/QC-notes/.

Watrous, John. 2018. *The theory of quantum information*. Cambridge university press.

Wei, Tzu-Chieh, Affleck, Ian, and Raussendorf, Robert. 2011. The 2D AKLT state is a universal quantum computational resource. *Bulletin of the American Physical Society*, **56**.

Xanadu. *PennyLane*. https://pennylane.ai/.

Zhou, Xinlan, Leung, Debbie W., and Chuang, Isaac L. 2000. Methodology for quantum logic gate construction. *Phys. Rev. A*, **62**(Oct), 052316.

Index