**An Introduction to Quantum Computing**

SIGGRAPH 2025 Course Notes
10 – 14 August, 2025
Vancouver, British Columbia

Andrew Glassner
Wētā FX

aglassner@wetafx.co.nz
www.glassner.com

These notes are adapted from my book,
*Quantum Computing: From Concepts to Code*
published by No Starch Press.


SIGGRAPH Courses '25, August 10-14, 2025, Vancouver, BC, Canada
ACM 979-8-4007-1543-3/2025/08. 10.1145/3721241.3733983

## Abstract

Computer graphics relies on computing hardware for everything from animation to rendering. An emerging new technology exploits the properties of quantum objects to offer radically new ways to think about, create, and run algorithms. For example, quantum computers can evaluate many different input values simultaneously, where "many" can be larger than the number of atoms in the visible universe. The catch is that only one output can be obtained from each run.

Quantum computing may change how we think about computer graphics. For example, future quantum computers may be able to intersect astronomical numbers of rays with a similarly massive database of objects in a single execution, or evaluate enormous numbers of simulation parameters in parallel to return the one set that produces a desired result.

This course describes, without advanced math, the core ideas of quantum computing. We start with the quantum version of the classical bit (called a qubit) and the basic operators that modify qubits. We introduce the four essential properties that distinguish quantum computers from familiar classical computers: superposition, interference, entanglement, and measurement. We show how these properties are orchestrated to build quantum algorithms.

We conclude with a brief overview of some of the most well-known quantum algorithms, and some of their possible applications in computer graphics.

Quantum computers are already here, and are increasing in size and reliability at a rapid pace. Open-source simulators abound, and small quantum computers are even available for free use on the web.

Classical hardware has served computer graphics well. Quantum computing offers a fundamentally new way to design and execute algorithms, which could change our field in fundamental ways. Now is the perfect time to starting thinking about quantum computing for graphics.

## Speaker's Bio

Andrew Glassner is a Distinguished Research Scientist at Wētā FX.

Glassner served as Papers Chair of SIGGRAPH '94, Founding Editor of the Journal of Computer Graphics Tools, and Editor-in-Chief of ACM Transactions on Graphics. Books he wrote or edited include *Principles of Digital Image Synthesis*, the *Graphics Gems* series, *An Introduction to Ray Tracing*, and *Deep Learning: A Visual Approach*. He has written and/or directed several short films, animations, and online internet games. Andrew has given many SIGGRAPH presentations that share fascinating ideas in clear and approachable ways.

Andrew holds a PhD in Computer Science from UNC Chapel Hill.

# Contents

# Section 1
# A Curious Deck of Cards

Quantum computing is exciting stuff! It stretches our brains with a bunch of new, challenging, and very cool ideas. These notes are all about getting familiar with these ideas.

This section is deliberately *not technical*. The math doesn't go past arithmetic, and there's no physics.

While you're reading this section, I encourage you to switch off the part of your brain that focuses on details, and give free rein to the wild, free-associating part of your brain that soaks up cool new ideas. We'll come back to the details later. Right now, we're after big-picture stuff. Relax and enjoy the ride.

Today's computers are called called **von Neumann machines** or **classical computers**. Their physical implementations are built around electronic devices. We like programming these devices with high-level languages that abstract away the details of the underlying hardware. We may have a general sense that there are things happening at the level of electrons, but we think about programs in terms of abstract ideas like iteration and subroutines. Those ideas all grew out of the capabilities of the hardware.

Quantum computing is based on a hardware model that has little in common with the hardware of classical computers. This hardware leads to a philosophy of solving problems that, while still algorithmic, is so radically different to conventional languages that it's essentially a new way to think about programming. For example, there are no data structures, no iteration, and no subroutines (though there might be, one day).

To keep our wits when working with these devices, we need some kind of mental model that describes what kind of things they can do, and how. In these notes, that model will be a deck of electronic playing cards.

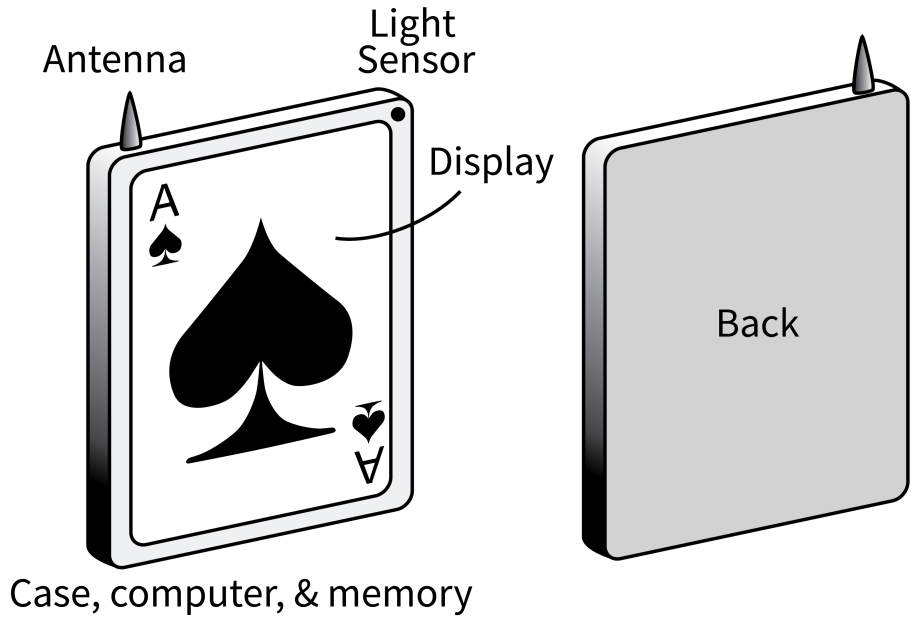


Figure 1: An electronic playing card.

# 1 An Electronic Playing Card

Our mental model of quantum computing uses computerized playing cards to stand in for quantum objects. These cards are unusual but they're not radical. You could build them right now with everyday parts if you wanted to.

These cards have four unusual properties, called **superposition**, **entanglement**, **measurement**, and **interference**. I'll refer to this quartet with the acronym SEMI. Our goal in this section is to become familiar with these four ideas by seeing how the cards work.

Each card is a little computer with a display and some other electronics, as shown in Figure 1.

Our cards will approximate real quantum behavior, but because this is only a metaphor, the match won't be perfect. And I'll skip some details. Again, we're only after a high-level overview at this point.

Every card we'll use in these notes looks like Figure 1. With an exception we'll get to at the end of the section, all the cards that are manufactured have identical hardware and software.

Each card is a thin box. Its most obvious feature is a display that takes up most of one big face. There's also a light sensor next to the display, and an antenna on top for communications. Inside the box, there's a computer, some memory embedded in the computer chip, and a battery.

Each card is shipped from the manufacturer in a standby mode, where the computer is turned off. The light sensor is powered, and in a moment we'll see that it's used to turn the computer on.

The display is unusual because it's *write-once*: once we draw a picture on the display it retains that picture forever. When we receive the cards from the manufacturer, the displays are blank.

When we first open up the box containing our cards, the light sensor notes that it's gone environment has gone from dark to light. When we later place the card face down on a table, the light sensor detects that it's no longer receiving light. The card primes itself to act when its turned face-up by waking up the computer.

At some point, when the card is turned face up, the sensor once again detects that it's receiving light. It sends another signal to the the computer reporting that the card has been turned face up. The computer responds by determining the value of the card, and then drawing the corresponding picture on the display.

To prevent cheating, we can never directly access the card's memory. Only the card's computer can read from, or write to, its internal memory. But we get some limited, one-way access through the card's antenna, with which we can send instructions to the computer telling it to manipulate the memory on our behalf. The computer will carry out any instructions we send it, but aside from the picture it draws on the display when the card is turned face-up, there's no way for the computer to report back to us anything else, including what's in the memory.

When the card has been turned face up, and the computer has drawn a picture on the display, it erases its internal memory and turns off all the electronics. The image on the display remains that way, fixed forever. From then on, the card is functionally no different than one printed on card stock.

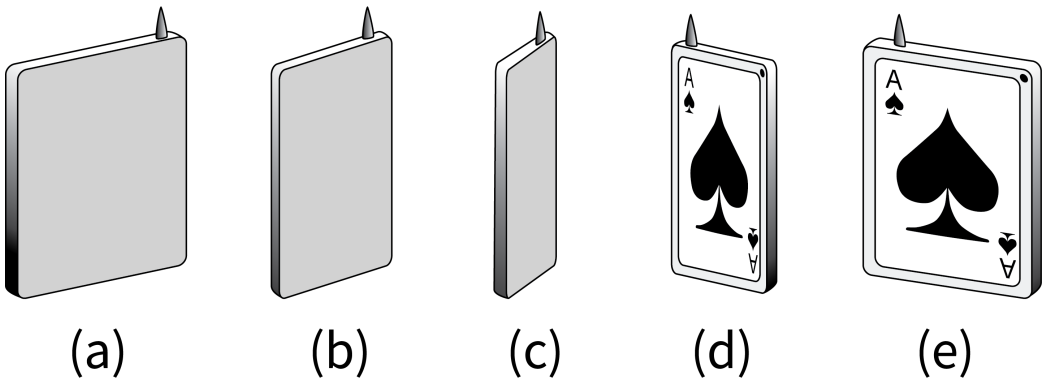


Figure 2: What happens when flip a card for the first time. We can consider all of these steps as happening sequentially and instantaneously. (a) The card is face-down. The display is blank. (b) When the card is lifts a little, the light sensor detects the change and turns on the electronics. (c) Immediately, the computer consults its internal memory and chooses one card. (d) The computer draws the card's value on the display. The computer clears the memory and turns off all electronics. (e) We see the card's value on the display.

# 2 States and Superposition

To play with these cards, we start by placing them on a table with their displays down. As we discussed, when any card is then turned face-up, the light sensor detects the change in light, and it triggers the computer inside to write the card's value to the display.

The computer determines what picture to draw based on the information in its memory.

For example, if upon being triggered the computer determines that the card is a three of diamonds, it will draw the picture for a three of diamonds on the display, and the display will stay that way forever. Figure 2 shows a visual summary of these steps. Note that that everything happens immediately as soon as any light hits the sensor, so by the time we can see the display it's already holding its final picture.

Since the computer decides what to draw to the display based on what's in the memory, let's look more closely at what gets stored there.

The memory holds a list of card values, like the three of diamonds, or

| state | probability |
|---|---|
| 3♣ | 0.5 |
| 9♦ | 0.5 |

Figure 3: A list inside the memory containing two states: the three of clubs with a probability of 0.5, and the nine of diamonds also with a probability of 0.5.

| state | probability |
|---|---|
| 3♣ | 2/3 |
| 9♦ | 1/3 |

Figure 4: A different list inside the memory containing the same two states, but with different probabilities. The three of clubs has a probability of 2/3, and the nine of diamonds has a probability of 1/3.

the nine of clubs. Each such value has an associated probability, which is a number from 0 to 1. This identifies how likely it is that, when the card is turned over, the computer will pick that value for the card.

There's a nice, more general word for "card value" that we can use to describe the "value" of any object: it's called a **state**. We say that an object that has the "value" of a state is *in* that state.

For example, suppose the list holds the three of clubs and the nine of diamonds, each with a probability of 0.5, as in Figure 3. Because the probability of each state tells us the chance that the computer will select it, and these are the only available states, the card is equally likely to end up as a three of diamonds or nine of clubs.

Alternatively, suppose the list looks like Figure 4. Here the three of clubs has a probability of 2/3, and the nine of diamonds has a probability of 1/3. That means the three is twice as likely to be chosen by the computer as the nine.

We have a special name for a list of states, each with a probability: we call it a **superposition**. The concept of a superposition is critical to

| state | probability |
|---|---|
| 3♦ | 1/3 |
| 5♠ | 1/2 |
| K♣ | 1/6 |

(a)

| state | probability |
|---|---|
| K♣ | 1/6 |
| 5♠ | 1/2 |
| 3♦ | 1/3 |

(b)

Figure 5: The order of the states (and their probabilities) in the superposition doesn't matter. The superposition in part (a) is the same as the superposition in part (b).

quantum computation, and it's used by almost every quantum algorithm. A superposition is never empty. A superposition in our cards is always a list of at least one state, but possibly many more, each with a probability. The memory in each card is used to store a single superposition (that is, a list of states and their probabilities), and nothing else.

Though we usually write a superposition as a list, the order in which the states are listed is irrelevant. This is illustrated in Figure 5.

Writing the superposition as a list necessarily requires that they're listed in some order. To get closer to the idea that the order doesn't matter, it might be helpful to think of a superposition not as a list, but as a bag full of objects representing states. These objects can have any shape we like, from slabs to balls to whales. To choose a state we close our eyes and reach into the bag. The larger a state is, the more likely we are to select it. The idea is shown in Figure 6

I'd draw superpositions as bags throughout these notes, but I think that would make them harder to take in at a glance. So while I'll use the traditional list approach, always remember the order of the items is irrelevant.

Because the computer inside the card chooses a state from the superposition at random when the card is turned over, we are fundamentally unable to predict what drawing will end up on the display. The best prediction we can make is to write out the superposition itself, identifying each possible state the card can end up in, along with the probability that state will be

Figure 6: States are like balls in a bag. We choose one at random. The bigger each state is, the more likely it is that we'll select it.

selected by the computer.

We sometimes say that before the card its turned over, its state *is* the superposition in its memory. That is, the superposition is itself the state of the card before the computer's choice is made.

Since only one state ends up on the display, it's sometimes natural to want to unwind this process and recover what the superposition was *before* we turned the card over.

But this is impossible! Remember that I said that after the computer chooses a state, it draws the corresponding picture to the display, erases the memory, and shuts itself down. So there's no way to unravel the process or figure out what the superposition was, based just on what is displayed after we turn the card over.

Let's return to a face-down card. If the card's memory holds, say, a superposition of three states (with their probabilities), it might be tempting to think of this as the card being in "all three states at once," but that would be a misleading description. If the card was in all three states, it would be showing all three pictures on the display. Instead, it's in the superposition state, which is a new kind of hybrid idea composed of these three states and their probabilities. It's a collection of possible outcomes.

When all the probabilities in a superposition are the same, we say that we're working with a **equal superposition** or **uniform superposition**.

## 2.1 Initialization

When the manufacturer makes these cards, they have to determine how to initialize each card's memory.

If they wanted to emulate traditional cards, then the manufacturer could make a deck of 52 cards where each card's memory was a superposition of just one state, with a probability of 1. If each card is initialized with a different single state in its probability, then when we turn over all the cards we'll get one of each of the 52 cards in a normal deck.

What a waste of potential! We might as well have just bought a deck of normal cards. So let's make things more interesting. By changing the initial memory, we'll make our cards unsuitable for many existing games, but we could open the door to new games.

We could initialize every card with a superposition that contains each of the card's 52 possible states, and gives each possible state the same probability of 1/52, as shown in Figure 7.


Figure 7: Each of the 52 states has a probability of 1/52.

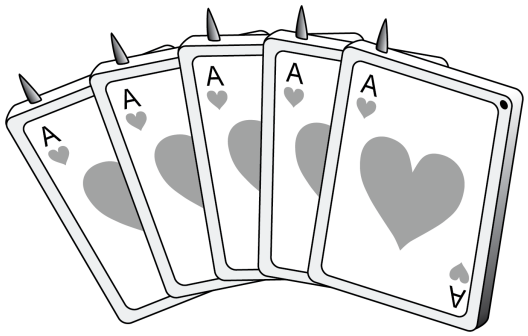

Figure 8: Even if you get this terrific hand, your opponent might also hold the same cards!

This is unlike shuffling a traditional deck because there's nothing preventing the same state being chosen by multiple cards, since each card makes its own random selection from its own memory. It's perfectly possible that if we turned over the first five of these cards, we could get, say, the jack of clubs for each one.

We'd definitely need to come up with new rules for games played with a deck initialized this way. Imagine a poker game where you reveal your winning hand is five aces of hearts, as in Figure 8, only to find your opponent has the identical hand!

For the examples in the rest of these notes, we'll use a variety of different initial superpositions.

# 3 Measurement

We'll say that when we turn over a card, we're **measuring** the card, or **making a measurement**. This term might seem odd for a card, but it will make better sense when we're using some kind of instrument to observe, or

measure, the state of a card, rather than just our eyes.

When we measure a card (that is, we turn it over), we're triggering an automatic and unavoidable process where the computer in the card consults the superposition in its memory, chooses a single state based on its probability, draws that card to the display permanently, erases the memory, and finally shuts itself down. This all happens essentially instantaneously. We can't stop the process, pause it, peek at intermediate results, or modify the steps in any way.

Drawing on the mathematical language for this process, we say that measurement causes the superposition to **collapse**. You might think of a superposition as a row of upright dominoes, each printed with a state and its probability. Then the act of measurement causes them all to fall over (or collapse) except for the single domino that's left standing.

If we're given a card and we don't know its state, its superposition will be a mystery to us, because the computer is prohibited from ever telling us what's in its memory. In Section 4 we'll see commands that we can send to the computer to get it to change the memory on our behalf, but we get no feedback on those requests.

Measurement is the only way we ever get to learn *anything* about the actual contents of a card's memory. And even then we learn only one thing, and it's about the past: we learn that the state shown on the display *was* in the memory before it was wiped clean. We don't learn about the identity of any other state in the superposition, or even if there were other such states. And we don't learn about the probabilities of any states at all, except that the one that appears on the display had a probability that wasn't 0.

The inability to read a card's memory makes it hard to check if the cards were properly initialized, or if the computer inside each card has been correctly executing our instructions. For example, suppose we get a new deck of cards where we're told that every card was initialized as in Figure 7 where the superposition holds all 52 states, each with a probability of 1/52. Then we turn over the first card and it shows the three of hearts. For all we know, that card's superposition could have actually been just that single state, with a probability of 1. Or the three of hearts could have been one of several states in the card's superposition. There's absolutely no way to know.

There is a way to check up on the manufacturer, though. We could gather up lots of cards that we believe all share the same superposition, and turn them all over. We'd expect that after enough cards have been turned over (or

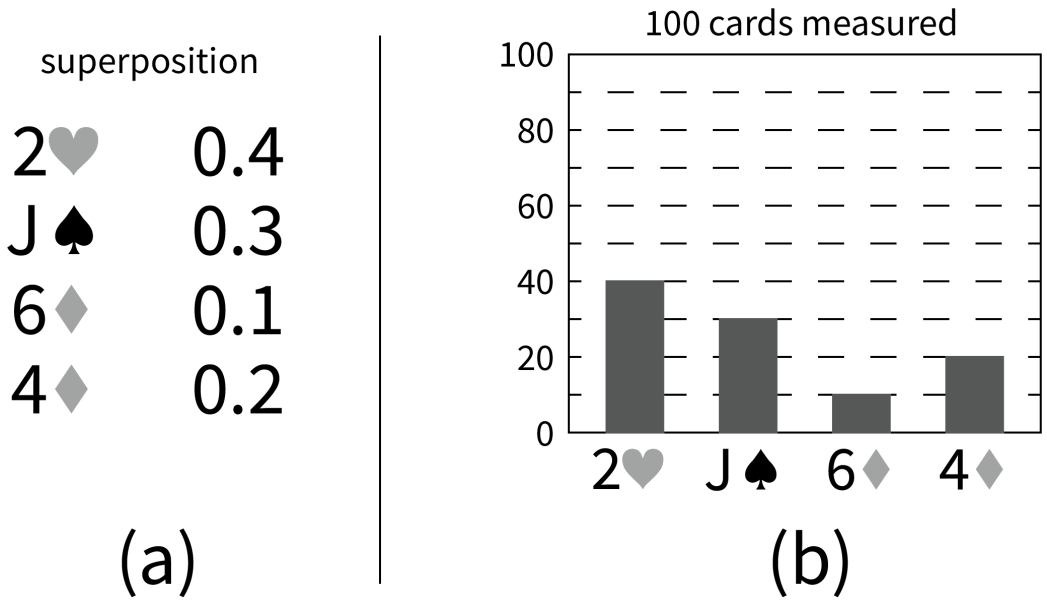


Figure 9: If we turn over many cards that were all in the same superposition, we expect that the relative numbers of each state match the probabilities in the superposition. a) The starting superposition of every card. b) The number of cards of each value observed when we turn over 100 such cards.

measured), the relative populations of each card value reflect the common superposition that all the cards started with. This is illustrated in Figure 9.

This time-consuming process is the only way we can learn more about a superposition other than observing the single state resulting from a measurement.

# 4 Operating on Cards

So far we've learned what happens when we turn over a card that was initialized by the manufacturer.

But we'd like to actually do things with our cards, like play games with them. Since we can't read the memory inside a card, we can imagine betting games based around what state will show up when a card is turned over.

Since we know the states of the cards when the manufacturer sends them

to us, to create interesting games we'll want to change their superpositions somehow. We'll do just this using the antenna on the top of each card. The antenna is connected to the computer, so we can send messages to the computer instructing it manipulate the superposition in the card's memory for us.

Imagine a game where you and a friend are taking turns sending messages to three face-down cards, changing their superpositions by a little each time. You both know the initial states that the manufacturer put them in, but each of you secretly change that superposition a little bit on each turn. Maybe your bet is that, when the cards are turned over, all the values will be less than five, and your friend is betting that they will all be diamonds. In this case, it's conceivable either or both of you could win. But if your friend is betting that all the cards will be greater than seven, at least one of you is bound to lose.

Each message that either of you sends to a card will have the name of a routine that the computer knows how to execute, perhaps followed by a list of real numbers and states that tell that routine specifically what to do.

Using the languages of mathematics and quantum mechanics, we call each of these routines an **operator**, and we say that when we tell the computer to execute one of these operators, we're **operating** on the card, or **applying an operator**. Each of the numbers and states that accompanies an operator is called an **argument**.

Let's start with an operator we'll name *include*. This takes two arguments: a state and a probability.

When the computer receives this message, it first looks in its memory to see if that state is in the current superposition. If the state isn't there, then the computer appends the state and its probability to the superposition. Figure 10 parts (a) through (c) illustrates this process.

Part (d) of Figure 10 shows an additional step that we perform after executing every operator: we divide all the probabilities by their sum after the operator was applied. The result is a new list of scaled probabilities that add up to 1. In this example, the the sum in part (c) is $0.4+0.6+0.2 = 1.2$, so we divide each probability in part (c) by 1.2 to get the final probability values in part (d). Using two fractional digits for each result, we have $0.33+0.5+0.13 \approx 1$.

This process is called **normalization**, and it's important because for the numbers in the superposition to fit the definition of being probabilities, and thus enabling the computer to choose one of them based on the rules of

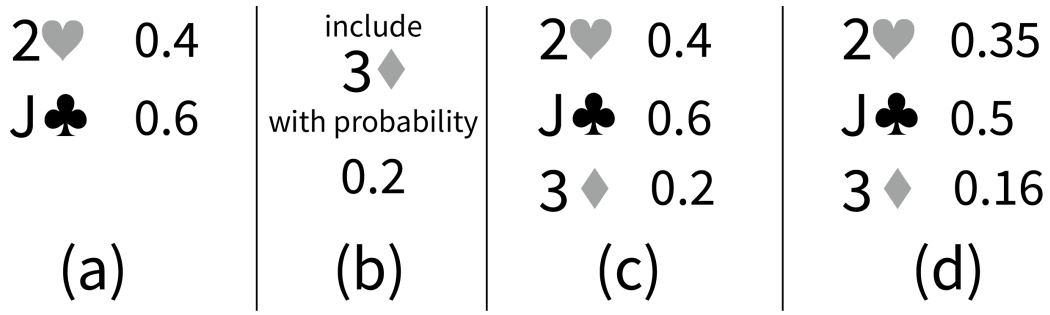


Figure 10: Including a new state and probability into the superposition. a) The superposition before the message arrives. b) The message says to include the state three of diamonds with a probability of 0.2. c) Since the three of diamonds isn't already in the superposition, the computer appends it to the list with the given probability of 0.2. d) The probabilities are then uniformly scaled so that they add up to 1.

probabilities, they have to add up to 1.

Every operation we perform from now on will be followed by a normalization step, which I'll explicitly include in the figures.

Now let's consider what happens if the state is already present. Then the computer replaces the current probability for that state with the sum of the current probability and the probability in the argument, as shown in Figure 11.

We can make up lots of operators, but we need only a few to do everything we'll be interested in.

Suppose you and a friend each start a game by guessing which value a card will have when it's turned over. Then you both get to send some number of messages, and finally you turn the card face-up and see what value it shows. What you'd like is to be sure that, when all the messages have been sent, the card's superposition will contain only the one state corresponding to the value you bet on, with a probability of 1. Then when you turn the card over, you can be sure it will display the picture for that state, giving you a victory. Of course, your friend is hoping it will show the state they bet on.

Let's look at a few more possible operators we might ask our cards to carry out for us. The more operators we have, the more different kinds of games we can invent.

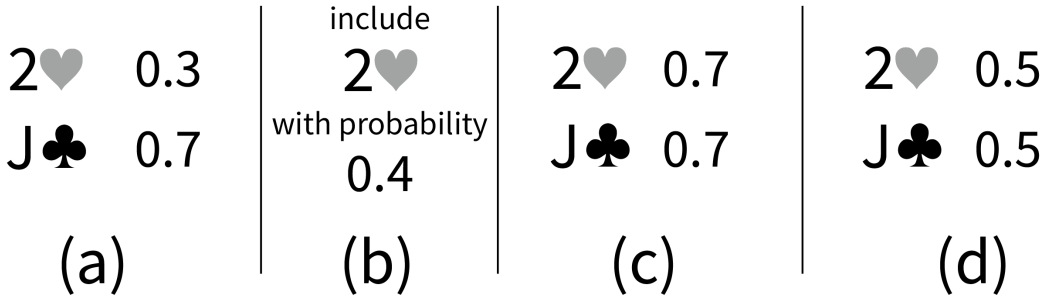


Figure 11: Including a new state and probability into the superposition. a) The superposition before the message arrives. b) The message says to include the state two of hearts with a probability of 0.4. c) Since the two of hearts is already in the superposition with a probability of 0.3, the computer replaces that probability with $0.3 + 0.4 = 0.7$. d) The probabilities are then uniformly scaled so that they add up to 1.

Let's invent an operator that we'll call *flip*. Given a single state as an argument, the computer responds by replacing the current probability for that state with 1 minus that current probability (followed by normalization, as always). If the state isn't already in the superposition, then we pretend it was there with a probability of 0, so we append that state to the list with a probability of $1 - 0 = 1$.

So if the message is "flip two-of-hearts," and the current probability for the two of hearts is, say, 0.4, then the computer will replace that 0.4 with $1 - 0.4 = 0.6$, as shown in Figure 12. As always, we follow up the operator with a normalization step.

The *flip* command specified just a single state to modify. Let's imagine a new operator called *swap*. This command takes two arguments that identify two states, telling the computer to exchange the probabilities of those two states, as shown in Figure 13. Since we've only moved the probabilities around, normalizing the superposition doesn't have any effect.

The key thing to remember about all of the manipulations we've seen so far (and others we can imagine) is that while the computer can read from its memory, and write to it, we can't do either. There is no way for us to read back the values of these probabilities, ever. Our one and only way to learn anything about the memory is to turn the card over, and then we can infer that the value shown on the display had a non-zero probability before the

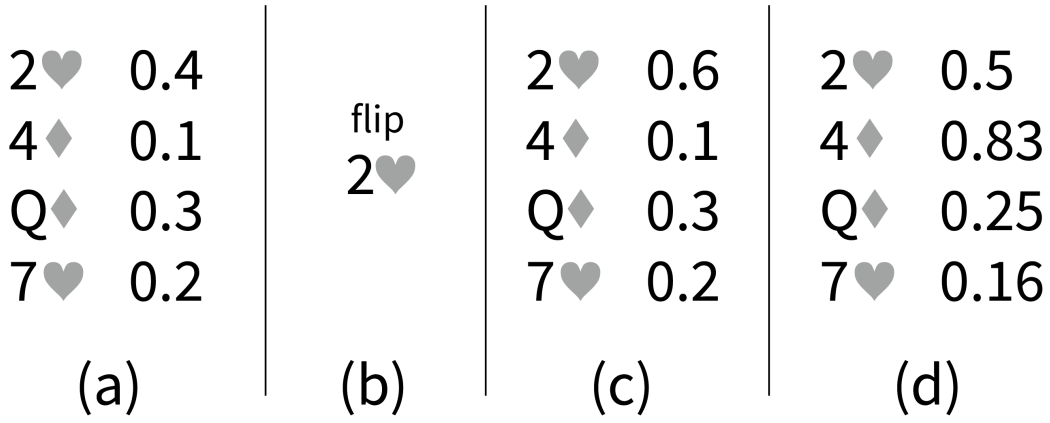


Figure 12: Carrying out the *flip* operator. a) The initial probabilities for a card. b) The instruction "flip two of hearts" arrives. c) After executing the instruction. d) After normalizing step (c).

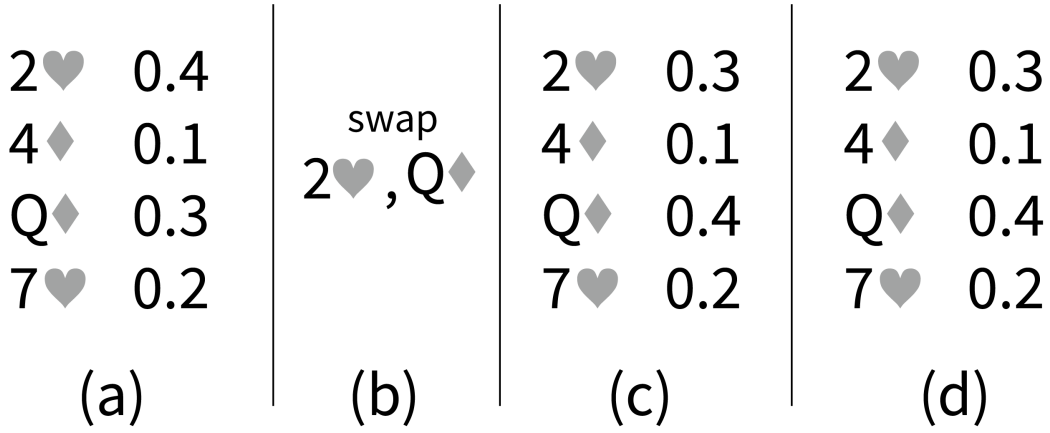


Figure 13: Carrying out the *swap* operator. a) The initial probabilities for a card. b) The instruction "swap two-of-hearts queen-of-diamonds" arrives. c) After executing the instruction. c) After normalizing step (c) (no change).

card was turned over. Otherwise, we just have to trust that our operations are being correctly carried out.

# 5 Interference

Over the course of play, we will often want to increase or decrease the probabilities of different states. We can measure the card after any operation, because we know that thanks to always performing normalization after every operation, the probabilities in the superposition are guaranteed to add up to 1. Thus the computer can legitimately use the rules of probability to randomly select one state.

But what if we decrease the value of a probability so much that it becomes negative? The normalization step we've seen doesn't do anything to address that possibility. Is that going to be a problem?

A negative probability is just fine while we're applying operators to the card, because eventually we might change that probability again and make it greater than 0. But what if that doesn't happen?

If the measurement step encounters a "probability" that isn't a value from 0 to 1, then the rules of probability are broken, and the computer won't know what to do. The measurement process will break, perhaps causing the card's computer to crash. That's not good!

We would like to be able to reduce probabilities without constantly worrying about whether they become negative.

We solve this problem by saving not the probability in our superpositions, but a different number, which we call the **amplitude**. Unlike a probability, an amplitude can be any real number: positive, negative, bigger than 1, and so on.

To find the probability of a state, multiply the amplitude times itself. In other words, the probability is the amplitude squared (sometimes we say it's the *square of the amplitude*).

This two-step mechanism means that we don't have to constantly check if the number associated with a state in a superposition is going negative, and then doing something about that, because when we multiply a real number times itself, the result is always positive.

For example, if we want a state to have a probability of, say, 0.3, then we store an amplitude of $\sqrt{0.3}$. If later, as a result of our messages to the card, that amplitude become $-\sqrt{0.3}$, it's no problem, because $(-\sqrt{0.3})^2 =$

0.3. So this mechanism of storing the amplitude, and squaring it to get the probability, ensures that all the values used by the measurement will be positive, and we're back in business.

We'll need to make a corresponding change to our normalization step, so from now on, we will say that normalization scales the *amplitudes* (no longer the probabilities) in a superposition so that their squared values add up to 1. So it's the probabilities that add up to 1, and not necessarily the amplitudes. When we refer to "normalization" and show it in our figures, we will always mean this new version.

Now we can manipulate the amplitudes freely. Normalization will make sure that the squares of the states in a superposition add up to 1 (which means none of them can be greater than 1), and it also means we always our convention that every probability is 0 or positive. The measuring process will always work and the card will never crash.

Whew.

Figure 14 shows an example of this new convention. Here we use the *include* instruction as usual, only now with an amplitude of $-0.7$. The amplitude of the two of diamonds goes from 0.1 to $-0.6$. Adding up the squares of the amplitudes (the leftmost number in column d) gives us 0.65, so we divide each squared amplitude by that amount to get the probabilities, on the right of column d. Now the squared amplitudes all add up to 1, and so are valid probabilities.

There's a special term for the action of the include operation where two amplitudes are added together. Using the language from physics that describes how waves interact, we call this process **interference**.

When the result of interference is an amplitude for a state that's less than the amplitude it started with, we call this **destructive interference**. You can think of the "destructive" term as telling us that the starting amplitude is partly destroyed, or reduced, or destructed, causing it to become less than it was. In Figure 14 it actually became negative.

We saw the opposite effect in Figure 11, where the sum of the old and new amplitudes created a new value larger than either amplitude. That's called **constructive interference**. You might think of "constructive" meaning that the values get together to "construct" a new, larger value.

By carefully adding just the right positive and negative amounts to the amplitudes for every state in the superposition, we can adjust them so that the state we want to measure (usually the answer to a problem we're trying to solve) has a large probability, and all other states have small probabilities.


Figure 14: Subtracting some probability from a state. a) The amplitudes of the superposition before the message arrives. b) The message says to include the state two of diamonds with an amplitude of $-0.7$. c) Since the two of diamonds is already in the superposition with an amplitude of 0.1, the computer adds the old amplitude and the new to get $-0.6$. d) The *squares* of the amplitudes are then uniformly divided by the square root of their sum, 0.65, so that they add up to 1.

That way, when we turn over the card, there's a very good chance we'll see our answer. The bigger the disparity in the probabilities, the more likely we are to get the result we want.

Carried to an extreme, we can completely remove a state from a superposition by using destructive interference to set that state's amplitude to 0 (the computer could literally then remove that state from the superposition in memory, or leave it there with a amplitude of 0. Either way, it have a probability of 0 and thus will never be chosen and will never show up on the display).

The trick to removing a state from a superposition is to send the card an *include* command with an amplitude that's exactly the opposite of the value it currently has, as in Figure 15.

To pull off this maneuver requires knowing the value of the state's amplitude, so we can exactly cancel it. If we know how the card was initialized, and every change we've made to it since then, we might be able to manually keep track of the superposition ourselves. But when our friend is making secret changes in addition to our changes, all we can do is guess at the amplitudes. Guessing our opponent's move is part of many games, including this one!

When we measure our card at the end of a calculation, if our desired state


Figure 15: Removing a state from a the superposition. a) The superposition before the message arrives. b) The message says to include the state nine of diamonds with an amplitude of $-0.2$. c) Since the nine of diamonds is already in the superposition with an amplitude of 0.2, the computer adds the old amplitude and the new to get $0.2 + (-0.2) = 0$. d) The amplitudes are then uniformly scaled so that their squared values add up to 1.

has a much larger probability than any other state, there's a good chance we'll observe that answer on the display.

# 6 Entanglement

Let's give our cards one last special ability.

So far, we've used the antennas to send messages to the computers inside our cards. Now we'll include a second kind of messaging, this time from one card to another.

Let's revisit Figure 7, which illustrated an initial state for a deck where every card has a superposition of all 52 states, each with a probability of 1/52. Each card in this deck is identical to every other card, so there's no way to tell one from another.

Now let's say that before sealing up this deck of cards, the manufacturer picks two cards from the deck (which two cards doesn't matter, since they're all the same), and then *links them together*.

Specifically, the manufacturer will include a routine in the software for each card of the pair that enables it to send and receive a single message with the other card in the pair. Only one kind of message can be exchanged, and it can only be sent once. The message is sent secretly, so no other card can detect that the message was even sent, let alone intercept and read it.

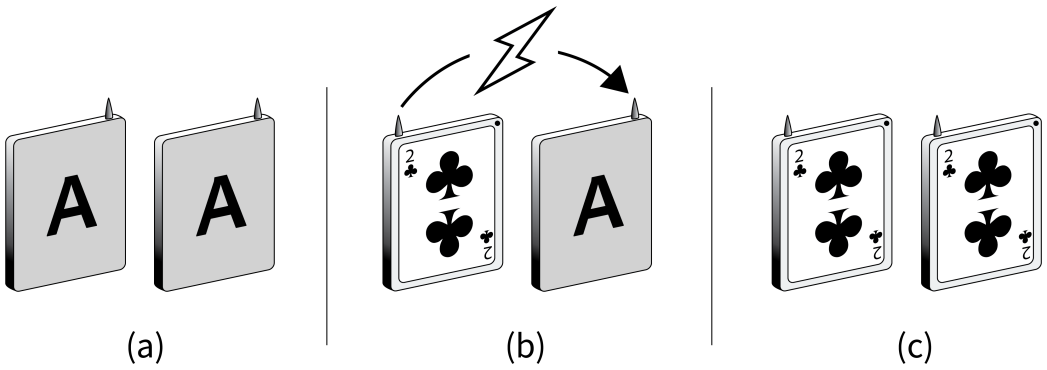


Figure 16: a) A pair of linked cards, both face down. b) When the left card is turned over to reveal the two of clubs, it sends a message to the other card reporting this event. c) If and when the other card is turned over, it will show the two of clubs as well.

The one message that can be sent between cards in a pair is this: "I have been turned over and I am showing the following state:", followed by the state that was randomly chosen and shown on the card. Both cards in the pair are able to send and receive this message, but once one cards sends it, the other never does, because there's no need. Once a card has received this message, the sending card has already determined its state.

When one card in the pair is turned over, it sends this message to the other card. When the receiving card gets this message it will do something extreme: it will erase its memory, and replace it with a superposition of a single state. That will be the state from the message, and it will get a probability of 1. This means that if and when the receiving card is turned over, it will definitely show the very same state as the first card. We say that the card will show this value **with certainty**.

Figure 16 shows the idea visually.

The two cards are otherwise identical, so there's no predetermined sender or receiver. Whichever card is turned over first becomes the sender, and the other becomes the receiver (for this discussion, we'll assume that one card is always turned over before the other).

We say that two cards linked together in this way are **entangled**.

In our everyday world, it will take some time for this message to travel from one card to the other, and then some time for the receiving card to

carry out the steps required to process the message. The farther apart the cards are from one another, the longer it will take for the message to travel, and at some point they'll be so far apart that the message will be too weak for the receiving card to pick it up.

These issues complicate the discussion without adding anything useful, so I'll follow a practical strategy and ignore them. I'll just declare from then now on, this message message gets communicated instantly, at any distance, and the receiving card carries out its entire response instantly.

The manufacturer can mark entangled pairs of cards by printing a shared mark on the back of each. For example, they could put a letter A on the backs of the first pair that they entangle, a letter B on the next pair, and so on. The order in which these labels are applied doesn't matter. Their only purpose is to let us identify pairs of entangled cards. If we don't want to know about which cards are entangled, the labels can be left off.

The manufacturer could create entangled groups of any number of cards, but we'll stick with pairs for this discussion.

Every card in the deck could, in principle, be entangled with another card, so a deck of 52 cards could contain up to 26 entangled pairs. Not all cards must be part of an entangled pair.

Recall that when both cards in an entangled pair are face down, our initialization means that every state is in the superposition with a probability of 1/52. So when the first card is turned over, it can be any of the 52 possible states. The special job of entanglement is to then change the other card so that, when it's turned over, it will definitely have the same state as the first card.

Entanglement is an intimate connection. Although each card has its own computer and memory, we can think of the pair as sharing a single superposition. When either card is measured, because they're connected, that shared superposition collapses.

If we take this point of view, then we can't really talk about the two cards as having separate identities, or separate superpositions. Because we can't talk about the superposition of one card without including the other, we think of the two cards as a single unit, or a pair that shares a superposition.

## 6.1 Entanglement In Action

Let's use entanglement to get out of a tricky situation.

Suppose you run a small casino that specializes in the card game 21. If you're not familiar with the game, here's a brief recap of the a typical version of the core gameplay.

Initially, the dealer has a face-down deck of cards and you (the player) have none. Play proceeds in independent rounds called **hands**. Let's suppose it's just you and the dealer. You start a hand by placing a bet. The dealer give you two **private cards**, dealt face down, and then gives themselves two face-down cards. You can look at your private cards, but you don't reveal them to anyone else. Your goal is to keep asking for cards to reach a total as close as you can get to 21 without going over (every number card is worth its numerical value, royalty cards are worth 10 points, and each ace can be individually either 1 or 11 at your choice, which you don't have to declare until the hand is over). To play, you can either announce that you want one more card, or that you're stopping. If you ask for another card, it's dealt face up, so we call it a **public card**. You can keep asking for more public cards, one at a time, until you decide to stop, or your total (including your secret cards) is at or over 21. If you hit exactly 21, you win immediately and get back your bet and more. If you go over 21, you lose immediately and the dealer collects your bet. If you stop before reaching 21, the dealer turns over their private cards, and then gives themselves more cards according to fixed rules, until the rules tell them to stop, they hit exactly 21, or they go over 21. If the dealer went over 21, or your total is greater than the dealer's total, you win and get back your bet and some more. There are rules for ties and lots of variations, but this is all we'll need for our discussion. Usually the game is played with one dealer and multiple players, each of whom is playing independently against the dealer. We'll say that when the hand is over, everyone turns over their two secret cards for all to see.

Your casino is special because, unlike your competitors, you use our special electronic cards. You buy the cards initialized with an equal superposition of all 52 values, so every card has an equal probability of showing, when it's looked at, any value. These unusual cards give your casino an extra spice nobody else can offer. We'll say that when a player peeks at their secret cards, that lets enough light hit the sensor to trigger the collapse into a single state that's drawn to the display As usual, only the player is able to view their secret cards.

One day you get a call from a famous, rich, but elusive player. They'd like to play at your casino next week and spend a whole lot of money. But they've recently had some bad press and they don't want to go out in public.

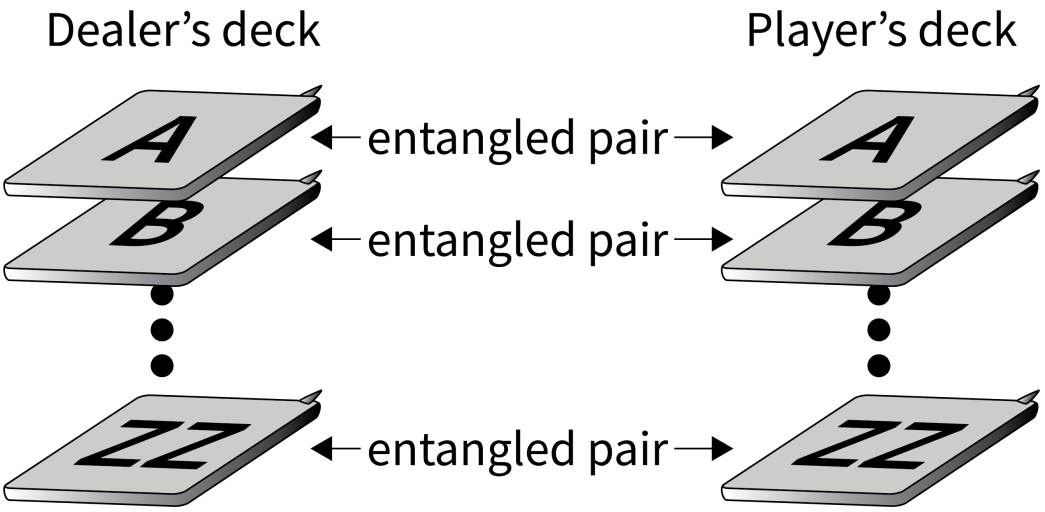


Figure 17: Two pairs of decks of cards. The pair of cards at the same position in each deck are entangled with each other.

You promise them every kind of privacy, but they aren't interested. Instead, they suggest playing from home with a two-way video system. Both you and they will have cameras pointed straight down at the cards.

You really want this player, and you have no reason not to trust them, but you've seen too many spy movies to not be suspicious of this kind of video set-up, particularly when large sums of money are involved. There are just too many opportunities for foul play.

And then it hits you: use entanglement! You can entangle the cards so that nobody has to trust anyone else. It's not a big deal to include this, since you're already using electronic cards.

You suggest to the player that you order two decks of cards from the manufacturer, just like your usual cards, except that the top card in each deck together form an entangled pair, as do the next-to-top cards, and so on. So each deck of 52 cards is card-by-card entangled with the other deck, as in Figure 17. You can order multiple such decks if you expect the game to go on for a while.

You propose that your dealer and the player each play with one of these decks. Every time the dealer deals a card from their deck, the player does the same thing with their own deck. Otherwise the game plays as usual.

Happily, the player agrees, the cards are ordered, and the game is on!

Let's see why neither party now needs to trust the other. Each hand begins with the dealer dealing two cards for the player from their deck, which they place face down. The player mirrors this action with their own deck, dealing themselves two face-down cards. Because they're electronic cards where every value is equally likely, at this point nobody can say what the values of the cards are. Now the player privately looks at their cards, causing their superpositions to collapse and the displays to show their final, randomly-chosen values. At that moment, the two face-down cards on the dealer's table *also* collapse to those same states!

The game continues this way. Each time the player asks for another card, the dealer takes one from their deck and places it on the table face up. At the same time, the player deals one card from their deck and places it face up. Thanks to entanglement, while nobody can predict what these cards will show, once turned face up they will always show the same value.

The real benefit of this comes at the end of the hand, when everyone turns over their private cards. We're *guaranteed* that they they will match! If they don't, either the cards have malfunctioned, or someone cheated. Neither party has to trust the other.

We've solved the trust problem!

Well, not quite. While we've prevented one way of cheating, we've created another.

Suppose that the player, upon receiving their deck of cards in the mail, immediately opens it up and looks at every card. Their cards all collapse to specific values, and so do all the cards in the dealer's deck, which might still be in the mail! Now the player stacks their cards back up and carefully puts them back in the box. Later, in the actual game, the player can play perfectly, since they always know what card is coming next. It's a clean cheat, because there's no way for the dealer to know that the player peeked at the cards beforehand.

In other words, when the the player looked at their deck, they not only caused their cards to collapse to individual values, they caused the dealer's cards to also collapse to those same values. There's no way for the dealer to detect this, because they can't ever get any information from a card except by turning it over and looking at it.

The moral of this tale is that entanglement is powerful, but subtle.

We can make an interesting variation on entanglement: instead of both cards showing the same state, they could show *opposite* states. Playing cards

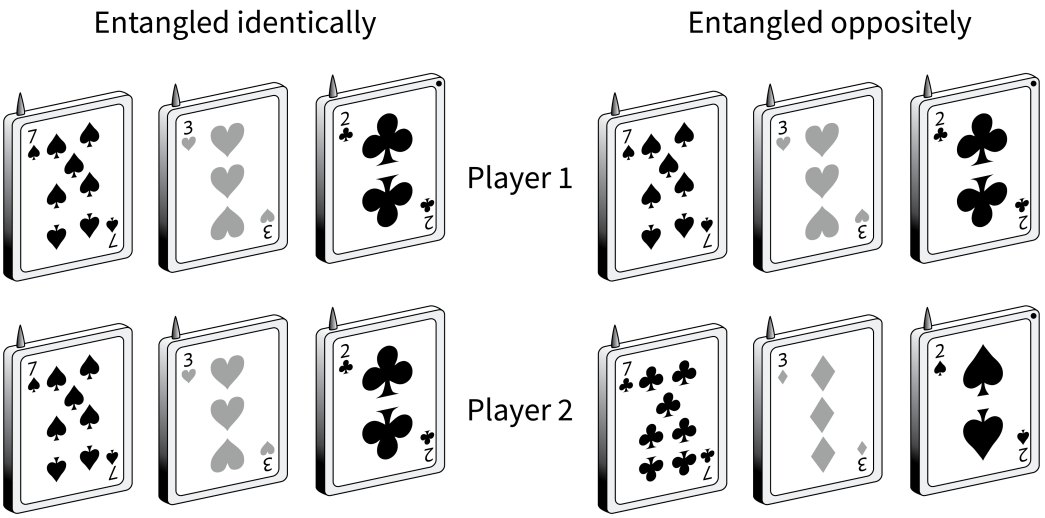


Figure 18: Two types of entanglement. Assume that Player 1 turns over their cards first. Left: If the cards are entangled identically, then Player 2 will find their cards are the same. Right: If the cards are entangled oppositely, then Player 2 will see the opposite (the other suit of the same color) of each of Player 1's cards.

don't have a natural sense of "opposite," so for for the moment let's say that the opposite of any card is the card with the same number, but with the other suit with the same color. For example, if we have the 3 of clubs, its opposite is the 3 of spades, since spades is the other black suit. In the same way, the opposite of the 8 of hearts is the 8 of diamonds.

So if the first card in a pair of entangled opposites is, say, the 7 of spades, we know that the other card, if and when we turn it over, will definitely be the 7 of clubs. The idea is shown in Figure 18.

Whether both cards of an entangled pair will show the same or opposite states is a decision made by the manufacturer when they create the cards.

We can make up lots of cool new games for these electronic playing cards. If you're feeling inspired, you can buy some chips, write some code, and build a bunch of these cards today!

# 7 Summary and Discussion

Congratulations!

You're now familiar with the fundamental ideas that make quantum computers special. The properties of these playing cards are a very close match to the properties of real quantum devices, which you'll exploit if you choose to go deeper into quantum computing.

Let's recap the properties of our cards.

We saw that each card can be characterized by a **state**, which identifies one of the 52 possible values for a playing card.

A state might also be a list of one or more other states, each with their own **amplitude**. That amplitude is squared to give a **probability** for that state. This list is called a **superposition**.

A card that is in superposition isn't in any of the states in the list, nor in all the states at once. Instead, it's in the superposition state, or a set of potential states.

There is only one way we can learn anything about the superposition in a card's memory: by **measuring** the card, or turning it over to see the image on the display.

This causes the computer to select a single state from the card's superposition, guided by the probability (or the squared amplitude) for each state listed there. The magnitude of each amplitude is squared to produce a probability for that state. The more probable the state, the more likely it will be chosen. If the card's superposition contains more than one state with a non-zero probability, than we cannot predict which one will be the result of the measurement.

All we learn from measurement is the identity of one state, and the knowledge that that state had been in the superposition with a non-zero probability.

We say that measurement causes the card's superposition state to **collapse** to a single state.

Though we cannot directly read or write to the superposition in the card's memory, we can send instructions to the computer asking it to manipulate that memory for us. These instructions are called **operators**.

One operator we saw tells the computer to include a new state (and associated probability) into the card's existing superposition. If the state is already in the superposition, the old and new probabilities are added together, and the sum is saved with that state. If the result causes the combined probability to be greater than it used to be, we call this process **constructive interference**. If the probability decreases, we call the process **destructive interference**.

Any time one or more amplitudes in a superposition are changed, we scale all the amplitudes so that the sum of their squares adds up to 1. We call this step **normalization**.

Pairs of cards can be **entangled**, so that if one card in the pair is measured, the other's superposition immediately collapses as well. If and when the second card is measured, it is guaranteed to show the same state as the first (or, depending on how the pair of cards is configured, the opposite state).

And that's it!

We've seen the essential characteristics of SEMI, the four qualities that underlie all quantum computing: **superposition**, **entanglement**, **measurement**, and **interference**. These principles are the core building blocks of how quantum objects can perform quantum computing.

When we write a quantum program, we usually start out with a superposition of all the possible states our computer can hold (which can be an astronomically large number of states). Then we apply a series of operators to those states to change their amplitudes. These operators apply to all the states in the superposition simultaneously, so we're changing huge numbers of amplitudes at every step.

The art of programming a quantum computer is in designing just what operators should be applied at each step. The overall goal is that when we finally measure our superposition and cause it to collapse to a single state, the identity of that state is useful information for us. It might be the final solution to our problem, or a useful piece in finding that solution.

We do all of this without the conventional tools we're used to, like data structures and loops. Instead, we design a series of operations. These operations can be simple, complicated, straightforward, or fiendishly subtle.

Just like classical programming!

The big advantage of the quantum approach is that it can effectively evaluate vast numbers of inputs simultaneously. While a classical computer must plod through each potential solution one at a time, a quantum computer can evaluate them all in parallel. The catch, though, is that after any given run of the computer, when we measure the output, we obtain a relatively tiny amount of information: we get back one state. Nothing more. We don't know what other states might have solved our problem, we don't know how probable the state we measured was, none of it. We just get the state.

In practice, we therefore usually run quantum algorithms many times and use the distribution of the states we measure to tell us more about what the machine computed.

Because of their special SEMI properties, and their unusual approach to programming, quantum computers offer an exciting, radically new way to think about algorithms and computing.

# Section 2
# Deutsch's Algorithm

This section of our course notes presents an informal mathematical treatment of a famous, small quantum algorithm. My goal is to show you what these algorithms look like.

Because these are course notes, and not a book-length discussion, the presentation here is deliberately informal mathematically. Nothing here is incorrect, but I'll skip over background information, context, proofs, and so on. Like Section 1, the intent is to share the big picture and structure of a quantum algorithm, without digging into all the details and mathematical infrastructure.

If you want to dig into those details, that's great! Almost everything we do in quantum computing is based on linear algebra, which you probably are already familiar with from your computer graphics work. There's some new notation and a bunch of conventions to get used to, but you won't find yourself getting lost in strange math you've never encountered before. You can find all this information in the references listed at the end of these notes.

For now, set rigor aside in favor of the overall technique.

A couple of words of language. Just as a **bit** is the fundamental unit of classical computation, the **qubit** (pronounced KYOO-bit) is the fundamental unit of quantum computation. A quantum algorithm usually starts with one more qubits that start out in a standard superposition of a single state. We then modify that superposition repeatedly, perhaps expanding it to include more states or using interference to remove some states, changing the amplitudes at each step. Finally, we measure the qubits.

When measured, qubits return one of a fixed number of return values. Surprisingly, there are only two such output values. Regardless of the technology that implements the quantum hardware, we give these two return values the names 0 and 1, establishing a close connection with conventional computing.

The bottom line is that when our computation is done and we measure our qubits, we get back either a 0 or 1 from each qubit. Taken together, those results give us a binary number. That number is the output of the algorithm.

| name | $f(0)$ | $f(1)$ |
|---|---|---|
| $f_a$ | 0 | 0 |
| $f_b$ | 0 | 1 |
| $f_c$ | 1 | 0 |
| $f_d$ | 1 | 1 |

Table 1: The four possible one-bit functions. The left column is the function name. The middle column is that function's output for an input of 0, and the right column is the output for an input of 1.

# 1 Deutsch's Problem

One of the first quantum algorithms was designed to solve a toy problem, which we call the **balanced or constant problem**. It's also called **Deutsch's problem**, named for David Deutsch who both invented the problem and published a quantum algorithm for solving it [Deutsch, 1985].

This is a contrived, simple problem. Its value is not that it solves something we need practically, but in how it demonstrates the structure of a quantum algorithm as a series of steps that transform the amplitudes of the states in a superposition.

Here's the problem we'd like to solve. Suppose that a friend provides us with a tiny circuit that computes a function that takes a single classical bit as input, and produces a single bit as output.

There are only four possible functions, summarized in Table 1.

We say that $f_a$ and $f_d$ are *constant*, since they give the same output for both inputs. We call $f_b$ and $f_c$ *balanced*, since they return 0 for one input and 1 for the other.

Our friend can choose any one of these four functions, but they won't tell us which. They give us the circuit, but they wrap it up in some way so that we can't peer inside and see what it does. All we can do is give it an input, and receive back its output.

It's reasonable to expect that we'd like to determine which function our friend chose, but our goal is more modest. We don't care which function they gave us. Our quest is only to identify if it's one of the two constant functions, or one of the two balanced functions. Specifically which function they gave us is a question for another day.

We say that our goal is to determine which **class** of function they chose.

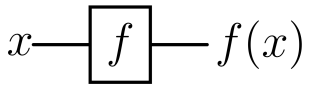


Figure 1: Writing $f$ graphically.

This problem is too simple to have any practical value, but that very simplicity makes it a great way to look at a real quantum algorithm.

A diagram of this function is shown in Figure 1.

Before we get into anything quantum, let's see how to answer the question on a traditional, or *classical* computer. Let's start by arbitrarily choosing to evaluate the function first with the input 0, and then save the output. Based on this single execution of their secret function, which we call a single **query** of the function, we can't say yet if the function is constant or balanced. After all, evaluating it with 1 could return either 0 or 1. So we'd need to make two queries to the function to answer the question, once with an input of 0 and then again with an input of 1. Now we know everything about $f$ and we can tell which class it belongs to. There's just no way to come up with the answer with fewer than two queries.

Unless you have a quantum computer! Deutsch's algorithm gives us the answer with *one* query of the function!

There are two key principles behind this feat. One is the delicately choreographed dance of superposition and interference that makes up the algorithm. The second is the magic of **quantum parallelism**. Let's see what that's all about.

The input to the function will be a superposition of both inputs, 0 and 1. Because of how quantum systems work, the function will evaluate *both* inputs, 0 and 1, independently and in parallel (whether it actually evaluates both outputs, or only *seems* to do so, is an open question that we'll come back to later. For these notes, I'll assume that both inputs are indeed evaluated simultaneously).

The output of the function will also be a superposition of $f(0)$ and $f(1)$. Then we'll use interference to combine the two states in the superposition. If $f$ is constant, some states in the superposition will interfere constructively, or double up their influence, and others will interfere destructively, or cancel each other out. If $f$ is balanced, a different set of states interfere constructively, and a different set of states interfere destructively. So the surviving

states depend on the nature of $f$. When we ultimately measure the output, we'll get back a result of 0 or 1, corresponding to a constant or balanced version of $f$.

To recap, we'll give the function a superposition of both inputs, and get back a superposition of both outputs. Applying interference, some states will interfere constructively and others will interfere destructively. The remaining state will, when we finally make a measurement, give us a result of either 0 or 1, corresponding to the function being constant or balanced.

# 2 Oracles

Now we come to one of the key rules of how we manipulate quantum states: every change must be **invertible**. If we know the output of some operation, we must always be able to precisely determine the input.

Nobody can say exactly why this is the case. It's a property of nature. In fact, it's one of the assumptions, or postulates, of quantum mechanics, the theory that quantum computing is built upon. This is simply how our universe works, like the way gravity behaves or how nothing can move faster than light.

Unfortunately, a "black box" that implements one of the four functions in Table 1 is not invertible (we also say it's not **reversible**). Let's say that we know the output of the black box is 0. What was the input? There's no way to know. Since the box could be implementing any of the four functions, the input could have been 0 or 1.

Nature requires that every step in a quantum computer must be invertible. Can we somehow change the black box so it can be inverted?

Perhaps the easiest way to do that is to add a second output to the black box that outputs the input value. So now we have two outputs: one that holds the function's input, and one that holds the function's output.

If we're going to be reversible, a system with two outputs needs to also have two inputs. It's just conservation of information: if two bits come out, then if we reverse the system and use those two bits as inputs, we need two bits on the other end. Let's call this second input $y$, and we'll make sure that our friend's circuit has two inputs and two outputs. Let's draw this new box as in Figure 2.

But now we have a new problem, because to be reversible we'll need to have $y$ also appear as an output, since it's an input, so we'll need three inputs,

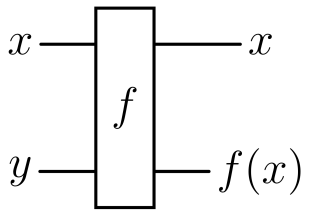


Figure 2: Including a second input

| $x$ | $y$ | $x \oplus y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Table 2: The exclusive-or (XOR) truth table. For any two input bits $x$ and $y$, the rightmost column shows their XOR, written $x \oplus y$.

requiring another output, requiring another input, requiring another output, and we'll never be done.

There's a clever solution to this that uses the **XOR** (or **exclusive-or**) operation. This is a two-bit operation that returns a 0 if both inputs are the same, and 1 otherwise. For input bits $x$ and $y$, we write the XOR as $x \oplus y$. The truth table for XOR is shown in Table 2.

Instead of delivering $f(x)$ to the output, let's set up the black box to deliver the XOR of $y$ and $f(x)$, which we can write as $y \oplus f(x)$. Now the system looks like Figure 3.

Suppose we place a second copy of this black box immediately after the

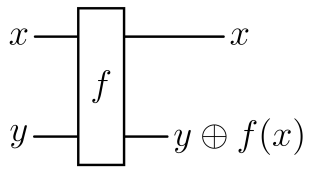


Figure 3: Improving the second output.

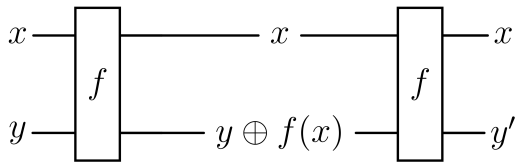


Figure 4: Repeating the black box.

first, as in Figure 4. The lower output of the first black box, as we've seen, is $y \oplus f(x)$. The second black box receives this as its $y$ input, and that is XOR'd with $f(x)$ again. The lower output, $y'$, will be the same as the original $y$, as shown in Equation 1, where the third and fourth steps use Table 2.

$$\begin{aligned} y' &= (y \oplus f(x)) \oplus f(x) && \text{Apply } f \text{ twice} \\ &= y \oplus (f(x) \oplus f(x)) && \text{Because XOR is associative} \\ &= y \oplus 0 && \text{The XOR } a \oplus a = 0 \text{ for any } a \\ &= y && \text{The XOR } a \oplus 0 = a \text{ for any } a \end{aligned} \tag{1}$$

This confirms that the black box version of $f$ in Figure 3 is reversible.

In quantum computing, this kind of black box is often called an **oracle**. This means its a function that we can provide with inputs and produces outputs, but we can know nothing more about it. We can use this language to say that our friend provides an oracle that takes in two inputs, $x$ and $y$. The first output is $x$, and the second is $y \oplus f(x)$.

Since $y$ is a bit, it's either 0 or 1. The oracle is reversible for both inputs. We usually assume that it starts out as 0.

Now let's take all of this and move into the quantum realm.

# 3 Qubits

We're going to replace the bits in the last section with quantum bits, or **qubits**. Like classical bits, when we measure a qubit we get one of two values. They could have been given any names, like hot and cold, or green and blue, but happily the two quantum results were given the familiar names 0 and 1.

We'd like to mark our inputs and outputs as quantum states. That is, rather than write $x$ or $y$, which we've been using to refer to bits, we'd like

to write *something-x* and *something-y* to emphasize that these are quantum bits, or qubits, which can represent much more information than just 0 or 1 (though they always collapse to either 0 or 1 when measured).

We identify a symbol as being a qubit with a new set of symbols that belong to a system called **braket notation**, or **Dirac notation** after Paul Dirac, who invented it [Dirac, 1939]. The part of Dirac notation we'll use here is to mark a quantum state by placing it between a vertical bar and a horizontally-compressed greater-than sign. So the quantum version of $x$ is $|x\rangle$, and the quantum version of $y$ is $|y\rangle$.

Suppose we have a large number of qubits that we're guaranteed to all be in the same superposition, and we measure them all. If we get back 0 every time, we'd conclude that those qubits were in a superposition that contained only a single state that always led to a measurement of 0. We'll call that state $|0\rangle$. Similarly, if every measurement is 1, then we'd conclude that every qubit was in a superposition of only the one state that always led to a measurement of 1. We'll call that state $|1\rangle$.

We say that if a qubit is in the state $|0\rangle$, then we'll measure a value of 0 **with certainty**. In the same way, when we measure a qubit in the state $|1\rangle$, we will, with certainty, get back a value of 1.

Just as classical bits can be implemented with little magnetic donuts, or moving charges in a chip, or even ping-pong balls on mousetraps, so too can qubits be implemented in many different ways. So don't get hung up on what a qubit might be inside one particular quantum device or another. Just think of it as something – perhaps something abstract – that is described by a quantum state.

We call these quantum symbols **kets** (this name is part of a pun that emerges when we use these symbols in more complicated ways). So $|x\rangle$ is a ket, pronounced "ket-x," and $|1\rangle$ is also a ket, pronounced "ket-one." Again, the symbols that mark a ket are just there to tell us that the object between the bar and angle is a quantum superposition.

The inputs and outputs of the oracle are all quantum states, so we should write them as kets. The quantum version of Figure 3 is shown in Figure 5. The only difference is that I've written the inputs and outputs as quantum states, rather than bits.

You might have noticed that I wrote the output of the oracle as $|y \oplus f(x)\rangle$. Shouldn't there be some kets inside of there, since the terms are $|y\rangle$ and $|f(x)\rangle$? The convention is that we can avoid clutter by writing one ket surrounding the whole expression. This convention also lets us write the

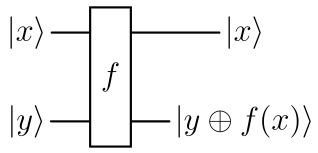


Figure 5: Writing Figure 3 with quantum states going in and out.

output as $f(x)$, referring to just $x$, rather than using $|x\rangle$ and writing $f(|x\rangle)$. The idea is that when the input $|x\rangle$ is $|0\rangle$, we treat it as the bit 0 and produce the output $f(0)$, and when the input is $|1\rangle$ we produce $f(1)$. This widespread and conventional notational agreement lets us cut down on a lot of clutter.

That wraps up the preliminaries. Now we can dig into how Deutsch's algorithm solves our problem with only a single query to the oracle.

# 4 Deutsch's Algorithm

The balanced or constant problem was invented by David Deutsch, who also invented the quantum solution for it. This solution is now known as **Deutsch's algorithm** after its creator.

Deutsch's algorithm uses three of the four properties that we discussed earlier: superposition, interference, and measurement.

Quantum algorithms can be written with text, like most programming languages that we're used to. But more frequently they're represented graphically. The drawings are called **circuits** because of their resemblance to a schematic for an electrical or electronic circuit. We often use the word "circuit" as a synonym for "algorithm."

For our purposes, a quantum circuit consists of a set of horizontal lines, each representing a single qubit. The lines are read left to right, so each qubit begins with an initial value at the far left, and ends up with a final quantum state at the far right (or until it gets measured, as we'll see below). There are rules on how to construct these circuits so that they can be realized on actual quantum hardware, but we won't go into those rules here. Every quantum circuit we'll discuss will be a legitimate system that can be realized on hardware.

To affect the amplitudes of qubits we draw boxes along these lines, such that one or more qubits enter each box on its left and the same number of

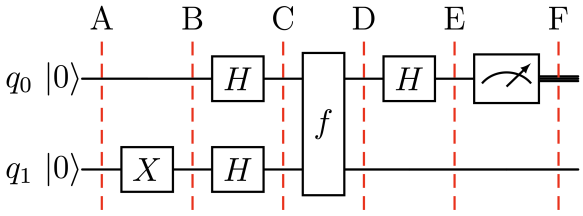


Figure 6: Deutsch's Algorithm. The dashed vertical lines and their labels are decorations to help us discuss the values of the qubits at different steps along the algorithm.

qubits exit the box on its right. These boxes represent what mathematicians call **operators**, which in quantum computing we often call **qugates** (they're often called **gates**, but I don't like that for two reasons. First, the term is too easily confused with classical gates, like XOR above. Second, it's just consistent language. Quantum versions of bits are called qubits, so quantum versions of gates should be called qugates). With this language, we can say that the oracle in Figure 5 is a two-qubit qugate.

Deutsch's algorithm makes use of three types of qugates: the two-qubit oracle given to us by our friend, and two single-qubit qugates called $X$ and $H$, which we'll describe in a moment. The whole algorithm, or circuit, is shown in Figure 6.

Let's look at the pieces, and then we'll put them together.

## 4.1 What the Qugates Do

Let's look at the icon in the upper right first, on the upper line between markers $E$ and $F$. This is called a **meter**, and it represents the act of measuring a qubit. The result of the meter for us is a classical bit (either 0 or 1). We signify that this output is a classical bit by using a double line, to distinguish it from the single lines elsewhere in the circuit that carry quantum bits, or qubits. The output of this meter will be the output of our circuit, telling us whether the oracle provided by our friend belongs to the constant or balanced categories.

Curiously, we can (and do) ignore the final value of the lower qubit. We could measure it if we liked, but everything we want to learn from this circuit

is given to us by the upper qubit. We sometimes use the term **auxiliary qubit** to refer to a qubit like the lower one in this figure, which is part of the algorithm but not needed for the output (a deprecated term that you'll sometimes see is **ancillary qubit**).

Deutsch's algorithm uses a single $X$ qugate, which is also called the $NOT$ qugate. It's the quantum version of the classical **inverter**. If its input is the quantum state $|0\rangle$, then the output of $X$ (written $X|0\rangle$) will be $|1\rangle$. Similarly, $X|1\rangle = |0\rangle$.

The other qugate is called $H$, which stands for **Hadamard**, named for Jaques Hadamard [Wikipedia, 2022]. The Hadamard qugate is perhaps the most important and interesting qugate in quantum computing.

When the input to an $H$ qugate is $|0\rangle$, the qugate turns this into an equal superposition of both $|0\rangle$ and $|1\rangle$. We can write this result in Equation 2.

$$H|0\rangle = \vee|0\rangle + \vee|1\rangle = \vee(|0\rangle + |1\rangle) \tag{2}$$

Here, the symbol $\vee$ refers to $1/\sqrt{2}$. Remember that the probability of measuring a state from a superposition is given by the square of the magnitude of its amplitude. Since $H$ creates a superposition where both $|0\rangle$ and $|1\rangle$ have the same probability, they must have the same amplitudes. If two identical numbers add up to 1 (as probabilities must), those probabilities must each be $1/2$. Thus their corresponding amplitudes are $\sqrt{1/2}$, to which I've assigned the symbol $\vee$. The symbol $\vee$ is a lot easier to read than having lots of $\sqrt{1/2}$ or $1/\sqrt{2}$ terms everywhere. This formal definition is in Equation 3.

$$\vee \triangleq \frac{1}{\sqrt{2}} \tag{3}$$

The output in Equation 2 crops up frequently, so it's useful to have a little shorthand for it. We write $\vee(|0\rangle + |1\rangle)$ with the ket $|+\rangle$.

When the input to a Hadamard gate is $|1\rangle$, its output is a little different: we get the difference between $|0\rangle$ and $|1\rangle$, rather than their sum, as shown in Equation 4.

$$H|1\rangle = \vee|0\rangle - \vee|1\rangle = \vee(|0\rangle - |1\rangle) \tag{4}$$

The only difference between $H|0\rangle$ and $H|1\rangle$ is that $|1\rangle$ is added in the first case, and subtracted in the second.

Just as $H|0\rangle = |+\rangle$, we have a shorthand for $\vee(|0\rangle - |1\rangle)$, which we write as $|-\rangle$.

| | |
|---|---|
| | $H\|0\rangle = \|+\rangle = \vee(\|0\rangle + \|1\rangle)$ |
| $X\|0\rangle = \|1\rangle$ | $H\|1\rangle = \|-\rangle = \vee(\|0\rangle - \|1\rangle)$ |
| $X\|1\rangle = \|0\rangle$ | $H\|+\rangle = \|0\rangle$ |
| | $H\|-\rangle = \|1\rangle$ |

Table 3: The outputs of $X$ and $H$ for inputs $|0\rangle$ and $|1\rangle$.

I said earlier that every qugate must be invertible, and both the $X$ and $H$ qugates satisfy this requirement. They're even nicer, in that each of these qugate is its own inverse. So for any input $|x\rangle$, applying $X$ twice, or $X(X|x\rangle)$, gives us back $|x\rangle$. So $X(X|0\rangle) = X|1\rangle = |0\rangle$, and $X(X|1\rangle) = X|0\rangle = |1\rangle$. Similarly, $H(H|0\rangle) = H|+\rangle = |0\rangle$, and $H(H|1\rangle) = H|-\rangle = |1\rangle$.

There's nothing magical about how either $X$ or $H$ work. These aren't mysterious gifts of nature. They're things we humans invented and defined so that they produces these outputs. Happily, they're also things we can implement in real hardware, so both the $X$ and $H$ qugates are theoretical ideas and practical components in a quantum circuit.

Let's collect the properties of both of these qugates in Table 3. We won't need the outputs of $X$ for inputs $|+\rangle$ and $|-\rangle$, so for simplicity I've left them out.

I've described both $X$ and $H$ in terms of how they process a qubit that's either just $|0\rangle$ or just $|1\rangle$. But we know that there will be superpositions around, because $H$ makes them. What happens when these qugates are presented with a superposition as input?

The answer here has to come from experiments, not math. The math is nothing more than how we write down what we believe is going on in the real world. Any prediction made by the math has to be checked, and if it's wrong, then the math has to change. So let's describe what actually happens when we apply a superposition to these qugates.

When we give $X$ a superposition, like that produced by $H|0\rangle$, it does something amazing. *It seems to process both states in the superposition simultaneously and independently.* That is, it processes each state **in parallel**. We call this **quantum parallelism**. The output is a new superposition resulting from applying the operation of the qugate to each input independently.

This always happens, regardless of whether the input superposition consists of two states, or two billion, or more states than there are atoms in

the universe [Baker, 2021]. Every state in the superposition gets processed simultaneously, boom, in the time it takes to process one of them, and all the results appear in a single new superposition.

Nobody is sure exactly how this happens (or even if it actually *does* happen, or it just *seems* to happen). What we do know is that this describes the result. There are lots of theories about how qugates manage to do this parallel processing, and more theories are coming up all the time. Feel free to invent one yourself if you like. Maybe the qugate splits the universe into two, processes each state in the superposition in its own universe, and then merges the universes together again [Everett III, 1957]. Sure, maybe. Or maybe the qugate manipulates time so that nothing actually gets computed until we make a measurement, and then time goes backwards so that the qugate computes only the state in the superposition that corresponds to the value we measured [Lesovik et al., 2017]. Sure, maybe that's right instead.

It's lots of fun to think about, and since nobody knows the real answer, you can let your imagination run wild.

But what we do know, in practical terms, is that we can *interpret* the action of a qugate *as if* it's literally processing every state in the superposition, and presenting a new superposition of the results.

Let's write this mathematically for the $X$ qugate in Equation 5. I'll write out each step.

$$
\begin{aligned}
& X(\vee|0\rangle + \vee|1\rangle) && \\
=& X \vee |0\rangle + X \vee |1\rangle && \text{Distribute } X \text{ over the superposition} \\
=& \vee X|0\rangle + \vee X|1\rangle && \text{Move the number } \vee \\
=& \vee |1\rangle + \vee|0\rangle && \text{Apply } X \text{ to each state independently} \\
=& \vee (|1\rangle + |0\rangle) && \text{Collect the } \vee \\
=& \vee (|0\rangle + |1\rangle) && \text{The order of states doesn't matter}
\end{aligned}
\tag{5}
$$

The amazing thing is that the math correctly matches the experiments. Applying $X$ to the superposition $H|0\rangle$, as in Equation 5, gives us back an equal superposition of both outputs.

In this case, both input states had equal amplitudes, so the output of $X$ is the same as its input. If the amplitudes were different, $X$ would swap them. For example, if the input superposition was $\sqrt{1/3}|0\rangle + \sqrt{2/3}|1\rangle$, the result of applying $X$ would be $\sqrt{2/3}|0\rangle + \sqrt{1/3}|1\rangle$ (we need the square roots because the the squared amplitudes are probabilities, and thus must sum to

1).

So the math and the experiments agree that applying a superposition to $X$ produces a new superposition consisting of applying $X$ to each of the input states simultaneously and independently. The output states have the same amplitudes as their corresponding inputs.

This remarkable behavior isn't limited to $X$. *All qugates work this way.*

What happens if we apply $H$ to a superposition? Given the principle I just stated, we should expect an output superposition of $H$ applied to each state in the input superposition.

Let's work this out step by step for the superposition $H|0\rangle$ in Equation 6. All we need are the definitions of $H|0\rangle$ and $H|1\rangle$ from Table 3 above.

$$
\begin{aligned}
& H(H|0\rangle) && \text{Apply } H \text{ to } H|0\rangle \\
=& H(\vee|0\rangle + \vee|1\rangle) && \text{Expand } H|0\rangle \text{ with Table 3} \\
=& \vee H|0\rangle + \vee H|1\rangle && \text{Distribute } H \text{ and move } \vee \\
=& \vee \left[ \vee (|0\rangle + |1\rangle) \right] + \vee \left[ \vee (|0\rangle - |1\rangle) \right] && \text{Use Table 3} \\
=& \vee^2 |0\rangle + \vee^2 |1\rangle + \vee^2 |0\rangle - \vee^2 |1\rangle && \text{Expand the terms} \\
=& 2 \vee^2 |0\rangle && \text{(!) Because } \vee^2|1\rangle - \vee^2|1\rangle = 0 \\
=& 2\frac{1}{2}|0\rangle && \text{Using } \vee^2 = 1/2 \\
=& |0\rangle
\end{aligned}
\tag{6}
$$

Line 6, marked with a bang, is extra super important and fascinating. *This is interference in action.* The state $|1\rangle$ appears twice, first with an amplitude of $\vee^2$ coming from $H|0\rangle$, and then second with an amplitude of $-\vee^2$ coming from $H|1\rangle$. The amplitudes $\vee^2 - \vee^2 = 0$ combine to produce **total destructive interference**, causing the state $|1\rangle$ to end up with an amplitude of 0, and thus *completely disappear from the superposition*!

What's left is $|0\rangle$ with an amplitude of $\vee^2$ from both superpositions. Those amplitudes combine *constructively* into $2\vee^2 = 1$, giving us a result of $|0\rangle$.

This math again matches experimental reality. If we start with a qubit in state $|0\rangle$, apply an $H$ qugate, apply an $H$ qugate again, and measure the result, we get 0 every time, confirming that the state going into the measurement is $|0\rangle$ and nothing else.

The qugate $H$ is remarkable: it both *creates* and *destroys* superpositions!

For completeness, I'll repeat Equation 6 again, step by step, but this time I'll start with $|1\rangle$. The result is in Equation 7.

$$
\begin{aligned}
& H(H|1\rangle) && \text{Apply } H \text{ to } H|1\rangle \\
=& H(\vee|0\rangle - \vee|1\rangle) && \text{Expand } H|1\rangle \text{ with Table 3} \\
=& \vee H|0\rangle - \vee H|1\rangle && \text{Distribute } H \text{ and move } \vee \\
=& \vee \big[\vee(|0\rangle + |1\rangle)\big] - \vee\big[\vee(|0\rangle - |1\rangle)\big] && \text{Use Table 3} \\
=& \vee^2|0\rangle + \vee^2|1\rangle - \vee^2|0\rangle + \vee^2|1\rangle && \text{Expand the terms} \\
=& 2\vee^2|1\rangle && \text{(!) Because } \vee^2|0\rangle - \vee^2|0\rangle = 0 \\
=& 2\frac{1}{2}|1\rangle && \text{Using } \vee^2 = 1/2 \\
=& |1\rangle
\end{aligned}
\tag{7}
$$

So the math is telling us that if we apply $H$ to $|1\rangle$, creating a superposition, and then apply $H$ to this superposition, we should get back $|1\rangle$. And indeed, that's just what happens in real experiments.

This time, it's state $|0\rangle$ that undergoes total destructive interference and ends up with an amplitude of 0, causing it to disappear from the superposition, and $|1\rangle$ goes through constructive interference to end up with an amplitude of 1.

## 4.2 Putting it All Together

Now we're ready to see how Deutsch's algorithm in Figure 6 solves our problem in a single query. I'll use the vertical dashed lines named A through E to identify the values of the two qubits at different times along the circuit. For convenience, I'll repeat Figure 6 here as Figure 7.

Deutsch's algorithm involves two qubits, and so far we've only seen one. How do we write this combined system of two qubits? The answer involves a mathematical technique we haven't discussed yet. Using that technique is important in more complicated circuits, but we can avoid a big digression and take a more casual approach for now. I'll write a two qubit system by writing the two kets side by side, placing the upper qubit on the left and the lower qubit on the right. So if at some point the upper qubit is $|0\rangle$ and the lower qubit is $|1\rangle$, I'll write the system as $|0\rangle|1\rangle$. For this discussion, we can treat this as having the usual properties of two objects multiplied together

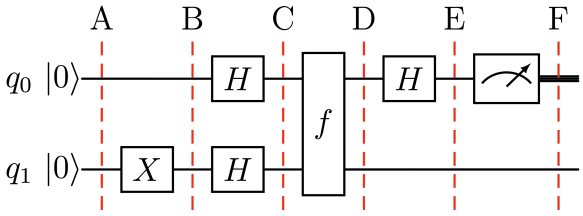


Figure 7: Deutsch's Algorithm. The dashed vertical lines and their labels are decorations to help us discuss the values of the qubits at different steps along the algorithm.

(I'm cutting a pretty big mathematical corner here, and while this approach is fine for this discussion, when we "combine qubits" in general we need to use a different and more complicated mechanism called the *tensor product*).

To reduce clutter, I'll write the math that follows without being overly strict or formal, so we don't have to digress into technicalities. In particular, I'll apply qugates to qubits in equations to match what we're doing in the drawing of the circuit. This is a pretty informal way to go about things, but in this case it works out correctly, and saves us a big detour into mathematical formalities.

As you can see in Figure 6, I've given the upper qubit the name $q_0$ and the lower qubit the name $q_1$, just so I don't have to always refer to "the upper qubit" and "the lower qubit." Note that these aren't in ket notation because these labels are the names of the qubits, not their states.

## 4.3 Deutsch's Algorithm

Now we have everything we need to see how Deutsch's algorithm works.

From here on out, it's just a lot of algebraic symbol manipulation and bookkeeping, applying the definitions above. There's nothing to it but just following the rules and writing down the results at each step. A symbolic math program could do the job for us in a millisecond or less.

Before we roll up our sleeves and dig in, I want to give you the big picture for how this will go. First, we'll set up the input to the oracle so that it's a superposition of both inputs, $|0\rangle$ and $|1\rangle$. Then we'll write down the outputs of the oracle, and then juggle them around algebraically until we end up

with a product of two terms. The first term will involve $|0\rangle$ and $|1\rangle$, and the second will be something more complicated, which we won't care about. The heart of the algorithm is that the expression describing the upper qubit $q_0$ will be either a sum or difference of $|0\rangle$ and $|1\rangle$, depending on whether $f$ is constant or balanced. We'll then feed these superpositions into an $H$ qugate to give us just $|0\rangle$ or $|1\rangle$ alone, which we will measure.

On first exposure, there probably won't be anything obvious or apparent about this process. It probably won't result in an "Aha!" moment when it all comes clear to you. The process is just algebra, but it can be surprising to see superposition and interference come together to cause terms to fall out just right so that we can obtain our answer.

This subtle orchestration of complicated maneuvers is at the heart of why designing quantum algorithms is challenging. We'll come back to this point later. For now, I suggest you follow the steps, which are nothing beyond what we've already seen, to see that the algorithm does indeed give us the answer we want.

With all of that said, let's dig in and do some algebra!

Starting at A, both qubits are $|0\rangle$, so we have the combined system state $|0\rangle|0\rangle$. Then we apply $X$ to $q_1$, flipping it from $|0\rangle$ to $|1\rangle$, giving us the new system state $|0\rangle|1\rangle$. These two system states are summarized in Equation 8. I'll write these system states as "equal" to A and B, since those letters are just names, and not quantum objects themselves.

$$\begin{aligned} A &= |0\rangle|0\rangle \\ B &= |0\rangle|1\rangle \end{aligned} \tag{8}$$

Now we'll apply an $H$ qugate to each qubit, giving us Equation 9.

$$\begin{aligned} C &= H|0\rangle\, H|1\rangle \\ &= \vee(|0\rangle + |1\rangle) \vee (|0\rangle - |1\rangle) \\ &= \vee^2(|0\rangle|0\rangle - |0\rangle|1\rangle + |1\rangle|0\rangle - |1\rangle|1\rangle) \end{aligned} \tag{9}$$

In the last line I used our agreement to treat side by side kets as two objects combined with normal multiplication.

Let's do a quick spot check that we're still obeying the rule that the probabilities of all the states in a superposition must add up to 1. Equation 9 is a superposition of four different input states, each with an amplitude of either $\vee^2$ or $-\vee^2$. When we square these amplitudes each one gives us a

probability of 1/4, and there are four of them, so they sum to 1 and we're obeying the rules of probabilities.

Now we'll apply the oracle. As we've seen, the output is a new system of two qubits. The first output is the same as the first input, because the oracle passes that through without change. The second output is the second input XOR'd with the output of the oracle on the first qubit, interpreted as a bit. For example, if the oracle was given the input $|0\rangle|1\rangle$, it would produce $|0\rangle|1 \oplus f(0)\rangle$.

Since the oracle is given to us as a quantum gate, it processes each state in its input superposition independently and simultaneously, like every other qugate. That means it provides us with an output superposition of the four independent results. So lets apply $f$ to each of the four states in the superposition $C$ from Equation 9, giving us the system at step D. The steps are written out in Equation 10.

I warned you that it would be a lot of bookkeeping! But all I'm going to do here is usual multiplication and gathering of terms.

$$\begin{aligned} D &= fC \\ &= f \vee^2 (|0\rangle|0\rangle - |0\rangle|1\rangle + |1\rangle|0\rangle - |1\rangle|1\rangle) \\ &= \vee^2(f(|0\rangle|0\rangle) - f(|0\rangle|1\rangle) + f(|1\rangle|0\rangle) - f(|1\rangle|1\rangle)) \\ &= \vee^2(|0\rangle|0 \oplus f(0)\rangle - |0\rangle|1 \oplus f(0)\rangle + |1\rangle|0 \oplus f(1)\rangle - |1\rangle|1 \oplus f(1)\rangle) \\ &= \vee^2(|0\rangle|f(0)\rangle - |0\rangle|1 \oplus f(0)\rangle + |1\rangle|f(1)\rangle - |1\rangle|1 \oplus f(1)\rangle) \end{aligned} \tag{10}$$

In the second line I replaced C with the last line of Equation 9. On the third line I distributed $f$ so that it applied independently to each term. In the fourth line I used our definition of $f$ to write the result of using it, and the last line was a simplification where I replaced $|0 \oplus f(0)\rangle$ with $|f(0)\rangle$, and similarly replaced $|0 \oplus f(1)\rangle$ with $|f(1)\rangle$, since the XOR of 0 and any bit $a$ is $a$.

So the last line of Equation 10 can be written as Equation 11. Starting now, I'll reduce some clutter by writing $f_0$ for $f(0)$ and $f_1$ for $f(1)$. Also, note that for any bit $a$, $1 \oplus a$ gives us the complement of $a$, which I'll write as $\bar{a}$. That is, 0 becomes 1 and 1 becomes 0. So I'll write $1 \oplus f_0$ as $\bar{f}_0$, and $1 \oplus f_1$ as $\bar{f}_1$.

$$D = \vee^2(|0\rangle|f_0\rangle - |0\rangle|\bar{f}_0\rangle + |1\rangle|f_1\rangle - |1\rangle|\bar{f}_1\rangle) \tag{11}$$

Now we'll use a clever trick and follow the rest of the circuit assuming that $f$ is constant [Gharibian, 2021]. Then we'll see what happens when $f$ is balanced.

If $f$ is constant, then $f_0 = f_1$. So let's write $f_0$ everywhere, and then gather up the terms and simplify. This will give us a simpler version of $D$, which I'll call $D_c$. The steps are in Equation 12.

$$\begin{aligned} D_c &= \vee^2(|0\rangle|f_0\rangle - |0\rangle|\bar{f}_0\rangle + |1\rangle|f_0\rangle - |1\rangle|\bar{f}_0\rangle) \\ &= \vee^2(\left[(|0\rangle + |1\rangle)\right)|f_0\rangle - (|0\rangle + |1\rangle)\,|\bar{f}_0\rangle) \\ &= \vee^2((|0\rangle + |1\rangle)\ (|f_0\rangle - |\bar{f}_0\rangle)) \end{aligned} \tag{12}$$

On the first line I repeated our expression for $D$ from Equation 10. On the second line I gathered up the terms on $|f_0\rangle$ and $\bar{f}_0$, and then on the last line I noted that $(|0\rangle + |1\rangle)$ was common to both.

Wait a second. That first term, $(|0\rangle + |1\rangle)$, looks familiar. Except for a factor of $\vee$, it's $H|0\rangle$ from Equation 2! And we can get that $\vee$ from the $\vee^2$ at the start of the equation. So if we apply $H$ to the first qubit, it will turn into $|0\rangle$. If you look at Figure 6, that's just what we do to get to label E. Let's call that $E_c$ (for the constant case), and walk through the math in Equation 13. In the first line I moved one of the $\vee$ terms so that it becomes part of the state that's given to $H$.

$$\begin{aligned} E_c &= \vee(H(\vee(|0\rangle + |1\rangle))\ (|f_0\rangle - |\bar{f}_0\rangle)) \\ &= \vee(H|+\rangle\ (|f_0\rangle - |\bar{f}_0\rangle)) \\ &= \vee(|0\rangle\ (|f_0\rangle - |\bar{f}_0\rangle)) \end{aligned} \tag{13}$$

Now we're ready to measure the first qubit. We know what we're going to get, because that first qubit is $|0\rangle$. It's a superposition of only one state, $|0\rangle$ and nothing else. So we are assured that we'll measure the corresponding bit 0 **with certainty**.

The only assumption we made here is that $f$ is constant, and we found that if that's the case, when we measure $q_0$ we'll get 0 every time, without fail.

That's great!

But we're not done, because we haven't looked at what happens when $f$ is balanced. Maybe that always puts the first qubit into the state $|0\rangle$, or maybe it just does that sometimes. So let's turn our attention to the balanced case. We'll pick up the thread with our expression for D in Equation 11, just before we made any assumptions about $f$.

The key insight is that since $f$ is a binary function, with the only values being 0 and 1, then $\bar{f}_0 = f_1$ and $\bar{f}_1 = f_0$.

So we can replace those XOR expressions in Equation 11 with their simpler forms, giving us the balanced version of D, which I'll call $D_b$. The steps are in Equation 14.

$$\begin{aligned} D_b &= \vee^2(|0\rangle|f_0\rangle - |0\rangle|\bar{f}_0\rangle + |1\rangle|f_1\rangle - |1\rangle|\bar{f}_1\rangle) \\ &= \vee^2(|0\rangle|f_0\rangle - |0\rangle|f_1\rangle + |1\rangle|f_1\rangle - |1\rangle|f_0\rangle) \\ &= \vee^2((|0\rangle - |1\rangle)|f_0\rangle \; - \; (|0\rangle - |1\rangle)|f_1\rangle)) \\ &= \vee^2((|0\rangle - |1\rangle)\,(|f_0\rangle - |f_1\rangle)) \end{aligned} \tag{14}$$

On the first line I wrote $D$ from Equation 11. On the second line I replaced the XOR terms with their equivalent shorter versions. On the third line I gathered up the terms on $|f_0\rangle$ and $|f_1\rangle$, and on the last line I noted that both of these were being multiplied by $|0\rangle - |1\rangle$.

You're probably way ahead of me by now. That first qubit is $|-\rangle$! So as we discovered, if we apply $H$ to $|-\rangle$, we get back $H|-\rangle = |1\rangle$, Equation 15.

$$\begin{aligned} E_b &= \vee(H(\vee(|0\rangle - |1\rangle))\,(|f_0\rangle - |f_1\rangle)) \\ &= \vee(H|-\rangle\,(|f_0\rangle - |f_1\rangle) \\ &= \vee(|1\rangle\,(|f_0\rangle - |f_1\rangle)) \end{aligned} \tag{15}$$

When we now measure the first qubit, we will get back 1 with certainty. Our only assumption was that $f$ was balanced.

This wraps up our analysis of Deutsch's algorithm! It's all there. There's nothing missing, nothing left over.

When $f$ always returns 0 or 1 for both inputs, it's constant, and when we measure qubit $q_0$ we will always get 0. Otherwise, when $f$ always returns 0 for one input and 1 for the other, it's balanced, and when we measure qubit $q_0$ we will always get 1.

Thanks to superposition and interference, followed by measurement, we only needed to consult the oracle a single time to get our answer! That's something that *no classical computer can ever do.*

As we saw in the circuit diagram, we only care about qubit $q_0$ when we're done. The expressions for $E_c$ and $E_b$ tell us that $q_1$ is $\vee(|f_0\rangle - |\bar{f}_0\rangle)$ and $\vee(|f_0\rangle - |f_1\rangle)$ respectively, but that information is unimportant for us, since all we want to do is know if $f$ is constant or balanced, and $q_0$ tells us that.

# 5 Discussion

As I mentioned above, you may be feeling, let's say, unsatisfied. Maybe you followed every step of the math and you agreed with the result, but perhaps you have a nagging feeling that you're not sure *why* it all works.

Don't worry.

Getting more familiar with this through repeated exposure, and through learning other algorithms, will make this gradually feel more and more sensible and comfortable. This will also help you move past the details of the algebra and better see the sequence of changes that they are bringing about.

I suggest one way to get started is to think about the input to the oracle $f$. The $|x\rangle$ input is a superposition of $|0\rangle$ and $|1\rangle$, so we know the oracle, being a qugate, will process each of these possibilities independently, and output a superposition of the outputs. At the same time, since $y$ came into the oracle in the superposition $|-\rangle$, the output value on line $q_1$ is *also* in superposition, this time of $|0\rangle$ and $-|1\rangle$. The difference with $|x\rangle$ is that the $|1\rangle$ has an amplitude of $-1$. This difference is essential, and a key to using interference to get our result.

The other key is that we now have *four* parallel executions performed by $f$, each a combination of one of the $|x\rangle$ states and one of the $|y\rangle$ states.

We can see this in our expression for D in Equation 11, which has has four different terms that depend on $f$. We want to find a way to knock out two of those four terms. Then we'll be able to factor what's left into one term involving $|0\rangle$ and $|1\rangle$ (on qubit $q_0$), and another term with all the $f$ stuff (on qubit $q_1$).

Asserting that $f_1 = f_0$ knocks out the terms involving $f_1$, leaving us with just two terms involving $f_0$ and $\bar{f}_0$. On the other hand, asserting that $\bar{f}_0 = f_1$ and $\bar{f}_1 = f_0$ knocks out the terms involving XORs, leaving us with just terms involving $f_0$ and $f_1$.

I don't know how Deutsch came up with this algorithm, but we can imagine someone setting up the circuit, looking at the output, and after some messing about with the algebra, realizing that these two choices led to two different measurements. The choices could be described by saying that $f$ was either constant or balanced. So Deutsch might then have worked backwards, and said that distinguishing between constant and balanced forms of $f$ was actually the whole purpose of the algorithm!

To keep the discussion moving forward I slid past something quite important, so let's go back and address it.

I said earlier that the upper output of the oracle is just the upper input, passed through without change. That's exactly true in the classical version in Figure 3, where the inputs were bits.

But we've seen that in the quantum version of Figure 5, the upper output of the oracle *does* change. As we saw for label C, the input is $|+\rangle$, but the output can be either $|+\rangle$ or $|-\rangle$. That second version isn't the same as the input! I must have cheated somewhere, right?

Yes, I did, but only because I was casual with what it meant for the upper input to be output "without changing." What I meant was "without a change that we can directly measure." I know, that's pretty subtle, so let's expand it a little.

If we measure $|+\rangle = \vee(|0\rangle + |1\rangle)$, then both $|0\rangle$ and $|1\rangle$ have amplitudes $\vee$, so each has a probability of $\vee^2 = 1/2$. We have an equal chance of measuring either 0 or 1. If we instead measure the subtracted version, $|-\rangle = \vee(|0\rangle - |1\rangle)$, *the squared amplitudes are the same*! That's because the squared amplitude of $|0\rangle$ is $\vee^2 = 1/2$, and the squared amplitude of $|1\rangle$ is $(-\vee)^2 = 1/2$. Just as in the first case, we have an equal chance of measuring either 0 or 1. *There is no way to distinguish these two states with a direct measurement.* If we dug deeper into the math, you'd see that there are no loopholes here.

On the other hand, we *can* perform further operations on the states (like apply an $H$, as we do here) and get different results that ultimately lead to different measurements, as long as that's all happening inside the computer. We can't see it, and we can't measure it. The only way we can learn about what's happening inside our machine is to make a measurement, and *no measurement can distinguish between the superposition state* $\vee(|0\rangle + |1\rangle)$ *and the superposition state* $\vee(|0\rangle - |1\rangle)$. So that's why we can treat the output superposition with $-|1\rangle$ as "the same" as the input superposition with $|1\rangle$.

This is a subtle point and you might be thinking of ways to circumvent the issue, but in over 100 years of quantum mechanics nobody has found a way, either in theory or practice, to distinguish these states with a measurement. We can't tell them apart. But – and this is the phenomenon that we depend on – when the states are inside the quantum computer they *are* different, and further processing (like applying an $H$ qugate) can make use of that difference, ultimately resulting in different measurements.

If you're scratching your head a little over this distinction, welcome to the world of quantum objects. Quantum-sized things just don't behave the same way as things at our human scale, and there are lots of surprises. Many of these differences can be challenging get used to, like this one.

If you'd like to understand this better, you can consult the references at the end of these notes to learn the mathematical theory which I've deliberately skipped here. Then you'll be able to see for yourself where these unexpected phenomena come from. As I mentioned earlier, a really good piece of news is that you'll rarely need anything more complicated than the linear algebra you're already familiar with from computer graphics!

Quantum algorithms can feel like magic tricks. This is one reason why designing new algorithms is both hard and incredibly fulfilling. When you manage to find a way to juggle all the pieces together just right, it's a deeply satisfying feeling, like you've solved a fiendish logic puzzle or pulled off an acrobatic stunt. Either way, it's an amazing feat.

# 6 Some Other Algorithms

Deutsch's algorithm is perhaps the simplest of the famous quantum algorithms that people study in quantum computing courses. A more generalized version of the algorithm that works for functions of multiple bits is called the **Deutsch-Josza algorithm** [Wikipedia, 2023b]. Other important algorithms are also named for their developers.

The **Bernstein-Vazirani algorithm** is another "black-box" method. [Wikipedia, 2023a]. Here the oracle contains a secret bitstring $s$, and for every input bitstring $x$ the oracle returns the bitwise XOR of $s$ and $x$. The goal is to work out $s$. That is, we're trying to guess a secret number of an arbitrary number of bits. Amazingly, this algorithm reveals the value of $s$ with a single query of the oracle.

**Simon's Algorithm** supposes that we have an oracle that has one of two possible forms [Simon, 1997]. Its input and output are both bitstrings of any given length. The oracle either returns a unique bitstring for each input (that is, the function is one-to-one), or it sends pairs of inputs to the same output (so it's two-to-one). Simon's algorithm can tell us which of these two types of functions we've got by using exponentially fewer queries to the oracle than required by a classical algorithm.

These algorithms fall into the category of *demonstration algorithms*, or perhaps *contrived algorithms*. They don't solve a problem that we expect is going to really come up in practice. Instead, they moved the field of quantum computing forward by inventing techniques that can be used for other, more practical problems.

**Grover's Algorithm** is one of these practical tools that people actually use to solve problems [Grover, 1996] [Qiskit, 2022a]. It's a searching technique. Suppose you have an unstructured list of $N$ items, and there's one in particular you want to locate. Imagine that you have an oracle you can query that returns 0 for every input except for the value you want, where it returns 1. Compared to classical algorithms, Grover's algorithm is able to find the one entry you want with only about $\sqrt{N}$ queries of the oracle.

There are utility algorithm of great value. The **Quantum Phase Estimation** algorithm, or **QPE**, lets us learn something about quantum states that is normally difficult to measure [Qiskit, 2022b]. The **Quantum Fourier Transform** algorithm, or **QFT**, brings the outrageously useful and ubiquitous Fourier transform to the quantum realm [Wikipedia, 2023d].

Perhaps the most famous and widely-discussed quantum algorithm is **Shor's Algorithm** [Shor, 1994] [Shor, 1996] [IBM, 2022]. This algorithm provides a way to factor integers into products of prime numbers, using the QFT and a wealth of insight from number theory. That might sound like another obscure mathematical operation, but prime factoring lies at the heart of much modern encryption software, including the algorithms that are used to protect internet traffic, banking, and all kinds of personal and organizational data. The security of these systems is based on the observation that nobody knows of a way to factor enormous integers on classical computers in a practical manner. So if a security system is based on the idea that it cannot be cracked in less than, say, 1000 years of computing time, that seems pretty secure. Shor's algorithm changes this dramatically because it uses quantum computing to factor numbers in a vastly more efficient, and thus faster, way. Cracking that encryption software can allow anyone to read any of those messages, and even send supposedly authentic messages between systems. So you could add a few million dollars to your personal bank account with a single message, or discover the security measures used by your local museum to safeguard precious art, or uncover the identity of your favorite internet streamer, among thousands of other acts of secrecy violation.

Don't expect to get rich quick, though, as running Shor's algorithm for large integers is still beyond the capability of the quantum computers we're able to build today. That's changing fast, though, so people are working very hard to find alternative encryption methods that cannot be efficiently cracked, even with quantum computers.

The resources section offer you places to read about all of these algorithms and many others, and even find explicit quantum source code to run them.

# Section 3
# Applications to Computer Graphics

Some people have already started exploring applications of quantum computing to computer graphics.

As you might guess, ray tracing is a popular target. The algorithm itself is so inherently parallel that it just feels like a great fit to quantum computing. There's been a steady flow of papers on ray tracing and quantum computing starting in 1991 [Pevzner and Hess, 1991] and continuing [Glassner, 2001a] [Glassner, 2001b] [Glassner, 2001c] [Johnston, 2015] [Alves et al., 2019] [Lu and Lin, 2022] [Santos et al., 2022] [Zhang et al., 2022].

There are some challenges to getting this working.

One class of issues has to do with communications. Usually the overall approach is to build a **hybrid** algorithm of a classical computer and a quantum computer, where the quantum computer is considered a kind of special purpose co-processor dedicated to intersecting rays with objects. So the classical computer sends rays to the quantum circuit, and gets back something to do with intersections of those rays. Those communications between classical and quantum computers can be slow, and can eat into the time savings we get from quantum computation.

Another class of issues is how to give the quantum computer enough information about the scene for it to compute intersections of rays with objects. Sometimes this is abstracted away, often into a Grover-style oracle which tells us if an intersection has occurred or not. The problem is that any such oracle must be actually implemented in a real circuit, and if it contains a list of all the objects in the scene, that becomes a *very big* quantum circuit, perhaps much bigger than today's computers are able to efficiently and accurately implement. We could have the classical computer download only those objects that it knows are possible candidates for intersections with a specific batch of one or more rays, but again those communications, and encoding the information into the quantum computer, whether as a Grover-style oracle or in some other form, will take time and resources inside the quantum computer.

Further complicating matters is that we don't really have general floating-point math available on quantum computers today, though people are working on it [Häner et al., 2018] [Seidel et al., 2021].

So while the concept of using quantum parallelism to accelerate ray tracing is very appealing, it still seems to face some practical challenges.

Another place quantum parallelism can help is in simulation. One way to work out the parameters for a simulation is to define starting and goal states for some system (say, a robot arm or a cloud of smoke), and then we try a large number of parameter sets looking for something that gets us from the start to the goal, or perhaps just close to it. Again, quantum parallelism has promise to help us identify the best such set of parameters.

Suppose we can encode our system in a quantum algorithm, and we can also encode our parameter sets in terms of the bitstrings or superpositions that go into the algorithm, and our outputs in terms of the bitstrings we measure when the algorithm has run. Then we can use the quantum computer to evaluate inconceivably vast numbers of parameter sets to find the one that does the best job.

In this task, communications is probably less of an issue than in ray tracing. But challenges here include the need to embed the simulation in some way into the quantum circuit, and managing the numerical computation without robust and practical access to floating-point calculations.

In general, quantum computation is appealing, at least in theory, for any task where we want to evaluate vast numbers of inputs to identify the one that works best for us.

For example, suppose we want to find the best prompt, or sequence of words, to provide to an image generator to produce a specific image. We might imagine putting the entire image generator into a quantum computer. Then we can create a superposition containing every possible sequence of words from some starting pool, and run the image generator on every one of those prompts *in parallel*. Then we compare those outputs with the image we started with, and return the sequence that produced the image that matched it the best.

This kind of thinking could let us turn lots of algorithms that rely on direct computation on their heads, and replace computation of output numbers with extraction of the best set of numbers from a huge starting pool.

For example, consider denoising. Currently we take in information about samples in a pixel, and from them we are able to compute a denoised output, or perhaps instead a signal that we need more samples to reconstruct that pixel to some desired degree of accuracy. Ultimately, pixels end up as integers. If we use 16-bit integers per color channel, that's a total of $3 \times 16 = 48$ bits.

So suppose we replace our computation with an evaluation function that assigns a score to any given RGB triplet, telling us how well that triplet performs when used as the final denoised value for the pixel. Then we could load up the circuit with all the data we have for a particular pixel, and have it *simultaneously evaluate every 48-bit integer* to find the one (or ones) that best denoises that pixel. Again, this could require more horsepower and accuracy than quantum computers can provide us today, but quantum computing hardware is improving in almost all respects at a rapid pace.

# Section 4
# Resources

### Quantum Mechanics

Quantum computing is built on the theory of quantum mechanics. There are many fine introductory books for quantum mechanics. Two that I particularly like are [Susskind and Friedman, 2014] and [Griffiths and Schroeter, 2018], but there are so many that it's worth the time to look at a few to find the best fit for you.

### Quantum Computing

The king of textbooks for quantum computing is [Neilsen and Chuang, 2011]. It has all the benefits and drawbacks of being a textbook. A more approachable book is [Mermin, 2007]. Many sets of lecture notes from courses in quantum computing are available online. Three that I really like are [Gharibian, 2021], [Simha, 2022], and [Young, 2021]. The documentation on the Qiskit site is often quite good [Qiskit, 2023].

There are lots of quantum gates out there that you'll find useful. Big reference lists of common qugates can be found at [IBM, 2023b] and [Wikipedia, 2023c].

### Simulators and Real Quantum Computers

You can write and run quantum algorithms today on a wide variety of simulators. A big list of simulators for many languages and platforms can be found at [Quantiki, 2023]. Well-known environments for building and simulating quantum circuits include Qiskit [Qiskit, 2023], Amazon Braket [Braket, 2023], and the IBM Quantum Experience [IBM, 2023a]. The IBM offering has a graphical front end so you can draw your circuits. Qiskit has a simulator called `Statevector` that lets you actually pull out the probabilties for all the possible states at any point in your circuit.

If you wan to use actual hardware, several providers offer limited access to small quantum computers *for free*. For example, as of this writing (early 2025), both IBM and Amazon offer a "free tier" on their quantum computers. The plans vary from one provider to the next, as well as over time, but typically they offer a few hours of free use every month on relatively small quantum computers, such as 5- and 7-qubit models. Typically you'll create

your circuit, submit it to the system, and after a while you'll get to the head of the queue. The system will then run your program, and send you back the results. Even though these computers are small, you can do a lot with just a few qubits. And there's nothing like running your code on a real, live quantum computer to be sure that you know what you're doing.

### Drawing Quantum Circuits

A great way to share your quantum circuits is to draw them. There are several packages that are designed to make good-looking circuits in LaTeX, including *quantikz* [Kay, 2020], *qcircuit* [Scholten et al., 2023], *yquant* [Desef, 2023]. All of these are built on LaTeX, so they're as brittle and unforgiving as LaTeX itself, with equally horrendous error messaging. For those reasons, I usually use *quantikz*, which I've found has the simplest syntax and almost always can be coerced into behaving, though to draw a complicated circuit I have to slowly build it up one tiny step at a time, recompiling the document after each adjustment. Nevertheless, when you have things dialed in, the diagrams are nice. All the quantum diagrams in these notes were drawn with *quantikz*.

### Philosophy

The philosophy of quantum mechanics, and by extension quantum computing, is fascinating. Arguments over what it all means have been raging for over a century, with no sign of stopping [Einstein et al., 1935] [Szabó, 2023] [Weisberger, 2019].

# 7 Acknowledgements

Thank you to Stephen Drucker, Adam Finkelstein, and Eric Haines for illuminating conversations, and my employer, Wētā FX, for support and providing such a wonderful and creative work atmosphere. Thanks also to many of the fine independent coffeeshops in Seattle, where I did much of my brainstorming, outlining, and figure drafting of these notes: Cafe Diva, Cafe Ladro, Cafe Vita, Distant Worlds Coffeehouse, Firehouse Coffee, The Fremont Coffee Company, and Red Arrow Coffee.

# References

[Alves et al., 2019] Alves, C. A., Santos, L. P., and Bashford-Rogers, T. (2019). A quantum algorithm for ray casting using an orthographic camera. In *2019 International Conference on Graphics and Interaction*.

[Baker, 2021] Baker, H. (2021). How many atoms are in the observable universe? https://www.space.com/how-many-atoms-in-universe.

[Braket, 2023] Braket (2023). Amazon braket. https://aws.amazon.com/braket/.

[Desef, 2023] Desef, B. (2023). yquant. https://github.com/projekter/yquant.

[Deutsch, 1985] Deutsch, D. (1985). Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society A.*, 85(400):97–117. https://www.daviddeutsch.org.uk/wp-content/deutsch85.pdf.

[Dirac, 1939] Dirac, P. (1939). A new notation for quantum mechanics. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 35, pages 416–418.

[Einstein et al., 1935] Einstein, A., Podolsky, B., and Rosen, N. (1935). Can quantum mechanical description of physical reality be considered complete? *Physical Review*, 47:777–780. https://journals.aps.org/pr/abstract/10.1103/PhysRev.47.777.

[Everett III, 1957] Everett III, H. (1957). Relative state formulation of quantum mechanics. *Reviews of Modern Physics*, 29(3). https://typeset.io/papers/relative-state-formulation-of-quantum-mechanics-3crdqguh8o.

[Gharibian, 2021] Gharibian, S. (2021). Introduction to quantum computation. *Paderborn University*.

[Glassner, 2001a] Glassner, A. (2001a). Quantum computing, Part 1. https://www.glassner.com/wp-content/uploads/2014/04/CG-CGA-PDF-01-07-Quantum-Computing-1-July01.pdf.

[Glassner, 2001b] Glassner, A. (2001b). Quantum computing, Part 2. https://www.glassner.com/wp-content/uploads/2014/04/CG-CGA-PDF-01-09-Quantum-Computing-2-Sept01.pdf.

[Glassner, 2001c] Glassner, A. (2001c). Quantum computing, Part 3. https://www.glassner.com/wp-content/uploads/2014/04/CG-CGA-PDF-01-11-Quantum-Computing-3-Nov01.pdf.

[Griffiths and Schroeter, 2018] Griffiths, D. J. and Schroeter, D. F. (2018). *Introduction to Quantum Mechanics*. Cambridge University Press, 3rd edition.

[Grover, 1996] Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. https://arxiv.org/abs/quant-ph/9605043.

[Häner et al., 2018] Häner, T., Soeken, M., Roetteler, M., and Svore, K. M. (2018). *Quantum circuits for floating-point arithmetic*, pages 162–174. Reversible Computation. Springer. https://arxiv.org/abs/1807.02023.

[IBM, 2022] IBM (2022). Shor's algorithm. https://quantum-computing.ibm.com/composer/docs/iqx/guide/shors-algorithm.

[IBM, 2023a] IBM (2023a). IBM Quantum Experience. https://quantum-computing.ibm.com/.

[IBM, 2023b] IBM (2023b). Single qubit gates. https://qiskit.org/textbook/ch-states/single-qubit-gates.html.

[Johnston, 2015] Johnston, E. (2015). An exploratory study in quantum acceleration of ray tracing. https://www.machinelevel.com/qc/doc/Quantum%20Ray%20Tracing.pdf.

[Kay, 2020] Kay, A. (2020). Tutorial on the Quantikz Package. https://arxiv.org/abs/1809.03842.

[Lesovik et al., 2017] Lesovik, G. B., Sadovskyy, I. A., Suslov, M. V., Lebedev, A. V., and Vinokur, V. M. (2017). Arrow of time and its reversal on the IBM quantum computer. *Scientific Reports*, 9.

[Lu and Lin, 2022] Lu, X. and Lin, H. (2022). A framework for quantum ray tracing. https://arxiv.org/abs/2203.15451.

[Mermin, 2007] Mermin, N. D. (2007). *Quantum Computer Science: An Introduction*. Cambridge University Press, Cambridge, UK.

[Neilsen and Chuang, 2011] Neilsen, M. A. and Chuang, I. L. (2011). *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, first edition.

[Pevzner and Hess, 1991] Pevzner, V. and Hess, H. (1991). Quantum ray tracing: A new approach to quantum transport in mesoscopic systems. https://link.springer.com/chapter/10.1007/978-1-4757-2124-9_45.

[Qiskit, 2022a] Qiskit (2022a). Grover's algorithm. https://qiskit.org/textbook/ch-algorithms/grover.html.

[Qiskit, 2022b] Qiskit (2022b). Quantum phase estimation. https://qiskit.org/textbook/ch-algorithms/quantum-phase-estimation.html.

[Qiskit, 2023] Qiskit (2023). Qiskit documentation. https://qiskit.org/.

[Quantiki, 2023] Quantiki (2023). List of QC simulators. https://quantiki.org/wiki/list-qc-simulators.

[Santos et al., 2022] Santos, L. P., Bashford-Rogers, T., Barbosa, J., and Navrátil, P. (2022). Towards quantum ray tracing. https://arxiv.org/abs/2204.12797.

[Scholten et al., 2023] Scholten, T. L., Eastin, B., and Flammia, S. (2023). qcircuit – macros to generate quantum circuits. https://www.ctan.org/pkg/qcircuit.

[Seidel et al., 2021] Seidel, R., Tcholtchev, N., Sebastian, B., Becker, C. K.-U., and Hauswirth, M. (2021). Efficient floating point arithmetic for quantum computers. https://arxiv.org/abs/2112.10537.

[Shor, 1994] Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press. https://math.mit.edu/~shor/papers/algsfqc-dlf.pdf.

[Shor, 1996] Shor, P. W. (1996). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. https://arxiv.org/pdf/quant-ph/9508027.pdf.

[Simha, 2022] Simha, R. (2022). Introduction to quantum computing. https://www2.seas.gwu.edu/~simhaweb/quantum/modules/.

[Simon, 1997] Simon, D. R. (1997). On the power of quantum computation. *SIAM Journal on Computing*, 26(5):1474–1483.

[Susskind and Friedman, 2014] Susskind, L. and Friedman, A. (2014). *Quantum Mechanics: The Theoretical Minimimum*. Basic Books.

[Szabó, 2023] Szabó, L. E. (2023). The Einstein-Podolsky-Rosen argument and the Bell inequalities. https://iep.utm.edu/einstein-podolsky-rosen-argument-bell-inequalities/.

[Weisberger, 2019] Weisberger, M. (2019). 'god plays dice with the universe,' Einstein writes in letter about his qualms with quantum theory. https://quantumcomputing.stackexchange.com/questions/2263/how-do-i-show-that-a-two-qubit-state-is-an-entangled-state.

[Wikipedia, 2022] Wikipedia (2022). Jaques Hadamard. https://en.wikipedia.org/wiki/Jacques_Hadamard.

[Wikipedia, 2023a] Wikipedia (2023a). Bernstein–Vazirani algorithm. https://en.wikipedia.org/wiki/Bernstein%E2%80%93Vazirani_algorithm.

[Wikipedia, 2023b] Wikipedia (2023b). The Deutsch-Josza algorithm. https://en.wikipedia.org/wiki/Deutsch-Jozsa_algorithm.

[Wikipedia, 2023c] Wikipedia (2023c). List of quantum logic gates. https://en.wikipedia.org/wiki/List_of_quantum_logic_gates.

[Wikipedia, 2023d] Wikipedia (2023d). Quantum Fourier transform. https://en.wikipedia.org/wiki/Quantum_Fourier_transform.

[Young, 2021] Young, P. (2021). An undergraduate course on quantum computing (2nd edition). *UC Santa Cruz*.

[Zhang et al., 2022] Zhang, Y., Orth, A., England, D., and Sussman, B. (2022). Ray tracing with quantum correlated photons to image a three-dimensional scene. *Physical Review A*, 105.