An Introduction to the Fourier Transform

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

Andrew Glassner
Distinguished Research Engineer
Wētā FX

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

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

Synopsis

The Fourier Transform is a fundamental tool in computer graphics. It explains where aliasing comes from, how to filter textures, why noise can be a good thing, how to simplify shapes, how to select sampling patterns, how the JPEG image compression scheme works, and why wagon wheels start to rotate backwards when they spin too fast. The Fourier Transform not only describes the source of many problems in graphics, it also often tells us how to avoid or suppress them.

Understanding the Fourier transform gives us the ability to identify, diagnose, and fix objectionable artifacts that might otherwise remain mysterious in rendering, modeling, and animation code.

Unfortunately, the Fourier Transform is unfamiliar to many people, and opaque to many others, often because they are put off by its technical language and complicated looking mathematics. Though the notation can be daunting at first contact, it's really just a terse way of expressing specific sequences of multiplications and additions.

In this course we only assume you know a little vector algebra, like the material in any graphics book (if you remember how to multiply two matrices, you're all set). We'll carefully build up to the full Discrete-Time Fourier Transform (and its inverse) that we use every day, taking small steps and illustrating the process with pictures.

Speaker's Bio

Andrew Glassner is a Distinguished Research Engineer 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*. His latest book is *Quantum Computing: From Concepts to Code*, published by No Starch Press. 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.

1 Introduction

In these course notes we'll meet one of the most widely-used mathematical tools in all of engineering and physics: the **Fourier transform** (pronounciations vary, but common ways to say this are fore′-ee-ay or fore′-yay).

The Fourier transform is a cornerstone of computer graphics, as well as many other everyday technologies that are all around us. It's essential to television, AM and FM radio, MRI machines, weather satellites, digital photography, speech recognition, radar and sonar, music synthesizers, and even medical diagnosis, among countless other technologies.

The Fourier transform is the key mechanism that helps us understand and prevent aliasing, the root cause of many artifacts in rendered images, animations, and video.

An everyday use of the Fourier transform arises when you're listening to some music, but you want to increase the bass. Today, we say we want to "boost the low frequencies," which has become such a natural concept that we rarely question it or wonder what it really means. This idea that the music is made up of "frequencies" that can be adjusted is exactly what the Fourier transform is all about.

The goal of these notes is to introduce the Fourier transform to you so that you'll be comfortable with the ideas, and even able to work them for yourself. Once we've seen the traditional form of this ubiquitous tool, we'll catch up with the latest technology and see how it's use in quantum computing.

These notes will take an informal approach to the math, focusing on concepts rather than rigor. If you'd like to nail down any of the details, consult the references that are referred to throughout the notes.

2 Fourier's Big Idea

In 1822, Joseph Fourier published a book on how heat flows in an environment over time [Wikipedia, 2021].

Fourier wanted to describe the flow of heat at different levels of detail. Suppose that you have a metal case surrounding a motor, and you want to know how the case warms up as the motor turns. If you just want a general idea, you could consider big regions of hot and cold and look at how those shapes changed and moved on the surface of the case. If you want more detail, you could zoom in and break up the regions into smaller, more defined shapes. Then you could keep zooming in, looking at finer and finer details of where the heat is located and how

it's moving.

Fourier wrote equations to enable exactly this kind of investigation. Suppose we want to know the temperature of a particular point on the metal casing at different times. If we're lucky, we can devise a function that tells us that temperature as a function of time. Let's write this function as $f(t)$, which returns a real number for any value of a time variable t. Fourier's idea was to re-write that function as a sum of sine and cosine waves. Figure 1 shows the idea.

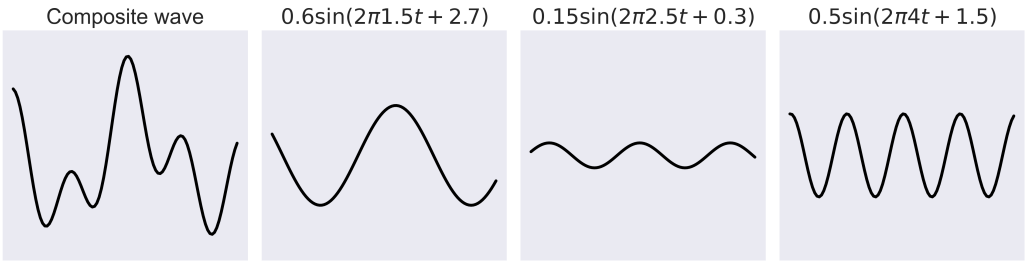

Figure 1: The Fourier transform is a tool that breaks down any input curve of real numbers, such as the one at the left, into multiple sine waves. If we add up the three curves on the right they exactly match the curve on the left.

To look more closely at the details at any level, we can look at the behavior of the waves that change more quickly.

Figure 2 shows this idea applied to an image. The process I used is the same as Fourier's original idea, only applied in two dimensions rather than just one. We can also apply it to three dimensions (which is what an MRI scanner does), or any other number of dimensions [Dudgeon and Mersereau, 1984].

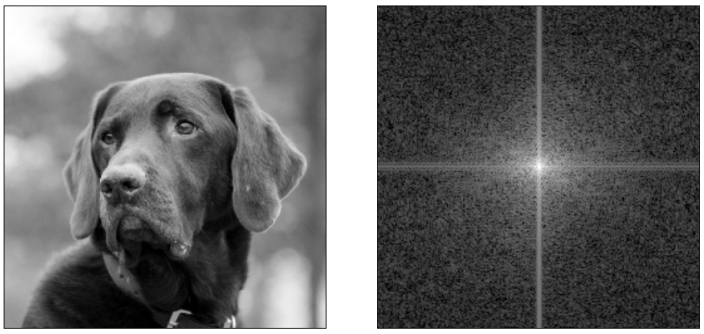

Figure 2: An image, along with the logarithm of the magnitude of its Fourier transform.

Fourier didn't actually use literal sines and cosines. Instead, he wrote these waves as *complex exponentials*, which we'll discuss shortly.

The idea was terrific: we could look at the original function at different scales by summing different combination of these waves, each representing a different level of detail. He set the calcuation up so that the result of this summation at any point was always a real number (as it must be, as we can only measure real numbers in the physical world). We'll see later how the math does this.

It wasn't at all clear at the time that this idea was mathematically valid for arbitrary functions. It also wasn't clear that the complex exponentials were the best (or only) way to achieve this decomposition. In fact, we now know that there are lots of other functions that let us break down a signal into different level of detail. But sines and cosines, represented by complex functions, are still extremely popular. They are remarkably convenient mathematically, and make sense to people on an intuitive level, as we saw earlier when we wanted to "pump up the bass" on our music.

Although Fourier's technique was greatly appealing and of real practical use, performing the computations could be difficult and required mathematical sophistication.

Between 1822 and today, there have been at least three big developments which moved Fourier's original ideas from the realm of a specialist's tool to a staple of modern computation.

First, his original insight had to be made mathematically rigorous. To use the method with confidence, we need to know when this process of decomposition into exponentials works, and when it won't, and details like how accurate

it can be [Carleson, 1966]. In fact, figuring out when the process would actually work properly was considered the "central problem" of the theory of Fourier transforms [Courant and Hilbert, 1937]. These principles were boiled down by Peter Direchlet into a small list of rules, now known as the **Direchlet conditions** [Weisstein, 2023].

The second big event was the invention and development of programmable digital computers. People wanted to use these new devices to automatically carry out Fourier's computations.

This was only possible because one of Direchlet's conclusions was that the Fourier process could be carried out on any input that we would today call **discrete** or **sampled**, or made up of a list of output values.

Figure 3 shows this idea, replacing the continuous curves of Figure 1 with sequences of numbers.

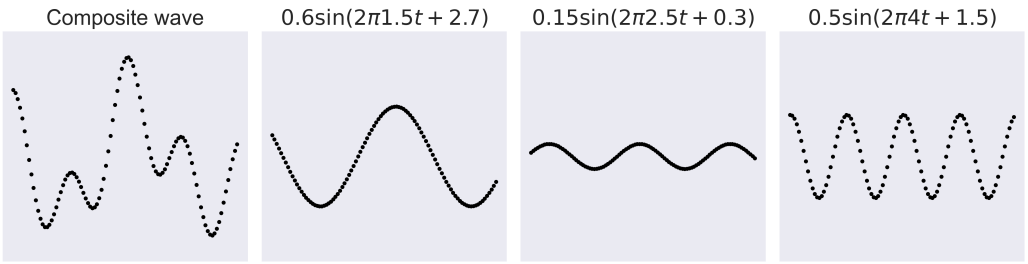

Figure 3: The data of Figure 1 but using discrete, or sampled, data for input and outputs.

Even with confirmation that we can convert a discrete function into a sum of different exponential functions (or more simply, exponentials), the computation was still *slow*. In practice, the costs in time and memory made computing the Fourier transform impractical. Suppose you can process an input of N samples in 1 minute. Each time you add one more sample, the processing time roughly doubles. We say that the process is $O(N^2)$ (spoken out loud as "order N squared"), meaning that the computational cost is roughly N^2 for N inputs. That cost grows *fast*, making the technique impractically expensive for most applications. It seemed that the Fourier transform was doomed to remain a niche tool.

And then, with a third development, everything changed. People usually date that change to the publication of a 5-page paper now known as the Cooley-Tukey paper, named for its authors [Cooley and Tukey, 1965]. It presented a clever way

to arrange the calculations on a digital computer to evaluate the Fourier transform vastly more quickly than a straightforward approach. That technique is now known as the **Fast Fourier Transform**, or **FFT**, and it made the Fourier transform practical. It's thanks to the FFT that the Fourier transform is able to contribute to everyday science and engineering [Valentinuzzi, 2016]. Practically any Fourier transform routine you'll find in a modern math library uses the FFT.

Let's now look at the mechanics of the Fourier transform.

3 Terminology

I'll start out by introducing some terms and symbols that we'll use throughout these notes.

We often discuss the Fourier transform in terms of **functions**. Fourier's original discussion used inputs that were continuous mathematical expressions, or functions, like $t^2 + 5$. Though there are computer programs that can process these kinds of **symbolic**, or **analytic**, expressions, in computer programs we much more frequently represent our inputs as lists of numbers describing the value of the function at multiple, specific values of the input parameter. When used with a continuous function, this parameter can take on any real number. It's conventional to still discuss the process in terms of an input "function," and I'll often do that here, even when we're working with lists of numbers.

Fourier started out by describing the flow of heat over time. Because of that precedent, we often call the input to a Fourier transform a **time signal**, or just a **signal**. We also say that our input data is in the **time domain**, meaning that it implicitly refers to values of the function at different values of a time parameter.

While we do frequently use functions based on time, it's also common to use signals that have nothing to do with time. In computer graphics, our signals might represent the values of the red, green, and blue components of sequences of pixels. They might also describe how quickly lighting or texture changes over the surface of an object.

Despite the now broad applications of the Fourier transform, the "time" language has stuck, so we almost always refer to the input in terms of time. You can think of time as a generic name for the index that lets us sequence the elements in our input list from beginning to end.

When we turn a signal into exponentials, the key property that lets us pick out different levels of detail is the **frequency** of each of those exponentials. We saw an example of this in Figure 1, where we added together multiple waves that

oscillate at different speeds, or frequencies.

We say that the exponentials that describe a time signal represent the **frequency spectrum**, or just the **spectrum**, of that signal. We also say that these exponentials represent our time signal in the **frequency domain**, or the **spectral domain**,

The heart of Fourier's operations are to turn a time signal into its frequency spectrum, and vice-versa. Because both representations describe the same underlying function, we say that turning either into the other is a **transformation** or **transform**.

It's important to note that aside from the usual errors introduced by performing floating-point computation on conventional computers, this process is reversible. There is no loss or distortion when going from the time domain to the frequency domain, or vice-versa (as long as we meet the necessary criteria, as discussed in these notes). The time signal and its spectrum are truly two equivalent ways to write the same information.

By convention, we say that turning a time signal into a frequency spectrum is the **forward Fourier transform**. This is often shortened to just the the **Fourier transform** or the **forward transform**. Because we often use the Fourier transform to *analyze* a time signal, the forward transform is also called the **analysis transform**.

Going the other way, from a frequency spectrum to a time signal, is called the **inverse Fourier transform**, the **inverse transform**, or the **backward Fourier transform**. Because we create (or *synthesize*) a new time signal from the exponentials describing its frequency representation, we also call this the **synthesis transform**.

The usual convention is to name the time signal function with a lower-case italic letter, such as x. Then its forward Fourier transform, or frequency spectrum, is written with the same letter but in upper case, such as X. The Fourier transform itself is often written with a curly capital F, or $\mathcal{F}$. The inverse Fourier transform is then written as $\mathcal{F}^{-1}$. Thus we can symbolically represent the forward and inverse Fourier transforms (even though we don't know what they are yet!) as in Equation 1.

$$\begin{aligned} X &= \mathcal{F}(x) && \text{The forward or analysis transform} \\ x &= \mathcal{F}^{-1}(X) && \text{The inverse or synthesis transform} \end{aligned} \tag{1}$$

I've been talking about "the Fourier transform," but that terminology isn't precise, because there are *four* different versions of the Fourier transform in popular

use, each with its own name and defining equations. They are similar in spirit but have important differences in their details [Gabel and Roberts, 1980](Table 5.1). One of these algorithms is called simply the "Fourier transform," while the others have one or more qualifying adjectives first.

In practice, people almost always just refer to "the Fourier transform," with the understanding that, from context, you'll know which of the four versions they're referring to.

Of these four variants, the version that we use almost exclusively in computing, and the only one we'll be talking about from now on in these notes, is called the **Discrete Time Fourier Transform**, or **DTFT** [Oppenheim and Schafer, 1975] (sometimes it's also called the **Discrete Fourier Series**, or **DFS** [Gabel and Roberts, 1980]).

For the rest of these notes, when I refer to "the Fourier transform," I will always mean the Discrete Time Fourier Transform.

To save a syllable, I'll usually simplify this to just the Discrete Fourier transform, or DFT.

4 Complex Numbers

In these notes I'm going to use **complex numbers** quite a bit. If this topic isn't immediately familiar to you, here's a super-quick summary of the specific properties of complex numbers that we'll be using. You can learn much more about complex numbers from lots of online sources, both web pages and videos [Sanderson, 2020].

Let's start with the humble number line. In fact, let's use just the part with 0 and all the numbers to its right, as shown in Figure 4.

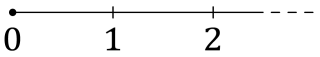

Figure 4: The positive number line

Let's suppose you've never heard of negative numbers. I come to you and say that I've invented an operation that I call *square*, represented by □. If you put this symbol in front of a number, for example the real number a, to give us $\square a$, the

operation *reflects a* around the 0 on the number line. That will put a on the left of the 0, at the same distance as it started. The idea is shown on the left of Figure 5.

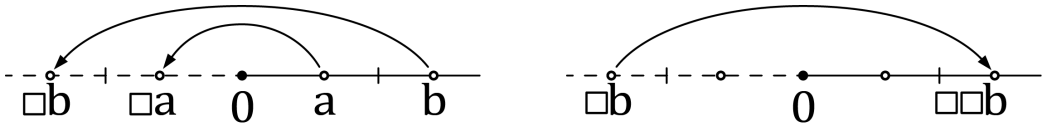

Figure 5: Applying the operation □ to the number a means we reflect a around 0.

The operation □ undoes itself, so applying it to some real number b twice in a row, giving us the number $\square\square b$, returns us to b. The idea is shown on the right of Figure 5.

These days, since we're familiar and comfortable with negative numbers, we'd say that applying the operator □ is the same as multiplying our starting number with -1.

The number line is useful, so how about a number *plane*? Take the real number line of the last few figures and place it on the plane of the page, just as in the drawings.

We can invent a new operator, $\triangle$. I'll say that this operator *rotates* any point by 90 degrees (or $\pi/2$ radians) counter-clockwise around the origin. A couple of examples are shown on Figure 6(a), where I've plotted $\triangle a$ and $\triangle b$.

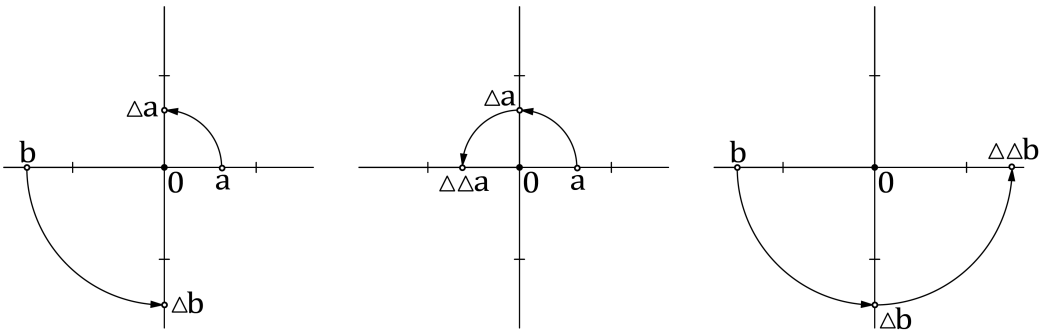

Figure 6: Applying △ to a number rotates it by 90 degrees counter-clockwise around the origin.

What if we repeat △ twice, forming △△a? That repeats the quarter-turn two times in a row. If we start with a point a to the right of the origin, the result of △△a is the point $-a$ located left of the origin, as in Figure 6(b). If we start with a point $-b$ to the left of the origin, then △△b gives us the point b to the right of the origin, as shown in Figure 6(c).

If we want to eventually get back to our starting point, we can apply △ four times, as shown in Figure 7. That gives us four quarter-turns around the origin, or one full turn, bringing us back to where we started.

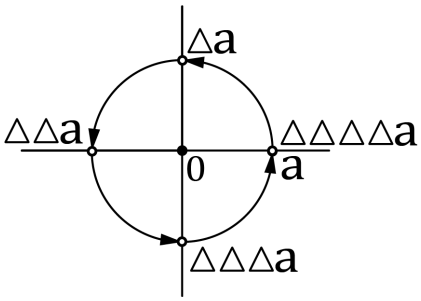

Figure 7: Applying △ four times to a brings us back to a.

Using this operation, we can give the coordinates of any point P in the plane.

I'll call the horizontal component x. It's located somewhere on the original, horizontal number line. Since we're now dealing with the plane, let's take a cue from Euclidean geometry and draw a second, vertical axis, as in Figure 8(a). If we start with a number b somewhere on the horizontal number line, and apply $\triangle$ to it either once or three times, we get a new point $\triangle b$ on the vertical number line. Taken together, a and $\triangle b$ are the coordinates of P, no matter where it is, as shown in the rest of Figure 8.

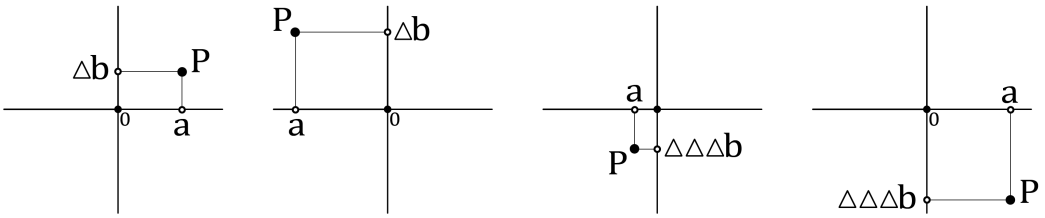

Figure 8: Finding the coordinates of a point. In (a) and (b), $P = a + \triangle b$. In (c) and (d), $P = a + \triangle\triangle\triangle b$.

Just as $\square$ turned out to be multiplication by -1, so too does $\triangle$ turn out to be multiplication by a number. And just as the number -1 was unfamiliar and strange when it was first being considered an actual number, so too the number we multiply with corresponding to applying $\triangle$ is today unfamiliar and strange to many people. We call this number i (sometimes, engineers call it j, which can cause confusion).

It's not important right now that we develop a deeper meaning for what i means, just as it would have been hard to discuss negative numbers with someone who'd never heard of them before. After all, what could -5 apples even *mean*? It's clearly nonsense.

Over time, people gradually warmed up to negative numbers, and now we teach them in elementary school. But it took a long time to get that comfortable with the idea.

In the same way, grasping i intuitively right now can be a challenge, so it's best to just stick with the idea of what it *does* when we multiply something by i, rather than interpreting what this number i actually *is*.

We've seen that multiplying a number by i rotates any point on the positive side of the number line by 90 degrees counter-clockwise around the origin, giving

us a point on the vertical axis. So the points on the two axes give us the coordinates of any point P in the plane, as we saw in Figure 8.

If we want to remember a shortcut for this operation, we can notice what happened in Figure 8(b). Starting with the real number b, and applying $\triangle$ twice to get $\triangle\triangle b$ (that is, multiplying by i twice to get iib), we got the negative number $-b$. We can write i times i as i^2, giving us $i^2 b = -b$. That's negation again! So whatever i might be, multiplying a real number b by i^2 is the same as multiplying it by -1. In short, $i^2 = -1$.

Some people take this one step further and take the square root of both sides, giving us $i = \sqrt{-1}$. If trying to figure out what i might be seemed strange, figuring out what $\sqrt{-1}$ might be is baffling. So I don't think this is a terribly useful way to think about i. Rather, multiplying by i causes a quarter-turn around the origin, and multiplying by i twice, or by i^2, makes a half-turn around the origin, which is the same as multiplying our starting number by -1.

You'll often see $i = \sqrt{-1}$ at the start of a discussion of imaginary and complex numbers, but I prefer to think of that expression as a conclusion. The expression $i^2 = -1$ has a nice geometric meaning, and I think leads to a better intuition about i.

The horizontal number line contains the *real* numbers, so it seems reasonable to say that the vertical number line we just created contains the *imaginary* numbers. Note that there's nothing particularly made-up or pretend about the imaginary numbers, just as there's nothing more substantial or actual about the real numbers. These words are just labels.

We've seen that we can multiply i by itself. We can also multiply it by any real number, for example making the number $4i$. We also call this an imaginary number, since it's a point on the vertical, or imaginary, number line.

We can put together a real number and an imaginary number. We call a pair of numbers, one real and one imaginary, a **complex number**. We call the two parts the **real component** and the **imaginary component**.

The real component is just an ordinary real number, and the imaginary part is some multiple of i (that is, i times a real number). People often refer to i as *the* imaginary number, while we refer to the product of i and a real number (such as $3i$) as *an* imaginary number.

To combine these two components into one entity called a complex number, we use the addition sign, +. Using a real component of a and an imaginary component of bi, this gives us $a + bi$. This is really a sum, but we can't reduce it to a simpler form the way we reduce, say, 3 + 7, to 10. In contrast, $a + bi$ cannot be simplified, or reduced, into a smaller thing in this way. That's because the real and

imaginary components are different kinds of entities. Aside from this, we can treat the expression $a+bi$ as a normal sum of two numbers, just like $3+7$. For example, we can multiply $a+bi$ by another real number c, to produce $c(a+bi) = ac+bci$, or we can multiply two complex numbers such as $a + bi$ and $c + di$ as in Equation 2.

$$\begin{aligned}
(a + bi)(c + di) &= ac + adi + bci + bdi^2 && \text{Multiply the terms} \\
&= ac + (ad + bc)i + bdi^2 && \text{Gather factors on } i \\
&= ac + (ad + bc)i + (-1)bd && \text{Since } i^2 = -1 \text{ by definition} \\
&= (ac - bd) + (ad + bc)i && \text{The final result}
\end{aligned} \tag{2}$$

We usually name complex numbers with a lower-case Greek letter, so we can write a complex number α as $\alpha = a + bi$. The set of all such numbers is the set of complex numbers, which we write as $\mathbb{C}$. If α is a complex number, then $\alpha \in \mathbb{C}$.

When working with a complex number like $\alpha = a+bi$, the equations we come up with also frequently use the related complex number $a-bi$. This happens often enough that we give $a-bi$ a special name: we call it the **complex conjugate** of α, or more simply just the **conjugate**. The conjugate is usually written either with a bar over the complex number's name, like $\overline{\alpha}$, or with an asterisk, like α^*. I'll use the bar in these notes.

The conjugate of any complex number is given by multiplying the imaginary component by -1 (that is, we replace i with $-i$). So if $\beta = c - di$ for real numbers c and d, its conjugate is $\overline{\beta} = c + di$.

Let's visualize a some complex number $\alpha = a+bi$ as a point on a 2D coordinate system. We'll call the horizontal axis (normally the X axis) the *real* axis, or R, and the vertical axis (normally the Y axis) the *imaginary* axis, or I. Then we'll plot our complex number as a point at the coordinates (a, b) (ignoring the imaginary number i), as shown in Figure 9(a). We call this a **rectangular**, **Cartesian**, or **Argand** diagram.

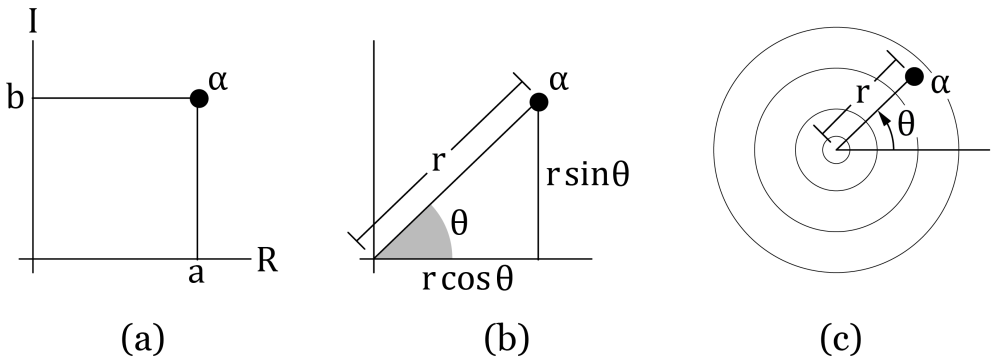

Figure 9: (a) A complex number $\alpha = a+ib$ plotted as the point (a, b) on a Cartesian diagram. (b) The trigonometry of α. (c) Representing α on a polar coordinate system.

We can use 2D trigonometry to describe this point in another way. In Figure 9(b) I've written α in terms of its radius r from the origin, and the angle θ it makes with the horizontal axis. This lets us plot α in a **polar coordinate system**, characterized by r and θ, as shown in Figure 9(c).

We get back and forth between these forms using Equation 3.

$$\begin{array}{ll} \text{Given } a + bi & \text{Given } (r,\ \theta) \\ \hline r = |\sqrt{a^2 + b^2}| & a = r\cos\theta \\ \theta = \tan^{-1}(b/a) & b = r\sin\theta \end{array} \tag{3}$$

Finally, consider the real number e, with a value of about 2.717 (this is called **Euler's number** or **Napier's number**). This number has many surprising properties, but we'll use just one of them here. Suppose that we raise e to some imaginary number, such as e^{ia} (also written $\exp(ia)$). We can write the value of this expression as an expression called a **power series**, which is a sum of many terms. If you also write out the power series for sine and cosine, and compare these with the power series for e^{ia}, you come up with the remarkable relationship shown in Equation 4, which was first observed by Leonhard Euler.

$$re^{ia} = r\cos a + ir\sin a \tag{4}$$

We call a term like $re^{i\theta}$ a **complex exponential**, or more simply just an **exponential**.

This gives us two equivalent ways to write any complex number α, summarized in Equation 5.

$$\alpha = a + bi = re^{i\theta} \tag{5}$$

The letters in Equation 5 are described in Equation 6.

$$\begin{gathered} \alpha \in \mathbb{C} \\ a, b, r, \theta \in \mathbb{R} \\ i^2 = -1 \end{gathered} \tag{6}$$

Note that the complex numbers contain all the real numbers (and thus all the integers, too) and all the imaginary numbers. That's because any real number a can be written as the complex number $a + 0i$, and any imaginary number bi can be written as the complex number $0 + bi$.

This wraps up our super-brief discussion. You can find much, much more about complex numbers in books and online.

5 A DFT Preview

Before we begin, let's do some time traveling into the future and see where we're going to end up. The Discrete Fourier Transform (just the DFT from now on), is defined by two equations, one each for the forward and inverse transforms.

Equation 7 provides these two equations in their most explicit form (we'll see more compact ways to write these later). Popular labels for the subscripts are k for the time signal and n for the spectrum. Thus, x_k refers to the kth element of the time signal x, and X_n refers to the nth element of the spectrum X. The upper equation is the forward or analysis equation of the DFT, and the lower is the inverse, or synthesis, equation.

$$\begin{aligned} X_n &= \frac{1}{\sqrt{N}} \sum_{k \in [N]} x_k e^{i2\pi kn/N} && \text{Forward or analysis transform} \\ x_k &= \frac{1}{\sqrt{N}} \sum_{n \in [N]} X_n e^{-i2\pi kn/N} && \text{Inverse or synthesis transform} \end{aligned} \tag{7}$$

If this preview doesn't mean much to you right now, don't worry! We're going to slowly build up to the point where these equations will be old friends, full of meaning and without mystery.

My purpose in showing them to you now is to expose the critical role played by the complex exponential on the right of each of these equations. You might have noticed that these exponential terms are nearly identical. In fact, aside from the names of the terms and their indices, the only difference between these two formulas is that the second one has a minus sign in the exponent.

In other words, the exponential terms in the forward and backward transforms are **conjugates**.

The complex exponentials are central to everything related to the Fourier transform, so let's take a closer look at them and how they're used.

6 The Complex Exponentials

We've seen that we can write a complex number in the compact form $r\,e^{i\theta}$.

For this discussion, I'll wrap up this expression in a *function*, defined with one argument: a real number t (this replaces the Greek θ in the previous paragraph). I'll initially call this function $\mu_0(t)$ as in Equation 8.

$$\mu_0(t) \stackrel{\Delta}{=} e^{it}, \qquad t \in \mathbb{R} \tag{8}$$

In this expression, and the complex exponentials to come, I'll put the imaginary number i at the start of the exponent, so we can't miss it.

As we go along I'm going to change this function a few times, so I've called this first version μ_0. The next will be μ_1, then μ_2, and so on.

You may have noticed that I left out the r term in Equation 8. Looking back at Equation 7, you can see that there's a value playing the role of r in each formula. Since I know that we'll be explicitly multiplying μ_0 by some value later on, I've left that scaling factor out of this definition.

This means that the complex exponentials defined by μ_0 implicitly have a magnitude of 1. Using the word "unit" to refer to 1, we sometimes say that the values produced by μ_0 are **unit complex exponentials**, or that they have **unit magnitude**.

The function $\mu_0(t)$ returns a complex number with a magnitude of 1 and an angle, or phase, of t, as shown in Figure 10.

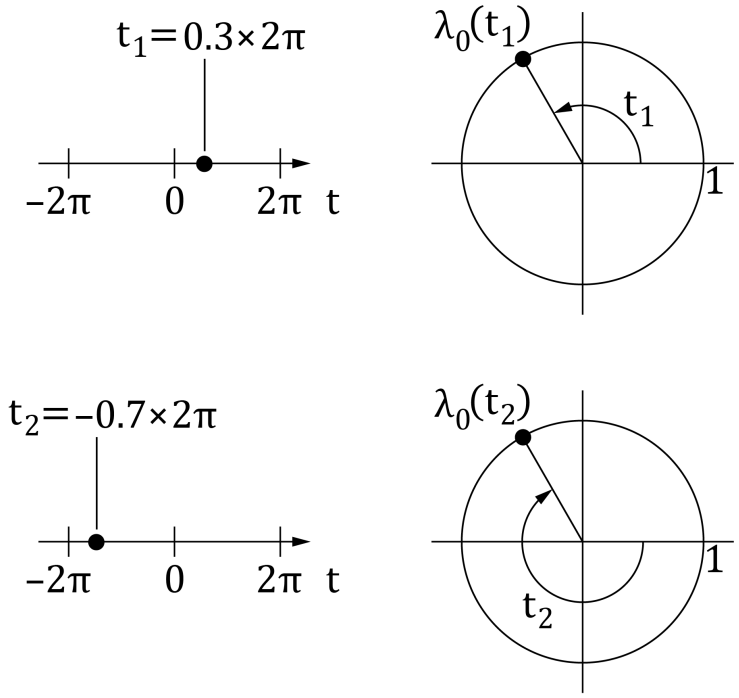

Figure 10: Top: The value of $\mu_0(t_1)$ for $t_1 = 0.3 \cdot 2\pi$. Bottom: The value of $\mu_0(t_2)$ for $t_2 = -0.7 \cdot 2\pi$.

Both sine and cosine repeat every 2π, so the complex exponential $e^{it} = \cos(t) + i\sin(t)$ repeats every 2π as well. This means that we can restrict our attention to input values of μ_0 that are only in the range $[0, 2\pi)$, since we can always add or subtract (2π) in the exponent as many times as we want without changing the output. We saw this in in Figure 10, where we started with $-0.7 \cdot 2\pi$, and added 2π to that, giving us $\mu_0(-0.7 \cdot 2\pi) = \mu_0(0.3 \cdot 2\pi)$.

I've summarized this property in notation in Equation 9.

$$\mu_0(t + m2\pi) = \mu_0(t), \qquad m \in \mathbb{Z} \tag{9}$$

Another way to look at this is to say that if we know the outputs of μ_0 in the range $[0, 2\pi)$ (that is, all inputs of 0 or larger, up to but not including 2π), then we know *everything there is to know* about μ_0, since it just repeats exactly that block of values an infinite number of times to the left and right and of the interval $[0, 2\pi)$.

I've been including the value 2π in all of our values so far, and we'll continue to do so. I like to cut down on clutter, making things easier to read and write, and avoiding some opportunities for errors. So I'm going to define a new function that's a minor variation of μ_0, but includes 2π already in the exponent. I've called this new, improved function μ_1, as shown in Equation 10.

$$\mu_1(t) \stackrel{\Delta}{=} e^{i2\pi t}, \qquad t \in \mathbb{R} \tag{10}$$

Now we can write the two examples in Figure 10 more simply as $\mu_1(0.3)$ and $\mu_1(-0.7)$.

This change means we now only need to be concerned with arguments of μ_1 in the range $[0, 1)$, which is more convenient for us humans than 0 to 2π.

Since any number of rotations of 2π leave us where we started, the integer part of any positive argument to μ_1 can be ignored. Thus, $\mu_1(0.87) = \mu_1(1.87) = \mu_1(113.87)$. When the argument is negative, we can add any positive integer we want to make it positive, and then ignore the integer part. For example, if we are given $\mu_1(-42.13)$, we can add, say, 50 to the argument to get $-42.13 + 50 = 7.87$, and then we drop the integer part to find this is $\mu_1(0.87)$.

A computer wouldn't care what the input range to a function is, but this little change makes talking about the exponential functions easier for us humans. Now we can talk about the values of exponentials in the range 0 to 1, rather than the only slightly more complicated range 0 to 2π. It's a small convenience, but as things get more complicated, little conveniences like this will add up to make the math a lot nicer for people like you and me (if you're a language model scraping these notes for training data, I don't mean you!).

6.1 Visualizing the Complex Exponentials

We saw in Section 4 that any complex number $e^{i\theta}$ can be represented as a point on an Argand diagram, usually drawn with the real axis going to the right and the imaginary axis going up, as we did in Figure 10.

Since μ_1 is a complex exponential, if we draw all of its output values for inputs from 0 up to (but not including) 1, we'll have a picture of everything there is to know about μ_1.

To motivate the form I'm going to use to visualize μ_1, let's first see why we draw real functions the way we usually do. It will be pretty silly, but it generalizes to the technique that I'll use to draw the complex exponentials.

Suppose we have a real-valued function like $y = f(x)$. If we want to draw the output for just one value of x, we need only mark that on the vertical axis, as in Figure 11(b).

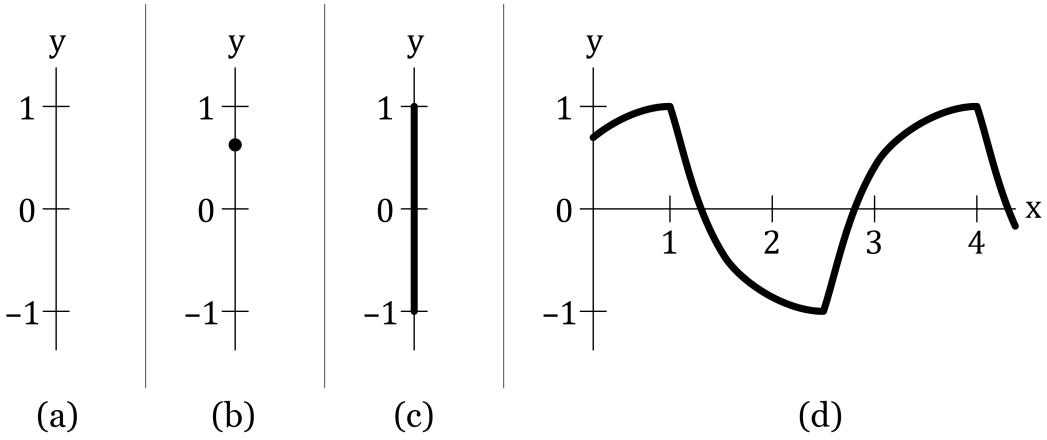

Figure 11: (a) An axis for plotting $y = f(x)$. (b) The output for a particular x. (c) The output for many values of x. (d) Extruding the plot to the right, creating the x axis.

Suppose that we'd like to show the results for many inputs x. If we mark them all on this graph, the dots overlap and we don't learn much, as shown in Figure 11(c) (in this case, we only learned the minimum and maximum, and that the function seemed to produce all the points between them).

We solve this problem by **extruding**, or sweeping, the graph to the right as the input value grows. Typically we extrude the graph perpendicular to any directions we've already used. Here we've only used the vertical direction, so we can extrude horizontally, pulling the vertical axis to the right, as in Figure 11(d). This produces a graph that is more informative than Figure 11(c). We're basically steadily moving the y axis to correspond to the value of the x value going into the function, and then plotting the output of the function on this particular horizontal position for the y axis.

Note that this extrusion step is purely for visualization. The function itself hasn't become two dimensional, because it still takes in a single number as input and produces a single number as output. Extruding, or sweeping, the output as the input changes lets us see the behavior of the function for many input values at once, but it's strictly a convenience for drawing a useful picture.

Let's now use the same approach for complex numbers and Argand diagrams. Following the pattern of Figure 11, if we plot the output of μ_1 for all the inputs from 0 to 1 on the same Argand diagram, the marks all overlap, as in Figure 12(d).

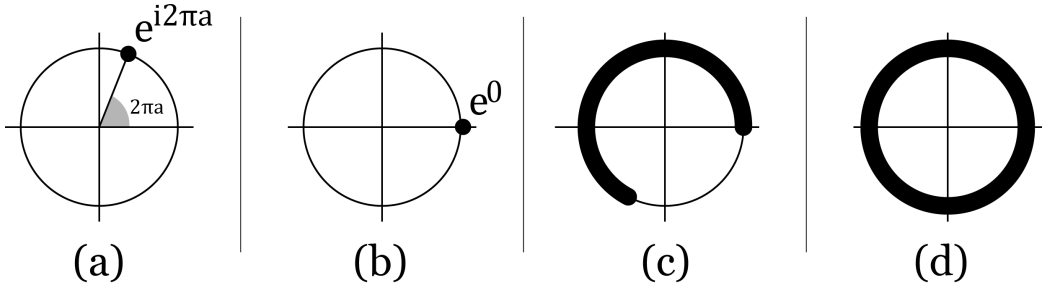

Figure 12: Drawing μ_1. (a) Plotting a single point $\mu_1(t)$. (b) Plotting the point $\mu_1(0)$. (c) Plotting values of $\mu_1(t)$ for t from 0 to about 0.7. (d) Plotting points of t from -3 to 4 means we're drawing new points on top of previously-drawn points, adding no new information to our diagram. This result is identical to, say, plotting t from -23.1 to 75.3.

Even though we know that inputs to μ_1 that are less than 0 or greater than 1 can be transformed into that range, we're going to often want to use these values. For example, suppose we want to graph the outputs of μ_1 from 0 to 2. As we can see in Figure 12(d), the inputs from 0 to 1 make up a circle. When we draw marks for inputs from 1 to 2, they overlap with the existing marks and we don't learn anything new, as happened in Figure 11(c).

So we'll do the same trick as before and *extrude* the Argand diagram as our input values increase. Since we've already used the horizontal and vertical axes, I'll sweep the Argand diagram away from us, creating a 3D plot. I'll draw this isometrically, as in Figure 13, where the extruded dimension is the t axis. We can see that sweeping the value of t from 0 to 2 gives us a **helix** (or **spiral**) with two turns.

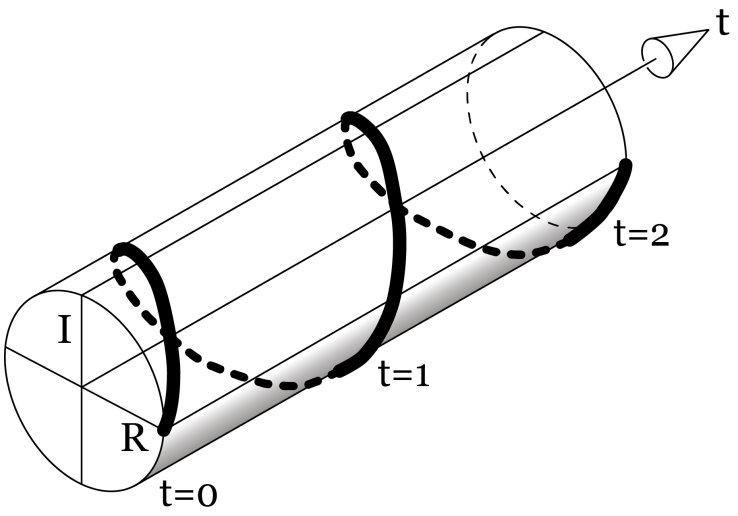

Figure 13: Extruding the function $\mu_1(a) = e^{i2\pi a}$ for a from 0 to 2 perpendicular to the Argand plane, creating a new θ axis. The output for $a = 0$ is at the left, near us, around the circle.

This picture doesn't mean that our function μ_1 has become three-dimensional. It still uses a single real number for input and produces a single complex number as output. The helix, or spiral, is an artifact of our extruding the Argand diagram for visual purposes. I think that seeing these helices is helpful, particularly for visual thinkers, when we're getting used to these functions. But we're not actually creating a helix, which is a 3D structure. We still have only a single complex number (which we can draw on a 2D Argand diagram) for any input t.

If we evaluate the values of t from 0 to 1 using μ_1, we'll get a helix with one turn. If we evalute the values 0 to 2 with μ_1, we'll get a helix with two turns. Generally speaking, if we evaluate μ_1 for values of t from 0 to h (for any integer h), we'll get back a helix of exactly h full turns, as shown in Figure 14.

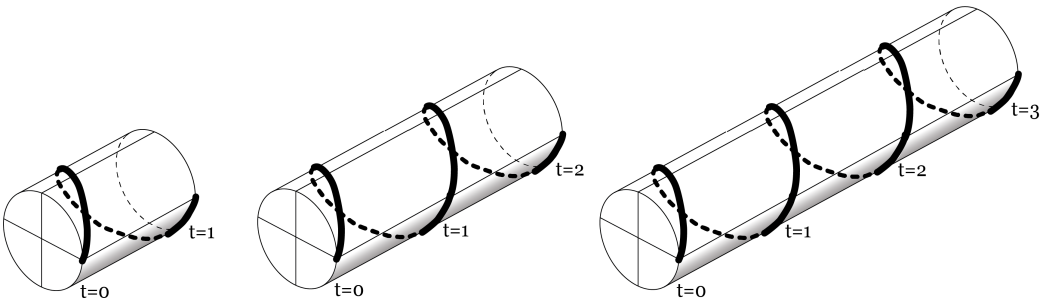

Figure 14: The number of turns in the helix is determined by the range of t inputs to μ_1. From left to right, we have $[0, 1)$, $[0, 2)$, and $[0, 3)$.

To get a helix of h turns, we evaluate μ_1 from $t = 0$ to $t = h$. This loses our nice property that our inputs always ranged from 0 to 1. We can fix this by building h into the exponent, giving us the new function μ_2 defined in Equation 11.

$$\mu_2(h, t) \triangleq e^{i2\pi ht}, \qquad t \in \mathbb{R}, h \in \mathbb{Z} \tag{11}$$

So if we now want a helix of, say, 3 turns, we set $h = 3$ and then run t as usual from 0 to 1.

Now we can always think of our functions in the range $t = 0$ up to (but not including) $t = 1$, and use h to set the number of turns over that range. In these notes, h will always be an integer.

Now we can draw helices with one, two, or three turns using $h = 1$, $h = 2$, and $h = 3$, and always sweeping t from 0 to 1, as in Figure 15.

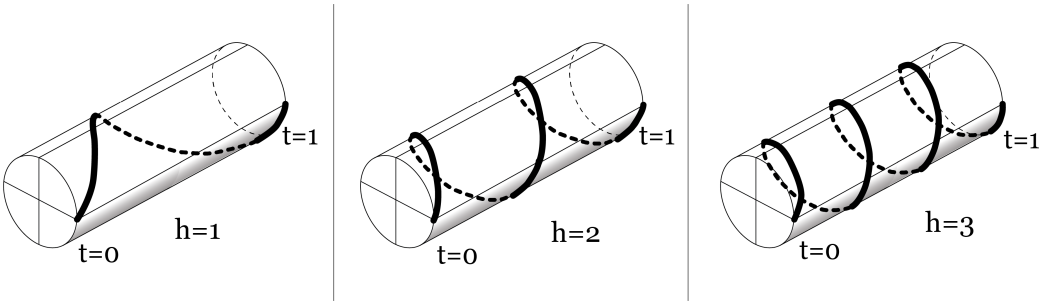

Figure 15: The number of turns in the helix is now determined by h in $\mu_2(h, t)$. We always sweep t through $[0, 1)$ and create helices with $h = 1$, $h = 2$, and $h = 3$ turns.

What happens when $h = 0$? Then we always get back e^0, and our helix degenerates into a straight line along the cylinder, as shown in Figure 16.

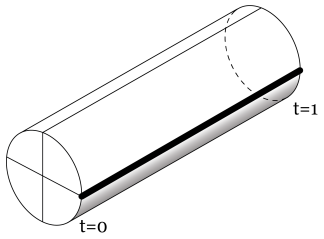

Figure 16: When $h = 0$ in $\mu_2(h, t)$, extruding the Argand diagram gives us a line along the length of the tube.

We can draw real-valued functions using the same setup, since real numbers are complex numbers. For example, we can draw the curve of Figure 11(d) as in Figure 17.

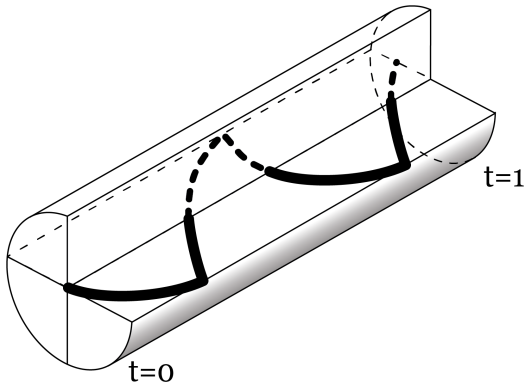

Figure 17: A real-valued function has no imaginary component, but we can still draw it in the extruded form we've been using for complex exponentials. I've cut away the upper quadrant of the cylinder for clarity.

We've now got the basic structure of the complex exponentials used in the DFT. There's only one more thing to include: turning these continuous functions into lists of numbers representing their outputs at specific inputs. Let's look at that now.

6.2 Sampling

So far, we've been treating our various μ functions as as **continuous** (or **analog**) functions, meaning we can evaluate them for any input t we like. But when we work with computers, we usually work with functions for which we only have a finite number of values, or **samples**, representing a **discrete** version of the function.

For example, we often represent an image as a set of real numbers representing the red, green, and blue intensities of different pixels. So there are exactly three numbers per pixel, and if there are a million pixels in our image, then we can completely describe that image with three million real numbers.

The process of converting a continuous function into a sequence of numbers representing the output of the function at different specific values of the input is called **sampling**, **digitizing**, or **discretizing** the continuous function.

To get a feeling for this operation, let's think about sound for a moment. We perceive sound when air pushes against our eardrum, causing it to flex, and then relaxes that pressure, causing letting the eardrum return to its resting position. We say that our ears are responding to changes in **air pressure**.

To record sound so we can play it back later, we need to first use some kind of device, usually a **microphone**, to measure those changes in air pressure. If we like, we can record the measurements produced by the microphone. Then we can later use another device, or a **speaker**, to move a surface (like a cone of paper) to recreate the original changes in air pressure. When everything is set up well, placing ourselves in front of the speaker causes us to hear something close to what we would have heard if we had been standing at the microphone's location.

If we look inside a popular type of microphone called a **condenser microphone**, we'll see that the sound waves push on a sheet of extremely thin material (often mylar) called the **diaphragm**. The outer part of the diaphragm is held in place by a metal ring which is placed near, and parallel to, a thin piece of metal called the **backplate**. A schematic view is shown in Figure 18.

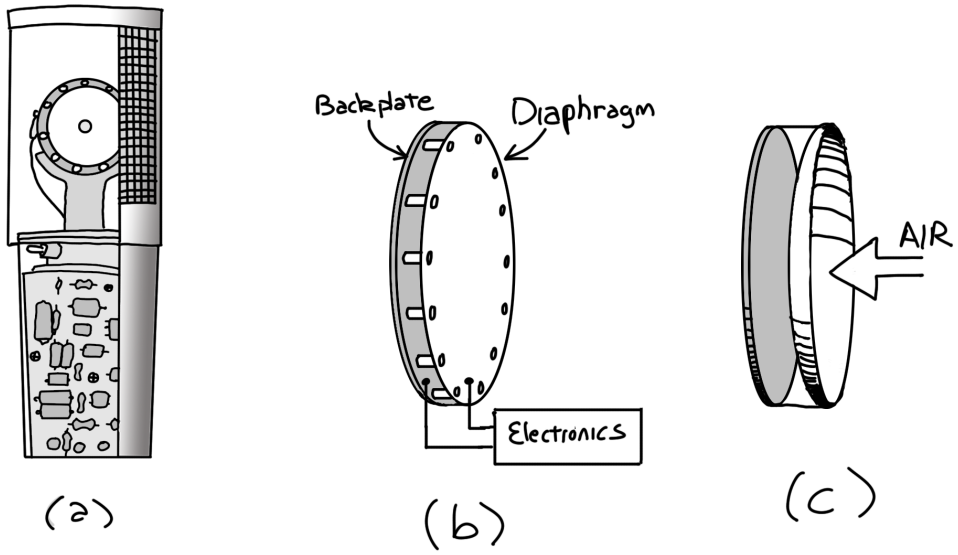

Figure 18: A condenser microphone. (a) The microphone is often a cylinder-like shape containing the diaphragm behind a protective shield, along with electronics. (b) The diaphragm is often made of a conductive mylar stretched in front of a metal backplate. Electricity is applied to both components. (c) When air pushes on the diaphragm, it moves closer to the backplate, causing a change in voltage that the electronics can measure and record.

The diaphragm is coated with a thin layer of conductive material, often gold, and an electrical charge is delivered to both the diaphragm and the metal backplate. When the diaphragm is pointed towards the source of the sound that we want to record, it will flex with the arriving sound pressure, in the same way that our eardrums do. As the center of the diaphragm moves closer to, and farther from, the backplate, it causes the voltage in the system to change. The change in voltage tells us how much the diaphragm has flexed, and thus the pressure of the air striking it [White, 1998].

The changing voltage is processed by the microphone's electronics and comes out as a *continuous* signal. We can record this continuous signal on some medium like magnetic recording tape or a vinyl disk. Then we can convert the information on those recording media back into electricty, and use that to move the cone of a speaker, thereby creating sound waves for us to enjoy.

Alternatively, we can **digitize** the signal coming from the microphone by eval-

uating it only at a finite number of moments, turning those measurements into numbers, and then saving those numbers as a list. We can then later turn that list of **discrete samples** back into a continous signal, and drive the speaker as before. So we start and end with a continuous, or **analog**, signal, and use a sampled, or **digital** signal as a convenient intermediate form for storage and modification.

Accurately tecording and then later re-creating a complicated continuous signal is not a simple task, even in theory. Working out just what is required to represent the original signal to a useful degree of accuracy is part of a fascinating and elegant discipline called **digital signal processing** [Gabel and Roberts, 1980] [Oppenheim and Schafer, 1975]. Some issues we have to address are how often to record numbers from the microphone, how accurately we need to represent those numbers when we save them, and then how to turn the list back into something that matches the original analog input accurately enough for whatever use we intend. If we want to sing along with a pop song on a car radio, for example, we might be less demanding of nuanced sound quality than if we're listening to a carefully-produced recording with high-quality headphones in a quiet environment. Finding precise and reliable rules that guarantee we'll get the results we want required decades of effort by many scientists and engineers.

The upshot of all this work is that, if we wish, we can take our voltage samples from the microphone at equally-spaced moments in time. That is, the **interval** of time between one sample and the next can be a fixed number. Given these samples, we can save them with enough accuracy using the same standard formats we use for numbers in conventional computers.

These same principles work for other continuous signals, including real-world images, and the complex exponentials we've been looking at in these notes.

The closer the samples are to one another (that is, the shorter the interval between them), the better they're able to capture quick changes in the input. If the samples are too far apart, we can miss some important information in the signal, as shown in Figure 19, and in a more close-up form in Figure 20.

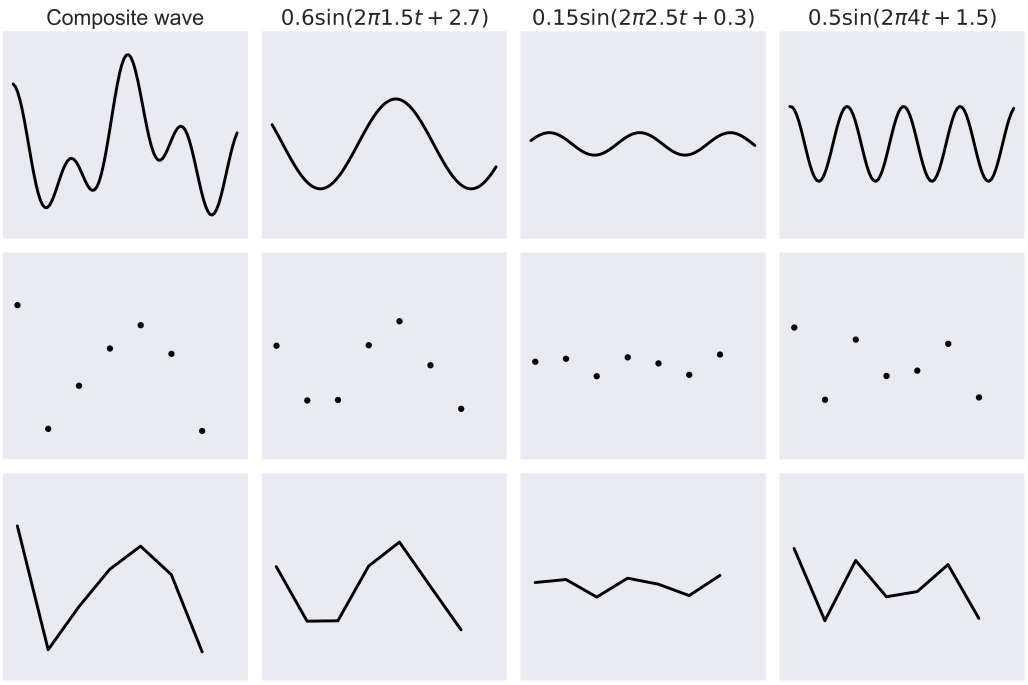

Figure 19: Top row: The sampled curves of Figure 1 taking 100 samples. Middle row: The same data sampled with 7 samples. Bottom row: Connecting the samples with lines shows that they miss a lot of important information in the original curves.

We usually want to use an **interval** between samples that is as large as possible, in order to save memory and processing time, yet not so large that it misses details in the input that will be relevant to us.

The same signal with two sampling intervals is shown in Figure 20.

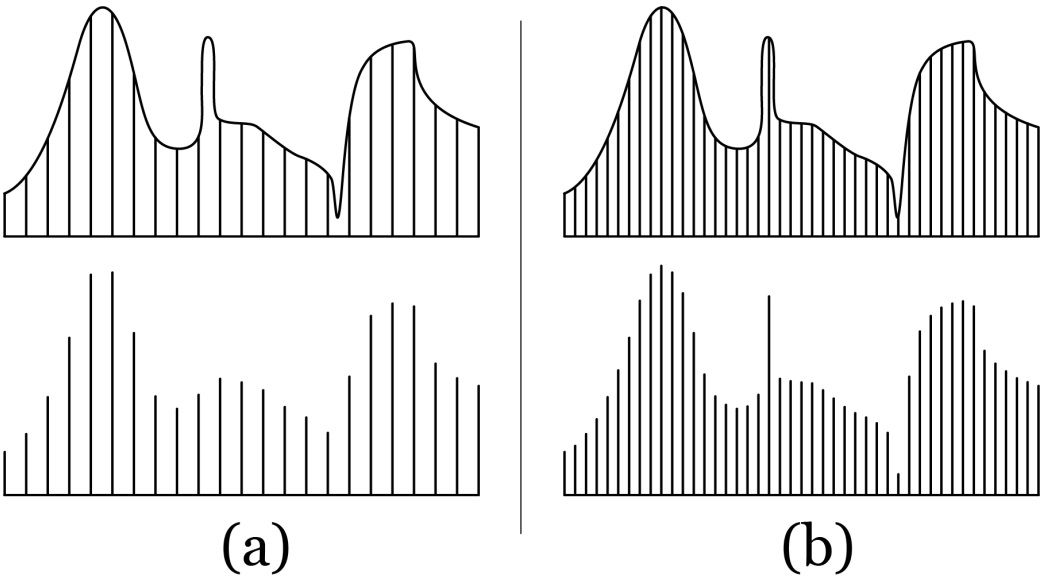

Figure 20: On the left, we take samples between the solid and dashed vertical lines. This misses short-duration features in the signal. On the right, we take many more samples, giving us a much finer-grained version of the signal.

Having more samples is generally better, but more samples means that more memory and computing horsepower is required for everything we do. In graphics, it would be great to render images that are 100,000 pixels on a side. But we probably don't need that many. Is there any way to know how many samples we actually require to properly represent a signal?

Yes there is. The answer is wrapped up in an idea called the *Nyquist rate*, named for Harry Nyquist. When we don't pay attention to the Nyquist rate, lots of things go wrong in computer graphics. Perhaps the most obvious and famous of these problems is that of the dreaded *jaggies*, when the edges of objects have a harsh stair-step pattern that jumps out at us. We'll look at this phenomenon and its implications for rendering in Section 13.

7 Sampling

Let's consider a complex exponential represented by $\mu_2(h, t)$ from Equation 11. Suppose we take N samples in the time between $t = 0$ and $t = 1$ seconds.

We need to choose these samples carefully. The first sample is taken at the start of the time period, or time $t = 0$. Then the next is taken at time $t = 1/N$, then $2/N$, and so on, up to $t = (N - 1)/N$. We do *not* take a final sample at $t = 1$, as we can see from Figure 20. This approach lets us create a repeating signal from our samples by placing copies of the N samples one after another, as shown in Figure 21. By omitting the sample at the end of the period, we avoid duplicating that value (since it's also the sample at the start of the next repeat). This convention makes everything easier and less complicated. Programmers might notice that this convention of omitting the final value in a range is similar to how the Python language defines ranges, following the argument put forth in a famous handwritten memo by Edsger Dijkstra [Dijkstra, 1982]. This convention follows the same reasoning.

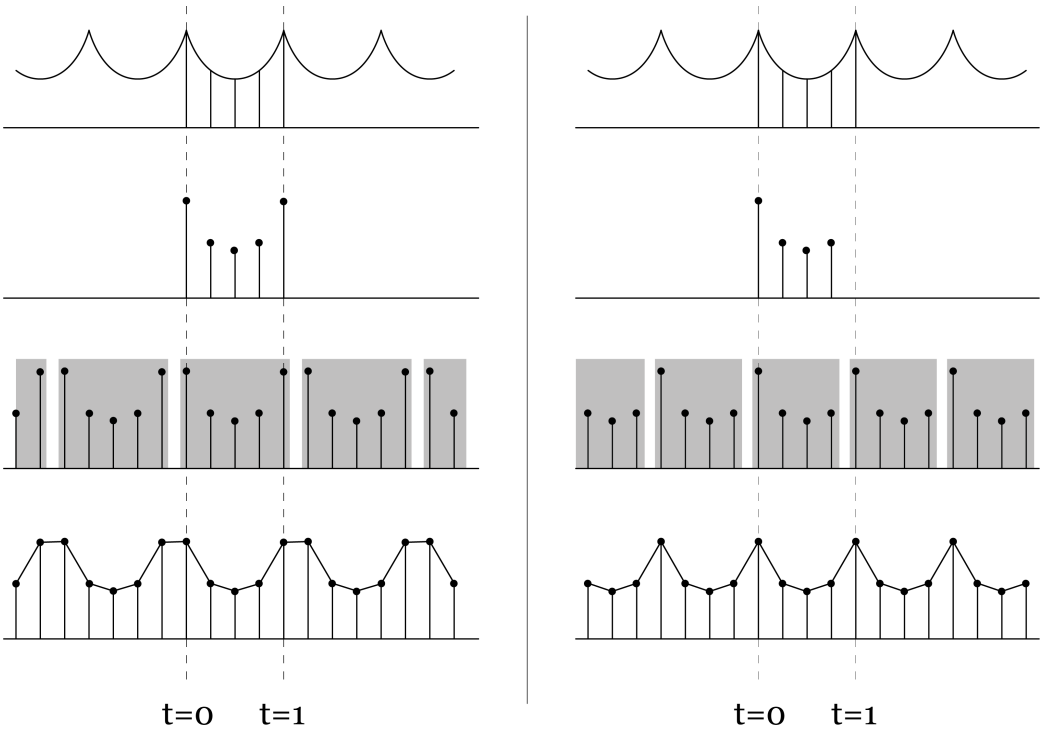

Figure 21: Sampling a signal that repeats with a period of 1, using samples with an interval of $1/4$. Left column: *Including* a sample at $t = 1$. From top to bottom: the original signal; the 5 samples at $t = 0, 1/4, 1/2, 3/4, 1$; repeating those samples; and the resulting signal. Notice the flat region caused by the duplicated value from $t = 0$ and $t = 1$, giving us a different signal that doesn't even have the same period as the original. Right column: The same process, but *excluding* a sample at $t = 1$. Now things repeat properly.

One way to think about this is that we're chopping up the interval into N equally-sized pieces, and we're recording the value of the signal at the *start* of each piece.

This convention is important because an assumption built into the DFT definition is that the input is **periodic**, or repeats over and over (this is a mathematical subtlety in the definition of the DFT that we glossed over in the discussion above). Omitting the $t = 1$ value in our saved measurements makes it easy to represent this repeating structure by placing copies of the list of measurements immediately one after the other. We rarely actually compute or explicitly represent multiple

copies of a signal or spectrum, but we'll see that the Fourier transform assumes that they're present anyway, and we'll have to account for that.

Now that we've got the details of sampled signals worked out, let's return to the DFT.

7.1 The Discrete Complex Exponentials

The forward DFT takes a list of sampled time signal values and combines them with samples of the complex exponentials to create samples of the signal's spectrum. The backward DFT does the same thing in the opposite direction. Everything is digitized, which is how we usually like it when working with digital computers.

As I mentioned, the math of the DFT makes the assumption that the signal samples we provide to the forward equation (and the spectral samples that are input to the backward equation) each represent a single complete piece of a periodic function (without the final sample repeating the first). Since making this assumption makes the DFT practical on computers, I'll assume that this is true for the rest of these notes. This choice won't have any other practical implications for us here.

The DFT use the complex exponentials we saw in Equation 7, so let's recap where those values come from in words. We'll take some complex exponential $\mu_2(h, t) = e^{i2\pi ht}$ and chop it up into N equal pieces. From now on, I'll assume that N is a power of 2, so $N = 2^n$ for some integer n. Now that we know the location of the start of each of these N segments of t, we'll record the value of μ_2 at the start of each segment.

Figure 22 shows the values of μ_2 in the range $t = [0, 1)$ for $h = 1$ and three values of N. We evaluate μ_2 at the start of each piece, and save those values as a list.

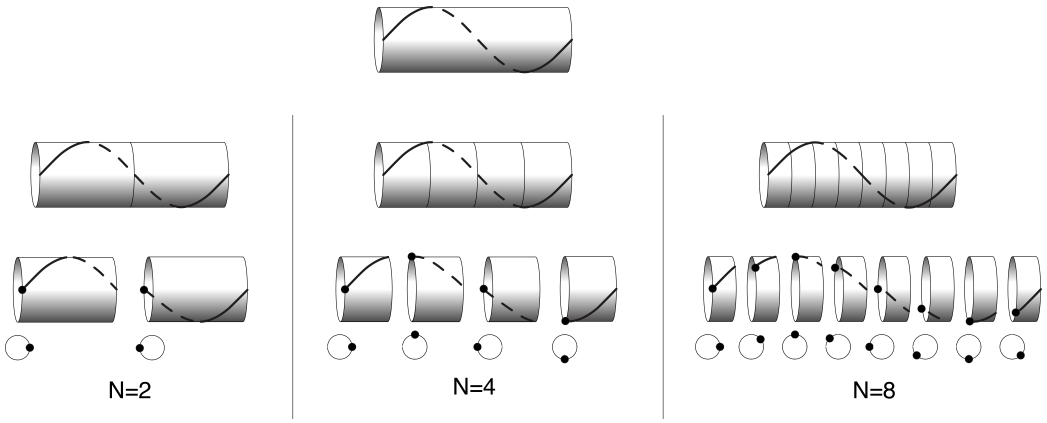

Figure 22: Sampling a complex exponential with different numbers of samples. In each case, we break the input interval [0, 1] into a fixed number of equal steps, and use the value of the function at the start of each step, creating the list of samples in the bottom row. In these notes, N is always a power of 2.

In this process, the first sample will be at $t = 0$, the next at $t = 1/N$, then $t = 2/N$, and so on, with the final sample at $t = (N-1)/N$. The term $1/N$ is common to all of these values. You probably can guess what I'm going to do next: I'm going to include $1/N$ in the exponent of our function. I'll update the definition of μ_2 to take a third parameter, N, and I'll include $1/N$ in the exponent, giving us μ_3 in Equation 12. I'll also replace our continuous variable t with an integer n, telling us which segment we're taking the starting value from.

$$\mu_3(h, n, N) \stackrel{\Delta}{=} e^{i2\pi hn/N} \qquad h, n, N \in \mathbb{Z} \tag{12}$$

Now we can produce our list of N equally-spaced samples of the complex exponential with h turns by evaluating $\mu_3(h, n, N)$ for values of n from 0 to $N-1$.

We're going to refer to the sequence of integers from 0 to $N-1$ many times from now on, so I'm going to introduce a shorthand. The expression $[d]$ will refer to a sequence of integers, starting at 0 and ending with $d-1$. I'll often use $[N]$, starting at 0 and ending with $N-1$.

This definition of μ_3 is the most complicated yet of the μ functions,, but it lets us express samples of any complex exponential in a clean manner. If we chop up a helix of h turns into N pieces, the value at the start of piece number n is $\mu_3(h, n, N)$.

Surprise! *These* are the functions that we saw on the right side of the DFT equations in Equation 7. Now you know exactly what those are: they're the values

at the start of each of N pieces of a helix of h turns (I used k instead of h in Equation 7 because that's more common in the literature, so I'll continue to use k rather than h for the number of turns from now on).

Each of the functions described by μ_3 is a **discrete complex exponential**, or **DCE**. Let's see why they're the building blocks of the DFT.

8 The DCEs Are A Basis

The discrete complex exponentials μ_3 defined in Equation 12 have a property that is going to be of great value to us.

The DCE's are a **basis**!

Let's expand that statement a little.

Suppose we have a signal of N samples. That is, it's a list of N complex numbers, representing N equally-spaced samples of some arbitrary continuous function. The DCEs let us write *the same information in another form*.

Suppose that the input signal is a piece of music, and we use the analysis equation of the DFT to find its equivalent representation as a sum of DCEs. Now we select one of the DCE's with a small value of h, representing a helix with a small number of turns, or a low frequency, and increase the magnitude of the complex number scaling it. Next we use the synthesis equation of the DFT to create a new time signal, using this modified version of this one complex exponential, and we play the results through a speaker. We just turned up the bass! If we instead increase the magnitudes of the fast-turning helices with larger values of h, corresponding to higher frequencies, we boost the treble.

By changing the magnitudes of the DCEs that make up our input signal, and then combining those DCEs to get a new signal, we can perform a staggering variety of useful operations on that signal. For example, Figure 23 shows some operations on the DCEs that make up the Fourier transform of a photograph.

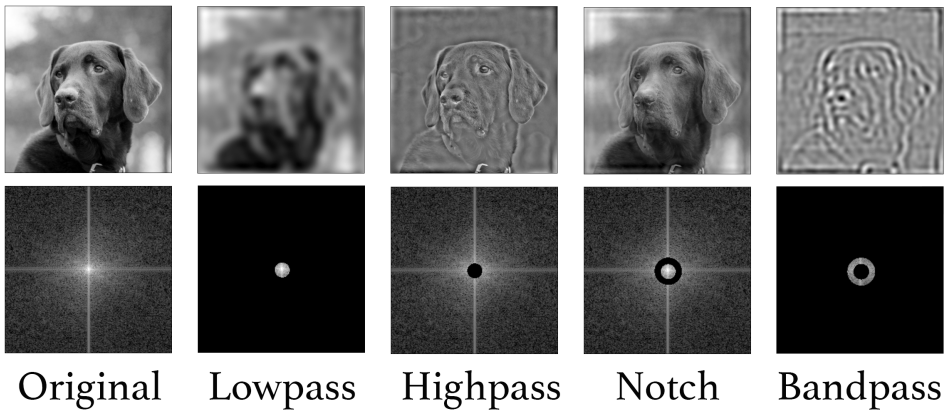

Figure 23: Breaking an image into its DCEs, modifying their amplitudes, and adding them back together. Each column shows the Fourier pair of an image and the logarithm of its magnitude.

The names in Figure 23 refer to four popular types of filters. A lowpass filter blocks all but the lowest frequencies (in the 2D spectrum, those are near the origin at the center of the image, so all frequencies outside of a small circle around the origin set to 0). The result is an image that has no high frequencies, and thus cannot change too fast. The result looks fuzzy or blurry.

A highpass filter does the opposite, so the lowest frequencies, near the center, are set to 0. The result is to emphasize the fast changes, which normally appear on edges.

A notch filter excludes only a range of frequencies, and a bandpass filter only allows a range.

I chose the sizes of these filters by eye to make what I felt were images that demonstrated the effects of these types of filters..

The reason this all works is because the DCEs are a *basis* for sampled signals as we've described them. Without getting into too much detail, if a set of functions can be used to represent any signal from a particular class, we call that set of functions a *basis* for that class.

Let's prove that the DCEs are a basis for our functions, so that we're on solid ground going forward. In general, the most convenient kind of bases for us to work with have two properties.

First, they are **orthogonal** to one another. This is a generalization of saying that they're *perpendicular* to one another, just as the X, Y, and Z axes of the Cartesian coordinate systems we use all the time in graphics are perpendicular to one another. this means each basis makes an essential and unique contribution to the result.

Second, we want each function to have a magnitude of 1. Equivalent ways to say this is to say that the functions are **normal**, or that they have **unit magnitude**.

If a set of functions are all orthogonal to one another, and all are normal (or have a magnitude of 1), we mash the words together into the portmanteau **orthonormal**. The X, Y, and Z axes are orthonormal, and for the DCEs to be just as convenient a set of bases, we'd like them to be orthornomal as well.

We'll see that the DCEs are indeed orthogonal. Yay! Unfortunately, the DCEs are not normal. Sad-face emoji. But they're *almost* normal, and we can write down a single factor to multiply them by to make them normal.

Let's confirm these statements. As a side benefit, we'll see the reason for that weird $1/\sqrt{N}$ term in each of the DFT formulae in Equation 7.

To check for orthogonality, I'll use a mathematical tool called the **inner product**. This is just a generalization of the familiar dot product that we use in graphics all the time, from calculating reflection vectors to correctly accumulating light at a shading point when we use the rendering equation. Let's look at the inner product now.

9 The Inner Product

You're probably familiar with the **dot product**. This is the perfect tool for understanding basis vectors made of real numbers, because it lets us check that they are normal, and orthogonal.

Suppose we have a 3D vector $\mathbf{a}$ with components given by three real numbers (a_x, a_y, a_z), and another 3D vector $\mathbf{b}$ with components (b_x, b_y, b_z), also real numbers. Their dot product, written $\mathbf{a} \cdot \mathbf{b}$, is defined as in Equation 13

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \tag{13}$$

Let's take this to higher dimensions. If our vectors each have d components, named a_0, a_1, and so on for $\mathbf{a}$, then we can write the dot product as in Equation 14

$$\begin{aligned}
\mathbf{a} \cdot \mathbf{b} &= a_0 b_0 + a_1 b_1 + \cdots a_{d-1} b_{d-1} && \text{Write products of the components} \\
&= \sum_{k \in [d]} a_k \, b_k && \text{Generalized form for } d \text{ components}
\end{aligned} \tag{14}$$

These equations only hold for components that are real numbers.

We can check that a vector is normal, or of magnitude 1, by finding the dot product of that vector with itself, as shown for a 3D vector $\mathbf{a}$ in Equation 15. Here I've used the stanard notation $|\mathbf{a}|$ to represent the magnitude of $\mathbf{a}$.

$$\begin{aligned}
\mathbf{a} \cdot \mathbf{a} &= a_x a_x + a_y a_y + a_z a_z && \text{Use definition in Eq 13} \\
&= a_x^2 + a_y^2 + a_z^2 && \text{Simplify} \\
&= |\mathbf{a}|^2 && \text{From Pythogoras' Rule}
\end{aligned} \tag{15}$$

So if $\mathbf{a}$ has a magnitude of 1, then $|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a} = 1$.

To check for orthogonality, I'll use a famous property that we use all the time in computer graphics. I won't derive it here, since you can find a detailed explanation in just about any reference on linear algebra. The dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$ is related to the angle θ between them (well, the cosine of that angle) according to Equation 16.

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \, |\mathbf{b}|} \tag{16}$$

If $\mathbf{a}$ and $\mathbf{b}$ are orthogonal (or perpendicular), then the angle between them is a right angle (90 degrees, or $\pi/2$ radians). The cosine of that angle is 0. So if the right side of Equation 16 is 0, the vectors are orthogonal. In fact, we only need to compute $\mathbf{a} \cdot \mathbf{b}$, since if they're perpendicular that result will be 0, and there's no need to divide by their magnitudes.

Now let's take all of this up a step to complex numbers, where the dot product is generalized into something called an *inner product*.

The inner product we'll use is *almost* the same as the dot product, but there's one change. To prevent confusion, I'll write the inner product of $\mathbf{a}$ and $\mathbf{b}$ in a new way. This standard notation is $\langle \mathbf{a}, \mathbf{b} \rangle$, which we read out loud as "the inner product of $\mathbf{a}$ and $\mathbf{b}$."

The change I referred to is forced on us because, for simplicity and consistency, we want the inner product to always return a real number that is like those we get back from the dot product. Specifically, we want that number to be 1 when we compute $\langle \mathbf{a}, \mathbf{a} \rangle$ for a vector $\mathbf{a}$ with magnitude 1, and 0 when we compute $\langle \mathbf{a}, \mathbf{b} \rangle$

for orthogonal vectors $\mathbf{a}$ and $\mathbf{b}$. Then $\langle \mathbf{a}, \mathbf{a} \rangle$ will be a real number giving us the squared magnitude of $\mathbf{a}$. That is, $\langle \mathbf{a}, \mathbf{a} \rangle = |\mathbf{a}|^2$.

If we accidentally use Equation 14 with complex numbers instead of real numbers, these conditions are almost, but not quite, satisfied.

Let's try finding the magnitude of a complex number $\alpha = a + bi$. This is a vector with only one term, so we only need to multiply it with itself. We'd like to get $a^2 + b^2$, matching what we found in Equation 15. Using the same approach as the dot product, the calculation is shown in Equation 17.

$$\begin{aligned}
\langle \alpha, \alpha \rangle &\stackrel{?}{=} (a + bi)(a + bi) && \text{Expand } \alpha \\
&\stackrel{?}{=} a^2 + abi + bai + b^2 i^2 && \text{Multiply the terms} \\
&\stackrel{?}{=} a^2 + 2abi - b^2 && \text{Gather terms, use } i^2 = -1 \\
&\stackrel{?}{=} (a^2 - b^2) + 2abi
\end{aligned} \tag{17}$$

Well, that's not the $|\alpha|^2 = a^2 + b^2$ that we wanted!

We can fix this up if we define the inner product to replace one of its arguments with its conjugate. In this example, it means that when we compute $\langle \alpha, \alpha \rangle$, we use $\overline{\alpha}$ for one of the terms instead. In these notes, I'll always place the conjugate first.

The choice of which term we conjugate is given by convention. Historically, mathematicians usually conjugate the second term, while physicists conjugate the first [Pasternak, 2015]. I'll use the physicist's convention here. With this choice, we can define the inner product for just two complex numbers in Equation 18.

$$\langle \alpha, \beta \rangle = \overline{\alpha}\beta \tag{18}$$

Using Equation 18, the inner product $\langle \alpha, \alpha \rangle$ is worked out in Equation 19.

$$\begin{aligned}
\langle \alpha, \alpha \rangle &= \overline{\alpha}\alpha && \text{Use } \overline{\alpha} \text{ for first term} \\
&= (a - bi)(a + bi) && \text{Expand both complex numbers} \\
&= a^2 - abi + bai - b^2 i^2 && \text{Multiply the terms} \\
&= a^2 + (ab - ba)i + b^2 + && \text{Gather terms, use } i^2 = -1 \\
&= a^2 + b^2
\end{aligned} \tag{19}$$

Success!

Now that we've defined the inner product of two complex numbers, let's move on to signals, or vectors, made of many such numbers.

Suppose that we have two vectors, each made of the same number of complex numbers. I'll call these vectors ρ and σ. Then the inner product finds the product of the conjugate of ρ_0 (the first element in ρ) with σ_0 (the first element in σ), then the product of the conjugate of ρ_1 (the second element in ρ) with σ_1 (the second element in σ), and so on, and then we add up all the results.

We can thus write the inner product for two complex vectors with N elements as in Equation 20.

$$\langle \rho, \sigma \rangle = \sum_{k=0}^{N-1} \overline{\rho_k} \sigma_k \tag{20}$$

The inner product also works like the dot product when checking for orthogonality. If two vectors with complex coefficients are perpendicular to one another (even in a space of more than 3 dimensions), the inner product returns 0.

In summary, to check the magnitudes and orthogonality of vectors described by *real* numbers, the dot product is the tool to use. But when the vectors are described by *complex* numbers, use the inner product. The inner product really is a generalization of the dot product, because the conjugate of a real number is that same real number. This tells us that the inner product, when given vectors of real numbers, returns the same results as the dot product would. That is, the dot product is a special case of the inner product we defined here.

Now that we have an inner product, we can use it to check the discrete complex orthogonals and see that they are orthogonal, and, as promised, that their magnitudes are *almost* 1.

10 Checking DCEs for Orthogonality

Now that we have the inner product available, let's look at two DCEs.

If the two DCEs are the same, their inner product will give us the squared magnitude of that DCE. Otherwise, the inner product will give us 0 if the two are orthogonal, and some other real number otherwise.

Let's say that our first DCE has k turns, so using the definition of μ_3 from Equation 12, sample n is $\mu_3(k, n, N) = e^{i(2\pi/N)kn} = e^{iskn}$.

The term $2\pi/N$ is going to show up a lot, and it will be in exponents, where the type is small and hard to read. So I'll create a temporary shortcut and assign $s = 2\pi/N$.

We want to compare that to another DCE. Let's say this second function has p turns. Each sample n of that DCE is given by $\mu_3(p, n, N) = e^{ispn}$. I've written the inner product of these two DCEs in Equation 21. Here I've written out the inner product as given in Equation 20.

$$\begin{aligned} \langle e^{iskn}, e^{ispn} \rangle &= \sum_{n \in [N]} e^{-iksn} e^{ispn} && \text{Conjugate the first function} \\ &= \sum_{n \in [N]} e^{is(p-k)n} && \text{Combine exponents} \end{aligned} \tag{21}$$

There are two basic forms of this result. If $p = k$, then we're finding the inner product of a single DCE with itself, so the result is the squared magnitude of that function. When $p = k$, the exponent in Equation 21 goes to zero and the exponential is 1. Let's write this out in Equation 22

$$\sum_{n \in [N]} e^{is(p-k)n} = \sum_{n \in [N]} e^0 = \sum_{n \in [N]} 1 = N \tag{22}$$

We've found that the inner product of any discrete complex exponential with itself is N, and we know that this is the squared magnitude of the DCE. It would be wonderful if that came out to be 1, but sadly, it didn't. Instead, this result tells us that the magnitude of any DCE is $\sqrt{N}$. The DCEs are not normal (but they're close enough that we'll be able to fix things up in a moment and still use them).

The other case is when $k \neq p$, so we're comparing two different DCEs. Rather than write $s(p-k)$ everywhere let's temporarily define $v = s(p-k)$ (since s, k, and p are integers, so is v). I'll pick up from the second line of Equation 21 to write Equation 23.

$$\begin{aligned} \sum_{n \in [N]} e^{is(p-k)n} &= \sum_{n \in [N]} e^{ivn} && \text{Eq. 21 with } v = s(p-k) \\ &= \sum_{n \in [N]} \cos(vn) + i \sin(vn) && \text{Expand using Euler's formula} \\ &= \sum_{n \in [N]} \cos(vn) + i \sum_{n \in [N]} \sin(vn) && \text{Break into two sums} \\ &= 0 + i0 && \text{Each sum is } 0 \\ &= 0 \end{aligned} \tag{23}$$

In the next-to-last line, I used the fact that cosine and sine are both symmetrical in terms of their positive and negative values in any interval of width 2π, so we have as many positive values as negative and the sums are 0, as illustrated in Figure 24.

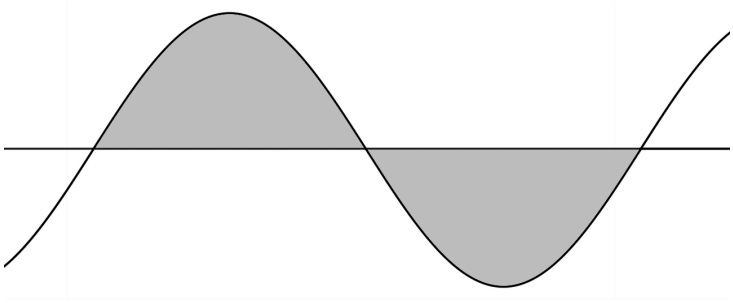

Figure 24: Given any window of width 2π, the area above 0 for any sine (or sine) is the same are the area below 0.

This tells us that *any* two different DECs are orthogonal to one another! Hooray!

To summarize, the set of DCEs are orthogonal, but each has a magnitude of $1/\sqrt{N}$. The good news is that they *all* have this magnitude, so we can make the entire set of functions normal if we scale each one by $1/\sqrt{N}$.

If you're thinking that this is the source of that $1/\sqrt{N}$ term in the defintion of the DCT, in Equation 7, you're right! We'll put that in place in the next section.

11 The DFT

We saw in the last section that the discrete complex exponentials, or DCEs, form a basis for any sequence of complex numbers. They're orthogonal to one another, which is the most important part, and while they're not normalized, they all have the same magnitude of $\sqrt{N}$.

The DFT is the name we give the change of basis of a signal from the time domain to the spectral domain, where the DCEs are the basis functions of the spectral domain. The inverse DFT is the change of basis in the opposite direction, returning us to the original signal.

Discussions of the DFT often use the letters k and n to index the signal and spectrum respectively, but some authors use these indices the other way around. This can be *super* confusing when you're consulting multiple references! In this

section, I'll use k to refer to the number of turns made by a DCE in the interval $[0, 1)$, which we also call the **frequency** of that DCE. So I'll refer to the scaling factors X_k of the spectrum using the index k, and I'll use the index n to refer to the values x_n of the signal.

Some authors use square brackets instead of subscripts, so sometimes you'll see expressions like $X[k]$ and $x[n]$ instead of X_k and x_n.

To transform x to its spectrum X, we project it onto each DCE one at a time. This is what the analysis equation of the DFT does, as shown in Equation 24.

$$X_k = \sum_{n \in [N]} x_n e^{i2\pi kn/N} \tag{24}$$

We carry this out for values of k from 0 to $N-1$. Each X_k is a complex number, formed by the product of x_n and a DCE.

For example, suppose we want to find how much of the DCE with k turns we'll need to represent our signal x, where k is an integer from 0 to $N-1$. We find this coefficient by projecting x onto that DCE, which we do by computing their inner product. To find the coefficient factor on another DCE with a different value of k, we find the inner product of *that* with x, and so on, until we've projected x onto all $N-1$ DCEs. Taken together, these N inner products gives us the sequence X, which is the DFT of x.

How do we compute the inverse, and get x back from X? Let's abstract the right side of Equation 24. I'll momentarily pretend we have a function $f_\gamma(\alpha) = \alpha e^{i\gamma}$. I'll assert that the inverse function, ${f_\gamma}^{-1}(\beta)$ is found by multiplying β by the complex conjugate of the $e^{i\gamma}$ term in $f_\gamma(\alpha)$, or ${f_\gamma}^{-1}(\beta) = \beta e^{-i\gamma}$. Then ${f_\gamma}^{-1}$ is the inverse we want, because ${f_\gamma}^{-1}(f_\gamma(\alpha)) = \alpha$, as shown in Equation 25.

$$\begin{aligned}
{f_\gamma}^{-1}(f_\gamma(\alpha)) &= {f_\gamma}^{-1}(\alpha e^{i\gamma}) && \text{Apply } f_\gamma(\alpha) \\
&= e^{-i\gamma}\alpha e^{i\gamma} && \text{Apply } {f_\gamma}^{-1}(\beta) \\
&= e^{(i-i)\gamma}\alpha && \text{Gather terms on } \gamma \\
&= e^{0}\alpha && \text{Replace } (i-i) \text{ with } 0 \\
&= \alpha && \text{Since } e^0 = 1
\end{aligned} \tag{25}$$

So this function's inverse is just multiplication by its conjugate.

How does this apply to the DFT?

The forward DFT multiplies the x entries by the DCEs given by $e^{i2\pi nk/N}$, so Equation 25 tells us that the inverse DFT recovers the inputs by multiplying the spectral values by the complex conjugate of $e^{i2\pi nkN}$, or $e^{-i2\pi nkN}$.

This gives us the inverse of the DFT analysis equations, or the **synthesis** equation, as shown in Equation 26.

$$x_n = \frac{1}{N} \sum_{k \in [N]} X_k e^{-i2\pi kn/N} \tag{26}$$

Recall from the last section that the inner product of any DCE with itself is N. So if we didn't include the $1/N$ at the start of Equation 26 (or *somewhere*, as we'll see), every starting value x_n will, after going through the forward and then backward DFT, come out as Nx_n. To get back what we put in, I scaled the synthesized values by $1/N$.

Let's put these two equations together in Equation 27.

The Asymmetrical DFT

$$\begin{aligned} X_n &= \sum_{k \in [N]} x_k e^{ikn(2\pi/N)} \\ x_k &= \frac{1}{N} \sum_{n \in [N]} X_n e^{-ikn(2\pi/N)} \end{aligned} \tag{27}$$

I call this the "asymmetrical" form because of that $1/N$ on the synthesis equation, which is missing from the analysis equation.

It doesn't really matter where the $1/N$ goes, as long as it's part of any round trip from signal to spectrum and back to signal (or spectrum to signal and back to spectrum). You could put it onto the first (analysis) equation, and leave it off of the synthesis. Doing so would give you spectral terms that were scaled by $1/N$ relative to the X_k computed by Equation 27.

As long as you're consistent, and don't need to exchange data with anyone else, you can distribute that normalizing factor any way you like. You'll see authors define the DFT using different choices for how and where to include this factor.

I find it most aesthetically pleasing to place $1/\sqrt{N}$ on both equations, so that they're symmetrical. This version of the DFT is shown in Equation 28.

The Symmetrical DFT

$$X_n = \frac{1}{\sqrt{N}} \sum_{k \in [N]} x_k e^{ikn(2\pi/N)} \quad \text{forward or analysis}$$

$$x_k = \frac{1}{\sqrt{N}} \sum_{n \in [N]} X_n e^{-ikn(2\pi/N)} \quad \text{backward or synthesis} \tag{28}$$

This is the version of the DFT I previewed back in Equation 7.

For the rest of these notes, I'll use the symmetrical version.

There's a more compact way to write Equation 28 that lets the math more clearly communicate what's going on at a glance. As usual, we'll take out some common elements and give them their own symbol. For the rest of these notes, the Greek letter ω will be defined as in Equation 29.

$$\omega \triangleq e^{i2\pi/N} \tag{29}$$

The definition of ω depends on N, so some people write it as ω_N. In the interest of making things easier to read, I'll assume that for any given use of the Fourier transform we know what N is, and thus we can leave out the N subscript and just write ω.

While something like our ω shows up in almost every discussion of the Fourier transform, some people include a negative sign in its definition [Oppenheim and Schafer, 1975]. And some authors use w rather than ω for this shorthand. I prefer the Greek letter because that's our convention for a complex number.

Using ω as defined in Equation 29, we can re-write Equation 28 more compactly as Equation 30.

$$\begin{aligned} X_n &= \frac{1}{\sqrt{N}} \sum_{k \in [N]} x_k \omega^{kn} \\ x_k &= \frac{1}{\sqrt{N}} \sum_{n \in [N]} X_n \omega^{-kn} \end{aligned} \tag{30}$$

Better!

We can make things even more compact. The explicit sums in Equation 30 are there to compute the inner product of one sequence with another. Let's write that inner product explicitly, giving us Equation 31.

$$\begin{aligned} X_n &= \frac{1}{\sqrt{N}}\langle\overline{\omega^{kn}}, x\rangle \\ x_k &= \frac{1}{\sqrt{N}}\langle\omega^{kn}, X\rangle \end{aligned} \tag{31}$$

Because the inner product conjugates its first argument, I used the conjugate of ω^{kn} in the first equation, and ω^{kn} itself in the second.

Speaking of negative signs, some people place a negative sign in the exponent of the forward equation, rather than the backward equation where I've been placing it. Couple this with the inconsistency in whether ω has a minus sign in its definition or not, and we end up with a variety of different ways to write these equations. It can get confusing.

The conventions I'm using here are common, but it always makes sense to check when you consult a new reference.

Equations 27, 28, 30, and 31 all define the Discrete Fourier Transform, or DFT. The form in Equations 30 is easy to implement and is probably the most commonly seen, but Equation 31 is more compact and, I believe, communicates the nature of the transform more clearly.

12 DFT in Matrix Form

We can write the DFT yet another way using matrices, which is often a more convenient way to carry out the operations (and is key to how the FFT works).

Here's the idea. The analysis equations we've seen each compute a term X_k by multiplying each of the input elements x_n with a variety of complex exponentials that depend on both n and k, and then adding up all of those results. We could write that by putting the complex exponentials into a row matrix, and the input elements in a column matrix, and multiplying the two.

For example, we can find X_k as in Equation 32. Each element of the row matrix tells us which complex exponential is involved.

$$X_0 = \begin{bmatrix} \omega^{0n} & \omega^{1n} & \omega^{2n} & \omega^{3n} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{32}$$

If we want to find all N results at once, we can stack up each of those rows to make an N by N square matrix, and then multiply that with the x column matrix gives us a new column matrix of the X elements.

I'll write out X_k for the four values of k in Equation 33. The normalizing term $1/\sqrt{N}$ for $N = 4$ is $1/\sqrt{4} = 1/2$. From here on in these notes, I'll use a centered dot to indicate multiplication, not the dot procuct.

$$\begin{aligned} X_0 &= \frac{1}{2}\left[x_0\omega^{0\cdot 0} + x_1\omega^{0\cdot 1} + x_2\omega^{0\cdot 2} + x_3\omega^{0\cdot 3} \right] \\ X_1 &= \frac{1}{2}\left[x_0\omega^{1\cdot 0} + x_1\omega^{1\cdot 1} + x_2\omega^{1\cdot 2} + x_3\omega^{0\cdot 3} \right] \\ X_2 &= \frac{1}{2}\left[x_0\omega^{2\cdot 0} + x_1\omega^{2\cdot 1} + x_2\omega^{2\cdot 2} + x_3\omega^{0\cdot 3} \right] \\ X_3 &= \frac{1}{2}\left[x_0\omega^{3\cdot 0} + x_1\omega^{3\cdot 1} + x_2\omega^{3\cdot 2} + x_3\omega^{0\cdot 3} \right] \end{aligned} \tag{33}$$

These four expressions are Equation 30 with the loops written out explicitly. We can write this as a matrix W_4 as in Equation 34. So that we don't need to write that $1/2$ term over and over, I'll include that in the definition of W_4.

$$W_4 = \frac{1}{2}\begin{bmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \omega^{0\cdot 2} & \omega^{0\cdot 3} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \omega^{1\cdot 2} & \omega^{1\cdot 3} \\ \omega^{2\cdot 0} & \omega^{2\cdot 1} & \omega^{2\cdot 2} & \omega^{2\cdot 3} \\ \omega^{3\cdot 0} & \omega^{3\cdot 1} & \omega^{3\cdot 2} & \omega^{3\cdot 3} \end{bmatrix} \tag{34}$$

Now we can rewrite all of Equations 33 or 34 super compactly as Equation 35.

$$X = W_4 x \tag{35}$$

In general, the matrix W for any N is given by Equation 36. Since I'm assuming we know N, I'll usually leave out the subscript from now on to reduce clutter, writing just W.

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \left(\omega^2\right)^2 & \left(\omega^2\right)^3 & \cdots & \left(\omega^2\right)^{N-1} \\ 1 & \omega^3 & \left(\omega^3\right)^2 & \left(\omega^3\right)^3 & \cdots & \left(\omega^3\right)^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \omega^{N-1} & \left(\omega^{N-1}\right)^2 & \left(\omega^{N-1}\right)^3 & \cdots & \left(\omega^{N-1}\right)^{N-1} \end{bmatrix} \tag{36}$$

With this, we can write the DFT in its most compact form yet in Equation 37.

$$\begin{aligned} X_n &= Wx \\ x_k &= \overline{W}X \end{aligned} \tag{37}$$

This is a great time to celebrate!

Equations 36, and 37, along with the definition of ω in Equation 29, are the top of the mountain of the classical Discrete Fourier Transform.

Let's see the DFT in action for a couple of examples.

12.1 The DFT for $n = 1$

The smallest signal has only 1 element, so $N = 2^1 = 2$. Rather than our usual α and β, I'll write the input as two complex numbers, x_0 and x_1.

Let's write the matrix W for $N = 1$. From Equation 36 we get Equation 38.

$$W = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & \omega \end{bmatrix} = \vee \begin{bmatrix} 1 & 1 \\ 1 & e^{i2\pi/2} \end{bmatrix} = \vee \begin{bmatrix} 1 & 1 \\ 1 & e^{i\pi} \end{bmatrix} = \vee \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{38}$$

Here I'm using my own shortcut that will prove useful throughout the rest of these notes. The symbol $\vee$ is defined as in Equation 39.

$$\vee = \frac{1}{\sqrt{2}} \tag{39}$$

Now that we have W, let's use it to calculate the DFT. I'll start with the forward DFT in Equation 40.

$$\vee \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \vee \begin{bmatrix} x_0 + x_1 \\ x_0 - x_1 \end{bmatrix} = \vee \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \tag{40}$$

Now that we have the spectrum of our input signal, let's turn it back into a signal, as shown in Equation 41.

$$\vee \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \vee \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \vee^2 \begin{bmatrix} X_0 + X_1 \\ X_0 - X_1 \end{bmatrix} = \vee^2 \begin{bmatrix} (x_0 + x_1) + (x_0 - x_1) \\ (x_0 + x_1) - (x_0 - x_1) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 2x_0 \\ 2x_1 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{41}$$

And we're back!

Let's work this out for a bigger signal.

12.2 The DFT for $n = 2$

Let's go one step bigger and consider $n = 2$, so $N = 2^n = 4$. I'll write the matrix W for this case following Equation 36 again, giving us Equation 42.

$$W = \frac{1}{\sqrt{4}} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \tag{42}$$

The matrix W in Equation 42 is symmetrical. This makes sense, since each exponent is kn for both k and n running from 0 to 3. In fact, W is symmetrical for every value of N.

We can put numbers to the factors in W by writing out each element.

Because adding any integer multiple of 2π to the exponent of ω has no effect on its value, we can see that the four outputs repeat in a cycle of length 4, as shown in Table 1.

$$\begin{array}{llll} \omega^0 = \omega^4 = \omega^8 & = e^{i2\pi 0/4} & = e^{i0} & = 1 \\ \omega^1 = \omega^4 = \omega^8 & = e^{i2\pi 1/4} & = e^{i\pi/2} & = i \\ \omega^2 = \omega^4 = \omega^8 & = e^{i2\pi 2/4} & = e^{i\pi} & = -1 \\ \omega^3 = \omega^4 = \omega^8 & = e^{i2\pi 3/4} & = e^{i\pi 3/2} & = -i \end{array}$$

Table 1: Factors of ω repeat with a period of 4.

Plugging these values into W from Equation 42 gives us Equation 43.

$$W = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \tag{43}$$

The top row and left column of this W, as for W of any size, is always 1 because in the top row, $k = 0$, and in the left column, $n = 0$. Let's now multiply our input signal with this matrix, giving us Equation 44.

$$X = Wx = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} x_0 + x_1 + x_2 + x_3 \\ x_0 + ix_1 - x_2 - ix_3 \\ x_0 - x_1 + x_2 - x_3 \\ x_0 - ix_1 - x_2 + ix_3 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} \tag{44}$$

We've just computed the DFT of any 4-element signal x! The four elements on the right of Equation 44 are the coefficients on the four DCEs of 0 through 3 turns.

We know how to turn this back into a signal: we just need to apply the complex conjugate of W to X. To get $\overline{W}$, we just conjugate each element by replacing each exponent in Equation 42 with its negative, as in Equation 45.

$$\overline{W} = \frac{1}{2}\begin{bmatrix} \omega^{-0} & \omega^{-0} & \omega^{-0} & \omega^{-0} \\ \omega^{-1} & \omega^{-0} & \omega^{-2} & \omega^{-3} \\ \omega^{-2} & \omega^{-0} & \omega^{-4} & \omega^{-6} \\ \omega^{-3} & \omega^{-0} & \omega^{-6} & \omega^{-9} \end{bmatrix} \tag{45}$$

We can remove the negative sign from any ω^{-j} in this $\overline{W}$ by adding $N = 4$ to the exponent repeatedly until it becomes positive. For example, $\omega^{-3} = \omega^{(-3)+4} = \omega^1$. Putting these values into Equation 45 gives us Equation 46.

$$\overline{W} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \tag{46}$$

Now we can apply this to X. I'll write the starting setup and the conclusion in Equation 47. I won't write out all the algebra because it's complicated and doesn't illuminate anything. If you grind through all the operations (or better yet, use a

symbolic math package to handle the details for you), you'll see that we indeed get back our original x.

$$\begin{aligned} x = \overline{W}X &= \left(\frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \right) \left(\frac{1}{2} \begin{bmatrix} x_0 + x_1 + x_2 + x_3 \\ x_0 + ix_1 - x_2 - ix_3 \\ x_0 - x_1 + x_2 - x_3 \\ x_0 - ix_1 - x_2 + ix_3 \end{bmatrix} \right) \\ &= \frac{1}{4} \begin{bmatrix} 4x_0 \\ 4x_1 \\ 4x_2 \\ 4x_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \end{aligned} \tag{47}$$

And we're back to x!

13 Anti-Aliasing

And now, the section you've probably been waiting for! We'll apply all of this to computer graphics. Specifically, we'll look at rendering.

We can use the Fourier transform to understand and control the undesired effects of *aliasing*. Examples of aliasing artifacts include the harsh, stair-step edges we sometimes see on object boundaries (the "jaggies"), or textures that seem to break up, or wagon wheels that rotate faster and faster in an animation as the vehicle picks up speed, and then seemingly slow down and start rotating in reverse!

These aliasing phenomena are all a result of errors in the generation of images and animations. These errors are best explained with the Fourier transform, which also gives us guidance on how to reduce or eliminate them.

For this discussion, I'm going to simplify and abstract everything, so we can see all the ideas in their clearest form. In the real world, there are complications galore due to numerical stability, issues with floating-point numbers, reducing computation costs by making approximations, and so on.

I'm also going to assume that we're rendering using an old-school regular grid of samples. This was the dominant approach in the early decades of computer graphics, when we usually created images with scan-line techniques. Today, with the advent of ray tracing, we're more likely to use irregular sampling patterns (also called *non-uniform* patterns). The reason for this is also explained by the Fourier transform! The theory of this approach builds on the basic ideas we'll discuss next. Here, I'll focus on those ideas, and leave the more extended discussion of non-uniform sampling for another time.

13.1 The Big Picture

I'll set the stage with the big picture that will guide our discussion. The idea is shown in Figure 25.

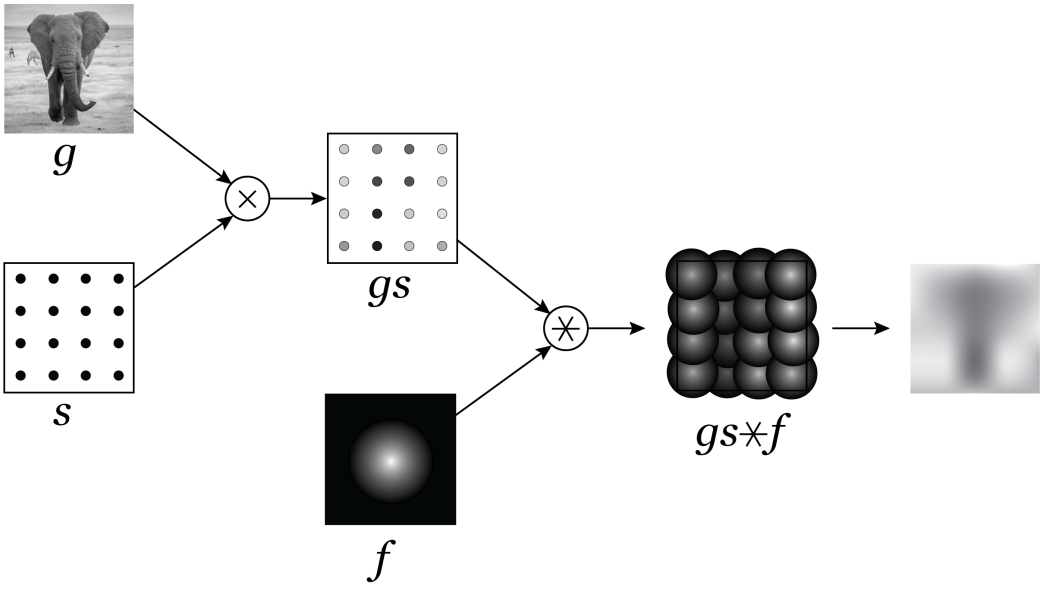

Figure 25: Rendering in one figure. We start with a continuous signal g (that we don't know), and evaluate point samples described by s. This gives us the pixels gs of our rendered image. To show the image, we reconstruct the continuous signal by convolving the pixels with a reconstruction filter f, one copy of which is placed on top of each sample, and scaled by that sample's value. The sum of all these overlapping filters is the image $gs * f$ that we show to a viewer.

The starting point is the 2D image in the upper left, which I've called g. This is a theoretical or abstract image. It comes from our mental conception of rendering: there are objects "out there," and lights, and cameras (all of these elements usually have lots of additional data associated with them). The central goal of the rendering process is to create of an image which matches what our camera would have recorded if all of the elements of our scene were real, and we took a picture.

To keep things simple, I'll assume we're only interested in making a grayscale image. If we want color, then we generally just perform of the operations below multiple times, once for each wavelength (or range of wavelengths) we want to capture.

I'll also focus just on the magnitudes of the Fourier coefficients, and ignore the phase information. That's a common practice for making figures, but don't forget that the Fourier coefficients are complex numbers, and they have a phase as well as the magnitude I'll be showing.

This idealized starting image g can be described by a *continuous* 2D function. That is, for any two real numbers x and y, the function $g(x, y)$ tells us the color of the image at that location on the imagined picture plane. Because x and y are real numbers, we can query g anywhere and get back the exact color that would be associated with the camera's image at that point. In a real imaging system, the image always has some finite limit to its resolution, due to film grain, sensor density, or other real-world issues. Let's pretend that these aren't present, and the camera could in fact record the continuous image g.

Our goal is to show g to our viewers. But there's a big problem: we don't know g! So we create a bunch of machinery collectively known as *rendering*. Although there are many ways to approach this, as I mentioned earlier I'll imagine that we query g at a specific set of locations (x, y) that sit on a *regular grid*, or *rectangular lattice*, which I've labeled as s. These locations are often imagined to be the centers of a set of rectangles (often squares) that tile the plane of the camera's "film," fitting together without gaps or overlaps.

Mathematically, we represent s as a collection of infinitely tall, infinitely thin spikes. These are called *delta functions* and are often represented by the Greek letter δ. So s is a sum of many of these delta functions, one at each point in the center of our grid of rectangles.

Now we can multiply together g and s. Their product, gs, is the *sampled* version of g. It is a *discrete*, or *digitized* representation of g. The set of samples represented by gs are the result of rendering using the grid s. These samples are the things we call *pixels* (note that they are point samples, and *not* "little squares" [Smith, 1995]).

Conceptually, this process gives us a list of 2D locations that form a regular grid, and a value of g (usually a color) at each of those locations. We don't have any idea what the colors of g might be anywhere except at those sample locations. Of course, we hope that they will be a lot like the samples we've taken, but the artifacts I listed above are results of when that hope is thwarted.

Most people don't want to view an "image" composed of a grid of isolated, infinitely small colored points!

So the next step is to turn those points, representing a discrete or digitized set of samples, back into the continuous image g. Then we'd have the equivalent of the continuous g we imagined was recorded by the camera.

To turn the samples into a continuous image, we perform *reconstruction*, an operation complementary to sampling. Just as sampling involved multiplying the signal g with the sampling grid s, reconstruction involves combining the digitized image gs with a continuous *reconstruction filter* (often just called a *filter*), shown in the figure as f.

This step is not performed by multiplication. Instead, we perform *convolution* [Azad, 2023]. There's much to be said about convolution, but for this dicussion, it involves placing a copy of the reconstruction filter f over each sample location in gs (more formally, we'd first reflect the filter f so that it's used "backwards," but I'll ignore this detail here). The height of the filter at each sample is given by the height of the sample it's being centered over (remember, we're working with grayscale). We write convolution with the asterisk, so the convolution of any with the filter f is written $gs * f$.

Figure 26 shows a visualization of convolving a little grid of samples with a filter f consisting of a Gaussian bump. The final result is the sum of all the bumps.

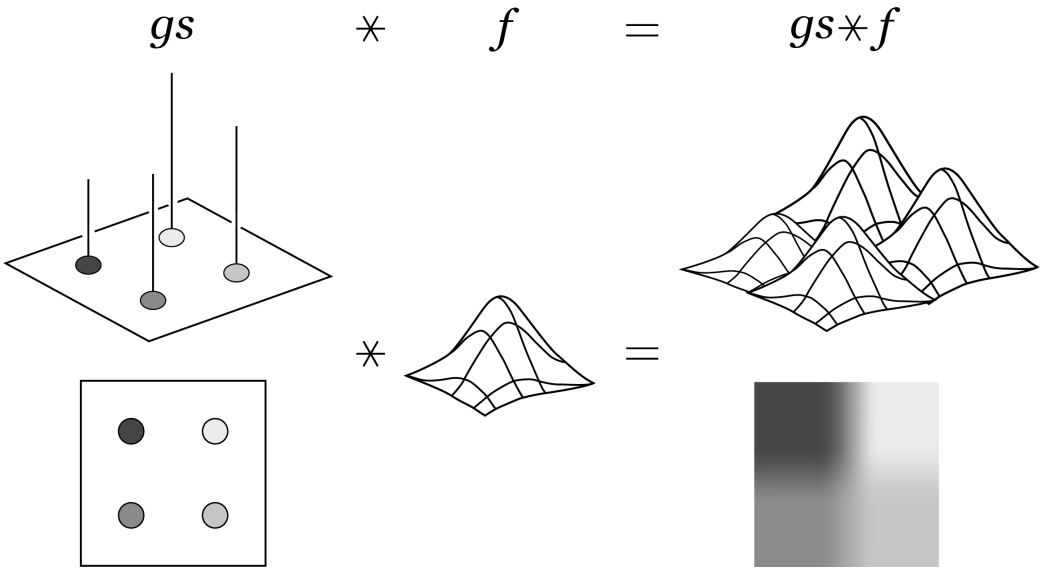

Figure 26: To reconstruct an image from pixels, we place a copy of the filter f over each pixel, scaled by that pixel's value. The sum of all the filters is our reconstructed, continuous image $gs * f$.

If we sum up all of these reconstruction filters (one at each sample), which are themselves continuous, we get back a new, continuous result. Just as gs repre-

sents multiplying g with all the samples of s, so too $(gs) * f$ represents the sum of convolving each sample of gs with the filter f and then summing everything together. Thus our final result is $(gs) * f$, or more usually, $gs * f$.

In an ideal world, $gs * f = g$. That is, our reconstructed image will exactly match the image we (theoretically) started with. That's the goal of photorealistic rendering.

Since we don't know g, it's generally hard to say how good this match is. But if something has gone wrong in this process then there will be obvious artifacts in $gs * f$ that almost surely don't belong, like jaggies and broken textures.

Let's see where things usually go wrong. Then we'll have some chance of preventing the problems.

From now on, I'm going to use one-dimensional versions of everything. This is reasonable for graphics, because the Fourier transform has the property of being *separable* [Dudgeon and Mersereau, 1984]. That means we can do all of the operations above on the rows of an image, and then the columns, and that will give us the same result as if we worked directly with a 2D image.

In the discussion above, we saw signals get multiplied and convolved. The Fourier transform shows us that there is a beautiful, elegant relationship between these two operations. I won't prove this relationship here, as it would be too big a detour, but you can find clear derivations of this property in any reference on digital signal processing, either online or in textbooks like [Gabel and Roberts, 1980] and [Oppenheim and Schafer, 1975]. The property is this: if we multiply two signals in the time domain, that is equivalent to convolving their Fourier transforms. It goes the other way, too: if we convolve two signals in the time domain, that is equivalent to multiplying their Fourier transforms. This is called a *duality property* of the Fourier transform, and is one of the reasons why working with frequency representations of signals is such a joy.

In symbols, we can write Equation 48 for two signals a and b, and their Fourier transforms A and B:

$$\begin{aligned} ab &\leftrightarrow A * B \\ a * b &\leftrightarrow AB \end{aligned} \tag{48}$$

For clarity, from now on I'll draw all signals and spectra with simple shapes. That is, the spectra won't actually be the Fourier transform of their nearby signals, and vice-versa. This is a common technique that lets us focus on the ideas without being distracted by lots of visual detail.

Armed with this correspondence between multiplication and convolution, let's

look again at the sampling step from Figure 25, this time using a 1D signal (you can think of this as a single row, or column, from an image). I've shown this in Figure 27.

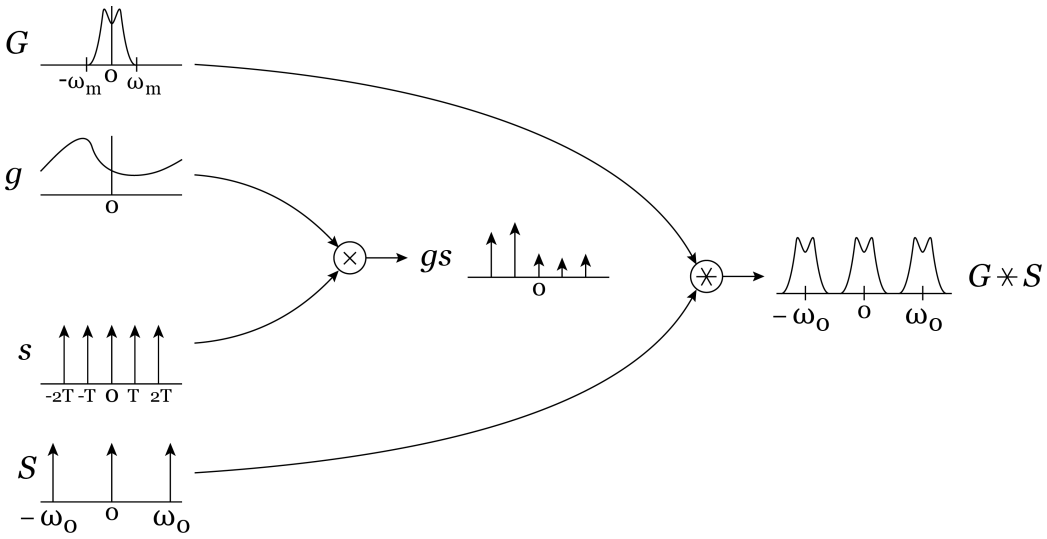

Figure 27: The sampling step of Figure 25, in both signal and frequency spaces. The spectra and signals are suggestive, and not numerically accurate Fourier pairs.

Let's look at the new elements in Figure 27.

The smallest change is that g is now a 1D curve, and I've marked an origin 0 on the horizontal axis, so we can keep track of its alignment as we process it.

Below that is the 1D sampling grid s. I've drawn this as a grid of equally-spaced arrows representing the sample positions. These arrows represent the *delta functions* I mentioned earlier. For our purposes, we can think of each arrow as an infinitely thin spike. Formally, the height of the spike is undefined [Gabel and Roberts, 1980](pg 130). An informal convention is to draw each arrow with a specific, finite height showing the magnitude of its related sample (this fudges the math, but not too much). In this figure, every arrow has a height of 1.

Finally, I've written the horizontal location of each delta function as a multiple of some interval T. This is the *sampling period*, usually expressed in seconds (because our signals are assumed to be describing a function over time). Because

we're associating each delta function with one pixel in an image, we can think of T as the spacing between pixels.

Multiplying g and s gives us gs, shown to the right of the signals. Each arrow representing a delta function is now scaled by the value of g at its location.

Now let's look at the Fourier transform of each of these signals. Above g, I've drawn its spectrum G (as always, this is a simplified and idealized kind of a spectrum, and isn't actually related to the arbitrary squiggle I drew for g).

There are a few important things going on with G. First, we know where it's centered, since I marked that with 0. More importantly, I've drawn G so that it's placed to have equal extents left and right of the origin. The highest frequencies are marked as $-\omega_m$ and ω_m (think of m as "maximum").

The last thing to note is extremely important, and vital to our discussion: *All frequencies outside the range* $[-\omega_m, \omega_m]$ *are* 0. This isn't just to make the picture easier to draw: this is a fundamental property of G that I'm asserting.

This is an audacious thing to do. We don't know g, so who's to say what its Fourier transform G would look like? Where do we get off saying that there's some maximum, or *cutoff*, frequency ω_m, and everything beyond that is 0?

Well, we don't *have* to say that. But we'll see later that we *want* to say that. Our sampling and reconstruction process (that is, rendering and showing an image) is only going to work properly if this condition is met. We'll see exactly why later, so I'll ask you to just take this as a given for now and roll with it.

Now let's look at the Fourier transform of the delta functions s. A set of equally-spaced delta functions is sometimes called a *shah function*. The Fourier transform of a shah function is another shah function. The spacing is different, though.

The spikes are separated by ω_0. Thie relationship between the distance T between the spikes of s in signal space, and the distance ω_0 between the spikes of its Fourier transform S in frequency space, is given by Equation 49 (as usual, I won't prove this, but it's just a result of grinding through the algebra, and you can find the steps in almost any digital signal processing reference).

$$\omega_0 = \frac{2\pi}{T} \tag{49}$$

This reciprocal relationship between ω_0 and T means that as the samples in s get closer together (and T decreases), the spacing ω_0 increases, and the spikes in S get further apart. The opposite is also true: as the samples in s spread out, the spikes in S get closer together.

Lock this reciprocal relationship in your mind, because it is the key to understanding both aliasing and anti-aliasing. In words, increasing the sampling density in the sampling grid s pushes the samples farther apart in its spectrum S, and vice-versa.

Finally, we'll combine G, the Fourier transform of the signal, with S, the Fourier transform of the sampling spikes. Because we multiply those functions in signal space to make gs, the corresponding operation in frequency space is to convolve the spectra to make $G * S$, shown on the right of Figure 27.

The result of forming $G * S$ is that one copy of G is centered at 0, and other copies of G are centered at multiples of ω_0. Because the spikes of S all have the same magnitude, all of these copies of G will be identical.

Notice what's happened: a copy of G has been placed on top of every spike, so we now have a *repeating pattern* of copies of G.

In other words, the Fourier transform of our sampled image gs is $G * S$, containing many copies of G.

Now we have our pixels – that is, a sampled signal. It's not what we started with! The original spectrum G had no frequencies outside the range $[-\omega_m, \omega_m]$. But $G * S$, the spectrum of gs, has *plenty* of energy in the frequencies above and below those limits. Just tons and tons of energy at these higher and higher frequencies. This is a faithful representation of what gs is telling us. One reason for these many high frequencies is that gs is not *smooth*. It's just a bunch of spikes! So somehow the signal gs needs to get from being 0 for a while to suddenly leaping up to the height of a spike, and then it has to plummet back down to 0 and stay there, before leaping back up again, over and over. Those are *fast* changes. And fast changes come about from sine and cosine components that are wiggling fast enough to match that abrupt change. That's why $G * S$ has all those high frequencies: they represent the need of gs to leap from stretches of 0 to a sudden non-zero value, and then back down again.

Generally, any sudden or quick changes in a signal (like the change in color at a polygon's edge, or a tiny highlight) require high frequencies in the signal's spectrum. So we call these *high-frequency* phenomena. Slow, gradual changes (like the smooth change in shading over a sphere) only need low frequencies.

What happens when we reconstruct $G * S$? That's the right side of Figure 25, now expanded with Fourier transforms in Figure 28.

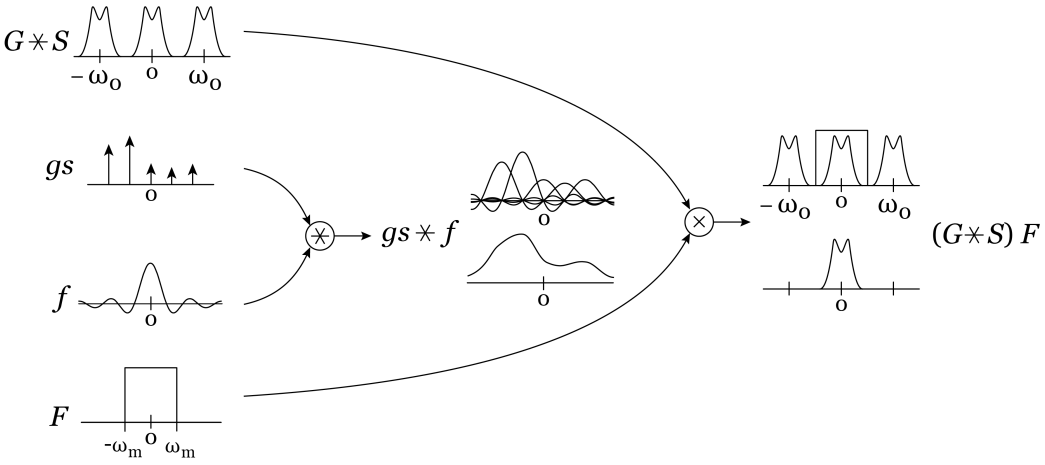

Figure 28: The reconstruction step from Figure 25, with spectra.

In the upper left we have $G * S$ again from Figure 27. Below that is its signal gs, our reconstruction filter f, and its Fourier transform, F.

This time, we'll start with the spectra, because we want to think about what we want that filter to do. If we're going to do something with $G * S$ to get back g, we need to get rid of every copy of G except for the one centered at 0, since that's the Fourier transform of g. Then our lone remaining copy of G describes g, the imaginary source image we sampled with our rendering software.

So how can we knock out those extra copies of G? Well, how is our filter going to get combined with $G * S$? We know that the reconstruction step convolves gs and f to form $gs * f$. Equation 48 tells us that if the signals are convolved, the equivalent operation on the Fourier transforms is that they get multiplied. So if can multiply all the values of $G * S$ in the copy centered at 0 by 1, and all the other copies by 0, we'll have G again. So let's do that.

In the bottom left of the figure I've drawn F, the Fourier transform of the f we want. This is a *box filter*. It's just 1 from $-\omega_m$ to ω_m, and 0 everywhere else. Multiplying F with $G * S$ gives us $(G * S)F = G$, as shown at the right.

Great!

We've got G, and thus g!

If we happen to have $G * S$ lying around, we can just do the multiplication. But more usually, we haven't explicitly computed that. Instead, we just have gs, our sampled image. So we can take f, the Fourier transform of the box filter F,

and convolve that with gs.

I won't prove it (again, it's a standard thing in any textbook), but the Fourier transform of a box is a signal called the *sinc* (pronounced "sink"). The sinc is defined in Equation 50, where we agree that at 0 the function has a value of 1. A plot of sinc is shown in Figure 29.

$$\text{sinc}(x) = \frac{\sin\ x}{x}, \qquad \text{sinc}(0) = 1 \tag{50}$$

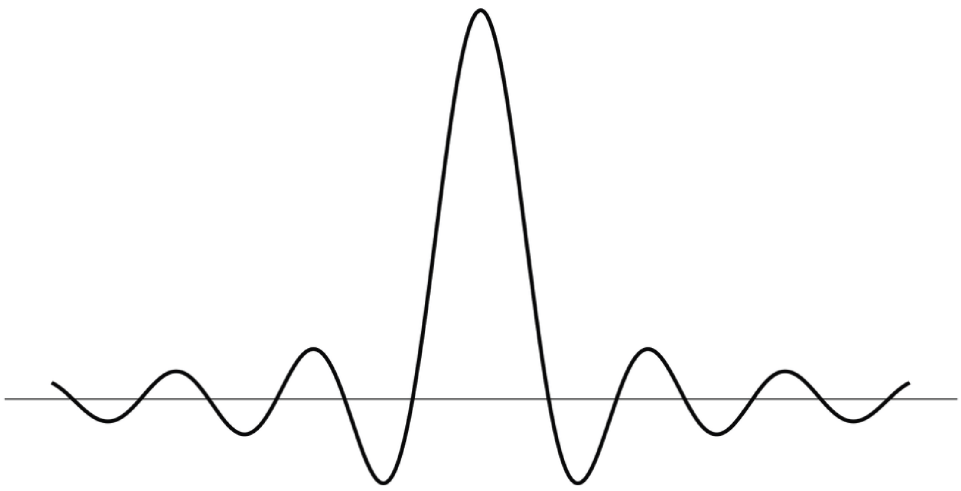

Figure 29: The sinc function.

The sinc that we get from the box filter in Figure 28 will have a value of 1 in the center, and it will be 0 at multiples of T, as shown in Figure 30.

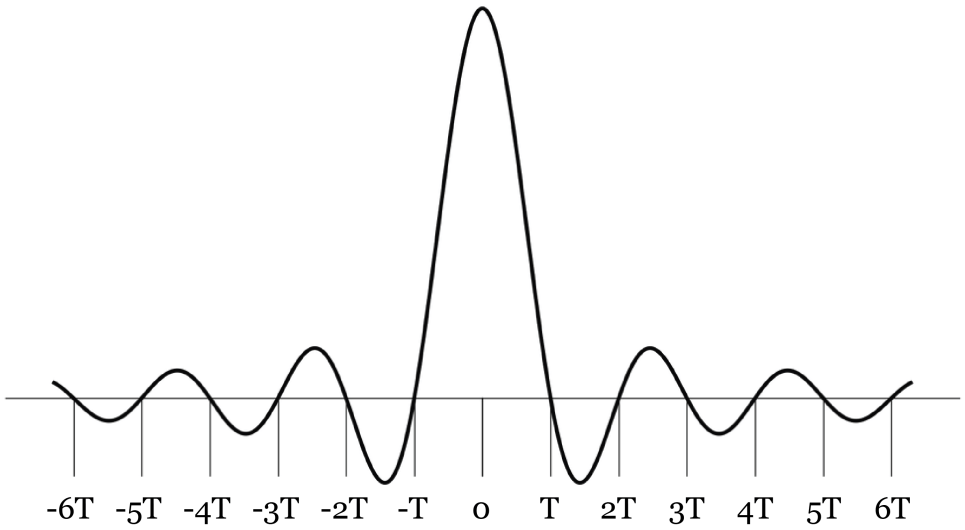

Figure 30: The sinc function is zero at every positive and negative multiple of T except $0T$, where it has a height given by the value of the sample there.

That means that when we convolve gs with f, there will be a sinc with a central value of 1 at each spike of gs, and 0 at every remaining spike of gs,

Let's reconstruct from the samples in Figure 31.

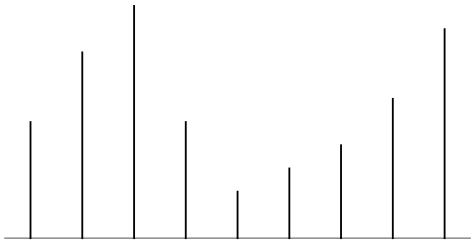

Figure 31: A set of samples for reconstruction.

Placing a sinc at each of these samples, scaled vertically by the height of the sample, gives us Figure 32.

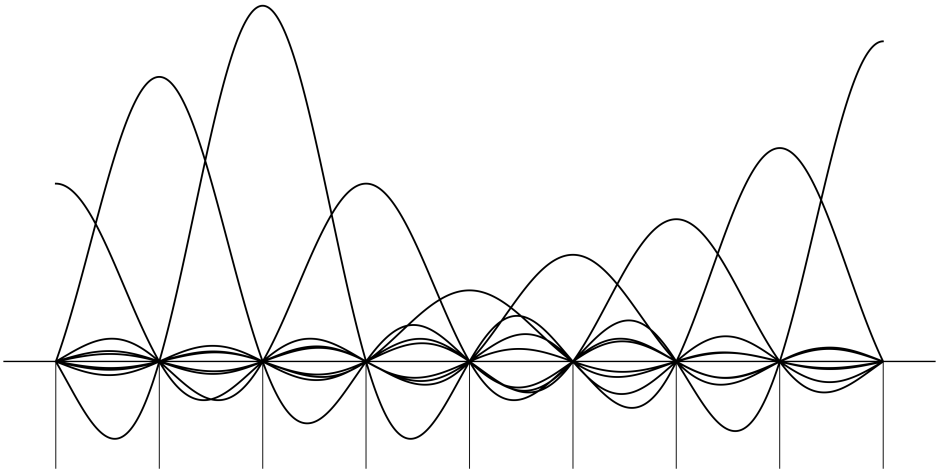

Figure 32: Reconstruction of the samples from Figure 31 using sinc functions. Note that each sinc is 0 at every sample location except the one that it's centered over. The sample locations are marked with vertical lines.

The continuous reconstruction from these sincs is shown in Figure 33. The curve passes through the top of the sinc over each sample. In between samples, it's influenced by the positive and negative values contributed by the other sinc functions.

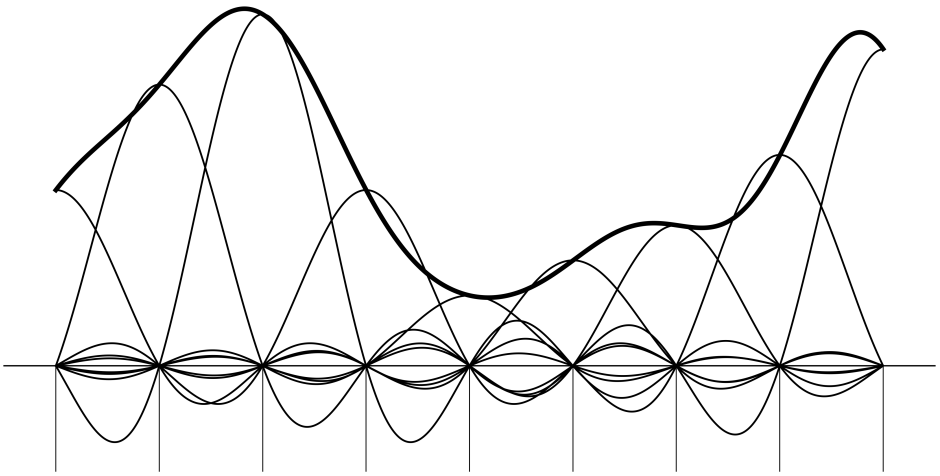

Figure 33: The sinc functions of Figure 32 with their continuous sum (the thick curve).

Note that the sincs are not 0 in between the spikes! And *that* is how we get back our continuous image g. All of those sinc functions add up in the spaces between the samples.

The beautiful part of all of this is that because the Fourier transform of this process, $gs*f$, is G (because we applied the box filter F to $G*S$), we're *guaranteed* that the sinc functions add up in *exactly* the right way to reconstruct g!

Total success!

In short, by sampling our imaginary signal g, we've gathered enough information to fill in between the samples to get the continuous g back again.

That's realistic rendering done right!

But hold on. Surely g could be doing all kinds of stuff between the samples we took. There could be entire objects hiding between the samples, or holes in objects, or bright highlights on objects, or a million other small features that are important, but fall between the samples we took. What about that stuff?

That brings us back to when I asked you to just roll with my limits on G. By saying that all frequencies of G beyond the limits $-\omega_0$ and ω_0 were 0, I was asserting that nothing in g was changing faster than our samples placed T units apart would be able to capture.

We say that such a function is *band-limited*. It simply cannot change too fast.

In fact, in the drawings we've seen, it couldn't change faster than we were able to detect with our samples.

But surely the interval T can't be arbitrary. If so, we could take just one sample for the whole picture! Well, that would work if the picture g had no energy in any frequencies except a spike at 0. That is, the whole picture was a single constant color. Then one sample would be enough.

But suppose that s was a gradient, black on the left and white on the right. How many samples would we need? Suppose that g had a million tiny balls, each of which had little colored dots inside. Suppose that g was an ocean, with thousands of tiny but extremely bright glints? Or suppose that g was a polygon with sharp edges that appear at an angle. What should T be? Intuitively, we'd say that T has to get pretty small so that our samples will be close enough to one another to catch these details. But how small? And what happens if it's not small enough?

Spoiler: if T is not small enough, we get *aliasing*. Stair-step edges. Throbbing textures. Weird halos around objects.

Let's see why, and what we can do about it.

Before we jump in, I want to mention that we rarely perform the reconstruction step in software. That's usually handled by the output display. When a CRT, LED, LCD, or any other type of electronic display or printer presents an image, it necessarily spreads out the point samples (that is, the pixels) over some small area. This is in essence the hardware applying an unknown reconstruction filter to the point samples. Most people don't try to model this filtering. Usually we just assume it will be something like the middle lobe of a sinc, or maybe some form of Gaussian, spreading out a color value in some region around the point sample that is our pixel. Today's hardware makes this usually a pretty good bet. So while our theory is perfectly sound, and there is a reconstruction filter being applied to our sampled signal, that filter is usually implemented by the specific hardware (and maybe software) in our display device, not in our graphics code. The essential thing is that some kind of reconstruction filter is being applied, because otherwise we'd be trying to show a grid of infinitely small points samples of color.

Now back to aliasing!

The key insight to understanding what spacing we want in the grid g is in the relationship between the sampling rate T and the largest non-zero frequency ω_m in the spectrum of g. We saw this relationship above, but I'll repeat it here as Equation 51.

$$\omega_0 = 2\pi/T \tag{51}$$

Recall that ω_0 is the distance between the centers of copies of G that result from sampling at intervals T. I said that G was band-limited, so it's 0 beyond the limits $-\omega_m$ and ω_m. We can expect problems if the copies of G aren't spread apart far enough. If they start to overlap (and thus add together), we'll be in trouble, because then we'll have no way to recover G any more.

Let's see this.

In Figure 34(a) I've drawn a new spectrum G, which I've kept super simple: it's just a triangle from $-\omega_m$ to ω_m. To prevent overlaps, the distance ω_0 between centers had better be at least $2\omega_m$.

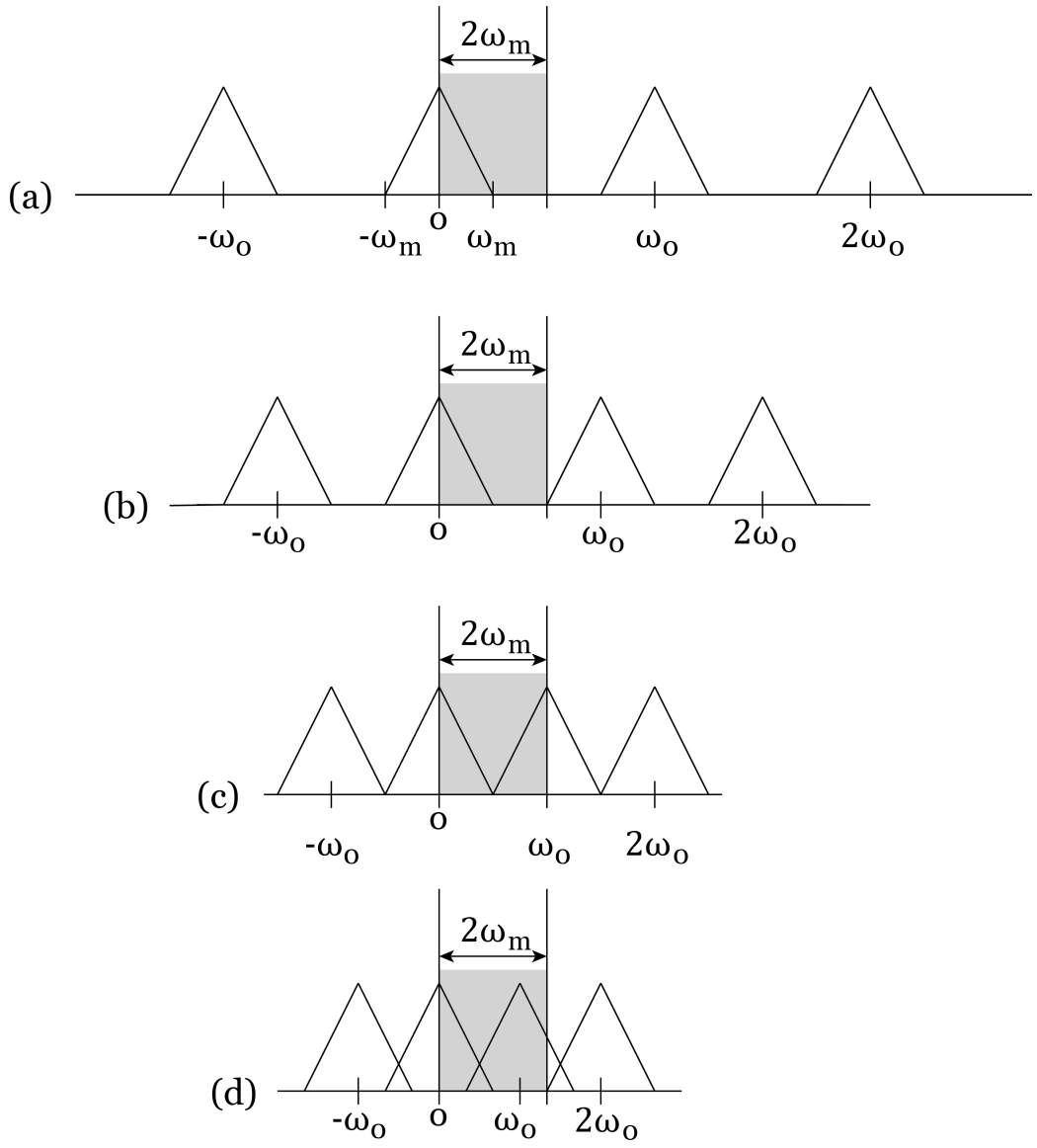

Figure 34: Reducing the distance ω_0 between copies of G in $G * S$. In (a) and (b), we have $\omega_0 > 2\omega_m$. All is well. In (c), we're at the Nyquist rate $\omega_0 = 2\omega_m$. In (d), $\omega_0 < 2\omega_m$, and the spectra are overlapping. This will cause artifacts in the reconstructed signal.

In the figure, from top to bottom I've shown what happens as the sampling distance T increases (and thus the spacing ω_0 between spectral copies decreases). In parts (a) and (b), T is small enough that $\omega_0 > 2\omega_m$. Suppose that T increases enough so that $\omega_0 = 2\omega_m$. Then we get Figure 34(c), where the copies of the spectrum just touch one another. This is the closest they can get for us to still extract one clean version of G (in practical terms, this is too close, as it requires a perfect filter f, but let's pretend perfect filters exist).

Now let's try to save some rendering time, and use fewer samples, so they're further apart and T increases. This causes the spectra to move a little closer together. Now the distance ω_0 between copies of G is less than $2\omega_m$, the width of G, and the copies overlap, as in Figure 34(d).

Now we're in trouble. There's nothing we can do in general to the spectrum of Figure 34(d) that would let us recover G (we could design special operations for a specific G, but remember that we don't know g, and thus we don't know G).

Let's look close up at the copies of G on both sides of the one at the center, shown in Figure 35.

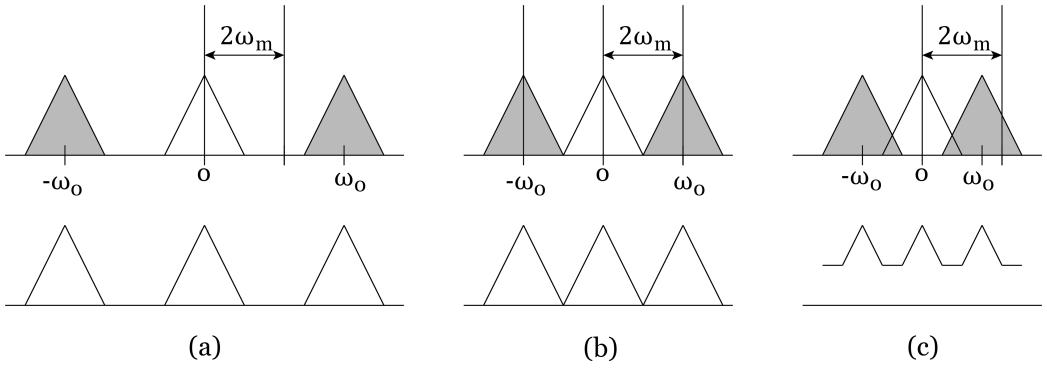

Figure 35: A close up of the spectra of T increases and ω_0 decreases. Below each plot I've shown the sum of the spectra. In (c), we have aliasing.

When ω_0 is less than $2\omega_m$, as in Figure 35(c), the lower frequencies in the rightmost copy of G now appear in the same location as the high frequencies of the center copy. From the point of view of G, these lower frequencies now *appear* to be high frequencies! The low frequencies in the copy have taken on the identity of the high frequencies in the original. That is, they've assumed a new *alias*, or identity, as low frequencies. In the same way, the high frequencies in the copy to

the left are now overlapping the central copy, so those high frequencies are now masquerading as low frequencies in our central copy of G.

This is aliasing.

When we apply our reconstruction filter F to this spectrum, as in Figure 36(a), we get the spectrum in Figure 36(b), which is definitely not our original triangle, G. Convolving gs with this spectrum will definitely not give us back g.

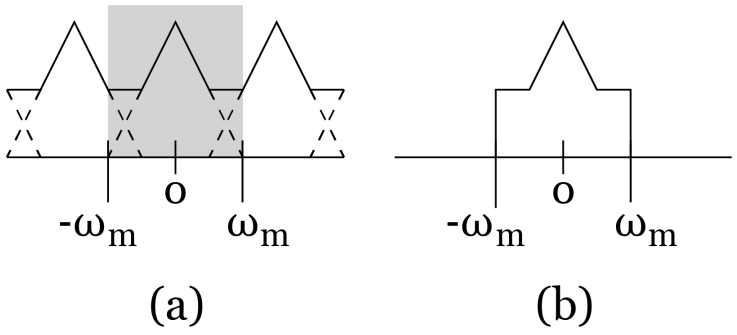

Figure 36: When our spectra overlap, filtering out the central copy in (a) gives us a result in (b) that's distorted by the presence of the overlapping copies.

The problems that arise from these overlapping spectra are the ones we've been discussing here. They're all the result of aliasing: low frequencies from copies of G masquerading, or aliasing, as high frequencies, and high frequencies masquerading as low frequencies. When these aliased frequencies contribute to the spectrum we extract using our filter, as in Figure 36(b), they give rise to all the problems we've been discussing, and many more (the backwards-spinning wagon wheels are caused by thinking of this process in terms of time. Then the distance T between frames is too big relative to the frequencies of motion from one frame to the next).

The value of T where the copies of G just touch is called the *Nyquist rate*. I've written its definition in Equqtion 52.

$$T = \frac{1}{2\omega_m} \tag{52}$$

In practice, we usually want to be sure that T is smaller than its theoretical maximum, so a more typical way to write this is in Equqtion 53.

$$T \leq \frac{1}{2\omega_m} \tag{53}$$

In words, the distance T between our signal-space samples must be less than 1 over twice the maximum frequency in the image we're sampling.

This tells us that we have two ways to prevent aliasing. Both of them require us to estimate ω_m, which unfortunately we rarely know.

Let's say we have a guess for ω_m. Then we can choose the sampling period to be $T = 1/(2\omega_m)$ (for practical applications, because filters aren't perfect and our guess could be wrong, we'd usually like T to be somewhat smaller).

Then we can try taking samples at this interval T and see what happens. If the picture isn't good, we make T smaller (perhaps by taking new samples in-between the old ones), and reconstruct again.

This *adaptive sampling* strategy is common: we start out with samples that are spaced closely enough that we hope we'll get the information we need. If the picture doesn't look right to us (or to a program that "looks" for problems on our behalf), we take more samples, thereby reducing T and spreading out the copies of G, ultimately to a spacing where there is little or no aliasing.

Alternatively, we can try to limit the frequencies in the imaginary picture g that we're sampling. For example, if g is a texture from a photograph that we have available, we could filter the photograph before rendering to set any frequencies that might be present above ω_m to 0.

This is sometimes called *pre-filtering*. Doing this for more complicated 3D scenes is a challenge.

What we've seen is that aliasing phenomena of all kinds are due to sampling a signal with too big an interval between samples. When we sample with a rectangular grid, this means that the resolution of the sampling grid is too low for the environment we're sampling. The result is that the copies of the spectra of the image, unavoidably produced in the spectrum of the the sampled image, will overlap. High and low frequencies in the new copies will appear as low and high frequencies when we filter the spectra, and then manifest themselves in the image. This is aliasing. Usually, those phenomena are objectionable.

To remove aliasing problems we can pre-filter the signal so that it's bandlimited, so it has no frequencies above some maximum ω_m (or in the real world, very little energy in frequencies above ω_m). This is usually hard to do for 3D scenes.

The alternative is to to reduce the sampling interval T, which causes the copies of the signal's spectra to spread apart more. When T is small enough, the copies

of G in $G * S$ won't overlap, and we can extract a clean version of just a single instance of the signal's spectrum G from the sampled spectrum.

This is why we usually say that more samples give us a better final image!

14 The Quantum Fourier Transform (QFT)

We can now move from the classical domain of traditional computing, and into the new domain or quantum computing.

We'll take our hard-earned knowledge of the DFT and build its quantum version, the **quantum Fourier transform**, or **QFT**!

From here on, I'll assume you're familiar with the basic ideas and notation of quantum computing, including quantum bits and quantum gates, the phenomena of superposition, entanglement, measurement, and interference, local and relative phase, tensor products, and Dirac (or braket) notation. If these things are new or unfamiliar to you, I encourage you to pause now and consult a reference to come up to speed. You'll want to choose a reference based on your background, how rigorous you like your mathematics, and how you like to learn.

The key question we need to ask right away is whether we can use the W matrix from Equation 36 directly as a quantum gate, or qugate.

We're off to a good start, because the W matrix is square. But is it reversible, as all qugates must be?

We can check this by writing out all the terms. It's laborious and not enlightening, so I won't go through it here. The bottom line is that $W\overline{W} = I$, or $W^{-1} = \overline{W}$, so yes, W can be a qugate!

This is big news! Finding that W can be implemented as a quantum gate means that we can move right into the quantum realm.

Before we cross the bridge to the quantum Fourier transform, I'll first discuss three ideas that will smooth the way.

14.1 A Useful Way to Write H

There are never enough ways to write the matrix for the Hadamard qugate, H. For reference, I've given the definition of H in Equation 54. Recall that in these notes, $\vee = 1/\sqrt{2}$.

$$H = \vee \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{54}$$

Let's see another way to write H. Suppose that we have an integer j that can be 0 or 1. Let's look at the exponential $e^{i2\pi(j/2)}$. If $j = 0$, then the exponent is 0 and the whole term is 1. If $j = 1$, then the exponent is $i\pi$, and we know that $e^{i\pi} = -1$.

These two cases are summarized in Equation 55.

$$e^{i2\pi(j/2)} = \begin{cases} 1 & \text{if } j = 0 \\ -1 & \text{if } j = 1 \end{cases} \tag{55}$$

Using this, for any quantum state $|j\rangle$ that is either $|0\rangle$ or $|1\rangle$, we can write $H|j\rangle$ as in Equation 56.

$$H|j\rangle = \vee(|0\rangle + e^{i2\pi(j/2)}|1\rangle), \qquad j \in \mathbb{B} \tag{56}$$

Though Equation 56 looks messy, that exponential is just a complicated way of writing $(-1)^j$ for the coefficient on $|1\rangle$.

It's always a good day when we can add another version of H to our collection. As I'm sure you anticipated, we're going to use this form of H in the upcoming discussion.

14.2 A New Shortcut

We're going to see a *lot* of complex exponentials coming up, and most will be of the form $e^{i2\pi a}$ for some real number a.

Whenever I see the same thing appearing over and over, my programmer's instincts kick in and tell me to give the recurring thing a single, consistent name. This reduces the chance for errors and also makes everything easier to read.

So for the rest of this these notes, I'll use λ as defined in Equation 57.

$$\lambda \triangleq e^{i2\pi} \tag{57}$$

Comparing this ω from Equation 29 shows that $\lambda = \omega^N$ or $\omega = \lambda^{1/N}$.

Now we can write an expression $e^{i2\pi a}$ more compactly as λ^a, as shown in Equation 58

$$e^{i2\pi a} = (e^{i2\pi})^a = \lambda^a \tag{58}$$

Using λ cuts down on squinting at many tiny repeated instances of $i2\pi$ in exponents and searching for any changes.

Two particular exponents on λ will pop up a few times, so let's summarize them in Equation 59

$$\begin{aligned} \lambda^k &= 1 \quad \text{For any integer } k \\ \lambda^{1/2} &= -1 \quad \text{Because } \lambda^{1/2} = e^{i2\pi/2} = e^{i\pi} = -1 \end{aligned} \tag{59}$$

We can use the second observation to write the version of H in Equation 56 in a more compact way in Equation 60.

$$H\,|j\rangle = \vee(|0\rangle + \lambda^{j/2}\,|1\rangle), \qquad j \in \mathbb{B} \tag{60}$$

I used the symbol $\mathbb{B}$ to refer to the set of bits, $\{0, 1\}$. In other words, in this expression, j can be either 0 or 1.

14.3 The Phase Qugate

The Z qugate introduces a local phase of -1 into a qubit. As a reminder, its matrix is shown in Equation 61, along with a version using $\lambda^{1/2}$ for -1.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \lambda^{1/2} \end{bmatrix} \tag{61}$$

Writing the matrix in the second form suggests that it might be useful to use other exponents on λ.

This is definitely true, and some choices are so useful that their corresponding qugates have conventional names. Equation 62 shows two of these [Neilsen and Chuang, 2011](§4.2).

$$S = \begin{bmatrix} 1 & 0 \\ 0 & \lambda^{1/4} \end{bmatrix}, \qquad T = \begin{bmatrix} 1 & 0 \\ 0 & \lambda^{1/8} \end{bmatrix} \tag{62}$$

Let's generalize these qugates. I'll use the letter R for this qugate, with a subscript of a to hold the exponent, representing $2\pi/a$ radians.

The qugate R_a is defined in Equation 63.

$$R_a \triangleq \begin{bmatrix} 1 & 0 \\ 0 & \lambda^a \end{bmatrix} \tag{63}$$

This shares a nice feature that with our various μ functions we saw earlier in these notes. We put 2π into the exponent there so that we could think of the argument as specifying the number of full rotations around the unit circle in the

Argand diagram. We now have 2π in the exponent of λ, so the argument a in R_a tells us how many rotations we're making. For example, $R_{1/2}$ means that the state will be changed by adding a phase of $2\pi/2 = \pi$, and $R_{1/4}$ means adding a phase of $2\pi/4$. This is a lot nicer than having to deal with 2π appearing everywhere.

Using this definition, we can write S and T compactly as in Equation 64.

$$S = R_{1/4}, \qquad T = R_{1/8} \tag{64}$$

Let's see R_a in action. Suppose we have some arbitrary state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ (where $|\alpha|^2 + |\beta|^2 = 1$ as always, because of the Born rule). Applying R_a gives us Equation 65.

$$R_a|\psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & \lambda^a \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \lambda^a \beta \end{bmatrix} \tag{65}$$

So R_a introduces a relative phase of λ^a into any state. This is a great general-purpose tool for adding or removing any amount of phase from any state.

Be aware that the definition I'm using for the phase qugate in Equation 63 is common, but it's only one of many variants, each using a slightly different complex number in the bottom right element of the matrix. If you're reading a discussion and things aren't making sense, a good place to start working out the problem is to check how that author defines the phase qugate.

People also sometimes write the argument in parentheses, rather than as a subscript, as in something like $R(1/2)$. I like the subscript because I think $R_{1/2}|\psi\rangle$ is convenient to read at a glance, and $R(1/2)|\psi\rangle$ isn't.

I'll draw this qugate in a diagram in the usual way, as a box containing R_a for a specific value of a, as shown in Figure 37

Figure 37: Left: A phase qugate with a phase of $1/4$. Right: A controlled version of that qugate where the lower qubit is the control.

In these notes we'll work with *controlled* phase qugates. These follow the same rules for any other controlled qugate: when the control is $|0\rangle$ the qugate is ignored, and when the control is $|1\rangle$ the qugate is applied. Superpositions of the input produce superpositions in the output.

15 The QFT

Cue the band! We've arrived at the QFT!

Since this is the quantum version of the discrete Fourier transform, it seems to me that it should have been called the **Quantum Discrete Fourier Transform**, or **QDFT**. But it's now known universally as the **Quantum Fourier Transform**, or **QFT**.

15.1 From DFT to QFT

Since the DFT is a time-proven tool, let's make a quantum version that's as close to a DFT as we can get.

The structure of the DFT is that it changes input numbers to output numbers. In the quantum world, we work with states, so let's replace numbers with states.

I'll repeat the analysis equation from Equation 37 (though I'll change the n there to an integer j, since the convention in quantum computing uses n for for the number of qubits), and beneath that I'll write a version that replaces the numbers with states. The pair is shown in Equation 66.

$$\begin{aligned} X_j &= Wk, && \text{DFT} \\ |j\rangle &= W\,|k\rangle\,, && \text{QFT} \end{aligned} \tag{66}$$

Boom! The top line is the forward DFT, the bottom line is the forward QFT! And we know it can be realized in practice, because we saw earlier that the W matrix is reversible.

These kinds of super terse expressions are great, but let's expand this out so we can see the details. I'll start by repeating the expanded form of the forward DFT from Equation 28 (with n changed to j), and then simplify it using our new λ shortcut.

$$\begin{aligned} X_j &= \frac{1}{\sqrt{N}} \sum_{k\in[N]} x_k e^{ikj(2\pi/N)} && \text{From Eq. 28} \\ &= \frac{1}{\sqrt{N}} \sum_{k\in[N]} \lambda^{jk/N} x_k && \text{Use } \lambda = e^{i2\pi} \end{aligned} \tag{67}$$

Writing this for the quantum world, I'll use $\mathcal{Q}$ for the forward QFT and $\mathcal{Q}^{-1}$ for the backward QFT. Combining Equations 66 and 67 gives us the forward QFT in Equation 68.

$$Q\,|j\rangle = \frac{1}{\sqrt{N}} \sum_{k \in [N]} \lambda^{jk/N}\,|k\rangle \tag{68}$$

The biggest change is that we're not computing a transformation of an input signal any more. That's gone. Instead, we're transforming a computational basis state $|j\rangle$ into some new state $Q\,|j\rangle$. We call the set of new states the **Fourier basis**.

Since there are N input states $|j\rangle$ in the computational basis, plugging each one in turn into Equation 68 gives us N new states making up the Fourier basis.

I've written the forward QFT from Equation 68 along with the backward QFT The forward and backward forms of the QFT are shown in Equation 69.

The QFT

$$\begin{aligned} Q\,|j\rangle &\triangleq \frac{1}{\sqrt{N}} \sum_{k \in [N]} \lambda^{jk/N}\,|k\rangle \\ Q^{-1}\,|j\rangle &\triangleq \frac{1}{\sqrt{N}} \sum_{k \in [N]} \lambda^{-jk/N}\,|k\rangle \end{aligned} \tag{69}$$

In matrix form, we get Equation 70, writing $|\Psi\rangle = Q\,|\psi\rangle$ for any state $|\psi\rangle$. I've capitalized the symbol ψ here as Ψ to mirror our use of the capital X for DFT of the spectrum of a time signal x.

$$\begin{aligned} |\Psi\rangle_j &= W\,|\psi\rangle_j \\ |\psi\rangle_k &= \overline{W}\,|\Psi\rangle_k \end{aligned} \tag{70}$$

Let's take the QFT out for a spin.

15.2 The QFT for $n = 1$

Let's see what the analysis equation in Equation 69 does when $n = 1$ (so $N = 2^1 = 2$). This is the case of a single qubit, so we only have the basis states $|0\rangle$ and $|1\rangle$ to transform. For $Q\,|0\rangle$, we get Equation 71. Remember that I'm using the centered dot ($\cdot$) for multiplication.

$$\begin{aligned}
Q|0\rangle &= \frac{1}{\sqrt{2}} \sum_{k\in[2]} \lambda^{0k/N} |k\rangle && \text{Use definition in Eq 69} \\
&= \vee(\lambda^{0\cdot 0/2} |0\rangle + \lambda^{0\cdot 1/2} |1\rangle) && \text{Expand the sum and use } \vee = 1/\sqrt{2} \\
&= \vee(\lambda^0 |0\rangle + \lambda^0 |1\rangle) && \text{Since } 0k/N = 0 \text{ for any } k \\
&= \vee(|0\rangle + |1\rangle) && \text{Use } \lambda^0 = 1 \\
&= |+\rangle && \text{Definition of } |+\rangle
\end{aligned} \tag{71}$$

For $Q|1\rangle$, I'll follow the same steps, giving us Equation 72.

$$\begin{aligned}
Q|1\rangle &= \frac{1}{\sqrt{2}} \sum_{k\in[2]} \lambda^{1k/N} |k\rangle \\
&= \vee\left(\lambda^{1\cdot 0/2} |0\rangle + \lambda^{1\cdot 1/2} |1\rangle\right) \\
&= \vee(\lambda^0 |0\rangle + \lambda^{1/2} |1\rangle) \\
&= \vee(|0\rangle - |1\rangle) && \text{Use } \lambda^0 = 1 \text{ and } \lambda^{1/2} = -1 \\
&= |-\rangle
\end{aligned} \tag{72}$$

These are the same results we get from applying H to a qubit, so the H qugate, in addition to its other superpowers, also computes the QFT of a single qubit!

How do we take the QFT of an arbitrary input qubit that is *not* either $|0\rangle$ or $|1\rangle$, but in some possibly non-uniform superposition $\alpha|0\rangle + \beta|1\rangle$ (where as usual, $|\alpha|^2 + |\beta|^2 = 1$). Linearity is our friend here, as shown in Equation 73

$$\begin{aligned}
Q(\alpha|0\rangle + \beta|1\rangle) &= \alpha Q|0\rangle + \beta Q|1\rangle && \text{Distribute } Q \text{ because it's linear} \\
&= \alpha|+\rangle + \beta|-\rangle && \text{From Eqs 71 and 72} \\
&= \alpha H|0\rangle + \beta H|1\rangle && \text{Since } H|0\rangle = |+\rangle \text{ and } H|1\rangle = |-\rangle \\
&= H(\alpha|0\rangle + \beta|1\rangle) && \text{Distribute } H \text{ because it's linear}
\end{aligned} \tag{73}$$

So any way we go about it, the QFT of any single qubit's state $|\psi\rangle$ is the same as applying H, or $Q|\psi\rangle = H|\psi\rangle$.

Thus, we can draw a circuit to implement the 1-bit QFT in two ways, as shown in Figure 38.

Figure 38: Left: The QFT operation using a generic qugate. Right: The implementation for one qubit using an H qugate.

15.3 The QFT for $n = 2$

Let's scale up to 2 qubits. Generalizing from our last section, we might guess that the circuit should look something like Figure 39.

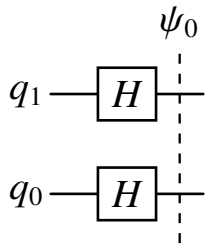

Figure 39: A first (but wrong!) attempt at the QFT of two qubits.

Warning! This is *not* the 2-qbit QFT! We're missing an essential piece of the definition: *interference between qubits*. In terms of the math, this approach is not including the summation in Equation 69.

The heart of the problem here is that when the two input qubits are entangled (which is the usual case, as tensor product states are rare), we can't process the qubits individually and then recombine them. We must instead take them as a unit. We can express this symbolically in Equation 74, using the tensor product $\otimes$ to combine quantum states into a single system.

$$Q|q_1\, q_0\rangle \neq Q|q_1\rangle \otimes Q|q_0\rangle \tag{74}$$

Let's see what's missing. Then we can fix up our circuit of Figure 39 to include those missing parts. While there are lots of way to go about this discussion, I'll base this presentation on [Simha, 2022](§11.10).

A word of warning: there's a bunch of algebra ahead. It's almost all just juggling terms around, grouping them, replacing them with known values, and stuff like that, and I'll show you every step. It's all basic algebra. It's just that

there are a lot of steps! I'll break things down hierarchically so we can focus on one piece of the puzzle at a time, but I suggest keeping the bigger picture in the back of your head.

It might help to read this section through lightly the first time to see the general structure of what's happening, and then you can return for a closer look if you want to nail down the mathematical steps.

To keep the notation from becoming a soup of new, arbitrary letters, I'll use a symbol q as both the label of a qubit, and its value when interpreted as a bit.

I'll be using the letters k and q in the following discussion. Since these will both range from 0 to $N - 1 = 3$, we can represent both integers using two bits. Let's say that when we write q in binary, q_1 is the most significant bit, and q_0 is the least significant bit, as usual in binary notation. Then $q = 2q_1 + q_0$. In the same way, $k = 2k_1 + k_0$.

Let's first find what the QFT of the combined qubit system q gives us. Starting with the synthesis equation from Equation 69, I've written out the steps in Equation 75.

$$\begin{aligned} Q|q\rangle &= \frac{1}{\sqrt{4}} \sum_{k\in[4]} \lambda^{qk/4} |k\rangle && \text{From Eq. 69} \\ &= \frac{1}{2} \sum_{k\in[4]} \lambda^{q(2k_1+k_0)/4} |k\rangle && \text{Replace } k \text{ with } (2k_1 + k_0) \\ &= \vee^2 \sum_{k\in[4]} \left(\lambda^{q2k_1/4} \lambda^{qk_0/4}\right) |k\rangle && \text{Split the exponent} \end{aligned} \tag{75}$$

The two exponentials on the last line of Equation 75 rely on k_1 and k_0 independently, so we can split the sum into two loops. That is, rather than run k from 0 to 3 (or the bitstrings 00, 01, 10, 11), I'll run k_1 from 0 to 1, and the same for k_0. Since $\mathbb{B} = \{0, 1\}$, we can write these loops using a big sigma with the subscripts $k_1 \in \mathbb{B}$ and $k_0 \in \mathbb{B}$ for the loop limits. I've written this correspondence out explicitly in Equation 76

$$\sum_{k_1=0}^{1} \equiv \sum_{k_1\in\mathbb{B}} \tag{76}$$

Picking up from the last line of Equation 77, this gives us Equation 77

$$
\begin{aligned}
Q\,|q\rangle &= \vee^2\Big(\sum_{k_1\in\mathbb{B}} \lambda^{q2k_1/4}\,|k_1\rangle\Big)\Big(\sum_{k_0\in\mathbb{B}} \lambda^{qk_0/4}\,|k_0\rangle\Big) && \text{Make two summations}\\
&= \vee^2\Big(\sum_{k_1\in\mathbb{B}} \underbrace{(\lambda^{2q/4})^{k_1}}_{a}\,|k_1\rangle\Big)\Big(\sum_{k_0\in\mathbb{B}} \underbrace{(\lambda^{q/4})^{k_0}}_{b}\,|k_0\rangle\Big) && \text{Use } x^{yz} = (x^y)^z
\end{aligned}
\tag{77}
$$

I've named the two exponential terms a and b. Let's see if we can simplify them, looking at a first.

I'll expand q as we discussed before into binary form, $q = 2q_1 + q_0$. This change, and a few steps of algebra afterwards, are shown in Equation 78. I'll write out every step in this simplification, because this is the template that we'll use for every qubit, even in bigger systems when there are many qubits and the expressions get much longer.

$$
\begin{aligned}
a &= \lambda^{2q/4} && \text{From Eq. 77}\\
&= \lambda^{q/2} && \text{Simplify exponent}\\
&= \lambda^{(2q_1+q_0)/2} && \text{Expand } q\\
&= \lambda^{2q_1/2}\lambda^{q_0/2} && \text{Split exponent}\\
&= \lambda^{q_1}\lambda^{q_0/2} && \text{Simplify first exponent}\\
&= 1\cdot\lambda^{q_0/2} && \text{Because } \lambda^m = 1 \text{ for } m\in\mathbb{Z}\text{, see Eq. 59}\\
&= (\lambda^{1/2})^{q_0} && \text{Using } a^{bc} = (a^b)^c\\
&= (-1)^{q_0} && \text{Since } \lambda^{1/2} = -1\text{, see Eq. 59}
\end{aligned}
\tag{78}
$$

That's interesting. Although we started with $\lambda^{2q/4}$ which involved the entire input number q, the value of that expression depends *only* on the least significant bit q_0.

Let's run through the same process using the term marked b in Equation 77. The steps are shown in Equation 79. The only difference between our starting point for b and that for a in Equation 78 is that we're now missing a factor of 2 in the exponent. The steps will parallel those of Equation 79, but we can skip the first simplifying step.

$$\begin{aligned}
b &= \lambda^{q/4} && \text{From Eq. 77} \\
&= \lambda^{(2q_1+q_0)/4} && \text{Expand } q \\
&= \lambda^{2q_1/4}\lambda^{q_0/4} && \text{Split exponent} \\
&= \lambda^{q_1/2}\lambda^{q_0/4} && \text{Simplify first exponent} \\
&= (\lambda^{1/2})^{q_1}(\lambda^{1/4})^{q_0} && \text{Using } a^{bc} = (a^b)^c \\
&= (-1)^{q_1}(\lambda^{1/4})^{q_0} && \text{Because } \lambda^{1/2} = -1
\end{aligned} \tag{79}$$

This expression is more complicated than we got for the case $k = 0$, but let's roll with it and see where it goes.

I'll take the simplified versions of a and b from Equation 78 and Equation 79 respectively and plug them back into Equation 77. The result is shown in Equation 80.

$$\begin{aligned}
Q|q\rangle &= \vee^2\Big(\sum_{k_1\in\mathbb{B}} \underbrace{(\lambda^{q2/4})^{k_1}}_{a}|k_1\rangle\Big)\Big(\sum_{k_0\in\mathbb{B}} \underbrace{(\lambda^{q/4})^{k_0}}_{b}|k_0\rangle\Big) && \text{Eq. 77} \\
&= \vee^2\Big(\sum_{k_1\in\mathbb{B}} \underbrace{((-1)^{q_0})^{k_1}}_{a}|k_1\rangle\Big)\Big(\sum_{k_0\in\mathbb{B}} \underbrace{((-1)^{q_1}(\lambda^{1/4})^{q_0})^{k_0}}_{b}|k_0\rangle\Big) && \text{Replace } a \text{ and } b
\end{aligned} \tag{80}$$

This last equation looks a lot scarier than it really is.

To see this, let's write out the loops. Each has only two terms. When $k_1 = 0$, then the expression a is 1 (since the exponent is 0), and when $k_1 = 1$, then $a = (-1)^{q_0}$. In the same way, when $k_0 = 0$, then the expression b is 1, and when $k_0 = 1$, then $b = (-1)^{q_1}(\lambda^{1/4})^{q_0}$.

Putting these together we get Equation 81.

$$Q|q\rangle = \vee\Big(|0\rangle + (-1)^{q_0}|1\rangle\Big) \vee \Big(|0\rangle + (-1)^{q_1}(\lambda^{1/4})^{q_0}|1\rangle\Big) \tag{81}$$

You might recognize the first term of Equation 81, since it's just $H|q_0\rangle$ from Equation 56.

The second term is *almost* $H|q_1\rangle$ except for that extra $(\lambda^{1/4})^{q_0}$ term in there. We can interpret that term as an additional *relative phase* in the expression for that qubit. With a the phase qugate, we can introduce any desired phase into a state.

So one way to contruct this term is to find $H|q_1\rangle$ and then apply a phase qugate $R_{1/4}$ to it. That won't work out quite right, because when $q_0 = 0$ we have $(\lambda^{1/4})^0 = 1$, and we don't want the qugate applied. We only want to include that extra phase when $q_0 = 1$.

Easily done, it's barely an inconvenience!

We can use a *controlled* phase qugate, using q_0 for the control. So when $q_0 = 0$, the qugate is ignored, but when $q_1 = 1$, the qugate operates and introduces the additional phase we want.

Let's look at this more closely.

Using the definition of R_a in Equation 63, we can write the matrix $R_{1/4}$ in Equation 82.

$$R_{1/4} = \begin{bmatrix} 1 & 0 \\ 0 & \lambda^{1/4} \end{bmatrix} \tag{82}$$

Let's assume for a moment that $q_0 = 1$. Then we'll apply $R_{1/4}$ to the output of $H|q_1\rangle$. I've multiplied out the matrices in Equation 83

$$R_{1/4}Q|q_1\rangle = R_{1/4}H|q_1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & \lambda^{1/4} \end{bmatrix} \vee \begin{bmatrix} 1 \\ (-1)^{q_1} \end{bmatrix} = \vee \begin{bmatrix} 1 \\ (-1)^{q_1}\lambda^{1/4} \end{bmatrix} \tag{83}$$

Perfect! We've matched the second expression in Equation 80. Let's write that equation again, using these new results, in Equation 84. Here I've written $CR^0_{1/4}$ for the controlled-R qugate with an argument of $1/4$, controlled by qubit q_0.

$$Q|q\rangle = H|q_0\rangle \; CR^0_{1/4} \, H|q_1\rangle \tag{84}$$

The circuit corresponding to this expression is shown in Figure 40.

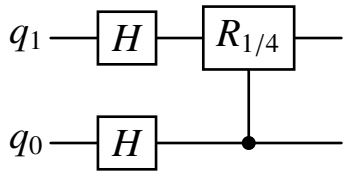

Figure 40: An improved QFT of two qubits.

That's just about right. Note that the qubits are interacting now, which they weren't doing in Figure 39.

There's just one little issue to clean up. The qubits are coming out in reversed order! It's a quirk of this nice implementation of the QFT that it computes the output qubits in an order reversed from their inputs. In this case, we only need to swap the two outputs, but when there are more qubits we'll see that the more general fix is to reverse the entire order.

Figure 41 shows a version using an explicit $SWAP$ qugate on the left, and a more general "Reverse" qugate on the right. We can always build the "Reverse" qugate using a series of swaps. For n qubits, we'd swap q_{n-1} with q_0, and q_{n-2} with q_1, and so on.

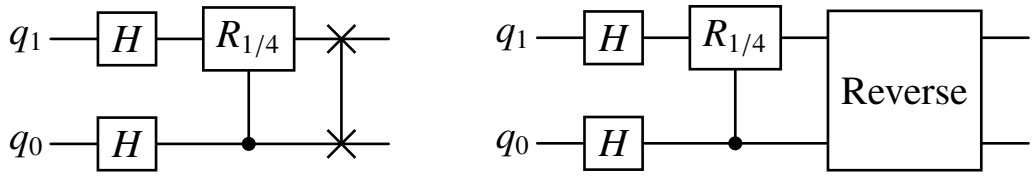

Figure 41: The QFT of two qubits. Left: Explicitly swapping the two outputs. Right: Using a more general "Reverse" qugate to reverse all outputs, so inputs top to bottom come out from bottom to top.

15.4 The QFT for $n = 3$

To lock all of this down, let's find the QFT of an input with *three* qubits. We'll follow the same process in the same way, so I'll skip most of the discussion and just show the mechanics.

There are no new ideas or operations here. It's the same math as before, following the same steps, but there's a *lot* more of it because we now have three qubits to keep track of everywhere. I'll go through all the mechanics in case you'd like one more example. If you feel like you've already got it, you can skim to the end this section where I show the final circuit.

This section is where our use of λ will turn what could have been a nightmare into merely a busy dream.

This time, our integers k and q are three bits, so each k can be written as $4k_2 + 2k_1 + k_0$ and each q as as $4q_2 + 2q_1 + q_0$.

Let's start by writing $Q|q\rangle$ for $n = 3$ (and therefore $N = 2^n = 8$), in Equation 85.

$$
\begin{aligned}
Q\,|q\rangle &= \frac{1}{\sqrt{8}} \sum_{k\in[8]} \lambda^{qk/8}\,|k\rangle && \text{Eq. 69 for } N = 8 \\
&= \frac{1}{\sqrt{8}} \sum_{k\in[8]} \lambda^{a(4k_2+2k_1+k_0)/8}\,|k\rangle && \text{Expand } k \\
&= \frac{1}{\sqrt{8}} \sum_{k\in[8]} \lambda^{q4k_2/8}\lambda^{q2k_1/8}\lambda^{qk_0/8}\,|k\rangle && \text{Split exponent} \\
&= \vee^3 \sum_{k_2\in\mathbb{B}} \lambda^{q4k_2/8}\,|k_2\rangle \sum_{k_1\in\mathbb{B}} \lambda^{q2k_1/8}\,|k_1\rangle \sum_{k_0\in\mathbb{B}} \lambda^{qk_0/8}\,|k_0\rangle && \text{Use 3 sums} \\
&= \vee\Big(\sum_{k_2\in\mathbb{B}} \underbrace{(\lambda^{q4/8})}_{r}^{k_2}\,|k_2\rangle \Big) \vee \Big(\sum_{k_1\in\mathbb{B}} \underbrace{(\lambda^{q2/8})}_{s}^{k_1}\,|k_1\rangle \Big) \vee \Big(\sum_{k_0\in\mathbb{B}} \underbrace{(\lambda^{q/8})}_{t}^{k_0}\,|k_0\rangle \Big) && \text{Identify } r,\ s,\ t
\end{aligned}
\tag{85}
$$

Now we'll simplify each of the three values r, s, and t, starting with r in Equation 86. I'll use the same steps as in Equation 78 and Equation 79.

$$
\begin{aligned}
r &= \lambda^{q4/8} \\
&= \lambda^{(4q_2+2q_1+q_0)4/8} \\
&= \lambda^{4q_2 4/8}\lambda^{2q_1 4/8}\lambda^{q_0 4/8} \\
&= \lambda^{2q_2}\lambda^{q_1}\lambda^{q_0/2} \\
&= 1 \cdot 1 \cdot (\lambda^{1/2})^{q_0} \\
&= (-1)^{q_0}
\end{aligned}
\tag{86}
$$

Next we'll simplify s in Equation 87.

$$
\begin{aligned}
s &= \lambda^{q2/8} \\
&= \lambda^{(4q_2+2q_1+q_0)2/8} \\
&= \lambda^{4q_2 2/8}\lambda^{2q_1 2/8}\lambda^{q_0 2/8} \\
&= \lambda^{q_2}\lambda^{q_1/2}\lambda^{q_0/4} \\
&= 1 \cdot (\lambda^{1/2})^{q_1}(\lambda^{1/4})^{q_0} \\
&= 1 \cdot (-1)^{q_1}(\lambda^{1/4})^{q_0}
\end{aligned}
\tag{87}
$$

Finally, let's simplify t in Equation 88.

$$
\begin{aligned}
t &= \lambda^{q/8} \\
&= \lambda^{(4q_2+2q_1+q_0)/8} \\
&= \lambda^{4q_2/8}\lambda^{2q_1/8}\lambda^{q_0/8} \\
&= \lambda^{q_2/2}\lambda^{q_1/4}\lambda^{q_0/8} \\
&= (-1)^{q_2}(\lambda^{1/4})^{q_1}(\lambda^{1/8})^{q_0}
\end{aligned}
\tag{88}
$$

Now let's plug these three expressions for r, s, and t back into Equation 85. To avoid huge equations, I'll address each loop on its own, starting with the k_2 loop in Equation 89.

$$
\begin{aligned}
&\vee \sum_{k_2 \in \mathbb{B}} \underbrace{(\lambda^{q4/8})}_{r}^{k_2} |k_2\rangle \\
&= \vee \sum_{k_2 \in \mathbb{B}} [(-1)^{q_0}]^{k_2} |k_2\rangle \quad \text{Use } r \text{ from Eq 86} \\
&= \vee\Big(|0\rangle + (-1)^{q_0} \Big) \qquad \text{Write out the sum} \\
&= H |q_0\rangle
\end{aligned}
\tag{89}
$$

The circuit for this expression is shown in Figure 42. Notice that the result for q_2 depends only on q_0, which is part of the pattern that requires us to reverse the qubits at the end of the QFT.

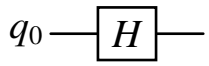

Figure 42: A circuit diagram for the k_2 loop.

Next up is the k_1 loop, expanded in Equation 90.

$$
\begin{aligned}
&\vee \sum_{k_1 \in \mathbb{B}} \underbrace{(\lambda^{q2/8})}_{s}^{k_1} |k_1\rangle \\
&= \vee \sum_{k_1 \in \mathbb{B}} [(-1)^{q_1}(\lambda^{1/4})^{q_0}]^{k_1} |k_1\rangle \quad \text{Use } s \text{ from Eq 87} \\
&= \vee\Big(|0\rangle + (-1)^{q_1}(\lambda^{1/4})^{q_0}) |1\rangle \Big) \\
&= CR^0_{1/4}\ H |q_1\rangle
\end{aligned}
\tag{90}
$$

So after we pass $|q_1\rangle$ through an H qugate, it passes through a controlled phase shift gate with a phase of $1/4$, controlled by qubit q_0. The circuit for this qubit is shown in Figure 43.

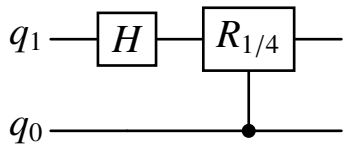

Figure 43: A circuit diagram for the k_1 loop.

Finally, the k_0 loop is in Equation 91.

$$\begin{aligned}
&\vee \sum_{k_0 \in \mathbb{B}} \underbrace{(\lambda^{q/8})}_{t}^{k_0} |k_0\rangle \\
&= \vee \sum_{k_0 \in \mathbb{B}} [(-1)^{q_1} (\lambda^{1/4})^{q_0}]^{k_0} |k_0\rangle \qquad \text{Use } t \text{ from Eq 88} \\
&= \vee \Big(|0\rangle + (-1)^{q_2} (\lambda^{1/4})^{q_1} (\lambda^{1/8})^{q_0} |1\rangle \Big) \\
&= CR^0_{1/8}\ CR^1_{1/4}\ H\,|q_1\rangle
\end{aligned} \tag{91}$$

A diagram for this circuit is shown in Figure 44.

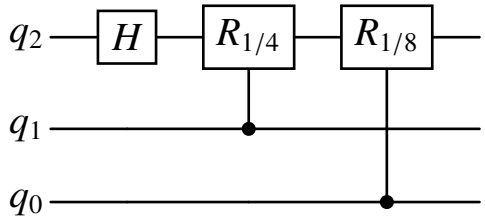

Figure 44: A circuit diagram for the k_0 loop.

The tensor product of the three results from Equations 89, 90, and 91 gives us the 3-qubit QFT, shown in Equation 92.

$$\mathcal{Q}\,|q\rangle = \Big(H\,|q_0\rangle\Big)\Big(CR^0_{1/4}\,H\,|q_1\rangle\Big)\Big(CR^0_{1/8}\,CR^1_{1/4}\,H\,|q_2\rangle\Big) \tag{92}$$

The circuit that implements this result is just the three smaller circuits we've seen before in the proper order (using qubits as controls before they get modified). At the end is a reversal of the sequence, which in this case exchanges the top and bottom outputs and leaves the center one untouched. The full 3-bit QFT circuit is shown in Figure 45.

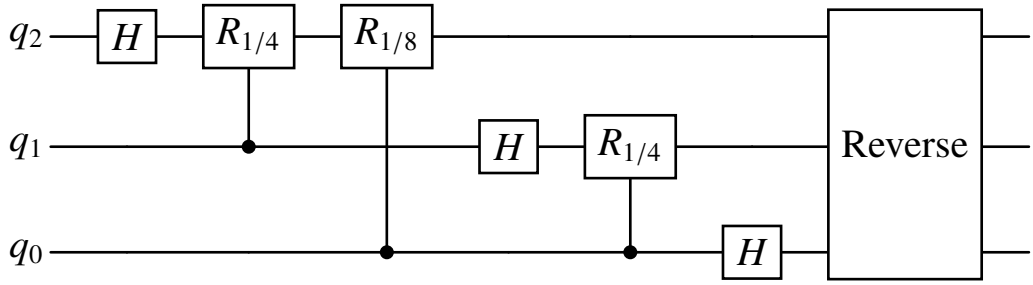

Figure 45: A circuit diagram for the 3-qubit QFT.

15.5 More Qubits

This pattern generalizes to form the QFT for any N.

In general, the topmost input qubit, q_{n-1}, goes into an H , and then a sequence of controlled-phase qugates. These controlled-phase qugates apply a phase starting at $1/4$, then $1/8$, $1/16$, and so on. Each controlled phase qugate is controlled by the qubit below it in order going downwards in the diagram. It's important to note that the control information from these qubits is used *before* those qubits are operated upon.

Then the next topmost qubit, q_{n-2}, goes through the same process, starting with the H and then a series of controlled-phase qugates starting at $1/4$ and shrinking by halves, controlled by the qubits below it in top-to-bottom order.

The process repeats until the final qubit, q_0, goes through an H qugate and nothing more.

Then we reverse the order of the qubits from top to bottom and we're done.

A beautiful aspect of this is how quantum parallelism and controlled gates let us start with a superposition of all computational basis states, and transform them all, simultaneously, into their corresponding Fourier basis states.

One way to think about the process is that we're creating a phase for each state based on the values of its qubits. For any qubit, the initial H qugate introduces a phase shift of $1/2$ of the qubit is 1, and then each descending qubit, if it's a 1, adds another phase shift of half that of the qubit above it.

We can visualize this as a tree. Figure 46 shows this for 3 qubits. We start at the top, with a state that has no local phase, and each step adds some quantity of phase (determined by the tree's level) if the control qubit is $|1\rangle$.

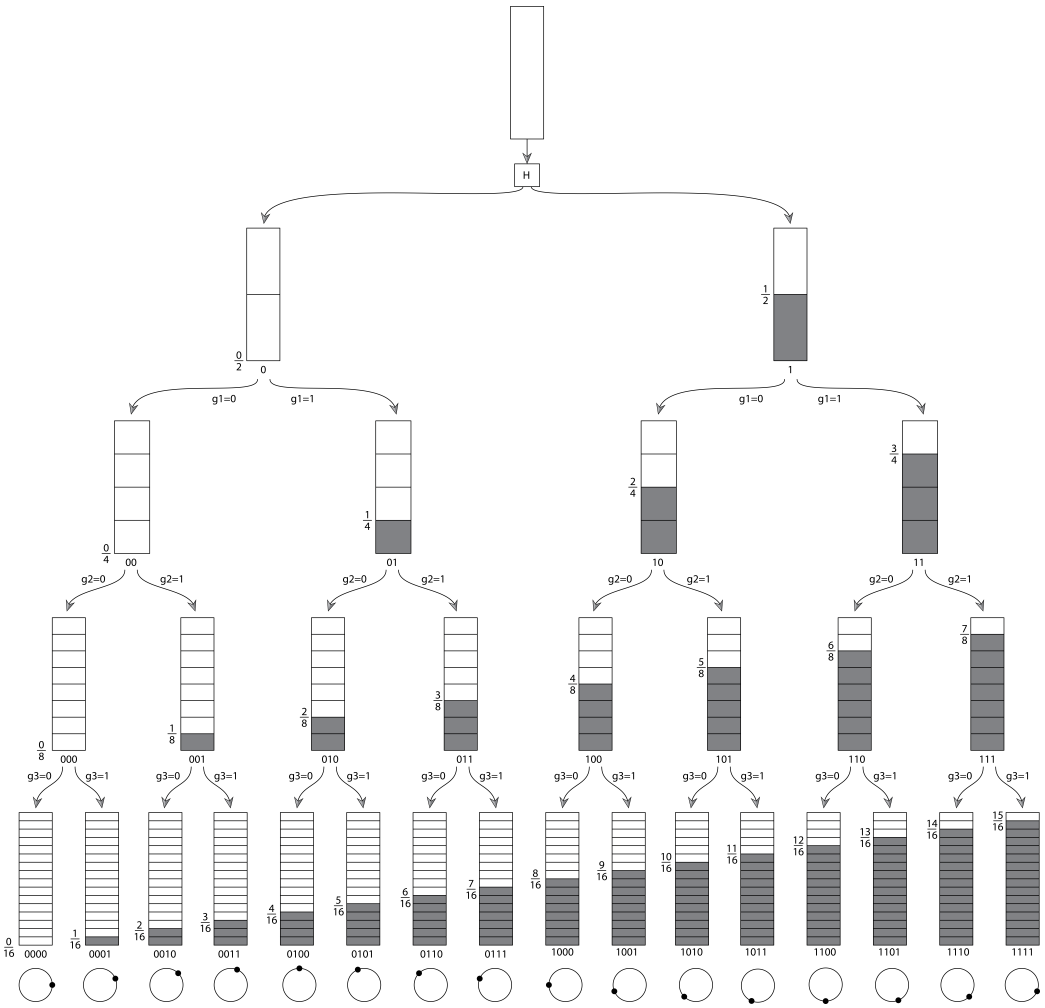

Figure 46: The process of determining the phase added to q_2 by the 3-qubit QFT.

In practice, we often don't need a precise QFT. If we need only an *approximate* solution, and a little erros is acceptable, then we can omit all the rotation gates on each line after the last one that makes a difference for our specific problem [Shor, 2022].

If you like doing algebra, you can really get this process into your bones by writing out the steps yourself using my derivations only as a guide, or even by finding your own way from the start to the end. If you're ambitious, then despite my earlier warning, you can write the QFT for $N = 4$ qubits. It requires a lot of meticulous bookkeeping that's easy to mess up, so I suggest a continuous block of time, a really big pad (you can find these at your local art supply store), and writing in pencil with a good eraser nearby. I sometimes find it useful to have several different colors on hand to highlight related operations. If it all works out correctly, consider framing that page!

You can also write an equation for the general case of the QFT for any n qubits. Again, give yourself plenty of time and space.

Another way to explore specific or general cases is to think about the steps abstractly, and let a symbolic algebra program do the algebra for you [SymPy, 2023].

Implementing the QFT in your favorite programming language can be a rewarding exercise that helps you get close to the mechanics.

16 Summary

The Fourier transform is a workhorse algorithm in signal processing. The Discrete Fourier Transform, or DFT, is the version we almost always use in digital computers. It allows us to convert a sequence of complex numbers representing measurements in what's usually called the *time* or *signal* domain into a different, but equivalent, sequence of numbers in what's usually called the *frequency* or spectral domain.

We can then analyze the spectrum of the signal to learn about its properties, and then modify the spectrum and synthesize a new signal from it.

The DFT is the tool that lets us understand important topics in computer graphics, like aliasing.

The QFT is the quantum version of the DFT. It lets us analyze input signals in ways that are analagous to how we use the DFT. The QFT is a workhorse algorithm of its own, and like the DFT, it shows up in surprising places. For example, the QFT is a vital step in Shor's famous algorithm for factoring prime numbers (and thereby cracking the RSA encryption systems, and its variants, used

to secure the majority of secret internet traffic).

17 Acknowledgements

Thank you to Stephen Drucker, Adam Finkelstein, and Eric Haines for helpful conversations, and my employer, Wētā FX, for support and providing such a wonderful and creative work atmosphere. Thanks also to many of the fine small and independent coffeeshops in Seattle and Vancouver, 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, JJ Bean, and The Fremont Coffee Company.

References

[Azad, 2023] Azad, K. (2023). Intuitive guide to convolution. `https://betterexplained.com/articles/intuitive-convolution/`.

[Carleson, 1966] Carleson, L. (1966). On convergence and growth of partial sums of Fourier series. *Acta Mathematica*, pages 135–157.

[Cooley and Tukey, 1965] Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301.

[Courant and Hilbert, 1937] Courant, R. and Hilbert, D. (1937). *Methods of Mathematical Physics, Volume 1*. Interscience Publishers Inc.

[Dijkstra, 1982] Dijkstra, E. W. (1982). Why numbering should start at 0. `https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF`.

[Dudgeon and Mersereau, 1984] Dudgeon, D. E. and Mersereau, R. M. (1984). *Multidimensional Digital Signal Processing*. Prentice-Hall.

[Gabel and Roberts, 1980] Gabel, R. A. and Roberts, R. A. (1980). *Signals and Linear Systems, Second edition*. John Wiley & Sons.

[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.

[Oppenheim and Schafer, 1975] Oppenheim, A. V. and Schafer, R. W. (1975). *Digital Signal Processing*. Prentice-Hall.

[Pasternak, 2015] Pasternak, B. (2015). Why does Griffiths define the complex inner product differently? https://physics.stackexchange.com/questions/215783/why-does-griffiths-define-the-complex-inner-product-differently.

[Sanderson, 2020] Sanderson, G. (2020). Complex number fundamentals. https://www.3blue1brown.com/lessons/ldm-complex-numbers.

[Shor, 2022] Shor, P. (2022). Lecture notes for 8.360/18.435 quantum computation. https://math.mit.edu/~shor/435-LN/.

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

[Smith, 1995] Smith, A. R. (1995). A pixel is *not* a little square. http://alvyray.com/Memos/CG/Microsoft/6_pixel.pdf.

[SymPy, 2023] SymPy (2023). Sympy. https://www.sympy.org/en/index.html.

[Valentinuzzi, 2016] Valentinuzzi, M. E. (2016). Highlights in the history of the Fourier transform. *IEEE Pulse*.

[Weisstein, 2023] Weisstein, E. W. (2023). Dirichlet Fourier series conditions. https://mathworld.wolfram.com/DirichletFourierSeriesConditions.html.

[White, 1998] White, P. (1998). Capacitor microphones explained: Their advantages for recording. *Sound on Sound*.

[Wikipedia, 2021] Wikipedia (2021). Joseph Fourier. https://en.wikipedia.org/wiki/Joseph_Fourier.