

A report on the Fourier Transform, representation in Desmos, and use in computing through Digital-to-Analog conversion and Video Games

Adomas Giedraitis

June 8, 2026

1 Abstract

This report investigates the Fourier Transform, beginning with the mathematics behind wave superposition, before explaining how a Fourier Transform is made, and then deriving a version reproducible in Desmos using Euler's Formula. After this, the report explores how computers turn real-world audio signals using ADC conversion into digital signals, using the Nyquist-Shannon Sampling Theorem, and what happens when this rule is broken, in the form of aliasing. After this, is a derivation of a Discrete Fourier Transform, which is then further optimised into the Cooley-Tukey Radix 2 FFT algorithm, reducing computational complexity to the point where modern computers can run them with very little computing power. Finally, the report examines a real-world application of these FFT algorithms, in the form of the wave simulation system used in the video games, Sea of Thieves.

1.1 Introduction

The Fourier Transform is a way of decomposing an input non-periodic wave into multiple frequency components. This can be useful for determining the strongest frequency in a signal or isolating certain frequencies in an audio recording. In this paper I will go through an example wave to deconstruct, how this can be reproduced in Desmos, how to optimise the Fourier Transform to enable it to be run with minimal processing power, and also the real-world application of this technology. As I have been producing music since the COVID-19 pandemic, I know that a core idea of sound synthesis and even composing chords, is stacking different frequency waves on top of each other to create new waves and more complex sounds. But then I wondered, how you would go backwards from your wave with mixed frequencies to finding your fundamental frequencies again? How does AI extract the chords from an audio file without having the original sheet music in front of them? How does an EQ work, displaying multiple different frequency levels? My research led me to this concept, and here I will explain what I have learned and apply the transform to a specific wave function, to isolate its composite frequencies, before talking about how computers can perform this transform as fast as possible, and how video games use this technology to simulate in-game aspects such as ocean waves.

2 How do waves combine?

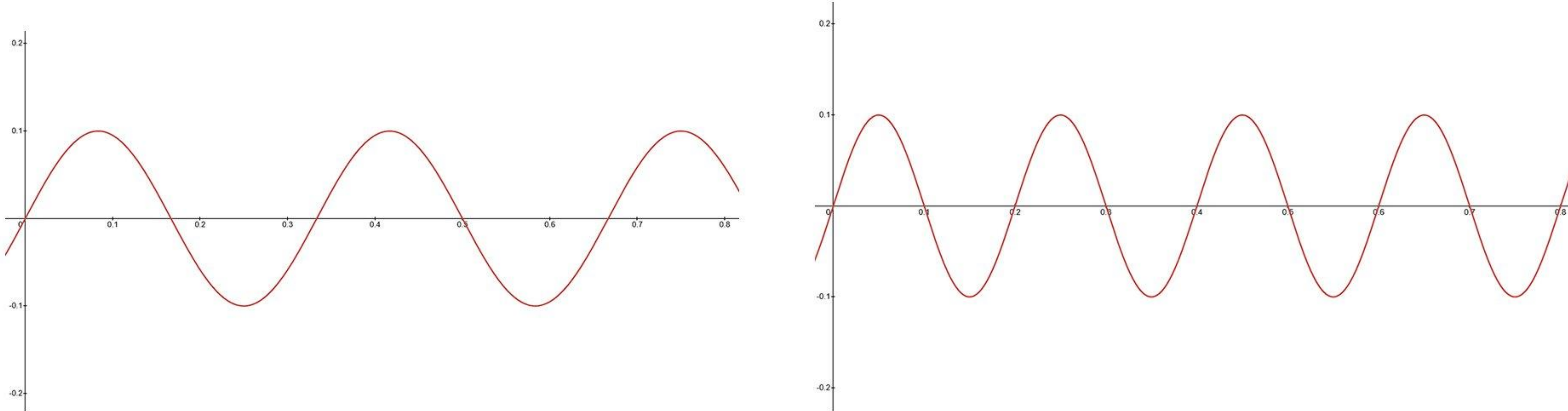
To understand the concept of a Fourier Transform, you must first understand how waves combine when they meet (or played at the same time.) This can be explained by the principle of superposition, which states:

When 2 or more waves overlap in the same space, the total displacement at any point is the sum of the individual displacements of each wave.

If you take a sine wave, the displacement of the wave can be a positive or negative value. If the value of displacement of both waves is positive (or both are negative) when they meet at a point, then the displacements are added together, but if the value of displacement of both waves are positive and negative respectively, then they end up cancelling out. The former mechanism is called constructive interference, and the latter is called destructive interference. This is how waves interact.

2.0.1 An example of this in action

For simplicity, let us take 2 waves, a 3 Hz signal and a 5 Hz signal with equal amplitude, and plot them on 2 separate graphs, with the y-axis representing displacement, and the x-axis representing time in seconds.



The wave function I am therefore using for this demonstration is given by

$$y = \sin(2\pi ft) \quad (1)$$

...where f is the aforementioned frequency of the waves, and t is time. Therefore, the first wave function can be written as

$$y = \sin(6\pi t) \quad (2)$$

... and the second wave function can be written as

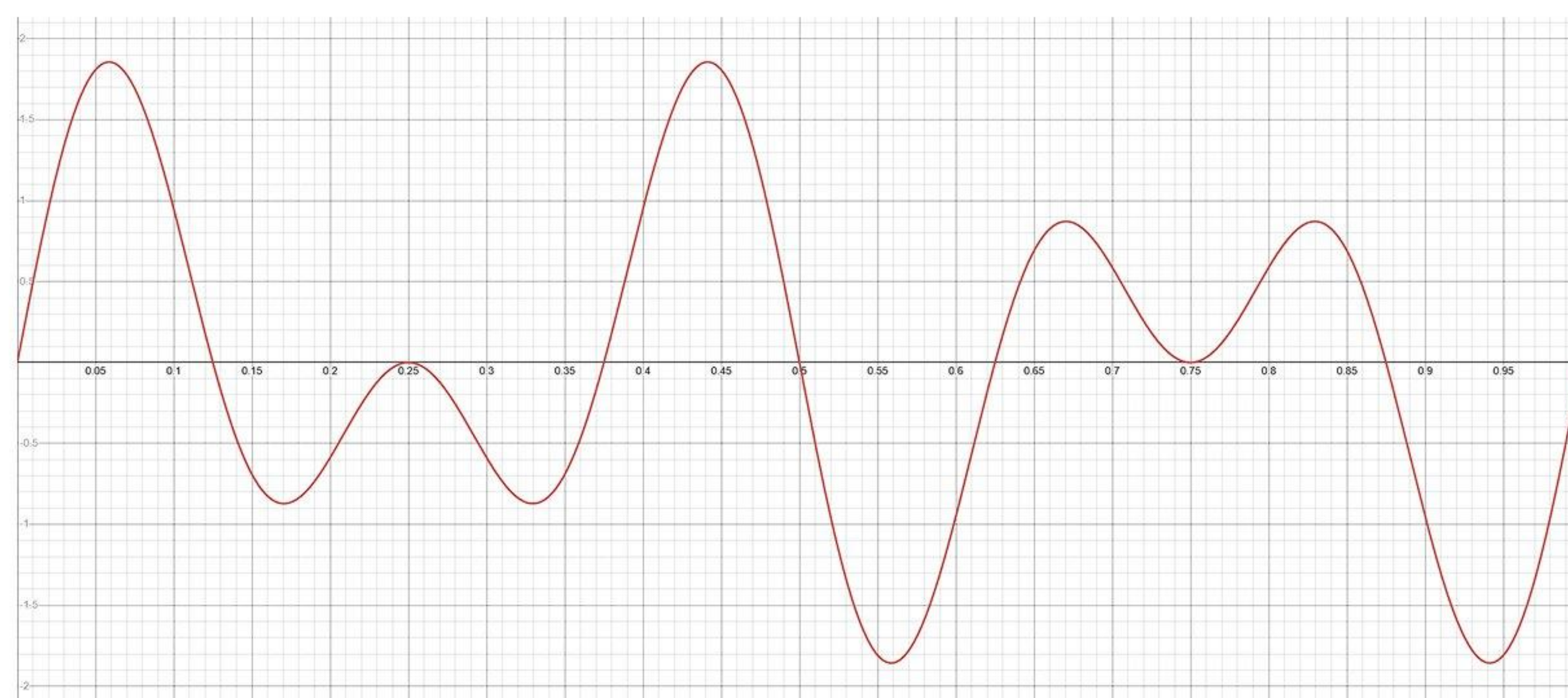
$$y = \sin(10\pi t) \quad (3)$$

Now that we have 2 defined wave functions, what happens when they interfere?

We can simulate these by adding these waves together all in one function, written as

$$y = \sin(6\pi t) + \sin(10\pi t) \quad (4)$$

Now we can see what this might look like.



As you can see, there are all manners of constructive and destructive interference happening, and we can see that we have generated a whole new type of wave. But how can we take a signal like this (even a non periodic one) and deconstruct it back into its original frequencies again? Let's break it down.

3 How to construct a Fourier transform

To begin to find out the resultant original wave's frequencies, we can plot this wave on the complex plane, centering around the origin. We can do this by multiplying our resultant wave function by

$$e^{-2\pi ift}$$

where our new function can be defined as $g(t, f)$. This results in...

$$g(t, f) = (\sin(6\pi t) + \sin(10\pi t))e^{-2\pi i f t} \quad (5)$$

You may wonder why we now have frequency as the variable of the new function instead of time. This is because we will need to vary f to create different windings around the graph.

Since this will get more complicated than it needs to be very quickly if we keep it in this form, let's just represent our original wave function as $h(t)$, so now it looks more manageable.

$$g(t, f) = h(t)e^{-2\pi i f t} \quad (6)$$

Now that we have our function plotted on the complex plane, we can vary f to create different windings of the graph around the origin. To find the original frequencies we were looking for, we can choose a number of points on the graph for a certain frequency, then average the complex positions themselves to find a so-called "center of mass" of the graph, also known as the centroid. We can do this with this function.

$$A_N(f) = \frac{1}{N} \sum_{k=1}^N h(t_k) e^{-2\pi i f t_k} \quad (7)$$

where N is the total number of points sampled, and t_k represents the time of the k 'th sample. As N increases, the number of points sampled becomes larger, and therefore our result becomes more and more accurate, as N tends to infinity, so this ends up transforming into a continuous integral over time. This is how we can represent this.

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-2\pi i f t} dt \quad (8)$$

Now since the output of this function is an imaginary value, we need to convert it to a real value to plot it on our new graph. We can do this by taking the modulus of the output of our function.

$$|H(f)| \quad (9)$$

Now to finally visually represent this, we can plot the modulus of $H(f)$ on a graph, where the x-axis is our varying frequency, and the y-axis is the value of the modulus of $H(f)$. However, how can we take all of this math and use a software like Desmos to compute all of it?

4 Reconstruction of Fourier Transform using Euler's Formula

The problem with Desmos is that the software does not support integrating imaginary numbers directly. What we need to do is represent our original imaginary component, that being...

$$e^{-2\pi i f t} \quad (10)$$

in another way. We can do this using a handy formula first published by Euler in 1748, which states...

$$e^{ix} = \cos x + i \sin x \quad (11)$$

Where $\cos x$ is our real component, and $i \sin x$ is our imaginary component.

To map this directly to Euler's formula, we identify the exponent's angle component as

$$x = -2\pi f t \quad (12)$$

We can now plug this value for x into Euler's Formula, therefore getting...

$$e^{-2\pi i f t} = \cos(-2\pi f t) + i \sin(-2\pi f t) \quad (13)$$

To clean this up, we can note that cosine is an even function...

$$\cos(x) = \cos(-x) \quad (14)$$

therefore we can rewrite the real portion of our equation as just...

$$\cos(2\pi ft) \quad (15)$$

Additionally, we can notice that sine is an odd function, as going into a negative angle flips the vertical direction of sine. And so, we can pull out the negative sign in our imaginary component to get...

$$-i \sin(2\pi ft) \quad (16)$$

Putting these two simplifications together, we obtain the equation...

$$e^{-2\pi i ft} = \cos(2\pi ft) - i \sin(2\pi ft) \quad (17)$$

Now we can multiply both sides of our equation by our original wave function, $h(t)$ to obtain...

$$h(t)e^{-2\pi i ft} = h(t) \cos(2\pi ft) - h(t)i \sin(2\pi ft) \quad (18)$$

You now can notice that the left side of our equation is equal to the one we integrated to get $H(f)$, so we can safely assume this new form of our equation produces the same results if it is integrated with bounds at infinity and negative infinity.

4.1 Plotting in Desmos

Using this new form, we can now integrate our real and imaginary components separately, which Desmos can easily do. Let's define our functions as...

$$h(t) = \sin(6\pi t) + \sin(10\pi t) \quad (19)$$

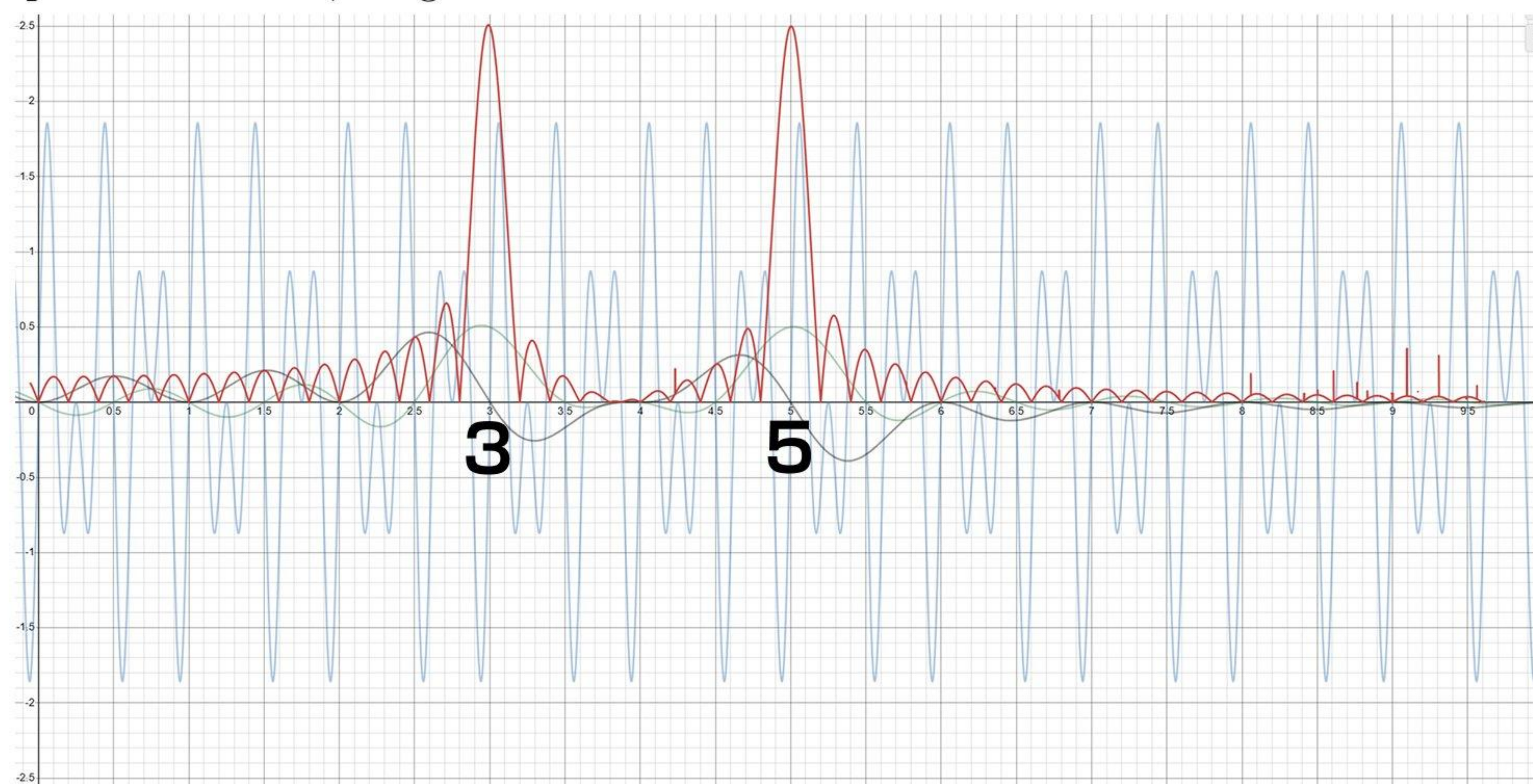
$$p(f) = \int_0^L h(t) \sin(2\pi ft) dt \quad (20)$$

$$q(f) = \int_0^L h(t) \cos(2\pi ft) dt \quad (21)$$

$$|H(f)| = \sqrt{(p(f))^2 + (q(f))^2} \quad (22)$$

...where $p(f)$ represents our imaginary component and $q(f)$ represents our real component.

The reason why L is used instead of infinity is because you cannot type in infinity in Desmos, therefore we need to approximate it by making L a very large number. (In the following image, $L = 5$, Desmos struggles to render the modulus function when L is higher, results may vary on your hardware, it may take incredibly long or not work at all.) Additionally, you may have realised that i has disappeared from our function of $p(f)$. This is the main reason why we express our function in two separate imaginary and real parts, as we can ignore i ; as it is just a scalar quantity, and the negative has disappeared from the imaginary function, as when it is squared it will end up as positive anyways, so we can leave it out. This means that we can express it in a Cartesian co-ordinate system, which Desmos supports. Please note that f needs to be replaced with x when entered into Desmos, as the variable on the x-axis cannot be changed, but I have kept it as f for this paper for consistency. So when we finally plot and set L to a positive number, we get...



...a clear graph (in red) with peaks at 3 and 5, which is the exact frequencies that made up our $h(t)$ wave in the first place! If L is increased, then the peaks become sharper and sharper, and makes it even more evident our individual frequencies are 3 and 5. However, this slightly simplified version of the Fourier transform made for Desmos highlights one key problem with the function as L increases, tending towards a more continuous function, especially when computers get in the mix, that being this process is very resource intensive and inefficient. How can we turn this algorithm into one that computers can easily deal with?

5 How computers turn the real into the digital

In order for a computer to process your wave using Fourier transform if it was recorded from a microphone for example, it must undergo Analog-to-Digital Conversion (ADC), where a chip in your microphone converts the continuous analog voltage waveform coming from fluctuations of a tiny coil. This coil picks up low-pressure and high-pressure areas of sound waves, which the chip then converts into binary data, which is then sent to your computer as a digital signal. However, a tiny detail of the technology means that the continuity of the signal is not reflected when converted to a digital output. So what is going on? Well, inside the chip there is a piece of circuitry called the Sample and Hold circuit, which takes the voltage level that is expressed when a sound wave moves the diaphragm and stores it temporarily, to give the chip time to compute the signal into a digital output. This set time is called the sampling rate, and is an incredibly important concept in the world of audio recording and music production. A higher sampling rate means higher quality audio, but ends up requiring more storage as more data is encoded in the same amount of time. Therefore, on old hardware, it may be intensive to process such a data heavy signal. Additionally, the digital signal is non-continuous in another way, where the raw amplitude of the analog signal cannot be exactly computed, so the chip needs to round the value to a certain amount, as binary data can only be as granular as the difference between a binary '0' and '1' rather than an infinite number of decimal points. This is called the bit depth, where the number of bits is the number of binary digits that the amplitude signal is quantised to.

8 Bit system = 2^8 possible volume levels = 256 possible volume levels
 16 Bit system = 2^{16} possible volume levels = 65536 possible volume levels
 24 Bit system = 2^{24} possible volume levels = 16777216 possible volume levels

The base of the exponent is always 2 because binary code can only be two values, on (1) and off (0), and the number of bits is just the number of binary numbers the digital signal is made of, with the corresponding possible volume levels resulting in the number of permutations of ones and zeros that are possible.

In short, computers struggle to deal with incredibly large values as it takes longer to process them, and theoretically if we had a fully continuous input into the computer, the processing time would be infinite. You can start to see why this is an issue. Fortunately there is a rule where our sample size doesn't have to be an infinite frequency, for our ears to be unable to tell the difference between an analog and a digital signal.

5.1 The Nyquist-Shannon Sampling Theorem

Let us take our sampling rate as f_s in Hz . The Nyquist-Shannon Sampling Theorem is a way to give us a precise mathematical statement which can set an upper limit on what sampling rate is acceptable to use. The theorem states that...

$$f_s > 2f_{\max} \quad (23)$$

This means that a continuous signal can be perfectly reconstructed from its discrete (non-continuous) samples if the sampling frequency is greater than twice the highest frequency component that is

present in the original signal. But what happens if the sampling frequency is less than twice the highest frequency component that is present in the original signal?

5.2 Aliasing

When an Analog-to-Digital converter samples a wave, it simply capturing isolated dots in time. If you sample a wave fast enough, then it is evident how to connect the dots between the points to reform your wave. But if your sampling frequency is too low, the distance between your isolated dots is big enough that your high-frequency wave oscillates entirely in-between them. As a result, the computer does not pick up the true peaks, meaning that you end up with a completely different and lower frequency than the original signal. This is the main reason why the standard sample rate is 44.1 *kHz*.

5.3 Why 44.1 kHz?

44.1 kHz is the general sampling frequency for many applications that render audio, or hard-coded into ADC's. If we use our Nyquist-Shannon Sampling Theorem from earlier, we can determine that our highest frequency that is being taken into account here is 22.05 kHz. This is nearly identical to the upper range of human hearing, quoted as around 20kHz. This figure of 22.05kHz gives some leeway for some people who can distinguish higher frequencies, but doesn't go so high as to use too high of a sampling rate that takes up unnecessary processing power. To prevent frequencies above 22.05kHz being captured and aliased into lower frequencies which then become audible as a metallic unpleasant ringing sound, most ADC's and audio rendering engines just cutoff all frequencies above that threshold before sampling, therefore removing any frequencies that can be aliased, and ensuring a clean, crisp sound.

5.4 The issue at hand

However, 44.1 kHz (44100 Hz) is still an incredibly short time interval to be sampling over. If we do...

$$\frac{1}{44100} \quad (24)$$

to get our seconds between each sampled interval, we get 0.00002267573 seconds or **22.7 μ s**. A human blink takes **350000 μ s**! If we take this rate of sampling and apply it to a three minute long song, we would expect to obtain...

$$N = 180 \text{ s} \times 44,100 \text{ samples/s} = 7,938,000 \text{ discrete samples} \quad (25)$$

We can clearly see that we are just running into our "L being too large to compute problem" in a whole new way, because making a computer sample this many points and constructing them back into a waveform would take an unreasonable amount of processing power. However, nearly all of the everyday Fourier transform you see performed in software like this uses what is called a Fast Fourier Transform.

6 Discrete Fourier Transform

To start to obtain a function that can be computed easily by computers, let us start with our original Fourier transform.

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-2\pi i f t} dt \quad (26)$$

We can continue using our composite wave function $h(t)$ for this demonstration. We now understand that computers can't take direct integrals, as you would need to sample an infinite number of points. Instead, we can turn our continuous function of time into discrete indices, where we measure the wave at specific intervals of time instead, like how a clock reads in seconds. We can therefore rewrite t as...

$$t \rightarrow n \cdot T_s \quad (27)$$

Now that time isn't continuous anymore, using an integral is now impossible. What we need to do instead is switch to a Riemann Summation, which simply adds up the area of tightly packed, very

narrow rectangles under the graph. If we substitute our new discrete time, and change our integral to a Riemann Summation, we get...

$$X(f) = \sum_{n=0}^{N-1} x_n e^{-2\pi i f(nT_s)T_s} \quad (28)$$

This looks pretty scary, so let's clarify what some parts of this function do. x_n is the height of the n -th rectangular column, determined by the amplitude of the input wave, and T_s is just the fixed width of every rectangle, defined by your sampling interval. (In this case, 44100Hz.) Now we are nearly done, however, our frequency value is still continuous, which as we know by now is exactly what we need to eliminate. We can do this by slicing our range of frequencies into numerous frequency bins. This is done by defining f as...

$$f = k \cdot \frac{f_s}{N} \quad (29)$$

Substituting this new version of f now seems like a massive nightmare considering the number of variables in our exponent of e , however once we do this, an elegant simplification appears. Since frequency is represented as s^{-1} , and time is s^1 , when multiplied together they cancel out, turning our exponent...

$$-i2\pi \left(k \frac{f_s}{N} \right) (nT_s) \quad (30)$$

...into a very elegant mathematical ratio of...

$$-i \frac{2\pi}{N} kn \quad (31)$$

...and if we take this exponent and put it back into our now-finished Discrete Fourier Transform equation, we get...

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i \frac{2\pi}{N} kn} \quad (32)$$

This equation essentially tells the computer to take your set of N samples, multiply each sample by a specific complex co-ordinate on the complex plane to form the windings around the origin, and adding them all together, and the peaks in the output of this function show you the frequencies that are present in your wave. However, is there a way where you can make this process even more computationally efficient?

7 Fast Fourier Transform (Cooley-Tukey Radix-2)

To prevent our equation from looking even more cluttered than it will be, we can use a shorthand for our mathematical ratio we derived earlier, called the Twiddle Function. This is also used to reduce computing times. Let's define this...

$$W_N = e^{-i \frac{2\pi}{N}} \quad (33)$$

...so now our equation looks like this...

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn} \quad (34)$$

What we need to do now is assume that N is an even power of two, so that we can split our summation into two different piles: even-indexed ($n = 0, 2, 4, \dots$) and odd-indexed samples. ($n = 1, 3, 5, \dots$) To separate them, we define our new counting variable as $n = 2m$, to represent the even indices, and $n = 2m + 1$, therefore our counter m only needs to run up to $N/2 - 1$. We can now rewrite this summation by splitting it into two halves.

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} W_N^{k(2m)} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} W_N^{k(2m+1)} \quad (35)$$

Next, let's expand W_N^{2mk} for the even half and simplify.

$$W_N^{2mk} = \left(e^{-i\frac{2\pi}{N}}\right)^{2mk} = e^{-i\frac{2\pi}{(N/2)}mk} = W_{N/2}^{mk} \quad (36)$$

Let's do this with our odd half as well.

$$W_N^{k(2m+1)} = W_N^{2mk} \cdot W_N^k = W_{N/2}^{mk} \cdot W_N^k \quad (37)$$

Now, let's substitute these simplified twiddle factors back into our split summation equation...

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} W_{N/2}^{mk} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} \left(W_{N/2}^{mk} \cdot W_N^k\right) \quad (38)$$

We can now pull out W_N^k from our odd component...

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} W_{N/2}^{mk} + W_N^k \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} W_{N/2}^{mk} \quad (39)$$

...and we now have turned our single N-point DFT into 2 smaller N/2-point DFT's. Let's resimplify this by defining our even and odd functions as E_k and O_k respectively.

$$\sum_{m=0}^{\frac{N}{2}-1} x_{2m} W_{N/2}^{mk} = E_k \quad (40)$$

$$\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} W_{N/2}^{mk} = O_k \quad (41)$$

$$X_k = E_k + W_N^k O_k \quad (42)$$

Now we need to calculate the upper half of our frequencies, where the index is now $k + N/2$. Let's plug it in.

$$X_{k+\frac{N}{2}} = E_{k+\frac{N}{2}} + W_N^{k+\frac{N}{2}} O_{k+\frac{N}{2}} \quad (43)$$

Because E_k and O_k are periodic outputs of an N/2-point transform, they naturally repeat their values exactly in the second half

$$E_{k+\frac{N}{2}} = E_k \quad \text{and} \quad O_{k+\frac{N}{2}} = O_k \quad (44)$$

Let's now split the exponent of our twiddle factor...

$$W_N^{k+\frac{N}{2}} = W_N^k \cdot W_N^{\frac{N}{2}} \quad (45)$$

We can then simplify further by actually finding out what $W_N^{\frac{N}{2}}$ equals to by plugging it back into Euler's Identity...

$$W_N^{\frac{N}{2}} = e^{-i\frac{2\pi}{N}(\frac{N}{2})} = e^{-i\pi} = \cos(-\pi) + i\sin(-\pi) = -1 + 0 = -1 \quad (46)$$

Now that we know it equals to -1, the sign for W_N^k just becomes negative as it is multiplied by -1.

$$W_N^{k+\frac{N}{2}} = -W_N^k \quad (47)$$

By substituting these periodic identities back in, we get the two fundamental equations that define the Radix-2 FFT:

$$\begin{aligned} X_k &= E_k + W_N^k O_k \\ X_{k+\frac{N}{2}} &= E_k - W_N^k O_k \end{aligned}$$

The reason these 2 functions are so effective and fast is that these 2 functions are so similar, other than the sign in the middle, which means that the computer only loops from $k = 0$ to $N/2 - 1$. In every iteration of this loop, it finds the precalculated value of E_k , computes $W_N^k O_k$ (the most computationally difficult part), then all the computer needs to do is to subtract and add this value from E_k to get 2 separate final answers at practically the same time.

7.1 What does this actually mean?

To look at the big picture, we can see that the number of operations needed to analyse fully a 44.1 kHz audio file with 16 bit depth is decreased a whopping 99.5 percent, therefore on a modern CPU, tasks like this are extremely easy to execute. While the math might look complicated, the CPU manages to work with these sets of equations much easier than anything that is continuous.

Table 1: Computational Workload Comparison for 1 Second of 44.1 kHz Audio

Analysis Method	Math Per Block	Blocks Per Second	Total Operations Per Second
Standard DFT	~4.19 Million	21.5	90,177,536
FFT Butterfly	~22.5 Thousand	21.5	484,352

The reason that this process is now so efficient, is essentially because the operations per second done by a regular DFT is shown by $O(N^2)$, where N is our number of discrete samples, whereas we have obtained an FFT formula that instead decreases our operations per second to $O(N \log_2 N)$, which greatly increases the speed of our processing with the same amount of samples needed to be analysed. This is the core idea as to why FFT's are so incredibly efficient. However, this can not just be applied to audio and sound engineering, but every young child's favorite hobby, **video games**.

8 Wave Simulation in Sea of Thieves

In 2018, the British studio Rare, known for legendary games such as "Banjo-Kazooie" and "GoldenEye 007" released the award-winning game Sea of Thieves, a story driven, first-person, online multiplayer experience. Sea of Thieves is a game where players can set sail with a group of up to four friends, hunting for treasure, fighting skeleton armies, and battling sea monsters and other players. One of its key features and selling points is its in-game ocean and wave simulation, widely regarded as one of the most impressive and dynamic ocean representations in modern gaming. This is because instead of using old techniques like summing many sine waves together, or just tiling an ocean texture, which makes oceans look unrealistic and unimmersive, and takes a large amount of GPU (Graphical Processing Unit) resources, developers Rare use a FFT transform algorithm to define the ocean waves, making ocean simulation not only look stunning and dynamic, but also take a relatively low computing power to compute.



<https://www.spritesanddice.com/reviews/review-sea-thieves/>

This algorithm was first designed by Jerry Tessendorf in 2001, and has also been used in other games such as "Crysis" and "Assassin's Creed IV". Let's break down the math of this algorithm. ¹

¹Note: this video by Acerola looking at the process of implementing this whole program and how other video games use it is a wonderful video to watch, go check it out! <https://youtu.be/yPfaGLeUa7k?si=5ZWimYVwOP8OfwyM>

8.1 The Fourier wave simulation itself

Before we do any math, we need to designate the dimensions of the simulation grid, such as 512x512 or 256x256, using powers of 2 for the FFT to work at full efficiency, and we define this grid with set dimensions, like a 100m by 100m grid. Next, we generate our wave vectors, where every single point on our grid corresponds to a specific wave vector.

$$\vec{k} = (k_x, k_z) \quad (48)$$

This is the direction our wave is pointing, where its length is inversely proportional to the wavelength, therefore a large value of k corresponds to large ocean swells, and a small value of k corresponds to tiny surface ripples. Additionally, let's define a variable L as the largest possible wavelength that can physically be created by your current wind speed, where g is the value for gravity, and V is your velocity. It essentially acts as a size cap for your wave.

$$L = \frac{V^2}{g} \quad (49)$$

Since the vector contains both horizontal and vertical directions, we can define these separately as...

$$k_x = \frac{2\pi n}{L_x}, k_z = \frac{2\pi m}{L_z}, \quad (50)$$

where n is the row number and m is the column number that you are calculating, and L is our constant that we defined earlier. Now that we have our grid and our points defined, we need a mathematical model that tells us how much energy should exist at each frequency, based on real world physics. For this, we turn to our Phillips Spectrum formula.

$$P_h(\vec{k}) = A \frac{e^{-\frac{1}{(kL)^2}}}{k^4} \left| \hat{k} \cdot \hat{w} \right|^2 \quad (51)$$

Where the parameters are defined as follows:

$\vec{k} = (k_x, k_z)$ The 2D wave vector representing the spatial frequency and directional propagation of a wave component.

$k = |\vec{k}|$ The magnitude of the wave vector, where $k = \sqrt{k_x^2 + k_z^2}$.

$\hat{k}$ The normalized direction vector of the wave, calculated as $\frac{\vec{k}}{k}$.

A A global numeric scaling constant that dictates the maximum possible amplitude of the waves.

$\hat{w}$ A normalized unit vector pointing in the direction of the wind.

The reciprocal of K^4 represents the gravity decay, where large values of k rapidly lose energy, penalizing high frequencies. $\left| \hat{k} \cdot \hat{w} \right|^2$ is our wave dot product. If a wave component is traveling perfectly parallel to the wind direction, the dot product equals 1 (or -1). When squared, it becomes 1, allowing that wave component to receive maximum energy from the wind. If a wave is trying to travel at a right angle (90°) to the wind, the dot product drops straight to 0. When squared, it completely zeroes out the equation for that frequency. This ensures that the simulated ocean waves mostly roll in the direction of the wind, rather than traveling chaotically in random directions. If we use just this function, our ocean would look like a very sterile, boring geometric pattern. We need some randomness. We can do this by multiplying the root of our Phillips Spectrum formula by a set of random complex numbers drawn from a gaussian bell curve.

$$h_0(\vec{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\vec{k})} \quad (52)$$

You take the square root of the Phillips function because of a fundamental law in wave physics: the energy of a wave is proportional to the square of its height/amplitude, therefore the value we want to vary here is the energy of the wave. The reason that we use a gaussian bell curve specifically is

essentially because it heavily values values in between -1 and 1, and uses numbers outside of that range as outliers. This provides a sweet spot so that our waves aren't too chaotic and extreme, but also are not too boring and regular.

Now that we have our initial static frequency grid, holding a large, realistic combination of wave amplitudes and phases, but now we need to make it vary over time. To animate the static wave field over time, the simulation utilizes the deep-water dispersion relation. This hydrodynamic equation dictates the angular frequency ω (the temporal propagation speed) of a wave component as a function of its spatial scale:

$$\omega = \sqrt{gk} \quad (53)$$

Where:

ω represents the angular frequency of the wave component, measured in radians per second (rad/s).

g represents the acceleration due to gravity ($\approx 9.81 \text{ m/s}^2$).

k represents the magnitude of the 2D wave vector, where $k = |\vec{k}| = \sqrt{k_x^2 + k_z^2}$.

One fundamental equation in fluid dynamics is $\omega = \sqrt{gk \tanh(kh)}$, where h represents the depth of the ocean, but the reason that this is not included in our angular frequency equation, is that the sea depth in Sea of Thieves is assumed to be very large, therefore as h tends to infinity, $\tanh$ is closer and closer to one, therefore we ignore it, reducing our calculation time. Now that we have defined ω , let's take our static equation and multiply it by our spinning complex exponentials to move the waves backwards and forwards.

$$h(\vec{k}, t) = h_0(\vec{k})e^{i\omega t} + h_0^*(-\vec{k})e^{-i\omega t} \quad (54)$$

We now have a grid of all our ocean frequencies during an exact point in time. If we only simulate vertical height, our waves will look just like rolling, regular sinusoidal hills. We can create more choppiness using some horizontal displacement.

$$h_x(\vec{k}, t) = -i\frac{k_x}{k}h(\vec{k}, t), \quad h_z(\vec{k}, t) = -i\frac{k_z}{k}h(\vec{k}, t) \quad (55)$$

What we have done is create two extra frequency grids, by multiplying our height data by our normalized wave vectors, meaning that we now have three separate frequency grids: Vertical Height, Horizontal X-shift, and Horizontal Z-shift.

Now that we have our 3 Fourier transform functions, we can turn them back into computer mesh data using an inverse Fourier transform algorithm used here, for all 3 co-ordinate components separately.

$$y(x, z, t) = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} h(\vec{k}, t) \exp\left(2\pi i \left(\frac{nx}{N} + \frac{mz}{M}\right)\right) \quad (56)$$

$$\Delta x(x, z, t) = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} h_x(\vec{k}, t) \exp\left(2\pi i \left(\frac{nx}{N} + \frac{mz}{M}\right)\right) \quad (57)$$

$$\Delta z(x, z, t) = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} h_z(\vec{k}, t) \exp\left(2\pi i \left(\frac{nx}{N} + \frac{mz}{M}\right)\right) \quad (58)$$

Where the output values y , Δx , and Δz represent the real-valued spatial coordinates used to displace the flat ocean vertex mesh at the horizontal grid intersections (x, z) . Finally, the graphics card renders a completely flat 3D mesh grid of vertices. The vertex shader looks up the coordinates in our freshly baked iFFT maps and physically offsets each point:

$$\vec{X}_{\text{final}} = \vec{X}_{\text{flat}} + \begin{pmatrix} x\text{-displacement} \\ y\text{-height} \\ z\text{-displacement} \end{pmatrix}$$

Because the x and z displacements push the geometry horizontally toward the peaks while the y displacement lifts them up, the flat mesh naturally bunches together at the top, forming the iconic,

cinematic wave crests that make virtual oceans look breathtaking after we overlay our water texture on top of our sets of 3D, time-varied points. Now that the heavy lifting of the math is finished, we can enhance the waves realism by determining where two points of two different waves horizontal displacements cross each-other, in reality meaning where two waves crash together, and when this happens, you can blend a white bubbly texture onto the exact spot where the waves meet, simulating them interfering with eachother, and making them look like they are actually interacting.

9 Conclusion

By defining how the original Fourier transform equation is represented, and then reconstructing the Fourier transform from the ground up into a series of equations using Euler's formula and some mathematical rearrangement, before then explaining a way to optimise Fourier transform for audio processing on a computer, I have been able to create a Fourier transform that is reproducible in Desmos without using any complex modes, and explain how Fourier transform can be optimised for use in computing. Additionally, I have showed how to derive a way to create a Discrete Fourier Transform that can be processed by computers, and how it can be further optimised into a Fast Fourier Transform algorithm for use in video games and other digital applications.

10 Bibliography

3Blue1Brown's video was an incredible way of visualising Fourier transform, and the main portion of my report used this video as a source. Additionally, Acerola's video showing his journey with trying to synthesise waves in Unity, including the math used to do so, provided a valuable insight on the process which I reflected upon in the paper, and deserves to be checked out. ChatGPT and Google Gemini were used when I needed specific concepts explained in a different way, producing some of the $\text{\LaTeX}$ code used in this document, and helped with sense checking and grammatical checking this report, and helped with the derivation of the harder concepts especially, with double checking from relevant Wikipedia articles, such as: https://en.wikipedia.org/wiki/Fast_Fourier_transform https://en.wikipedia.org/wiki/CoolyTukey_FFT_algorithm https://en.wikipedia.org/wiki/NyquistShannon_sampling_theorem Other articles used for fragments of information or smaller topics of interest are detailed in my citations along with these main sources detailed in this bibliography.

References

- [1] Sanderson, G. [3Blue1Brown]. (2018, Jan 26). *But what is the Fourier Transform? A visual introduction.* [Video]. YouTube. <https://youtu.be/spUNpyF58BY>
- [2] Acerola. (2023, Nov 2). *I Put Sea of Thieves Waves In My Game.* [Video]. YouTube. <https://youtu.be/yPfaGLeUa7k>
- [3] Daniel M. (2017, Mar 20). *Energy of a Wave.* [Video]. YouTube. <https://www.youtube.com/watch?v=H3tIX0inH4I&t=12s>
- [4] OpenAI. (2024). *ChatGPT* (GPT-4) / Google. (2024). *Gemini*. [Large Language Models]. Used for $\text{\LaTeX}$ formatting, grammar checks, and mathematical sense-checking.
- [5] Encyclopedia Britannica. (n.d.). *Euler's formula*. Britannica. <https://www.britannica.com/science/Eulers-formula>
- [6] Audacity Team. (n.d.). *Sample Rates*. Audacity Manual. https://manual.audacityteam.org/man/sample_rates.html
- [7] Wikipedia contributors. (n.d.). *Aliasing*. Wikipedia. <https://en.wikipedia.org/wiki/Aliasing>
- [8] Wikipedia contributors. (n.d.). *Cooley–Tukey FFT algorithm*. Wikipedia. https://en.wikipedia.org/wiki/CoolyTukey_FFT_algorithm

- [9] Wikipedia contributors. (n.d.). *Discrete Fourier transform*. Wikipedia. https://en.wikipedia.org/wiki/Discrete_Fourier_transform
- [10] Wikipedia contributors. (n.d.). *Fast Fourier transform*. Wikipedia. https://en.wikipedia.org/wiki/Fast_Fourier_transform
- [11] Wikipedia contributors. (n.d.). *List of video games developed by Rare*. Wikipedia. https://en.wikipedia.org/wiki/List_of_video_games_developed_by_Rare
- [12] Wikipedia contributors. (n.d.). *Nyquist–Shannon sampling theorem*. Wikipedia. https://en.wikipedia.org/wiki/NyquistShannon_sampling_theorem

11 Reflective Log

While I was attempting to visually demonstrate the output of a Fourier transform, I had a large difficulty working with the limitations of Desmos, however through some testing and help online, I have devised a step-by-step method of deriving the equations needed to input into Desmos, which I have otherwise not been able to find on the internet, and since I had the original idea of just the deriving Fourier transform, I have also come across numerous other ideas such as Fast Fourier Transform and its use in the video game "Sea of Thieves". I used 3Blue1Brown's video as the base for the process of derivation of the original Fourier transform equation, and I have learnt a large amount of how the Fourier transform works using their various visual demonstrations. While this topic is incredibly advanced (something typically taught in the second or third year of university,) I believe that this project has enabled me to explain at least the general overview of how a Fourier transform works, and this project demonstrates my ability to tackle advanced-level mathematical concepts, which I found challenged me beyond the standard A-Level curriculum. I have also discovered that writing reports is something that I find fun, and is something I will very much enjoy undergoing in university. One thing I may regret is not getting my information from even more sources, however I think 3Blue1Brown's video on the topic is incredibly comprehensive and served as the bulk of my inspiration and reasoning, whereas Wikipedia and sources linked on the Wikipedia pages ended up corroborating the information disclosed by AI software. The link to my Desmos project where the group of Fourier functions can be observed is here: <https://www.desmos.com/calculator/cguid0cftk>