Kantonsschule im Lee
HS 2022/23

Programming Audio Plugins

Maturitätsarbeit

Charly Finn Schubertrügmer

supervised by Thomas Graf and submitted on January 9, 2023

Abstract

The aim of this work is to provide a comprehensive explanation of how to code an audio plugin in the audio application framework JUCE as well an introduction to the mathematics involved. In particular lays the groundwork to code a synthesizer that represents all ”reasonable” sound waves on a line through an infinite dimensional space filling curve and an effect plugin that implements the Volterra series to unify different nonlinear time invariant effects.

Preface

I have always been interested in both music and mathematics. I have been playing the violin since I was five and the hammered dulcimer since I was eight. Back when I was eleven years old and in my last year of primary school the game ”Geometry Dash” got me into the EDM genre, and I wanted to try my hand at creating my own electronic music. I was given the program FL Studio for my twelfth birthday and started playing around. But it soon became clear that creating sounds I was happy with was a difficult task. I couldn’t get my synths to growl, whine or whatever else I was trying to do. It simply didn’t sound like what the professionals produced. So, I stopped trying. But as I learned more mathematics, especially during COVID, where time was abundant, my fascination of audio processing reignited. I wanted to know the inner workings of the synthesizers and effects I had experimented with years earlier. Maybe now I could finally design the sounds I wanted to. Maybe now I could learn this skill that had eluded me. I had also started to develop my own ideas for audio plugins that I wanted to create. I did some research and put that idea to the side as a candidate for the matura-topic, and here we are.

I would like to take this opportunity to thank Jakob Traber for checking parts of the text for comprehensibility, he has been an enormous help in improving the readability of the text. I would also like to thank Lukas Nieswand and Josia John for the support I have received from them when I was learning C++ and similarly my supervisor for helping me with LaTeXand my work more generally though I regret not asking for help more often. I would like to thank Josia John especially for also creating the images 2.9 and 2.10 when time was running short. I need to thank my dad for providing the tools and materials used to create this booklet. Finally, I would like to thank all my friends and both my parents for the emotional support they have provided throughout the years.

Contents

Chapter 1

Introduction

Nowadays most of the audio you hear online underwent some editing process on a computer, be it splicing together different recordings, the reduction of noise in a recording, the balancing of loudness throughout different times in the recording or a multitude of other possible tweaks. Music is especially digitized with each instrument often being recorded separately and undergoing a separate processing chain if it is being recorded at all. Many instruments are being replaced by software synthesizers that sound increasingly similar to their real-world counterparts. But synthesizers can also be used to create completely new sounds, opening the door to a world of possibilities we could never before imagine. Audio effects have similarly been used creatively. One prominent example is the controversial use of strong Autotune to create a robotic sounding voice. And all this stands on the shoulders of the software behind it all.

Most audio production takes place in a so called digital audio workstation (DAW) in which multiple recordings can be arranged, layered and edited and which typically include features to play back virtual instruments off of a virtual piano roll. DAWs usually also allow the use of third party software called plugins to be used as both synthesizers and effects in addition to those included in the installation of the DAW itself. This gave rise to a market of often pricey audio plugins which makes it all the more valuable to know how to program one yourself.

This work will go through the process of programming such an audio plugin.

Chapter 2

Mathematical Groundwork

2.1 Prerequisites

Below is a list of topics the reader should be familiar with before reading this work along with references to resources covering them:

- Complex number basics
- Linear algebra fundamentals [9]
- Taylor series [10]

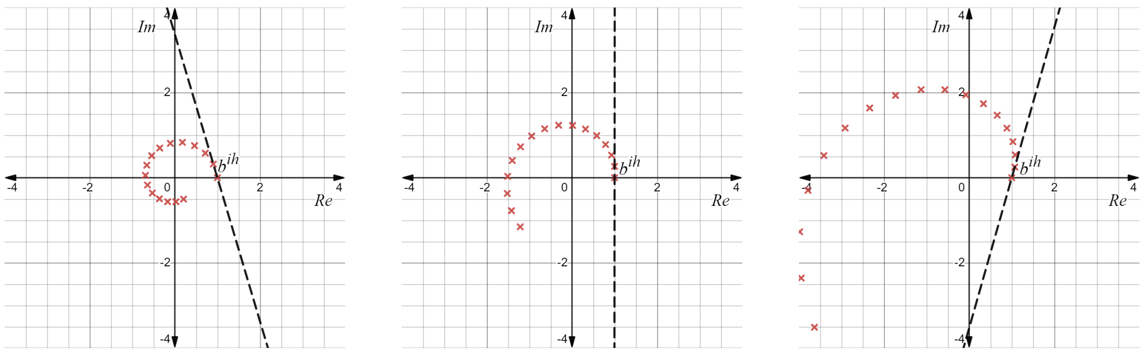
2.2 Complex exponential

The concept of a complex valued exponent will appear a lot later so I am including an explanation here though the definition may take time and practice to properly digest.

So far, we have extended the domain of the function $f(t) = b^t$ (for positive real b) from the positive integers to the the real numbers using the property that $b^{s+t} = b^s \cdot b^t$ ($\Rightarrow (b^t)^n = b^{nt}$ for integer n) and the assertions that b^t is a continuous function (and that it is real valued for real t). These properties alone are not enough to uniquely define b^t for complex t. To see this, observe that if b^{ih} is defined for some $h \in \mathbb{R}$ we can derive definitions for b^{x+inh}, $n \in \mathbb{N}$, $x \in \mathbb{R}$. Now define b^{ih} to approach $b^0 = 1$ as $h \to 0$ from any direction in the complex plane and any velocity of your choosing. No matter which direction and velocity is chosen, this will lead to a continuous extension of b^t to the complex numbers (see 2.1).

To remedy this we will add the further constraint that b^t is differentiable (in fact we only need it to be differentiable at 0). Since the derivative of b^t is $\ln b$ at $t = 0$, $\lim_{\substack{h\to 0\\ h\in\mathbb{R}}} \frac{b^{0+ih}-b^0}{ih} = \ln(b) \implies b^{0+ih} - b^0 \to ih\ln(b)$ as $h \to 0$, $h \in \mathbb{R}$, i.e. b^{ih} approaches $b^0 = 1$ from the purely imaginary direction with velocity $\ln(b)$ as $h \to 0$ as in figure 2.1b. This in conjunction with the other properties uniquely defines b^{it} to be $(\cos t + i \sin t)^{\ln b}$ for real t. The graph of b^{it} is a helix through the complex plane around the t-axis with angular frequency $\ln b$ (see 2.2). Important special cases are e^{it} which has angular frequency 1 and $e^{2\pi it}$ which has angular frequency 2π i.e. it has frequency 1. This definition also retains the property that $\frac{d}{dt}e^{\omega t} = \omega e^{\omega t}$.

c^t for complex $c = |c|e^{\varphi i}$ is usually defined as $|c|^t e^{\varphi it}$ such that $-\pi < \varphi \leq \pi$.



(a) (b) (c)

Figure 2.1: Red: b^{ih} and consequent definitions for b^{inh}. Dashed: The direction of approach.

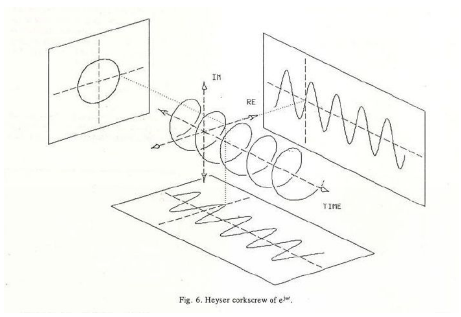


Figure 2.2: The graph of e^{it} is a helix. [4]

2.3 Linear operators and integral transforms

Let $M\{x(\tau_M)\}(t)$ be an operator (e.g. an audio effect), $x(t)$, $y(t)$ be complex valued functions (e.g. sound waves) and $a \in \mathbb{C}$ a constant:
$M\{x(\tau_M)\}(t)$ is said to be linear if

$$M\{x(\tau_M) + y(\tau_M)\}(t) = M\{x(\tau_M)\}(t) + M\{y(\tau_M)\}(t) \tag{2.3.1}$$

and

$$M\{a \cdot x(\tau_M)\}(t) = a \cdot M\{x(\tau_M)\}(t) \tag{2.3.2}$$

that is, applying M after adding x and y has the same effect as applying M to x and y individually, then adding $Mx + My$.

Given an operator M and a function x to apply M to, we can make use of the linearity of M by writing x as a sum of "spike functions" centered at different points in time and applying M to each "spike" individually first before summing them up again. Define $\delta[t]$ to be such a spike function that is 1 at $t = 0$ and 0 everywhere else. Spikes centered at different times τ are generated by the expression $\delta[t - \tau]$.

$$x(t) = \sum_{\tau} x(\tau) \cdot \delta[t - \tau] \tag{2.3.3}$$

thus:

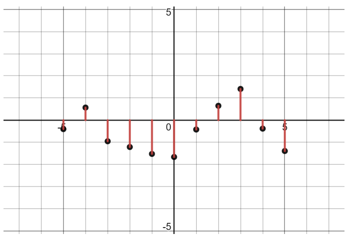


Figure 2.3: $x(t)$ can be written as a sum of "spike functions".

$$\begin{aligned}
M\{x(\tau_M)\}(t) &= M\left\{\sum_\tau x(\tau)\cdot\delta[\tau_M-\tau]\right\}(t) \\
&= \sum_\tau M\{x(\tau)\cdot\delta[\tau_M-\tau]\}(t) \\
&= \sum_\tau x(\tau)\cdot M\{\delta[\tau_M-\tau]\}(t)
\end{aligned}$$

defining $M\{\delta[\tau_M-\tau]\}(t) := m(\tau,t)$:

$$= \sum_\tau x(\tau)\cdot m(\tau,t) \tag{2.3.4}$$

m is called the kernel function of the transform. You may recognize this as the formula for matrix multiplication because that is exactly what it is.

This expression works fine for discrete functions but it has the downside that, if the sampling period changes by a factor of T, all else being equal, the resulting amplitude will be scaled by $1/T$. To keep the amplitude constant we multiply the expression by T:

$$\sum_\tau x(\tau)\cdot m(\tau,t)\cdot T \tag{2.3.5}$$

And as T tends to 0 the expression tends to the integral:

$$\int_\tau x(\tau)\cdot m(\tau,t)d\tau \tag{2.3.6}$$

This type of expression is termed an integral transform.

2.3.1 Composition

The composition $L = MN$ of two linear operators M, N is itself a linear operator and its kernel function is given by:

$$l(\tau,t) = \sum_f m(f,t)\cdot n(\tau,f) \tag{2.3.7}$$

Where l, m, n are the kernel functions of L, M, N respectively.

Or for the continuous case:

$$l(\tau,t) = \int_f m(f,t)\cdot n(\tau,f)df \tag{2.3.8}$$

Derivation for the discrete case:

$$\begin{aligned}
L\{x(\tau_L)\}(t) &= \sum_f m(f,t)\sum_\tau x(\tau)\cdot n(\tau,f) \\
&= \sum_\tau x(\tau)\underbrace{\sum_f m(f,t)\cdot n(\tau,f)}_{\text{kernel function}}
\end{aligned} \tag{2.3.9}$$

The derivation for the continuous case is analogous.

2.4 Linear time invariant systems and convolution

Again, let $M\{x(\tau_M)\}(t)$ be an operator, $x(t)$ be a complex valued function and $\tau_M, t, t_0 \in \mathbb{R}$ (or more generally elements of some group under addition); $M\{x(\tau_M)\}(t)$ is said to be time invariant if

$$M\{x(\tau_M + t_0)\}(t) = M\{x(\tau_M)\}(t + t_0) \tag{2.4.1}$$

that is, applying M after shifting x in time by some t_0 has the same effect as applying M first, then the shift in time.

Examples of operators that are both linear and time invariant (LTI) include scaling, time shifting itself, differentiation (and differencing), and real world examples like equalizers, blur (weighted averaging over time) and reverb.

When M is time invariant, aside from being linear, the kernel function can be expressed as a single variable function:

$$M\{\delta[\tau_M - \tau]\}(t) = M\{\delta[\tau_M]\}(t - \tau) := m(t - \tau) \tag{2.4.2}$$

$m(t)$ is also called the system's impulse response because it represents the output, i.e. response, of the system given an impulse, i.e. spike function, centered at 0.

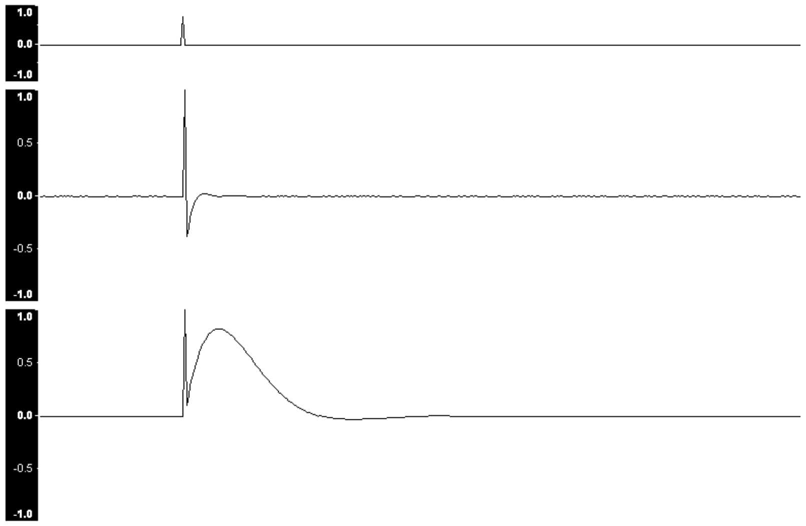


Figure 2.4: The Impulse response from a simple audio system. Showing, from top to bottom, the original impulse, the response after high frequency boosting, and the response after low frequency boosting.[5]

Replacing $m(\tau, t)$ with $m(t - \tau)$ in 2.3.4 we get:

$$M\{x(\tau_M)\}(t) = \sum_{\tau} x(\tau) \cdot m(t - \tau) \tag{2.4.3}$$

The last expression is called the convolution of x with m and is denoted $x * m$. By substitution one can show that $x * m = m * x$ is commutative.

Analogously for continuous applications:

$$\int_{\tau} x(\tau) \cdot m(t - \tau) d\tau \tag{2.4.4}$$

2.4.1 Eigenfunctions

We might be interested in finding eigenfunctions for LTI operators. For this, it suffices to find a set of eigenfunctions of all time shift operators. Let $x(t)$ denote one such eigenfunction. Then:

$$x(0+\tau) = \lambda_\tau \cdot x(0) \tag{2.4.5}$$

applying a time shift of τ m times yields:

$$x(m \cdot \tau) = \lambda_\tau^m \cdot x(0) \tag{2.4.6}$$

Put in words, the eigenfunctions of the time shift operator are exponential functions and can be expressed as Ae^{st} for $A, s \in \mathbb{C}$. These functions are also eigenfunctions of all other LTI operators: Let m denote the impulse response of the LTI operator[8]:

$$\begin{aligned} &\int_\tau Ae^{s\tau} \cdot m(t-\tau)d\tau \\ =&\int_\tau Ae^{s(t-\tau)} \cdot m(\tau)d\tau \\ =&\int_\tau Ae^{st} \cdot e^{-s\tau} \cdot m(\tau)d\tau \\ =&\underbrace{Ae^{st}}_{\text{Input}} \underbrace{\int_\tau e^{-s\tau} \cdot m(\tau)d\tau}_{\text{Scalar}} \end{aligned} \tag{2.4.7}$$

2.5 Fourier transform

Knowing that exponential functions are eigenfunctions of LTI operators we might be interested in decomposing functions into a sum of exponential functions. This will make calculations easier as will become apparent in section 2.5.1. One such decomposition is the Fourier transform, which concerns itself with decomposing functions into periodic exponentials of frequency f i.e. complex exponentials of the form $e^{2\pi ift}$.
The Fourier transform of a function $x(t)$ is defined as the function $\hat{x}(f)$ such that:

$$\int_{-\infty}^{\infty} \hat{x}(f) \cdot e^{2\pi ift} df = x(t) \tag{2.5.1}$$

That is, for each frequency f, $\hat{x}(f)$ returns the amplitude of the complex exponential of that frequency "contained" in x such that summing over all complex exponentials with their respective amplitudes returns x. The process of summing over the complex exponentials as above is called the inverse Fourier transform.

If x is periodic then it will only have frequency components at integer multiples of the fundamental frequency $f_0 = \frac{1}{T}$. In this case it makes sense to define $\hat{x}$ as the Fourier series coefficients:

$$\sum_{n=-\infty}^{\infty} \hat{x}[n] \cdot e^{2\pi int/T} = x(t) \tag{2.5.2}$$

(this definition is not equivalent to 2.5.1)

If we were to define a discrete Fourier transform (DFT) to act on such a periodic function which was sampled at N discrete points in *time*, for the DFT to be unique and invertible, the inverse discrete Fourier transform (IDFT) must also act on N discrete points in *frequency*. This follows from the properties of linear transformations. We thus define the the IDFT $\hat{x}$ of x via a truncated Fourier series with period N:

$$\sum_{f=1}^{N} \hat{x}[f] \cdot e^{2\pi ift/N} = x[t] \tag{2.5.3}$$

Sometimes the IDFT is scaled by a factor of $\frac{1}{\sqrt{N}}$ so as to make it a unitary operator, and we will use this definition moving forward:

$$\frac{1}{\sqrt{N}} \sum_{f=1}^{N} \hat{x}[f] \cdot e^{2\pi ift/N} = x[t] \tag{2.5.4}$$

it is worth noting that the continuous inverse Fourier transform 2.5.1 is also unitary.

One could also choose to center the summation around 0 for odd N so as to make it possible for the imaginary components of the positive and negative frequency components to cancel:

$$\frac{1}{\sqrt{N}} \sum_{f=-(N-1)/2}^{(N-1)/2} \hat{x}[f] \cdot e^{2\pi ift/N} = x[t] \tag{2.5.5}$$

To derive a closed form expression for the Fourier transform recall that a matrix is unitary when its inverse equals its conjugate transpose. Analogously, a linear operator M (on L^2) is unitary when its kernel function m is related to the kernel function of its inverse n by: $m(\tau, t) = \overline{n(t, \tau)}$, where the bar denotes complex conjugation. If the inverse Fourier transform is indeed unitary then *its* inverse, i.e. the Fourier transform itself, will be given by:

$$\hat{x}(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-2\pi ift} dt \tag{2.5.6}$$

(Indeed, this is the eigenvalue of the convolution operator from 2.4.7.) Similarly, the discrete Fourier transform would be:

$$\hat{x}[f] = \frac{1}{\sqrt{N}} \sum_{t=1}^{N} x[t] \cdot e^{-2\pi ift/N} \tag{2.5.7}$$

Let us verify that this is correct in the discrete case (with $N > 1$) using the following identity on the way:
For $N > 1$:

$$\Sigma = \sum_{f=1}^{N} e^{2\pi if/N} = 0 \tag{2.5.8}$$

To prove this, notice that the set $S = \{e^{2\pi in/N} | n \in \mathbb{N}\}$ of Nth roots of unity is invariant with respect to elementwise multiplication with one of its elements; multiplication by some $z \in S$ is one-to-one, because it is invertible, and onto, because the product of two Nth roots of unity is another Nth root of unity. This means that Σ must satisfy: $z\Sigma = \Sigma$. Now, for $N > 1$, there exists a $z \in S, z \neq 1$ and thus the equation $z\Sigma = \Sigma$ can only be satisfied if $\Sigma = 0$.

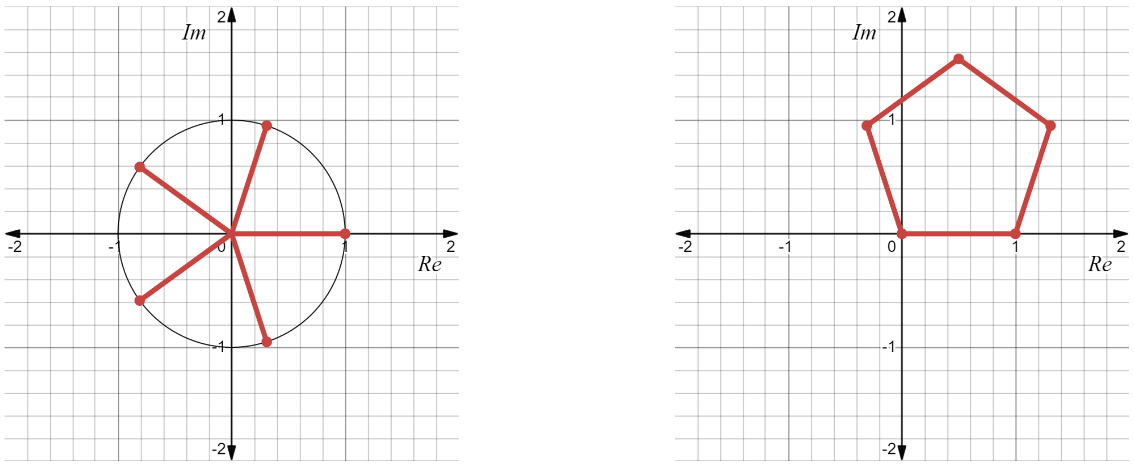


Figure 2.5: Visualization of 2.5.8. Left: the fifth roots of unity. Right: the sum of the fifth roots of unity.

Notice also that, for $t \in \mathbb{N}$, S^t is also a set of roots of unity which, as long as N does not divide t, has strictly more than one element, so 2.5.8 applies to:

$$\sum_{f=1}^{N} (e^{2\pi i f/N})^t = \sum_{f=1}^{N} e^{2\pi i f t/N} = 0 \tag{2.5.9}$$

When $t \nmid N$.

Now we need to show that the composition of the inverse Fourier transform with 2.5.7 is the identity operator:
By 2.3.7, the kernel function of the composition of 2.5.7 and the inverse Fourier transform 2.5.4 is given by:

$$\begin{aligned} &\sum_{f=1}^{N} \frac{1}{\sqrt{N}} e^{2\pi i f t/N} \cdot \frac{1}{\sqrt{N}} e^{-2\pi i f \tau/N} \\ =&\frac{1}{N} \sum_{f=1}^{N} e^{2\pi i f(t-\tau)/N} \\ =&\begin{cases} 1, & \tau = t \\ 0, & \tau \neq t \end{cases} \end{aligned} \tag{2.5.10}$$

Which is the kernel function of the identity operator.

2.5.1 Convolution theorem

Using the Fourier transform, convolution of functions x and m can be written as:

$$\begin{aligned} x * m &= \int_\tau x(\tau) \cdot m(t-\tau) d\tau \\ &= \int_f \underbrace{\hat{x}(f) \cdot e^{2\pi i f t}}_{\text{eigenfunction}} \underbrace{\overbrace{\int_\tau e^{-2\pi i f \tau} \cdot m(\tau) d\tau}^{\hat{m}(f)}}_{\text{eigenvalue from 2.4.7}} df \\ &= \int_f \hat{x}(f) \cdot \hat{m}(f) \cdot e^{2\pi i f t} df \end{aligned} \tag{2.5.11}$$

This result is known as the convolution theorem.

2.5.2 Physical meaning

If the function $x(t)$ to be Fourier transformed is purely real, then $\hat{x}(f) = \overline{\hat{x}(-f)}$ so that the imaginary parts of the positive and negative frequency components cancel, effectively reducing the Fourier transform to a decomposition into shifted cosine waves. This means, the Fourier transform can be interpreted as decomposing x into a sum of so called harmonic oscillations.

A harmonic oscillator is a system that exerts a restoring force proportional to its displacement:

$$F = -kx \tag{2.5.12}$$

From which the potential energy follows:

$$E_{pot} = \int_0^x ks^2 ds = \frac{1}{2} kx^2 \tag{2.5.13}$$

A spring, for instance, is modeled as a harmonic oscillator. The mass at the end of a spring has the kinetic energy:

$$E_{kin} = \frac{1}{2} mv^2 \tag{2.5.14}$$

Consider the coordinate space (x, y) with $x = x$ and $y = \frac{\sqrt{m}}{\sqrt{k}} v$. The coordinates are chosen to be proportional to the square roots of the potential and kinetic energies respectively. For a given point (x, y) in that space, its derivative in y direction is:

$$\begin{aligned} \frac{dy}{dt} &= \frac{\sqrt{m}}{\sqrt{k}} \cdot \frac{dv}{dt} \\ &= \frac{\sqrt{m}}{\sqrt{k}} \cdot \frac{-kx}{m} \\ &= \frac{-\sqrt{k}}{\sqrt{m}} x \end{aligned} \tag{2.5.15}$$

and similarly its derivative in x direction:

$$\frac{dx}{dt} = v = \frac{\sqrt{k}}{\sqrt{m}} y \tag{2.5.16}$$

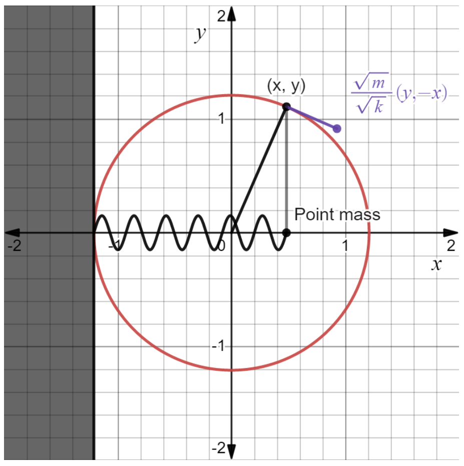


Figure 2.6: The point moves around a circular trajectory.

That is, for a point (x, y) in this space, its direction vector is given by the scaled $90°$ rotation $\frac{\sqrt{k}}{\sqrt{m}}(y, -x)$ and because of this the point moves along a trajectory of a circle with velocity $\frac{\sqrt{k}}{\sqrt{m}}r = \frac{\sqrt{k}}{\sqrt{m}}x_{\max}$ (where r is the radius of the circle) or angular frequency $\frac{\sqrt{k}}{\sqrt{m}}$ (independent of r or the energy in the system!). When regarding only the displacement axis, the mass at the end of the spring moves in a shifted and scaled cosine wave.

2.5.3 Fast Fourier transform

Not only is the Fourier transform useful for us humans to understand what frequencies are present in a sound, it can also be computed quickly (in $O(n \log n)$ time) which, in conjunction with the convolution theorem, makes it useful to speed up calculations. (A naive implementation of convolution would take $O(n^2)$ time.) This, and other reasons, give the Fourier transform applications in theoretical computer science, image processing, statistics and more aside from the familiar applications in signal processing (audio, radio etc.) and physics.

The algorithm that performs the Fourier transform in $O(n \log n)$ is called the fast Fourier transform (FFT). Here is a reference to a video explaining a simple form of the algorithm: [11].

2.5.4 Aliasing

In a computer signals are stored as samples at discrete points in time. The density of these points in time are given by the sampling frequency (or sample rate) f_s. As one would expect, some information is lost going from a continuous sound wave to a sampled one. For example the complex exponential $e^{2\pi i t f_s/2}$ of frequency $\frac{f_s}{2}$ would be sampled the same as its complex conjugate $e^{-2\pi i t f_s/2}$. $e^{2\pi i t f_s 2}$ is thus called an alias of $e^{-2\pi i t f_s/2}$. Higher frequencies $\frac{f_s}{2} + \Delta f$ also have aliases: $e^{2\pi i (f_s/2 + \Delta f)t} = e^{2\pi i t f_s/2} \cdot e^{2\pi i \Delta f t}$ is sampled the same as $e^{-2\pi i t f_s/2} \cdot e^{2\pi i \Delta f t} = e^{2\pi i(-f_s/2 + \Delta f)t}$. We must thus make an assumption about which alias should be used when our sampled sound is played back.

In 2.5.5 one such assumption is made, namely that the highest absolute frequency component present during sampling is $\frac{1}{N}\frac{N}{2} = \frac{1}{2}$ *per sample.* To calculate the

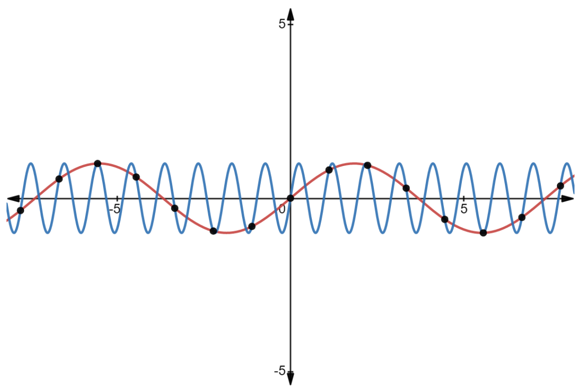


Figure 2.7: The red and blue sine waves are aliases of one another; they get sampled the same

frequency in Hz we multiply by the sampling frequency f_s given in $\frac{\text{samples}}{\text{second}}$ to get $\frac{f_s}{2}$.

If higher frequencies were present during sampling then they would be reconstructed as a low frequency alias in a phenomenon known as aliasing. It is therefore recommended to filter out the high frequencies when sampling when possible. This is either done by multiplying the frequency spectrum $\hat{x}$ by the indicator function for the range $[-\frac{f_s}{2}, \frac{f_s}{2}]$ or equivalently, by the convolution theorem, by convolving with its Fourier transform; the function $\operatorname{sinc}(\pi f_s t) = \frac{\sin(\pi f_s t)}{\pi f_s t}$.

2.6 Volterra series

In this chapter we will focus on time invariant systems without requiring they be linear. Given a function $x(t)$ with period T, a time invariant operator is guaranteed to preserve that period since $x(t+T) = x(t)$ and therefore $M\{x(\tau)\}(t+T) = M\{x(\tau+T)\}(t) = M\{x(\tau)\}(t)$. Say $x(t) = a_1 e^{f_1 t} + a_2 e^{f_2 t} + ...$, then its period T is the least common multiple of the individual components' periods $T = \operatorname{lcm}(T_1, T_2, ...)$ where $T_n = 1/f_n$. Phrased in terms of frequencies; the frequency f of $x(t)$ is the greatest common divisor of the component frequencies $f = \gcd(f_1, f_2, ...)$. This means that M can generate new frequency components of frequencies at integer multiples of f. This phenomenon is called intermodulation distortion and the new frequency components are called intermodulation products.

To approximate or describe a nonlinear but time invariant operator M one can employ a so called Volterra series. The Volterra series is essentially a time invariant, infinite dimensional Taylor series around $x_0 = 0$. To understand this, we need to define derivatives for multidimensional functions. Here I will introduce the Fréchet derivative to generalize the derivative of real valued functions of one real variable. The Fréchet derivative D of a function (or operator) $m : \boldsymbol{X} \to \boldsymbol{Y}$ when it exists, is defined as the bounded linear operator A, such that:

$$\lim_{||h||\to 0} \frac{||m(x+h) - m(x) - Ah||_{\boldsymbol{Y}}}{||h||_{\boldsymbol{X}}} = 0 \tag{2.6.1}$$

This means that the Fréchet derivative Dm of m is a function from X to the space of linear functions from $\boldsymbol{X}$ to $\boldsymbol{Y}$, denoted $\boldsymbol{L}(\boldsymbol{X}, \boldsymbol{Y})$: $Dm : \boldsymbol{X} \to \boldsymbol{L}(\boldsymbol{X}, \boldsymbol{Y})$. So, $Dm(x_0)(x_1)$, which is linear in all arguments except x_0 . Higher order derivatives are then of the form $D^n m(x_0)(x_1)...(x_{n-1})(x_n)$ and are linear in all arguments except x_0

Analogous to a Taylor series for one dimensional functions, we wish to approximate $m(x)$ at $x_0 = 0$. The zeroth order approximation, as before, is $m(x_0)$, the first order approximation is $m(x_0) + Dm(x_0)(x - x_0)$, the second $m(x_0) + Dm(x_0)(x - x_0) + \frac{1}{2}D^2m(x_0)(x - x_0)(x - x_0)$ and so on. Each term being a multilinear operator, and for sufficiently well behaved m, we can write the generalized Taylor series the following way:

$$m(x)(t) = m(x_0)(t) + \sum_{n=1}^{\infty} \int_{\tau_1} \cdots \int_{\tau_n} \frac{1}{n!} d_n(\tau_1, \ldots, \tau_n, t) \prod_{j=1}^{n} (x(\tau_j) - x_0(\tau_j)) d\tau_j \quad (2.6.2)$$

Where $d_n(\tau_1, \ldots, \tau_n, t) = D^n m(x_0)(\delta[\tau_D - \tau_1] \cdots \delta[\tau_D - \tau_n])(t)$ is the kernel function of $D^n m(x_0)$.

Let m be time invariant, $x_0 = 0$, define $h_n(\tau_1, \ldots, \tau_n) = d_n(-\tau_1, \ldots, -\tau_n, 0)$ and notice $m(0)(t)$ is independent of t because a time invariant operator of a constant function is a constant function. Then this can be simplified to obtain the Volterra series:

$$\begin{aligned} m(x)(t) &= m(0)(t) + \sum_{n=1}^{\infty} \int_{\tau_1} \cdots \int_{\tau_n} \frac{1}{n!} d_n(\tau_1, \ldots, \tau_n, t) \prod_{j=1}^{n} x(\tau_j) d\tau_j \\ &= m(0) + \sum_{n=1}^{\infty} \int_{\tau_1} \cdots \int_{\tau_n} \frac{1}{n!} h_n(t - \tau_1, \ldots, t - \tau_n) \prod_{j=1}^{n} x(\tau_j) d\tau_j \quad (2.6.3) \\ &= m(0) + \sum_{n=1}^{\infty} \int_{\tau_1} \cdots \int_{\tau_n} \frac{1}{n!} h_n(\tau_1, \ldots, \tau_n) \prod_{j=1}^{n} x(t - \tau_j) d\tau_j \end{aligned}$$

2.7 Mapping space on a line

Here's a fun fact: you can map all real numbers onto an interval like $[-1, 1]$. For instance the map $x \rightarrow \frac{2}{\pi}\arctan(x)$ is one such map. It is, perhaps even more fascinatingly, possible to map 2D space onto 1D space. For instance given the coordinates of a 2D point $(...x_1x_0.x_{-1}x_{-2}..., ...y_1y_0.y_{-1}y_{-2}...)$ we can interleave their digits to obtain a single number: $x_1y_1x_0y_0.x_{-1}y_{-1}x_{-2}y_{-2}...$. (see 2.10) Indeed this works for any number of finite dimensions. Moreover, there exist mappings that are continuous (see 2.9).We can, in fact, devise a similar mapping valid for an infinite amount of dimensions: Given a sequence $x[n]$ of numbers in $[0, 1]$ with the mth most significant bit of the nth term denoted $x[n]_m$ (with the indexes starting at 1) we can map the sequence $x[n]$ to the real number $0.x[1]_1x[2]_1x[1]_2x[3]_1x[2]_2\ldots$ (see 2.11). These schemes are collectively termed space-filling curves (or, often, the term "curve" is reserved for continuous mappings).

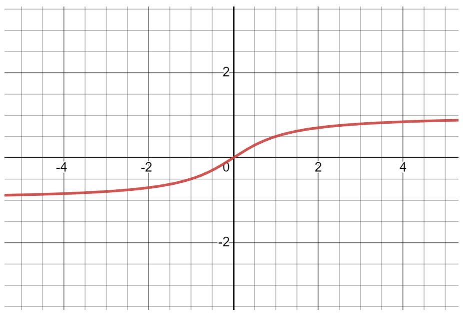


Figure 2.8: the function $\frac{2}{\pi}\arctan(x)$

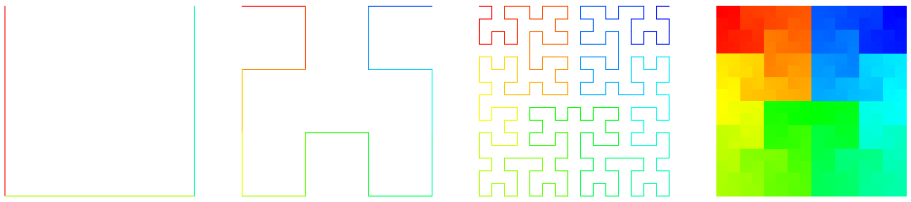


Figure 2.9: The Hilbert curve. Iterations 1, 2, 4, 8. A continuous map from $[0,1]$ to $[0,1] \times [0,1]$. [6]

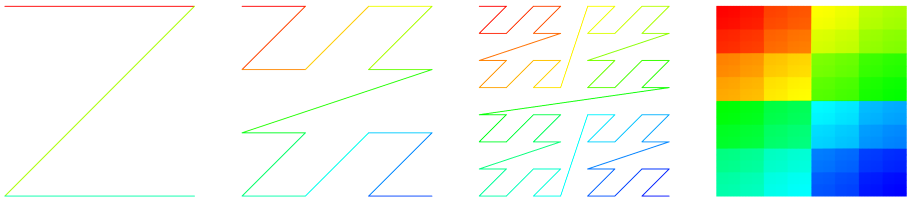


Figure 2.10: The Z order (or Morton curve). Iterations 1, 2, 4, 8. A discontinuous map from $[0,1]$ to $[0,1] \times [0,1]$. Achieved by interleaving the bits of the x and y coordinates. [7]

Inspired by this, we might be interested in the prospect of making a synthesizer that could generate any "reasonable" sound wave by varying a single parameter. Or perhaps having one parameter for the pitch of the sound, another for the volume and yet another for the timbre. We may formulate the first idea as trying to find a surjective mapping from the real numbers to the set of real functions.

Unfortunately, the set of real functions is larger in cardinality than the set of real numbers. Let $\{-1,1\}^{\mathbb{R}}$ denote the set of functions $x : \mathbb{R} \to \{-1,1\}$. Clearly $\{-1,1\}^{\mathbb{R}}$ is a subset of $\mathbb{R}^{\mathbb{R}}$; the set of functions from real numbers to real numbers. Now we can use Cantor's famous diagonal argument to prove that $|\{-1,1\}^{\mathbb{R}}| > |\mathbb{R}|$: Suppose there exists a function $L : \mathbb{R}^+ \to \{-1,1\}^{\mathbb{R}}$ whose range is all of $\{-1,1\}^{\mathbb{R}}$ i.e. for any $x \in \{-1,1\}^{\mathbb{R}}$ there exists an $r \in \mathbb{R}^+$ such that $L(r)(t) = x(t)$. Then we could construct a function $d(t) = -L(t)(t) \in \{-1,1\}^{\mathbb{R}}$. Then for any r, $d(t) \neq L(r)(t)$. But this means d is not in the range of L which contradicts our assumption.

This is not much of an issue however, restricting ourselves to the set of continuous real functions for instance already, perhaps surprisingly, yields a cardinality of $|\mathbb{R}|$. This is because one can map the set of continuous functions to the set of real sequences $\mathbb{R}^{\mathbb{N}}$, which we have already proven to have the same cardinality as $\mathbb{R}$, by mapping each continuous function to its values on all the rational points. Since the rational points are dense in $\mathbb{R}$, this determines the function. [1]

If we restrict ourselves to periodic functions of a specific frequency and sound measure (e.g. loudness/energy) we can identify each such function with its Fourier series and constrain its sound measure to a certain value. Choosing the sound measure $\mu\{x\} = \sum_f |\hat{x}(f)|^2$ for f being the multiples of the fundamental frequency, functions of equal measure conveniently lie on a sphere (by the Pythagorean theorem) on which each function can be given a spherical coordinate. The sequence of coordinate values can then be mapped onto the real numbers as mentioned before.

It would be convenient if we could devise such an infinite dimensional space filling scheme that was also continuous. Unfortunately this may very well be impossible. Consider the following impossibility result: There cannot be a uniformly continuous

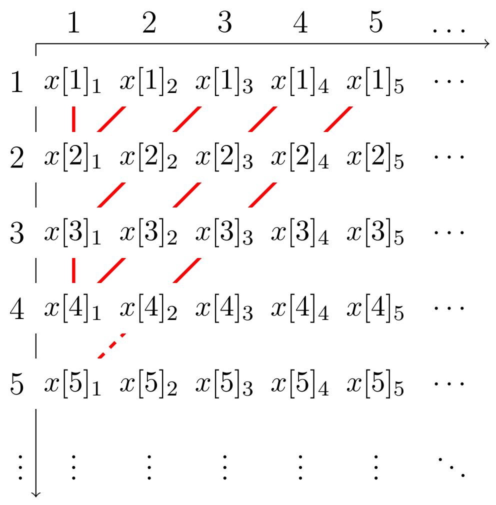

1 2 3 4 5 ...
1 $x[1]_1$ $x[1]_2$ $x[1]_3$ $x[1]_4$ $x[1]_5$ ⋯
2 $x[2]_1$ $x[2]_2$ $x[2]_3$ $x[2]_4$ $x[2]_5$ ⋯
3 $x[3]_1$ $x[3]_2$ $x[3]_3$ $x[3]_4$ $x[3]_5$ ⋯
4 $x[4]_1$ $x[4]_2$ $x[4]_3$ $x[4]_4$ $x[4]_5$ ⋯
5 $x[5]_1$ $x[5]_2$ $x[5]_3$ $x[5]_4$ $x[5]_5$ ⋯

Figure 2.11: mapping a sequence of values in $[0, 1]$ to $[0, 1]$

map m from $[0, 1]$ to the subset of ℓ^2 of sequences with elements in $[0, 1]$, denoted here as $\mathbb{N}^{[0,1]}$, because if there were, then, for a distance $\varepsilon < 1$, there would be some cutoff bit after which changing bits would no longer result in a change greater than ε which means that there are finitely many bits that effect m by more than ε and hence a finite amount N of images of m that are less than ε away from any other element of $\mathbb{N}^{[0,1]}$. But as the number of dimensions $D \to \infty$ the combined measure of the regions no further than ε from this finite set of images tends to $\lim_{D\to\infty} N\varepsilon^D = 0$ while the measure of the entire space stays 1 which concludes the contradiction.
This impossibility result can certainly be generalized but I won't do so here due to time constraints.

Chapter 3

Code

3.1 The JUCE framework

There are multiple formats audio plugins come in. A major one is Steinberg's virtual studio technology (VST) format. Steinberg does provide a software developement kit (SDK)[2] but unfortunately it has quite poor documentation. That is why I will be using the JUCE framework [3]. Not only is it better documented, but it also compiles the code you write into multiple different formats. JUCE does require you to pay if you want to commercialize your plugin though. Both the Steinberg VST SDK and JUCE are in `C++`.
When you create a new audio plugin project in JUCE (see [12]) the framework provides you with four files meant to be edited:

- PluginEditor.cpp
- PluginEditor.h
- PluginProcessor.cpp
- PluginProcessor.h

The editor files are meant for the user interface while the processor files are meant for the actual calculations.
Inside the PluginProcessor.cpp file there are three main functions you may want to edit:

- The constructor of the `PluginNameAudioProcessor` class, which is where you might initialize your plugin
- The `prepareToPlay` function, which is called before playback and passes the sample rate (and the expected maximum size of the audio blocks)
- The `processBlock` method, which is where the actual processing happens

The `processBlock` method passes a buffer, with variable name `buffer`, that serves to store both the input and the output audio and a separate musical instrument digital interface (MIDI) buffer, with variable name `midiMessages`, that serves to store both the input and output MIDI messages. Memory can be deallocated in the `releaseResources` function called after playback ends or the destructor of the `PluginNameAudioProcessor` class depending on the nature of the memory.
Variables used in PluginProcessor.cpp can be declared in PluginProcessor.h.

3.2 First sounds

Let us first get our feet wet by coding synthesizer that plays a sine wave at A4 = 440Hz. Later we will extend its functionality to a polyphonic MIDI wavetable synthesizer.
We need to do the following:

1. Initialize a lookup table for our sine wave inside the constructor
2. compute the conversion factor `freqToSampleRatio` for converting frequencies in $\frac{\text{cycles}}{\text{second}}$ to frequencies in $\frac{\text{lookup table samples}}{\text{buffer samples}}$ in the `prepareToPlay` function
3. write to the output buffer in the `processBlock` method
4. declare all the variables left to declare in PluginProcessor.h

Here's the code:
Step 1:

```
SynthAudioProcessor::SynthAudioProcessor()
//...
{
    for (int i = 0; i < lookupSamples; ++i)
    {
        lookupArray[i]
        = sin(juce::MathConstants<float>::twoPi * i /
   lookupSamples);
    }
}
```

Step 2:

```
void SynthAudioProcessor::prepareToPlay (double sampleRate, int
   samplesPerBlock)
{
    freqToSampleRatio = lookupSamples / sampleRate;
}
```

Step 3:

```
void SynthAudioProcessor::processBlock
(juce::AudioBuffer<float>& buffer,
juce::MidiBuffer& midiMessages)
{
    //clear any residual garbage in the buffer
    buffer.clear();

    //we write to all channels in the buffer by writing
    //to all channels in buffer.getArrayOfWritePointers()
    auto* buffers=buffer.getArrayOfWritePointers();
    for (auto sample = 0;
    sample < buffer.getNumSamples();
    ++sample)
    {
        for (auto channel = 0;
        channel < buffer.getNumChannels();
        ++channel)
        {
```

```cpp
            buffers[channel][sample] = lookupArray[(int)position];
        }

        //Increment position. When position reaches
        //the end of the table it has to wrap around
        if ((position += A4 * freqToSampleRatio)
        > (float)lookupSamples)
            position -= (float)lookupSamples;
    }
}
```

Step 4:

```cpp
class SynthPlaygroundAudioProcessor  : public juce::AudioProcessor
//...
{
//...
private:
    const float A4 = 440.f;
    static const int lookupSamples = 1 << 10;
    float lookupArray[lookupSamples] = {};
    float freqToSampleRatio = 0;
    float position = 0;
}
```

(To test out the plugin see [12].)

3.3 Coding a MIDI synthesizer

Implementing MIDI functionality makes things more complicated. The MIDI protocol communicates whether a note is playing by sending a NoteOn message when it starts and a NoteOff message when it ends. These messages are stored in the MIDI buffer. We will implement MIDI-handling by keeping a list of playing ”voices”, iterating through the MIDI messages in the buffer, for each message rendering the section between this message and the previous one and then updating the list of voices according to the MIDI message supplied. Let us define the list of voices to be of type `std::map<int, VoiceData>` where VoiceData is a class that stores the variable `VoiceData::position` for each voice (delete the old variable `position`) and `frequency * freqToSampleRatio` as `delta`:

```cpp
class SynthPlaygroundAudioProcessor  : public juce::AudioProcessor
//...
{
//...
private:
    //...
    struct VoiceData
    {
        float position = 0;
        float delta = 0;
        VoiceData(float initPosition, float initDelta)
        {
            position = initPosition;
            delta = initDelta;
        }
    };
    std::map<int, VoiceData> voices;
}
```

Now in the processBlock method:

```
void LineSynthAudioProcessor::processBlock
(juce::AudioBuffer<float>& buffer,
juce::MidiBuffer& midiMessages)
{
    buffer.clear();

    //Add empty buffer end message
    //This is necessary to ensure iterating through
    //midiMessages reaches the end of the buffer
    midiMessages.addEvent(juce::MidiMessage(), buffer.
  getNumSamples());

    auto* buffers = buffer.getArrayOfWritePointers();
    auto lastSample = 0;
    for (auto midiMetadata : midiMessages)
    {
        for (auto sample = lastSample;
        sample < midiMetadata.samplePosition;
        ++sample)
        {
            for (auto& voice : voices)
            {
                for (auto channel = 0;
                channel < buffer.getNumChannels();
                ++channel)
                {
                    buffers[channel][sample]
                    += lookupArray[(int)voice.second.position];
                }
                if ((voice.second.position += voice.second.delta
  )
                > (float)lookupSamples)
                    voice.second.position -= (float)
  lookupSamples;
            }
        }

        auto midiMessage = midiMetadata.getMessage();
        if (midiMessage.isNoteOn())
        {
            float frequency = A4 * pow(2.f,
                    float(midiMessage.getNoteNumber() - 69) / 12.f);
               voices.insert({ midiMessage.getNoteNumber(),
                    VoiceData(0.f, freqToSampleRatio * frequency) })
     ;
          }
          else if (midiMessage.isNoteOff())
          {
               voices.erase(midiMessage.getNoteNumber());
          }
          lastSample = midiMetadata.samplePosition;
     }
}
```

3.4 Anti-aliasing

If we had chosen not to initialize `lookupArray` with a sine wave but instead something with a lot of high harmonics we may run into aliasing (see 2.5.4). To hear this, initialize `lookupArray` with a square wave.

```
SynthAudioProcessor::SynthAudioProcessor()
//...
{
     for (int i = 0; i < lookupSamples; ++i)
     {
          lookupArray[i] = (i < (lookupSamples / 2)) ? -0.5f : 0.5
     f;
     }
}
```

You should be listening for frequencies that sound inharmonic, distorted or anything that makes the synth sound different in the higher registers.
So how do we avoid aliasing while preserving the harmonic richness when possible? Well as per 2.5.4 there are two simple ways to fix this:

- Store the Fourier transform of `lookupArray` and perform an FFT every time a voice is added with the aliasing frequencies removed
- Convolve `lookupArray` with an approximation of the sinc function either when a voice gets added or when it is read.

Both methods incur a sizeable performance penalty.
Instead we will create an array of lookup tables that, for each power of two, stores the waveform with all higher frequencies removed. To determine which lookup table to use we calculate the highest non-aliasing harmonic h from the fundamental frequency f and the sampling frequency f_s:

$$f * h < \frac{f_s}{2} \tag{3.4.1}$$

$$h < \frac{f_s}{2f} \tag{3.4.2}$$

Which gives the highest non aliasing power of 2 to be $\left\lfloor \log_2(\frac{f_s}{2f}) \right\rfloor = \left\lfloor \log_2(\frac{f_s}{2}) - \log_2(f) \right\rfloor$. To implement this we have to create an instance of `juce::dsp::FFT` of order `fftOrder` (make sure you add the dsp module in the Projucer app [see [13]]), make `fftOrder` lookup tables and add a pointer to our VoiceData class that points to the lookup table to be used. I will also add a variable `logNyquist` to store $\log_2(\frac{f_s}{2})$ and another array of floats `fftArray`. We are going to use the functions `juce::dsp::FFT::performRealOnlyForwardTransform()` and `juce::dsp::FFT::performRealOnlyInverseTransform()`. `juce::dsp::FFT::performRealOnlyForwardTransform()` will replace the input array with the real and complex components of each successive frequency component in alternating fashion. `juce::dsp::FFT::performRealOnlyForwardTransform()` actually operates on an array of floats but because each frequency component is stored as *two* floats we need to pass an array of twice the size of the amount of data stored within it. Similarly `juce::dsp::FFT::performRealOnlyInverseTransform()` will operate on arrays with twice the data as the output. All this is to say that if `fftOrder` is the order of the FFT and `lookupSamples = 1 << (fftOrder)` is the amount of samples-in-use in `lookupArray` and `fftArray` we need `lookupArray` and `fftArray` to have length `lookupSamples * 2` to be able to use the FFT on them. This is how the variable declarations in PluginProcessor.h should look:

```
    class SynthPlaygroundAudioProcessor  : public juce::
  AudioProcessor
//...
{
//...
private:
    const float A4 = 440.f;
    static const int fftOrder = 10;
    static const int lookupSamples = 1 << (fftOrder);
    float fftArray[lookupSamples * 2] = {};
    float lookupArray[fftOrder][lookupSamples * 2] = {};
    float freqToSampleRatio = 0;
    float logNyquist = 1;
    juce::dsp::FFT fft{ fftOrder };
    struct VoiceData
    {
        float position = 0;
        float delta = 0;
        float* dealiasedLookupArray = nullptr;
        VoiceData(float initPosition, float initDelta, float*
  initDealiasedLookupArray)
        {
            position = initPosition;
            delta = initDelta;
            dealiasedLookupArray = initDealiasedLookupArray;
        }
    };
}
```

When we initialize our plugin with our square wave we want to generate the dealiased versions as well:

```
    SynthAudioProcessor::SynthAudioProcessor()
//...
```

```
{
    //initialize fftArray with our square wave
    for (int sample = 0; sample < lookupSamples; ++sample)
    {
        fftArray[sample] = (sample < (lookupSamples / 2)) ? -0.5
   f : 0.5f;
    }

    //The true flag indicates that fft should only calculate
    //nonnegative frequencies.
    fft.performRealOnlyForwardTransform(fftArray, true);

    //create dealiased copies
    for (int cutoffOrder = 0; cutoffOrder < fftOrder; ++
   cutoffOrder)
    {
        //Copy all frequency components up to cutoffOrder.
        //Leave the rest as 0.
        //Remember, the frequency component at cutoffOrder is
        //stored as two floats at indexes
        //2 * cutoffOrder and (2 * cutoffOrder) + 1
        for (int i = 0; i < (1 << cutoffOrder) * 2; ++i)
        {
            lookupArray[cutoffOrder][i] = fftArray[i];
        }
        fft.performRealOnlyInverseTransform(lookupArray[
   cutoffOrder]);
    }
}
```

Calculate `logNyquist` in the `prepareToPlay` function:

```
void SynthPlaygroundAudioProcessor::prepareToPlay (double
   sampleRate, int samplesPerBlock)
{
    freqToSampleRatio = lookupSamples / sampleRate;
    logNyquist = log2(sampleRate / 2);
}
```

In the innermost for loop in `processBlock` replace `buffers[channel][sample] += LookupArray[(int)voice.second.position];` with `buffers[channel][sample] += voice.second.dealiasedLookupArray[(int)voice.second.position];` and change the handling of a NoteOn event to:

```
if (midiMessage.isNoteOn())
{
    float frequency = A4 * pow(2.f,
        float(midiMessage.getNoteNumber() - 69) / 12.f);
    voices.insert({ midiMessage.getNoteNumber(),
        VoiceData{ 0.f, freqToSampleRatio * frequency,
        lookupArray[int(logNyquist - log2(frequency))] } });
}
```

Chapter 4

Discussion

As is apparent, I have made the effort to explain a lot of math that I didn't end up using. I did intend on using all of what was explained in 2 but due to very poor time management I failed to finish on time. As I had mentioned in 2.7 I was, and still am, going to make a synthesizer plug-in that encodes all ”reasonable” sound waves on a line (which could hopefully be represented on a slider). Unfortunately I am currently stuck at the debugging stage. Similarly, I was, and still am, going to make an effect plugin which can approximate other effect plugins via the Volterra series. I have not gotten started with that yet. The repositories for both will be provided in the Appendix.
Here is a run-down of the Ideas that I had not gotten to implementing:
For the space-filling curve synthesizer I was going to do the mapping in two stages:

1. Map the 1D input to a sequence of real numbers
2. Interpret the sequence as some kind of spherical coordinate

Step 2 is to ensure that all inputs gave the same output volume. Here are a couple of mappings I wanted to implement for step 1:

- Splicing Hilbert curves of successive dimensions end to end such that the resulting mapping would be continuous. This would have the downside of being ridiculously redundant but the advantage of being more gradual than most alternatives. It could also not represent sounds with infinitely many overtones.
- Going through the array in figure 2.11 in a Z order. This is probably the easiest to implement and does not have the downsides of the previous option. It, however has the downside of being discontinuous even when low-pass filtered.
- Mapping the elements of 2.11 with indexes n, m, $n + m = N$ for some fixed N into a first iteration Hilbert curve and splicing together. Could possibly preserve the advantages of the previous option while being continuous when low-pass filtered. (I am not quite sure about this one yet)

Here are a couple of mappings I wanted to implement for step 2:

- Interpret the input sequence as a spherical coordinate in frequency space
- Interpret the input sequence as a polyspherical coordinate in the time domain

Overall I am quite unhappy with how this project turned out. I can only hope that I've learned my lessons. If I were to give my past self some advice it would be:

- Meet with the supervisor more often to discuss ”checkpoints”
- Test functions in separate programs to see if they work

and of course:

- Start earlier

Glossary

angular frequency Angle traversed per unit time. 2, 10

DAW digital audio workstation. 1
DFT discrete Fourier transform. 7

eigenfunction Analogue of an eigenvector for linear operators. An eigenfunction f of a linear operator M has the property that applying M is identical to scaling f by a factor λ; $M\{f(\tau_M)\}(t) = \lambda f(t)$. 6, 9
equalizer An audio effect that amplifies certain frequencies and suppresses others. 5

FFT fast Fourier transform. 10, 19, 20
fundamental frequency Inverse of the period of a periodic function. 6, 19

IDFT inverse discrete Fourier transform. 7
indicator function A function that returns 1 when the argument is in the set and 0 otherwise. 11

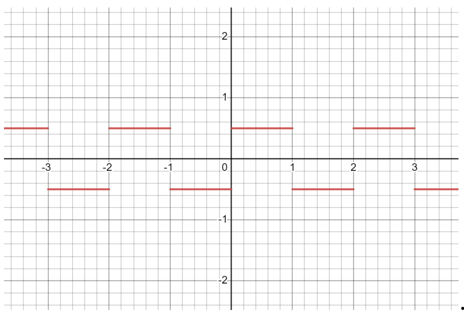
L^2 The space of square-integrable functions, that is, for our purposes, the set of functions $x : \mathbb{R} \to \mathbb{C}$ for which $\int_{\mathbb{R}} |x(t)|^2 dt$ converges. It is equipped with the inner product $\langle x, y\rangle = \int_{\mathbb{R}} x(t)\overline{y(t)}dt$. 7
ℓ^2 The space of square-summable sequences, that is, for our purposes, the set of sequences $x : \mathbb{N} \to \mathbb{C}$ for which $\sum_{\mathbb{R}} |x(t)|^2$ converges. It is equipped with the inner product $\langle x, y\rangle = \sum_{t\in\mathbb{N}} x(t)\overline{y(t)}$. 14

MIDI musical instrument digital interface. 15–17
multilinear Linear in all its arguments. 12

operator Another term for function. Used to emphasize that it maps functions to functions. 3, 5, 6, 11
orthonormal Used to describe real matrices that preserve the dot product, i.e. if M is a real matrix acting on vectors $\mathbf{u}, \mathbf{v}$ then $\mathbf{u}^\mathsf{T}\mathbf{v} = (M\mathbf{u})^\mathsf{T} M\mathbf{v}$. This implies that the images of the standard basis vectors $(\mathbf{e}_i, \mathbf{e}_j,\ i \neq j)$ remain orthogonal $(\mathbf{e}_i^\mathsf{T}\mathbf{e}_j = 0)$ and are of unit length (i.e. normalized; hence ortho*normal*)$(\mathbf{e}_i^\mathsf{T}\mathbf{e}_i = 1)$. Since an orthonormal matrix preserves the dot product $I^\mathsf{T}\mathbf{v} = (MI)^\mathsf{T} M\mathbf{v}$ (where I is the identity matrix) whence $M^{-1} = M^\mathsf{T}$.

reverb The persistence of sound after it is produced. 5
root of unity A root of unity of order N is a number ω such that $\omega^N = 1$. 7

SDK software developement kit. 15

square wave 

. 19

uniformly continuous Used to describe functions. A function x is said to be uniformly continuous if for all $\epsilon > 0$ there exists a $\delta > 0$ such that for all s, t a distance less that δ apart $x(s), x(t)$ are less than a distance ϵ apart.. 13

unitary Used to describe operators that preserve the inner product of their space. In L^2 the kernel function of the inverse of a unitary operator n, similar to orthonormal matrices, is given by the the conjugate transpose of the kernel function of the operator m: $n(\tau, t) = \overline{m(t, \tau)}$. 7, *see* orthonormal

VST virtual studio technology. 15

wavetable A lookup table for a sound wave. 16

Bibliography

[1] Dejan Govc (https://math.stackexchange.com/users/19588/dejan-govc). *Cardinality of set of real continuous functions.* Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/271641 (version: 2013-01-06). eprint: `https://math.stackexchange.com/q/271641`. URL: `https://math.stackexchange.com/q/271641`.

[2] *3rd-Party Developers Support & SDKs - Steinberg.* URL: `https://www.steinberg.net/developers/`.

[3] *Download JUCE.* URL: `https://juce.com/get-juce/download`.

[4] Andrew Duncan. *The Analytic Impulse.* URL: `http://andrewduncan.net/air/`.

[5] Iain. *Impulse.* URL: `https://commons.wikimedia.org/w/index.php?curid=1665418`.

[6] Josia John. *Hilbert curve.*

[7] Josia John. *Z order.*

[8] *Linear time-invariant system - Wikipedia.* Exponentials as eigenfunctions. URL: `https://en.wikipedia.org/wiki/Linear_time-invariant_system#Exponentials_as_eigenfunctions`.

[9] Grant Sanderson. *Essence of Linear Algebra.* URL: `https://youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab`.

[10] Grant Sanderson. *Taylor series.* Chapter 11, Essence of calculus. URL: `https://youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab`.

[11] *The Fast Fourier Transform (FFT).* Most Ingenious Algorithm Ever? URL: `https://youtu.be/h7apO7q16V0`.

[12] *Tutorial: Create a basic Audio/MIDI plugin.* Part 1: Setting up. URL: `https://docs.juce.com/master/tutorial_create_projucer_basic_plugin.html`.

[13] *Tutorial: Projucer Part 1: Getting started with the Projucer.* URL: `https://docs.juce.com/master/tutorial_new_projucer_project.html`.

Appendix

A Github repo containing this document as a PDF and the plugins once I finish them: