Introduction

The dawn of the digital age has brought fundamental changes to numerous areas of science as well as our everyday lives. In general, one may observe that many phenomena and aspects of the physical world — hardware — have been replaced by virtual simulations — software. This is also true for the realm of electronic music production. Whereas music synthesizers such as the legendary "Moog" previously produced sounds through analog circuitry, digital synthesizers can nowadays be implemented with common programming languages such as C++.

The question that remains, of course, is: how? Where does one start on day zero? What knowledge is required? What resources are available off and online? What are the best practices? How can somebody with no previous experience in generating sound through software build an entire synthesis system, from scratch? Even though building a software synthesizer is itself a big enough mountain to climb, the real difficulty lies in finding the path that leads one up this mountain.

Two publications that are especially valuable when attempting to build a software synthesizer are *BasicSynth: Creating a Music Synthesizer in Software*, by Daniel R. Mitchell, and *The Scientist and Engineer's Guide to Digital Signal Processing*, by Steven W. Smith. The first book is a practical, hands-on guide to understanding and implementing the concepts behind digital synthesizers. It provides simple, straight-forward explanations as well as pseudo-code examples on a variety of subjects that are also discussed in this thesis. The real value of Mitchell's publication is that it simplifies and condenses years of industry practices and digital signal processing (DSP) knowledge into a single book. However, *BasicSynth* abstracts certain topics, such as digital filters, too much. Where its explanations do not suffice for a full, or at least practical, understanding, *The Scientist and Engineer's Guide to Digital Signal Processing* is an excellent alternative. Steven W. Smith's book explains concepts of Digital Signal Processing (DSP) in great detail, and is to some extent on the opposite end of the spectrum of explanation depth, when compared to *BasicSynth*. It should be noted, however, that Smith's publication is not at all focused on audio processing or digital music, but on DSP in general. (Mitchell, 2008) (Smith, 1999)

This thesis is intended to dicuss the most important steps on the path from zero lines of code to a full synthesis system, implemented in C++. While providing many programming snippets and even full class definitions[1], a special focus will be put on explaining the fundamental ideas behind the programming, which very often lie in the realm of mathematics or general digital signal processing. The reason for this is that once a theoretical understanding has been established, the practical implementation becomes a trivial task.

[1]C++ program samples of up to 50 lines of code are displayed in-line with the text, longer samples are found in Appendix A.

The first chapter will introduce some rudimentary concepts and phenomena of sound in the digital realm, as well as explain how digital sound differs to its analog counterpart. Subsequent chapters examine, among other things, how computer music is generated, modulated, filtered, synthesized and finally made audible. Images of sound samples in the time and frequency domain, as well as diagrams to abstract certain concepts, will be provided. Moreover, programming relationships, such as inheritance diagrams, between parts of the synthesizer implemented for this thesis, called *Anthem*, are also displayed when relevant. Lastly, excerpts of email or online forum exchanges will be given when they contributed to the necessary knowledge.

It should be made clear that this thesis is not a "tutorial" on how to program a complete synthesis system in C++. It is also not designed to be a reference for theoretical concepts of digital signal processing. Rather, practice and theory will be combined in the most pragmatic way possible.

Chapter 1

From Analog to Digital

Our everyday experience of sound is an entirely analog one. When a physical object emits or reflects a sound wave into space and towards our ears, the signal produced consists of an infinite set of values, spaced apart in infinitesimal intervals. Due to the fact that such a signal has an amplitude value at every single point in time, it is called a continuous signal. (Smith, 1999, p. 11) Figure 1.1 displays the continuous representation of a sine wave.

`img/cont.png`

Figure 1.1: The continuous representation of a typical sine wave. In this case, both the signal's frequency f as well as the maximum elongation from the equilibrium a are equal to 1.

While continuous signals and the idea of an infinite, uncountable set of values are easy to model in mathematics and physics — the analog world, computers — in the digital world — effectively have no means by which to represent something that is infinite, since computer memory is a finite resource. Therefore, signals in the digital domain are discrete, meaning they are composed of periodic *samples*. A sample is a discrete recording of a continuous signal's amplitude, taken in a constant time interval. The process by which a continuous signal is converted to a discrete signal is called *quantization*, *digitization* or simply *analog-to-digital-conversion* (Smith, 1999, p. 35-36). Quantization essentially converts an analog function of amplitude to a digital function of location in computer memory over time (Burk, Polansky, Repetto, Roberts and Rockmore, 2011, Section 2.1). The reverse process of converting discrete samples to a continuous signal is called *digital-to-analog-conversion*. Figure 1.2 shows the discrete representation of a sine wave, the same signal that was previously shown as a continuous signal in Figure 1.1. (Mitchell, 2008, p. 16-17)

Figure 1.2: The discrete representation of a typical sine wave.

1.1 Sample Rate

The sample rate (often referred to as sampling rate or sampling frequency), commonly denoted by f_s, is the rate at which samples of a continuous signal are taken to quantize it. The value of the sample rate is measured in Hertz (Hz) or samples per second. Common values for audio sampling rates are 44.1 kHz, a frequency originally chosen by Sony in 1979 that is still used for Compact Discs, and 48 kHz, the standard audio sampling rate used today. (Mitchell, 2008, p. 18) (Colletti, 2013) The reciprocal of the sample rate yields the sampling interval, denoted by T_s and measured in seconds, which is the time period after which a single sample is taken from a continuous signal:

$$T_s = \frac{1}{f_s} \tag{1.1}$$

The reciprocal of the sample interval again yields the sampling rate:

$$f_s = \frac{1}{T_s} \tag{1.2}$$

1.2 Nyquist Limit

The sample rate also determines the range of frequencies that can be represented by a digital sound system. The reason for this is that only frequencies that are less than or equal to one half of the sampling rate can be "properly sampled". To sample a signal "properly" means to be able to reconstruct a continuous signal, given a set of discrete samples, exactly, i.e. without any *quantization errors*. This is only possible if the frequency of a signal allows at least one sample per cycle to be taken above the equilibrium and at least one sample below. The value of one half of the sample rate is called the *Nyquist frequency* or *Nyquist limit*, named after Harry Nyquist, who first described the Nyquist limit and associated phenomena together with Claude Shannon in the 1940s, stating that "a continuous signal can be properly sampled, only if it does not contain frequency components above one half the sampling rate". Any frequencies above the Nyquist limit lead to *aliasing*, which is discussed in the next section. (Smith, 1999, p. 40)
Given the definition of the Nyquist limit and considering the fact that the limit of human hearing is approximately 20 kHz (Cutnell & Johnson, 1998, p. 466), the reason for which the two most common audio sample rates are 40 kHz and above is clear: they were chosen to allow the "proper" representation of the entire range of frequencies audible to humans, since a sample rate of 40 kHz or higher meets the Nyquist requirement of a sample rate at least twice the maximum frequency component of the signal to be sampled (the Nyquist limit), in this case ca. 20 Khz.

1.3 Aliasing

When a signal's frequency exceeds the Nyquist limit, it is said to produce an *alias*, a new signal with a different frequency that is indistinguishable from the original signal when sampled. This is due to the fact that a signal with a frequency component above the Nyquist limit no longer has one sample taken above and one below the zero level for each cycle, but at arbitrary points of the original signal. When these points are connected, they yield an entirely different signal. For example, if the sinusoid depicted in Figure 1.3, with a frequency of 4 Hz, is sampled at a sample rate of 5 Hertz, shown in Figure 1.4, meaning the frequency of the continuous signal is higher than the Nyquist limit (here 2.5 Hz), the reconstructed signal, approximated in Figure 1.5, will look completely different from the original sinusoid. "This phenomenon of [signals] changing frequency during sampling is called aliasing, [...] an example of improper sampling". (Smith, 1999, p. 40) img/orig.png

Figure 1.3: A sinusoid with a frequency of 4 Hz.

img/sampled.png

Figure 1.4: A sinusoid with a frequency of 4 Hz, sampled at a sample rate of 5 Hz. According to the Nyquist Theorem, this signal is sampled improperly, as the frequency of the continuous signal is not less than or equal to one half of the sample rate, in this case 2.5 Hz.

img/approx.png

Figure 1.5: An approximated representation of the signal created from the sampling process shown in Figure 1.4. Visually as well as acoustically, the new signal is completely different from the original signal. However, in terms of sampling, they are indistinguishable from each other due to improper sampling. These signals are said to be *aliases* of each other.

Chapter 2

Generating Sound

The following sections will outline how digital sound can be generated in theory and implemented in practice, using the C++ programming language.

2.1 Simple Waveforms

The simplest possible audio waveform is a sine wave. As a function of time, it can be mathematically represented by Equation 2.1, where A is the maximum amplitude of the signal, f the frequency in Hertz and ϕ an initial phase offset in radians.

$$f_s(t) = A\sin(2\pi ft + \phi) \tag{2.1}$$

A computer program to compute the values of a sine wave with a variable duration, implemented in C++, is shown in Listing **??**. Another waveform similar to the sine wave is the cosine wave, which differs only in a phase offset of 90°or $\frac{\pi}{2}$ radians, as shown in Equation 2.2.

$$f_c(t) = A\cos(2\pi ft + \phi) = A\sin(2\pi ft + \phi + \frac{\pi}{2}) \tag{2.2}$$

Therefore, the program from Listing **??** could be modified to compute a cosine wave by changing line 22 from:

to

2.2 Complex Waveforms

Now that the process of creating simple sine and cosine waves has been discussed, the generation of more complex waveforms can be examined. Generally, there are two methods by which complex waveforms can be created in a digital synthesis system: mathematical calculation and additive synthesis. In the first case — mathematical calculation, waveforms are computed according to certain mathematical formulae and thus yield *perfect* or *exact* waveforms, such as a square wave that is equal to its maximum amplitude exactly one half of a period and equal to its minimum amplitude for the rest of the period. While these waveforms produce a very crisp and clear sound, they are rarely found in nature due to their degree of perfection and are consequently rather undesirable for a music synthesizer.

Table 2.1: C++ code to represent a single partial in a Fourier Series.

The second method of generating complex waveforms, additive synthesis, produces waveforms that, despite not being mathematically perfect, are closer to the waveforms found naturally. This method involves the summation of a theoretically infinite, practically finite set of sine and cosine waves with varying parameters. Additive synthesis is often called Fourier Synthesis, after the 18th century French scientist, Joseph Fourier, who first described the process and associated phenomena of summing sine and cosine waves to produce complex waveforms. This calculation of a complex, periodic waveform from a sum of sine and cosine functions is also referred to as a Fourier Transform or a Fourier Series, both part of the Fourier Theorem. In a Fourier Series, a single sine or cosine component is either called a harmonic, an overtone or a partial. All three name the same idea of a waveform with a frequency that is an *integer multiple* of some fundamental pitch or frequency. (Mitchell, 2008, p. 64) Throughout this thesis the term *partial* will be preferred.
Equation 2.13 gives the general definition of a discrete Fourier Transform and Equation 2.14 shows a simplified version of Equation 2.13. Table 2.1 presents a C++ struct to represent a single partial and Listing **??** a piece of C++ code to compute one period of any Fourier Series.

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos(\omega n t) + b_n \sin(\omega n t))$$

Equation 2.13: Formula to calculate an infinite Fourier series, where $\frac{a_n}{2}$ is the center amplitude, a_n and b_n the partial amplitudes and ω the angular frequency, which is equal to $2\pi f$.

$$f(t) = \sum_{n=1}^{N} a_n \sin(\omega n t + \phi_n)$$

Equation 2.14: Simplificiation of Equation 2.13. Note the change from a computationally impossible infinite series to a more practical finite series. Because a cosine wave is a sine wave shifted by 90°or $\frac{\pi}{2}$ radians, the $\cos$ function can be eliminated and replaced by an appropiate $\sin$ function with a phase shift ϕ_n.

The following paragraphs will analyze how three of the most common waveforms found in digital synthesizers — the square, the sawtooth and the triangle wave — can be generated via additive synthesis.

Square Waves

When speaking of additive synthesis, a square wave is the result of summing all odd-numbered partials (3rd, 5th, 7th etc.) at a respective amplitude equal to the reciprocal of their partial number ($\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{7}$ etc.). The amplitude of each partial must decrease with increasing partial numbers to prevent amplitude overflow. A mathematical equation for such a square wave with N partials is given by Equation 2.3, where $2n - 1$ makes the series use only odd partials. A good maximum number of partials N for near-perfect but still naturally sounding waveforms is

64, a value determined empirically. Higher numbers have not been found to produce significant improvements in sound quality. Table 2.2 displays the C++ code needed to produce one period of a square wave in conjuction with the `additive` function from Listing **??**. Figure 2.1 shows the result of summing 2, 4, 8, 16, 32 and finally 64 partials.

$$f(t) = \sum_{n=1}^{N} \frac{1}{2n-1} \sin(\omega(2n-1)t) \tag{2.3}$$

img/square.png

Figure 2.1: Square waves with 2, 4, 8, 16, 32 and 64 partials.

Table 2.2: C++ code for a square wave with 64 partials.

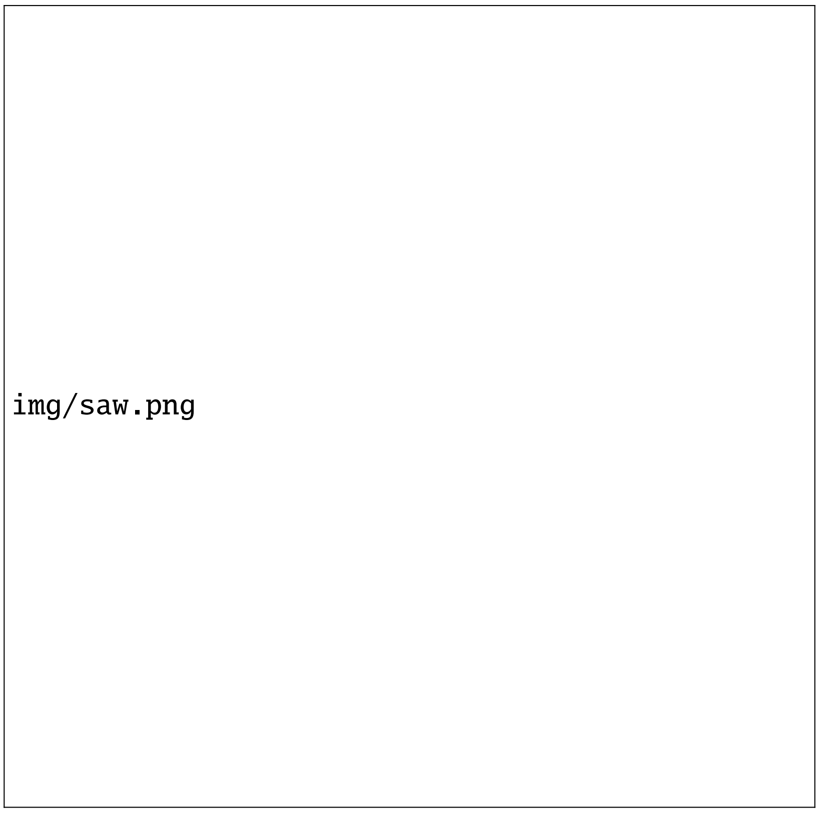

Figure 2.2: Sawtooth waves with 2, 4, 8, 16, 32 and 64 partials.

Table 2.3: C++ code for a sawtooth wave with 64 partials.

Sawtooth Waves

A sawtooth wave is slightly simpler to create through additive synthesis, as it requires the summation of every partial rather than only the odd-numbered ones. The respective amplitude is again the reciprocal of the partial number. Equation 2.4 gives a mathematical definition for a sawtooth wave, Figure 2.2 displays sawtooth functions with various partial numbers and Table 2.3 shows C++ code to generate such functions.

$$f(t) = \sum_{n=1}^{N} \frac{1}{n} \sin(\omega n t) \tag{2.4}$$

Figure 2.3: Triangle waves with 2, 4, 8, 16, 32 and 64 partials. Note that already 2 partials produce a very good approximation of a triangle wave.

Table 2.4: C++ code for a triangle wave with 64 partials.

Triangle Waves

The process of generating triangle waves additively differs from previous waveforms. The amplitude of each partial is no longer the reciprocal of the partial number, $\frac{1}{n}$, but now of the partial number squared: $\frac{1}{n^2}$. Moreover, the sign of the amplitude alternates for each partial in the series. As for square waves, only odd-numbered partials are used. Mathematically, such a triangle wave is defined as shown in Equation 2.5 or, more concisely, in Equation 2.6. Figure 2.3 displays such a triangle wave with various partial numbers and Table 2.4 implements C++ code to compute a triangle wave.

$$f(t) = \sum_{n=1}^{\frac{N}{2}} \frac{1}{(4n-3)^2} \sin(\omega(4n-3)t) - \frac{1}{(4n-1)^2} \sin(\omega(4n-1)t) \tag{2.5}$$

$$f(t) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)^2} \sin(\omega(2n+1)t) \tag{2.6}$$

2.3 Wavetables

Following the discussion of the creation of complex waveform, the two options for playing back waveforms in a digital synthesis system must be examined: continuous real-time calculation of waveform samples or lookup from a table that has been calculated once and then written to disk — a so-called Wavetable. To keep matters short, the second method was found to be computationally more efficient and thus the better choice, as memory is a more easily expended resource than computational speed.

2.3.1 Implementation

A Wavetable is a table, practically speaking an array, in which pre-calculated waveform amplitude values are stored for lookup. The main benefit of a Wavetable is that individual samples need not be computed periodically and in real-time. Rather, samples can be retrieved simply by dereferencing and subsequently incrementing a Wavetable index. If the Wavetable holds sample values for a one-second period, the frequency of the waveform can be adjusted during playback by multiplying the increment value by some factor other than one. "For example, if [one] increment[s] by two [instead of one], [one can] scan the table in half the time and produce a frequency twice the original frequency. Other increment values will yield proportionate frequencies." (Mitchell, 2008, p. 80-81) The *fundamental increment* of a Wavetable refers to the value by which a table index must be incremented after each sample to traverse a waveform at a frequency of 1 Hz. A formula to calculate the fundamental increment i_{fund} is shown in Equation 2.7, where L is the Wavetable length and the f_s the sample rate. To alter the frequency f of the played-back waveform, Equation 2.8 can be used to calculate the appropriate increment value i.

$$i_{fund} = \frac{L}{f_s} \tag{2.7}$$

$$i = i_{fund} \cdot f = \frac{L}{f_s} \cdot f \tag{2.8}$$

2.3.2 Interpolation

For almost all configurations of frequencies, table lengths and sample rates, the table index i produced by Equation 2.8 will not be an integer. Since using a floating point number as an index for an array is a syntactically illegal operation in C++, there are two options. The first is to truncate or round the table index to an integer, thus "introducing a quantization error into the signal [...]". (Mitchell, 2008, p. 84) This is a suboptimal solution which would result in a change of phase and consequently distortion. The second option, interpolation, tries to approximate the true value from the sample at the current index and at the subsequent one. Interpolation is achieved by summation of the sample value at the floored, integral part of the table index, $\lfloor i \rfloor$, with the difference between this sample and the sample value at the next table index, $\lfloor i \rfloor + i$, multiplied by the fractional part of the table index, $i - \lfloor i \rfloor$. Table 2.5 displays the calculation of a sample by means of interpolation in pseudo-code and Table 2.6 in C++. (based on pseudo-code, Mitchell, 2008, p. 85)

Table 2.5: An interplation algorithm in pseudo-code.

Table 2.6: Full C++ template function to interpolate a value from a table, given a fractional index.

2.3.3 Table Length

The length of the Wavetable must be chosen carefully and consider both memory efficiency and waveform resolution. An equation to calculate the size of a single wavetable in Kilobytes is given by Equation 2.9, where L is the table length and N the number of bytes provided by the resolution of the data type used for samples, e.g. 8 bytes for the double-precision floating-point data type `double`.

$$Size = \frac{L \cdot N}{1024} \tag{2.9}$$

Daniel R. Mitchell advises that the length of the table be a power of two for maximum efficiency. (Mitchell, 2008, p. 86) Moreover, during an E-Mail exchange with Jari Kleimola, author of the 2005 master's thesis "Design and Implementation of a Software Sound Synthesizer", it was discovered that "as a rule of thumb", the table length should not exceed the processor cache size. A relevant excerpt of this e-mail exchange is depicted in Figure 2.4. Considering both pieces of advice, it was decided that a Wavetable size of 4096 (2^{12}), which translates to 32 KB of memory, would be suitable.

One important fact to mention, also discussed by Jari Kleimola, is that because the interpolation algorithm from Table 2.6 must have access to the sample value at index $i+1$, i being the current index, an additional sample must be appended to the Wavetable to avoid a `BAD_ACCESS` error when i is at the last valid position in the table. This added sample has the same value as the first sample in the table to avoid a discontinuity. Therefore, the period of the waveform actually only fills 4095 of the 4096 indices of the Wavetable, as the 4096th sample is equal to the first.

img/jari.png

Figure 2.4: An excerpt of an E-Mail exchange with Jari Kleimola.

2.4 Noise

A noisy signal is a signal in which some or all samples take on random values. Generally, noise is considered as something to avoid, as it may lead to unwanted distortion of a signal. Nevertheless, noise can be used as an interesting effect when creating music. Its uses include, for example, the modeling of the sound of wind or the crashing of water waves against the shore of a beach. Some people enjoy the change in texture noise induces in a sound, others find noise relaxing and even listen to it while studying. (Mitchell, 2008, p. 76) Unlike all audio signals[1] presented so far, noise cannot[2] be stored in a Wavetable, as it must be random throughout its duration and not repeat periodically for a proper sensation of truly *random* noise. Another interesting fact about noise is that it is common to associate certain forms of noise with colors. The *color* of a noise signal describes, acoustically speaking, the *texture* or *timbre* [4] of the sound produced, as well as, scientifically speaking, the spectral power density and frequency content of the signal.

White noise is the most frequencly encountered form, or color, of noise. It is a random signal in its purest and most un-filtered form. In such a signal, all possible frequencies are found with a uniform probability distribution, meaning they are distributed at equal intensity throughout the signal. The association of noise with colors actually stems from the connection between white noise and white light, which is said to be composed of almost all color components at an approximately equal distribution. Figure 2.5 shows a typical white noise signal in the time domain, Figure 2.6 gives a close-up view of Figure 2.5, Figure 2.7 displays the frequency spectrum of a white noise signal and Table 2.7 shows the implementation of a simple C++ class to produce white noise.

[1]The term "waveform" would be incorrect here, as noise is not periodic and thus cannot really be seen as a waveform.

[2]Noise could theoretically be stored in a Wavetable, of course. However, even a very large Wavetable destroys the randomness property to some extent and would thus invalidate the idea behind noise being truly random.

[4]Timbre is a rather vague term used by musicians and audio engineers to describe the properties, such as pitch, tone and intensity, of an audio signal's sound that distinguish it from other sounds. The *Oxford Dictionary of English* defines timbre as "the character or quality of a musical sound or voice as distinct from its pitch and intensity".

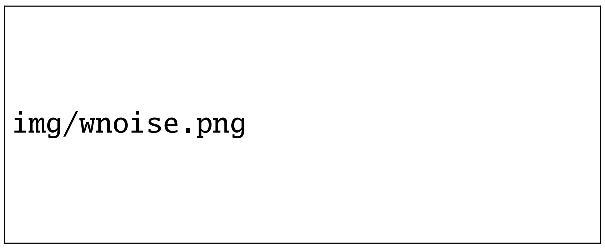

Figure 2.5: A typical white noise signal.

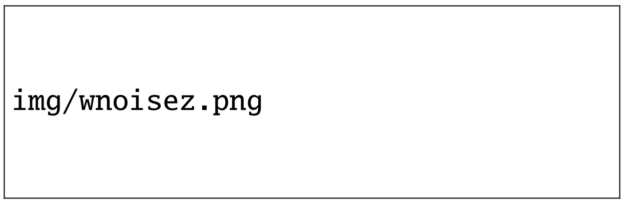

Figure 2.6: A close-up view of Figure 2.5. This Figure shows nicely how individual sample values are completely random and independent from each other.

Figure 2.7: The signal from Figures 2.5 and 2.6 in the frequency domain. This frequency spectrum analysis proves the fact that white noise has a "flat" frequency spectrum, meaning that all frequencies are distributed uniformly and at (approximately) equal intensity.

Table 2.7: A simple C++ class to produce white noise. `rgen_` is a random number generator following the Mersenne-Twister algorithm, to retrieve uniformly distributed values from the `dist_` distribution in the range of -1 to 1. `tick()` returns a random white noise sample.

Chapter 3

Modulating Sound

One of the most interesting aspects of any synthesizer, digital as well as analog, is its capability to modify or *modulate* sound in a variety of ways. To modulate a sound means to change its amplitude, pitch, timbre, or any other property of a signal to produce an, often entirely, new sound. This chapter will examine and explain two of the most popular means of modulation in a digital synthesis system, Envelopes and Low Frequency Oscillators (LFOs).

3.1 Envelopes

When a pianist hits a key on a piano, the amplitude of the sound produced increases from zero, no sound, to some maximum amplitude which depends on a multitude of factors such as how hard the musician pressed the key, what material the piano string is made of, the influence of air friction and so on. After reaching the maximum loudness, the amplitude decreases until the piano string stops oscillating, resulting in renewed silence. To model such an evolution of amplitude over time, digital musicians use a modulation technique commonly referred to as an "Envelope".

3.1.1 Envelope segments

The following sections outline the creation of and terminology used for single segments of an envelope.

ADSR

A common concept associated with Envelopes is "ADSR", which stands for "Attack, Decay, Sustain, Release". These four terms name the four possible states an Envelope segment can take on. An Attack segment is any segment where the initial amplitude, at the start of a segment, is less than the final amplitude, at the end of the segment — the amplitude increases. Conversely, a Decay segment signifies a decrease in amplitude from a higher to a lower value. When the loudness of a signal stays constant for the full duration of an interval, this interval is termed a "Sustain" segment. While the three segment types just mentioned all describe the modulation of a signal's loudness when the key of a piano or synthesizer is still being pressed, the last segment type, a "Release" segment, refers to the change in loudness once the key is *released*. Figure 3.1 depicts an abstract representation of a typical ADSR envelope. Figure 3.2 shows a 440 Hz sine wave before the application of an ADSR envelope and Figure 3.3 displays the same signal after an ADSR envelope has been applied to it.

img/adsr.png

Figure 3.1: An Envelope with an Attack, a Decay, a Sustain and finally a Release segment. Source: `http://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/ADSR_parameter.svg/500px-ADSR_parameter.svg.png`

img/envb.png

Figure 3.2: A 440 Hz sine wave.

img/enva.png

Figure 3.3: The same signal from Figure 3.2 with an ADSR Envelope overlayed on it.

img/rates.png

Figure 3.4: An envelope where r is equal to 2, then to 1, then to 0.5.

Table 3.1: Two member functions of the EnvSegSeq class (Envelope Segment Sequence).

Mathematical Definition and Rate Types

Mathematically, an Envelope segment can be modeled using a simple power function of the general form presented in Equation 3.1, where a_{final} is the final amplitude at the end of the segment, a_{start} the initial amplitude at the beginning of the segment and r the parameter responsible for the shape or "rate" of the segment. When r, which must kept between 0 and ∞ (practically speaking some value around 10), is equal to 1, the segment has a linear rate and is thus a straight line connecting the initial amplitude with the final loudness. If r is greater than 1, the function becomes a power function and consequently exhibits a non-linear increase or decrease in amplitude. Lastly, if r is less than 1 but greater than 0, the function is termed a "radical function", since any term of the form $x^{\frac{a}{b}}$ can be re-written to the form $\sqrt[b]{x^a}$, where the numerator a becomes the power of the variable and the denominator b the radicand. Figure 3.4 displays an envelope whose first segment has $r = 2$, a quadratic increase, after which the sound decays linearly, before increasing again, this time r being equal to $\frac{1}{2}$ (a square root function). A C++ class for single Envelope segments is shown in Listing **??**.

$$a(t) = (a_{final} - a_{start}) \cdot t^r + a_{start} \tag{3.1}$$

3.1.2 Full Envelopes

Creating full Envelopes with a variable number of segments requires little more work than implementing a state-machine, which checks whether the current sample count is greater than the length of the Envelope segment currently active. If the sample count is still less than the length of the segment, one retrieves Envelope values from the current segment, else the Envelope progresses to the next segment. Additionally, it should be possible for the user to loop between segments of an Envelope a variable number of times before moving on to the release segment. Table 3.1 displays two member functions of an Envelope class created for this thesis, which return an Envelope value from the current Envelope segment and allow for the updating of the sample count.
Envelopes have many uses. Some require flexible segments which allow for the adjusting of individual segments' lengths, others need all segments to have a constant length. Some give the user the possibility to modulate individual segments by making them behave like a sine function, others do not. Fortunately, C++'s inheritance features make it very easy and efficient to construct such a variety of different classes that may or may not share relevant features. The

Figure 3.5: The inheritance diagram for the `Envelope` class. The `Unit` and `ModUnit` classes are two abstract classes that will be explained in later parts of this thesis.

inheritance diagram for the final `Envelope` class created for the purpose of this thesis reflects how all of these individual class can play together to yield the wanted features for a class. This inheritance diagram is displayed in Figure 3.5.

3.2 Low Frequency Oscillators

A Low Frequency Oscillator, abbreviated "LFO", is an oscillator operating at very low frequencies, typically in the range below human hearing (0-20 Hz), used solely for the purposes of modulating other signals. The most common parameter to be modulated by an LFO in a synthesizer is the amplitude of another oscillator, to produce effects such as the well known "vibrato" effect. In this case, were the frequency of the LFO to be in the audible range, one would use the term Amplitude Modulation (AM), which is also a method for synthesizing sound. Equation 3.2 shows how an LFO can be used to change the amplitude of another oscillator[1]. Figure 3.6 shows a 440 Hz sine wave, Figure 3.7 displays the same sine wave now modulated by a 2 Hz LFO and in Figure 3.8 the frequency of the LFO has been increased to 20 Hz to produce a vibrato effect. It should be noted that an LFO can also modulate any other parameter, such as the rate of an Envelope segment.

$$O(t) = (A_{osc} + (A_{lfo} \cdot sin(\omega_{lfo}t + \phi_{lfo}))) \cdot sin(\omega_{osc}t + \phi_{osc}) \tag{3.2}$$

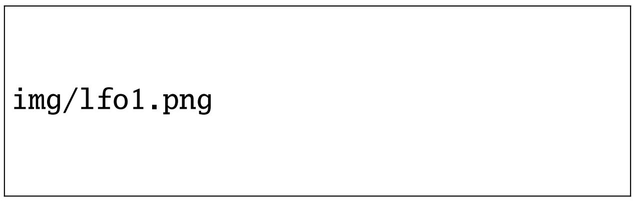

Figure 3.6: A 440 Hz sine wave.

[1]This is the same equation as for AM.

img/lfo2.png

Figure 3.7: The sine wave from Figure 3.6 modulated by an LFO with a frequency of 2 Hertz. Note how the maximum amplitude of the underlying signal now follows the waveform of a sine wave. Because the LFO's amplitude is the same as that of the oscillator (the "carrier" wave), the loudness is twice as high at its maximum, and zero at its minimum.

img/lfo3.png

Figure 3.8: The sine wave from Figure 3.6 modulated by an LFO with a frequency of 20 Hertz. This signal is said to have a "vibrato" sound to it.

Concerning the implementation of an LFO in a computer program, the process could be as simple as re-naming a class used for an oscillator:

In the synthesizer created for this thesis, called *Anthem*, the distinction between an LFO and an oscillator is the possibility to modulate an LFO's parameters, for example using an Envelope or another LFO, whereas the `Oscillator` class is an abstract base class whose sole purpose is to be an interface to a Wavetable. This property of the `Oscillator` class is used by both the `LFO` and the `Operator` class, who both inherit from the `Oscillator` class. The `Operator` class is the sound generation unit the user actually interfaces with from the Graphical User Interface (GUI) of *Anthem*. It is derived from the `Oscillator` class because it ultimately needs to generate sound samples, but it is its own class because it has various other features used for sound synthesis. These features are discussed in a later chapter. The relationships just described lead to the inheritance diagram shown in Figure 3.9.

img/osc-derived.png

Figure 3.9: Inheritance diagram showing the relationship between an `LFO`, an `Operator` and their base class, the `Oscillator` class.

3.3 The ModDock system

The majority of digital synthesizers, such as Propellerhead's[2] *Thor* or Native Instruments'[2] *FM8* synthesizer, implement modulation in a static way. Instead of making it possible to use an LFO or an Envelope to modulate any parameter in a synthesis system, units, e.g. an oscillator, have dedicated LFOs and Envelopes, which modulate only one parameter — mostly amplitude — and only for this unit. On the other hand, some synthesizers like *Massive*, also created by Native Instruments, implement a system where LFOs and Envelopes can be used by a variable number of units, for a variable number of parameters. For this thesis and the synthesizer created for it, such a system, here called the "ModDock" system, was emulated. The following sections will outline the process of defining and implementing the ModDock system.

3.3.1 Problem statement

In short, the ModDock system should make it possible to modulate a variable number of parameters of a variable number of units of the synthesizer, with any of a fixed number of LFOs, Envelopes or similar *Modulation Units*. Consequently, each unit in the synthesis system should have what will be known as a "ModDock", a set of "docks", the number of which depends on the number of parameters the unit makes it possible to modulate, where the user may insert a Modulation Unit. Due to the fact that many units may be modulated by one Modulation Unit, the Modulation Unit must not update its value until all units have been modulated by it. For example, the sample count of an Envelope must stay the same until all dependent units have retrieved the current Envelope value from it. Moreover, a unit should be able to adjust the depth of modulation by a Modulation Unit, i.e. it should be possible to have only 50% of an LFO's signal affect the parameter it is modulating. Finally, there should be a way of *side-chaining* Modulation Units so that one Modulation Unit in a dock modulates the depth of another Modulation Unit in that same dock, the signal of which may again side-chain the depth of another Modulation Unit in that ModDock. The final modulation of a parameter will be the average of all modulation values of a ModDock affecting that parameter.

3.3.2 Implementation

In order to let units of the synthesizer created for this thesis share certain behaviour and member functions, as well as to distinguish between the traits and tasks certain units have that others do not, it was necessary to develop an inheritance structure that would satisfy these requirements. In *Anthem*, a *Unit* is defined as any object with modulateable parameters. A Unit has necessary member variables and functions to permit its parameters to be modulated by other *Modulation Units*. A Modulation Unit, short *ModUnit*, shall be defined as any Unit that can modulate a parameter of a Unit. The ModUnit class includes the *pure virtual*[3] `modulate` method, which takes a sample as one of its arguments and then returns a modulated version of that sample. The form of modulation depends entirely on the implementation of the `modulate` method by the class that inherits from the ModUnit class, as the ModUnit class itself does not implement any standard way of modulating a sample. Figure 3.10 displays the inheritance diagram for the Unit class.

[2]"Propellerhead Software is a music software company, based in Stockholm, Sweden, and founded in 1994." Source: *Wikipedia* `http://en.wikipedia.org/wiki/Propellerhead_Software`. Homepage: `https://www.propellerheads.se`

[3]In C++, a pure virtual function is a function that does not have any standard implementation and thus requires the derived classes of the class that declares the pure virtual function to implement that function on their own. A class that declares a pure virtual method is termed an *abstract* class and may not be instantiated.

img/unit-inherit.png

Figure 3.10: The inheritance diagram of the Unit class. GenUnits are Units that generate samples. EffectUnits apply some effect to a sample, e.g. a sample delay.

The `modulate` method

All classes derived from the ModUnit class must implement the `modulate` method, which takes the sample to modulate, a depth value between 0 (no modulation) and 1 (full modulation) as well as a maximum boundary as its arguments. In the case of LFOs, the maximum boundary parameter determines the value that is added to the sample. For example, when the amplitude of a unit is modulated, the maximum boundary is 1, meaning that the value added to the sample is in the range of $[-A_{LFO} \cdot 1; A_{LFO} \cdot 1]$, whereas for the rate parameter of Envelope segments, where the maximum boundary is 10, the range is $[-A_{LFO} \cdot 10; A_{LFO} \cdot 10]$. The declaration of the `modulate` method in the ModUnit class is given below.

This method was the main motivation behind the creation of the ModUnit class and is especially important for ModDocks, as it makes it possible to polymorphically access the `modulate` method of any ModUnit through a pointer-to-ModUnit (`ModUnit*`). This means that each ModUnit can have its own definition of what it means to modulate a sample. Table 3.2 shows the definition of the `modulate` method in the LFO and Table 3.3 in the Envelope class.

Table 3.2: The implementation of the `modulate` method for LFOs.

Table 3.3: The implementation of the `modulate` method for Envelopes. Note that for the Envelope class, the paramter `maximum` is not relevant, which is why it is never used. This shows that all that matters is that the method returns a modulated sample — what "modulate" means is up to the class that implements it.

ModDocks

A ModDock is simply a collection of ModUnits. The ModDock collects all the modulated samples of individual ModUnits in the dock and returns an average over all samples. For example, if one LFO adds a value of 0.3 to the amplitude parameter of a Unit with a current amplitude of 0.5 and another subtracts a value of 0.1, the final amplitude of that Unit will be 0.6, as $\frac{(0.5+0.3)+(0.5-0.1)}{2} = 0.6$. Moreover, this means that if the two LFOs were to add/subtract the same absolute value but with a different sign, for example -0.4 and 0.4, the net difference would be 0 and the amplitude would remain 0.5. Something else the ModDock takes care of is boundary checking and value adjustment. Meaning that, continuing the example given above, were an LFO to add a value of ± 1 to the base amplitude of 0.5, the amplitude would not oscillate in the range $[-1.5; 1.5]$. Rather, the ModDock ensures that the value trespasses neither the maximum boundary nor the minimum boundary, which is another parameter supplied to the ModDock. Therefore, the amplitude value would remain in the optimal range of $[-1; 1]$. Alongside the maximum and the minimum boundary, the Unit who owns the ModDock can pass the current base value of the parameter to be modulated to the ModDock. In the aforementioned example, the base value would be 0.5. Also, the ModDock can store the depth of modulation of individual ModUnits. Because both the `depth` and the `maximum` parameter of the `modulate` method for ModUnits are now stored in an instance of the `ModDock` class, the `modulate` method has a much simpler declaration in the `ModDock` class:

What this simplification of the `modulate` method requires, however, is that the maximum and minimum boundary as well as an initial base value are passed to the relevant ModDock in the construction of the Unit that owns it. Additionally, the ModDock must be notified whenever the base value changes. Table 3.4 shows how these parameters may be passed to a ModDock and updated when necessary.

Table 3.4

Sidechaining

One of the most interesting and equally complicated tasks encountered when creating the ModDock system was the implementation of side-chaining. Side-chaining makes it possible to have one ModUnit in a ModDock modulate the depth parameter of another ModUnit in that same ModDock. Borrowing from the terminology of digital communication, a ModUnit that side-chains another ModUnit is termed a "Master" and the ModUnit being side-chained is called a "Slave". Moreover, it was decided that any ModUnit that is not a Master shall be called a non-Master. Therefore, a non-Master is either a Slave or not involved in any side-chaining relationship at all — a normal ModUnit. It should be noted that the signal of a Master does not affect the final modulation value of a ModDock directly, i.e. the modulation value of a Master is not taken into consideration during the averaging process described above, but only indirectly, by modulating the depth of a Slave. Therefore, a ModUnit can be either a Master and contribute to the final modulation indirectly or be a non-Master and contribute to the final value directly. It should be noted that it is entirely possible to have one Master modulate multiple Slaves and for one Slave to have multiple Masters. Figure 3.11 depicts these relationships in a flow-chart. Figure 3.12 shows a scan of early sketches created while implementing side-chaining.

What Figure 3.12 also displays is that there are two possible ways to implement side-chaining. The first method, which was ultimately not chosen, is to give each ModUnit in the ModDock its own ModDock where Masters could be inserted. The second method involves internally linking Masters and Slaves within the ModDock. This second implementation was finally decided to be the better one as it does not require the instantiation of a full new ModDock for each ModUnit, while providing the same functionality. Listing **??** shows all private members of the `ModDock` class. Special attention should be payed to the `ModItem` struct, which is essential to the implementation of side-chaining and is very similar to the concept of a Linked-List. Each `ModItem` stores the aforementioned pointer-to-ModUnit to access the `modulate` method of the ModUnit. Moreover, there is one vector of indices for all Masters of that particular ModItem and one vector of indices for all of its Slaves. These indices refer to the positions of Masters/Slaves in the vector where all ModItems are stored, the `modItems_` vector. When the user sets up a side-chain relationship between two ModItems of a ModDock, the index of the Slave is added to the Slave vector of the Master and the index of the Master is inserted into the Master vector of the Slave. Should the user wish to "un-side-chain" two ModItems, their indices are removed from the each other's appropriate vector. Furthermore, the `ModItem` struct holds a baseDepth variable. This variable is similar to the baseValue of the ModDock, in the sense that is the ModItem's original depth which serves as the base value for modulation by other ModItems. The modulation of a Slave by its Masters is implemented in the exact same way that a parameter of a Unit is modulated by a ModDock's ModUnits. Modulated Slave samples are summed and averaged over their number. Listing **??** gives the full definition of the `modulate` method of the ModDock class. In lines 9 to 35, Slaves are modulated by their Masters. Subsequently, in lines 39 to 62, the sample passed to the function from the Unit who owns the ModDock is modulated by all non-Masters and then finally returned.

img/flowsc.png

Figure 3.11

img/scsketch.jpeg

Figure 3.12

Chapter 4

Synthesizing Sound

The most important feature of a synthesizer is its capability to synthesize sound. Synthesizing sound means to combine two or more (audio) signals to produce a new signal. The many possibilities to synthesize sound waves with variable parameters enable musicians to create an uncountable number of different sounds. A synthesizer can be configured to resemble natural sounds such as that of wind or water waves, can emulate other instruments like pianos, guitars or bass drums and, finally, a synthesizer can also be used to produce entirely new, electronic sounds. However, not all synthesis methods available to the creator of a digital synthesizer are equally suited to the various possible sounds just described. Common methods of synthesis are Additive Synthesis, Granular Synthesis, Amplitude Modulation Synthesis and Frequency Modulation Synthesis. This chapter aims to examine and describe the latter.

4.1 Frequency Modulation Synthesis

In Frequency Modulation (FM) Synthesis, one signal, termed the "modulator", is used to modulate the frequency of another signal — the "carrier". This produces very complex changes in the carrier's frequency spectrum and introduces a theoretically infinite set of side-bands, which contribute to the characteristic sound and timbre of FM Synthesis. The fact that Frequency Modulation can be used to synthesize audio signals was first discovered by John Chowning, who he described the mathematical principles and practical implications of FM Synthesis in his 1973 paper "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation". In 1974 his employer, Stanford University, licensed his invention to the Yamaha Corporation, a Japanese technology firm, which went on to create the first FM synthesizers, including the very popular DX-7 model. Many digital FM synthesizers, such as Native Instrument's *FM8*, try to emulate the DX-7. (Electronic Music Wiki, `http://electronicmusic.wikia.com/wiki/DX7`, accessed 4 January 2015) Equation 4.3 gives a full mathematical definition for Frequency Modulation (Synthesis). What Equation 4.3 shows is that FM Synthesis works by summing the carrier's instantaneous frequency ω_c, the carrier signal being $c(t)$, with the output of the modulator signal $m(t)$, thereby varying the carrier frequency periodically. The degree of frequency variation Δf_c depends on the modulator's amplitude A_m. Therefore, $\Delta f_c = A_m$. Figure 4.1 shows how frequency modulation effects a signal. Note that this Figure should only show how FM works. It is not a realistic example of FM *Synthesis*, as the modulator frequency is not in the audible range.

img/fmlow.png

Figure 4.1: The first figure shows the carrier signal $c(t)$ with its frequency f_c equal to 100 Hz. The signal beneath the carrier is that of the modulator, $m(t)$, which has a frequency f_m of 5 Hz. When the modulator amplitude A_m is increased to 50 (instead of ~ 0.4, as shown here) and used to modulate the frequency of the carrier, the last signal $f(t)$ is produced. While these figures show how FM works, they are not a good example of FM synthesis, because the modulator frequncy is not in the audible range.

$$c(t) = A_c \cdot \sin(\omega_c t + \phi_c) \qquad (4.1) \qquad\qquad m(t) = A_m \cdot \sin(\omega_m t + \phi_m) \qquad (4.2)$$

$$f(t) = A_c \cdot \sin((\omega_c + m(t))t + \phi_c) = A_c \cdot \sin((\omega_c + A_m \cdot \sin(\omega_m t + \phi_m))t + \phi_c) \qquad (4.3)$$

4.2 Sidebands

One of the most noticeable effects of FM Synthesis is that it adds *sidebands* to a carrier signal's frequency spectrum. Sidebands are frequency components higher or lower than the carrier frequency, whose spectral position, amplitude as well as relative spacing depends on two factors: the ratio between the carrier and modulator frequency, referred to as the "C:M ratio", and the index of modulation β, which in turn depends on the modulator signal's amplitude and frequency.

4.3 C:M Ratio

The spacing and positions of sidebands on the carrier signal's frequency spectrum depends on the ratio between the frequency of the carrier signal, C, and that of the modulator, M. This $C : M$ ratio gives insight into a variety of properties of a frequency-modulated sound. Most importantly, when the $C : M$ ratio is known, Equation 4.4 gives all the relative frequency values of the theoretically infinite set of sidebands. Equation 4.5 describes how to calculate ω_{sb_n}, the angular frequency of the n-th sideband, absolutely. What these equations show is that for any given $C : M$ ratio, the sidebands are found at relative frequencies $C + M, C + 2M, C + 3M, ...$ and absolute frequencies $\omega_c + \omega_m, \omega_c + 2\omega_m, \omega_c + 3\omega_m, ...$ Hertz.

$$\omega_{sb_n} = C \pm n \cdot M, \text{where } n \in [0; \infty[\tag{4.4}$$

$$\omega_{sb_n} = \omega_c \pm n \cdot \omega_m, \text{where } n \in [0; \infty[\tag{4.5}$$

When examining these two equations one may notice that there also exist sidebands with negative frequencies, found relatively at $C - M, C - 2M, C - 3M$ and so on. Simply put, a signal with a negative frequency is equal to its positive-frequency counterpart but with inverted amplitude, which can also be seen as a 180° or π radian phase-shift (`http://www.sfu.ca/~truax/fmtut.html`, accessed 30 December 2014). Equation 4.6 defines this formally. What this also means is that if there is a sideband at a frequency f and another sideband at $-f$ Hertz, these two sidebands will phase-cancel completely if their amplitudes are the same. If not, the positive side-band will be reduced in amplitude proportionally. Consequently, it may happen that also the original carrier frequency is reduced in amplitude, for example at a $C : M$ ratio of 1:2, as the first "lower" sideband, meaning with a lower frequency than the carrier, is at $C - M = C - 2C = -C$ Hertz.

$$A \cdot \sin(-\omega t + \phi) = -A \cdot \sin(\omega t + \phi) = A \cdot \sin(\omega t + \phi - \pi) \tag{4.6}$$

Figure 4.2 shows the frequency spectrum of a carrier signal with a frequency f_c[1] of 200 Hz, modulated by a modulator signal with its frequency f_m equal to 100 Hz. The carrier frequency is seen on the spectrum as the peak at 200 Hz. The other peaks are the sidebands. Because the $C : M$ ratio here is $2 : 1$, the first two lower sidebands are found at relative positions $C - M = 2 - 1 = 1$ and $C - 2M = 2 - 2 = 0$, relative position 2 being the carrier. The first two upper sidebands have absolute frequency values of $\omega_c + \omega_m = 200 + 100 = 300$ and $\omega_c + 2 \cdot \omega_m = 200 + 200 = 400$ Hertz.

[1]Not to be confused with the cutoff frequency in the context of filters, which is also denoted by f_c

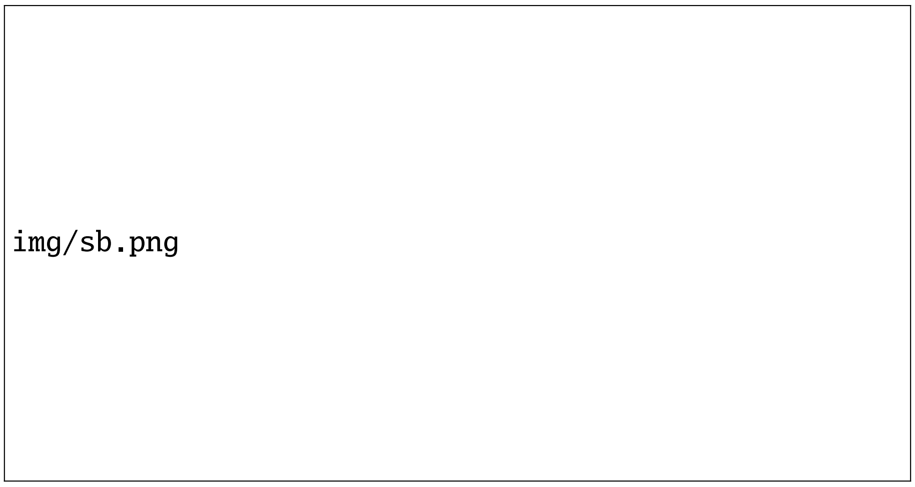

Figure 4.2: The frequency spectrum of a $C : M$ ratio of $2 : 1$, where carrier frequency f_c is 200 and the modulator frequency f_m 100 Hertz.

4.4 Index of Modulation

Theoretically, Frequency Modulation Synthesis produces an infinite number of sidebands. However, sidebands reach inaudible levels very quickly, making the series practically finite. In general, the amplitude of individual sidebands depends on the "index of modulation" and the Bessel Function values that result from it. A definition for the index of modulation, denoted by β, is given in Equation 4.7. Because the variation in carrier frequency, Δf_c, depends directly on the amplitude of the modulator, Equation 4.7 can be re-written as Equation 4.8.

$$\beta = \frac{\Delta f_c}{f_m} \tag{4.7}$$

$$\beta = \frac{\Delta f_c}{f_m} = \frac{A_m}{f_m} \tag{4.8}$$

The index of modulation can be used to determine the amplitude of individual sidebands, if input into the Bessel Function, shown in Equation 4.9, where n is the sideband to calculate the amplitude for. Table 4.1 shows amplitude values for the n-th sideband, given an index of modulation β. These values were derived from the Bessel Function. Only values above 0.01 are shown.

$$J(n, \beta) = \sum_{k=0}^{\infty} \frac{(-1)^k \cdot (\frac{\beta}{2})^{n+2k}}{(n+k)! \cdot k!} \tag{4.9}$$

4.5 Bandwidth

The bandwidth of a signal describes the range of frequencies that signal occupies on the frequency spectrum. The bandwidth of a frequency-modulated signal is theoretically infinite. However, it was already shown that the amplitude values of a carrier's sidebands quickly become inaudible and thus negligible. A rule of thumb for calculating the bandwidth B of an FM signal is the so-called "Carson Bandwidth Rule", given in Equation 4.10.

$$B \approx 2(\Delta f_c + f_m) = 2(A_m + f_m) = 2f_m(1 + \beta) \tag{4.10}$$

β	Sideband								
	Carrier	1	2	3	4	5	6	7	8
0	1								
0.25	0.98	0.12							
0.5	0.94	0.24	0.03						
1.0	0.77	0.44	0.11	0.02					
1.5	0.51	0.56	0.23	0.06	0.01				
2.0	0.22	0.58	0.35	0.13	0.03				
3.0	-0.26	0.34	0.49	0.31	0.13	0.04	0.01		
4.0	-0.40	-0.07	0.36	0.43	0.28	0.13	0.05	0.02	
5.0	-0.18	- 0.33	0.05	0.36	0.39	0.26	0.13	0.05	0.02

Table 4.1

4.6 Algorithms and Operators

When speaking of FM Synthesis, it is common to refer to oscillators as "operators". This naming convention stems from the Yamaha DX-7 series. So far, FM Synthesis was only discussed for two operators, a carrier and a modulator. However, it is entirely possible to perform FM Synthesis with more than two operators, by modulating any number of operators either in series, or in parallel. When three operators A, B and C are connected in series, A modulates B, which in turn modulates C. If A and B are connected in parallel, but in series with C, C is first modulated by A and then by B. The same result is achieved if the sum of signals A and B is used to modulate C. In general, a configuration of operators is referred to as an "algorithm". The number of possible algorithms increases with an increasing number of operators. In the synthesizer created for this thesis, four operators A, B, C and D are used. The possible algorithms for these four operators are shown in Figure 4.3.

4.7 Implementation

The `Operator` class implements an FM operator and inherits from the `Oscillator` class. Its frequency is modulated by adding an offset to its Wavetable index increment, proportional to the frequency variation caused by the modulator. Table 4.2 shows how this offset is calculated and added to the Wavetable index whenever the Operator is updated. Listing **??** shows how the `FM` class, which takes pointers to four Operators, implements the various algorithms shown in Figure 4.3.

Table 4.2: Two member functions from the `Operator` class that show how the frequency of an `Operator` object can be modulated.

img/fmalgs.png

Figure 4.3: Signal flows for FM Synthesis algorithms with four operators, A, B, C and D. A direct connection between two operators means modulation of the bottom operator by the top operator. Signals of operators positioned side-by-side are summed.

Conclusion

Digital music synthesis and audio processing are ongoing fields of study. New synthesis methods — such as physical modelling — as well as incumbent ones are actively being researched and perfected, to provide ever crisper and higher-quality sound. Reverberation algorithms implemented in professional audio software have become many times more complex than the Schroeder Reverb presented in chapter 6, making digital sound resemble its natural counterpart ever more strikingly. Filters are continuously being improved as well, ensuring that frequency responses have the fastest possible roll-off in the transition band, highest amount of attenuation in the stop band and least in the pass band.

This thesis clearly did not attempt to examine such state-of-the-art techniques or cutting-edge research. An entire thesis would have to be dedicated to every chapter — if not every section — presented here, if it intended to examine what current methods yield the best possible results. Rather, what this thesis aimed at was describing the most straight-forward and well-established practices, that can be ameliorated and refined by the interested reader in the future.

However, one may argue that attempting to build a complete synthesis system that implements not the simplest techniques, but those that produce best results, is an undertaking too large for a single person. For this reason, the synthesizer created for the purpose of this thesis, *Anthem*, will be open-sourced and made available to the online community, for free and with no restrictions. The goal is to make *Anthem* the go-to alternative to commercial music synthesizers, built by audio programming enthusiasts from all over the world, who will hopefully improve on the code base written for this thesis. The journey does not end here. It begins.