Developing a Digital Synthesizer in C++

Peter Goldsborough
petergoldsborough@hotmail.com

December 29, 2014

Contents

Chapter 1

Sound in the Digital Realm

1.1 What is a Music Synthesizer?

1.2 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.

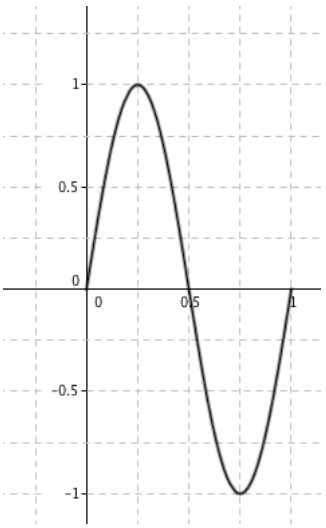


Figure 1.1: The continuous representation of a typical sine wave. In this case, both the signal's frequency f as well as the maximum elongation from the equilibrium a are equal to 1.

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 (Mitchell, 2008, p. 16). 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) (Mitchell, 2008, p. 16). 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* (Mitchell, 2008, p. 17). 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.

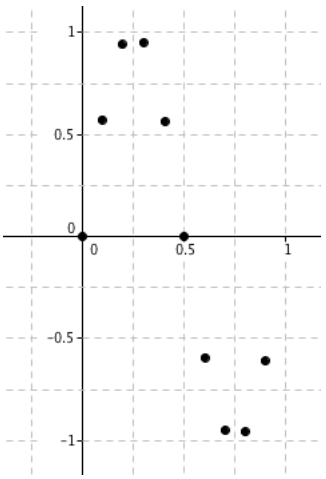


Figure 1.2: The discrete representation of a typical sine wave.

1.3 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. The value of the sample rate is given 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:

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

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

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

1.4 Nyquist Limit

The sample rate determines the range of frequencies that can be represented by a digital sound system, as only frequencies that are less than or equal to one half of the sampling rate, where it is possible to take at least one sample above the equilibrium and at least one sample below the equilibrium for every cycle (Mitchell, 2008, p. 18), 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*. 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" (Smith, 1999, p. 40). Any frequencies above the Nyquist limit lead to *aliasing*, which is discussed in the next section.

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 audio frequency range, since a sample rate of 40 kHz 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.

1.5 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, which, when reconstructed, 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 samples per second, 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).

1.6 Overflow

Another interesting property of digital sound, which is not encountered in the analog world, is that it can overflow. When we attempt to increase the loudness of something in the analog world, e.g. by hitting a drum more intensely, the expected result is a louder sound. In the digital realm however, it may occur that attempting to increase a signal's amplitude does not result in an increased loudness, but in distortion. The cause of this phenomenon lies in the way digital audio is stored. Since computer memory is a finite resource, each sample has a dedicated portion of computer memory allocated to it. For example, the Waveform Audio File Format (WAVE), a common computer file format for audio data, stores each sample of an audio track as a 16-bit signed integer. A 16-bit signed integer gives a possible range of -2^{16-1} to 2^{16-1} ($16 - 1$ because the most significant bit is used as the sign bit in two's complement

representation). This means that a signal with an amplitude of 1 will be stored as 32767, an amplitude of 0.5 as 16384, an amplitude of -1 as -32768 and so on. If one tries to increase the amplitude of a signal whose value has already saturated the available range and space allocated to it, in this case 32767 on the positive end and -32768 on the negative end, the result is that the integer with which the sample is stored *overflows*. Because WAVE files (and many other storage media) store samples as *signed* integers, overflow always results in a change of sign:

$$32767_{10} = 0111111111111111_2$$

$$0111111111111111_2 + 1 = 1000000000000000_2 = -32768_{10}$$

$$1000000000000000_2 - 1 = 0111111111111111_2 = 32767_{10}$$

A visualization of the result of increasing the amplitude of a signal with a saturated value (an amplitude of 1), shown in Figure 1.6, is given in Figure 1.7.

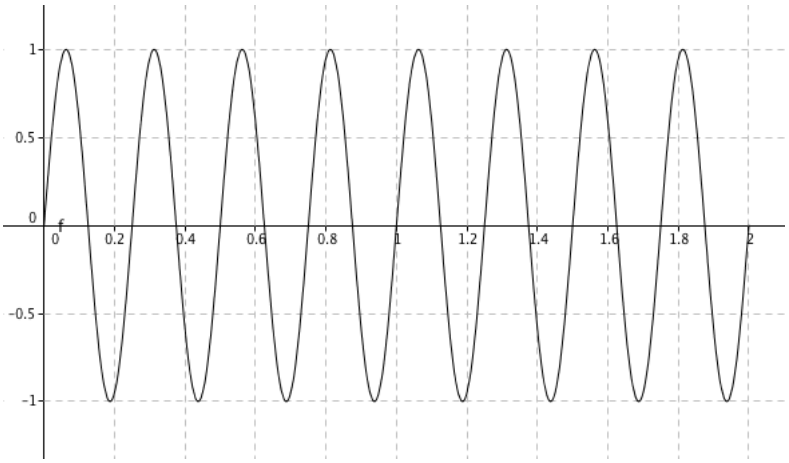


Figure 1.3: A sinusoid with a frequency of 4 Hz.

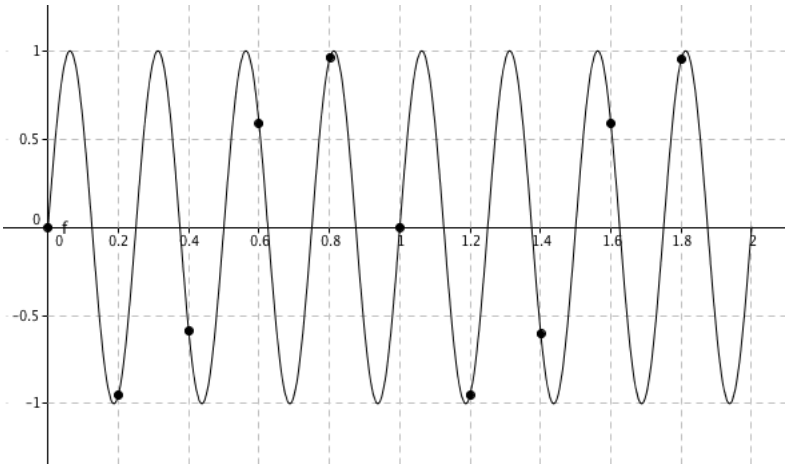


Figure 1.4: A sinusoid with a frequency of 4 Hz, sampled at a sample rate of 5 Hz. According to the Nyquist Theorem, this signal is sampled improperly, as the frequency of the continuous signal is not less than or equal to one half of the sample rate, in this case 2.5 Hz.

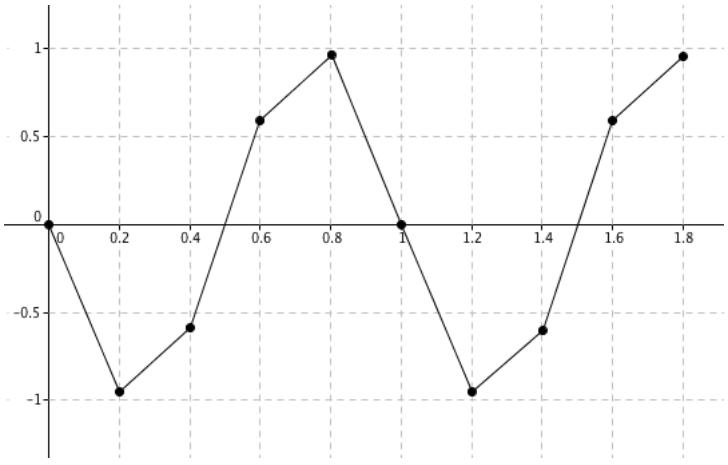


Figure 1.5: An approximated representation of the signal created from the sampling process shown in Figure 1.4. Visually as well as acoustically, the new signal is completely different from the original signal. However, in terms of sampling, they are indistinguishable from each other due to improper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

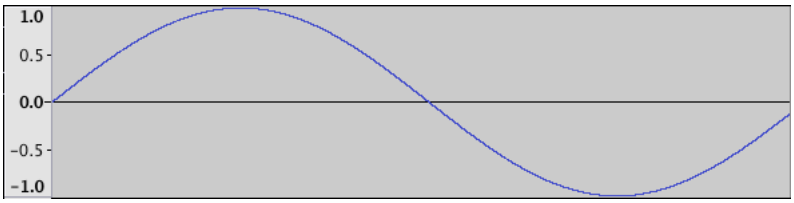


Figure 1.6: A typical sinusoidal signal with an amplitude of 1. The integer range provided by the allocated memory for the top-most sample is saturated, meaning it is equal to 32767_{10} or 0111111111111111_2.

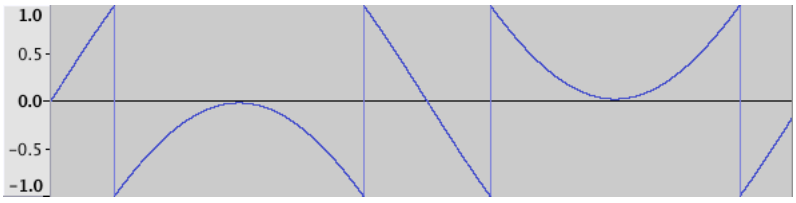


Figure 1.7: What happens when the amplitude of the signal from Figure 1.6 is increased by a factor of 2. Multiple samples have overflowed and thus changed their sign. Because of the way two's-complement representation is implemented, the signal continues it's path as if no overflow had ever occurred. The only difference being, of course, that the sign has changed mid-way.

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 waveform is the 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 A.1. 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:

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

Therefore, the program from Listing A.1 could be modified to compute a cosine wave by changing line 22 from:

```
22 |   double phase = 0;
```

to

```
22 |   double phase = pi/2.0;
```

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 or additive synthesis.

2.2.1 Mathematical Calculation of Complex Waveforms

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 the maximum amplitude exactly one half of a period and equal to the 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. Nevertheless, they are considerably useful for modulating other signals, as tiny acoustical imperfections such as those found in additively synthesized waveforms can result in unwanted distortion which is not encountered when using mathematically calculated waveforms. Therefore, exact waveforms are the best choice for modulation sources such as Low Frequency Oscillators (LFOs), which are discussed in later chapters. (Mitchell, 2008, p. 71)

The following paragraphs will analyze how four of the most common waveforms found in digital synthesizers, the square, the sawtooth, the ramp and the triangle wave, can be generated via mathematical calculation.

Note: There have found to be disparities in literature over which waveform is a sawtooth and which a ramp wave. This thesis will consider a sawtooth wave as descending from maximum to minimum amplitude with time and a ramp wave as ascending from minimum to maximum amplitude with time.

Square Waves

Ideally, a square wave is equal to its maximum amplitude for exactly one half of a period and equal to its minimum amplitude for the other half of the same period. A single period of a square wave can be calculated as shown in Equation 2.3, where the independent variable t as well as the period T can be either in samples or in seconds. A mathematical Equation for a full, periodic square wave function is given by Equation 2.4, where t is time in seconds and the frequency f in Hertz. An equivalent C++ computer program is shown in Table 2.1.

$$f(t) = \begin{cases} 1, & \text{if } 0 \leq t < \dfrac{T}{2} \\ -1, & \text{if } \dfrac{T}{2} \leq t < T \end{cases} \qquad (2.3)$$

$$f(t) = \begin{cases} 1, & \text{if } \sin(2\pi f t) > 0 \\ -1, & \text{otherwise} \end{cases} \qquad (2.4)$$

Sawtooth Waves

An ideal sawtooth wave descends from its maximum amplitude to its minimum amplitude linearly before jumping back to the maximum amplitude at the beginning of the next period. A mathematical Equation for a single period of such a sawtooth wave function, calculated directly from the phase, is given by Equation 2.5 (Mitchell, 2008, p. 68). Alternatively, the function can depend on time or on samples, as shown by Equation 2.6, where T is the period. A computer program to compute one period of a sawtooth wave is given in Table 2.2.

$$f(\phi) = \left\{ -\frac{\phi}{\pi} + 1, \quad \text{if } 0 \leq \phi < 2\pi \right. \qquad (2.5)$$

$$f(t) = \left\{ -\frac{2t}{T} + 1, \quad \text{if } 0 \leq t < 1 \right. \qquad (2.6)$$

```
double* square(const unsigned int period)
{
    // the sample buffer
    double * buffer = new double [period + 1];

    // time for one sample
    double sampleTime = 1.0 / period;

    // the midpoint of the period
    double mid = 0.5;

    double value = 0;

    // fill the sample buffer
    for (int n = 0; n < period; n++)
    {
        buffer[n] = (value < mid) ? -1 : 1;

        value += sampleTime;
    }

    return buffer;
}
```

Table 2.1: C++ code to generate and return one period of a square wave, where `period` is the period duration in samples. Note that this function increments in sample time, measured in seconds, rather than actual samples. This prevents a one-sample quantization error at the mid-point, since time can always be halved whereas a sample is a fixed entity and cannot be broken down any further.

```
double* sawtooth(const unsigned int period)
{
    // the sample buffer
    double * buffer = new double [period];

    // how much we must decrement the
    // index by at each iteration
    double incr = -2.0 / period;

    double value = 1;

    for (int n = 0; n < period; n++)
    {
        buffer[n] = value;

        value += incr;
    }

    return buffer;
}
```

Table 2.2: C++ code to generate one period of a sawtooth wave function, where `period` is the period duration in samples.

Ramp Waves

A ramp wave is, quite simply, an inverted sawtooth wave. It ascends from its minimum amplitude to its maximum amplitude linearly, after which it jumps back down to the minimum amplitude. Consequently, Equation 2.7 and 2.8 differ from sawtooth Equation 2.5 and 2.6 only in their sign and offset. The equivalent C++ implementation shown in Table 2.3 also reflects these differences.

$$f(\phi) = \begin{cases} \frac{\phi}{\pi} - 1, & \text{if } 0 \leq \phi < 2\pi \end{cases} \qquad (2.7)$$

$$f(t) = \begin{cases} \frac{2t}{T} - 1, & \text{if } 0 \leq t < 1 \end{cases} \qquad (2.8)$$

```
1  double* ramp(const unsigned int period)
2  {
3      // the sample buffer
4      double * buffer = new double [period];
5
6      // index incrementor
7      double incr = 2.0 / period;
8
9      double value = -1;
10
11     for (int n = 0; n < period; n++)
12     {
13         buffer[n] = value;
14
15         value += incr;
16     }
17
18     return buffer;
19 }
```

Table 2.3: C++ ramp wave generator. This code differs from the program shown in Table 2.2 solely in the amplitude offset (-1 instead of 1) and the increment, which is now positive.

Triangle waves

A triangle wave can be seen as a combination of a ramp wave and a sawtooth wave, or as a linear, "edgy", sine wave. It increments from its minimum amplitude to its maximum amplitude linearly one half of a period and decrements back to the minimum during the other half. Simply put, "[a] triangle wave is a linear increment or decrement that switches direction every π radians" (Mitchell, 2008, p. 69). A mathematical definition for one period of a triangle wave is given by Equation 2.9, where ϕ is the phase in radians. If ϕ is kept in the range of $[-\pi; \pi]$ rather than the usual range of $[0; 2\pi]$, the subtraction of π can be eliminated, yielding Equation 2.10. If the dependent variable is time, in seconds, or samples, Equation 2.11 can be used for a range of $[0; T]$, where T is the period, and Equation 2.12 for a range of $[-\frac{T}{2}; \frac{T}{2}]$. A C++ implementation is shown in Table 2.4.

$$f(\phi) = \left\{ 1 - \frac{2|\phi - \pi|}{\pi}, \quad \text{if } 0 \leq \phi < 2\pi \right. \tag{2.9}$$

$$f(\phi) = \left\{ 1 - \frac{2|\phi|}{\pi}, \quad \text{if } -\pi \leq \phi < \pi \right. \tag{2.10}$$

$$f(t) = \left\{ 1 - \frac{4|t - \frac{T}{2}|}{T}, \quad \text{if } 0 \leq t < 1 \right. \tag{2.11}$$

$$f(t) = \left\{ 1 - \frac{4|t|}{T}, \quad \text{if } -\frac{T}{2} \leq t < \frac{T}{2} \right. \tag{2.12}$$

2.2.2 Additive Synthesis

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/or cosine waves with varying parameters and 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 called a

```
double* triangle(const unsigned int period) const
{
    double* buffer = new double[period];

    double value = -1;

    // 4.0 because we're incrementing/decrementing
    // half the period and the range is 2, so it's
    // actually 2 / period / 2.
    double incr = 4.0 / period;

    // Boolean to indicate direction
    bool reachedMid = false;

    for (unsigned int n = 0; n < period; n++)
    {
        wt[n] = value;

        // Increment or decrement depending
        // on the current direction
        value += (reachedMid) ? -incr : incr;

        // Change direction every time
        // the value hits a maximum
        if (value >= 1 || value <= -1)
        { reachedMid = !reachedMid; }
    }

    return buffer;
}
```

Table 2.4: C++ program to compute one period of a triangle wave.

Fourier Transform or a Fourier Series, both part of the Fourier Theorem. In a Fourier Series, a single sine/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. Equation 2.14 shows a simplified version of Equation 2.13. Table 2.5 presents a C++ struct to represent a single partial and Listing A.2 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.

```
struct Partial
{
  Partial(unsigned short number, double ampl, double phsOffs = 0)
    : num(number), amp(ampl), phaseOffs(phsOffs)
  { }

  /*! The Partial's number, stays const. */
  const unsigned short num;

  /*! The amplitude value. */
  double amp;

  /*! A phase offset */
  double phaseOffs;
};
```

Table 2.5: C++ code to represent a single partial in a Fourier Series.

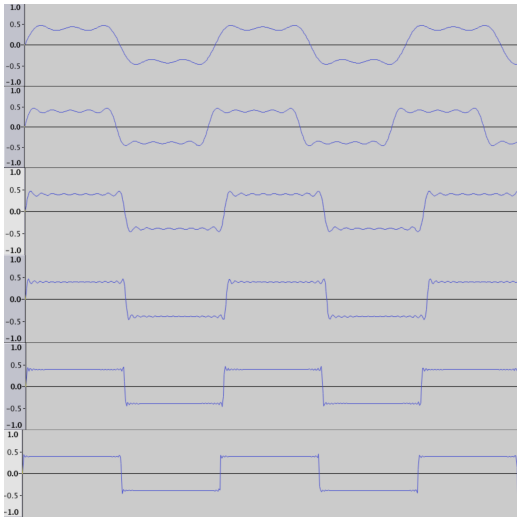


Figure 2.1: Square waves with 2, 4, 8, 16, 32 and 64 partials.

Table 2.6: C++ code for a square wave with 64 partials.

```
std::vector<Partial> vec;

for (int i = 1; i <= 128; i += 2)
{
  vec.push_back(Partial(i, 1.0/i));
}

double* buffer = additive(vec.begin(),
                          vec.end(),
                          48000)
```

The following paragraphs will examine how the four waveforms presented in Section 2.2.1, the square, the sawtooth, the ramp and the triangle wave, can be synthesized additively.

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 2.13, 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 2.6 displays the C++ code needed to produce one period of a square wave in conjuction with the `additive` function from Tables **??** and **??**. Figure 2.1 shows the result of summing 2, 4, 8, 16, 32 and finally 64 partials.

$$f(t) = \sum_{n=1}^{N} \frac{1}{2n-1} \sin(\omega(2n-1)t) \tag{2.13}$$

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 2.14 gives a mathematical definition for a sawtooth wave, Figure 2.2 displays sawtooth functions with various partial numbers and Table 2.7 shows C++ code to generate such functions.

$$f(t) = \sum_{n=1}^{N} \frac{1}{n} \sin(\omega n t) \tag{2.14}$$

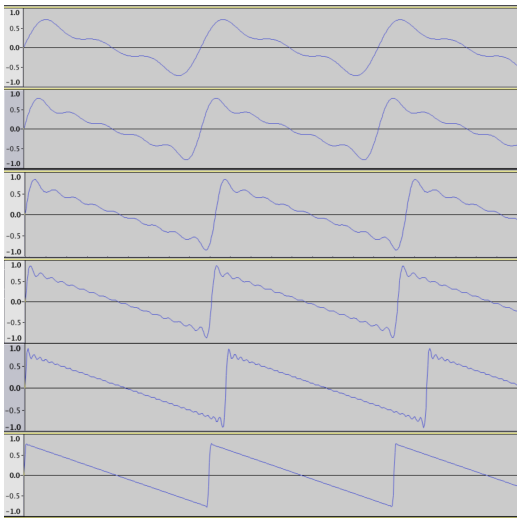


Figure 2.2: Sawtooth waves with 2, 4, 8, 16, 32 and 64 partials.

Table 2.7: C++ code for a sawtooth wave with 64 partials.

```
1  std::vector<Partial> vec;
2
3  for (int i = 1; i < 64; ++i)
4  {
5    vec.push_back(Partial(i, 1.0/i));
6  }
7
8  double* buffer = additive(vec.begin(),
9                            vec.end(),
10                           48000)
```

Ramp Waves

Ramp waves are essentially the inverse of sawtooth waves. Therefore, we can simply sum sinusoids as we did for a sawtooth function but change the sign of each partial's amplitude to negative instead of positive. Equation 2.15 gives the mathematical definition, Figure 2.3 displays a set of ramp waveforms with different partial numbers and Table 2.8 again shows the accompanying C++ code.

$$f(t) = \sum_{n=1}^{N} -\frac{1}{n} \sin(\omega n t) \tag{2.15}$$

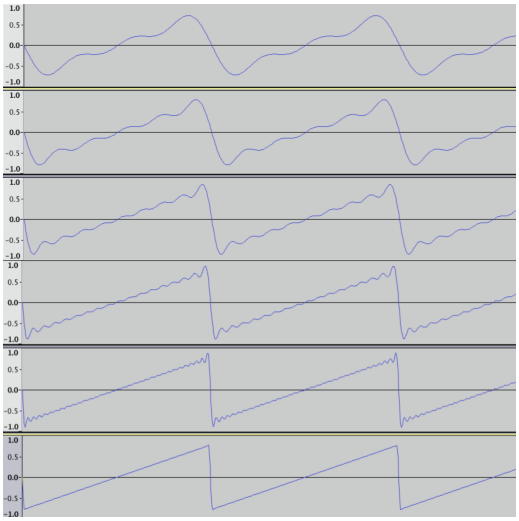


Figure 2.3: Ramp waves with 2, 4, 8, 16, 32 and 64 partials.

Table 2.8: C++ code for a ramp wave with 64 partials.

```
1  std::vector<Partial> vec;
2
3  for (int i = 1; i < 64; ++i)
4  {
5    vec.push_back(Partial(i, -1.0/i));
6  }
7
8  double* buffer = additive(vec.begin(),
9                            vec.end(),
10                           48000)
```

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 the inverse 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 2.16 or, more concisely, in Equation 2.17. Figure 2.4 displays such a triangle wave with various partial numbers (2,4,8,16,32 and 64) and Table 2.9 implements C++ code to compute a triangle wave.

$$f(t) = \sum_{n=1}^{\frac{N}{2}} \frac{1}{(4n-3)^2} \sin(\omega(4n-3)t) - \frac{1}{(4n-1)^2} \sin(\omega(4n-1)t) \tag{2.16}$$

$$f(t) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)^2} \sin(\omega(2n+1)t) \tag{2.17}$$

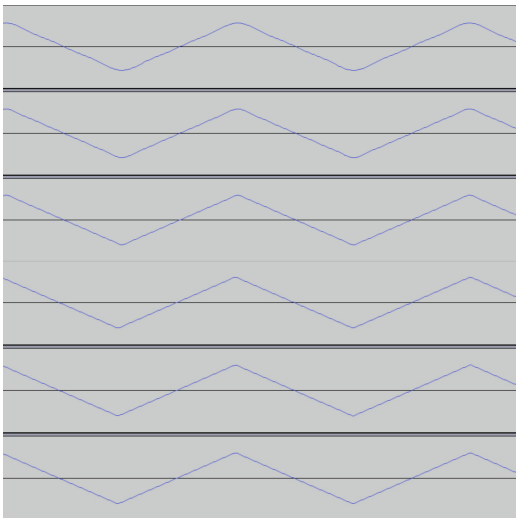


Figure 2.4: Triangle waves with 2, 4, 8, 16, 32 and 64 partials. Note that already 2 partials produce a very good approximation of a triangle wave.

Table 2.9: C++ code for a triangle wave with 64 partials.

```
1  std::vector<Partial> vec;
2
3  double amp = -1;
4
5  for(int i = 1; i <= 128; i += 2)
6  {
7    amp = (amp > 0) ? (-1.0/(i*i)) : (1.0/(i*i));
8
9    vec.push_back(Partial(i,amp));
10 }
11
12 double* buffer = additive(vec.begin(),
13                            vec.end(),
14                            48000);
```

2.2.3 Smooth Waveforms

One noticeable problem with square, sawtooth and ramp waves is that they have very sharp transitions at certain points in their waveforms, such as the square wave at the midpoint, $\frac{T}{2}$, where the amplitude jumps from its maximum to its minimum. While these transitions and jumps in amplitude contribute to the characteristic sound of these waves when used for music, they can pose a problem when they are used for modulation, as sharp transitions in amplitude translate to clicking sounds acoustically. One possible solution to reduce the impact of such transitions would be to overlay an audio envelope directly on the waveform, to bend or smooth the amplitude at the transition points. However, a more effective and direct solution would be to mathematically calculate a waveform that is already smoothed out at these critical points. The following two paragraphs propose a set of empirically determined functions to calculate "smooth" square and sawtooth waves.

Smooth Square Waves

Smooth square waves can be calculated mathematically as a set of four power functions with a very large exponent that change their sign after the midpoint. The exponent chosen, 50, was determined entirely experimentally. Equations 2.18 to 2.21 display the four power functions and their respective range of definition. Figure 2.5 shows the resulting visualization, where each individual power function is highlighted in a different colour as a visual aid.

$$a(t) = \begin{cases} (t-1)^{50} - 1, & \text{if } 0 \leq t < \frac{T}{4} \end{cases} \tag{2.18}$$

$$b(t) = \begin{cases} (t+0.5)^{50} - 1, & \text{if } \frac{T}{4} \leq t < \frac{T}{2} \end{cases} \tag{2.19}$$

$$c(t) = \begin{cases} -(t-1.5)^{50} + 1, & \text{if } \frac{T}{2} \leq t < \frac{3T}{4} \end{cases} \tag{2.20}$$

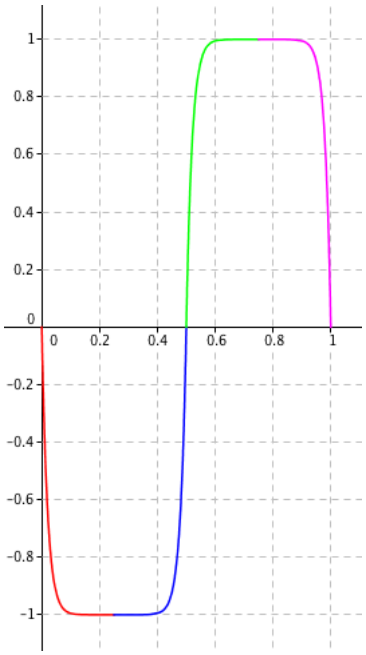


Figure 2.5: One period of a smooth square wave as the sum of four piecewise power functions.

$$d(t) = \begin{cases} -t^{50} + 1, & \text{if } \frac{3T}{4} \leq t < T \end{cases} \tag{2.21}$$

Smooth Sawtooth Waves

For sawtooth waves, the only critical area of transition is the point at which one period ends and the next begins, as the amplitude jumps back to the maximum from its minimum value. To make this transition smoother, one can use two very steep quadratic functions, one positive and one negative, that intersect at the midpoint of the transition. If one allocates nine tenths of one period for the normal descent of the sawtooth function from its maximum to its minimum, the transition can take place in the last tenth of the period. Equation 2.22 shows an altered version of the sawtooth wave function introduced in Section 2.2.1 and Equations 2.23 and 2.24 the two aforementioned quadratic functions. Figure 2.6 visualizes the resulting waveform.

$$a(t) = \begin{cases} \frac{-2t}{0.9T} + 1 = \frac{-20t}{9T} + 1, & \text{if } 0 \leq t < \frac{9T}{10} \end{cases} \tag{2.22}$$

$$b(t) = \begin{cases} 400(t - 0.9)^2 - 1, & \text{if } \frac{9T}{10} \leq t < \frac{95T}{100} \end{cases} \tag{2.23}$$

$$c(t) = \begin{cases} -400(t - 1)^2 + 1, & \text{if } \frac{95T}{100} \leq t < T \end{cases} \tag{2.24}$$

Smooth Ramp Waves

Smooth ramp waves are created identically to smooth sawtooth waves, except that the sign of all piecewise functions as well as as their offsets change, as shown by Equations 2.25 to 2.27 as well as Figure 2.7.

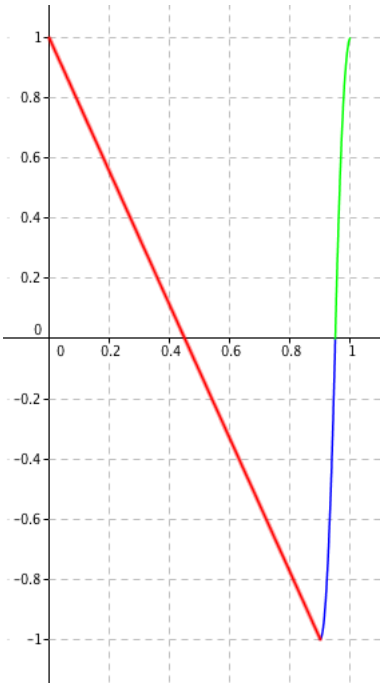


Figure 2.6: A smooth sawtooth wave.

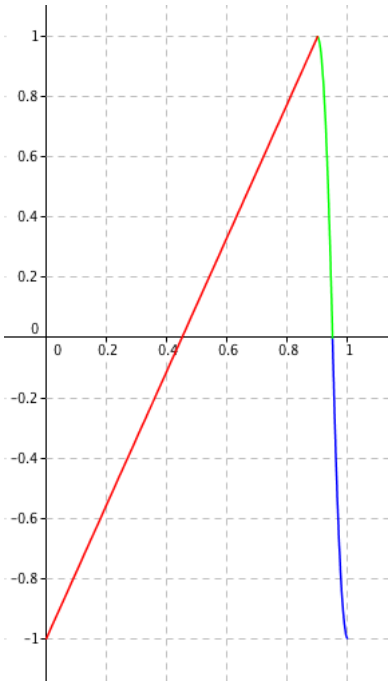


Figure 2.7: A smooth ramp wave.

$$a(t) = \begin{cases} \dfrac{2t}{0.9T} - 1 = \dfrac{-20t}{9T} + 1, & \text{if } 0 \leq t < \dfrac{9T}{10} \end{cases} \tag{2.25}$$

$$b(t) = \begin{cases} -400(t - 0.9)^2 + 1, & \text{if } \dfrac{9T}{10} \leq t < \dfrac{95T}{100} \end{cases} \tag{2.26}$$

$$c(t) = \begin{cases} 400(t - 1)^2 - 1, & \text{if } \dfrac{95T}{100} \leq t < T \end{cases} \tag{2.27}$$

2.2.4 Sine Waves with Different Bit Widths

The bit width of a waveform determines its resolution, i.e. the number of numerical values the amplitude of a signal can take. Common values for the resolution of a waveform are 64 bits, as provided by the C++ double-precision floating-point data type `double`, or 16 bits, as used by the Waveform Audio File Format (WAVE). Reducing the bit width of a waveform can give the impression of an "old-school" sound, as older synthesis systems from the 20th century were often limited to low bit-widths due to their less advanced hardware, compared to contemporary hardware configurations. Consequently, some digital synthesizers, such as Ableton's Operator, additionally provide waveforms with other bit widths such as 4 or 8 bits. Table 2.10 shows bit width values alongside the number of possibilities a sample can take when the given bit width is used. Note that in general, n bits yield 2^n possible values. Figures 2.8 and 2.9 show sine waves with a bit width of 3 and 4 bits, respectively.

Bit width	Possibilities
2	4
3	8
4	16
8	256
16	65536
32	4294967296
64	18446744073709551616

Table 2.10: Bit widths alongside the number of possible values a sample can take when the given bit width is used.

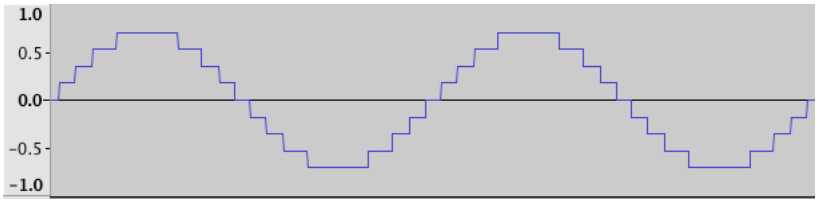


Figure 2.8: A 3-bit sine wave.

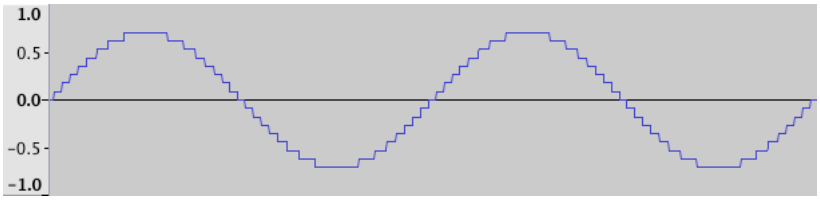


Figure 2.9: A 4-bit sine wave.

2.3 The Gibbs Phenomenon and the Lanczos Sigma Factor

Examining Figure 2.10, which displays an additively synthesized square wave function with 64 partials, one may observe that additive synthesis, the summation of sine waves to produce complex waveforms, produces an overshoot — slight ripples or "horns" — at the ends of each peak and trough of a waveform. This is known as the Gibbs Phenomenon, named after the American scientist Josiah Willard Gibbs who first described it in 1898, and is "the result of summing a finite series of partials rather than the infinite series of partials specified by the Fourier transform" (Mitchell, 2008, p. 67). Acoustically, this ringing artifact has little influence on the sound of the produced waveform. However, for modulation, this imperfection may render the resulting sound unsatisfactory. A common way to reduce the Gibbs Phenomenon is to apply the Lanczos Sigma (σ) Factor to each partial of a Fourier Series. This is often called *sigma-approximation*. The definition of the Lanczos Sigma Factor is given in Equation 2.28, where n is the current partial number and M the total number of partials in a Fourier Summation. In the `additive` function shown in Tables **??** and **??**, the Lanczos Sigma Factor is implemented in lines 48 to 54. Figure 2.11 shows the same square wave from Figure 2.10 after sigma-approximation.

$$\sigma = sinc(\frac{n\pi}{M}) = \frac{\sin(\frac{n\pi}{M})}{\frac{n\pi}{M}} \tag{2.28}$$

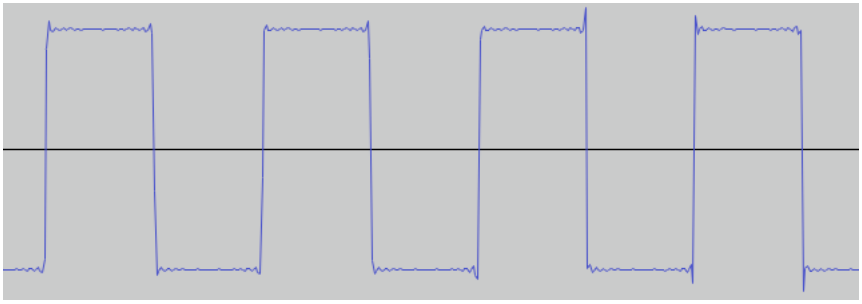


Figure 2.10: An additively synthesized square wave with 64 partials before sigma-approximation.

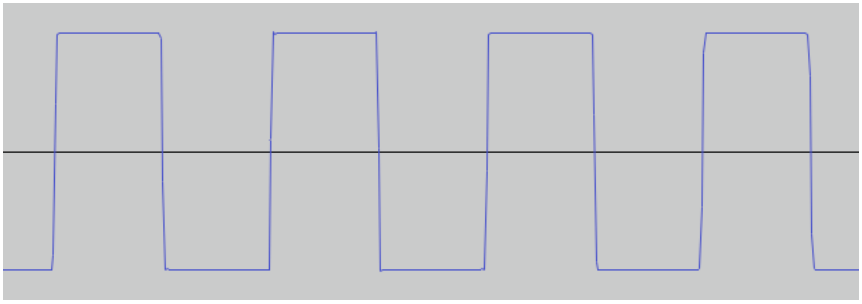


Figure 2.11: An additively synthesized square wave with 64 partials after sigma-approximation.

2.4 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.

2.4.1 Implementation

A Wavetable is a table (practically speaking an array) in which one stores pre-calculated waveform amplitude values for lookup. This way, the computation of individual samples as shown in Sections 2.2.1 and 2.2.2 can be replaced by the incrementing of a table index variable and retrieval of samples by dereferencing of the array at the current index. If a table holds sample values for one period of a waveform, at a frequency of $1Hz$, the frequency of the waveform can be adjusted during playback by multiplying the increment value by some factor other than one. "For example, if we increment by two [instead of one], we 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 for a table index, i_{fund}, the value by which a table index must be incremented after each sample to traverse a waveform at a frequency of $1Hz$, given the table length L and the samplerate f_s, is shown in Equation 2.29. To alter the frequency f of the played-back waveform, Equation 2.30 should be used to calculate the appropriate increment value i.

```
sample = table[integral] + ((table[integral + 1] - table[integral]) * fractional)
```

Table 2.11: An interplation algorithm in pseudo-code.

```
template<class T>
double interpolate(T table [], double index)
{
    long integral = static_cast<long>(index);  // The truncated integral part
    double fractional = index - integral;    // The remaining fractional part

    // grab the two items in-between which the actual value lies
    T value1 = table[integral];
    T value2 = table[integral+1];

    // interpolate: integer part + (fractional part * difference between value2 and value1)
    double final = value1 + ((value2 - value1) * fractional);

    return final;
}
```

Table 2.12: Full C++ template function to interpolate a value from a table, given a fractional index.

$$i_{fund} = \frac{L}{f_s} \tag{2.29}$$

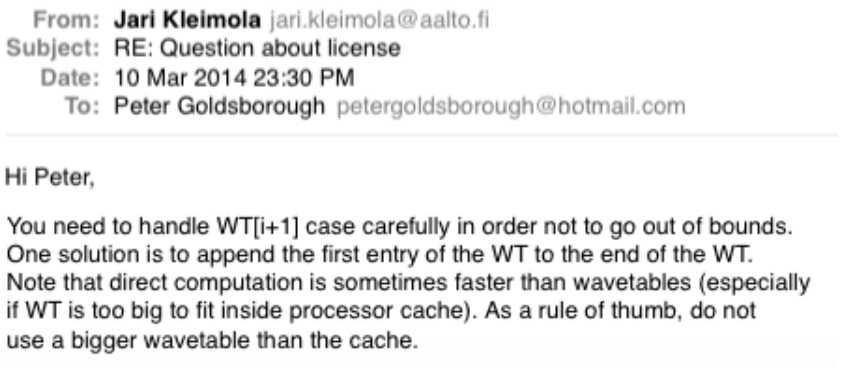
$$i = i_{fund} \times f = \frac{L}{f_s} \times f \tag{2.30}$$

2.4.2 Interpolation

For almost all configurations of frequencies, table lengths and sample rates, the table index i produced by Equation 2.30 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 [...]". This is obviously a suboptimal solution which would result in a change of phase and consequently distortion. (Mitchell, 2008, p. 84) 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 2.11 displays the calculation of a sample by means of interpolation in pseudo-code and Table 2.12 in C++. (based on pseudo-code, Mitchell, 2008, p. 85)

2.4.3 Table Length

The length of the Wavetable must be chosen carefully to be both memory efficient and at the same time provide a decently accurate waveform resolution. An equation to calculate the size of a single wavetable in Kilobytes is given by Equation 2.31, 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`.



From: **Jari Kleimola** jari.kleimola@aalto.fi
Subject: RE: Question about license
Date: 10 Mar 2014 23:30 PM
To: Peter Goldsborough petergoldsborough@hotmail.com

Hi Peter,

You need to handle WT[i+1] case carefully in order not to go out of bounds. One solution is to append the first entry of the WT to the end of the WT. Note that direct computation is sometimes faster than wavetables (especially if WT is too big to fit inside processor cache). As a rule of thumb, do not use a bigger wavetable than the cache.

Figure 2.12: An excerpt of an E-Mail exchange with Jari Kleimola.

$$Size = \frac{L \times N}{1024} \tag{2.31}$$

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 titled "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 2.12. 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 as seen in Figure 2.12, is that because the interpolation algorithm from Table 2.12 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. This added sample usually 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.

2.4.4 File Format

For maximum efficiency, the Wavetables are not created at program startup but read from a binary file. To prevent the synthesizer program from accidentally reading faulty files, some simple identification string must be added to the file. Therefore, Wavetable files contain a 6-char ID string with the name of the synthesizer, Anthem, after which the 32 KB of Wavetable data follow. Additionally, Wavetable files end with a `.wavetable` file extension. Table 2.13 displays a function to read such a Wavetable file and Figure 2.13 shows the first few bytes of a Wavetable file when opened as plain-text. The total size of a Wavetable file is exactly 32774 bytes, 32768 bytes (32KB) of Wavetable data and 6 bytes for the ID string.

```
ANTHEM{†Åå#Y?ö`)˜ä#i?'œ Ã•/r?·zO5É#y?÷?Ëº\l?À°≠µò/Ç?ìµ7$˘Ö?±@ˆ-d#å?õÆÕVƒGå?˜£" lè?‰¯≠ä;Hë?
BtoYd/í?®ïâ?älî?áq◀˛˜¸ï?ÀòòYÃêó?g"wË"ô?Î¥ÅË˘¥ö?êÆ†Gú?cñÙ°"Ÿù?®VKº-kü?Œä'—ô~†?µk:öG°?¶4µ˜ó
¢?ã5ñÎíŸ¢?b,Ç˜ä¢£?
```

Figure 2.13: The first few bytes of a Wavetable file.

```cpp
#include <fstream>
#include <stdexcept>

double* readWavetable(const std::string &fname)
{
  std::ifstream file(fname);

  if (! file.is_open())
  { throw std::runtime_error("Could not find wavetable file: " + fname); }

  if (! file.good())
  { throw std::runtime_error("Error opening wavetable: " + fname); }

  char signature[6];

  file.read(signature, 6);

  if (strncmp(signature, "ANTHEM", 6))
  { throw std::runtime_error("Invalid signature for Anthem file!"); }

  int len = 4096;
  int size = len * sizeof(double);

  double * wt = new double [len];

  file.read(reinterpret_cast<char*>(wt), size);

  return wt;
}
```

Table 2.13: C++ code to read a Wavetable file.

2.5 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.

2.5.1 Colors of Noise

The *color* of a noise signal describes, acoustically speaking, the *texture* or *timbre* [3] of the sound produced, as well as, scientifically speaking, the spectral power density and frequency content of the signal. The following paragraphs will examine and explain the creation of various noise colors, namely white, pink, red, violet and blue noise.

[2]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.

[3]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".

White Noise

White noise 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 2.14 shows a typical white noise signal in the time domain, Figure 2.15 gives a close-up view of Figure 2.14, Figure 2.14 displays the frequency spectrum of a white noise signal and Table 2.16 shows the implementation of a simple C++ class to produce white noise.

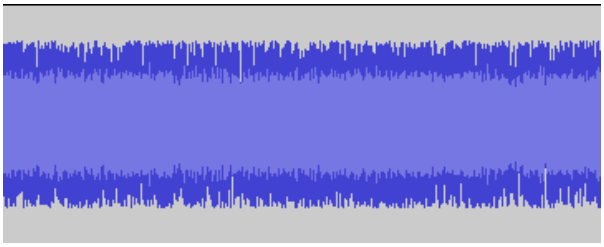


Table 2.14: A typical white noise signal.

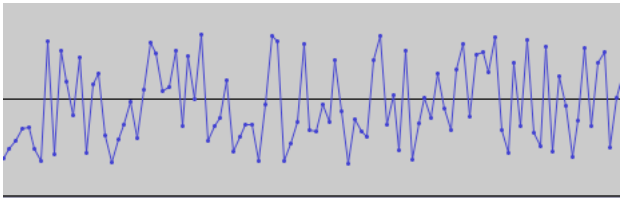


Table 2.15: A close-up view of Figure 2.14. This Figure shows nicely how individual sample values are completely random and independent from each other.

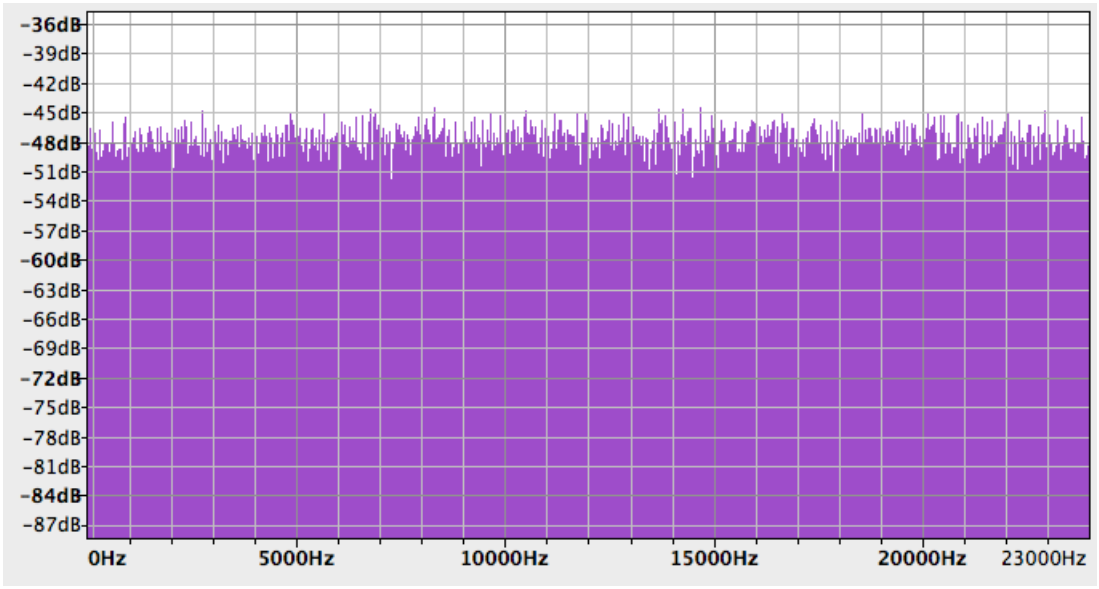


Figure 2.14: The signal from Figures 2.14 and 2.15 in the frequency domain. This frequency spectrum analysis proves the fact that white noise has a "flat" frequency spectrum, meaning that all frequencies are distributed uniformly and at (approximately) equal intensity.

```cpp
#include <random>
#include <ctime>

class Noise
{
    Noise()
    : dist_(-1,1)
    {
        // Seed random number generator
        rgen_.seed((unsigned)time(0));
    }

    double tick()
    {
        // Return noise sample
        return dist_(rgen_);
    }

private:

    /*! Mersenne-twister random number generator */
    std::mt19937 rgen_;

    /*! Random number distribution (uniform) */
    std::uniform_real_distribution<double> dist_;
};
```

Table 2.16: 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.

Pink Noise

Pink noise, sometimes referred to as $\frac{1}{f}$ noise, is white noise filtered with a 3dB/octave low-pass filter. In contrast to white noise, which has equal energy *per frequency component*, pink noise has equal energy *per octave*. This is a significant difference as humans hear on the basis of octaves (logarithmically), and not completely linearly, meaning that for the human ear the difference between 400 and 800 Hertz (one octave) sounds the same as the octave jump between 6 KHz and 11 Khz, even though the absolute difference between thoes two ranges is 4.6 Khz! To address this phenomenon, pink noise has more energy in the lower, narrower octave ranges and less energy in the wider, higher octave ranges. Therefore, the few high intensity values in the low frequency bands add up to equal the sum of lower intensity values in the wider, higher frequency octave bands. Figure 2.15 displays the frequency spectrum of pink noise. In the time domain, pink noise looks more or less like white noise, as can be seen in Figure 2.16.

Red Noise

Red Noise has an energy decrease of 6dB/Octave. Another method of generating red noise besides filtering white noise appropriately is integrating a white noise signal, as shown in Equation 2.32. Red noise is also called "Brown" or "Brownian" noise. This name is not given for its visual equivalent, but rather for it's resemblance to a signal produced by Brownian motion or "Random Walk", "the erratic random movement of microscopic particles in a fluid, as a result of continuous bombardment from molecules of the surrounding medium." (Oxford Dictionary of English, 2003). Figure 2.17 shows a red noise signal in the frequency domain and Figure 2.18 in the time domain.

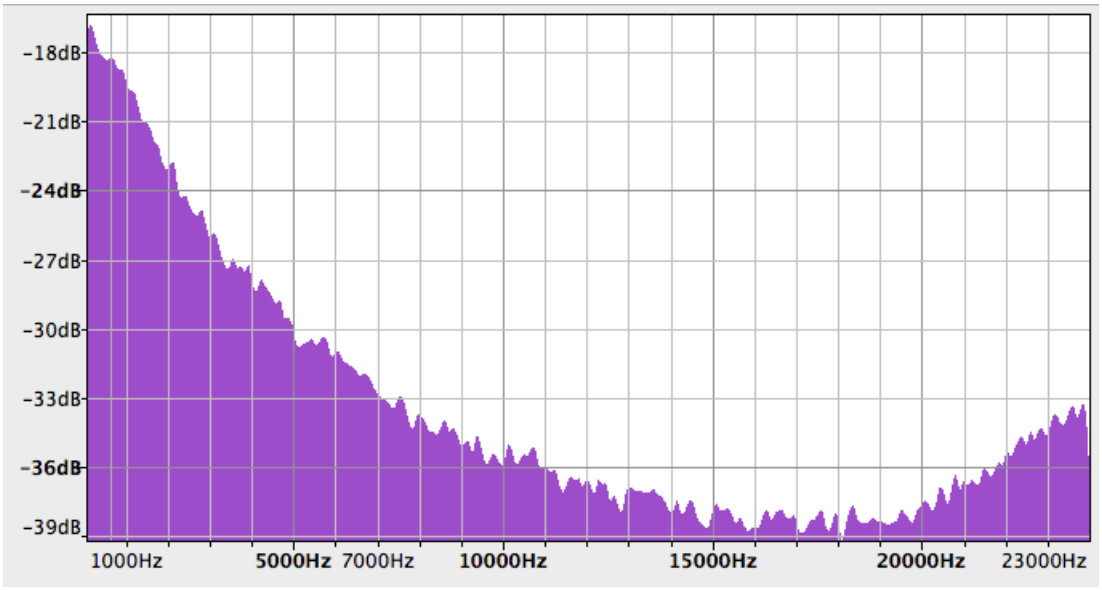


Figure 2.15: The frequency spectrum of pink noise. Note that the filter with which white noise was transformed into pink noise, discussed in a later chapter, is not perfect, resulting in the increase in frequency intensity towards the end of the spectrum. Ideally, the intensity should decrease linearly.

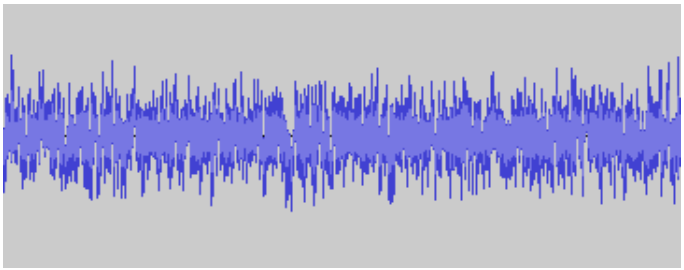


Figure 2.16: Pink noise in the time domain.

$$N_{red}(t) = \int_0^t N_{white}(t)dt \tag{2.32}$$

Blue Noise

Wherease pink and red noise show a decrease in intensity per frequency component, blue noise exhibits an **increase** in intensity by 3dB/Octave, making it sound like a high-pitched version of white noise. Blue noise can be created by making the filter used for pink noise a high-pass filter instead of a low-pass one, meaning that higher frequencies are filtered less than lower ones. Figure 2.19 shows a frequency spectrum of the blue noise created for the purpose of this thesis. The reader may question as to why the spectrum shown looks nothing like what was just described as the ideal spectrum of blue noise. As matter of fact, it was discovered that a frequency spectrum of this kind, combined with a decreased gain of -3dB, produces the same sound as an ideal blue noise signal. In the time domain, blue noise looks similar to the white noise depicted in Figure 2.14.

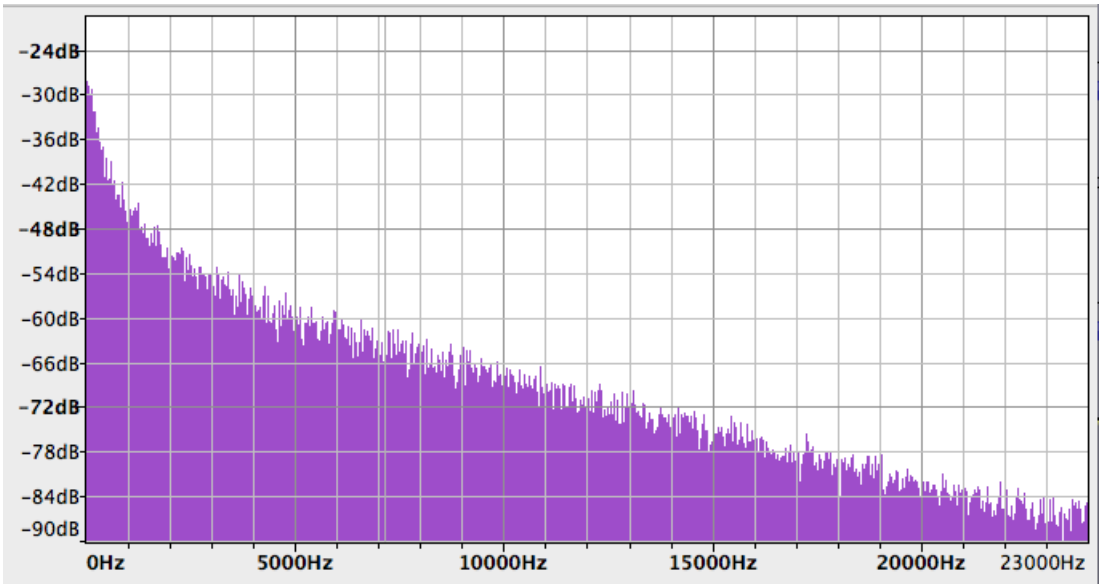


Figure 2.17: Red noise in the frequency domain.

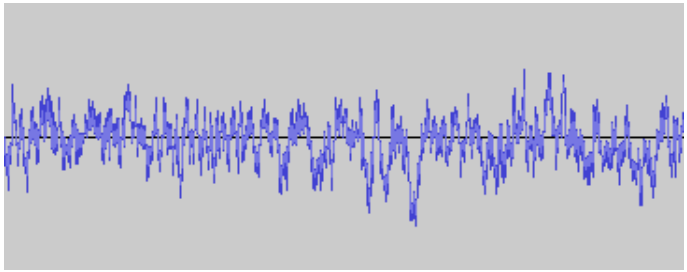


Figure 2.18: Red noise in the time domain.

Violet Noise

What blue noise is to pink noise, violet noise is to red noise. Violet noise increases by 6dB per octave and can also be created by differentiating a white noise signal, just as red noise is

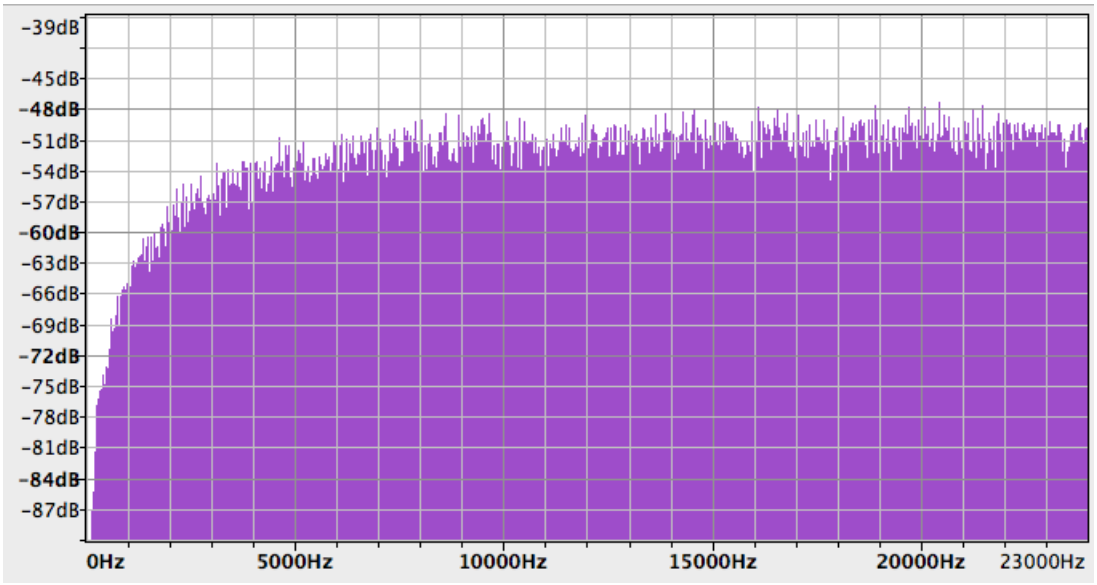


Figure 2.19: Blue noise in the frequency domain.

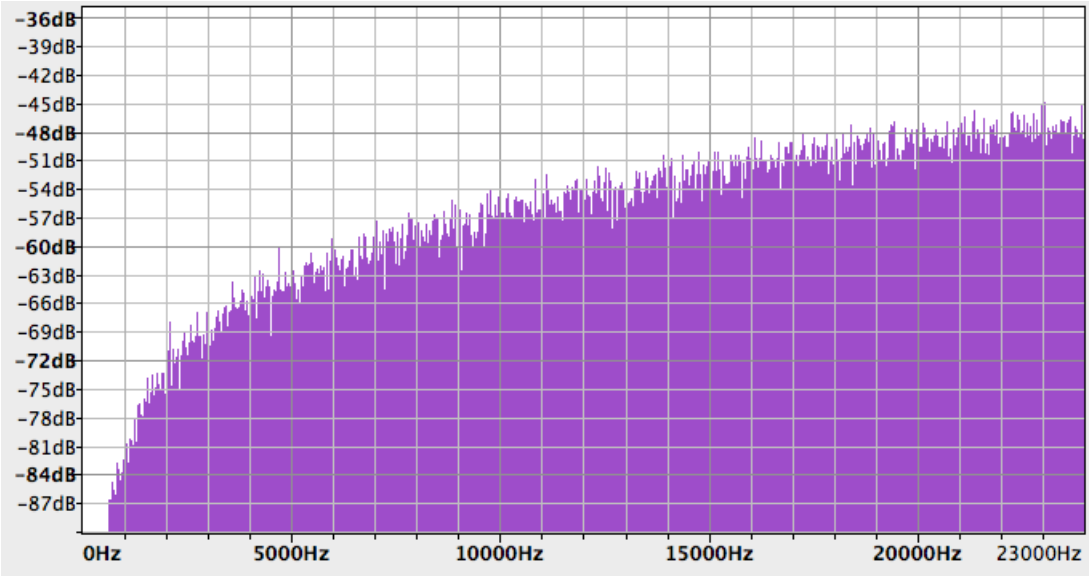


Figure 2.20: The frequency spectrum of a violet noise signal.

the result of integrating white noise. Acoustically, violet noise sounds like a very high-pitched "hissing" sound. Equation 2.33 shows how violet noise can be produced as a derivative of white noise and Figure 2.20 shows the frequency response of a violet noise signal.

$$N_{violet} = \frac{dN_{white}(t)}{dt} \tag{2.33}$$

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 his 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.

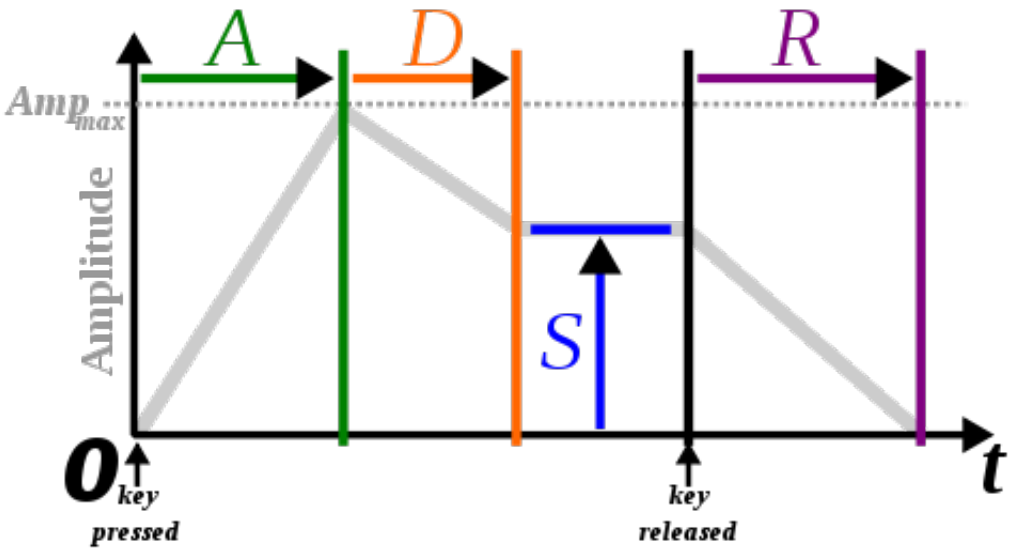


Figure 3.1: An Envelope with an Attack, a Decay, a Sustain and finally a Release segment. Source: `http://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/ADSR_parameter.svg/500px-ADSR_parameter.svg.png`

Figure 3.2: A 440 Hz sine wave.

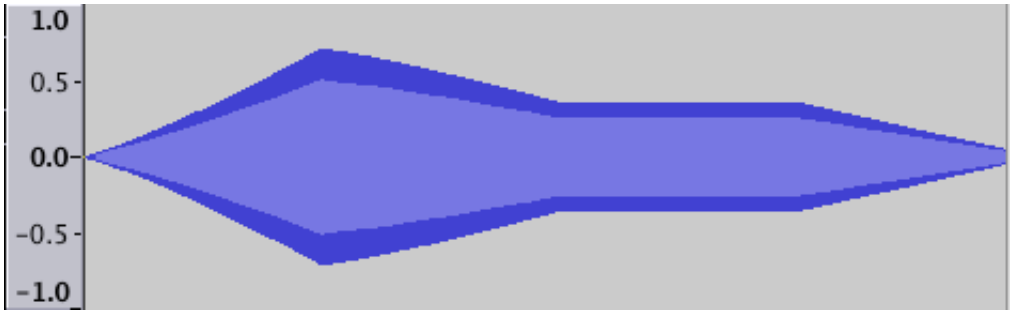


Figure 3.3: The same signal from Figure 3.2 with an ADSR Envelope overlayed on it.

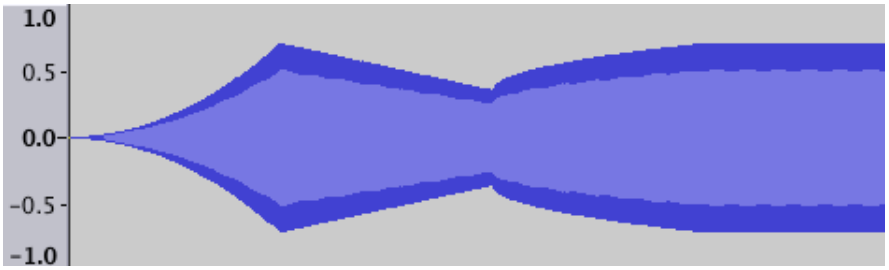


Figure 3.4: An envelope where r is equal to 2, then to 1, then to 0.5.

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.3.

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

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 update 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 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.

```
1  void EnvSegSeq::update()
2  {
3      currSample_++;
4      currSeg_->update();
5  }
6
7  double EnvSegSeq::tick()
8  {
9      if (currSample_ >= currSeg_->getLen())
10     {
11         // Increment segment
12         currSeg_++;
13
14         // Check if we need to reset the loop
15         if (currSeg_ == loopEnd_ && (loopInf_ || loopCount_ < loopMax_))
16         { resetLoop_(); }
17
18         // If we've reached the end, go back to last segment
19         // which will continue to tick its end amplitude value
20         else if (currSeg_ == segs_.end()) currSeg_--;
21
22         // else change
23         else changeSeg_(currSeg_);
24     }
25
26     return currSeg_->tick();
27 }
```

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

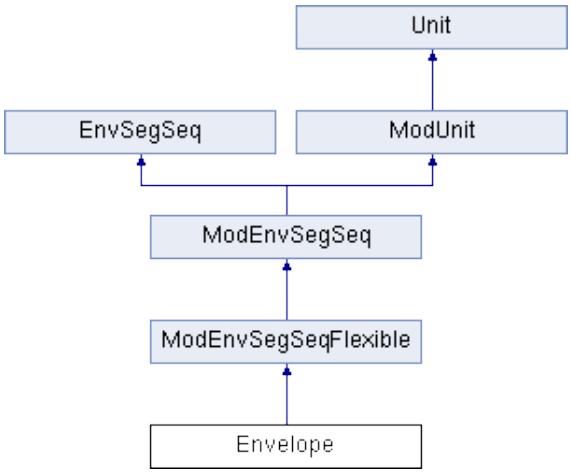


Figure 3.5: The inheritance diagram for the `Envelope` class. The `Unit` and `ModUnit` classes are two abstract classes that will be explained in later parts of this thesis.

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. Moreover, Amplitude Modulation is one of the two main techniques for transmitting information over electro-magentic waves, alongside Frequency Modulation (FM), but that is a different subject, entirely. 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.

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

Figure 3.6: A 440 Hz sine wave.

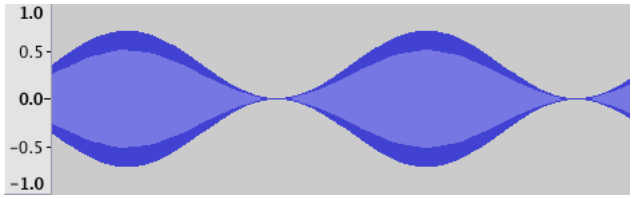


Figure 3.7: The sine wave from Figure 3.6 modulated by an LFO with a frequency of 2 Hertz. Note how the maximum amplitude of the underlying signal now follows the waveform of a sine wave. Because the LFO's amplitude is the same as that of the oscillator (the "carrier" wave), the loudness is twice as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

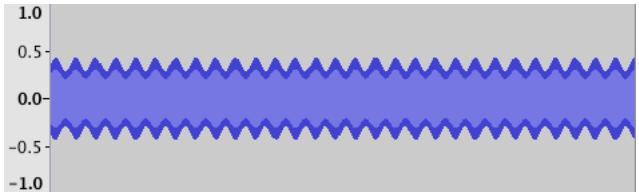


Figure 3.8: The sine wave from Figure 3.6 modulated by an LFO with a frequency of 20 Hertz. This signal is said to have a "vibrato" sound to it.

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

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:

```
1 | typedef OscillatorClass LFOClass.
```

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.

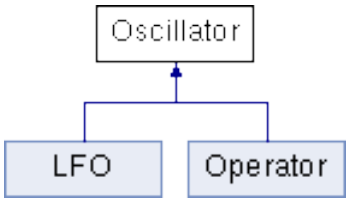


Figure 3.9: Inheritance diagram showing the relationship between an `LFO`, an `Operator` and their base class, the `Oscillator` class.

3.2.1 LFO Sequences

A common sight in many modern digital synthesizers, such as Native Instruments'[2] "Massive" synthesizer, is a form of step-sequencer for Low Frequency Oscillators. This modulation unit is essentially a synthesis between the concept of an Envelope and that of an LFO. It has a variable number of segments of fixed length, each with their own LFO to change the waveform of the segment. Figure 3.10 displays what Native Instruments calls a "Stepper".

For the purpose of this thesis, such an LFO sequence was created. Regarding the implementation in C++, there is little difference between an LFO Sequence and an Envelope Segment Sequence, except for the fact that now each individual segment has an LFO that can be accessed through the interface of the class to change the parameters or the waveform of each LFO. One of the two parameters that deserves to be discussed, however, is the frequency or "rate" of such an LFO sequence. Just like the frequency of an oscillator describes the number of cycles its waveform completes every second, the rate of an LFO Sequence determines how often the entire sequence is traversed in one second. Practically speaking, this involves adjusting the length of individual segments in such a way that the combined length equals the wanted duration. To give an example: if the user wishes to change the rate of an LFO Sequence with 10 segments to 1 Hz, each individual segment must have a duration of 0.1 seconds (100 milliseconds), so that at the end of the sequence, exactly 10 segments $\times$ 0.1 seconds $=$ 1 second will have passed.

[2]"Native Instruments is a technology company that develops software and hardware for music production and DJ-ing", based out of Berlin, Germany. Source: *Wikpedia:* `http://en.wikipedia.org/wiki/Native_Instruments`. Homepage: `http://www.native-instruments.com/en/`

Figure 3.10: An LFO sequence, here named a "Stepper". Source: `http://www.askaudiomag.com/articles/massives-performer-and-stepper-lfos`

The length of a single segment can be derived from Equation 3.3, where r is the rate of the LFO Sequence and N the number of segments in the sequence.

$$l_{seg} = \frac{1}{r \times N} \tag{3.3}$$

The second parameter worthy of mentioning is the frequency of individual segments' modulation LFOs. As mentioned, the frequency of an oscillator or of an LFO sets the cycles per second of their waveform. However, when the length of a single segment in an LFO sequence is only very small, e.g. 100 milliseconds, having an LFO modulate the segment at a common frequency of, say, 1 Hz, would mean that only one tenth of the LFO's period is completed. To have the LFO complete a full cycle, the user would have to adjust the frequency of the LFO in such a way that it can complete one period within the duration of the segment (in this case, 10 Hz). This may still seem like a manageable task for the user. However, once parameter values become a little more complicated, giving birth to cases where the user has to figure out that to have 5 cycles in a segment with a duration of 0.4 seconds, the frequency needs to be exactly 12.5 Hz, the objective reader must admit that this is not intuitive behaviour. Therefore, it was decided that for modulation LFOs in LFO sequences, the rate should not be defined as cycles per second, but as cycles per *segment*. This means that setting the rate of a segment's LFO to 5 Hertz results in 5 cycles per second *regardless* of the segment's duration. Naturally, the task of calculating the correct frequency for the LFO still has to be performed by the computer program. The *real* frequency can be calculated using Equation 3.4 when the duration d is in seconds and Equation 3.5 when the duration is in samples, as is the case for a digital synthesizer. In both Equations $f_{displayed}$ stands for the frequency the user specifies, the cycles per segment, and f_{real} for the actual frequency in cycles per second. Furthermore, in Equation 3.5, f_s denotes the sample rate.

$$f_{real} = \frac{f_{displayed}}{d} \tag{3.4}$$

$$f_{real} = \frac{f_s}{d} \times f_{displayed} \tag{3.5}$$

```
void LFOSeq::setRate(double Hz)
{
    if (Hz < 0 || Hz > 10)
    { throw std::invalid_argument("Rate cannot be less than zero or greater 10!"); }

    rate_ = Hz;

    resizeSegsFromRate_(rate_);
}

void LFOSeq::resizeSegsFromRate_(double rate)
{
    // get the period, divide up into _segNum pieces
    segLen_ = (Global::samplerate / rate) / segs_.size();

    // Set all segments' lengths
    for (int i = 0; i < segs_.size(); i++)
    {
        segs_[i].setLen(segLen_);

        // Scale frequency of mods according to length
        setScaledModFreq_(i);
    }
}

void LFOSeq::setScaledModFreq_(seg_t seg)
{
    // Set scaled frequency to frequency of lfo
    lfos_[seg].lfo.setFrequency(getScaledModFreqValue(lfos_[seg].freq));
}

double LFOSeq::getScaledModFreqValue(double freq) const
{
    // Since the rate is in cycles per segment
    // and not cycles per second, we get the
    // "period" of the segment and multiply that
    // by the rate, giving the mod wave's frequency.

    if (! segLen_) return 0;

    // To go from samples to Hertz, simply
    // divide the samplerate by the length
    // in samples e.g. 44100 / 22050 = 2 Hz
    double temp = Global::samplerate / static_cast<double>(segLen_);

    // Multiply by wanted frequency
    return freq * temp;
}
```

Table 3.2: Relevant member functions from the `LFOSeq` class for calculating the correct rate for individual segments as well as for the entire sequence.

3.3 The ModDock system

The majority of digital synthesizers, such as Propellerhead's[3] *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 [4]. 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

The following paragraphs will outline the process of implementing such a ModDock system.

Units and ModUnits

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. Units have necessary member variables and functions to permit that Unit's 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*[5] `modulate` method, which takes a sample and 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 3.11 displays the inheritance diagram for the Unit class.

[4]"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`

[4]I, Peter Goldsborough, want to explicitly state that I did not invent the idea of, what *I* call, the "ModDock" system. Full credit goes to the software engineers behind and anybody involved in the creation of Native Instruments *Massive*. I merely implemented it and modified it slightly.

[5]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.

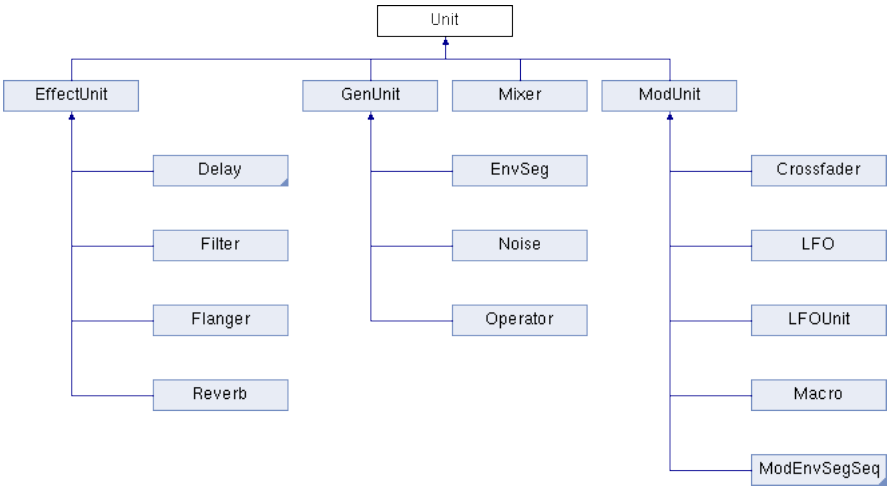


Figure 3.11: The inheritance diagram of the Unit class. GenUnits are Units that generate samples. EffectUnits apply some effect to a sample, e.g. a sample delay.

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 $[-A_{LFO} \times 1; A_{LFO} \times 1]$, whereas for the rate parameter of Envelope segments, where the maximum boundary is 10, the range is $[-A_{LFO} \times 10; A_{LFO} \times 10]$. The declaration of the `modulate` method in the ModUnit class is given below.

```
1 |   virtual double modulate(double sample, double depth, double maximum) = 0;
```

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 3.3 shows the definition of the `modulate` method in the LFO and Table 3.4 in the Envelope class.

```
1 | double LFO::modulate(double sample, double depth, double maximum)
2 | {
3 |   return sample + (maximum * Oscillator::tick() * depth * amp_);
4 | }
```

Table 3.3: The implementation of the `modulate` method for LFOs.

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

```
double Envelope::modulate(double sample, double depth, double)
{
   return sample * tick_() * depth * amp_;
}
```

Table 3.4: 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.

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. This means 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:

```
  double modulate(double sample);
```

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 3.5 shows how these parameters may be passed to a ModDock and updated when necessary.

```
AnyUnit::AnyUnit()
{
    modDock.setHigherBoundary(1);
    modDock.setLowerBoundary(0);
    modDock.setBaseValue(0.5);
}

void AnyUnit::setAmp(double amp)
{
    modDock.setBaseValue(amp);
}
```

Table 3.5

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. A non-Master is any ModUnit that 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 directly 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 3.12 depicts these relationships in a flow-chart. Figure 3.13 shows a scan of early sketches created while implementing side-chaining.

What Figure 3.13 also displays is that there are two possible ways to implement side-chaining. The first method, which was ultimately not chosen, is to have each ModUnit in the ModDock have 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 A.4 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.

Moreover, 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 A.5 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 finally returned.

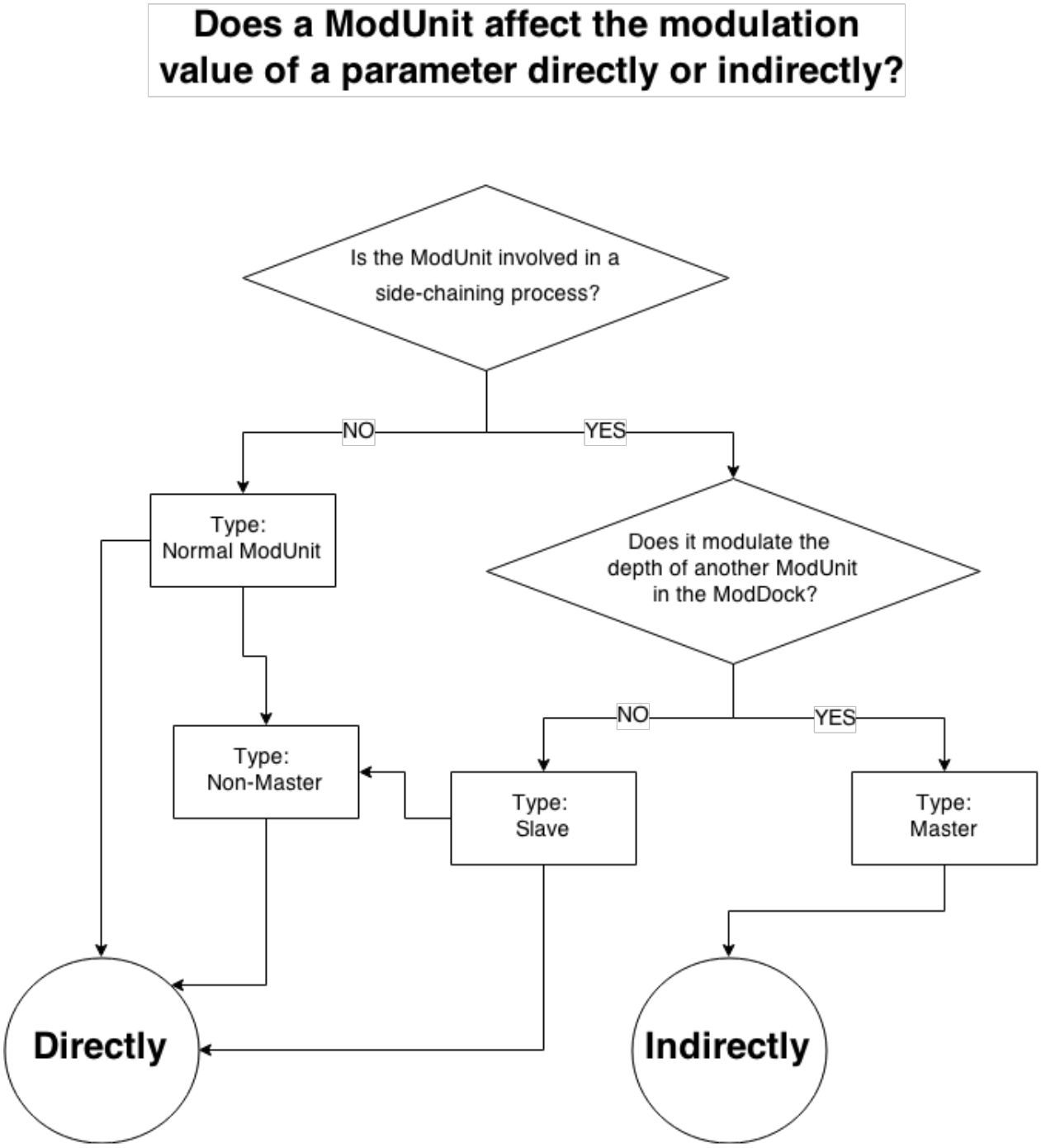

Does a ModUnit affect the modulation value of a parameter directly or indirectly?
Is the ModUnit involved in a side-chaining process?
NO
YES
Type: Normal ModUnit
Does it modulate the depth of another ModUnit in the ModDock?
NO
YES
Type: Non-Master
Type: Slave
Type: Master
Directly
Indirectly

Figure 3.12

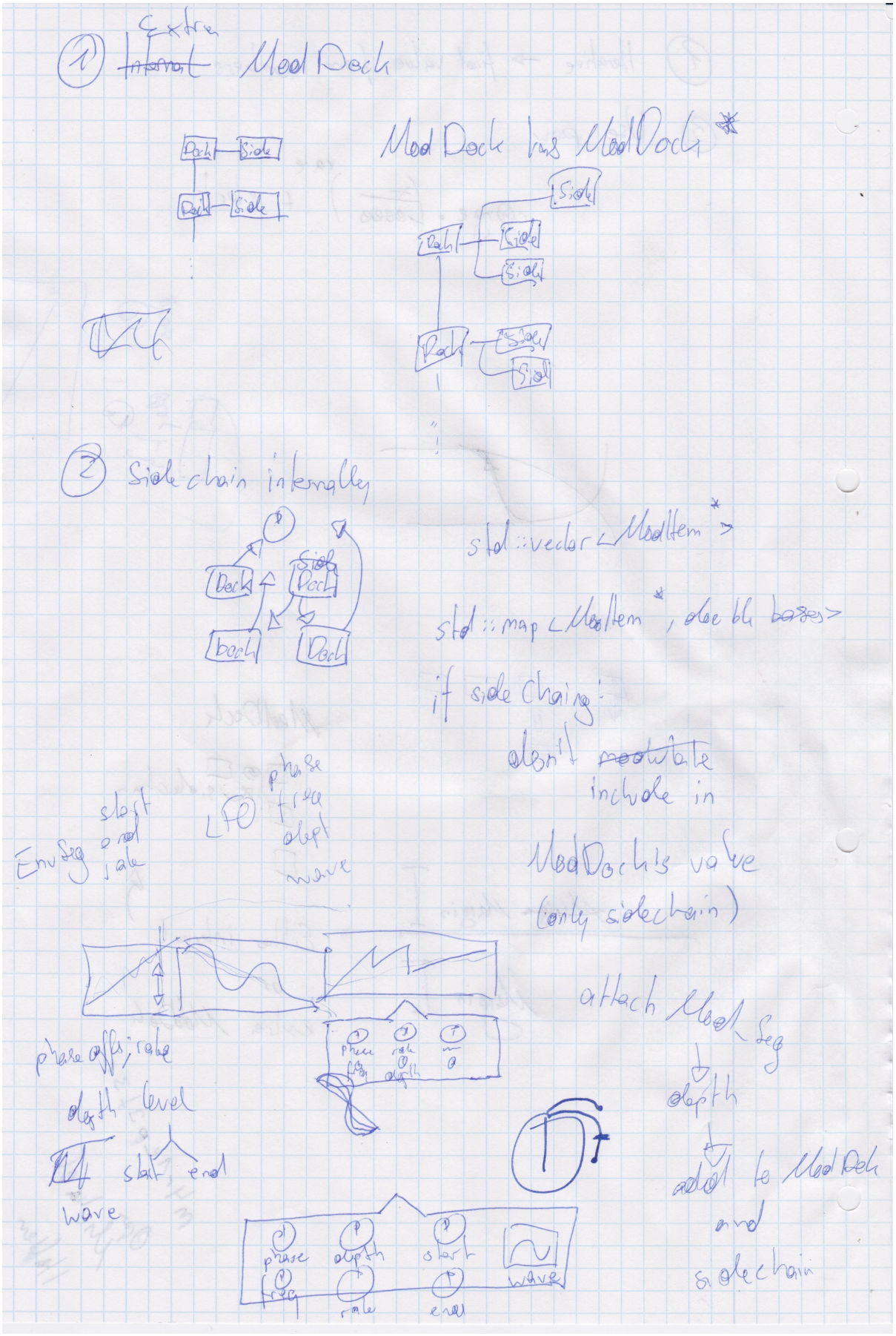

ModDock has ModDock*
Side chain internally
std::vector<ModItem*>
std::map<ModItem*, double> bases>
if side chain:
don't modulate
include in
ModDock's value
(only sidechain)
attach ModLfo
depth
add to ModDock
and
sidechain
EnvSeg
start
end
rate
LFO
phase
freq
depth
wave
phase offs; rate
depth level
start end
wave
phase
depth
start
wave
freq
rate
end

Figure 3.13

Chapter 4

Crossfading Sound

In digital music, it is often useful to crossfade or "mix" between two signals[1]. When two signals A and B are used to produce a new signal C, crossfading means to control the amount by which signal C is composed of signal B in dependence of the amount by which it is composed of signal A. This is especially interesting when the two signals stem from ModUnits, making it possible to mix between their modulation signals. This idea lead to the creation of the `Crossfader` class, which is itself a ModUnit and will be described in this section. Equation 4.1 defines the concept of crossfading, in its most basic form, mathematically.

$$C(t) = (A(t) \times x) + (B(t) \times (1 - x)), \text{where } x \in [0; 1] \tag{4.1}$$

The following sections will examine three techniques of crossfading as described by Daniel R. Mitchell in his book *BasicSynth: Creating a Music Synthesizer in Software*, pages 96 to 99. Additionally, their implementation in C++ will be outlined.

4.1 Linear Crossfading

Linear crossfading is the simplest form of crossfading. Its mathematical definition was already given in Equation 4.1, showing that the crossfade value for the "right" signal, B, is indirectly proportional to the value for the "left" signal, A. This means that when the left value increases, the right value decreases and vice-versa. It should be noted that it is common to provide a range of $[-1; 1]$ or $[-100; 100]$ (preferred) instead of the mathematically easier range of $[0; 1]$, simply to make the interaction more intuitive to the user. Therefore, Equation 4.1 must be re-formed to Equation 4.2. Table 4.1 shows crossfade values alongside the resulting multipliers for signals A and B.

$$C(t) = (A(t) \times \frac{100 - x}{200}) + (B(t) \times \frac{100 + x}{200}), \text{where } x \in [-100; 100] \tag{4.2}$$

4.2 Crossfading with the $\sin$ function

Another technique of crossfading involves the use of the $\sin$ function to compute the multipliers for signals A and B. This is said to improve the "smoothness" of the mix (Mitchell, 2008, p. 98). Because the $\sin$ function operates with radians, the crossfade values from Equation 4.2

[1] Crossfading is most commonly known as the technique a DJ may use to mix between two pieces of music during his/her performance.

Value	Multiplier A	Multiplier B
-100	1	0
-75	0.875	0.125
-50	0.75	0.25
-25	0.625	0.375
0	0.5	0.5
25	0.375	0.625
50	0.25	0.75
75	0.125	0.875
100	0	1

Table 4.1: Crossfade values alongside the resulting multipliers for signals A and B.

must be multiplied with $\frac{\pi}{2}$, leading to Equation 4.3 for cross-fading with the $\sin$ function. It should be noted that the center value for this technique is not 0.5, but $\frac{\sqrt{2}}{2}$ or $\sin(\frac{\pi}{4})$. Table 4.2 gives cross-fade values using the $\sin$ function and the multipliers that result from them. Figure 4.1 shows how this method of crossfading compares to linear crossfading in a coordinate system.

$$C(t) = (A(t) \times \sin(\frac{100 - x}{200} \times \frac{\pi}{2})) + (B(t) \times \sin(\frac{100 + x}{200} \times \frac{\pi}{2})), \text{where } x \in [-100; 100] \quad (4.3)$$

Value	Multiplier A	Multiplier B
-100	1	0
-75	0.980	0.195
-50	0.924	0.383
-25	0.831	0.556
0	0.707	0.707
25	0.556	0.831
50	0.383	0.924
75	0.195	0.980
100	0	1

Table 4.2: Crossfade values when using the $\sin$ function, alongside the resulting multipliers for signals A and B.

4.3 Radical Crossfading

Another method of non-linear crossfading involves the computation of signal multipliers according to a square-root function. Similar to crossfading with the $\sin$ function, values do not always add up to 1 for radical crossfading. Moreover, the middle value is again $\frac{\sqrt{2}}{2}$ or $\sin(\frac{\pi}{4})$. Equation 4.4 shows the appropriate mathematical definition for radical crossfading and Table 4.5 displays crossfade values alongside resultant multipliers. Additionally, Figure 4.2 puts radical crossfading into a coordinate-system.

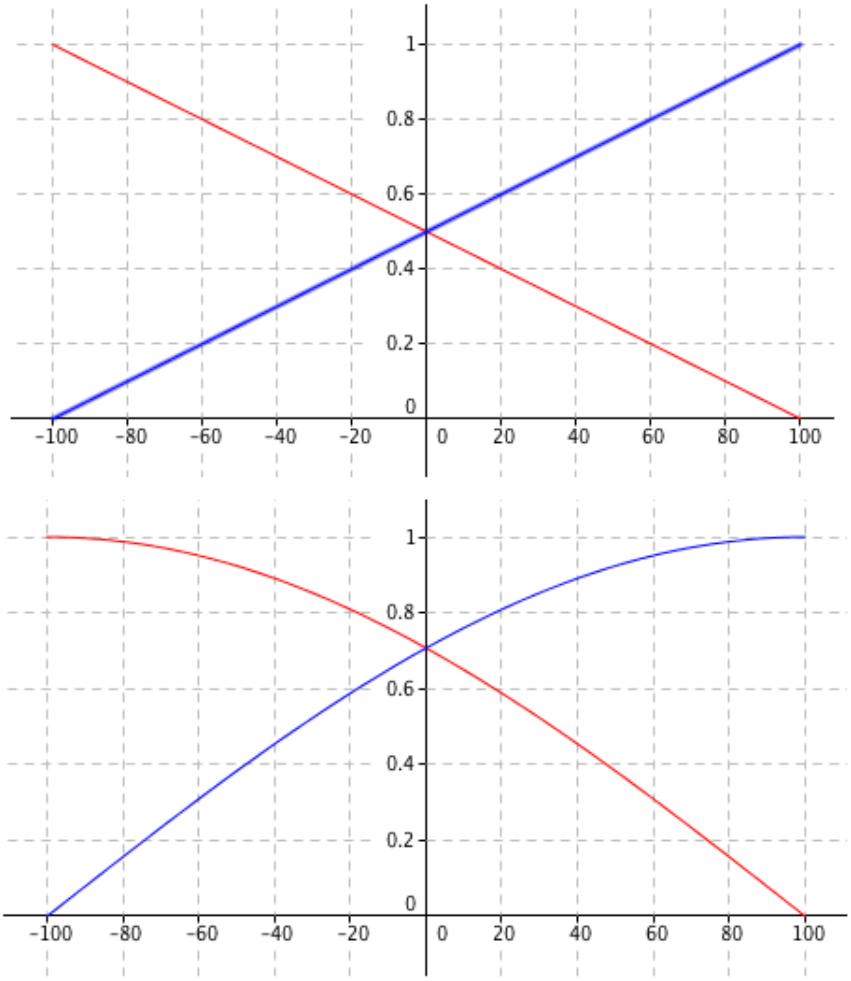


Figure 4.1: The top graph depicts linear crossfading and the bottom graph crossfading with the $\sin$ function. The red line is the multiplier for signal A and the blue line is the multiplier for signal B.

$$C(t) = (A(t) \times \sqrt{\frac{100 - x}{200}}) + (B(t) \times \sqrt{\frac{100 + x}{200}}), \text{where } x \in [-100; 100] \tag{4.4}$$

Value	Multiplier A	Multiplier B
-100	1	0
-75	0.935	0.354
-50	0.866	0.500
-25	0.791	0.612
0	0.707	0.707
25	0.612	0.791
50	0.500	0.866
75	0.354	0.935
100	0	1

Table 4.3: Radical crossfade values with resultant multipliers for signals A and B.

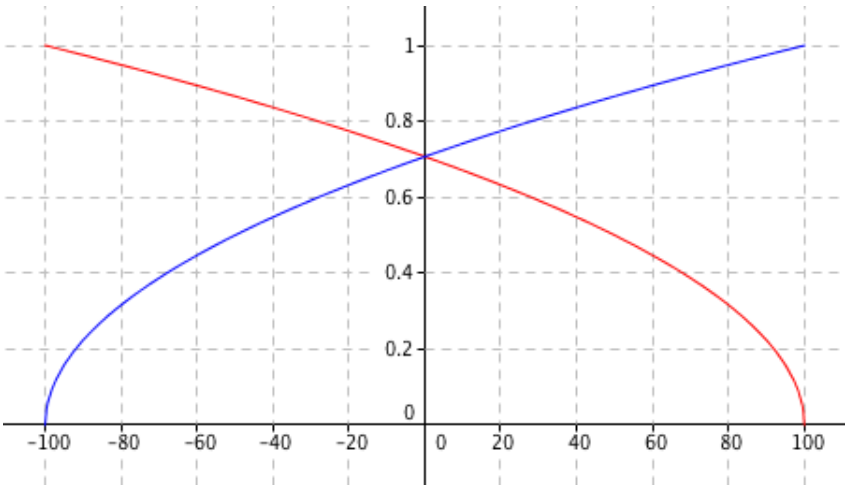


Figure 4.2: Radical crossfading shown in a coordinate-system. The red line is the multiplier for signal A and the blue line is the multiplier for signal B.

4.4 Scaling Crossfade values

It is possible to scale crossfade values resulting from radical crossfading as well as from crossfading with the $\sin$ function in such way that the multipliers at the center crossfade value are again both 0.5 instead of $\frac{\sqrt{2}}{2}$. To do so, one simply needs to multiply all multipliers by $\frac{\sqrt{2}}{2}$, since $(\frac{\sqrt{2}}{2})^2 = 0.5$. This leads to Equations 4.5, for crossfading with the $\sin$ function, and 4.6, for radical crossfading as well as Tables 4.5 and 4.6 for respective multipliers. One caveat of scaling is that all values become smaller.

$$C(t) = (A(t) \times \sin(\frac{100 - x}{200} \times \frac{\pi}{2}) \times \frac{\sqrt{2}}{2}) + (B(t) \times \sin(\frac{100 + x}{200} \times \frac{\pi}{2}) \times \frac{\sqrt{2}}{2}), \text{ where } x \in [-100; 100] \tag{4.5}$$

$$C(t) = (A(t) \times \sqrt{\frac{100 - x}{200}} \times \frac{\sqrt{2}}{2}) + (B(t) \times \sqrt{\frac{100 + x}{200}} \times \frac{\sqrt{2}}{2}), \text{ where } x \in [-100; 100] \tag{4.6}$$

Value	Multiplier A	Multiplier B
-100	0.707	0
-75	0.694	0.138
-50	0.653	0.270
-25	0.588	0.393
0	0.5	0.5
25	0.393	0.588
50	0.270	0.653
75	0.138	0.694
100	0	0.707

Table 4.4: Scaled values for crossfading with the $\sin$ function, alongside the resulting multipliers for signals A and B.

Value	Multiplier A	Multiplier B
-100	0.707	0
-75	0.661	0.25
-50	0.612	0.354
-25	0.559	0.433
0	0.5	0.5
25	0.433	0.559
50	0.354	0.612
75	0.25	0.661
100	0	0.707

Table 4.5: Scaled radical crossfade values with resultant multipliers for signals A and B.

4.5 Creating Crossfade Tables

Because crossfade values are constant, i.e. there is only one possible tuple of multipliers for each crossfade value, it was decided that these multipliers should be stored in a table once and then looked-up. This is a lot more efficient than having to calculate each multiplier at every request. Similar to Wavetables, these crossfade tables are written to a file and then written into computer memory at program start-up. The only difference between Wavetable files and crossfade table files is that the former are stored in binary and the latter in plain-text, as the the size of crossfade tables is only 201 instead of 4096, the size of a Wavetable. Table 4.6 shows a computer program, this time written in Python, to compute and store crossfade tables.

```
import math

lin = []
sin = []
sqrt = []

sinsc = []
sqrtsc = []

m = math.pi / 2
q = math.sqrt(2) / 2

for n in range(201):

    val = n - 100

    left = (100 - val) / 200

    right = (100 + val) / 200

    lin.append((left,right))

    sin.append((math.sin(left * m), math.sin(right * m)))

    sqrt.append((math.sqrt(left),math.sqrt(right)))

    sinsc.append((sin[n][0] * q, sin[n][1] * q))

    sqrtsc.append((sqrt[n][0] * q, sqrt[n][1] * q))


tables = [lin,sin,sqrt,sinsc,sqrtsc]

names = ["linear", "sine", "sqrt", "sine_scaled", "sqrt_scaled"]

for n,t in zip(names,tables):

    t = ["{} {}".format(i[0], i[1]) for i in t]

    s = "\n".join(t)

    with open(n+".table","wt") as f:

        f.write(s)
```

Table 4.6

Chapter 5

Filtering Sound

A feature any digital synthesizer should include is the possibility to filter sound. In digital signal processing (DSP) and more specifically in audio processing, to filter a signal means to "alter [its] frequency spectrum [...] by amplifying or attenuating selected frequencies". (Mitchell, 2008, p. 102).

The discussion of digital filters is closely linked to the concept of a signal's phase and how the phase relationship between two *input* signals influences the properties of the *output* signal produced when the two input signals interact and interfere either constructively or destructively. A simple examples of phase relationships from the analog realm is the situation where signals A and B are exactly identical and consequently have a phase difference $\Delta\phi$of 0. If these signals are summed, the resulting signal C will have an amplitude twice as high or low as that of signal A or B in every single point of its signal — the input signals interfere *constructively*. Conversely, if signals A and B have a phase shift of 180°or π radians, the two signals interefere *destructively* and result in signal C having a constant amplitude of exactly 0, as the amplitude values of signals A and B cancel each other out in each point of the signal.

In digital systems, where signals are represented discretely, i.e. with periodically recorded samples, this behaviour can be modeled by applying a *sample delay*. (Mitchell, 2008, p. 103) For example, if a signal that is periodically represented by 10 samples is summed with a version of itself that is delayed by 5 samples, that version will effectively be phase-shifted by 180°and thus the summation of these two signals will result in silence. Had only one sample been delayed instead of five, the phase difference $\Delta\phi$ would have only been 36°or $\frac{2\pi}{10}$ radians — one tenth of a period. The amount of phase difference $\Delta\phi$ depends on both the number of samples a signal is represented by, so the sample rate of the system, and the number of samples that are delayed. An equation to calculate the phase difference in radians, given these parameters, is shown in Equation 5.1, where f is the frequency of the signal, f_s the sample rate and N the number of samples delayed. If 2π is replaced by 360, this Equation yields the same phase difference in degrees.

$$\Delta\phi = \frac{N \times f}{f_s} \times 2\pi \tag{5.1}$$

From Equation 5.1 it can be deduced that a sample delay causes different phase relationships for different frequencies of the original signal. To give an example: let one sample be delayed in a system with a sample rate f_s equal to 20 kHz. In such a system, a signal with a frequency

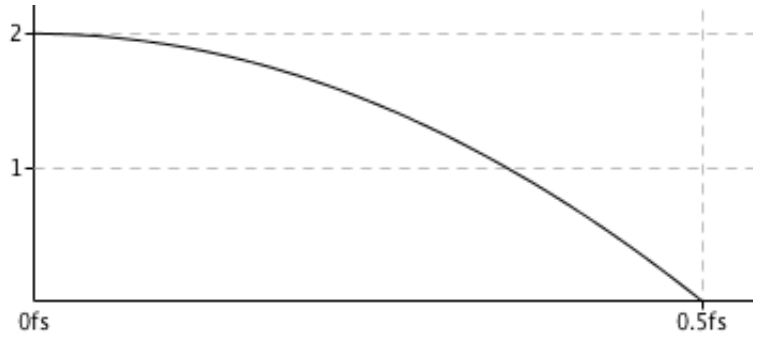


Figure 5.1: The frequency response of a one-sample-delay filter. The most constructive interference is at 0 Hz, a signal that is constant with $f(t) = 1$. Such a signal with a frequency of 0 Hz is also called DC, for direct current, because in analog circuits a DC current is constant. The most destructive interference occurs at $\frac{f_s}{2}$ Hz.

of 1 Hz is represented by 20000 samples. Consequently, summing such a signal with a version of itself delayed by one sample would make the resulting, filtered signal almost twice as loud, as $\Delta\phi$ is only about 0.018°. However, a signal with the maximum possible frequency of 10 Khz (the Nyquist limit) is represented by only 2 samples. Therefore, delaying one sample equals a phase shift of 180°, which results in silence when the signal is summed with the delayed version of itself. This introduces the concept of a filter's *frequency response*, which describes how a filter influences a signal's frequency spectrum. For the filter just described, the frequency response would look approximately like Figure 5.1.

One way to change the frequency response of a filter is to weight samples of the input signal as well as the delayed samples with *filter coefficients* (Mitchell, 2008, p. 105, 110). Whereas previously a one-sample delay could be modeled by Equation 5.2 (Mitchell, 2008, p. 104), where y_n is the output, x_n the current and x_{n-1} the delayed sample, Equation 5.3 (Mitchell, 2008, p. 103), where a and b are the respective filter coefficients, must be used when samples are weighted with coefficients. Visually, such a one-sample delay can be represented with the diagram shown in Figure 5.2.

$$y_n = x_n + x_{n-1} \tag{5.2}$$

$$y_n = bx_n + ax_{n-1} \tag{5.3}$$

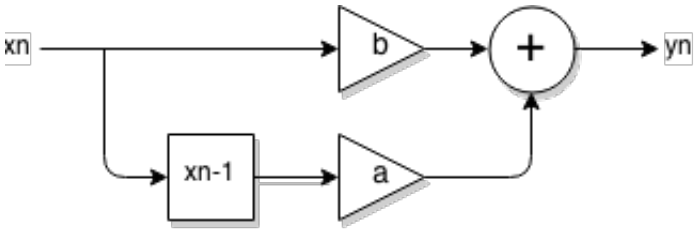


Figure 5.2: A one-sample delay with filter coefficients a and b. Triangles are amplifiers/attenuators. Samples are summed at the circle with the + at its center.

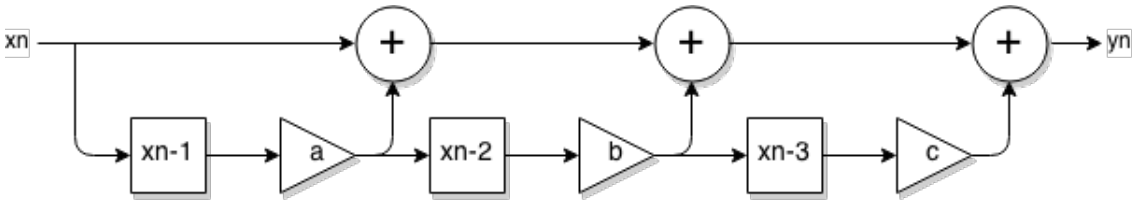


Figure 5.3: A three-sample delay filter.

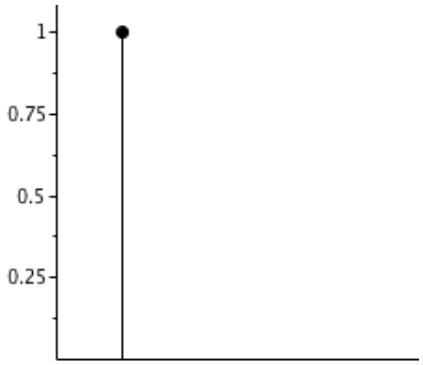


Figure 5.4: A delta function, also called a unit impulse. The first sample has an amplitude of 1 while all other samples are equal to 0.

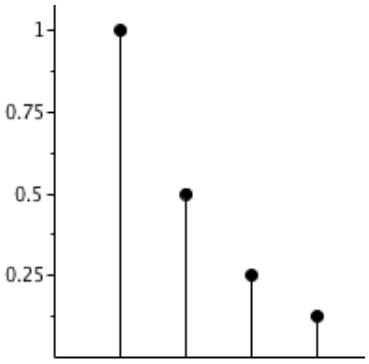


Figure 5.5: The impulse response of the three-sample delay filter depicted in Figure 5.3.

5.1 FIR and IIR Filters

While a filter's frequency response describes how the frequency content of a signal is affected by that filter, its influence on a signal in the time domain is given by the filter's *impulse response* or *kernel*. More precisely, an impulse response determines what a filter outputs when its input is a delta function, δ, which has a normalized *impulse*, meaning a sample with an amplitude of 1, as its first sample while all other subsequent samples have an amplitude of 0. The δ function is also referred to as the *unit impulse*. (Smith, 1999, p. 108) For example, if the filter coefficients of the three-sample delay filter shown in Figure 5.3 are all equal to 0.5 ($a = b = c = 0.5$), leading to Equation 5.4, then Figure 5.5 gives the impulse response of this filter when it is fed the δ function depicted in Figure 5.4. What this impulse response shows is that if a sample x_n with an amplitude of 1 is input into this filter, the first sample output will be $1 + \frac{0}{2} + \frac{0}{2} + \frac{0}{2} = 1$. The second sample to exit this filter will then be $0 + \frac{1}{2} + \frac{0}{2} + \frac{0}{2} = 0.5$, where, following the definition of the δ function given above, the new x_n is equal to 0, while the previous x_n that was equal to 1 has moved into the position of x_{n-1}. The third sample will be equal to $0 + \frac{0}{2} + \frac{0.5}{2} + \frac{0}{2} = 0.25$ and finally the fourth sample has a value of $\frac{1}{8}$.

$$y_n = x_n + \frac{x_{n-1}}{2} + \frac{x_{n-2}}{2} + \frac{x_{n-3}}{2} \tag{5.4}$$

The operation that was just performed on the δ function is called "convolution". Convolution has its own mathematical symbol: $*$. To convolve a signal means to apply an impulse response to every of its samples. Table 5.1 shows a C++ program to perform convolution (based on pseudo-code cf. Mitchell, 2008, p. 107).

```cpp
#include <vector>

std::vector<double> convolve(const std::vector<double>& signal,
                             const std::vector<double>& impulseResponse)
{
    std::vector<double> output(signal.size(),0);

    for(std::vector<double>::size_type i = 0, endI = signal.size();
        i < endI;
        ++i)
    {
        for(std::vector<double>::size_type j = 0, endJ = impulseResponse.size();
            j < endJ;
            ++j)
        {
            output[i + j] += signal[i] * impulseResponse[j];
        }
    }

    return output;
}
```

Table 5.1: A C++ function to convolve a signal vector with an impulse response vector. It should be noted that to convolve a signal vector of size m with an impulse response vector of size n, the output vector must have a minimum size of $m + (n - 1)$.

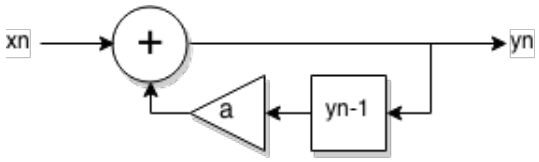


Figure 5.6: An IIR delay filter.

The filters described so far are called Finite Impulse Response (FIR) filters, because their impulse response is of finite size, like the one shown in Figure 5.5, caused by the Filter depicted in Figure 5.3. Given that in this three-sample delay filter all filter coefficients have the same value of 0.5, the diagram could actually be re-arranged and simplified if the three delays are replaced by a single *recursive* delay. Instead of processing an input sample, a recursive delay is fed an output sample, which it then amplifies or attenuates and feeds back into the signal. This re-arranged filter, also called a *feedback loop*, is shown in Figure 5.6. Because this feedback loop could theoretically go on forever, such a filter is called an Infinite Impulse Response (IIR) filter. Equation 5.5 gives a mathematical definition for the filter shown in Figure 5.6. To prove that this filter results in the same response as the FIR filter from before, it can be fed a δ function. Because y_{n-1} is initially 0, the first output y_n will simply be $1 + \frac{0}{2}$. For the second sample, y_{n-1} takes on the value of the previous output, which was 1. Therefore, the second output sample is $0 + \frac{1}{2} = 0.5$, the next sample $0 + \frac{0.5}{2}$ and so on. Because this is an IIR filter, this process could theoretically go on ad infinitum. x_n is equal to 0 for all samples other than the first because of the way the δ function is defined.

$$y_n = x_n + \frac{y_{n-1}}{2} \tag{5.5}$$

5.2 Bi-Quad Filters

A common arrangement of FIR and IIR filters is a so-called bi-quad filter. A "bi-quad filter combines one and two sample feedforward delays (FIR) with one and two sample feedback delays (IIR)" (Mitchell, 2008, p. 109). Figure 5.7 shows such a bi-quad filter. The Equation to calculate the sample output of a bi-quad filter is given by Equation 5.6.

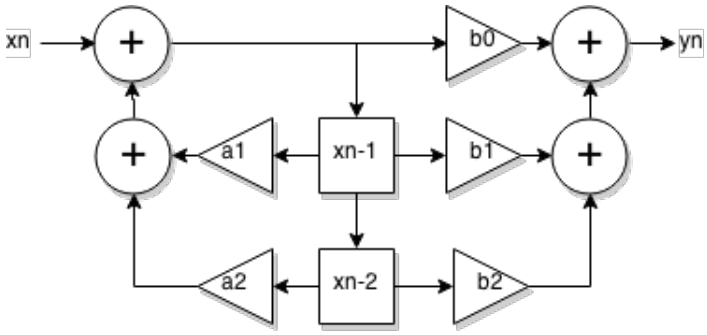


Figure 5.7: A bi-quad filter in direct form II. In direct form I, the filter is turned inside-out. The result is the same, however direct form II is a little more concise to implement.

$$y_n = b_0(x_n - a_1 y_{n-1} - a_2 y_{n-2}) + b_1 x_{n-1} + b_2 x_{n-2} \tag{5.6}$$

5.2.1 Implementation

In the synthesizer created for this thesis, the `Filter` class implements a bi-quad filter. Table 5.2 shows the `Filter` class' `process` method, which takes an input sample and returns a filtered output sample.

```
1  double Filter::process(double sample)
2  {
3      double temp = sample
4      - (coefA1_ * delayA_)
5      - (coefA2_ * delayB_);
6
7      double output = (coefB0_ * temp)
8      + (coefB1_ * delayA_)
9      + (coefB2_ * delayB_);
10
11
12     // Store values into delay line
13     delayB_ = delayA_;
14     delayA_ = temp;
15
16     return output;
17 }
```

Table 5.2: C++ function to filter an input sample and return an output sample. This function implements the bi-quad filter equation given by Equation 5.6.

5.3 Filter Types

Bi-quad filters can be used to create a variety of different filter types and corresponding frequency responses, simply by adjusting the filter's coefficients. When examining a frequency response, the term **passband** is used for the frequency range that should ideally not be affected by the filter. Conversely, the **stopband** are those frequency components the filter should, ideally, silence completely. The frequency at which the transition from minimum to maximum attenuation occurs is called the **cutoff frequency** and is denoted by f_c. Because no filter is perfect, this transition usually does not happen precisely at the specified cutoff frequency. Rather, there is a certain **transition band**, which is the range of frequencies where the transition takes place. Within this transition band, the cutoff frequency is defined as the frequency component where the amplitude reaches $\frac{\sqrt{2}}{2}$ (0.707...) or -3dB in power[1]. If the transition band is narrow, the filter is said to have a fast **roll-off**. If it is rather wide, the roll-off is said to be slow. (Smith, 1999, p. 268) Figure 5.8 depicts a typical frequency response and labels the relevant bands and frequencies. It should be noted that bi-quad filters and IIR filters in general show quite slow roll-off.

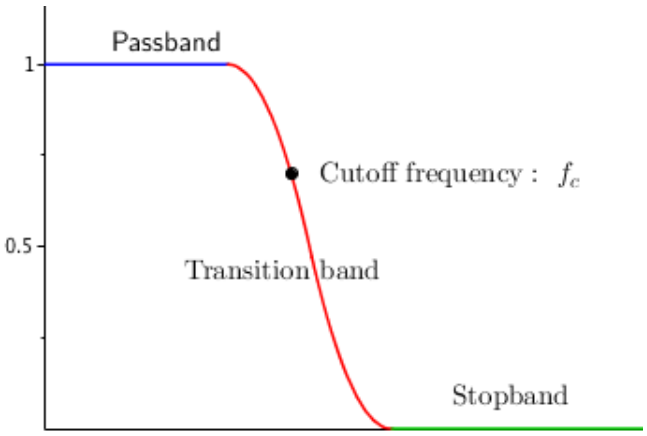


Figure 5.8: The frequency response of a filter with a relatively fast roll-off.

[1]A signal's power is measured in decibels (dB), which is a logarithmic unit of measurement. 20dB equals an increase in amplitude by one order of magnitude ($\times 10$) and -20dB a decrease by one order of magnitude ($\times 0.1$).

5.3.1 Low-Pass Filters

Low-Pass filters have their passband in the lower frequency ranges and their stopband in the higher ranges. A visualization of a low-pass bi-quad filter's frequency response when applied to a white noise signal[2] is given in Figure 5.8.

5.3.2 High-Pass Filters

A high-pass filter is a low-pass filter whose spectrum has either been *inverted* (flipped top-for-bottom) or *reversed* (flipped left-for-right) (Smith, 1999, p. 271). As a consequence, high-pass filters let high frequencies pass, while stopping lower frequencies. Figure 5.10 shows an appropriate frequency response.

5.3.3 Band-Pass Filters

Band-pass filters let frequency components in a specific range, e.g. 3000 to 5000 Hz, pass, while stopping all other frequencies lower or higher than the range of the passband. Such a filter can be created by first sending a signal through a low-pass and then through a high-pass filter. The passband of the filter created will then be the range of frequencies where the passband of the high-pass filter intersects that of the low-pass filter, since all other ranges are in a stopband. Figure 5.11 shows the frequency response of a band-pass filter.

5.3.4 Band-Reject Filters

A band-reject filter, often referred to as a "notch" filter, can be seen as the opposite of a band-pass filter. While a band-pass filter stops all frequencies except for those immediately around the cutoff frequency, a band-reject filter stops only the frequencies at the cutoff frequency and lets all others pass. Besides using the correct filter coefficients for a bi-quad filter, a band-reject filter can also be created by using a low-pass and a high-pass filter *in parallel*. This involves copying a signal, then sending one copy of the original signal through a low-pass filter and the other copy through a high-pass filter. The final signal is the sum of the signal output by the high-pass filter and the signal output by the low-pass filter. Figure 5.12 displays the frequency response of a band-reject filter.

[2] A white noise signal is used here because it has a flat frequency spectrum when un-filtered.

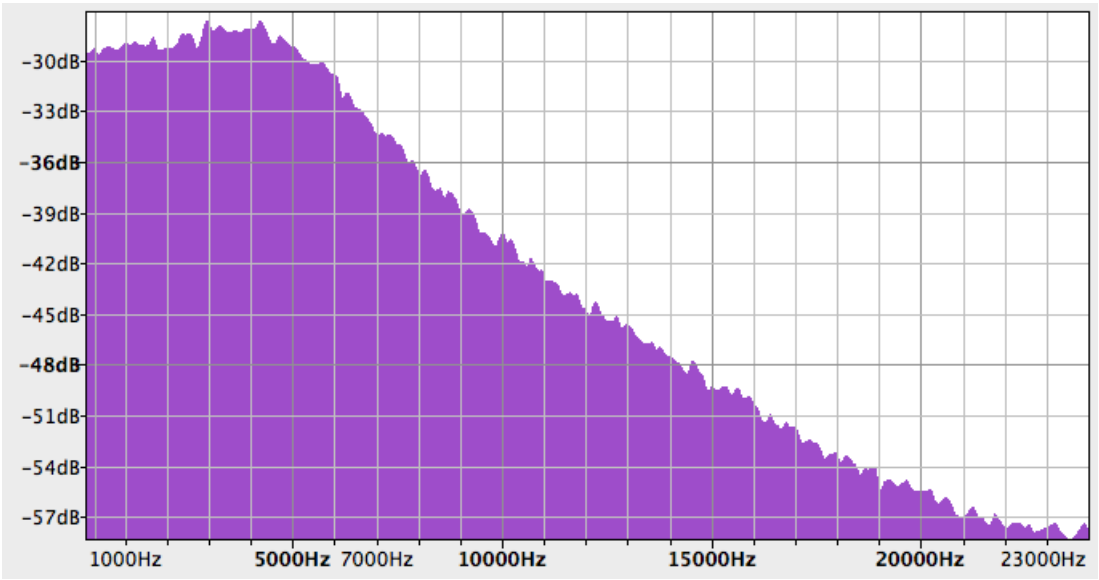


Figure 5.9: The frequency response of a low-pass bi-quad filter where $f_c = 5000$ Hz.

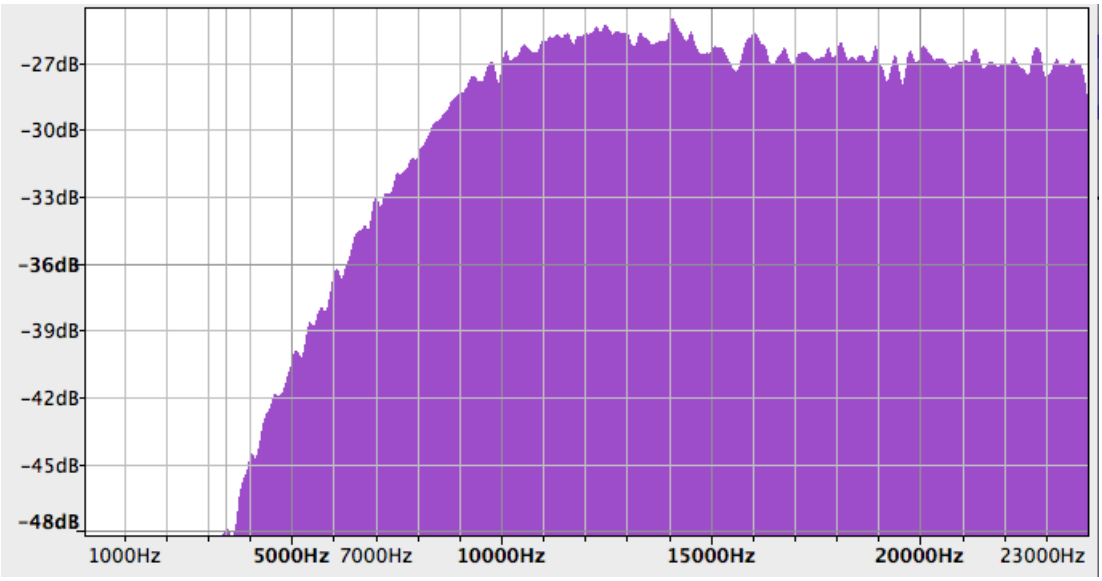


Figure 5.10: The frequency response of a high-pass bi-quad filter where $f_c = 10000$ Hz.

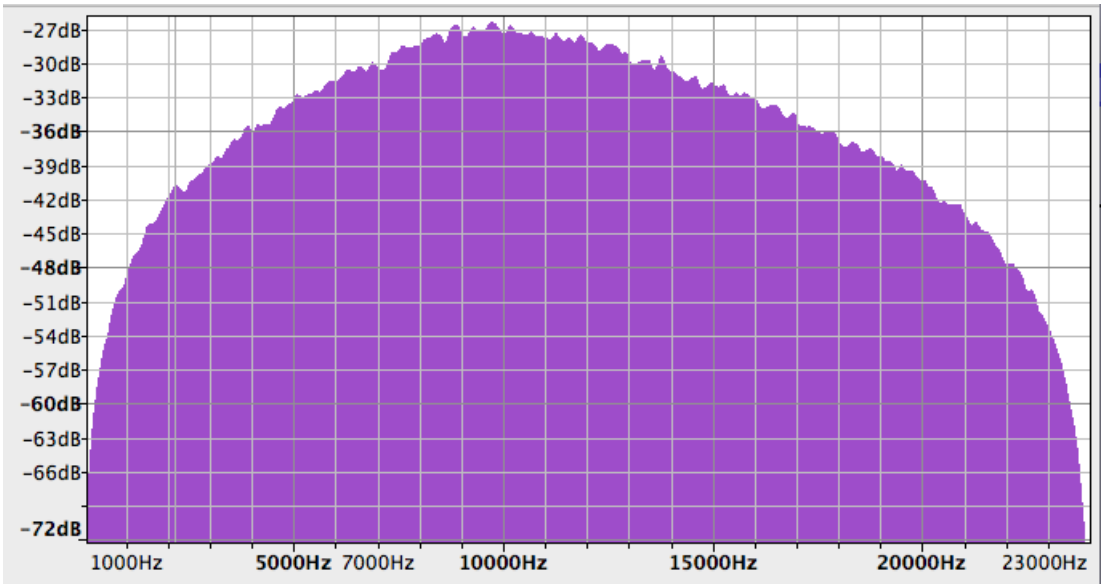


Figure 5.11: The frequency response of a band-pass bi-quad filter where $f_c = 10000$ Hz.

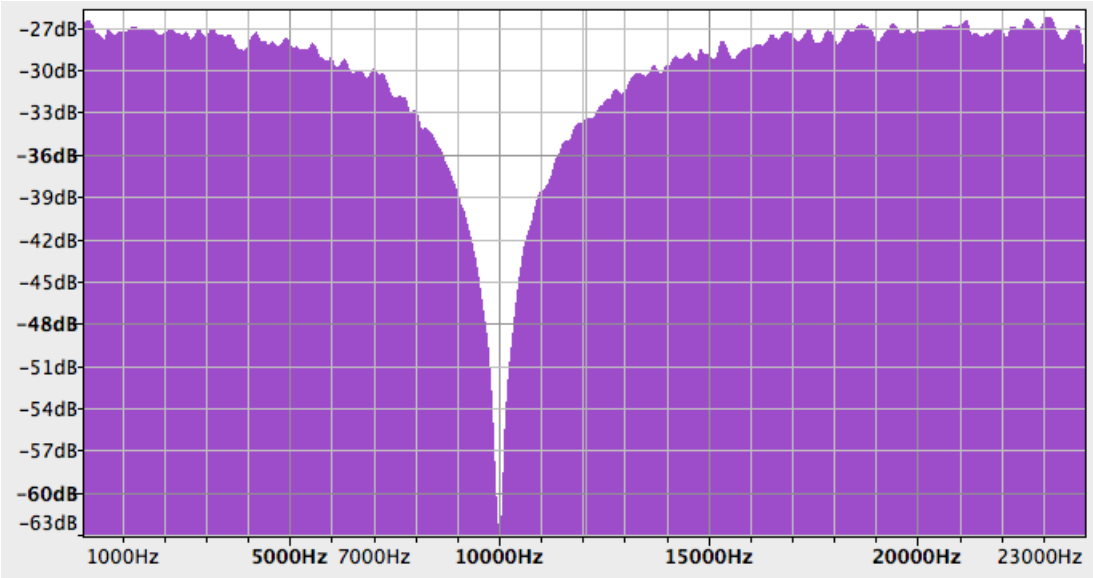


Figure 5.12: The frequency response of a band-reject bi-quad filter where $f_c = 10000$ Hz.

5.3.5 All-Pass Filters

A very special type of filter is the so-called all-pass filter. As its name implies, an all-pass filter lets all frequencies pass and stops none. What could possibly be the use of such a filter? The answer lies in the second alteration a filter performs on a signal: a change in phase. All-pass filters are used to change a signal's phase while leaving its amplitude un-altered at all frequencies.

5.4 Filter Coefficients

Let $\omega = \frac{2\pi f_c}{f_s}$ and $\alpha = \frac{\sin(\omega)}{2Q}$, where Q is the filter's *Quality Factor*[3], then Table 5.3 gives the values the various filter coefficients of a bi-quad filter must take on to achieve the wanted frequency response. These values were determined by Robert Bristow-Johnson from analog filter circuits [4].

Filter Type	a_1	a_2	b_0	b_1	b_2
Low-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1-\cos(\omega)}{2(1+\alpha)}$	$\frac{1-\cos(\omega)}{1+\alpha}$	$\frac{1-\cos(\omega)}{2(1+\alpha)}$
High-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1+\cos(\omega)}{2(1+\alpha)}$	$\frac{-(1+\cos(\omega))}{1+\alpha}$	$\frac{1+\cos(\omega)}{2(1+\alpha)}$
Band-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{\sin(\omega)}{2(1+\alpha)}$	0	$\frac{-\sin(\omega)}{2(1+\alpha)}$
Band-Reject	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1}{1+\alpha}$	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1}{1+\alpha}$
All-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{-2\cos(\omega)}{1+\alpha}$	1

Table 5.3: Filter coefficients for a bi-quad filter.

[4]Source: `http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt`

[4]The Quality Factor can be said to control to some extent the bandwith of a filter's transition band as well as the amplitude peak of the passband before transitioning to the stopband.

Chapter 6

Applying Effects to Sound

While the digital revolution made it possible to create popular effects like Echos, Flangers or Reverbs more efficiently using logic circuits and digital signal processing, most effects found in digital synthesizers today have a long history of analog implementation. Rock stars like Jimi Hendrix, famous for using the "Wah-Wah" effect, applied effects to their sound long before digitization. The following sections will describe four common effects and explain their implementation.

6.1 Delay Lines and the Delay Effect

Delay lines store samples and *delay* their retrieval by a certain number of sample times. Just like delay lines are at the foundation of many digital filters, they are also the primary building block of a great number of different effects. Consequently, they must be examined and discussed thoroughly before attempting to create any other digital effects. However, having a well-founded understanding of delay lines also makes the implementation of many effects a trivial task. It should be noted that when a delay line is used as an effect simply to delay samples, it will be referred to as a "Delay" effect.

6.1.1 Simple Delay Lines

If the time by which a sample is delayed is constant for each sample, a delay line can be implemented using a simple First-In-First-Out (FIFO) data-structure such as a Queue. Figure 6.1 visualizes such a delay line and Table 6.1 shows a C++ implementation. (Mitchell, 2008, p. 118)

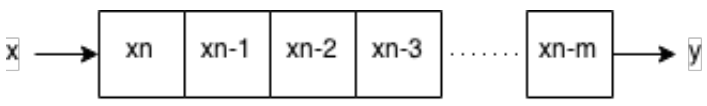


Figure 6.1: Visualization of a simple FIFO delay line.

```
class Delay
{
public:

    typedef unsigned long index_t;

    Delay(index_t length)
    : line_(new double [length]()),
      length_(length)
    { }

    ~Delay()
    {
        delete [] line_;
    }

    double process(double sample)
    {
        // Retrieve output sample
        double output = line_[0];

        // Move all elements forward by one
        for (index_t i = 0, j = 1; j < length_; ++i, ++j)
        {
            line_[i] = line_[j];
        }

        // Push the new sample into
        // into the delay line
        line_[length_ - 1] = sample;

        return output;
    }

private:

    index_t length_;

    double* line_;
};
```

Table 6.1: A C++ class that implements a very simple delay line of fixed size.

6.1.2 Flexible Delay Lines

While simple delay lines are easy to implement, they are not efficient if the delay time is not constant, as changing the size would require the re-allocation of memory to suit the new size and subsequent copying of all items from the old delay line into the new one. Moreover, the moving of all samples by one index after retrieving the latest output sample is highly inefficient, especially for long delay times. A more reasonable approach is to implement a delay line as a circular buffer with a fixed maximum length and then to vary the distance between a read and a write iterator relative to the current delay time. The circularity is achieved by wrapping the read or the write index back to the front of the buffer if either reaches the end. Figure 6.2 shows such a circular buffer in an abstract visualization, where the write iterator, which points to the position where incoming samples are stored into the delay line, is currently at index 0. The Figure also shows two possible positions of the read iterator, from which delayed samples are retrieved, which both result in a different delay time. It should be noted that read iterator A and B do not exist simultaneously, they just show possible indices for the read iterator. Read position A would cause a delay of 5 sample times while read position B would result in a 20 sample delay. Figure 6.3 shows the same buffer and iterators in a more realistic sequential layout, as it is stored in computer memory.

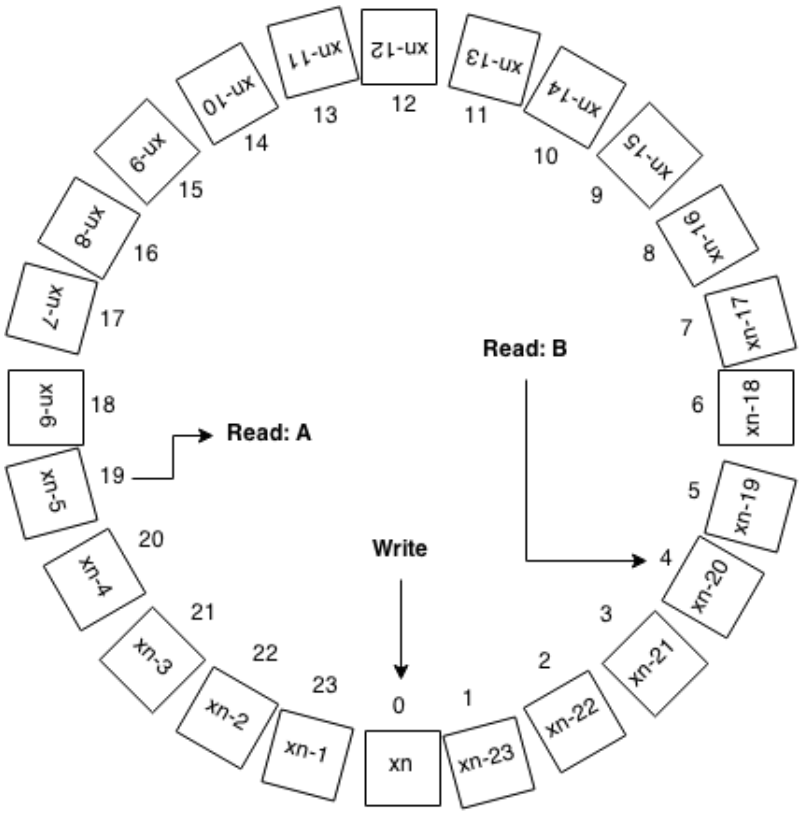


Figure 6.2: A circular delay buffer with two different possible read positions A and B. The labels inside the boxes are the relative sample delays and the labels outside the boxes are the sequential indices. The write iterator is where new samples are stored and the read iterators are where delayed samples are retrieved from.

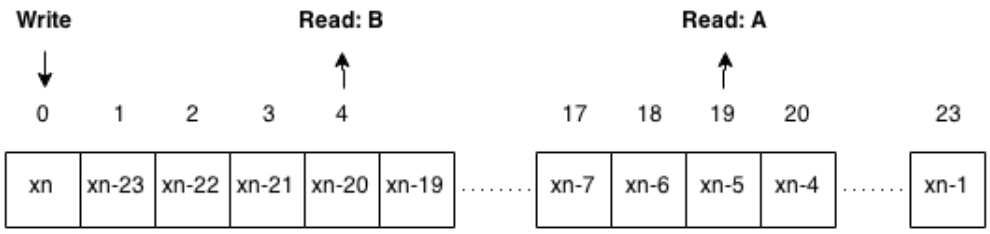


Figure 6.3: The buffer from Figure 6.2 in a sequential layout.

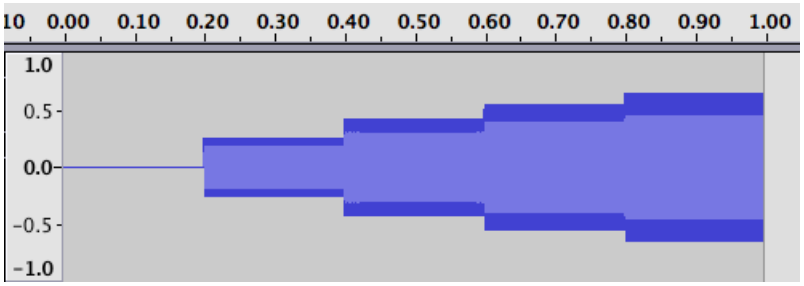


Figure 6.4: A sound wave resulting from a resonator. The initial silence is caused by the delay line filling up and delayed samples consequently still being equal to 0. The increase in amplitude per delay time (200 ms) is due to the feedback control. Because the decay is not 0, the amplitude does not double per delay time as it would if there were no decay and full feedback. Rather, the sound decays with time and would reach its maximum after 4 seconds.

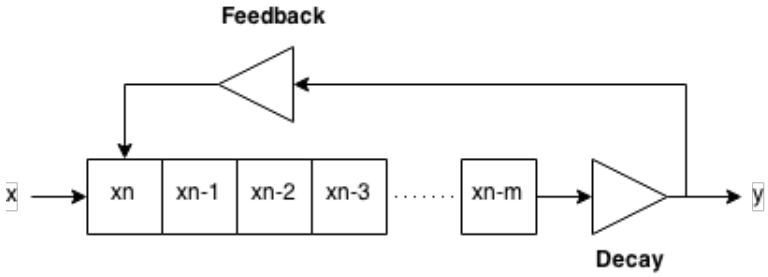


Figure 6.5: A resonating delay line.

6.1.3 Interpolation

It is also possible to have fractional delay values if samples are interpolated. Equation 6.1 shows how this interpolation process is performed, where i is the integral and f the fractional portion of the n-sample delay. Note that this is the same interpolation algorithm that is performed to retrieve sample values from fractional Wavetable indices.

$$y_n = x_i + ((x_{i-1} - x_i) \times f) \tag{6.1}$$

6.1.4 Feedback and Decay

Two parameters commonly associated with delay lines are *feedback* and *decay*. The feedback control of a delay line determines how much of the output signal is fed back into the delay line together with the input signal. The decay parameter determines the level of attenuation of the output signal as it leaves the delay line. It is "[...] applied each time the [fed-back] sample travels through the delay line and is equivalent to the dampening, or decay, of a signal over time". When these two parameters are used together, such a delay line is called a "Resonator". Figure 6.5 shows a block diagram for such a resonator and Figure 6.4 displays a sound wave resulting from a resonator. (Mitchell, 2008, p. 120)

6.1.5 Dry/Wet Control

Almost every effect, including the Delay effect, has what is called a "dry/wet" control. If the dry signal is the unchanged input signal and the wet signal the fully processed output signal,

the dry/wet control determines how much of an effect is applied to the dry signal. Equation 6.2 gives a mathematical definition for this principle, where dw is the dry/wet value between 0 (no effect) and 1 (full effect), x_n the unchanged, "dry", input sample and y_n the "wet" output signal from the effect.

$$z_n = (x_n \times (1 - dw)) + (y_n \times dw) \tag{6.2}$$

6.1.6 Implementation

In the synthesizer created for this thesis, a flexible delay line with all the properties and controls just mentioned is implemented in the `Delay` class. Relevant processing and updating methods of the `Delay` class are shown in Table 6.2.

```
1  double Delay::dryWet_(double originalSample, double processedSample)
2  {
3      return (originalSample * (1 - dw_)) + (processedSample * dw_);
4  }
5
6  void Delay::writeAndIncrement_(double sample)
7  {
8      *write_ = sample;
9
10     if (++write_ >= buffer_.end())
11     {
12         write_ -= buffer_.size();
13     }
14 }
15
16 double Delay::process(double sample)
17 {
18     const_iterator read = write_ - readIntegral_;
19
20     if (read < buffer_.begin())
21     {
22         read += buffer_.size();
23     }
24
25     if (! readIntegral_)
26     {
27         writeAndIncrement_(sample);
28     }
29
30     double output = *read;
31
32     if (--read < buffer_.begin())
33     {
34         read += buffer_.size();
35     }
36
37     output += (*read - output) * readFractional_;
38
39     output *= decayValue_;
40
41     if (readIntegral_)
42     {
43         writeAndIncrement_(sample + (output * feedback_));
44     }
45
46     return dryWet_(sample, output);
47 }
```

Table 6.2: Relevant member functions of the `Delay` class that implements a flexible delay line with feedback, decay, interpolation and dry/wet control.

6.2 Echo

The first effect that can be implemented very easily using just delay lines is the Echo effect. An echo is the result of summing an input sample with the output sample of a delay line. An implementation from the `Echo` class, derived from the `Delay` class, is shown in Table 6.3.

```
1 double Echo::process(double sample)
2 {
3     double output = sample + Delay::process(sample);
4
5     return dryWet_(sample, output);
6 }
```

Table 6.3: The `process` method of the `Echo` class. This shows that an Echo is just the input sample summed with the output from the delay line.

6.3 Flanger

A Flanger effect is created by mixing a signal with a delayed version of itself and varying the time by which it is delayed with a Low Frequency Oscillator. The middle delay time around which the LFO oscillates is called the "center" value. The term "depth" is used for the value that is added or subtracted to the center value periodically. For example, if the center value is 10 ms and the depth 4 ms, the delay time will oscillate between 6 ms and 14 ms. The "rate" parameter controls how fast the LFO oscillates. A Flanger's "feedback" control is not the same as for delay lines, as it does not control how much of the output signal is fed back into the delay line. Rather, the feedback value determines how much of the sample at the center delay value is subtracted from the input sample. Varying the feedback controls the coloration and brightness of the sound produced. Lastly, a Flanger also has a dry/wet parameter, whose function was already described. Figure 6.6 depicts a block diagram for a Flanger. The sound produced by a Flanger effect can be described as a "swooshing" sound. (Mitchell, 2008, p. 136) Table 6.4 shows the `Flanger` class' `process` method.

6.4 Reverb

When a musician plucks a string on his guitar or bangs a percussion drum in a closed room, the sound that reaches a listener's ears does not stem solely from the sound's source. Rather, the sound emited from the musical instrument reflects off the room's walls and ceiling as well as any other object it meets on the way to the listener. This mixture of original and reflectd sound is called reverberation (Mitchell, 2008, p. 129). Naturally, the degree of reverberation depends, among other things, to a large part on the space of the room in which the sound is emitted. For example, an organ will sound differently if played in a large cathedral than it will in a small classroom. Therefore, any parameters that control the degree of reverberation essentially determine the size of the "virtual room" in which the sound is played. Most reverberators, reverbs for short, allow the user to control the reverb "time" and the reverb "rate". The reverb time determines how long it takes for the sound to reach inaudible levels, while the reverb rate controls at what rate this fading out occurs. Finally, there is also a dry/wet control as for all other effects as well.

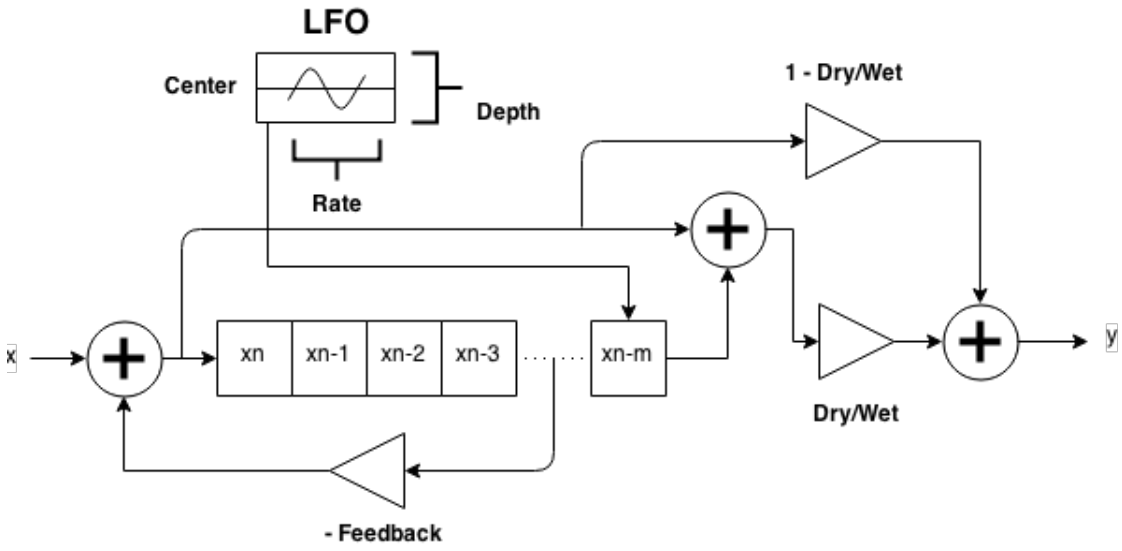


Figure 6.6: Block diagram for a Flanger effect. The LFO controls the delay length. The feedback is negative because it is subtracted from the input sample.

```
1  double Flanger::process(double sample)
2  {
3      double output = sample;
4
5      // Check for feedback
6      if (feedback_)
7      {
8          output -= delay_->offset(center_) * feedback_;
9      }
10
11     // Calculate new length by modulation. Modulation
12     // depth and maximum are 1 because the LFO's amplitude
13     // is the delay depth value
14     double length = lfo_->modulate(center_, 1, 1);
15
16     // Increment LFO
17     lfo_->update();
18
19     // Set the new length
20     delay_->setDelayTime(length);
21
22     // Retrieve new sample
23     output += delay_->process(output);
24
25     // Apply dry/wet
26     return dryWet_(sample, output);
27 }
```

Table 6.4: Member function of the `Flanger` class that implements flanging.

6.4.1 Schroeder Reverb

A relatively well-known and moderately popular reverb algorithm is the so-called "Schroeder Reverb", which consists of four parallel delay lines whose signals are fed into two all-pass filters connected in series. Figure 6.7 depicts a block diagram for a Schroeder Reverb. To simulate how sound waves reflect off different surfaces in a room, the delay lines are given different lengths. The decay parameter of these delay lines is determined by the reverb time and rate. The Schroeder Reverb's all-pass filters have fixed delay times and decay values. Table 6.5 gives these various delay line lengths and decay values. (Mitchell, 2008, p. 133)

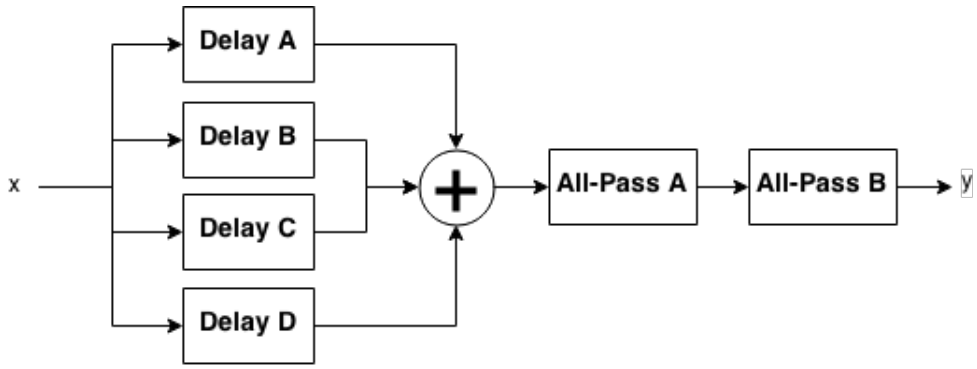


Figure 6.7: Block diagram for a Schroeder Reverb effect.

Component	Delay Time	Decay Value
Delay A	0.0297	Variable
Delay B	0.0371	Variable
Delay C	0.0411	Variable
Delay D	0.0437	Variable
All-Pass A	0.09638	0.0050
All-Pass B	0.03292	0.0017

Table 6.5: Delay times and decay values for a Schroeder Reverb.

6.4.2 Implementation

Table 6.6 shows relevant member of function of the `Reverb` class, which implements a Schroeder Reverb.

```
Reverb::Reverb(double reverbTime, double reverbRate, double dryWet)
: delays_(new Delay[4]),
  allPasses_(new AllPassDelay [2])
{

    delays_[0].setDelayTime(0.0437);
    delays_[1].setDelayTime(0.0411);
    delays_[2].setDelayTime(0.0371);
    delays_[3].setDelayTime(0.0297);

    allPasses_[0].setDecayTime(0.0050);
    allPasses_[0].setDelayTime(0.09638);

    allPasses_[1].setDecayTime(0.0017);
    allPasses_[1].setDelayTime(0.03292);

    setReverbTime(reverbTime);
    setReverbRate(reverbRate);
}

double Reverb::process(double sample)
{
    double output = 0;

    for (unsigned short i = 0; i < 4; ++i)
    {
        output += delays_[i].process(sample);
    }

    output = allPasses_[1].process(allPasses_[0].process(output));

    return dryWet_(sample, output);
}

void Reverb::setReverbRate(double reverbRate)
{
    for (unsigned short i = 0; i < 4; ++i)
    {
        delays_[i].setDecayRate(reverbRate);
    }
}

void Reverb::setReverbTime(double reverbTime)
{
    for (unsigned short i = 0; i < 4; ++i)
    {
        delays_[i].setDecayTime(reverbTime);
    }
}
```

Table 6.6: This code exerpt from the `Reverb` class shows how the Schroeder Reverb's delay lines and all-pass filters are initialized and then used to reveberate a signal.

Appendices

Appendix A

Code Listings

```cpp
#include <cmath>

int main(int argc, char * argv[])
{
  // The sample rate, 48 kHz.
  const unsigned short samplerate = 48000;

  // The duration of the generated sine wave, in seconds.
  const unsigned long duration = 1;

  // The number of samples that will be generated, derived
  // from the number of samples per second (the sample rate)
  // and the number of seconds to be generated for.
  const unsigned long numberOfSamples = duration * samplerate;

  const double pi = 3.141592653589793;

  const double twoPi = 6.28318530717958;

  // The frequency of the sine wave
  double frequency = 1;

  // The phase counter. This variable can be seen as phi.
  double phase = 0;

  // The amount by which the phase is incremented for each
  // sample. Since one period of a sine wave has 2 pi radians,
  // dividing that value by the sample rate yields the amount
  // of radians by which the phase needs to be incremented to
  // reach a full 2 pi radians.
  double phaseIncrement = frequency * twoPi / samplerate;

  // The maximum amplitude of the signal, should not exceed 1.
  double maxAmplitude = 0.8;

  // The buffer in which the samples will be stored.
  double * buffer = new double[numberOfSamples];

  // For every sample.
  for (unsigned long n = 0; n < numberOfSamples; ++n)
  {
      // Calculate the sample.
      buffer[n] = maxAmplitude * sin(phase);

      // Increment the phase by the appropriate
      // amount of radians.
      phase += phaseIncrement;

      // Check if two pi have been reached and
      // reset if so.
      if (phase >= twoPi)
      {
          phase -= twoPi;
      }
  }

  // Further processing ...

  // Free the buffer memory.
  delete [] buffer;
}
```

Listing A.1: C++ implementation of a complete sine wave generator.

```
template <class PartItr>
double* additive(PartItr start,
                 PartItr end,
                 unsigned long length,
                 double masterAmp = 1,
                 bool sigmaAprox = false,
                 unsigned int bitWidth = 16)
{
    static const double pi = 3.141592653589793;

    static const double twoPi = 6.28318530717958;

    // calculate number of partials
    unsigned long partNum = end - start;

    double * buffer = new double [length];

    double * amp = new double [partNum];         // the amplitudes
    double * phase = new double [partNum];       // the current phase
    double * phaseIncr = new double [partNum];   // the phase increments

    // constant sigma constant part
    double sigmaK = pi / partNum;

    // variable part
    double sigmaV;

    // convert the bit number to decimal
    bitWidth = pow(2, bitWidth);

    // the fundamental increment of one period
    // in radians
    static double fundIncr = twoPi / length;

    // fill the arrays with the respective partial values
    for (unsigned long p = 0; start != end; ++p, ++start)
    {
        // initial phase
        phase[p] = start->phaseOffs;

        // fundIncr is two ÏĂ / tablelength
        phaseIncr[p] = fundIncr * start->num;

        // reduce amplitude if necessary
        amp[p] = start->amp * masterAmp;

        // apply sigma approximation conditionally
        if (sigmaAprox)
        {
            // following the formula
            sigmaV = sigmaK * start->num;

            amp[p] *= sin(sigmaV) / sigmaV;
        }
    }

    // fill the wavetable
    for (unsigned int n = 0; n < length; n++)
    {
        double value = 0;

        // do additive magic
        for (unsigned short p = 0; p < partNum; p++)
        {
            value += sin(phase[p]) * amp[p];

            phase[p] += phaseIncr[p];

            if (phase[p] >= twoPi)
            { phase[p] -= twoPi; }

            // round if necessary
```

```
73              if (bitWidth < 65536)
74              {
75                  Util::round(value, bitWidth);
76              }
77
78              buffer[n] = value;
79          }
80      }
81
82      delete [] phase;
83      delete [] phaseIncr;
84      delete [] amp;
85
86      return buffer;
87  }
```

Listing A.2: C++ program to produce one period of an additively synthesized complex waveform, given a start and end iterator to a container of partials, a buffer length, a maximum, "master", amplitude, a boolean whether or not to apply sigma approximation and lastly a maximum bit width parameter.

```
#include <cmath>
#include <stdexcept>

class EnvSeg
{

public:

    typedef unsigned long len_t;

    EnvSeg(double startLevel = 0,
           double endLevel = 0,
           len_t len = 0,
           double rate = 1)

    : startLevel_(startLevel), endLevel_(endLevel),
    rate_(rate), curr_(0),len_(len)

    {
        calcRange_();
        calcIncr_();
    }

    double tick()
    {
        // If the segment is still supposed to
        // tick after reaching the end amplitude
        // just return the end amplitude

        if (curr_ >= 1 || ! len_) return endLevel_;

        return range_ * pow(curr_,rate_) + startLevel_;
    }

    void update()
    {
        // Increment curr_
        curr_ += incr_;
    }

    void setLen(len_t sampleLen)
    {
        len_ = sampleLen;

        calcIncr_();
    }

    len_t getLen() const
    {
        return len_;
    }

    void setRate(double rate)
    {
        if (rate > 10 || rate < 0)
        { throw std::invalid_argument("Rate must be between 0 and 10"); }

        rate_ = rate;
    }

    double getRate() const
    {
        return rate_;
    }

    void setEndLevel(double lv)
    {
        if (lv > 1 || lv < 0)
        { throw std::invalid_argument("Level must be between 0 and 1"); }

        endLevel_ = lv;

```

```
73          calcRange_();
74      }
75
76      double getEndLevel() const
77      {
78          return endLevel_;
79      }
80
81      void setStartLevel(double lv)
82      {
83          if (lv > 1 || lv < 0)
84          { throw std::invalid_argument("Level must be between 0 and 1"); }
85
86          startLevel_ = lv;
87
88          calcRange_();
89      }
90
91      double getStartLevel() const
92      {
93          return startLevel_;
94      }
95
96      void reset()
97      {
98          curr_ = 0;
99      }
100
101 private:
102
103     /*! Calculates the amplitude range and assigns it to range_ */
104     void calcRange_()
105     {
106         // The range between start and end
107         range_ = endLevel_ - startLevel_;
108     }
109
110     /*! Calculates the increment for curr_ and assigns it to incr_ */
111     void calcIncr_()
112     {
113         incr_ = (len_) ? 1.0/len_ : 0;
114     }
115
116     /*! The rate determining the type (lin,log,exp) */
117     double rate_;
118
119     /*! Starting amplitude */
120     double startLevel_;
121
122     /*! End amplitude */
123     double endLevel_;
124
125     /*! Difference between end and start amplitude */
126     double range_;
127
128     /*! Current segment value */
129     double curr_;
130
131     /*! Increment value for curr_ */
132     double incr_;
133
134     /*! Length of segment in samples */
135     len_t len_;
136 };
```

Listing A.3: C++ implementation of single Envelope segments.

```
1  private:
2
3      struct ModItem;
4
5      typedef std::vector<ModItem> modVec;
6
7      // Not using iterators because they're invalidated when
8      // pushing back/erasing from modItems_ and not using
9      // pointers to ModItems because then it's difficult
10     // to interact with modItems_
11     typedef std::vector<modVec::size_type> indexVec;
12
13     typedef indexVec::iterator indexVecItr;
14
15     typedef indexVec::const_iterator indexVecItr_const;
16
17     /*! A ModItem contains a ModUnit*, its depth value as well as */
18     struct ModItem
19     {
20         ModItem(ModUnit* modUnit, double dpth = 1)
21         : mod(modUnit), depth(dpth), baseDepth(dpth)
22         { }
23
24         /*! The actual ModUnit pointer. */
25         ModUnit* mod;
26
27         /*! The current modulation depth. */
28         double depth;
29
30         /*! For sidechaining */
31         double baseDepth;
32
33         /*! Vector of indices of all masters in modItems_*/
34         indexVec masters;
35
36         /* Vector of indices of slaves of this ModItem in modItems_ */
37         indexVec slaves;
38     };
39
40     /*! Indices of all ModItems that are masters for sidechaining
41      and thus don't contribute to the ModDocks modulation value */
42     indexVec masterItems_;
43
44     /*! Pointer to all ModItems excluding sidechaining masters */
45     indexVec nonMasterItems_;
46
47     /*! All ModItems */
48     modVec modItems_;
49
50     /*! This is the base value that the modulation happens around */
51     double baseValue_;
52
53     /*! Lower boundary value to scale to when modulation trespasses it */
54     double lowerBoundary_;
55
56     /*! Higher boundary value to scale to when modulation trespasses it */
57     double higherBoundary_;
```

Listing A.4: Private members of the `ModDock` class.

```
double ModDock::modulate(double sample)
{
    // If ModDock is not in use, return original sample immediately
    if (! inUse()) return sample;

    // Sidechaining

    // For every non-master
    for (indexVecItr nonMasterItr = nonMasterItems_.begin(), nonMasterEnd =
    nonMasterItems_.end();
         nonMasterItr != nonMasterEnd;
         ++nonMasterItr)
    {
        // If it isn't a slave, nothing to do
        if (! isSlave(*nonMasterItr)) continue;

        ModItem& slave = modItems_[*nonMasterItr];

        // Set to zero initially
        slave.depth = 0;

        // Then sum up the depth from all masters
        for (indexVecItr_const masterItr = slave.masters.begin(), masterEnd =
    slave.masters.end();
             masterItr != masterEnd;
             ++masterItr)
        {
            ModItem& master = modItems_[*masterItr];

            // Using the baseDepth as the base value and the master's depth as
            // the depth for modulation and 1 as the maximum boundary
            slave.depth += master.mod->modulate(slave.baseDepth, master.depth, 1);
        }

        // Average the depth
        slave.depth /= slave.masters.size();
    }

    // Modulation

    double temp = 0;

    // Get modulation from all non-master items
    for(indexVecItr_const itr = nonMasterItems_.begin(), end = nonMasterItems_.end();
        itr != end;
        ++itr)
    {
        // Add to result so we can average later
        // Use the sample as base, the modItem's depth as depth and the
        // higherBoundary as maximum
        temp += modItems_[*itr].mod->modulate(sample,
                                              modItems_[*itr].depth,
                                              higherBoundary_);
    }

    // Average
    sample = temp / nonMasterItems_.size();

    // Boundary checking
    if (sample > higherBoundary_) { sample = higherBoundary_; }

    else if (sample < lowerBoundary_) { sample = lowerBoundary_; }

    return sample;
}
```

Listing A.5: Definition of the `modulate` method in the `ModDock` class.

Bibliography

[1] Daniel R. Mitchell,
BasicSynth: Creating a Music Synthesizer in Software.
Publisher: Author.
1st Edition,
2008.

[2] Steven W. Smith,
The Scientist and Engineer's Guide to Digital Signal Processing.
California Technical Publishing,
San Diego, California,
2nd Edition,
1999.

[3] Ed: Catherine Soanes and Angus Stevenson
Oxford Dictionary of English
Oxford University Press,
Oxford,
2003.

[4] Phil Burk, Larry Polansky, Douglas Repetto, Mary Roberts and Dan Rockmore
Music and Computers: A Historical and Theoretical Appraoch.
2011
http://music.columbia.edu/cmc/MusicAndComputers/
Accessed: 22 December 2014.

[5] Gordon Reid,
Synth Secrets, Part 12: An Introduction To Frequency Modulation.
http://www.soundonsound.com/sos/apr00/articles/synthsecrets.htm
Accessed: 8 October 2014.

[6] Justin Colletti,
The Science of Sample Rates (When Higher Is Better âĂŤ– And When It Isn't)

2013.
http://www.trustmeimascientist.com/2013/02/04/
the-science-of-sample-rates-when-higher-is-better-and-when-it-isnt/
Accessed: 17 December 2014.

[7] John D. Cutnell and Kenneth W. Johnson,
Physics.
Wiley,
New York,
4th Edition,
1998.

List of Figures