Developing a Digital Synthesizer in C++

Peter Goldsborough
petergoldsborough@hotmail.com

December 23, 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.
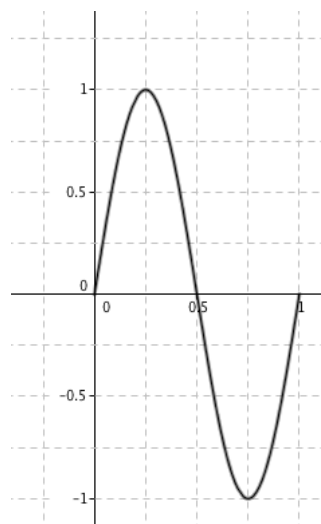
Figure 1.1: The continuous representation of a typical sine wave. In this case, both the signal's frequency f as well as the maximum elongation from the equilibrium a are equal to 1.

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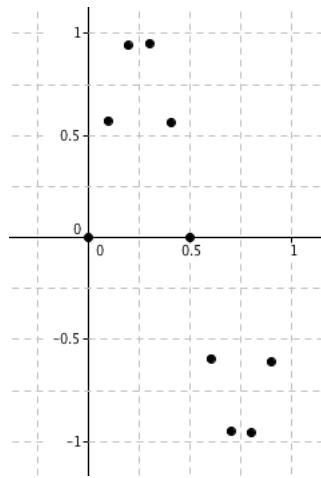
Figure 1.2: The discrete representation of a typical sine wave.

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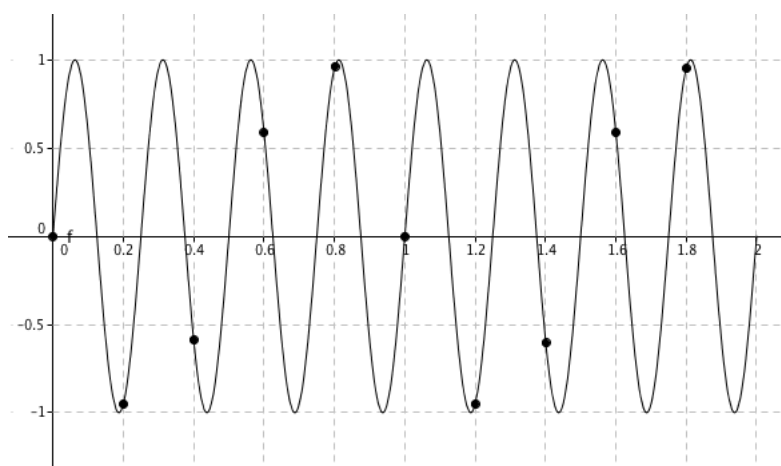
$$32767_{10} = 0111111111111111_2$$

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

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

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.
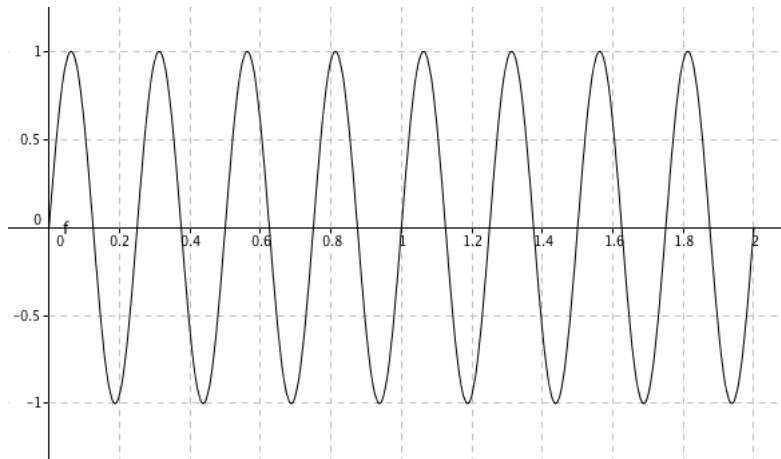

**Figure 1.3:** A sinusoid with a frequency of 4 Hz.

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.
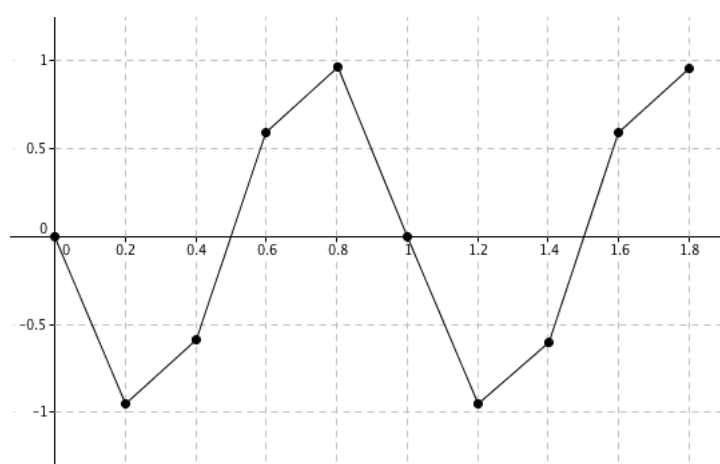
Figure 1.5: An approximated representation of the signal created from the sampling process shown in Figure 1.4. Visually as well as acoustically, the new signal is completely different from the original signal. However, in terms of sampling, they are indistinguishable from each other due to improper sampling. These signals are said to be *aliases* of each other.
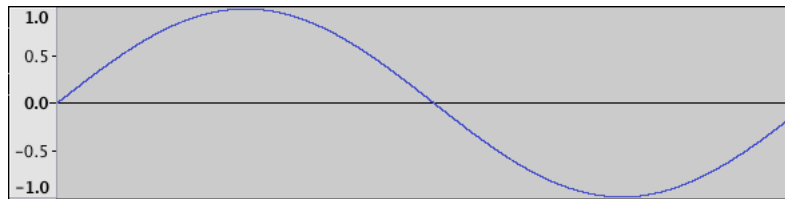
Figure 1.6: A typical sinusoidal signal with an amplitude of 1. The integer range provided by the allocated memory for the top-most sample is saturated, meaning it is equal to 32767_{10} or 0111111111111111_2.
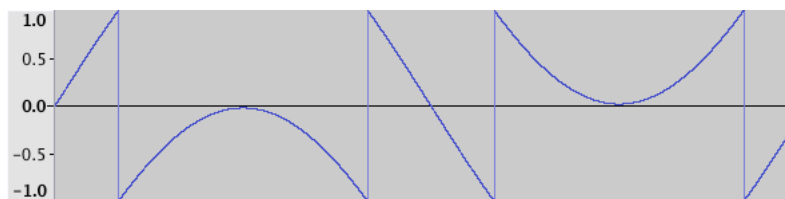
Figure 1.7: What happens when the amplitude of the signal from Figure 1.6 is increased by a factor of 2. Multiple samples have overflowed and thus changed their sign. Because of the way two's-complement representation is implemented, the signal continues it's path as if no overflow had ever occurred. 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:

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

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 $\dfrac{\pi}{2}$ radians:

$$f_c(t) = A\cos(2\pi f t + \phi) = A\sin(2\pi f t + \phi + 90) \tag{2.2}$$

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.

$$f(t) = \begin{cases} 1, & \text{if } 0 \leq t < \dfrac{T}{2} \\ -1, & \text{if } \dfrac{T}{2} \leq t < T \end{cases} \quad (2.3) \qquad f(t) = \begin{cases} 1, & \text{if } \sin(2\pi f t) > 0 \\ -1, & \text{otherwise} \end{cases} \quad (2.4)$$

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.

$$f(\phi) = \begin{cases} -\dfrac{\phi}{\pi} + 1, & \text{if } 0 \leq \phi < 2\pi \end{cases} \quad (2.5) \qquad f(t) = \begin{cases} -\dfrac{2t}{T} + 1, & \text{if } 0 \leq t < 1 \end{cases} \quad (2.6)$$

```cpp
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.

```cpp
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.

$$f(\phi) = \begin{cases} \dfrac{\phi}{\pi} - 1, & \text{if } 0 \leq \phi < 2\pi \end{cases} \quad (2.7) \qquad f(t) = \begin{cases} \dfrac{2t}{T} - 1, & \text{if } 0 \leq t < 1 \end{cases} \quad (2.8)$$

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

    // index incrementor
    double incr = 2.0 / period;

    double value = -1;

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

        value += incr;
    }

    return buffer;
}
```

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.

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

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

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

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

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

```cpp
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.

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

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 $2\pi f$.

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

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.

```
1    struct Partial
2    {
3      Partial(unsigned short number, double ampl, double phsOffs = 0)
4        : num(number), amp(ampl), phaseOffs(phsOffs)
5      { }
6
7      /*! The Partial's number, stays const. */
8      const unsigned short num;
9
10     /*! The amplitude value. */
11     double amp;
12
13     /*! A phase offset */
14     double phaseOffs;
15   };
```

Table 2.5: C++ code to represent a single partial in a Fourier Series.
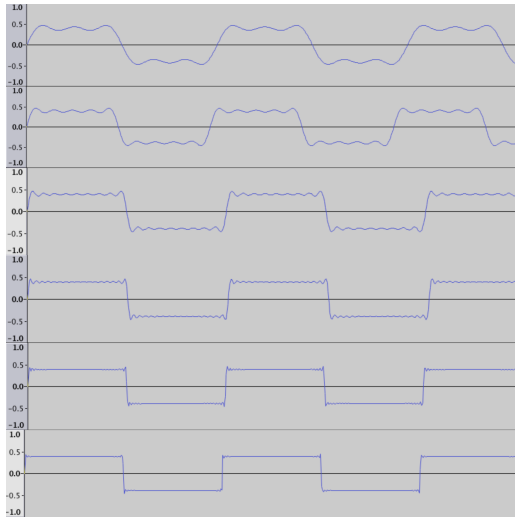
Figure 2.1: Square waves with 2, 4, 8, 16, 32 and 64 partials.

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

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

$$f(t) = \sum_{n=1}^{N} \frac{1}{n} \sin(\omega n t) \tag{2.14}$$
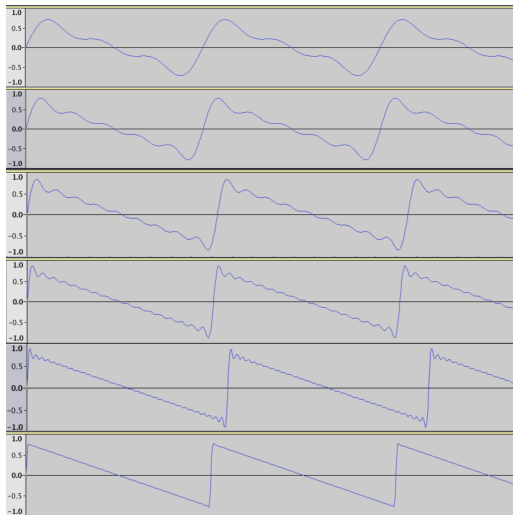
Figure 2.2: Sawtooth waves with 2, 4, 8, 16, 32 and 64 partials.

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)
```
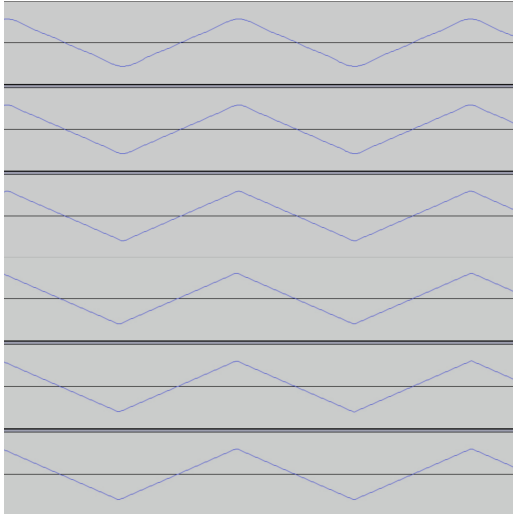
Ramp Waves

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.

$$f(t) = \sum_{n=1}^{N} -\frac{1}{n} \sin(\omega n t) \tag{2.15}$$
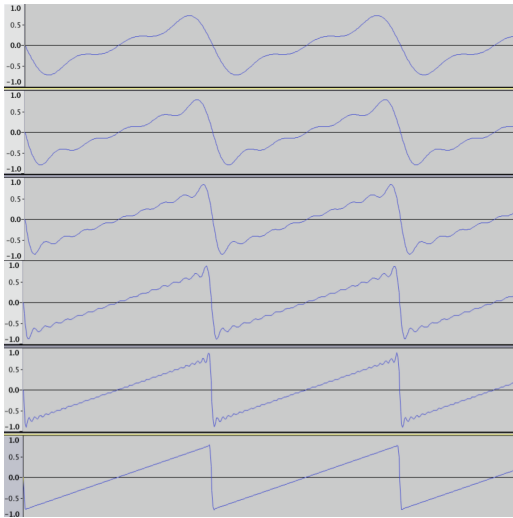
Figure 2.3: Ramp waves with 2, 4, 8, 16, 32 and 64 partials.

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.

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

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

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

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

double amp = -1;

for(int i = 1; i <= 128; i += 2)
{
  amp = (amp > 0) ? (-1.0/(i*i)) : (1.0/(i*i));

  vec.push_back(Partial(i,amp));
}

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

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.

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.
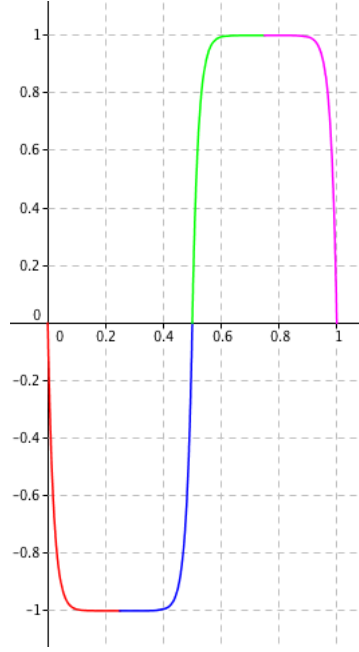
Smooth Square Waves

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.

$$a(t) = \left\{ (t-1)^{50} - 1, \quad \text{if } 0 \leq t < \frac{T}{4} \right. \tag{2.18}$$

$$b(t) = \left\{ (t+0.5)^{50} - 1, \quad \text{if } \frac{T}{4} \leq t < \frac{T}{2} \right. \tag{2.19}$$

$$c(t) = \left\{ -(t-1.5)^{50} + 1, \quad \text{if } \frac{T}{2} \leq t < \frac{3T}{4} \right. \tag{2.20}$$

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

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

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.

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

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

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

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.
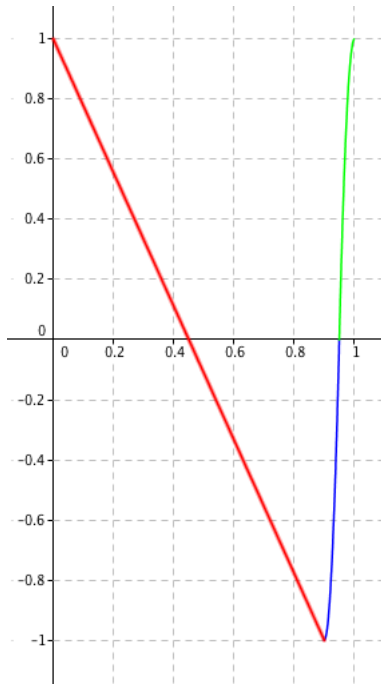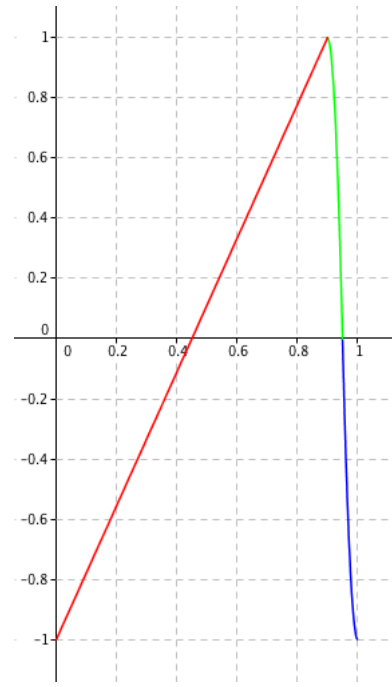
Figure 2.6: A smooth sawtooth wave.

Figure 2.7: A smooth ramp wave.

$$a(t) = \left\{ \frac{2t}{0.9T} - 1 = \frac{-20t}{9T} + 1, \quad \text{if } 0 \le t < \frac{9T}{10} \right. \tag{2.25}$$

$$b(t) = \left\{ -400(t - 0.9)^2 + 1, \quad \text{if } \frac{9T}{10} \le t < \frac{95T}{100} \right. \tag{2.26}$$

$$c(t) = \left\{ 400(t - 1)^2 - 1, \quad \text{if } \frac{95T}{100} \le t < T \right. \tag{2.27}$$

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.
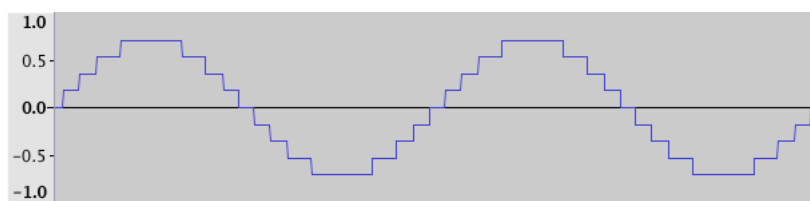

Figure 2.8: A 3-bit sine wave.
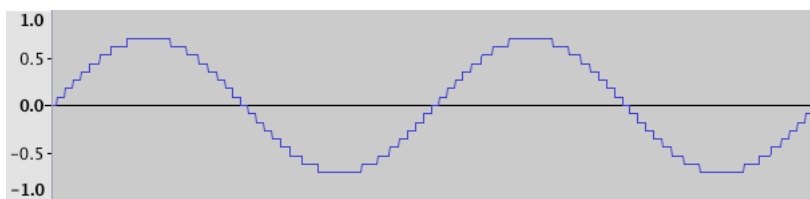

Figure 2.9: A 4-bit sine wave.

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.

$$\sigma = sinc(\frac{n\pi}{M}) = \frac{\sin(\frac{n\pi}{M})}{\frac{n\pi}{M}} \tag{2.28}$$
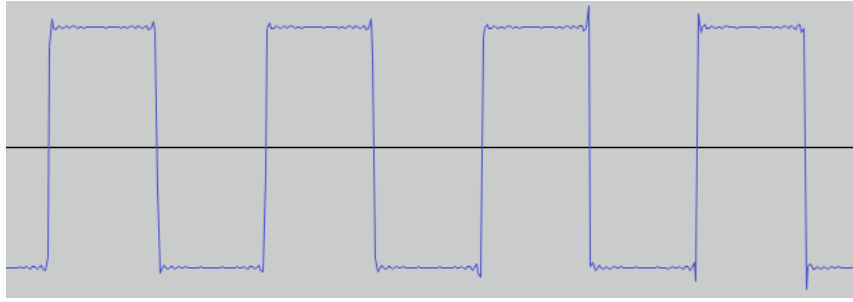
Figure 2.10:   An additively synthesized square wave with 64 partials before sigma-approximation.
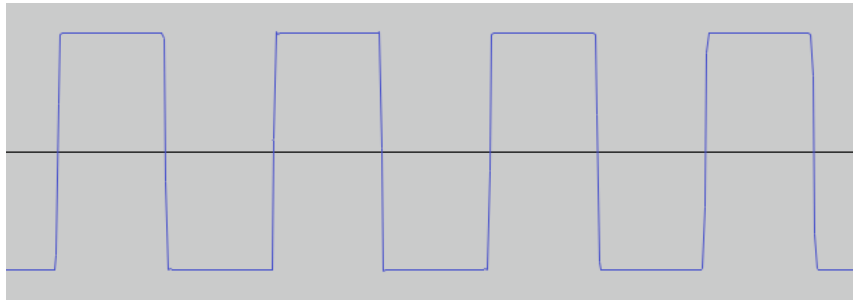
Figure 2.11:   An additively synthesized square wave with 64 partials after sigma-approximation.

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 wave-form 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 propor-tionate 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.

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

Table 2.11: An interplation algorithm in pseudo-code.

```
1   template<class T>
2   double interpolate(T table [], double index)
3   {
4       long integral = static_cast<long>(index);  // The truncated integral part
5       double fractional = index - integral;    // The remaining fractional part
6
7       // grab the two items in-between which the actual value lies
8       T value1 = table[integral];
9       T value2 = table[integral+1];
10
11      // interpolate: integer part + (fractional part * difference between value2 and value1)
12      double final = value1 + ((value2 - value1) * fractional);
13
14      return final;
15  }
```

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{†Áå#Y?ö`)˜ä#i?'œ Ã•/r?·z05É#y?÷?˪\l?À°≠µò/Ç?ìµ7$ˇÖ?±@ˆ−d#â?õÆÕVƒGå?˜£" lè?‰¯≠ä;Hë?
BtoYd/í?®ïâ?älî?áq•˛¨¸ï?ÀòòYÃêó?g"wË"ô?Î¥ÅËˇ¥ö?ê&†Gú?cñÙ°"Ÿù?®VKº−kü?Œä'−ô~†?µk:öG°?¶4µ˜ó
¢?ã5ﬂÍíŸ¢?b,Ç˜ä¢£?
```

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

```cpp
 1  #include <fstream>
 2  #include <stdexcept>
 3
 4  double* readWavetable(const std::string &fname)
 5  {
 6    std::ifstream file(fname);
 7
 8    if (! file.is_open())
 9    { throw std::runtime_error("Could not find wavetable file: " + fname); }
10
11    if (! file.good())
12    { throw std::runtime_error("Error opening wavetable: " + fname); }
13
14    char signature[6];
15
16    file.read(signature, 6);
17
18    if (strncmp(signature, "ANTHEM", 6))
19    { throw std::runtime_error("Invalid signature for Anthem file!"); }
20
21    int len = 4096;
22    int size = len * sizeof(double);
23
24    double * wt = new double [len];
25
26    file.read(reinterpret_cast<char*>(wt), size);
27
28    return wt;
29  }
```

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.
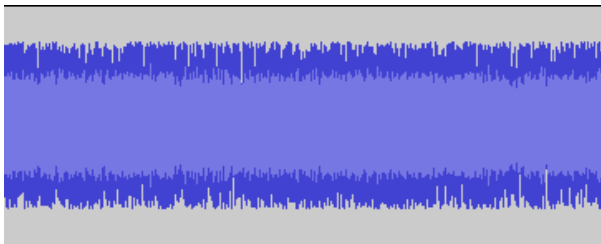
Table 2.14: A typical white noise signal.
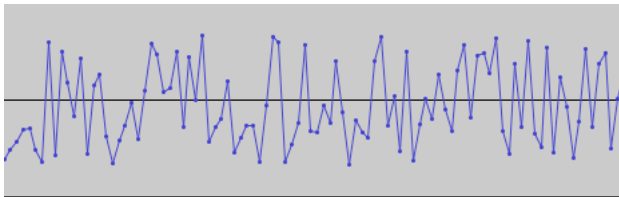
Table 2.15: A close-up view of Figure 2.14. This Figure shows nicely how individual sample values are completely random and independent from each other.
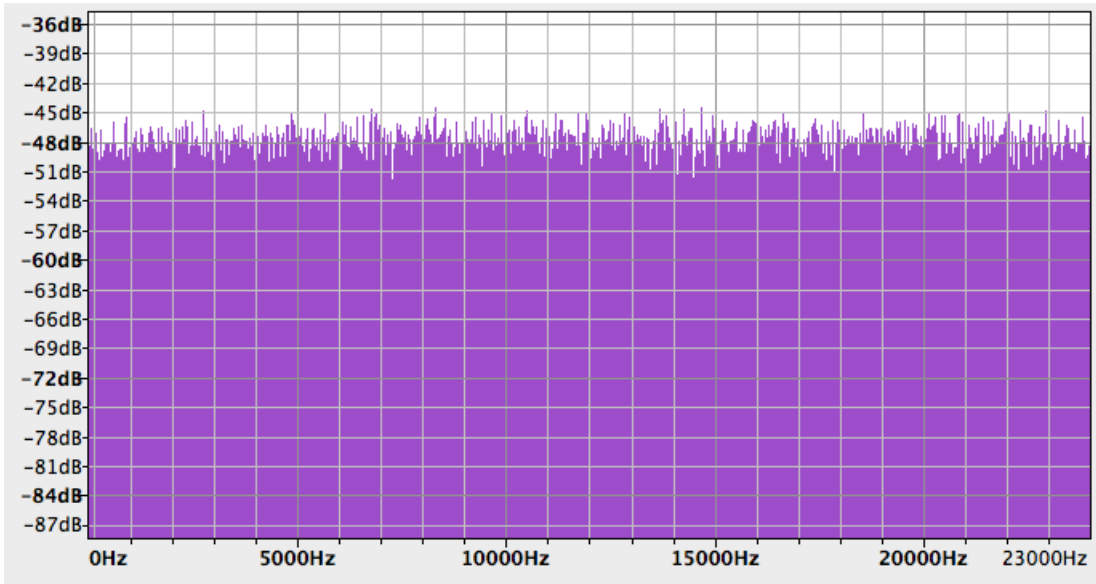
Figure 2.14: The signal from Figures 2.14 and 2.15 in the frequency domain. This frequency spectrum analysis proves the fact that white noise has a "flat" frequency spectrum, meaning that all frequencies are distributed uniformly and at (approximately) equal intensity.

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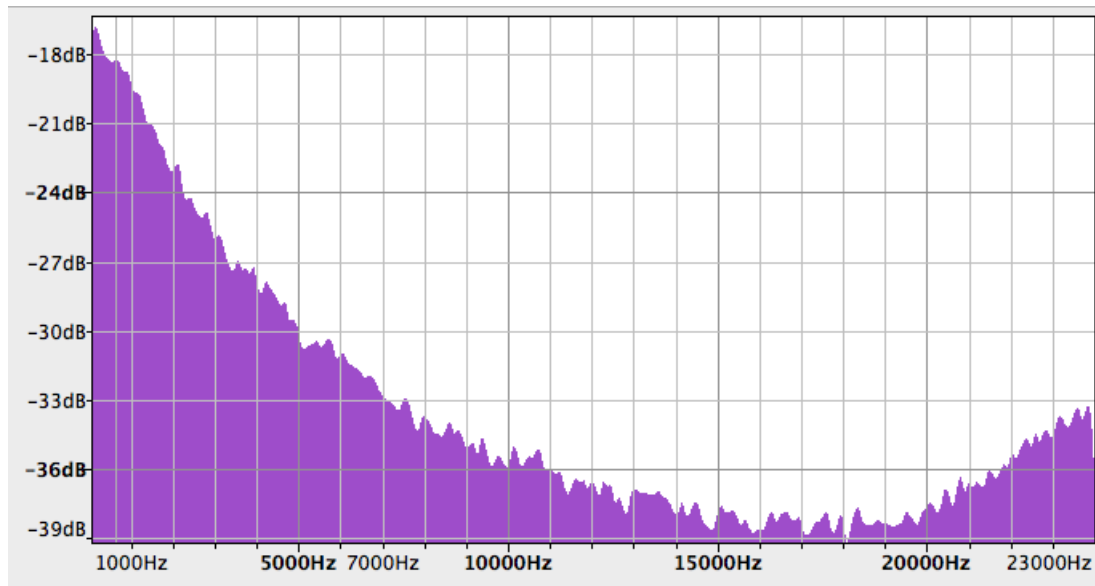
Figure 2.15: The frequency spectrum of pink noise. Note that the filter with which white noise was transformed into pink noise, discussed in a later chapter, is not perfect, resulting in the increase in frequency intensity towards the end of the spectrum. Ideally, the intensity should decrease linearly.
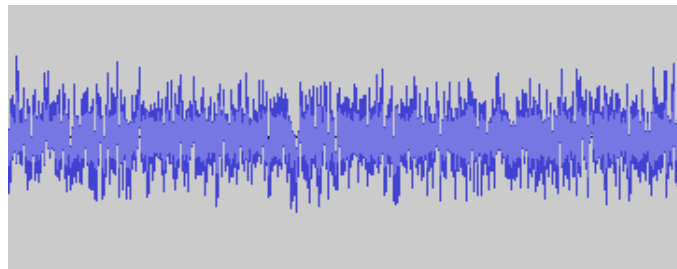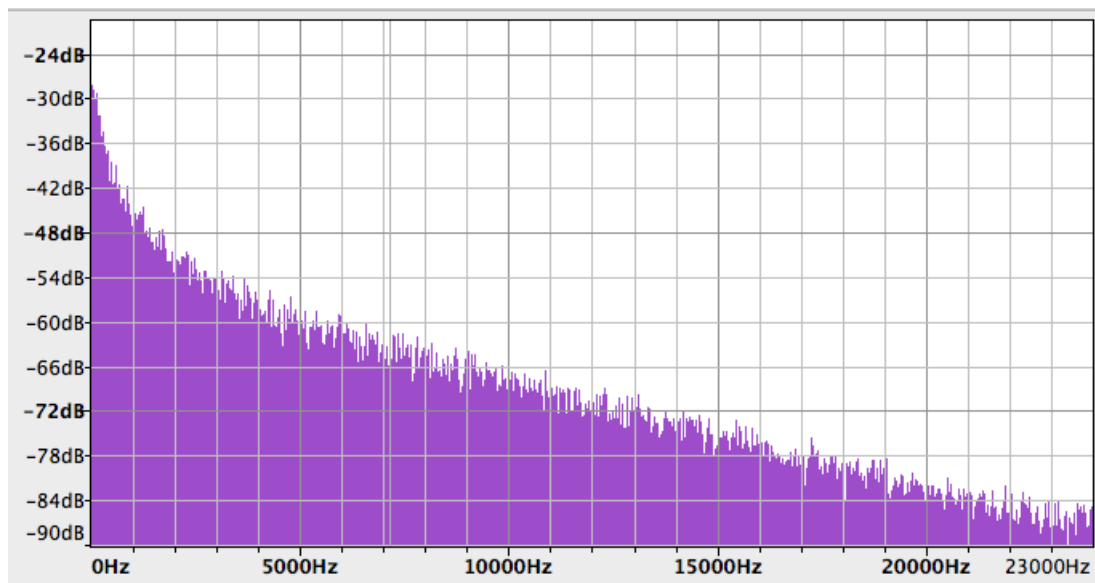

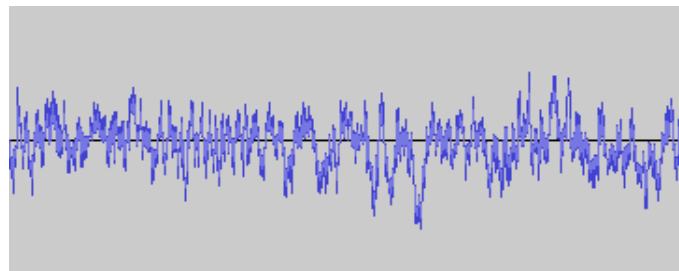**Figure 2.16:** Pink noise in the time domain.

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

Blue Noise

Whereas 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
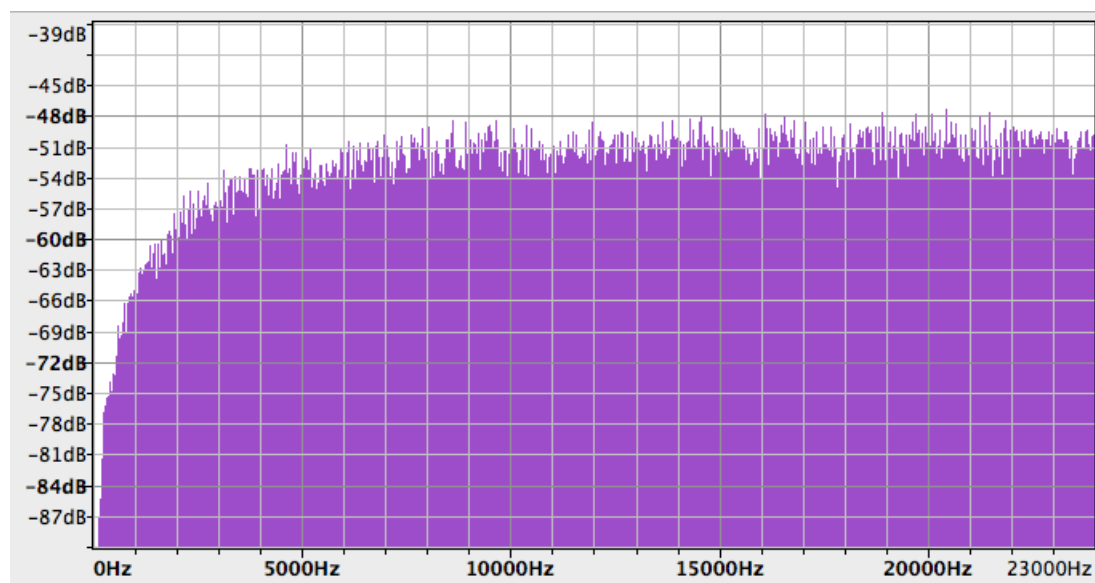
Figure 2.19: Blue noise in the frequency domain.
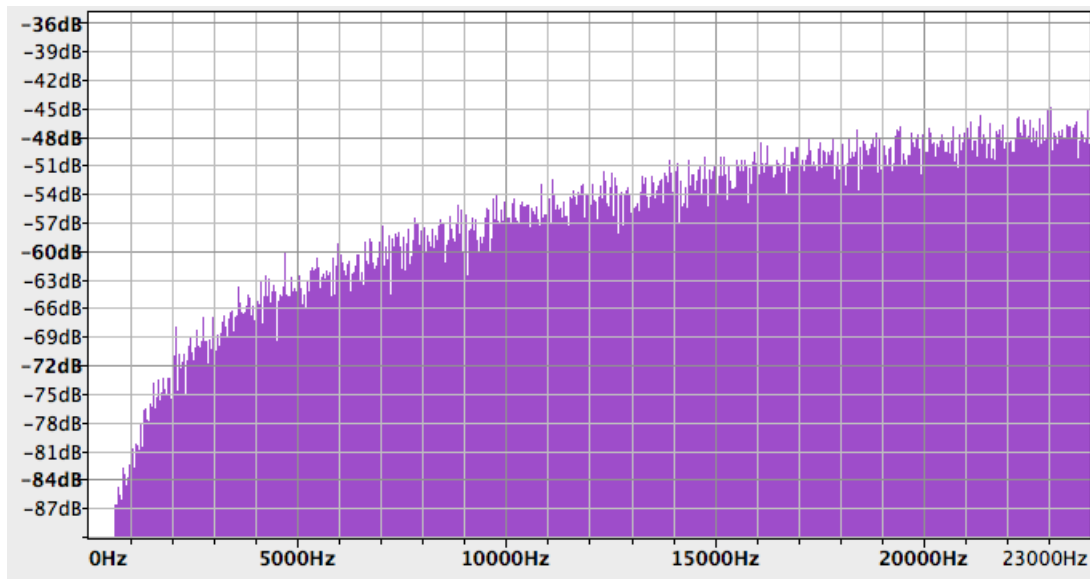
Figure 2.20: The frequency spectrum of a violet noise signal.

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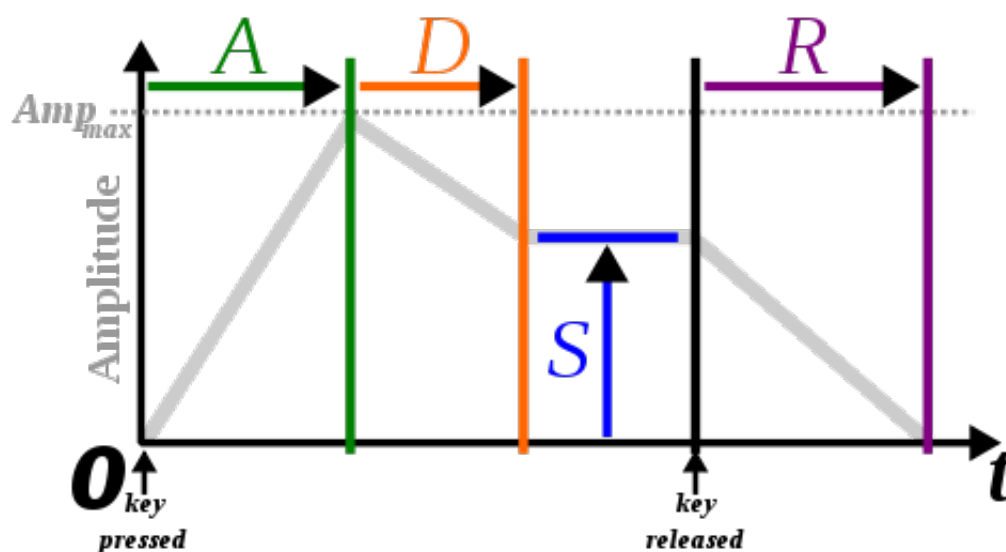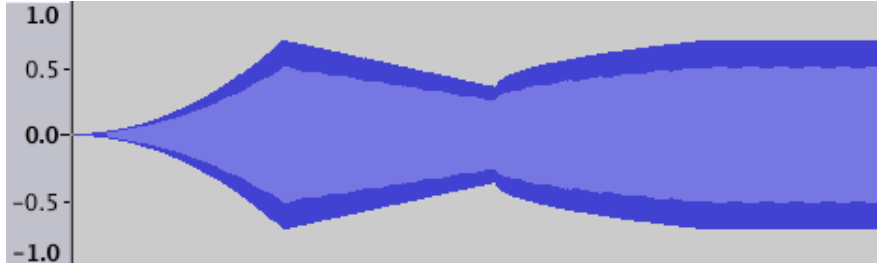
Figure 3.1: An Envelope with an Attack, a Decay, a Sustain and finally a Release segment. Source: `http://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/ ADSR_parameter.svg/500px-ADSR_parameter.svg.png`

Figure 3.2: A 440 Hz sine wave.
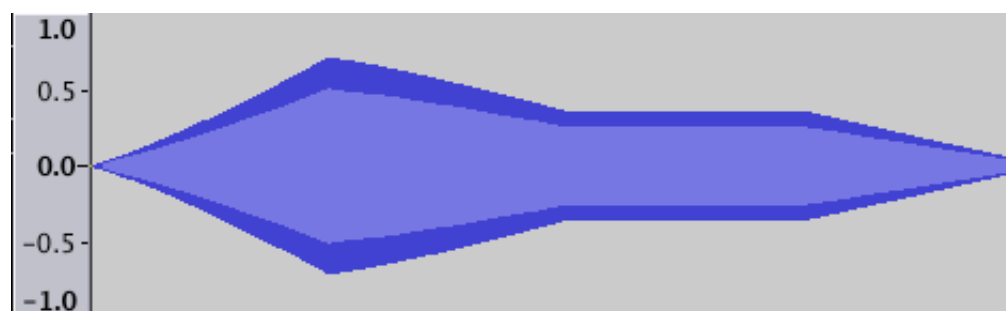
Figure 3.3: The same signal from Figure 3.2 with an ADSR Envelope overlayed on it.

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.

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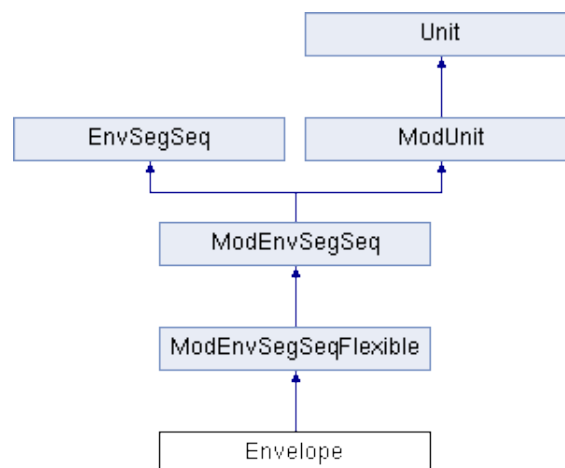
Figure 3.5: The inheritance diagram for the `Envelope` class. The `Unit` and `ModUnit` classes are two abstract classes that will be explained in later parts of this thesis.

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.

```cpp
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.

```cpp
#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.

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