Developing a Digital Synthesizer in C++

Peter Goldsborough
petergoldsborough@hotmail.com

December 20, 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.

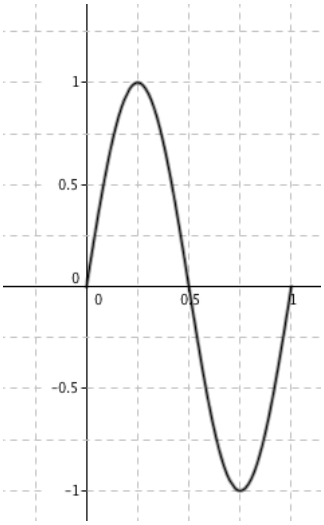

Figure 1.1: The continuous representation of a typical sine wave. In this case, both the signal's frequency f as well as the maximum elongation from the equilibrium a are equal to 1.

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). 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.

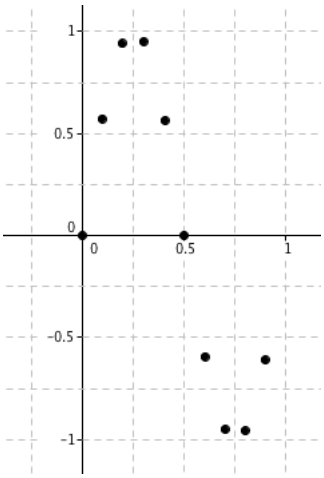

Figure 1.2: The discrete representation of a typical sine wave.

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).

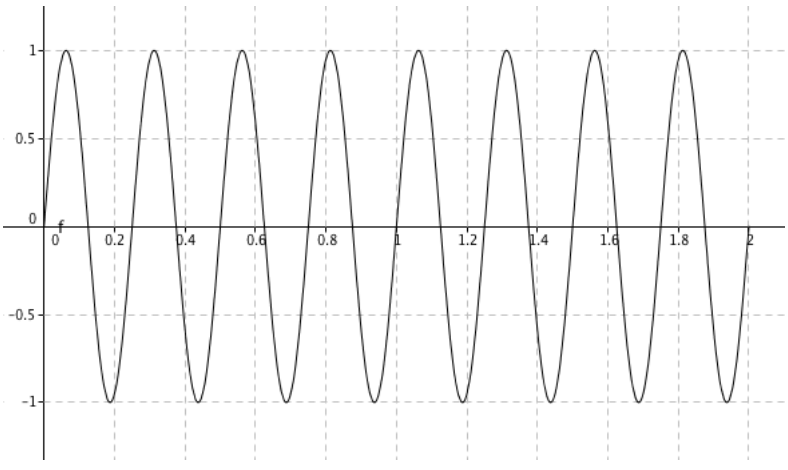

Figure 1.3: A sinusoid with a frequency of 4 Hz.

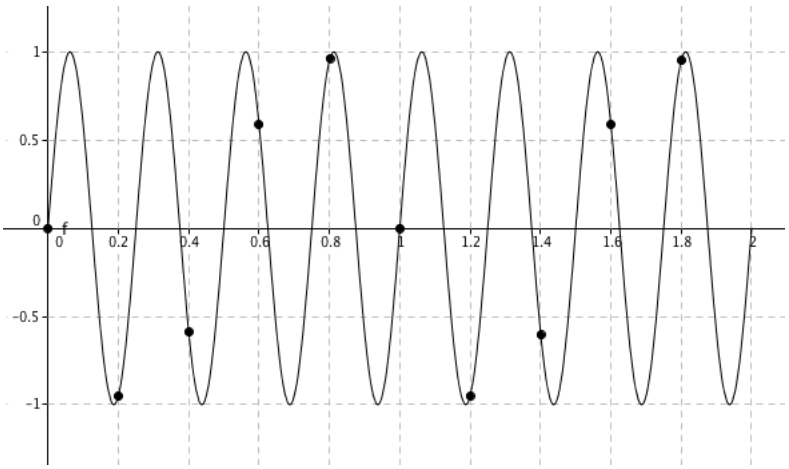

Figure 1.4: A sinusoid with a frequency of 4 Hz, sampled at a sample rate of 5 Hz. According to the Nyquist Theorem, this signal is sampled improperly, as the frequency of the continuous signal is not less than or equal to one half of the sample rate, in this case 2.5 Hz.

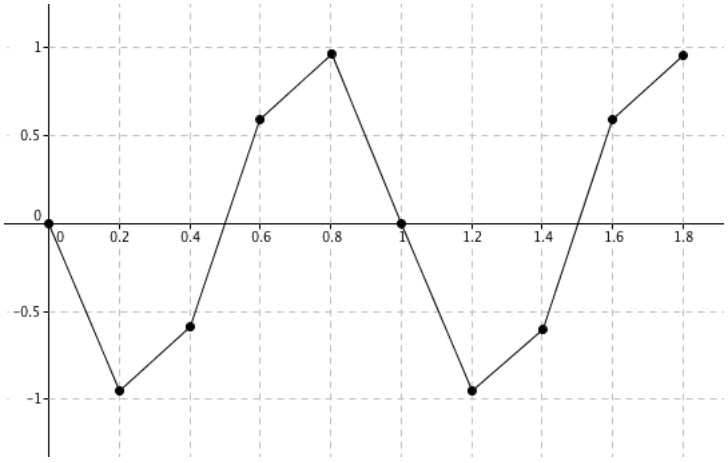

Figure 1.5: An approximated representation of the signal created from the sampling process shown in Figure 1.4. Visually as well as acoustically, the new signal is completely different from the original signal. However, in terms of sampling, they are indistinguishable from each other due to improper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

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.

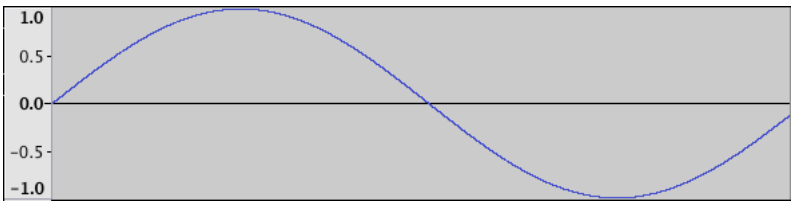

Figure 1.6: A typical sinusoidal signal with an amplitude of 1. The integer range provided by the allocated memory for the top-most sample is saturated, meaning it is equal to 32767_{10} or 0111111111111111_2.

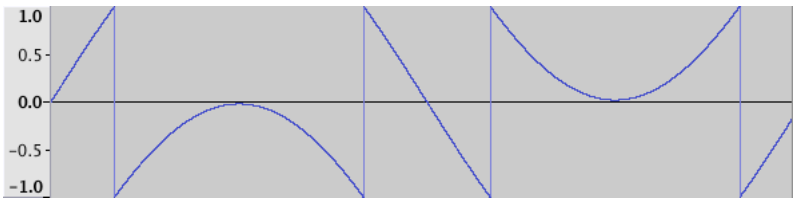

Figure 1.7: What happens when the amplitude of the signal from Figure 1.6 is increased by a factor of 2. Multiple samples have overflowed and thus changed their sign. Because of the way two's-complement representation is implemented, the signal continues it's path as if no overflow had ever occurred.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

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

Therefore, the program from Table **??** 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.

```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;
}
```

Table 2.1: C++ implementation of a complete sine wave generator.

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 2.2.

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

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

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 2.3.

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

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

```
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 2.2: 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 2.3: 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 2.4 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 2.4: C++ ramp wave generator. This code differs from the program shown in Table 2.3 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 2.5.

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

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

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

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

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

```
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 2.5: C++ program to compute one period of a triangle wave.

called a 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 2.6 presents a C++ struct to represent a single partial and Tables 2.7 and 2.8 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 2.6: C++ code to represent a single partial in a Fourier Series.

```
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;
        }
    }
```

Table 2.7

```
56 // fill the wavetable
57 for (unsigned int n = 0; n < length; n++)
58 {
59     double value = 0;
60
61     // do additive magic
62     for (unsigned short p = 0; p < partNum; p++)
63     {
64         value += sin(phase[p]) * amp[p];
65
66         phase[p] += phaseIncr[p];
67
68         if (phase[p] >= twoPi)
69         { phase[p] -= twoPi; }
70
71     // round if necessary
72     if (bitWidth < 65536)
73     {
74         Util::round(value, bitWidth);
75     }
76
77     buffer[n] = value;
78 }
79
80 delete [] phase;
81 delete [] phaseIncr;
82 delete [] amp;
83
84 return buffer;
85 }
```

Table 2.8: *Continuation of Table 2.7.* 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.

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 or Fourier 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 do not produce significant improvements in sound quality. Table 2.9 displays the C++ code needed to produce one period of a square wave in conjuction with the `additive` function from Tables 2.7 and 2.8. Figure 2.1 shows the result of summing 2, 4, 8, 16, 32 and finally 64 partials.

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

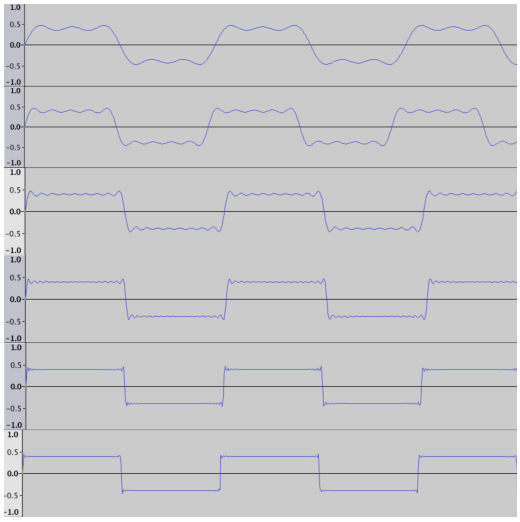

Figure 2.1

Table 2.9

```
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)
```

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 2.10 shows C++ code to generate such functions.

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

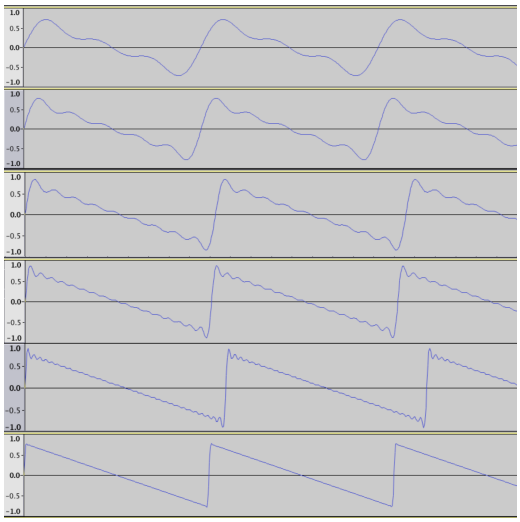

Figure 2.2

Table 2.10

```
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 2.11 again shows the accompanying C++ code.

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

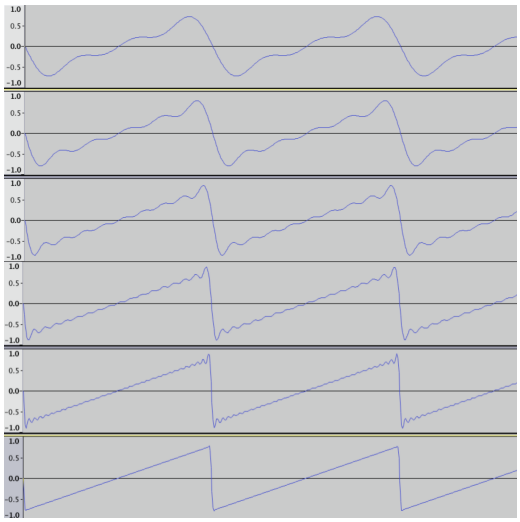

Figure 2.3

Table 2.11

```
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 of a triangle waveform 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 2.12 implements C++ code to compute a triangle wave.

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

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

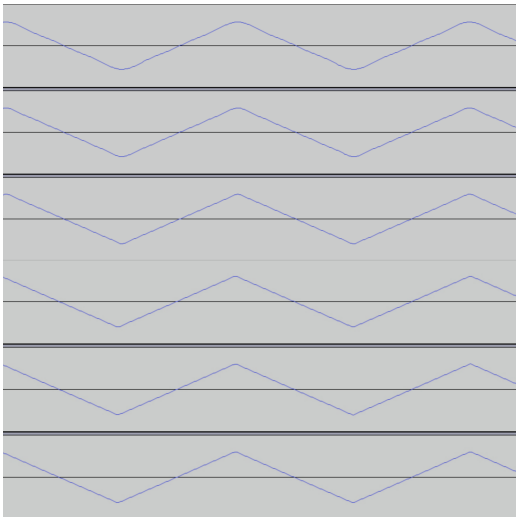

Figure 2.4

Table 2.12

```
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);
```

2.3 Real-time calculation versus Wavetable lookup

Before the generation of more complex waveforms than the sine and cosine waves from section 2.1 can be discussed, 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.

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

[4] 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.

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

List of Figures