TITLE

Alexander Liebendörfer

December 1, 2013

Contents

1 Preface

1.1 Motivation

2 Introduction

2.1 Description

2.1.1 Synthesizers

3 Tools

The only physical tools used were a computer with peripherals, and a sound system. Several software tools were used though, and I will briefly discuss the most important ones here.

The software itself is programmed in *Java* (by Oracle), and hence is dependent of the *Java runtime environment* (JRE) and all it's requirements. As interface between my software and the OS sound system I used the standard *Java Sound API*, which is part of the javax module package. The GUI itself was made through a popular GUI library called *Swing*, which is also used in many professional applications.

On top of the tools directly implicated in the making of the software, a few additional ones were used to facilitate it. A rather famous IDE named *Eclipse* was used for all Java programming, and of enormous importance was the version control system *Git* (with a few extra features from the webservice *Github*). *Python* and *Veusz* were slightly used for debugging and graphing purposes, especially for displaying sound output as an image, but have no direct part in the software.

Finally, this text was created with *LaTeX* and the *TikZ* graphing package for all graphs. Certain parts were created with help of the geometry package *Geogebra*, but were always modified afterwards.

4 Implementation

4.1 Pipes and Modules

To achieve a completely modular synthesizer, we already need to give up a fixed circuit and have to use the module concept. On top of that, because of recursive loops, we need to actually let the data flow between the different modules, and not just sum the effects of the modules recursively. This means we need at least 1 buffer between every module. We also need a clear way to connect those modules, so a pipe system for connecting inputs and outputs of different modules will be necessary. And finally, some way to inject the current MIDI data and duration of every note as well as a way to capture the processed sound for output is needed.

4.1.1 Basic system

In this specific implementation, modules are pure functions, or atleast behave as such towards the outside. A module is always connected to other modules via so-called pipes, which contain the actual data. Modules can then load the data from their input pipes, process it, and dump the result into their output pipes. The main advantage of this system is that it's stable and will work with any wiring setup, even feedback loops. In such closed feedback systems, there will be a propagation delay depending on the amount of pipes the data has to travel, and so no infinite recursion appears. Splitting data and doing parallel processing before merging it again is also implicitely supported, as every module and pipe only acts locally. One major issue is execution order. A recursive ”pull” query where every module calls the modules it depends on for their output cannot work, infinite recursion is a possibility. Even without recursion, there is always the risk that some part of the circuit did not get updated, which might affect the rest. I haven't found a clean solution to this without drawbacks, the one I chose just makes every module execute once in more or less random order (it is not truely random, in fact it is in order of creation), which in worse case causes a signal to wait an extra frame between two modules but guarantees that every module is executed once and only once per frame, and done so in a consistent way.

Injection of user input and retrieval of the sound output is best done with modules dedicated to that. These modules possess only 1 input or output port respectively, and do not by themselves do anything. They just serve as convienient place to connect pipes to, and the main overarching engine can simply read and write the contents of their pipes.

4.1.2 Dealing with Time

A tricky problem is dealing with time-sensitive modules, like envelopes or oscillators. The input does not implicitely declare it's duration, so there needs to be a way for the pipes to also carry starting time of a signal. For this, I implemented a separate counter in every pipe, which showed the time at which that signal started. When a module is done with processing and copies the signal data into its output, it also copies in the counter. A negative time can be used as a marker that the pipe is inactive and that module doesn't even need to process anything. Every module can define a special pipe which is taken as reference for the source time, which is especially important when conflicts are possible. If a module does not define such a pipe, the highest starting time (the most recent signal) of all input pipes is assumed to be the reference.

4.1.3 Stereo/Mono and Polyphony support

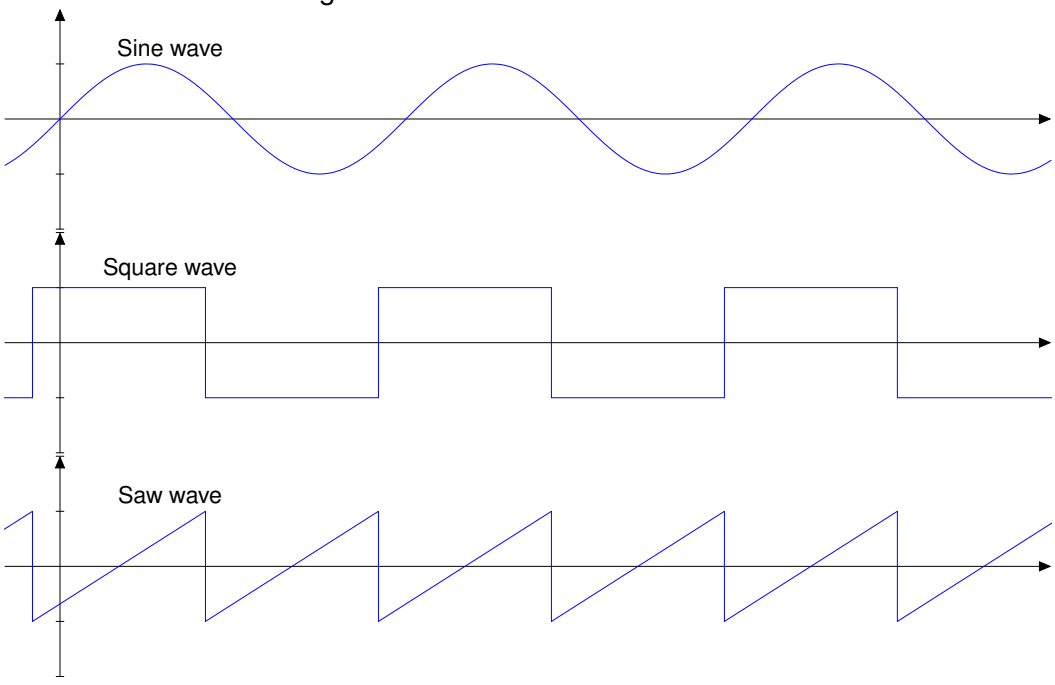
Making one sound is nice, but for making anything remotely musical one must be able to play several sounds at once. Infinite simultaneous sounds would be ideal, but is technically impossible. A number of different channels though, usually around 8 or 16, is simple to implement. Every sound buffer in the pipe is transformed into an array of buffers, and the starting time is transformed into an array. Modules then just iterate between all channels and behave normally for each pipe. Additionally, a nice feature would be stereo support. My implementation lacks stereo support in the java api calls and the gui, but the engine itself and all the modules and pipes can use stereo. For that, all buffers across all channels are duplicated, to be calculated separately. In fact, there is little difference between stereo and a second layer of channels for polyphony, and so it is treated much the same way.

4.2 Oscillators

Oscillators are generators that can create a periodic waveform. They are the basis of additive and subtractive synthesis by creating a basic wave to be shaped and modified by every other module. Usually osillators create a few types of mathematically well-defined primitive waveforms, like sine or square waves. Apart from the wave function, they also possess two parameters, frequency and phase. Frequency describes the amount of oscillations per second, and a difference in frequency is percieved as a difference in pitch. Phase is very difficult to percieve audibly in most cases, it is the initial offset the oscillation started with. Contrarily to frequency, phase has a bounded range, usually from 0 to 2π or from 0 to 1, depending on the implementation.

In this synthesizer, the 3 most fundamental waveforms are supported. These are the sine wave, the ”square wave” or pulse wave, and the popular sawtooth-wave, also called saw-wave.

Figure 1: Common basic waveforms



4.2.1 Basic implementation

As already noted, oscillators only require two parameters, phase and frequency. It is quite easy to deal with stereo behavior, as the channels are independent, one can simply first calculate the left side and then the right side. Oscillators also depend heavily on the global time and on the pipe starting time. Through this the current time since starting to oscillate can be calculated trivially, and it is generally a good idea to do modulo the period of the wave as well. Afterwards, the resulting signal is calculated differently depending on the type.

```
double x = time * frequency + phase;
switch (osc_type)
{
        case SINE_WAVE:
               return Math.sin(x*2*Math.PI);

        case SQUARE_WAVE:
               if (x <= 0.5)
               {
                   return 1;
               }
               else
               {
                   return -1;
               }

        case SAW_WAVE:
               return 2*(x - Math.floor(x)) - 1;
}
```

4.2.2 Anti-Aliased implementation

The previous approach worked fine for sine waves, but in both other waveforms there is a regular discontinuity. In the frequency range, this translates to an infinite frequency, inducing aliasing, as the Nyquist frequency cannot be infinite. There are many, many approaches to fixing this problem in general, it is a recurring obstacle in everything from 3D-graphics to font rendering, and it affects synthesis as well. Luckily, the two waveforms we need are special; their frequencies are easily represented as infinite sums. On top of that, we do not need a particularly fast solution, so we can settle for a mathematically simple and complete one.

The saw wave, in the frequency domain, is nothing but the sum of every harmonic (whole-number multiples of the base frequency) with a 1/k decaying amplitude. ($f =$

the base frequency, $F_N =$ the Nyquist frequency)

$$H(f) = \frac{2}{\pi} \sum_{k=1}^{\infty} \frac{\sin(k * f)}{k} \tag{1}$$

Note: The $\frac{2}{\pi}$ normalises the result between -1 and 1.

Of course, instead of summing to infinity, we could stop summing as soon as $k * f > F_N$, therefore only including overtones under the Nyquist frequency.

$$H(f) = \frac{2}{\pi} \sum_{k=1}^{\frac{F_N}{f}} \frac{\sin(k * f)}{k} \tag{2}$$

The same can be done for square waves. Square waves have nearly an identical frequency sum, except only odd harmonics have a non-zero amplitude (this also doubles the normalizing factor, since the range is still -1 to 1 but only half the elements are summed).

$$H(f) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{\sin((2 * k - 1) * f)}{2 * k - 1} \tag{3}$$

becomes

$$H(f) = \frac{4}{\pi} \sum_{k=1}^{\frac{F_N}{2f}} \frac{\sin((2 * k - 1) * f)}{2 * k - 1} \tag{4}$$

which again, only includes all parts of the square wave which are under the Nyquist frequency; a perfect solution, if we have the luxury of enough computing time to calculate this sum (which, for this synthesizer, we do).

In code, the final result looks like this:

```
double x = time * frequency + phase;
switch (osc_type)
{
        case SINE_WAVE:
                return Math.sin(x*2*Math.PI);

        case SAW_WAVE:
                result = 0;
                // Sawtooth = infinite sum of all harmonics with A=1/n for
                    nth harmonic
                // Source: http://en.wikipedia.org/wiki/Sawtooth_wave
                for (int k=1; k*freq<Constants.SAMPLING_RATE/2; k++)
                {
                        result += Math.sin(x*2*Math.PI*k)/k;
                }
                // Keep result -1 <= x <= 1, not -1.0903783642160645 because
                    of imprecision
                return Math.min(1, Math.max(-1, 2*result/Math.PI));

        case SQUARE_WAVE:
                result = 0;
                // Square = infinite sum of odd harmonics with A=1/n for nth
                    harmonic
                // Source:
                    http://en.wikipedia.org/wiki/Square_wave#Examining_the_square_wave
                for (int k=1; k*freq<Constants.SAMPLING_RATE/2; k+=2)
                {
                        result += Math.sin(x*2*Math.PI*k)/k;
                }
                // Keep result -1 <= x <= 1, not -1.0903783642160645 because
                    of imprecision
                return Math.min(1, Math.max(-1, 4*result/Math.PI));
}
```

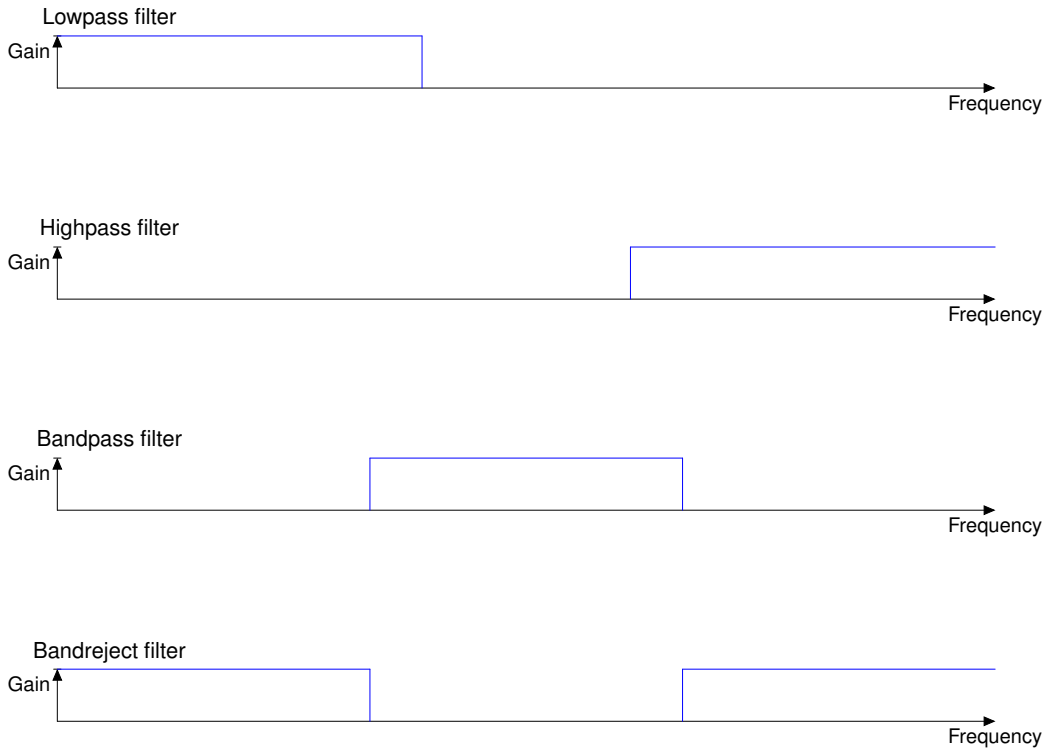
This code gives off perfect anti-aliasing and is fast enough for use in our synthesizer.

4.3 Filters

Filters are an enormously complex subject and a field for themselves, having a wide range of applications beyond music especially in communications and data analysis. The general definition of a filter is very broad, I am going to refer specifically to filters that modify the gain as a function of frequency. Even of these, innumerable examples exist, so I will concentrate on the commonly used musical filters. By far the most famous of those are the lowpass and highpass filters. A lowpass filter, as the name would suggest, lets low frequencies pass while blocking high frequencies. A highpass filter does just the opposite. There are more types of filters common in synthesizers, most notably bandpass and bandreject (also called notch) filters, but I have not implemented them, so I won't do more than mention them here.

All of the following filters have at least one very important parameter, the cutoff

Figure 2: Ideal filter frequency responses



frequency. Informally stated, this is the location of the filter. Raising the cutoff frequency of a lowpass filter for example allows more frequencies to pass through.

4.3.1 Implementation of a lowpass and highpass filter with FFT

The plan is to first convert the time-domain signal into frequency domain, using the Fast Fourier Transform (FFT), multiply it with our filter function, and then convert it back (using the Inverse Fast Fourier Transform, or IFFT). Contrarily to the ideal mathematical idea in the previous section, we have several limitations. For one, we do not have an input of infinite length, the input will come in small blocks at a time. Related is the problem that frequency is tied to the concept of a period. Our signals will only be approximately periodic, but certainly not precisely so. The result is that whenever we attempt to transform the filtered signal back into time-domain, it becomes a periodic signal, and both extremities change. This causes discontinuities between different frames, which results in audible buzzing. There is

a way to avoid this, using the overlap-add method, which depends on buffering a bit of extra data of previous frames and including it in the FFT, causing the IFFT to create a longer wave, which can be cut at the right place with much less of an error. It makes things worse though, because the FFT/IFFT depends on a signal length of 2^n (without either sacrificing a lot performance or using black magic), which means that the real signal length has to be chosen so that added with the overlap constant, it gives a 2^n signal. This whole implementation turns out to be rather complex, and quickly became a problem to program. So I searched for simpler methods with less book-keeping involved, and once I had found one I gave up on this particular method.

4.3.2 Implementation using (2-pole) recursive filters

One property of conversion between time and frequency domain when considering two functions is what happens to multiplication.

$$j(t) = h(t) \times g(t) \iff J(t) = H(t) * G(t) \tag{5}$$

and vis-versa

$$j(t) = h(t) * g(t) \iff J(t) = H(t) \times G(t) \tag{6}$$

What this means is that multiplying the frequency spectrums of two waves is identical to convolving their time domain functions. This means we could also simply pass our filter through an IFFT (the result is called a convolution kernel), and convolve our signal with that. Convolution works the following way

$$y[n] = h[0]x[n] + h[1]x[n-1] + h[2]x[n-2] + \dots \tag{7}$$

Note: Convolution also exists for continous functions, but we are dealing with discrete signals so I am switching to that notation.

The problem with convolution is that most interesting filters have very long impulse responses (the time-domain converse of the frequency response), ideal box-filters like we attempted before even possess infinite impulse responses. We can give up on ideal box-filters, it then becomes a story of trying to get as close as possible while keeping computation low. One method for this are recursive filters; Instead of

just using previous input values, we also use previous output values.

$$y[n] = b_0x[n]+b_1x[n-1]+b_2x[n-2]+\cdots+a_0y[n]+a_1y[n-1]+a_2y[n-2]+\ldots \quad (8)$$

While not being of obvious use, this allows us to bypass long convolution kernels. The mathematics for deriving the constants a_0,b_0,... is called the Z-transform and is quite complex. Skipping the details, I chose to copy an analog electronical circuit working with up to two previous values (hence being called a 2-pole filter). The equation becomes

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + a_0y[n] + a_1y[n-1] \quad (9)$$

The frequency response function isn't a perfect box filter anymore, but it is similar enough to work as well, and is much simpler. It has two parameters, the cutoff frequency (f_c) as per before, and also a strange parameter commonly called Q or resonance. Sampling frequency will be denoted as F_s. First defining ω_0 which we will need:

$$\omega_0 = \frac{2\pi f_c}{F_s} \quad (10)$$

The transfer function (equal to the frequency response) of the lowpass filter:

$$H(f) = \frac{\omega_0^2}{f^2 + \frac{\omega_0}{Q}f + \omega_0^2} \quad (11)$$

Converting this into coefficients for the recursive equation is a complex procedure, so I will directly jump to the actual values. First, we create a new constant a_0 and negate a_1 and a_2 to simplify later equations. Our new recursive equation becomes:

$$y[n] = \frac{b_0}{a_0}x[n] + \frac{b_1}{a_0}x[n-1] + \frac{b_2}{a_0}x[n-2] - \frac{a_1}{a_0}y[n-1] - \frac{a_2}{a_0}y[n-2] \quad (12)$$

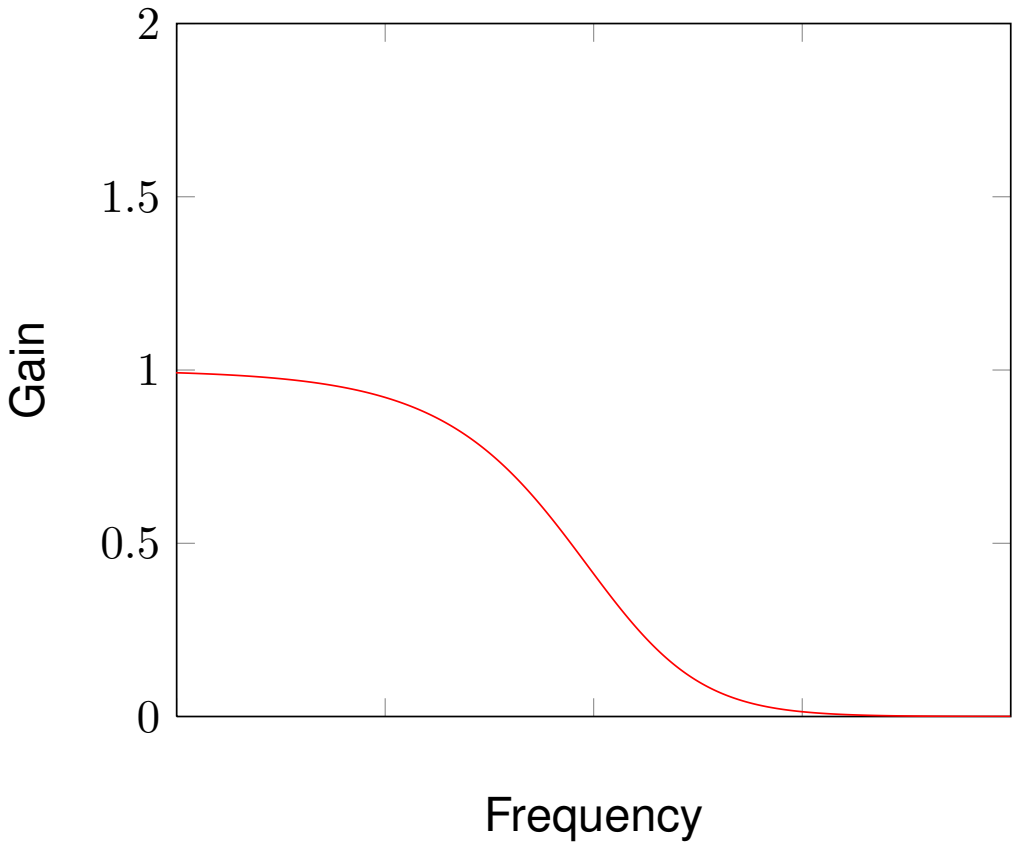
We first calculate α:

$$\alpha = \frac{\sin(\omega_0)}{2Q} \quad (13)$$

The coefficients for a lowpass filter:

$$b_0 = \frac{1-\cos(\omega_0)}{2} \quad (14)$$

Figure 3: 2-pole lowpass filter frequency response



$$b_1 = 1 - \cos(\omega_0) \tag{15}$$

$$b_2 = \frac{1 - \cos(\omega_0)}{2} \tag{16}$$

$$a_0 = 1 + \alpha \tag{17}$$

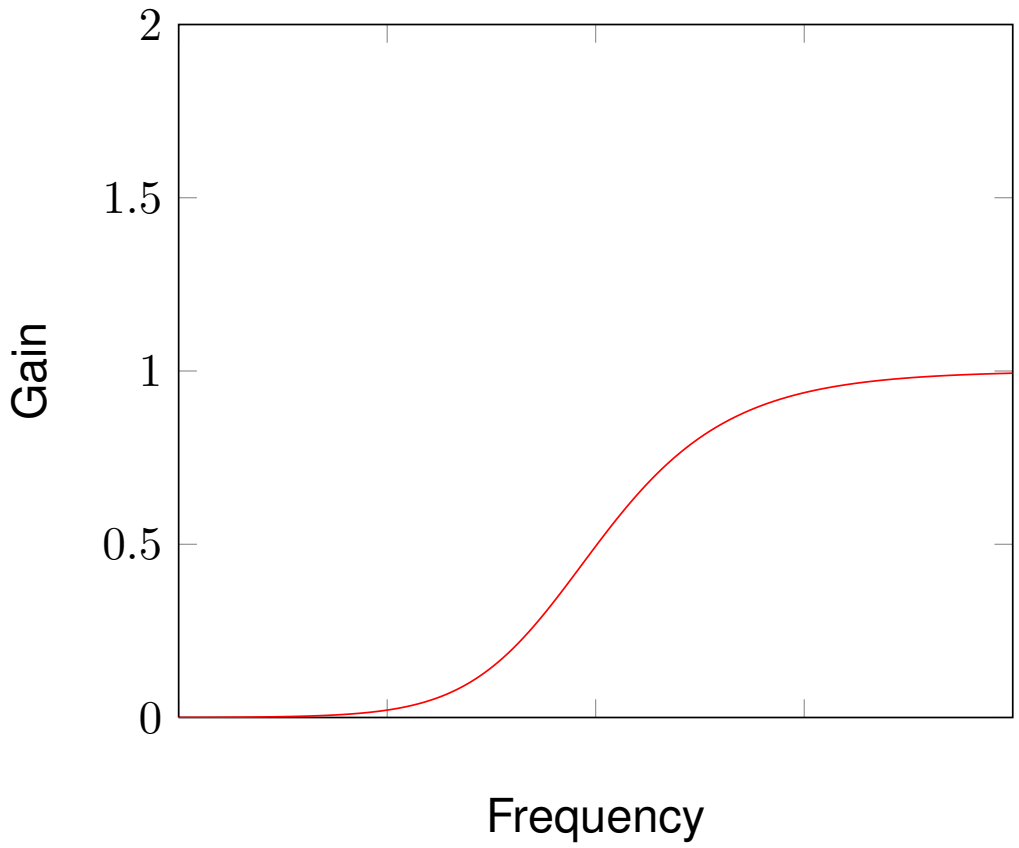
$$a_1 = -2\cos(\omega_0) \tag{18}$$

$$a_2 = 1 - \alpha \tag{19}$$

The transfer function for the highpass filter:

$$H(f) = \frac{f^2}{f^2 + \frac{\omega_0}{Q} f + \omega_0^2} \tag{20}$$

Figure 4: 2-pole highpass filter frequency response



The coefficients for a highpass filter (following the same recursive equation as the lowpass filter):

$$b_0 = \frac{1 + \cos(\omega_0)}{2} \tag{21}$$

$$b_1 = -1 - \cos(\omega_0)) \tag{22}$$

$$b_2 = \frac{1 + \cos(\omega_0)}{2} \tag{23}$$

$$a_0 = 1 + \alpha \tag{24}$$

$$a_1 = -2\cos(\omega_0) \tag{25}$$

$$a_2 = 1 - \alpha \tag{26}$$

This system has the huge advantage of being local, only the last two values of the previous frame must be buffered. It does have it's flaws though, mainly that phase is not kept constant throughout the transform, and this means it is possible that the signal oversteps the -1/1 bound. Being a feedback loop, this error then enters every other value and corrupts every future signal, even future frames. To limit this, I simply placed a max() bound at the appropriate place, preventing at least any distortion from spreading.

4.4 Other Primitives

Apart from the ”interesting” modules, there is a list of modules that are required in some form or other but whose inner workings are very simple, and that do not exist as explicit modules in most synthesizers.

4.4.1 Constants

Many modules or circuits need paramters, in fact anything not dependent on the input signal is a parameter. These parameters cannot be edited directly inside the module, since everything is supposed to be modular. To solve this problem with as little change to the engine as possible I simply created a ”Constants” module which streams a single value the whole time. It does this on all channels, since it is a source of input as much as the standard note input is. There is also a specific GUI for Constants, allowing the user to create one and directly type into a writeable box. This is the only way to parameters in this synthesizer.

4.4.2 Merger and Copyer

When attempting more complex kinds of synthesized sounds, it often becomes useful to split a source signal into different parts, edit those parts separately and then bring them together again. This, along with many other usecases, requires a module to duplicate a signal, and another one to merge them together again. The copying module is not very complex; all data in the input pipe gets copied identically in both output pipes. If more outputs are wanted, the user can chain copiers, although in principle the module supports more than two outputs. A merger module is slightly more complex, as there are (at least) two intuitive ways of combining signals. One can multiply them, or one can add them (overlaying the sounds). To give the user freedom of choice, I split the merger into two types, similarly to the oscillator, and a gui menu can be used to control the merging operation of each merger module.

4.4.3 Range Modifier

The purpose of a range modifier is simple; it's job is to linearly project a signal between two bounds to two other bounds. 4 parameters are required, the starting bounds s_{min} and s_{max} and the desired bounds d_{min} and d_{max}.

$$m = \frac{d_{max} - d_{min}}{s_{max} - s_{min}} \tag{27}$$

$$b = d_{min} - (s_{min} * ration) \tag{28}$$

$$y[t] = x[t] * m + b \tag{29}$$

This is especially useful if one needs to amplify or attenuate a signal, all you have to do is change the output bounds to match your desired result, and the signal will automatically get scaled to size.

4.5 GUI

5 Results

5.1 Capabilities and Limitations of this program

6 Looking forward

References

[1] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub, 1 edition, 1997. Can be found online at `http://www.dspguide.com/whatdsp.htm`.