Implementing a simple FM synthesizer with fixed modulation ratios in FAUST

Sound Synthesis: Building instruments with FAUST SoSe 2020

Teacher: Henrik von Coler
Students: Felix Weiß (349146), Jonas Körwer (382313)

Introduction and motivation

The goal of this project is to create a polyphonic FM synthesizer that creates harmonic sounds quickly and intuitively in the FAUST programming language. This is achieved through stacking six carrier oscillators, each with their own FM modulation oscillators. The frequency of the modulators is a variable to be set by the user, however it is limited to fixed integer ratios to only allow for harmonic sounds.

The idea for this project is inspired by two existing commercial synthesizers:
The Duality Oscillator of the Reaktor Blocks Kodiak family and the Moog Subharmonicon.
The Duality oscillator is, as the name suggests, an oscillator to be used in the software modular environment Reaktor Blocks. It comprises two oscillators which can be synced, frequency, amplitude and pulse-width modulation and a waveshaping processor. The frequency of the primary oscillator can be manually set or triggered via a pitch signal, while the secondary oscillator can be synced to the pitch of the primary oscillator or to a integer ratio or multiple of it. This latter feature, when modulated, provided quite interesting sounds when experimenting with the block and was the primary inspiration for this project.
The Moog Subharmonicon is a semi-modular polyrhythmic analog synthesizer that essentially allows the user to create sequences of chords via two main oscillators and four so called sub-oscillators. The sub-oscillators' frequencies are derived from the main oscillators frequencies resulting in the aforementioned harmonic chord structures. Two built-in sequencers then allow for polyrhythmic arrangement of these chords. It's property of stacking voices to create harmonic chords has inspired the structure of the subsequently described synth.
This project builds on the principle of fixed integer ratios to create harmonic signals via frequency modulation, and extends it by applying it to a polyphonic synthesizer to be played on a keyboard. Both existing instruments are not polyphonic, and one could argue that their strengths only fully develop when used in a sequencing environment. This synthesizer however strives when being played via a keyboard since both velocity and aftertouch influence modulation depth.
The goal of this project was to create a VST plug-in to be used in Ableton, as this was the DAW we used.

Theoretical foundation and implementation in FAUST

A mathematical model for FM Synthesis in the audio domain was first proposed in 1973 in the widely popular paper “The Synthesis of Complex Audio Spectra by Means of Frequency Modulation” by John M. Chowning. While this document will not reproduce his full argumentation, the main conclusions which are important for the implementation and resulting sound of the described FM synthesizer are shown.

In short, the underlying mathematical formula for FM modulated signals can be described as:

$$e = Asin(\alpha t + Isin\beta t)$$

where

- e = the instantaneous amplitude of the modulated carrier
- α = the carrier frequency in rad/s
- β = the modulating frequency in rad/s
- I = the modulation index, which is set individually in our implementation

Real world implementations, whether analog or digital, tend to use a second oscillator to create the modulation frequency. In the FAUST language this also holds true and the above mentioned basic formula for a modulated signal can be represented through:

```
os.osc(carrierFreq+os.osc(modulatorFreq)*index)
```

where

`os.osc` = a sine wave oscillator

Chowning (1973) goes on to demonstrate that with increasing modulation index the resulting spectrum includes more sideband components around the carrier frequency.
This is demonstrated in figure 1.

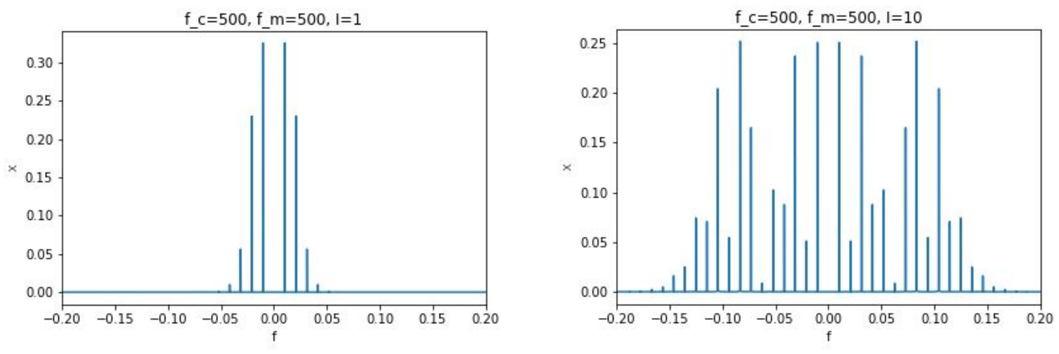


Fig 1: Spectrum of a modulated sinusoid signal of 500 Hz with Fm = 500 and modulation indices I = 1 and I = 10.

Furthermore, the position of these sidebands is determined by the ratio of the carrier frequency to the modulator frequency. When an integer ratio is applied, the relationships of the sidebands to the carrier frequencies can also be described in simple integer ratios, resulting in a harmonic sound. The opposite holds true for inharmonic signals. Figure 3 and 4 show this by applying a $\frac{2}{1}$ and $\frac{1}{\sqrt{2}}$ carrier- to modulator-frequency ratio respectively.

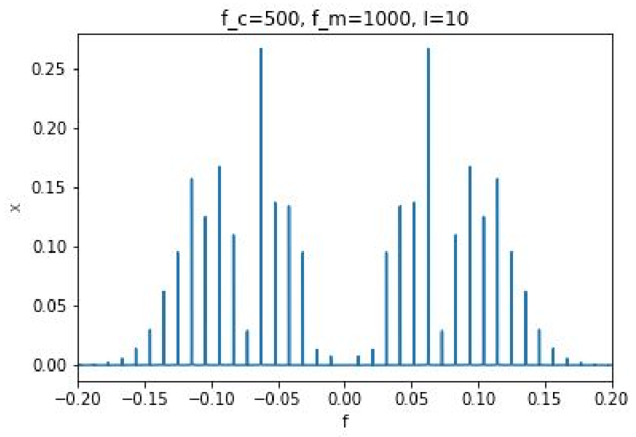


*Fig 3: Demonstration of harmonic sidebands. Spectrum of a modulated sinusoid signal of 500 Hz with Fm = 2*Fc and modulation index = 10*

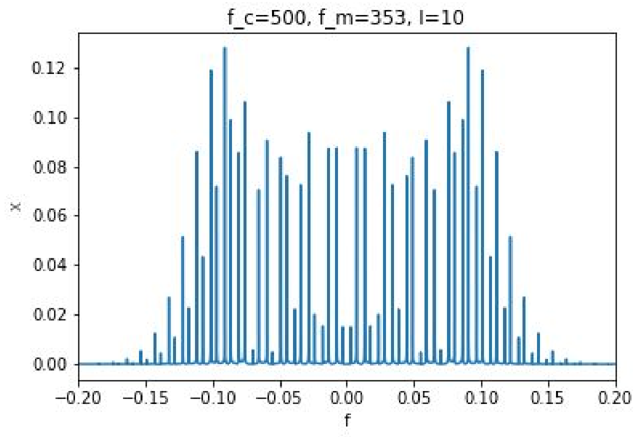


*Fig 4: Demonstration of inharmonic sidebands. Spectrum of a modulated sinusoid signal of 500 Hz with Fm = $\frac{1}{\sqrt{2}}$ * Fc and modulation index = 10*

In the herein discussed FM synthesizer, only fixed integer ratios for the modulation frequency are implemented while the modulation index is adjustable by the musician to allow for complex sounds that are always harmonic. Additionally the modulation index is scaled via an ADSR envelope to allow for the creation of a more dynamic signal. Each of the 6 oscillators has its own modulation envelope.

The full implementation of one oscillator voice in FAUST is as follows:

```
os.osc((c_f* modFreq1))
```

where modFreq1 is the modulation frequency which is calculated in another process like so:

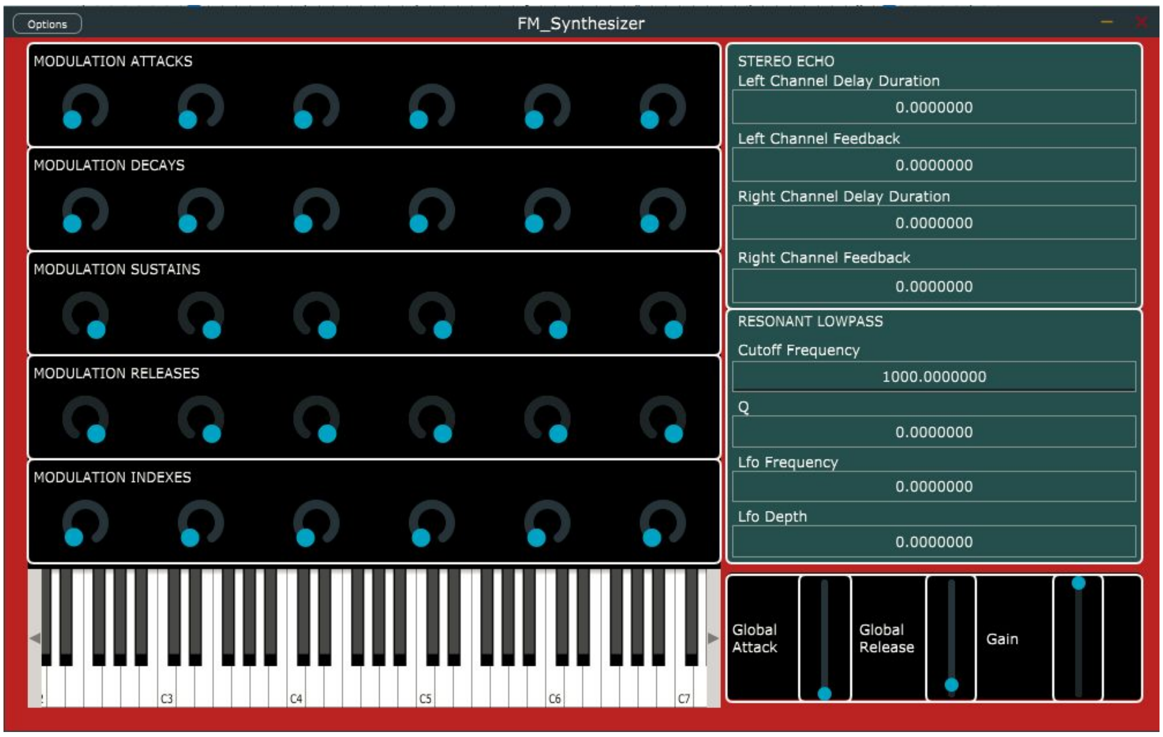
```
modFreq1 = ((os.osc(mod_freq1)*index1)*en.adsre(a1,d1,s1,r1,gate))
```

In total there are six of these voices in our synth which are summed together before being processed through a global gain control and a resonant lowpass filter with low frequency modulation. Initially an echo effect was also included as the final step in the audio chain, however this led to performance issues in the final VST export and was therefore discarded.

Implementation of the Faust DSP into the JUCE Framework

One of the benefits of programming in Faust is its compiler. It can be used to generate very optimised C++ code. This code can then for example be incorporated in projects created with JUCE. For generating the C++ code we used the compiler of the Faust Online IDE and exported the project as source code for juce-poly. The result is downloaded and consists of a dsp.h, a dsp.cpp and a readme file.
After that you use the Projucer to create a basic standalone plugin and drag the faust dsp header and .cpp file into the project folder. The Projucer is able to create a framework for different types of applications. The basic standalone plugin consists of a plugin processor which handles for example the dsp and midi messages and a plugin editor which handles the GUI layout. The resulting code is opened in an integrated development environment of your choice where it will be compiled and built. The C++ code for our project is found in the git repository under “Juce”.
In both header files created by the Projucer, PluginProcessor.h and PluginEditor.h, you must include the DspFaust.h file. In addition to that you create a private member of the DspFaust class in the PluginProcessor.h. With that member, which we named simply dspFaust, it’s possible to communicate between the faust dsp, the plugin processor and the plugin editor of JUCE. In the constructor of the PluginProcessor you call the function start() on the dspFaust object which starts the dsp process. In the destructor you do the opposite with the function stop(). In the plugin editor you create all the sliders and buttons that are needed for the faust application. For each of the implemented controls a function needs to be declared and defined in the PluginProcessor.h and PluginProcessor.ccp respectively. In those functions you call the function setParamValue() on the dspFaust object, which needs the address to the actual parameter in the DspFaust.cpp. Those addresses are listed in the readme file that the Faust online compiler created.
In addition to the controls needed by the faust application, we added a MidiKeyboardComponent, which displays a midi keyboard that can be clicked to play notes. To send the Midi messages from the displayed keyboard to the faust dsp the keyOn() and keyOff() methods are called on the dspFaust object.
In the constructor of the PluginEditor we make all the controls visible, edit the look in its paint() method and organise the controls in the resized() method. To send the Midi messages from the displayed keyboard to the faust dsp the keyOn() and keyOff() method is used.

The following picture shows the finished standalone synthesizer.



STEREO ECHO

Left Channel Delay Duration: 0.0000000

Left Channel Feedback: 0.0000000

Right Channel Delay Duration: 0.0000000

Right Channel Feedback: 0.0000000

RESONANT LOWPASS

Cutoff Frequency: 1000.0000000

Q: 0.0000000

Lfo Frequency: 0.0000000

Lfo Depth: 0.0000000

Fig 4: Graphical User Interface created with JUCE

Limitations and Conclusion

Implementing our idea in FAUST turned out to be fairly straightforward. Stacking oscillators with individual controls and experimenting with various effects in the chain was easy and fast, mainly because of the language used. Being able to export to multiple formats could be seen as another major advantage of the FAUST framework, however we experienced many issues in creating the final VST instrument. The delay effect led to major performance issues, and MIDI was and still is unreliable unfortunately. Overall, the final instrument does provide a good sound with acceptable performance.

The combination of the dsp code written in Faust embedded in the JUCE framework is very powerful. Complex dsp algorithms can be easily written in Faust, whereas JUCE handles the compatibility with different operating systems as well as providing good looking GUI components. Disadvantages lay in the generated C++ Code of the Faust compiler, it is hard to read which makes it even harder to manipulate. A simple thing like enabling an external MIDI device to send notes to the dspFaust object can be quite a challenge.

References:

Chowning, J. M. (1973). The synthesis of complex audio spectra by means of frequency modulation. *Journal of the audio engineering society*, *21*(7), 526-534.
Retrieved here

The code used to produce the graphs in this document draws heavily from the work of Henrik von Coler: Formula & Spectra