Sound processing: FM Synthesizer

Simon Andersen
Aalborg University CPH
sand22@student.aau.dk

ABSTRACT

This paper is about the implementation of John M. Chowning's traditional FM synthesis algorithm in the JUCE platform as a MIDI plugin. As well as an description of the algorithm in use and a technical one. The complete J. Chowning implementation were not fully achieved, but it was managed to setup the necessary means of a MIDI synthesizer doing simple frequency modulation.

1. INTRODUCTION

From the findings of John M. Chowning's, the spectacle of frequency modulation or FM was explored in used in musical composition [1] [2]. What J. Chowning discovered was a computationally cheap approach to produce intricate dynamic timbres of sound digitally [1]. By this he could with his own examples imitate real-world sounds. His discovery lead to the release of synthesizers based upon J. Chowning's concept of working with FM Synthesis.

2. PROBLEM STATEMENT

The objective of this project is to implement a traditional FM synthesizer in the JUCE platform with the architecture from John M. Chowning.

3. FREQUENCY MODULATION

Modulation is the process of altering data by adding additional information to a signal [3]. This is often used in the alteration of frequency, phase or amplitude of an oscillator. In it's basic form, modulation in sound processing is constructed by the use of two oscillators known as the *carrier & modulator*. The carrier is the signal that is being altered whereas the modulator is one that transform the carrier[3]. Specifically in FM synthesis it is the frequency component of the of the carrier oscillator that is being modulated. The carrier and modulator signal can be seen as steady waveforms where the frequency and amplitude can be altered. Then the output is a varying waveform to reflect the modulation as seen in figure 1.

This technology has been used in a variety of areas in example data transfer[2]. What is interesting and what J. Chowning were able to utilize in FM synthesis was the fact that the change of frequency from the modulator signal produced new harmonics that could be translated into musicality. These harmonics are known as 'sidebands'.

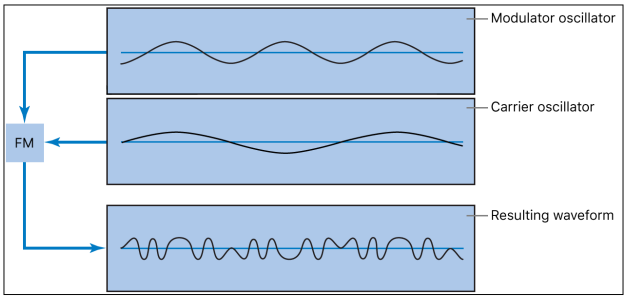

Figure 1: Two oscillators creating a different waveform output in FM synthesis. (courtesy of Apple Logic Pro Guide [4])

This is exactly what makes FM stand out and create the timbre of the sound[3].

The FM Synthesis computation includes the components of amplitude and frequency of both the carrier and modulator. These components can be defined as $A_{carrier}, A_{modulator}$ & $f_{carrier}, f_{modulator}$. Importantly the $A_{modulator}$ component is known as the *modulation index* or *modulation depth* specifying the magnitude of modulation attributed to the carrier. This is therefore a configuration parameter that is adjustable meaning a value of zero will provide no frequency modulation.

3.1 Algorithm

The simplest form that can describe the relationship of the carrier and modulator signal in FM synthesis is the following equation[2]:

$$y(t) = A_{carrier} \cdot sin(2 \cdot \pi \cdot f_{carrier} \cdot t + A_{mod} \cdot sin(2 \cdot \pi \cdot f_{mod} \cdot t)$$

Whereas it becomes clear to denote that the amplitude component or modulation index is the factor that controls the amount of modulation applied. For the usability component of FM synthesis it lies in the modulation index and -frequency that are the most crucial feature. The increase in modulation index means a wider spread of sidebands. This manages the timbre of the sound produced.

To implement the specific FM algorithm in software the equation is found as being the following [2]:

$$y[n] = A_c \cdot cos(2 \cdot \pi \cdot f_c \cdot \frac{n}{f_s} + \frac{A_{mod}}{f_{mod}} \cdot sin(2 \cdot \pi \cdot f_{mod} \cdot \frac{n}{f_s})$$

Where n is the number of sample and f_s is sample rate.

3.2 Chowning FM

In the implementation of J. Chowning's FM synthesizer there is one last element in highlighting the timbral abilities of the algorithm[1]. This is found in the dynamic variation of the adjustable parameters. To achieve this the parameters can be controlled by applying envelopes that changes them over time. With the implementation of said envelopes it rewards the synthesizer of resembling the qualities of an acoustic instrument[1]. For this a simple implementation design can be seen on figure 2.

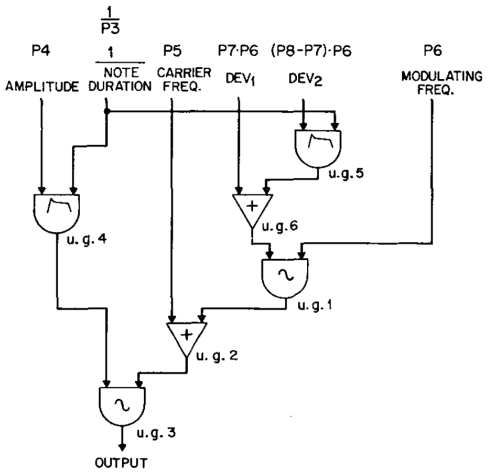

Figure 2: Simple design scheme of FM synthesizer (Courtesy of J. Chowning [1]).

The dynamic spectra produced from the FM synthesis is closely related to the modulation index[1]. So when we monitor an increase of the index the entire spectre does as well. Chowning himself stated for a more controllable system a formant peak can be added in the spectrum[1]. Generating a more manageable circuit as seen on 3.

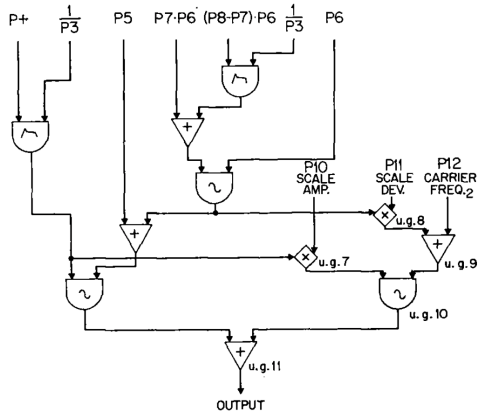

Figure 3: Complex design scheme of FM synthesizer (Courtesy of J. Chowning [1]).

4. PROJECT IMPLEMENTATION

The implementation made for this project was developed in the C++ framework JUCE [1] [5]. The following implementation takes great inspiration from the *"Let's Build a Synth VST Plug-in with JUCE"* [6] from YouTube channel *'The Audio Programmer'* [2] channel on YouTube. The implemented code from the tutorial can also be seen in the GitHub repository [3] from the project. The tutorial was meant to be used as the setup of a MIDI synthesizer which is developed in the playlist. When the architecture for the synth was build the addition of the Chowning FM synthesizer design would be implemented on top of the foundation.

4.1 JUCE design

The plugin platform of audio implementation was used in the standalone format. The specific classes used for the synthesizer was the following:

- **Synthesizer object**
  Defined as the *"Base class for a musical device that can play sounds"*.
- **2 Oscillators**
  Sine wave oscillators.
- **2 Envelope objects**
  Object controlling the attack, decay, sustain and release of an audio stream.
- **2 Envelope parameter objects**
  Object managing and saving the specific parameters set with a respective envelope object.
- **AudioBuffer**
  Object used when working, processing and managing samples.

4.2 Setup

The first part of the implementation consist of the setup for the ability to create a synthesizer system through JUCE. This means adding the JUCE Synthesier class to the program. Then followed by defining the sub-classes from the *Synthesizer* class in order to describe which sounds is accessible for the plugin and allow playback of the chosen sounds. The set of *SynthSound(.h)* & *SynthVoices(.h, .cpp)* classes were added for this purpose and connected to the plugin processor. For the current implementation one voice was configured making it a monophonic synthesizer and applied to the sound class.

Importantly the synthesizer contains the *renderNextBlock()* method that continuously should be used in order to create the audio and is where the FM Synthesis processing takes place. Other than that the MIDI compatibility is applied in the dependencies of the project allowing the plugin to process midi information when it occurs. The controlling methods of determining when a note is played and stopped is made available by the *noteOn & noteOff* methods used for the envelope segment. Then at last the sample rate is passed into the reference of the synthesizer of the main

[1] https://juce.com/

[2] https://www.youtube.com/@TheAudioProgrammer

[3] https://github.com/TheAudioProgrammer/tapSynth

plugin processor through the prepareToPlay() method. In addition, preparing the specific voices and calling the synthesizers preperation method. Now the plugin's synthesizer object is ready to do the processing shown on figure 4.

```
synth.setCurrentPlaybackSampleRate(sampleRate);

for (int i = 0; i < synth.getNumVoices(); i++) {
    if (auto voice = dynamic_cast<SynthVoices*>(synth.getVoice(i))) {
        voice->prepareToPlay(sampleRate, samplesPerBlock, getTotalNum
    }
}
```

Figure 4: Example on the implemented frequency modulator calculation)

4.3 FM Implementation

The two implemented oscillators in the system should resemble the carrier and modulator. Whereas one of them would change the frequency of the carrier signal. To be able to achieve this feature the system was supposed to do sample by sample processing. This was managed by adding the *AudioBlock* component from the dsp library of JUCE. The aforementioned component would enable access to check sample by sample instead of using an entire block of data. This way, the system are iterating over every channel and sample in the block. Also when doing the processing of the carrier signal we want to add the *ProcessContextReplacing* dsp component. Because we want to change the context information that is passed into the process method of the carrier. Since a single block is applied for both in- and output. Then in this architecture we are able to use the specific sample in order to do the processing calculation as seen on figure 5.

```
for (int channelBlock = 0; channelBlock < audioBlock.getNumChannels(); channelBlock++) {
    for (int sampleBlock = 0; sampleBlock < audioBlock.getNumSamples(); sampleBlock++) {
        freqModulator = modulator.processSample(audioBlock.getSample(channelBlock, sampleBlock)) *
            depthModulator;
    }
}
```

Figure 5: Example on the implemented frequency modulator calculation)

4.4 Parametric control

For the J. Chowning FM Synthesis we want the frequency and modulation index to be controlled. These parameters were saved managed as the following.

```
juce::ParameterID{"FREQFM", 1}, "FreqFM"
juce::ParameterID{"DEPTHFM", 1}, "DepthFM"
```

Whereas 'FreqFM' is defined as the frequency set of the modulator and the 'DepthFM' is the variation of how much of the frequency should change over time. Now as mentioned before in the processing segment the calculation for each sample is created by following equation:

$$mod_{index} = sample_{mod} \cdot f_{depth}$$

The modulation index is then applied when the frequency is set for the carrier signal in the startNote() method by the following.

```
carrier.setFrequncy(MIDI + freqModulator)
```

Now mentioned in the J. Chowning architecture two envelopes are required to produce the dynamic spectra. These envelopes were added as ADSR JUCE classes. One of the modulators were supposed to modify the amplitude of the carrier signal whereas the second should change the modulation index of the modulation signal. The first envelope was able to be implemented but applying the second envelope to the mod_{index} failed.

4.5 UI

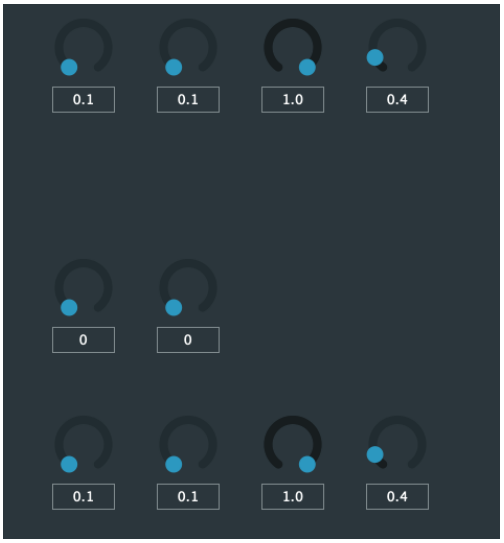

Figure 6: Example on the implemented frequency modulator calculation)

For the user interface referred to in figure 6, simple knobs were added to configure the individual envelopes of attack, decay, sustain and release and incorporated with the parameter changes. The lower row of four knobs just doesn't do anything for now, since these were reserved for the modulation index envelope. The middle knobs being two beside each other modifies the parameters for the modulator signal. The right being the specified frequency and the left being the depth.

4.6 Result

The algorithm and architecture from J. Chowning were supposed to be implemented but ended up being more of a simplified frequency modulation synthesizer were the modulation index ratio is not implemented probably. This was due to not fully understanding the means of the parameters that had to be used for Chowning's architecture.

5. CONCLUSION

The objective of this project was to implement a traditional Chowning FM synthesizer in JUCE with the architecture specified in the paper *"The synthesis of complex audio spectra by means of frequency modulation"* [1]. I managed to complete the setup of a synthesizer architecture in JUCE. Though, I weren't able to implement the

full Chowning FM synth I did implement simple frequency modulation. My understanding of how the conversion from the algorithm to the system weren't extensive enough. So I wasn't able to translate the traditional FM architecture into code.

References

[1] J. M. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the audio engineering society*, vol. 21, no. 7, pp. 526–534, 1973.

[2] I. to Computer Music Editorial. Chapter four: Synthesis. [Online]. Available: https://cmtext.indiana.edu/synthesis/chapter4_fm.php

[3] T. H. Park, *Introduction to digital signal processing: Computer musically speaking*. World Scientific, 2009.

[4] A. L. P. G. Editorial. Frequency modulation (fm) synthesis. [Online]. Available: https://support.apple.com/da-dk/guide/logicpro/lgsife418213/mac

[5] J. Editorial. Juce documentation. [Online]. Available: https://docs.juce.com/master/index.html

[6] T. A. Programmer. Let's build a synth vst plug-in with juce (2020). [Online]. Available: https://www.youtube.com/playlist?list=PLLgJJsrdwhPwJimt5vtHtNmu63OucmPck