RAVENSBURG-WEINGARTEN UNIVERSITY

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

IN COLLABORATION WITH

UNIVERSITY OF REYKJAVÍK

Phase Modulation audio synthesis on a μcontroller platform

Bachelor Thesis

Author:
Hallmar Gauti Halldórsson
Matrikel:
29404
Contact:
halligauti@hotmail.com

Supervisor I:
Prof. Dr. rer. nat. Markus Pfeil
Supervisor II:
Adjunct Baldur Thorgilsson

February 20, 2021

i Statement of Authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Reykjavík, February 21, 2021 .....

ii Acknowledgments

I would like to thank first of my two instructors, Markus Pfeil and Baldur Thorgilsson for guiding me through this project and giving insightful information and advice throughout the writing of this thesis and helping me with problems that have come up in this project. I would also like to thank the University of Reykjavík and their willingness to help me with finishing my last semester with them and to find a suitable instructor at their school. Further thanks go to my family who have helped me throughout my studies and have always been supportive of my obsession of audio related technology and music. And last but not least I would like to thank Morgan Greene for helping me go over the grammar in this thesis and for improving the text immensely.

Contents

iii Abbreviations

- **ADC**: analog to digital converter
- **DAC**: digital to analog converter
- **SoC**: system on a chip
- **PM**: phase modulation
- **FM**: frequency modulation
- **DSP**: digital signal processing
- **THD**: total harmonic distortion
- **Fs**: sample frequency
- $\Delta\Sigma$: delta sigma
- **MSB**: most significant bit
- **SCK**: serial clock
- **WS**: word select
- **SD**: serial data
- **MOSI**: master out slave in
- **D/C**: data/command
- **CS**: chip select
- **CLK**: clock
- **EMI**: Electromagnetic interference
- **FFT**: fast fourier transform
- **UI**: user interface
- **IDE**: integrated development enviroment
- **PWM**: pulse width modulated
- **MIDI**:Musical Instrument Digital Interface
- **THD+N**: Total harmonic distortion plus noise
- **RF**: Radio frequency

5 Introduction

Audio synthesis is a way of creating musical sounds with electronics and it can be played with a keyboard or any other control device, be it a standard 12 note keyboard or another type of sensor.

This method is not a commonly known topic but audio synthesis has woven itself into our western musical culture with notable artists such as Genesis, Pink Floyd, New Order, Kraftwerk and alot more contemporary artists. Additionally audio synthesis has a long history of innovators and engineers figuring out new ways to make new sounds [**?**]. In order to synthesize audio, a sound engineer/synthesist starts with a basic waveshape like a sinusoidal wave or a square wave and decides what kind of synthesis they want to apply.

A simple method for audio synthesis is the application of reductive synthesis. Reductive synthesis is a method where the sound engineer filters out the higher harmonics of a harmonically rich waveform or the sound engineer can apply additive synthesis where he/she/they adds higher or lower harmonics to a sine wave. There are many more complicated methods that have been developed since initial experimentation in audio synthesis. Some examples are phase distortion synthesis, frequency modulation synthesis, granular synthesis and more [**?**].

My focus will be on phase modulation synthesis and documentation on the hardware platform that I designed for synthesizing phase modulated sounds. This type of synthesis can be implemented in various ways and configurations to be able to make sounds. In its most basic form we have a sinusoidal oscillator that is being phase modulated by a second sinusoidal oscillator. The frequency relation and modulation intensity between these two oscillators will change the timbre of the sounds being generated.

5.1 State of the art

The synthesizer industry is an industry defined by two seperate products, forward thinking innovative designs [**?**] [**?**] [**?**] and re-issued classic designs with nostalgic or vintage appeal [**?**] [**?**] [**?**]. The landscape of creating synthesis methods has changed alot for the past decades, it has gone from being a very costly effort and time consuming mission to a fairly cheap and relatively quick design path, as is shown with most small companies that are making innovating products in the synthesizer industry.

The synthesis method that I am writing this thesis on has been around for quite some time [**?**]. The main distinction that I have from the original method is that my application of the synthesis method has multiple choosable waveshapes where as most phase modulation synthesis platforms don't [**?**] [**?**] [**?**]. I have also not seen any synthesizers that label themselves as "phase modulation synthesizer" in the industry, most of them always label themselves as frequency modulation synthesizers but most often they use phase modulation because computationally it is easier to add [**?**, p.8 eq. 22] than to multiply [**?**, p. 2 eq.1].

This all adds up to the fact that the hardware platform, named Fjöl, that is based on this method definately has something new to offer to the synthesizer industry with the variable waveshapes and its proper use of the name phase modulation synthesizer.

6 Overview

My main goal with this thesis is to give a consistent and good explanation of phase modulation synthesis and to give a good overview of the hardware platform that I designed to do such synthesis. My driving motivation behind this is that I have never found a good summary or documentation on this subject in particular online. What is also a huge motivation for me is simply the very interesting and pleasing sounds that this kind of synthesis can create.

In this document I will outline the design of my microcontroller platform, the theoretical framework of phase modulation synthesis, the programming of my design and the measurements and calculations required for the hardware platform. I will also include pictures and oscilloscope measurements of different aspects of the hardware platform, for example oscilloscope pictures of the phase modulated waveforms.

These following tools were used to design, plan and program the hardware platform:

- KiCAD for circuit design
- Matlab for simulation of PM synthesis and plotting oscilloscope measurements
- Arduino IDE for compiling
- Sublime Text for text editing/programming
- Teensy Audio Design tool for audio path design
- Github for project documentation

7 Hardware

7.1 Overview

The hardware that I designed has been based on other designs of mine and of another open-source hardware project called Clouds which was designed by Émilie Gilet of Mutable Instruments [**?**]. It's all designed around the Teensy 4.0 microcontroller which sports an ARM Cortex M7 processor, has two 12-bit ADC converter chips, SPI interface, MIDI interface, i^2s interface and alot of other features [**?**]. The overall schematic and design files for the system can be viewed via the links in the footnote on this page [1]. A picture of the front panel UI of the hardware platform can be found in Chapter **??** of this thesis.

A block diagram of the hardware is shown in figure **??**. The hardware consists of the Teensy microcontroller, the control voltage conditioners, low pass filter, opto-isolator for the MIDI input, output buffering and more.

I would also like to note that the power supply is a bipolar 12V supply. I did not design a specific power supply for the platform since my platform adheres to the Eurorack synthesizer standard which was created by Dieter Doepfer [**?**].

[1] https://github.com/hallmar/Bachelor-thesis/tree/main/Hardware

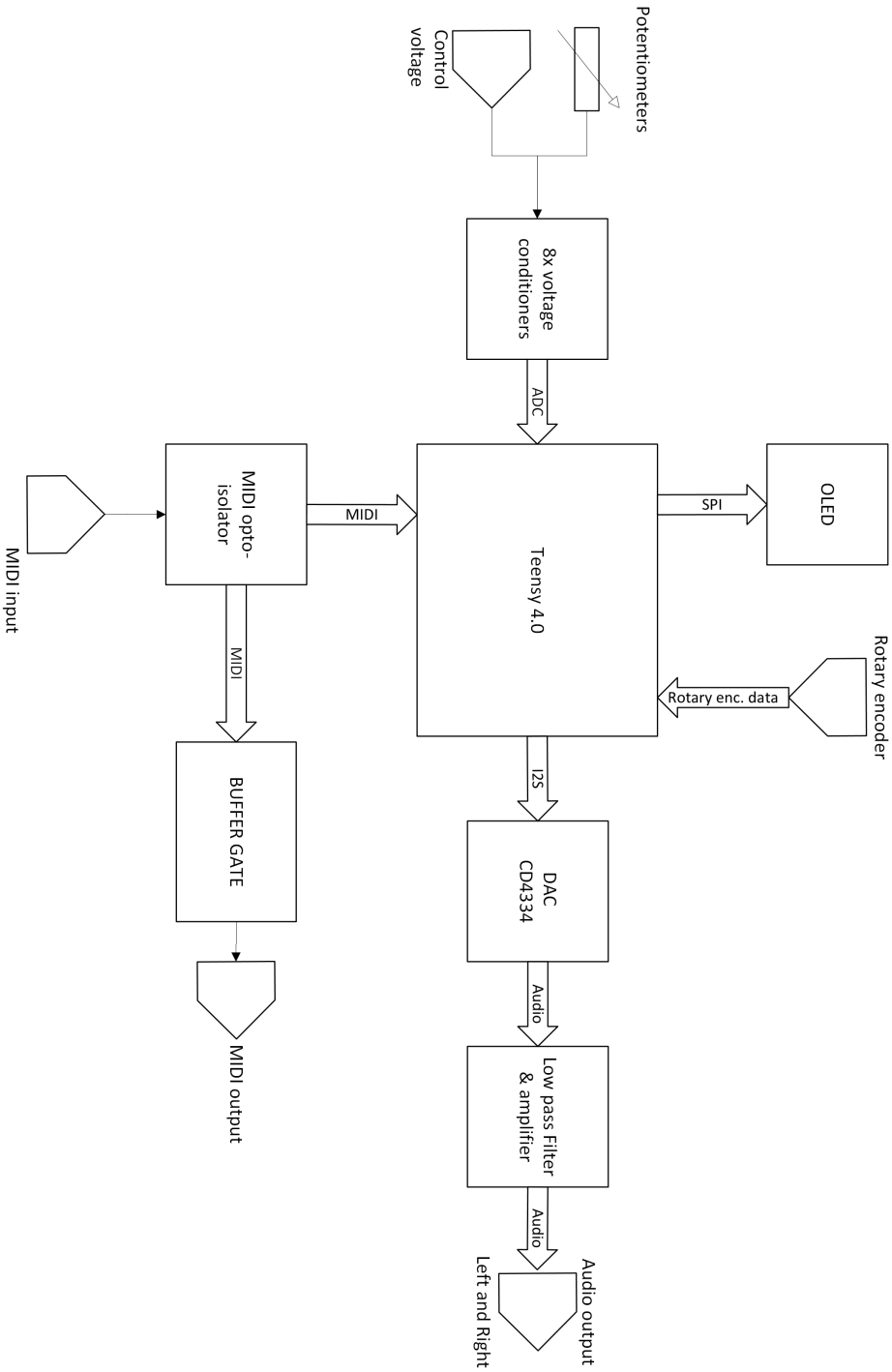

Figure 1: Block diagram of the hardware architecture

As can be seen in figure **??** there are multiple voltage conditioners that take in either control voltage from an outside source or a control voltage from the potentiometers which are mounted directly on the device and allow control over the algorithm.

There is a rotary encoder along with one pushbutton that can also control other miscellaneous things within the program, such as selecting a MIDI channel, selecting which type of PM synthesis algorithm is used and ect.

Then there is a digital to audio converter chip named CD4334 from Cirrus Logic which changes an i^2s serial datastream into an analog audio signal, which then goes through a filter and an amplifier.

And last but not least there is the MIDI input, which is a serial standard for controlling musical instruments. There is an opto isolator to isolate the ground between the sender and receiver of the MIDI serial stream. The MIDI standard is discussed further in Chapter **??**.

7.2 Processor

As mentioned before the processor/microcontroller is a Teensy 4.0 SoC. My main reason for choosing this microcontroller is because I have lots of experience with it and the designers of the Teensy also have an open access and extensive audio processing library of different audio functions. This makes the goal of implementing PM synthesis much easier since I did not have to program the audio DSP from scratch.
There are other options available over the Teensy, for example the Electrosmith Daisy [**?**] which has a large audio DSP library and costs around the same as a Teensy but I chose not to use that since I'm not that familiar with it yet and it would have made things harder than they should be.

The specifications of the Teensy microcontroller are as follows:

- ARM Cortex-M7 at 600Mhz and over-clockable
- 1024K RAM
- 2048K Flash
- 2 USB ports, both 480 MBit/sec
- 3 CAN Bus (1 with CAN FD)
- 2 i^2s Digital Audio
- 1 S/PDIF Digital Audio
- 1 SDIO (4 bit) native SD
- 3 SPI, all with 16 word FIFO
- 3 i^2c, all with 4 byte FIFO
- 7 Serial, all with 4 byte FIFO
- 32 general purpose DMA channels
- 31 PWM pins
- 40 digital pins, all interrupt capable
- 14 analog pins, 2 ADCs on chip
- Cryptographic Acceleration
- Random Number Generator
- RTC for date/time
- Programmable FlexIO
- Pixel Processing Pipeline
- Peripheral cross triggering
- Power On/Off management

A photo of the Teensy 4.0 microcontroller can be seen in figure **??**.

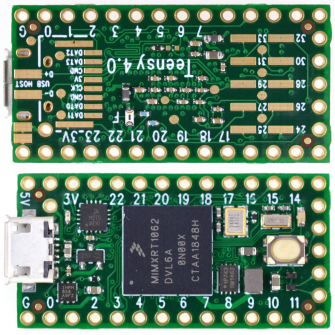

Figure 2: Teensy 4.0 microcontroller

7.3 Digital to analog audio conversion

For this project I chose the Cirrus Logic CS4334 DAC chip. I chose this DAC chip because it has a small 8 pin footprint, uses the i^2s(Chapter **??**) standard for receiving serial audio data, has a Total harmonic distortion plus noise(THD+N) performance of -88dB [**?**, p.6], dynamic range of 94dB [**?**, p.6], can receive up to 24bit data [**?**, p.1] and has a built in low pass reconstruction filter [**?**, p. 1].
It is worth mentioning that it's always better to have an extra low pass filter at the output to further increase the performance of the chip by filtering out frequency components that you cannot hear but will increase noise if not removed [**?**, p. 3].

During the creation of this project I did run in to problems with this DAC which will be discussed in Chapter **??** and a possible solution/replacement chip is mentioned in Chapter **??**.

A block diagram that describes the overall function of the DAC chip [**?**, p. 12] can be seen in figure **??**.

The general function is that the serial data goes into an interpolator which works as the datasheet states [**?**, p. 12]:

> The digital interpolation filter increases the sample rate, Fs, by a factor of 4 and is followed by a 32x digital sample-and-hold (16x in HRM). This filter eliminates images of the baseband audio signal which exist at multiples of the input sample rate. The resulting frequency spectrum has images of the input signal at multiples of 4 Fs. These images are easily removed by the on-chip analog low-pass filter and a simple external analog filter.

The Teensy audio library uses BRM or Base Rate Mode, in which case the sample rate is 44.1kHz.

The digital signal then goes into a fourth order $\Delta\Sigma$ modulator [**?**] which converts the interpolated signal into a 1-bit data stream which is at 128x Fs datarate [**?**, p. 12].

Then there is a switched capacitor DAC circuit that takes the 1-bit data stream which switches the Vref, which in this case is +5V into an operational amplifier with a capacitor in its feedback path which then produces a smooth analog waveform [**?**, p. 3]. The general idea of this circuit is shown in figure **??**.

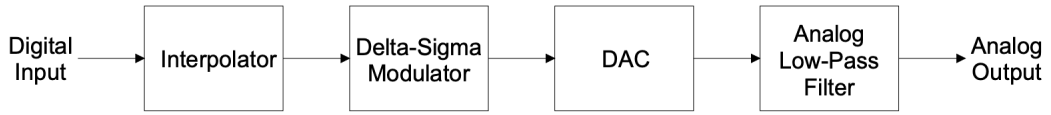

Figure 3: Block diagram of one audio channel of the DAC chip

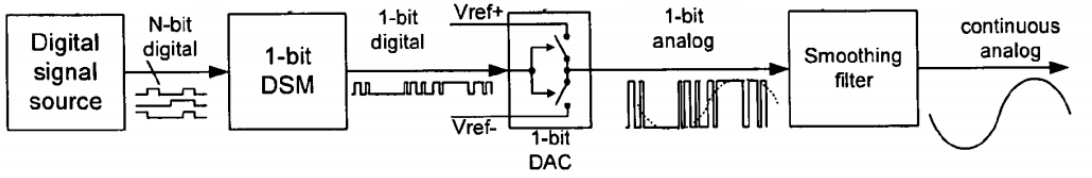

Figure 4: Switched capacitor Digital to analog converter [**?**]

With the sample rate(Fs) of 44.1kHz and 18 to 24-bit resolution the performance of the chip is as noted in the datasheet [**?**, p. 6]:

- -88dB Total harmonic distortion + noise(THD+N)
- +0.05/-0.25 dB De-emphasis error
- $\frac{9}{44100Hz} = 204\mu$ second Group Delay[2]
- 94dB dynamic range

[2]Time it takes for the digital data to be produced at the analog output

7.3.1 i^2s serial standard

i^2s is a serial communication standard which is specifically tailored for transmitting and receiving a high fidelity audio signal of various sample rates and bit depth. The signals that are used in this standard are as follows:

- Serial clock or SCK which is the master clock for the data transmission
- Word Select or WS which tells the receiver whether it's receiving the right or left channel of the audio signal
- Serial Data or SD which is the data for each sample point of the corresponding channel

An optional master clock can be implemented but that is up to each manufacturer of DAC or ADC chips, in my case the CS4334 chip requires a master clock for the $\Delta\Sigma$ block within the DAC itself.

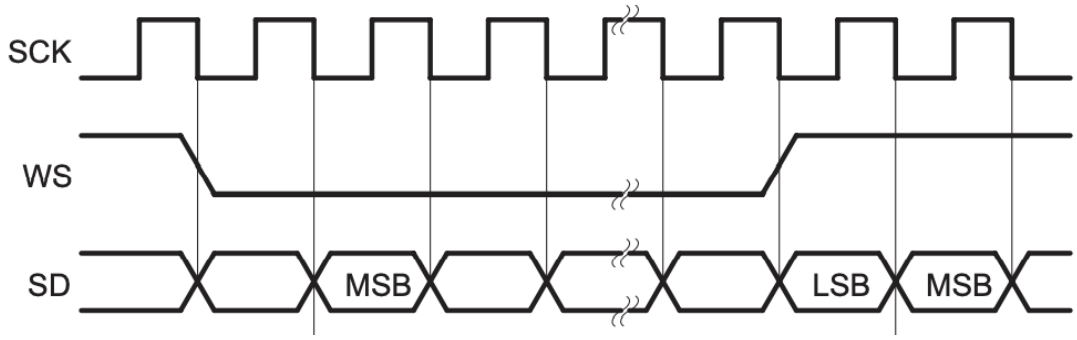

Figure 5: Clock timing diagram of i^2s standard

Figure **??** shows the i^2s stream and note that the MSB is transmitted first, this is because the receiver does not know how many bits it will receive, it only stops receiving the data when the WS pin changes its state. When the WS pin changes it indicates that the i^2s transmitter is changing which channel the i^2s receiver is receiving. WS is 0 for left channel and 1 for right channel.
As an example the DAC is enabled to receive up to 24 bits then the transmitter of the data could send just 16 bits, change the WS pin state and then the DAC would fill up the lower 8 bits with zeroes [**?**, p. 2].

This serial data is synchronous with the falling edge of SCK.

7.3.2 Low pass filter

I chose to use an active filter by Cirrus Logic which they designed for their evaluation board [**?**, p. 7] for the CS4334 chip.

This particular filter is a 3-pole active low pass [**?**, p. 3] that has a cutoff at around 62.45kHz according to LTSpice simulation given in figure **??**.
The first two poles of the filter in figure **??** are determined by R_1, C_1, R_2, C_2 and the output impedance of the DAC chip.

The output impedance of the DAC chip was measured by using the circuit in figure **??**, when the output wave was reduced by half then the output impedance of the DAC is the same as the potentiometer value at that given time. The value that I got with this measurement was 300Ω.

With this impedance value I had all of the necessary values I needed to simulate the low pass filter in LTSpice and to plot the frequency response as shown in figure **??**.

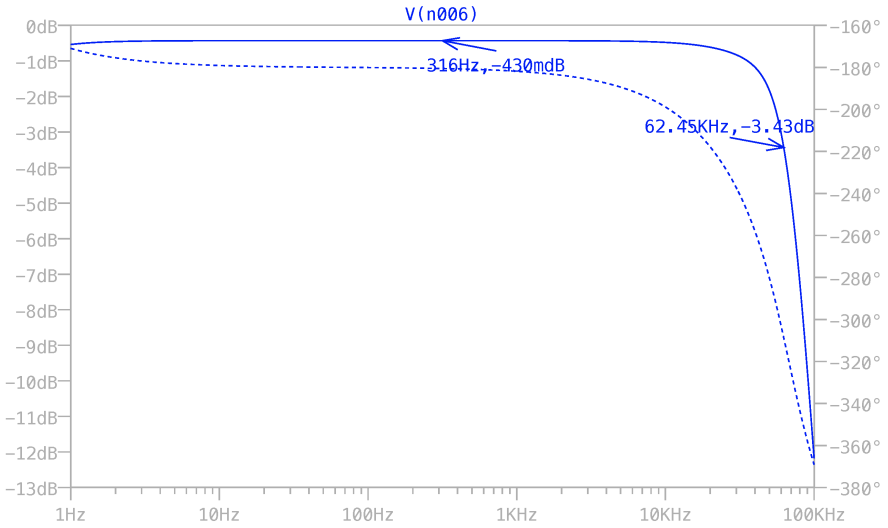

Figure 6: Bode plot of third order low pass filter

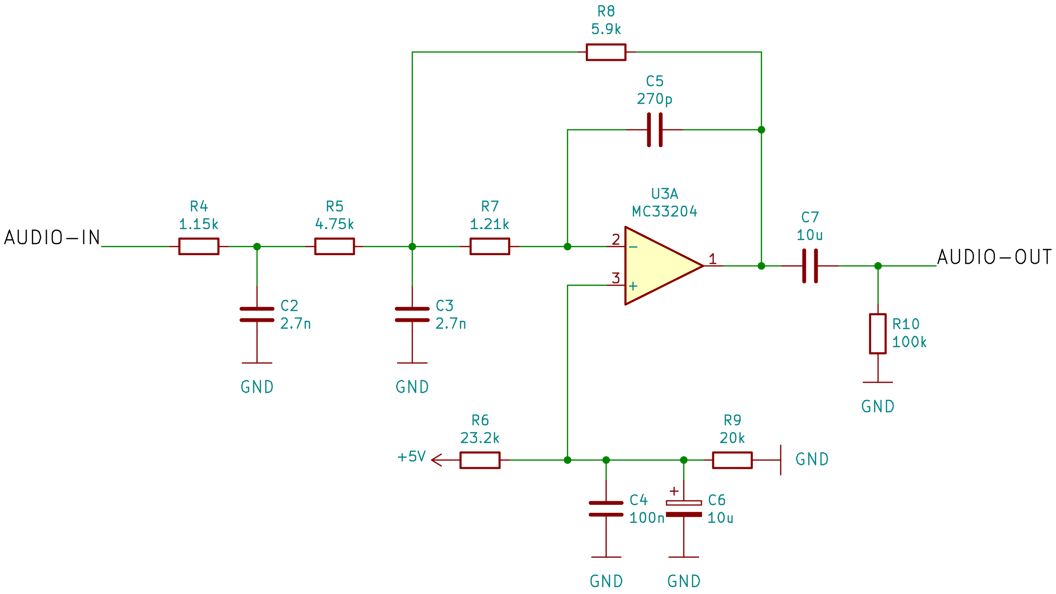

Figure 7: Schematic of the third order low pass filter

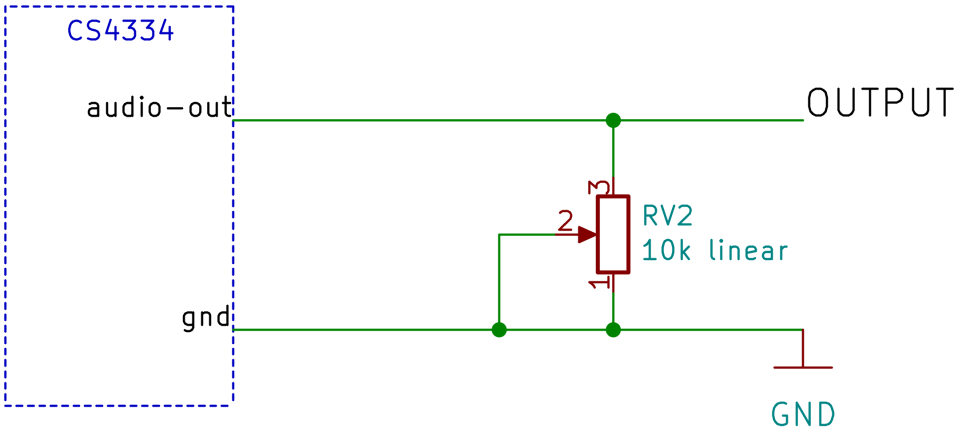

Figure 8: DAC output impedance measurement circuit

7.3.3 Problems

One of the biggest problems that I ran into while testing out this DAC and the corresponding output filter is the overshoot of a square wave output. A potential reason for this is that the switched capacitor filter in the DAC is producing this overshoot because of the sharp transient of the square wave.
The problem is shown in figure **??**. In order to address this issue partially, I applied a digital low pass filter that has a cut-off at around 15kHz. This does not correct the issue entirely as a ringing/overshoot sound can still be heard. And having a low pass filter at 15kHz severely limits the bandwidth of the output since the typical audio frequency range is up to 20kHz.

However this method does reduce the sinewave/overshoot by 1/3 of its original amplitude at approximately -9dB. In order to do this I added a Chamberlin 12dB/oct low pass filter block in the Teensy Audio Design Tool(See Chapter **??**) and set the cutoff frequency to 15kHz. The software block diagram is shown in figure **??**.

These waveforms were measured with a digital oscilloscope(Rohde und Schwarz RTB2004) which I then exported to a .csv file that I reconstructed in Matlab.

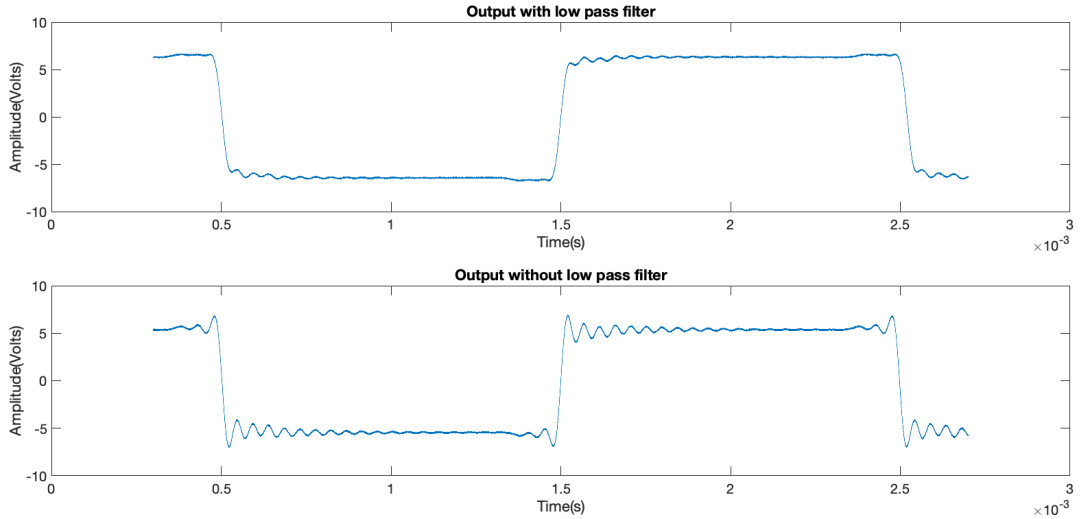

Figure 9: Square wave output with a low pass filter at 15kHz and no low pass filter

To eliminate the possibility of the Teensy audio library from being the cause of the 20kHz ringing/overshoot issue I did some analysis of the serial data that is being sent to the DAC chip with the Rohde und Schwarz oscilloscope that was capable of exporting values from a serial stream. The waveform that is being transmitted to the DAC via i^2s and the resulting waveform directly at the output of the DAC is shown in figure **??**. Note the delay from the serial data being transmitted to the DAC and the analog wave at the output, this is normal. This is explained with the total group delay(tgd) in the CS4334 datasheet [**?**, p. 7] which is $\frac{9}{F_s} = \frac{9}{44100Hz} = 204\mu seconds$. This further cements the theory that it is the DAC chip itself that is behaving this way since the actual data being transmitted is a well behaved perfect square wave.

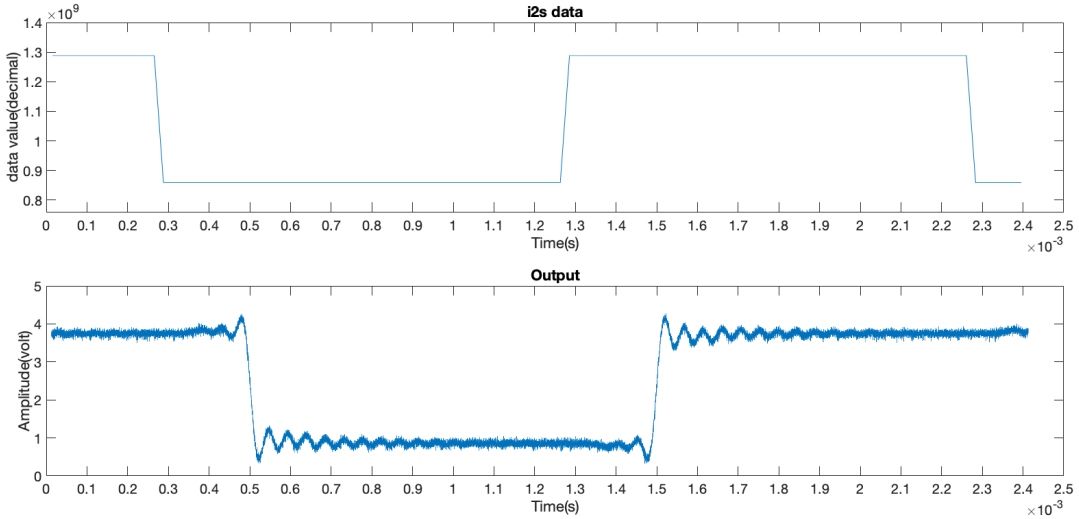

Figure 10: Serial data and the resulting DAC output with the sine wave problem

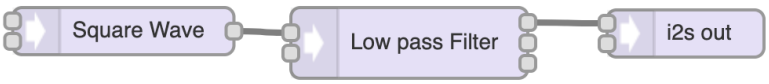

Figure 11: Digital filter with 15kHz cut-off

7.4 IO circuits

7.4.1 Control voltage inputs

For the hardware platform I wanted to have a more flexible human interface other than one rotary encoder and a pushbutton. So I opted for eight different potentiometers that can control various aspects of the algorithm/program.
I also implemented seven control-voltage inputs that are mapped to each potentiometer respectively[3].

These control-voltage inputs could let a user control a parameter inside the program by inputting an oscillator into the input that has a very low frequency which would control the same parameter as its respective potentiometer.

In figure **??** is the schematic for one input which includes a control voltage input and a potentiometer. This design is forked from a design of Émilie Gilet called 'Clouds' [**?**, p. 2]. It should be noted that the voltage rails for the op amp is from 0V to +3.3V. This is to ensure that the output of the op amp does not go above +3.3V or below 0V which would ruin the ADC of the Teensy 4.0 microcontroller.

The input circuit is a simple inverting mixing op-amp circuit with a potentiometer as an offset that mixes with the CV-CTL1 jack input. When an input of a sine wave of a peak to peak value of 5V and an offset of 2.5V is inputted, the user will get an inverted sine wave(with respects to the input) with a 3.3V peak to peak value and a 1.15V offset.

To calculate the output of this circuit I first have to define the inputs and outputs of it.

- CV-CTL1 Jack input := $U_1(t)$
- Center wiper of potentiometer := $U_2(t)$
- Output of op amp/CV-Output := $U_3(t)$

The formulas for calculating the output $U_3(t)$ if $U_2(t) = 0$ is

- $U_1(t) = (2.5V \cdot sin(2 \cdot \pi \cdot \frac{1}{t})) + 2.5V$
- Gain for this input is: $-\frac{R4}{R3} = -\frac{66.5k\Omega}{100k\Omega} = -0.665$
- The output for this input is then: $U_3(t) = -0.665 \cdot U_1(t)$

[3] I decided not to have voltage control over the potentiometer named 'attack' since I did not have room for another mini-jack connector and it is probably the unlikeliest parameter to be voltage controlled.

The formulas for calculating the output $U_3(t)$ if $U_1(t) = 0$ is

- The gain for this input is $-\frac{66.5k\Omega}{200k\Omega} = -0.3325$
- The output for this input is then: $U_3(t) = U_2(t) \cdot -0.3325 = -0.3325 \cdot U_2(t)$
- minimum value for $U_2(t)$ is -10V and maximum value is 0V. The graph of $U_2(t)$ is trivial; just a straight line from minimum to maximum with a slope depending on how fast the user turns the potentiometer.

Now I superimpose those two formulas and get

- $U_3(t) = (-0.665 \cdot U_1(t)) + (-0.3325 \cdot U_2(t))$
- With $U_3(t)$ being limited between 0V and 3.3V

An oscilloscope screenshot of $U_1(t)$ and $U_3(t)$ if $U_2(t) = -10V$ can be seen in figure **??**.

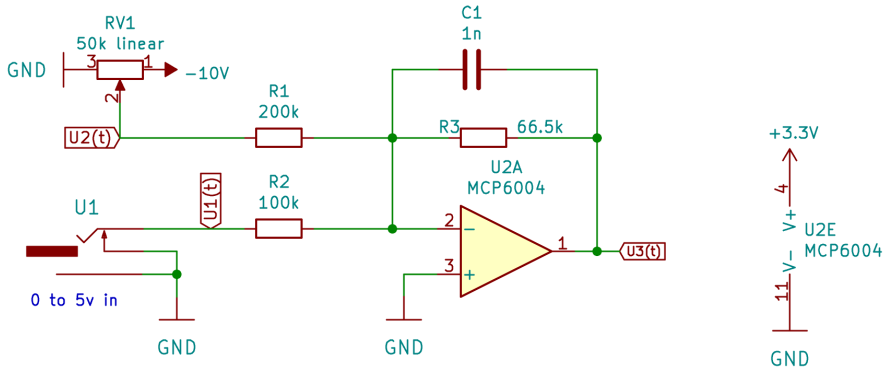

Figure 12: CV input circuit

I did run into minor problems with noise on the output of this circuit and I have yet to solve it properly other than in the firmware by reducing the ADC reading from 12 bits to 11 bits. A couple of theories for reducing this noise are listed below:

- Supply the op-amp with a different +3.3V supply than the one supplied by the Teensy
- Have a special filtered analog ground for the op amp that is seperated from the digital circuitry
- Redesign the PCB layout with low noise considerations such as better ground layout and shorter traces

I did not have the time to test these theories but I will in the future.

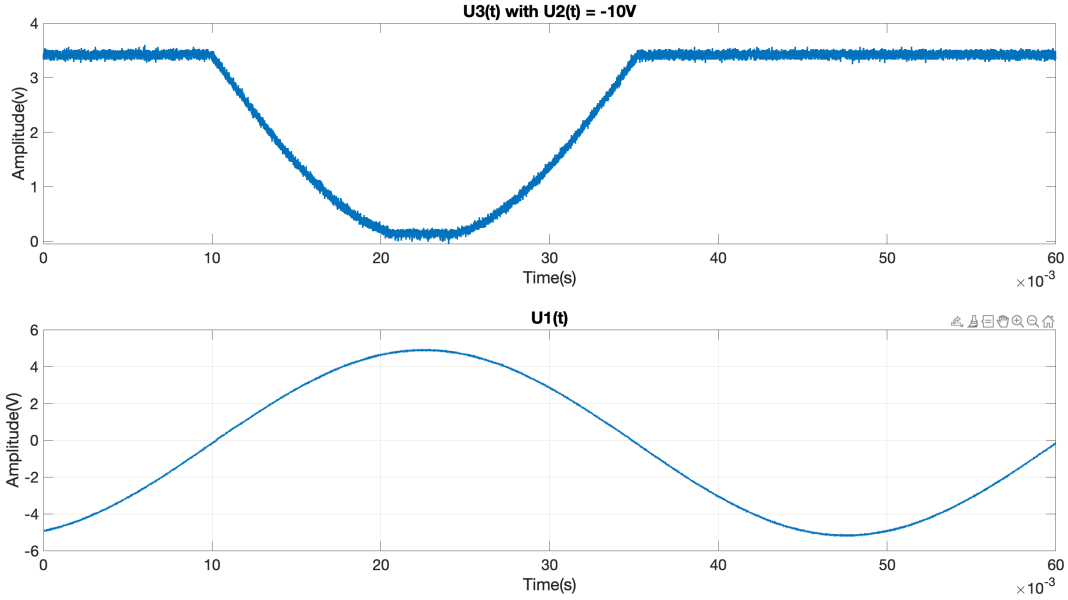

Figure 13: Scope picture of U_3 and U_1

7.4.2 Voltage amplifier output

Since the CS4334 DAC has an output amplitude of around 3.5V peak to peak [**?**, p 7], I had to amplify it to get it to the 10V peak to peak Eurorack standard levels [**?**, Paragraph 6]. I did this with a simple non-inverting voltage amplifier that has variable gain. The gain G for this amplifier is:

- $Gain(G) = 1 + \frac{R_{V1}}{R_1}$ with $R_1 = 1k\Omega$ and $RV1$ min $= 0\Omega$ and max $= 10\text{k}\Omega$
- Maximum gain $= 11$, minimum gain $= 1$
- So in order to get the 10V peak to peak the gain has to be $= \frac{10V}{3.5V} = 2.85$
- Then $R_{V1} = (2.85 - 1) \cdot R_1 = 1.85k\Omega$

The gain can vary since the internal DSP is not always outputing its maximum peak to peak all the time and often times I had to change amplitudes within the program to prevent internal DSP clipping. This is why I decided to have a variable resistor to change the gain since it was the most flexible way to match the voltage levels to the Eurorack standard.

The reason for R_2 is to prevent the op-amp from being damaged if the output audio jack is grounded, with this resistor the high voltage audio signal is dissipated over this resistor instead of the op-amp. R_2 is also used to "set" the output impedance which is measured in Chapter **??**.

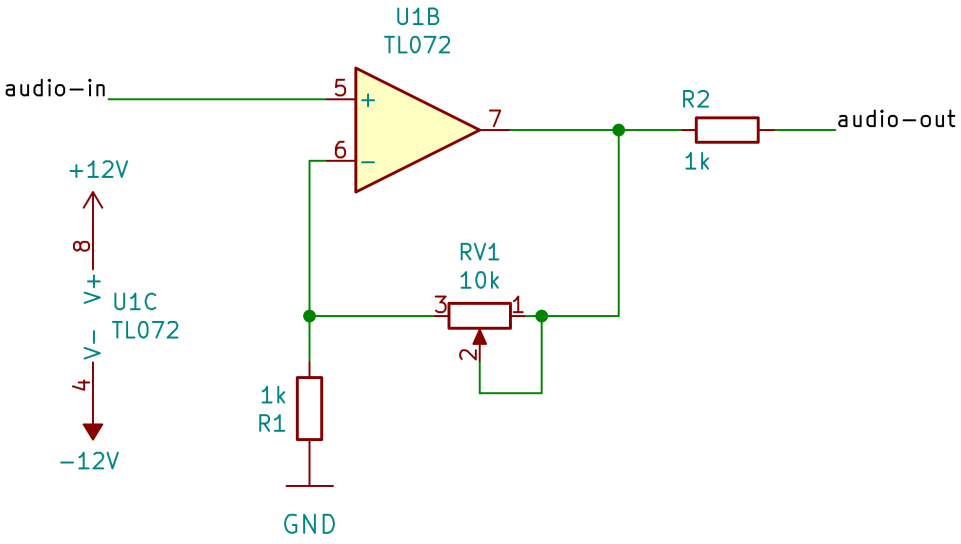

Figure 14: Voltage amplifier

7.4.3 1-bit gate output

My design also includes a digital 1-bit output. The reason for this output is so that when the microcontroller receives a musical note value from the MIDI receiver it will also send a +5V gate pulse to the GATE-OUT jack connector. This can be used to trigger certain events within another device with a gate input.

The way this works is the DIGITAL IN pin is an active LOW input, so when the pin is LOW the transistor will not conduct and R_{12} will pull up the output thus resulting in +5V at the GATE-OUT. When the DIGITAL IN pin goes HIGH the transistor will conduct thus pulling DOWN the GATE OUT resulting in 0V at the jack output.

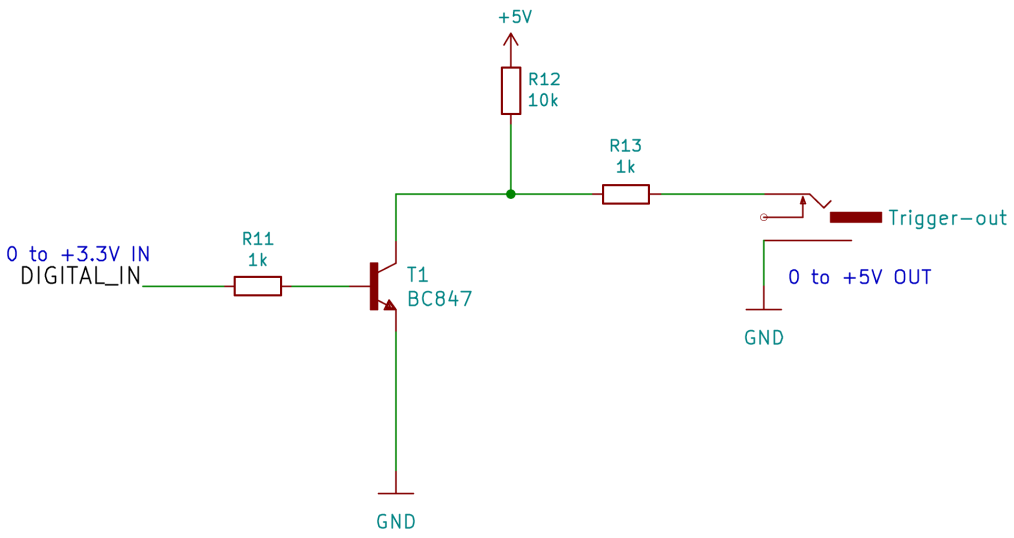

Figure 15: Gate output circuit

7.5 OLED

The OLED that I chose is a 1.3" 128x64 OLED that is manufactured by Adafruit, the reason for doing so is that this OLED has alot of support and documentation [**?**] from Adafruit. It has a driver chip named SSD1306 which can be communicated with using either i^2c or SPI [**?**]. It can be used with either 3.3V or 5V devices, in my case the Teensy 4.0 is a 3.3V device. Adafruit have designed this OLED with an auto-reset circuit [**?**] so in my hardware design I didn't have to connect the reset pin of the OLED.

The driver for this OLED also has a lot of software support and libraries for showing graphics and more, so it was an obvious choice for easily getting a graphical User-Interface on the screen. The OLED screen with a UI that I designed can be seen in figure **??**.

The driver for this OLED calls for 4 data pins for it to be able to work with SPI:

- MOSI, Data or Master Out Slave In. This is the data that the Master(Teensy) is sending to the Slave(OLED). For the SSD1306 it's an 8 bit register that can either hold 8 bits of (visual) data or 8 bit command data [**?**, p. 14].
- D/C or Data/Command. This is to tell the SSD1306 chip whether it's receiving data(D/C = 1) or a command(D/C = 0) [**?**, p. 14].
- CS or Chip Select, this to either enable or disable the chip [**?**, p. 13].
- CL, CLK or Clock, this is the serial clock for the chip [**?**, p. 13].

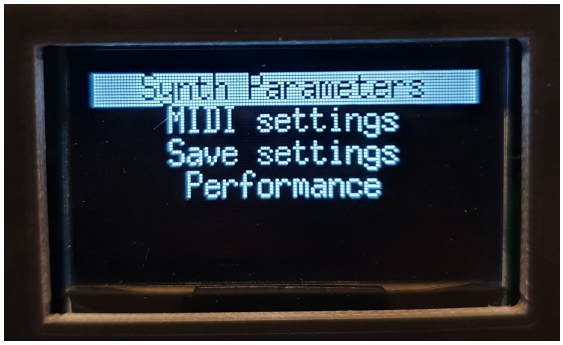

Figure 16: OLED with current user interface

7.6 MIDI 1.0 receiving and buffering

Musical Instrument Digital Interface (MIDI) is a standard for transmitting various musical information between musical devices. A musician can for example have a 49 note MIDI keyboard that can send MIDI data via its MIDI OUT port that contains the note the musician just hit on the keyboard, whether they just pushed the note or released it(Note on or off) and at what velocity the musician pushed the note. Other control data such as keyboard aftertouch can also be interpreted through MIDI. MIDI was standardized in 1983 by the MIDI Manufacturers Association [**?**].

It has become the most common way to send all kinds of information between musical instruments and control surfaces.

MIDI is an asynchronous serial data interface that has a baud rate of 31250, with 1 start bit, 8 data bits and one stop bit [**?**, p. 2]. The MIDI interface is a 5mA current loop that has a logical zero when the current is on [**?**, p.3]. With the MIDI interface, there is a requirement to isolate the ground between the transmitter and receiver to avoid ground loops which would result in high bit-rate errors due to erroneous currents [**?**, p. 3].

Originally the MIDI interface was designed for a +5V TTL logic standard but it has been updated to include a +3.3V logic standard [**?**, p. 4], because of the development of CMOS technology which can have lower voltage levels [**?**].

The only thing that needs to be changed is a couple of resistors for the MIDI output circuit.

This circuit is designed according to the recommended circuit in the MIDI specification document [**?**, p. 3]. This is including the optional MIDI THRU circuit, the optional EMI noise suppression ferrite beads that limit the current at 100Mhz which will reduce the amplitude of high frequency signals at around 100Mhz [**?**, p.6] and an optional capacitor that improves high frequency shielding by shorting high frequencies(10MHz) to ground [**?**, p.6].

The circuit for receiving a MIDI signal and buffering it to another output is shown in figure **??**, additionally the recommended circuit from MIDI manufacturers association is also shown in figure **??**.

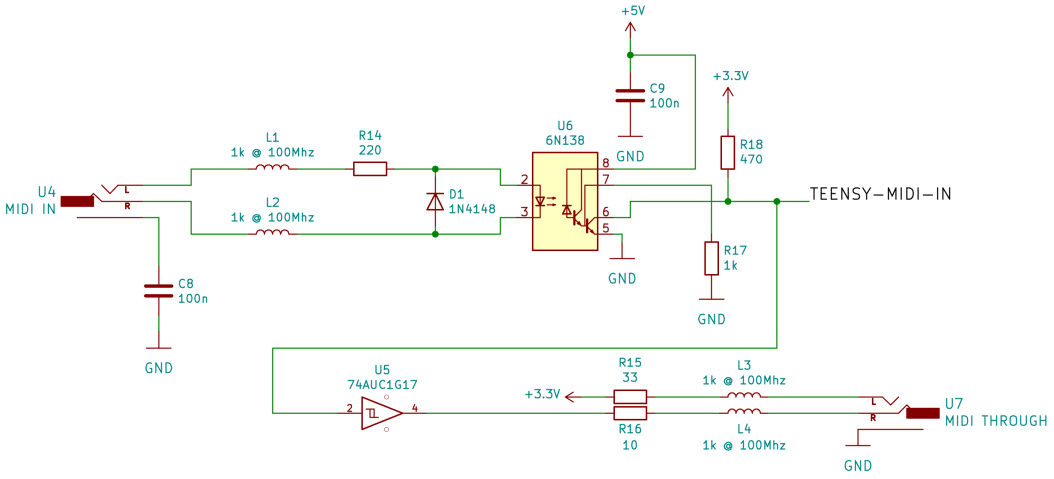

Figure 17: MIDI in and out circuit in my hardware platform

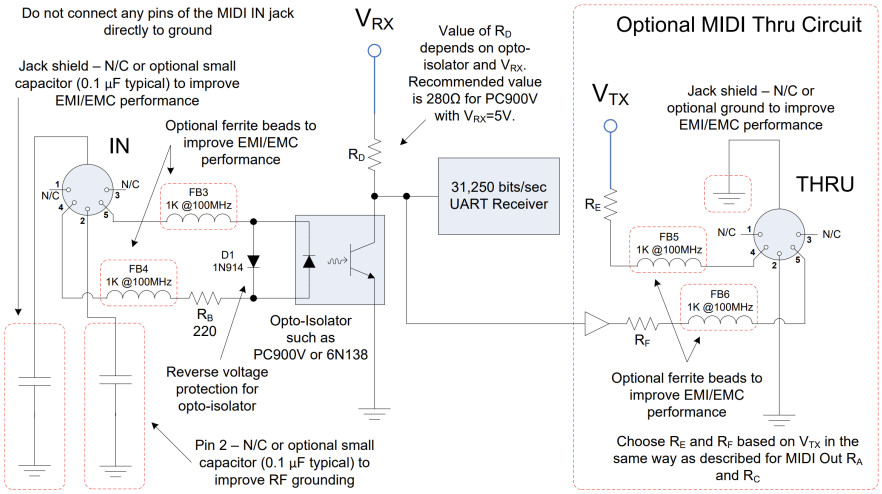

Figure 18: Recommended MIDI in and out circuit

I decided to use a 6N138 optocoupler since I've used it for previous designs and I have confirmed it to work in previous projects. In the 6N138 optocoupler, when there is current going through the LED at pin 2 and 3, the first transistor of the darlington pair starts conducting. Then the second transistor will be conducting and thus producing a logical zero at the output pin.
Since the Teensy is a 3.3V device I put a pullup resistor, R_{15}, connected to +3.3V at the output. The output is also buffered via a non inverting Schmitt trigger and then the output of the Schmitt trigger is routed to the MIDI THRU output jack.

7.7 Analog input and output characteristics

7.7.1 Analog input impedance

The analog input impedance value is guaranteed by theory. Since the control voltage input is an inverting operational amplifier circuit it can be assumed that the input impedance for CV-IN is 100kΩ. This is because the inverting input of the op-amp is a virtual ground [**?**].

I confirmed the input impedance value by putting a 100kΩ resistor in series with the 100kΩ input resistor and then measuring at point A as is depicted in **??**. I then checked if the voltage at point A was half of U1 (t), it was. This confirmed that the input impedance of the circuit was 100kΩ.

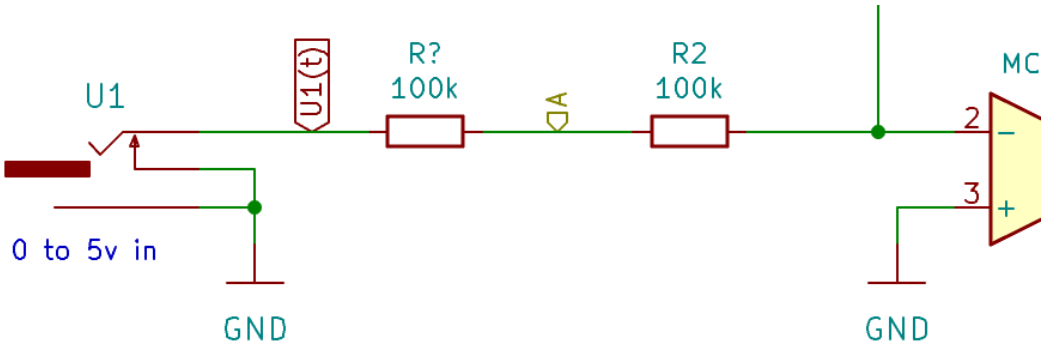

Figure 19: Input impedance measurement circuit

7.7.2 Analog output impedance

To measure the analog output impedance of the amplifier from Chapter **??**, I used the same method as in Chapter **??**.

The method is that the impedance output of the amplifier is seen as a resistor in series from the viewpoint of another circuit. So if the output impedance of this circuit and the input impedance of another circuit is the same then the output wave is halved since they form a voltage divider. In the case of this circuit I connected a potentiometer that is connected at the output and to ground. I then tuned this potentiometer until the output was half of what it was before I connected the potentiometer. The value that I got from this measurement method was 1kΩ.

7.7.3 THD+N of audio output

The definition of the THD+N value is the ratio between the RMS value of a signal and the RMS value of all other frequency components, including distortion, between the frequency 10Hz to 20kHz [**?**, p.22].
To calculate the THD+N value of the output of my hardware platform I generated a sine wave at 500Hz and did an FFT analysis with a Rode Und Schwarz RTB2004 oscilloscope and exported the data from that analysis to a .csv file. Then I imported the data into Matlab, got the RMS value for the sine wave and calculated the mean value of the underlying noise and distortion and simply subtracted the sine wave's amplitude from the mean value of the noise and distortion.

- $ND_{mean}(dbmV)$ = mean of all noise and distortion from 0Hz to 20kHz = -62.7455dBmV
- $Signal_{mean}(dbmV)$ = mean of sine wave signal = 26.0331dBmV
- THD+N = $ND_{mean} - Signal_{mean} = -62.7455dBmV - 26.0331dBmV = -88.7786dBmV$

From this I can say that the THD+N value is -88.7786dB. This is around the same as the datasheet states for the CS4334 chip [**?**, p. 6]. For Base-Rate-Mode the THD+N value is -88dB. The FFT measurement from the oscilloscope can be seen in figure **??**. With the data point for the 500hz sinewave marked.

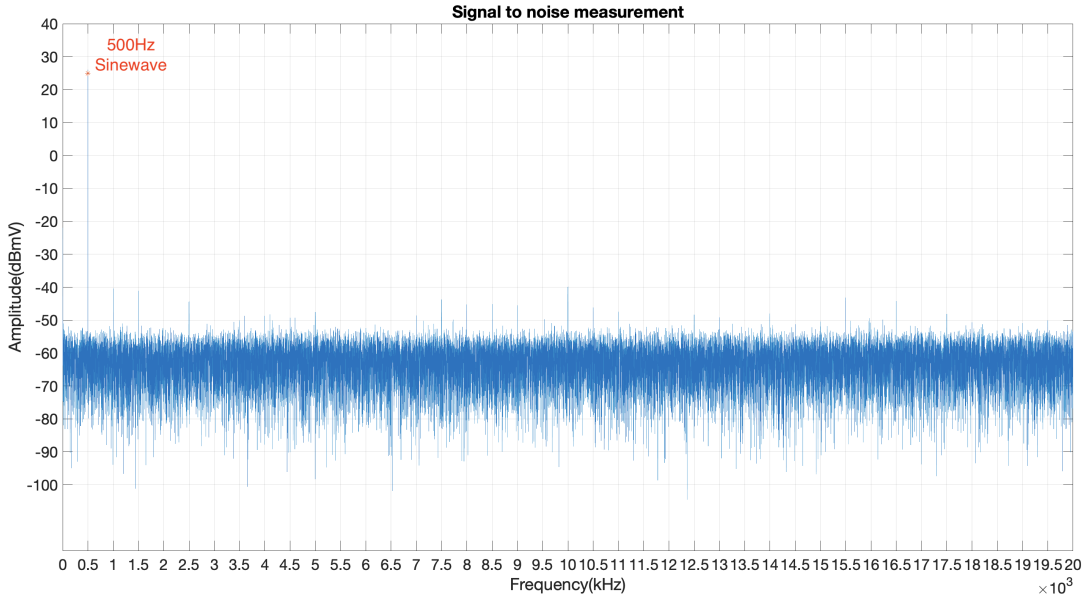

Figure 20: FFT analysis of the output with a 500Hz sinewave

8 Phase Modulation Synthesis

In this chapter I will cover the calculation and simulation of PM synthesis using Matlab and compare it to real life measurements of my hardware platform using an oscilloscope and a spectrum analyzer. There are many more applications for PM synthesis that my study will discuss. I will only cover one basic algorithm that can, despite not being that complicated, deliver quite complicated harmonic tones and interesting sonic results.

With PM synthesis there are some common words and concepts that are very important to know to be able to understand it. The most common words are carrier and modulator. Carrier and modulator are often referred as operators in the musical synthesis world. The basis for PM synthesis is a carrier being phase modulated by some sort of modulator and that modulator could also be phase modulated by another modulator. This relationship has resulted in users and designers referring to the each unit, either carrier or modulator, as operators.

I will take the three operator system in figure **??** as an example. The carrier is one operator and the two modulators are the other two operators. Every operator has its own parameters for their amplitude and frequency. The operator that has an audio output has the fundamental frequency f1 and the other operators have a frequency which is a multiple or fraction of the fundamental frequency f1.

In this study, I only calculated, simulated and measured PM synthesis with sine waves since doing square, triangle and sawtooth waveshapes would increase the amount of calculations and measurements fourfold.

The most common implementation is to only have sine waves for this type of synthesis but in my application in Chapter **??** of this synthesis I will have another parameter that can change the waveshapes. The parameters of each block are defined below:

- f_1 := operator 1 frequency
- α := amplitude of operator 2
- β := amplitude of operator 3
- γ := frequency ratio between frequency of operator 2 and operator 1 or $f_2 = f_1 \cdot \gamma$
- δ := frequency ratio between frequency of operator 3 and operator 1 or $f_3 = f_1 \cdot \delta$

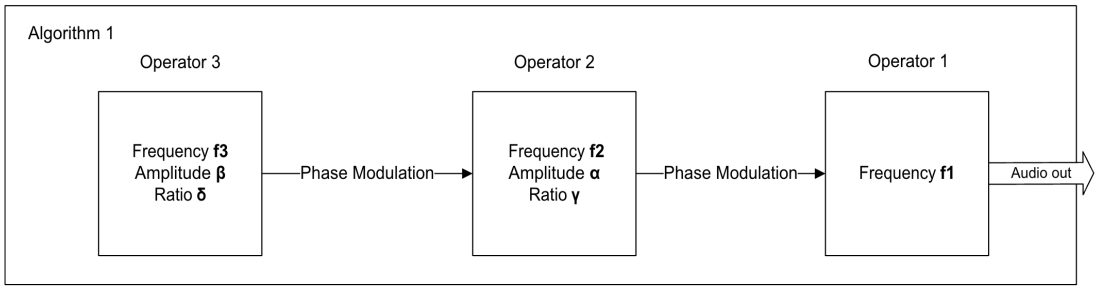

Figure 21: Block diagram of a common 3 operator phase modulation algorithm

8.1 Calculation

One of the first steps required for understanding PM synthesis is understanding the very simple calculations that are done to achieve this synthesis. First I will define the formulas used for each operator of algorithm 1 as shown in figure **??**.

- Sine wave standard form: $sin(2 \cdot \pi \cdot t + \phi)$ with $\phi = phase$
- $t_1 = \dfrac{1}{f_1}$
- operator 3 $= op_3(t) = sin(2 \cdot \pi \cdot \dfrac{t_1}{\delta} + \phi_3)$
- operator 2 $= op_2(t) = sin(2 \cdot \pi \cdot \dfrac{t_1}{\gamma} + \phi_2)$
- operator 1 $= op_1(t) = sin(2 \cdot \pi \cdot t_1 + \phi_1)$

Now the next step is very simple, I just replace each ϕ for each operator with its respective phase modulator.

- $\phi_3 = 0$ (nothing modulating phase of operator 3)
- $\phi_2 = \pi \cdot op_3(t) \cdot \beta$ (operator 3 modulating phase of operator 2)
- $\phi_1 = \pi \cdot op_2(t) \cdot \alpha$ (operator 2 modulating phase of operator 1)
- $op_3(t) = sin(2 \cdot \pi \cdot \dfrac{t_1}{\delta})$
- $op_2(t) = sin(2 \cdot \pi \cdot \dfrac{t_1}{\gamma} + \pi \cdot op_3(t) \cdot \beta)$
- $op_1(t) = sin(2 \cdot \pi \cdot t_1 + \pi \cdot op_2(t) \cdot \alpha)$

8.2 Simulation

Now I will show some minor Matlab code and figures from Matlab to show both a waveform generated with PM synthesis and the spectral analysis of that waveform. I will include 6 waveforms and FFT analyses which showcase how an audio engineer can start with a basic sine wave and generate very complex musical waveforms.

A three dimensional plot of all possible α values between 0 and 1 can be seen in figures **??**, **??**, **??**, **??**, **??** and **??**. This gives a good picture of how operator 1 is 'distorted' or 'morphed' by operator 2. When β is bigger than zero, operator 1 carries frequency components from operator 3. Operator 1 waveform and its FFT analysis with $\alpha = 0.5$ can be seen in figure **??** with a fundamental frequency of 500Hz, harmonic frequency at 1500Hz and 2500Hz. In figure **??** operator 1 resembles a square wave. These overtones match how the waveform looks like because in a Fourier Series expansion a square wave is simply an odd multiplication of the fundamental sine wave and then added together [**?**].

One very important aspect of PM synthesis is the ratio between the operators. When the ratio is an even number the upper harmonics are very pleasing to the ear but when the ratio is an odd number or a fraction other than 0.5 the upper harmonics produce dissonant sounds. The dissonant sounds are great for creating bell like sounds and other atonal sounds. The variation of different sounds that can be achieved is very large and as can be shown in the figures mentioned above, the various waveforms that can be generated are very interesting in both appearance and sound.

8.3 Example code

In the example code included in the Appendix, I have shown the calculation itself to compute the functions and the plot it produced in Matlab. Additional labeling and scaling of the axes will be left to the reader. This example code with additional labeling was uploaded to the same Github repository as previously mentioned in Chapter **??**.

In listing **??**, I had to invert the op1(t) function to directly compare it to the output of my hardware platform. This is because the low pass filter in Chapter **??** has an inverting operational amplifier configuration. This does not change the sound in any way but simply changes how the waveform is visualized.

8.4 Spectral analysis and waveforms

This chapter subsection will show the same waveforms that are shown in Chapter **??**. The waveforms were synthesized on my hardware platform and not simulated. As can be seen in figures **??** to **??** the output of the hardware platform and the MATLAB simulation are the same as far as the waveshape itself goes.

The frequency spectrum analysis done by the Rode und Schwarz oscilloscope showed approximately the same results as the FFT analysis in Matlab in Chapter **??**. The oscilloscope frequency analysis shows the noise that is associated with the hardware platform but overall the frequency analysis is around the same as the Matlab simulation.
It is also worth noting that the frequency analysis done by the oscilloscope uses dBmV measurement whereas the simulation simply deals with linear amplitudes. But overall it still shows the relevant information such as higher order harmonics.

9 Software

The software for demonstrating PM synthesis is not an overly complicated process. However, other factors such as receiving MIDI notes and controlling different aspects of the synthesis algorithm can get pretty large. In this chapter I will mainly outline the biggest factors of the software itself and I will have some minor example code.
The whole code for the application of PM synthesis in Chapter **??** can be seen on my Github. The Github link can be found in Chapter **??**.

9.1 Teensy Audio Design Tool

The first step of designing and programming for an audio application with a Teensy microcontroller is to start up the Teensy Audio Design Tool designed by PJRC [**?**]. This software enables the programmer to set up and design the architecture of the audio application and generate a setup code that would otherwise be very cumbersome and time consuming to write. A screenshot of the current system architecture for demonstrating PM synthesis is shown in figure **??**.

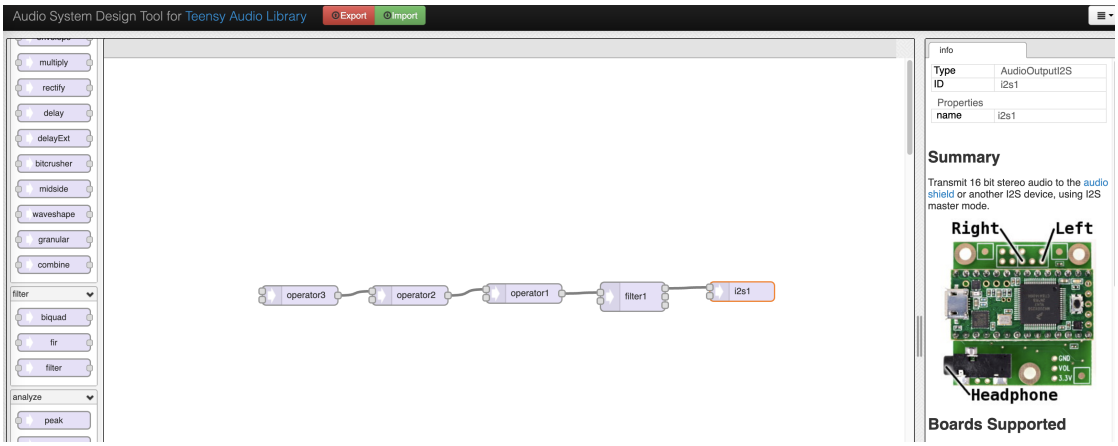

Figure 22: Teensy Audio Design online tool

The resulting pre-configured code using this Audio Design Tool is shown in listing **??**. It instantiates the objects needed for this architecture and then it makes the necessary audio connections. This is then copied into the relevant Arduino project.

9.2 Arduino

In this chapter I will briefly go over the Arduino IDE tool that I used for compiling code and what libraries I used for simulating phase modulation synthesis.

As I discussed in Chapter **??** the design uses a Teensy 4.0 microcontroller and it can be programmed with Arduino IDE with an addon called Teensyduino made by PJRC [**?**].

Arduino is a Java based IDE that implements the Processing/Wiring language [**?**]. The programming itself is done in the C++ language and the file extensions all end with .ino. The most important aspect of programming with Arduino is the huge plethora of code libraries that are available.
The libraries that I used mainly for the implementation of PM synthesis and interfacing with the hardware platform are:

- 'ResponsiveAnalogRead' for reading and smoothing out noise from ADC readings
- 'Audio' for implementing the Teensy Audio Library. This library is neccessary for the Teensy Audio Design tool.
- 'SPI' for interfacing with the OLED via SPI
- 'u8g2' for sending data and graphics to the OLED
- 'EEPROM' for storing settings data and retrieving it again after a reboot of the hardware plaftorm
- 'MIDI' for receiving and sending MIDI messages.
- 'Bounce2' for debouncing button inputs and detecting rising and falling edges

I found that the code editor in the Arduino IDE was severely limiting so I started using Sublimetext to edit all of the .ino files for the project and only used Arduino IDE for compiling and uploading the code to the Teensy using the Teensy Loader [**?**].

9.3 Handling MIDI notes

In this chapter I will go over the algorithm that is receiving and allocating notes to the different voices. There are four voices total and each voice receives a note value and a note on or off message. Using the MIDI Arduino library the programmer can assign functions to the interrupt message tied to receiving a MIDI message. This is done using the lines shown in listing 3. The method that I chose for voice allocation is called 'piano mode' [**?**]. If the musician pushes a note then that note gets assigned to a voice, such as voice 1. Then the next note ON message that is received is allocated to voice 2 and so on. But if the musician releases the first note before he/she/they push the third then the third note does not get assigned to voice 1, it is assigned to voice 3.

Now let's say that all 4 voices allocated to their individual notes, if the musician pushes a 5th note then that 5th note takes over the 'oldest' note that is assigned to a voice, so it steals the note.

In the program there is a history array that keeps track of what the oldest assigned voice is. One array that keeps track of what voice is playing what note and the third array contains information whether that voice is playing or not. When the program receives a note OFF message, it finds a voice that is playing and has the same note as the note OFF message. It then silences that particular voice. The code for the voice allocation algorithm can be found in listing **??**.

10 Application: Fjöl synthesizer

The final application of phase modulated synthesis can be demonstrated with the hardware platform I designed, named Fjöl, that is designed for the Eurorack synth format designed by Dieter Doepfer [**?**] [**?**]. The frontpanel can be seen in figure **??**. A link to the Github for Fjöl can be found in the footnotes[4]

Figure 23: Fjöl front panel

[4] https://github.com/hallmar/Eurorack-Modules/tree/master/Fj%C3%B6l

The Fjöl synthesizer can be controlled via MIDI serial commands and various parameters can be controlled by either the potentiometers on the front, the rotary encoder or via a control voltage input.

The end product is anticipated to have a lot of features which are yet to be programmed fully, but most of the core functionality has been programmed and published in my Github. The list of core features is:

- 4 note/voice polyphony
- MIDI in and buffered out(passthrough)
- 3 operator PM synthesis algorithm per voice with 4 different waveforms for each operator
- 1 sub-octave PWM oscillator per voice
- 1 extra oscillator per voice with 4 different waveforms
- 6 knobs attached to a specific function within the module
- 2 knobs freely assignable to a select set of functions within the module
- Spread the 4 voices across the stereo field with the -4.5dB pan law [**?**]

The architecture per voice for this synthesizer is shown in figure **??**.

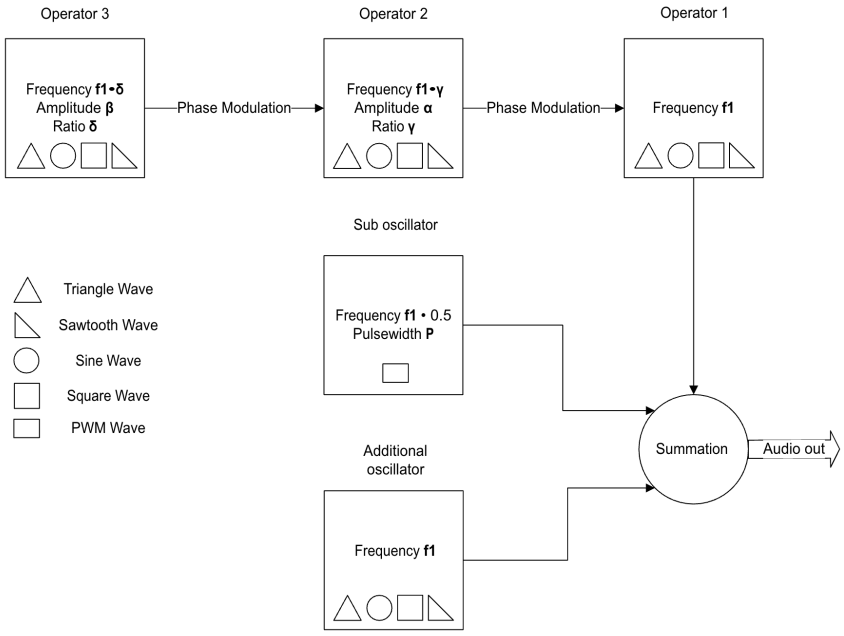

Figure 24: Fjöl voice architecture

11 Conclusion

In conclusion the hardware part of this thesis has produced satisfactory results based on my goals for this project. The THD+N is at around -88dB which is satisfactory but other DAC chips do have better THD+N performance, this will improve when I replace the current DAC chip with another as mentioned later in this chapter. The user interface works well albeit it being complicated to program, the voltage conditioners work as I expected despite minor noise issues which reduce the ADC resolution and the Teensy performs very well.

Based on my initial designs and expected limitations of the hardware, it was surprising that the Teensy could handle interfacing with an OLED, receiving MIDI data, handling eight different ADC calculations and synthesize audio, but with a 720Mhz overclocked ARM processor it did the job very well.

But the platform is not completely free of problems, the main problem is the overshoot described in chapter **??**. This to me is unacceptable since I can clearly hear the overshoot problem with and without the digital filter. I will probably have to replace the current DAC with another. A potential solution could be the PCM5100 from Texas Instruments which has an build in de-emphasis filter for 44.1kHz sample rates [**?**, p. 3] and it also has a lower THD+N value of -93dB [**?**, p.5] than the CS4334. The CS4334 does not have a de-emphasis filter with suitable bandwidth at 44.1kHz sample rate [**?**, p. 14]. I will most likely also have to change the low pass filter cutoff frequency to 20kHz at the output to further reduce the sharp transients of these waves.

The simulation of phase modulated synthesis went very well despite being quite time intensive to program in Matlab. In addition the synthesized waveforms using my platform came out exactly the same as the waveforms that I simulated in Matlab. Although there was some minor difference from the FFT analysis outputs. The main difference was the accuracy difference. This is most likely because the oscilloscope took a very long sample window of maybe around 100 periods and analyzed the frequency spectrum from that, whereas the Matlab program that I wrote only took a sample window of around 10 periods.

Largely I achieved what I wanted to with this thesis. I explained and simulated how to synthesize PM synthesis while qualifying my simulations using a hardware platform of my own design. This thesis is meant to be a helpful starting point for future sound engineers researching PM synthesis methods, simulation and hardware design and it also achieved in doing so.

References

[1] 120 years of electronic musical instruments. http://120years.net/.

[2] Adafruit. OLED screen. https://www.adafruit.com/product/938.

[3] Arduino. Arduino IDE Github. https://github.com/arduino/Arduino/tree/1.8.11.

[4] MIDI Manufacturers Association. MIDI standard technical document. https://mitxela.com/other/ca33.pdf.

[5] Uwe Beis. An Introduction to Delta Sigma modulation. https://www.beis.de/Elektronik/DeltaSigma/DeltaSigma.html.

[6] John Chowning. The synthesis of complex audio spectra by means of frequency modulation. https://www.aes.org/e-lib/browse.cfm?elib=1954, 1973.

[7] Sequential Circuits. Sequential Circuits Prophet 5 reissue. https://www.sequential.com/product/prophet-5/.

[8] Dieter Doepfer. Doepfer construction details of Eurorack standard. http://www.doepfer.de/a100_man/a100m_e.htm.

[9] Dieter Doepfer. Doepfer technical details of Eurorack standard. http://www.doepfer.de/a100_man/a100t_e.html.

[10] E-RM. Polygogo synthesizer oscillator using Polygonal synthesis. https://www.e-rm.de/polygogo/.

[11] Noise Engineering. Cursus Iteritas oscillator. https://www.noiseengineering.us/shop/cursus-iteritas.

[12] Peter Frith and Wolfson MIcroelectronics. Switched capacitor DAC. https://patentimages.storage.googleapis.com/b3/95/d5/4ea122f2fcd8bd/US7102557.pdf.

[13] Émilie Gilet. Clouds module. https://mutable-instruments.net/modules/clouds/downloads/clouds_v30.pdf.

[14] Émilie Gilet. Voice allocation algorithm description. https://synth-diy.org/pipermail/synth-diy/2014-June/044044.html.

[15] Xavier Hosxe. PreenFM3 FM synthesizer. https://github.com/Ixox/preenfm3/wiki/Quickstart.

[16] Hyperphysics. Sound synthesis methods. http://hyperphysics.phy-astr.gsu.edu/hbase/Audio/synth.html.

[17] Texas Instruments. PCM5100 DAC. https://www.ti.com/lit/ds/symlink/pcm5100.pdf?ts=1613814145446&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FPCM5100.

[18] Robert Keim. Understanding the virtual short in op amp circuits. https://www.allaboutcircuits.com/technical-articles/understanding-the-virtual-short-in-op-amp-circuits/.

[19] Peter Kirn. Reissued Korg synthesizers. https://cdm.link/2021/01/all-the-korg-synth-news-mini-arp-modular-a-reissued-classic-more-wavetables/.

[20] Korg. Korg Opsix FM synthesizer. https://www.korg.com/us/products/synthesizers/opsix/.

[21] Cirrus Logic. CS4334 datasheet. https://statics.cirrus.com/pubs/proDatasheet/CS4334-5-8-9_F7.pdf.

[22] Cirrus Logic. CS4334 evaluation board. https://statics.cirrus.com/pubs/rdDatasheet/cs4334eb.pdf.

[23] Wolfram Mathworld. Fourier Series article. https://mathworld.wolfram.com/FourierSeries.html.

[24] PJRC Electronic projects. PJRC teensyduino addon. https://www.pjrc.com/teensy/td_download.html.

[25] PJRC Electronic projects. Teensy 4.0. https://www.pjrc.com/store/teensy40.html.

[26] PJRC Electronic projects. Teensy audio design tool. https://www.pjrc.com/teensy/gui/.

[27] PJRC Electronic projects. Teensy uploader. https://www.pjrc.com/teensy/loader.html.

[28] Steve Rickinson. Polivoks synthesizer reiussued. https://www.deephouseamsterdam.com/classic-soviet-era-polivok-synth-reissued/.

[29] Philips Semiconductors. i^2s serial standard. https://web.archive.org/web/20070102004400/http://www.nxp.com/acrobat_download/various/I2SBUS.pdf.

[30] Gianpaolo Evangelista Sergio Cavaliere and Aldo Piccialli. Synthesis by phase modulation and its implementation in hardware. https://www.jstor.org/stable/3679835?seq=1, 1988.

[31] Electro Smith. Daisy Seed, DSP microcontroller platform. https://www.electro-smith.com/daisy/daisy.

[32] Sparkfun. CMOS logic levels. https://learn.sparkfun.com/tutorials/logic-levels/all#33-v-cmos-logic-levels.

[33] Vector synth. Vector synthesizer. https://www.vectorsynth.com/.

[34] Solomon Systech. SSD1306 Chip datasheet. https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf.

[35] Carnegie Mellon University. Loudness concepts and panning laws. https://www.cs.cmu.edu/~music/icm-online/readings/panlaws/.

[36] Wikipedia. DX7 synthesizer. https://en.wikipedia.org/wiki/Yamaha_DX7.

Appendix

Listing 1: Audio design tool export

```
// GUItool: begin automatically generated code
AudioSynthWaveformModulated operator3; //initiate operator3
AudioSynthWaveformModulated operator2;  //initiate operator2
AudioSynthWaveformModulated operator1;   //initiate operator1
AudioFilterStateVariable filter1;      //initiate filter
AudioOutputI2S           i2s1;          //initiate i2s output to CD4334 chip
//In the next lines all of the connections between the objects or audio blocks are made
AudioConnection          patchCord1(operator3, 0, operator2, 0); //connect output of operator 3 to phase modulation input of operator 2
AudioConnection          patchCord4(operator2, 0, operator1, 0); //connect output of operator 2 to phase modulation input of operator 1
AudioConnection          patchCord6(operator1, 0, filter1, 0);
AudioConnection          patchCord9(filter1, 0, i2s1, 0);
// GUItool: end automatically generated code
```

Listing 2: Matlab code

```
fs = 96000; %set sample rate
t = (0:1/fs:0.004-1/fs); %create time vector with sample rate
f1 = 500; %operator 1 frequency
%Operator 3
beta = 0.5; %amplitude operator 3
delta = 5;
f3 = delta*f1;
op3 = sin(2*pi*f3*t);
%Operator 2
alpha = 0.5; %amplitude operator 2
gamma = 2;
f2 = gamma*f1;
op2 = sin(2*pi*f2*t+pi*op3*beta);
%Operator 1
op1 = sin(2*pi*f1*t + pi*op2*alpha);
%Plot operator 1
plot(t,-op1,'LineWidth',5);
%Calculate FFT of operator 1
y = fft(op1, 20000); %do a discrete fast fourier transform with 20000 points
f = (0:length(y)-1)*fs/length(y); %make an f vector
y_abs = abs(y); %scale y so that amplitude of 1 in t domain corresponds to 1 in f domain
y_max = max(y_abs);
y_scale = y_abs/y_max;
plot(f,y_scale, 'LineWidth',3); %plot fourier transform
```

Listing 3: MIDI interrupt assignment

```
MIDI.setHandleNoteOff(handleMIDIoff); //midi note off interrupt routine
MIDI.setHandleNoteOn(handleMIDIon); //midi note on interrupt routine
MIDI.setHandleControlChange(controlchange); //midi control message interrupt
    routine
void handleMIDIon(byte channel, byte note, byte velocity)
{
  if (channel == (midichannel)) //if the message is on the user set channel then
      call note ON
  {
    noteON(note);
  }
}
void handleMIDIoff(byte channel, byte note, byte velocity)
{
  if (channel == (midichannel))  //if the message is on the user set channel then
      call note OFF
  {
    noteOFF(note);
  }
}
```

Listing 4: Voice allocation algorithm

```
void noteOFF(byte note)
{
  if ((voicetrig[0] == 1) && (note == voicenote[0]))
  {
    //turn the note OFF on voice 1
    voicetrig[0] = 0;
  }
  // same is repeated for the rest of the voices
}
void noteON(byte note)
{
    if (voicehist[0] == 1)
    {
      voicenote[0] = note;
      voicetrig[0] = 1;
      //shift voicehistory by 1 to the left
      temp = voicehist[3];
      voicehist[3] = voicehist[2];
      voicehist[2] = voicehist[1];
      voicehist[1] = voicehist[0];
      voicehist[0] = temp;
      //turn the note ON on voice 1
      return;
    }
  // same is repeated for the rest of the voices
  }
```

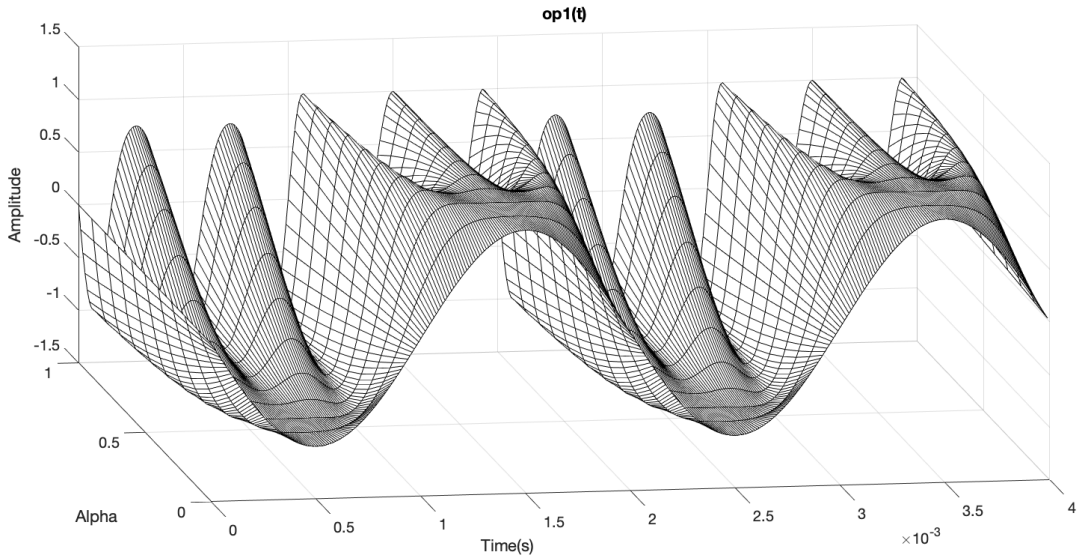

Figure 25: $f_1 = 500Hz$, $\beta = 0$ and $\gamma = 2$

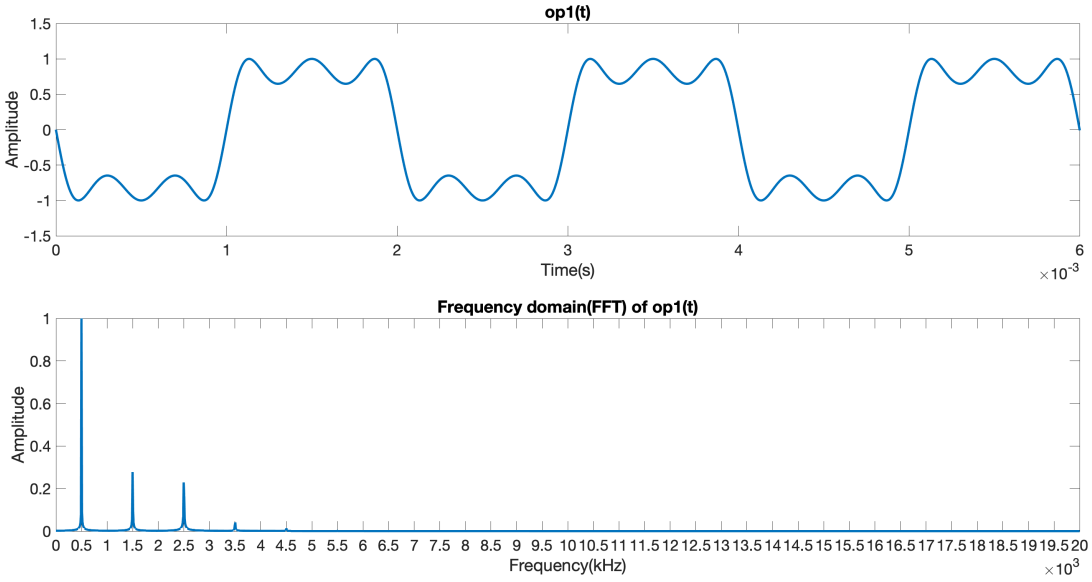

Figure 26: FFT analysis with $f_1 = 500Hz$, $\alpha = 0.5$, $\beta = 0$ and $\gamma = 2$

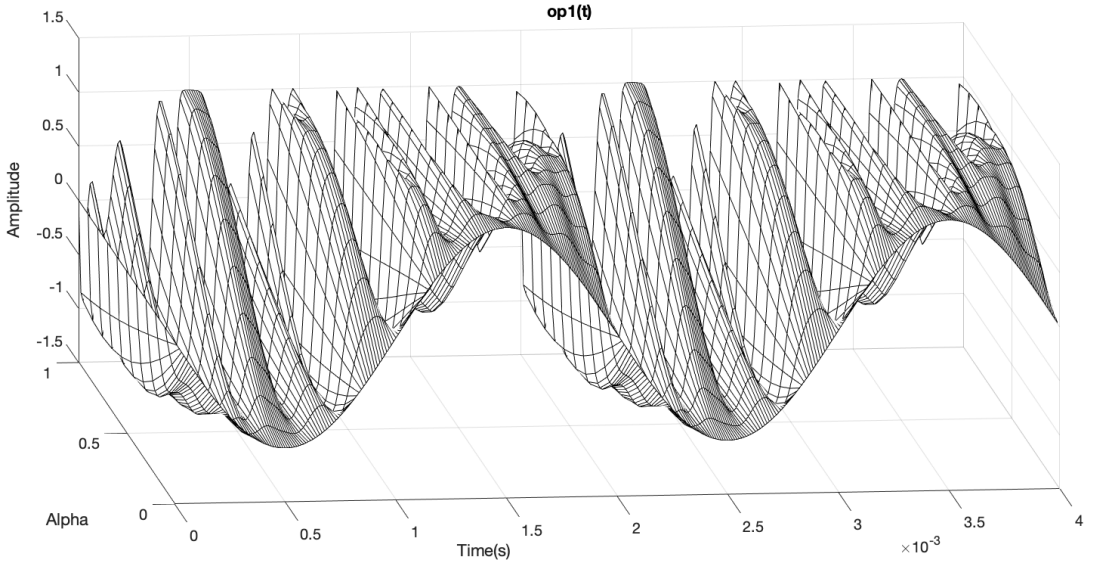

Figure 27: Simulated PM wave with $f_1 = 500Hz$, $\beta = 0.5$, $\delta = 4$ and $\gamma = 2$

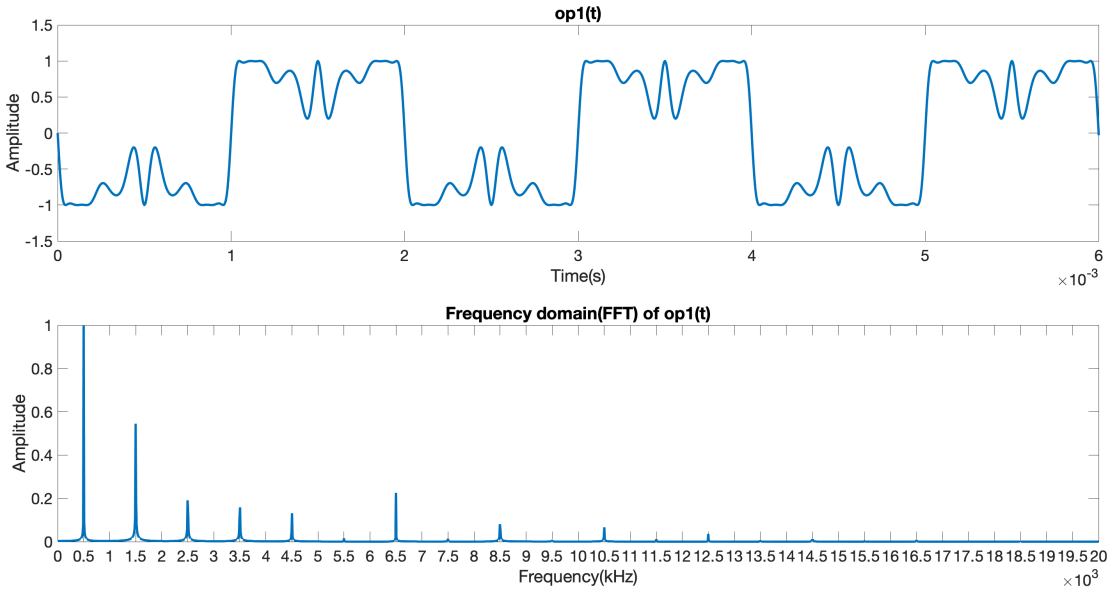

Figure 28: Simulated PM wave and FFT analysis with $f_1 = 500Hz$, $\alpha = 0.5$, $\beta = 0.5$, $\delta = 4$ and $\gamma = 3$

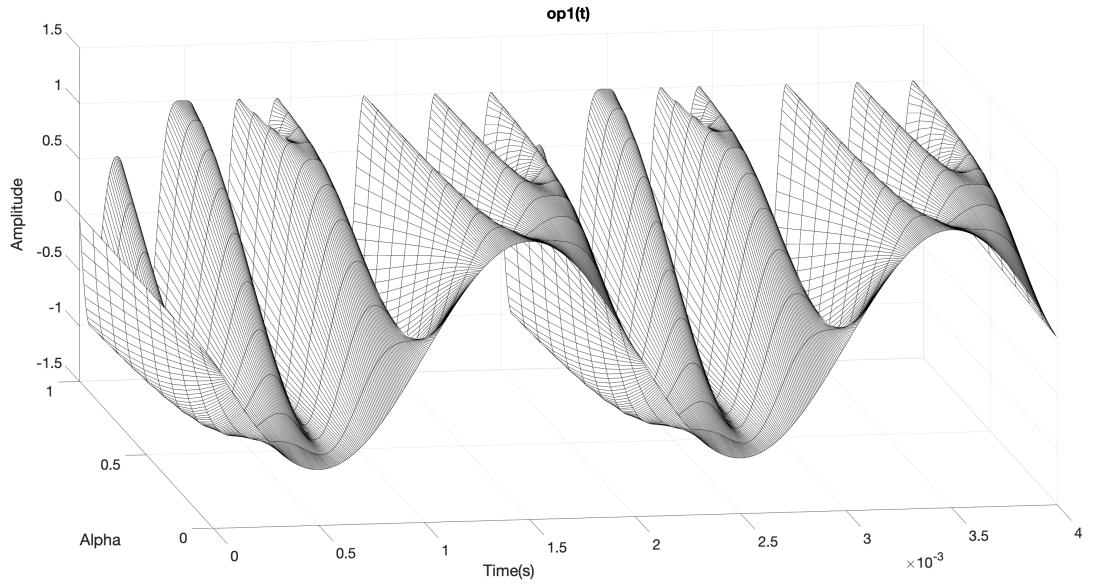

Figure 29: Simulated PM wave with $f_1 = 500Hz$, $\beta = 0$ and $\gamma = 3$

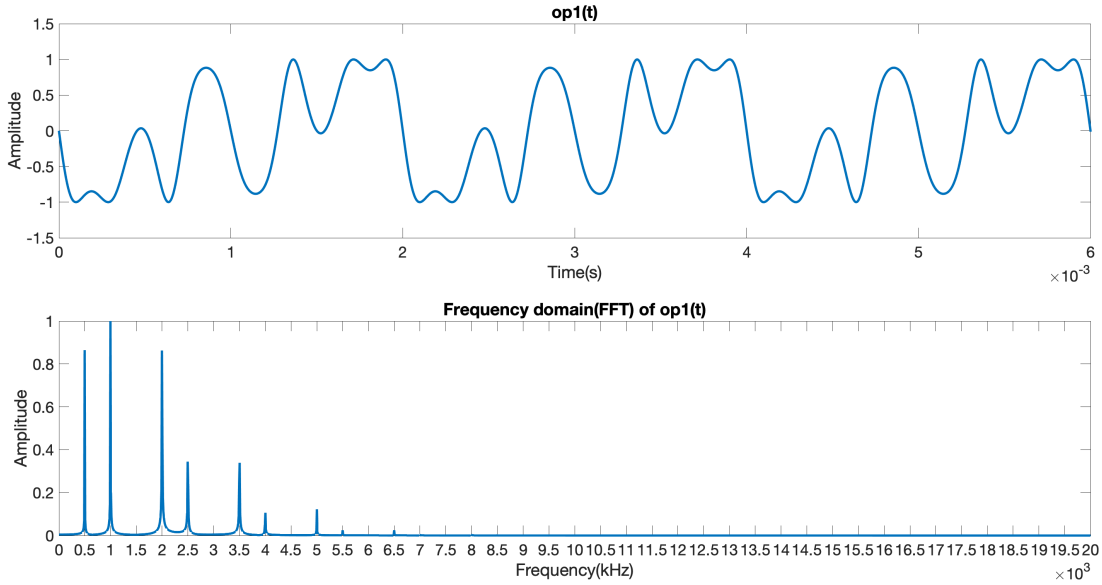

Figure 30: Simulated PM wave and FFT analysis with $f_1 = 500Hz$, $\alpha = 0.5$, $\beta = 0$ and $\gamma = 3$

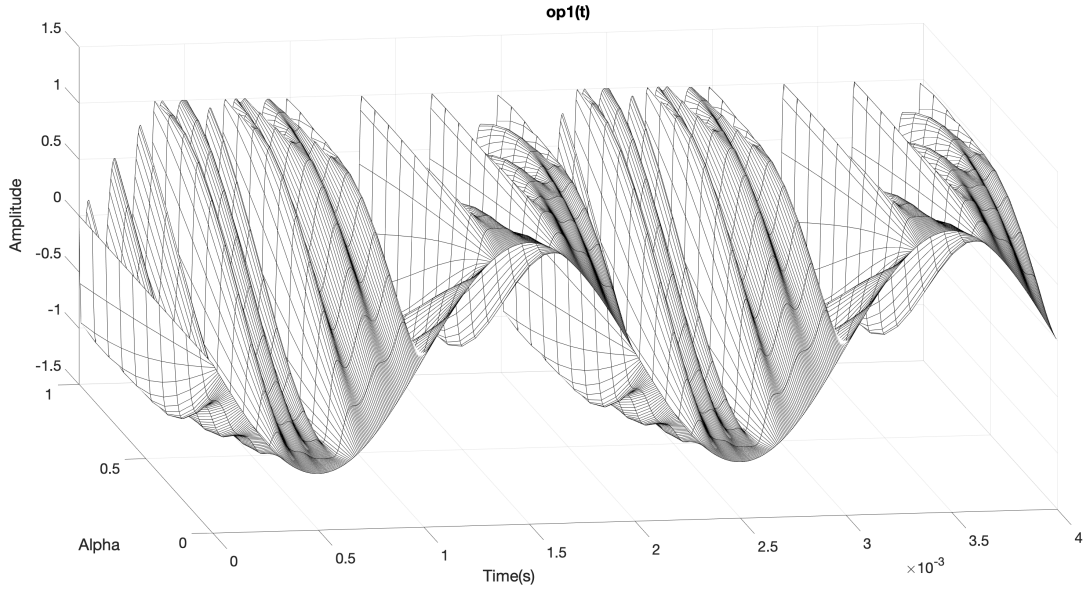

Figure 31: Simulated PM wave with $f_1 = 500Hz$, $\beta = 0.5$, $\delta = 6$ and $\gamma = 3$

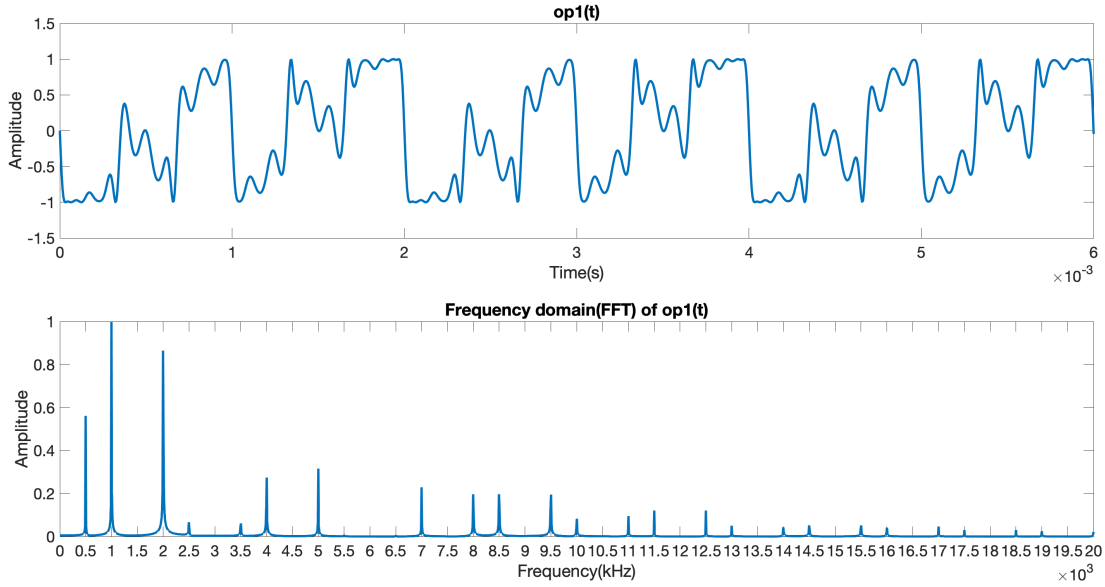

Figure 32: Simulated PM wave and FFT analysis with $f_1 = 500Hz$, $\alpha = 0.5$, $\beta = 0.5$, $\delta = 6$ and $\gamma = 3$

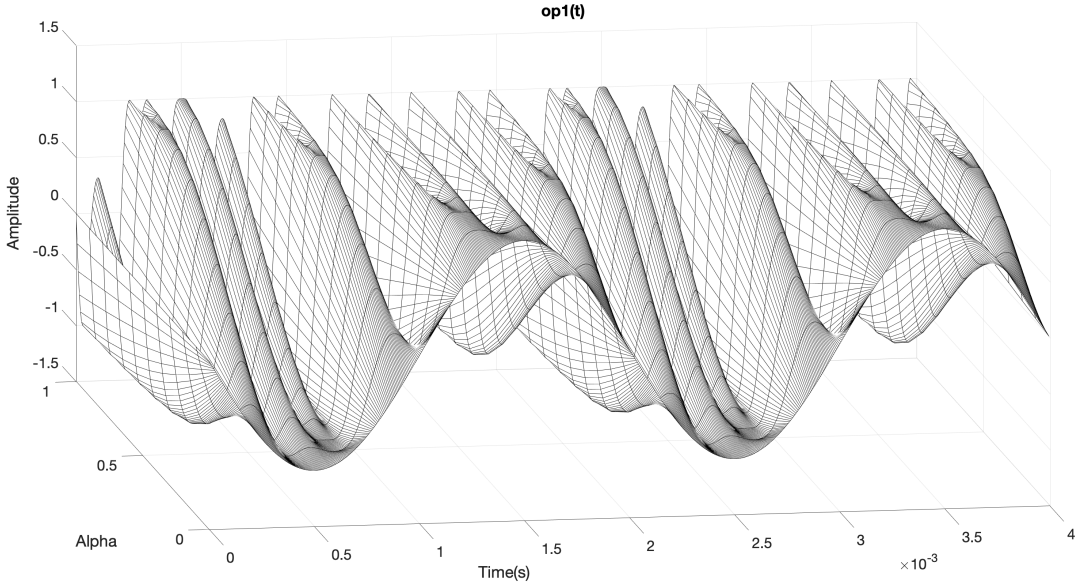

Figure 33: Simulated PM wave with $f_1 = 500Hz$, $\beta = 0$ and $\gamma = 5$

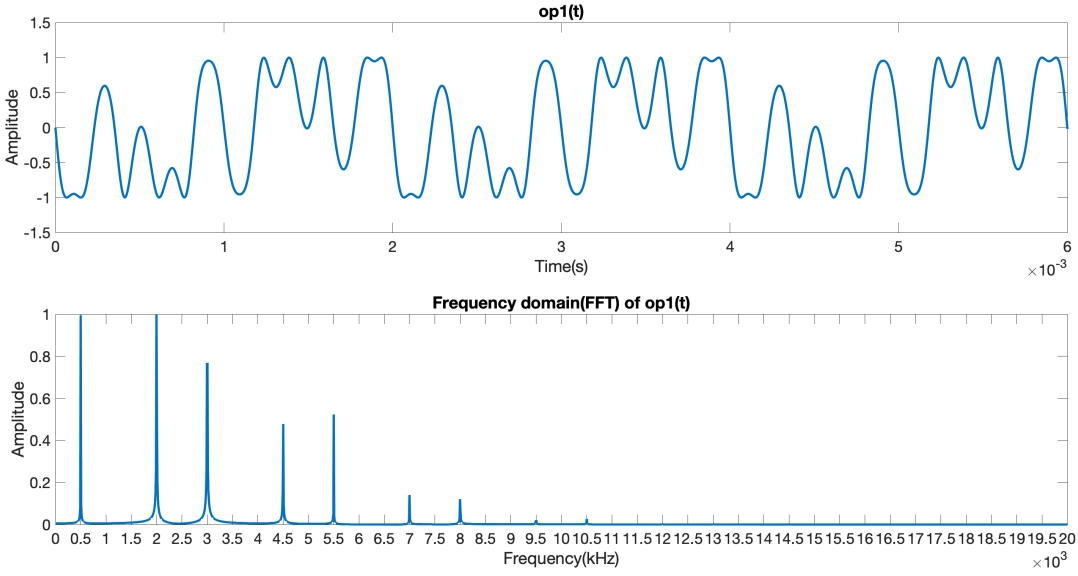

Figure 34: Simulated PM wave and FFT analysis with $f_1 = 500Hz$, $\alpha = 0.5$, $\beta = 0$ and $\gamma = 5$

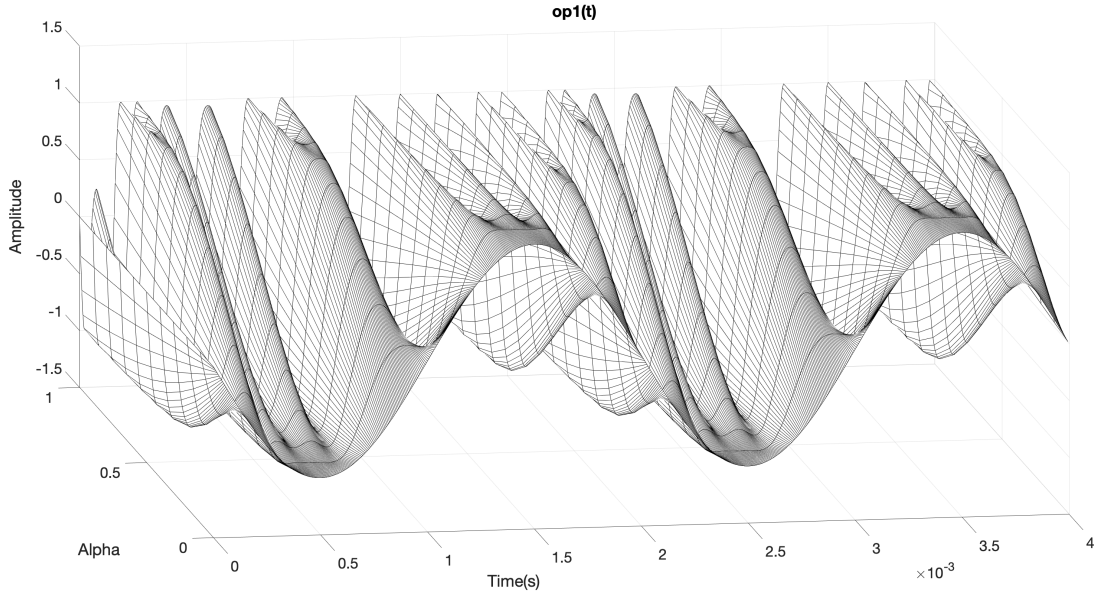

Figure 35: Simulated PM wave with $f_1 = 500Hz$, $\beta = 0.5$, $\delta = 1$ and $\gamma = 5$

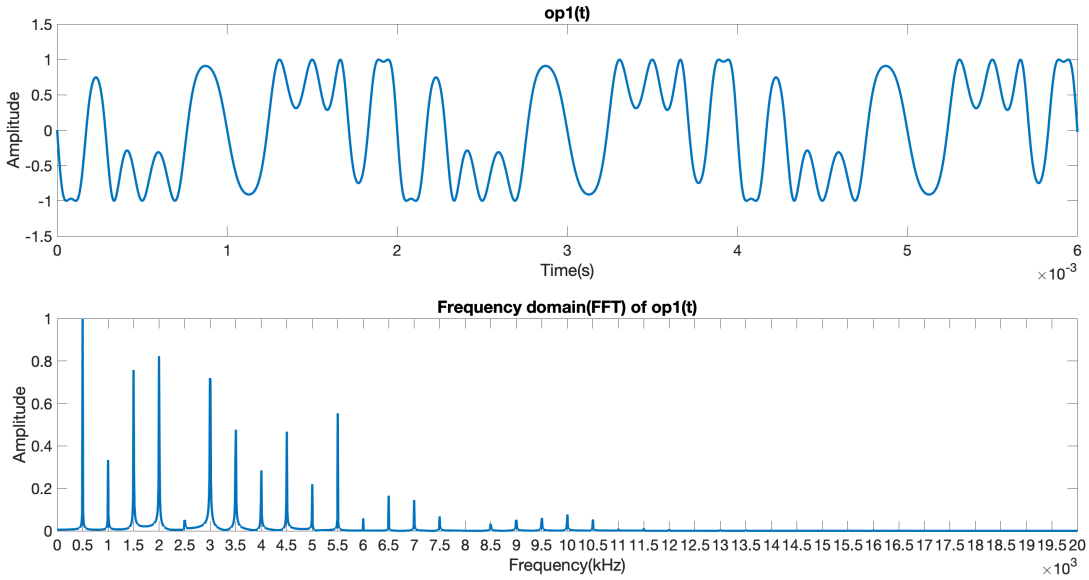

Figure 36: Simulated PM wave and FFT analysis with $f_1 = 500Hz$, $\alpha = 0.5$, $\beta = 0.5$, $\delta = 1$ and $\gamma = 5$

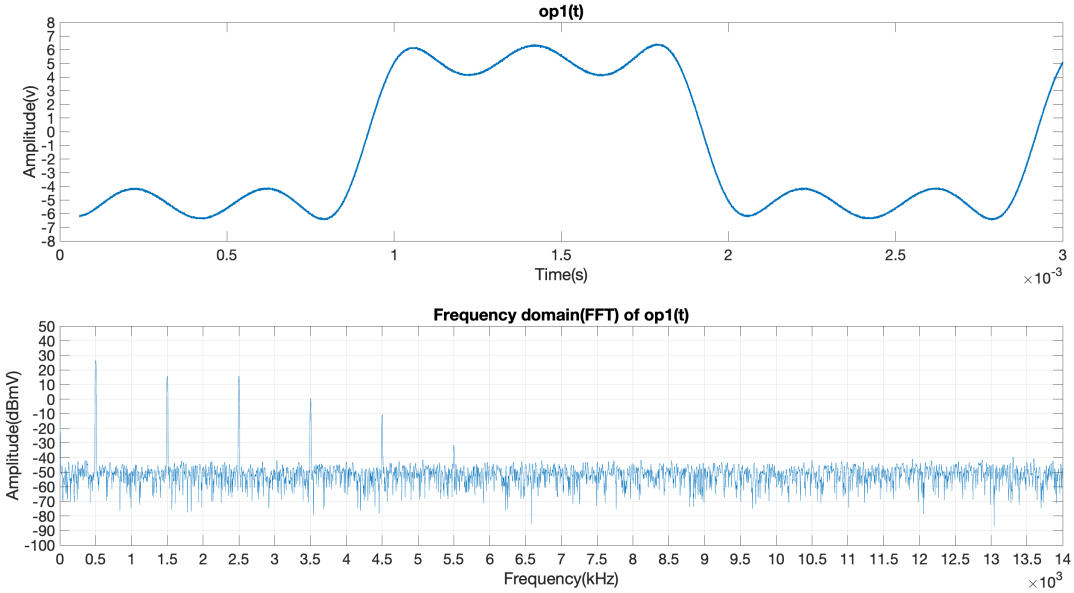

Figure 37: Synthesized PM wave and FFT analysis with $f_1 = 500Hz$, $\beta = 0$ and $\gamma = 2$

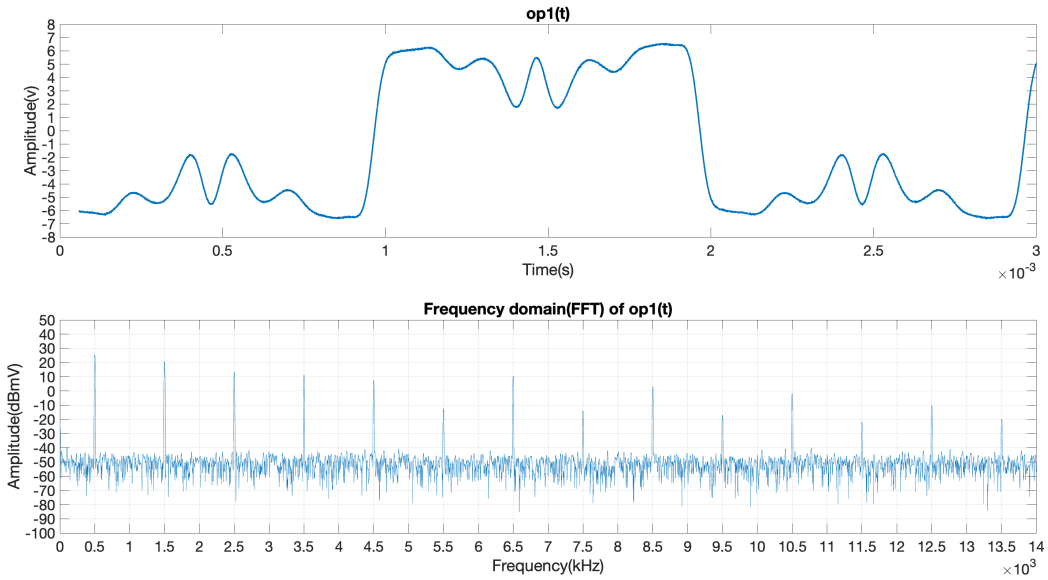

Figure 38: Synthesized PM wave and FFT analysis with $f_1 = 500Hz$, $\beta = 0.5$, $\delta = 4$ and $\gamma = 2$

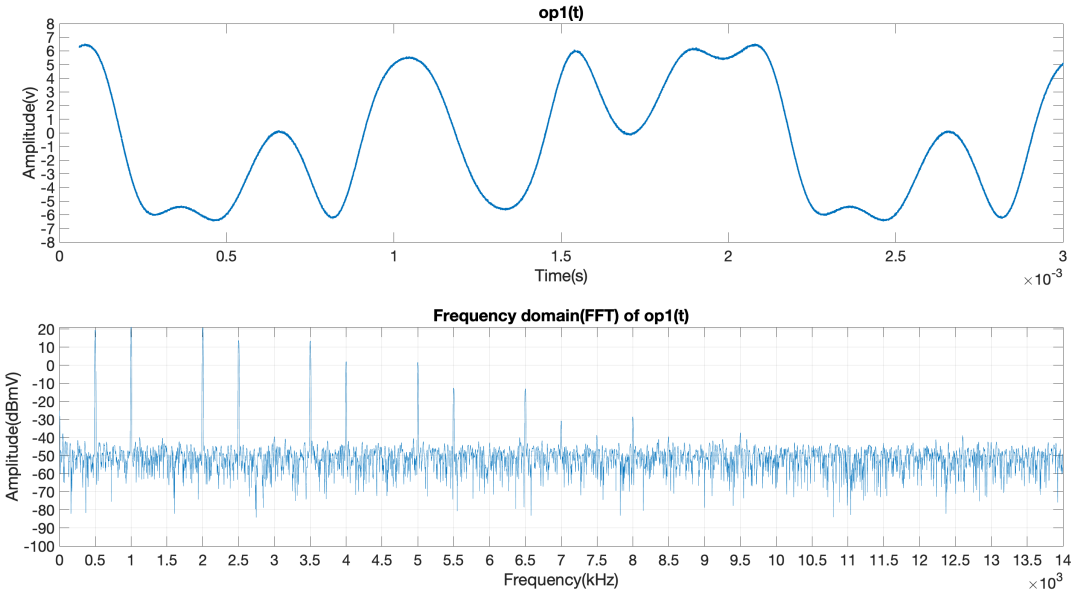

Figure 39: Synthesized PM wave and FFT analysis with $f_1 = 500Hz$, $\beta = 0$ and $\gamma = 3$

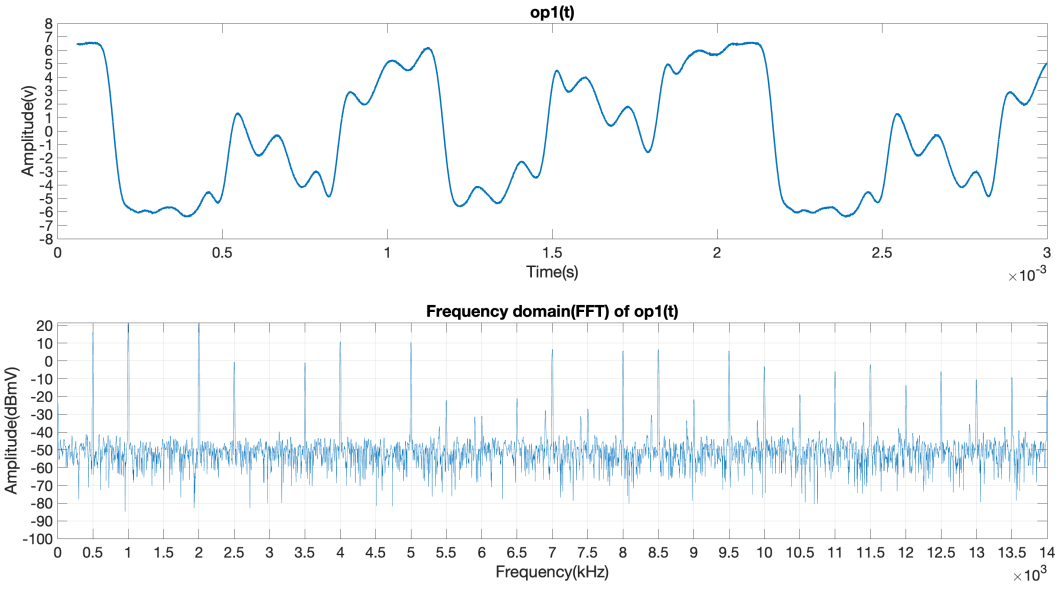

Figure 40: Synthesized PM wave and FFT analysis with $f_1 = 500Hz$, $\beta = 0.5$, $\delta = 6$ and $\gamma = 3$

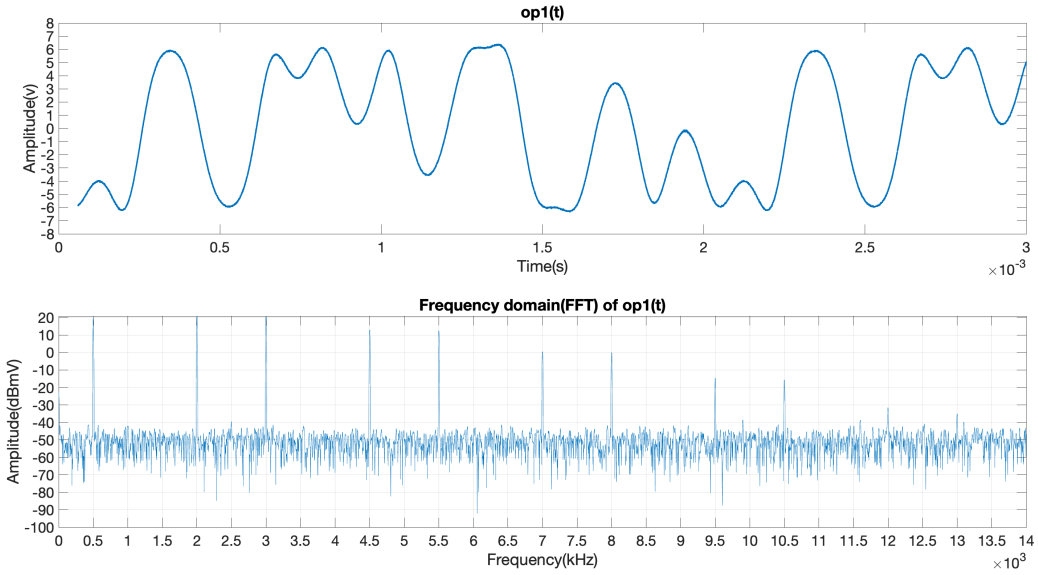

Figure 41: Synthesized PM wave and FFT analysis with $f_1 = 500Hz$, $\beta = 0$ and $\gamma = 3$

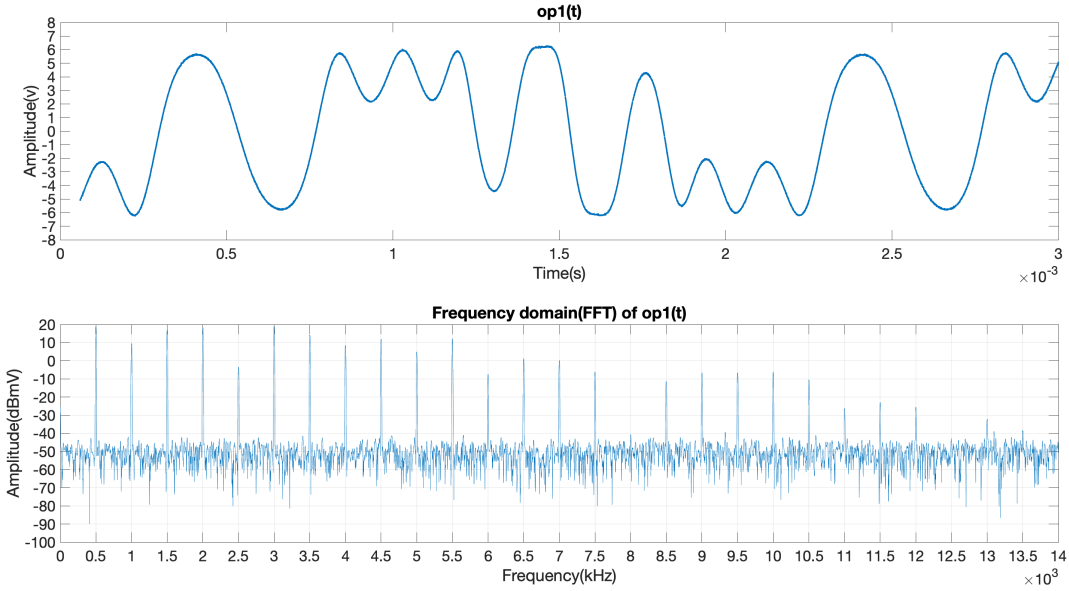

Figure 42: Synthesized PM wave and FFT analysis with $f_1 = 500Hz$, $\beta = 0.5$, $\delta = 1$ and $\gamma = 3$