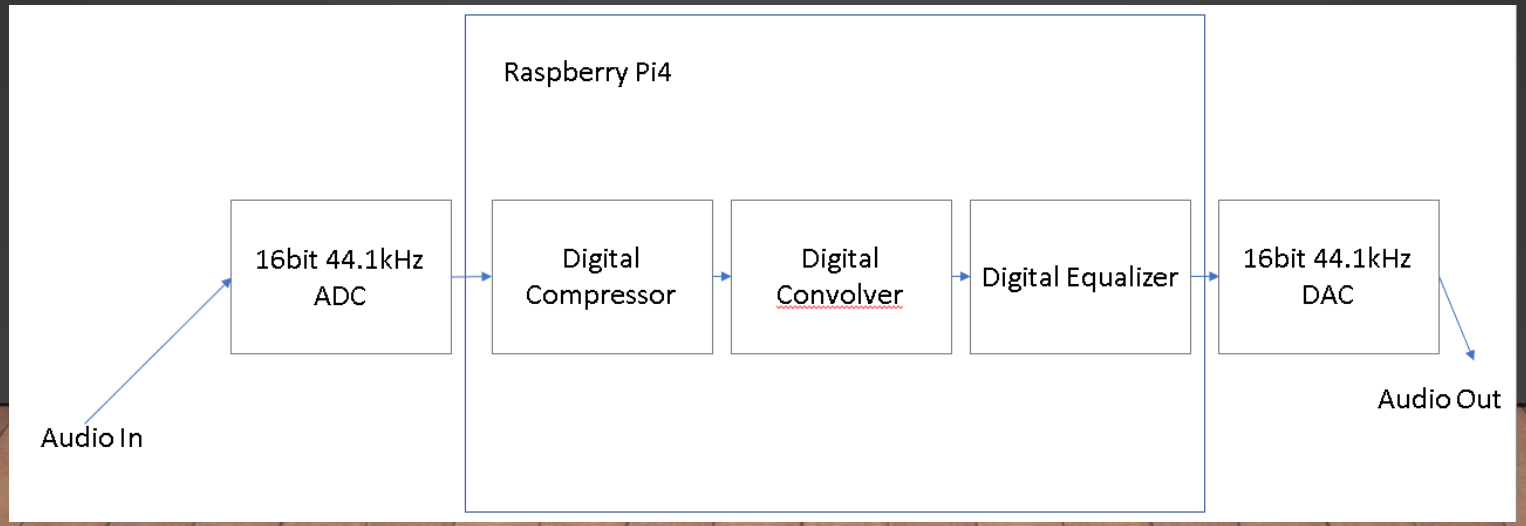
FXBOX: DIGITAL AUDIO PROCESSING

DANTE VILORIA, EECE 490B

PROJECT OVERVIEW

Specifications were

- 16-bit, 44.1kHz audio
- Delay less than 15ms
- Realtime implementations of a digital compressor, convolver, and equalizer



TOOLS USED

Hardware

- Raspberry Pi 4B 8GB
- HiFiBerry DAC+ADC
- Adafruit 20x4 Character LCD
- Adafruit USB backpack

Software

- Python 3.9
  - Numpy, Scipy for data processing
  - Pynput, Pyserial for the UI
  - Pyaudio for real time audio I/O
- MATLAB
  - For testing purposes

RASPBERRY PI 4/HIFIBERRY DAC + ADC SPECIFICATIONS

Maximum input voltage	2.1Vrms	4.2Vrms for balanced input
Maximum output voltage	2.1Vrms	
ADC signal-to-noise ratio	110db	typical
DAC signal-to-noise ratio	112db	typical
ADC THD+N	-85db	typical
DAC THD+N	-93db	typical
Input voltage for lowest distortions	0.8Vrms	typical
Input gain (configurable with Jumpers)	0dB, 12dB, 32dB	
Power consumption	<0.3W	
Sample rates	44.1-192kHz	

Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
1GB, 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)
2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
Gigabit Ethernet
2 USB 3.0 ports; 2 USB 2.0 ports.
Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
2 × micro-HDMI ports (up to 4kp60 supported)
2-lane MIPI DSI display port
2-lane MIPI CSI camera port
4-pole stereo audio and composite video port
H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
OpenGL ES 3.1, Vulkan 1.0
Micro-SD card slot for loading operating system and data storage
5V DC via USB-C connector (minimum 3A*)
5V DC via GPIO header (minimum 3A*)
Power over Ethernet (PoE) enabled (requires separate PoE HAT)
Operating temperature: 0 – 50 degrees C ambient

DIGITAL COMPRESSOR

Purpose

- To lower the volume of loud sounds
- This evens out of the volume level of an audio signal

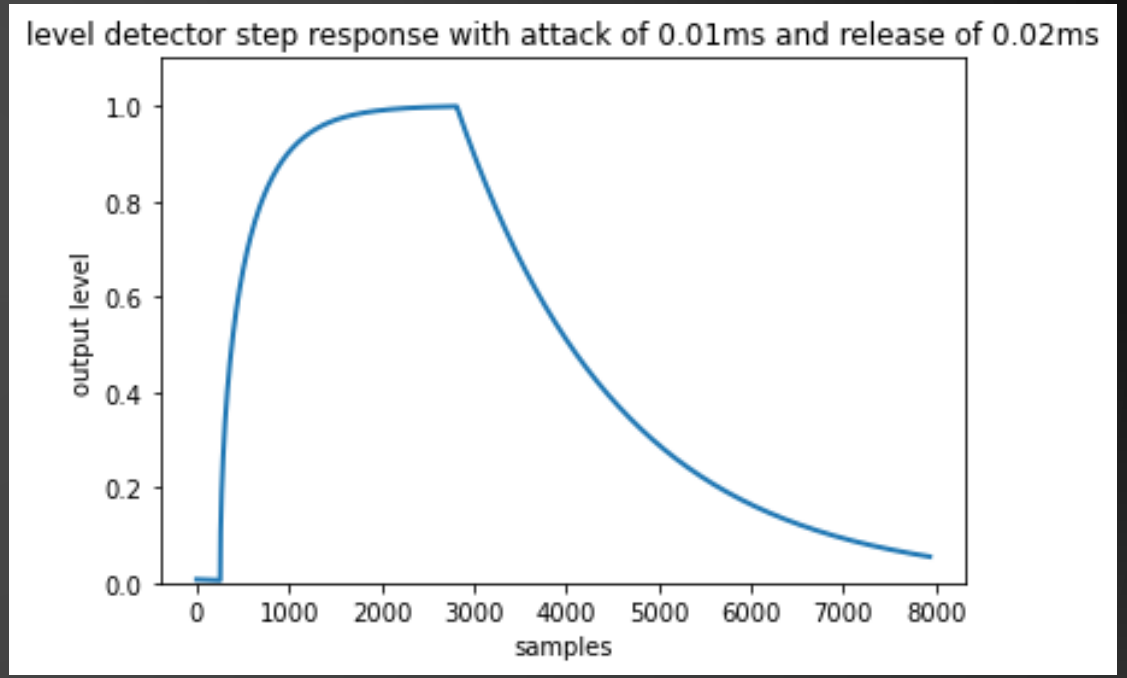
DIGITAL COMPRESSOR ARCHITECTURE

Many compressors can be constructed with the following blocks [1]

- Absolute value block – get the absolute value of the signal
- Linear to dB block – convert a linear audio signal to dB
- dB to linear block – convert a dB audio signal to linear
- Level detector – Create a smooth signal of the audio's volume level
- Gain computer – Calculates the what amount of gain should be applied to the signal

LEVEL DETECTOR [1]

- Contains two 1-pole lowpass filters
- One is the attack LPF, the other is the release LPF
- The attack LPF is used if the current sample is louder than the previous sample
- The release LPF is used otherwise
- The time constants of these LPF filters are referred to as the attack and release times respectively
- Attack and release affect how fast the compressor can react to changes in volume
- They are separate because usually it is desirable to have a fast attack and a slow release

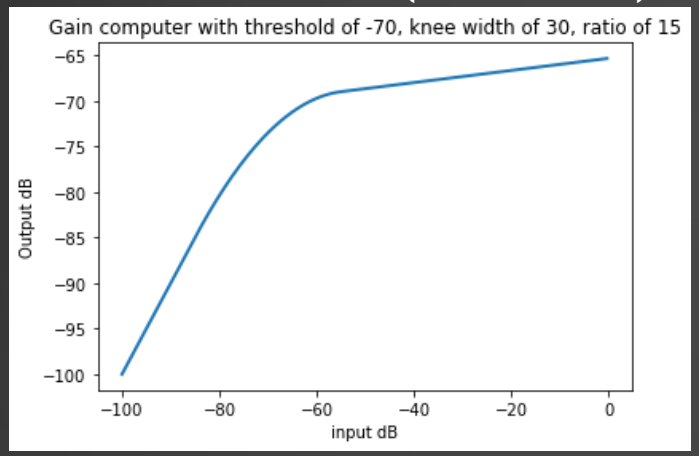


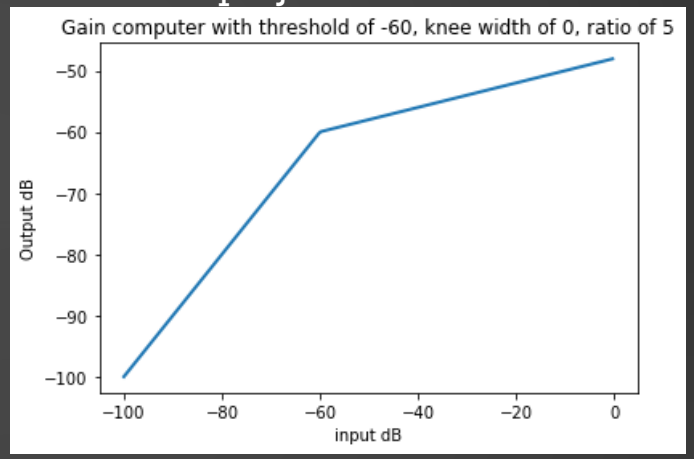
GAIN COMPUTER [1]

The gain computer maps an input dB level to an output dB level

The gain computer has the parameters of threshold, ratio, and knee width

The gain computer of Giannoulis et al (shown below) was used in this project



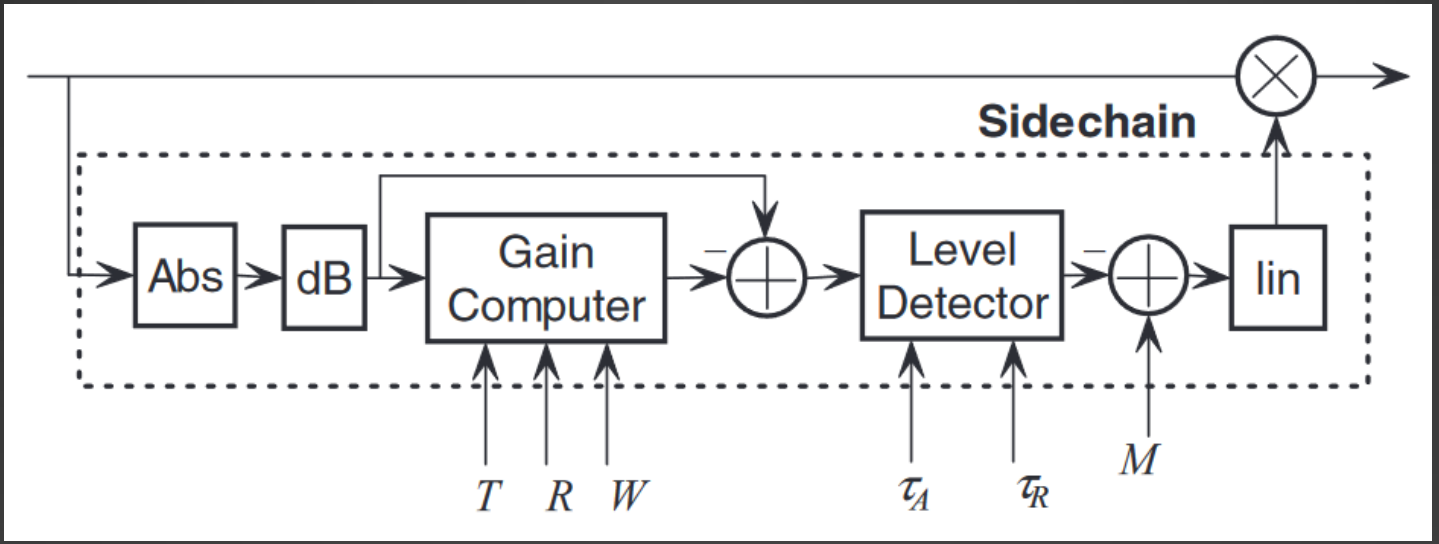


$$y_G = \begin{cases} x_G & 2(x_G - T) < -W \\ x_G + (1/R - 1)(x_G - T + W/2)^2/(2W) & 2|(x_G - T)| \leq W \\ T + (x_G - T)/R & 2(x_G - T) > W \end{cases}$$

x_G *is the input dB,* y_G *is the output dB,* T *is the threshold,* R *is the ratio,* W *is the knee width* [1]

COMPRESSOR ARCHITECTURE

- The log-domain compressor architecture of Giannoulis et al was implemented in this project [1]



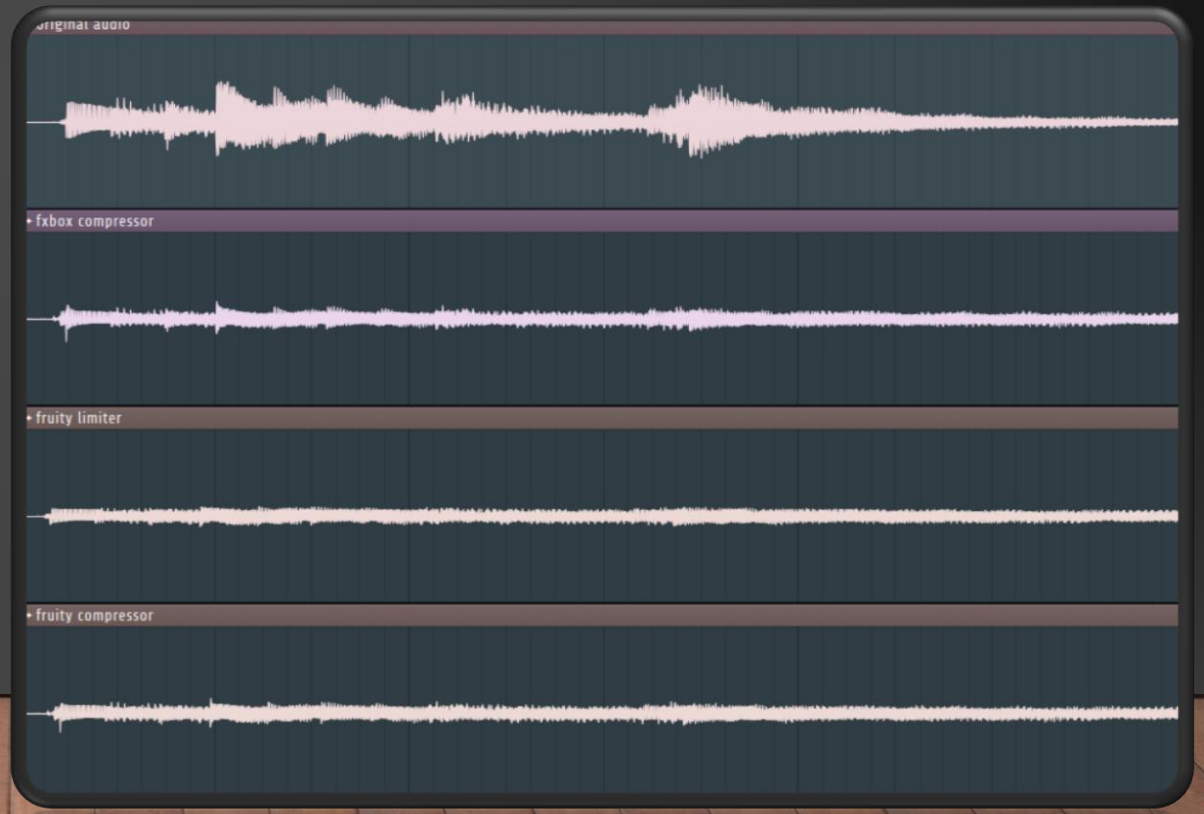
COMPRESSOR TEST RESULTS

Unfortunately, it is difficult to test compressors in a "scientific" way because there is no standard for compressors

- So, we will compare the fxbox compressor to two other compressors on the market
- The fxbox compressor offers similar results to the other two compressors, so this is a success
  - In addition to the previously mentioned parameters, digital compressors can have features such as a hold time or RMS level detector that can cause differences between compressors

Settings

- Threshold of -40dB
- Knee width of 0dB
- Ratio of 7
- Attack of 1ms
- Release of 500ms



CONVOLVER

- Convolvers have many uses, but they are typically used to add realistic sounding reverberations to an audio signal
- The convolver will convolve an incoming audio signal with an impulse file in real time
- Impulse files with a length of up to 5 seconds (220.5k taps) were confirmed to be working on a buffer size of 256
- If we are working with impulses that are up to five seconds long, it is impractical to convolve each audio buffer with the entire impulse [2][4]
- In order to accomplish this, we use the linear property of convolution in order to break the impulse into smaller parts and do convolutions when they are needed

PARTITIONED CONVOLUTION

The idea is largely credited to William G. Gardner [2]

- Impulse partitions start at one buffer length [4]
- Partitions get larger until they reach the size at which the computer can most efficiently compute FFTs [4]
  - Usually 4096 – 16384 samples, but it depends on the system

General optimizations that are used are

- Computing convolutions via fourier transform [2]
- Using real fourier transforms [2][4]
- Only using transform lengths that are powers of 2 [2]
- Pre-computing the fourier transforms of impulse partitions [2][4]

IMPULSE PARTITIONING

A custom partitioning algorithm was created for this project

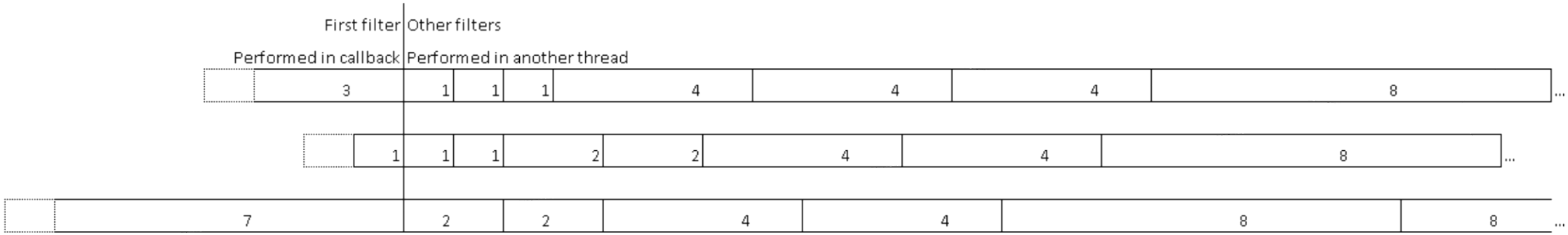
The parameters are first filter power, power start, power step, power stop, and height

A partition of size n is convolved with n previous buffers, except for the first filter which is always convolved with a single buffer

For partition 1: first filter power = 2, power start = 0, power step = 2, height = 3

For partition 2: first filter power =, 1 power start = 0, power step = 1, height = 2

For partition 3: first filter power = 3, power start = 1, power step = 2, height = 2



PARTITIONED CONVOLUTION IMPLEMENTATION PT 1

In practice the following structures are needed:

- Previous buffers – an array containing previous buffers
  - The number of previous buffers that need to be stored is the length of the longest partition
- Convolution buffer – an array where the results of the convolutions are added to
  - Needs to be at least the length of the impulse
- Impulse partitioner – decides how an impulse file should be partitioned, runs once at startup or whenever the impulse is changed
- Impulse partitions – an array of objects that encapsulates a filter partition and contains info about it such as its offset, how many buffer lengths it is, etc.
- Convolution queue(s) – one or many queues that contain the convolutions that need to be done
  - In this project a single priority queue based on a convolution's deadline was used

PARTITIONED CONVOLUTION IMPLEMENTATION PT 2

In the callback function:

- Calculate convolution of first filter with most recent buffer
- Add result to the convolution buffer
- Determine which other convolutions can be performed and add them to the priority queue

In the worker thread:

- Get a convolution task from the priority queue
- Get the appropriate number of previous buffers
- Calculate the convolution
- Add the result to the convolution buffer

PARTITIONED CONVOLUTION IMPLEMENTATION PT 3

Dry signal – the input signal to the covnolver

Wet signal – the result of the convolution of the input signal and impulse

The final convolver output is the dry signal added to the wet signal

- The wet gain parameter is the gain of the wet signal
- The dry gain parameter is the gain of the dry signal

CONVOLVER TEST RESULTS

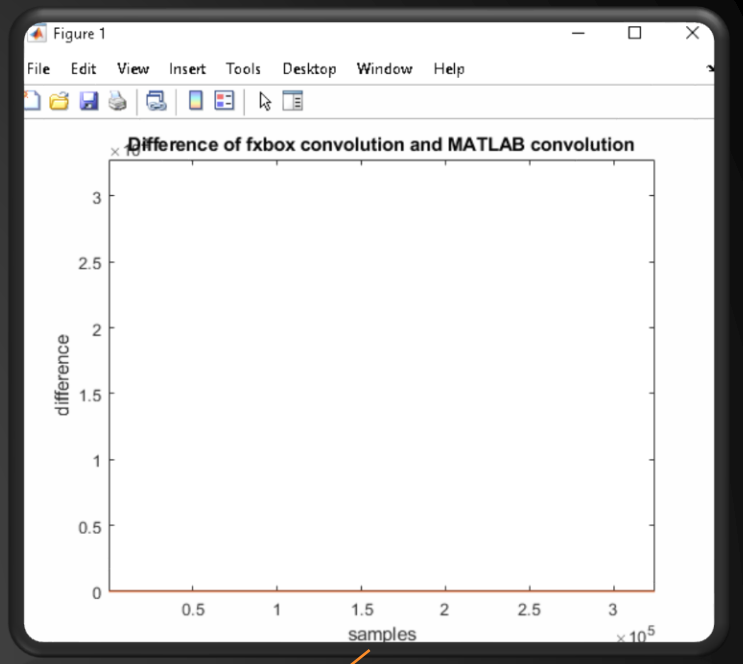
Convolution is a well-defined mathematical function

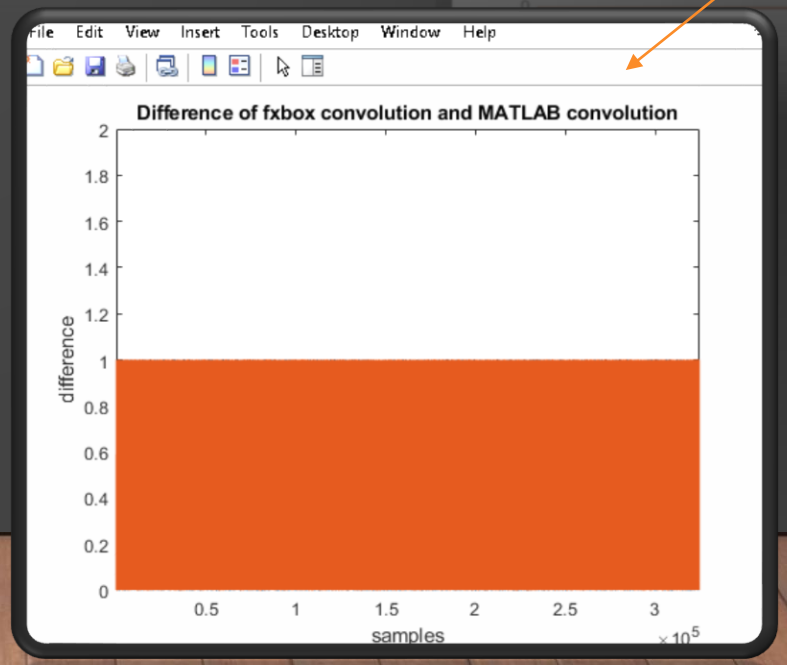
The ideal situation is for the fxbox convolver to produce the same results as a 'traditional' convolution

In order to test this, we convolve an audio file with an impulse using the fxbox's convolver and compare the result to MATLAB's conv() command by subtracting the results from each other

The result is that the difference between the two methods is either 0 or 1

- This is the ideal result for a 16-bit integer wave file as this indicates that the only errors are rounding errors



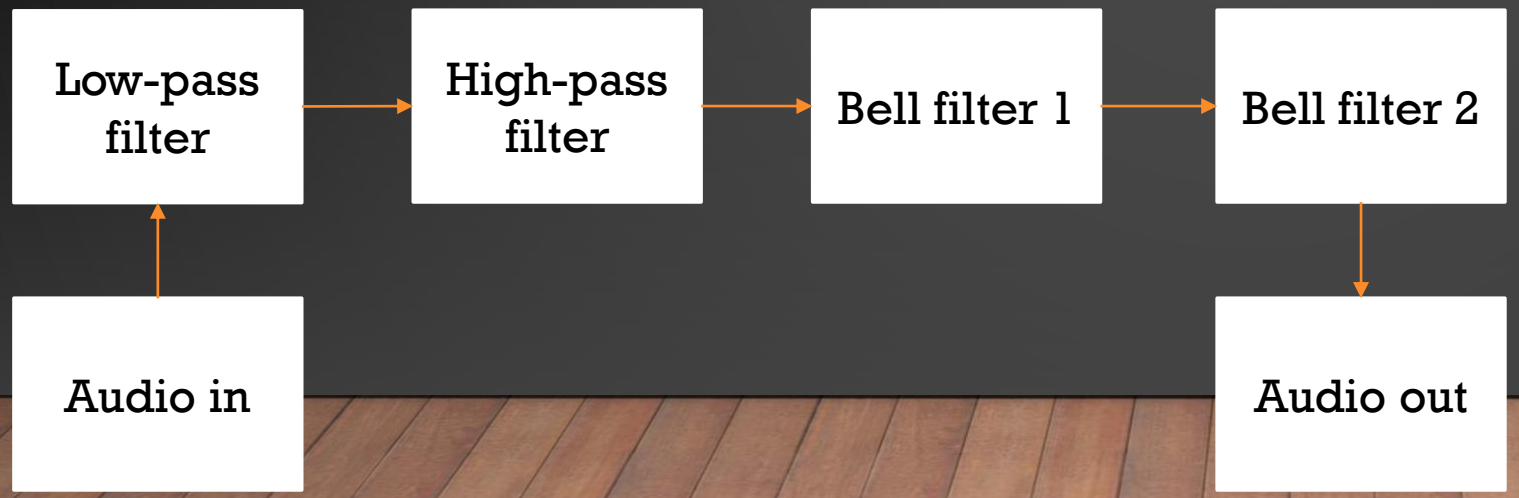


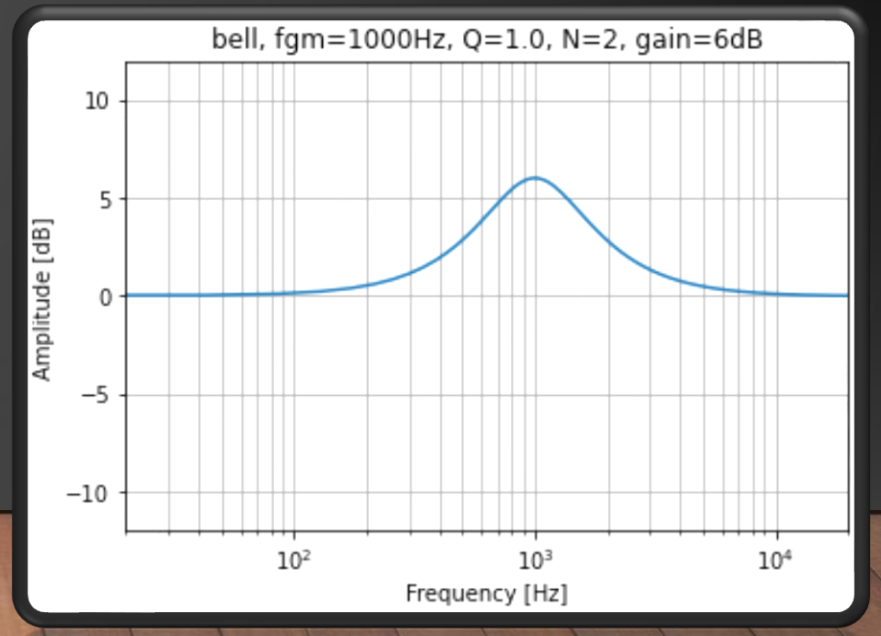
DIGITAL EQUALIZER

The digital equalizer contains four filters that are cascaded

A low-pass, high-pass and two bell filters (also called peaking filters)

- A bell filter is a band-pass filter + an all-pass filter
- Its purpose is to attenuate or boost by a desired dB value at its center frequency but preserve the rest of the signal





DIGITAL FILTER DESIGN

The filters are designed to the user's specification

- Cutoff frequency and order for low-pass and high-pass filters
- Center frequency, bandwidth (as a factor of the center frequency) and gain applied at the center frequency

Butterworth filters were chosen because of their level gain across the passband and smooth transition band which makes them desirable for audio applications

The filter coefficients themselves can be designed via Scipy's butter() command which will design a filter given an order and cutoff frequency

- This is sufficient to design the low-pass and band-pass filters, but the peaking filters require extra design work

BELL FILTER DESIGN 1: BAND-PASS DESIGN

We want to choose upper f_1 and lower f_2 cutoff frequencies for a center frequency f_{gm} and bandwidth as a multiple of the center frequency Q^{-1}

$$(1)\ f_{gm} = \sqrt{f_1 f_2} \text{ where } f_1 = R f_{gm} \ and \ f_2 = \frac{1}{R} f_{gm} [5]$$

$$(2)\ Q = \frac{f_{gm}}{B} \text{ where B} = f_1 - f_2$$

$$R = \frac{Q^{-1} + \sqrt{Q^{-2} + 4}}{2} \ is \ derived \ from \ (1) \ and \ (2)$$

Having found R from f_{gm} and Q^{-1}, appropriate values of f_1 and f_2 can be calculated

BELL FILTER DESIGN 2

We want to apply KdB of gain at the center frequency of the bell filter

The transfer function of a bell filter is

$$Bell(s) = 1 + g * BPF(s)$$

Where $BPF(s)$ is a band-pass filter and g is the linear gain applied at the band-pass filter's center frequency in order to obtain KdB of gain

The total linear gain applied at the Bell filter's center frequency is

$$K_{lin} = 1 + g$$

Converting to dB and solving for g yields

$$g = 10^{K_{dB}/20} - 1$$

DIGITAL IIR FILTERING ALGORITHM [3]

- An implementation of a direct form II difference equation was created in order to perform the filtering

$$a[0]y[n] + a[1]y[n\ -1] +\ \ldots + a[N]y[n-N] = b[0]x[0] + b[1]x[n\ -1] + \cdots + b[M]x[n-M]$$

$$Y(Z) = \frac{b[0] + b[1]Z^{-1} + \cdots + b[M]Z^{-M}}{a[0] + a[1]Z^{-1} + \cdots + a[N]Z^{-N}}$$

EQUALIZER TEST RESULTS

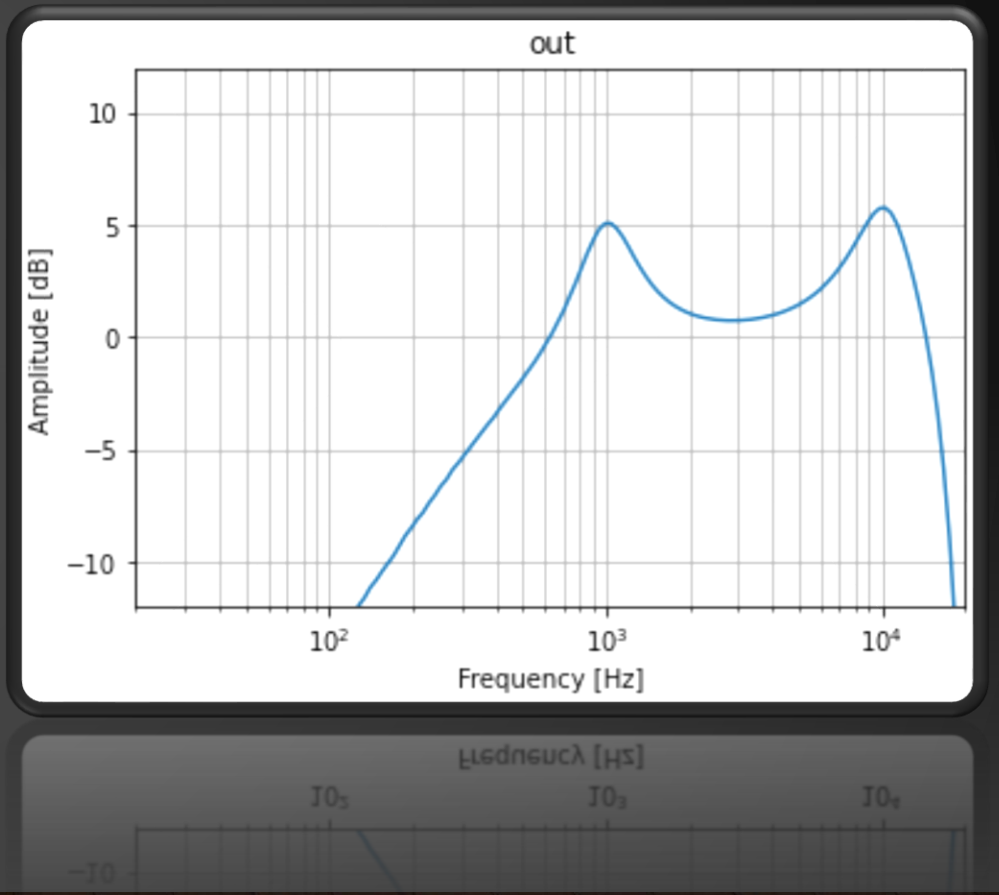
The frequency response was measured experimentally in order to test the equalizer

A wave file was generated that increased in frequency every 0.2 seconds and spanned 20Hz to 20kHz

This file was run through the fxbox's equalizer and used to determine the frequency response of the equalizer with settings

- High-pass at 500Hz
- Low-pass at 15kHz
- Bell at 1kHz and at 10kHz, each with 6dB gain

The fxbox's equalizer was able to generate a frequency response that reflects these settings



USER INTERFACE

- A user interface was implemented via a 20x4 character LCD screen
- It allows the user to control the audio effects via a keyboard
- Keys 1-4 select a menu item, the 0 key goes backwards into the menu
- The +/- keys can increment/decrement a value on the screen

RESULTS

- The individual subsystems (compressor, convolver, equalizer) were confirmed to be functioning properly
- All three of the subsystems could be enabled and run in realtime with a sampling frequency of 44.1kHz buffer size of 256
- The UI works but is rough around the edges
- There are minor issues that occur when settings are changed
  - These could be fixed and are due to some poor design choices
- Overall, the project was a success

Demonstration: https://www.youtube.com/watch?v=CQCDXM29WBg

REFERENCES

[1] D. Giannoulis and M. Massberg and J. Reiss, Digital Dynamic Range Compressor Design

[2] F. Wefers, Partitioned Convolution Algorithms for Real-time Auralization

[3] scipy.signal.lfilter, https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lfilter.html

[4] W. G. Gardner, Efficient Convolution without Input-Output Delay

[5] V. Välimäki and J. Reiss, All About Audio Equalization: Solutions and Frontiers