Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Ngoc Thy Pham

Real time audio processing on a Raspberry Pi using partitioned convolution

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Ngoc Thy Pham

Real time audio processing on a Raspberry Pi using partitioned convolution

Bachelorthesisbased on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Aplied Sciences

Supervising examiner : Prof. Dr. Klaus Jünemann
Second Examiner : Prof. Dr. Robert Heß

Day of delivery 7. September 2017

Ngoc Thy Pham

Title of the Bachelorthesis

Real time audio processing on a Raspberry Pi using partitioned convolution

Keywords

Raspberry Pi, FIR filter, IIR filer, direct convolution, fast block convolution, partioned convolution, BruteFIR, JACK Audio Connection Kit, Cirrus Logic Audio Card.

Abstract

The aim of this thesis is to implement a digital FIR filter on a Raspberry Pi with the help of Cirrus Logic Audio Card. The process of capturing from input, filtering and delivering the audio samples to the output must happen in real time.There are several convolution techniques are used including direct convolution, fast block convolution and partioned convolution in order to compare the speed of them and find the optimum technique for the filtering process. Thanks to BruteFIR (using fast block convolution and partioned convolution), the FIR filter can be run with a huge amount of coefficients in real time. This filter can be redesigned easily by changing its coefficients and is applied to design a filter for a loud speaker to compensate its amplitude response.

Ngoc Thy Pham

Titel der Arbeit

Echtzeit-Audioverarbeitung auf einem Raspberry Pi mit Hilfe der segmentierten Faltung

Stichworte

Raspberry Pi, FIR filter, IIR filer, schnelle Faltung, segmentierte Faltung, BruteFIR, JACK Audio Connection Kit, Cirrus Logic Audiokarte.

Kurzzusammenfassung

Das Ziel dieser Arbeit ist die Implementierung eines FIR Filters auf einem Raspberry Pi mit Cirrus Logic Audiokarte. Der ganze Prozess, die Abtastwerte vom Eingang zu erhalten, zu filtern und zum Ausgang zu schicken, muss in Echtzeit ablaufen. Verschiedene Filtertechniken, unter anderem direkte Faltung, schnelle Blockfaltung und segmentierte Faltung, werden im Hinblick auf Geschwindigkeit verglichen, um die optimale Technik fÃ¼r die Filterung zu identifizieren. Mit Hilfe der BruteFIR Bibliothek (die die schnelle Blockfaltung und segmentierte Faltung verwendet) kann ein FIR Filter mit einer sehr groÃen Anzahl von Koeffizienten in Echtzeit betrieben werden. Dieses Filter kann leicht durch Anpassen der Koeffizienten konfiguriert werden. Als Anwendung wird ein Filter zur Kompensation des Amplitudengangs eines Lautsprechers entwickelt.

Contents

List of Tables

List of Figures

1. Introduction

1.1. Goals of the thesis

The purpose of this thesis is to implement a digital filter using a Raspberry Pi which fulfills these requirements:

- Receive sound data as input signal.
- Get access to the sample values and convolve them with the coefficients of a digital FIR filter.
- Deliver the processed samples to the output so that they can be sent to a loud speaker.
- The work-flow must happen in real time.

After the filter is implemented, the Raspberry Pi is used to design a compensating filter along with two measurements are performed:

- Frequency response of a given loudspeaker.
- The combination of the compensating filter and the loudspeaker to confirm that the amplitude response is constant.

1.2. Digital filter

In signal processing realm, a digital filter is a discrete-time system which performs computations on digital signals, in order to reduce or amplify some components or features from those signals.

Digital filters are classified by their structure:

- Non-recursive filter: the current output value depends only on the input values.
- Recursive filter: the current output value depends on the input values and the previous output value.

According to the structures above, there are two primary types of filters in digital realm: Finite Impulse Response (FIR) filters, and Infinite Impulse Response (IIR) filters.

- FIR filters: these are non-recursive filters, which have finite-duration impulse responses.
- IIR filters: these are recursive filters, whose impulse responses do not become zero at a certain time, but continue indefinitely.

This thesis mainly focuses on the FIR filters since they are more stable, easy to implement and are guaranteed to have linear phase. In addition, an IIR filter is also implemented in this thesis.

1.3. Raspberry Pi

[Raspberry-Pi-Foundation (2017)]

Raspberry Pi is a low-cost single-board computer but has a high performance which is developed by the Raspberry Pi Foundation in United Kingdom.

There are different models of Raspberry Pis with various kinds of hardwares and performances have been released.

Raspberry Pi 2 Model B is used in this thesis: this is the second generation of Raspberry Pi which replaced the Raspberry Pi 1 Model B+ in February 2015.

Raspberry Pi 2 Model B specification:

- A 900MHz quad-core ARM Cortex-A7 CPU
- 1GB RAM
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- Ethernet port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)

- Micro SD card slot
- VideoCore IV 3D graphics core

1.4. Cirrus Logic Audio Card

[Cirrus-Logic (2014)]

Raspberry Pi, whilst being equipped with audio capability, remains limited in a number of ways. The limitations are intended to maintain Raspberry Pi's low price point, but still represent a limitation to users interested in exploring the audio capability of Raspberry Pi.

In terms of audio, there are no ways to capture audio using Raspberry Pi alone, and audio output is limited to two paths; analogue, via its onboard 3.5 mm stereo output jack, and digital, via its onboard HDMI output. Whilst the HDMI output provides the potential for high quality rendering of audio (depending on what HDMI devices are used to finally convert audio from its digital format to an analogue signal), the audio quality from the analogue 3.5 mm stereo output jack is universally recognized as being of an 'acceptable' quality level - no more.

The most important limitation is Raspberry Pi's lack flexibility in terms of multiple types of audio input sources, and outputs.

Cirrus Logic Audio Card is designed by element14 and Cirrus Logic in partnership addresses the above by providing a rich set of high quality audio features, including the following:

- Compatible with Raspberry Pi B+ and A+ onwards (with 40-pin extended GPIO, and no P5 connector)
- Analogue line-level output and input
- Digital stereo audio input and output (SPDIF)
- High quality headphone output, with microphone facility (for headphones with boom microphone)
- Onboard stereo digital microphones based on MEMS technology
- Ability to render High Definition (HD) Audio
- Arrives bundled with five High Definition (HD) audio files to demonstrate the systems capability.
- Onboard power amplifier for directly driving loudspeakers. (Requires headers to be fitted.1)

- Expansion header to allow connection to host boards other than Raspberry Pi.
- Back power protection, allows Raspberry Pi to be powered from the Cirrus Logic Audio card, allowing for convenient set up.
- New universal software support for both this card, and the former Wolfson Audio Card.

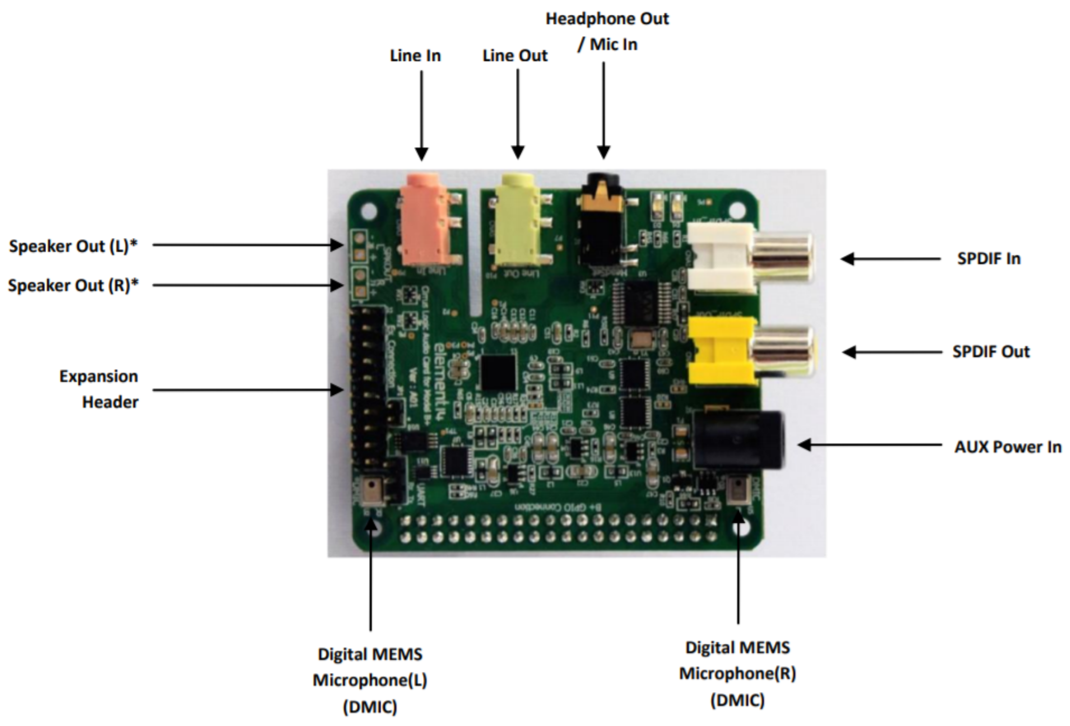

Figure 1.1.: Connections to and from the Cirrus Logic Audio Card

2. Background

2.1. Digital FIR filter

2.1.1. Definition

A digital Nth order FIR filter is a digital filter, which has the below transfer function:

$$H(z) = b_0 + b_{-1}z^{-1} + \ldots + b_N z^{-N} \tag{2.1}$$

The impulse response of this filter takes exactly N + 1 samples before it settles to zero.

FIR filter is the most common type of filter which is implemented in software.

2.1.2. Properties

FIR filters have some primary advantages:

- Easy to be designed to have linear phase.
- Simple to be implemented.
- Are always stable.
- Can be realized efficiently in hardware.

In comparison to IIR filters, FIR filters often require much higher order which leads to more memory and calculations needed to be able to archive the given response characteristics.

2.1.3. Filter design

The main purpose of designing an FIR filter which has a specific frequency response is to find the coefficients and the order of it. In the direct form FIR filter, these coefficients represent the impulse responses of the filter. The outputs of the signal can then be calculated by the convolution between the filter's coefficients and the previous signal inputs.

There are different popular methods are used to design an FIR filter:

- Window design method
- Frequency sampling method
- Weighted least squares design
- Parks-McClellan method
- FFT algorithms

The details of the methods above are out of this thesis's scope.

2.1.4. Output calculation

Direct convolution

When a signal passes through an Nth order FIR filter, each value of the output will be calculated by the weighted sum of the previous input values:

$$y[n] = b_0x[n] + b_1x[n-1] + ... + b_Nx[n-N] = \sum_{i=0}^{N} b_i.x[n-i] \tag{2.2}$$

Where:

- x[n]: the input values
- y[n]: the output values
- bi: the impulse response of the filter at the ith instant, also the coefficient of the FIR filter in case of direct form.

Equation 2.2 is known as discrete convolution.

Fast convolution and partion convolution

Circular convolution Suppose two signals h(k) and x(k) are finite, $h(k) \neq 0$ for $0 \leq k < L$ and $x(k) \neq 0$ for $0 \leq k < M$. Then the linear convolution between h(k) and x(k) can be expressed as:

$$y(k) = \sum_{i=0}^{L-1} h(i)x(k-i), \qquad 0 \leq k < L+M-1 \tag{2.3}$$

As a result, the length of the output signal of the convolution between an L-point signal and an M-point signal is M + L - 1.

There is another form of convolution when the length of the result is equal to the length of the two operands called circular convolution. To define this form, a periodic extension of an N-point signal x(k) is used and is defined as follows:

$$x_p(k) \triangleq x[mod(k,N)] \tag{2.4}$$

Consequently, the periodic extension of x(k) extends x(k) in both positive and negative direction periodically.

With this periodic extension $x_p(k)$, the circular convolution of the N-point signals h(k) and x(k) which is denoted $y_c(k) = h(k) \;\; o \;\; x(k)$ can be defined as:

$$h(k) \;\; o \;\; x(k) \triangleq \sum_{i=0}^{N-1} h(i)x_p(k-i), \qquad 0 \leq k < N \tag{2.5}$$

The circular convolution can be calculated using matrix multiplication. Assume h and y_c are N x 1 column vectors which contain values of the signal h(k) and $y_c(k) = h(k) \;\; o \;\; x(k)$:

$$h = [h(0), h(1), ..., h(N-1)]^T \tag{2.6}$$

$$y_c = [y_c(0), y_c(1), ..., y_c(N-1)]^T \tag{2.7}$$

And C(x) is an N x N matrix whose columns are downward rotations of x(i):

$$C(x) = \begin{bmatrix} x(0) & x(N-1) & x(N-2) & \dots & x(1) \\ x(1) & x(0) & x(N-1) & \dots & x(2) \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ x(N-1) & x(N-2) & x(N-3) & \dots & x(0) \end{bmatrix} \tag{2.8}$$

Then the circular convolution can be expressed in vector form as:

$$y_c = C(x).h \tag{2.9}$$

Zero padding Since the circular convolution $y_c(k)$ is periodic with a period of N, it has a different response than linear convolution. However, there is a preprocessing step that can be performed in order to achieve linear convolution from circular convolution.

Suppose $h_z(k)$ and $x_z(k)$ with length of N = L + M - 1 are zero-padded versions of two signals L-point h(k) and M-point x(k):

$$h_z = [h(0), h(1), ..., h(L-1), 0, ..., 0], \qquad \text{(M-1 zeros at the end)} \tag{2.10}$$

$$x_z = [x(0), x(1), ..., x(M-1), 0, ..., 0], \qquad \text{(L-1 zeros at the end)} \tag{2.11}$$

If $x_{zp}(k)$ is the periodic extension of $x_z(k)$, then the circular convolution $y_c(k)$ is:

$$\begin{aligned} y_c(k) &= h_z(k) \;\; o \;\; x_z(k) \\ &= \sum_{i=0}^{N-1} h_z(i) x_{zp}(k-i) \\ &= \sum_{i=0}^{L-1} h_z(i) x_{zp}(k-i), \;\; 0 \le k < N \end{aligned} \tag{2.12}$$

In equation 2.12, the minimum value of k - i is at k = 0 and i = L - 1 (since $0 \le k < N$ and $0 \le i < L-1$) and equal to - (L - i). Because of L - 1 zeros padded to the end of $x_z(i)$, $x_{zp}(-i) = 0$ for $0 \le i < L$. Consequently, $x_{zp}(k-i)$ in equation 2.12 can be replaced by $x_z(i)$ which leads to linear convolution.

In conclusion, linear convolution of two signals can be calculated using circular convolution between the zero-padded versions of those signals:

$$h(k) \star x(k) = h_z(k) \;\; o \;\; x_z(k), \qquad 0 \leq k < N \tag{2.13}$$

Fast convolution From the properties of Discrete Fourier Transform, the circular convolution in time domain is equal to the multiplication in frequency domain:

$$DFT\{h_z(k) \;\; o \;\; x_z(k)\} = H_z(i)X_z(i) \tag{2.14}$$

As a result:

$$h_z(k) \;\; o \;\; x_z(k) = IDFT\{H_z(i)X_z(i)\}, \qquad 0 \leq k < N \tag{2.15}$$

If the length of zero-padded versions is chosen such that it is equal to the smallest integer power of two which is greater or equal to L + M - 1 (N = nextpow2(L + M - 1)). Then the Fast Fourier Transform can be used to compute the DFTs of $h_z(k)$ and $x_z(k)$ and the IDFT of $H_z(i)X_z(i)$. This results in the following version of linear convolution called fast convolution:

$$h_z(k) \;\; \star \;\; x_z(k) = IFFT\{H_z(i)X_z(i)\}, \;\; 0 \leq k < L + M - 1 \tag{2.16}$$

The total number of real floating point operations per second (FLOPS) required to perform a fast convolution of two L-point signals is:

$$n_{fast} = 12L.log_2(2L) + 8L + 4 \;\; FLOPs \tag{2.17}$$

While direct convolution requires:

$$n_{dir} = 2L^2, \;\; FLOPs \tag{2.18}$$

From table 2.1, for small values of L direct convolution will be faster than fast convolution. However, because the L^2 grows faster than the $L.log_2(2L)$ term, eventually the fast convolution will outperform direct convolution.

L	Fast convolution [FLOPs]	Direct convolution [FLOPs]
2	68	8
4	180	32
8	452	128
16	1092	512
32	2564	2048
64	5892	8192
128	13316	32768
256	29700	131072
512	65540	524288
1024	143364	2097152
2048	311300	8388608
4096	671748	33554432
8192	1441796	134217728
16384	3080196	536870912

Table 2.1.: Speed comparison between fast convolution and direct convolution

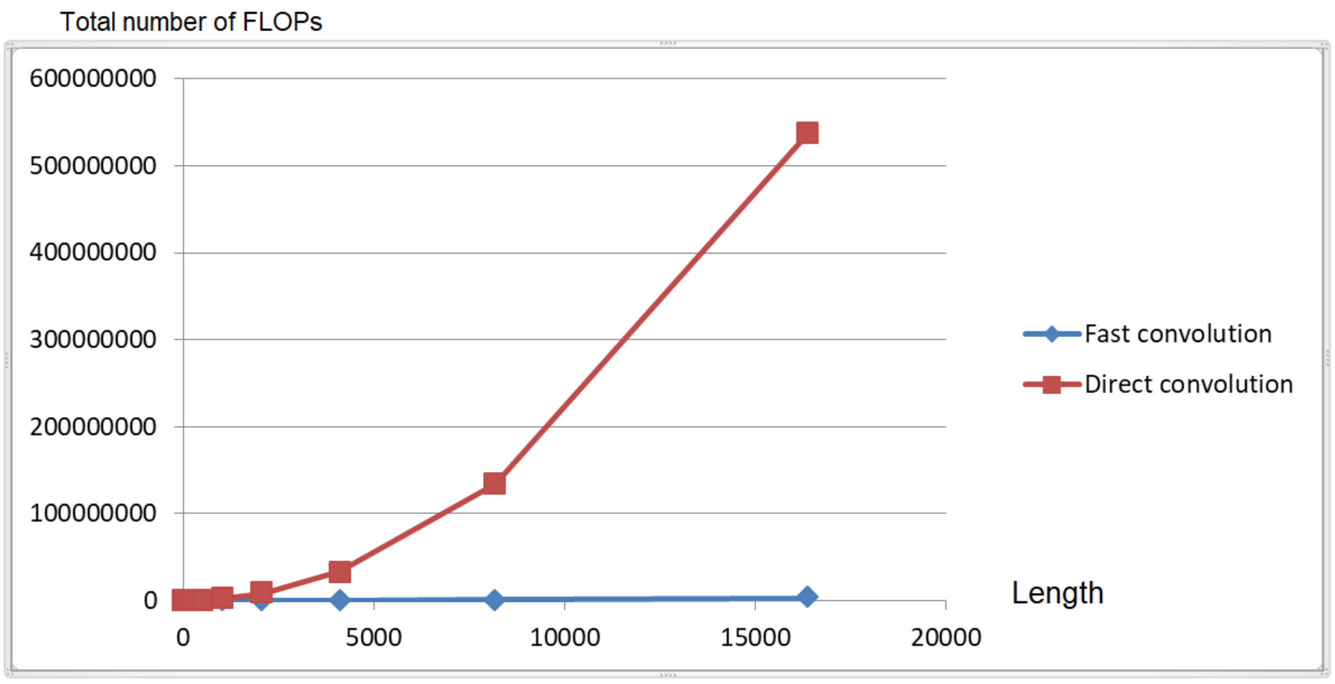

Figure 2.1.: Speed comparison between fast convolution and direct convolution

Fast block convolution In practical, the input signal length is often indefinite. In cases like these, the number of input samples M is very large, which leads to the computation of an FFT of length N = L + M + p may not be practical.

This problem can be solved using a technique called fast block convolution. The basic idea is to split the input into smaller parts of length L. Each part is convolved with the L-point impulse response h(k). When the results are combined correctly, the original (L + M - 1)-point convolution can be recovered.

Note that x(k) can be padded with up to L - 1 zeros, if needed, so that the length of the zero-padded input $x_z(k)$ is Q.L (Q is integer and Q > 1). Then $x_z(k)$ can be expressed as a sum of Q blocks of length L:

$$\begin{gathered} x_z(k) = \sum_{i=0}^{Q-1} x_i(k - iL), \quad 0 \le k < M \\ with\ x_i(k) \triangleq \begin{cases} x(k+iL), & 0 \le k < L \\ 0, & otherwise \end{cases} \end{gathered} \tag{2.19}$$

Using those equations above, the linear convolution between x(k) and h(k) is then:

$$h(k) \star x(k) = \sum_{i=0}^{Q-1} h(k) \star x_i(k - iL) = \sum_{i=0}^{Q-1} y_i(k - iL) \tag{2.20}$$

$y_i(k)$ is the convolution of h(k) with the ith subsignal $x_i(k)$. That is,

$$y_i(k) = h(k) \star x_i(k), \quad 0 \le i < Q \tag{2.21}$$

If h(k) and $x_i(k)$ are padded in such a way that the length of the zero-padded versions are the power of two, then FFT can be used. The results must then be shifted and added as in 2.20. The resulting procedure, known as overlap-add method of block convolution, is summarized in the following algorithm: [Schilling und Harris (2015)]

1. Compute
   - $M_{save} = M$
   - $r = L - mod(M, L)$
   - $M = M + r$
   - $x_z = [x(0), ..., x(M_{save} - 1), 0, ..., 0]^T \in R^M$

- $Q = M/L$
- $N = 2^{ceil[log_2(2L-1)]}$
- $h_z = [h(0), ..., h(L-1), 0, ..., 0]^T \in R^N$
- $H_z = FFT\{h_z(k)\}$
- $y_0 = [0, ..., 0]^T \in R^{L(Q-1)+N}$

2. For i = 0 to Q - 1 compute
   - $x_i(k) = x_z(k + iL), 0 \leq k < L$
   - $x_{iz}(k) = [x_i(0), ..., x_i(L-1), 0, ..., 0]^T \in R^N$
   - $X_{iz} = FFT\{x_{iz}(k)\}$
   - $y_i(k) = IFFT\{H_z(i)X_{iz}(i)\}$
   - $y_0(k) = y_0(k) + y_i(k - Li), Li \leq k < Li + 2N - 1$
3. Set
   - $y(k) = y_0(k), 0 \leq k < L + M_{save} - 1$

Note that in step 1, the original number of input samples is saved in M_{save}.

Partioned convolution [Battenberg und Avizienis (2011)]

Partitioned convolution is a technique for efficiently performing time-domain convolution with low inherent latency. It is particularly useful for computing the convolution of audio signals with long (>1 second) impulse responses in real time.

There are two types of partitioned convolution: uniform partitioned convolution and non-uniform partitioned convolution. In thesis's scope, only the uniform partitioned convolution is focused.

In order to obtain a compromise between computational efficiency and latency, the impulse response is partitioned into a series of smaller sub-filters which can be run in parallel with appropriate delays inserted. Each sub-filter's output is computed using a block convolution method, and the outputs of all sub-filters are summed to produce a block of the output signal.

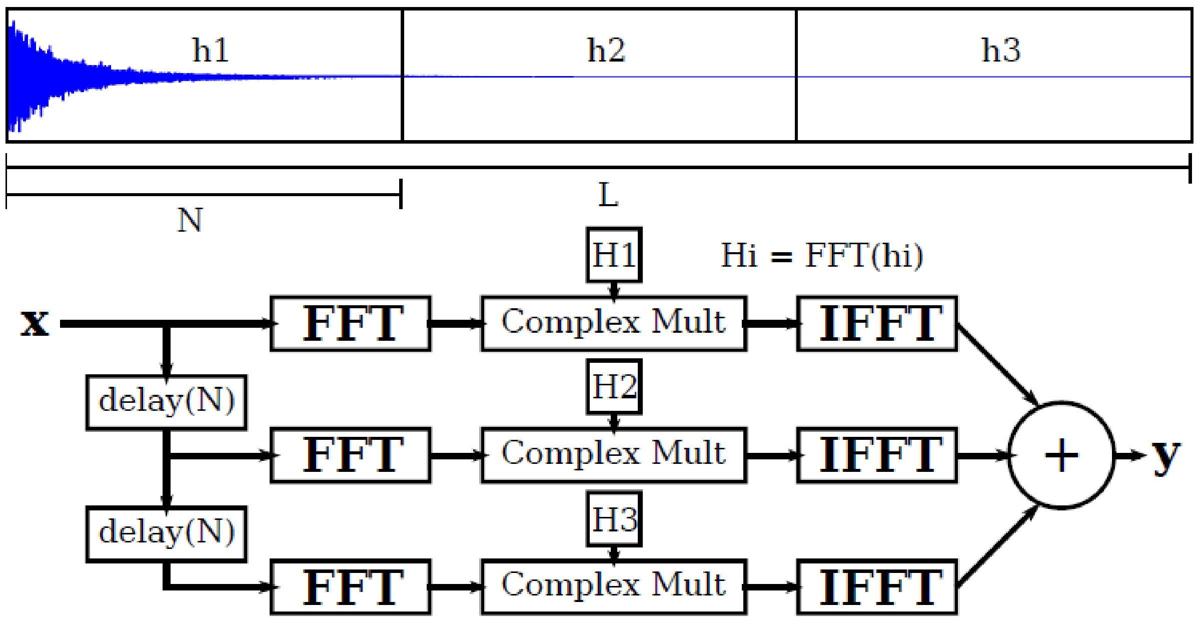

Figure 2.2.: Top - Partitioning of an impulse response into 3 parts. Bottom - Steps involved in computing the above 3-part uniform partitioning.

If the original length-L filter is partitioned into sub-filters of size N, O(logN) operations per sub-filter per output sample are performed but the latency is reduced from L to N.

Within this uniform partitioning scheme, previous computed forward FFTs can be saved for reusing in subsequent sub-filters. Additionally, according to the linearity property of FFT, the complex frequency domain output of each sub-filter can be summed up before taking the inverse FFT, which reduces the number of inverse FFTs.

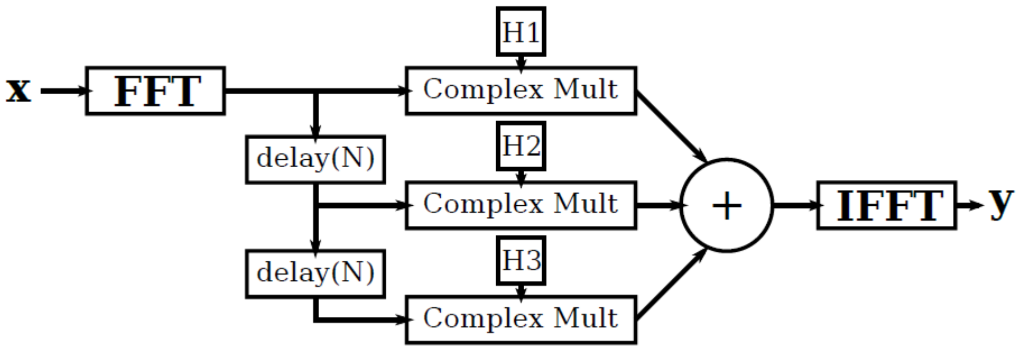

Figure 2.3.: Frequency-domain Delay Line (FDL).

3. Design

3.1. Advanced Linux Sound Architecture (ALSA)

[ALSA (2017)]

The Advanced Linux Sound Architecture (ALSA) provides audio and MIDI functionality to the Linux operating system. ALSA has the following significant features:

- Efficient support for all types of audio interfaces, from consumer sound cards to professional multichannel audio interfaces.
- Fully modularized sound drivers.
- SMP and thread-safe design.
- User space library (alsa-lib) to simplify application programming and provide higher level functionality.
- Support for the older Open Sound System (OSS) API, providing binary compatibility for most OSS programs.

3.2. JACK Audio Connection Kit - infrastructure for audio applications

[JACK-Audio-Connection-Kit (2017a)]

JACK (JACK Audio Connection Kit) refers to an API and two implementations of this API, jack1 and jack2. It provides a basic infrastructure for audio applications to communicate with each other and with audio hardware. Through JACK, users are enabled to build powerful systems for signal processing and music production.[quoted from http://jackaudio.org/faq/about.html].

In this thesis, JACK acts as an Audio Server which receives the audio signals from its input ports, passes the signals though an Audio Client and delivers the output signals from the client to its output ports in real time. Note that the sampling rate and audio format can be configured using JACK.

The Audio Client mentioned above is the FIR filter which is implemented using normal convolution or partitioned convolution with BruteFIR.

3.3. Normal FIR filter client (normal convolution)

JACK provides a library built on C programming language for building audio application (which acts as an Audio Client). With this library, an FIR filter can be built. The FIR filter receives the audio samples from the input through JACK, then convolves them with the FIR coefficients and send the processed signals to the output though JACK again.

This application uses normal linear convolution to implement the filter, in order to compare the efficiency between it and the partitioned convolution.

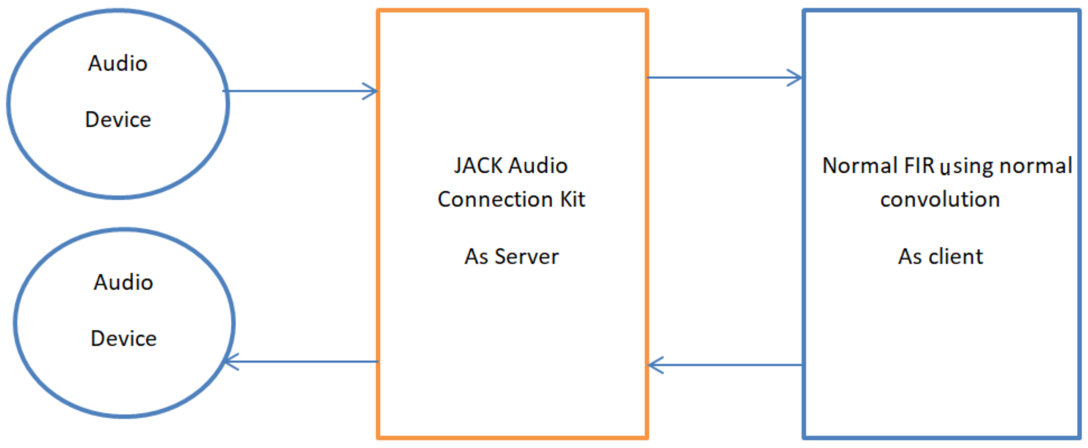
Audio Device
Audio Device
JACK Audio Connection Kit
As Server
Normal FIR using normal convolution
As client

Figure 3.1.: Direct convolution FIR filter client.

3.4. FIR filter using fast block convolution and partioned convolution with BruteFIR

BruteFIR is a software convolution engine, a program for applying long FIR filters to multi-channel digital audio, either offline or in real time. Its basic operation is specified through a configuration file, and filters, attenuation and delay can be changed in runtime through a simple command line interface.

BruteFIR uses the high-performance FFTW library for the Fast Fourier Transform calculations. Partitioned convolution is also included in BruteFIR which makes the FFTs shorter.

BruteFIR provides a module named JACK I/O module. With this module, BruteFIR can be connected to JACK Audio Connection Kit as a client.

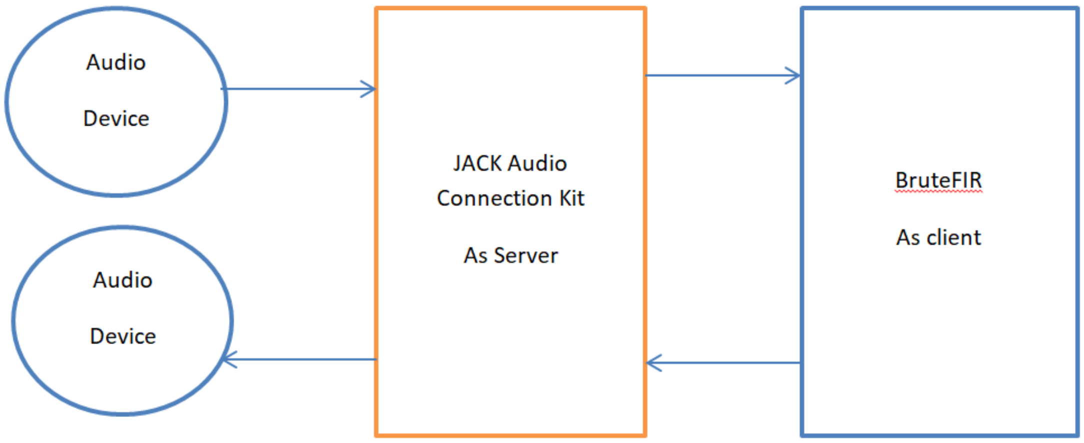

Figure 3.2.: BruteFIR as a client.

4. Implementation

4.1. JACK Audio Connection Kit - Installation

JACK is already included in the image file from Cirrus Logic Audio Card (see Appendix part "Cirrus Logic Audio Card - Install software to run on Raspberry Pi") so there is no need to install it again.

However, there are a few things need to be configured before using JACK:

1. Download a kernel with FLL patch included (from this link: https://drive.google.com/uc?export=download&id=0BzIaxMH3N5O1S1JyTkJ4Z090cHc or from attached DVD). File name is "linux-image-3.18.9-v7cludlmmapfll_3.18.9-v7cludlmmapfll-4_armhf.deb" then run the following command to install it:
   ```
   sudo dpkg -i linux-image-3.18.9-v7cludlmmapfll_3.18.9-v7cludlmmapfll-4_armhf.deb
   ```
2. Change /boot/config.txt to contain
   ```
   kernel=vmlinuz-3.18.9-v7cludlmmapfll
   ```
3. To be able to write the client code, JACK library need to be installed:
   ```
   sudo apt-get install libjack-jackd2-dev
   ```

After everything is installed, JACK server can be started with the following command:

```
jackd -r -d alsa -d hw:0 -r [rate] -p [period]
```

- rate: Specify the sample rate. The default is 48000.
- Period: Specify the number of frames between JACK process() calls. This value must be a power of 2, and the default is 1024.

4.2. Direct convovultion FIR filter

4.2.1. Circular buffer

Circular buffer is a data structure which uses a single, fixed size array to serve as a First In First Out (FIFO) buffer. The circular buffer usually has two indices, start index and end index (or head and tail). These indices do not point to any fix position. When a new element is inserted, the end index is increased by one, or become zero if it reaches the last position. Similarly, when an element is removed, the start index is increased by one, or become zero when it reaches the last position.

This circular buffer is used to store all values of the audio input samples. The FIFO characteristic is very suitable to calculate the convolution between the input signal and the filter coefficients.

There are many different ways to implement the structure of the circular buffer depends on different purposes. For the FIR implementation purpose, its structure can be:

```
// Structure of the circular buffer
typedef struct ring_buffer {
  double values[SAMPLEFILTER_TAP_NUM]; // Store values of the input signal
  int size;// The current size of the circular buffer (the number of elements)
  int start; // The start index of the circular buffer.
} ring_buffer;

// Initial state of the buffer
void init(ring_buffer *buffer) {
  buffer->size = SAMPLEFILTER_TAP_NUM;
  buffer->start = 0;
}
```

Listing 4.1: Circular buffer implementation - structure

The end index is not included in the structure above since it can be calculated using the start index and the number of elements in the buffer. The maximum size of the buffer is equal to the number of the FIR filter's coefficient to make it easier for the convolution, note that the buffer is filled with zeros at the beginning.

There are also two functions which can be performed on this buffer: push a new element on the top and pop an element from the bottom. The start index and buffer's current size are recalculated accordingly.

```
void push(ring_buffer *buffer, double value) {
  if (isFull(buffer))
```

```
    printf("buffer␣overflow");
  else {
    int index = (buffer->start + buffer->size) % SAMPLEFILTER_TAP_NUM;
    buffer->values[index] = value;
    buffer->size++;
  }
}

double pop(ring_buffer *buffer) {
  if (isEmpty(buffer)) {
    printf("Buffer␣is␣empty");
    return 0;
  } else {
    double value = buffer->values[buffer->start];
    buffer->start = (buffer->start >= SAMPLEFILTER_TAP_NUM - 1) ?
                    0 : buffer->start + 1;
    buffer->size--;
    return value;
  }
}
```

Listing 4.2: Circular buffer implementation - functions

Additionally, this version of circular buffer has a function to get the value of an element stored in the buffer using its index. This index is the index calculated based on the distance from the start index, not from zero index:

```
double getFromIndex(ring_buffer *buffer, int index) {
  return buffer->values[(buffer->start + index) % SAMPLEFILTER_TAP_NUM];
}
```

4.2.2. Direct convolution between the FIR filter's coefficients and the audio samples

From the direct convolution equation 2.2, one output sample (filtered_value) can be calculated in C code as follow:

```
for (i = 0; i < SAMPLEFILTER_TAP_NUM; i++) {
  filtered_value +=
    getFromIndex(&sample_buffer, buffer_index--) * filter_taps[i];
}
```

Listing 4.3: Direct convolution implementation

The input signal is stored in sample_buffer while the filter coefficients are stored in filter_tap. These two arrays have the same length and equal to the number of coefficients to make it easier to calculate the convolution. Note that the sample_buffer is filled with zero at the very first step.

Each time a new sample come, the oldest sample is popped out and the new one is pushed into the sample_buffer. After that, a new output sample can be calculated using the for loop above.

To get the input audio samples, JACK provides a callback function called "process" which is executed repeatedly until JACK is stopped. Each time this function is executed, it delivers one period of the sampled audio input. The FIR filtering code must be written inside this function to be able to handle the input and output audio samples.

(From ALSA's perspective, a period is associated with the number of ALSA frames captured/played between two hardware interrupts by the audio interface. And a frame is a single (PCM) value that represents the amplitude of the sampled audio input signal at a given point in time, on a single channel. This period can be adjusted via command line which is used to execute JACK).

The output after being processed by the FIR filter also has to be delivered in a period. In other word, the output samples cannot be delivered one by one, but have to be in bulk. For example: if there are 1024 input samples are taken each time the "process" functions is executed, then 1024 output samples must be prepared before they can be sent to the output port.

This is the pseudo code for the FIR filter:

1. Initialize the length sample_buffer to be the same as filter_taps.
2. For each time "process" run:
   a) Get one period of audio samples (the number of audio samples is NFRAMES)
   b) Initialize the length of the filtered samples (filtered_samples) to be equal to NFRAMES.
   c) Loop from 0 to NFRAMES:
      - Pop out the oldest element from sample_buffer.
      - Push new audio sample from input into sample_buffer.
      - Calculate the ouput sample using for loop in code 4.3
      - Add the output into filtered_samples array.

3. Delivier the output in filtered_samples array.

This is the implementation code:

```
int
process (jack_nframes_t nframes, void *arg)
{
  jack_default_audio_sample_t
      *in, // Input samples
      *out, // The output will be sent to JACK
      *filtered_samples; // The output processed by the filter,
                         // will be copied to *out

  // Get the input samples
  in = jack_port_get_buffer (input_port, nframes);

  // Initialze filtered_sample length (=nframes)
  filtered_samples = malloc(sizeof (jack_default_audio_sample_t) * nframes);

  // Initialze the output
  out = jack_port_get_buffer (output_port, nframes);

  int i;

  int buffer_index,
  input_samples_index = 0,
  filtered_samples_index = 0;

  double filtered_value;

  while (filtered_samples_index < NFRAMES) {

    buffer_index = SAMPLEFILTER_TAP_NUM - 1;
    filtered_value = 0;

    // Pop the oldest element from sample_buffer
    pop(&sample_buffer);

    // Push new sample from input to the sample_buffer
    push(&sample_buffer, in[input_samples_index++]);

    // Calculate the output sample - convolution
    for (i = 0; i < SAMPLEFILTER_TAP_NUM; i++) {
      filtered_value +=
        getFromIndex(&sample_buffer, buffer_index--) * filter_taps[i];
    }

    // Add the calculated sample into the output array
    filtered_samples[filtered_samples_index++] = filtered_value;

  }

  // Copy filtered_sample to output of JACK
  memcpy (out, filtered_samples,
  sizeof (jack_default_audio_sample_t) * nframes);

  return 0;
}
```

Listing 4.4: Direct convolution FIR filter implementation

To build the client, execute command:

```
gcc -Wall [filename.c] -o [filename] ‘pkg-config -cflags -libs jack‘
```

4.3. BruteFIR

4.3.1. Installation

Installing Brute FIR is as simple as running these two commands:

```
sudo apt-get update
sudo apt-get install brutefir
```

4.3.2. Configuration

Brute FIR can be connected as a client to JACK using its module named JACK I/O.

To do that, simply overwrite file ".brutefir_defaults" as follow:

```
## DEFAULT GENERAL SETTINGS ##

float_bits: 32;              # internal floating point precision
sampling_rate: 44100;        # sampling rate in Hz of audio interfaces
filter_length: 65536;        # length of filters
config_file: "~/.brutefir_config"; # standard location of main config file
overflow_warnings: true;     # echo warnings to stderr if overflow occurs
```

```
show_progress: true;         # echo filtering progress to stderr
max_dither_table_size: 0;    # maximum size in bytes of precalculated dither
allow_poll_mode: false;      # allow use of input poll mode
modules_path: ".";           # extra path where to find BruteFIR modules
monitor_rate: true;         # monitor sample rate
powersave: false;            # pause filtering when input is zero
lock_memory: true;           # try to lock memory if realtime prio is set
sdf_length: −1;              # subsample filter half length in samples
convolver_config: "~/.brutefir_convolver"; # location of convolver config file

## COEFF DEFAULTS ##

coeff {
  format: "TEXT";      # file format
  attenuation: 0.0;    # attenuation in dB
  blocks: −1;          # how long in blocks
  skip: 0;             # how many bytes to skip
  shared_mem: false;   # allocate in shared memory
};

## INPUT DEFAULTS ##

input {
  device: "jack" {};   # module and parameters to get audio
  sample: "S32_LE";    # sample format
  channels: 2/0,1;     # number of open channels / which to use
  delay: 0,0;          # delay in samples for each channel
  maxdelay: −1;        # max delay for variable delays
  subdelay: 0,0;       # subsample delay in 1/100th sample for each channel
  mute: false,false;   # mute active on startup for each channel
};

## OUTPUT DEFAULTS ##

output {
  device: "jack" {};   # module and parameters to put audio
  sample: "S32_LE";    # sample format
  channels: 2/0,1;     # number of open channels / which to use
  delay: 0,0;          # delay in samples for each channel
  maxdelay: −1;        # max delay for variable delays
  subdelay: 0,0;       # subsample delay in 1/100th sample for each channel
  mute: false,false;   # mute active on startup for each channel
  dither: false;       # apply dither
};
```

```
## FILTER DEFAULTS ##

filter {
  process: -1;         # process index to run in (-1 means auto)
  delay: 0;            # predelay, in blocks
  crossfade: false;    # crossfade when coefficient is changed
};
```

Listing 4.5: Content of file .brutefir_defaults

Basically, there are five important sections in the configuration file need to be rewrite (most of the values are provided by JACK by default):

- General settings. Here the general parameters for BruteFIR is set up.
- Coefficient settings. Parameters for files where-from filter coefficients are loaded.
- Input settings. Settings for digital audio inputs.
- Output settings. Settings for digital audio outputs.
- Filter settings. Parameters for the FIR filters.

File 4.5 contains all the default configurations. When Brute FIR is executed, those configurations will take effect unless there is any of them is overwritten in a custom configuration file named ".brutefir_config". In that case, the custom configurations will take effect instead of the old ones.

This is one example of ".brutefir_config":

```
sampling_rate: 44100;
filter_length: 2048;
modules_path: ".";
convolver_config: "~/.brutefir_convolver";

coeff "FIR␣filter␣coefficients" {
  filename: "coeffs.txt";
  format: "TEXT";
};

input "left", "right" {
  device: "jack" {ports: "system:capture_1", "system:capture_2";};
  sample: "AUTO";
  channels: 2;
};
```

```
output "left", "right" {
  device: "jack" {ports: "system:playback_1", "system:playback_2";};
  sample: "AUTO";
  channels: 2;
};

filter "FIR␣filter" {
  process: −1;
  delay: 0;
  inputs: "left", "right";
  outputs: "left", "right";
  coeff: "FIR␣filter␣coefficients";
};
```

Listing 4.6: Content of file .brutefir_config

File 4.6 defines filter length of 2048 taps instead of 65536 taps in the default configuration, with sampling rate of 44100. All the coefficients are stored in a text file named "coeffs.txt". The input and output are connected to JACK input ports (system:capture_1, system:capture_2) and output ports (system:playback_1, system:playback_2) on 2 channels.

4.3.3. Running

To run the Brute FIR client, simply run command:
`BruteFIR`

5. Testing

5.1. Frequency response of Normal FIR filter

Figure 5.1 is the amplitude response of a low-pass FIR filter using direct convolution with:

- Passband: from 0 to 400 Hz
- Stopband: from 600 to 22050 Hz
- 257 taps

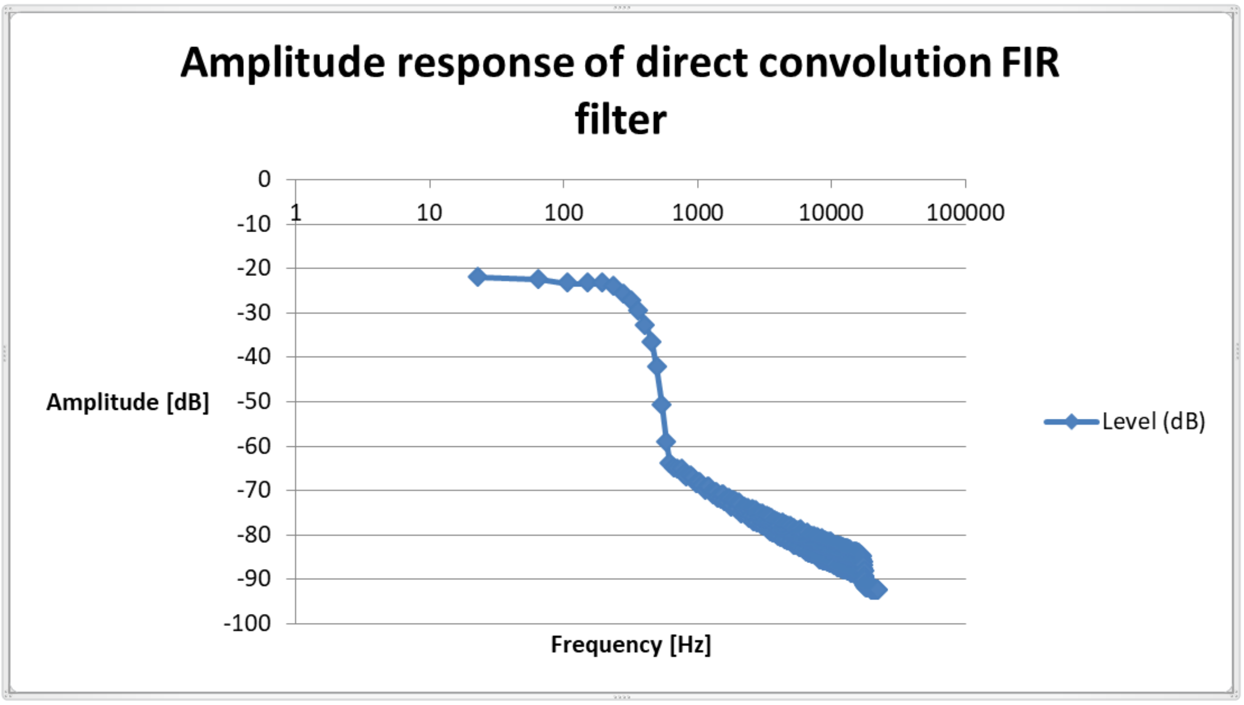

Figure 5.1.: Amplitude response of the direct convolution filter

5.2. Frequency response of BruteFIR

BeFigure 5.2 low is the amplitude response of a low-pass FIR filter using BruteFIR with:

- Passband: from 0 to 400 Hz
- Stopband: from 600 to 22050 Hz
- 257 taps

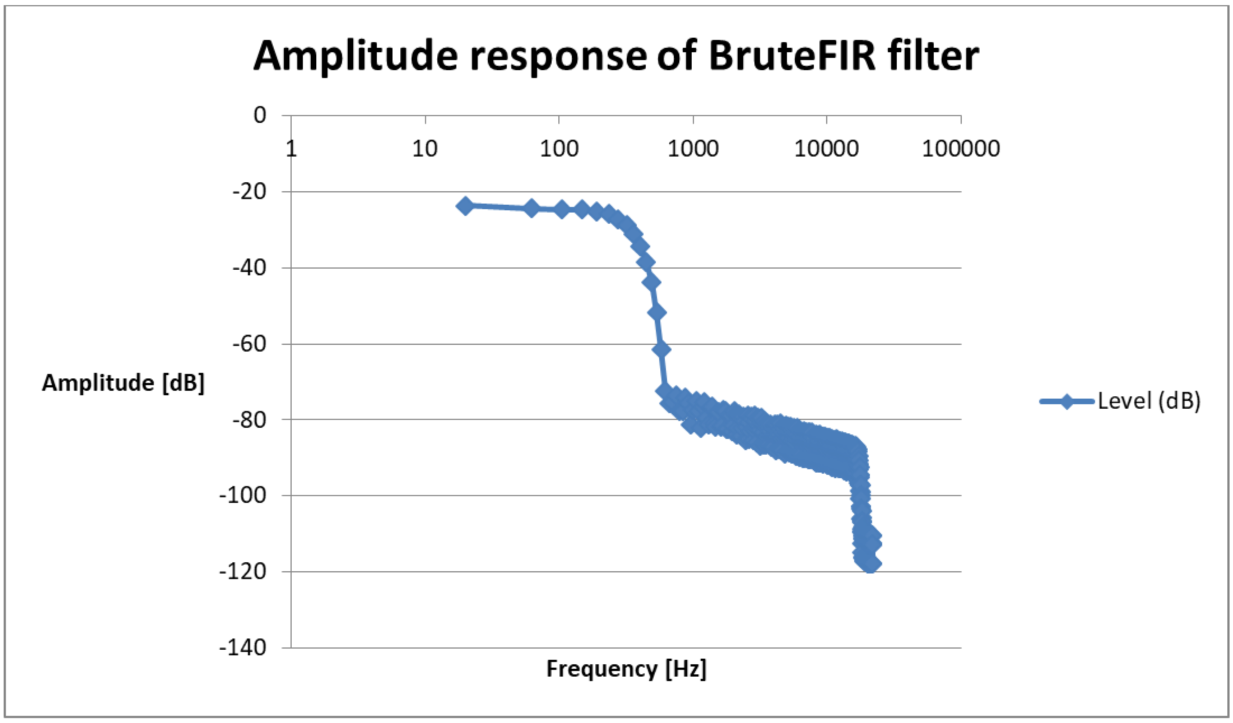

Figure 5.2.: Amplitude response of BruteFIR filter

5.3. Sampling time of direct convolution filter

As the sampling frequency is 44100 Hz, this filter can only work in real time if the number of taps is smaller than 257.

taps	t in second
10	1.78711E-06
20	3.06641E-06
30	4.48242E-06
40	5.46875E-06
50	6.49414E-06
60	7.79297E-06
70	8.54492E-06
80	7.49024E-06
90	8.08594E-06
100	8.84766E-06
110	9.67773E-06
120	1.10165E-05
130	1.1709E-05
140	1.21875E-05
150	1.28809E-05
160	1.4248E-05
170	1.46289E-05
180	1.56641E-05
190	1.65137E-05
200	1.71582E-05
210	0.000018125
220	1.86816E-05
230	1.94922E-05
240	2.14258E-05
250	2.11816E-05
255	2.23828E-05
256	2.01465E-05
257	2.23145E-05

Table 5.1.: Sampling time of direct convolution filter

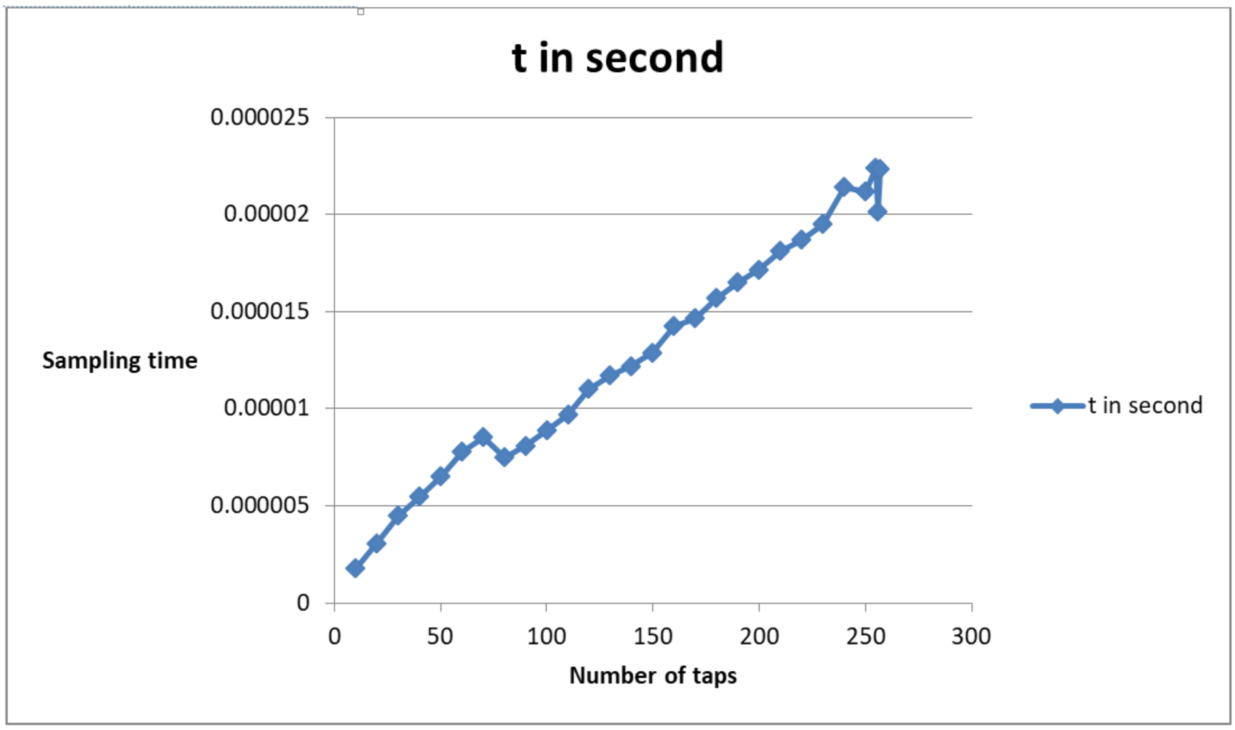

Figure 5.3.: Sampling time of direct convolution filter

5.4. Sampling time of BruteFIR filter

5.4.1. Sampling time with 1 partion

number of taps with 1 partition	Sampling time in second
1024	1.80176E-06
2048	3.39453E-06
4096	8.20215E-06
8192	1.46875E-05
16384	4.70068E-05

Table 5.2.: Sampling time of BruteFIR with 1 partion

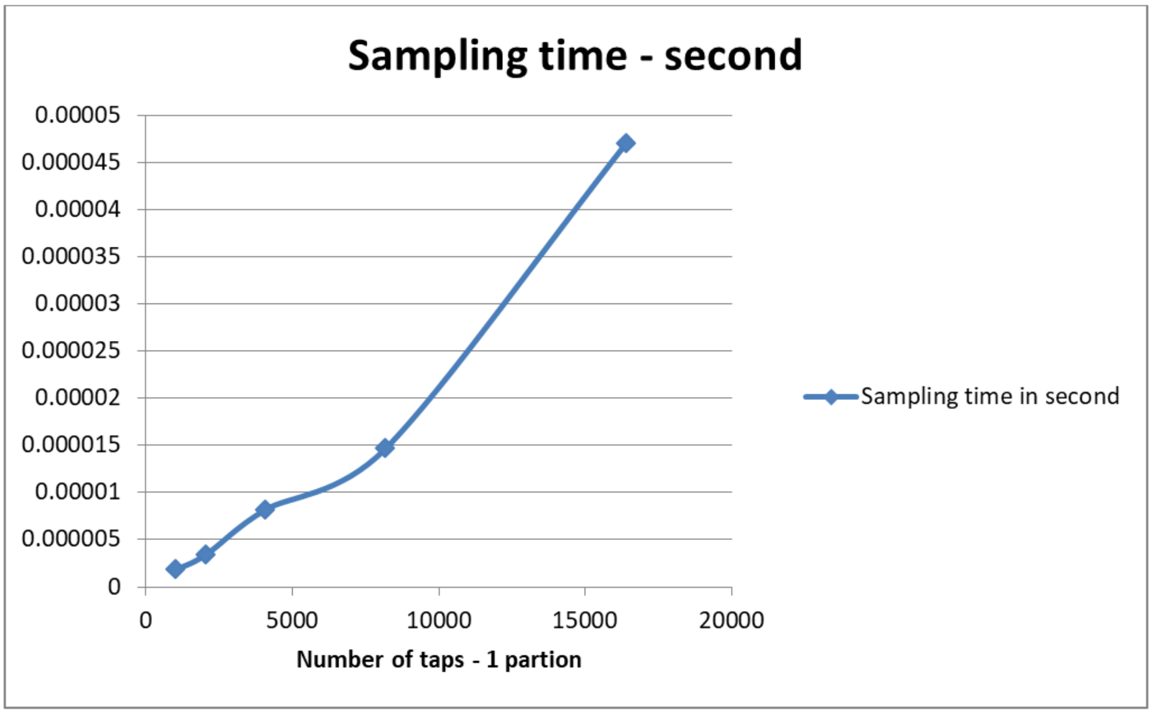

Figure 5.4.: Sampling time of BruteFIR with 1 partion

5.4.2. Sampling time with various partions

Number of partions with 16384 taps	Sampling time
1	4.70068E-05
2	1.81133E-05
4	9.91504E-06
8	5.2998E-06
16	2.38086E-06

Table 5.3.: Sampling time of BruteFIR filter with various partions

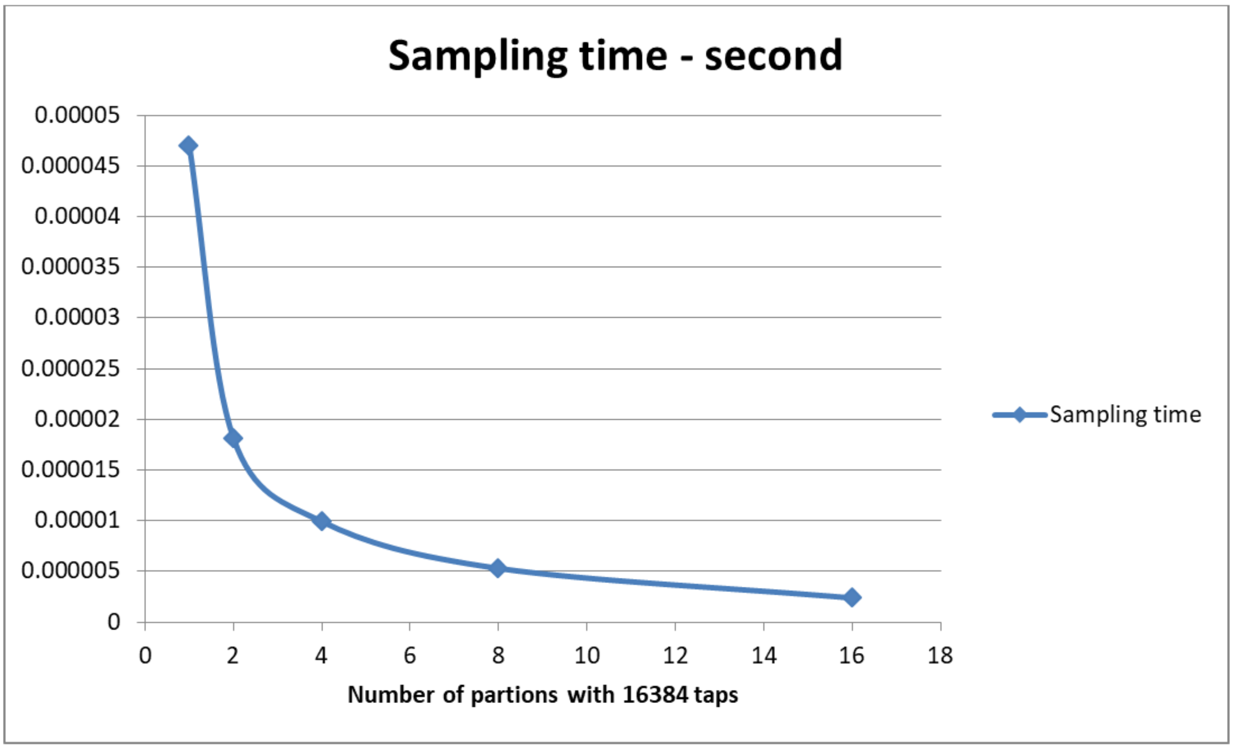

Figure 5.5.: Sampling time of BruteFIR filter with various partions

It can be observed from table 5.3 that, with 1 partion, BruteFIR can work in real time only if the number of taps is from 8192 and below. However, when the number of partions increase, the sampling speed is increased significantly.

5.4.3. Sampling time comparision between Normal FIR filter and Brute FIR filter

number of taps with 1 partition	BruteFIR sampling time [s]	Normal FIR sampling time [s]
1024	0.001801758	0.086523436
2048	0.003394531	0.173242182
4096	0.008202148	0.358691424
8192	0.0146875	0.737890601
16384	0.047006836	1.409505248

Table 5.4.: Sampling time comparison between Normal FIR filter and Brute FIR filter

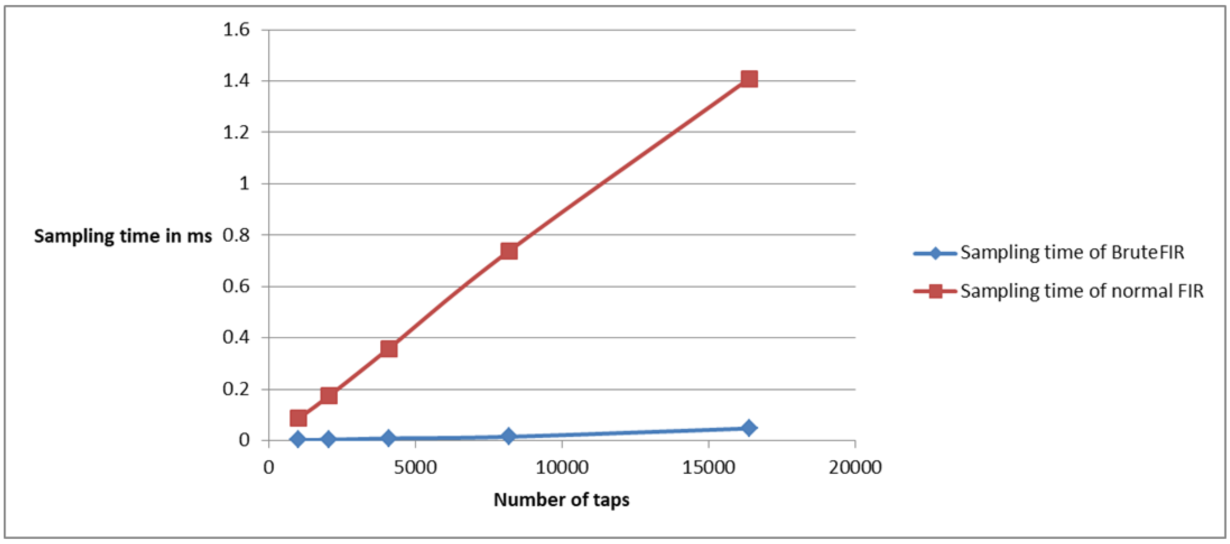

Figure 5.6.: Sampling time comparison between Normal FIR filter and Brute FIR filter

From the data in table 5.4, it can be observed that the speed of Brute FIR using FFT convolution techniques is must faster than the speed of the normal FIR filter using direct convolution.

5.4.4. Sampling time comparision between each number of taps and their partions

From table 5.5 and figure 5.7, with the same number of taps, the filter works faster if it has more partitions. But there is an optimize number of partitions, if the number of partitions are larger than this, then the filter will work slower.

Number of partitions	Number of Partitions (power of 2)	1024 taps	2048 taps	4096 taps	8192 taps	16384 taps	32768 taps	65536 taps	131072 taps	262144 taps	524288 taps	1048576 taps
1	0	0.001295	0.003774	0.008379	0.015081	0.039276367	0.0822979	0.2007305	0.55026953125	1.34624707	2.309982422	5.282570313
2	1		0.001696	0.004161	0.008644	0.014521484	0.043625	0.0847275	0.19821875000	0.63208496	1.29884668	2.429537109
4	2			0.002043	0.00355	0.009042969	0.0225254	0.0402295	0.08324511719	0.20308203	0.6015	1.396071289
8	3				0.002643	0.005314453	0.0121563	0.0190869	0.04019238281	0.08618164	0.220908203	0.568103516
16	4					0.002311523	0.0061494	0.014291	0.02433691406	0.05058691	0.11949707	0.206493164
32	5						0.0044316	0.0085332	0.01895507813	0.03375391	0.080707031	0.115533203
64	6							0.0071689	0.01433203125	0.02975293	0.053898438	0.10690918
128	7								0.01374316406	0.03012891	0.049848633	0.092543945
256	8									0.02409082	0.040986328	0.089209961
512	9										0.040871094	0.099810547
1024	10											0.088123047

Table 5.5.: Sampling time comparision between each number of taps and their partions

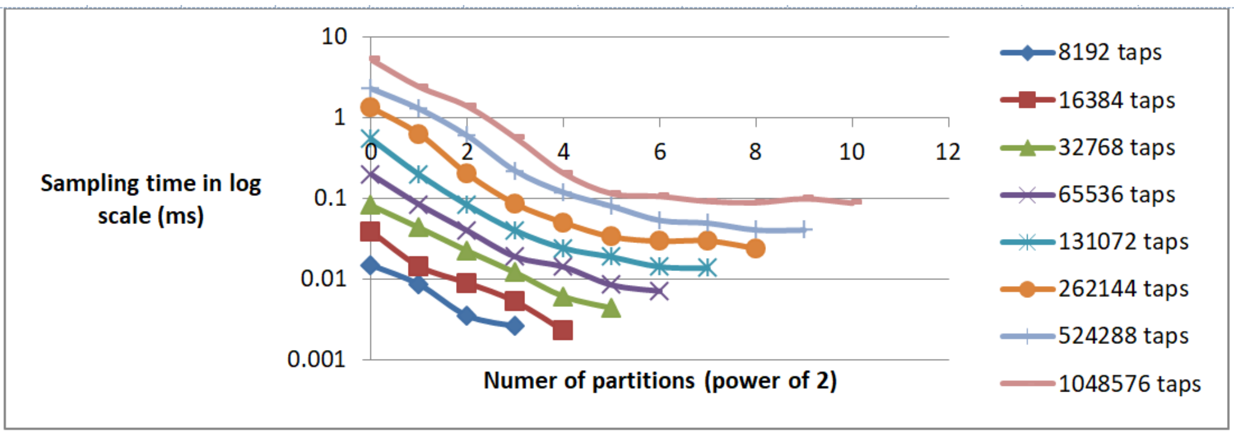

Figure 5.7.: Sampling time comparision between each number of taps and their partions

5.4.5. Optimum number of partitions for each number of taps

Number of taps	optimum partiontions
1024	1
2048	2
4096	4
8192	8
16384	16
32768	32
65536	64
131072	128
262144	256
524288	512
1048576	1024

Table 5.6.: Optimum number of partitions for each number of taps

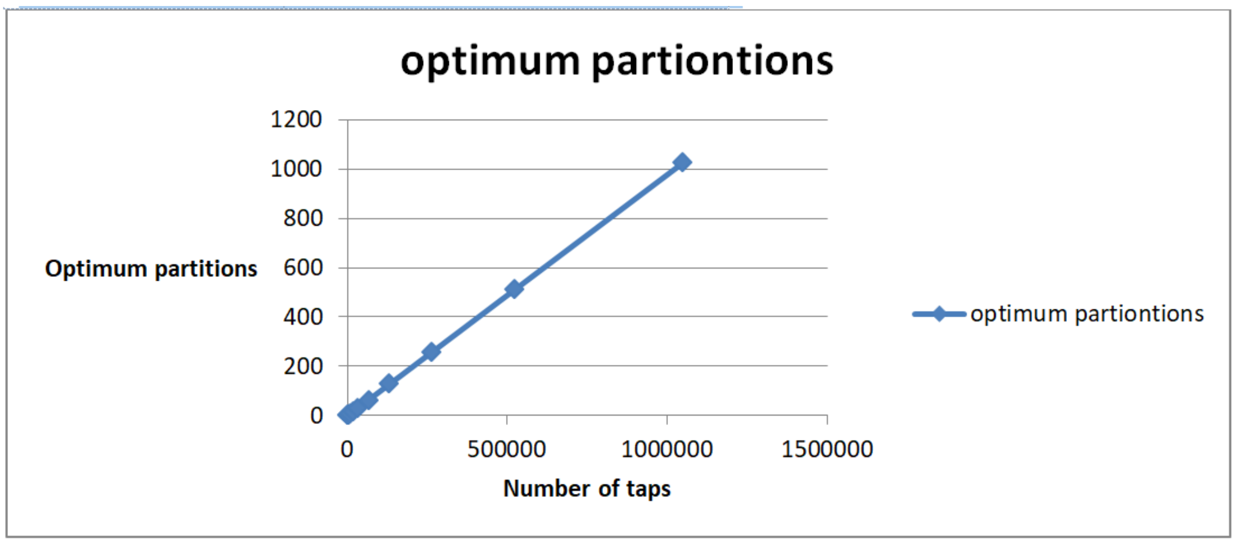

Figure 5.8.: Optimum number of partitions for each number of taps

6. Application

6.1. Measure the frequency response of a given loud speaker

To measure the frequency response of a loud speaker, several equipment are used:

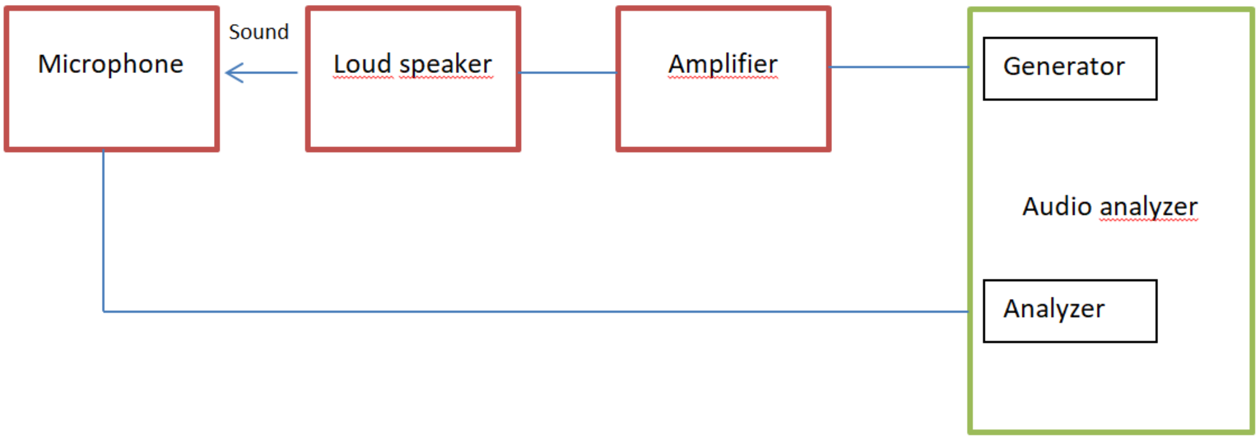

Figure 6.1.: Equipments setup layout for frequency response measurement

The audio analyzer generates the frequency from 0 Hz to 20 kHz to the loud speaker. The micro phone records the sound from the loud speaker and passes it to the analyzer.

Result:

As it is shown from figure 6.2, the amplitude response of this loud speaker is not constant. For that reason, a compensating FIR filter can be attached in between in order to make the amplitude response of this loud speaker become constant - or a flat line in the graph.

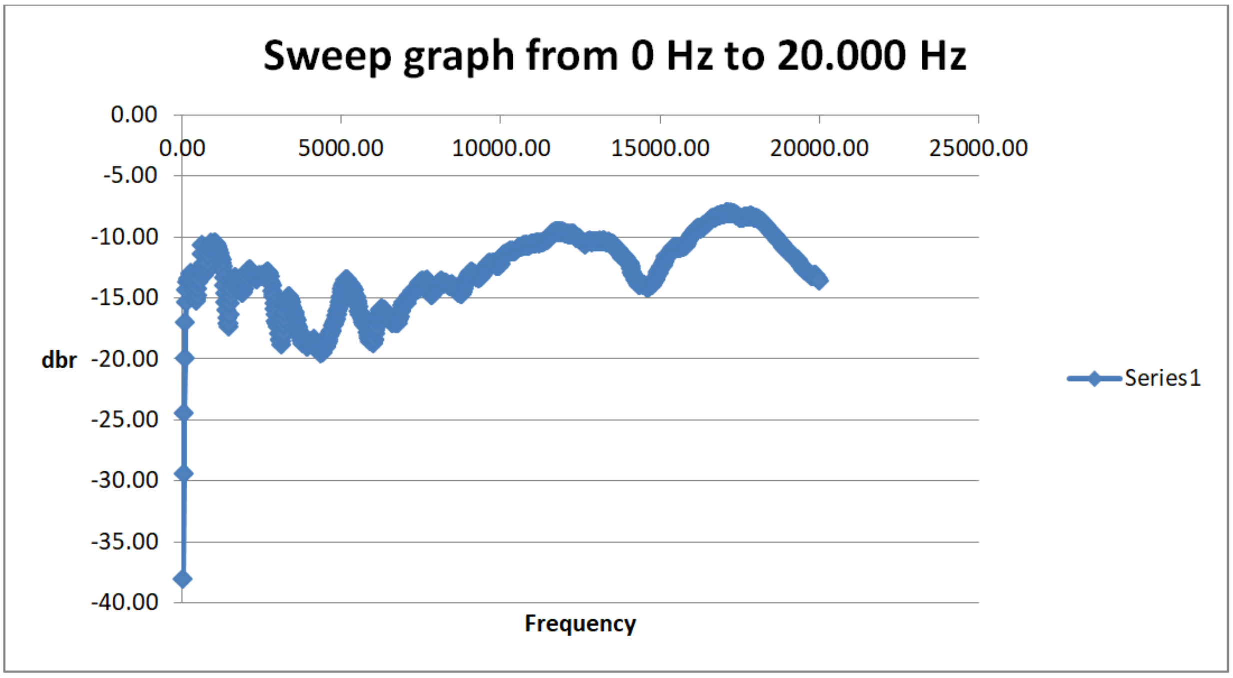

Figure 6.2.: Measured amplitude response of the loud speaker

6.2. Amplitude response compensatiing FIR filter

The purpose of this filter is to compensate the amplitude response of a loud speaker to make it a constant. Assume that $H_{comp}(j\omega)$ is the frequency response of the compensating filter and $H(j\omega)$ is the frequency response of the loud speaker, then this equation must be sastified:

$$H_{comp}(j\omega).H(j\omega) = a \tag{6.1}$$

Frequency response of the filter must be:

$$H_{comp}(j\omega) = \frac{a}{H(j\omega)} \tag{6.2}$$

Amplitude response of the filter:

$$A_{comp}(j\omega) := 20lg(|H_{comp}(j\omega)|) = 20lg(|\frac{a}{H(j\omega)}|) = 20lg(a) - A(j\omega) \tag{6.3}$$

With $A(j\omega)$ is the amplitude response of the loud speaker, can be got from audio analyzer machine.

After the amplitude response of the filter is calculated, its frequency response and coefficients can then be calculated also.

Those coefficients can be used by Brute FIR and run on the Raspberry Pi to act as the compensating filter for the loud speaker.

Result:

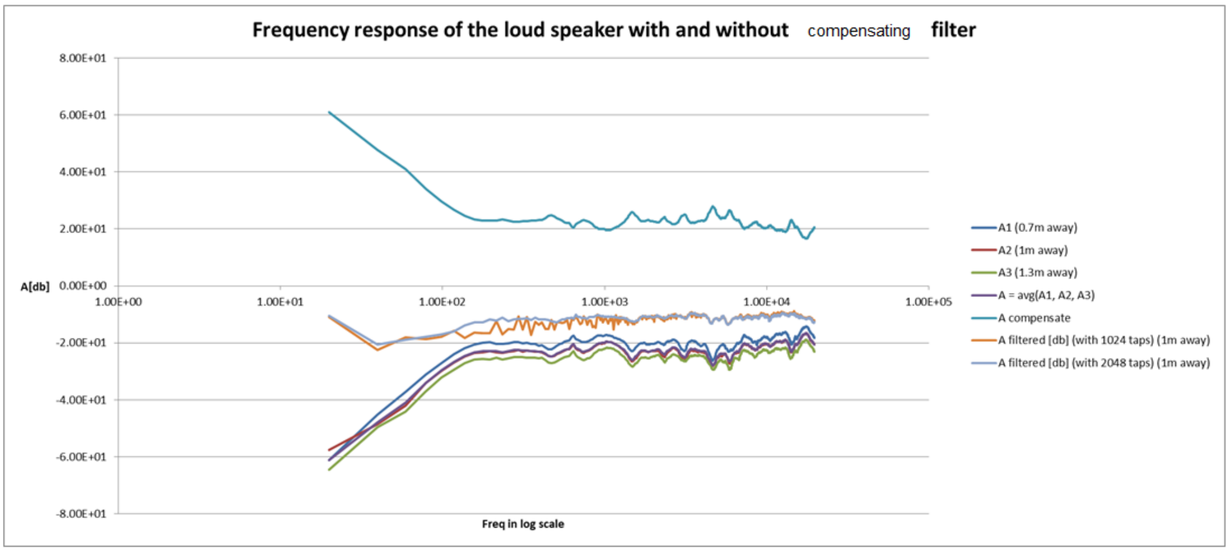

Figure 6.3.: Amplitude responses of the loud speaker with and without compensating filter

It can be observed from figure 6.3 that the amplitude response line of the filtered loud speaker is "flatter" than the original line. And the higher the number of taps is, the smoother the line is.

7. Conclusion

With a cheap price but high performance piece of hardware, FIR filter can be easily implemented and adjusted thanks to Raspberry Pi. With the help of Cirrus Logic Audio Card, audio samples can be captured, processed and delivered to the output with very high quality.

BruteFIR provides the ability to convolve unlimited sound samples with huge amount of filter's coefficients in real time using advanced techniques such as FFT block convolution and Partition convolution.

As a result, almost flawless FIR can be built and applied to solve real life problems. One example is to compensate the amplitude response of a loud speaker to make it constant. After applying this compensating filter and observing the result from the graph, the amplitude response line is significantly flatter.

Bibliography

[ALSA 2017] ALSA: *Advanced Linux Sound Architecture (ALSA) project homepage*. 2017. – URL https://www.alsa-project.org/main/index.php/Main_Page. – Zugriffsdatum: 2017-06-17. – Available in file /referenced_documents/alsa-project_org.pdf on attached DVD

[Battenberg und Avizienis 2011] BATTENBERG, Eric ; AVIZIENIS, Rimas: *IMPLEMENTING REAL-TIME PARTITIONED CONVOLUTION ALGORITHMS ON CONVENTIONAL OPERATING SYSTEMS*. 2011. – URL http://www.ericbattenberg.com/. – Available in file /referenced_documents/PartionConvolution.pdf on attached DVD

[Cirrus-Logic 2014] CIRRUS-LOGIC: *Cirrus Logic Audio Card User Manual*. 2014. – URL https://www.element14.com/community/docs/DOC-72078?ICID=CirrusLogicAudio-topMain-usermanual. – Available in file /referenced_documents/Cirrus Logic Audio Card for B+ and A+ onwards V1.02.pdf on attached DVD

[Element14-community 2017] ELEMENT14-COMMUNITY: *'Cirrus Logic Audio Card working on the Raspberry Pi 2' discussion*. 2017. – URL https://www.element14.com/community/message/147024/l/re-cirrus-logic-audio-card-working-on-the-raspberry-pi-2#147024. – Zugriffsdatum: 2017-06-17. – Available in file /referenced_documents/Cirrus Logic Audio Card working on the Raspberry Pi.pdf on attached DVD

[Ifeachor und Jervis 2001] IFEACHOR, Emmanuel ; JERVIS, Barrie: *Digital Signal Processing: A Practical Approach (2nd Edition)*. Prentice Hall, 2001. – ISBN 978-0201596199

[JACK-Audio-Connection-Kit 2017a] JACK-AUDIO-CONNECTION-KIT: *JACK Audio Connection Kit definition*. 2017. – URL http://jackaudio.org/faq/about.html. – Zugriffsdatum: 2017-06-17. – Available in file /referenced_documents/jackaudio_org.pdf on attached DVD

[JACK-Audio-Connection-Kit 2017b] JACK-AUDIO-CONNECTION-KIT: *WalkThrough Dev SimpleAudioClient*. 2017. – URL https://github.com/jackaudio/jackaudio.github.com/wiki/WalkThrough_Dev_

SimpleAudioClient. – Zugriffsdatum: 2017-06-17. – Available in file /referenced_documents/WalkThrough_Dev_SimpleAudioClient.pdf on attached DVD

[Jackson 1995] JACKSON, Leland B.: *Digital Filters and Signal Processing: With MATLAB Exercises, 3rd Edition*. Kluwer Academic, 1995. – ISBN 978-0792395591

[Mitra 2001] MITRA, Sanjit K.: *Digital Signal Processing: A Computer-Based Approach*. Mcgraw-Hill College, 2001. – ISBN 978-0072522617

[Oppenheim und Schafer 2009] OPPENHEIM, Alan V. ; SCHAFER, Ronald W.: *Discrete-Time Signal Processing, 3rd Edition*. Pearson, 2009. – ISBN 978-0131988422

[Proakis und Manolakis 2006] PROAKIS, John G. ; MANOLAKIS, Dimitris G.: *Digital Signal Processing*. Prentice Hall, 2006. – ISBN 978-0131873742

[Raspberry-Pi-Foundation 2017] RASPBERRY-PI-FOUNDATION: *RASPBERRY PI 2 MODEL B*. 2017. – URL https://www.raspberrypi.org/products/raspberry-pi-2-model-b/. – Zugriffsdatum: 2017-06-17. – Available in file /referenced_documents/Raspberry Pi 2 Model B - Raspberry Pi.pdf on attached DVD

[Schilling und Harris 2015] SCHILLING, Robert J. ; HARRIS, Sandra L.: *Digital Signal Processing using MATLAB*. Cengage Learning, 2015. – ISBN 978-1-305-63660-6

[Torger 2016] TORGER, Anders: *BruteFIR*. 2016. – URL https://www.ludd.ltu.se/~torger/brutefir.html. – Zugriffsdatum: 2017-06-17. – Available in file /referenced_documents/BruteFIR.pdf on attached DVD

A. Appendix

A.1. Cirrus Logic Audio Card

A.1.1. Connect to Raspberry Pi

Cirrus-Logic (2014)

This Cirrus Logic soundcard has been designed to plug in to Raspberry Pi simply and easily. It is compatible with Raspberry Pi with 40-pin extended GPIO, such as models A+ and B+.

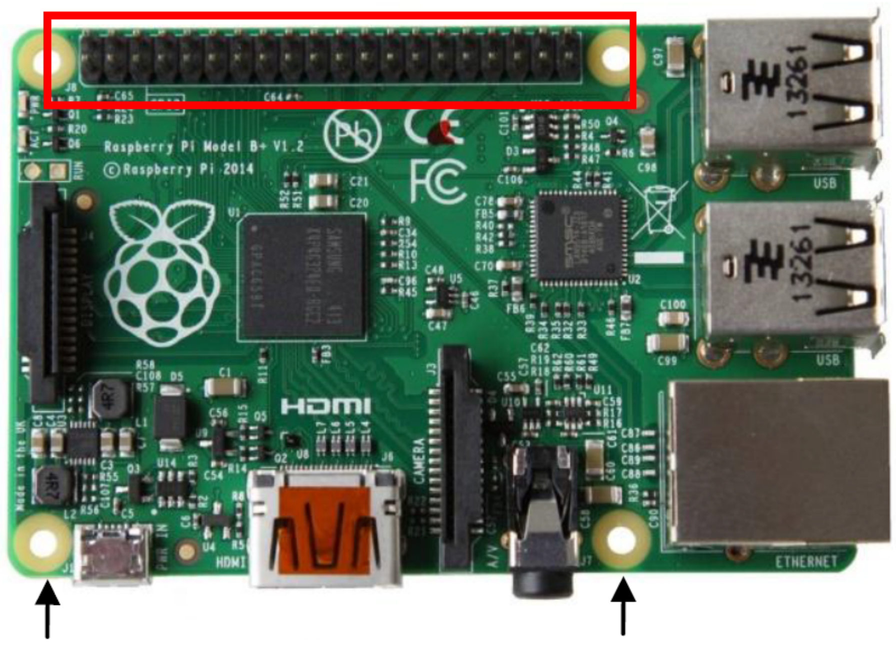

Figure A.1.: Raspberry Pi with 40-pin extended GPIO connector outlined in red

The Cirrus Logic Audio Card simply pushes onto the top of Raspberry Pi. Inside the box, are two plastic pillars and four screws that facilitate this.

- Mount the two pillars at the two mounting holes identified in Diagram A.1. Don't over tighten the screws.
- Push the Cirrus Logic Audio Card down onto Raspberry Pi's 40-pin connector, ensuring the pins are aligned.
- Use the remaining two screws to secure the board to the pillars underneath it.

A.1.2. Install software to run on Raspberry Pi

There is a fully integrated all-in-one package that contains all the drivers and settings needed to work straight out of the box:

- Download the most recent image file from www.element14.com/cirruslogic_ac
- The file .img has been compressed into a .zip format in order to minimize download time. That file needs to be restored to its original form again. After being restored, there is a single file called an Image File, or .img. This is a snapshot of what needs to be written to the microSD Card.
- Install the .img file to the microSD Card. There is a popular tool is called Win32DiskImager, and it is available at this address https://launchpad.net/win32-image-writer can be used to finish this process.
- Once this process has been completed, remove the SD Card and insert it into the SD card slot of the Raspberry Pi.
- Power up the Raspberry Pi and Cirrus Logic Audio Card. The red LED will light up on the Raspberry Pi, indicating the main chip has started up, and then the green LED will begin to flash, indicating data is being read from/written to the microSD Card. The TV/monitor (provided it's switched on of course) will begin to show the Linux boot sequence.
- This image has been designed to boot into the Graphical User Interface, so that when the boot process has been completed, the Raspbian desktop will be presented. Once this appears, it is ready to go.
- In order to maximize the space available on the microSD card, it is recommended to perform a file system expand operation. This can be achieved as follows:
  - Either start LXterminal on the desktop, or use the command line screen, and type the following command: sudo raspi-config <Enter>
  - Select the first option, Expand Filesystem, and allow the process to complete.

– At the next reboot, the file system size will have been resized to fit the microSD card.

A.1.3. IIR filter implementation on Raspberry Pi

The IIR filter can be also implemented manually on the Raspberry Pi using the second-order section IIR cascade structure:

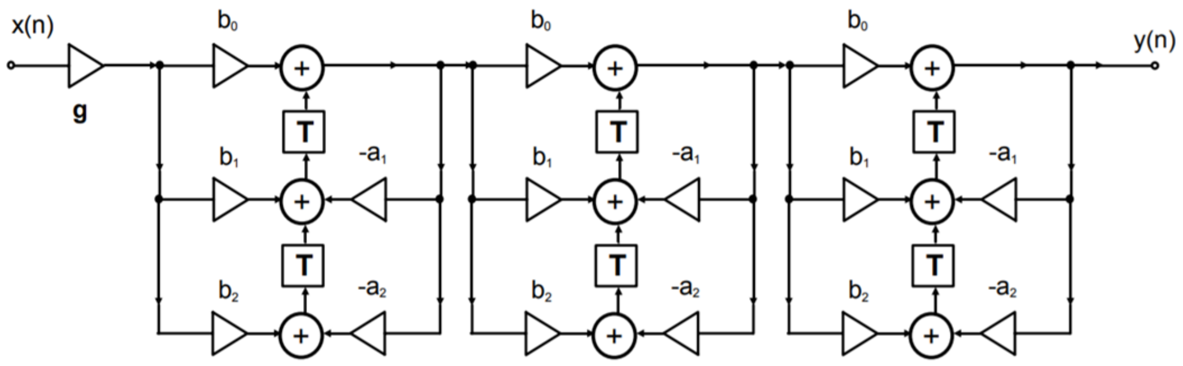

Figure A.2.: Cascade structure of a second-order IIR filter.

Code:

```
int
process (jack_nframes_t nframes, void *arg)
{
  jack_default_audio_sample_t *in, *out, *filtered_samples;

  in = jack_port_get_buffer (input_port, nframes);
  out = jack_port_get_buffer (output_port, nframes);

  filtered_samples = malloc(sizeof (jack_default_audio_sample_t) * nframes);

  for (i = 0; i < nframes; i++) {

    x = in[i];

    for (j = 0; j < SECTION; j++) {
      y = x * coeff[j][0] + w0[j];
      w0[j] = x * coeff[j][1] - y * coeff[j][3] + w1[j];
      w1[j] = x * coeff[j][2] - y * coeff[j][4];
      x = y;
    }

    filtered_samples[i] = y;
  }

  memcpy (out, filtered_samples,
  sizeof (jack_default_audio_sample_t) * nframes);

  return 0;
}
```

B. Appendix

B.1. Source code

The source code is stored in "/source_code" folder on the attached DVD. It contains:

- "/BruteFIR_configuration" folder:
  - '.brutefir_defaults': Default configuration of BruteFIR, which is edited to be able to connect to JACK as a client.
  - '.brutefir_config': User's custom configuration of BruteFIR. The sampling rate, filter's length, coefficient file, input / output ports, etc. can be declared here.
  - 'coeffs.txt': This file stores the coefficients of BruteFIR filter.
- "/Direct_convolution_implementation" folder:
  - 'headers' folder:
    - 'cirbuffer.c': Implementation of circular buffer.
    - 'taps.c': Defines the coefficients of the filters in an array.
    - 'fir.h': Defines number of frames in one period, number of filter's taps, and the buffer's size.
  - 'fir_filter_implementation.c': Implementation of direct FIR filter.
  - 'iir_filter_implemetation.c': Implementation of IIR filter.

B.2. Softwares

The necessary softwares used in this thesis are stored in "/softwares" folder on the attached DVD, it contains:

- 'cirrus_logic_audio_card_all_pi_versions.img': Image file from Cirrus Logic Audio Card, fully integrated all-in-one package that contains all the drivers and settings you need to work straight out of the box.
- linux-image-3.18.9-v7cludlmmapfll_3.18.9-v7cludlmmapfll-4_armhf.deb: Kernel with FLL patch included, to solve "alsa mmap-based access is not possible" problem.

Declaration

I declare within the meaning of section 25(4) of the Ex-amination and Study Regulations of the International De-gree Course Information Engineering that: this Bachelor report has been completed by myself inde-pendently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, September 7, 2017

City, Date

sign