0.1 To SKO

Hello Søren

We have written some more stuff in the following:

- Equalizer from "Transformation to the digital domain" and onwards. - Also related to that is appendix C about the bilinear transform. Does it make sense and is it okay to put it in an appendix? - wifi design, the parts about 5.2.2 "Bit rate and package loss test".

Questions: - At low frequencies the output signal from the filters have a lot of noise. There is a correlation between how much the signal is downsampled and how much noice there is. Could this be a result of zero-padding or are we handling the filtering wrong? See 'Programming of filters' for code examples. - We are having trouble finding good sources on up- and downsampling as well as zero padding so if you have some that would be helpful.

Gr512.

Ps. Do you like figure 5.1?

Contents

Introduction 1

1.1 Frequency Responses of audio systems

Hi-Fi audio equipment like amplifiers and loudspeakers, are very popular pieces of household electronics. People are seemingly willing to spend a lot of money on sound systems that is able to accurately reproduce sound. Amongst different types of sound-reproducing devices, the transducers[1] are generally the worst at accurately reproducing the actual sound, especially for loudspeakers[13]. Purely electronic devices, like amplifiers, are generally easier to control.

A key feature of high quality sound systems is their frequency response.The frequency response is a term describing the variations in amplitude and phase from the original signal as a function of frequency. It is generally desired to have a "flat" frequency response (i.e. the amplitude- and phase response of the output signal behaves the same way regardless of the frequency of the input signal).

This is hard to achieve in real loudspeakers because of resonances in the physical part of the loudspeaker, which will result in certain frequencies being played louder than others.

The Loudspeaker driver that converts the electric signal to sound waves only has a certain limited range of frequencies where it works optimally and loudspeakers made with only one driver (Full-range drivers) also usually have a hard time reproducing every frequency at a similar amplitude, especially when playing at high levels of power. A lot of loudspeaker systems are therefore made using multiple drivers, that each handles a certain band of frequencies.

There are different design and construction challenges for each type of driver depending on which part of the audio spectrum it should reproduce, but in general drivers for high frequencies are smaller than drivers for lower frequencies. Since drivers for reproducing the lowest part of the frequency spectrum (20 - 200 Hz) are usually quite big and use a lot of power. For this reason, Loudspeaker systems with limited size and power usage (like PC speakers) are usually physically incapable of reproducing these frequencies at the same amplitude as higher frequencies.

1.2 Acoustic properties of a room

Even with a perfectly flat frequency response in a sound system, the sound that is actually being perceived by the listener might still be different than the original audio because of the acoustics of the surrounding room.

Room acoustics is a whole subject onto itself, but while this section will not go depth about every physical phenomenon that can influence the sound quality of a room, it will try to briefly cover how the surrounding room can influence the perceived sound quality for a listener.

When a sound wave hits an object in its path, a combination of three things happen. Some of

[1] Devices that convert one type energy into another.

the wave will be reflected off of the surface of the object. The rest of the wave will travel through the object, where some of the waves energy will be absorbed by the material and transformed into heat. Whatever is left of the wave will be transmitted through the object and continue along its direction of propagation. An illustration of this can be seen on figure 1.1

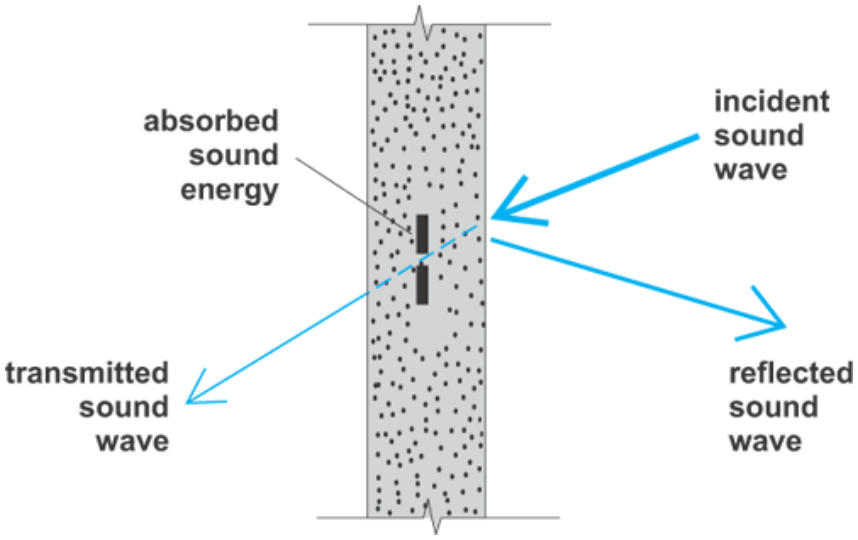

Figure 1.1: sound interacting with a wall. Illustration from [10] should be remade with vector graphics

How much of the sound wave is reflected, absorbed or transmitted through, is dependent on both the material of the object, the angle of incidence, and the frequency of the sound wave. rule of thumb for different building materials If the object being struck by the sound wave is significantly larger in area than the incoming wave, then the reflected sound wave will be reflected at an angle equal to the angle of the incoming wave. For example, a room with large and bare concrete walls will reflect specific frequencies a lot. If the waves hits a surface that is comparable in size to its wavelength, the waves will diffuse and scatter somewhat evenly across the room.

Acoustic treatment of rooms is often done by fitting rooms with object that will diffuse incoming sound ways, a covering surfaces with sound absorbent materials. This is done to reduce reflected waves in a room.

One problem associated with reflections is the emergence of interference patterns between a sound wave in a direct path to the listener the sound wave reflected off a surface. Since the reflected wave has traveled a longer distance than the direct wave, the to waveforms will likely be out of phase with each other and interference will occur. This can cause the resulting wave to either have a larger or smaller amplitude than the original wave. An example of how this can occur can be seen in figure 1.2

Since a given material absorbs and reflects sound waves differently at different frequencies, and since reflected waves are attenuated a rate given by its frequency according to Stoke's Law of sound attenuation, different frequencies will be affected by interference at the listeners position differently for different frequencies. These phenomena can have a quite significant effect since a wave with a frequency of 50 Hz can behave very differently from one with a frequency of 5000

[2] The lines representing the sound waves can be seen as tangents to the crests of the wave

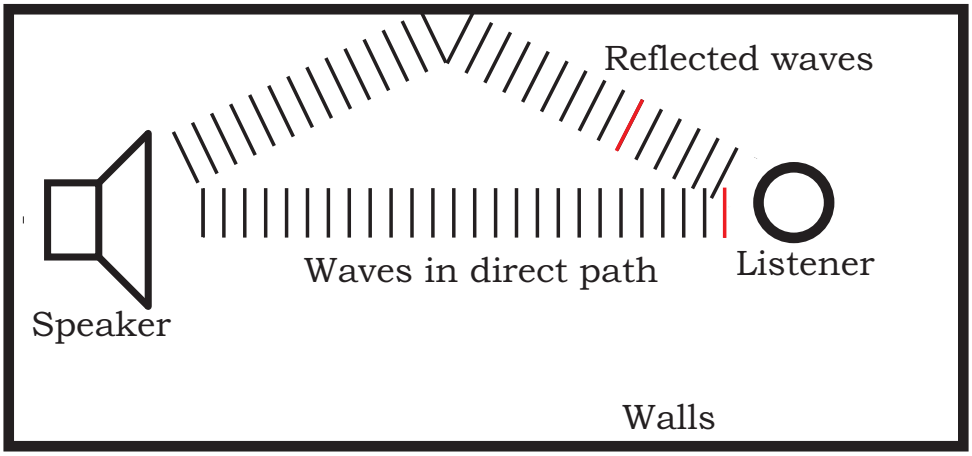

Figure 1.2: Sound traveling towards a listener along different paths[2]

Hz. And since music in general feature a pretty wide spectrum of frequencies [3], the frequency response of a given piece of music can change a lot by traveling through a room. Furthermore this behavior can vary a lot based on the position of the speaker, the size of the room, the building materials, and the interior decoration of the room, in ways that might not be obvious to the average listener, who just wants to sit at his/her favorite listening spot.

There is an important phenomenon that provides a general idea about the behaviour of a room and it is independent from the listening point, the reverberation. The reverberation is the greater or the less persistence of the sound when a sound source stops emitting, and it depends on the intensity of the sound source and the threshold of hearing.

1.3 Summation of Problem

In the previous sections it has been shown that a piece of audio can often be altered during playback by different frequencies being represented differently. Furthermore, the processes which alters the characteristics of the sound might not be immediately obvious or even fixable for the user of the sound system. Since this is a 5th semester project in Electronics and IT, the project has been is meant to revolve around an electronic system that can interact with its surroundings by measuring some form of signal, processing this data, and generating a new signal based on this processing. For this reason It has been decided to work with a system that can measure the frequency response of a sound system, and correct the frequency response based on these measurements. The following chapter will deal different theories and technologies that relate to designing such a system.

[3] 20 Hz to 20 kHz

Theory 2

This chapter will contain the analysis of the theories that make the project possible.

2.1 Equalizer

Citation needed
There are digital and analog equalizers, but in this section the focus will be on analog equalizers.

In the field of audio, an equalizer is a device that modifies the response in frequency of the signal, and adapts it to the user, solving all kinds of problems. These devices are able to either amplify or attenuate one or more frequency bands.

There are different types of equalizers: High-pass and low-pass filters, shelving filters, graphic equalizers and parametric equalizers.

A high-pass filter is an electronic circuit that passes high frequencies and attenuates low ones. The same for the low-pass filters, that allows low frequencies to pass but attenuate the high ones.[17]

On the other hand, shelving filters (figure 2.1) are used as tone controls, as they can attenuate or amplify a signal above or below a certain frequency. [17]

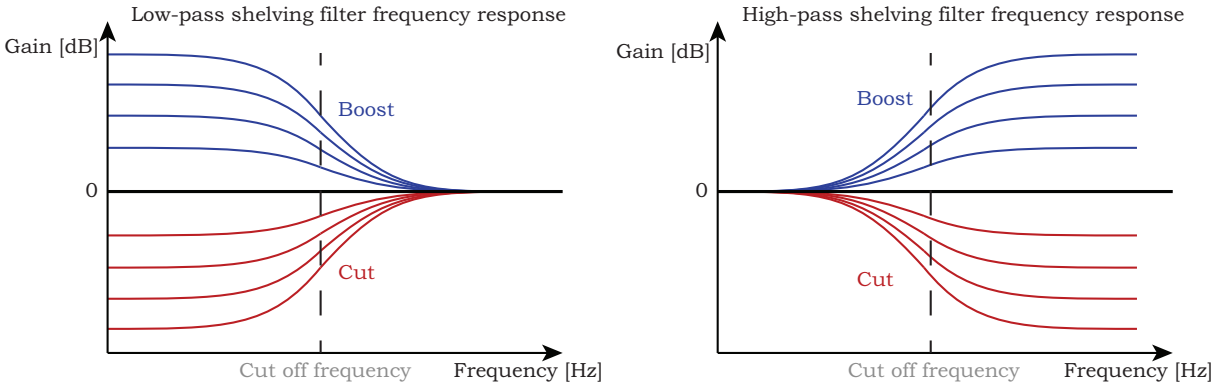

Figure 2.1: Shelving filter. Image from [14]

A graphic equalizer (figure 2.2, however, is composed by some band-pass filters. It takes the name from the physical position that the faders are placed, making possible to see in a easy way the changes made. [17]

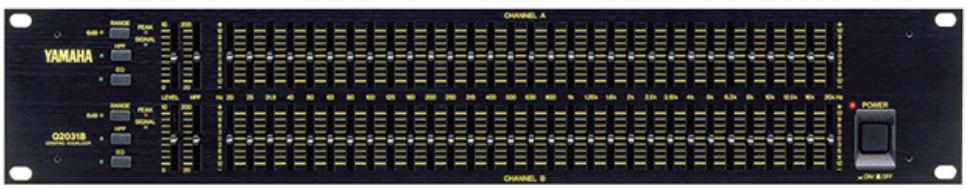

Figure 2.2: Graphic equalizer. Image from [15]

Finally, the parametric equalizer (figure 2.3), allows the individual control of three main parameters of the filters that compose the equalizer: amplitude, center frequency and bandwidth. [17]

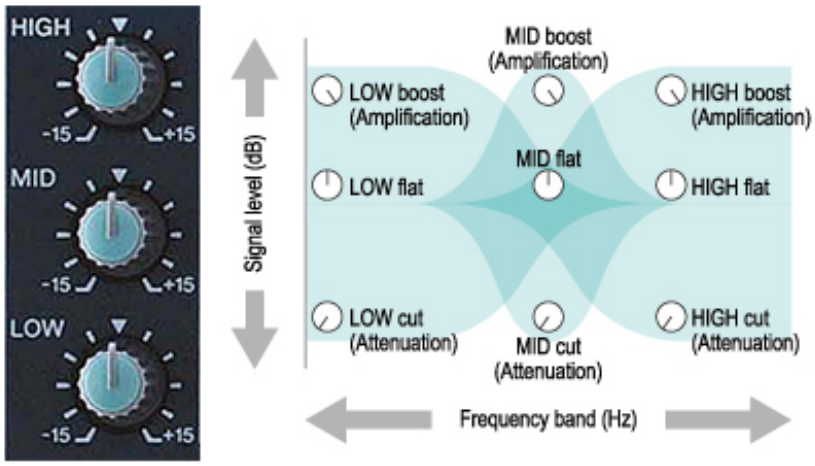

Figure 2.3: Parametric equalizer. Image from [15]

The analog filter design is mainly limited by the cost of the different elements, so a selection of better components will lead to a better equalizer.

It is well known that most of the signals contain noise, but analog filters have the disadvantage that they add noise to the signal through for example heat and electrostatic noise. [7]

2.2 Digital sound

When sound have to be transported over longer distances, or stored for later use, it might be necessary to convert it into a digital signal. This can be done by converting the sound to an analog signal trough a microphone and then converting it to a digital signal trough a analog to ADC (analog to digital converter). The digital signal can then be stored as a string of bits, which can be read by computers and for example be filtered ore shared via the internet. Furthermore it is possible to make calculations based on the digital signal such as equalization or addition of multiple signals.

Sampling frequency

To ensure that the audio signal can be recreated as an AC-signal, the sampling frequency has to be taken into account. The sampling frequency determines the number of samples, there has to be played pr second. According to the Nyquist sampling theorem[20] the sampling frequency should be above two times the highest frequency of the signal. If this is not the case, some frequencies could be represented as a different frequency, as seen in figure 2.4.

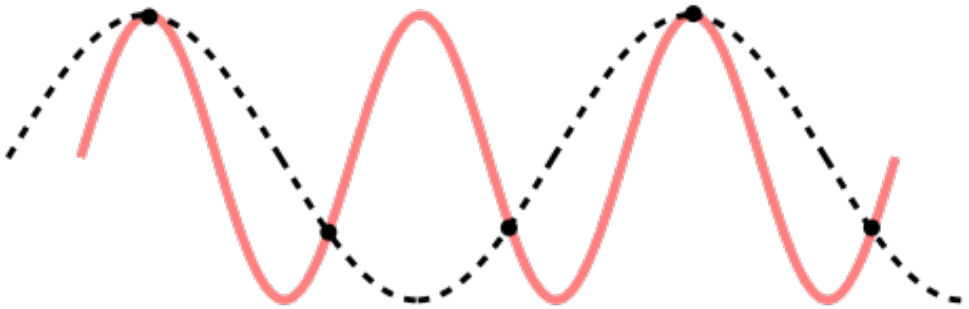

Figure 2.4: Falsely represented signal, due to the sampling frequency being to low. Figure from source [19]

Since the Nyquist sampling theorem states that the sampling frequency should be at least twice the highest frequency of the signal, the sampling frequency should be at least 40 kHz, when sampling audio signals. According to the standard IEC-60908[5] the sampling frequency of sound should be 44,1 kHz. This way audio signals can be recreated up to 22,05 kHz.

Quantization

When storing audio in a file it is defined by a string of bits that matches the amplitude of a specific sample. Here the amount of bits used to determine the sample, also known as bit depht, is a big factor in the quality of the sound. This means that the amount of bits used per sample determents how well the sound will be when converted to an analog signal. For a 4 bit signal, there would therefore be 16 different values for a sample, and for a 16 bit signal there would be 65536. An illustration of the correlation between an analog signal and its representation in bits, can be seen in figure 2.5.

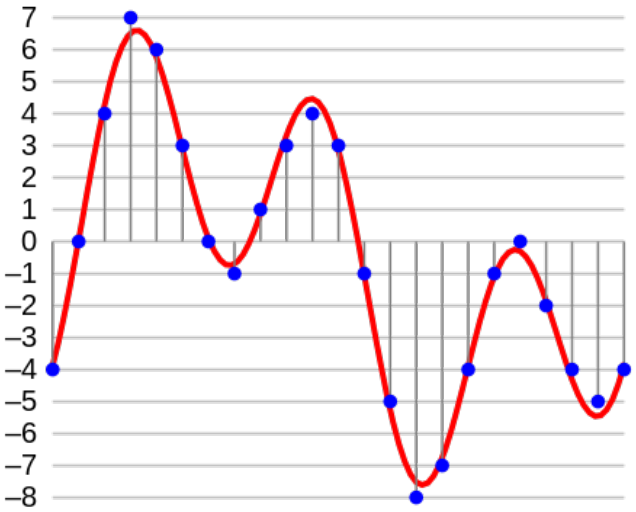

Figure 2.5: Audio signal and its representation in bits. Figure from [18]

CD quality sound is usually stored in a 16 bit resolution. In sound studios they use a bit depth of 24 or above to ensure a proper D/A conversion. If there is a need for adding two 16 bit signals together, 17 bits are needed to avoid clipping.

When audio is converted from analog to digital, there can occur an error in the sampling process. This happens because the digital value might not be exactly the same as the analog value. This

error is referred to as quantization. The quantization error can be ± 0,5 times the value of the least significant bit(LSB). The signal-to-quantization-noise ratio (SQNR) can be calculated as seen in equation 2.2.1.

$$\text{SQNR} = 20 \cdot \log_{10}(2^Q) \quad [\text{dB}] \tag{2.2.1}$$

where:

Q = Signal bit-dept [·]

Using the equation, a 1 bit signal would have an SQNR of 6,02 dB. For every bit added to the bit dept the SQNR goes up another 6,02 dB.

2.3 Digital signal processing

Digital signal processing and analog signal processing both is a description of a process to modulate an input. Digital signal processing utilizes digital filters to achieve this modulation. This makes digital signal processing much more versatile than analog signal processing because digital filters can both emulate analog filters, and because it is not limited to passive and active components, but to basic arithmetic, can therefore do a lot more[16]. Additionally there are no outside noise effecting a digital signal, which therefore does not have to be taken into account.

Digital filters

Finish this when lectures about digital filters are done.

Filters are used to selectively choose some wanted frequencies to be allowed through, and cut out other frequencies. Active filters can also be used to amplify or attenuate the specified frequencies.

A digital filter takes the input sequence of numbers from sampling the signal and computes a new sequence of numbers based on the filters transfer function. However digital filters cannot amplify a signal, only scale it up to the maximum bit value determined by the bit debt. Digital filters can only do basic operations such as: addition, subtraction, multiplication and division, more complex functions must be a combination of these [16] [12].

With digital filters the accuracy of the calculations of the functions can be controlled much more precisely than analog filters. The more complex a function and the closer the more accurate the function needs to be to the ideal, the longer it takes to compute. High computation times limits digital filters usability for real time applications. For real time applications either the filters complexity and accuracy needs to be lowered to decrease the computation time, or a fast digital signal processor is needed [12].

Due to the fact digital filters are calculated in a processor, they can be changed easily compared to analog filters, where the passive or active components would have to be changed.

new stuff, might be changed

There is generally two ways of implementing a digital filter: FIR filters (FIR) and IIR filters (IIR).

IIR Filters have an impulse response that continues forever, just like analog filters made from resistors, capacitors and inductors.

This means that an IIR filter can be designed by taking the transfer function of a known analog filter and transforming it to the Z-domain using Z-transformation and later to the digital domain using reverse Z-transform.

In this way, a digital filter can be given be a difference equation, which is a relatively easy type of equation to solve for a computer to solve given that the computer is able to store a certain number of previous inputs and outputs. In that sense a IIR filter has to have some kind of "feedback".

The other type of filters, FIR filters, on the other hand, have impulse responses that becomes zero after a certain point.

For FIR filter with an order n, the impulse response will consist of $n+1$ samples. the output will be computed as a convolution sum of the impulse response and the last $n+1$ input samples.

FIR filters has several advantages over IIR filters. The impulse response of a FIR filter can be based on an ideal impulse response. This means that high order FIR filter can closely resemble ideal filters. FIR filters cannot be unstable since the impulse response will always be zero after a certain point. It is also very easy to design a FIR filter with a linear phase response by making the impulse response symmetric.

However, these advantages comes at a cost. For a FIR filter to have as sharp a cutoff as a IIR filter, the FIR filter needs to be several orders bigger. Because of this it is more computationally expensive to implement a FIR filter than a IIR filter.

2.4 The amazing world of sound cards

In order for a computer to play and record audio, it needs a device that can convert analog sound to digital, and vise versa. A sound card also allows the user to perform some degree of digital signal manipulation, where the most common would be to manipulate the audio with an equalizer.

A sound card has a lot of interfaces, depending on the type of sound card. One of these could be a digital input or output from a CD-ROM drive or a MIDI(Musical Instrument Digital Interface). Depending on the type of sound card and the price range, the interfaces of the sound card differs, but the most common is analog input to a microphone or analog outputs to loudspeakers.

Since a sound card performs its operations digitally, all analog signal needs to be converted to digital in order to make recordings, and digital signal must be converted to analog for audio playback. This is done with an ADC(Analog to Digital Converter) and a DAC(Digital to Analog Converter). The resolution of these components depends on the price range, but a resolution of 16 bits is needed for audio in CD quality.[11][21]

For the digital audio processing there are mainly two options. Either use to computers CPU or to use a Digital Signal Processor(DSP).

In order to control and issue commands to the sound card, some form of device driver is needed. The driver handles the data connection between the sound card and an operating system. The driver needed depends mostly on the users OS. For Windows users the drivers will generally be written by the manufactures and will therefore be licensed. Linux users on the other hand,

have a bit more freedom. The most wildly used driver is called ALSA(Advanced Linux Sound Architecture). One of the bigger advantages with the ALSA-driver compared to most Windows drivers, is that it allows the user to directly interact with the kernel and therefore the hardware.[2]

2.5 Platforms for digital signal processing

In order to correct the soundlevels of the loudspeaker compared to the measured sound, a platform that can process and compare the audio is needed.

For real time digital signal processing the highly specialized digital signal processors (DSP) was created, these had a lot less functionality than normal CPU's but were faster for processing digital signals. Today however CPU's have gotten so fast that they can be used for most real time digital signal processing.

Microcontroller

A microcontroller is a circuit created for and used to control embedded systems. A microcontroller usually contains a CPU and some random access memory (RAM), besides that it has input output ports, so that it can be used in a embedded system[9].

A microcontroller will typically be set to run a single program indefinitely. This makes it possible to use the CPU[1] quite effectively. It also removes the overhead of trying to effectively schedule the CPU usage between different processes.

A very widely used series of microcontrollers are the Arduinos. The most common Arduinos are equipped with an ARM CPU with a clock speed in the range of about 8-80 MHz.

Computer

Another option for a platform for digital signal processing, could be a fully fledged computer. Besides a CPU and RAM, a complete computer will often have some form of permanent memory, network capabilities, a dedicated graphical processing unit (GPU), and multiple ports for communication with peripherals. A complete computer is also easily implemented with a operating system, which makes it easier for the CPU to handle multiple simultaneous processes with the help of a kernel.

While you normally think of a desktop PC or a laptop, when you hear the term "computer", There has recently been an emergence of small single board computers, where you have a complete computer implemented on a single PCB. The most well known example of a single board computer is the Rasberry Pi.

[1] which is often somewhat slow compared to CPU's used in PC's

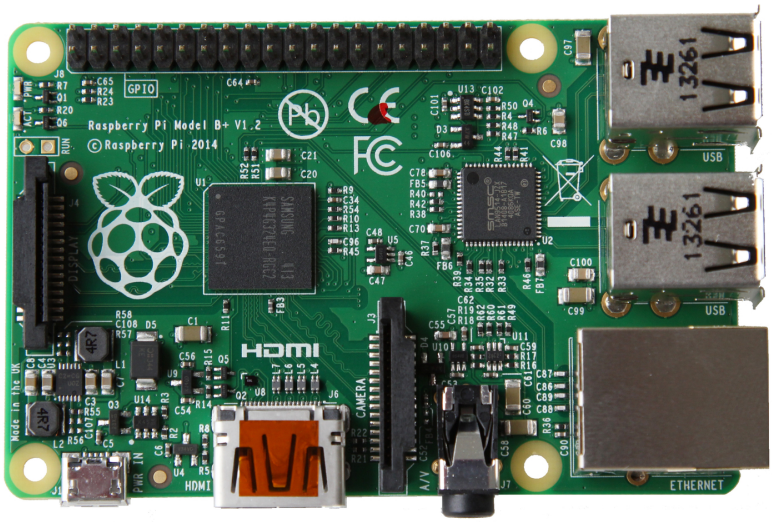

Figure 2.6: Rasberry Pi model B+. Image from [1]

There are multiple versions of the Raspberry Pi, each version with slightly different hardware. Each type has an ARM processor with a clock speed of at least 700 MHz which is several times faster than even the fastest Arduino. This fact might help with negating the extra overhead associated with running a program in an operating system instead of an embedded system. Another benefit of using a computer, is that it might be possible to use already existing drivers to communicate with a sound card.

2.6 Wireless communication

While the system may not necessarily need a wireless connection. Yet do to the curriculum and because it is expected to be of convenience, some useful forms of wireless communication will be studied to a basic level, so that it is possible to make a suitable decision for the design phase of the Self adjusting loudspeaker system.

Commonly used wireless systems

Because there is lot of different types of communication systems, some which is not relevant for the scope of this project, we will try to list some of the more commonly used systems, which is expected to work within the scope of the project.

Wireless Local Area Network

In general most of people with wireless access to the internet, are using either cellular or Wireless Local Area Network (WLAN for short). Cellular communication will be discussed later. The most common implementation of WLAN is through WIFI, which is based around the IEEE 802.11 standard. There are many varriations in 802.11 such as

do we want elaboration on frames and CSMA/CA here? or later if we "choose" to use wifi later on in the project

Wireless Personal Area Netowrk

PersonalArea Netowrk or PAN for short, and the wireless version is call WPAN (wireless PAN). The idea behind WPAN is like that of the WLAN, except the range is typical limited to *personal* space. The range is a lot shorter compared to WLAN, often around 10 meters (but can be wider for industrial purposes), where WLAN often have a range that of a house. WPAN is based on IEEE 802.15 standard. Probably the widest implementation of WPAN is the bluetooth standard (the IEEE 802.15.1 standard).

Bluetooth

Bluetooth was originally meant as a wireless alternative of the RS-232 standard. The RS-232 is no longer in use by the average person, due to the slow transmission speed, short maximal cable length and large physical connector size. It is still in use in commercial usage. So bluetooth is designed to be a short range wireless communication standard, working in areas with interference. This is done using something frequency hopping spread spectrum (FHSS) in the 2.4 to 2.5 GHz ISM band, sending small packages at up to 79 changing frequency channels. It works in a master slave relation with up to seven slaves. Later on power consumption and bit rate have been improved significantly. It is possible to achieve bit rates up to 2.1 Mb/s [4] and with the use of 802.11 AMP (Alternate MAC/PHY) bit rates of 54 Mb/s is possible. Adaptive frequency hopping spread spectrum (AFH) was also introduced, mainly to avoid interference with WIFI.

Zigbee

is this worth mentioning??

Another usage of the IEEE 802.15 standard, 802.15.4 to be exact, is the zigbee protocol. This protocol is meant for begin low on price, power consumption, bit rate and low range (though a mesh grid can be a solution to this). So this technology is not useful for this project since the bit rate is about 250 kbit/s using the 2.4 Ghz ISM band. While with the right compression of the sound data, it its possible to achieve a real time system. It would create serious limitations of the system, eg. sampling and transmitting raw sound data at the 44.1 kHz sampling rate with a bit depth of 16 bits is not possible, since this would require 705,6 kb/s. Do we want to make this delimitation here, or should we wait to the delimitation chapter?

Should we mention technologies such as 4G (and cellular in general) and radio broadcasting such as FM-radio?

Problem Statement 3

3.1 Project Scope

The preceding analysis, in chapter 2 of potential technologies, it is now possible to outline the project.

As mentioned earlier, the goal of the project is to make a system that automatically can correct the frequency response of an audio system. It has been decided that the specific solution that will be described in this project, is a system that can measure audio in a room. The measured audio will then be compared to the original audio signal, and an equalizer will then be used to adjust for an uneven frequency response, and then correct imminent audio signal.

It would be preferable to be able to configure the equalizer for many different kinds of frequency responses. Therefore it has been decided to implement an equalizer using a series of digital band pass filters, that can be automatically adjusted independently. This in analogous to the graphic equalizers described in section 2.1.

Delimitations

A big advantage for the imagined system would be if the microphone could be freely moved around the room. Therefore, it has been decided to include some form of wireless communication in the implementation.

It has been decided to use WIFI (one of the IEEE 802.11 standards). The decision is based on the two most reasonable communication standards WIFI and bluetooth, of these WIFI has been chosen. WIFI is chosen because it has the highest bandwidth making the initial design easier, because bandwidth will not be critical and optimization can be made late on if need be.

For this reason it has been chosen to implement the system on two separate devices that can communicate wirelessly: one with a microphone to measure sound system, and another device to do the signal processing.

To save time and money, and to make it easier to implement a prototype it has been decided not to make everything in the project from scratch. It would be more relevant to use already existing hardware.

It has been decided to implement both devices on Raspberry Pi single board computers (specifically a Raspberry Pi 3 for the stationary unit, and a Raspberry Pi Zero W for the microphone unit). The conversion from analog signals to digital will also be done using already existing sound cards and drivers. The same goes for microphones and wireless transmitters/receivers.

In general the hardware will be chosen mostly with availability, simplicity and ease of use in mind.

Since it might not be possible to correct the whole audio spectrum, it has been decided to put a lower limit on the frequencies that will be corrected. where and why there.

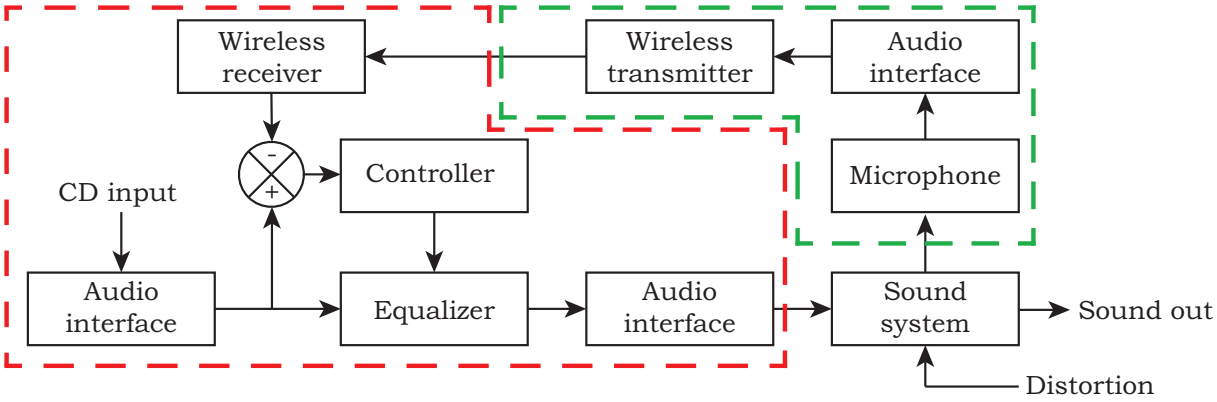

Figure 3.1: Diagram of project

table over chosen hardware

more if necessary

3.2 Problem statement

How does?

Requirements specification 4

In this chapter the specific requirements for the project will be established. This will include wanted functionality and how this will be divided into subsystems, with their own requirements.

4.1 Functional requirements

The system should be able to:

1.1 Receive an analog audio input signal.
1.2 Output a corrected analog audio signal in CD standard quality.
1.3 Correct the frequency response of the signal compared to the input within TBD dB.
1.4 Change the frequency response by TBD dB for each alteration guess: 1 dB
1.5 Modulate frequencies from TBD Hz to TBD Hz guess: 50 Hz - 20 kHz
1.6 Correct the signal within TBD ms.
1.7 Run on two separate devices that uses wireless communication.

4.2 Division into subsystems

asdf

Based around the functional requirements outlined above, the functionalities of the system have been divided into four different subcategories. These can be seen in figure 4.1.

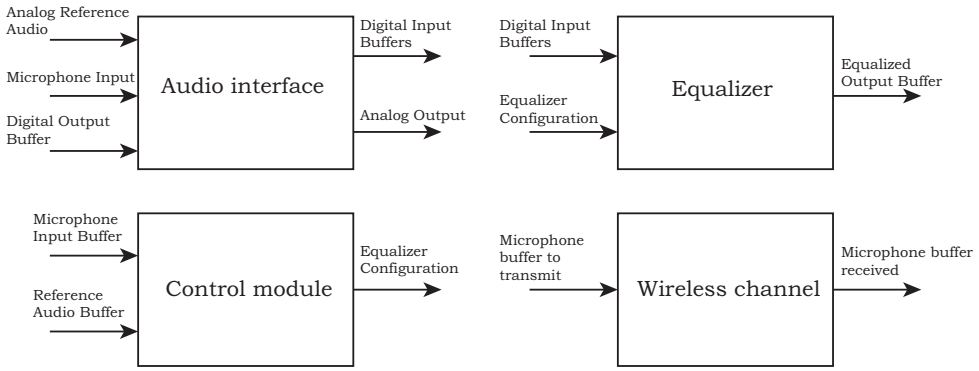

Figure 4.1: some description

Audio interface(s)

This module is tasked with converting the analog reference signal and the signal captured by the microphone to digital signals. Furthermore it has to convert the corrected digital signal to an analog output signal. It will also be tasked with storing the sampled input in correctly sized buffers for further use by the system.

Equalizer

This module is tasked with dividing the input signal into a number of frequency bands, using parallel band pass filters. Each band pass filter will need to have an adjustable gain.

Controller

The controller module is supposed to perform frequency analysis on a buffer of both the measured audio signal and the input signal. The results of these analyses are compared, and the equalizer is configured to try and minimize the differences of the frequency responses of the two signals.

Wireless channel

The last of the four modules has the job of ensuring that the two raspberry pi's can communicate over WiFi at all times. More specifically it need to make sure that the microphones device is always transmitting its measured audio signal to the stationary device. For this to be possible the wireless channel needs a bit rate of at least:

$$sample\ rate \cdot bit\ depth \cdot mono\ input = bit\ rate \tag{4.2.1}$$

the sample rate and bit rate has been decided to be 44.1kHz and 16 bit respectively ref to something. Since only one microphone is in use, and more is not functionally necessary for the system, only mono is recorded. which means the necessary bit rate for continuous streaming of the sound audio signal is:

$$44.1\text{kHz} \cdot 16\text{bits} = 705.6\text{kbps} \tag{4.2.2}$$

Which is relatively low bit rate, and should easily be achievable. So it has been decided that we should prepare for a stereo input instead, which means a bit rate of 1.411Mbps is needed. Furthermore some headroom would also desirable, so headders, flags and tails can be implemented if need be. An immediate bit rate with the added headroom would be 2Mbps.

qwer

2.1 Audio interface
2.2 Equalizer
2.3 Controller
2.4 Wireless channel

The requirements for the modules is an early draft

4.3 Requirements for audio interface

The audio interface module should be able to:

3.1 Convert an analog audio signal to a digital signal.
3.2 Convert an digital audio signal to a analog signal.
3.3 Sample audio signals with a sampling frequency of 44,1 kHz and a bit debt of TBD bits.
3.4 Store input and measured data in two buffers of TBD size.
3.5 Convert the signal within TBD ms. Less then the total delay

4.4 Requirements for equalizer

The equalizer module should be able to:

4.1 Capable of dividing a digital signal into 10 frequency bands.
4.2 Adjust the amplitude of the individual frequency bands with - TBD dB.
4.3 Make the frequency adjustments based on input from the controller module.
4.4 ****Adjust the signal within TBD ms. Less then the total delay****
4.5 have a maximum group delay lower than TBD ms.
4.6 have an amplitude response of ± 1 dB when all bands are at maximum gain.

4.5 Requirements for controller

The control module should be able to:

5.1 Analyze frequencies from TBD Hz to TBD Hz guess: 50 Hz - 20 kHz
5.2 Change the frequency response by TBD dB for each alteration guess: 1 dB
5.3 Correct the frequency response of the signal compared to the input within TBD dB.
5.4 Correct the signal within TBD ms. Less then the total delay

4.6 Requirements for wireless channel

The wireless channel module should be able to:

6.1 transmitting and receiving data between the two raspberry pi's at a constant rate of 2Mbps.
6.2 Communicate using the IEEE 802.11 standard.
6.3 Send data with TBD% package loss or below.

Design 5

This is a nice header.

5.1 Design of audio interface

In this section, the design and implementation of the audio interface will be covered.

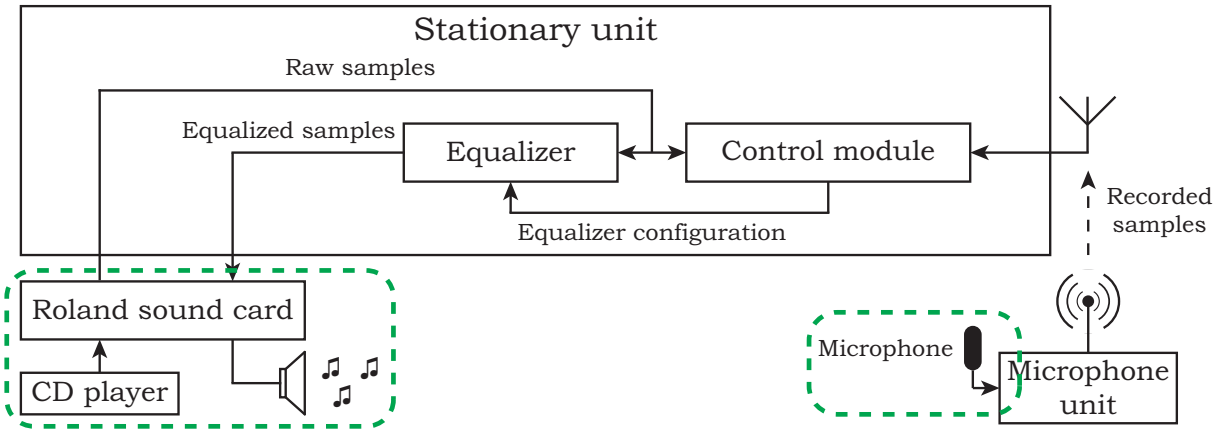

Figure 5.1: You are here

The purpose of the audio interface is to convert a reference analog signal, and the signal captured by the microphone, to digital signals. These digital signal are to be saved, so the controller can analyze and compare them. Finally it must be able to convert a digital signal to an analog, after the equalizer has made its potential adjustments.

The module can be split into two separate parts. A part that records the reference signal and plays the corrected signal, and the part that records the signal being played, and makes it ready to be transmitted by the wireless channel.

The recording of the reference signal and playback of the corrected signal is done by the UA-25EX sound card by Roland[8], which contains a 24 bit DSP with a sampling rate up to 96 kHz and two channel input and output. The sound card can be seen in figure 5.2.

Figure 5.2: Roland UA-25EX

The recording of the signal being played is done by a AK5371 USB microphone [3], which has 2 channels, a 16 bit A/D converter and a sampling rate of 44,1 kHz.

In order to communicate between either the Roland sound card, or the USB-microphone and their respective Raspberry PI, the ALSA-API, as described in section 2.4, is used. To ease the understanding of the designing of the audio interface, some basic ALSA terminology will be explained. Instead of working in samples, a term called frames is used. A frame consists of a sample from each channel, and since two channels is being used with two bytes in each sample, the size of one frame is in this case 4 bytes.

When streaming audio, or any kind of data for that matter, a constant transmission of frames is needed. This is not feasible since the CPU most likely has other jobs as well, and if the sample is not received at the exact right moment, the audio will click. This can be resolved by using a buffer, so data can be transfered to the sound card in batches, which will result in latency. Using this buffer as a ring buffer so new frames just needs to be put in after the previous frames. The size of this ring buffer will determine the maximum amount of latency.

Another problem occurs when applying the ring buffer. Assuming the ring buffer is completely filled with frames when it is completely empty it is completely emptied, maybe?, it is not guaranteed that the new data will arrive in time for the new first frame to be played. Therefore ALSA uses a term named periods. A period is either a specific amount of frames or time between status updates. If the period size for example is 512 frames, an interrupt will be made every time 512 frames has been played, meaning it is time to fill the buffer with new data. This also means that the ring buffer should at least have the double size of the period size, since it should be possible to fill the ring buffer with new data right after the interrupt has been made.

This leads to the design the audio interface, where the amount of latency in the system has to be considered. For the recording using the USB-microphone, the latency does not matter, since the data is being used, to adjust the equalizer. On the other hand, the Roland sound card must be able to both record and play in "real time". In this case, real time is when the recorded data is being played fast enough for the user not to notice a time difference between pressing play and samples being played by the sound card. A maximum latency of 100 ms is considered to be enough. I know it's a guess but can we find any source on how quickly a person would notice, or would it even matter if everything is delayed this amount of time?

The latency is the time difference between a frame is put into the buffer and the same frame is

played. Since the amount of frames being played every second is determined by the sampling rate, the maximum latency can be found as:

$$latency = \frac{ringBuffer_{size}}{2 \cdot samplingRate} \qquad [s] \tag{5.1.1}$$

In order to play in stereo the sampling rate must be doubled, which is the case with the ring buffer where the samples are interleaved. It has been decided to use a ring buffer with a size of 2048 frames, which is 4096 samples, and since the sampling rate is 44,1 kHz, the maximum latency is:

$$latency = \frac{2048 \text{ frames}}{2 \cdot 44100 \ \frac{\text{samples}}{\text{s}}} = 23{,}2\,\text{ms} \tag{5.1.2}$$

As seen, the latency is well bellow the requirement, and with the chosen ring buffer size of 2048 frames, and period size of 512 frames, it is now possible to design the actual audio interface.

5.1.1 Recording of reference audio and playback of corrected audio

As mentioned, the recording of the reference audio and playback of the corrected audio is done by the UA-25EX sound card, which is controlled by the placeholder unit. First the sound card must be initialized for both recording and playback. Starting with initialization of the recording, hardware parameters of the sound card must be set it is usable. This is done by first loading the default parameters and afterwards edit the parameters of interest. These parameters are in this case the streaming mode, the sampling rate, the format, the number of channels, and how to read the sampled data. Since the reference audio needs to be recorded, the streaming mode is set to "capture". Because the specs dictates cd quality, the sampling rate is set to 44100 Hz, and the bit-depth is set to 16 bits per sample. In this case the format is little endian since the placeholder unit has an ARM-processor, which uses little endian. It has been decided to record in stereo because most music is in stereo. It would be a shame to simply discard one of the channels, and therefore the number of channels is set to two.

The initialization of the sound card in playback mode is done in almost the same way as in recording mode. The only difference that the streaming mode is set to "playback" instead of "capture". When the sound card has been initialized to both record and playback, the actual reading and writing of data can take place. In figure 5.3, the general flow of the record/playback can be seen. Here the buffer is accessible by other parts of the system, so adjustments of the frames are possible.

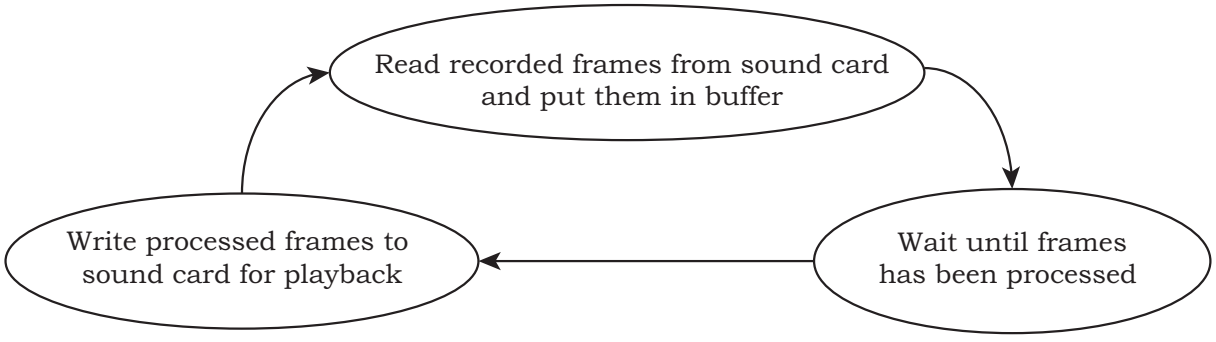

Figure 5.3: Flow of recording reference audio and playback of corrected audio.

Before filling the buffer with recorded frames, it necessary to check how many frames are available for capturing, so the number of recorded frames won't exceed the size of the buffer. Afterwards the number of available frames are read and stored. This can be seen in code example 5.1.

Code example 5.1: Reading of available frames

```
1 int avail = snd_pcm_avail_update(recordHandle); //Updates the amount of available frames.
2
3 if (avail > 0) {
4         if (avail > frames){  //If there are too many frames, only read the maximum amount possible.
5                 avail = frames;
6         }
7         snd_pcm_readi(recordHandle,buffer,avail); //Reads interleaved frames and stores them in the buffer.
8 }
```

Stuff about how to wait.

When the frames has been processed by the control unit, they are written to the sound card for playback. This is done almost the same way as when reading frames. The amount of available space in the sound card buffer is checked before writing any frames. If the free space exceeds the amount of frames, all the frames are written, else an overrun will occur. With a ring buffer size four times as large as the period size, this should not be possible. Writing frames to the sound card can be seen in code example 5.2. space check can maybe omitted, but need test to be sure.

Code example 5.2: Reading of available frames

```
1 int avail = snd_pcm_avail_update(playbackHandle); //Updates the amount of available frames.
2
3 if (avail > 0){
4         if (avail > frames){
5                 avail = frames;
6         }
7         snd_pcm_writei(playbackHandle,buffer,avail); //Writes interleaved frames to the sound card.
8 }
```

Make nice tail

5.1.2 Recording of played signal

The recording of the played signal is done by the USB-microphone which is controlled by the microphone unit. The task is to record a number of frames for transmitting by the wireless channel.

On figure 5.4 can the overall functionality be seen.

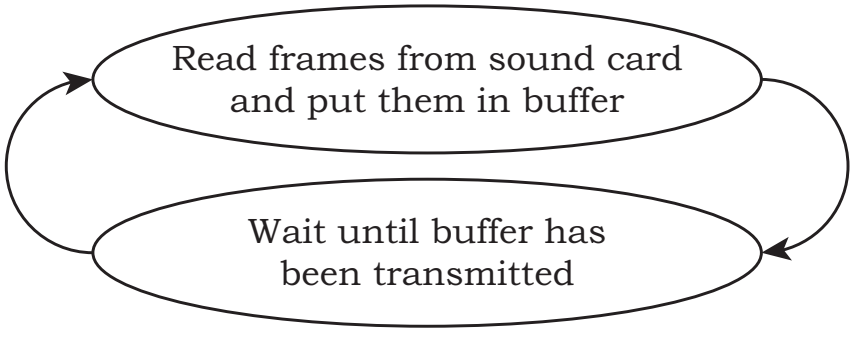

Figure 5.4: Flow of recording the played signal.

Since ALSA is being used for the USB-microphone as well, both the initialization and use is almost the same as for the placeholder unit. The only difference is that the USB-microphone is initialized to use one channel, and the amount of samples is therefore halved.

I know it's a short sections, but there is not really anything new to write.

Make an even nicer tail for the audio interface.

5.2 Wireless interface design

In this section the design of the wireless connection between the control unit and the microphone unit will be described. The wireless connection must be designed to live up to the specifications from section 4.6. From the specifications the following requirements needs to be fulfilled: The wireless connection needs a data rate of at least 2 Mbps, it should be using a WiFi protocol and should have a maximum package loss of TBD package loss.

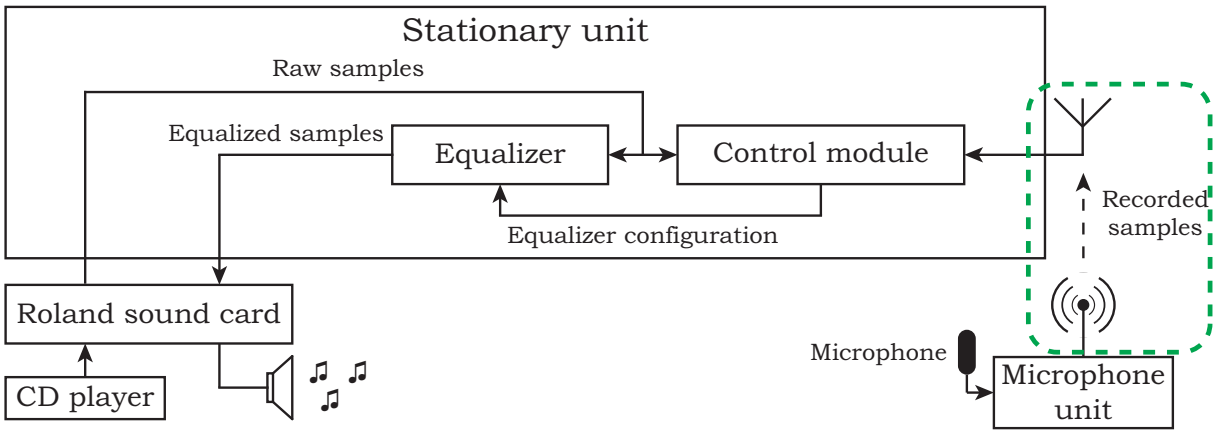

Figure 5.5: You are here

5.2.1 connection structure and protocol suite?

There are two obvious possibilities for setting up a connection between the two Raspberry Pi's: Either a direct connection, where one of them acts as a router, or a indirect connection through an already existing router. Since the two Raspberry Pi's doesn't need to be connected to anything but each other, and since a preexisting router would not be dedicated to this connection, which means some of the routers bandwidth capacity would be used for other purposes. A direct connection is more reliable and therefore preferable

To setup a direct connection between the two Raspberry Pi's, the stationary unit is set to Ad hoc mode. Ad hoc is latin for: "For this specific purpose" and is used in wireless networks to describe a connection that does not use any preexisting infrastructure. The stationary unit is used as host a SSID is set. To make it possible for the microphone unit to connect, there is need for a administrator of IP addresses. for this to be done automatically an DHCP (Dynamic Host Configuration Protocol) is installed. The final step for establishing a connection is that the microphone unit also needs to be set to Ad hoc mode, and connect to the stationary unit using its SSID.

With connection established between the two Raspberry Pi's data can be send, for this purpose a transport protocol suite is needed. for this many options exists, the two most prominent are

UDP (User Datagram Protocol) and TCP (Transmission Control Protocol).

the protocol suite TCP is the most widely used protocol suite on the Internet. Because compared to UDP it offers a lot of options for the users of the connection that UDP do not. Options like congestion control, connection control, retransmission capabilities and transmission windows.

When a connection is established a "three-way-handshake" is used, see figure 5.6, this is done to insure the connection is possible, and to determine parameters for the data transmission. These parameters determines things such as the size of the segments which is transfered and the window scale factor, used for determining the amount of bytes transferred before an acknowledgement is needed for the receiver. These options and parameters makes the TCP a connection-oriented protocol. The TCP header is at least 20 Bytes in size, but can be bigger depending on some if options are in .

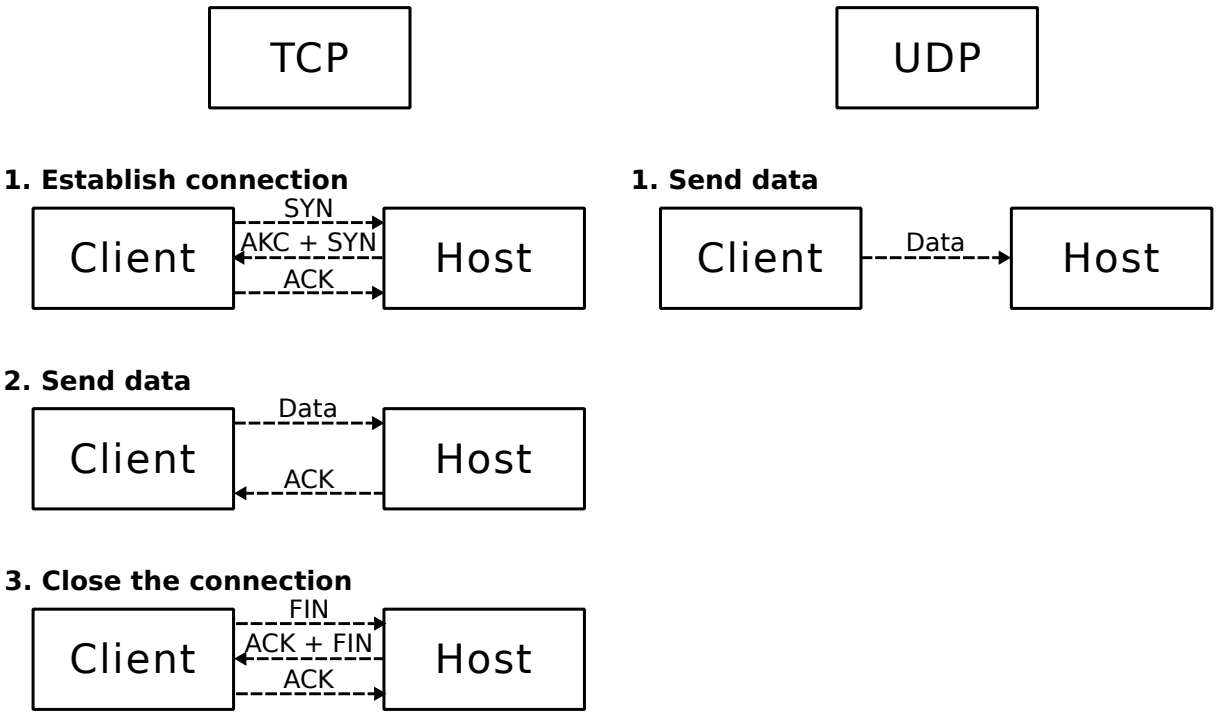

Figure 5.6: TCP hand need cite aau Jens Slides

The protocol suite UDP is connectionless which compared to TCP means that the connection is not set up prior to the data transfer that also means the every time a transmission is done a new connection is established. The UDP header only contains a port description, a check sum and a length. This means that the UDP header is very small compared to TCP header, the UDP header is a minimum of 8 Bytes.

The complete system is a real time system, and is established over a ad hoc connection. This means that there are no use of the TCP protocols features such as congestion control and retransmission, other than this the TCP-header is much larger than that of UDP-header. Making the UDP protocol the choice for the complete system. This reduces overhead and complexity of the system.

5.2.2 connection using socket programming

To establish a UDP connection between a host and a client, the host must establish a socket as seen in Code example 5.3.

Code example 5.3: UDP socket creation

```
#include <netinet/in.h>
if((servSock = socket(PF_INET,SOCK_DGRAM, IPPROTO_UDP)) < 0)
        error("socket creation\n");
```

The socket is created, using the socket function, this function takes three inputs: family, type and protocol. The input parameter `PF\_INET` specifies the socket to use the `IPv4-protocol`, `SOCK\_DGRAM` and `IPPROTO\_UDP` specifies the socket to be UDP protocol suite. is sock_dgram and ipproto_udp two sides of the same coin??

The host now has to bind the socket to the address family, an IP address and a port, seen in 5.4.

Code example 5.4: UDP bind

```
struct sockaddr_in server;
server.sin_family            = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port              = htons(atoi(argv[1]));

if(bind(servSock, (struct sockaddr *) &server,sizeof(server)) < 0)
        error("Port is in use\n");
```

Before the socket can be bound the input parameters must be specified, this is done in a struct of the type `sockaddr\_in` which is specified in the `netinet/in.h` library. As can be seen in the code example 5.4 the family is and so is the port, the port is determent as input to the c program -wat??. The input address is set to `INADDR\_ANY` which means it is specified to all available address on the network.

The bind function takes three inputs: the socket id of the socket specified previously, a pointer to the struct `sockaddr\_in server`, and the size of this struct.

The host now enters an infinite loop where the host waits to receive data from the client, in this case the microphone unit.

Code example 5.5: UDP receive from client

```
#define x 1024
int y = x*2;
int16_t Data[x];

struct sockaddr_in client;
clientSize = sizeof(struct sockaddr_in);
while(1)
{
        if((recvMsgSize = recvfrom(servSock,Data,y,0,(struct sockaddr *) &client,  &clientSize)) < 0)
                error("Error in receiving data\n");
...
```

The size of the array where the data is stored after being received is determined by the size of the audio values send from the microphone unit - see section 5.1.

The `recvfrom` function takes six inputs as can be seen in code example 5.5: First input specifies the socket id. The second specify what kind of data and where it should be saved. The third determines how many bytes should be received, since it is int16_t, every value is 2 bytes in size thus y is (2048) twice the size of x, the reason for the y value is that each frame consists of two channels each defined to be 512 values, making the total 2048 bytes to be send. Making y the about of bytes to be transmitted and x the number of values. The fourth determines flags, but is not used in the UDP protocol?. The fifth determines the source address and saves it into the client struct similar to the server struct, see code example 5.4. The sixth inputs the size of the client struct.

Then For the client to connect to the host socket, it first has to create a client socket and bind to the right port. This is done in the same way as for the server see code example 5.3, and 5.4, the port has to be the same as that for the server.

For the client to be able to send data it has to setup a server struct with the IP and port of the host socket.

Code example 5.6: UDP bind

```
#define x SOME KIND OF VALUE
int y = x*2;
int16_t Data[x];

struct sockaddr_in server;
server.sin_family            = AF_INET;
server.sin_addr.s_addr = inet_addr(argv[2]);
server.sin_port              = htons(atoi(argv[1]));

while(1)
{
        if(recvMsgSize = sendto(clientSock,Data,y,0, (struct sock_addr*) &server,sizeof(server)) < 0)
                error("Error in sending data");
}
```

The address and the port, is inputs for the microphone unit's program, this could however be fixed to 192.168.1.1 since the microphone unit always has to connect to the stationary unit, which is hosting the ad hoc server.

As can be seen in the code example 5.6 the `sendto` function takes similar inputs as the `recvfrom` function, and the data transmit and received must both be the same size and type to ensure no mistakes happen. the difference is that the `sendto` function transmit the data to the already known IP and port of the host.

To make sure that the host computer does not read the data from the client incorrectly, due to different usage of endian. All data is converted from the host's byte order to network's byte order, and back again when received by the host.

To be certain that the data arrives at the control unit correctly and not corrupted, an checksum is implemented. To

Bit rate and package loss test

What remains to be determent is, if the transmission speed between the stationary unit and microphone unit, is at least 2Mbps and if the package loss between the two is an issue or not.

Approximate bite rate

In A a test is made to give an approximate of the bit rate between the units. A large file consisting of 400MB of integers where made and transmitted from the microphone unit to the stationary unit. Time is measured of how long the transmission takes to complete, and an approximate of the bit rate is computed. An approximate bit rate of around 4 to 5.5 MB/s was achieved, much higher than the required 2Mbps (0.25MB/s).

Measurement of package loss

B A measurement were conducted with intent to gain knowledge, as to which extent package loss should be expected. No package loss were recorded, which is expected to be because, it is only the two units that is users of the ad hoc network. Therefore it has been decided that only minimal implementation, to correct for package loss is needed. Namely the usage of checksum. The programmed checksum for the package loss measurement, will be adjusted and used for the total system. Which means that the stationary unit will know, if there have been a package loss, and then the stationary unit will be able to decide how to handle the package loss, which will be described in section X!?!?

5.3 Design of Equalizer

In this section the design of the equalizer module will be described. The equalizer must be designed to live up to the specifications from section 4.4.

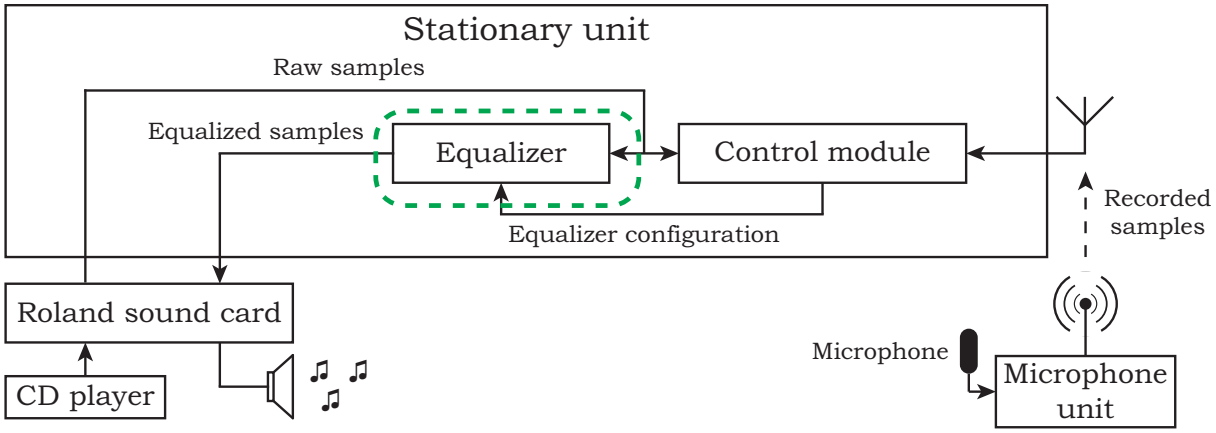

Figure 5.7: You are here

From the requirements specification, it is given that the equalizer should be divided into 10 frequency bands. These 10 frequency bands can be implemented using 10 bandpass filters.

5.3.1 Determination of center frequencies

For all the bands to be equally distinct from each other, the center frequencies of the bandpass filters must be distanced logarithmically from each other something something how sound works.

The center frequencies are calculated by starting from 1 kHz and then multiplying or dividing by 2, so that each center frequency is placed an octave apart. This nicely covers the audible

frequency range by having the lowest frequency band around $\omega_{c1} = \frac{1000\text{Hz}}{2^5} = \frac{1000\text{Hz}}{32} = 31,25\text{Hz}$ and the highest frequency range around $\omega_{c10} = 2^4 \cdot 1000\text{Hz} = 16 \cdot 1000\text{Hz} = 16\text{kHz}$

All the center frequencies can thus be written as $2^n \cdot 1000\text{Hz}$ where $n = -5, -4, -3, ...3, 4$.

ω_{C1}	ω_{C2}	ω_{C3}	ω_{C4}	ω_{C5}
$2\pi \cdot 31,25$ Hz	$2\pi \cdot 62,5$ Hz	$2\pi \cdot 125$ Hz	$2\pi \cdot 250$ Hz	$2\pi \cdot 500$ Hz
ω_{C6}	ω_{C7}	ω_{C8}	ω_{C9}	ω_{C10}
$2\pi \cdot 1$ kHz	$2\pi \cdot 2$ kHz	$2\pi \cdot 4$ kHz	$2\pi \cdot 8$ kHz	$2\pi \cdot 16$ kHz

Table 5.1: Center frequencies spaced one octave apart

These 10 center frequencies are similar to the center frequencies specified in the ISO 266 standard for octave band equalizers [6].

5.3.2 Determination of filter transfer functions

The passband of the bandpass filters should cover the whole frequency range from TBD - TBD specified in section 4.4, so that the equalizer will have a flat passband in the neutral configuration[1].

For each bandpass filter to handle an equally significant amount of frequencies, the cutoff frequencies of two filters next to each other should meet at the logarithmic mid point of their center. Thus the edge frequencies of one of the bandpass filters can be formulated as:

$$\omega_{Hn} = \sqrt{\omega_{Cn} \cdot \omega_{Cn+1}} \tag{5.3.1}$$

and

$$\omega_{Ln} = \sqrt{\omega_{Cn} \cdot \omega_{Cn-1}} \tag{5.3.2}$$

Where ω_{hn} is the higher edge frequency and ω_{ln} is the lower edge frequency of bandpass filter number n.

Since the center frequencies are spaced exactly one octave apart, the ratio between the edge frequencies can be calculated as:

$$\omega_{Hn} = \sqrt{2 \cdot \omega_{Cn}^2} \iff \frac{\omega_{Hn}}{\omega_{Cn}} = \sqrt{2} \tag{5.3.3}$$

and

$$\omega_{Hn} = \sqrt{\frac{1}{2} \cdot \omega_{Cn}^2} \iff \frac{\omega_{Ln}}{\omega_{Cn}} = \frac{1}{\sqrt{2}} \tag{5.3.4}$$

Considering this it can also be shown that the passband of any single filter covers exactly an octave.

$$\frac{\frac{\omega_{Ln}}{\omega_{Cn}}}{\frac{\omega_{Ln}}{\omega_{Cn}}} = \frac{\omega_{Hn}}{\omega_{Ln}} = \frac{\sqrt{2}}{\frac{1}{\sqrt{2}}} = 2 \iff \omega_{Hn} = 2 \cdot \omega_{Ln} \tag{5.3.5}$$

In an ideal world the frequency response of the equalizer should then look like figure 5.8

[1] Meaning that all the bandpass filters have the same gain.

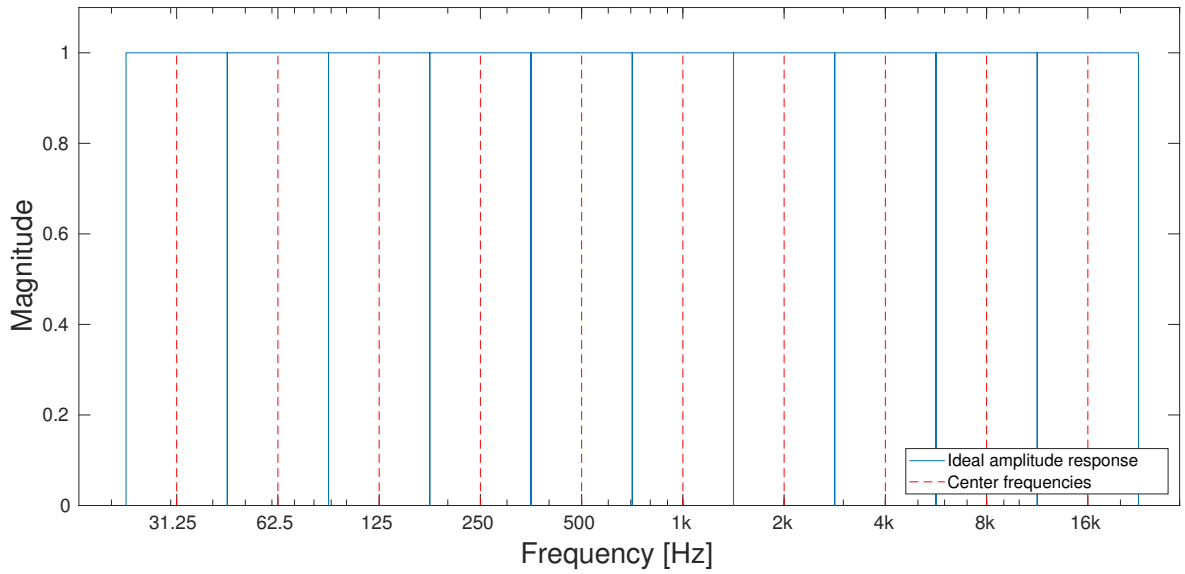

Figure 5.8: Ideal amplitude response of the equalizer.

Since the equalizer is supposed to be implemented on a Raspberry Pi, the bandpass filter will have to be digital filters. There are generally two ways of designing a digital filter as can be seen in section 2.3: FIR and IIR. The signal processing of the equalizer will have to be made in approximately real time on the processor of the stationary raspberry pi. Since IIR filters needs fewer computations, it makes the most sense to implement the filters as IIR filters. Another reason to use IIR filters is that the filters should be able to handle very low frequencies compared to the sampling frequency of 44.1 kHz.

To get the desired frequency response given by the requirements in section 4.4, continuous time filters, that meets these requirements, will be designed initially. They will then afterwards be converted to the digital domain.

calculations of filter order showing that the necessary order is 4. Been written plz read. Been read; is k.

Since it is desired to have as flat a pass band as possible, it is chosen that the filters will be implemented as Butterworth filters. The order of the filters can be calculated using the following equation:

$$n \geq \frac{1}{2 \cdot \log(\frac{f_s}{f_p})} \cdot \log\left(\frac{10^{\alpha_s/10} - 1}{10^{\alpha_p/10} - 1}\right) \tag{5.3.6}$$

Where:

n	= filter order	$[\cdot]$
f_p	= pass frequency	[Hz]
f_s	= stop frequency	[Hz]
α_p	= pass attenuation	[dB]
α_s	= stop attenuation	[dB]

To calculate this the bandpass filter around 1 kHz will be used as a reference point. Using equation 5.3.3 the pass frequency can be calculated as $\sqrt{2}$ times the center frequency. In this case the pass frequency will be 1414 Hz. Because this is the edge of the filter the attenuation

at the cutoff frequency would be 3 dB. Since a Butterworth filter is being used, it will require a large filter order to have no influence on the surrounding filters. Therefore is has been decided that the attenuation at the next filters cutoff frequency should be at least 20 dB. As before the cutoff for the next filter will be $\sqrt{2}$ times the center frequency. This would be at 2828 Hz. With this the necessary order of the filter can be calculated.

$$n \geq \frac{1}{2 \cdot \log(\frac{2828\text{Hz}}{1414\text{Hz}})} \cdot \log\left(\frac{10^{20\text{dB}/10} - 1}{10^{3\text{dB}/10} - 1}\right) = 3.32 \tag{5.3.7}$$

Since it is not possible to have a decimal number as the filter order, the value for n will be rounded up and n = 4. Given that a bandpass filter is needed the order would be twice as high, as this would be a combination of a 4th order high and lowpass filter. As starting point, bandpass filters with the same center frequencies and bandwidth as earlier, will be implemented at 8th order bandpass filters.

To see if these filters can fulfill the requirements, all 10 bandpass filters along with the sum of all of them are plotted in a bode plot in MATLAB.

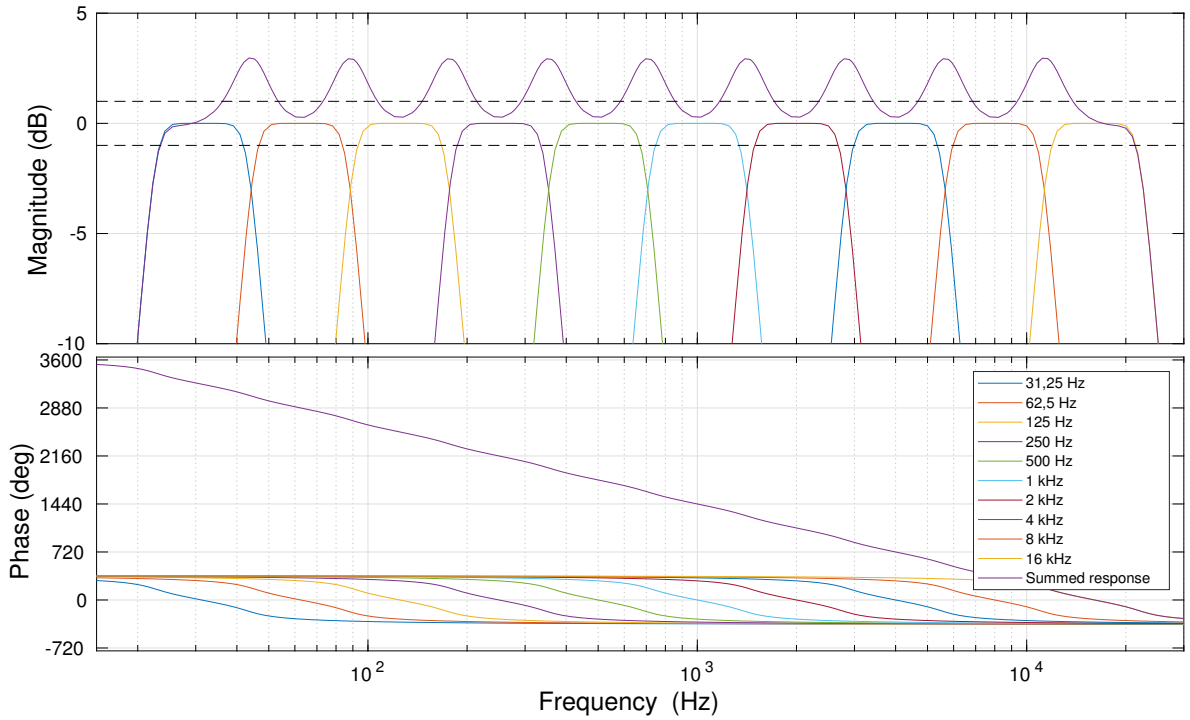

Figure 5.9: Bode plot of bandpass filters as 8th order Butterworth

As can be seen in figure 5.9, these bandpass filters cannot fulfill the requirement of a bandpass ripple at ± 1 dB. It can be noticed that the gain where the bands meet ($\omega_{Hn} = \omega_{Ln+1}$), are significantly higher than 1 dB. This is not totally unexpected since two gains of -3 dB are added together at the point there the bands meet. This should give a gain at these points of about $\frac{2}{\sqrt{2}} = \sqrt{2^2}$ which is $20 \cdot \log\sqrt{2} = 3\text{dB}$. Since the filters will be Butterworth filters, the edge frequencies will always have a gain of -3 dB. Therefore it is impossible to have a gain of less than 1 dB if the bands meet at the edge frequencies.

[2] This is assuming that the other bands have an insignificant effect at this frequency.

But as a consolation the phase looks similar to a straight line, which corresponds to constant group delay, which is also desired. This makes sense as the bands are an octave apart. Since all the bands are the same just shifted an octave along, the phase responses will behave the same way. The results is that when the phase response of one the filters starts to flatten out, one of the other filters phase will start to decrease. This makes the sum of all the phases look sort of like a straight line along the passband.

To lower the additional gain, the bandwidth of each band will be reduced to lower the gain of the filters at the point where the bandpass filters meet. It is desired to both keep the center frequencies from section 5.3.1 and have each band be the same size. The bands are shrunken by changing the ratio between the center frequencies ω_{Cn} and the - 3 dB frequencies ω_{Ln} and ω_{H-n} defined as $r = \frac{\omega_{Ln}}{\omega_{Cn}} = \frac{\omega_{Cn}}{\omega_{Hn}}$, which was initially set to $r = \frac{1}{\sqrt{2}} \approx 0.7017$. As this ratio r approaches 1, the -3 dB frequencies will get closer to the center frequencies, which will make the bands more narrow. To find a value of r that will satisfy the requirements, the magnitude response of the summed filter responses has been plotted in MATLAB. These can be seen in figure 5.10.

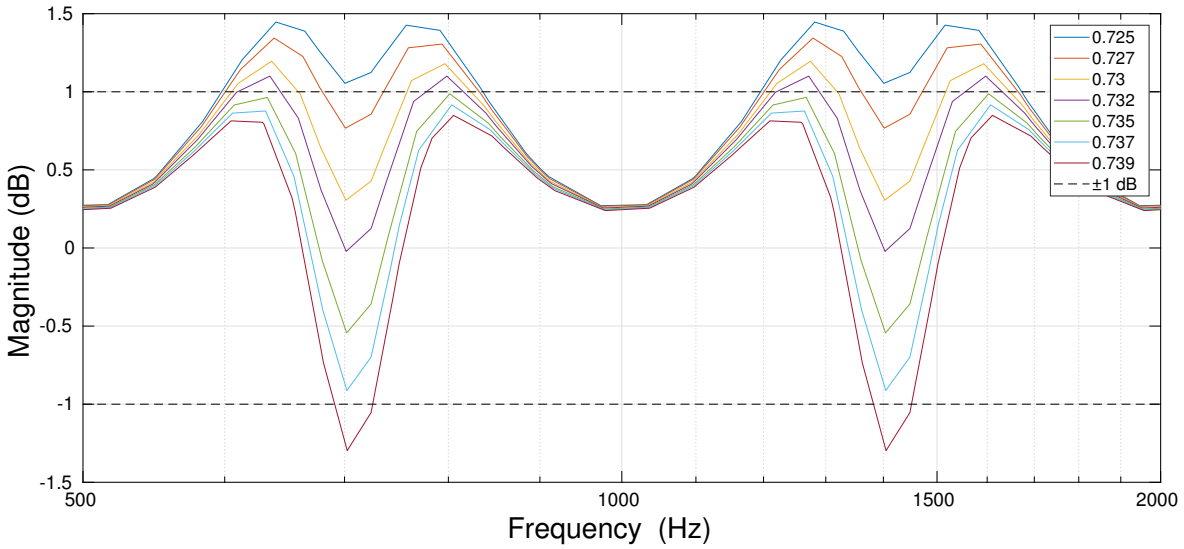

Figure 5.10: Summed filter responses with different values of r, around the 1 kHz band.

By graphical estimation it can be seen that the value of $r = 0.73$ results in the least amount of fluctuations. It can be seen that the graph with $r = 0.73$ goes from about 0.2 dB to 1.2 dB. To make the midpoint be at 0 dB the output needs to be attenuated with a factor of $-\frac{(1.2-0.2)\text{dB}}{2} = -0.5\text{dB} = 0.94$. New 3 dB frequencies can now be calculated as

$$f_{Ln} = 0.73 \cdot f_{Cn} \tag{5.3.8}$$

and

$$f_{Hn} = \frac{f_{Cn}}{0.73} \tag{5.3.9}$$

These values are computed for each center frequency and are listed in table 5.2

Band	f_C [Hz]	f_L [Hz]	f_H [Hz]
1	31.25	22.813	42.808
2	62.5	45.625	85.616
3	125	91.250	171.23
4	250	182.50	342.47
5	500	365.00	684.93
6	1 k	730.00	1.370 k
7	2 k	1.460 k	2.740 k
8	4 k	2.900 k	5.480 k
9	8 k	5.840 k	10.959 k
10	16 k	11.680 k	21.918 k

Table 5.2: A bunch of frequencies

Since the edge frequencies of adjacent bandpass filters are still pretty close to each other, this frequency shifting is assumed to not have a big impact on the phase response of the equalizer compared to the one seen in figure 5.9.

It is now possible to determine the transfer functions of each of the bandpass filters.

A way to determine the transfer function of an arbitrary 8th order bandpass filter of a specific filter type, is to initially look at a normalized 4th order lowpass filter of the same type, and then perform a frequency transformation.

The coefficients of a 4th order Butterworth lowpass filter can be looked up in a table. The transfer function can then be written as:

$$H_{LPP}(s) = \frac{1}{s^4 + 2.613s^3 + 3.414s^2 + 2.613s + 1} \tag{5.3.10}$$

where $s = j\omega$ [Rad/s]

To transform this into a bandpass filter, the pass band is shifted along the frequency axis, to be centered around a given center frequency. This shifts the s in the transfer function to $S+\omega_C$. To implement a zero at zero and poles on the each edge of the passband, s is further transformed to $\frac{S^2+\omega_C^2}{S}$. Lastly the transfer function is frequency scaled to cover a given bandwidth $B = \omega_H - \omega_L$, which gives a transformation of:

$$s = \frac{S^2 + \omega_C^2}{S \cdot B} = \frac{S^2 + \omega_C^2}{S \cdot (\omega_H - \omega_L)} \tag{5.3.11}$$

The transfer functions for the 8th order bandpass filter can thus be found as:

$$
\begin{aligned}
H_{BP}(s) &= H_{LPP}\left(\frac{S^2+\omega_C^2}{S\cdot B}\right) \\
&= \frac{1}{\left(\frac{S^2+\omega_C^2}{S\cdot B}\right)^4 + 2.613\left(\frac{S^2+\omega_C^2}{S\cdot B}\right)^3 + 3.414\left(\frac{S^2+\omega_C^2}{S\cdot B}\right)^2 + 2.613\left(\frac{S^2+\omega_C^2}{S\cdot B}\right) + 1} \\
&= \frac{(S\cdot B)^4}{(S^2+\omega_C^2)^4 + 2.613(S^2+\omega_C^2)^3\cdot SB + 3.414(S^2+\omega_C^2)^2\cdot (SB)^2 + 2.613(S^2+\omega_C^2)\cdot (SB)^3 + (SB)^4}
\end{aligned} \tag{5.3.12}
$$

It can be seen that the system has turned into an 8th order filter with 4 zeros at zero and 8 poles i.e. an 8th order bandpass filter. Thus the 10 bandpass filter transfer function can be found by inserting the values found in table 5.2 into equation 5.3.12.

As an example the transfer function on the filter centered around 8 kHz will be calculated.

For this filter it is known that:

$$
\begin{aligned}
B_9 &= 2\pi\cdot(f_{H9}-f_{L9}) = 2\pi\cdot(10.96\text{kHz}-5.84\text{kHz}) = 32.16\cdot 10^3\text{rad/s} \\
\omega_{C9} &= 2\pi\cdot f_{C9} = 2\pi\cdot 8\text{kHz} = 50.26\cdot 10^3\text{rad/s}
\end{aligned}
$$

By inserting these values into equation 5.3.12 we get the following transfer function:

$$
H_9(s) = \frac{9.85\text{E}17s^4}{s^8 + 8.41\text{E}4s^7 + 1.36\text{E}10s^6 + 7.24\text{E}14s^5 + 5.72\text{E}19s^4 + 1.83\text{E}24s^3 + 8.71\text{E}28s^2 + 1.36\text{E}33s + 4.08\text{E}37} \tag{5.3.13}
$$

Solving the roots of the polynomial in the denominator gives you that poles of the filter.

$$
p_9 = \{-7.910 \pm 66.940i, -16.745 \pm 54.603i, -12.970 \pm 42.295, -4.399 \pm 37.225\}\cdot 10^3\text{rad/s}
$$

5.3.3 Transformation to the digital domain

Since the equalizer is going to be implemented on a raspberry pi, it is necessary that the filters handles time discreet signals. Accordingly the transfer functions will be transformed to the z-domain. There are a couple of different ways to obtain a z-domain transfer function from an s-domain one. For this specific task the bilinear transformation is chosen since this method largely preserves the shape of the original Butterworth filters. The bilinear transform is derived in appendix C, but in short the bilinear transform simply maps the s-plane to the z-plane.This is done by replacing s with:

$$
s = \frac{2}{T_d}\frac{z-1}{z+1} \tag{5.3.14}
$$

As it can be seen from equation C.0.11 it is necessary to pre-warp the s-domain transfer function to get the desired z-domain transfer function using the formula:

$$
\omega_a = \frac{2}{T_d}\tan\left(\frac{\omega_d T_d}{2}\right) \qquad [\text{rad/s}] \tag{5.3.15}
$$

As the audio interface of the system is sampling and playing back at a sampling rate of 44.1 kHz, the discretization time T_d is set to $(44.1\text{kHz})^{-1} = 22.676\mu\text{s}$. To preserve the shape of the bandpass filters it is important that both the higher and lower cutoff frequencies are preserved. This is accomplished by pre-warping these frequencies, and redoing the lowpass to bandpass transform form equation 5.3.12 with these new values. Once again the filter centered around 8 kHz is used as an example.

$$\begin{aligned}\omega_{L,a} &= \frac{2}{T_d}\tan\left(\frac{\omega_{L,d}T_d}{2}\right) = \frac{2}{22.676\mu\text{s}}\tan\left(\frac{2\pi\cdot 5.84\text{kHz}\cdot 22.676\mu\text{s}}{2}\right) \\ &= 38.968\cdot 10^3\text{rad/s}\end{aligned} \tag{5.3.16}$$

$$\begin{aligned}\omega_{H,a} &= \frac{2}{T_d}\tan\left(\frac{\omega_{H,d}T_d}{2}\right) = \frac{2}{22.676\mu\text{s}}\tan\left(\frac{2\pi\cdot 10.96\text{kHz}\cdot 22.676\mu\text{s}}{2}\right) \\ &= 87.373\cdot 10^3\text{rad/s}\end{aligned} \tag{5.3.17}$$

These values can be used to compute a new center frequency and bandwidth.

$$\begin{aligned}B_a &= \omega_{H,a} - \omega_{L,a} = 87.373\cdot 10^3\text{rad/s} - 38.968\cdot 10^3\text{rad/s} = 48.405\cdot 10^3\text{rad/s} \\ \omega_{C,a} &= \sqrt{\omega_{H,a}\cdot\omega_{L,a}} = \sqrt{87.373\cdot 10^3\text{rad/s}\cdot 38.968\cdot 10^3\text{rad/s}} = 58.351\cdot 10^3\text{rad/s}\end{aligned} \tag{5.3.18}$$

By once again inserting these values into equation 5.3.12, a pre-warped analog transfer function has been found in equation.

$$H_{a,9}(s) = \frac{5.49\text{E}18s^4}{s^8 + 1.26\text{E}5s^7 + 2.16\text{E}10s^6 + 1.59\text{E}15s^5 + 1.30E20s^4 + 5.41\text{E}24s^3 + 2.51\text{E}29s^2 + 5.0\text{E}33s + 1.34\text{E}38} \tag{5.3.19}$$

This transfer function is then converted to the z-domain using the bilinear transform.

$$\begin{aligned}H_9(z) &= H_{a,9}\left(\frac{2}{T_d}\frac{z-1}{z+1}\right) = H_{a,9}\left(\frac{2}{22.676\mu\text{s}}\frac{z-1}{z+1}\right) \\ &= \frac{0.007978z^8 - 0.03191z^6 + 0.04787z^4 - 0.03191z^2 + 0.007978}{z^8 - 2.39z^7 + 4.293z^6 - 4.885z^5 + 4.599z^4 - 3.023z^3 + 1.637z^2 - 0.5459z + 0.1413}\end{aligned} \tag{5.3.20}$$

It can be seen from figure 5.11 that this new filter transfer function $H_9(z)$ largely retains the frequency response of the original analog filter $H_9(s)$ from equation 5.3.13.

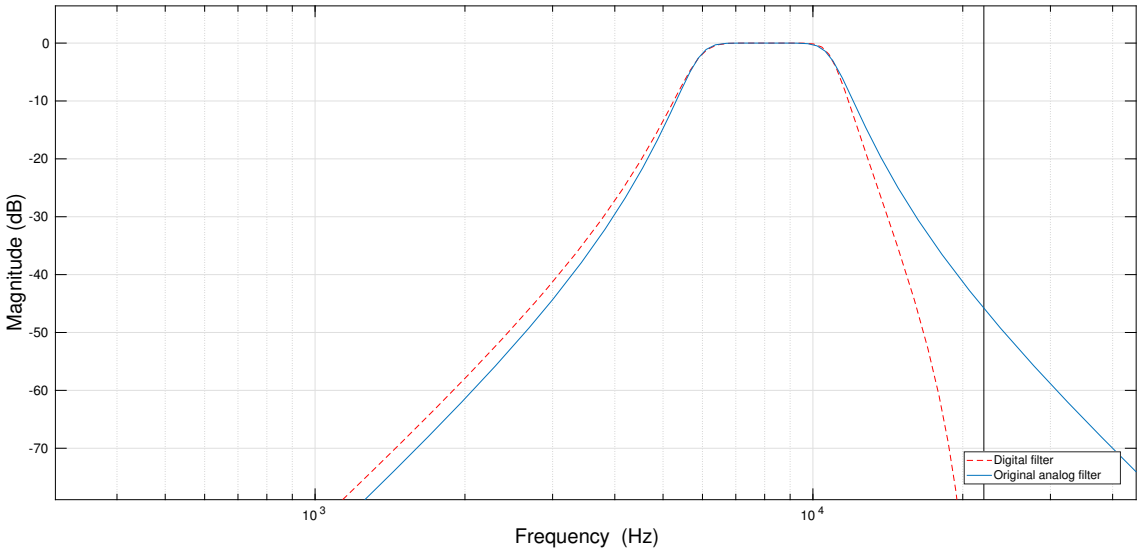

Figure 5.11: The 8 kHz band transfer function in s- and z-domain.

Though it can be seen that the digital filter attenuates lower frequencies less than the original analog filter, this is assumed to not have a big effect since deviation only looks significant at frequencies where the magnitude is already under -20 dB. At these frequencies there will be other bands with a gain of approximately 0 dB, so the deviations are assumed to have a negligible effect on the overall response of the equalizer.

It is possible to directly convert the transfer function from equation 5.3.20 but since the transfer function is quite unwieldy it has been decided to implement it as 4 2nd order filters in cascade. Another reason for this is to ensure that poles of a filter are not close enough together to give error in the coefficients due to quantization cite signal procesisng with ove. This can be ensured by only letting a single filter have a single complex conjugated pair of poles.

These 4 filters are each constructed from two poles and two zeros from complete transfer function $H_9(z)$. The poles and zeros are found by solving the roots of the denominator and numerator in $H_9(z)$ respectively. The poles and zeros can be seen in figure 5.12

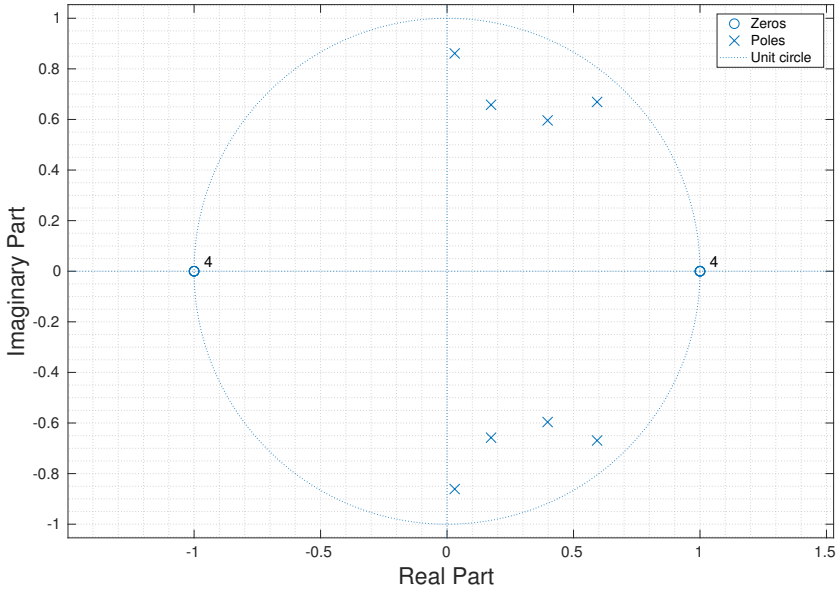

Figure 5.12: The poles and zeroes of the digital filter $H_9(z)$.

There are 4 zeros in 1 (0 Hz) and 4 zeros in -1 (half the sampling frequency $F_s/2 = 22.05$ kHz). Once again there are four pairs of complex conjugated pole pairs in

$$\text{Poles} = \{0.5932 \pm 0.6692j, 0.0303 \pm 0.8610j, 0.3976 \pm 0.5962j, 0.1740 \pm 0.6580j\} \quad (5.3.21)$$

All the 2nd order filters will each have a complex conjugated set of poles and two zeros. By constructing a 2nd order filter from the zeros at -1, the filter will be a lowpass filter, and by using the zeros at 1 the filter will be a highpass filter.

A common way$_{tm}$ to choose which poles should go with which zeros is to start with the pole closest to the unit circle and pair it with its closest zero. In this case the poles closest to the unit circle are the poles at $0.5932 \pm 0.6692j$.[3] These two poles will then be paired with two zeros in 1.

As the 8th order filter is timed by a constant of $7.978 \cdot 10^{-3}$ every 2nd order filter will be made with constant factor:

$$c = (7.978 \cdot 10^{-3})^{1/4} = 0.2989 \quad (5.3.22)$$

So the first filter of the cascade structure can now be determined as:

$$H_{9,1}(z) = c \cdot \frac{(z-1)(z-1)}{(z-p_1)(z-p_1^*)} = \frac{c - 2cz^{-1} + cz^{-2}}{1 - 1.186z^{-1} + 0.7997z^{-2}} \quad (5.3.23)$$

The same is repeated for the next closest pole in $, 0.0303 \pm 0.8610j$

$$H_{9,2}(z) = c \cdot \frac{(z-1)(z-1)}{(z-p_2)(z-p_2^*)} = \frac{c - 2cz^{-1} + cz^{-2}}{1 - 0.06054z^{-1} + 0.7423z^{-2}} \quad (5.3.24)$$

[3] This is determined by looking at the absolute value of the poles.

The next two filters will then be constructed with the remaining poles and the 4 zeros in -1. These filters will then be lowpass filters.

$$H_{9,3}(z) = c \cdot \frac{(z+1)(z+1)}{(z-p_3)(z-p_3^*)} = \frac{c + 2cz^{-1} + cz^{-2}}{1 - 0.7952z^{-1} + 0.5136z^{-2}}$$
$$H_{9,4}(z) = c \cdot \frac{(z+1)(z+1)}{(z-p_4)(z-p_4^*)} = \frac{c + 2cz^{-1} + cz^{-2}}{1 - 0.3481z^{-1} + 0.4632z^{-2}}$$

The amplitude responses of these 4 filter can b seen in figure 5.13.

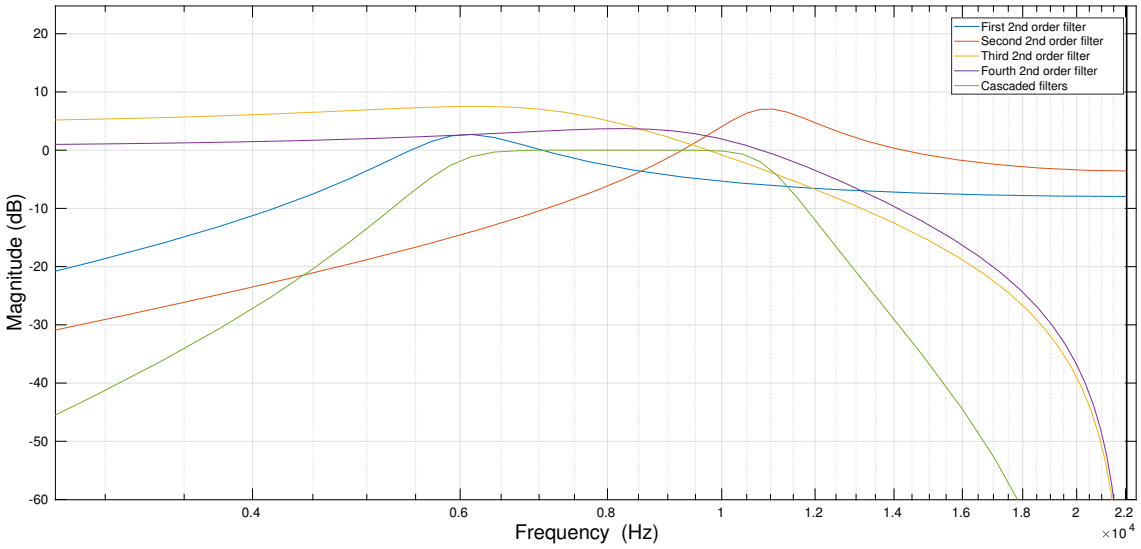

Figure 5.13: cascaded filters

Despite the fact that neither of the filter looks like a butterworth filter on their own, the result of the cascaded filters looks like the wanted filter. These transfer function can then be transformed into a difference equation by doing a inverse z-transform, like it is shown in equation 5.3.25.

$$H_{9,1}(z) = \frac{Y_{9,1}(z)}{X_{9,1}(z)} = \frac{0.2989 + 0.5977z^{-1} + 0.2989z^{-2}}{1 - 1.186z^{-1} + 0.7997z^{-2}} \tag{5.3.25}$$
$$\iff Y_{9,1}(z) \cdot (1 - 1.186z^{-1} + 0.7997z^{-2}) = X_{9,1}(z) \cdot (0.2989 + 0.5977z^{-1} + 0.2989z^{-2})$$
$$\iff y_{9,1}[n] - 1.186y_{9,1}[n-1] + 0.7997y_{9,1}[n-2] = 0.2989x_{9,1}[n] - 0.5977x_{9,1}[n-1] + 0.2989x_{9,1}[n-2]$$

The same can be done for all the other 2nd order filters.

Here comes a part explaining how the first 9 filter can be made using downsampling - It is not yet finished.

5.3.4 Downsampling

However not every bandpass filter will behave like they should when implemented in this way. This is illustrated in figure 5.14, where it can be seen that the 16 kHz band has a weird shape

compared to what the butterworth bandpass filter is supposed to look like, and every band bellow the 250 Hz band has neither the correct central frequency, bandwidth or gain.

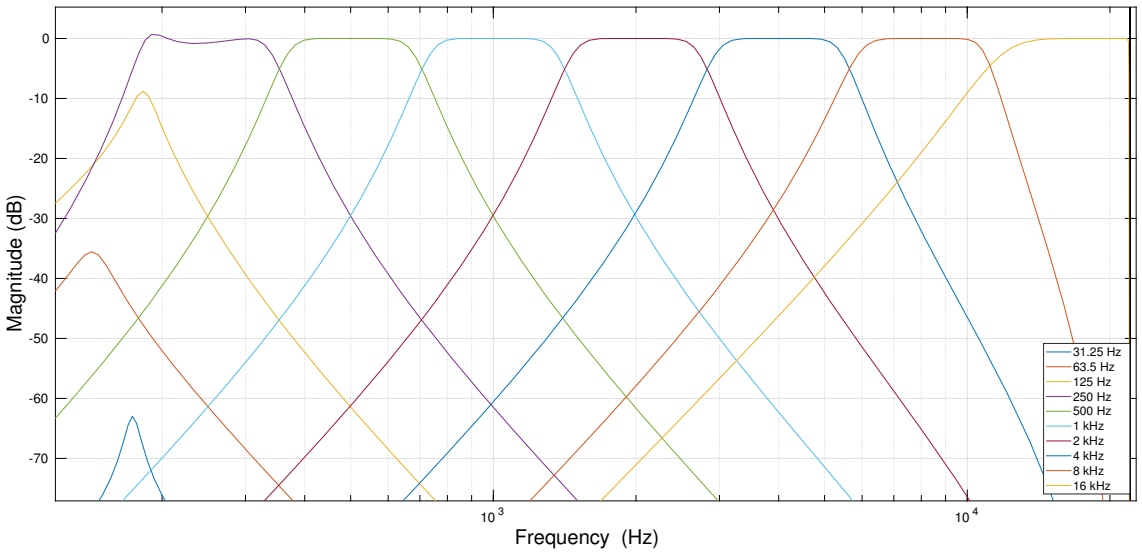

Figure 5.14: Amplitude responses of bandpass filters optained with the bilinear transform.

The reason for this distortion of the amplitude response can be seen by looking at the poles of the last four bands which can be seen in figure 5.15

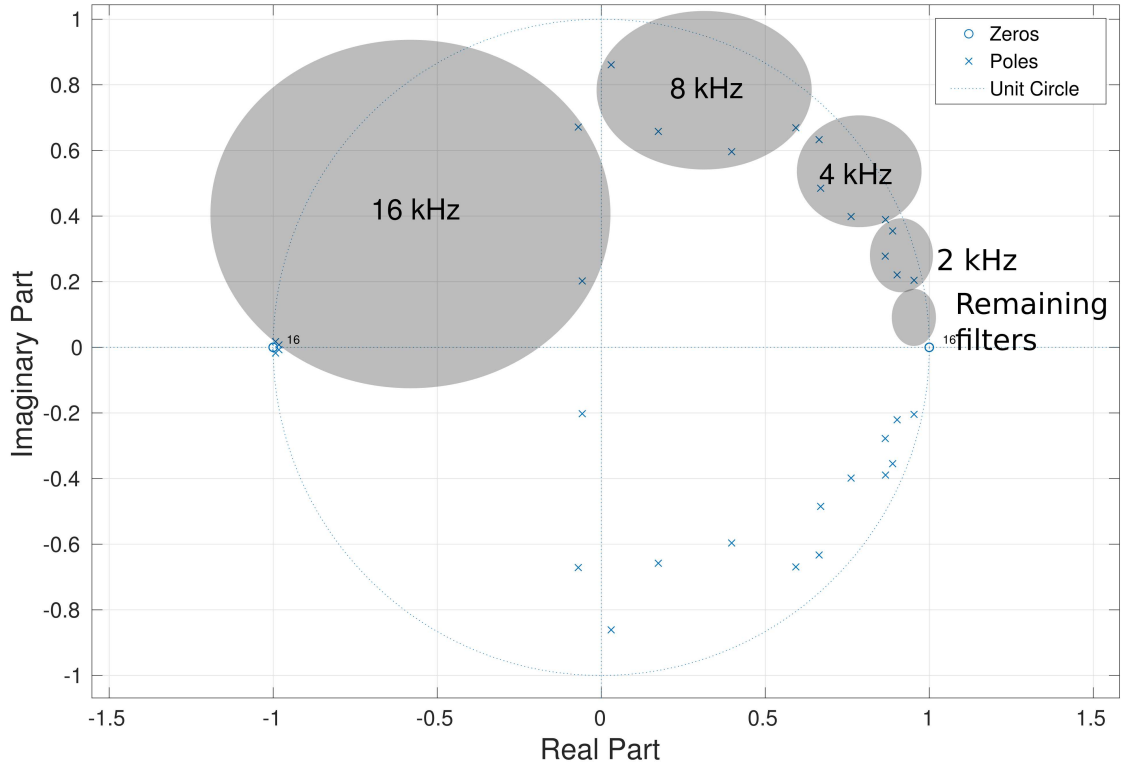

Figure 5.15: Poles and zeros of the 2 kHz to 16 kHz bands.

In figure 5.15 the 16 kHz band has to poles almost at half the sampling frequency (at -1). This causes the shape of the right side of the passband of the 16 kHz filter. As the central frequency of the bands gets closer to 0, the poles gets both more squished together closer to the unit circle at 1. Both of these properties leads to instability of the filters.

Obviously another way of implementing the filters is necessary.

The "frequency" of a digital filter transfer function is given as:

$$\omega = \frac{2\pi \cdot f}{f_s} \qquad \text{[rad/Sample]} \tag{5.3.26}$$

Questions for SKO: we have already used lower case omega as a symbol for radian frequency. is it okay to use it for both? if not which one should have another symbol?

Of course the same relation follows for the noteworthy frequencies like the central and edge frequencies.

Consider having a z-domain transfer function with a specific central frequency ω_c and using the half the sampling frequency originally intended.

$$\begin{aligned} & 2\pi \cdot f_c = \omega_c \cdot f_s \\ \iff & 2\pi \cdot \frac{f_c}{2} = \omega_c \cdot \frac{f_s}{2} \qquad \text{[rad/s]} \end{aligned} \tag{5.3.27}$$

From equation 5.3.27 it can be seen that by only sampling the input half the time, the actual frequency of the filter will behave as if the central frequency was an octave lower[4].

This technique is known as downsampling.

Since all the bandpass filters are an octave apart, you could for example sample the input of the 8 kHz filter found in section 5.3.3 at half the sampling rate of the input, and make the filter behave like the wanted 4 kHz filter. As the z-domain transfer function will remain the same, the poles and zeros are unchanged as well. This means that this new 4 kHz filter will have the exact same pole-zero plot as shown in figure **??** expect that the point at -1 corresponding to half the original sampling frequency 22.05 kHz will now correspond to half of this namely 11.025 kHz.

This can eliminate the problems related to poles being to close to each other and to the unit circle.

Another advantage is that a filter made from downsampling a filter an octave above will need only half as many computations since only every other sample is read.

So all the lower bands of the equalizer could essentially be made by taking a stable bandpass filter and downsampling it by a factor of 2^n where n is an integer.

something about upsampling and zero filling

checkpoint - might want to change this if stuff can't live up to requirements

[4] This of course also applies to the edge frequencies

It is of course always useful to lessen the number of computations the system has to do to filter one buffer.

Since the computations of the filtering is approximately halved when a filter is downsampled to an octave below, quartered when downsampled two octaves et.c, it is most efficient to have every filter downsampled as much as possible. For this reason it has been decided to implement every filter below the 8 kHz band, as a downsampled version of the 8 kHz filter exemplified in section 5.3.3. The amplitude responses of these filters can be seen in figure 5.16.

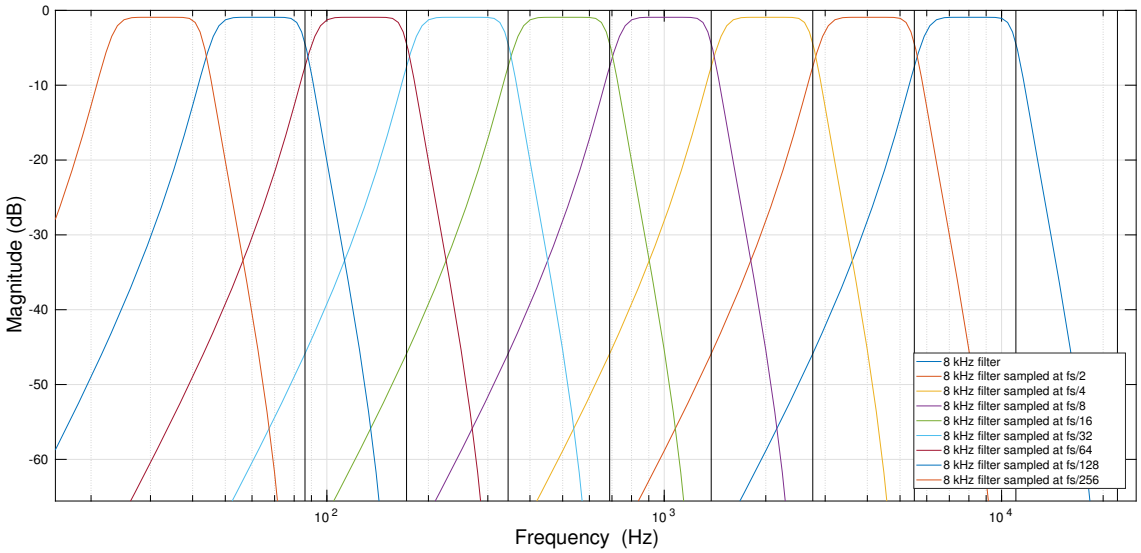

Figure 5.16: The first 9 filters implemented using the same transfer function

more to come

Programming of the filters

Now that the filters are calculated and transformed into discrete time domain, it is possible to program these in C. First the structure of the code is examined. Each filter is spit into two second order highpass and two second order lowpass filters, that can then be programmed so the output of one filter becomes the input of the next. When down sampling aliasing also becomes a problem so anti aliasing filters have been designed as well. The program flow for the implementation of the first nine filters can be seen in figure 5.17. The filter at 16 kHz is not included here because this have different coefficients.

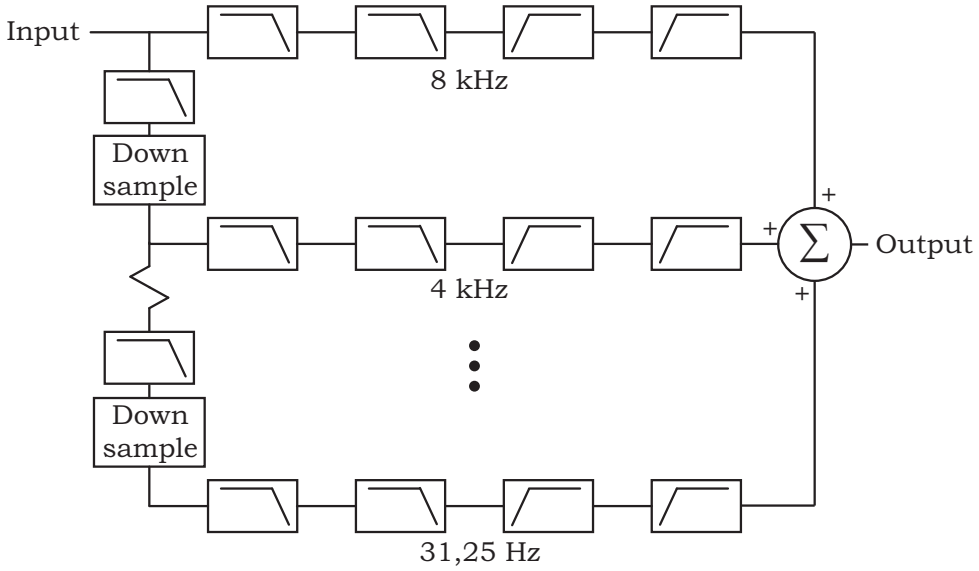

Figure 5.17: Sketch over the program flow for the filters

First off the input from the buffer is filtered using the 8 kHz bandpass. The output here is then saved in a buffer so it can be added together with the other filter outputs. Next the input is lowpass filtered using the previously described anti aliasing filter. Afterwards its downsampled and send to the next filter. This process is then repeated until the input has been downsampled and filtered through the 31.25 Hz filter. All the outputs from the filters are then summed and are ready to be played. First off the setup for the function that handles the bandpass filtering can be seen in code example 5.7.

Code example 5.7: Function for the bandpass filters

```
1 void bandpassFilter(int m, double ratio) {
2   double output;
3   int k = 0;
4   int dsRatio = power(2,m);
```

The bandpass function is getting two inputs, m and ratio. The input m is telling the function witch filter is currently being processed. Here a zero would be the 8 kHz filter, a one would be 4 kHz and so forth. The ratio input is used to scale the output so the controller can turn up or down for a specific bandpass filter. At the start of the function some variables are declared. The double output is used for the ratio calculations. The integer k is a variable used for controlling the loop that handles the filters. Next the downsampling ratio is calculated. This is done with the power function that returns two to the power of m. If the input have been downsampled two times and there will only be a fourth of the number of samples to process so the number of samples is calculated as the size of a frame divided by the downsampling ratio.

Code example 5.8: Function for the bandpass filters

```
1 while(k != frameSize/dsRatio) {
2   if(m == 0) {
3         x[m][0] = inputArray[k];
4   } else {
5         x[m][0] = aaArray[k];
6   }
7   // The filter calculations
8   y[m][0][0] = x[m][0] - 2*x[m][1] +  x[m][2] +  coef[0]*y[m][0][1] - coef[1]*y[m][0][2];
```

```
9   y[m][1][0] = y[m][0][0] - 2*y[m][0][1] + y[m][0][2] + coef[2]*y[m][1][1] - coef[3]*y[m][1][2];
10  y[m][2][0] = y[m][1][0] + 2*y[m][1][1] + y[m][1][2] + coef[4]*y[m][2][1] - coef[5]*y[m][2][2];
11  y[m][3][0] = y[m][2][0] + 2*y[m][2][1] + y[m][2][2] + coef[6]*y[m][3][1] - coef[7]*y[m][3][2];
12  output = ratio * con * y[m][3][0];
```

The next part of the function is a while loop that repeats the filter calculations until the available samples have been processed. This can be seen in code example 5.8. In the start of the while loop the input for the filter is updated. If it is the 8 kHz filter the input will come directly from the buffer else it will be the downsampled output from the anti aliasing filter. Now that the parameters for the filter and the inputs have been clarified, it is possible to do the actual filter calculations. The first line is equivalent to the first lowpass filter. The output of the first filter depends on the input, the previous inputs and the previous outputs of the filter. For all the filters there are some coefficients that the previous outputs needs to be multiplied with. These are stored in the array called coef. The output from the first filter is then used as the input for the next filter. This procedure is repeated until all the filters have been processed. After the filter calculations are done there is a constant that the output needs to be multiplied with. Here it is also multiplied with the ratio given by the controller.

Code example 5.9: Function for the bandpass filters

```
1     // Updating the varables
2     x[m][2] = x[m][1]; x[m][1] = x[m][0];
3     y[m][0][2] = y[m][0][1]; y[m][0][1] = y[m][0][0];
4     y[m][1][2] = y[m][1][1]; y[m][1][1] = y[m][1][0];
5     y[m][2][2] = y[m][2][1]; y[m][2][1] = y[m][2][0];
6     y[m][3][2] = y[m][3][1]; y[m][3][1] = y[m][3][0];
7     // Saving the data in an array for the output
8     outputArray[k*dsRatio] = outputArray[k*dsRatio] + output;
9     k++;
10  }
11  return;
12 }
```

The last part of the function can be seen in code example 5.9. Since the filter need the previous inputs and outputs in order to do the calculations the variables that are needed are updated one at a time. This variables are stored as global arrays so they can be saved between frames. The structure of the variables are a series of arrays where the first signifies the bandpass number, the next is equivalent to the filter number and the last is for the previous values. After the variables is updated the output for the rest of the system need to be updated as well. Because zero padding is needed the output is just added to their equivalent point in the output array. Last in the function the counter is updated for the next run trough of the while loop. With these lines of code the function for filtering and add justing a specific bandpass filter.

For downsampling and anti aliasing separate function have been made. This can be found in code example D.1 in the appendix. The code concerning the anti aliasing filter have the same structure as the code for the bandpass filters, so this will not be explained further. The downsampling can be found in code example 5.10.

Code example 5.10: Downsampling data

```
1 if(i%2 == 0) {
2       aaArray[i/2] = aaOut[n][0];
3 }
```

The downsampling is done by only saving every other sample in the output array for the anti aliasing filter. For the summed output of the bandpass filters zero padding is needed, however in this case it is not necessary. To save on memory the same array is being used for the output of the anti aliasing filter, as it is only needed one time in the bandpass filters. Afterwards the program moves to the next filter and need a new set of downsampled input values.

He is an idiot (”’\(O_o)/”’)

\/

5.4 Design of controller

The purpose of the controller is to regulate the gain of the different bands in the equalizer, based on audioSupposedToBe and audioIs

5.4.1 Ramblings of a mad man

The control unit must compare the reference with the actual audio. There is a lot of ways to do this. The best would probably be to DFT both of the mother humpers, however they are not in phase, so it is hard to compare. Even if the phase difference is found for one measurement, it could be different for the next. A lot of calculations is therefore needed to compare the two with DFTs. To be honest a DFT is kind of overkill since it gives all the info of all the frequencies, and we just need to know how to adjust the gain of the eq-bands.

Another way is to get the rms value from the different bands. Since the eq is implemented, it's easy to nick an RMS value at the right time. All that is needed is to get the RMS at the same bands from the mic. Of course there is a gain difference between the RMS's's, due to volume and distance from speaker to mic, however this should be the same across all the bands.

RMS-way is therefore best way since less calculation and the info we get is just what we need.

Glossary

Ad hoc Ad hoc is a wireless networks that does not use any preexisting infrastructure.. 21

ALSA Advanced Linux Sound Architecture. A set of software modules on the Linux kernel, that handles interaction with sound cards.. 8

Downsampling A way of changing the sample rate of a signal by only taking every n sample of some integer n.. 37

Equalizer An electronic device for adjusting the frequency response of a system.. 4

FIR Filter Finite impulse response filters. Filters whose impulse responses are zero after a certain point. see. 7

Frequency response The way a system output behaves in terms of magnitude and phase, as a function of the frequency of the input. 1

IIR Filter Infinite impulse response filters. Filters whose impulse responses continues forever. This property is known from the impulse response of analog filters. see. 7

Rasberry Pi Series of single board computers, often used for small projects.. 9

Bibliography

[1] Last seen: 21/09/17. URL: https://www.raspberrypi.org/app/uploads/2014/07/rsz_b-.jpg.

[2] *Advanced Linux Sound Architecture*. Last seen: 13/09/17. URL: https://wiki.archlinux.org/index.php/Advanced_Linux_Sound_Architecture.

[3] AKM. *AK5371 microphone*. Last seen: 23/10/2017. URL: https://media.digikey.com/pdf/Data%20Sheets/AKM%20Semiconductor%20Inc.%20PDFs/AK5371.pdf.

[4] *Bluetooth Core Specification v5.0*. Bluetooth SIG Proprietary. 2016. URL: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043&_ga=2.131475212.2088466289.1505901779-621389646.1505901779.

[5] International Electrotechnical Commission. *Audio Recording - Compact Disc digital audio system*. 1999.

[6] *ISO 266:1997 - Acoustics – Preferred frequencies*. ISO. 1997.

[7] Bjorn Roche. *Analog vs. Digital EQ*. Last seen: 20/09/17. URL: http://www.xowave.com/doc/effect/eq/analogvsdigital.shtml.

[8] Roland. *UA-25EX Sound card*. Last seen: 18/10/2017. URL: http://cdn.roland.com/assets/media/pdf/UA-25EX_OM.pdf.

[9] Margaret Rouse. *definition: microcontroller*. Last seen: 13/09/17. URL: http://internetofthingsagenda.techtarget.com/definition/microcontroller.

[10] Gary W. Siebein. "Architectural acoustics". In: *Access Science* (2014). URL: http://www.accessscience.com.zorac.aub.aau.dk/content/architectural-acoustics/048700.

[11] *Sound Card Components*. Last seen: 13/09/17. URL: https://www.pctechguide.com/sound-cards/sound-card-components.

[12] Steve Taranovich. *Integration Choices: Analog Filters vs. Digital Filters*. Last seen: 21/09/17. URL: https://www.planetanalog.com/author.asp?section_id=3065&doc_id=560512.

[13] Floys Toole. "Sound-reproducing systems". In: *Access Science* (2014). URL: http://accessscience.com.zorac.aub.aau.dk/content/637630.

[14] *Types of EQ*. Last seen: 25/09/17. URL: http://amusicproducer.com/types-of-eq/.

[15] *What is an equalizer?* Last seen: 20/09/17. URL: http://www.yamahaproaudio.com/global/en/training_support/selftraining/pa_guide_beginner/equlizer/.

[16] J. William Whikehart. *Signal Processing*. Last seen: 21/09/17. URL: https://www.accessscience.com/content/signal-processing/757755#757755s011.

[17] Wikipedia. *Equalization (audio)*. Last seen: 20/09/17. URL: https://en.wikipedia.org/wiki/Equalization_(audio).

[18] Wikipedia. *Figure about audio bit debt*. Last seen: 13/09/17. URL: https://en.wikipedia.org/wiki/Audio_bit_depth#/media/File:4-bit-linear-PCM.svg.

[19] Wikipedia. *Figure about Nyquist–Shannon sampling theorem*. Last seen: 12/09/17. URL: https://en.wikipedia.org/wiki/File:CPT-sound-nyquist-thereom-1.5percycle.svg.

[20] Wikipedia. *Nyquist–Shannon sampling theorem*. Last seen: 12/09/17. URL: https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem.

[21] Tracy V. Wilson. *How Sound Card Works*. Last seen: 13/09/17. URL: http://computer.howstuffworks.com/sound-card3.htm.

Wireless bit rate approximation A

Purpose

The purpose of this test is to find an approximation bit rate between the microphone unit and stationary unit, and to test the bite rate, on changing distance. A bit rate of 2 Mbps (0.25 MB/s) is needed specified in 4.6, it is expected that the step-up will achieve bite rates well above the wished 2 Mbps.

Procedure

To test the wireless connection speed 10.000 packets of 40.000 bytes, a total of 400 MB each, are send from the client to the host using the chosen UDP protocol. This is then timed so that the connection speed can be calculated. The time is taken using a stop watch on a cellphone, starting the same time the programs executes, and the time is stopped then the transmission is done.

Since this procedure is prone to human error, and the software in use is not fully optimized for raw transmission speed. The results will not be exact, yet since it is only desired to know if the bite rate is well above the bite rate required, or if an more extensive examination is needed.

Materials

- Raspberry Pi 3 Model B Vi 3
- Raspberry Pi Zero W
- Stopwatch
- Measuring Tape

For the test C code is used to send the packages UDP-Client_Speedtest and UDP-Server_Speedtest

Results

$$Messagelength = 10.000 Int32 = 40.000\text{MB} \tag{A.0.1}$$

$$Messagessend = 10.000 \tag{A.0.2}$$

$$Datasend = 10.000 \cdot 40.000 = 400\text{MB} \tag{A.0.3}$$

$$Distance = 0, 5, 1 \text{ and } 5\text{m} \tag{A.0.4}$$

Test	1	2	3	4	5	unit
Measured time	97	96	93	71	95	[s]
Distance	1	1	1	0,5	5	[m]
Connection speed	4,12	4,17	4,3	5,63	4,44	[MB/s]

Table A.1: Data from tests

Conclusion

The bit rate between the two raspberry Pi units is well above the requirements of 0.25MB/s, even with uncertainties such as distance between the units, human error in the test and interference from other sources on the 2.4 or 5.8 GHz bands.

Test of package loss rate B

Purpose

To test as to what extent package loss should be expected for the wireless interface. So a decision can be made whether package loss is an issue, and if so how it should be addressed.

Little to non package loss is expected, because of the units communicating in ad hoc mode. Meaning there is no other users to disturb the communication. The only disturbance would be heavy interference from other users of the same frequency as the ad hoc network uses (2.4 GHz).

Procedure

Is to continuously transmit a known value (every value possible by 16-bit) from the microphone un it to the stationary unit, a set amount of times. The stationary unit then sums these values, if the sum is incorrect it is considered a package loss, and the time and value is then noted. the test is running for 16 hours.

Materials

- Raspberry Pi 3 Model B Vi 3 (stationary unit)
- Raspberry Pi Zero W (microphone unit)

For the test C code is used to send the packages UDP-Client_packageloss and UDP-Server_packageloss

Results

The test have been conducted for 16 hours, and about 250 GB have been transmitted between the two units, no package loss is recorded during this test.

Conclusion

As expected there have not been a single package loss during the 16 hours, the test have been conducted.

The bilinear transform C

The bilinear transform is obtained by looking at the Laplace transform of a discreet time function i.e a continuous time function multiplied by an impulse train $\sum_{n=0}^{\infty} \delta(t - nT_d)$ where T_d is the sample time $1/f_s$.

$$x(t) \cdot \sum_{n=0}^{\infty} \delta(t - nT_d) \tag{C.0.1}$$

As the impulse train is zero at every $t \neq nT_d$ the equation can be rewritten as

$$\sum_{n=0}^{\infty} x(t)\delta(t - nT_d) = \sum_{n=0}^{\infty} x(nT_d)\delta(t - nT_d) \tag{C.0.2}$$

The discreet time function $x[n]$ is defined as the value of $x(t)$ at every sampling time, which is $x(nT_d)$ for every integer n. That means the equation can be written as

$$\sum_{n=0}^{\infty} x[n]\delta(t - nT_d) \tag{C.0.3}$$

By taking the Laplace transform of this you will get

$$\begin{aligned} X_d(s) &= \int_0^{\infty} \sum_{n=0}^{\infty} x[n] \cdot \delta(t - nT_d) \cdot e^{-st} \mathrm{d}t \\ &= \sum_{n=0}^{\infty} x[n] \int_0^{\infty} \cdot \delta(t - nT_d) \cdot e^{-st} \mathrm{d}t \end{aligned} \tag{C.0.4}$$

As the delta function has a value of 1 at exactly $t = nT_d$ the integral reduces to just e^{-snT_d}.

$$X_d(s) = \sum_{n=0}^{\infty} x[n]e^{-snT_d} \tag{C.0.5}$$

This is exactly the same as the z-transform

$$X(z) = \sum_{n=0}^{\infty} x[n]z^{-n} \tag{C.0.6}$$

where $z = e^{sT_d}$

Isolating for s you get

$$\begin{aligned} z = e^{sT_d} &\iff \\ sT_D = \ln(z) &\iff \\ s = \frac{1}{T_d} \ln(z) & \end{aligned} \tag{C.0.7}$$

Thereby a transfer function in the analog s-domain can be transformed to the digital z-domain by simply replacing s with $\frac{1}{T_d}\ln(z)$. However since it is desired to keep the transfer functions linear after the transformation, $\ln(z)$ is approximated to $2\frac{z-1}{z+1}$. The variable s can then be given as

$$s \approx \frac{2}{T_d}\frac{z-1}{z+1} \tag{C.0.8}$$

By substituting s in an analog transfer function with this expression you will therefor get an equivalent transfer function in the z-domain.

Frequency warping

One problem with the bilinear transform is that the frequency axis of a discreet transfer function does not exactly match the frequency axis of th analog transfer function it is obtained from. This can be seen by substituting s and z with their corresponding frequency axis' in the transformation.

$$j\omega_a = \frac{2}{T_d}\frac{e^{j\omega_d T_d}-1}{e^{j\omega_d T_d}+1} = \frac{2}{T_d}\frac{1-e^{-j\omega_d T_d}}{1+e^{-j\omega_d T_d}} \tag{C.0.9}$$

By simplifying this you optain:

$$\begin{aligned} j\omega_a &= \frac{2}{T_d}\frac{e^{j\omega_d T_d/2}\cdot e^{-j\omega_d T_d/2} - e^{-j\omega_d T_d/2}\cdot e^{-j\omega_d T_d/2}}{e^{j\omega_d T_d/2}\cdot e^{-j\omega_d T_d/2} + e^{-j\omega_d/2}\cdot e^{-j\omega_d/2}} \\ &= \frac{2}{T_d}\frac{e^{-j\omega_d T_d/2}\cdot(e^{j\omega_d T_d/2} - e^{-j\omega_d T_d/2})}{e^{-j\omega_d T_d/2}\cdot(e^{j\omega_d T_d/2} + e^{-j\omega_d T_d/2}} \\ &= \frac{2}{T_d T_d}\frac{2j\cdot\sin(\omega_d T_d/2)}{2\cdot\cos(\omega_d T_d/2)} \\ &= \frac{2j}{T_d}\tan(\frac{\omega_d T_d}{2}) \leftrightarrow \end{aligned} \tag{C.0.10}$$

$$\omega_a = \frac{2}{T_d}\tan(\frac{\omega_d T_d}{2}) \qquad [\text{rad/s}] \tag{C.0.11}$$

and the inverse:

$$\omega_d = \frac{2}{T_d}\arctan(\frac{\omega_a T_d}{2}) \qquad [\text{rad/s}] \tag{C.0.12}$$

This means that if you are looking for a digital filter with a specific behavior at a specific frequency w_d, this this filter can be obtained from an analog filter with this behavior at $w_a = \frac{2}{T_d}\tan(\frac{\omega_d T_d}{2})$.

The important frequencies in an analog system e.g. the cutoff frequencies of a filter should for this reason by pre-warped using the expression from equation C.0.11, if it is desired for the digital system to behave in the same way.

C-code used in the project D

Code example D.1: Function for the anti aliasing filters and downsampling

```
1  void antiAliasing(int n) {
2    int i = 0;
3    int dsRatio = power(2,n);
4    while(i != frameSize/dsRatio) {
5      if(n == 0) {
6        aaIn[n][0] = inputArray[i];
7      } else {
8        aaIn[n][0] = aaArray[i];
9      }
10
11     aaOut[n][0] = 0.2132*aaIn[n][0] + 0.4264*aaIn[n][1] + 0.2132*aaIn[n][2] + 0.3392*aaOut[n][1] -
           0.192*aaOut[n][2];
12     aaIn[n][2] = aaIn[n][1]; aaIn[n][1] = aaIn[n][0];
13     aaOut[n][2] = aaOut[n][1]; aaOut[n][1] = aaOut[n][0];
14     // Downsampling
15     if(i%2 == 0) {
16       aaArray[i/2] = aaOut[n][0];
17     }
18     i++;
19   }
20   return;
21 }
```

Matlab scripts for equalizer calculations

E

Code example E.1: Calculation of filter edge frequencies and transfer functions

```
w0 = [31.25 62.5 125 250 500 1E3 2E3 4E3 8E3 16E3] * 2 * pi;

ratio = 0.73;
order = 4;

for i = 1:10 % The calculations are done for each bandpass filter
    WL(i) = w0(i) * ratio;
    WH(i) = w0(i) * 1/ratio;

    [a(i,:), b(i,:)] = butter(order, [WL(i) WH(i)], 's'); % When 'butter()' receives 2D vector
        as 2nd input, it will automatically make a bandpass filter with WL(i) and WH(i) as edge
        frequencies and with an order of twice the first input.
    H(i) = 0.92 * tf(a(i,:), b(i,:));
end
```

Code example E.2: Bilinear transform of the 8 kHz filter

```
fs = 44100;

WLd = WL(9)/fs;
WHd = WH(9)/fs;

OL = 2*fs*tan(WLd/2); % prewarped edge frequencies
OH = 2*fs*tan(WHd/2);

%optaining S-domain transferfunction
order = 4;
[a, b] = butter(order, [OL OH], 's');
H = tf(a, b);    % Optaining the prewarped transferfunction

%optaining z-domian transferfunction(s)
HD = c2d(H,1/fs,'tustin'); % Perfoming the bilinear transform on the prewarped transfer
     function.
[N,D] = tfdata(HD);  % Optains the numeratorand denominator of the transfer function
N = cell2mat(N);
D = cell2mat(D);
zeroes = roots(N);
poles = roots(D);
zplane(zeroes,poles);
```

Code example E.3: Divison into 4 biquads

```
ts = 1/fs;
z = tf('z',ts);
K = 0.007172; % Constant factor of original transfer function.
```

```
4  c = K^(1/4);
5
6  TF1 = c * ((z - zeroes(1))*(z - zeroes(2)))/((z - poles(1))*(z - poles(2)));
7  TF2 = c * ((z - zeroes(5))*(z - zeroes(6) ))/((z - poles(3))*(z - poles(4)));
8  TF3 = c * ((z - zeroes(3))*(z - zeroes(4)))/((z - poles(5))*(z - poles(6)));
9  TF4 = c * ((z - zeroes(8))*(z - zeroes(7)))/((z - poles(7))*(z - poles(8)));
10 TFT = TF1*TF2*TF3*TF4;
```