Ministerul Educaţiei al Republicii Moldova

Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică şi Microelectronică
Catedra Automatică şi Tehnologii Informaţionale

Admis la susţinere
Şef de catedră: prof.univ.dr. Beşliu V.
_____
„____" _____ 2016

ANALIZA COMPARATIVĂ A ALGORITMILOR DE PROCESARE A SUNETULUI
ANALYSIS OF DIGITAL SOUND PROCESSING ALGORITHMS

Teză de master în
Tehnologii Informaționale

Masterand: _____(Roman Postanciuc)
Conducător: _____ (dr. Dumitru Ciorbă)

Chişinău, 2016

Adnotare

Teza cu tema ”Analiza Comparativă a Algoritmilor de Procesare a Sunetului” constitue din intorducere, trei capitole, concluzii, bibliografie cu 8 titluri, 24 figuri, 2 tabele.

Cuvinte cheie utilizate: Transformata Fourier, Funcție de Transfer, Frecvență de Eșantionare, Filtru Digital, Filtru Analog, Caracteristică de Frecvență, Caracteristică de Fază.

Domeniul de studiu şi obiectivele tezei propuse constituie: studierea și examinarea algoritmelor de procesare a sunetului în general, și a algoritmelor de filtrare a sunetului în particular, implementarea algoritmilor examinate intr-un program funcțional care are oferi capabilități de filtrare a sunetulu identice sau aproape identice de cele ale filtrelor analogice.

Originalitatea științifică a lucrării constă în metoda de utilizare a filtrelor, in sisteme audio digitale cu multe canale, care ne permit direcționarea sunetului spre diferite tipuri de difuzoare eliminînd filtrele analogice, sau, cel putzin, ușurînd modelarea și construirea acestora avînd la dispozoție un filtru digital ușor modificabil.

Abstract

This master thesis under the title ”Analysis of Digital Sound Processing Alorithms” consists of introduction, three chapters, conclusions, bibliography containing 8 titles, 24 figures and 2 tables.

Keywords used: Fourier Transform, Transfer Function, Sampling Frequency, Digital Filter, Analogue Filter, Frequency Response, Phase Response.

The Domain of Study and the objectives of this master thesis are: acknowledgement and analysis of digital sound processing algorithms in general and digital sound filtering in particular, implementation of examined algorithms into a fully functional application having sound filtering capabilities virtually identical to the ones of analogue filters.

The scientific genuineness of this paperwork consists of the method of filters implementation in multi-channel digital audio systems, which allows us to redirect the output signal to different types of audio speakers, excluding analogue filters or, at the very least, facilitating the analogue filter design and building process, having a easily adjustable digital filter.

Contents

Introduction 6

1. The structure of digital audio and the generation of digital audio 7

1.1 Sound recording and digital audio structure 7

1.2 Dynamic range and the bit depth 8

1.3 Sampling rate 11

1.4 Logarithmic scale. Amplitude vs loudness 12

1.5 Waveform generation 13

1.6 Employment of oscillators for waveform generation 16

1.7 Common issues related to representation of digital audio (PCM) 17

2. The Mathematics of Digital Audio Processing. Digital Audio Filters 19

2.1 Usage of Discrete Fourier Transform in Digital Audio Processing 19

2.2 Analysis of digital audio employing the Short Time Fourier Transform and signal convolution 22

2.2.1 Rectangular window 23

2.2.2 Sinc function and the flat-top window 25

2.2.3 Hamming and Hanning windows 26

2.2.4 Blackman-Harris Window 27

2.2.5 MLT/Sine window 27

2.3 Audio filters and their parameters 28

2.4 Analogue filters 31

2.5 The Infinite Impulse Response Filters 32

2.6 Filter stability. Frequency and phase response. 32

2.7 Finite Impulse Response (FIR) filters. Linear phase response 34

2.8 Zero pole analysis 35

2.9 The Standard Second-Order Biquad Filter 36

3. Implementation of digital audio filters 37

3.1 Filtering the signal via spectral changing 37
3.2 Direct filtering of a signal using digital filters 40
3.3 Product implementation 43
Conclusions 46
Appendix A. Code snippets 47
Appendix B Terms and definitions 59
Bibliography 61

Introduction

Today, digital audio is everywhere. It is used in telephony, television and in devices like smartphones, game consoles, tablets and multimedia players. Moreover every other devices, is already a multimedia device, for example, all personal computers, telephones, cars, and, occasionaly, even the fridges have an audio reproduction capability. In other words, there's a lot of digital sound around us, sound that needs to be processed, stored, transformed, transmitted, compressed, coded and played. With so much digital sound around it is hard to underestimate the importance of digital audio processing techniques. However, it is, yet, too yearly to say that digital audio devices are replacing analogue audio devices. In particular, there are no digital speaker, one has to use analogue, phisical devices to reproduce the sounds, and these phisical devices have certain shortcoming that we are aiming to solve using the digital audio techniques.

The scope of this master thesis is to define a way to use the digital audio processing techniques as a way to solve and, eventualy, to eliminate the problems specific to the design and construction of analogue audio filters, which are quite many. The process of designing the audio filters relies heavily on the specifications of speakers and, the first problem that we encounter here, is that the speakers electrom-mechanical paramters (Thiele-Small parameters) and analogue filter components are change over time, are affected by the heat, and the specified parameters themselves might have low precision. This problem also appears during the speaker enclosure design, which is, as well, affecting the speaker characteristics, and requires changes in the filter design. Another problem encountered during the analogue filter design is that, while a two or three speaker system is relatively easily solved primarily because it was heaviliy studied – a system consisting of 4 and more speakers becomes a complex problem, which is one fo the reasons why such systems are not widespread.

The solutions that we are going to describe int this work are primarily based on two aspects: the application of Fourier Transform to the digital audio signals and, accordingly, the application of Z transform to the digital filters design process. The output of this work should be a program embodying the subjects discusses in this paper and the area of application of this program is as follows: a) the process of designing of analogue filters is facilitaed by designing them in a digital form first, and the the defined design is applied as a model to the analogue filter; b) the analogue filter is replaced altogether by the digital filter, thus eliminating a whole range of problems associating the filter.

1. The structure of digital audio and the generation of digital audio

1.1 Sound recording and digital audio structure

Sound, or audio, at its lowest level is the vibration of a physical substance, in our case – the air. The first known successful attempt of storing the audio goes back to the second half of 19 century, 1877, when Thomas Alva Edison patented the phonograph. This started the era of analogue sound recordings, which, as the name speaks, dealt with continuous analogue sounds being recorded, stored and reproduced. This era included the Acoustic period, when the sound recording and reproducing was performed exclusively by mechanical devices; Electrical period, started by the introduction of electrical microphones, loudspeakers and amplifiers and the Magnetic period, which began with the introduction of magnetic tape recording.

The era that is of our interest, the current, Digital era, started in 1975, with the adoption of Pulse-Code Modulation technique (PCM), which is currently a de-facto standard in digital audio recording[*]. The basic idea behind PCM consists in the following: a continuous analogue signal is processed portion by portion (this process is called discretization), for each portion of signal the signal level is measured and, then, rounded to the nearest value within a finite set of discrete values or levels (this process is called quantification). The units holding the information about one such portion of signal and its level are commonly referred to as samples. [1]

The method used to encode the sound on computer is a more specific version of PCM called Linear Pulse Code Modulation. The specifics of this method, is that the quantization levels are linearly uniform, as opposed to other PCM encoding techniques where the quantization levels may vary as a function of amplitude. Although, as was just mentioned, PCM is a more general term – it is commonly used to describe data encoded as LPCM. [2]

[*] The PCM technique was invented much earlier, but it wasn't used as a digital recording/reproducing method up until the 1970's when digital devices and personal computers began to spread

1.2 Dynamic range and the bit depth

As it was mentioned in previous rubric, the process of quantification is the casting of the signal level within a sample to a certain value or level included in a predefined set of values. The need in quantification is quite obvious, since the number of possible real signal levels is not a finite number, and signal levels themselves are, typically, irrational numbers. Accordingly, in order to express the signal level associated with each sample we need to be able to store it using a finite set of rational numbers. [3]

As one might guess, quantification is an irreversible process, i. e. it is not possible to reconstruct the *exact* original signal due to the fact that one quantified digital value represents a range of input signal levels, hence such a stored (and, eventually, reconstructed) signal represents an approximation of the original analogue signal. The difference between an original signal level and its quantized value is usually called *quantization error*. Obviously, to have a smaller quantization error and a digital signal level closer to the original – we need to have more discrete signal levels representing a smaller range of analogue signal levels. A schematic example of quantization process is presented in the figure 1.1 below.

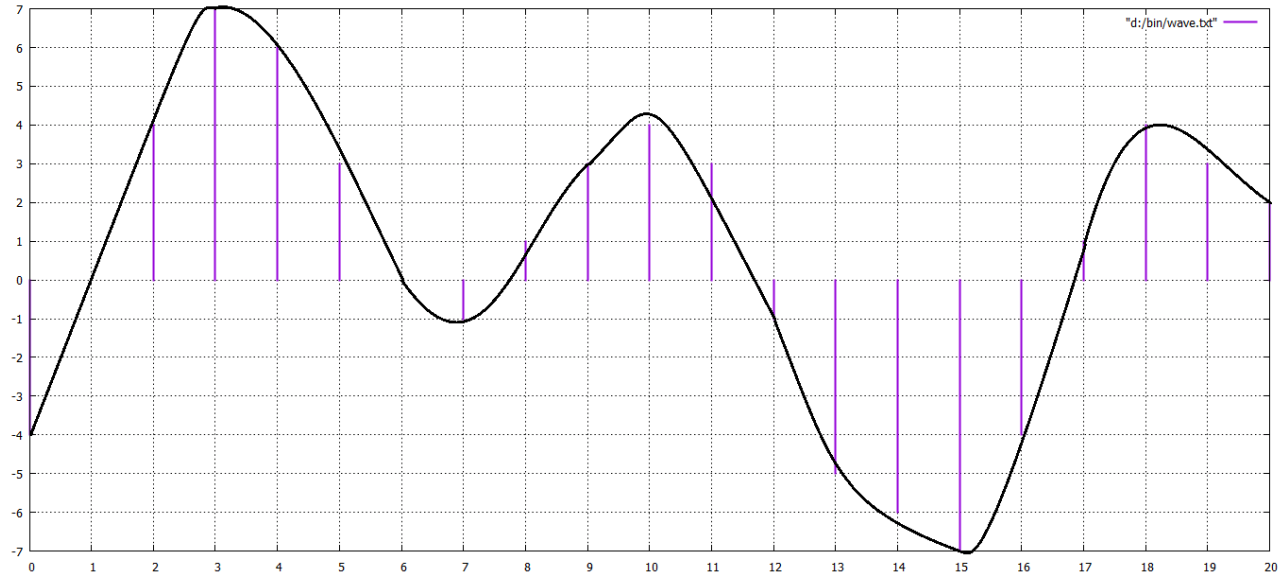

Figure 1.1 – An analogue signal (in red) encoded to 4-bit PCM digital samples (in blue); the bit depth is 4, so each sample's amplitude is one of 16 possible values.

The most common bit depths used today are 16 bit per sample, common to CD Audio, and the vast majority of computer sound cards, which represents a *short integer* (2 bytes) value, and 24 bits per sample (3 bytes) typical to DVD-Audio, Blue Ray Disc format and hi-end sound cards. Bit depths of 4-bit and 8-bit per

sample were used widely during the 80's era of videogames, and is normally associated with low quality audio, but it is almost never supported by modern software (e. g. Audacity, one of the most popular open source audio editors has 16 bits/sample as its lowest resolution option). 24 bits and 32 bit / sample (4 byte int or float) is a popular bit depth value used in audio programming and studio mastering processes, when engineers need to perform editing and resampling without losing the original sound quality, the resulting digital audio is often downsampled to 16 bits/sample. [4] While 24 bits dynamic range is the standard for DVD Audio and could, theoretically, support a signal to noise ratio of 144 dB (human ear's dynamic range is ~130 dB) – 32 and 64 bits depth is considered excessive and is not used outside of professional sampling / programming / editing environments: a 32 bits depth signal is 65,536 times the quality of 16 bits CD Audio, and in real life, a substantial amount of those bits would be just very low level noise.

It is also worth mentioning that the term *bit depth* is only meaningful in reference to a PCM digital signal. Non-PCM formats, such as lossy compression formats like MPEG, do not have associated bit depths. The table 1.1 below lists some of the most popular bit depths and associate signal-to-noise ratio (SNR).

Table 1.1 Signal-to-noise ratio and resolution of bit depths

# bits	SNR	Possible integer values (per sample)	Base ten signed range (per sample)
4	24.08 dB	16	−8 to +7
8	48.16 dB	256	−128 to +127
11	66.22 dB	2048	−1024 to +1023
16	96.33 dB	65,536	−32,768 to +32,767
20	120.41 dB	1,048,576	−524,288 to +524,287
24	144.49 dB	16,777,216	−8,388,608 to +8,388,607
32	192.66 dB	4,294,967,296	−2,147,483,648 to +2,147,483,647
48	288.99 dB	281,474,976,710,656	−140,737,488,355,328 to +140,737,488,355,327
64	385.32 dB	18,446,744,073,709,551,616	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

1.3 Sampling rate

Another factor that defines the audio quality (more exactly – audio properties) is the sampling rate, also known as modulation frequency. According to *Nyquist-Shannon sampling theorem (a.k.a Nyquist–Shannon–Kotelnikov, Whittaker–Shannon–Kotelnikov, Whittaker–Nyquist–Kotelnikov–Shannon, and cardinal theorem of interpolation.)* the sampling rate which is sufficient to encode and successfully reconstruct a continuous function is twice the highest frequency carried by the function. [5] The highest frequency audible by human ear is 20 kHz, hence the standard sampling frequency of CD Audio and most of digital audio playback devices – 44100. For DVD Audio the standard sampling frequency is 48 kHz.

In terms of audio recording and reproducing – a sampling rate higher than 48 kHz would be excessive. Lower sampling rates are used, however, in telephony systems and radio transmission. In particular, the typical sampling rate on AM radio is 22.05 kHz (maximum reproducible frequency ~ 10 kHz), and for telephony systems the sampling rate of choice is 8 kHz (max ~ 3.6 kHz) which is enough for voice sampling (human voice frequency range is 300 Hz to 3400 Hz).

The usage of a sampling rate lower than Nyquist frequency is the origin of *aliasing* effect (see figure 1.2) and is causing data loss. A typical consequence of aliasing effects in audio is the loss of higher frequency sounds and, if we are talking about telephony (VoIP telephony in particular) a voice may sound lower, when a very low sampling rate is chosen.

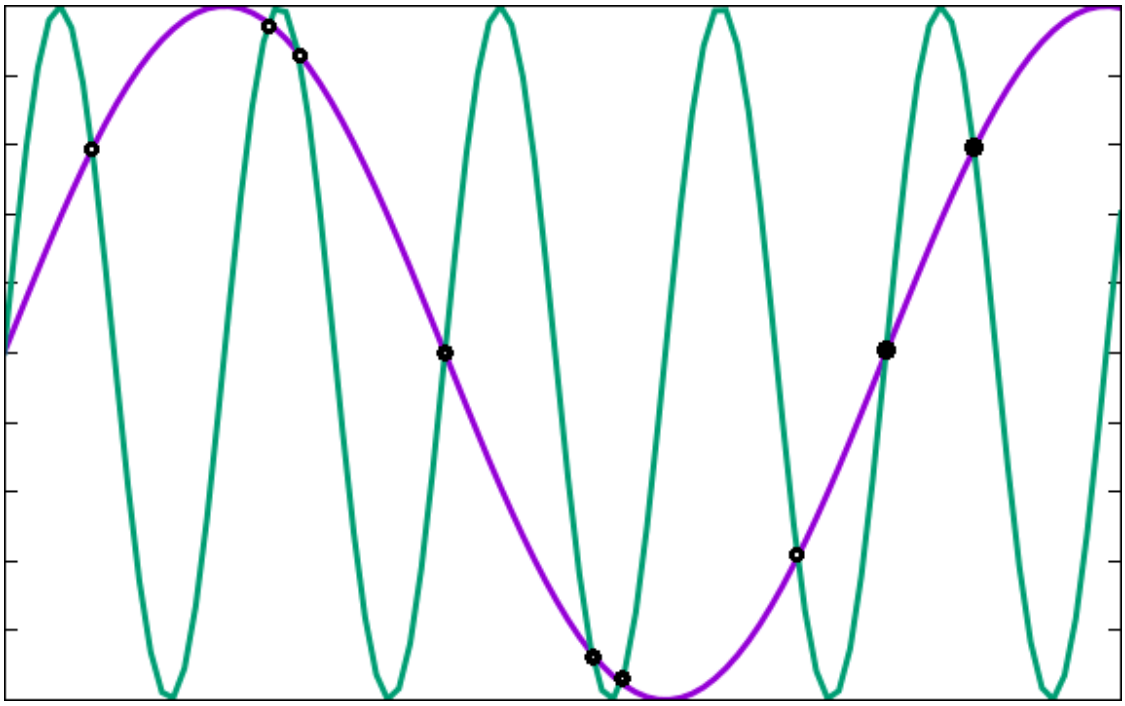

Figure 1.2 two sinusoidal signals that fit an identic set of samples.

1.4 Logarithmic scale. Amplitude vs loudness

The decibel measures sound pressure or electrical pressure (voltage) levels. It is a logarithmic unit that describes a ratio of two intensities, such as two different sound pressures, two different voltages, and so on. A *bel* (named after Alexander Graham Bell) is a base-ten logarithm of the ratio between two signals, the term *bel* is extremely rarely used though, e. g. it is more acceptable a notation of "hundreds of decibels" instead of "tens of bels".

There are 2 major reasons why the sound pressure is measured in decibels, which is a relative measure unit, instead of Pascals, or other fixed units. The first of it would be the extremely high dynamic range a human ear can distinguish. The highest and lowest such values measured in pascals would be 20 microPascals (.00002 Pa) which is an extremely small amount of atmospheric pressure and 200 Pascals on the other extreme, which is a huge range. To make it easier to work with such a range – it is easier to use a logarithmic scale. The other reason behind using a logarithmic reference value is the fact that the human ear itself perceives the sound pressure logarithmically, hence using a logarithmic scale accords more precisely or more naturally to the way human ear works. In particular, a small change in sound pressure at the bottom of logarithmic scale may represent a large change in signal level, while a relatively large change change at the top of logarithmic scale may represent a small linear change.[4][6]

Note that in digital systems, the highest signal level is 0dB, for example, a signal that reaches 50% of the maximum level at any point would reach −6 dBFS at that point, 6 dB below full scale.[4]

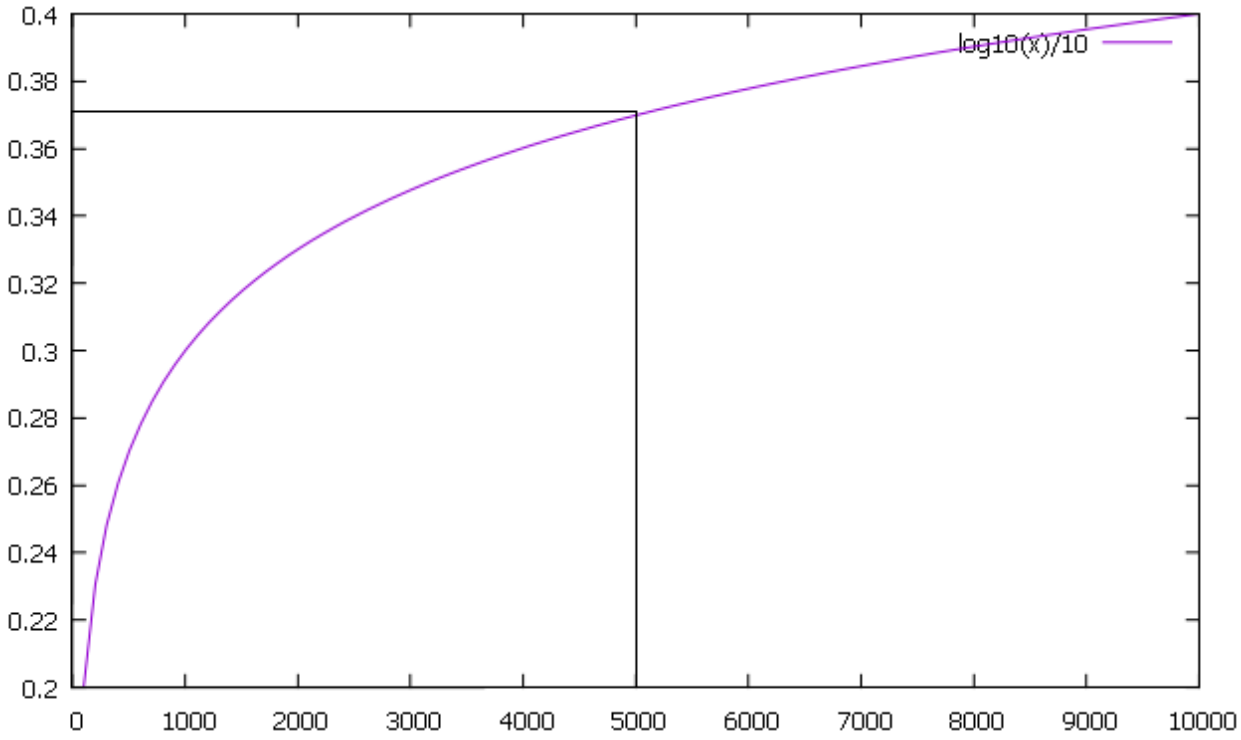

Figure 1.3 – Decibel scale log(x) for x between 0 – 10000.

1.5 Waveform generation

Some of the most fundamental procedures in audio programming is the generation of waveforms. Some of the most used waveforms in (not just in audio programming but in sound engineering in general) are:

Sine wave;

Square wave;

Triangle wave;

Upward/downward saw wave;

Examples of these waveforms are presented in fig. 1.4 below:

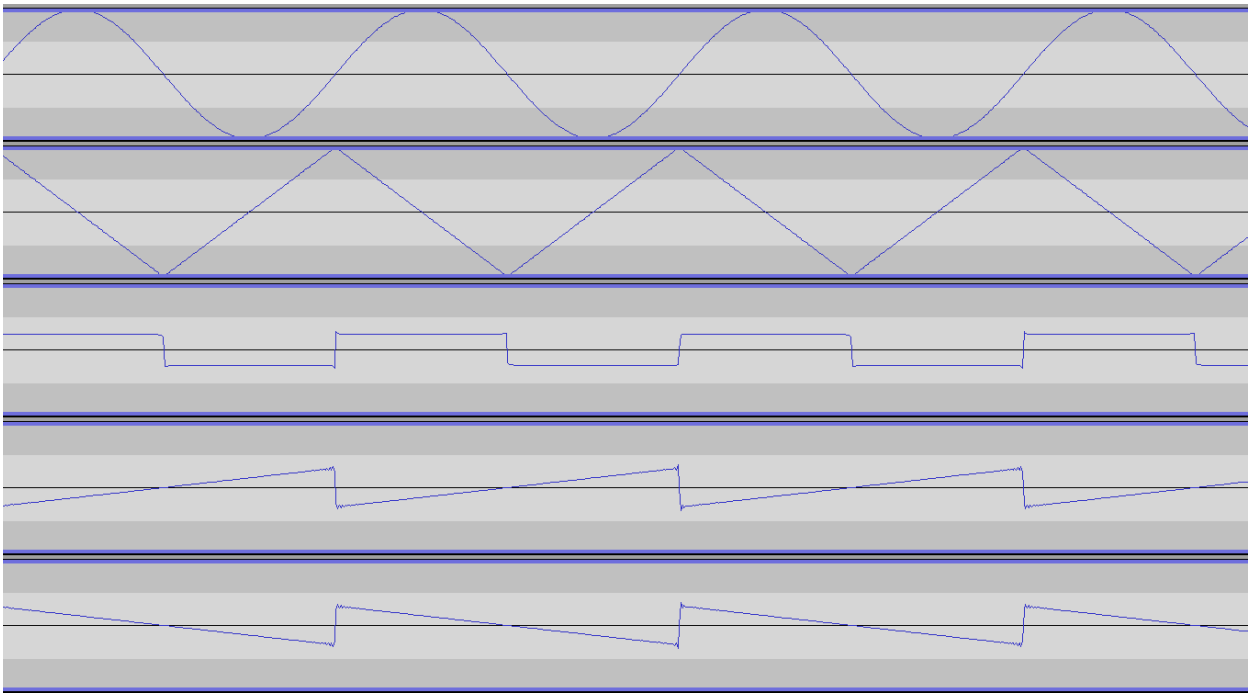

Figure 1.4 examples of non-sinusoidal waveforms in comparison with the sinusoid (at the top). The order of representation is: sine, triangle, square, upward (normal) saw, downward saw.

The *sine* wave or *sinusoid* is a mathematical curve that describes a smooth repetitive oscillation. There isn't much to say about it, except that it's the basis of all sound forms that we're going to examine. The code for generation of a sine wave is below. Snippet A3 in appendix A represents an example of sinusoidal waveform generation for sound synthesis purposes.

An ideal *square* wave is a non-sinusoidal periodic waveform in which the amplitude alternates at a steady frequency between fixed minimum and maximum values, with the same duration at minimum and maximum. In real life – an ideal square wave is impossible to create, and usually an approximation is used which consists of a summation of odd harmonics of the base wave. (see figure 1.5) Square waves are frequently used in digital switching circuits, mainly as timing references or "clock signals", and are generated by binary logic devices. See snippet A4 for an example of code generation implementing formula 1.1.

$$\int_{k=1}^{\infty} \frac{1}{2k-1} f_{2k-1}(t) \qquad (1.1)$$

Simplified Fourier series formula for square wave generation [4][6]

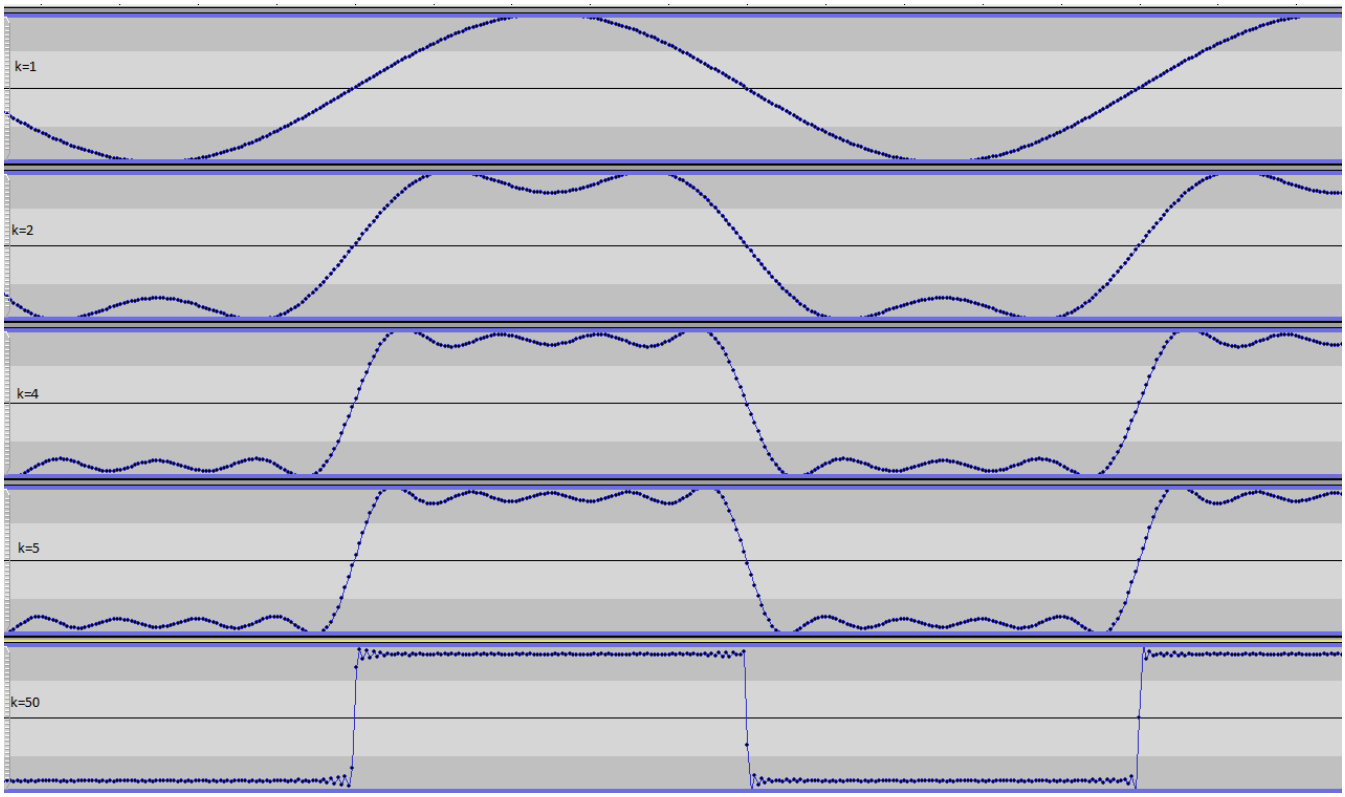

Figure 1.5 Square form generation where response is described by the integral above with the maximum value of k as indicated

A *triangle* wave (also called *triangular*) is a non-sinusoidal periodic waveform named for its triangular shape. Besides sound synthesis it is normally used as a reference waveform to test amplifiers and other analogue devices.

Like a square wave, the triangle wave contains only odd harmonics, demonstrating odd symmetry. However, the higher harmonics response is reduced much faster than in a square wave (proportional to the inverse square of the harmonic number). The implementation of triangle waveform generation can be found in code snippet A5.

$$\int_{k=1}^{\infty} \frac{1}{(2k-1)^2} f_{2k-1}(t) \tag{1.2}$$

Fourier series for triangle wave generation [4][6]

A *sawtooth* wave is a non-sinusoidal waveform characterized by periodic linear ascending and sharp drop which resembles the teeth of a saw (in the case of a *downward saw* waveform – it consists of linear descendings and sharp spikes). Sometimes, it is also named as an extreme case of a triangle wave described

above. The sawtooth wave is the form of the vertical and horizontal deviation signals used to generate a raster on CRT screens. See A6 for a generation code example.

Unlike the square or triangular wave – a sawtooth wave uses all harmonics:

$$\int_{k=1}^{\infty} \frac{-1}{k} f_k(t) \tag{1.3}$$

Fourier series for sawtooth waveform generation [4][6]

$$\int_{k=1}^{\infty} \frac{1}{k} f_k(t) \tag{1.4}$$

Fourier series for downward saw waveform generation [4][6]

1.6 Employment of oscillators for waveform generation

For generation of waveforms described above we are using the following approach: each type of waveform will be associated with an array of type double of custom *n*-size, storing a full waveform period. In the sine wave case – this will be just the *n* sample values for a sinusoid period. Sample code for each waveform is provided below. Note that the usage of a const double array gives us a significant performance boost, since we don't need to calculate each sample's value, which could be a very expensive operation, especially in the case of a higher number of harmonics. Code snippet A1 represents the initialization process described here.

A situation that occurs quite often in the process of audio signal generation and processing is the amplitude value overflow beyond the max amplitude of 1 (or -1 minimum), in the case of double or float valued samples. To handle such cases a normalization function is being used (code snippet A2).

To access the function value for the current phase – we are introducing a special oscillator structure, which holds the information about frequency, phase, sampling rate and the increment. A similar structure will be used to store the pointers to this oscillator data, a pointer to the array mentioned above and information regarding waveform information granularity (literally, a division of array size over sampling rate, which will be used to calculate the array step / increment as a function of frequency).

Finally, we are providing 2 tick functions to access the waveform information as the phase changes / cursor moves over the array, one of which provides a truncated representation of waveform (code snippet A7), and the second one – an interpolated representation. (code snippet A8) They are both performing, virtually, the same operation: read the current waveform value form the table and move the cursor to the next position. It has been decided to keep both since, while the interpolated tick function (linear interpolation, in

our case) should provide a smoother variant of the waveform, it performs poorly and is rarely used in real production code. Besides, the default settings used in this program are accurate enough to work for truncated lookup.

1.7 Common issues related to representation of digital audio (PCM)

Although, considered somewhat precise and accurate, digital audio has some bottlenecks that are worth mentioning as a part of this document.

*SHRT_MIN and SHRT_MAX***:** while 0 can be considered the same for all numeric formats, the minimum and maximum level that can be expressed in 16 bit are -32768 and 32767 accordingly. Whether the last negative level should be skipped (to preserve symmetry) or used (for space efficiency), is still a matter of debate. Note: although the minimum and maximum value for a float number are FLT_MAX and –FLT_MAX, float numbers drawback is that they are not actually able to represent the zero number, a float zero being an approximation (according to IEEE Standard 754 – Floating Point Numbers) hence -0 and +0 are distinct values even though they will return "true" if checked for equality.[4]

*Big endian vs Little endian***:** another thing to be taken in count when processing PCM data recorded on different platform in the endianness. Although, nowadays, most of the PCM audio formats are little endian – there are still big endian formats that are relatively widely used, and which should be supported. One of the most known Big Endian PCM audio formats is .AIFF (Audio Interchange File Format), associated primarily with Apple Macintosh computers[*] [4]

*Sample rate vs Frame rate***:** a frame, in terms of digital audio, is a set of samples related to the same time position that contain values for each channel, i. e. a frame for a mono audio source would consist of 1 sample, for a stereo audio source – 2 samples, for 5.1 audio – 6 samples, etc. Accordingly, panning a mono file to 2 channels, without data loss, would not only double it's size, but also its sampling rate. Thus, *sampling rate* term usually stands for *frame rate*.

Conversion from short to float, from int to double: different libraries and different sound streams are offering data in different format, the most common of which are short, int, float and double. The problem with converting from a floating point to a integer and vice versa does not simly reduce to casting, since float

[*] Note that Sound Blaster compatible soundcards have their roots in little endian IBM PC architecture, and requires PCM to be little endian, e. g., playing AIFF files on a PC requires conversion of audio stream from big to little endian

streams are commonly representing the stream in the -1.f to 1.f range, while the integer form is representing it in the SHRT_MIN/INT_MIN - SHRT_MAX/INT MAX**,** range which means that a more careful and complex mapping is required.

2. The Mathematics of Digital Audio Processing. Digital Audio Filters

In this chapter we will describe and provide some implementations of the Fourier Transform, which is the basis of Digital Signal processing in general and Digital Audio Processing in particular. The transform itself has been proposed as a method to describe some functions as an infinite sum of harmonics. In regards to Digital Signal/Audio processing, it is a method of decomposing a periodic signal into the frequencies that are present in the given signal. The variant of the transform that is to be used is the Discrete Fourier Transform, the term "Discrete" here meaning that we are dealing with a list of separate values of a given function of time, in other words – with samples. It can also be said the Fourier Transform converts (transforms) a function of time x(t) into a function of frequency X(f).Note, that, although the usual notation of a function is using the "*f*" letter, this could lead to a confusion, since this letter is also used to indicates the frequency, hence we will be using the letter "*x*" from here on.

2.1 Usage of Discrete Fourier Transform in Digital Audio Processing

As mentioned above, the Fourier transform is used to determine the frequency components of a signal. The frequency components graph is also known as spectra, and is widely used in signal analysis in general, and will be used in this particular master thesis as well.

Below we will present the formulas for DFT and Inverse DFT in trigonometric and exponential notation as well as some spectra examples. Note that in the figure 2.1 below, the sampling frequency is 1024 and the spectra pylons have a values of -0.5 and +0.5 and are located at points 0 and 1023, which corresponds to a frequency of -1 and 1, which corresponds to a sinusoid of frequency 1 and amplitude 1 (sin(-x)=-sin(x)).

Figure 2.2 is plotting the output of the same formula, but the input in this case is a square wave with a frequency of 5. The spectra pylons are thus located at 5, 15, 25 etc.

Appendix A9 contains an implementation of DFT which is customized for better display of output, in particular the division by number of samples (N) has been moved from IDFT sum to DFT, hence in its original form the spectra peaks are at -512 and 512 accordingly.

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-j\frac{2\pi kn}{N}} \qquad X_k = \sum_{n=0}^{N-1} x_n \left(cos\left(\frac{2\pi kn}{N}\right) - j * sin\left(\frac{2\pi kn}{N}\right)\right) \tag{2.1}$$

Discrete Fourier Transform in complex and trigonometric representation

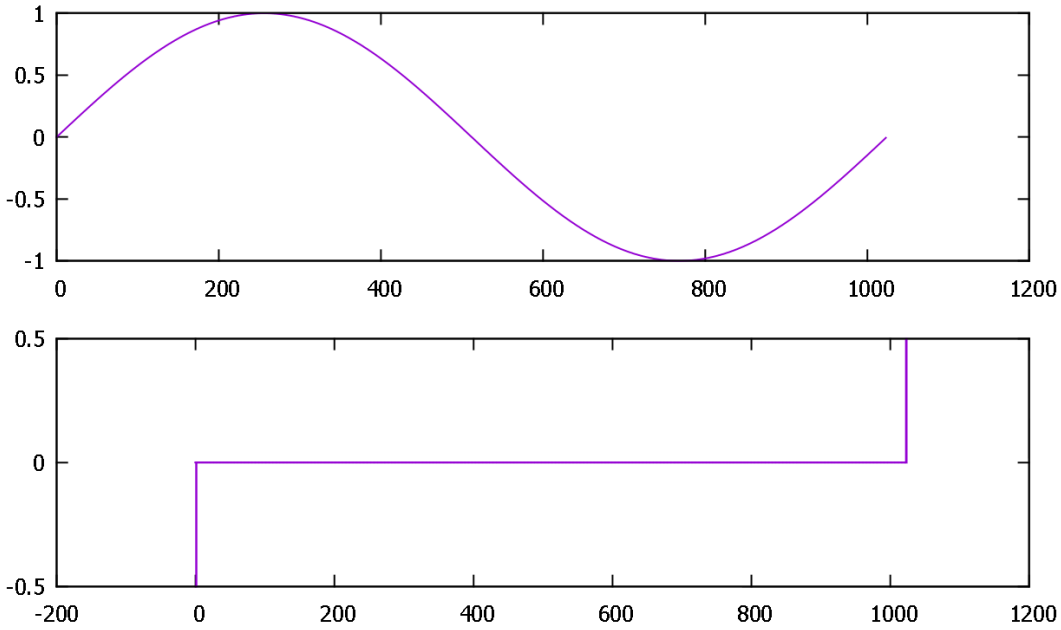

Figure 2.1 A sinusoid of frequency 1 and it's DFT

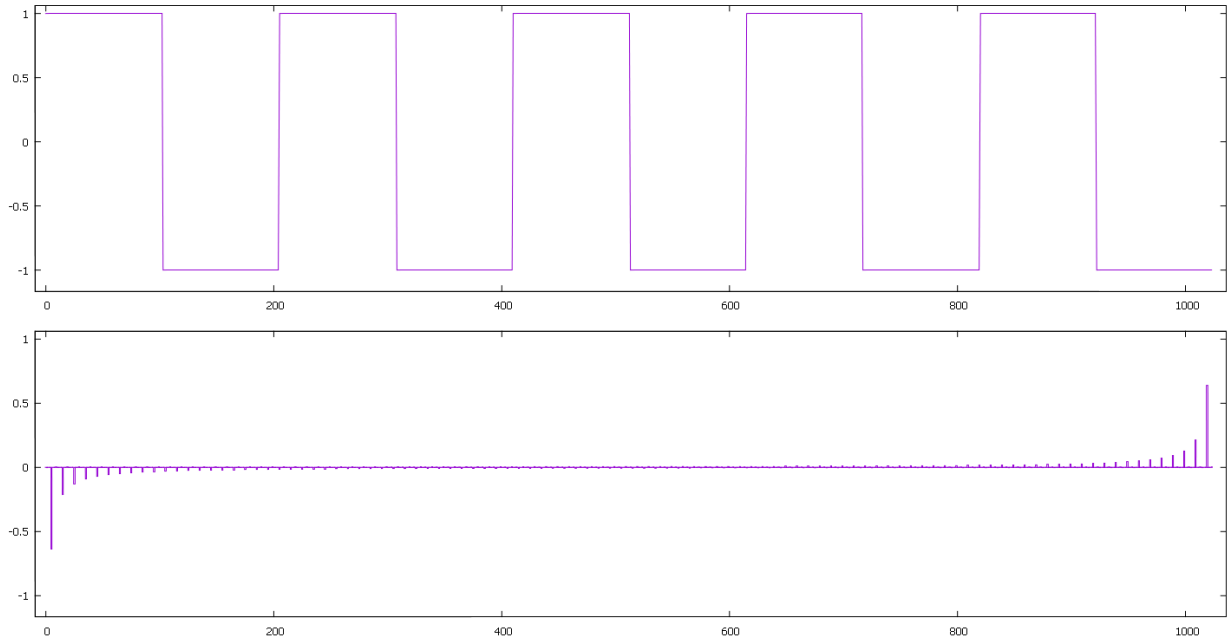

Figure 2.2 A square wave of frequency 5 and it's DFT

Below is a graphical representation of how the DFT actually works and detects the sinusoidal components. The image depicts a DFT taken on an 11 points sinusoid of frequency 1 and includes the components of the DFT sum for points 1 to 5 (first half). As you can see, the DFT components for

frequencies different from *f* and *-f* represent functions that are symmetric in the frequency domain, hence their integral is 0. Also, the closer frequency axis is to Nyquist frequency and, accordingly, the farthest to the given sinusoid frequency – the lower is the amplitude of the function. Obviously, the second half of the DFT sum functions will represent the same graphs with an inverted sign.

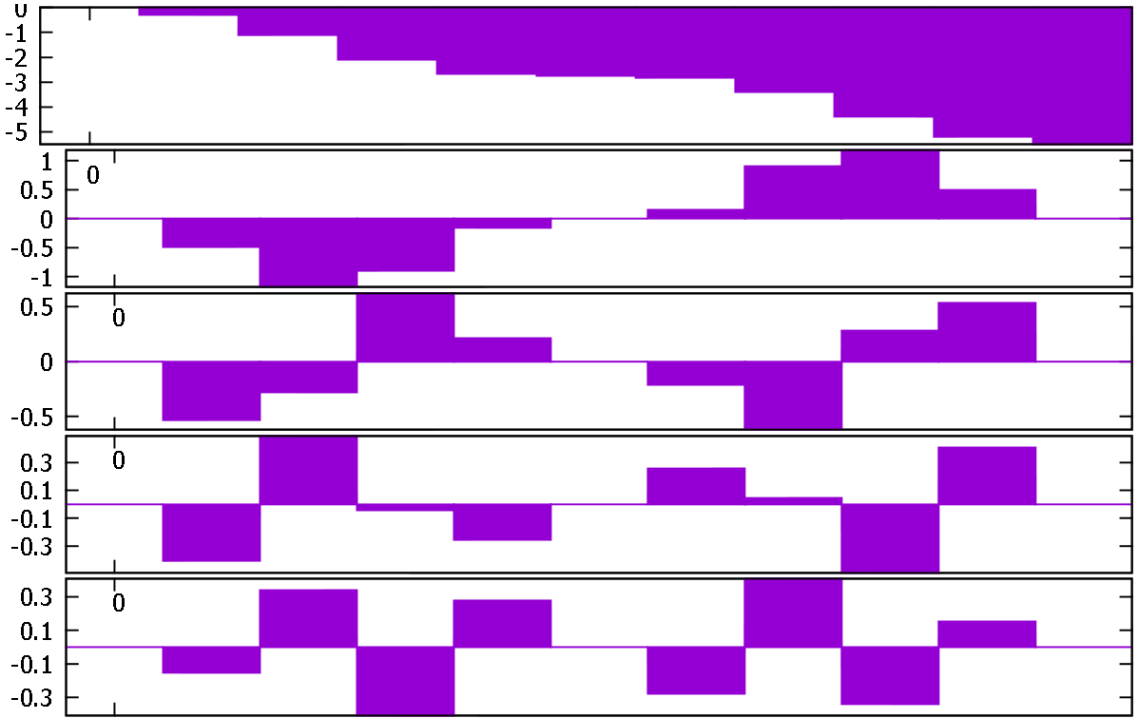

Fig 2.3 First half of DFT components for a 11 point sinusoid

The Inverse DFT (IDFT) has a very straightforward application – synthesis of a periodic signal from its spectra. Below you can see the formula for this operation, and the implementation can be found in appendix A10.

$$x_n = \frac{1}{N}\sum_{n=0}^{N-1} X_k \, e^{j\frac{2\pi kn}{N}} \qquad x_n = \frac{1}{N}\sum_{n=0}^{N-1} X_k \left(cos\left(\frac{2\pi kn}{N}\right) + j * sin\left(\frac{2\pi kn}{N}\right)\right) \qquad (2.2)$$

Inverse Discrete Fourier Transform in complex and trigonometric notation.

As one may guess even just by looking at the straightforward implementation of DFT and IDFT – the code is quite slow, using two trigonometric operations and several multiplications and divisions is quite slow (quadratic) and is not suitable for real-time signal processing. The importance of Fast Fourier Transform (FFT and IFFT) invented and reinvented several times over the course of centuries cannot, thus, be underestimated. The idea behind FFT is that it computes the transformations by splitting the input signal into smaller portions, the number of portions being a power of 2. As a result, it manages to reduce the complexity

of computing the DFT from quadratic $O(N^2)$, which arises if one simply applies the definition of DFT, to linearithmic $O(N \log(N))$, where N is the data size. There are several variants of FFT, the most common one being the Cooley–Tukey FFT algorithm (which is a reinvention of Gauss algorithm, formalized even before the Fourier theorem), while the fastest (having the lowest arithmetic operation count) FFT known at the time of writing is the Split-radix FFT algorithm. The preferred approach for this particular work is to use the FFTW library, hence no code snippets will be presented.

2.2 Analysis of digital audio employing the Short Time Fourier Transform and signal convolution

In a Hilbert (vector) space, the switch to frequency domain is regarded a change of base, which results in the following aspect of the DFT, important from a general DSP point of view: convolution of the 2 signals in the time domain equals their multiplication in the frequency domain, and vice versa: convolution of 2 signals in the frequency domain equals their multiplication in the time domain. Consequently, deconvolution in one domain results in division in another domain.

Some of the most important consequences of this aspect, with regards to digital audio processing, are the possibility of applying the impulse of a digital filter to an input signal, the facilitation of digital filter design in general, and the possibility of a windowed approach to signal analysis described below.

The DFT in its original form is providing limited to a specific point in time, or period of time. This is, in general, acceptable in a case of a strictly periodic signal, like solar cycles or tides periodicity, when only a base frequency needs to be identified. For efficient audio processing, though, we often need to observe the changes in the frequency domain, and to make our program react accordingly. In this case, the regular approach is to take a signal snapshot of a predefined size at a specific point in time, this process being called windowing. Short Time Fourier Transform (STFT) or, more correctly, discrete time DTFT, is the name given to the process of applying the DFT to a several samples or portions of signal. The formula is below:

$$STFT\{x[n]\}(m,\omega) \equiv X(m,\omega) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-j\omega t} \quad (2,3)$$

Short Time Fourier Transform**.** *w[m]* is the applied window [6]

There are several types of windows that can be applied, some of the most common being the Rectangular, Hamming and Blackman windows and their derivatives. Applying a window to a portion of signal usually means that the contents of the signal (the samples) are multiplied with coefficients according to the window shape. These types of windows are going to be discussed in details below.

2.2.1 Rectangular window

Rectangular window is the simplest, and at the same time – the least efficient type of window. This is the only window where the signal sample values are not multiplied with any coefficients (or are multiplied by 1). It's not very useful in signal analysis because of the simple fact that it creates discontinuities at the edges of the window, i. e. in the case of a sinusoid – the window may include the wave starting from, say, $\pi/8$ phase, and ending at, say, $15\pi/7$ (see figure 2.4 below). In a practical sense this means that the rectangular window usually generates spectral anomalies, like non zero values Such anomalies are not causing any serious problems in only one case: when we are analyzing a signal consisting of sinusoids of comparable size, moreover, in such cases the frequency resolution of the rectangular window is the best. However, speaking of real life sounds, this situation is virtually non-occurring, so whenever we are dealing with sinusoids of different amplitude – the spectra produced by a rectangular window will literally be a mess. This property is called *low dynamic range*. [7]

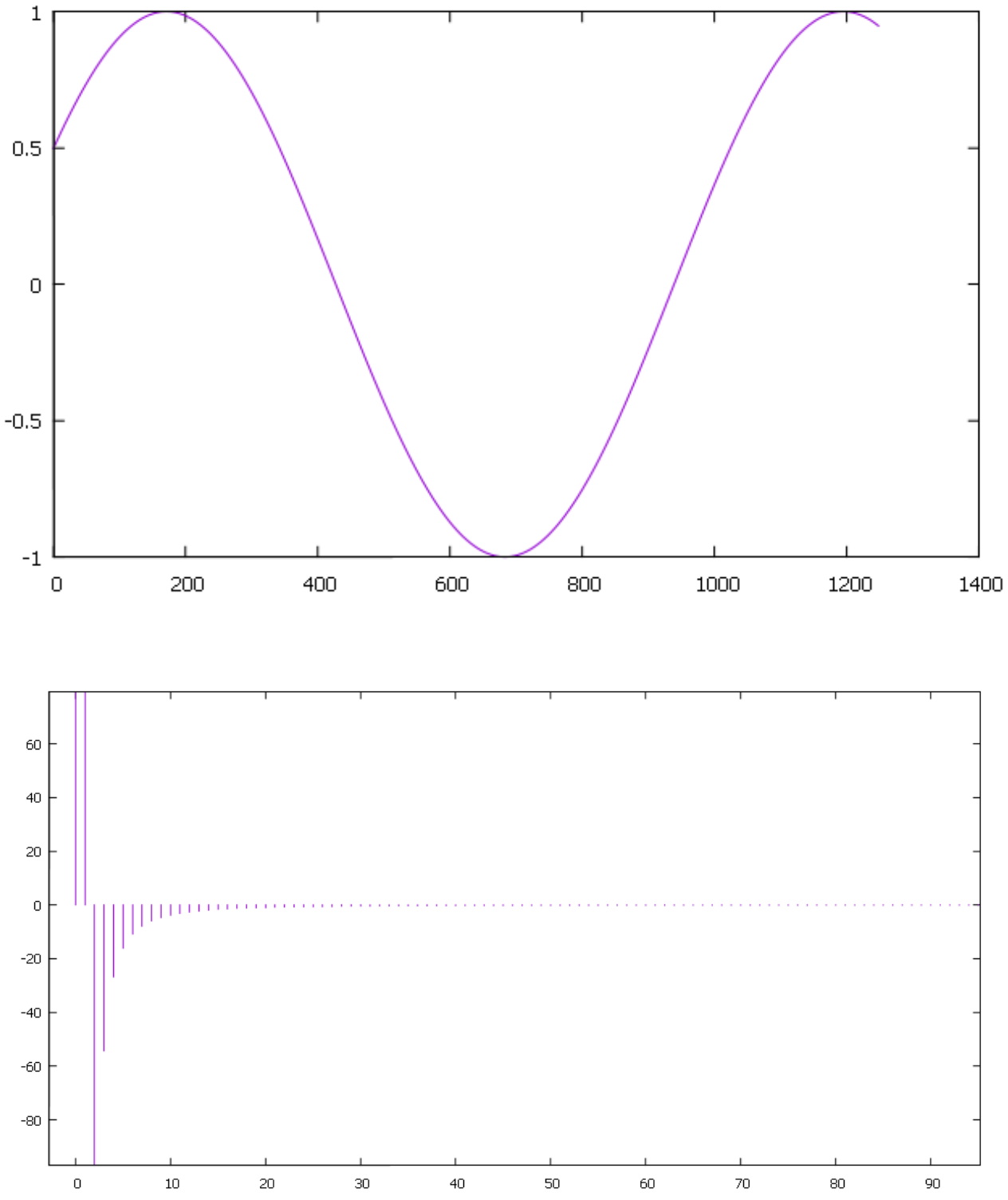

Figure 2.4 A discontinued sinusoid and its spectra (only a portion of the spectra is show)

2.2.2 Sinc function and the flat-top window

The sinc function is widely used in signal processing (in its truncated or adjusted form) and is the basis of 2 important application, the windowed-sinc filter, which is an approximation of an ideal low-pass filter, the non-windowed sinc filter being the ideal one. The shape of a sinc function (as well as of a filter kernel of a low-pass filter) is the same as of a flat-top window (see figure 2.5 below) and is sometimes (but not always) used to calculate the shape of such a window. The flat top window, as opposed to rectangular window, has a high dynamic range, which makes suitable for detecting amplitudes of sinusoidal components with a low measurement error, but this happens at the cost of frequency resolution. The flat-top window shape is commonly calculated as a sum of cosines as are other non-rectangular windows described here. The formulas for the sinc and the flat top window are below:

$$sinc(x) = \frac{\sin(\pi x)}{\pi x} \tag{2.4}$$

sinc function

$$w(n) = \sum_{k=0}^{K} a_k \cos\left(\frac{2\pi k n}{N}\right) \tag{2.5}$$

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right) + a_4 \cos\left(\frac{8\pi n}{N-1}\right)$$

a_0=0.21557895, a_1=0.41663158, a_2=0.277263158, a_3=0.083578947, a_4=0.006947368

Generalized flat-top function and the four term approximation [7][8]

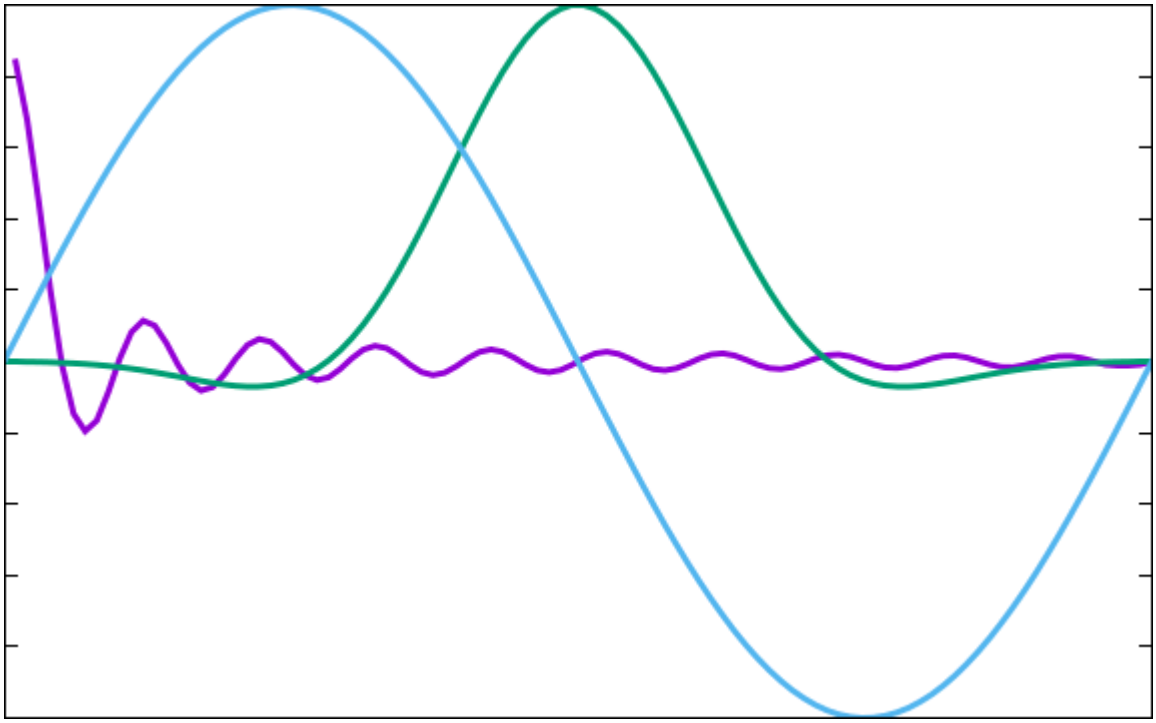

Figure 2.5 Sinc function, flat-top function and sinusoid for the same frequency and scale (N) in the 0-2π range.

2.2.3 Hamming and Hanning windows

As was mentioned in the previous section – rectangular window has a major flaw: the signal it traps is usually a discontinued signal which, in a general case, will produce certain undesirable artifact in the spectra. Of course, one would expect discontinuities from any, not only the rectangular window. The general solution to this is to apply to the signal a window having a shape that tends to 0 at its edges, which in the Hamming and Hann window case (and several other cases) means applying a raised half of cosine function. The cosine shape is built using three shifted and scaled *aliased-sinc* functions (a. k. a. *periodic sinc function*). The Hamming family of windows in general, provides a balance between frequency resolution an amplitude accuracy, providing an error level at around 1%, and are a standard in telephony(8 bit audio), but for higher quality audio there are better suited windows.[7]

The formulas 2.6 and 2.7 below describe the Hamming and Hann windows, and the sinc function.

$$w(n)=\alpha-(1-\alpha)\cos\left(2\pi\frac{n}{N-1}\right) \qquad 0\leq n<N \tag{2.6}$$

General formula for Hamming and Hanning windows [7][8]

Hann α=0.5

Hamming α=0.54

$$\text{asinc}(\omega)=\frac{\sin(\omega M/2)}{M\sin(\omega/2)} \tag{2.7}$$

Aliased-sinc function where ω is the angular frequency (ω=2πf) [7][8]

2.2.4 Blackman-Harris Window

The Blackman-Harris (BH) window family is a straightforward generalization of the Hamming family described above. Given the generalized Hamming family was constructed using a summation of three shifted and scaled aliased-sinc functions, the Blackman-Harris family is obtained by adding still more shifted sinc functions. The formula is exactly the same as the flat-top window formula, only the coefficients are different, below are the coefficients for a three term BH window. A 7 term window is able to provide a dynamic range enough for currently used precision levels (up to 32 bit quantization), but at the cost of frequency resolution. Note that reducing a BH window to 2 terms represents a Hamming window, and a 1 term is, obviously, a rectangular window.

a0 = 0.4243801, a1 = 0.4973406, a2 = 0.0782793 [7][8]

2.2.5 MLT/Sine window

The Modulated Lapped Transform uses the sine window defined in the formula 2.8 below:

$$w(n)=\sin\left[\left(n+\frac{1}{2}\right)\frac{\pi}{2M}\right] \tag{2.8}$$

MLT/Sine window where n=0,1,2,..,2M-1 [7][8]

The MLT/sine window is used in MPEG-1, Layer 3 (MP3 format), MPEG-2 AAC, and MPEG-4 compression, is considered to be asymptotically optimal, and has the smallest moment of inertia over all windows.[6] The spectra for this, and all other windows mentioned in this chapter is below, in figure 2.6

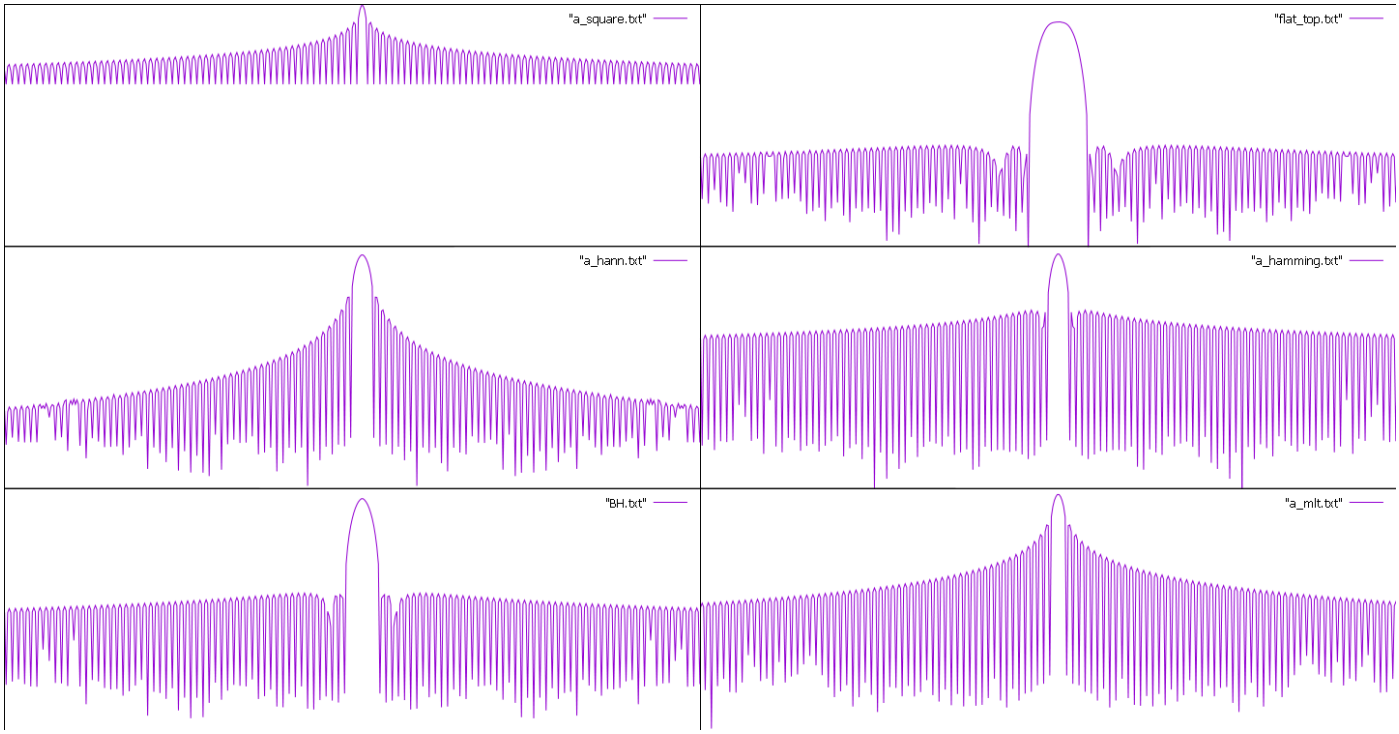

Figure 2.6 Magnitude of DTFT of windows normalized around 0 on logarithmic scale.

2.3 Audio filters and their parameters

Filters are used to do 2 things: to reduce the level of a frequency (or a range of frequencies) or to emphasize them. A filter is a linear time invariant process (i. e. its behavior does not change over time), which presumes that it does not add new components to the signal. What it does do, though, is change the relative level of the frequency components that make up a signal, i. e. it changes the spectrum. Naturally, the changes applied to the signal should follow a predefined, static (Linear Time Invariant) profile. The profile is referred to as the frequency response of the filter. Figure 2.7 provides an example of a low-pass filter response applied to a signal with an uniform spectrum.

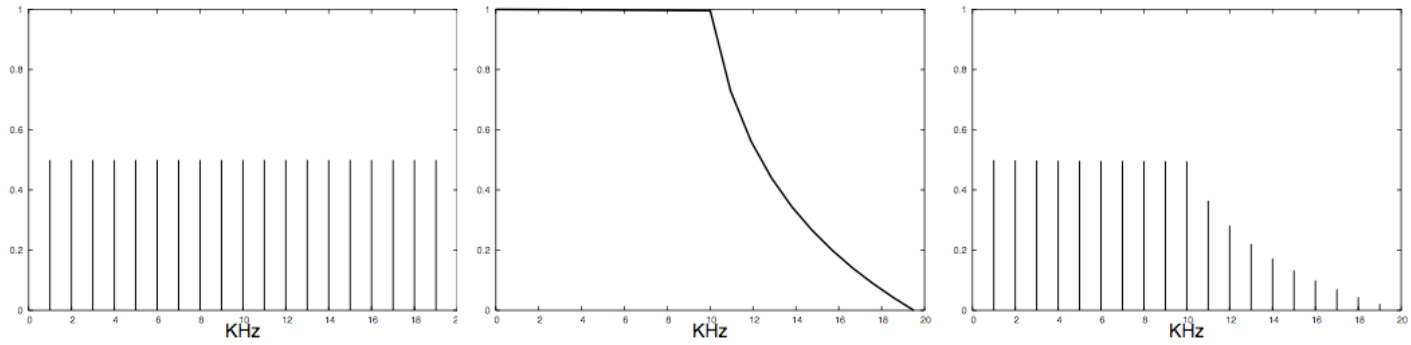

Figure 2.7 The filtering process in the frequency domain [4]

Other filter types are, high-pass, which attenuates the lower frequencies, leaving the upper frequencies unchanged, the band-pass filter, which attenuates everything out of a certain range, and the band-reject filter, which attenuates the frequency components within a certain rage. Figure 2.8 below provides examples for filter response similar to the one above.

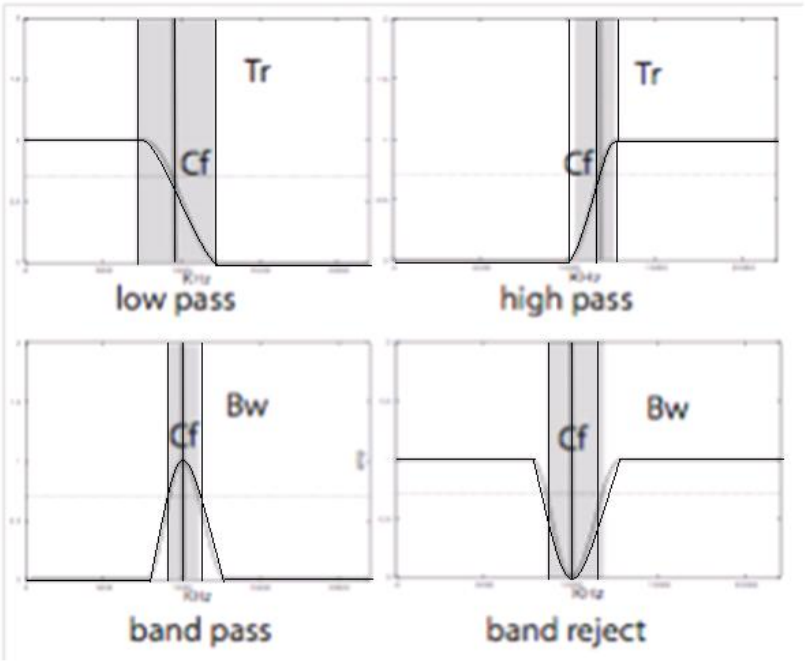

Figure 2.8 Parameters of the 4 basic filter types [4]

Tr – transition region;

Cf (High-pass, Low-pass) – Cutoff frequency

Cf (Band-pass, Band-reject) – Center frequency

The figure also shows that filter parameters are inter-related, offering a variety of ways to describe a filter. In each plot the light horizontal line indicates the -3dB level. This is the primary reference level for describing a filter. Its meaning differs slightly, depending on the filter type:

Low pass and high pass: The -3dB point defines the cut-off frequency – the frequency point at which the filter response has fallen by -3dB. The shaded area in these plots indicates the width of the transition region between the pass band (maximum response) and the stop band (minimum response).

Band pass and band reject: The central vertical line indicates the center frequency of the filter. The response is nominally symmetrical on either side of the center frequency. The shaded area in these plots indicates the bandwidth of the filter – again, bounded at the -3dB level above and below the center frequency.

A number of alternative terms may be employed to describe the same filter, depending on the application and the primary design objectives:

Stop-band attenuation: applies to low pass and high pass filters. In the ideal plots shown, the response transitions rapidly from the pass band to what appears to be zero response in the stop band – within which the signal is completely blocked. This would correspond to maximum stop band attenuation.

Slope: Even in a so-called brick-wall filter (low pass or high pass) the transition width in a realizable filter cannot be infinitely narrow. An alternative definition expresses this in terms of the slope of the transition – how rapidly the response falls off towards the stop band. In many filters employed for musical purposes, the slope is relatively shallow. This parameter is marked as Q (Q is usually understood to mean quality) which is defined as the ratio of cutoff frequency to bandwidth

$$Q = \frac{f_c}{B_w} \tag{2.9}$$

Slope (quality) of a filter.

Note that in a filter implemented in terms of a constant Q, the bandwidth (measured in Hz) increases linearly with frequency, so that regardless of the cutoff frequency, the bandwidth will cover the same pitch interval. Constant Q is an important property of many filters used for high-quality audio purposes.

2.4 Analogue filters

The basis of analog filters is 3 elements, resistors, capacitors and inductors. We will list their properties, just in case: a resistor changes only the amplitude of a signal, a capacitor reduces the amplitude of lower frequencies, while the higher frequencies are passing (almost) unchanged, and an inductor is acting the opposite way, it reduces the higher frequencies amplitude, while the lower ones are passing through. An analogue filter is, thus, a combination of the three in different forms. Figure 2.9 depicts some examples of the 4 basic filter types, implemented as 1st order Butterworth filters.

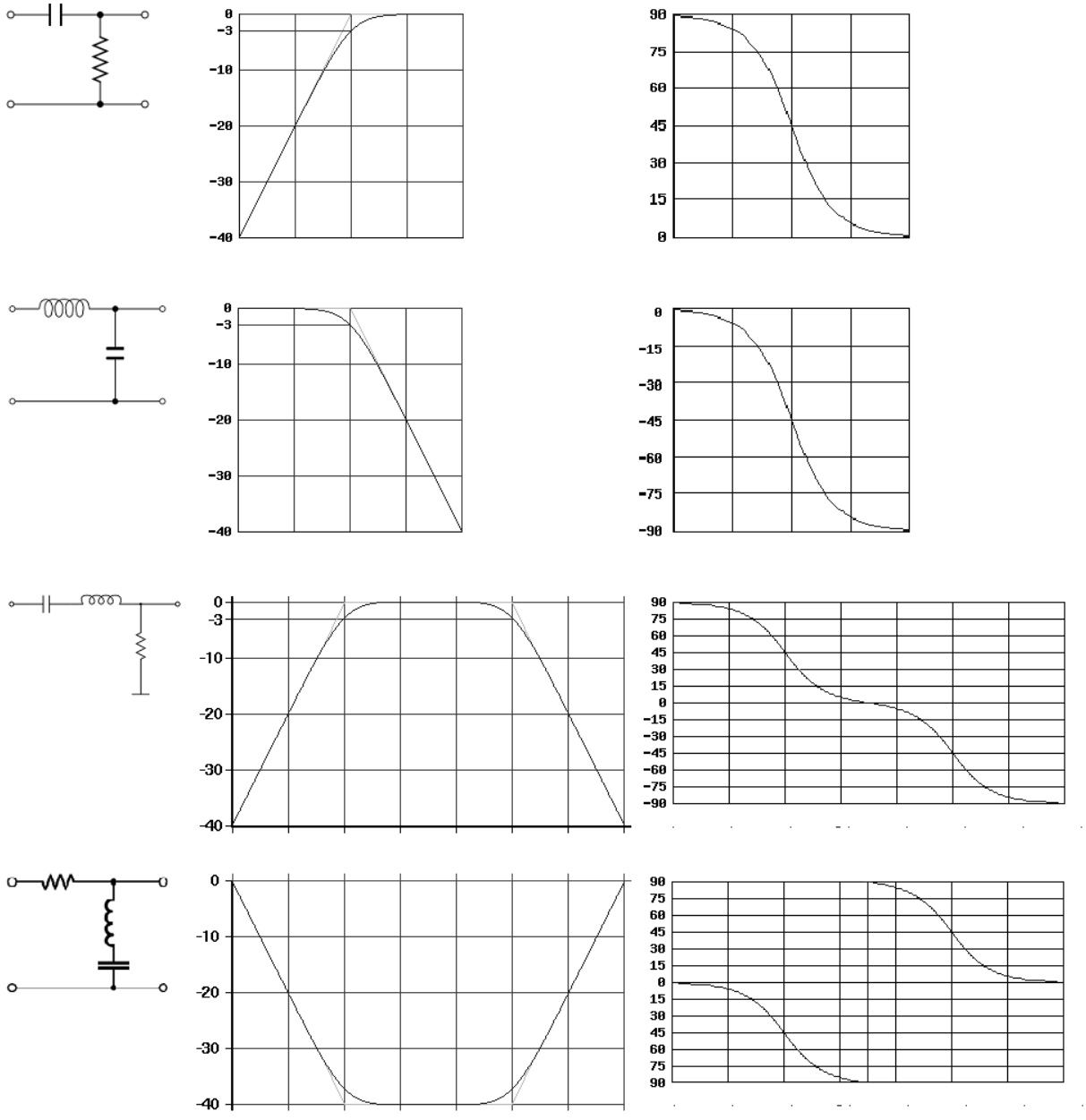

Figure 2.9 High-pass, low-pass, band-pass and band-reject analogue filters

2.5 The Infinite Impulse Response Filters

The simplest IIR digital filter used in DSP is the first order recursive filter. The order of a filter is then defined by the greatest delay distance, (whether of the input or the output), measured in samples. A filter that uses feedback is termed a recursive filter, and, like the general delay line with feedback, has an infinite impulse response. Figure 2.10 shows the schematic representation of such a filter, while the equation 2.10 describing it is below:

$$y[n] = ax[n] + by[n-1] \tag{2.10}$$

First order recursive filter, where x – input signal, y – output signal, n – current sample, a and b – coefficients.

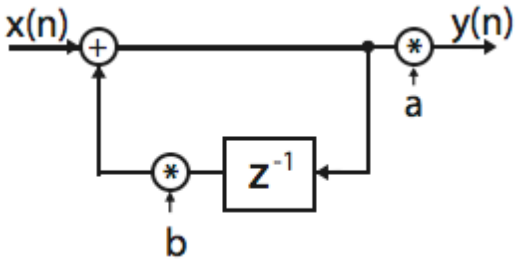

Figure 2.10 First order recursive filter flow diagram [4]

z^{-1} is called z-transform or unit delay operator and signifies a delay of one sample, and is related to the complex exponential as follows: $z=Ae^{j\varphi}$ where A denotes the magnitude and φ is the phase (angle).

2.6 Filter stability. Frequency and phase response.

The infinite impulse response filters are exclusive to the digital world, i. e. there are no IIR analogue filters. Obviously, if the gain of such a filter exceeds 1 – the output will increase exponentially, making the filter unstable and, hence, unusable. By contrast with the FIR filters, which only remove energy from a signal – an IIR filter is able to add it to a signal, which makes it quite easy to increase the output level above the input. To analyze this situation, we will be using the Z transform operation which is defined as follows:

$$X(z) = Z\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{2.11}$$

Z-transform X(z) of a discrete time signal x[n]

With the help of Z transform a filter's transfer function can be calculated as follows:

$$H(z) \triangleq \frac{Y(z)}{X(z)} \tag{2.12}$$

Transfer Function [7][8]

In the case of a simple recursive filter defined in 2.7 the calculus flow is:

y[n]=ax[n]+by[n-1]

Z{y[n]}=Z{ax[n]+by[n-1]}

According to the shift theorem for Z transform: $Z\{x[n-k]\}=z^{-k}X(z)$, hence the transform can be simplified as follows:

$Y(z)=aX(z)+bz^{-1}Y(z)$

$(1-bz^{-1})Y(z)=aX(z)$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a}{1-bz^{-1}}$$

Now, since the frequency response of a filter is complex valued function, it has an amplitude and a phase. The amplitude is equivalent to the transfer function magnitude |H(z)|. The phase response is calculated as for any complex number and is the arctangent of the imaginary part of H(z) divided by its real part. Applying this to a digital filter transfer function we will be identifying the phase of the input signal X(z), the phase of the output signal Y(z), and the resulting phase response will be the difference between these 2 signals.[7][8] For the example above, the phase response will be calculated as follows:

$$H(z) = \frac{a}{1 - bz^{-1}} = \frac{a}{1 - b\big(\cos(\omega) - jsin(\omega)\big)} = \frac{a}{(1 - b\cos(\omega)) + jb\sin(\omega)}$$

$$\angle H(z) = arctan(a) - (1 - b\cos(\omega)) + jb\sin(\omega) = \arctan(a) - \arctan\left(\frac{bsin(\omega)}{1 - bcos(\omega)}\right)$$

The phase response for such a filter is easily calculated, and as we can see in the figure 2.11 it could be made linear.

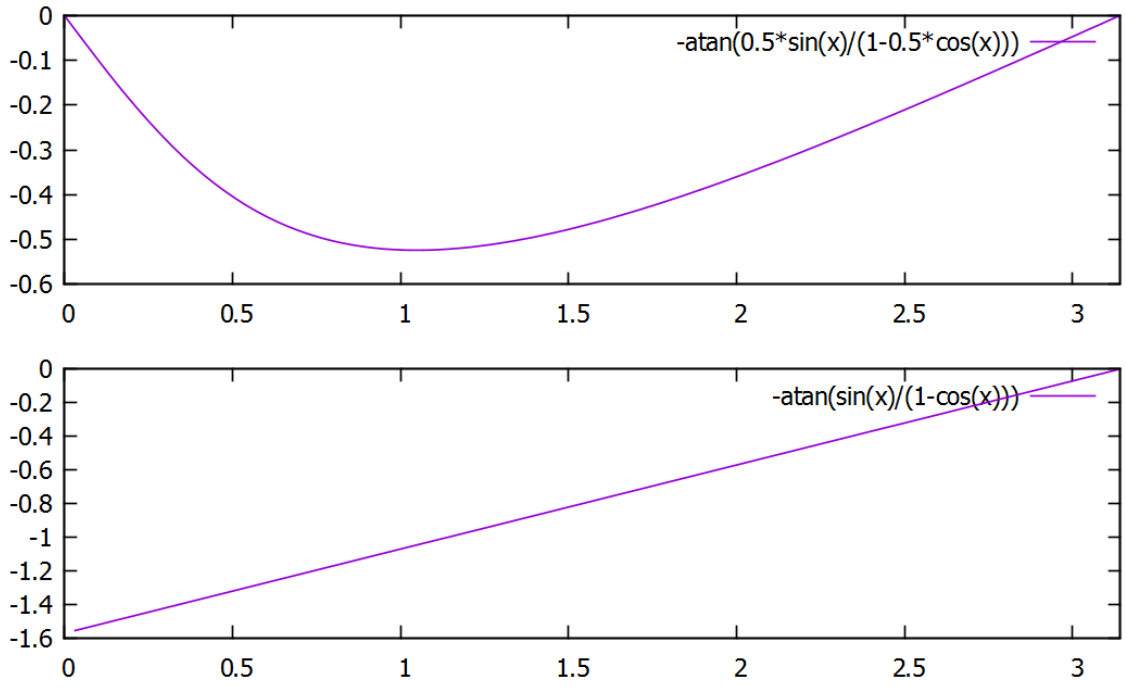

Figure 2.11 Phase response for a=1, b=0.5 and b=1

2.7 Finite Impulse Response (FIR) filters. Linear phase response

A typical FIR filter is a non-recursive digital filter which corresponds to a delay line without feedback. As the filter is non-recursive, its impulse response will be finite, hence the term Finite Impulse Response. A typical filter equation is as in the formula 2.13 below.

$$y[n] = a0x[n] + a1x[n-1] + a2x[n-2] \quad (2.13)$$

$$H(z)=a0+a1z^{-1}+a2z^{-2}$$

Equation and transfer function of a second order non-recursive filter.

In general, all non-recursive filters, can be designed such that the phase response will be flat, but this usually means that the Q of such a filter will not be very high. [4]

Generally, a FIR filter defines its impulse response by its coefficients. When applying a filter to a signal, the signal is convolved with the impulse response of a filter, i. e. the impulse response is calculated for each incoming sample, and then they are added together. Obviously – direct convolution (a. k. a. slow convolution) is only efficient for a small number of coefficients. For a more or less complex filter – the usual approach is to apply FFT to the signal and perform the multiplication with the filter's spectra. [4] In a general case, the solution is to determine which operation consumes more processor time, direct convolution, or FFT-

Multiplication-IFFT, and as the filter complexity increases the FFT variant becomes a solution of choice, especially in the case of real-time signal processing. The sample code for direct convolution is available in appendix A10.

2.8 Zero pole analysis

An usual practice for analyzing the filter response is to use the complex (Z) plane. As mentioned above, for a IIR filter the gain could cause a problem. For $H(z)=a/(1-bz^{-1})$ the problem will arise when z=b which likely to happen at a certain frequency, at which the denominator becomes 0 and the transfer function – infinite. Such a point in the transfer function is called a pole. In the case of a first order recursive filter, such a point is located at 0 or Nyquist frequency. In terms of the complex (Z) plane, this means that a first-order pole is confined to the real axis. The generalized formula for second and higher order filters is:

$$H(z) = \frac{1}{(1+b0z^{-1})(1+b1z^{-2})} = \frac{z^3}{(z+b0)(z^2+b1)} \tag{2.14}$$

Generalized formula for filters of second and higher order [7][8]

Here we have a second order filter, with two possible poles when z becomes equal to either b0 or $j\sqrt{b1}$ (or simply $\sqrt{|b1|}$, if b1 has a negative sign). These values for z are the roots of the quadratic expression in the denominator. In practice, these poles need to form a complex conjugate pair, i.e. they should be symmetrically opposite each other either side of the real (horizontal) axis in the Z plane, so that the resulting signal (adding of the two complex numbers) is always real. The formula for calculating the poles is as follows:

$$p0_k = -b0^{\frac{1}{M_1}} e^{j2\pi\frac{k}{M_1}}, k = 0,2,\dots,M_1 - 1 \tag{2.15}$$

$$p1_k = -b1^{\frac{1}{M_2}} e^{j2\pi\frac{k}{M_2}}, k = 0,2,\dots,M_2 - 1$$

Calculation of filter poles. [7][8]

In the case of a second order filter described above, the poles will be located at –b0 and $-\sqrt{b1}$. Note that in the case when nominator also contains z operator polynomials – the formula doesn't change, but the according points are called zeros, for obvious reasons.

2.9 The Standard Second-Order Biquad Filter

The biquad filter combines two non-recursive coefficients with two recursive coefficients. A further coefficient is applied as a gain factor to the input sample, giving five coefficients in all, therefore the filter supports a maximum of two poles and two zeroes. This filter architecture is the most common one used on general purpose computers. The word “biquad” is short for “bi-quadratic”, which for a digital filter means that the transfer function is a ratio of two quadratic equations. The figure 2.12 depicts the schematic representation of a biquad filter in *direct form 1*, the standard equation and transfer function are in the formula 2.16 below.

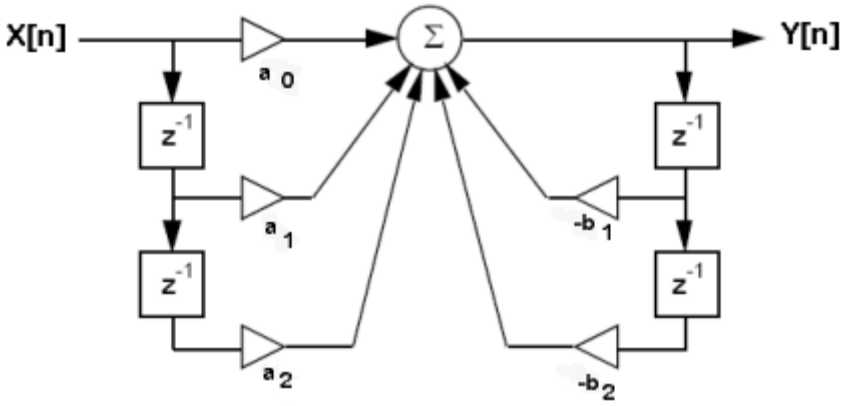

Figure 2.12 biquad filter flow diagram [4]

$$y[n] = a0x[n] + a1x[n-1] + a2x[n-2] – b1y[n-1] – b2y[n-2] \quad (2.16)$$

$$H(z) = \frac{a0+a1z^{-1}+a2z^{-2}}{1+b1z^{-1}+b2z^{-2}}$$

Biquad filter standard equation and transfer function [7][8]

The code snippets are available in appendix A11

3. Implementation of digital audio filters

3.1 Filtering the signal via spectral changing

One of the most straightforward methods of digital signal filtering is the modification of signal spectra via the convolution with the impulse of a filter. As a matter of fact, it shouldn't even be a filter impulse, it could be just a spectra of an arbitrary shape that we want to apply to our signal, however in this case certain problems might appear. In particular, diminishing or removing some of the spectral components is very likely to produce clipping in the reconstructed signal, which usually means that it needs to be corrected using dithering and noise shaping techniques. As part of this master thesis a few examples of such a filtering technique have been created, two of which we are going to describe here.

The first example is the convolution of a signal with the windowed sinc function. The sinc function has been chosen since, as mentioned earlier, it is considered to be an ideal low-pass filter. However, since this function does not produce a finite output, the windowed variant of the function is going to be applied. This implies some limitations, as, the narrower the window, the less accurate is the filter, which could even generate some artifacts. The code snippet is below, please note that in a real production code a sinc function table would not be calculated for each signal spectra, and that usually the sinc function is multiplied by a window function (e. g. by Hanning window), the square window being an actually rare choice.

```
for(i=0;i<fftsize;i++)
        sinc[i]=(sin(i)/i);
sinc[0]=1.f;
fft(sinc, sincspec,fftsize);
for(i=0;i<1;i++)
        specframe[i]*=sincspec[i];
ifft(specframe, sigframe, fftsize);
```

The figures 3.1 and 3.2 below depict the spectra of an input signal, and a sinc filtered output signal spectra.

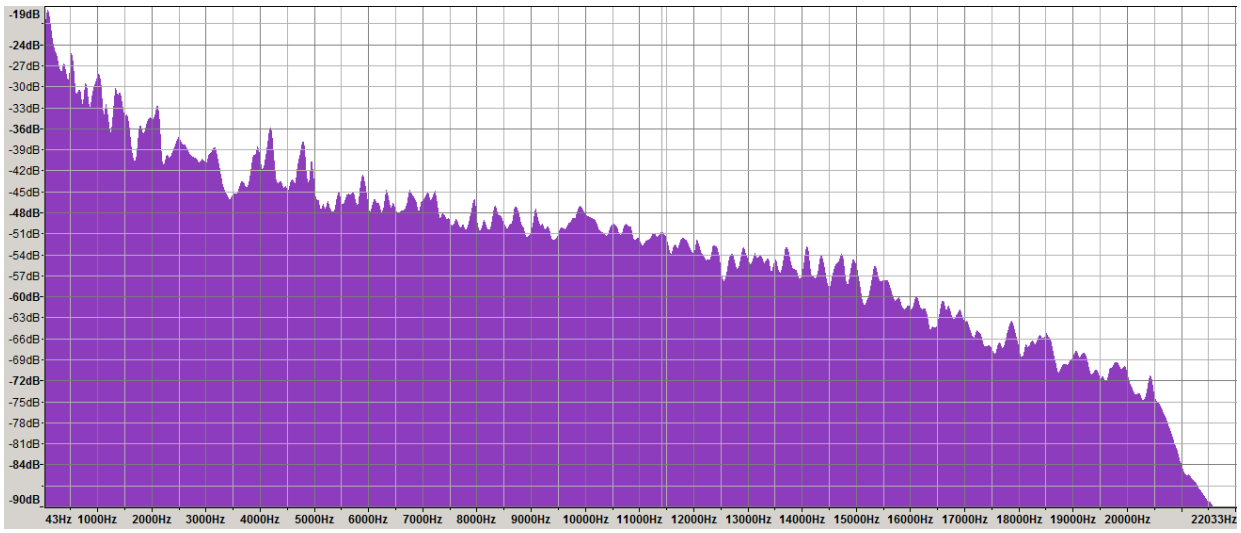

Figure 3.1 Input signal spectra

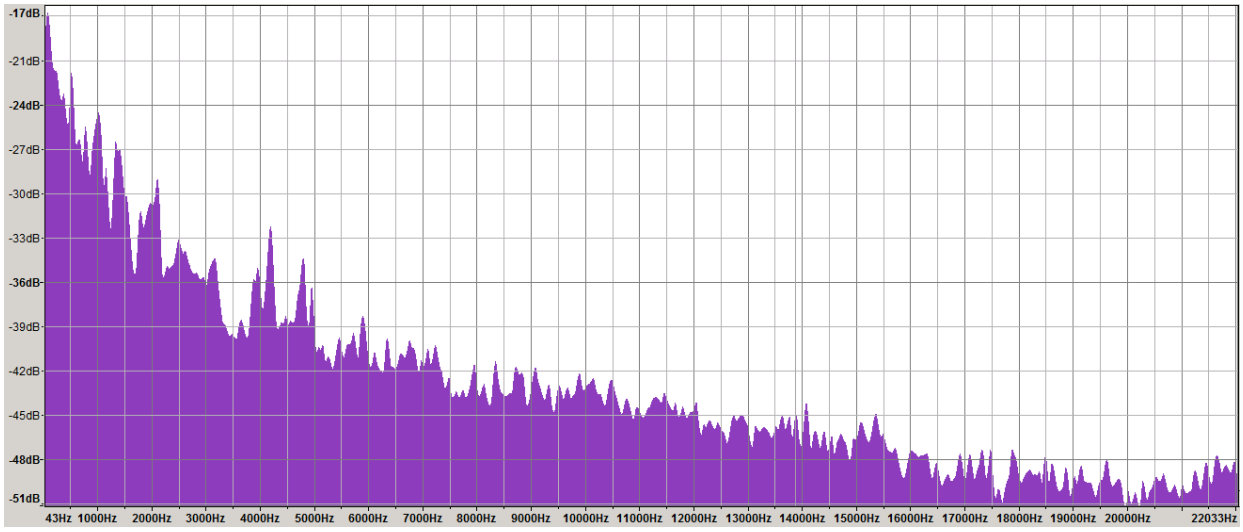

Figure 3.2 Output signal spectra with sinc filter applied

The second method can be called a brute force method and involves direct changes to certain portions of spectra. For example, if we want to lower the frequency components of the input signal by 100, we can do this simply as in the code below:

```
fft(sigframe, specframe, fftsize);
for(i=100;i>=0;i--)
specframe[i]/=100;
ifft(specframe, sigframe, fftsize);
```

The figure 3.3 shows the spectral diagram of the same signal before and 3.4 after processing the signal.

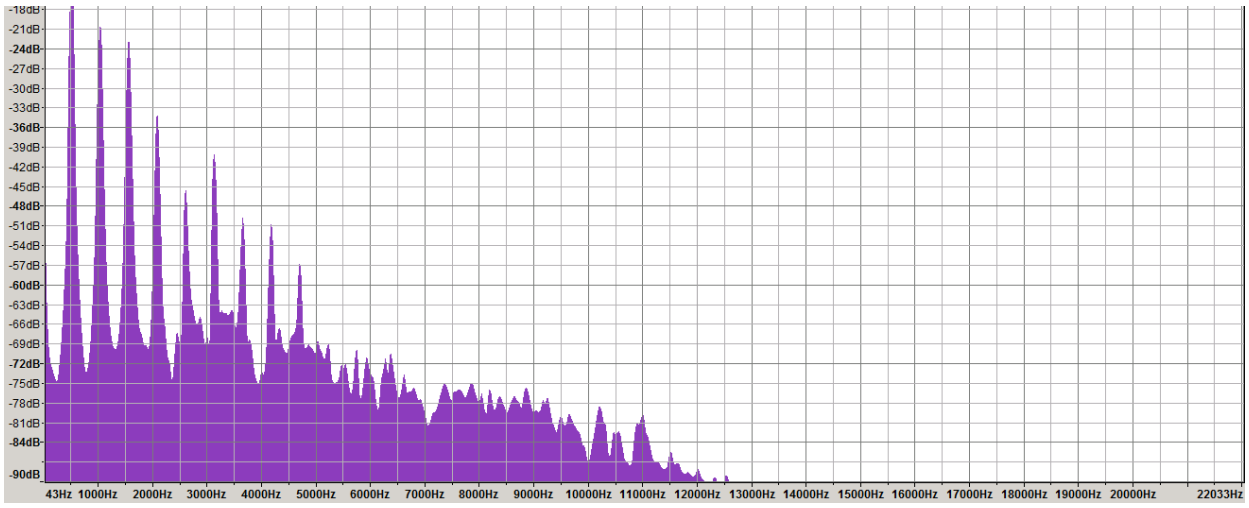

Figure 3.2 Input signal spectra

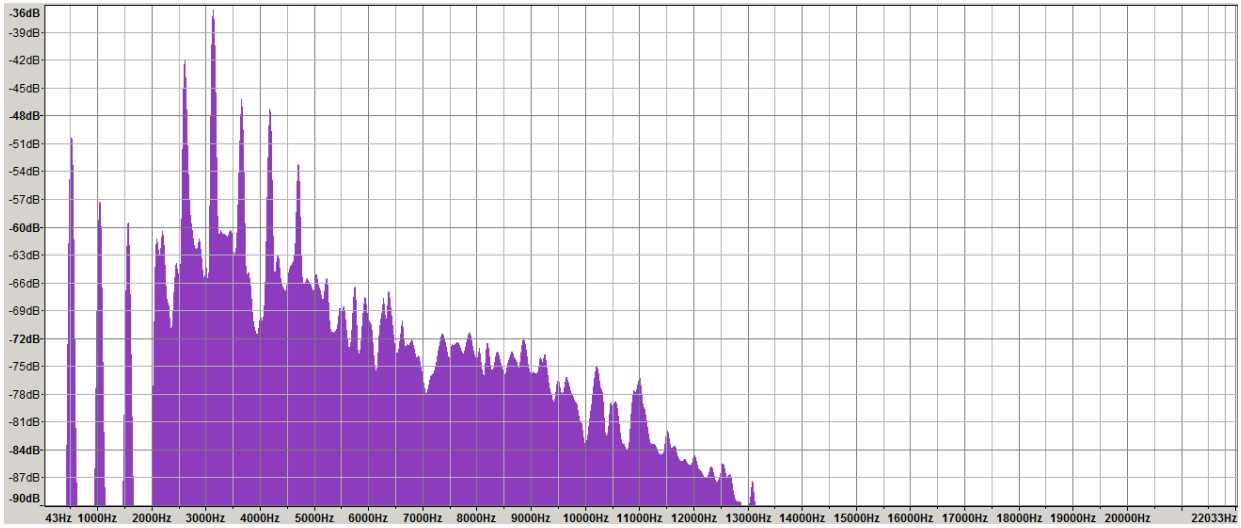

Figure 3.4 Output signal spectra

3.2 Direct filtering of a signal using digital filters

Speaking of digital audio, one of the most important aspects of it is the music reproduction. Now, while the music is slowly but firmly turning digital, the base aspects, such as filtering and redirection of audio has been well established in the analogue audio era, in particular, the standard audio filters used, e. g. for splitting the audio signal between a woofer and a tweeter speaker, are the Butterworth filters of different order. design of digital filters is commonly based on the analogue filters transfer function, which in a digital case can be simply reimplemented using input and output samples instead of capacitors and inductors, thus letting the engineer deal exclusively with the mathematics of the process, not worrying about analogue audio equipment materials quality and wearing. To avoid any possible misunderstanding, we need to mention here that the transfer function of an analogue and a digital filter of same type and order in its analogue and digital form don't match. The transfer function of a digital version of a filter is generated using a method called bilinear transform, which is not a subject of this paper.

In the application developed as part of master thesis, we have implemented the digital analogues of Butterworth filters (Low Pass, High Pass, Band Pass, and Band Reject) based on the formula first described by Charles Dodge and Thomas Jersey in the book "Computer Music". [4]. The implementation uses an intermediate variable C and needs the values of sampling frequency and cutoff frequency. For a band-pass and a band-reject filter, the algorithm also requires the desired bandwidth value and calculates an additional variable D. There is no much sense in listing here the calculation process for all the filters, hence only high-pass filter example is included:

$$C = \frac{1}{tan\left(\pi\frac{f_c}{f_s}\right)}$$

$a0 = 1/(1+C*\sqrt{2} + C^2)$

$a1=-2*a0$

$a2=a0$

$b1=2*(C^2-1)*a0$

$b2=a0*(1- C*\sqrt{2} + C^2)$

Having the calculated filter coefficients, the transfer function obtains a complete form, and we can now use it to calculate filters frequency response and phase response. A filters frequency response is defined by

the transfer function magnitude, i. e. $\sqrt{(Im^2(H(z)) + Re^2(H(z)))}$ and a filters phase response is defined by the arctangent of ratio of imaginary to real component, i. e. arctan(Im(H(z)) / Re(H(z))) [7]

Accordingly, to identify the amplitude and the phase of filter we just need to calculate the imaginary and real part of function transfer:

$Ximag = a1*\sin(\omega)+ a2*\sin(2*i\omega)$

$Xreal = a0 +a1*\cos(\omega) + coeffs.a2*\cos(2*\omega)$

$Yimag =b1*\sin(\omega)+ b2*\sin(2*\omega)$

$Yreal = 1 + b1*\cos(\omega) + b2*\cos(2* \omega)$

$\Theta(z) = \angle H(z) = \angle X(z) - \angle Y(z) = \arctan(Ximag/Xreal) - \arctan(Yimag/Yreal)$ [7]

$\Phi(z) = \sqrt{(Ximag^2 + Xreal^2) / (Yimag^2 + Yreal^2)}$ [7]

These formulas can easily be implemented in a programming language of choice, thus leading us to possibility of obtaining the phase and frequency response in a real-time manner, which is implemented in the application.

Below you will see an example of a code for such a filter which is quite simple and resembles the biquad filter design, but it includes the coefficient calculations; it is a modified version of filter functions provided by Richard Dobson as supporting materials for the “Audio Programming Book”[4].

```
void bw_update_lp(BQCOEFFS* coeffs, double cutoff, double bandwidth, double srate)
{
        double C;
        C =  1.0 / tan(cutoff * M_PI/srate);
        coeffs->a0 = 1.0 / (1 + sqrt(2) * C + (C * C));
        coeffs->a1 = 2.0 * coeffs->a0;
        coeffs->a2 = coeffs->a0;
        coeffs->b1 = 2.0 * coeffs->a0 * (1.0 - (C*C));
        coeffs->b2 = coeffs->a0 * (1.0 - sqrt(2) * C + (C * C));
}

void bw_update_hp(BQCOEFFS* coeffs, double cutoff, double bandwidth,double srate)
{
        double C;
        C =  tan(cutoff * M_PI/srate);
        coeffs->a0 = 1.0 / (1 + sqrt(2) * C + (C * C));
        coeffs->a1 = -2.0 * coeffs->a0;
        coeffs->a2 = coeffs->a0;
        coeffs->b1 = 2.0 * coeffs->a0 * ((C*C) - 1.0);
        coeffs->b2 = coeffs->a0 * (1.0 - sqrt(2) * C + (C * C));
}

void bw_update_bp(BQCOEFFS* coeffs, double cutoff, double bandwidth,double srate)
{
        double C,D;

        C =  1.0 / tan(bandwidth * cutoff * M_PI/srate);
        D = 2.0 * cos(2*cutoff * M_PI/srate);
        coeffs->a0 = 1.0 / (1.0 + C);
        coeffs->a1 = 0.0;
        coeffs->a2 = - coeffs->a0;
        coeffs->b1 = - coeffs->a0 * C * D;
        coeffs->b2 = coeffs->a0 * (C - 1.0);
}


void bw_update_br(BQCOEFFS* coeffs, double cutoff, double bandwidth,double srate)
{
        double C,D;
        C =  tan(bandwidth * cutoff * M_PI/srate);
        D =  2.0 * cos(2*cutoff * M_PI/srate);
        coeffs->a0 = 1.0 / (1.0 + C);
        coeffs->a1 = -coeffs->a0 * D;;
        coeffs->a2 = coeffs->a0;
        coeffs->b1 = -coeffs->a0 * D;
        coeffs->b2 = coeffs->a0 * (1.0 - C);
}
```

Another widely used filter that must be included here is the all-pass filter, also known as a phase filter. A popular analogue filter design of this type is known under the name of Linkwitz-Riley, and its purpose is to correct the phase shift.

The filters design can be explained as follows: as the transfer function shows – the filter zeros may freely exist outside the unit circle, i. e. they can be higher than 1, which is exactly the case when we have an all-pass filter design. The filters zeros are set to be conjugated so that they exactly cancel each other. An example of an equation is below:

y[n] = 0.8 x[n] - 1.6x[n-1] + x[n-2] + 1.6 y[n-1] - 0.8y[n-2]

$$H(z) = \frac{0.8 - 1.6z^{-1} + z^{-2}}{1 - 1.6z^{-1} + 0.8z^{-2}}$$

Formula 3.11 Example of an all-pass filter design [4]

As its name suggests, this filter passes all frequencies equally – the frequency response is flat. The phase response, though, is far from flat, and is exactly what make this filter usable. The example is in figure 3.5 below.

Figure 3.5 All-pass filter phase response

This characteristic of all-pass filter is used to correct the phase shift occurring with regular filters, and can be used in combination with them.

3.3 Product implementation

The main application that has been created as part of this master thesis has the following features: real-time spectral display for input and output signal, real-time filter adjustment for each channel separately, automatic calculation and display of filters frequency and phase response, and is primarily meant to be used with multi-channel (i. e. more than two channels) soundcards. The implementation of the frequency and phase response is a direct transcription of the according formulas for transfer function:

```
void filtResponse(BUTTERWORTH * bwf, double* resp, double* phase){
        BQCOEFFS coeffs = get_bw_coefficients(bwf);
        int i;
        int sampleRate=1024;
        for(i=0;i<sampleRate;i++){
                double imagX, imagY, realX, realY;
                imagX=-1* (coeffs.a1*sin(i*M_PI/sampleRate) +
                        coeffs.a2*sin(2*i*M_PI/sampleRate));
                realX=(coeffs.a0 + coeffs.a1*cos(i*M_PI/sampleRate) +
                        coeffs.a2*cos(2*i*M_PI/sampleRate));
                imagY=-1 * (coeffs.b1*sin(i*M_PI/sampleRate) +
                        coeffs.b2*sin(2*i*M_PI/sampleRate));
                realY=(1 + coeffs.b1*cos(i*M_PI/sampleRate) +
                        coeffs.b2*cos(2*i*M_PI/sampleRate));
                phase[i] = atan2(imagX,realX) -atan2(imagY,realY);
                resp[i]=sqrt((imagX*imagX + realX*realX)/(imagY*imagY + realY*realY));
        }
}
```

The figure 3.6 below is a screenshot of the program with a high-pass filter applied to the left channel and a low-pass filter applied to the right channel.

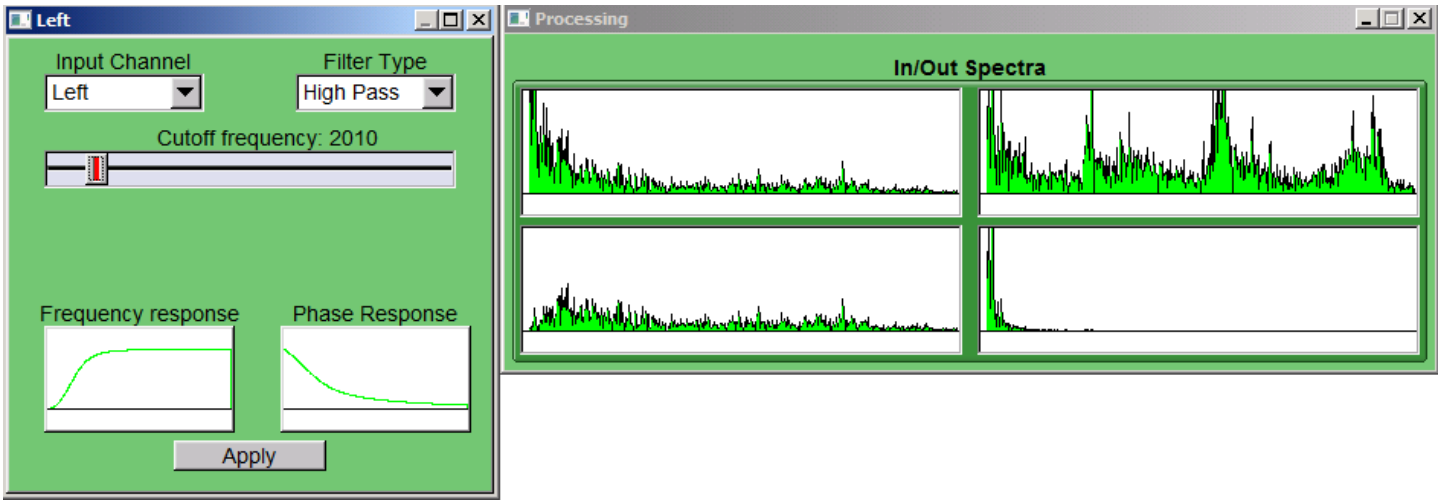

Figure 3.6 An image of application displaying input and output spectra for 2 channels and the filter configuration window.

Conclusions

This paperwork included a study of digital audio processing, the structure of digital audio, the most important algorithms of digital audio processing, and the mathematical principles that form the basis of digital sound processing in general and digital filters in particular. It described the methods related to the Fourier Transform and its variants which are being used in digital signal processing, the methods of spectral analysis and modification of signal, the direct digital (non-spectral) signal filters, the most important variants of digital filters, and the methods of calculation of a filter frequency response and phase response.

A key principle of this work is the analysis of the transfer function as the primary method of digital filter design, as well as provision of variants of implementation for the most popular filter types. Based on identified algorithms and principles – a software product has been created as part of this master thesis which provides a solution to the most common problems occurring in analogue audio filtering, in particular the problem of analogue filter design, the possibility of replacement of analogue filters with their digital counterparts providing a possibility of displaying of input and output signal spectra as a means to ease the analysis of filtered signal versus the original one. Another feature of the aforementioned product is the real-time calculation and display of a filter's frequency and phase response, and support for multichannel audio cards (up to 8 channels) and mono/stereo input, with the possibility of extension of number of both the input and output channels.

The product is a cross-platform application, and has been developed and tested using MinGW GCC compiler in the Win32 variant and QtCreator, CMake and GCC in the Linux variant as development environments. It includes libmad MP3 decoder, libsndfile, libogg, libvorbis and libflac libraries for audio files support. Signal spectra is calculated using the fftw library. For audio output it uses the PortAudio library, and FLTK with Fluid as front-end library in both Linux and Win32 variants. Some parts of the application are based on examples and tutorials included in "The Audio Programming Book" and are covered by GPL license.

Appendix A. Code snippets

A1 Initialization section[4]

```
typedef struct t_gtable
{
        double* table;
        unsigned long length;
} GTABLE;

GTABLE* new_gtable(unsigned long length){
   unsigned long i;
   GTABLE* gtable = NULL;
   if(length == 0)
      return NULL;
   gtable = (GTABLE*) malloc(sizeof(GTABLE));

   gtable->table = (double*) malloc((length+1)*sizeof(double));
   if(gtable->table == NULL){
      free(gtable);
      return NULL;
   }
   for(i=0;i<=length;i++)
      gtable->table[i]=.0;
   gtable->length = length;
   return gtable;
}

void gtable_free(GTABLE** gtable){
   if(gtable && *gtable && (*gtable)->table){
      free((*gtable)->table);
      free((*gtable));
      *gtable=NULL;
   }
}
```

A2 Normalization function [4]

```
static void normalize_gtable(GTABLE* gtable){
   double amp,maxamp=.0;
   unsigned long i=0;
   unsigned long length = gtable->length;
   for(i=0;i<length;i++){
      amp = fabs(gtable->table[i]);
      if(maxamp<amp)
         maxamp=amp;
   }
   maxamp = 1. / maxamp;
   for(i=0;i<length;i++)
      gtable->table[i] *= maxamp;
   gtable->table[i]=gtable->table[0];
}
```

A3 Sinusoidal waveform generation [4]

```
GTABLE* new_sine(unsigned long length){
   unsigned long i;
   double step;
   GTABLE* gtable = NULL;

   gtable = new_gtable(length);
   if(gtable==NULL)
      return NULL;


   step = 2 * M_PI / length;

   for(i=0;i<length;i++)
      gtable->table[i] = sin(step*i);

   gtable->table[i]=gtable->table[0];
   normalize_gtable(gtable);
   return gtable;
}
```

A4 Square waveform generation [4]

```
GTABLE* new_square(unsigned long length, unsigned long nharms){
   unsigned long i,j;
   double step,amp;
   double* table;
   unsigned long harmonic = 1;
   GTABLE* gtable = NULL;
   if(length == 0 || nharms == 0|| nharms >= length/2)
      return NULL;

   gtable = new_gtable(length);
   if(gtable==NULL)
      return NULL;
   table=gtable->table;
   step = (M_PI * 2) / length;
   for(i=0;i<nharms;i++){
      amp=1./ harmonic;
      for(j=0;j<length;j++)
         table[j] += amp * sin(step * harmonic * j);
      harmonic +=2;
   }
   normalize_gtable(gtable);
   return gtable;
}
```

A5 Triangle waveform generation [4]

```
GTABLE* new_triangle(unsigned long length, unsigned long nharms){
   unsigned long i,j;
   double step,amp;
   double* table;
   unsigned long harmonic = 1;
   GTABLE* gtable = NULL;
   if(length == 0 || nharms == 0|| nharms >= length/2)
      return NULL;
   gtable = new_gtable(length);
   if(gtable==NULL)
      return NULL;
   table=gtable->table;
   step = (M_PI * 2) / length;
   for(i=0;i<nharms;i++){
      amp=1./ (harmonic*harmonic);
      for(j=0;j<length;j++)
         table[j] += amp * cos(step * harmonic * j);
      harmonic +=2;
   }
   normalize_gtable(gtable);
   return gtable;
}
```

A6 Sawtooth waveform generation [4]

```
static GTABLE* new_saw(unsigned long length, unsigned long nharms, int up){
   unsigned long i,j;
   double step,amp,val=1.;
   double* table;
   unsigned long harmonic = 1;
   GTABLE* gtable = NULL;
   if(length == 0 || nharms == 0|| nharms >= length/2)
      return NULL;

   gtable = new_gtable(length);
   if(gtable==NULL)
      return NULL;

   table=gtable->table;

   step = (M_PI * 2) / length;
   if(up) val=-1.;
   for(i=0;i<nharms;i++){
      amp=val/ harmonic;
      for(j=0;j<length;j++)
         table[j] += amp * sin(step * harmonic * j);
      harmonic ++;
   }
   normalize_gtable(gtable);
   return gtable;
}

GTABLE* new_upsaw(unsigned long length, unsigned long nharms){
   return new_saw(length, nharms,1);
}

GTABLE* new_downsaw(unsigned long length, unsigned long nharms){
   return new_saw(length, nharms,0);
}
```

A7 Truncated table lookup[4]

```
double tabtick(OSCILT* p_osc, double freq){
   int index = (int) p_osc->osc.currentphase;
   double dtablen = p_osc->dtablen;
   double curphase = p_osc->osc.currentphase;
   double* table = p_osc->gtable->table;
   if(p_osc->osc.currentfrequency != freq){
      p_osc->osc.currentfrequency = freq;
      p_osc->osc.increment = p_osc->sizeovrsr * p_osc->osc.currentfrequency;
   }
   curphase += p_osc->osc.increment;
   while(curphase >= dtablen)
      curphase -= dtablen;
   while(curphase < .0)
      curphase += dtablen;
   p_osc->osc.currentphase = curphase;
   return table[index];
}
```

A8 Interpolated table lookup[4]

```
double interpolatedtabtick(OSCILT* p_osc, double freq){
   int base_index = (int) p_osc->osc.currentphase;
   unsigned long next_index = base_index+1;
   double frac,slope,val;
   double dtablen = p_osc->dtablen;
   double curphase = p_osc->osc.currentphase;
   double* table = p_osc->gtable->table;
   if(p_osc->osc.currentfrequency != freq){
      p_osc->osc.currentfrequency = freq;
      p_osc->osc.increment = p_osc->sizeovrsr * p_osc->osc.currentfrequency;
   }
   frac = curphase - base_index;
   val = table[base_index];
   slope = table[next_index] - val;
   val+=(frac*slope);
   curphase += p_osc->osc.increment;
   while(curphase >= dtablen)
      curphase -= dtablen;
   while(curphase < .0)
      curphase += dtablen;
   p_osc->osc.currentphase = curphase;
   return val;
}
```

A9 Discrete Fourier Transform implementation

```
void dft(float *in, float *outReal, float *outImag, int N){
   int i,n;
   for(i = 0; i < N; i++){
      outReal[i] = outImag[i] = .0f;
      for(n = 0; n < N; n++){
         outReal[i] += in[n] * cosf( i * n * M_PI * 2 / N );
         outImag[i] -= in[n] * sinf( i * n * M_PI * 2 / N );
      }
      outReal[i] /= N;
      outImag[i] /= N;
   }
}


void idft(float *in, float *outReal, float *outImag, int N){
   int i,n;
   for(i = 0; i < N; i++){
      in[i]= .0f;
      int n;
      for(n = 0; n < N; n++){
         in[i] += outReal[n] * cosf( i * n * M_PI * 2 / N )
                  - outImag[n] * sinf( i * n * M_PI / N );
      }
   }
}
```

A10 Direct (slow) convolution [4]

```
typedef struct f_conv
{
        unsigned long nm1;
        double* coeffs;
        double* past;
        long pindex;
} CONV;


CONV* new_conv(unsigned long ncoeffs, double coeffs[])
{
   CONV* conv = NULL;

   if(ncoeffs == 0)
        return NULL;
   conv = (CONV*) malloc(sizeof(CONV));
   if(conv == NULL)
        return NULL;
   conv->coeffs = (double*) malloc(ncoeffs * sizeof(double));
   if(conv->coeffs == NULL){
        free(conv);
        return NULL;
   }
   conv->past = (double*) malloc(ncoeffs * sizeof(double));
   if(conv->past == NULL){
        free(conv->coeffs);
        free(conv);
        return NULL;
   }
   memcpy(conv->coeffs,coeffs,ncoeffs * sizeof(double));
   memset(conv->past,0, ncoeffs * sizeof(double));
   conv->nm1 = ncoeffs - 1;
   conv->pindex = 0;
   return conv;
}
```

```
double conv_tick(CONV* conv,double input)
{
   unsigned long i,p;
   long pindex = conv->pindex;
   double output;
   double* coeffs = conv->coeffs;
   double* past = conv->past;

   past[pindex] = input;
   p = pindex;
   output = 0.0;
   for(i=0; i <= conv->nm1; i++){
         output += coeffs[i] * past[p++];
         if(p > conv->nm1)
         p = 0;
   }
   if(--pindex < 0)
      pindex = conv->nm1;
   conv->pindex = pindex;
   return output;
}

void conv_free(CONV* conv)
{
   if(conv){
      if(conv->past)
         free(conv->past);
      if(conv->coeffs)
         free(conv->coeffs);
      conv->past = conv->coeffs = NULL;
   }
}
```

A11 Standard biquad filter implementation [4]

```
typedef struct bq_coeffs
{
        double a0,a1,a2,b1,b2;
} BQCOEFFS;

typedef struct bq_filter {
   BQCOEFFS coeffs;
   double x1,x2,y1,y2;
} BQ_FILTER;


void biquad_init(BQ_FILTER* filter)
{
   filter->x1 = filter->x2 = filter->y1 = filter->y2 = 0.0;
   filter->coeffs.a0 = filter->coeffs.a1 = filter.coeffs.a2
      = filter.coeffs.b1 = filter.coeffs.b2 = 0.0;
}

void  biquad_set_coeffs(BQ_FILTER* filter, const BQCOEFFS* coeffs)
{
   filter->coeffs.a0 = coeffs->a0;
   filter->coeffs.a1 = coeffs->a1;
   filter->coeffs.a2 = coeffs->a2;
   filter->coeffs.b1 = coeffs->b1;
   filter->coeffs.b2 = coeffs->b2;
}


double biquad_tick(BQ_FILTER* filter,double input)
{
        double output;

        output =  filter->coeffs.a0 * input + filter->coeffs.a1 * filter->x1 + filter->coeffs.a2 * fil-ter-
>x2
                - filter->coeffs.b1 * filter->y1 - filter->coeffs.b2 * filter->y2;
        filter->y2 = filter->y1;
        filter->y1 = output;
        filter->x2 = filter->x1;
        filter->x1 = input;

        return output;
}
```

Appendix B Terms and definitions

Term	Definition
PCM	Pulse-code modulation is a method used to digitally represent sampled analog signals. It is the standard form of digital audio in computers, Compact Discs, digital telephony and other digital audio applications. In a PCM stream, the amplitude of the analog signal is sampled regularly at uniform intervals, and each sample is quantized to the nearest value within a range of digital steps.
Discretization	In mathematics, discretization concerns the process of transferring continuous functions, models, and equations into discrete counterparts. This process is usually carried out as a first step toward making them suitable for numerical evaluation and implementation on digital computers.
Sampling	In signal processing, sampling is the reduction of a continuous signal to a discrete signal. A common example is the conversion of a sound wave (a continuous signal) to a sequence of samples (a discrete-time signal).
Quantization	In mathematics and digital signal processing, is the process of mapping a large set of input values to a (countable) smaller set.
Bit depth	Bit depth is the number of bits of information in each sample, and it directly corresponds to the resolution of each sample.
Sampling rate	Sample rate is the number of samples of audio carried per second, measured in Hz or kHz. Bandwidth is the difference between the highest and lowest frequencies carried in an audio stream.
Decibel	The decibel (dB) is a logarithmic unit that expresses the ratio of two values of a physical quantity, often power or intensity. In acoustics it is used as a unit of sound pressure level.
Aliasing	In signal processing and related disciplines, aliasing is an effect that causes different signals to become indistinguishable (or aliases of one another) when sampled.

Normalization	Audio normalization is the application of a constant amount of gain to an audio recording to bring the average or peak amplitude to a target level (the norm).
DFT	Discrete Fourier Transform – decomposition of signal into harmonics or spectral components
IDFT	Inverse DFT – the inverse operation of recomposing the signal from harmonic components.
FFT	Fast Fourier Transform. Transformation of signal usually based on the Cooley-Tukey algorithm.
Transfer Function	A mathematical formula describing the ratio of the output of a system to the input of a system

Bibliography

1. "Linear Pulse Code Modulated Audio (LPCM)". Library of Congress. Retrieved 2015-09-29. http://www.digitalpreservation.gov/formats/fdd/fdd000011.shtml

2. “Pulse Code Modulation”. Library of congress. Retrieved 2015-09-30. http://www.digitalpreservation.gov/formats/fdd/fdd000016.shtml

3. Thompson, Dan (2005) – “Understanding Audio.” Berklee Press. ISBN 978-0-634-00959-4.

4. Richard Boulanger, Victor Lazzarini (2011) – “The Audio Programming Book”, MIT Press, ISBN: 9780262014465

5. C. E. Shannon, "Communication in the presence of noise", Proc. Institute of Radio Engineers, vol. 37, no.1, pp. 10–21, Jan. 1949. Reprint as classic paper in: Proc. IEEE, Vol. 86, No. 2, (Feb 1998). Online copy retrieved 2015-09-25 http://www.stanford.edu/class/ee104/shannonpaper.pdf

6. Julius O. Smith III (2007) – “Mathematics of the Discrete Fourier Transform (DFT) with audio applications. Second Edition” W3K Publishing, ISBN 978-0-9745607-4-8. Online Book Retrieved 2015-11-05: https://ccrma.stanford.edu/~jos/mdft/mdft.html

7. Julius O. Smith III (2007) – “Introduction to Digital Filters with Audio Applications” W3K Publishing, ISBN 978-0-9745607-1-7.

8. Julius O. Smith III (2011) – “Spectral Audio Signal Processing” W3K Publishing, ISBN 978-0974560731. Online Book Retrieved 2015-11-05: http://www.dsprelated.com/freebooks/sasp/