Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Exploring use of non-negative matrix factorization for lossy audio compression

Bc. Tomáš Drbota

Department of Theoretical Computer Science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

April 23, 2019

TODO

Acknowledgements

TODO

Declaration

Citation of this thesis

Drbota, Tomáš. *Exploring use of non-negative matrix factorization for lossy audio compression.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

TODO

Klíčová slova TODO

Abstract

Non-negative matrix factorization has been successfully applied in various scenarios, mostly for analyzing large chunks of data and finding patterns in them for later use. Due to the nature of NMF, it has also seen some use in the field of image compression.

The purpose of this thesis is to research possible uses of non-negative matrix factorization in the problem of audio compression. A reference audio encoder and decoder using NMF will be implemented and various experiments using this encoder will be conducted. The results will be measured and compared to existing audio compressing solutions.

Keywords lossy, audio, compression, processing, nmf, encoding

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's age of smartphones and other portable electronic devices capable of connecting to the internet, nearly everyone has access to this giant (and still growing) library of various media, including music and other audio. However, to transmit or store all of this data in its raw uncompressed form, a large amount of bandwidth and storage would be required.

.. need for compression ..

.. common methods of audio compression ..

.. mp3 opus ..

.. this work tries nmf ..

.. state of art ..

.. then design and implement ..

.. measure results ..

Chapter 2

Digital audio

Sound as we know it can be defined as a physical wave travelling through air or another means. [1] It can be measured as change in air pressure surrounding an object. Once we have this electrical representation of the wave, we can convert it back and consequently play using speakers.

In the real world, these sound waves are generally composed of many different kinds of waves, with differing frequencies and amplitudes. The human ear can tell the difference between high (whistling) and low frequencies (drums), and knowledge of this will be useful later when we are discussing audio encoding.

2.1 Notation

Please see Table 2.1 for the notation used throughout this thesis.

2.2 Important terms

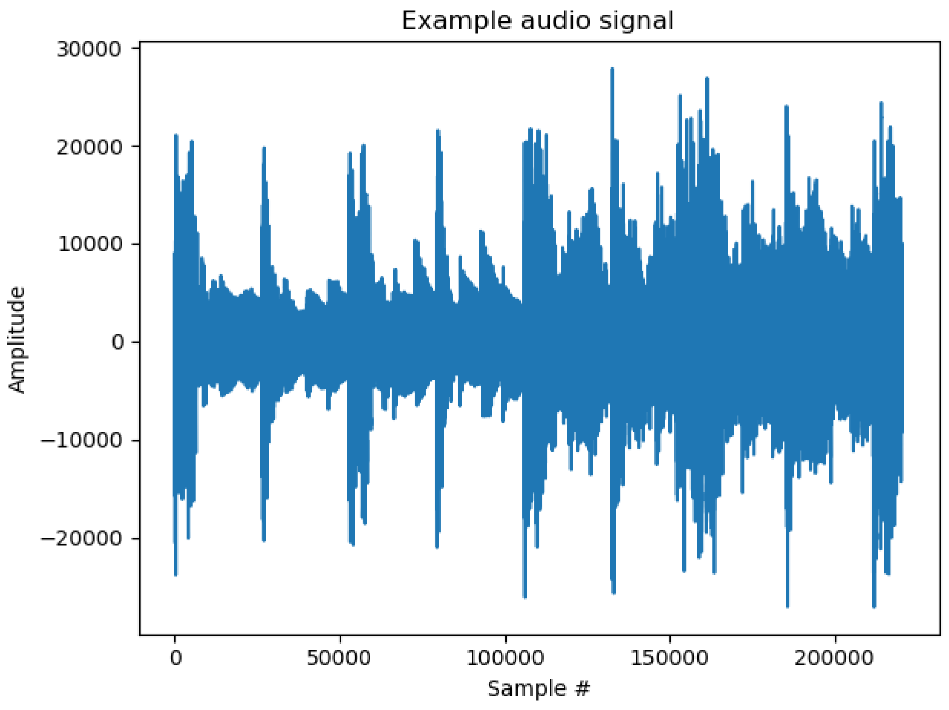
In this section several terms used throughout this thesis will be described.

2.2.1 Sampling

Sampling in this context refers to the way we convert an analogue signal into a digital one. When we use something like a microphone, what's happening is that it measures the pressure waves generated by the sound around it at regular intervals. The result of this operation is the amplitude of the wave, or the size of the vibration at that point in time.

To obtain an actual usable sound wave of various frequencies and amplitudes, we will need many samples - generally tens of thousands per second. The rate at which we collect these samples is called the *sampling rate*, denoted F_s.

Figure 2.1: An example of an audio signal represented in PCM form.



2.2.2 Nyquist frequency

It is proven that we can fully represent an arbitrary signal $x(t)$ by its samples x_n [2], but there are some conditions to this. If we don't sample a given signal enough times per second, there might be frequencies high enough that our sampler won't be able to record them correctly.

This is called the *Sampling Theorem* and the idea is that we can only accurately record all the frequencies in a signal if our sampling rate F_s is at least twice as large as the largest frequency contained in the signal F_{max}. [3]

In essence, a continuous signal can only be fully represented by its samples if:

$$F_s \geq 2F_{max} \tag{2.1}$$

Looking at it from another angle, it also means that given a sampling rate F_s, the highest frequency we can record is equal to $\frac{F_s}{2}$, and that is what's called the *Nyquist frequency* or *Nyquist limit*.

Table 2.1: Digital audio notation

t	symbol representing a time value in seconds
τ	symbol representing a ”slow” time, time index with a lower resolution than t
ξ	symbol representing a frequency value in hertz
F_s	symbol representing a sampling rate of an audio signal
z	symbol representing a complex number
$\phi(z)$	symbol representing the phase of a complex number
$\|z\|$	symbol representing the magnitude of a complex number
$Q(x)$	function representing a uniform quantizer
Δ	symbol representing the step size of a uniform quantizer
$F(x)$	function representing μ-law compression
$x(t)$	function representing the amplitude of a continuous signal at a time t
x_n	sequence representing the amplitude of a discrete signal indexed by n
$w(t)$	continuous windowing function at a time t
w_n	discrete windowing function indexed by n
$X(\xi)$	function representing the frequency component of a signal for a frequency ξ
$S(\xi)$	the Fourier transform of a continuous signal
S_k	the discrete Fourier transform of a discrete signal
$S(\tau, \xi)$	the short-time Fourier transform of a continuous signal
$S_{k,\xi}$	the discrete short-time Fourier transform of a discrete signal
M_k	the Modified discrete cosine transform of a discrete signal

2.2.3 Quantization

Quantization is a process where we restrict a large set of values, possibly continuous, into a smaller set, generally discrete. Each of the values from the smaller set represents a *quantization level*, and has a range of values that fall into it.

For example, when we sample an analogue signal, what we get back is the amplitude represented by a voltage, which can be considered a set of infinitely many real numbers. In order to use these values on our computers for digital processing, these voltages must first be quantized. [2] An example of a common choice would be signed 16-bit PCM, where each sample is converted to an integer between $-2^{15} = -32768$ and $2^{15} - 1 = 32767$, representing the quantized amplitude for the given sample.

Quantization is also common in digital audio processing when we already have discrete values, e.g. when quantizing MDCT values (described later) in order to further reduce the range and allow for more efficient compression.

There are two kinds of quantization: uniform and non-uniform. The main difference between them is that when using uniform quantization, the quantization levels are uniformly spaced, i.e. the difference between any two quantized values is the same when the value is converted to the original range, whereas with non-uniform quantization the difference between different levels may vary.

2.2.3.1 Uniform quantization

The practical part uses uniform quantization extensively, therefore it's important to mention some extra details surrounding it.

In a uniform quantizer, there is a step size, denoted Δ, which is the value difference between two successive quantization levels.

We differentiate between a *mid-rise quantizer* and a *mid-tread* quantizer. The main difference between them is how they handle quantization of the area around 0. [4]

If we imagine quantization in the form of a staircase function, then in a mid-rise quantizer the "rise" of the stairs is the part that crosses the 0 point, whereas in a mid-tread quantizer it's the "flat" part of the staircase-like function (that one treads on), hence the naming.

In practice, in the context of a uniform quantizer, this means that a mid-rise quantizer (as opposed to a mid-tread quantizer) does not have a value to represent the zero value of the given range, but instead has the zero value as a classification threshold.

A mid-tread quantizer is defined as:

$$Q_t(x) = \Delta \cdot \left\lfloor \frac{x}{\Delta} + \frac{1}{2} \right\rfloor \tag{2.2}$$

A mid-rise quantizer is defined as:

$$Q_r(x) = \Delta \cdot \left(\left\lfloor \frac{x}{\Delta} \right\rfloor + \frac{1}{2} \right) \tag{2.3}$$

2.2.3.2 μ-law companding

μ-law (or mu-law) companding is an algorithm employing non-uniform quantization in order to reduce the quantization error and required bandwidth of a signal transmitting system. In practice it is mostly used in telephony and other applications that deal with a limited dynamic range.

Companding is combined from the words *compression* and *expanding*. Compression because the non-uniform quantization reduces the amount of bits needed to encode the peak amplitudes, and expanding on the other end to cancel the effect of the non-uniform compressor. There is one more intermediate step between them, specifically uniform quantization of the compressed signal. [5]

It has several important effects:

- the larger values get a bit smaller by getting rid of the least significant bits

- the lower values become higher due to conversion to a logarithmic scale

- some perceptible noise is masked due to boosting of the intended signal

μ-law compression is defined as: [6]

$$F(x) = \operatorname{sgn}(x)\frac{\ln(1+\mu|x|)}{\ln(1+\mu)} \qquad \text{for } -1 \leq x \leq 1 \tag{2.4}$$

And the inverse function for expanding:

$$F^{-1}(y) = \operatorname{sgn}(y)\left(\frac{1}{\mu}\right)\left((1+\mu)^{|y|} - 1\right) \qquad \text{for } -1 \leq y \leq 1 \tag{2.5}$$

Where μ is the compression parameter. The higher the value of μ, the higher a companded value $F(x)$ will be in average, and this increase will be more noticeable for lower x.

2.2.4 Windowing

A windowing function w_n is a function that is equal to 0 outside of some chosen interval, where its value is defined by an expression. It is generally symmetrical, though this is not a rule. Windowing functions are often applied to audio signals by multiplying them with the windowing function before further processing to avoid various artifacts.

There is no one "best window", there's usually a trade off, for example a window used to prevent aliasing will lower the accuracy of frequency identification (as is the case in the Hann window). [2] In fact, most modern audio codecs use different kinds of windows depending on the musical characteristics of the current block and the ones that follow. [7]

2.2.4.1 Rectangular window

The rectangular window is the simplest one, as it does not modify the original signal at all, and as such is often used to represent e.g. a part of a periodic signal. It's equivalent to:

$$w_n^R = 1 \tag{2.6}$$

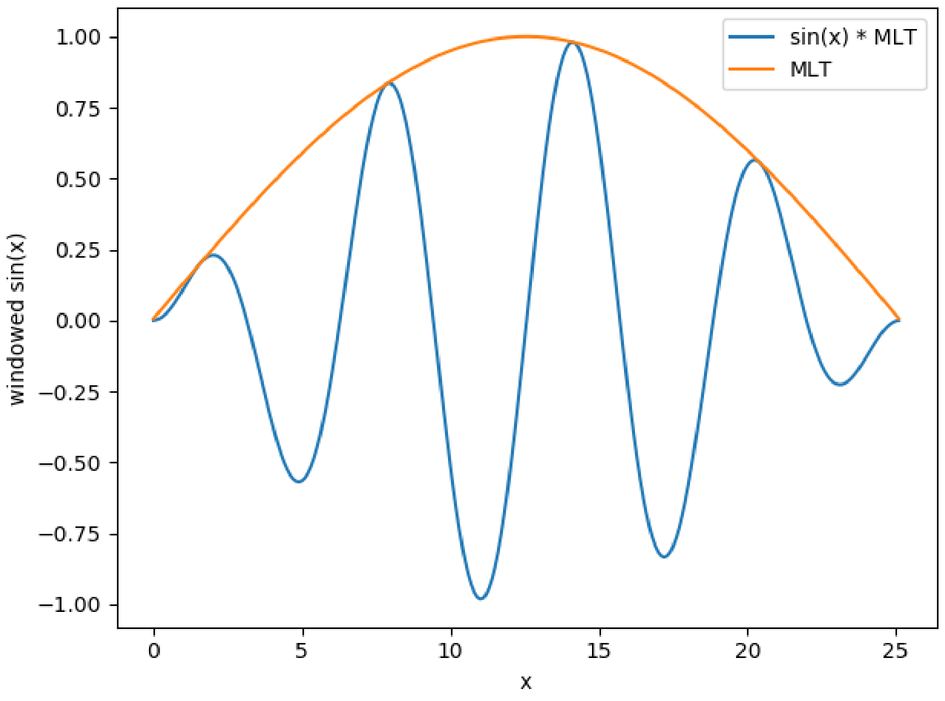
2.2.4.2 Hann window

A Hann window is a simple sine-based window, defined as:

$$w_n^H = \sin^2\left(\frac{\pi n}{N}\right) \tag{2.7}$$

This window is what we'll be using most of the time with the short-time Fourier transform.

Figure 2.2: An example of windowing: a sine function windowed by MLT.



2.2.4.3 Modulated lapped transform

Modulated lapped transform (MLT) is another sine-based window, used for example in the MP3 codec. It has a comparatively low computational complexity and is simple to implement. [8] It's defined as:

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

It is what we will be using with the MDCT due to its simplicity of implementation compared to the results it provides.

2.2.4.4 Kaiser-Bessel derived window

The Kaiser-bessel derived window (KBD) is designed to be used with the modified discrete cosine transform (MDCT), and is used for example in the AC-3 codec. It's defined as follows [2]:

$$w_n^D = \begin{cases} \sqrt{\frac{\sum_{i=0}^{n} w_i}{\sum_{i=0}^{\frac{N}{2}} w_i}} & \text{if } 0 \leq n < \frac{N}{2} \\ \sqrt{\frac{\sum_{i=0}^{N-1-n} w_i}{\sum_{i=0}^{\frac{N}{2}} w_i}} & \text{if } \frac{N}{2} \leq n < N-1 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

2.2.5 Transient

A transient is a high amplitude sound with a short duration. It's significant because during block-based audio encoding, if a transient occurs on the border between two blocks it may cause what's called *pre-echo* - essentially a type of artifact where a sound is being heard before it should occur.

To mitigate artifacts caused by transients, it's important to use proper windowing.

2.2.6 Aliasing

Aliasing is a type of artifact that occurs when the sampling rate is too low. Due to the way sampling works, frequencies above the *Nyquist limit* are mirrored to lower frequencies, creating a noticeable distortion. [2]

This can be eliminated by either choosing an appropriately higher sampling rate, or by passing the signal through a low-pass filter to eliminate the higher frequency content that would cause aliasing.

2.2.7 Spectral leakage

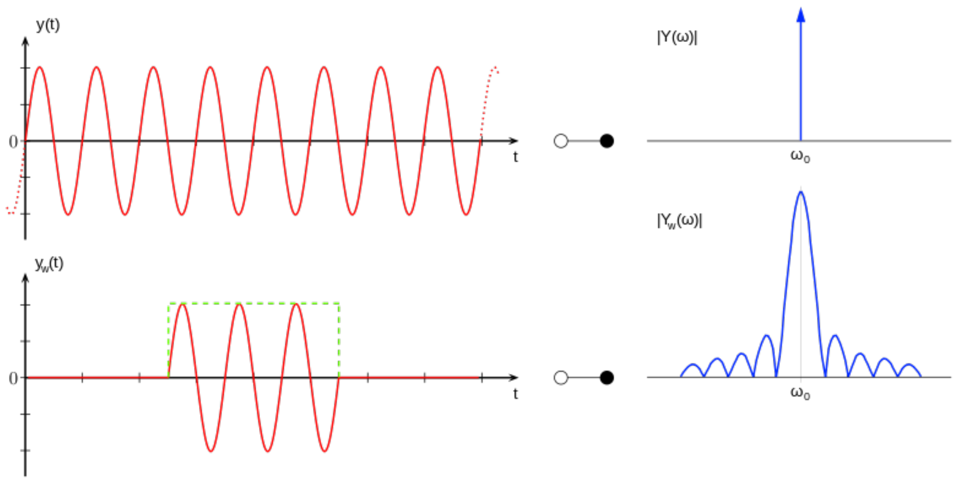
As we will see later, the Fourier transform (and other Fourier-related transforms) effectively projects the signal into infinity, as if it was a periodic signal. If we apply the transform to an unmodified signal, it's similar to using a rectangular window on a periodic function, where the signal ends abruptly at the edges. This in turn leads to *spectral leakage* - the resulting Fourier transform will produce non-zero values even for frequencies that aren't contained in the original signal at all.

The way to combat spectral leakage is again, by using proper windowing. By smoothing out the function using a window so that it approaches 0 along the edges of the signal, we can mitigate some spectral leakage, although not all of it.

2.2.8 Scalloping

A phenomena that occurs when using a discrete Fourier (or Fourier-related) transform. Since the frequencies are split into ranges rather than exact values,

Figure 2.3: An example of a sine function's spectral leakage after application of a rectangular window. Image by Walter Dvorak published under the CC BY 3.0 license.



it's possible that a single frequency may fall into two adjacent bins, providing inaccurate frequency content readings.

To deal with scalloping, a large enough block size in relation to the sampling rate must be chosen to lower the ranges of the bins.

2.3 Digital audio representation

Most commonly, the amount of air pressure is sampled many times a second and after being processed this information is stored as a discrete-time signal using numerical representations - this is what's known as a *digital audio signal.* This entire process is called *digital audio encoding.*

By sampling the audio signal, we will potentially be losing out on some information, but given a high enough sampling rate, the result will be imperceptible to the human ear. For general purpose audio and music, the standard sampling rate is 48 kHz, alternatively 44.1 kHz from the compact disk era.

Once we have our digital signal, there are two distinct kinds of ways we can represent, or, encode it. Both of them have many different data models for encoding [1], but in this work I am only going to focus on the most relevant ones.

2.3.1 Time domain representation

In the time domain, the signal is simply represented as a function of time, where t is the time and $x(t)$ is the raw amplitude, or air pressure, at that point. [2]

This is the most straightforward representation since it directly correlates to how the signal is being captured in the first place. However, as we will see later, this format is not ideal for storing audio data with any sort of compression.

2.3.1.1 PCM

In the time domain, the most basic encoding we can use is PCM (Pulse Code Modulation). After sampling a signal at uniform intervals, the discrete values are quantized; that is, each range of values is assigned a symbol in (usually) binary code.

For example using 16-bit signed PCM, each sample will be represented as a 16-bit signed integer, or in the case of multiple channels, N 16-bit signed integers, where N is the amount of channels.

PCM serves as a good base for what we are going to talk about next - Frequency domain representation and encoding.

2.3.2 Frequency domain representation

While it's simple to understand and work with for the computer with samples in the form of a sequence of amplitudes, it's difficult to run any sort of meaningful analysis on such data. To better grasp the structure of the audio we're working with, it would be helpful to be able to decompose it into its basic building blocks, so to speak. And that's where frequency based representation comes in.

The goal here is to represent the signal as not a function of time, but rather a function of frequency $X(\xi)$. That is, instead of having a simple sequence of amplitudes, we will have information about the magnitude for each component from a set of frequency ranges. This description alone is generally more compact than the PCM representation [2] on top of providing us with useful information about the signal, so it will serve as a good entry point to our compression schemes.

2.3.2.1 Fourier transform

Fourier transform is the first and arguably the most used tool for converting a signal from a function of time $x(t)$ into a function of frequency $X(\xi)$.

It is based on the *Fourier series*, which is essentially a representation of a periodic function as the linear combination of sines and cosines. [9] However, the main difference is that our function need not be periodic.

The Fourier transform of a continuous signal x is defined as: [10]

$$S(\xi) = \int_{-\infty}^{\infty} x(t)e^{-2\pi it\xi}dt \tag{2.10}$$

If we inspect the formula, we can notice that Fourier transform essentially projects our signal into infinity - this wouldn't be a problem if it was a periodic signal, but sampled audio is generally constrained by time. To prevent spectral leakage as a result, we must window the signal before processing it. [11]

The output is a complex number, which provides us with the means to find the magnitude and phase offset for the sinusoid of each frequency ξ.

The Fourier transform can also be inverted, providing us with an easy way to obtain the original signal back from its frequency components. The inverse transform is defined as:

$$x(t) = \int_{-\infty}^{\infty} S(\xi) e^{2\pi i t \xi} d\xi \tag{2.11}$$

However, seeing as our samples are discretely sampled, we will need to modify our transform accordingly.

The discrete Fourier transform of a discrete signal $x_0, x_1, ..., x_{N-1}$ is: [**?**]

$$S_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N} \tag{2.12}$$

And our inverse is:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} S_k e^{2\pi i k n / N} \tag{2.13}$$

In the discrete form, rather than finding the frequency content for a specific frequency ξ, we find the content of the k-th frequency bin. Since we have fewer values to work with, the frequency range is quantized to a degree and each bin contains an uniformly sized range of frequencies rather than a specific one.

The way that works is as following: if we for example run the discrete Fourier transform on 1152 samples recorded with a sampling rate $F_s = 44100$, we will end up with 1152 frequency bins, each containing the amplitude of a frequency range $\frac{F_s}{N} = \frac{44100}{1152} \approx 38$ Hz. However, due to the *Nyquist limit*, the only useful frequencies are up until $\frac{F_s}{2} = \frac{44100}{2} = 22050$ Hz, and so we only need to be concerned with the first half of the bins, i.e. $\frac{1152}{2} = 576$ bins, as the latter half mirrors the first and does not contain any useful information.

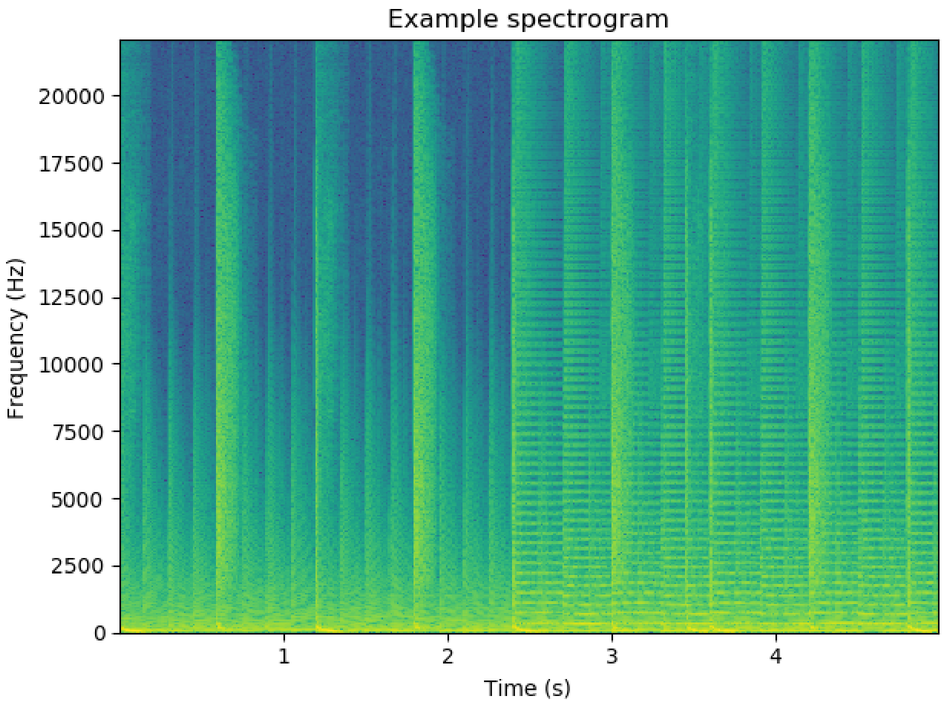
Due to the nature of this process, if we run the Fourier transform on our whole signal, we will only be able to analyse it as a whole. That means we won't be able to tell which parts of for example a song are quiet or if there are any parts with very high frequencies - we lose our temporal data.

To alleviate this problem, we can run Fourier transform on smaller chunks of the signal, analyse them separately and later join them back into the original signal. That is the essence of the Short-time Fourier transform.

2.3.2.2 Short-time Fourier transform

When using Short-time Fourier transform, or STFT for short, we first split the signal into smaller segments of equal size and then run Fourier transform on those separately. As such, our output can be projected into two dimensions - specifically a frequency spectrum as a function of time, a spectrogram.

Figure 2.4: An example of an audio spectrogram for the signal from Figure 2.1.



Doing it this way will let us see how the frequency components change over time instead of taking the spectrum of the entire signal.

As with regular Fourier transform, we'll need to window each segment of the signal, but there is a caveat. Since we have windowed segments, we may be losing some information at the edge of each segment leading to artifacts, and furthermore we may be losing information about transients. To solve this, we'll need to introduce overlapping windows - however, having an overlap will increase the amount of coefficients required.

The continuous version is defined as: [?]

$$S(\tau, \xi) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-2\pi i t \xi} dt \tag{2.14}$$

where w is the window function.

But again, as we have discrete samples, we will need to use a discrete short-time Fourier transform, specifically:

$$S_{k,\xi} = \sum_{n=-\infty}^{\infty} x_n w_{n-k} e^{-2\pi i \xi n} \tag{2.15}$$

And similarly to the regular Fourier Transform, short-time Fourier Transform is also invertible. [12]

STFT is commonly used for audio analysis (e.g. for generating spectrograms) but in this case it will be used as a means for our NMF compression.

2.3.2.3 Modified discrete cosine transform

Modified discrete cosine transform, or MDCT for short, has become the dominant means of lossy high-quality audio coding. [13]

It is what's known as a *lapped transform*. This means that when transforming a block into its MDCT coefficients, the basis function overlaps the block's boundaries. [14] In practice, what this means is that while we have blocks with overlapping windows as in the short-time Fourier transform, the number of coefficients remains the same as without while retaining the relevant properties.

As the name suggests, MDCT is based on the Discrete cosine transform [15], namely *DCT-IV*, where the main difference is the addition of lapping mentioned above. The type then specifies the boundary conditions, to determine what values the transform should have for points outside the original range.

What makes MDCT simpler to work with compared to Fourier transform is that not only do we not need more coefficients despite overlapping, they are also real numbers as opposed to complex numbers, lowering the amount of bytes necessary to store them.

It is a linear function $f : \mathbf{R}^{2N} \to \mathbf{R}^{N}$, defined as: [16]

$$M_k = \sum_{n=0}^{N-1} x_n \cos\left\{\frac{(2n+1+\frac{N}{2})(2k+1)\pi}{2N}\right\} \tag{2.16}$$

for $k = 0, 1, \ldots, \frac{N}{2} - 1$.

It is assumed that $x(n)$ is already windowed by an appropriate windowing function w.

MDCT is also invertible, and its inversion is defined as:

$$\bar{x}_n = \sum_{k=0}^{\frac{N}{2}-1} M_k \cos\left\{\frac{(2n+1+\frac{N}{2})(2k+1)\pi}{2N}\right\} \tag{2.17}$$

for $n = 0, 1, \dots, N - 1$.

It's important to note that the inverted transformed sequence $\bar{x}_n$ by itself does not correspond to the original signal x_n [17]. To achieve perfect invertibility, we must add subsequent overlapping blocks of the inverted MDCT (IMDCT). This method is called *time domain aliasing cancellation* [18], or TDAC for short. As the name suggests, it mainly helps remove artifacts on the boundaries between transform blocks.

2.4 Psychoacoustics

Apart from time-frequency representations being generally more compact, they also give us the ability to analyse, isolate or modify the frequency composition of a given signal. This comprises a large chunk of the audio compressing process.

The field of psychoacoustics studies sound perception - that is, how our ears work and how we perceive different kinds of sounds. There are many different characteristics to sound that need to be taken into account for a proper psychoacoustic analysis [19], split into several categories, namely:

tonal includes pitch, timbre, melody harmony

dynamic based on loudness

temporal involves time, duration, tempo and rhythm

qualitative represents harmonic constitution of the tone

For music, it's important to balance these four qualities appropriately. For compression, the most important qualities for us in scope of this work are going to be tonal (pitch) and dynamic (loudness).

2.4.1 Pitch

Pitch is a characteristic that comes from a frequency. The difference between the two is that pitch is our subjective perception of the tone whereas a frequency is an objective measure. Despite this fact, pitch is often quantified as a frequency using Hertz as its unit.

The lower bound of human hearing is around 20 Hz whereas the upper bound is most commonly cited as 20 000 Hz, or 20 kHz. [20] In a laboratory environment, people have been found to hear as low as 12 Hz. As people age, our hearing gets progressively worse and a healthy adult younger than 40 years can generally perceive frequencies only up to 15 kHz. [19]

The human ear is capable of distinguishing different frequencies fairly accurately, though accuracy gets lower with increasing frequency. It's easier for our ears to tell a difference between 500 Hz and 520 Hz compared to the difference between 5000 Hz and 5020 Hz. [21]

Furthermore, if we hear two different tones simultaneously, but their frequencies are close enough to one another, we may perceive them as a combination of tones rather than separate tones. Frequency ranges, or bands, where this phenomenon happens, are called *critical bands*. [22] It's also possible for one tone to mask the other entirely, and then we get what's called *auditory masking*. [23]

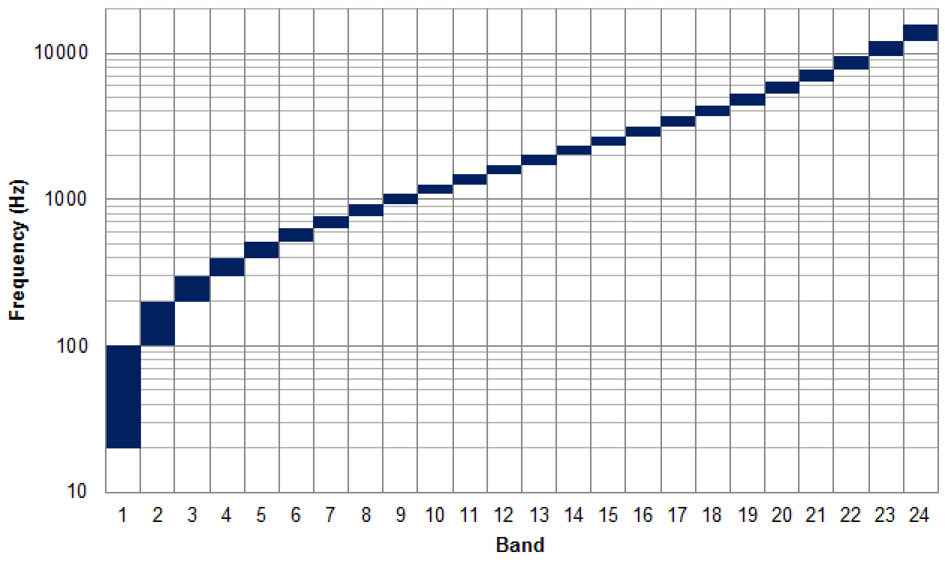
Based on the knowledge of the existence of these critical bands, it's possible to devise a system that specifies the range of each band in human hearing. One such scale that is commonly used is called the *Bark scale*.

2.4.1.1 Bark scale

The Bark scale ranges from 1 to 24 Barks, where each Bark corresponds to a single critical band of human hearing. [24] The perceived difference in pitch between each band should be the same, despite the scale not growing linearly in terms of frequency ranges. Specifically, until around 500 Hz, the scale is roughly linear, but above that it has a more logarithmic growth. [25]

The Bark scale is commonly used as reference for audio encoding codecs, as we will see later. Knowledge of these critical bands allows for more educated byte allocation during the quantization process when compressing a frequency domain representation.

Figure 2.5: The Bark scale. Image by ”Swpb” licensed under CC BY-SA 4.0.



2.4.2 Loudness

What people often decide as loudness is really called *sound pressure level* and it's measured in decibels (dB), however it has some shortcomings when it comes to psychoacoustic analysis.

It is defined as following: [26]

$$L_p = 20 \log_{10} \left(\frac{p}{p_0} \right) \mathrm{dB} \tag{2.18}$$

where p is a sound's sound pressure and p_0 is a reference sound pressure, also called the threshold of human hearing.

While this metric is very popular, it doesn't account for the fact that different frequencies have a different perceived loudness for a person's ears. [19] There is a lot of research in recent years into how different frequencies impact our perception and hearing [27], but that is out of scope of this work. For more information about the exact definitions of loudness, refer to [19].

2.4.3 Auditory masking

As mentioned above, when it comes to audio masking, and therefore audio compression, we must not only take into account the critical bands as per e.g. the Bark scale, but also their intensity.

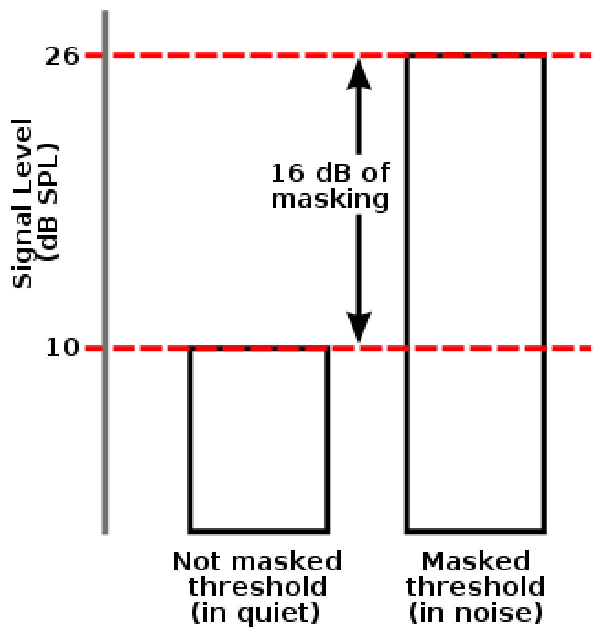
For example a lower frequency sound may mask one of a higher frequency, but the other way around does not apply. [23] Modern audio encoders take this into account and using this knowledge are able to eliminate sounds that exist in the original signal, but are not perceivable by humans.

There are two important different kinds of masking effects - *simultaneous* masking and *temporal* masking. [7]

Simultaneous masking is what I have hinted at above - when there are two sounds within the same critical band, the dominant one may mask other frequencies within the same band. This can be compensated to a degree by increasing the volume of the masked sound.

Temporal masking does not occur in the frequency domain, but the time domain. The essence is that a stronger tonal component may mask a weaker one if they appear within a small window of time in succession.

Figure 2.6: A hypothetical example of how a masker can shift the hearing threshold of a signal by 16 dB. Image created by Alan De Smet and published in the public domain.



Chapter 3

Non-negative matrix factorization

In today's age of big data, machine learning and various other fields, it's important to have ways to quickly analyse these datasets and ideally find patterns within. Non-negative matrix factorization is one of the paradigms suitable for that task.

In layman's terms, what NMF does is that when we are given a large set of data, for example a matrix representing books and their review scores from people, we can extract certain hidden "features" from it using NMF, in this case representing e.g. various genres (basis matrix) and how prominent they are in a given book (weight matrix). And then, using these two matrices, we are able to estimate or even predict what kind of books a user would like - this is a simple example of a possible recommendation algorithm.

3.1 Linear dimensionality reduction

Non-negative matrix factorization, or NMF, falls under *linear dimensionality reduction* techniques. These are used widely for noise filtering, feature selection or compression, among others.

LDR can be defined as following: [28]

$$x_j \approx \sum_{k=1}^{r} w_k h_j(k) \qquad \text{for some weights } h_j \in \mathbf{R}^r \tag{3.1}$$

where given a data set of size n, we define $x_j \in \mathbf{R}^p$ for $1 \leq j \leq n$, $r < \min(p, n)$, and $w_k \in \mathbf{R}^p$ for $1 \leq k \leq r$.

What this effectively means is that we represent p-dimensional data points in a r-dimensional linear subspace, with basis elements w_k and data coordinates given by vectors h_j. LDR defined in this manner is equivalent to low-rank matrix approximation, which is the essence of non-negative matrix factorization.

3.2 NMF definition

Non-negative matrix factorization solves the following NP-hard problem:

Given a non-negative matrix V, find non-negative matrix factors W and H such that:

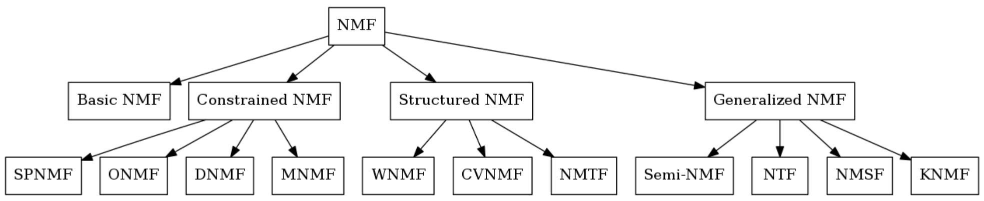
$$V \approx WH \tag{3.2}$$

That is, given a set of multivariate n-dimensional data vectors, we place these vectors in the columns of a $n \times m$ matrix V, where m is the amount of examples we have. We then approximately factorize this matrix into two different matrices: a $n \times r$ matrix W and a $r \times m$ matrix H. We generally choose $r < \min(n, m)$ (though this is not required) so that the two matrices are smaller than the original matrix V, essentially compressing it. [29] The variable r is usually referred to as the *rank* of the approximation.

3.3 Classification

NMF is as of currently still a relevant research topic, and has been explored by researchers from many different fields including mathematicians, statisticians, computer scientists or biologists. Given the wide range of use, over time it lead to different variations and additional constraints on the algorithms. Therefore, a taxonomy system was proposed in [30], outlined below.

Figure 3.1: The NMF classification as per [30].



3.3.1 Basic NMF

This is the basic model which only enforces non-negativity, and which all the following ones build upon.

However, due to its unconstrained nature, without any other constraints there are many possible solutions which may lead to the algorithm's performance to vary. Further constraints outlined below help in the search of unique solutions and optimizing for specific scenarios.

3.3.2 Constrained NMF (CNMF)

Constrained NMF imposes additional constraints on the resulting matrices, namely:

Sparse NMF SPNMF, sparseness constraint

Orthogonal NMF ONMF, orthogonality constraint

Discriminant NMF DNMF, couples discriminant information along with the decomposition

NMF on manifold MNMF, preserves local topological properties

3.3.3 Structured NMF (SNMF)

Structured NMF modifies standard factorization formulations:

Weighed NMF WNMF, attaches weights to different elements relative to their importance

Convolutive NMF CVNMF, considers time-frequency domain factorization

Non-negative Matrix Trifactorization NMTF, decomposes the data into three matrices

3.3.4 Generalized NMF (GNMF)

Generalized NMF can be considered a broader variant of Basic NMF, where conventional data types or factorization modes may be replaced with something different. It's split as follows:

Semi-NMF relaxes the non-negativity constraint on a specific factor matrix

Non-negative Tensor Factorization NTF, generalizes the model to higher dimensional tensors

Non-negative Matrix-set Factorization NMSF, extends the data sets from matrices to matrix-sets

Kernel NMF KNMF, non-linear model of NMF

3.4 Properties

The additional constraint of non-negativity is important, as results show that it leads to a natural higher sparseness in both the basis matrix (W) and the encoding matrix (H). Additionally, non-negativity leads to a parts-based representation, which is similar to how our brains are presumed to work, basically combining parts in an additive manner to form a whole instead of subtracting. [31] This sparseness makes it even easier to further compress the resulting matrices, saving us more space.

However this isn't without any downsides. While the concept of adding parts together seems to make a lot of sense, there is an issue. Since NMF employs a holistic approach, the additive parts learned by it in an unsupervised mode only considers features on a global level, and does not allow for representation of spatially localized features. [32]

So while on paper NMF might seem better than PCA or SVD for a parts-based representation, it only comes at a cost of increased complexity, and since both PCA and SVD have a more compact spectrum than NMF, we must consider if this is worth the trade-off. [30]

3.5 Algorithms

For the purposes of this thesis, we will only consider algorithms for Basic NMF. While NMF has been widely used in sound analysis etc. as we'll see below, its use for audio compression specifically is rare and there are limited resources to provide insight into utilizing possible constraints, therefore we will only be using the standard version.

Finding a decomposition of a matrix V into matrices W and H is an NP-hard problem, and as such, the resulting matrices are generally only approximated over a number of iterations of an optimization algorithm. What this means in practice is that it's likely a result we'll find is sub-optimal or a local minimum.

3.5.1 Cost function

When using iterative updates, in each step of the process we need to evaluate the quality of the approximation. The function that does this is called the *cost function*, or *objective function*.

There are two simple commonly used functions. Firstly, we can use squared Euclidean distance: [33]

$$||A - B||^2 = \sum_{ij} (A_{ij} - B_{ij})^2 \tag{3.3}$$

This is lower bounded by 0, which it only is equal to if $A = B$.

Another metric we can use is based on Kullback-Leibler divergence, and is defined as such: [29]

$$D(A||B) = \sum_{ij} \left(A_{ij} \log \frac{A_{ij}}{B_{ij}} - A_{ij} + B_{ij} \right) \tag{3.4}$$

3.5.2 Update rules

With the cost function in place, we now need a function to apply each iteration to try and minimize the value of the cost function. It has been found that a good compromise between speed and ease of implementation is to use what's called *multiplicative update rules.* [29] Despite being over 15 years old, they are still very commonly used exactly for this reason.

For non-increasing Euclidean distance $||V - WH||$, if W and H are at a stationary point of distance, we may use these rules:

$$H_{a\mu} \leftarrow H_{a\mu} \frac{(W^T V)_{a\mu}}{(W^T W H)_{a\mu}} \tag{3.5}$$

$$W_{ia} \leftarrow W_{ia} \frac{(V H^T)_{ia}}{(W H H^T)_{ia}} \tag{3.6}$$

And for non-increasing divergence $D(V||WH)$, if W and H are at a stationary point of divergence, we can use this:

$$H_{a\mu} \leftarrow H_{a\mu} \frac{\sum_i W_{ia} V_{i\mu} / (WH)_{i\mu}}{\sum_k W_{ka}} \tag{3.7}$$

$$W_{ia} \leftarrow W_{ia} \frac{\sum_\mu H_{a\mu} V_{i\mu} / (WH)_{i\mu}}{\sum_v H_{av}} \tag{3.8}$$

3.5.3 Initialization

Before we can use our update rules to iteratively optimize the cost function, we need some initial value to initialize the matrices W and H to, first. In practice, different initializations generally yield different solutions, so it may be worth considering. [34]

The most basic way of initialization is to simply initialize the matrices with random values. This generally works decently but you somewhat lose out on controlling the composition of the matrices. A small way to improve this is to generate random matrices a couple times and pick the one with the lowest cost function value, but the issue remains.

As per [34], there are tons of different ways to initialize the matrices, usually relating to constraints on the matrices, e.g. initialize H in such a way that none of the values are above a certain threshold etc. But given the low amount of research on audio compression using NMF, it's difficult to gauge which of these might work better for audio than others, and as such this work will not elaborate on different ones further.

3.6 Use in digital audio processing

In digital audio, Non-negative matrix factorization is mostly used as a tool for analysis rather than compression. The ability to extract hidden features from a given signal is useful for certain things.

For example NMF sees use in audio separation tasks. [35] The gist of this lies in first creating a spectrogram of the signal using STFT and then using NMF decomposition to isolate different kinds of sounds or instruments from it, roughly represented by the basis matrix.

Another example of use is musical transcription. [10] The idea here is to process an input signal via STFT, further filtering and NMF, to isolate individual notes from e.g. a piano piece.

Both of these methods have seen some success, but as we will see later, applying NMF to compression rather than analysis is not a simple task and might not even be worth it.

Chapter 4

Lossy audio compression

Compression can be split into two kinds - lossy and lossless. Using ”lossless” in the context of audio is a bit misleading, since sampling itself is a lossy process, but using a high enough sampling rate, we will not notice any difference, so sampled audio without any lossy compression will be our baseline.

For audio, lossless compression generally means taking some form of digital audio representation and losslessly compressing this data. This will preserve the signal in its entirety with a reduced bit-rate. However, due to size of such audio (an audio CD could only fit about 80 minutes of such music sampled at 44.1 kHz), it’s become more common to use a lossy format.

Lossy compression implies that there will be loss of data, and while this is true, thanks to the application of various psychoacoustic principles size of audio can be greatly reduced without altering human perception, leading to vastly smaller bit-rates for no real cost.

This work focuses on lossy audio compression, therefore only lossy codecs will be considered for comparison.

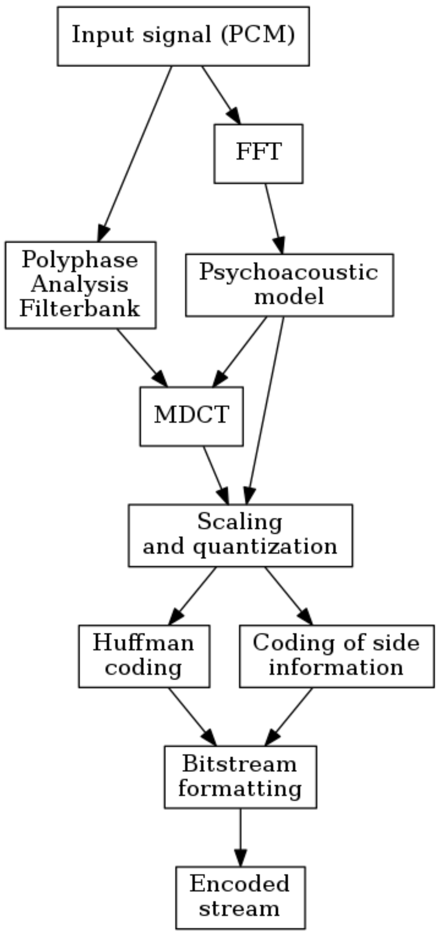
4.1 Overview

4.2 State of the art

Due to its qualities of efficiently compacting energy and mitigating artifacts at block boundaries, MDCT is the most commonly used transformation in modern lossy audio coding, and is employed in the most popular audio formats including MP3, Opus, Vorbis or AAC.

In this section, I will elaborate on some of the more popular ones to get an idea of what considerations go into writing a modern audio codec. They are also the ones that will be used for comparison with my own encoding scheme.

Figure 4.1: The MP3 encoding scheme, simplified.



4.2.1 MP3

MP3, or MPEG-1 Layer III has been standardized in 1991 and has since become widespread throughout a multitude of electronic devices as the de-facto standard for music storage. Its simplified encoding scheme can be seen in Figure 4.1.

It's a very powerful compression/decompression scheme capable of reducing the bit-rate of an audio stream by up to a factor of 12 without any noticeable (to humans) quality degradation. In other words, to transmit CD quality audio, it needs a bitrate of 128 kbps. [7]

The core of MP3 compression is the Modified discrete cosine transform. The signal represented in its PCM form is first split into 32 subbands using an analysis polyphase filterbank, and each of those is further split into 18 MDCT bins, so overall we end up with 576 MDCT frequency bins per frame.

These bins are then sorted into 22 scalefactor bands, which roughly correlate to the 24 bands of human hearing. The point of these bands is that you may individually scale each of them up or down depending on how much precision you need for that specific frequency range. This usually done by dividing and rounding the values in the band, losing a certain amount of information; this process will be reversed during decoding.

The signal is also analyzed using the Fourier transformation, which gives us frequency information for the signal in the same frame, and we can use this information to determine how much to scale each scalefactor band - e.g. if there's some weak sound that will be masked by another, we can assign the band it's in lower precision, saving data. [36]

Once we have the scaled and quantized data, we use the Huffman encoding to losslessly compress these values, and format this output into the final bitstream, encoding our audio.

4.2.2 Opus (CELT)

The Opus codec has been standardized fairly recently, in 2012 [37] compared to other widespread audio codecs. Opus was created from two core technologies - Skype's SILK codec based on linear prediction, and Xiph.Org's CELT codec, based on the MDCT. [38]

As a result, Opus is capable of performing in three different modes:

SILK used for speech signals

CELT used for high-bitrate speech and music

Hybrid both SILK and CELT used simultaneously

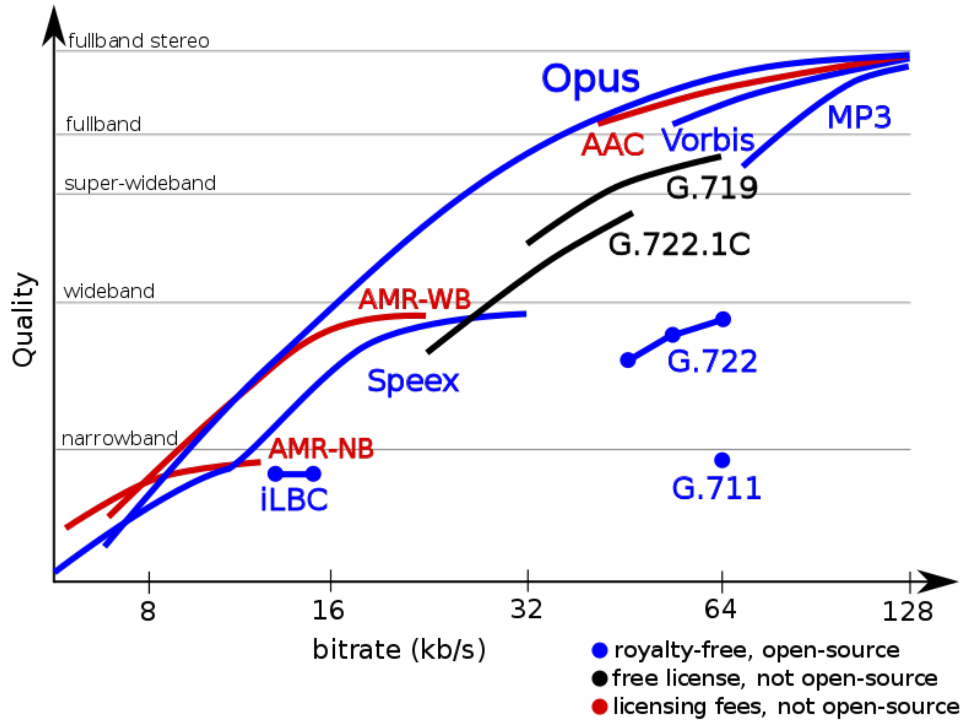
This thesis will mainly focus on Opus CELT due to it being more general purpose than SILK, which will make it easier to use for comparison with a NMF-based codec. SILK uses a technique known as Linear predictive coding, whose complexity and difficulty of implementation exceeds the scope of this work.

Opus is designed with real-time constraints in mind, that is, for example network music performances which require very low delays. Despite that, however, its compression is comparable to codecs with higher delays intended for streaming or storage. This makes it a good candidate to test against in this work, alongside MP3.

CELT (Constrained Energy Lapped Transform) is based on MDCT similar to MP3, but the main difference is that Opus uses specifically sized bands with ranges similar to the Bark scale in order to preserve the spectral envelope of the signal.

Similar to MP3, Opus CELT works on the basis of quantizing MDCT coefficients, but utilizes various kinds of prediction and focuses more on handling transients, and is much more complex in general. Through various optimizations Opus achieves similar results but at a reduced bitrate.

Figure 4.2: This figure attempts to summarize the results of a collection of listening tests, comparing various lossy audio codecs. Image attributed to Jean-Marc Valin and licensed under CC BY 3.0.



4.3 Compression using NMF

This section is separate from the state of the art, as NMF is not really used for audio compression in practice. There is one notable example that will be covered and to an extent implemented here, for more details refer to [39].

The proposed method uses a metric previously formulated in [40] called *noise-to-mask ratio*, or NMR for short. They successfully apply this metric onto non-negative matrix factorization, establishing a custom cost function to represent NMR and also propose update rules based on Euclidean distance for minimizing this cost function.

Results showed that NMR performed better than other, more general cost functions. Audio compressed using NMF + NMR had better perceptual quality and managed to encode signals with lots of transient sounds in better quality. However, NMR in its current form is part of a proprietary algorithm known as PEAQ [41], and while the standard is public, it is under-specified and a conforming implementation is not available, so this work will not use it.

This modified NMF is then used in what the paper refers to as *object-based audio coding*, and the principle is fairly simple to understand. Instead of using MDCT for conversion to the frequency domain, they use STFT, obtaining the frequency spectrum as a set of complex values. They then take the magnitude and phase of each of these complex values to obtain the *magnitude spectrogram* and *phase information*.

The magnitude spectrogram is then compressed using NMF (+NMR) among other things, whereas the phase information is entropy encoded separately and does not utilize NMF at all.

The conclusion is that while object-based audio coding done this way achieves reasonable bit-rate and quality on the magnitude spectrogram, it is difficult to efficiently encode the phase information and as such the bit-rate of this approach proved to be too high to be usable in practice compared to other, more common, MDCT based methods.

Chapter 5

Design

In this chapter I will give an overview of how this audio codec will be designed, and explaining the various decisions I made along the way.

I have decided to call this new codec *ANMF* (stands for Audio-NMF), using files with the extension *.anmfx*, where x represents the compression method. It will be implemented as a command line utility.

There are currently three different ANMF formats that you can choose from, and their main difference is which audio representation is being compressed by NMF. They are as follows:

ANMF-RAW denoted by r, compresses the signal in PCM form (time domain)

ANMF-MDCT denoted by m, compresses the signal transformed with MDCT (frequency domain)

ANMF-STFT denoted by s, compresses the signal transformed with STFT (frequency domain)

5.1 WAVE file

WAVE, WAV or Waveform audio is a file format for storing digitized audio, created as a joint design by the Microsoft Corporation and the IBM Corporation. They are built on top of the chunk-based RIFF format. For details on the specific structure of a WAVE file please refer to [42].

It stores raw uncompressed audio samples in PCM format along with some metadata and will serve as both the standard input and output to the ANMF codec. Most commonly used formats can be converted to and from WAV as well and thus it will serve as a good baseline.

Samples in this format can be represented by different datatypes, I chose 16-bit signed integers, i.e. each sample's amplitude is represented by a whole number between -32768 and 32767. The sample rate can vary, but a good standard value the experiments will use is 44.1 kHz, which corresponds to audio CD quality.

5.2 ANMF File structure

The base container for the compressed ANMF file is the same no matter which encoding method you use. The bytes are saved in little endian byte order.

Please refer to Table 5.1 and each encoding method's table for the specific file structure. The first eleven bytes are mostly set in stone other than the method specification, but after that it varies greatly.

Table 5.1: ANMF file structure

Bytes	Data type	Description
0-3	char[]	identifier string ”ANMF”
4	char	method used, can be ’S’, ’R’ or ’M’
5-6	uint16	# of channels
7-10	uint32	sample rate
11-?	enc_data	encoded data depending on the method

When serializing matrices to a file, the structure in Table 5.2 will be used, denoted by a ”matrix(*dt*)” datatype in the following tables, where *dt* stands for the datatype used for the matrix elements.

Table 5.2: Serialized matrix structure

Bytes (relative)	Data type	Description
0-3	uint32	amount of rows in the matrix
4-7	uint32	amount of columns in the matrix
8-$(8 + x - 1)$	*dt*	row-wise values of the matrix, $x = rows * columns$

There's also one more kind of matrix, a matrix using Huffman encoded quantized values. This will be denoted by a ”quant_matrix” datatype. For its specification please see Table 5.3.

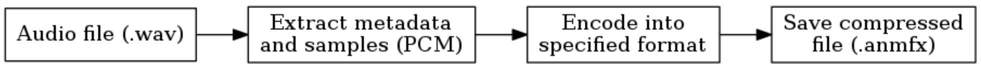
Table 5.3: Serialized quantized matrix structure

Bytes (relative)	Data type	Description
0-3	uint32	amount of rows in the matrix
4-7	uint32	length L of the Huffman encoded byte stream
8-$(8 + L - 1)$	byte[]	Huffman encoded byte stream representing the matrix

5.3 Encoder

The encoder is responsible for taking a raw audio file and encoding the data within, producing a compressed version of the original. Please refer to Figure 5.1 for a visual representation of the process.

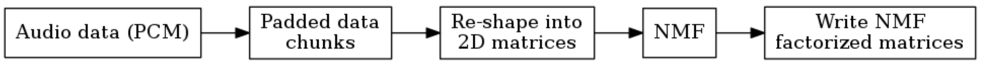
Figure 5.1: A high level overview of the ANMF audio encoder.


Audio file (.wav)
Extract metadata and samples (PCM)
Encode into specified format
Save compressed file (.anmfx)

Next, each format's encoding process will be outlined (third step in the figure). If the audio file has multiple channels, this process is repeated on each channel separately.

5.3.1 ANMF-RAW

Figure 5.2: The encoding scheme for ANMF-RAW.



ANMF-RAW works on the principle of applying NMF directly to the PCM audio samples x_n. However, the samples are initially an array of 16-bit signed integers, and as such, they need to be processed first before NMF can be used.

A chunk shape is specified to determine how many rows and columns each matrix will have before NMF. We choose a target matrix shape of 1152×200 to match the amount of samples per frame in the other methods.The sample array is then padded with zeroes to ensure there are enough elements at the end of the array to ensure every matrix can have the same shape and number of elements. The amount of padding must be written to the output so that we know the length of the original array when decoding.

Once the array is padded, we iterate over the samples and split them into equal chunks of size $rows * columns$. This array is then "folded" to produce a matrix of the desired shape. We then obtain a matrix of signed integers, so in order to be able to use NMF, we first need to get rid of all the negative values. To do that, we increment each chunk by the absolute value of its smallest element, guaranteeing that the lowest value in the matrix is ≥ 0.

Once we have this matrix, we proceed by applying basic random-initialized Euclidean-based NMF on it, obtaining the basis matrix W and coefficient matrix H. Lastly, for each chunk, we write the value we incremented the matrix by, and the two decomposition matrices.

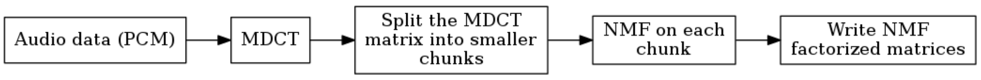
Table 5.4: ANMF-RAW data structure

Bytes (relative)	Data type	Description
0-3	uint32	amount of zeroes used to pad the samples
4-7	uint32	amount of chunks
8-?	data_chunk[]	NMF-compressed data chunks (refer to Table 5.5)

Table 5.5: ANMF-RAW structure of each data chunk

Bytes (relative)	Data type	Description
0-7	float64	absolute value that the matrix was incremented by
8-?	matrix(float32)	matrix W
?-?	matrix(float32)	matrix H

Figure 5.3: The encoding scheme for ANMF-MDCT.


Audio data (PCM)
MDCT
Split the MDCT matrix into smaller chunks
NMF on each chunk
Write NMF factorized matrices

5.3.2 ANMF-MDCT

In ANMF-MDCT, as the name suggests, the PCM input will be transformed using MDCT as per Section 2.3.2.3. Since this is a lapped transform, each transformed block will have a 50% overlap with the following block. We choose a frame size of $2N = 1152$ and split the signal into blocks of that size, thus each block will contain $N = 576$ coefficients from its own block, and another 576 from the following one.

The decision to have a block size of $N = 576$ stems from the fact that this will give us the same amount of frequency resolution that e.g. MP3 uses (as seen in Section 4.2.1), which proved to be enough for human hearing.

As before, we first pad the signal at the end with zeroes to align it to the desired block size. To prevent loss of data in the first and the last block due to the overlapping, we further pad the signal by an array of zeroes, equal in size to the size of a block, that is N zeroes both at the beginning and the end of the signal.

Then, we apply a windowing function on each block to bring the values near the edges closer to 0 to help mitigate spectral leakage. We use the MLT window w_n^M as defined in Section 2.2.4.3.

Finally, we apply the MDCT on each of the windowed blocks and obtain a matrix of MDCT coefficients in the form of real numbers.

This matrix is then split into smaller chunks. For example, if the MDCT matrix contains 576 rows and 1100 columns, we might split it into submatrices sized 576×200, with the last one being 576×100, as no padding is necessary here. This amount of chunks is written to the output. We increment the MDCT matrix by its lowest value to get rid of negative values. Basic random-initialized Euclidean-based NMF is then ran on each of the chunks separately and the decomposition matrices serialized.

Table 5.6: ANMF-MDCT data structure

Bytes (relative)	Data type	Description
0-3	uint32	amount of zeroes used to pad the samples
4-7	uint32	amount of MDCT submatrix chunks
8-?	data_chunk[]	NMF-compressed MDCT chunks (refer to Table 5.7)

Table 5.7: ANMF-MDCT structure of each data chunk

Bytes (relative)	Data type	Description
0-7	float64	absolute value that the matrix was incremented by
8-?	matrix(float32)	matrix W
?-?	matrix(float32)	matrix H

5.3.3 ANMF-STFT

The design of ANMF-STFT is based on the solution suggested in [39] with some changes along with only utilizing open source solutions.

Like with ANMF-MDCT, we choose a frame size of $N = 1152$, leading to a frequency resolution of 576 bins. We begin by properly padding the signal to N so that it's possible to be split into equal parts. We then use STFT with 50% overlap and a block size of N, which means we end up with twice the coefficients compared to MDCT, but this is not a major issue.

During STFT, we must again window each block, leading to overlapping windows. We use the Hann window w_n^H for this (as defined in Section 2.2.4.2).

Once STFT is finished, we end up with a matrix of Fourier transform coefficients in the form of complex numbers. Trying to apply NMF on the complex numbers directly would yield similar results to ANMF-MDCT, so we have to approach this differently.

If we visualise the complex valued elements in the complex plane, we can instead represent each element z as two separate values:

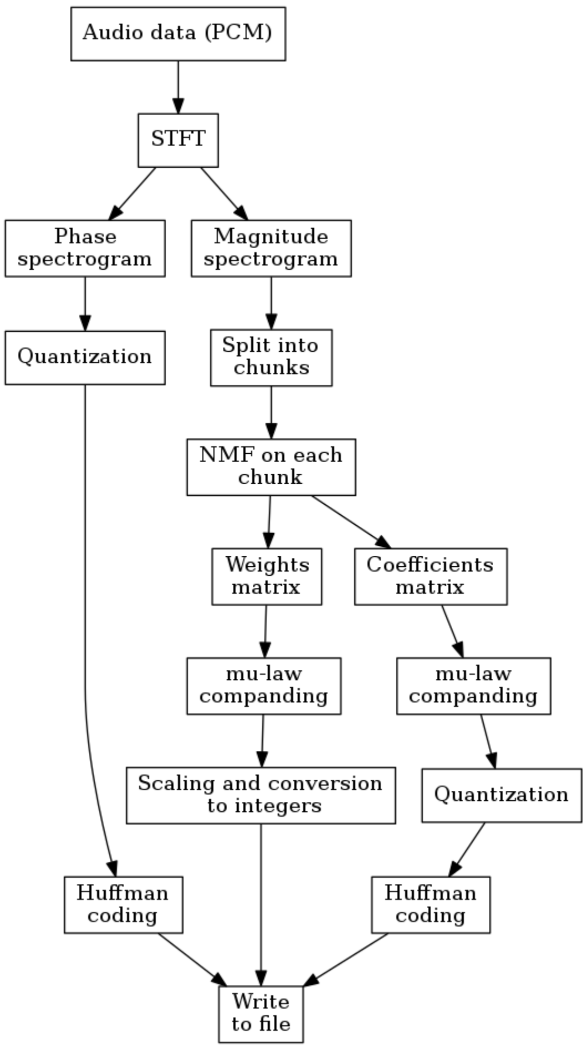
magnitude also called the modulus, geometrically it's the distance from 0

phase also called the argument, geometrically it's the angle from the real axis

To obtain the phase ϕ of a complex number $z = x + iy$, we can use the following formula:

$$\phi(z) = \arg(z) = \operatorname{arctan2}(y, x) \tag{5.1}$$

Figure 5.4: The encoding scheme for ANMF-STFT.



And to obtain the magnitude $|z|$ of the complex number:

$$|z| = \sqrt{x^2 + y^2} \tag{5.2}$$

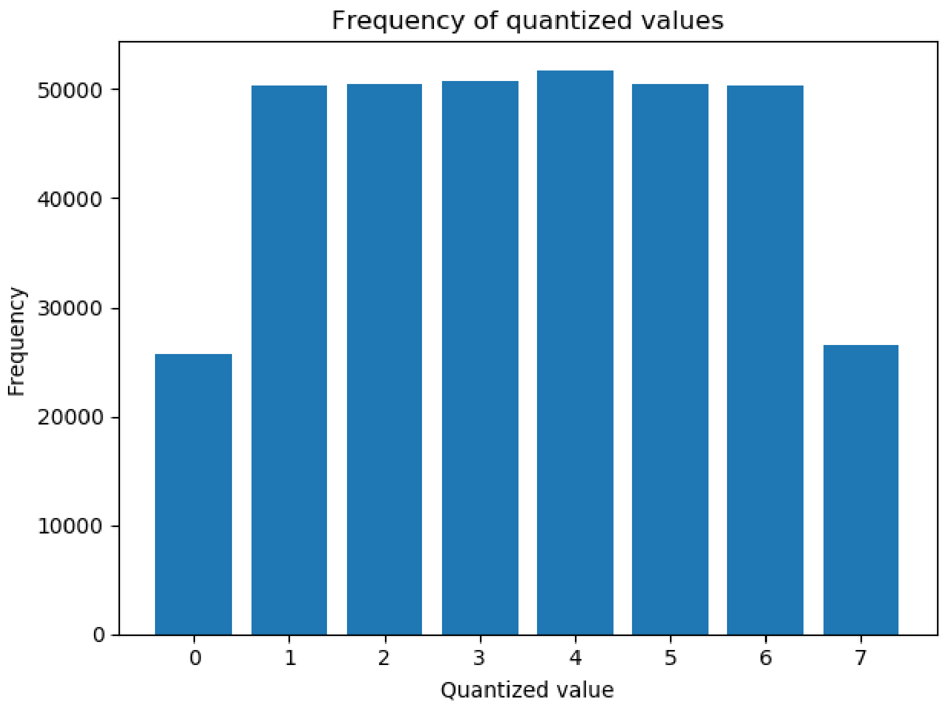
By calculating the magnitude and phase of every element in the STFT matrix individually, we obtain the magnitude spectrogram and the phase spectrogram respectively. We now need to encode both of them individually.

For the phase matrix, my experiments showed that applying NMF on it leads to a very noticeable loss in quality, so instead I opted for a different solution that ultimately ends up saving more space than NMF would.

The phase matrix contains values ranging from $-\pi$ to π. These values are uniformly quantized into 8 levels ($n_p = 3$ bits) as per Section 2.2.3.1. Due to the relative frequency of the boundary values $-\pi$ and π, a mid-tread quantizer is used. The frequency of each quantization level is visible in Figure 5.5.

Using these frequencies, we are able to then construct a Huffman coding table (refer to Section 5.3.3.1) and use it to losslessly encode the quantized phase matrix using at most 4 bits per value.

Figure 5.5: Frequency of each quantization level in the phase spectrogram. Data taken from the average of all the example audio files.



The magnitude spectrogram is where we actually use NMF. As it is effectively a matrix of signal amplitudes, it's less prone to noticeable quality loss if the values deviate a little. We split the magnitude matrix column-wise into smaller submatrices, e.g. $chunk_size = 500$.

For each submatrix we then increment it by its lowest value to get rid of negative values and then run basic random-initialized Euclidean-based NMF, obtaining the weight matrix W and coefficient matrix H.

For the next step, we apply μ-law compression to both the decomposition matrices, as defined in Section 2.2.3.2. In order to be able to do that, we first scale both matrices to the range $[0, 1]$. We do this by applying the following formula to all elements in both matrices:

$$M'_{ij} = \frac{M_{ij} - \min(M)}{|\max(M) - \min(M)|} \tag{5.3}$$

To restore the original scale later, we need to save both the minimum and maximum value, otherwise they will be lost in the non-uniform quantization.

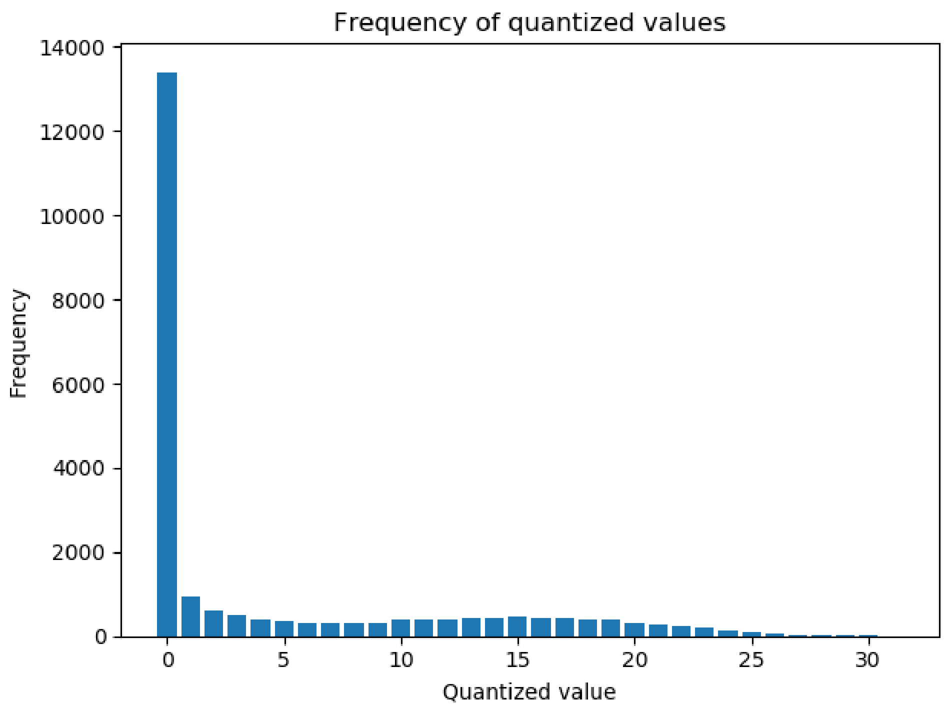
For the μ-law compression, we experiment with the value of μ later, but a good start is $\mu_W = 10^4$ and $\mu_H = 10^5$. We do this because we want to lower the amount of bits needed to represent each value, without losing the values near zero - that would lead to a large loss of quality, or in some cases, loss of the signal itself.

In the weight matrix, we then convert the compressed values into 32-bit unsigned integers, as going to values below that proved to cause loss of signal data, and this integer matrix is then serialized into the file.

However, in the case of the coefficient matrix H, we can go a lot further. We use a uniform quantizer with 32 levels ($n_h = 5$ bits) and we opt for a mid-tread quantizer again, as we want to be able to reconstruct a value of 0, i.e. feature not present.

Similarly to the phase matrix, we take these quantized values and look at their frequencies. The results of this analysis are visible on Figure 5.6.

Figure 5.6: Frequency of each quantization level in the decomposed coefficients matrix of the magnitude spectrogram. Data taken from the average of all the example audio files.



Using this information, we can construct another Huffman coding table, separate for this matrix. As we can see, even despite the μ-law compression, a lot of values still fall to a quantized value of 0.

In a Huffman table from these frequencies, we then assemble the shortest code to the value 0, i.e. we only need 1 bit. The longest code in a table constructed this way for 32 levels of quantization will only require 11 bits. Overall, we save a lot of space saving the matrix this way as opposed to storing the values as 32-bit floats or integers.

Table 5.8: ANMF-STFT data structure

Bytes (relative)	Data type	Description
0-3	uint32	amount of zeroes used to pad the samples
4-7	uint32	amount of STFT submatrix chunks
8-?	quant_matrix	quantized phase matrix
?-?	data_chunk[]	NMF-compressed magnitude chunks (refer to Table 5.9)

Table 5.9: ANMF-STFT structure of each magnitude submatrix

Bytes (relative)	Data type	Description
0-7	float64	absolute value that the matrix was incremented by
8-15	float64	minimum value before scaling to $[0, 1]$
16-23	float64	maximum value before scaling to $[0, 1]$
24-?	matrix(uint32)	μ-law compressed matrix W
?-?	quant_matrix	quantized μ-law compressed matrix H

5.3.3.1 Huffman encoding

Huffman code refers to an optimal prefix coding scheme often employed for lossless compression. It's used to compress a message that only consists of members from a finite, known beforehand set of symbols. [43]

The goal is to create a dictionary that maps each value to a sequence of bits, where none of the sequences is a prefix of another one, which means that there are no ambiguities when decoding a Huffman encoded message, and we do not need to store any information about where one code begins and where it ends.

The main characteristic of a Huffman code is that the more frequent a symbol is, the shorter code it will be assigned. So in the case of a system where a certain symbol is very frequent, a message consisting of mostly those symbols will be compressed greatly, with no loss of data.

Algorithm 1 describes the compression process, at the end of which we obtain a Huffman tree. By traversing this tree from the root and assigning every left child a 0 and every right child a 1 and concatenating these bits as we reach the symbols in the leafs, we obtain the Huffman code for the given symbol. The asymptotic complexity of this algorithm is $\mathcal{O}(n \log n)$, essentially

```
Input: a list of symbols and their probabilities
Output: a Huffman tree
queue ← new PriorityQueue();
foreach item x in input do
    node ← new Node();
    node.symbol ← x.symbol;
    node.prob ← x.probability;
    node.leftChild, node.rightChild ← null;
    queue.enqueue(node, node.prob);
end
while queue.length > 1 do
    node1 ← queue.dequeue();
    node2 ← queue.dequeue();
    newNode ← new Node();
    newNode.leftChild ← node1;
    newNode.rightChild ← node2;
    newNode.prob ← node1.prob + node2.prob;
    queue.enqueue(newNode, newNode.prob);
end
```

Algorithm 1: Huffman code compression

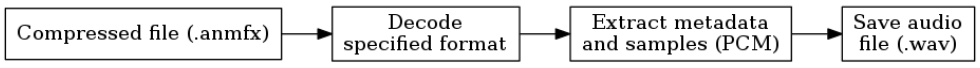
we need $\mathcal{O}(\log n)$ time to determine the highest priority in the queue, and there are $\mathcal{O}(n)$ iterations.

Often, the dictionary itself has to be encoded as well, but as the frequencies are roughly the same for each audio file, the tree can be fixed directly into the implementation.

5.4 Decoder

Similar to the encoder, the decoder simply reverses the encoding process as seen in Figure 5.7. As this process is fairly straightforward for each of the methods, it won't be elaborated on further.

Figure 5.7: A high level overview of the ANMF audio decoder.



CHAPTER 6

Implementation

This chapter explains the specific details of implementing the audio codec outlined in the previous chapter.

The codec was implemented entirely using *Python 3* (x64) [44] and the source code is available both as part of this thesis and on my personal GitHub profile [link].

The codec could be roughly split into three parts:

Command line utility the command line interface (CLI), interacts with the user

Audio library a set of classes and functions responsible for reading/writing audio files

Compression library a set of classes and functions responsible for encoding and decoding ANMF files

6.1 Used 3rd party libraries

The codec uses several libraries for things where a tried and tested method is both faster and more reliable than implementing my own. A brief description of each of them follows.

6.1.1 click

The `Click` [45] library lets you create command line interfaces in a descriptive manner without having to manually read user input. This helps prevent errors regarding unexpected parameters and such. It also automatically generates the help message describing the various options and arguments, all of which makes it suitable for a project like this.

6.1.2 pytest

`pytest` [46] is a framework for writing and executing tests in Python. To ensure certain functions work correctly and give expected results, several unit tests have been written.

6.1.3 dahuffman

To create Huffman trees and use Huffman coding, a library called `dahuffman` [47] was utilized.

It determines the frequencies either from a dictionary containing $value \rightarrow frequency$ pairs, or it infers them automatically from existing data (as per Section 5.3.3.1). Once the frequencies are determined, the library is capable of losslessly encoding and decoding arbitrary data containing the symbols from the Huffman tree into a serializable byte array.

If needed, the Huffman table including the exact code for each symbol can be printed to validate it works correctly.

A wrapper class was made around this library called `HuffmanCoder`, to facilitate easier usage and to be able to easily separate different coders using varying dictionaries. It has convenience methods `encode_int_matrix` and `decode_int_matrix`.

When encoding using the encoding method, it flattens the matrix into an array, then `dahuffman` encodes the array using the associated Huffman table, and finally the function returns both the bytes and the amount of rows in the original matrix.

Decoding does the reverse - it takes these bytes and the amount of rows, decodes the bytes into an array of values, and then re-shapes the array into the original matrix.

6.1.4 SciPy

`SciPy` [48] is short for *Scientific Python* and as the name suggests, this library provides the bulk of the mathematical support for the implementation.

`SciPy` as a whole is more of a stack, or software ecosystem, containing various sub-libraries (covered below). However, the `SciPy` library itself provides some algorithms which are used as described later.

6.1.4.1 NumPy

`NumPy` [49] is part of the `SciPy` stack and its most important feature for us is that it provides us with an easy way to create and manipulate N-dimensional array objects. In our case, for the most part it's 2D matrices.

It has a friendly syntax and all the core functions for creating matrices, mapping functions over them, multiplying them etc., all of which are very important. It also supports complex numbers and operations on them seamlessly.

Even though it is a Python library, a lot of its performance critical parts are written in lower level languages such as C and are optimized for complex calculations.

`NumPy` is heavily utilized in this work, as the problem of NMF mostly concerns 2D matrices and `NumPy` is ideal for working with those.

6.1.4.2 Matplotlib

`Matplotlib` [50] is also part of the `SciPy` stack and it's best described as a Python plotting library, capable of producing various (primarily 2D) plots and diagrams from data.

As it's closely integrated with `SciPy`, it can natively plot data structures created by e.g. `NumPy` without any boilerplate code necessary.

`Matplotlib` is responsible for most of the diagrams visible in this thesis.

6.2 Command line utility

The command line part of the program (and the entry point) was created using the `Click` library.

The CLI lets you specify which kind of compression you want to use, between ANMF-RAW, ANMF-MDCT and ANMF-STFT, defaulting to ANMF-STFT. When decompressing, it automatically determines the compression type from the file's extension.

The bulk of the work of the command line utility is to pass the input file's handle to the audio library and obtain a reference to the parsed audio data. This data is then passed to the compression library along with a reference to the output file handle, and once the target file is compressed or decompressed, the program returns.

6.3 Audio library

The audio library is capable of reading samples from audio files, or creating audio files from samples, in both cases the only supported format right now is 16-bit signed PCM WAV.

To facilitate reading and writing WAV files, I use the methods contained in the `scipy.io.wavfile` namespace, specifically `scipy.io.wavfile.read` and `scipy.io.wavfile.write`.

When a WAV file is read, its raw samples and metadata are stored in an internal class called `AudioData`. This class can then either call the compression library to save it as a compressed ANMF file, or it may call into `SciPy` again to save it into a WAV file.

6.4 Compression library

The compression library is a collection of three classes:

NMFCompressorRaw class responsible for compressing and decompressing ANMF-RAW data

NMFCompressorMDCT class responsible for compressing and decompressing ANMF-MDCT data

NMFCompressorSTFT class responsible for compressing and decompressing ANMF-STFT data

Each of those has two functions: `compress(audio_data, file_handle)` and `decompress(file_handle, audio_data)`. For compression, it reads the audio data and writes it to the target file. For decompression, it reads the raw data from the file and creates a new `AudioData` object, updating the reference with the decompressed data.

The encoding process has already been described in Section 5.3, so I will only include specific implementation details here. For the decoding process, just as before, it's doing the same, but inverse operations in the same order, so it will not be elaborated on in detail.

Unless mentioned otherwise, all matrix operations are implemented using `NumPy` and its associated data structures and functions. Writing data to file is done using the `struct.pack` function from the native Python library. During decoding, when the audio data is finalized, it must be converted back to a signed 16-bit integer, so that it can be saved to a WAV file.

6.4.1 Utility functions

To avoid code repetition, a lot of functionality is shared between the various methods using different helper functions. The most important ones are described below.

6.4.1.1 Matrix (de)serialization

There are two core functions for matrix serialization and deserialization to/from a file:

- `serialize_matrix(fd, matrix, dtype='f')`
- `deserialize_matrix(fd, dtype='f')`

For serialization, the input matrix is taken and converted to the target datatype, by default using a 32-bit float, but this can be changed by the caller0. Then, two 32-bit unsigned integers are written representing the amount of rows and columns in the matrix. After that, the matrix is flattened row-wise and each value is written to the file individually.

Deserialization works on the same principle. First we need to read the amount of rows and columns, and then read $rows \times columns \times sizeof(datatype)$ bytes to obtain a data array, which is then re-shaped to the original matrix and returned.

6.4.1.2 Array padding

If we want to run some transformation on an array of values, we may need the array's length to be divisible by a certain number, to be able to split it into equal parts.

There is a utility function for this with the signature `array_pad(array, n)`, where *array* is the array we want to pad and `n` is the number whose multiple the array's length should be.

We first find the necessary padding by calculating $padding = -length(array) \mod n$ and then add that amount of zeroes to the end of the array, returning both the new array and the amount of padding. The padded zeroes need to be removed again during decoding, otherwise we'll end up with a different signal than the original.

There is a variant of this function for convenience called `array_pad_split(array, n)`, which does the same as the former, but also splits the array into n-sized chunks, returning a list of the new subarrays and the padding at the end.

6.4.1.3 Matrix splitting

Similarly to the previous functions, there are times where have a matrix and we want to split it into smaller matrices. There is a function for this called `matrix_split(matrix, n)`.

It takes an input matrix $x \times y$, and splits it into $\lfloor \frac{x}{n} \rfloor$ submatrices, each sized $n \times y$. There is no padding necessary in this case.

6.4.1.4 Range scaling

There are a few instances where a value (or more often, an entire matrix of values) needs to be scaled within a certain range, e.g. for μ-law companding.

The design part already describes a simple equation to scale a range of values to the $[0, 1]$ range (Equation 5.3), but the actual implementation allows scaling from an arbitrary range to an arbitrary range. This function is called is `scale_val` and you can see its code in Listing 6.1.

Listing 6.1: Function for scaling value ranges

```
def scale_val(x, old_min, old_max, new_min, new_max):
        old_range = abs(old_max - old_min)
        new_range = abs(new_max - new_min)
```

```
    val = (((x - old_min) / old_range) * new_range) + new_min
    return val
```

6.4.1.5 Uniform quantization

In ANMF-STFT, it's often needed to uniformly quantize a range of values into N levels. To make this easier, a special class `UniformQuantizer` was implemented. It has three parameters:

min_val the minimum value in the range

max_val the maximum value in the range

levels the desired amount of quantization levels

Internally, it uses a mid-tread quantizer as defined in Equation 2.2. Rather than returning the quantized value itself, it returns an index in the range $[0, levels - 1]$.

This means, that given a `UniformQuantizer` and an index of a quantized value, it can trivially determine the actual quantized value x as:

$$x = min\_val + index \cdot step \tag{6.1}$$

Where step is defined as:

$$step = \frac{max\_val - min\_val}{levels - 1} \tag{6.2}$$

It has two methods: `quantize_value(value)` and `dequantize_index(index)`, depending on what you want to do.

6.4.1.6 NMF helpers

This set of functions helps use NMF by doing any processing necessary both before and after and serves as a sort of interface between the codec and NMF. It includes three functions:

- `nmf_matrix(matrix, max_iter, rank)`
- `nmf_matrix_original(W, H, min_val)`
- `increment_by_min(matrix)`

`increment_by_min` takes an input matrix, finds $min\_val = |\min(matrix)|$, and increments every element in the matrix by min_val, ensuring that the matrix does not contain any negative values.

`nmf_matrix` takes an input matrix, applies `increment_by_min` on it and then calls NMF, obtaining the weight and coefficient matrices. It then returns both of those matrices along with the minimum value it was initially incremented by.

`nmf_matrix_original` takes the decomposition matrices W and H along with the minimum value min_val, multiplies the two matrices and subtracts min_val from each element of the new matrix, to ensure the original range of values is preserved. It then returns the final approximated matrix.

6.4.2 NMF implementation

Basic NMF has been implemented in a class called `NMF`. It's a basic implementation for approximating a matrix V as a product of two matrices WH.

The class constructor takes six different arguments:

1. input matrix
2. maximum number of iterations
3. approximation rank
4. initialization function
5. cost function
6. update function

There are a few important functions:

- The `validate()` ensures that the input matrix does not contain any negative values, raising an exception otherwise.
- Cost function evaluation is done by the `eval_cost()` function.
- `factorize()` stands at the core of the class as it does the actual factorization, returning the factors W and H. Please refer to Listing 6.2 for the source code.

In its default settings, it uses the Euclidean cost function (Equation 3.3) and Euclidean updates (Equation 3.5), with the initial elements of W and H being randomly initialized from the range $[0, \max(V)]$.

Listing 6.2: Python code for the factorize function

```
def factorize(self):
        self.initialize(self)
        last_cost = None
        for i in range(self.max_iter):
                self.update(self)
                cost = self.eval_cost(self)
                if cost == last_cost:
                break
        last_cost = cost
        return self.W, self.H
```

6.4.3 ANMF-RAW

In my implementation of the ANMF-RAW method, there are three parameters that can be changed:

CHUNK_SHAPE a 2-tuple containing the number of rows and columns that specifies the shape of each chunk before NMF is applied

NMF_MAX_ITER maximum number of iterations of NMF before the algorithm is stopped

NMF_RANK target rank of the NMF approximation

If `CHUNK_SHAPE` is set to `None`, the entire array will be turned into a square matrix of dimensions $\lceil\sqrt{N}\rceil \times \lceil\sqrt{N}\rceil$, where N is the amount of samples in the audio signal for the given channel. However, this generally leads to a huge loss in quality and it is therefore recommended to always specify a shape tuple.

I use the utility function `array_pad_split` to first split the samples into equal parts of size corresponding to the shape of the desired matrix. Then, each of these parts is re-shaped into the proper shape using `numpy.reshape`, and the values are converted from signed 16-bit integer to a signed 32-bit integer to be able to increment the values in it. The amount of chunks along with the extra padding at the end is then written to the output.

For each submatrix from the original sample array, we apply NMF using the utility function `nmf_matrix` to calculate the NMF, and then write the minimum value to the file along with the serialized matrices W and H.

6.4.4 ANMF-MDCT

ANMF-MDCT uses the following parameters:

FRAME_SIZE how many samples a frame should contain, determines frequency resolution, must be an even number

NMF_CHUNK_SIZE number that specifies the number of frames in each submatrix before NMF is applied

NMF_MAX_ITER maximum number of iterations of NMF before the algorithm is stopped

NMF_RANK target rank of the NMF approximation

As no easy to use library was available, I had to implement MDCT on my own.

The signal needs to first be padded to block size using `array_pad`. Before we can apply MDCT, the signal is windowed using the MLT window, i.e. in every frame of size $N = FRAME\_SIZE$, we apply Equation 2.8 to every sample.

In the implementation, there are two different algorithms for calculating the MDCT of the input signal. One is called `mdct_slow` and the other `mdct_fast`.

The slow variant is a direct implementation of Equation 2.16 with the inverse following Equation 2.17. Unfortunately, as the name suggests, it is rather slow for use in production and only served as a reference for unit testing the fast algorithm.

The fast variant instead relies on two things:

1. MDCT can be directly derived from a Discrete cosine transform (DCT) [16]
2. DCT can be calculated quickly with the help of fast Fourier transform (FFT) [51]

Specifically, MDCT can be derived from a type 4 DCT (DCT-IV). We take each block of $FRAME\_SIZE$ samples and split it into four equal parts (a, b, c, d). Then, due to the boundary conditions, the MDCT of this block is equal to a DCT-IV of the inputs $(-c_R - d, a - b_R)$, where x_R refers to x in reverse order. To calculate the DCT-IV of the resulting array, we use the function `scipy.fftpack.dct(input, type=4)`. We normalize the resulting array by dividing every element in it by 2, making its output equal to the slow algorithm. The function finishes by returning the MDCT coefficient matrix along with the length of padding at the end of the array.

Once we have our MDCT, the resulting MDCT matrix is split using `matrix_split(mdct, NMF_CHUNK_SIZE)`. We write the length of the padding followed by the amount of the split submatrices as 32-bit unsigned integers.

Then, for each MDCT submatrix, we apply the utility function `nmf_matrix`, write the minimum value and then serialize both the matrices.

The decoding process is fairly straightforward other than inverting the MDCT, so that will be described next.

If we want to get the inverse MDCT (IMDCT), it gets a bit tricky. We first run inverse DCT-IV by using `scipy.fftpack.idct(input, type=4)`, giving us back the array in the form of $(-c_R - d, a - b_R)$, which we need to turn back into (a, b, c, d). By cleverly inverting, reversing and concatenating these subarrays, we are able to obtain an array in the form of $(a-b_R, b-a_R, c+d_R, d+c_R)$ (notice that some of the values are redundant). We apply the window again and multiply all the elements by two (to cancel out the division in the MDCT) and obtain nearly the original array. As the final step, we need to cancel out the "extra" values such as $-b_R$. To do that, we take two adjacent overlapping blocks and add them together, leaving us with the original array (a, b, c, d).

6.4.5 ANMF-STFT

ANMF-STFT uses the same parameters as ANMF-MDCT and some additional ones:

FRAME_SIZE how many samples a frame should contain, determines frequency resolution, must be an even number

NMF_CHUNK_SIZE number that specifies the number of frames in each submatrix before NMF is applied

NMF_MAX_ITER maximum number of iterations of NMF before the algorithm is stopped

NMF_RANK target rank of the NMF approximation

MU_LAW_W parameter μ for μ-law companding of the weight matrix W

MU_LAW_H parameter μ for μ-law companding of the coefficient matrix H

`SciPy` contains a function for the entire STFT calculation, called `scipy.signal.stft`. It handles windowing, overlapping and padding by itself, so it is very handy.

At its core, it uses the fast Fourier transform [52]. We choose an overlap of 50% by putting the number of data points per segment to $FRAME\_SIZE$ and the number of overlapping points to $\frac{FRAME\_SIZE}{2}$. For the window function, we choose the Hann window (Section 2.2.4.2).

Before further processing, the output STFT matrix is transposed (for the shape to be consistent with the matrix from MDCT), and then we find the phase matrix and the magnitude matrix by calling `numpy.angle` and `numpy.absolute`, respectively, on every element of the matrix.

The magnitude matrix is split into chunks of size NMF_CHUNK_SIZE using `matrix_split`, while the phase matrix is quantized using a `UniformQuantizer` in range $[-\pi, \pi]$ with 8 quantization levels and this matrix of quantized values is then Huffman encoded.

Several things are then written to the file in order: the amount of magnitude submatrices (uint32), number of rows in the phase matrix (uint32), length of the Huffman encoded phase matrix (uint32) and the Huffman encoded phase matrix bytes themselves.

The magnitude submatrices are then processed in order. First we use basic NMF via `nmf_matrix`, then scale both the decomposition matrices to the $[0, 1]$ range using `scale_val`, and then μ-law compress both using their respective $MU\_LAW\_[W|H]$ parameter as per Equation 2.4.

For matrix W, we use $scale_val$ to scale it to range $[0, 2^{32}]$ and change its datatype to unsigned 32-bit integers.

In the case of matrix H, it is uniformly quantized in the range $[0, 1]$ with 32 quantization levels, and then similar to the phase matrix, it's Huffman encoded using its associated Huffman table.

Then, for each submatrix, in order, we write to the file: the value we need to subtract after multiplying WH (float64), the original minimum and maximum of the matrix before it was scaled to $[0, 1]$ (float64), the scaled 32-bit integer matrix W using `serialize_matrix`, and finally the encoded matrix H in the same manner as the phase matrix.

When decoding, the process is again the same but in reverse order. To invert the STFT, we call `scipy.signal.istft` using the same parameters as before. It estimates the original signal by using the algorithm described in [53]. For this algorithm to work properly and attain perfect reconstruction of the original signal, the COLA constraint (Constant OverLap Add) [54] must be met. Using a Hann window at 50% overlap guarantees the satisfaction of this constraint.

Chapter 7

Evaluation

In this section you will find an evaluation of the different algorithms, including experiments with various parameters for them and comparison to existing solutions from Section 4.2.

7.1 Methodology

Audio compression quality is most often evaluated using a series of listening tests (such as the one pictured in Figure 4.2). However, due to a lack of resources to conduct such a thing, a different method must be used - although they are generally not as accurate.

One of the most used methods for objectively evaluating perceptible audio quality is PEAQ [41], however its specifications are insufficient, and its implementations proprietary. The exact algorithm and parameters aren't known for the most part. It comes in two variants:

PEAQ Basic intended for real-time use

PEAQ Advanced a more comprehensive model, intended for non real-time use

Even though neither of these is directly available, luckily, there are a few open-source alternatives that try to implement similar algorithms, with their quality measured by comparing their output to PEAQ, and by proxy, to listening tests.

One of the more prominent open-source solutions is GstPEAQ, which according to its paper [55] performs better than the other implementations. So while it does not conform to the PEAQ recommendation directly, its results are within an acceptable margin and thus this thesis will use GstPEAQ for quality evaluation.

7.2 GstPEAQ

GstPEAQ is a plugin for GStreamer [56] (a pipeline-based multimedia framework) and its source code is freely available at [57]. It implements both the Basic and Advanced mode of PEAQ as specified in [41], however as the standard is under-specified, educated guesses must be taken at points.

It's also important to mention that I could not get the Advanced mode to work without errors, and as such testing will be performed in Basic mode only.

And just like PEAQ, the algorithm's main output is a value known as *Objective Difference Grade* (ODG), which evaluates the perceptible impairment (quality difference) between the provided audio and the reference audio. It uses various psychoacoustic "features" of the signal to determine the grade, the details of which won't be covered here - please refer to either paper for specifics.

The ODG scale contains real values from 0 to -4, ranging from imperceptible difference to very annoying for the human ear. Please refer to Table 7.1 to see the full scale.

Table 7.1: PEAQ - Objective Difference Grade table

ODG	Impairment description
0.0	Imperceptible
-1.0	Perceptible, but not annoying
-2.0	Slightly annoying
-3.0	Annoying
-4.0	Very annoying

7.3 Evaluating results

In this section we will experiment with various parameters for all the encoding methods, find a good compromise between bitrate and ODG, and then compare those results to the reference codings (MP3, Opus) at similar bitrates.

The evaluating process will work in these steps:

1. compress example WAV file using ANMF
2. decompress ANMF back to a WAV file
3. measure ODG between old and new WAV file

In the case of MP3 or Opus files, they will be decoded to WAV for comparison.

7.3.1 Audio examples

There's a total of four example audio files in WAV format that will be used for testing. Please refer to Table 7.2 for a list. All of the examples are $\sim 5-10$ second excerpts from audio files using 44.1 kHz sampling rate and 16-bit signed integer samples. These files can be found in the `examples` folder of the implementation.

Table 7.2: List of tested audio files

ID	File name	Description
00	`piano16t.wav`	Clear sounding piano sounds
01	`henry16t.wav`	Average quality English voice
02	`swave16t.wav`	Simple electronic music
03	`taleena16t.wav`	Complex music including lyrics

7.3.2 Bitrate

The bitrate is the amount of bits that a computer needs to process per a unit of time, in this context it means how many bits are needed to play back 1 second of an audio file.

To find the bitrate of a WAV file with a sampling rate F_s, we can use the following equation:

$$bitrate = channelCount \cdot F_s \cdot bitsPerSample \tag{7.1}$$

So in regards to compressing the example files, our target bitrate is at most $2 \cdot 44100 \cdot 16 = 1411200$ bits per second (or ~ 1411 kbps).

To calculate the number of elements in an NMF-approximated matrix V of size $m \times n$ with approximation rank r, we can use the following equation:

$$size(NMF(V)) = r(m+n) \tag{7.2}$$

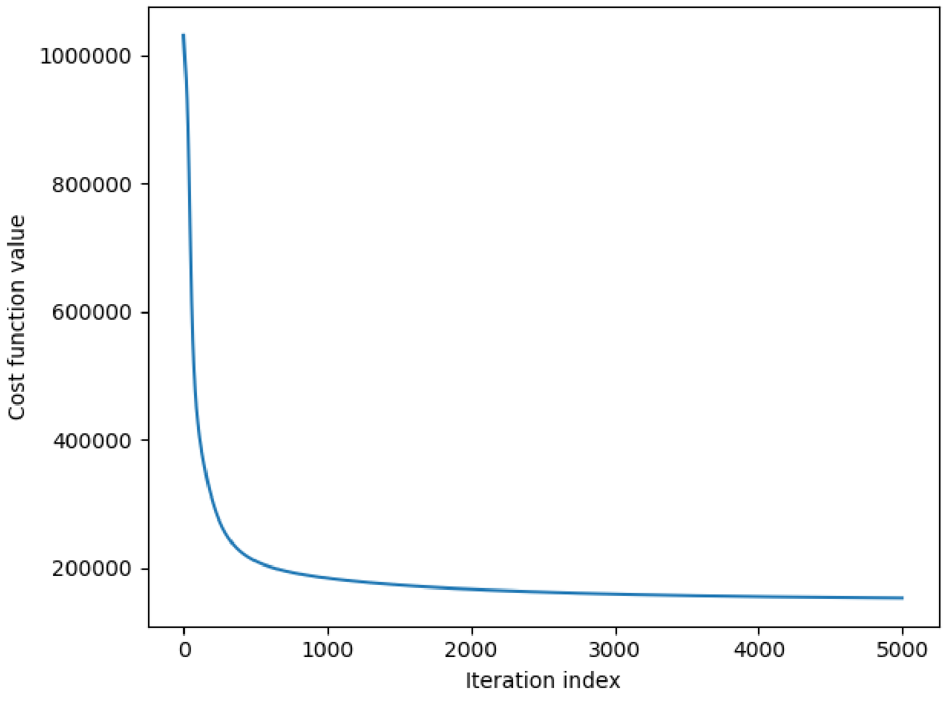
Proof: NMF approximates a matrix $m \times n$ into two matrices $m \times r$ and $r \times n$, therefore the total size is $mr + rn = r(m+n)$.

7.3.3 ANMF-RAW

For ANMF-RAW, we need to find a compromise between the size of one chunk and the rank of the NMF. First, we'll fix the shape to 1152×200 and experiment with the rank.

The maximum number of iterations has been fixed to 3000, since past that point the cost function is almost stable as seen in Figure 7.1.

Figure 7.1: The value of the cost function per iteration during NMF in ANMF-RAW at rank = 30.



In Table 7.3, you can see the results of changing the rank of the NMF approximation. The rows correspond to the example IDs from Table 7.2, and the columns correspond to the ODG for different rank values.

Table 7.3: ANMF-RAW rank experiment results

ID	$r = 30$	$r = 40$	$r = 45$	$r = 60$	$r = 75$
00	-3.873	-3.830	-3.811	-3.749	-3.650
01	-3.908	-3.902	-3.895	-3.861	-3.844
02	-3.868	-3.850	-3.837	-3.800	-3.759
03	-3.744	-3.519	-3.361	-2.825	-2.461

It's also worth mentioning that the runtime necessary for compression scales with the rank, as seen in Table 7.4.

Table 7.4: ANMF-RAW average runtime in seconds

ID	rank 30	rank 40	rank 45	rank 60	rank 75
00	63.378s	65.913s	67.082s	71.684s	75.839s

Interestingly enough, sample ID 03 notices the largest improvement despite being the more complex of the four, but when listening to the decompressed audio, it still sounds quite distorted and noticeable. Another looming issue is that at that rank the size of the compressed file approaches the original filesize, so we will need to lower the rank and try a lower chunk size instead.

To calculate the bitrate of ANMF-RAW, we must take into account the chunk size and the rank of the NMF approximation and then use Equation 7.2. There is some overhead for the metadata, but as it spans only a few bytes it will be disregarded. The bitrate of ANMF-RAW with a chunk size of $m \times n$ and rank r can be calculated as:

$$bitrate^{RAW} = r(m+n) \cdot \frac{F_s}{mn} \cdot bitsPerValue \cdot channelCount \tag{7.3}$$

Therefore, for a chunk 1152 × 200 with rank 75, we get:

$$bitrate^{RAW} = 75 \cdot (1152 + 200) \cdot \frac{44100}{1152 \cdot 200} \cdot 32 \cdot 2 = 1242150 \tag{7.4}$$

Our target is 1411 kbps and our compression with 1242 kbps is judged as ”annoying” by GstPEAQ. We will try lowering the chunk size and rank both to see if it has an effect. It was mentioned in Section 4.2.1 that it can compress music by up to a factor of 12 without noticeable loss in quality. Here, with this method, we are going to aim for at least around half that.

We take a rank of 40 and a target bitrate of 750 kbps. We pick e.g. 600 for m and solve the equation above, obtaining $n \approx 200$. However, when trying out these new parameters, the ODG barely changed from the previous shape, and further testing of this method was thus stopped due to unsatisfactory results.

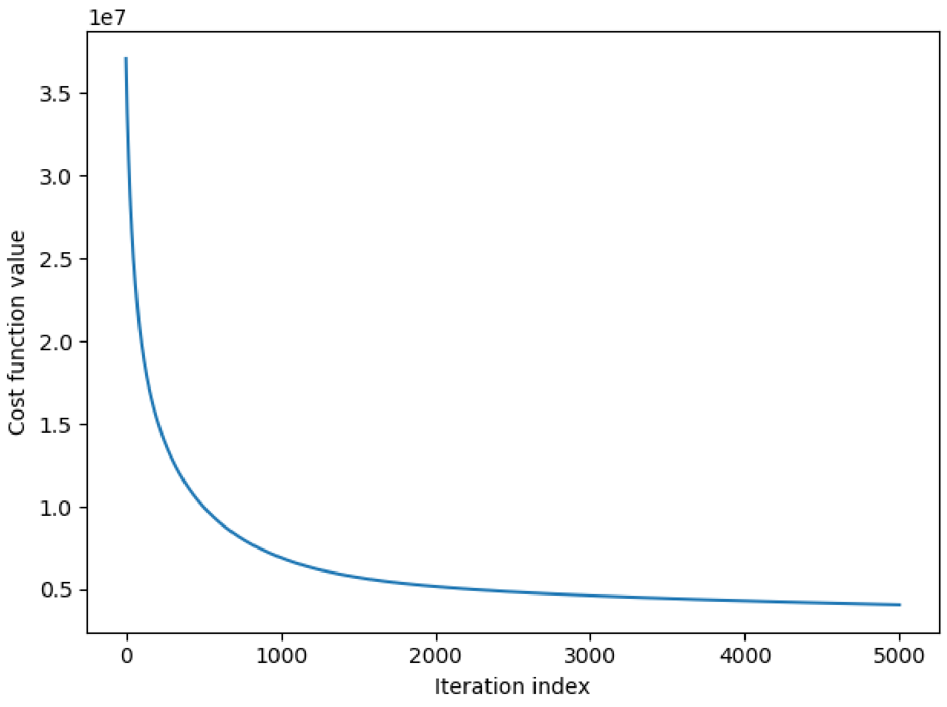
7.3.4 ANMF-MDCT

For MDCT we fix the frame size to 1152 for a good balance between frequency resolution and storage necessary. As seen in Figure 7.2, the approximation converges in the same manner as ANMF-RAW (using a chunk size of 200 and a rank of 40), but the values are much smaller. We will go with 3000 iterations as before, though, to get more accurate results.

Learning a lesson from the ANMF-RAW experiments, it’s worth figuring out whether ANMF-MDCT is even practical at all. To do that, we need to be able to calculate the bitrate of ANMF-MDCT first. Assuming rank r, sampling rate F_s, frame size S_f and chunk size S_c, our bitrate is:

$$bitrate^{MDCT} = r(\frac{S_f}{2} + S_c) \cdot \frac{F_s}{\frac{1}{2}S_f S_c} \cdot bitsPerValue \cdot channelCount \tag{7.5}$$

Figure 7.2: The value of the cost function per iteration during NMF in ANMF-MDCT at rank = 40.



And as before, our initial goal is a target bitrate of 750 kbps. To achieve that, we either use a higher rank and higher chunk size, or vice versa.

First, I tried to fix the rank to a relatively high value, specifically $r = 60$. If we fill the known values into the equation and solve for S_c, we obtain a chunk size of $S_c \approx 371$. Similarly, if we go with a small chunk size, e.g. $S_c = 100$, we obtain a rank of $r \approx 23$.

However, as we can see in Table 7.5, the ODG values are less than usable (and it sounds terrible to my ears as well). This seems to be mostly due to the fact that the phase data contained in the MDCT coefficients is very sensitive to inaccuracies and throws the signal off.

Out of curiosity, I tried increasing the frame size as well, but this only led to the signal being unrecognizable and there doesn't seem to be a good reason to do it in the first place.

Due to the fact that ANMF-MDCT did not produce usable results at a comparatively high bitrate, it will not be tested further.

7.3.5 ANMF-STFT

ANMF-STFT seemed to produce the best results during preliminary testing when I was implementing the codec, so given that and the large amount of

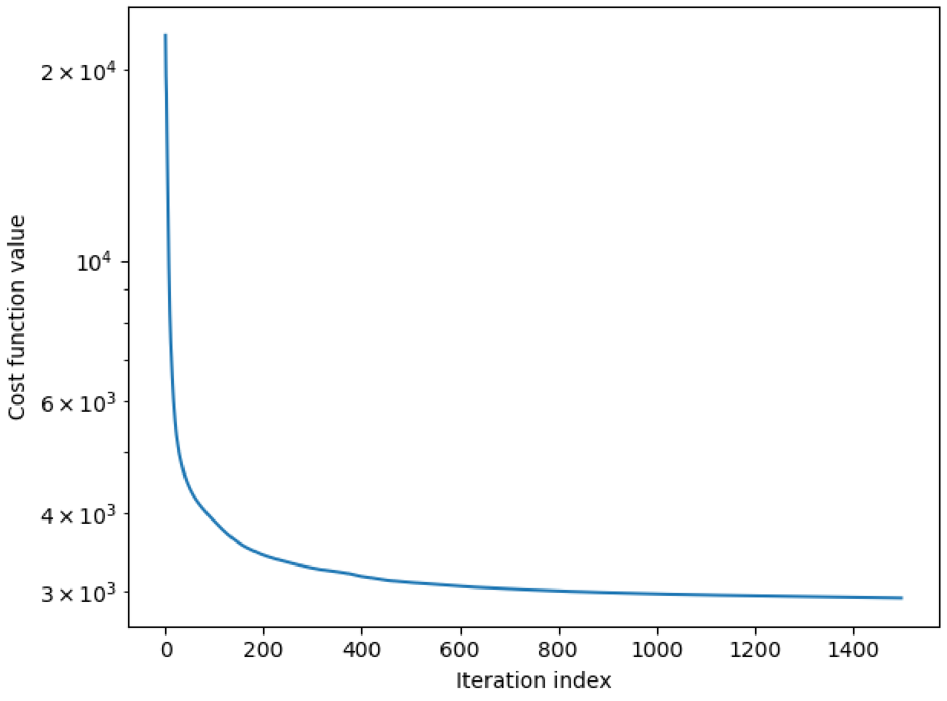
Table 7.5: ANMF-MDCT rank experiment results

ID	$r = 23, S_c = 100$	$r = 60, S_c = 371$
00	-3.910	-3.911
01	-3.913	-3.913
02	-3.909	-3.910
03	-3.878	-3.877

parameters, this method will be tested thoroughly.

For baseline parameters that seemed to work well, we choose frame size $S_f = 1152$, chunk size $S_c = 500$, $\mu_W = 10^4$ and $\mu_H = 10^5$. On Figure 7.3 we notice NMF converges a lot sooner than in the previous methods, so we will set max iterations to only 1000.

Figure 7.3: The value of the cost function per iteration during NMF in ANMF-STFT at rank = 40. Logarithmic scale is used for better visibility.



Calculating the bitrate here is going to be a bit different. Since we use Huffman coding, the bitrate will vary, since each value uses a different amount of bits, but we can still estimate it.

To estimate the "average" bitrate, we will take the Huffman table and interpret it as a random variable, specifically X_P for the phase spectrogram and X_H for the coefficient matrix of the magnitude spectrogram. As its output, we'll use the amount of bits necessary to encode a value. For the probability distribution of these values, we can use the data we obtained before as seen in Figures 5.5 and 5.6. We then find the expected values $E[X_P]$ and $E[X_H]$ for each of them.

The expected value $E[X]$ of a discrete random variable X is simply put an average of the possible values weighted by their respective probabilities. Formally, it is defined as [58]:

$$E[X] = \sum_i x_i p_i \tag{7.6}$$

By applying this model to the Huffman tables, we find that $E[X_P] \approx 3.072$ bits, and $E[X_H] \approx 2.952$ bits. We can then proceed with bitrate calculation.

To calculate the bitrate of ANMF-STFT, it will be easiest to calculate the bits necessary for the phase matrix and the coefficient matrix separately, and then add them together. The equations are:

For the phase matrix:

$$\begin{align} bitrate^{PHASE} &= E[X_P] \cdot \frac{S_f}{2} \cdot \frac{F_s}{\frac{1}{2}S_f} \tag{7.7} \\ &= E[X_P] \cdot F_s \tag{7.8} \end{align}$$

For the magnitude matrix:

$$\begin{align} bits^{W} &= r \cdot S_c \cdot 32 \tag{7.9} \\ bits^{H} &= r \cdot \frac{S_f}{2} \cdot E[X_H] \tag{7.10} \\ bitrate^{MAG} &= \left(bits^{W} + bits^{H}\right) \cdot \frac{F_s}{\frac{1}{2}S_f S_c} \tag{7.11} \\ &= \left(r \cdot S_c \cdot 32 + r \cdot \frac{S_f}{2} \cdot E[X_H]\right) \cdot \frac{F_s}{\frac{1}{2}S_f S_c} \tag{7.12} \\ &= r\left(32 \cdot S_c + \frac{S_f}{2} \cdot E[X_H]\right) \cdot \frac{F_s}{\frac{1}{2}S_f S_c} \tag{7.13} \end{align}$$

And finally:

$$\begin{align} bitrate^{STFT} &= (bitrate^{PHASE} + bitrate^{MAG}) \cdot channelCount \tag{7.14} \\ &= \left[E[X_P] \cdot F_s + r\left(32 \cdot S_c + \frac{S_f}{2} \cdot E[X_H]\right) \cdot \frac{F_s}{\frac{1}{2}S_f S_c}\right] \cdot channelCount \tag{7.15} \\ &= \left(\frac{E[X_H] \cdot r}{S_c} + \frac{64 \cdot r}{S_f} + E[X_P]\right) \cdot F_s \cdot channelCount \tag{7.16} \end{align}$$

Substituting the variables for our reference values, we end up with a bitrate of $\sim$ 487 kbps, which is lower than the previous methods, but at the same time, subjectively speaking, it actually sounds fairly promising.

However, upon closer inspection of the equations an alarming realization sets in. If we calculate the bitrate of the phase spectrogram, we find out it equals $3.072 \cdot 44100 = 135475.2$. This means is that the phase spectrogram alone is taking up roughly 135 kbps (per channel!), which is already more than the average MP3 file, and we are already compressing it with a total of only 3 bits per value. We run into the issue highlighted in [39], that being that there doesn't seem to be a good way to compress only the phase spectrogram. As mentioned earlier, applying NMF to the phase spectrogram does not work and produces barely recognizable sounds.

Regardless, I'm at least going to experiment on what I still can. The next step is to stabilize this bitrate while experimenting with the variables and measuring the quality. Once we find out the most influencing factors and end up with satisfactory audio quality, we can try lowering the bitrate further to try and match existing codecs.

Table 7.6: ANMF-STFT rank experiment results

ID	$r = 30$	$r = 40$	$r = 50$	$r = 60$	$r = 70$
00	-3.529	-3.217	-3.041	-2.838	-2.768
01	-2.140	-1.884	-1.714	-1.658	-1.499
02	-3.816	-3.758	-3.696	-3.640	-3.576
03	-2.491	-1.282	-1.081	-1.033	-0.806

Looking at Table 7.6, we can notice that by increasing the rank, the ODG gets lower. However, we can't increase the rank ad infinitum, since any increase means two things:

1. the required bitrate increases
2. the required number of NMF iterations increases

In short, the higher the rank, the more space it will take while also drastically increasing the needed computation time. Using Equation 7.13 and plotting the function (Figure 7.4), we find that the bitrate grows linearly with the rank. At rank 60, we are already at a large value of $\sim$ 162 kbps.

Next up, I will experiment with the chunk size keeping the other values as reference values.

In Table 7.7, we can see that the smaller the chunk size, the better the quality. This makes sense, since if the chunk is smaller, the input matrix to NMF is also smaller, which makes it easier to converge, but at the same time increases the bitrate. Again, by using Equation 7.13 and plotting it for S_c (Figure 7.5), we can notice a certain trend. Unlike rank, the chunk size

Figure 7.4: Bitrate (kbps) per channel in relation to NMF rank for $S_c = 500$.

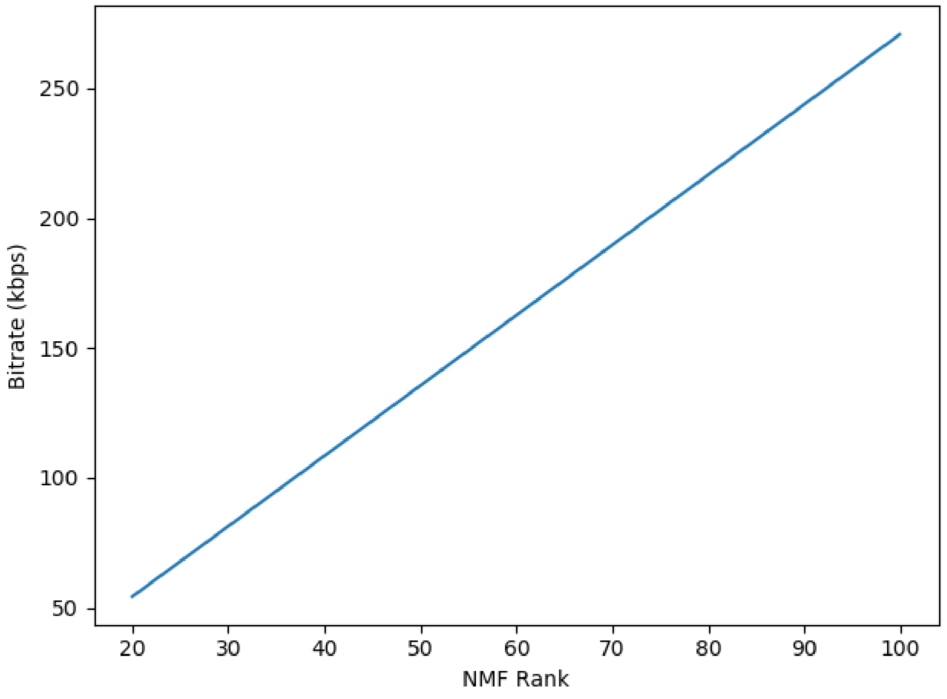


Table 7.7: ANMF-STFT chunk size experiment results

ID	$S_c = 100$	$S_c = 250$	$S_c = 400$	$S_c = 550$	$S_c = 700$
00	-2.598	-3.170	-3.211	-3.272	-3.291
01	-1.149	-1.612	-1.892	-1.925	-1.877
02	-3.495	-3.685	-3.769	-3.758	-3.772
03	-1.303	-1.322	-1.415	-1.681	-2.163

changes the bitrate in an exponential manner, so while lower is better, we need to be careful.

Between the experiments with rank and chunk size, I have decided to go for a compromise. To strike a balance between quality and bitrate, experiments from here on will use rank $r = 50$ and chunk size $S_c = 250$.

Finally, we move on to the last two parameters: μ_W and μ_H. For results, please refer to Tables 7.8 and 7.9.

μ_W and μ_H don't really seem to have too much effect, but it appears that the reference values $\mu_W = 10^4$ and $\mu_H = 5$ were pretty accurate in terms of what seems to give the best results. One thing of note is that mu_H has a rather large jump for audio example ID 03 (electronic music). I believe this is due to the fact that it contains sounds of similar frequencies, which doesn't play along well with μ-law companding for lower values of μ.

Figure 7.5: Bitrate (kbps) per channel in relation to chunk size for $r = 40$.

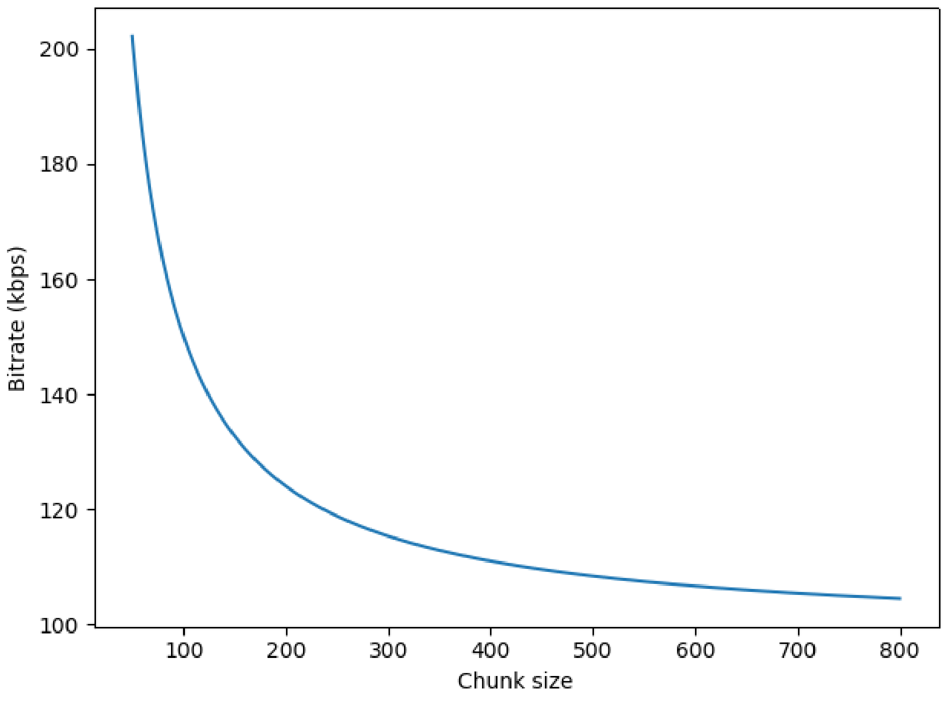


Table 7.8: ANMF-STFT μ_W experiment results

ID	$\mu_W = 10^2$	$\mu_W = 10^3$	$\mu_W = 10^4$	$\mu_W = 10^5$	$\mu_W = 10^6$
00	-2.932	-2.828	-2.862	-2.872	-2.821
01	-1.524	-1.483	-1.538	-1.577	-1.583
02	-3.618	-3.594	-3.532	-3.591	-3.536
03	-1.057	-1.113	-1.039	-0.995	-1.014

In conclusion, our final values are frame size $S_f = 1152$, chunk size $S_c = 250$, 1000 NMF iterations with an approximation rank $r = 50$, and for μ-law companding we have $\mu_W = 10^4$ and $\mu_H = 10^5$. If we plug these values into Equation 7.16, we obtain the following bitrate:

$$bitrate^{STFT} = \left(\frac{2.952 \cdot 50}{250} + \frac{64 \cdot 50}{1152} + 3.072\right) \cdot 44100 \cdot 2 \quad = 568023.68 \tag{7.17}$$

7.3.6 Comparison

Due to the low quality of ANMF-RAW and ANMF-MDCT, they will be omitted from comparison as it's not deemed necessary.

Table 7.9: ANMF-STFT μ_H experiment results

ID	$\mu_H = 10^2$	$\mu_H = 10^3$	$\mu_H = 10^4$	$\mu_H = 10^5$	$\mu_H = 10^6$
00	-2.946	-2.863	-2.833	-2.813	-2.883
01	-2.730	-2.279	-1.573	-1.474	-1.545
02	-3.405	-3.617	-3.610	-3.566	-3.591
03	-0.864	-0.950	-1.077	-0.840	-1.161

Our best lossy audio compression, ANMF-STFT, has reached a bitrate of $\sim$ 568 kbps, which is about four times the bitrate of the average MP3 file, and while the quality is ”good enough”, there are noticeable issues with it. Furthermore, experiments have shown that this form of compression does not seem to particularly excel at any specific kind of audio, though from my personal listening it was usable for speech (ID 00) and clear piano sounds (ID 01). While the ODG for music (ID 04) was not too high, the difference was noticeable enough to convince a user to turn to another format.

For comparison, we will use MP3 with a target bitrate of 320 kbps, and for Opus the target bitrate will be 256 kbps. For both of these, those bitrates are more or less the highest practically used values and they were picked to try and somewhat match the bitrate of ANMF-STFT.

To encode and decode the WAV files into and from the respective formats, certain tools were used. For MP3, I used `LAME` [59], and for Opus, I used `opus-tools` [60]. Afterwards, the ODG was measured using GstPEAQ. Please refer to Table 7.10 for the results.

Table 7.10: ANMF-STFT versus MP3/Opus

ID	320 kbps MP3	256 kbps Opus
00	-0.025	-0.088
01	-0.052	0
02	-0.044	0
03	0	0

As you can see, there is no contest. Both of the reference codecs perform much better than ANMF-STFT at a far lower bitrate.

Chapter **8**

Conclusion

The point of this thesis was to design and implement an audio codec with the ultimate goal of audio compression. While this goal was technically met, it leaves much to be desired.

In total, three different methods were designed and implemented:

ANMF-RAW compresses audio using NMF in the time domain directly

ANMF-MDCT compresses audio using NMF in the frequency domain via modified discrete cosine transform

ANMF-STFT compresses audio using NMF in the frequency domain via short-time Fourier transform

Out of these three methods, only ANMF-STFT ended up being usable in practice. However, similarly to the research it was inspired by, I ended up having a large portion of data in the form of a STFT phase matrix that wasn't feasibly compressible given current knowledge of this particular topic.

In the case of both ANMF-RAW and ANMF-MDCT, the data is far too fragile to be approximated without carefully crafted constraints, in which case a proper quantization (similar to MP3 and Opus) will yield far more reliable results than an approximation, i.e. NMF.

The main output of this work is therefore a command line tool capable of compressing arbitrary audio in WAV format to a third of its size, but it is still no match for contemporary codecs such as MP3 or Opus, which are capable of reducing the bitrate by a factor of up to twelve, with far less noticeable quality loss.

In its current form, it is not applicable to real-time use, but if the file structure is altered somewhat, it's definitely possible. It would require the data from each channel to be interweaved similar to a WAV file, so that it can be read sequentially as a stream of frames. Decompression is not a problem, as that boils down to simple matrix multiplication for the most part, which is not an expensive process given today's technology.

One thing that wasn't explored in-depth in terms of implementation was the application of psychoacoustics, to further trim ”unnecessary” sounds contained in the signal. While this is most likely feasible and would save some bytes, there is already a noticeable quality loss even when trying to preserve as much information as possible, and as such application of psychoacoustics would not make this codec any more competitive to its counterparts, which are able to compress audio without any apparent distortion.

Considering the low amount of works exploring this particular topic, I believe this thesis will serve as a decent reference for when somebody else may stumble across the same problem. It's also worth mentioning that this might be the first public implementation of this kind, and as such it might help the understanding of the methodology behind the process and possibly lead to improvement in the future.

Bibliography

[1] You, Y. *Audio Coding Theory and Applications*. Springer US, 2010.

[2] Bosi, M.; Goldberg, R. E. *Introduction to digital audio coding and standards*. Kluwer Academic Publishers, 2003.

[3] Marks Robert J., I. *Introduction to Shannon Sampling and Interpolation Theory*. 01 1991, ISBN 0387973915, doi:10.1007/978-1-4613-9708-3.

[4] Gersho, A. Quantization. *IEEE Communications Society Magazine*, volume 15, no. 5, Sep. 1977: pp. 16–16, ISSN 0148-9615, doi:10.1109/MCOM.1977.1089500.

[5] Brokish, C. W.; Lewis, M. A-Law and mu-Law Companding Implementations Using the TMS320C54x. Dec 1997. Available from: `http://www.ti.com/lit/an/spra163a/spra163a.pdf`

[6] Waveform Coding Techniques. February 2006. Available from: `https://www.cisco.com/c/en/us/support/docs/voice/h323/8123-waveform-coding.html`

[7] Raissi, R. The Theory Behind Mp3. 2002.

[8] Malvar, H. S. Lapped transforms for efficient transform/subband coding. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 38, no. 6, June 1990: pp. 969–978, ISSN 0096-3518, doi:10.1109/29.56057.

[9] Shatkay, H. The Fourier Transform - A Primer. Technical report, Providence, RI, USA, 1995.

[10] Recoskie, D.; Mann, R. Constrained Nonnegative Matrix Factorization with Applications to Music Transcription. *University of Waterloo Technical Report CS-2014-27*, Dec 2014. Available

from: https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/cs-2014-27.pdf

[11] Heinzel, G.; Rüdiger, A.; et al. Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows. *Max Plank Inst*, volume 12, 01 2002.

[12] Selesnick, I. W. Short-Time Fourier Transform and Its Inverse. Apr 2009. Available from: http://eeweb.poly.edu/iselesni/EL713/STFT/stft_inverse.pdf

[13] Wang, Y.; Vilermo, M. Modified discrete cosine transform - Its implications for audio coding and error concealment. *Advances in Engineering Software - AES*, volume 51, 01 2012.

[14] Malvar, H. S. *Signal Processing with Lapped Transforms*. Norwood, MA, USA: Artech House, Inc., 1992, ISBN 0890064679.

[15] Ahmed, N.; Natarajan, T.; et al. Discrete Cosine Transform. *IEEE Transactions on Computers*, volume C-23, no. 1, Jan 1974: pp. 90–93, ISSN 0018-9340, doi:10.1109/T-C.1974.223784.

[16] Babu, S. P. K.; Subramanian, K. Fast and Efficient Computation of MDCT / IMDCT Algorithms for MP 3 Applications. 2013.

[17] Princen, J.; BRADLEY, A. Analysis/Synthesis filter bank design based on time domain aliasing cancellation. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, volume 34, 11 1986: pp. 1153 – 1161, doi:10.1109/TASSP.1986.1164954.

[18] Princen, J.; Johnson, A.; et al. Subband/Transform coding using filter bank designs based on time domain aliasing cancellation. 05 1987, pp. 2161 – 2164, doi:10.1109/ICASSP.1987.1169405.

[19] Olson, H. *Music, Physics and Engineering*. Dover Books, Dover Publications, 1967, ISBN 9780486217697. Available from: https://books.google.cz/books?id=RUDTFBbb7jAC

[20] Rosen, S.; Howell, P. Signals and Systems for Speech and Hearing. *Acoustical Society of America Journal*, volume 94, 12 1993: p. 163, doi:10.1121/1.407176.

[21] ”smacdon”. Critical Bands in Human Hearing. Oct 2018. Available from: https://community.plm.automation.siemens.com/t5/Testing-Knowledge-Base/Critical-Bands-in-Human-Hearing/ta-p/416798

[22] Fletcher, H. Auditory Patterns. *Rev. Mod. Phys.*, volume 12, Jan 1940: pp. 47–65, doi:10.1103/RevModPhys.12.47. Available from: `https://link.aps.org/doi/10.1103/RevModPhys.12.47`

[23] Gelfand, S. *Hearing: An Introduction to Psychological and Physiological Acoustics*. 01 1990, 187 pp., doi:10.1201/b14858.

[24] Fastl, H.; Zwicker, E. *Psychoacoustics: Facts and Models*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN 3540231595.

[25] Hermes, D. J. The auditory filter. Available from: `http://home.ieis.tue.nl/dhermes/lectures/soundperception/04AuditoryFilter.html`

[26] Behar, A. Intensity and sound pressure level. *Journal of The Acoustical Society of America - J ACOUST SOC AMER*, volume 76, 08 1984, doi:10.1121/1.391117.

[27] Kuwano, S.; Namba, S.; et al. Advantages and Disadvantages of A-weighted Sound Pressure Level in Relation to Subjective Impression of Environmental Noise. *Noise Control Engineering Journal - NOISE CONTR ENG J*, volume 33, 11 1989, doi:10.3397/1.2827748.

[28] Gillis, N. The Why and How of Nonnegative Matrix Factorization. *Regularization, Optimization, Kernels, and Support Vector Machines*, volume 12, 01 2014.

[29] Lee, D.; Seung, H. Algorithms for Non-negative Matrix Factorization. *Adv. Neural Inform. Process. Syst.*, volume 13, 02 2001.

[30] Wang, Y.-X.; Zhang, Y.-J. Nonnegative Matrix Factorization: A Comprehensive Review. *IEEE Transactions on Knowledge and Data Engineering*, volume 25, no. 6, 2013: p. 1336–1353, doi:10.1109/tkde.2012.51.

[31] Lee, D.; Sebastian Seung, H. Learning the Parts of Objects by Non-Negative Matrix Factorization. *Nature*, volume 401, 11 1999: pp. 788–91, doi:10.1038/44565.

[32] Li, S.; Hou, X.; et al. Learning Spatially Localized, Parts-Based Representation. 01 2001, pp. 207–212, doi:10.1109/CVPR.2001.990477.

[33] Paatero, P. Least Squares Formulation of Robust Non-Negative Factor Analysis. *Chemometrics and Intelligent Laboratory Systems*, volume 37, 05 1997: pp. 23–35, doi:10.1016/S0169-7439(96)00044-5.

[34] Naik, G. *Non-negative Matrix Factorization Techniques: Advances in Theory and Applications*. 09 2015, ISBN 978-3-662-48331-2, doi:10.1007/978-3-662-48331-2.

[35] Févotte, C.; Vincent, E.; et al. *Single-channel audio source separation with NMF: divergences, constraints and algorithms.* 01 2017.

[36] Wilburn, T. The AudioFile: Understanding MP3 compression. Oct 2007. Available from: `https://arstechnica.com/features/2007/10/the-audiofile-understanding-mp3-compression/2/`

[37] Valin, e. a. Definition of the Opus Audio Codec. RFC 6716, RFC Editor, September 2012. Available from: `http://www.rfc-editor.org/rfc/rfc6716.txt`

[38] Valin, J.-M.; Maxwell, G.; et al. High-Quality, Low-Delay Music Coding in the Opus Codec. *135th Audio Engineering Society Convention 2013*, 01 2013: pp. 73–82.

[39] Nikunen, J.; Virtanen, T. Object-based audio coding using non-negative matrix factorization for the spectrogram representation. 05 2010.

[40] Nikunen, J.; Virtanen, T. Noise-to-mask ratio minimization by weighted non-negative matrix factorization. *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2010: pp. 25–28.

[41] BS, ITU-R. Method for objective measurements of perceived audio quality. 01 2006.

[42] Sapp, C. S. WAVE PCM soundfile format. Available from: `http://soundfile.sapp.org/doc/WaveFormat/`

[43] Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, volume 40, no. 9, Sep. 1952: pp. 1098–1101, ISSN 0096-8390, doi:10.1109/JRPROC.1952.273898.

[44] The Python Language Reference. Available from: `https://docs.python.org/3/reference/`

[45] Pallets team. Click. 2014. Available from: `https://palletsprojects.com/p/click`

[46] Krekel, H.; Oliveira, B.; et al. pytest 4.1.1. 2004. Available from: `https://github.com/pytest-dev/pytest`

[47] Lippens, S. dahuffman. 2017. Available from: `https://github.com/soxofaan/dahuffman`

[48] Jones, E.; Oliphant, T.; et al. SciPy: Open source scientific tools for Python. 2001–, [Online; accessed 2019/04/14]. Available from: `http://www.scipy.org/`

[49] Oliphant, T. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–, [Online; accessed 2019/04/14]. Available from: `http://www.numpy.org/`

[50] Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, volume 9, no. 3, 2007: pp. 90–95, doi:10.1109/MCSE.2007.55.

[51] Makhoul, J. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 28, no. 1, February 1980: pp. 27–34, ISSN 0096-3518, doi:10.1109/TASSP.1980.1163351.

[52] Oppenheim, A. V.; Schafer, R. W. *Discrete-Time Signal Processing.* Upper Saddle River, NJ, USA: Prentice Hall Press, third edition, 2009, ISBN 0131988425, 9780131988422.

[53] Griffin, D.; Lim, J. Signal estimation from modified short-time Fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 32, no. 2, April 1984: pp. 236–243, ISSN 0096-3518, doi:10.1109/TASSP.1984.1164317.

[54] Borß, C.; Martin, R. On the construction of window functions with constant-overlap-add constraint for arbitrary window shifts. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2012, ISSN 2379-190X, pp. 337–340, doi:10.1109/ICASSP.2012.6287885.

[55] Holters, M.; Zölzer, U. GstPEAQ – An Open Source Implementation of the PEAQ Algorithm. 12 2015.

[56] Taymans, W.; Baker, S.; et al. *GStreamer 1.8.3 Application Development Manual.* United Kingdom: Samurai Media Limited, 2016, ISBN 9789888406654, 9888406655.

[57] Holters, M. GstPEAQ. 2015. Available from: `https://github.com/HSU-ANT/gstpeaq`

[58] Ross, S. M. Introduction to Probability Models. 1972.

[59] About LAME. Available from: `http://lame.sourceforge.net/about.php`

[60] Opus downloads. Available from: `http://opus-codec.org/downloads/`

Appendix A

Acronyms

todo TODO

Appendix B

Contents of enclosed CD

```
readme.txt ........ the file with CD contents description
exe ................ the directory with executables
src ................ the directory of source codes
  wbdcm ............ implementation sources
  thesis ........... the directory of LaTeX source codes of the thesis
text ............... the thesis text directory
  thesis.pdf ....... the thesis text in PDF format
  thesis.ps ........ the thesis text in PS format
```