BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

LOSSLESS ENCODING OF SIGNALS FROM MICROPHONE ARRAY

BEZEZTRÁTOVÉ KÓDOVÁNÍ SIGNÁLŮ Z MIKROFONNÍHO POLE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ADRIÁN KÁLAZI**
AUTOR PRÁCE

SUPERVISOR **prof. Dr. Ing. JAN ČERNOCKÝ**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Kálazi Adrián**
Programme: Information Technology
Title: **Lossless Encoding of Signals from Microphone Array**
Category: Speech and Natural Language Processing

Assignment:

1. Get acquainted with the principles of loss-less audio coding, especially the FLAC codec, analyze its source code and re-implement part of it.
2. Get acquainted with the principles of microphone arrays and prepare examples of multi-channel speech recordings from a microphone array.
3. Suggest a technique for lossless coding of a multi-channel audio signal for a microphone array and implement it.
4. Evaluate the compression ratio and encoding speed and propose ways to improve them.
5. Implement and evaluate at least one improvement.
6. Consolidate your code into a library or a binary tool that can be used from the command line. Document your code properly.

Recommended literature:

- FLAC - Free Lossless Audio Codec, https://xiph.org/flac/
- Kumatani, K., McDonough, J., & Raj, B.: Microphone array processing for distant speech recognition: From close-talking microphones to far-field sensors. IEEE Signal Processing Magazine, 29(6), 127-140, 2012.

Requirements for the first semester:

- Items 1 and 2, significant advance in item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Černocký Jan, prof. Dr. Ing.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: May 9, 2022

Abstract

Lossless audio coding is increasingly important for properly archiving and preserving audio in its original form. To achieve a good compression ratio, lossless encoding techniques such as linear prediction and Rice coding are often applied to the original audio in order to minimize its entropy and preserve the original signal bit-precisely with a reduced size. This thesis explores the possibilities of efficiently encoding multi-channel audio in a way that exploits the similarity between multiple channels in order to achieve better compression ratios. This thesis also explores the techniques employed by FLAC in more depth while also providing solutions to a few problems that FLAC fails to address.

Abstrakt

Bezeztrátové kódování zvuku je stále důležitější pro správnou archivaci a uchování zvuku v původní podobě. Pro dosažení dobrého kompresního poměru se na původní zvuk často aplikují techniky bezztrátového kódování, jako je lineární predikce a Riceho kódování, aby se minimalizovala jeho entropie a zachoval se původní signál s bitovou přesností se zmenšenou velikostí. Tato práce zkoumá možnosti efektivního kódování vícekanálového zvuku způsobem, který využívá podobnosti mezi více kanály za účelem dosažení lepších kompresních poměrů. Tato práce také hlouběji zkoumá techniky používané FLAC-em a zároveň poskytuje řešení několika problémů, které FLAC neřeší.

Keywords

lossless speech coding, lossless audio coding, linear prediction, adaptive Rice coding, decorrelation, audio processing, dynamic audio encoding

Klíčová slova

bezeztrátové kódování řeči, bezeztrátové kódování zvuku, lineární predikce, adaptivní Riceho kódování, dekorelace, zpracování zvuku, dynamické kódování zvuku

Reference

KÁLAZI, Adrián. *Lossless Encoding of Signals from Microphone Array.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Dr. Ing. Jan Černocký

Lossless Encoding of Signals from Microphone Array

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of prof. Dr. Ing. Jan Černocký. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Adrián Kálazi
May 9, 2022

Acknowledgements

I would like to thank my supervisor Jan Černocký for his guidance, suggestions and know-how, which greatly helped during my work. I would also like to thank Vladimír Malenovký for the information he provided about codecs and multi-channel audio analysis. Last but not least, I would like to thank František Bernáth for his technical insight, which was very helpful during the final moments of writing this thesis.

Contents

Chapter 1

Introduction

Audio compression has an important role in reducing the strain on data storage across multiple fields. Since keeping up with increasing file sizes can be often challenging, compression has became the most important tool for keeping the rising storage demands in check.

Big data sets such as long conference recordings can often grow quickly and take up a substantial amount of space if left unencoded. To prevent this, audio is often encoded using audio codecs offering much better compression ratios than general purpose data compressors such as *Lempel–Ziv* (LZ) or *DEFLATE*.

These audio codecs come in two distinct groups: *lossy* and *lossless*. Lossy compressors employ techniques such as *modified discrete cosine transform* (MDCT) or *perceptual coding*. These techniques adapt the frequency spectrum of the audio by removing parts that are redundant for human perception. These codecs usually offer better compression ratios in exchange for introducing *compression errors*. These compression errors can be indistinguishable for humans and are therefore often ignored.

On the other hand lossless codecs offer a solution for storing audio without altering it in the process. In contrast to lossy codecs, these employ techniques that help reduce file sizes by lowering the intensity of audio signals while *preserving their entropy*. This is useful for archiving, high quality audio and a lot of other purposes. Some of the techniques lossless encoders employ are *linear prediction* and *entropy coding*.

Linear prediction takes advantage of the structure of speech by *estimating the formants* and removing their effect from the speech signal. This process often results in the remaining signal having a much lower intensity.

This low intensity signal can then be efficiently encoded using an entropy coder such as a *Rice coder*. These coders can encode small (and thus frequent) values efficiently while also being able to encode larger values. They can often be fine-tuned to a specific dynamic range using a parameter determined from the would-be encoded signal.

A microphone array is a set of microphones which produce multiple channels of often very similar audio signal. This fact can sometimes be exploited to our benefit.

1.1 Lossless Encoding of Signals from Microphone Array

The aim of this thesis is to design a *lossless audio codec*, capable of exploiting these similarities and achieving a comparatively better compression ratio for recordings with multiple channels.

To reach this goal, we first needed to understand the FLAC encoding and decoding process. Since the resulting codec is in Python, we had to re-implement these processes. Next, we had to analyze the multi-channel audio that is produced by microphone array. Previous theses from the Speech@FIT group proved to be a good source for this [12].

After we expanded our knowledge of these fields, we modified these processes and proposed a new codec. This codec was designed to achieve better compression ratio for multi-channel audio. The codec was implemented as a library and a companion executable. The resulting codec is finally evaluated and compared to FLAC.

1.2 Scope of chapters

The thesis is structure into thematically distinct chapters. Chapter 2 deals with the current state of the art. It also explains most of the techniques used for losslessly encoding audio. In chapter 3, we list the data used for development and testing. This data was acquired from publicly available corpuses. Chapter 4 is the core of this thesis. In this chapter we explain most of the proposed techniques to improve compression. We also reveal the impact of these techniques and explain the reasoning behind abandoning some of them. Chapter 5 deals with the implementation of our codec. This chapter contains the technical aspects of the codec implementation. We also explain the internal structure of the codec as well as the used libraries and dependencies. In chapter 6, we test the performance of our implemented codec and compare it to FLAC.

Chapter 2

State of the art

Compression involves the removal of redundant information from a certain data source in order to lower the size of that data while. This process is used in a wide range of applications for the sole purpose of storing data more efficiently. General purpose compression algorithms such as Lempel–Ziv (LZ) or DEFLATE can be useful for compressing data in general, but fail short on audio compression. This is due to audio data often having a large amount of information redundancy. Specialized audio compressors can exploit these properties using audio compressing techniques such as linear prediction.

This chapter explores the current state regarding audio compression techniques and different audio compression tools. It also offers a deep dive into the internal workings of FLAC and its components. The last section offers a basic understanding of encoding multi-channel audio recorded by microphone arrays.

2.1 Audio Compression formats

In this section we compare some of the most well known codecs and the techniques they employ to compress a given input. Some of these techniques such as linear predictive coding and entropy coding are later employed by our proposed codec in chapter 4.

These codecs can be divided into two main categories, lossy and lossless. While lossy codecs can achieve much better compression ratio and often even reduce the size of output files to a theoretical minimum, lossless codecs such as FLAC are still recommended for all cases where the accuracy matters [9].

2.1.1 Lossy

Audio compressors usually take advantage of lossy compression methods, such as modified dicrete cosine transforms (MDCT) or psychoacoustic models to achieve better compression ratios. These codecs offer better compression ratio by sacrificing minor details in the given input. This most often results in irreversibly altering the input audio signal and introducing compression errors [9]. These compression errors in most cases not have a significant impact on the human perception if done correctly. These compression errors might not be desirable in certain cases when digitally processing audio. These include algorithms used for speech recognition or audio processing using neural networks.

MP3

MP3 or formally MPEG-1/MPEG-2 Audio Layer III is a lossy audio compression format [9]. It is mostly used due to its effective storage of audio data. The core idea of MP3 is employing a modified discrete cosine transform (MDCT) and a psychoacoustic model to discard parts of the data which are mostly redundant for human perception.

Opus

Opus is a low-latency speech oriented codec developed by the Xiph.org foundation and standardized through RFC 6716 [10]. Opus was originally a successor to Speex, a codec specifically designed to be effective at encoding speech. Opus combines two distinctive algorithms, a linear prediction (LPC) based algorithm and a low-latency modified discrete cosine transform (MDCT). Opus also uses code-excited linear prediction (CELP) techniques in the frequency domain to increase its prediction efficiency. This allows Opus to code speech and even other audio efficiently, while maintaining a relatively good quality and minimal compression error [9].

Vorbis

Vorbis is a lossy compression format developed by the Xiph.org foundation as a replacement for MP3 [5]. The source code of Vorbis is open-source [1].

Vorbis uses a modified discrete cosine transform (MDCT) similarly to MP3. The resulting frequency domain is then split into a noise floor and residual. These are then quantized using codebook-based vector quantization. The quantized residuals are then entropy coded.

The performance of this codec is very similar to MP3, achieving an overall better performance than lossless codecs. Although this seems to be the case, Opus still offers a better performance for higher compression levels [9].

Vorbis served as our main inspiration alongside FLAC during our research.

2.1.2 Lossless

Lossless audio codecs preserve the integrity of the original audio data bit-precisely. This is often done by using techniques such as linear prediction and entropy coding which are lossless in nature. The most used lossless codec currently is FLAC[2] that served as the main inspiration for our codec. FLAC is explored in more detail in section 2.2.

WAV

Waveform Audio File Format (or WAVE) is a standard developed by Microsoft and IBM [9]. It is mostly used for storing audio data in a pulse-code modulated (PCM) format. All of the samples in a WAV file are raw and unencoded.

WavPack

WavPack is an open-source lossless audio compression format developed by David Bryant [1]. WavPack uses linear prediction in combination with entropy coding, which is very similar to how FLAC works. WavPack offers a predictor implemented purely in integer math and

[1] https://github.com/xiph/vorbis
[2] https://xiph.org/flac/

a special data encoder instead of Rice coding. This encoder is less efficient but offers easy adaptation to lossy encoding.

WavPacks also offers a hybrid mode, allowing the creation of two files, one of which is a standalone WavPack audio file with lossy encoding. The two files together can then be losslessly restored to the original audio track.

Shorten

Shorten is a fast, low complexity audio compressor originally developed by Tony Robinson [8]. It can operate in both lossy and lossless mode.

Shorten served as a predecessor to FLAC and works in much the same way, with the differences being relatively minor. FLAC trivially extends and improves the fixed predictors, LPC coefficient quantization, and Rice coding used in Shorten. [11] Shorten is no longer developed and is superseded by FLAC.

2.2 FLAC

FLAC (Free Lossless Audio Codec) is a lossless audio codec developed by the Xiph.Org foundation [11]. FLAC preserves the original audio bit-precisely while achieving a decent compression ratio. It uses mostly integer operations to preserve losslessness and improve the encoding speed.

The main techniques FLAC employs are Linear prediction (LPC) and Rice coding.

2.2.1 Encoding

The encoding process of FLAC follows a simple scheme as shown in figure 2.1. Most frames are encoded as LPC frames, which follow a predict -> partition -> Rice encode process. Constant and Verbatim frames are encoded directly.

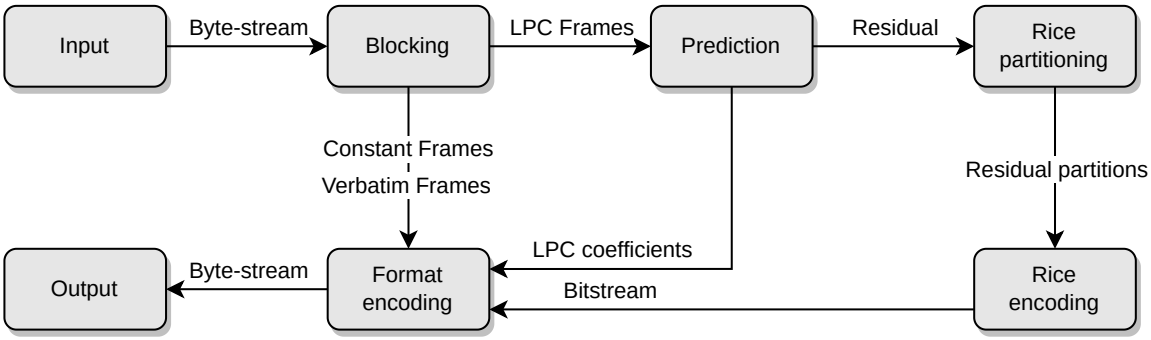

Figure 2.1: Block diagram of the FLAC encoder

2.2.2 Blocking

The first step of the encoding process is splitting the audio track into equal-length blocks. These blocks, called frames, are processed sequentially.

After slicing a multi-channel audio track, each frame consists of a number of subframes equal to the number of audio channels from the original audio file. These subframes are processed differently depending on their type. Frames which are not encoded as LPC frames

are classified as Constant (when each sample has the same constant value) or Verbatim (whose residual would have a higher intensity and thus require more bits to store than the actual source signal).

SubFrame types

FLAC offers four methods for modelling the input signal [11]:

1. *Verbatim* - `SUBFRAME_VERBATIM` - zero order predictor
   The residual is the signal itself. The residual coding stage is skipped and the samples are stored as they appear in the source file.

2. *Constant* - `SUBFRAME_CONSTANT` - constant value predictor (order 1)
   The signal is pure DC. This constant value is stored unencoded. The residual coding stage is skipped and only one value is stored in this subframe, during decoding it is expanded to the original constant valued signal.

3. *Fixed linear predictor* - `SUBFRAME_FIXED` - constant value predictor (order P)
   The predictor values are determined at compile time, only the order is stored. The residual is Rice encoded. This method is implemented in FLAC, though its use is often limited and the encoder itself favors conventional LPC frames

4. *FIR Linear prediction* - `SUBFRAME_LPC` - dynamic value predictor (order P)
   The LPC coefficients are calculated using the Levinson-Durbin method. Order and the quantized LPC coefficients are stored. The residual is Rice encoded. This is the standard LPC subframe which we will discuss and mostly use in our codec, although we later introduce a special version of this frame, more about this new subframe type can be found in section 4.1.2.

2.2.3 Decoding

The decoding process is similar but reversed. Since subframes already have their type and the LPC reconstruction is much simpler than prediction, this process is often much quicker and memory efficient than encoding.

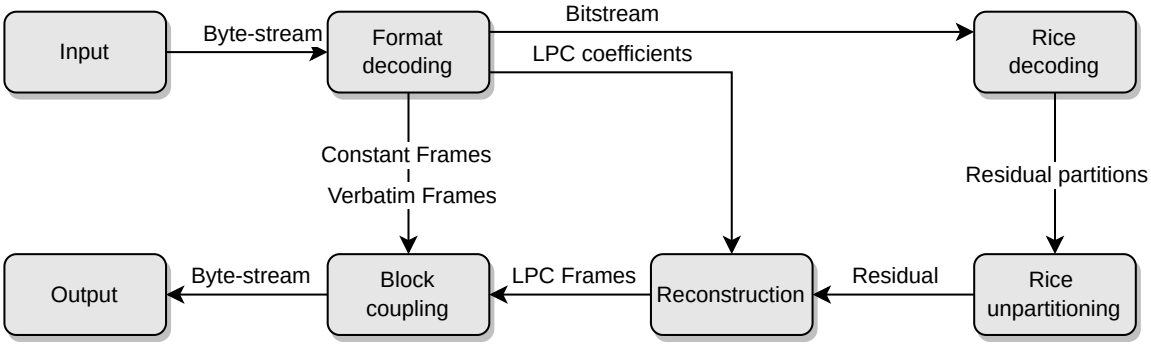

Figure 2.2: Block diagram of the FLAC decoder

2.2.4 Linear prediction

This section is heavily inspired by the Speech Signal Processing (ZRE) course at FIT [2]. Linear prediction is a speech analysis technique which represents the spectral envelope of

a signal of speech using a linear predictive model [6]. The predicted signal is often very similar to the original, as seen in figure 2.3, with spikes in the residual representing larger entropy.

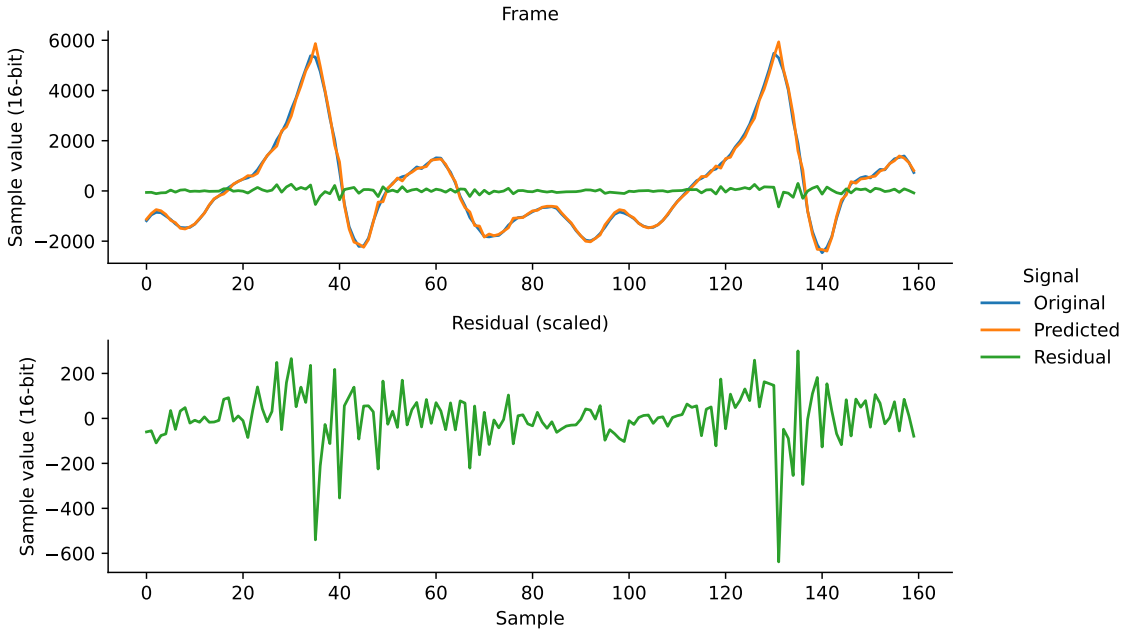

Figure 2.3: Visualization of linear prediction

Basics of LPC

Linear prediction can be represented by a set of filter coefficients. These coefficients are together called a predictor. They are used for constructing a FIR and an IIR filter. The FIR filter is used for prediction and the IIR filter for reconstruction of the original signal. Linear prediction only considers past samples for estimating the current sample.

The prediction part can by performed by using a convolutional filter:

$$s_r[n] \quad = \quad s[n] + \sum_{m=1}^{P} a_m \, s[n-m] \qquad \text{for } n \geq P. \tag{2.1}$$

The first P samples are left intact and stored as warm-up samples. These are used for reconstruction when decoding.

This predicted signal is removed from the original signal, lowering its intensity. This low-intensity residual, seen in figure 2.3, is later efficiently encoded using Rice coding as explained in section 2.2.5.

For restoring the original signal, a reverse of this process can be performed:

$$s[n] \quad = \quad s_r[n] - \sum_{m=1}^{P} a_m \, s[n-m] \qquad \text{for } n \geq P. \tag{2.2}$$

Note the use of previous $s[n]$ samples in this equation. Since this filter uses the values of previous output values, it is classified as an IIR filter. As for every IIR filter, it may become

unstable and introduce reconstruction issues. To prevent this, we check the stability of each constructed filter. This is explained later in this section.

Conventional filtering is performed using floating point coefficients. This is often done after the coefficients have been quantized and then dequantized to prevent quantization noise. This is not the case for FLAC and our proposed codec however, which both use a method involving the use of quantized coefficients. This quantization method is explored later in this section.

Autocorrelation

For calculating the LPC coefficients, FLAC uses the *correlation method* [2, page 54]. The signal can be optionally passed through a windowing function to improve the precision of the LPC coefficients. The window used by FLAC is the Tukey window. It has a narrow frequency response with periodic spikes which make it ideal for this purpose [3]. This window is later used by our own codec as seen in section 4.1.3.

The autocorrelation coefficients are calculated using:

$$R_k = \sum_{n=0}^{N-k-1} s[n]\, s[n+k]. \tag{2.3}$$

For computing the LPC coefficients, only the first $P+1$ autocorrelation coefficients are required.

Coefficient computation

As mentioned before, the FLAC reference encoder precomputes predictors up to a set maximum order. This subsection will describe the calculation of one set of these coefficients with order P.

For computing the actual coefficients, the autocorrelation coefficients R_n can be arranged in a symmetrical Toeplitz matrix and solved for a_n:

$$\begin{bmatrix} R_0 & R_1 & \cdots & R_{P-1} \\ R_1 & R_0 & \cdots & R_{P-2} \\ \vdots & \vdots & \ddots & \vdots \\ R_{P-1} & R_{P-2} & \cdots & R_0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_P \end{bmatrix} = - \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_P \end{bmatrix} \tag{2.4}$$

Ti find the coefficients a_n more efficiently, the the fast Levinson-Durbin recursion can be used [2]:

$$E^{(0)} = R(0) \tag{2.5}$$

$$k_i = -\left[R(i) + \sum_{j=1}^{i-1} a_j^{i-1} R(i-j)\right] / E^{(i-1)} \tag{2.6}$$

$$a_i^{(i)} = k_i \tag{2.7}$$

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)} \text{ for } 1 \le j \le i-1 \tag{2.8}$$

$$E^{(i)} = (1 - k_i^2) E^{(i-1)} \tag{2.9}$$

Most libraries already implement this method, such as SciPy[3].

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve_toeplitz.html

Order

The order of the predictor needs to be estimated, so we don't waste bits on redundant coefficients. The order represents how many past samples are considered when determining the value of the current sample. The order should be chosen based on the sampling frequency of the original signal.

The FLAC reference encoder precomputes the LPC coefficients up to a maximal given order. The encoder then estimates the best order to use (2.12) by calculating the expected bits per sample (2.11) that will be required to store the residual. This is calculated using the LPC error (2.10) from each set of LPC coefficients.

$$E_p = R_0 + \sum_{i=1}^{p} a_i \, R_i \tag{2.10}$$

$$b_{exp}[p] = 0.5 \, \log_2 \left(\frac{0.5 \, E_p}{N} \right) \tag{2.11}$$

$$P_{best} = \arg\min \left(b_{exp} \right) \tag{2.12}$$

This chosen order is then used to select one of the precomputed set of LPC coefficients which will be used to encode the subframe. This ensures that the predicted signal would be the best estimation the encoder can provide to the input signal. The order can be different for each subframe.

Coefficient quantization

The resulting coefficients are represented as floating point numbers, which are good for generating the predicted signal precisely but storing them requires 8 bytes (for 64-bit double precision). This is not ideal if we want to store them efficiently.

For this reason, the coefficients a_n are quantized. The lost precision does not significantly affect the predicted signal while significantly reducing the required space needed to store them. This quantization method is easily reversible and can later be used for calculating the residual using integer math. Additionally the precision information (or shift) needs to be stored as a constant k_{shift} for each frame:

$$k_{shift} = k_{prec} - \lceil \log_2 |a_{max}| \rceil - 1 \tag{2.13}$$

where $\lceil x \rceil$ denotes rounding up to the closest larger integer.

For computing the quantization shift (2.13), the desired precision in bits k_{prec} and the LPC coefficient with the largest absolute value a_{max} are necessary. The FLAC reference encoder estimates the optimal precision k_{prec} to use based on the block size and dynamic range of the original signal [11]. The quantized (QLP) coefficients can be calculated with:

$$q_i = \left\lfloor a_i \cdot 2^{k_{shift}} \right\rfloor \tag{2.14}$$

where $\lfloor x \rceil$ denotes rounding to the closest integer.

A dequantization coefficient k_{quant} is introduced, which reverses the effects of this quantization:

$$k_{quant} = \frac{1}{2^{k_{shift}}} \tag{2.15}$$

$$a_i = q_i \times k_{quant}. \tag{2.16}$$

The dequantization of the q_n coefficients is not necessary in the implementation though, since both the encoder and decoder only use the quantized coefficients. The shift (2.13) as well as the quantized LPC coefficients (2.14) are stored for each frame. The example results of this quantization process can be found in table 2.1.

Table 2.1: Example coefficients for predictor with order 8

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
LPC	-2.077	1.354	-0.494	0.615	-0.763	0.837	-0.625	0.174
QLP	-1064	694	-253	315	-391	429	-320	88

The quantized LPC coefficients (QLP) in table 2.1 are represented in an integer format with precision $k_{prec} = 12$ and $k_{shift} = 9$.

Alternative quantization methods

Direct quantization of LPC coefficients is dangerous, since quantization can cause the filter to be unstable. This can pose issues when decoding by making the residual irreversible. Alternative ways to quantize the LPC coefficients would be to transform them to:

- PARCOR coefficients - These coefficients are byproducts of the Levinson-Durbin recursion, shown in equation (2.6) as k_i. They are more tolerant towards quantization noise and more suitable for quantization.
- Line spectral pairs/frequencies [4] - LSPs have a few positive properties which include smaller sensitivity to quantization noise, ensured filter stability and the ability to be interpolated.

All of these methods seem promising and in theory should improve the encoding efficiency. FLAC and Straw though still use direct quantization with stability check, due to the reasons mentioned in section 4.1.3.

Prediction and residual

The prediction process, as explained at the start of this section, consists of filtering the frame using a modified FIR filter. This filtering is shown in figure 2.4.

[4] http://www.dspcsp.com/pdf/lsp.pdf

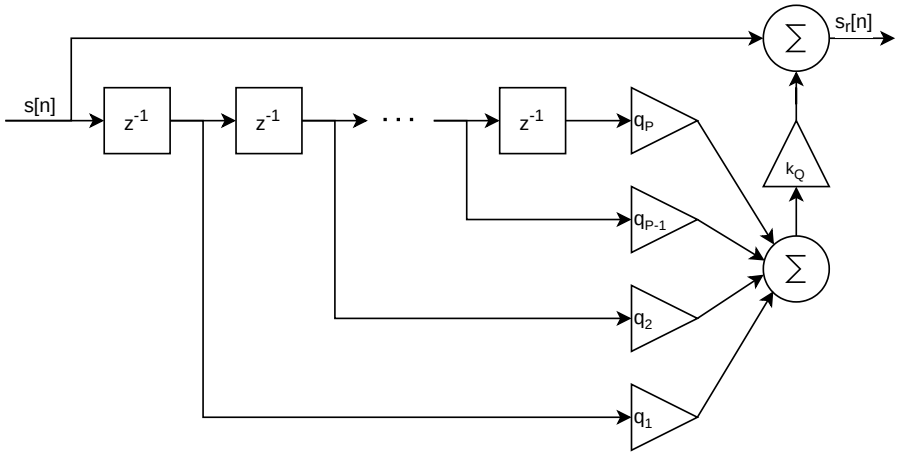

Figure 2.4: Signal prediction

For filtering the quantization has to be taken into account by multiplying the predicted signal by the dequantization constant k_{quant}:

$$s_r[n] \quad = \quad s[n] + \left\lfloor k_{quant} \sum_{m=1}^{P} q_m \, s[n-m] \right\rfloor \qquad \text{for } n \geq P \tag{2.17}$$

where $\lfloor x \rfloor$ denotes rounding down to the closest smaller integer.

This method allows for quick integer math. One set of quantized coefficients q_n is used for prediction and also reconstruction.

The first P warm-up samples are left intact and stored directly without compression. These warm-up samples are later used for reconstructing the original signal.

The resulting residual signal, seen in figure 2.3, has a significantly smaller intensity than the original signal. This fact can then later be exploited by an entropy coding scheme, such as Golomb or Rice coding as described in section 2.2.5.

Reconstruction

Reconstruction is performed using a modified IIR filter as shown in figure 2.5. Due to filtering using already quantized coefficients, the predicted signal must be multiplied by the dequantization constant k_{quant}. This k_{quant} constant is the same as the constant used for prediction, described in equation 2.15.

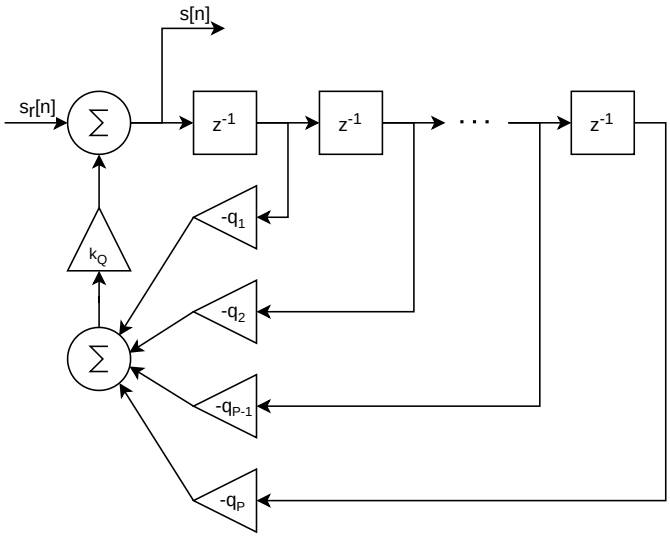

Figure 2.5: Signal reconstruction

The output signal s is initialized with P warmup samples, so the filtration only needs to be performed for $n \geq P$:.

$$s[n] \quad = \quad s_r[n] - \left\lfloor k_{quant} \sum_{m=1}^{P} q_m \, s[n-m] \right\rfloor \qquad \text{for } n \geq P. \tag{2.18}$$

2.2.5 Rice coding

The results of the linear prediction step are a set of LPC coefficients and the residual. We can assume that the residual contains a signal with relatively low intensity. This means that the values of this signal are closely centered around zero.

To efficiently encode these low intensity residuals, FLAC uses Rice coding [7]. Rice coding is a method for run-length data compression. These coders are very efficient at losslessly encoding low intensity signals.

Rice coding uses a constant parameter m for each encoded frame. For Rice coders this parameter can be stored more efficiently stored as a logarithm:

$$k \quad = \quad \log_2 m. \tag{2.19}$$

Rice coding is only defined for positive integers, so the residual must be mapped by an overlap and interleave scheme (2.20). This scheme maps positive values to positive even integers and negative values to positive odd integers.

$$y \quad = \quad \begin{cases} 2|x| & \text{for } x \geq 0 \\ 2|x| - 1 & \text{for } x < 0 \end{cases} . \tag{2.20}$$

The resulting interleaved residual is shown in figure 2.6.

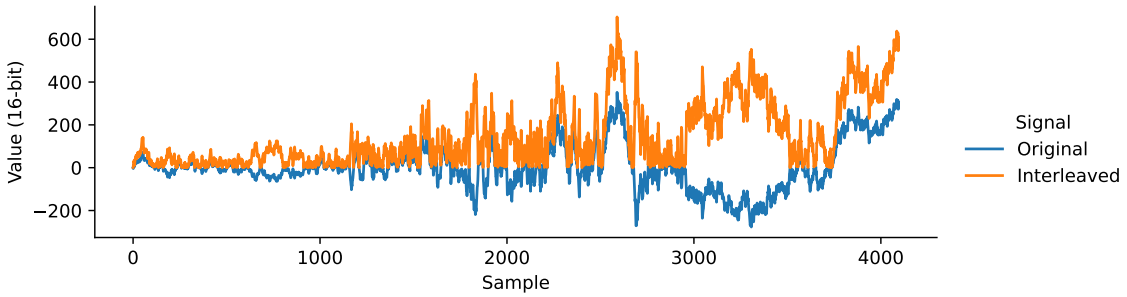

Figure 2.6: Interleaving of the residual

A simplified description of the Rice encoding algorithm is as follows:

1. $s[n]$ is interleaved.
2. The quotient $q = s/m$ is calculated. Since we are working with integers, q is rounded down.
3. q binary zeroes are written to the bitstream.
4. A binary one is written to the bitstream.
5. The k last bits of $s[n]$ are written to the bitstream.

Our Rice coding scheme, as shown in section 4.1.5, uses a modified version of this encoding algorithm with an adaptive coding parameter.

Partitioning

FLAC uses a partitioning scheme in which the residual is partitioned into several equal-length regions of contiguous samples, and each region is coded with its own Rice parameter m based on the mean of the region [11].

This effectively improves the encoding efficiency for signals with highly variable intensity but may fall short in certain cases, which is even acknowledged by the FLAC developers - „*The FLAC format has reserved space for other coding methods. Some possibilities for volunteers would be to explore better context-modeling of the Rice parameter*“ [11].

2.2.6 Problems of FLAC

For multi-channel audio, the FLAC format offers a basic mid-side interchannel decorrelation method. This is available for stereo streams only. If more than two channels are present, no additional processing is supported and the channels are encoded independently.

FLAC also limits the maximum number of channels to 8, which can pose a problem when encoding audio from microphone arrays with more than 8 inputs, as further described in section 2.3.1.

2.3 Microphone arrays

Microphone arrays consist of multiple microphones, arranged in a certain shape. Microphone arrays produce this audio in a multi-channel fashion. These channels are often very

similar in shape. To efficiently store these signals, FLAC currently offers no practical solution other than independently storing the signal from each channel.

2.3.1 Channel differences

Microphone arrays present a specific situation in which multiple (sometime more than 8) channels are present. Each of these channels can be assumed to have significant correlation with the others. The differences between these channels often consist of a combination of three effects - shift, gain and DC bias.

Our goal is minimizing the differences between individual channels while keeping their entropy mostly intact. This can be accomplished by a few reversible operations which increase the similarity between channels. Employing the use of a neural network is also possible, though this method will not be discussed in this thesis.

Shift

Shift or signal delay is a common characteristic of microphone arrays and is often caused by the sound waves hitting the individual microphones at different times. In some cases it can be the result of an amplifier design fault where each microphone has its own amplifier. These, often insignificant, differences between the individual amplifiers can also cause a certain delay.

Mitigating this can be as simple as just shifting each channel compared to the main or leading channel. This can be accomplished by using cross-correlation (2.21) to find the lag for each channel.

$$R_k = \sum_{n=0}^{N-k-1} s_1[n]\, s_2[n+k] \tag{2.21}$$

$$shift = \arg\max R_k \tag{2.22}$$

These lags can then be used for delaying the other channels. This can be accomplished by removing samples from the start and end of each channel and storing them separately. The effects of shift on a speech signal are shown in figure 2.7.

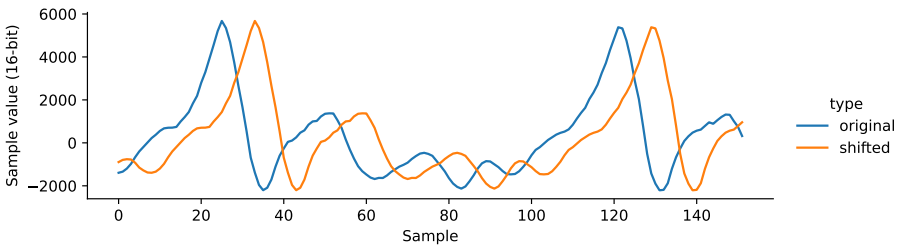

Figure 2.7: Effects of shift on a signal

Gain

Gain represents the intensity of each channel. Differences in gain are mostly tied to hardware differences between individual amplifiers in the microphone array. No amplifier is

identical to another and these differences cannot be ignored. In a few cases, hardware malfunction can cause a specific amplifier to drop or increase its gain more significantly:

$$E = \sqrt{Var(frame)}. \tag{2.23}$$

The simplest approach to counter this difference is to linearly scale each channel by a ratio determined by the energy of that channel compared to a main channel:

$$g_n = \frac{E_{max}}{E_n}. \tag{2.24}$$

The effects of gain difference on a speech signal are shown in figure 2.8.

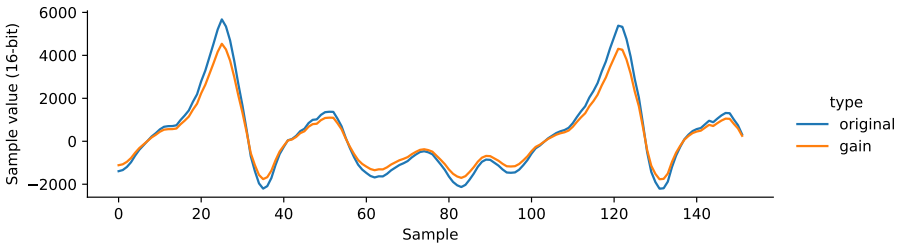

Figure 2.8: Effects of gain difference on a signal

DC bias

As gain and shift, DC bias is also often the result of hardware inconsistency. Some amplifiers can and often do add a DC bias, which causes the signal to not be centered around zero but instead around this arbitrary value. This bias essentially acts as a constant value added to each sample.

Subtracting the mean of a given frame from each sample is often the easiest method to eliminate these differences:

$$\bar{s} = \frac{1}{N}\sum_{n=0}^{N-1} s[n] \tag{2.25}$$

$$s[n] = s[n] - \bar{s}. \tag{2.26}$$

The effects of added DC bias on a speech signal are shown in figure 2.9.

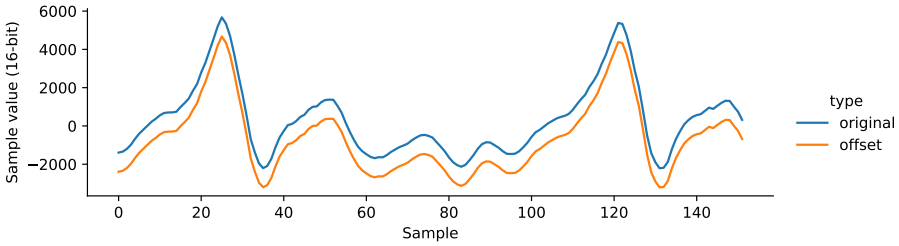

Figure 2.9: Effects of DC bias on a signal

Corrections

In reality, the signals recorded by microphone arrays are influenced by all of the effects mentioned above. Correcting the combination of these effects can be significantly more challenging. The orders of operations as well as the nature of the different microphones can cause differences that cannot be reversed by any combination of methods. Dealing with all of these factors requires a longer computation time and a more complex algorithm than just countering a specific feature/distortion.

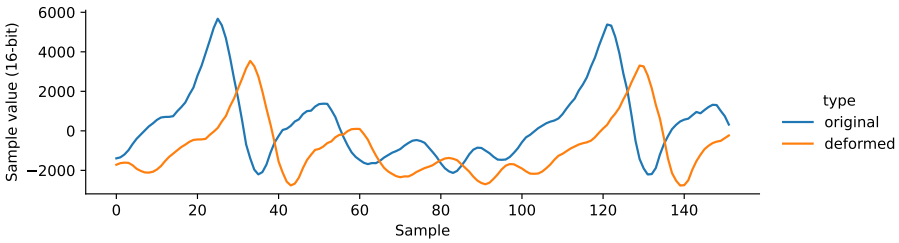

Figure 2.10: Effects of all distortions on a signal

In essence, by removing these differences the streams can often look very similar. This can be exploited in most cases but it does not mean that the signals are identical. In certain cases, the material of the microphone array as well as vibrations and other environmental noises can differently affect each channel. This can be ignored if a good heuristic is used for determining whether to perform this operation or just perform independent encoding.

These corrections can be done on a per-frame basis to be as effective as possible or just globally applied to the whole signal. More about correcting these factors and the effect of these corrections on the encoding can be found in section 4.1.1.

Chapter 3

Data

This chapter describes the recordings and other data used for development and later testing purposes. Most of the data has been provided by my supervisor and the Speech@FIT group [1].

The files used were part of different speech corpuses, most notably the AMI [2] and CHiME [3] corpus.

The common characteristic of all these audio tracks is their 16 kHz sampling frequency and 16-bit sample resolution in the `pcm_s16le` format. These properties will not be listed in the tables for readability.

3.1 AMI corpus

The parts of the AMI corpus that we used were recorded on 8-channel tabletop microphone arrays.

Development

The audio tracks used for development are shown in table 3.1. The development recordings originate from the Edinburgh `EN2001a` 1-hour meeting.

Table 3.1: Development files from the AMI corpus EN2001a

File Name	Channels	Duration	File Size
1min.wav	8	1:00	14.64 MiB
10min.wav	8	10:00	146.48 MiB

These audio tracks contain mostly speech in 8 channels. For development purposes, these tracks have been trimmed down to smaller duration.

The audio most used for development was the trimmed 1 minute track `1min.wav`. This audio track has been used for and can be seen in most of the tables used in chapter 4.

[1] https://speech.fit.vutbr.cz/
[2] https://groups.inf.ed.ac.uk/ami/corpus/
[3] http://spandh.dcs.shef.ac.uk/chime_challenge/CHiME5/data.html

Testing

For most testing purposes, we used the same audio tracks from meeting EN2001a. These testing recording were trimmed to longer a longer duration as shown in table 3.2.

Table 3.2: Testing files from the AMI corpus EN2001a

File Name	Channels	Duration	File Size
AMIa_a1.wav	8	1:00:00	878.90 MiB
AMIa_a2.wav	8	1:00:00	878.90 MiB
AMIa_a1_ch0.wav	1	1:00:00	109.86 MiB
AMIa_a2_ch0.wav	1	1:00:00	109.86 MiB

We also used the Edinburgh `EN2001b` meeting from the same corpus. The parameters of these recording can be found in table 3.3.

Table 3.3: Testing files from the AMI corpus EN2001b

File Name	Channels	Duration	File Size
AMIb_a1.wav	8	00:57:31	842.74 MiB
AMIb_a2.wav	8	00:57:31	842.74 MiB
AMIb_a1_ch0.wav	1	00:57:31	105.34 MiB
AMIb_a2_ch0.wav	1	00:57:31	105.34 MiB

3.2 CHiME corpus

To diversify the testing data set we also used a corpus from *The 5th CHiME Speech Separation and Recognition Challenge*. This corpus consists of 4-channel audio tracks, which we trimmed down to 30 minutes. The specs for these recordings can be found in table 3.4.

Table 3.4: Testing files from the CHiME5 corpus

File Name	Channels	Duration	File Size
S21_U01.wav	4	00:30:00	219.72 MiB
S21_U02.wav	4	00:30:00	219.72 MiB
S21_U03.wav	4	00:30:00	219.72 MiB
S21_U04.wav	4	00:30:00	219.72 MiB
S21_U05.wav	4	00:30:00	219.72 MiB
S21_U06.wav	4	00:30:00	219.72 MiB

Chapter 4

Proposed solution

Due to the issues presented by the FLAC codec and its lack of focus on multi-channel audio we developed a coding process. We addressed most of these issues in a way that would be beneficial to multi-channel audio encoding.

This chapter explains the proposed solution for a new codec based on FLAC with linear prediction as its main component.

4.1 Encoding process

While designing the encoding process of our codec, we drew much inspiration from FLAC in terms of structure and design. The main task was to re-implement much of what FLAC offers in Python and then build on top of a functioning architecture.

The final block diagram of our proposed codec may be seen in figure 4.1.

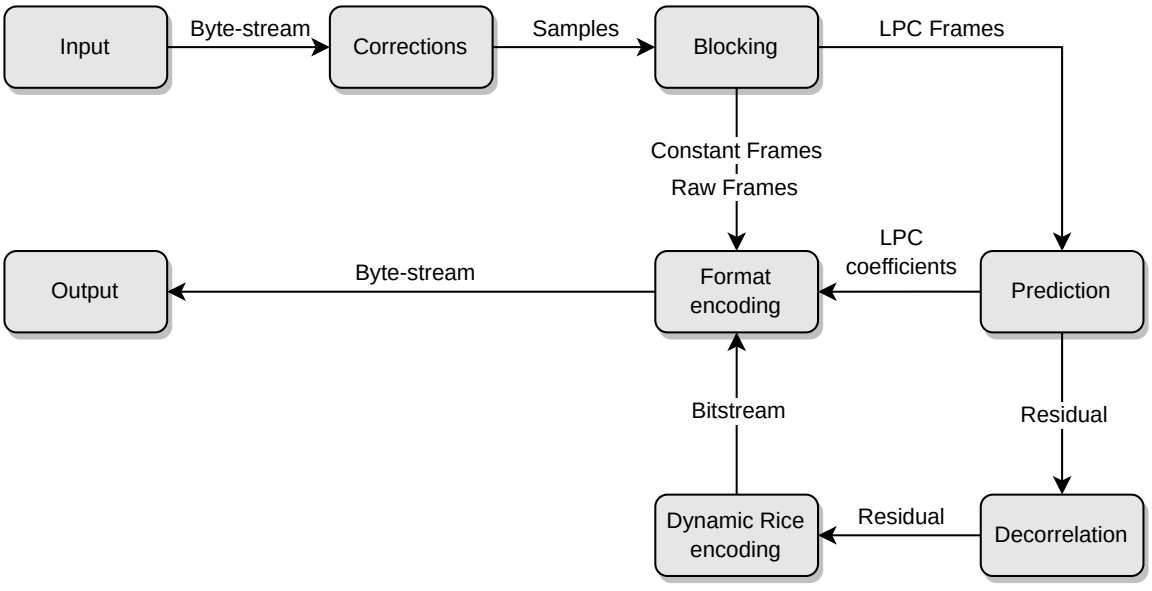

Figure 4.1: Block diagram of the Straw encoder

The main differences can be found in the way our codec handles and processes frames. FLAC by default encodes each subframe independently of the other subframes. Our proposed codec would first apply corrections to make the channels more similar. Then, it

would process the subframes of one frame together in one group and use one set of LPC coefficients for encoding each group. The resulting residuals would then be decorrelated to remove common parts from each residual. Our codec would also have a slightly different entropy coding method from FLAC.

All of these changes make our codec non-compatible with FLAC though. For this reason we heavily modified the FLAC format to suit our needs by devising a new encoding format. A detailed description of this new format can be found in appendix A.

4.1.1 Corrections

Corrections represent operations performed on the signals before framing. These operations are essential to make the channels as similar as possible.

The corrections in this section are determined and applied to a single frame. This is different in our implementation however, since these corrections are done globally (for the whole file, not per-frame). Globalizing these corrections can, however cause a few non-ideal frames. The reason behind this choice is later explained at the end of this section.

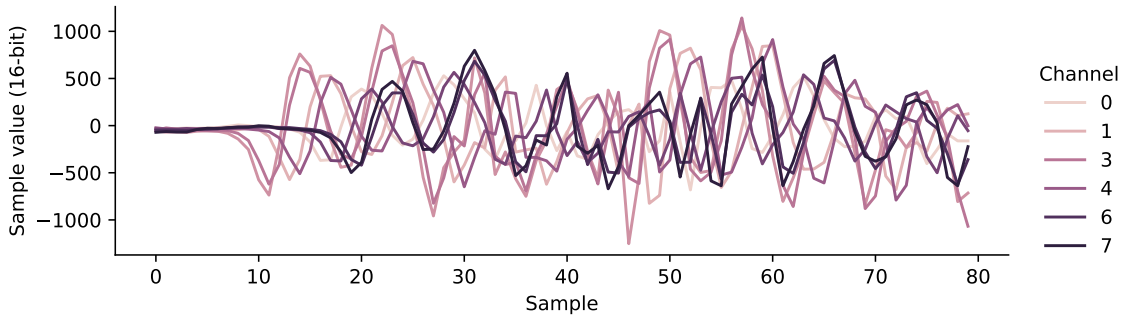

Figure 4.2: Frame before corrections

This section describes corrections as applied to a single multi-channel frame from one of our development recordings (1min.wav, samples shown are 18134:18214, more about the used recordings in chapter 3). For clarity, only 80 samples are shown from the middle of the specified frame. The original samples are shown in figure 4.2.

Gain correction

Gain correction is done simply by scaling each channel with a given ratio as explained in section 2.3.1. Since we are dealing with limited precision (e.g. 16-bit samples for PCM16), this operation inevitably alters the entropy of the signal due to rounding errors.

Downscaling would be non-reversible since it removes excess entropy without storing it anywhere. Upscaling can be implemented to be reversible using strict rounding, a process described in equation (4.2) and the reversal in section 4.2.4.

The process depends on finding the channel with the most energy, computing the scaling factors:

$$g_n = \frac{E_{max}}{E_n} \tag{4.1}$$

and scaling each weaker channel to the energy of the most energetic channel:

$$s_{eq}[n] \quad = \quad \lfloor s[n] \times g \rceil \quad \text{for } g \geq 1.0. \tag{4.2}$$

The individual gain differences $k[n]$ compared to the strongest channel are stored in a quantized format (4.3) in the `STREAMINFO` metadata block using a similar quantization method used for filtering, as seen in section 2.2.4:

$$g_q \quad = \quad \lfloor g * k_{quant} \rceil. \tag{4.3}$$

This introduces a different problem however – entropy increase. This is certainly not ideal for our use case and overall this operation had an adverse effect on the resulting file size as seen in table 4.1. An alternative idea was to perform this operation on the residuals, which showed a bit better results, as seen in table 4.1.

Table 4.1: Effects of gain correction on the encoding

Corrections	File Size	Ratio
None	5.27 MiB	36.02 %
Gain (on frame)	5.62 MiB	38.39 %
Gain (on residual)	5.54 MiB	37.86 %

While increasing the accuracy of the prediction with shared coefficients, this operation also increased the overall entropy of the signals. This resulted in significantly higher file sizes. This result was not ideal, so we chose to disable it by default in our codec, though it can be activated by the `-correct-gain` option or by adding `gain` to the corrections parameter of the `Encoder` constructor.

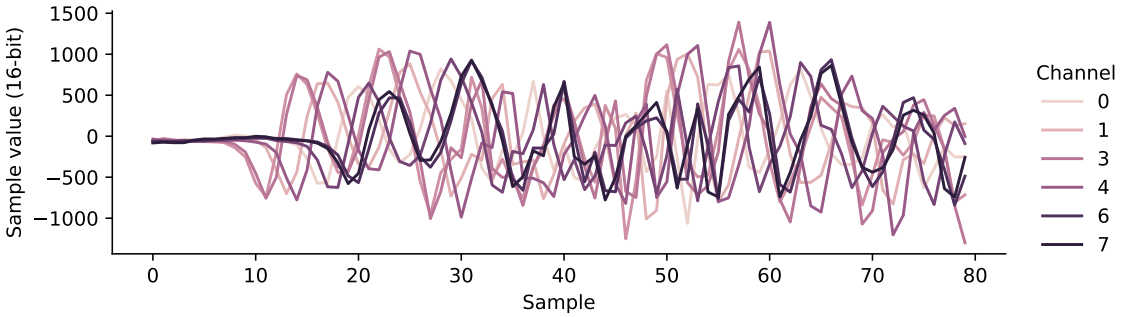

Figure 4.3: Frame with corrected gain

Shift correction

Shift correction is performed between the gain and bias corrections. This operation showed the most promise during the development process. First, the input signal is run through a Nuttall window function which increases the effectiveness of the shift estimation and thus has a positive effect by decreasing the file size as seen in table 4.2.

Table 4.2: Effects of windowing functions on the file size

Window	File Size	Ratio
None	5.30 MiB	36.16 %
Hamming	5.28 MiB	36.04 %
Nuttall	5.27 MiB	35.96 %

The windowed signal is then used for determining the leading channel by doing a double-sided cross-correlation method. These are performed using cross-correlation with two signals: $s[n]$ and the leading channel $p[n]$. The maximal lag is limited by a k_{maxlag} constant fixed by the encoder to $k_{maxlag} = 16$ which corresponds to a maximal distance $l = 0.34$ meters between the microphones:

$$k_{maxlag} = \frac{f_s \times l}{340.29}. \tag{4.4}$$

This double-sided correlation process works as follows:

1. The first channel is set as the reference.
2. A standard correlation is performed (4.5) for $k_{max} = k_{maxlag}/2 - 1$ coefficients.
3. The values for the negative lags are computed (4.6) for $k_{max} = k_{maxlag}/2$. These negative lags represent a lead over the reference channel.
4. The channel with the most negative lag is chosen as the leading channel $p[n]$ (negative lag compared to the first channel).
5. The leading channel is set as the reference.
6. The final shift values are determined using cross-correlation (4.5).

$$R_k = \sum_{n=0}^{N-k-1} p[n]\, s[n+k] \tag{4.5}$$

$$R_{-k} = -\sum_{n=0}^{N-k-1} s[n]\, p[n+k] \tag{4.6}$$

The leading channel as well as the individual shift values are stored in the `STREAMINFO` metadata block. The results of this operation on the encoding is the most significant out of all corrections as seen in table 4.3.

Table 4.3: Effects of shift correction on the encoding

Corrections	File Size	Ratio
None	5.27 MiB	36.02 %
Shift (on frame)	5.27 MiB	35.96 %
Shift (on residual)	5.27 MiB	35.97 %

This is probably due to the nature of our recording which was recorded on an 8-channel microphone array and as described in section 2.3, shift correction can have a significant effect on these recordings.

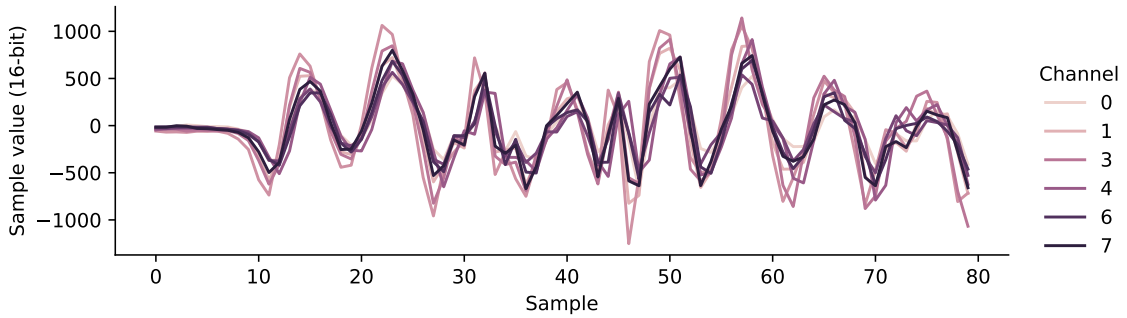

Figure 4.4: Frame with corrected shift

Bias correction

The mean of the signal is subtracted from each sample, a process described in section 2.3.1. This has a negative effect however, since it can happen that this operation can throw the values of the signal out of the valid range of its data type. Although overflows cannot happen during runtime (since we use a larger data type) these larger samples would not fit into the original bit-per-sample precision given by the source file. This is an issue mainly for raw frames and warmup samples since these are stored unencoded. To prevent overflows, these values have this correction later reversed before storing them in the Straw format.

The individual removed bias values are stored in the `STREAMINFO` metadata block.

Table 4.4: Effects of bias correction on the encoding

Corrections	File Size	Ratio
None	5.27 MiB	36.02 %
Bias (on frame)	5.27 MiB	36.01 %
Bias (on residual)	5.27 MiB	36.02 %

As seen in table 4.4, this correction only has a minor role in improving the encoding efficiency. This can be due to our recording mostly already being centered on 0 and not needing much correction. This was also confirmed during debugging where we have determined that for the used development recording, only one channel needed a bias correction. This correction was also only performed with the value of exactly 1 and the other channels were already properly centered on 0.

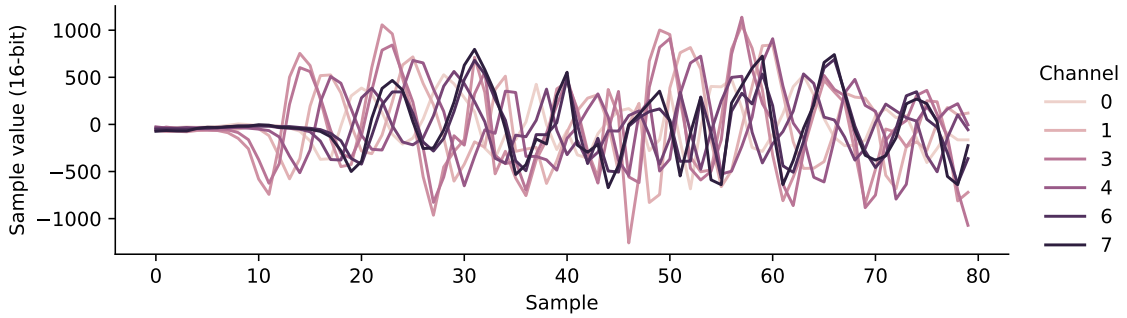

Figure 4.5: Frame with corrected bias

All corrections

After applying all of the above mentioned corrections (including gain correction), the resulting samples indeed seem to resemble each other to a significant degree, as shown in figure 4.6.

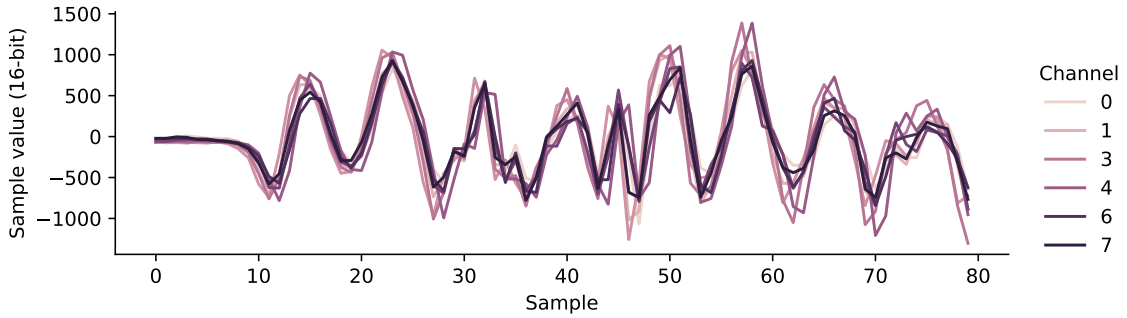

Figure 4.6: Frame with all corrections

Although this works reasonably well on a local scale, storing the additional information needed to reverse these corrections for each frame would cost more memory than it would effectively save, as seen in table 4.5:

Table 4.5: Effects of corrections on the encoding

Correction type	Corrections	File Size	Ratio	Time
None	0 B	5.27 MiB	36.02 %	1.285 s
Global	747 B	5.27 MiB	35.96 %	1.682 s
Local	185.29 KiB	5.28 MiB	36.06 %	2.403 s

storing all the parameters would require more memory that can be saved for each frame in the best case. The used corrections also excluded gain correction since it is not used in a production environment. The shown processing times are obtained in our development environment (AMD Ryzen 3600 with 12 threads) and they also show a significant overhead for local corrections. Although we have to admit that the selected development track already had very similar channels, the corrections did not show a very significant improvement.

For this reason we have chosen to apply these corrections globally. This allows us to sacrifice a very small amount of bytes in the `STREAMINFO` metadata block for a decent improvement in encoding efficiency. Although this does not seem to have a visible effect (as seen in figure 4.7) it does decrease the output file size as shown in table 4.5.

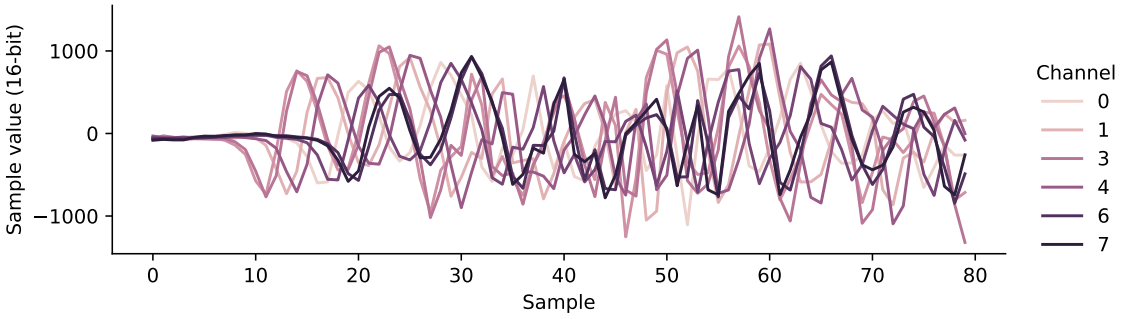

Figure 4.7: Frame with all global corrections

Frames vs Residuals

During development, we experimented with different combinations of corrections, both on frames and on residuals. The implementation of these corrections and the internal data structure of the implemented codec has allowed us to easily switch between global/local and frame/residual corrections. These experiments led us to believe that the most memory and time efficient method involved global corrections of the signal before framing. This is also because of the reasons mentioned above such as the problem with additional per-frame memory requirements.

As a last effort to integrate gain correction into the codec, we also tried applying gain correction on the residuals right before decorrelation. This however also resulted in negative effects on the resulting file size as seen in table 4.1.

Order of operations

The default order of operations in our codec is Shift correction followed by Bias correction. This order is not currently overridable by the standalone reference encoder. When imported and constructed as a class however, the order of operations and the operations themselves can be freely customized. The operations and their order is defined by the `Encoder` constructor parameter `do_corrections: tuple` which has a default value `(„shift", „bias")`.

4.1.2 Blocking

In order to properly process the input signals using linear prediction the signal needs to be sliced into smaller, separate and independent frames. This is performed by slicing the underlying Numpy array into smaller memory views which are then stored inside a pandas DataFrame. More about the internal structure of the reference implementation can be found in section 5.1.2.

Subrame types

Similarly to FLAC, our codec classifies the audio subframes with a type. The basic `SUBFRAME_LPC` and `SUBFRAME_CONSTANT` are analogous to FLAC, while `SUBFRAME_VERBATIM` has been renamed to `SUBFRAME_RAW`.

Our codec also introduces a new type of subframe: `SUBFRAME_LPC_COMMON`. This subframe does not have its own LPC coefficients and related fields, containing only warmup samples and the residual. These subframes are often in groups, with the first one being reclassified to `SUBFRAME_LPC` containing the shared LPC coefficients.

Dynamic blocking

One of our main ideas on how to improve encoding efficiency was to focus on better blocking techniques. We observed that FLAC by default uses fixed size blocks even if the FLAC format supports a dynamic blocking structure. The best block size for this approach seems to be around 2^{12} or 4096 samples which FLAC uses as a default fixed block size.

During development, we also observed that many frames are not uniform, as seen in figure 4.8. The energy of these frames can vary significantly during the duration of the frame. This can potentially have a negative effect on prediction since the predictor has to account for the whole frame and sudden spikes can worsen the performance of the prediction. We also observed that residuals from frames containing both silence (or low value white noise) and speech tended to have larger entropy than pure speech or silence frames. This resulted in a longer Rice encoded bitstream.

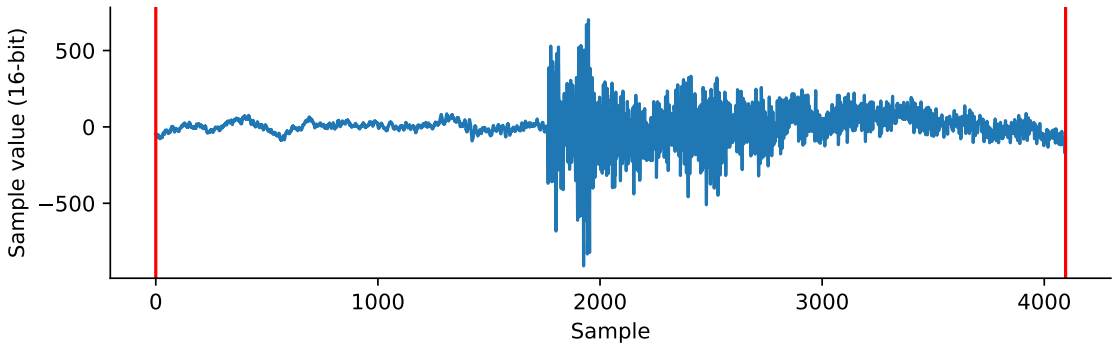

Figure 4.8: Frame with static block size

For this reason, we have decided to make our codec use dynamic block sizes whenever possible. This was doable as both formats (FLAC and Straw A) already support dynamic block sizes. This also meant that no changes were needed for the rest of the encoding process due to the modular structure of our code.

During our experimentation, we came to the conclusion that using a short-term energy estimation [2, page 38] for determining slicing points might be a good way to split the frames. These splits would split the frames into separate but bordering low and high energy frames. We created an algorithm which estimates the ideal slicing points based on the short time energy of the signal. These slicing points are later used for splitting the signals into frames.

This algorithm works as follows:

1. The short-term energy is calculated from the whole signal:

$$E[n] \quad = \quad \frac{1}{N_{res}} \sum_{m=0}^{N_{res}-1} x^2[n\,N_{res} + m] \quad \text{for } n < \frac{N}{N_{res}}. \tag{4.7}$$

   The resolution $N_{res} = 10$ determines the precision by which the slicing points are selected.

2. The threshold value is subtracted from the energies $E_r[n] = E[n] - E_{thr}$.

3. The zero crossings of E_r are marked as potential slicing points.

4. To ensure that the resulting frame size falls between the minimum and maximum block sizes specified by the encoder, some of these potential slicing points are merged and interpolated to prevent too small and too large frames, respectively.

Since the channels are presumed similar, determining the slicing points is done based on the first (main) channel only. Since the corrections were performed before blocking, all the other channels are presumed to be in sync with this channel. In figure 4.9 we can see a frame with highlighted slicing points and slicing in our new blocking design.

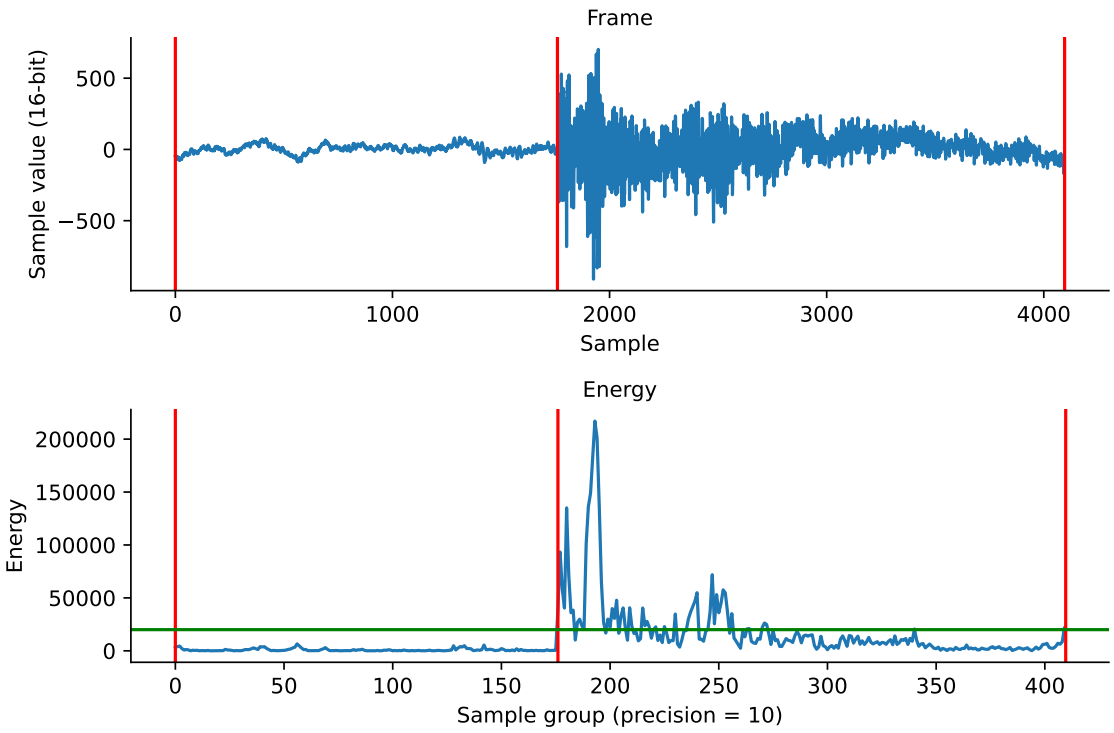

Figure 4.9: Sliced frame and its short-term energy

The energy threshold value $E_{thr} = 20000$ was the result of our experiments with different threshold values and block size combinations to find a combination that would reduce the output size the most. The results of this experiment can be seen in figure 4.10.

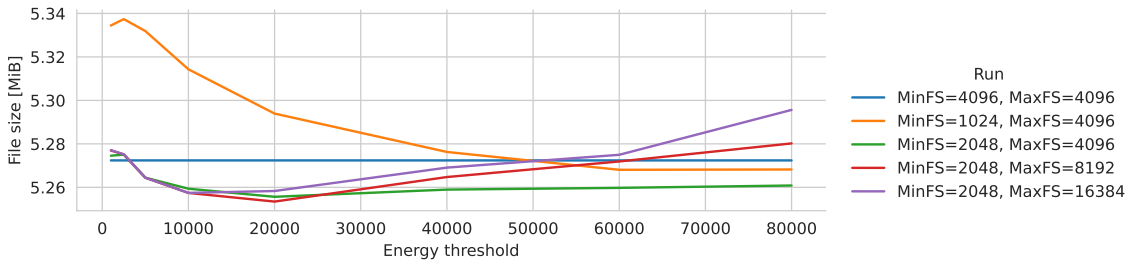

Figure 4.10: Effects of energy thresholds on dynamic blocking

From figure 4.10, we can see that the best results are achieved for an energy threshold of around 20000, although this can vary for different files. This value should fit most cases where the audio is composed mostly of speech. These long recordings often contain areas with mostly low audio level, and areas where speakers are active, which also corresponds to our test recordings.

A better estimation for the E_{thr} value could be based on the dynamic range of the signal, though this is not implemented in our codec.

As seen in table 4.6, this can improve the encoding if a good threshold value is chosen. The results shown in table 4.6 were obtained with a framing threshold value of 20000.

Table 4.6: Effects of frame size on the encoding

Min Size	Max Size	Frames	File Size	Ratio
4096	4096	235	5.27 MiB	36.00 %
1024	4096	654	5.29 MiB	36.14 %
2048	4096	380	5.26 MiB	35.88 %
2048	**8192**	**353**	**5.25 MiB**	**35.87 %**
2048	16384	344	5.26 MiB	35.90 %

Overriding the default parameters

The encoder specifies default values for frame size limits, framing threshold and resolution as shown in this section. These values can be overridden however, by supplying the standalone encoder with the arguments `-min-frame-size`, `-max-frame-size`, `-framing-threshold` and `-framing-resolution`.

The `Encoder` class also exposes these parameters to its public interface and they can be modified during the lifecycle of the encoder. These parameters are only taken into accord during blocking, which is done in `Encoder.load_file()`.

Subframes

The slicing points only define the frame borders. The size of subframes withing a frame remains equal. When slicing, the subframes are presumed to be in sync. This is due to the blocking process being performed after the correction have been applied. These subframes are then processed in groups and parallelized. More about the parallelization process can be found in section 4.1.6.

4.1.3 Linear prediction

Linear prediction is the main component of our codec, as it was for FLAC. This section attempts to explain how the prediction process works in the reference implementation of our codec. The prediction process is modelled as a separate block that takes a set of LPC subframes belonging to one frame as input and produces the quantized LPC coefficients (QLP) and residuals an outputs. The length of each subframe is equal, although the frame size can vary. All LPC related operations are implemented in the subpackage `straw.lpc`. A simplified block diagram of this process can be seen in figure 4.11.

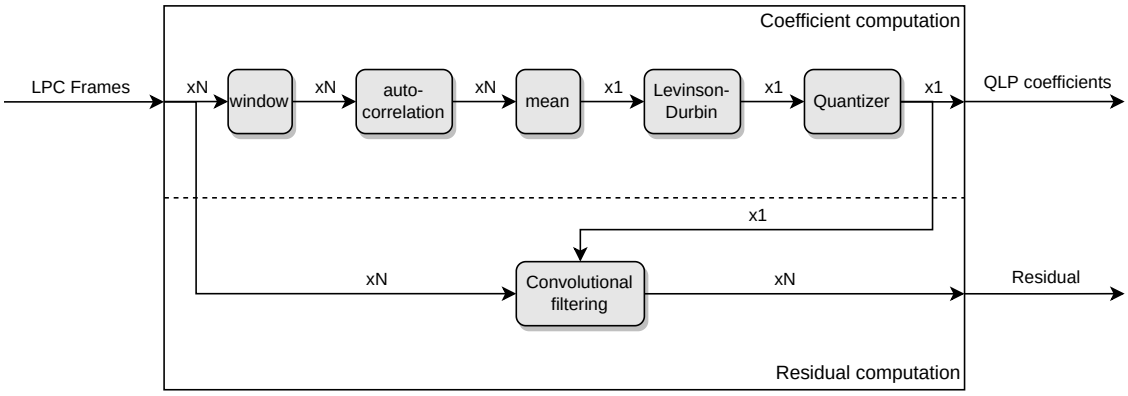

Figure 4.11: Block diagram of the prediction process

The exact process is as follows:

1. The subframes are converted to a normalized floating point format since rounding errors during this process can be ignored.
2. A window function is applied to each subframe - this is optional in FLAC
3. For each subframe, $P + 1$ autocorrelation coefficients are computed
4. The autocorrelation coefficients are averaged over all the subframes
5. The LPC coefficients are computed using `scipy.linalg.solve_toeplitz`. This function implements the efficient Levinson-Durbin recursion method, described in section 2.2.4.

The windowing is done using a Tukey window, also known as a tapered cosine window. This significantly improves the accuracy of the prediction as can be seen in table 4.7.

Table 4.7: Effects of windowing functions on the file size

Window	File Size	Ratio
None	5.36 MiB	36.59 %
Hamming	5.27 MiB	35.98 %
Tukey	5.27 MiB	35.96 %

Order

The LPC order is selected based on the calculated LPC error of each set of LPC coefficients. Straw A as well as FLAC [11] formats support an order of up to 32, while order 20 proved to be the most effective when a fixed order and one set of coefficients was used (for subframe type `LPC_COMMON`).

Quantization

The LPC coefficient quantization process is completely adapted from FLAC and rewritten in Cython.

Coefficient quantization methods

During development, one of the ideas for possible improvements over FLAC was the use of alternative quantization methods which would be less influenced by quantization errors. This has, however proved to make such an insignificant difference, as seen in table 4.8, that keeping these methods in the reference encoder was useless and we have abandoned this idea in favor of direct quantization. This might be caused by our relatively high precision of 12 bits for quantizing each coefficient. These methods might have an impact if a smaller precision is used for storing the quantized coefficients. Reducing the quantized coefficient precision and using one of these methods also would not have saved many bytes since the coefficients themselves require only a small percentage of the overall file size, 82.73 KiB out of 5.25 MiB (1.5 %).

Table 4.8: Effects of quantization types on the file size

Quantization type	File Size	Ratio
Direct	5.25 MiB	35.87 %
PARCOR	5.25 MiB	35.87 %
LSF	5.25 MiB	35.87 %

Due to these results, we abandon these alternative quantization methods and focus on other aspects of our codec.

Filter stability

A stability check is performed after the LPC coefficients were calculated to prevent invalid frames due to decoding issues caused by unstable IIR filtering. This is performed by finding the complex roots of the polynomial represented by the LPC coefficients and verifying that each pole falls inside the unit circle as shown in figure 4.12.

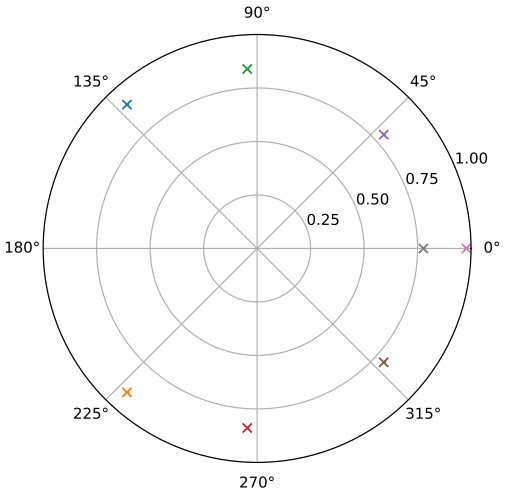

Figure 4.12: Complex roots of a stable LPC filter

If the filter is deemed unstable, the subframes are reclassified as raw frames and encoded accordingly.

4.1.4 Decorrelation

Decorrelations are operations for separation of common values from the channels performed on the residuals left after the prediction step. Our codec currently offers two types of decorrelation, although the use of the conditional subtraction method is discouraged due to its inferior performance compared to mid-side decorrelation.

Iterated mid-side decorrelation

This type of decorrelation is based on transforming two channels, L representing the left and R representing the right channel, into two distinct signals, one representing their sum (or more efficiently average), and the other the differences between them. This decorrelation method is also implemented in FLAC, though its use must be specifically requested by the user.

Our implementation extends this technique for more than two channels by doing the decorrelation in iterations when applicable, further decreasing the intensity of the residuals by separating the entropy from the channels. This method ensures that the similarity between the results of this step have minimal correlation between each other.

The mid-side transformation is described by:

$$d_{dif}[n] = s_L[n] - s_R[n] \tag{4.8}$$

$$d_{mid}[n] = \lfloor (s_L[n] + s_R[n])/2 \rfloor. \tag{4.9}$$

Since we only need the average of the two channels and we are dealing with integers, the MID signal can be divided, or more efficiently shifted to the right by 1 bit. The reverse of this process is shown in section 4.2.2.

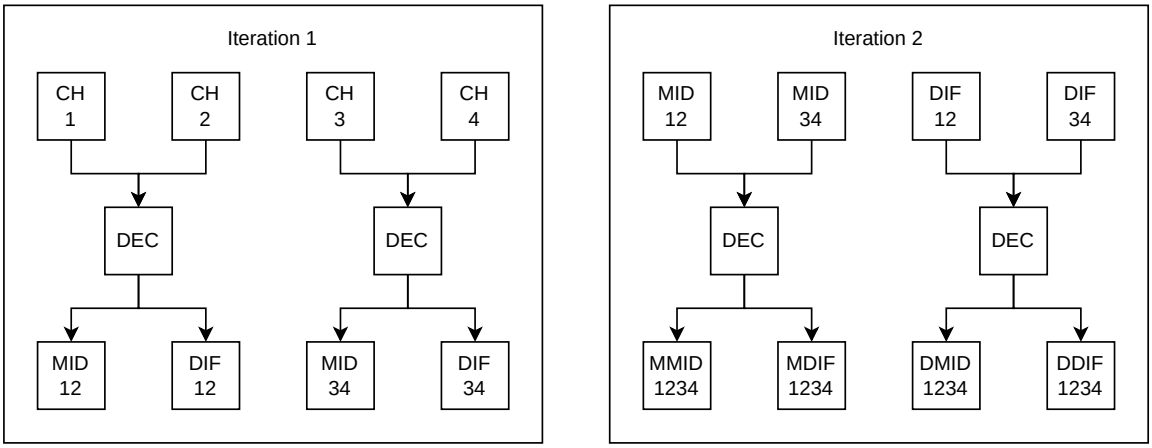

Figure 4.13: Iterated mid-side decorrelation

The effects of this technique on the encoding can be found in table 4.9.

Decorrelation by subtraction

Our initial idea for interchannel decorrelation was a simple subtraction since we assumed the resulting residuals would resemble each other in a way that could be easily leveraged.

This did not happen however, as seen in table 4.9. The average difference between the residuals in low-intensity areas was often larger than the residuals themselves. This resulted in our full-frame subtraction to be inefficient to a degree that it even decreased the overall encoding efficiency.

We observed, however, that the residuals are often very similar in places with higher intensity and frames where this happened tended to have a better decorrelation efficiency. The resulting implementation behaved in the following way:

$$d_i[n] = \begin{cases} r_i[n] - r_0[n] & \text{if } i > 0 \text{ and } Var(r_i - r_0) < Var(r_i) \\ r_i[n] & \text{else} \end{cases} \tag{4.10}$$

Since the size of the Rice coded residual depends on the signals variance, we decided to only store this decorrelated version if its variance is smaller than the variance of the original signal. This variance check ensured that this step would not increase the signals intensity and would have a beneficial effect on the eventual Rice coding stage. This required a single bit in the subframe header which indicated whether the frame was decorrelated.

This seemed to have a good effect on the overall output size as seen in table 4.9. This decorrelation method was however abandoned due to mid-side decorrelation having a much better performance (mainly in its iterated form).

Performance comparison

Table 4.9: Effects of decorrelation methods on compression

Decorelation method	File Size	Ratio
None	5.28 MiB	36.06 %
Sub	5.30 MiB	36.18 %
Sub (with variance check)	5.27 MiB	35.97 %
Mid-Side	5.25 MiB	35.86 %

As seen in table 4.9, the best performing decorrelation technique is mid-side. For this reason, this decorrelation method is enabled by default in the reference implementation of our codec. All residuals are considered to be in a decorrelated state when decoding.

4.1.5 Rice coding

Some frames may contain spikes of higher values which would be inefficient to code with a small Rice parameter due to the nature of this coding method [7]. FLAC partially solved this issue by using a variable width partitioning scheme (described in section 2.2.5) which performs adequately for most inputs with uniform probability distributions.

Our approach to this problem was different though. Straw encodes the residual in one part and uses a starting Rice parameter which is estimated based on the first samples of the residual to encode the frame. This parameter can vary during the encoding process depending on the values of the last k_{resp} encoded samples, k_{resp} being the responsiveness constant (similarly to linear prediction it does not use the current sample). The behavior of the Rice parameter m can be seen in figure 4.14. It is evident that this parameter is always a power of two.

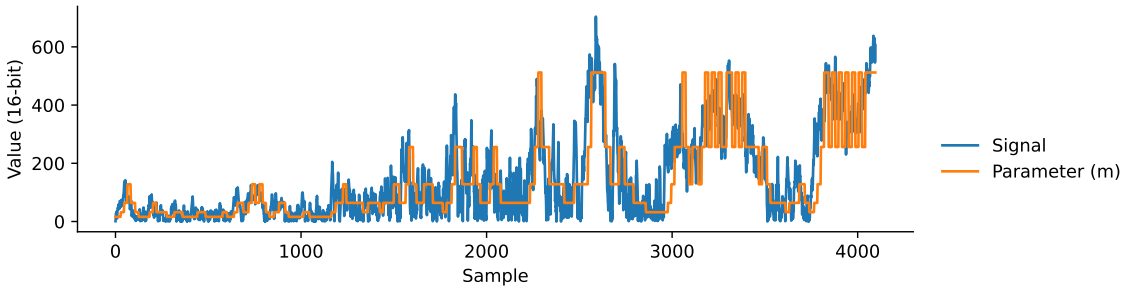

Figure 4.14: Behavior of Rice parameter during encoding

The algorithm of this adaptive process is an extension of the original algorithm described in section 2.2.5, with modifications shown in bold:

1. **Initialize a** *scale* **variable to 0.**
2. $s[n]$ is interleaved.
3. The quotient $q = s/m$ is calculated. Since we are working with integers, q is rounded down.

4. q binary zeroes are written to the bitstream.
5. A binary one is written to the bitstream.
6. The k last bits of $s[n]$ are written to the bitstream.
7. **Increment** k **if** $scale > k_{resp}$**, else if** $k > 0$ **decrement** k**.**
8. **Recalculate** $m = 2^k$**.**
9. **Increment** $scale$ **if** $s[n] > m$**, else if** $s < m$ **decrement** $scale$**, else** $scale = 0$**.**

The values of responsiveness can be adjusted using the `-rice-responsiveness` argument or by presenting the `Encoder` constructor with a responsiveness value. Through experiments, as seen in figure 4.15, we have found $k_{resp} = 20$ to be the best value for this constant.

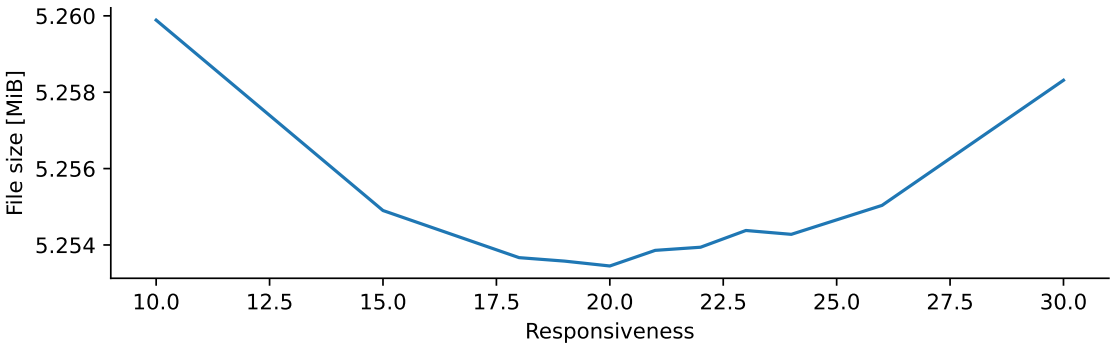

Figure 4.15: Effects of responsiveness on the file size

The parameter described in figure 4.14 is the Rice m parameter, however we only store the k parameter. The relationship between these parameters is simple:

$$m \quad = \quad 2^k. \tag{4.11}$$

The decoder can be simply adapted to this method since the parameter only depends on the last N decoded samples. All of these methods depend on choosing an ideal Rice parameter in the first place.

This adaptive coding method shows a significant improvement over a static Rice parameter. Most importantly, this method is more efficient for residuals which do not have a uniform distribution or have areas with significant spikes.

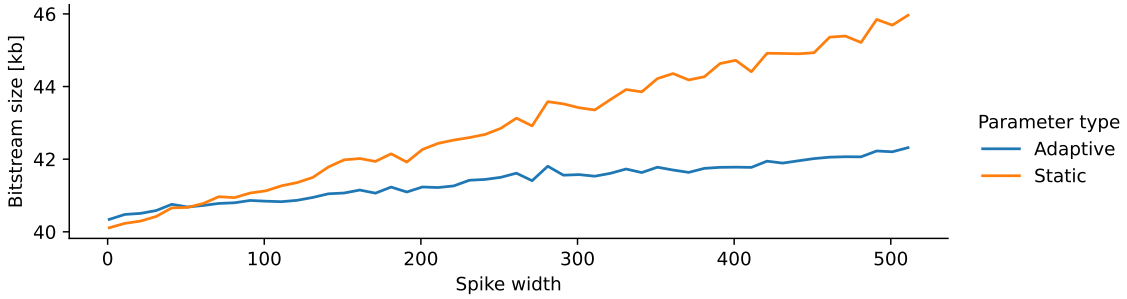

Figure 4.16: Static vs adaptive Rice coding

In figure 4.16, we introduced a spike of a given width to the mid-section of a random signal. From this figure, we can see that the adaptive coding process has much better stability and efficiency for residuals with non-consistent variance where the difference can be as high as 10% in certain cases.

We can also see that when the audio frame has a uniform energy distribution, the adaptive coding scheme may fall behind its static counterpart. This may however be ignored since most frames are not ideal and therefore this adaptive coding scheme is better in most cases.

Performance

From table 4.10, we can see a significant increase in encoding efficiency over encoding with a static Rice coding scheme. The audio file on which we developed this codec had significant energy differences in a few frames, these frames might have been encoded more efficiently with our coding scheme thus reducing the overall file size.

Table 4.10: Effects of adaptive Rice coding the file size

Rice coding type	File Size	Ratio
Static	5.53 MiB	37.76 %
Adaptive	5.25 MiB	35.86 %

4.1.6 Parallelization

The encoding process is parallelized on the frame level using a thread pool. After blocking the signal into individual subframes, all subframes from the same frame are grouped together and processed in parallel. This allows for an ideal environment for parallelization where each thread is computing a group of subframes.

Since the frames are independent of each other, this does not cause shared resource access issues such as race conditions. This also means that each thread has a similar memory range in which it operates since the samples are stored as a C-contiguous array where for each sample, there are contiguous subsamples.

4.2 Decoding process

The decoding process of our proposed codec also draws much similarity to FLAC in regards to its architecture. The decoding is, similarly to FLAC, comparatively faster than encoding as shown in section 6.2. The block diagram of the decoding process can be seen in figure 4.17.

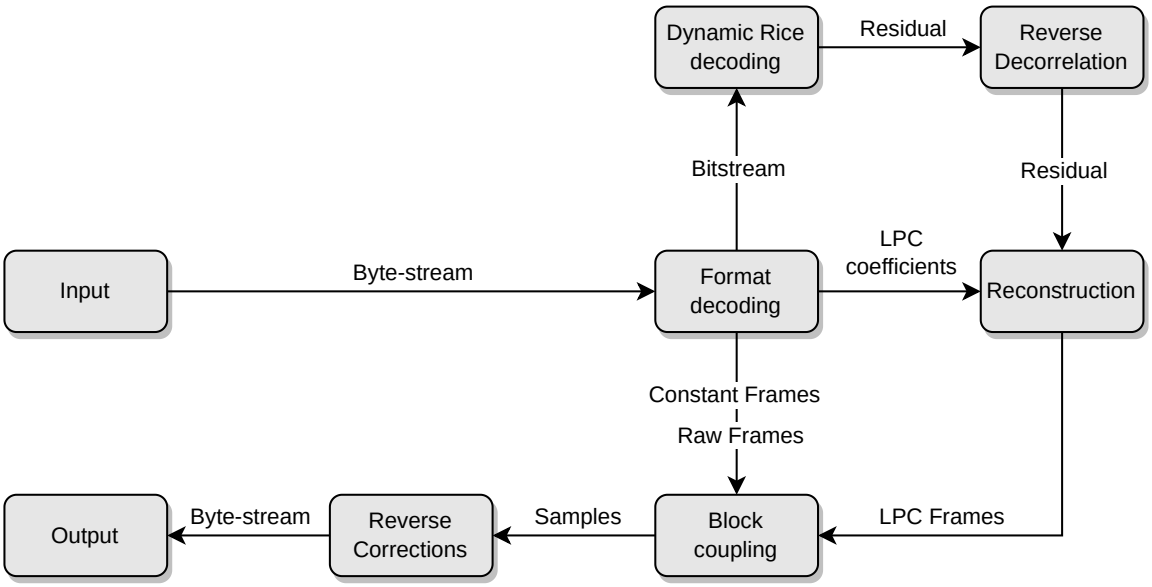

Figure 4.17: Block diagram of the Straw decoder

As we can see from figure 4.17, much of the decoding process is similar to the decoding process of FLAC (shown in section 2.2.3) but extended with correction and decorrelation reversal as well as a different Rice coding scheme. All of the mentioned changes compared to FLAC are described in section 4.1 in more detail.

Encoder-Decoder compatibility

The encoder and decoder are designed to work independently from each other. This is desirable, since the `.straw` input file contains all information the decoder needs to restore the original audio file. This information also includes values which can be checked to verify the data integrity, as further described in section 4.2.5.

This also removes the need for the user to specify the encoding parameters to the decoder, since the parameters that the encoder used are stored in the input file.

Parallelization

The decoding process is from its nature more time and memory efficient than the encoding process. Due to this fact, parallelization of the decoding process is not strictly necessary since the decoder can process even large input files quite efficiently. More about the results can be found in section 6.2.

4.2.1 Rice decoding

The Rice-encoded residual is stored in the RESIDUAL section of each subframe (see appenix A). Each residual block has stored its own starting 4-bit k Rice parameter. This parameter will most likely vary during the decoding process. Note however, that this parameter can exceed this 4-bit value during encoding and decoding. The value of the responsiveness constant k_{resp} can be acquired from the **STREAMINFO** metadata block.

During decoding, exactly the same algorithm for updating the Rice parameters k and m is applied as described in section 4.1.5.

4.2.2 Reverse Decorrelation

After the residual has been Rice decoded, the applied decorrelation needs to be reverted. This results in the original residuals, which can be then used for reconstructing the original signal along with the LPC coefficients.

The reverse of mid-side decorrelation is described by:

$$s_L[n] = d_{mid}[n] + \lfloor d_{dif}[n]/2 \rfloor + (d_{dif}[n] \bmod 2) \tag{4.12}$$

$$s_R[n] = d_{mid}[n] - \lfloor d_{dif}[n]/2 \rfloor. \tag{4.13}$$

This step is also done in an iterated manner, see section 4.1.4, with the order of iterations being exactly the reverse as used for decorrelation.

4.2.3 Reconstruction

The reconstruction is done using a modified IIR filtering, exactly as described in section 2.2.4. This step is mostly identical to reconstruction performed by the FLAC decoder. This filtering method is implemented as a compiled function in Cython.

In contrast to prediction (see section 4.1.3) the decoding is done in-place, since we know that reconstructing the frame using the given residual and LPC coefficients is required. This was not the case for the encoder which, after prediction, had to check if the variance of the residual was smaller and also ensure that the generated bitstream was smaller than a frame encoded as `SUBFRAME_RAW`. This prevented the encoder from doing prediction in-place and required almost twice as much memory as the decoding process for the same input.

The output of this step can be considered mostly similar to the source signal, although the corrections performed by the encoder still need to be accounted for.

4.2.4 Reverse Corrections

Reversing the corrections is the last step before the signals can be considered intact. Whether a correction reversal will be performed depends on flags and correction values stored in the `STREAMINFO` block. The corrections are then reversed in the reverse order that they have been performed.

Gain correction

The gain correction value is stored in quantized format, so it needs to be converted to a floating point format before proceeding (4.14). The k_{quant} constant is the same quantization constant as described in section 4.1.1.

$$g = \frac{g_q}{k_{quant}}. \tag{4.14}$$

The gain correction is reversed by dividing each channel by the gain correction value of the given channel 4.15.

$$s[n] = \lfloor s_{eq}[n]/g \rceil \quad \text{for } g \geq 1.0. \tag{4.15}$$

When a gain correction value is equal to 1.0, the gain correction of that channel is skipped for performance reasons.

Shift correction

Shift correction can be reversed by simply inserting the removed samples to the original signals. In the implementation, this is done efficiently by first allocating memory for the samples and then shifting the memory views of each frame by the given lag. This results in reserved space at the start and end of each signal equal to the missing samples, which are then loaded from the given metadata block.

Bias correction

Bias correction is the easiest to revert, by simply adding the bias removed by the original corrections. This is done for each sample.

4.2.5 Integrity checks

To verify the integrity of the decoded data, our proposed codec offers the same methods that FLAC employed in order to verify the integrity of each frame and the decoded samples.

CRC checks

Each frame header has a CRC-8 field for verifying the integrity of the header. The frame footer contains a CRC-16 field for verifying the integrity of each frame, including the frame header and subframes. This also includes the byte-alignment appended after the subframes

Losslessness check

To verify the losslessness and the overall integrity of the decoded data, an `md5` sum is performed on the decoded samples and compared to the value stored in the `STREAMINFO` metadata block.

Chapter 5

Implementation

An application suite containing a library and a self-contained launcher script were created to demonstrate the functionality of our proposed codec. This codec can be compared to the reference codec that FLAC offers as a baseline implementation.

We have chosen to implement this library in Python for its versatility and ease of integration. Since most applications dealing with speech recognition and synthesis are nowadays also written in Python, we have seen the potential to implement this codec in a Python package that could easily be integrated into other projects dealing with audio processing.

This library consists of the main package named straw (the code name of the project and the format) and numerous subpackages which usually deal with one specific issue and perform a specific task.

The project and its source code are available as open-source on GitHub [1].

5.1 The library

As most Python project, ours is structured as a package with multiple subpackages. Subpackages are self-contained directories containing Python modules (`.py` files) which can be called from any other subpackage or module. Each subpackage has its own interface consisting of top-level classes or functions from the given subpackage. This interface can be found in the `__init__.py` file in each subpackage directory.

Base classes

Most of the subpackages have a file `base.py` containing base classes. These classes are not used directly, but rather inherited by specific classes in the subpackage. These specific classes often have common properties while most of their implementation is completely different.

Extensions

Subpackages may also contain Cython extensions. These extensions use the format `ext_{subpackage_name}.pyx`. Extensions implement algorithms which would otherwise be slow, inefficient or too complicated in pure Python. Memory views are used extensively by these extensions to efficiently access data in NumPy arrays.

[1] https://github.com/KLZ-0/straw/

5.1.1 Subpackages

The following subpackages make up the main part of the library.

straw.codec

The codec subpackage contains the main encoder and decoder classes. This subpackage offers the most readable top-down implementation of both the encoding and decoding processes. It often calls different operations from the other subpackages and can therefore be considered a top-level subpackage.

straw.compute

Contains the class `ParallelCompute`. This class implements NumPy and Pandas parallelization. These parallelizations can be performed on different levels, such as numpy, DataFrame or Series (one row of a DataFrame).

straw.correctors

As the name suggests, this subpackage contains all classes which deal with corrections. Each of these corrections is located in a separate file. All corrections have a common base class and a uniform interface. This subpackage also contains the methods for residual decorrelation.

straw.io

The formatters and format definitions reside in this subpackage. This subpackage is used for loading and writing the straw format. There is also a basic FLAC reader/writer implementation since our codec used to be mostly compatible with FLAC (excluding the metadata blocks and a few more caveats).

This subpackage also contains the SlicedBitarray class. This class offers efficient bit-level access to the buffer of the input file when decoding. More about the bitarray module can be found in section 5.3. The encoding format-specific constants can also be found in this subpackage, these are used for uniform bitstream loading and writing.

This subpackage also contains an extension which implements efficient reading and writing of raw subframes. This is due to numpy not handling well bit widths other than 8/16/32/64.

straw.lpc

This subpackage contains operations specific to linear prediction. The interface of this subpackage exports functions which are defined in `wrappers.py`. These functions all expect input the standard DataFrame format. These functions call numpy level functions from `steps.py` which implement operations such as LPC computation, quantization, prediction and reconstruction as well as a custom implementation for autocorrelation. This subpackage contains an extension which implements efficient quantization and reconstruction algorithms.

straw.rice

The main interface to the Rice coder, the Ricer class is located in this subpackage.

We could not find any external library which would offer an efficient implementation of Rice coding. The Rice encoder had to be re-implemented in an efficient manner, which is almost impossible in pure Python. For this reason, we have chosen to use Cython for implementing the coding method as efficiently as possible.

The first implementation used calls to bitarray functions from Cython itself. This was was later replaced by the buffer protocol and direct byte-level access due to the delays caused by call delays from the Python interpreter. This buffer is later exported and a bitarray is constructed from it. This resulting bitarray is then written into the `RESIDUAL` section of the Straw format.

straw.utils

This subpackage contains utilities that could not be linked to any specific subpackage.

straw.static

This is not actually a subpackages but rather a module. This submodule contains static definitions for the internal structure of the codec.

straw.straw

Similarly to `straw.static`, this is not a subpackage but rather a compatibility module. This module contains functions `read()` and `write()` which work analogously to `soundfile.read()` and `soundfile.write()`. This module also contains a helper function `run()` which, when called from `main.py` runs the encoder or decoder as a standalone application.

figures

The figures subpackage is used for drawing figures and tables used in this thesis. It is not part of the straw package, but rather a separate package that imports functions and methods from the straw package.

This subpackage is not a part of the encoding/decoding process and is normally not loaded during execution. It can be activated by supplying `main.py` with the `-figures` argument and giving it an input file. The figures used in this thesis were generated by supplying the script with the `1min.wav` (see section 3 for more information about this recording).

5.1.2 Internal data structure

The codec internally works with pandas DataFrames and DataFrame slices. These DataFrames do not copy the data but only contain numpy memory views [2]. These memory views point to certain parts of the underlying numpy array where the actual samples are stored. This abstraction allows for easy parallelization while maintaining relatively low memory requirements.

Each subpackage exports classes which work on this level and accept this DataFrame format.

[2] https://docs.python.org/3/c-api/memoryview.html

Data types

Straw officially only supports the 16 bit pulse-code modulation (PCM) format. The 24 and 32 bit PCM formats are also available, but they are untested and should be avoided due to performance and/or memory issues.

5.1.3 The format

The FLAC format is well defined and has became the standard over the years. This format is not well suited to our needs however, since it has certain fields which are redundant for us. For this reason we decided to define our own format based on the original FLAC format. The full format bitstream specification can be found in appendix A.

The format is overall similar to that of FLAC. The `.straw` file is written after each frame has been processed. Frames consist of a group of subframes equal to the number of channels. These subframes represent the sound data, each of which has its type specifying how will it be encoded, see section 2.2.1.

Our format adds a special type of LPC subframe, named `LPC_COMMON`. This subframe does not have its own LPC coefficients. The first subframe from a group of these frames is changed to a standard LPC frame. This one frame then contains one set of LPC frames which are used to for the prediction process of every subframe of that group.

This extension allows us to use one set of LPC coefficients for a whole frame. This can lower the output file size for multi-channel files, while not impacting the quality of the prediction significantly.

The format also supports arbitrary number of channels. This is a significant extension compared to FLAC, which has an upper limit of 8 channels. It allows for effective encoding of signals from multiple channels without requiring multiple output files. The fields for the channel and frame numbers are dynamic in size. This is due to using a special form of UTF-8 encoding, which is extended to handle larger input. This method is also used by FLAC for frame numbers [11].

The reference implementation does not implement every aspect of the format though, namely the `METADATA_BLOCK` section. FLAC supports different metadata blocks for a number of applications. The Straw format specification supports these metadata blocks identically to FLAC, but their implementation was not critical for our application at the moment. For this reason they have been mostly left out of the reference codec.

Sampling frequency can be any integer up to 2^{20} Hz, or up to approximately 1 MHz.

5.1.4 Interface

The library offers global convenience functions `read()` and `write()`. These are analogous in function and semantics to `soundfile.read()` and `soundfile.write()`. For more fine-tuned control, the `Encoder` and `Decoder` classes can be utilized.

The input file `existing_file.wav` is a multi-channel WAV file.

```
import soundfile as sf
import straw

# Encoding
# Reading a wav file and writing a straw file
data, samplerate = sf.read("existing_file.wav")
straw.write("new_file.straw", data, samplerate)
```

```
# Decoding
# Reading a straw file and writing a wav file
data, samplerate = straw.read("existing_file.straw")
sf.write("new_file.wav", data, samplerate)
```

The use of the `Encoder` and `Decoder` classes should be preferred for Python projects that wish to integrate Straw as an IO library. This is advisable since the Python interpreter does not need to be reloaded.

5.2 Standalone executable

As well as a library, the reference implementation contains a wrapper script around the `Encoder` and `Decoder` classes that can be used as a standalone executable.

The currect installation and usage methods are also detailed in the supplied `README.md` markdown file.

5.2.1 Installation

The current version can be acquired from the Git repository, mentioned at the start of chapter 5 or from The Python Package Index (PyPI) [3]. This installation step is recommended but not necessary. This is further explained in section 5.2.2.

PyPI

```
# system-wide
pip install straw-codec
# local
pip install --user straw-codec
```

Directly from source

```
# must be run in the directory where setup.py is located
pip install .
```

5.2.2 Usage

If the installation was done locally, the executable is installed to `/.local/bin`. To use the executable from these installations, this directory must be added to `$PATH`.

```
export PATH=$PATH:~/.local/bin
```

After installation, the script should be executable from a shell:

```
# Show help and available options
straw -h
# Encode
straw -i /path/to/input.wav -o /path/to/output.straw
```

[3] https://pypi.org/project/straw-codec/

```
# Decode
straw -d -i /path/to/input.straw -o /path/to/output.wav
```

The launcher script offers much of the same functionality as the `Encoder` and `Decoder` classes themselves.

The current implementation of the standalone executable does not contain an option to load a multi-channel recording from multiple single-channel files, although it could be extended with that functionality relatively easily.

5.3 External libraries and dependencies

Pandas and NumPy

The encoder and decoder both utilize Pandas and NumPy to a very high extent. Operations are implemented on different levels depending on the context, for example wrappers which apply certain operation to a whole DataFrame slice while lower level functions do numpy-level operations.

Cython

We used Cython for implementing operations which are too heavy or inefficient in pure Python or those which do not have an efficient pure Python implementation such as Rice coding.

Bitarray

For writing bitstreams, Python does not offer a clear solution, so we used a library called bitarray, this library offers an object type which efficiently represents an array of booleans [4].

The io and ricer subpackages heavily use this library for bit operations. The drawback of this library is that while it offers much faster operations than pure Python, frequent calls from the Python interpreter itself still consume a lot of CPU time for operations like Rice coding or writing raw frames. For heavier operations, we utilized raw Cython code sometimes in combination with shared buffers from bitarrays thus mitigating this issue and achieving much better performance.

The bitarray library also offers access to the buffer protocol which allows us to overlay a standard file buffer over a bitarray and gain efficient bit-level access without storing the whole file in memory.

SciPy

SciPy is used for a few signal processing computations such as `scipy.linalg.solve_toeplitz` and windowing `scipy.signal.get_window`.

Other

There are a few libraries for very specific purposes, that are used in only a few places:

- **Crcmod** has been used for performing cyclic redundancy checks (CRC) on frames and their headers.

[4] https://pypi.org/project/bitarray/

- **Tqdm** as a progress bar for the standalone encoder/decoder script.
- **Soundfile** for storing and loading wav files, as well as for compatibility. reasons.

Optional

Matplotlib and Seaborn are used for plotting the figures used in this thesis, imports of these libraries are specific to the `figures` subpackage and are not mandatory unless figure generation is required.

Chapter 6

Testing

The final chapter of this thesis deals with tests performed on the implementation of our codec. As part of this testing process we also compare the performance of our codec with the FLAC reference encoder.

6.1 Testing conditions

For uniform results we have chosen the Merlin server [1] as our primary testing environment. Although we used Merlin, the test results shown in this chapter should be reproducible on any other machine.

Tools used

The version of FLAC in our development environment was 1.3.4. The version of FLAC on Merlin was 1.3.0. Unless specified otherwise, the FLAC codec was invoked using its default compression level. The FLAC analyzing option (`flac -a`) proved to be very helpful during development and testing.

For combining WAV files, we used `sox` [2] in version v14.4.1. This tool proved to be very useful for combining multiple WAV files into one single multi-channel file.

Performance

In most tests, FLAC performed significantly better in terms of encoding time. Straw often took more than 10x longer to encode the same file, as seen in table 6.1.

Table 6.1: Encoding times for the development recordings from the AMI corpus (EN2001a)

File Name	Duration	FLAC	Straw	Difference
1min.wav	1:00	0.33s	2.22s	6.72 ×
10min.wav	10:00	3.88s	20.70s	5.33 ×
AMIa_a1.wav	1:00:00	21.20s	233.31s	11.00 ×

Although the encoding time is an important part of every codec, our priority was on developing the encoding process. A few time-costly operations are still performed in pure

[1] https://merlin.fit.vutbr.cz/
[2] http://sox.sourceforge.net/

Python. Rewriting these critical parts of the codec could have resulted in better encoding times.

Memory usage may also become an issue. The encoder may use multiple times more memory than the file which is being encoded. We observed memory spikes of around 10x the size of the original file. This has to be taken into account when performing any kind of encoding with the reference codec. Parallelization does also have an impact on this issue, since it results in multiple temporary arrays being allocated at the same time. If experiencing memory issues, parallelization can be turned off with the `-no-parallel` encoder option.

6.2 Achieved results

In this section we explore the differences in the compression efficiency of Straw and FLAC. We expect Straw to perform better on multi-channel audio as well as having similar performance on single-channel audio.

Development recordings

For most of the development process, we used the recordings `1min.wav` and `10min.wav`. We were able to achieve a considerably better compression ratio than FLAC for these files, as seen in table 6.2.

Table 6.2: Test results for the development recordings from the AMI corpus (EN2001a)

File Name	CH	FLAC	Straw	Difference
1min.wav	8	37.26 %	35.86 %	-3.90 %
10min.wav	8	34.18 %	33.12 %	-3.20 %

AMI corpus

The first data set we tested was the extension of the development tracks from the AMI corpus, namely `EN2001a`. The results for this data set can be seen in table 6.3. We experimented with both single-channel as well as with multi-channel tracks for comparison.

Table 6.3: Test results for the files from the AMI corpus (EN2001a)

File Name	CH	FLAC	Straw	Difference
AMIa_a1.wav	8	34.02 %	33.17 %	-2.56 %
AMIa_a2.wav	8	33.96 %	33.30 %	-2.04 %
AMIa_a1_ch0.wav	1	32.69 %	32.32 %	-1.14 %
AMIa_a2_ch0.wav	1	31.07 %	30.64 %	-1.27 %

The results from table 6.3 show that Straw outperformed FLAC by an average of 2.30 % for multi-channel, and 1.21 % for single-channel compression.

The next data set we have examined was the meeting recording `EN2001b`.

Table 6.4: Test results for the files from the AMI corpus (EN2001b)

File Name	CH	FLAC	Straw	Difference
AMIb_a1.wav	8	36.02 %	35.68 %	-0.95 %
AMIb_a2.wav	8	34.46 %	34.31 %	-0.44 %
AMIb_a1_ch0.wav	1	34.68 %	34.78 %	+0.29 %
AMIb_a2_ch0.wav	1	34.69 %	34.93 %	+0.69 %

The results in table 6.4 indicate the advantage of Straw for encoding multi-channel audio. From these results we see that Straw still beats FLAC at multi-channel encoding by an average of 0.70 %. This cannot be said for single-channel tracks however, where Straw performed poorly compared to FLAC, being outperformed by 0.49 % on average.

CHiME corpus

As described in section 3.2, the CHiME5 corpus consists of multiple 4-channel recordings.

Table 6.5: Test results for the files from the CHiME5 corpus

File Name	CH	FLAC	Straw	Difference
S21_U01.wav	4	56.68 %	56.41 %	-0.48 %
S21_U02.wav	4	65.89 %	65.88 %	-0.02 %
S21_U03.wav	4	47.25 %	46.82 %	-0.92 %
S21_U04.wav	4	50.69 %	49.96 %	-1.46 %
S21_U05.wav	4	48.37 %	48.53 %	+0.33 %
S21_U06.wav	4	47.88 %	47.51 %	-0.78 %

Table 6.5 shows that even though the recordings from the CHiME data set only have 4 channels, Straw still manages to outperform FLAC by an average of 0.55 %.

6.3 Testing summary

From our tests, we can see that Straw performs very similarly to FLAC, with an average improvement of around 1 % over FLAC. This improvement can be attributed mostly to space saving caused by the use of common LPC coefficients as well as to our adaptive Rice coding scheme. This can also be seen in table 6.3, where we see an improvement even for single-channel audio.

Not every test showed a significant improvement however, and the results we got were not completely up to our expectations. Straw performed best on the development recordings, as seen in 6.2, which shows that the parameters of the encoder have been optimized solely for these recordings. We believe that using estimation methods for determining some of the parameters could improve the overall encoding efficiency on other data sets.

Chapter 7

Conclusion

The goal of this thesis was to propose and implement a codec which would use the redundancies present in multi-channel audio.

To reach this goal, we proposed multiple solutions to increase the similarity between channels. These corrections were later implemented along with a mid-side decorrelation method. Apart from these corrections, we extended the FLAC codec by using a dynamic blocking method. The slicing points for these new blocks were determined by the energy boundaries of the the source recording. The linear prediction process used in FLAC was extended by using a common set of LPC coefficients for multiple channels. The last change to the FLAC specification was the introduction of an adaptive Rice coding method.

Our experiments in chapter 6 show that the resulting compression efficiency is mostly similar to FLAC, with better compression for multi-channel audio. Although these results are positive, they did not meet our expectations. We thought that after the prediction stage, the residuals would be similar and could be decorrelated easily. This did not happen however, and decorrelating these signals did not result in a significant compression improvement.

We found that obtaining a compression ratio better than FLAC is a fairly hard task. While our experiments and tests proved to not show much improvement over FLAC, we gained a significant amount of experience regarding the audio compression field. We expect to continue the work on this codec even after this thesis is submitted, and expect to beat FLAC with a better margin in the future.

7.1 Future

Every project has its ideas for the future, this is also the case for this project. After contemplating on what we should have implemented during the development of our project we came up with a few further ideas. These improvements could possibly improve the compression efficiency of our codec.

Conditional separation of LPC groups

Separation of LPC groups is one possible improvement that we think would benefit the encoding process. It would involve computing the residuals using both common and separate LPC coefficients. This would allow for picking between encoding the separate residual and its LPC coefficients or the common residual. The latter method would work like our current implementation and leave the common LPC coefficients in the first `LPC_COMMON` subframe.

This method would need further research, but could improve the encoding efficiency in exchange for encoding time.

Additional DCT layer

A better replacement for the decorrelation and Rice coding stages could have been a reversible discrete cosine transform (RDCT) [4]. This transform is similar to the original DCT with modifications that allow its lossless reversal. This would make our codec work similarly to Opus, as described in section 2.1, while still being lossless. This could possibly improve the residual coding stage and achieve better results than the current entropy coding process.

Parameter Optimization

Our encoder exposes a lot of parameters which can be fine-tuned. These parameters include Rice responsiveness, frame size limits, framing resolution, framing threshold and many more. Optimizing and fine-tuning these parameters may, in our experience increase the overall compression ratio significantly. We came to this conclusion since our development recording was compressed around 3 % better than other recordings, as shown in chapter 6.

An alternative approach would be to estimate these parameters based on the input recordings.

Bibliography

[1] Bryant, D. *WavPack Audio Compression* [online]. 1998 [cit. 2022-05-05]. Available at: https://www.wavpack.com.

[2] Černocký, J. *Zpracování řečových signálů — studijní opora* [online]. Brno University of Technology, Faculty of information technology, december 2006 [cit. 2022-02-05]. Available at: http://www.fit.vutbr.cz/study/courses/ZRE/public/opora/zre_opora.pdf.

[3] Harris, F. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE* [online]. 1st ed. New York: Institute of Electrical and Electronics Engineers. january 1978, vol. 66, no. 1, p. 51–83, [cit. 2022-05-05]. DOI: 10.1109/PROC.1978.10837. ISSN 1558-2256. Available at: https://ieeexplore.ieee.org/document/1455106.

[4] Komatsu, K. and Sezaki, K. Reversible discrete cosine transform. In: IEEE, ed. *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)* [online]. 1st ed. New York: Institute of Electrical and Electronics Engineers, May 1998, vol. 3, p. 1769–1772 vol.3 [cit. 2022-05-05]. DOI: 10.1109/ICASSP.1998.681802. ISSN 1520-6149. Available at: https://ieeexplore.ieee.org/document/681802.

[5] Moffitt, J. Ogg Vorbis—Open, Free Audio—Set Your Media Free. *Linux Journal* [online]. 1st ed. Houston, TX: Belltown Media. january 2001, vol. 2001, 81es, p. 9–es, [cit. 2022-05-05]. DOI: 10.5555/364682.364691. ISSN 1075-3583. Available at: https://dl.acm.org/doi/10.5555/364682.364691.

[6] O'Shaughnessy, D. Linear predictive coding. *IEEE Potentials* [online]. 1st ed. New York: Institute of Electrical and Electronics Engineers. february 1988, vol. 7, no. 1, p. 29–32, [cit. 2022-05-05]. DOI: 10.1109/45.1890. ISSN 1558-1772. Available at: https://ieeexplore.ieee.org/document/1890.

[7] Rice, R. and Plaunt, J. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Transactions on Communication Technology* [online]. 1st ed. New York: Institute of Electrical and Electronics Engineers. december 1971, vol. 19, no. 6, p. 889–897, [cit. 2022-05-05]. DOI: 10.1109/TCOM.1971.1090789. ISSN 2162-2175. Available at: https://ieeexplore.ieee.org/document/1090789.

[8] Robinson, T. *SHORTEN: Simple lossless and near-lossless waveform compression* [online]. Cambridge: Department of Engineering, University of Cambridge, december

1994 [cit. 2022-05-05]. Available at:
http://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/robinson_tr156.pdf.

[9] Siegert, I., Lotz, A. F., Duong, L. L. and Wendemuth, A. Measuring the Impact of Audio Compression on the Spectral Quality of Speech Data. In: Jokisch, O., ed. *Studientexte zur Sprachkommunikation: Elektronische Sprachsignalverarbeitung* [online]. Dresden: TUDpress, 2016, p. 229–236 [cit. 2022-05-05]. ISBN 978-3-959080-40-8. Available at: https://www.essv.de/paper.php?id=344.

[10] Valin, J.-M., Vos, K. and Terriberry, T. Definition of the Opus Audio Codec. *RFC Index* [online]. 1st ed. RFC Editor. september 2012, no. 6716, [cit. 2022-05-05]. Request for Comments. DOI: 10.17487/RFC6716. Available at: https://www.rfc-editor.org/info/rfc6716.

[11] Xiph.Org Foundation. *FLAC - format* [online]. 2014 [cit. 2022-01-12]. Available at: https://xiph.org/flac/format.html.

[12] Žmolíková, K. *Far-field speech recognition.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of information technology. Supervisor Černocký, J.

Appendix A

The Straw format description

The following pages show the description of the Straw format. This description can also be found in the format.md [1] file.

[1] https://github.com/KLZ-0/straw/blob/master/doc/format.md

The .straw file format

Heavily based on the FLAC format

`~` before a value means the feature is not yet implemented in the encoder/decoder

Detailed format description

STREAM

- `<32>` "sTrW", the Straw stream marker in ASCII, meaning byte 0 of the stream is 0×73, followed by 0×54 0×72 0×57
- METADATA_BLOCK This is the mandatory STREAMINFO metadata block that has the basic properties of the
  stream
- FRAME+ One or more audio frames

METADATA_BLOCK

- METADATA_BLOCK_HEADER A block header that specifies the type and size of the metadata
  block data.
- METADATA_BLOCK_DATA

METADATA_BLOCK_HEADER

- `<1>` Last-metadata-block flag: '1' if this block is the last metadata block before the audio blocks, '0' otherwise.

- `<7>` BLOCK_TYPE

```
0 : STREAMINFO
~1 : PADDING
~2 : APPLICATION
~3 : SEEKTABLE
~4 : VORBIS_COMMENT
~5 : CUESHEET
~6 : PICTURE
~7-126 : reserved
~127 : invalid, to avoid confusion with a frame sync code
```

- `<0/24>` if(BLOCK_TYPE != STREAMINFO) Length (in bytes) of metadata to follow (does not include the size of the METADATA_BLOCK_HEADER)

METADATA_BLOCK_DATA

One of:

- METADATA_BLOCK_STREAMINFO
- ~METADATA_BLOCK_PADDING
- ~METADATA_BLOCK_APPLICATION
- ~METADATA_BLOCK_SEEKTABLE
- ~METADATA_BLOCK_VORBIS_COMMENT
- ~METADATA_BLOCK_CUESHEET
- ~METADATA_BLOCK_PICTURE

METADATA_BLOCK_STREAMINFO

- `<20>` Sample rate in Hz. Also, a value of 0 is invalid.
- `<8-?>` "UTF-8" coded (number of channels)-1.
- `<5>` (bits per sample)-1. 4 to 32 bits per sample.
- `<27>` Total number of frames
- `<36>` Total samples in stream. 'Samples' means inter-channel sample, i.e. one second of 44.1Khz audio will have 44100 samples regardless of the number of channels. A value of zero here means the number of total samples is unknown.
- `<128>` MD5 signature of the unencoded audio data. This allows the decoder to determine if an error exists in the audio data even when the error does not result in an invalid bitstream.
- `<8>` Rice coding responsiveness
- `<1>` Has shift correction
- `<8-?>` if (Has shift correction) "UTF-8" coded leading channel
- `<n*4>` if (Has shift correction) Shift needed for each channel compared to the leading channel, n = number of channels
- `<c*n*b>` if (Has shift correction) Removed samples start + end flattened, c = number of channels, n = number of removed samples (max lag), b = bits per sample
    - NOTE: the values are signed two's-complement

- `<1>` Has bias correction
- `<n*8>` if (Has bias correction) DC bias removed from each channel, n = number of channels
    - NOTE: the values are signed two's-complement
- `<1>` Has gain correction
- `<n*12>` if (Has gain correction == 1) Gain correction coefficients (factor) - 1.0, n = number of channels
    - These are unsigned quantized floating point numbers with the range (1 to inf) by for storage purposes 1.0 is subtracted since the coefficients are always larger than 1
    - The strongest channel will always have a factor of 1.0 (or 0 quantized)
- `<4>` if (Has gain correction == 1) Gain shift in bits
- `<?>` Zero-padding to byte alignment.

NOTES

The "UTF-8" coding is the same variable length code used to store compressed UCS-2, extended to handle larger input.

FRAME

- FRAME_HEADER
- SUBFRAME+ One SUBFRAME per channel.
- `<?>` Zero-padding to byte alignment.
- FRAME_FOOTER

FRAME_HEADER

- `<14>` Sync code '10101010101010'
- `<1>` Reserved

```
0 : mandatory value
1 : reserved
```

- `<1>` Block size length:

```
0 : get 8 bit exponent for (2^n) samples
1 : get 16 bit (blocksize-1)
```

- `<0/8>` elif(Block size length bit == 0) blocksize = (2^n) samples
- `<0/16>` if(Block size length bit == 1) 16 bit (blocksize-1)
- `<8-?>`: "UTF-8" coded frame number
- `<32>` Size of the frame in bytes (size including the header sync code and the frame footer)
- `<8>` CRC-8 (polynomial = x^8 + x^2 + x^1 + x^0, initialized with 0) of everything before the crc, including the sync code

FRAME_FOOTER

- `<16>` CRC-16 (polynomial = x^16 + x^15 + x^2 + x^0, initialized with 0) of everything before the crc, back to and including the frame header sync code

SUBFRAME

NOTE: Subframes are not byte-aligned

- SUBFRAME_HEADER
- SUBFRAME_DATA

SUBFRAME_HEADER

- `<2>` Subframe type:

```
00 : SUBFRAME_CONSTANT
01 : SUBFRAME_RAW
10 : reserved
11 : SUBFRAME_LPC
```

SUBFRAME_DATA

One of:

- SUBFRAME_CONSTANT
- SUBFRAME_RAW
- SUBFRAME_LPC
- SUBFRAME_LPC_COMMON

The SUBFRAME_HEADER specifies which one.

SUBFRAME_CONSTANT

- `<n>` Unencoded constant value of the subframe, n = bits-per-sample.

SUBFRAME_RAW

- `<n*i>` Unencoded samples of the subframe, n = bits-per-sample, i = frame's blocksize.

SUBFRAME_LPC

- `<5>` (LPC order) - 1
- `<4>` (Quantized linear predictor coefficients' precision in bits)-1.
- `<4>` Quantized linear predictor coefficient shift needed in bits
- `<bpc*order>` Unencoded predictor coefficients (qlp coeff precision * lpc order) (NOTE: the coefficients are signed two's-complement).
- SUBFRAME_LPC_COMMON

SUBFRAME_LPC_COMMON

- `<bps*order>` Unencoded warm-up samples (bits-per-sample * lpc order).
- RESIDUAL Encoded residual

RESIDUAL

- `<4>` Starting rice parameter
- `<->` Encoded residual n = frame's blocksize - predictor order