Uppsala University

Master thesis

Compression of data from microphone array

Author:
Axel Östfeldt

May 20, 2024

Abstract

Saab's *Acoustic Warfare* system utilizes microphone arrays to localize sound sources, with data processed on an FPGA for subsequent beamforming. This study explores the feasibility of implementing lossless compression algorithms on the FPGA to reduce the average transfer rate per array, aiming for levels below 90 Mbps, preferably under 45 Mbps. Various algorithms were researched and evaluated, with the Adjacent algorithm emerging as the most suitable for FPGA implementation due to its ability to utilize microphone correlation for compression. Testing in a Python environment confirmed its efficacy in achieving compression rates and maintaining lossless data integrity. Implementation via a VHDL testbench demonstrated successful compression and decompression without loss of information. The resulting transfer rate reductions ranged from 44 to 58 Mbps per array, contingent on the sound type. This study showcases the potential for enhancing system efficiency through FPGA-based compression algorithms.

Populärvetenskaplig sammanfattning

Saabs *Acoustic Warfare* system utnyttjar matrisuppsättningar med flera mikrofoner för att lokalisera ljudkällor. Inspelning av ljudet görs med hjälp av en FPGA på ett ZyboZ7 20-utvecklingskort. Data överförs sedan via Ethernet till en dator där riktningen på ljudkällan lokaliseras med hjälp av beamforming. I dagsläget överförs data med en hastighet på strax över 100 Mbps per mikrofon array. Detta projekt undersöker möjligheten att minska överföringshastigheten till under 90 Mbps per array med komprimeringsalgoritmer, idealiskt under 45 Mbps per array. Ytterligare ett krav är att algoritmerna ska vara förlustfria och fungera i realtid.

Flera lämpliga förlustfria komprimeringsalgoritmer för ljuddata utvärderades i Python-miljö. Kompressionsförmåga, dekompressionshastigheter, samt förmåga till förlustfri komprimering undersöktes. Efter analys valdes Adjacent-algoritmen ut som den mest lämplig för FPGA implementering. Den utnyttjar mikrofonernas korrelation genom att förutsäga nästa mikrofonsvärde från en närligande mikrofon och kodar skillnaden Rice codes.

Implementering av Adjacent algoritmen i VHDL miljö utvärderades med hjälp av en testbänk. Resultatet visade en minskad överföringshastigheten till cirka 44-58 Mbps per mikrofonmatris, beroende på ljudtyp, allt utan att någon information förlorats. Detta resultat visar potentialen för att effektivisera Saabs system genom komprimeringsalgoritmer.

Acknowledgement

I want to express my gratitude to Saab AB for granting me the opportunity to work on this project, and to the individuals within the organization for warmly welcoming me. A special thanks to my supervisor, Rasmus Söderström, who has guided me through this project and the workplace. I want to specifically thank Didrik Olofsson, Jacob Drotz, Joel Lindfors, Mats Järgenstedt, and Tom Setterblad for their assistance at Saab, helping me with the project and making me feel welcomed to the company.

I would also like to thank my project subject reader at Uppsala University, Uwe Zimmermann, for advising me on my report and helping me leave a contribution to the academic world.

Lastly, thanks to my fellow student Jonas Teglund, who has kept me good company during the project and who has been an excellent springboard to bounce ideas off of.

Table of contents

1 Abbreviations and acronyms

Abbrevation	Meaning
AXI	Advanced eXtensible Interface protocol
FLAC	Free Lossless Audio Codec
FPGA	Field Programmable Gate Arrays
LPC	Linear Predictive Coding
LSB	Least Significant Bit
MBps	Megabytes per second
Mbps	Megabits per second
MSB	Most Significant Bit
MSE	Mean-squared error
PCB	Printed Circuit Board
RLE	Run-Length Encoding
UDP	User Datagram Protocol

2 Introduction

In 1953 Dr Warren Weaver summarised the problem of communication in three levels:

- Level A. How accurate is the transmission of symbols? (The technical problem)
- Level B. How well do the symbols transmitted convey the desired meaning (The semantic problem)
- Level C. How effectively does the received meaning affect conduct in the desired way? (The effectiveness problem)

This paper will look at a communication problem from a Level A perspective. "Since Weaver wrote his paper over 70 years ago, the amount of data communicated worldwide has increased manifold.The question Dr. Weaver posed to formulate a Level A problem can be considered more relevant than ever[1].

(a) How does one measure amount of information?

(b) How does one measure the capacity of a communication channel?

(c) What are the characteristics of an efficient coding process?

(d) With efficient coding, at what rate can the communication channel convey information?

(e) How does the signal being transmitted being continuous, such as audio signal, affect the problem?

Although the questions posed in this paper use different semantics (a Level B problem, as Weaver would say), they are all relevant throughout this work. This paper focuses on the specific problem of compressing audio data with the aim of decreasing bit rate over a communication channel. To be able to do this relevant sources regarding information theory and compression algorithms will be examined. When summarizing the theory of compression algorithms, it is divided between the model and the coder. The model captures the probability of the symbols being sent, while the coder takes advantage of these probabilities to generate codes[2]. The information gathered will then be applied with Python and VHDL code in order to solve the problem of compressing audio data in a real-time environment.

2.1 Background

Saab AB has developed a system over the last couple of summers to localize sound sources under the alias *Acoustic Warfare*. The system utilizes beamforming to visualize the audio sources as a heat map over a webcam feed in real-time.

The system utilizes four PCBs containing 8x8 microphone arrays to record audio. Each PCB consists of 64 ICS-52000 microphones with an ADC, anti-aliasing filter, wide-band response, and omnidirectional directionality. The microphones on each PCB are daisy-chained together in groups of 16, creating four groups on each card. An FPGA on a Zybo-Z7 20 development board samples the audio using the microphone arrays. Below is a figure depicting the system:

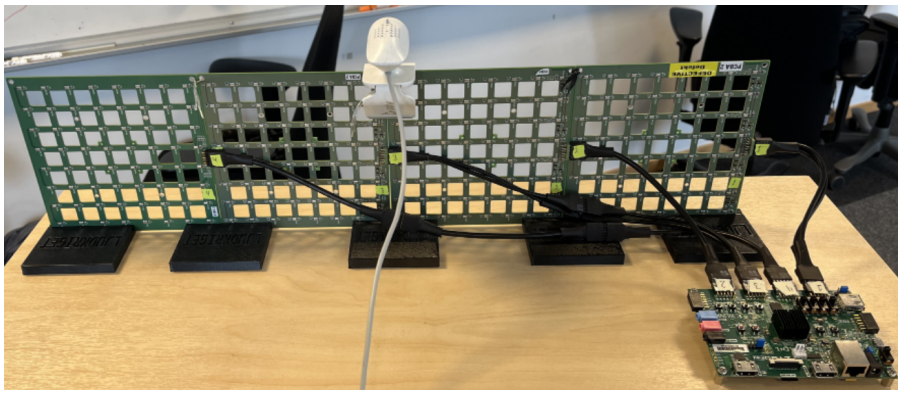


Figure 1: Setup of the Acoustic Warfare System

In the bottom left of Figure 1, the Zybo-Z7 20 development board can be seen. Four microphone arrays are connected to it, and a camera is attached in the middle of the arrays.

The sampling starts with an incoming WS-pulse and is then sampled at a frequency of 48,828.125 Hz. The microphones output 24 bits of data, which are padded with two's complement and concatenated to create 32 bits of data. The data is sent to a 32-bit Time-Division Multiplexing slot.

An AXI protocol is used to load the data packages into the package sender. The package sender on the FPGA board then utilizes the User Datagram Protocol (UDP) to transmit the packages to a package collector over an Ethernet connection. The data is transmitted in packages containing several 32-bit slots. The first two slots in each package contain a header with metadata: 8 bits for protocol version, followed by 8 bits for the number of arrays, then 16 bits for frequency information, and lastly 32 bits for the sample counter. The subsequent 256 slots contain 32-bit data from each microphone. Each package's total size is 8256 bits, or 1032 bytes. These packages are sent with a frequency of 48.828125 kHz, resulting in a transfer rate of circa 50.39 MBps, or 403.12 Mbps, when all four microphone arrays are used. This results in an average transfer rate of 12.6 MBps or 100.78 Mbps per array.

Once received on the PC side, the samples are stored in a buffer. This buffer is sized at 256 by 256, meaning 256 samples from each of the 256 microphones can be accommodated. It is continuously updated by the information sent over the Ethernet

cable, and old samples are overwritten. To perform beamforming, the samples are extracted from the buffer, processed, and a heat map is generated using various beamforming algorithms.[3]

2.2 Purpose

The purpose of this paper is to gather in-depth knowledge on the subject of data compression by finding, studying, and compiling relevant literature. This in-depth knowledge will be applied in a scientific and engineering manner to solve the technical problem of data transfer rate in Saab's *Acoustic Warfare* system. This involves implementing a compression algorithm for the FPGA board to reduce the average transfer rate. Such an implementation could enable the use of cheaper FPGA boards, which are typically limited to a 100 Mbps transfer rate using Ethernet.

2.3 Limitations

Compression algorithms are plentiful, but there is no algorithm that can compress all types of data perfectly. The data being handled in this project is strictly audio data. Therefore, the algorithms are limited to only looking into audio compression algorithms.

Compression algorithms are also usually divided into lossy and lossless algorithms. Lossy algorithms allow for some loss of data without corrupting the signal, while lossless algorithms compress all data and decompress it exactly to the original data (in theory). In audio compression, lossy algorithms are quite common, since they are usually used in the context of humans listening to audio in some format. Losing data in this context can be acceptable if it does not affect how it sounds to human ears; this could involve losing frequencies outside of the ear's range to save data, for example. In this project, the compression will be applied to a system that uses audio data to perform beamforming and localize the sound source, and not for listening to audio. Lossy algorithms have therefore been excluded, and the project has been limited to looking at lossless audio compression algorithms.

Several different algorithms that match the limitations set will be examined in this paper, namely: Shorten, LPC, FLAC, Adjacent, and some modified versions of these algorithms.

2.4 Goal

The goal is to develop a compression algorithm for the FPGA board to be utilized in Saab's *Acoustic Warfare* system, aiming to reduce the average transfer rate below 90 Mbps per array, and preferably down to 45 Mbps per array to be able to send data from two arrays over one ethernet cable at a transfer rate of 90 Mbps.

3 Methodology

3.1 Research and Information Gathering

The research for this project focused on two points of interest: the current system and compression algorithms.

The current system was examined by reviewing the report from the project conducted in *Acoustic Warfare*[3]. The *Acoustic Warfare* system, depicted in Figure 1, was used to record sounds to obtain a baseline dataset for testing compression algorithms. A mobile phone speaker was employed to play sounds including a 1 kHz tone, drone sound, and static noise, while moving the sound source around the room.

When exploring compression algorithms, emphasis was placed on identifying those commonly used for FPGA and audio data. For FPGA, GZIP was found to be the most common, which is a modified version of LZ77 compression algorithm [2]. However, for audio compression, it was noted that GZIP was not well-suited for the task. Among the algorithms considered most interesting for compressing audio in a lossless manner, no research on their implementation on FPGA could be found. This presents an intriguing scientific gap for exploration in this paper.

3.2 Python Programming

During the research section of the project, several models and a couple of coding schemes were found suitable for the task at hand. More information about these can be read in the theory section.

To test their performance, a replay script in Python was utilized to emulate the *Acoustic Warfare* system using the recorded data, enabling real-time testing. The viability of the raw data was tested by plotting the sound waves and analyzing the performance of specific microphones. Compression rate by only implementing coding schemes and not modeling the signals was also tested for the raw data.

The test using compression algorithms, both modeling and coding in combination, performed in sections 5.3, 5.4, 5.5, 5.6, 5.7, and 5.8 aim to determine performance in the following aspects:

- Ability to recreate original values correctly.
- Evaluate k value for Rice codes.
- Time to recreate original values from codewords.
- Compression rate.

When evaluating the time and compression rate performance, the tests were performed for 20 datablocks from all 64 microphones in array 2, and the result pre-

sented is an average value for one datablock. The compression rate was calculated by comparing the length of the codeword to the length of the original data without zero padding, meaning each sample is represented in 24 bits. The reasoning behind this is to display the algorithms ability to compress data in general in the results, and not specifically to the *Acoustic Warfare* system.

Evaluation of k values when using Rice codes was done with Shorten, LPC, and Adjacent by comparing an averaged compression rate. The tests were run over 20 datablocks, and the result is presented as the average value per datablock. When evaluating Rice codes with Shorten and LPC, the data from microphone 79 was used for all sound files, as well as microphone 20 for a 1 kHz tone to test on a silent microphone. When testing for Adjacent, all microphones in array 2 were used.

Following the research on compression algorithms, the most suitable one for the task was selected for implementation using Python. The recorded audio data with saved timestamps was used to simulate how the different algorithms would perform in a real-time environment. This process helped identify which algorithms showed the most promise and should be implemented in the *Acoustic Warfare* system. While compression ratio was central to determining the best compression algorithm, it was not the sole determining factor. Decompression speed and how well-suited the algorithm would be for implementation on an FPGA were also important considerations.

3.3 Implementation on Testbench

From the results the Adjacent algorithm with Rice codes was decided to be implemented in a VHDL testbench. For an in dept motivation for this decision see section 6.2 and 6.3. The compression algorithm was written by creating entities for modelling and coding.

3.3.1 AdjacentResiduals entity

This entity receives a datablock containing data with one sample from 64 microphones. Each sample represents the data as 24-bit signed integers. The entity calculates the residual according to the Adjacent modeling, as described in Section 4.3.4. It functions as a state machine with states for receiving datablocks, grabbing the next sample, calculating the residual, sending the residual, etc.

3.3.2 kCalculator entity

This entity summarizes the absolute value of all residuals in a datablock. The total sum is then compared to different levels to determine what k value to use when encoding them. These levels have been calculated by modifying Equation 19 by replacing $\mathbb{E}[x]$ with $\frac{Total,sum}{64}$. Division by 64 is performed since there are 64 samples

per datablock. By calculating the set levels for k values at different total sums before implementation, operations can be avoided in the FPGA environment, speeding up the process. The entity works as a state machine, with different states for reading residuals, summarizing their absolute value, and once a full datablock has been summarized, sending the k value.

3.3.3 RiceEncode entity

The RiceEncode entity Rice encodes a full datablock of residuals by employing the steps described in Section 4.2.3. It operates using a state machine with states for receiving datablocks of residuals and k value, several states for the encoding steps, states for sending a codeword for each sample, etc.

3.3.4 CodeWordAssembler entity

The CodeWordAssembler entity assembles all codewords for each sample in a datablock into one vector. This vector also contains metadata, with 5 bits for the k value used to encode the residuals and 8 bits to describe how many samples have been encoded. The entity operates as a state machine, with states for receiving codewords, assembling codewords, setting metadata, and sending the assembled codewords, etc.

3.3.5 Reading and writing data

The compression algorithm has been tested in isolation in the testbench, meaning that the reading of data from the microphones and sending of data via internet packages used in the *Acoustic Warfare* system are not utilized in the testbench. Instead, this is emulated by reading and writing from/to .txt files in the testbench. The data is read from a .txt file containing 256 samples from 64 microphones, with each sample saved as a 24-bit signed value. The data in the .txt file is created from the 1 kHz tone recorded using the *Acoustic Warfare* system. Once the full codeword has been assembled with the metadata, it is written to a .txt file. The data in this .txt file is then read into a Python script that uses the Adjacent and Rice code classes to verify that the VHDL entities have encoded the data correctly.

4 Theory

4.1 Information Theory

Information theory aims to measure the amount of information contained in messages sent during communication. The simplest method of measuring information is by taking the base-2 logarithm of the number of available choices. For example, if a communication channel has 16 alternative messages, each equally likely to be sent, the information is calculated as $\log_2 16 = 4$ bits. The unit for measuring information is "bit"; thus, the previous example contains 4 bits of information[1].

With compression algorithms, the goal is to reduce the number of bits sent over the communication channel. However, it is not possible for a lossless compression algorithm to compress all messages, especially when the message sent can contain any bit sequence. For instance, if a communication channel sends messages of 100 bits, there are 2^{100} different possible messages. If each of these messages were compressed by just one bit, resulting in all messages containing 99 bits, the communication channel would only be capable of sending 2^{99} different messages. This illustrates that in order to compress certain messages in the communication channel, other messages must be lengthened. In fact, for a set of input messages with fixed lengths, if one message is compressed, the average length of all possible inputs will always become longer than the original inputs. As it is not feasible to compress all messages, compression algorithms leverage message probabilities. They exploit the bias towards different messages by assigning shorter codes to messages with higher bias and longer codes to those with lower bias[2]. An example of this can be seen in telegraphy, where common letters, such as E, have shorter codes, while less common letters, such as Q, have longer codes[4]

4.1.1 Entropy

The concept of entropy is a measure of information. If there are n different possible messages that can be sent on the communication channel, each with the possibility p_i, the entropy is calculated using the equation:

$$H = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{1}$$

Where H is the entropy in bits per message. $H = 0$ can only be achieved if there is only one possible choice, where all p_i are zero except one. In all other cases, H is positive. If all the probabilities of p_i are equal, $H = \log_2 n$. This is the maximum value H can achieve for a given set of n possible messages. This illustrates how entropy relates to uncertainty. When there is only one possible choice, uncertainty vanishes, and entropy is zero. When all choices are equally possible, uncertainty is

at a maximum, and so is the entropy. As the probabilities converge, the entropy increases[4]. This aligns with lower entropy indicating more biased probabilities in the messages.

The concept of self-information represents how many bits of information a message contains and is a good indicator of how many bits the message should be encoded in. The self-information of a message can be calculated using the equation:

$$i(s) = \log_2 \frac{1}{p(s)} \tag{2}$$

Where *i(s)* is the self-information of the message and *p(s)* is the probability of the message. A message's information is inversely related to its probability, meaning messages with less information have higher probability and vice versa. Entropy is the probability-weighted average of the self-information for each message, with larger entropies representing a larger average of information per message. Since higher entropy indicates less biased probabilities, a higher average of information indicates a more random set of messages[2].

A noiseless communication channel with the capacity C bits per second used to transfer messages with entropy H bits information per message can never have an average transfer of information rate higher than $\frac{C}{H}$ bits per second[4]. The greater the uncertainty of the source, the more information is being transmitted. It follows that higher uncertainty of the source will need more bits to represent all messages, which will affect the information rate for a given communication channel[5].

4.2 Coding

4.2.1 Run-Length Encoding (RLE)

Run-Length Encoding (RLE) utilizes the fact that the same data item is repeated several times consecutively in the data. When a run of symbol d occurs n times in a row, it can be written as:

@nd

Where @ is an *escape character*, indicating to the decoder that an RLE sequence is coming, n represents how many times the data item is repeated, and d is the data item to repeat. A given run of data items can then be replaced with these three items. The compression factor from this can be calculated with the following equation:

$$\frac{N}{N - M(L-3)} \tag{3}$$

In equation 3, M is the number of repetitions in the data, L is the average length of repetition, and N is a data string[6].

4.2.2 Golomb Codes

Golomb codes are variable-length codes that assume larger integers have a lower probability of occurring[7]. Golomb code is ideal to use when the list of possible outcomes is infinite but follows a distribution law. In order to use Golomb code to encode and decode a non-negative integer n, a choice for the parameter m has to be made (m should preferably be set to an integer). The ideal choice of m can be derived from the following equation:

$$m = -\frac{\log 2}{\log p} \tag{4}$$

Where p is the probability of a "0" occurring in a bit stream. In cases where the ideal m is not an integer, the ideal dictionary will switch to and from different m values. For large m, the penalty for picking the nearest integer value is very small[8].

With n and m, it is possible to calculate the three quantities needed for Golomb coding using the following equations:[6]

$$q = \left\lfloor \frac{n}{m} \right\rfloor \tag{5}$$

$$r = n\%m \tag{6}$$

$$c = \lceil \log_2 m \rceil \tag{7}$$

Where q is the quotient and r is the remainder. The floor sign in equation 5 indicates that q is rounded down to the closest integer, and the ceiling sign in equation 7 indicates that c is rounded up to the closest integer. The quotient is coded as unary prefix, meaning that there are q 1's followed by a 0 (alternatively, q 0's followed by a 1). The remainder is then binary coded, but the length of the code depends on the value of r in the following way:

$$\text{If } r < 2^c - m \text{ then } b = c - 1$$

$$\text{If } r \geq 2^c - m \text{ then } b = c$$

Where b is the number of bits r is coded in. The first $2^c - m$ values of r, when $b =$ c - 1, are encoded in bits as normal. The rest, when $b = c$, are encoded so that the biggest possible residual consists of only 1's (the rest are written in bits decremented from this value, that is if $b = c = 3$ the largest residual is written as "111" and the second largest as "110" and so on until $r < 2^c - m$). An exception to this is when m is a power of 2; then r is always written in bits as normal with the length c. The remainder in bits is then appended to the unary-coded quotient to get the final code. In the table below are some examples of Golomb codes:

m/n	n = 0	n = 1	n = 2	n = 3	n = 4
2	0|0	0|1	10|0	10|1	110|0
3	0|0	0|10	0|11	10|0	10|10
4	0|00	0|01	0|10	0|11	10|00
5	0|00	0|01	0|10	0|110	0|111

Table 1: Some Golomb codes, on the left side of | is the quotient and on the right the remainder

It can be seen in Table 1 for $m = 5$ that even though $r = 3$ and $r = 4$ for $n = 3$ and $n = 4$, respectively, the binary value does not match the remainder value. This is because of how the binary value is set for r when $r < 2^c - m$, as stated earlier. If the data being handled by the Golomb code is signed, there are two extra steps to encoding. The sign bit for n is removed and saved as s. The encoding is done the same way as stated earlier; once this is done, the sign bit is added back onto the encoded message as the MSB.

In order for the decoder to correctly decode the Golomb code, it needs to know two things: the value of m and if the code is signed or unsigned. With m known, the decoder can calculate the value of c with Equation 7. Assuming the unsigned case for Golomb codes first, the following equations are used for decoding:

$$n = m * A + R \tag{8}$$

$$n = m * A + R' - (2^c - m) \tag{9}$$

The A value is always derived the same way, by counting how many 1's occur before the first 0 (or vice versa, depending on how the unary code is set up). If m is a power of 2, the c last bits are decoded to their binary value and denoted R. To get n, Equation 8 is used.

In cases where m is not a power of 2, the c-1 bits following the first 0 are denoted as R. R is then compared in the same way as r:

$$\text{If } R < 2^c - m \text{ then } B = A + c$$

$$\text{If } R \geq 2^c - m \text{ then } B = A + c + 1$$

Where B is the total length of the code word. If $B = A + c$, then R captures all the remaining bits in the code word. R can then be converted to an integer, and Equation number 8 can be used to calculate n. If $B = A + c + 1$, the c bits following the first 0 are denoted as R'. R' is converted to an integer, and Equation number 9 is used to calculate n[6].

For the signed decoding case in Golomb code, some extra steps are needed. The MSB of the Golomb code will be the signed bit; this is first removed and saved as S. The rest of the bits are decoded as previously stated. Equation numbers 8 and 9 will always give a positive n value. If the sign bit, S, is "1", an extra step is added where n = - n[6].

4.2.3 Rice codes

Golomb codes are simplified in cases where m is a power of 2 [8]; these cases are called Rice codes (or Golomb-Rice codes). Instead of choosing the parameter m, the parameter k is chosen in Rice codes. The relation between m and k is as follows:

$$m = 2^k$$

This simplifies the encoding. To code the binary value n in Rice code, the following steps are taken:

1. The sign bit is separated, denoted s, and will later be used as the MSB for the encoded n. This step is only done if the data contains signed values.
2. The k number of LSBs are separated, denoted r, and will later be the LSB for the encoded n.
3. The remaining bits are converted to an integer and unary coded, denoted q.

The final codeword will be as follows:

$$"s" + "q" + "r"$$

For decoding, the following steps are taken:

1. Save the MSB as the sign bit, if the data is signed. This is denoted S.
2. Count the following 0's after the sign bit until the first 1 occurs. Convert the number of 0's to binary, denoted Q.
3. Denote the last k bits that remain as R.

The binary value n will be obtained from:

$$"S" + "Q" + "R"$$

Rice coding allows for n to be encoded and decoded only using logical operations, which speeds up the computing for encoding and decoding. Especially the decoding is very computationally fast since it only requires one loop through the codeword to get the original n.

Rice codes are ideal when the data to be encoded follows a Laplace distribution. The value of k decides the amount of LSB to be included in the Rice code and is linearly related to the variance of the data to be encoded. This leads to a formula to compute the ideal k value for a dataset x:

$$k = \log_2(\log(2) \cdot \mathbb{E}[x]) \tag{10}$$

Where $\mathbb{E}[x]$ is the expected value over the data $x[6]$.

4.3 Modeling

4.3.1 Shorten

Shorten is a lossless compression model created by Tony Robinson specifically to compress audio by exploiting the significant sample-to-sample correlation often present in audio data. Shorten utilizes this correlation by predicting the waveform using a linear combination of previous samples. This can be represented by the following equation:

$$\hat{s}(t) = \sum_{i=1}^{p} a_i \cdot s(t-i) \tag{11}$$

Where:

- $\hat{s}(t)$ is the predicted value of the waveform at time t.
- a_i is a set of coefficients.
- $s(t)$ is the waveform value at time t.
- p determines how far back in the waveform samples are used to predict the current sample. (The typical value for p is 3; it never exceeds this in Shorten.)

With the predicted value $\hat{s}(t)$, a residual can be calculated using the equation:

$$e(t) = s(t) - \hat{s}(t) \tag{12}$$

Where $e(t)$ represents the residual, which is the difference between the predicted value and the actual value of the waveform.

The ideal values of a_i and p may vary throughout the waveform. Robinson suggests dividing the data into blocks and finding the optimal values for a_i and p in each block. The block size recommended by Robinson is 256 values [9]. If the file has more than one audio channel, Shorten divides each channel into separate blocks [6].

For calculating the prediction, Shorten uses one of the following equations with different levels of p [9]:

$$\hat{s}(t) = 0 \tag{13}$$

$$\hat{s}(t) = s(t-1) \tag{14}$$

$$\hat{s}(t) = 2s(t-1) - s(t-2) \tag{15}$$

$$\hat{s}(t) = 3s(t-1) - 3s(t-2) + s(t-3) \tag{16}$$

The choice between equations 13, 14, 15, or 16 depends on which gives the best residual versus the computational time for the current block. The residuals are variable-length coded. Correlation between audio samples implies that the residuals are small, and experiments show that they follow the Laplace distribution. This suits the residuals well to be encoded with Rice codes [6].

4.3.2 Linear Predictive Coding (LPC)

Linear Predictive Coding utilizes correlation between audio samples to create a linear predictor of future waveform values from preceding values, similarly to Shorten, with equation 11. The difference between LPC and Shorten is how it chooses its coefficients, a_i [6]. LPC does not have set coefficients for different orders but tries to find the coefficients that minimize the mean-squared error [10]. This can be done in three steps:

1. Finding the squared error:

$$d(t) = \left(s(t) - \sum_{i=1}^{n} a_i \cdot s(t-i) \right)^2$$

2. Differentiating the squared error with respect to coefficient a_p and setting the derivative to zero for $p = 1, 2, \ldots, n$:

$$\frac{\partial d(t)}{\partial a_p} = 0 \quad \text{for } p = 1, 2, \ldots, n$$

$$\Rightarrow -2 \cdot \left[\left(s(t) - \sum_{i=1}^{n} a_i \cdot s(t-i) \right) \cdot s(i-p) \right] = 0 \quad \text{for } p = 1, 2, \ldots, n$$

3. Taking the expected value:

$$\Rightarrow \sum_{i=1}^{n} a_i \cdot \mathbb{E}[s(t-i) \cdot s(t-p)] = \mathbb{E}[s(t) \cdot s(t-p)] \quad \text{for } p = 1, 2, \ldots, n$$

To simplify the expression, the autocorrelation of the waveform can be used, which is the correlation between two samples. The equation for the autocorrelation is as follows:

$$R_s(i) = \mathbb{E}[s(t) \cdot s(t+i)] \tag{17}$$

By applying equation 17 to the earlier expression, it can be represented as a matrix multiplication:

$$\begin{bmatrix} R(0) & R(1) & R(2) & ... & R(n-1) \\ R(1) & R(0) & R(1) & ... & R(n-2) \\ R(2) & R(1) & R(0) & ... & R(n-3) \\ . & . & . & ... & . \\ . & . & . & ... & . \\ . & . & . & ... & . \\ R(n-1) & R(n-2) & R(n-3) & ... & R(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ . \\ . \\ . \\ a_n \end{bmatrix} = \begin{bmatrix} R(1) \\ R(2) \\ R(3) \\ . \\ . \\ . \\ R(4) \end{bmatrix}$$

This can be written as $\mathbf{R}\mathrm{C} = \mathbf{P}$. Normally, solving the equation would require $\mathrm{O}(n^3)$ computations, but the matching diagonals of $\mathbf{R}$ and its symmetry allow for faster computations using the Levinson-Durbin algorithm[6].

The Levinson-Durbin algorithm is a recursive method that finds the ith-order optimal predictor by considering the optimum predictor of order $i-1$. The algorithm determines all optimal coefficients for orders 0 to n along with the corresponding mean-squared prediction error E. The algorithm is computed with the following steps:

$\mathrm{E}^0 = \mathrm{R}(0)$

for i = 1, 2,..., n

$k_i = \frac{R(i) - \sum_{j=1}^{i-1} a_j^{i-1} \cdot R(i-j)}{E^{i-1}}$

$a_i^i = k_i$

if i > 1 then for j = 1, 2,..., i-1

$a_j^i = a_j^{i-1} - k_i \cdot a_{i-j}^{i-1}$

end

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

end

E^i indicates the mean-squared prediction error for the coefficients of order i, and a_j^i indicates the jth coefficient for order i. For example, if a third-order predictor has the following coefficients:

$$\hat{s}(t) = 5 \cdot s(t-1) + 6 \cdot s(t-2) + 7 \cdot s(t-3)$$

The mean squared prediction error from this particular $\hat{s}(t)$ would be E^3, and the coefficients would be: $a_1^3 = 5$, $a_2^3 = 6$, and $a_3^3 = 7$ [10].

4.3.3 FLAC

FLAC stands for Free Lossless Audio Codec and is also a compression algorithm specialized for audio data. It shares a lot of commonalities with the previously mentioned Shorten and LPC; it also exploits the fact that audio data have high correlation between each sample. The steps FLAC takes to compress the code are yet another similarity it has to other audio compression algorithms; it follows the 4 steps:

1. **Blocking**: The data is divided into continuous blocks. These blocks may vary in size and span across several channels.
2. **Interchannel Decorrelation**: If the audio is stereo, meaning it uses two channels, the algorithm will create a mid and side signal from the average and difference of the left and right channels. Based on which of these gives the best result—left and right versus mid and side—it chooses which one to encode. This can vary from block to block.
3. **Prediction**: FLAC uses one of four predictors on each datablock and calculates the residual. Which predictor is chosen is based on which one gives the smallest residual.
4. **Residual coding:** The residual is passed through an encoder for the final compression step

Choosing a good block size is essential for the compression ratio. With every block, a frame header is sent containing metadata to the decoder in order for it to decode the data correctly. If the block size is too small, a lot of space is wasted on the frame header. On the other hand, if the block size is too large, the data can have a large variation in its characteristics and therefore make it hard for the encoder to find a good predictor. FLAC imposes a block size of 16 to 65535 samples in order to be able to compress the data optimally.

FLAC calculates the residual the same way as Shorten, with equation 12. There are four different methods FLAC can use to model its predictor, $\hat{s}(t)$:

1. **Verbatim**
2. **Constant**
3. **Fixed linear predictor**
4. **FIR Linear prediction**

Verbatim is essentially equation 13. Since this equation always predicts a zero, the residual will be the raw signal sent verbatim. This is the baseline against which all the other predictors are measured when FLAC decides which method to use.

Constant, as suggested by its name, is used when the data signal is a constant value. For this case, no predictor is used, and the code is instead encoded using RLE [11].

Fixed linear predictor uses the same predictors as those used in Shorten equations 14, 15, and 16. FLAC also implements a fourth-order predictor with the following equation [6]:

$$\hat{s}(t) = 4s(t-1) - 6s(t-2) + 4s(t-3) - s(t-4) \tag{18}$$

FIR linear prediction implements the same steps as LPC to calculate coefficients for the most accurate predictor, albeit at the cost of slower encoding. FLAC supports FIR linear predictors up to the 32nd order. Just like LPC, FLAC utilizes the Levinson-Durbin algorithm to calculate coefficients for the FIR predictor from the autocorrelation in the data.

FLAC can vary which method to use in each subblock, meaning different channels in the same block can use different methods to calculate the residual. The flexibility of FLAC improves its capability to compress data but also means that more calculations must be done to find the best option, and more metadata must be sent to the decoder side for it to know which method is used.

As mentioned earlier, for the prediction method *constant*, RLE is used to encode the data. In other cases, Rice codes are used. The value for k can be chosen in two different ways in FLAC. The first method is to base it on the variance of the entire residual and use it to encode the entire residual. The second method is to divide the residual into equal-length regions of contiguous samples and calculate each region's mean. The mean for each region will be used as the k value for that specific region[11].

4.3.4 Adjacent

The modelling of the input signal when compressing audio data, previously mentioned, all builds on predicting future samples from previous ones, exploiting the sample-to-sample correlation in audio data [6]. The system created in the *Acoustic Warfare* project does not use only one microphone but several arrays of microphones spread over a grid [3]. Assuming that the audio data recorded by one microphone would be similar to adjacent microphones, it could be possible to create a model that exploits mic-to-mic correlation rather than sample-to-sample. If this holds true, that could open up an avenue to predict the input signal for a microphone by looking at the previous one. This algorithm would calculate the residual by taking the difference between the current microphone and the previous microphone on the encoding side. Both Golomb and Rice codes could be used to encode this residual since they work better when larger residuals have a lower probability of occurring [7]. On the

decoding side, the original value could then be recreated by adding the residual to the previous microphone value. The first microphone in the array would need to be sent as either raw data or use another encoder to have a baseline microphone to calculate the first adjacent value.

It could also be possible to combine this modeling along with previously mentioned algorithms. By first calculating the residual as the difference between adjacent microphones, and then attempting to utilize correlation between the residuals. By taking the residuals from adjacent microphones and then applying, for example, the Shorten algorithm to predict the next residual, the residual might be able to become even smaller. If the probability of the residuals being smaller integers, Golomb and Rice codes become even more effective when compressing them [7].

5 Results

5.1 System

The system developed in the *Acoustic Warfare* project was used to record sound files of different scenarios to be later used for testing the compression algorithms. A mobile phone speaker served as a sound source, playing various sounds while moving around the room. The three sound recordings used for testing were: a 1 kHz tone, drone sounds, and static noise. The data recorded from the different sound waves can be seen in the plot below:

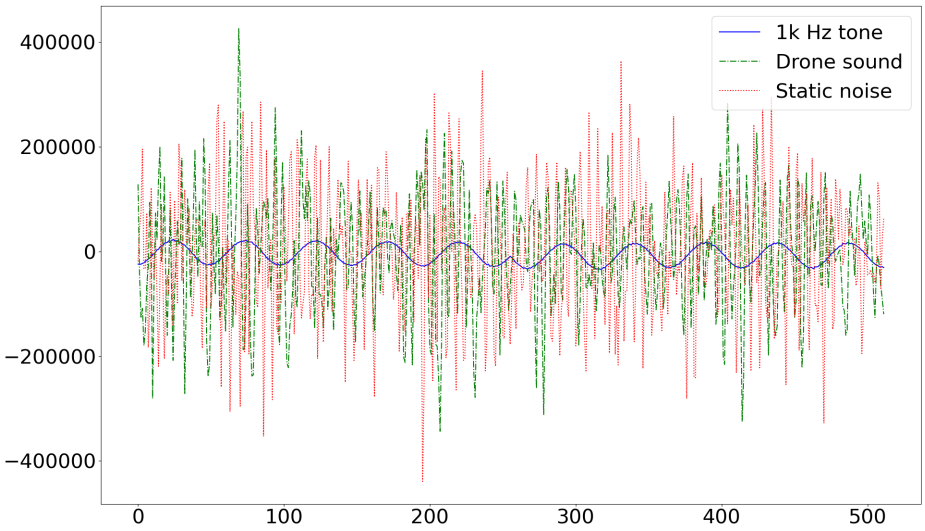
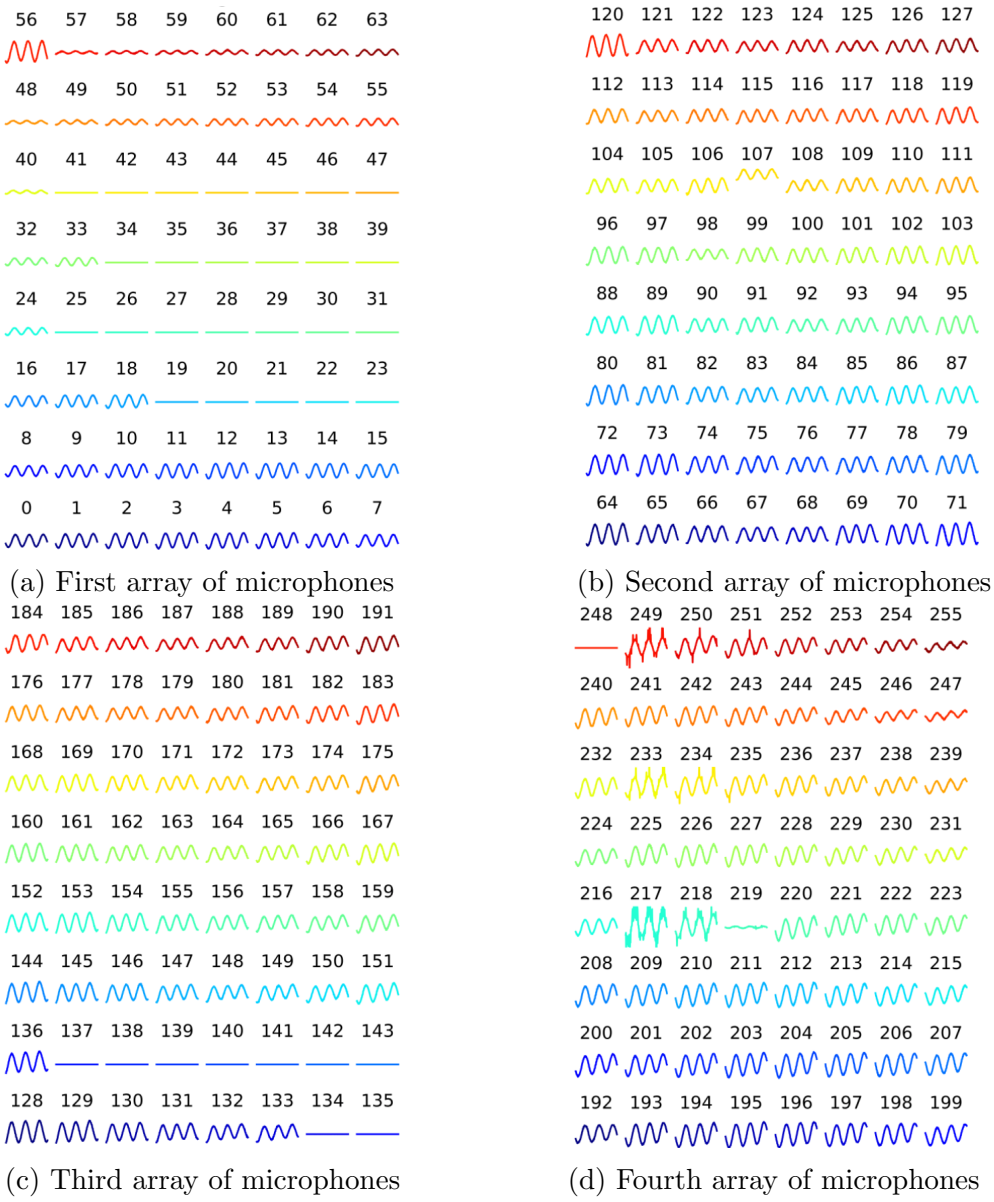


Figure 2: Sounds Used in Testing

The data displayed in Figure 2 was recorded using microphone number 79. Figures of each individual sound can be found in the appendix. The data in the *Acoustic Warfare* system is sent in *datablocks*, each containing 256 samples from all 256 microphones. The sound quality of the microphones was tested in the *Acoustic Warfare* project, and the results varied significantly. The performance of different microphones can be observed in the figures below[3]:



(a) First array of microphones

(b) Second array of microphones

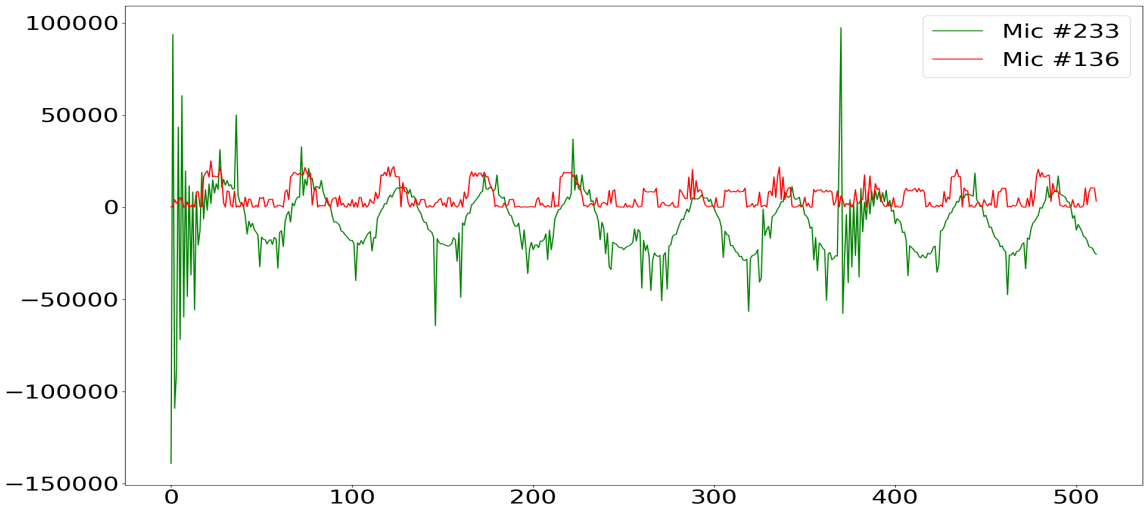
(c) Third array of microphones

(d) Fourth array of microphones

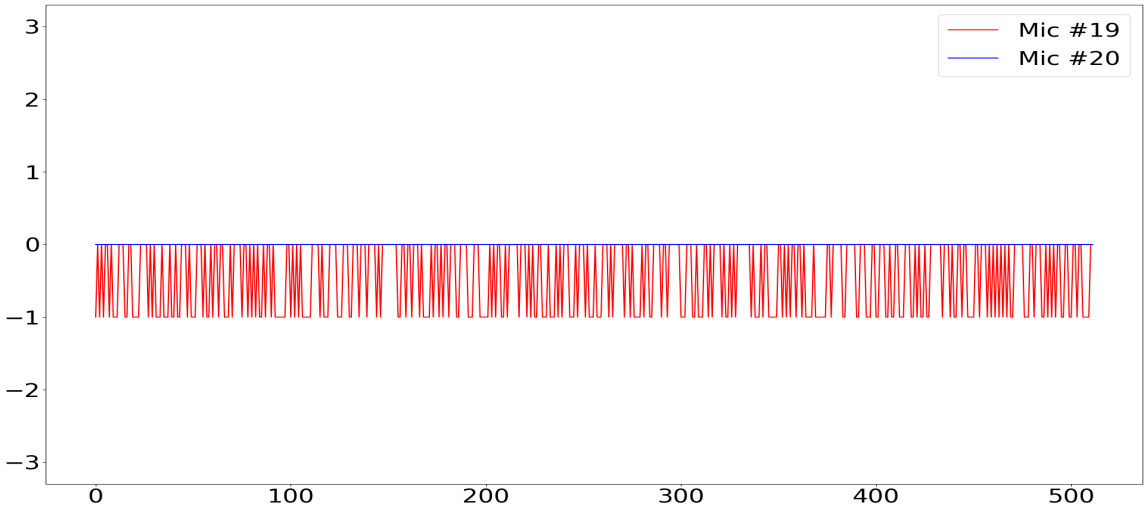
Figure 3: Displaying sound quality for all microphones

In Figure 3, it can be seen that most of the hardware has some faulty microphones. Only array 2, containing microphones numbered 64 to 127 in Figure 3b, is fully operational. The flat lines, such as those seen for microphones 19 and 20 in Figure 3a, indicate that the sound recorded with these microphones is silent. Larger peaks, such as the one observed for microphone 217 in Figure 3d, suggest that the recorded sound from these microphones is boosted to a tremendous volume. Additionally, there are microphones with the correct magnitude but other faults, such as unwanted spikes or failures to capture the correct sound wave. Clear examples of faulty microphones

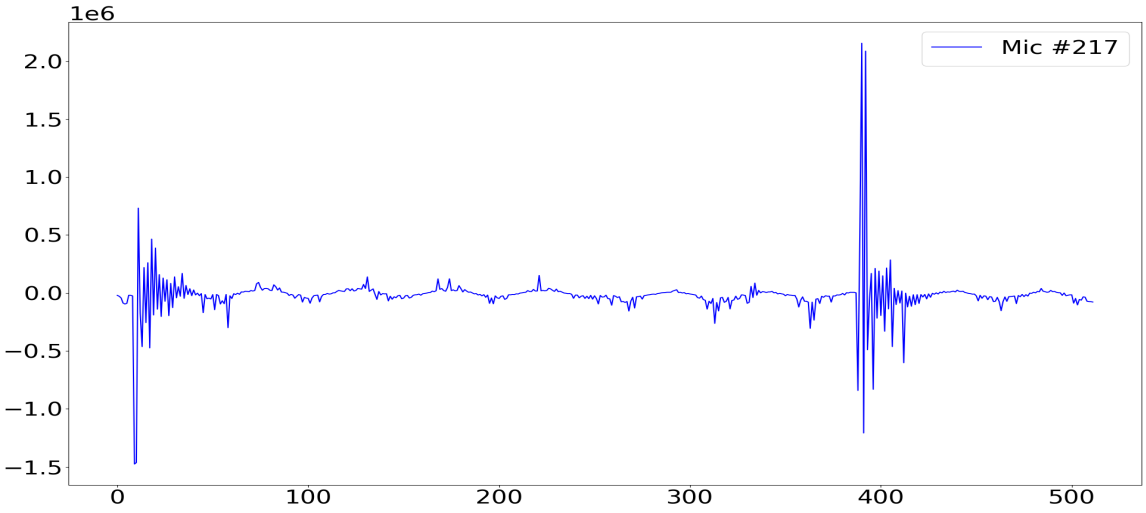
can be seen in the figures below:



(a) Example of faulty microphones



(b) Example of faulty microphones with low magnitude



(c) Example of faulty microphone with high magnitude

Figure 4: Displaying sound quality from different faulty microphones

The plots seen in Figure 4 depict the results of recording a 1 kHz tone using different faulty microphones. Microphone 233, displayed in Figure 4a, exhibits unwanted spikes, while microphone 136 in the same figure demonstrates how the microphone can fail to capture the correct sound wave. In Figure 4b, microphones 19 and 20 are examples of silent microphones. In Figure 4c, microphone 217 is shown with a large magnitude; please note the scale on the y-axis, which is 10^6. The individual plots of the microphones in Figures 4a and 4b have been included in the appendix.

5.2 Encoding Raw Data

Encoding the raw data values with Rice codes was tested to determine the achievable compression rate when only using coding without any modeling. The k value used for the Rice codes was calculated using equation 10. The test was performed on all three sound files, using both good and faulty microphones. The results can be seen in the figure below:

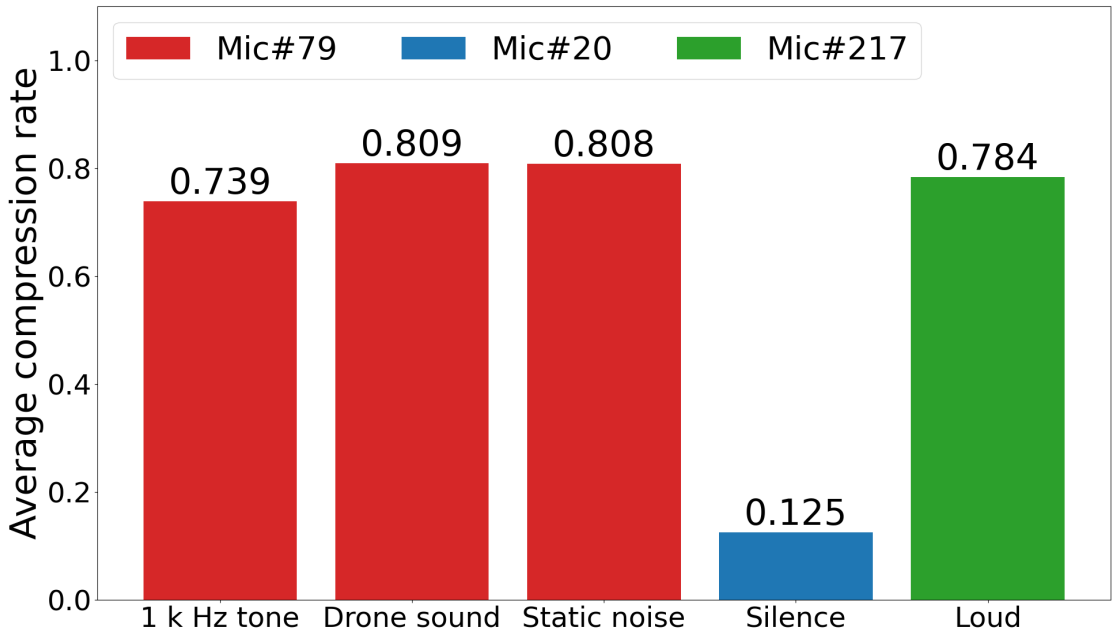


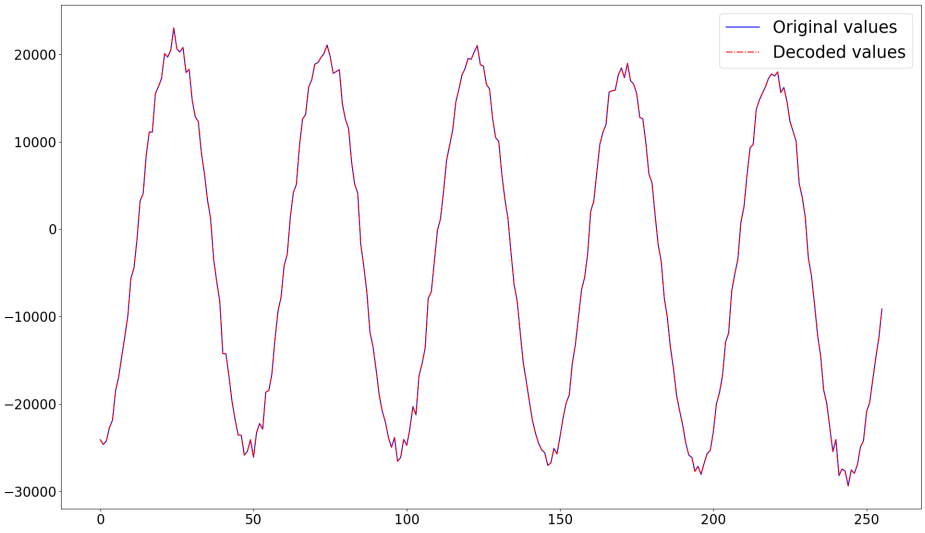
Figure 5: Rice coded data

In Figure 5, it can be observed that some compression gain is achievable by encoding the raw data without using any modeling.

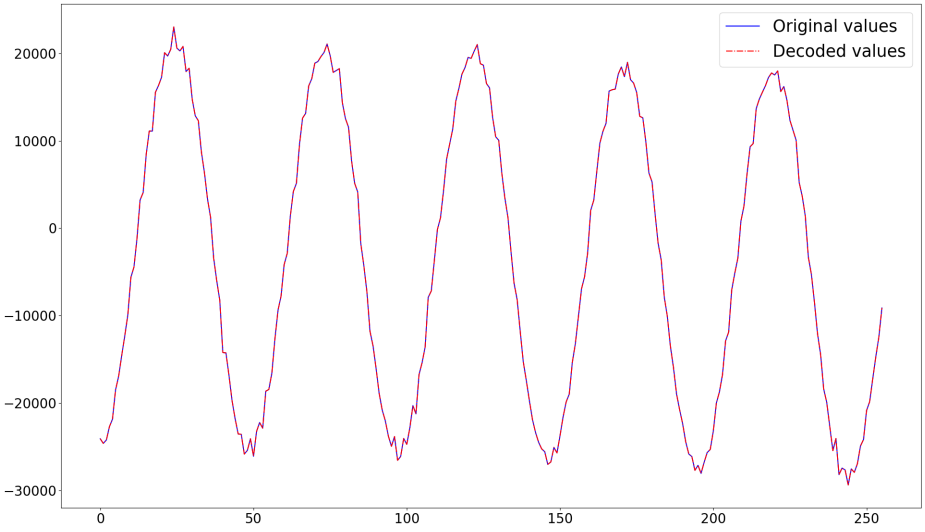
5.3 Shorten Python Implementation

5.3.1 Ability to recreate original values

The Shorten algorithm's ability to recreate the original inputs using both Rice and Golomb codes was tested. In the figures below, the performance using Shorten order 1 can be seen:



(a) Rice codes



(b) Golomb codes

Figure 6: Recreated values using Shorten order 1

Both plots of the original values and recreated values completely overlap in figures 6a and 6b, indicating that all values have been recreated correctly. Plots for the other

orders of Shorten can be found in the appendix, in figures 46, 47, and 48.

5.3.2 Evaluation of k value for Rice codes

Evaluation of varying k values for all orders of Shorten was conducted for all three sound files. The results are depicted in the figure below:

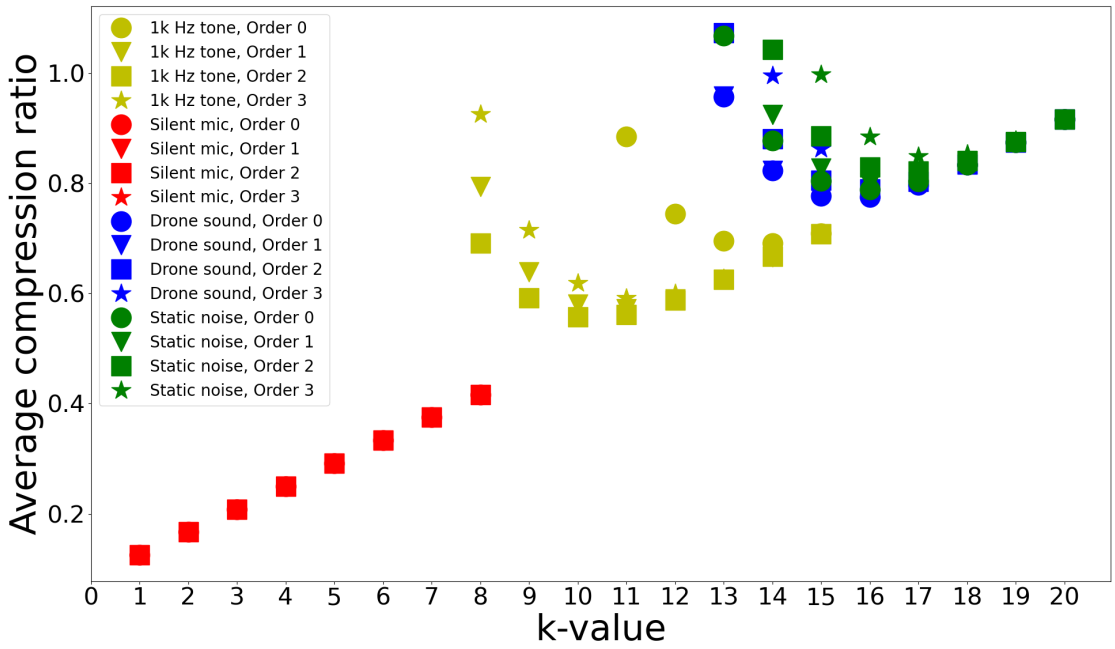


Figure 7: Compression rate for different k-values using Shorten

In Figure 7, it can be observed that the compression rate follows a parabolic curve for all cases except when k=1, which yields the best compression rate. In the appendix, Tables 2, 3, 4, and 5 contain the exact compression rates corresponding to some k-values for all orders of Shorten.

The best performing k-vlaues from Figure 7 was compared to the compression rate using k value derived from equation 10 and the compression rate from Golomb codes using the m-vlaue that yeilds the best results. The results for Shorten order 1 are presented in the figure below:

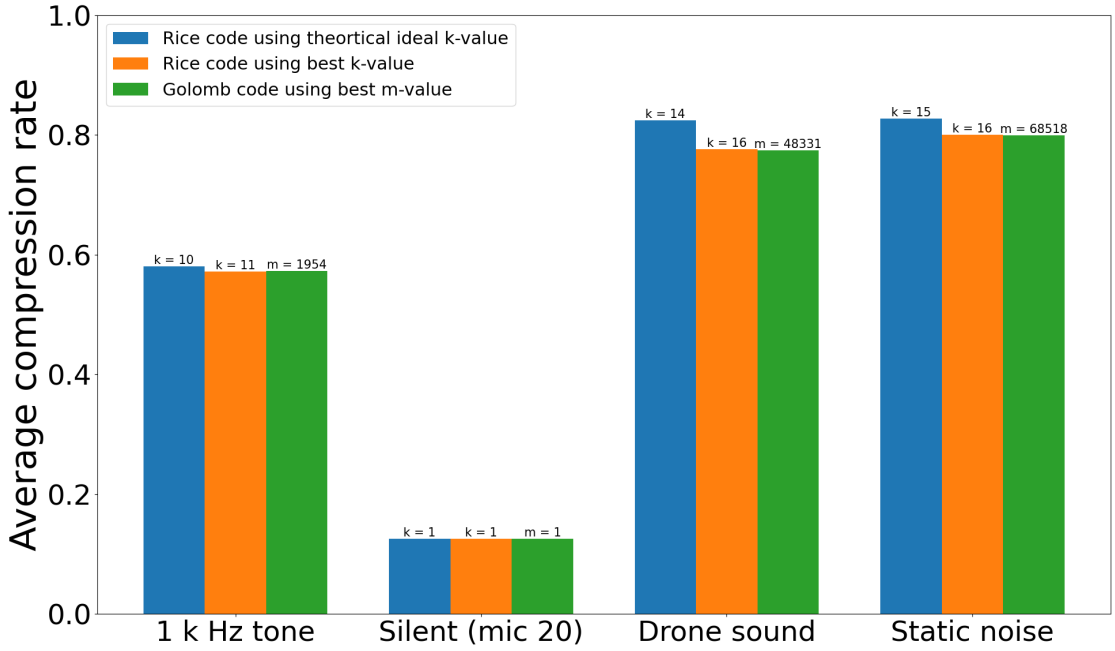


Figure 8: Average compression rates using Shorten order 1

In the appendix, figures for the other orders of Shorten can be found. In Figure 8, the performance of using Rice and Golomb codes is nearly identical when the k- and m-values are chosen correctly. Comparing the k-values that provide the best compression rate to the ideal k-values according to equation 10, it is evident that for all cases where k=1 is not the best k-value, they do not match. The k-value from equation 10 is always one or two values too low to achieve the best compression rate. This observation also applies when examining Figures 49, 50, and 51 in the appendix.

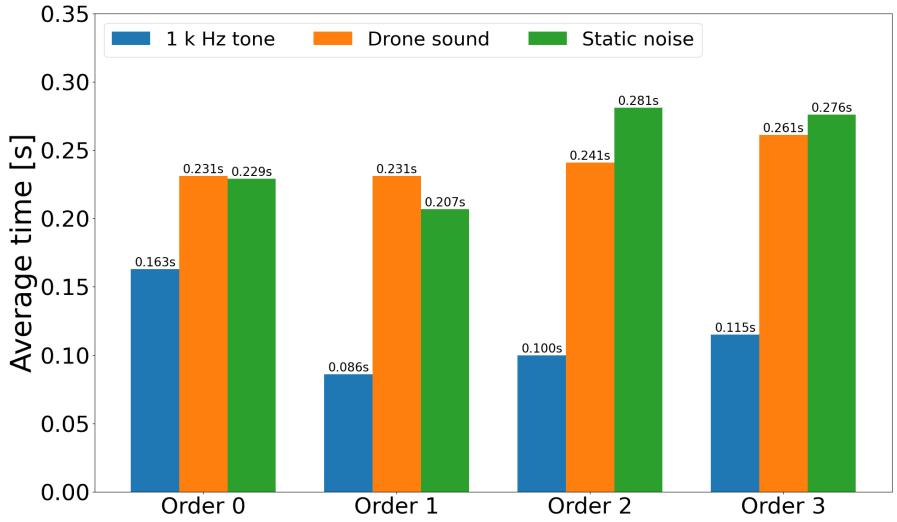
The equation for calculating the ideal k-value is adjusted based on the test data. Since the curves in Figure 7 are all parabolic except when k = 1, incrementing the result from equation 10 by 1 for all values where the result is not 1 will achieve a better compression rate. Therefore, the adjusted equation is as follows:

$$\begin{aligned} \text{E[x]} > 6.64 \rightarrow k = 1\log_2(\log(2) \cdot \mathbb{E}[x]) + 1 \\ \text{E[x]} \leq 6.64 \rightarrow k = 1 \end{aligned} \tag{19}$$

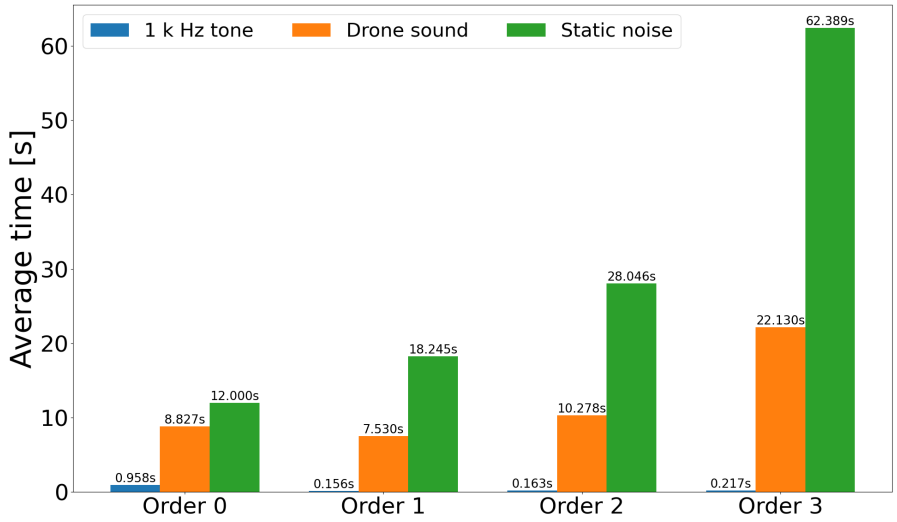
When $\mathbb{E}[x] = 6.64$, equation 10 gives k = 1; therefore, that is the limit set in equation 19.

5.3.3 Time to recreate values from codewords

The time it took to recreate the original values from the codewords was tested using Shorten with both Rice and Golomb codes. The results can be seen in the figures below:



(a) Using Rice codes



(b) Using Golomb codes

Figure 9: Average time to recreate input values from codewords

Comparing the times in Figures 9a and 9b, it can be seen that Rice codes outperform Golomb codes in all cases.

5.3.4 Compression rate

The compression rate using Shorten with Rice codes was evaluated using equation 19 to calculate k-values for Rice codes. The result can be seen in the figure below:

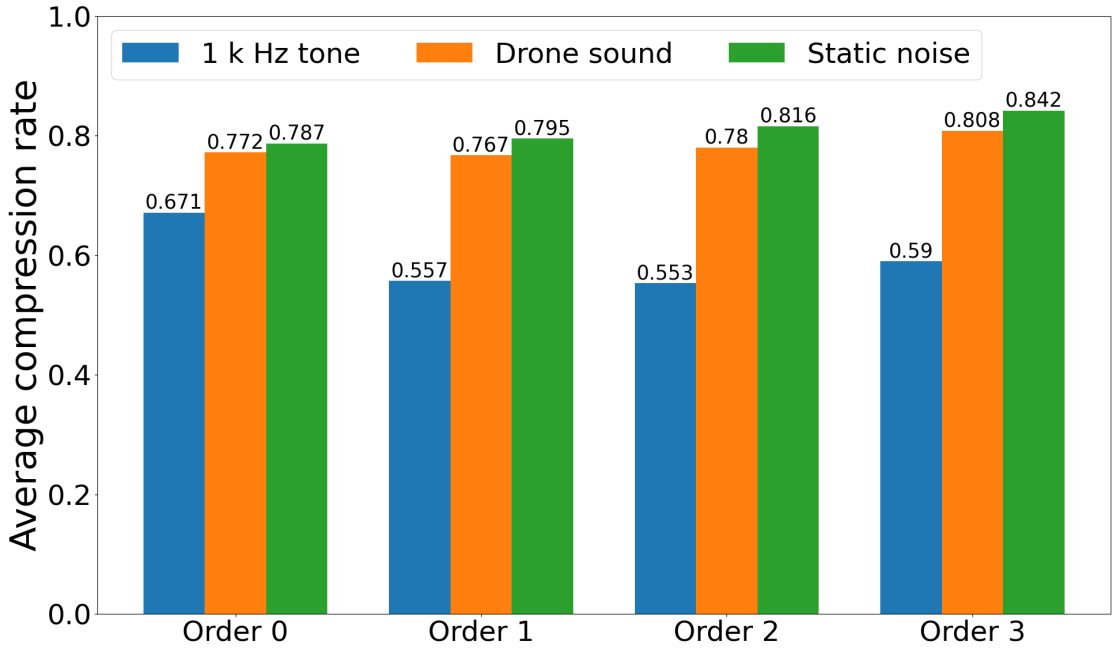


Figure 10: Compression rate using Shorten

In Figure 10, it can be observed that using Shorten with Rice codes achieved some compression for all orders on every sound file. The compression algorithm performed best with the 1 kHz tone sound file, with Shorten order 1 achieving the highest compression rate. For drone sound and static noise, the performance was similar for all orders of Shorten, with order 3 showing somewhat worse performance for both sound files.

The compression rates displayed in Figure 10 take into account the metadata that needs to be sent to the decoding side. For Shorten, the only metadata required is the k-value used for the Rice codes. This was set to 5 bits, as k is never less than 1, allowing k to take values from 1 to 33. The metadata for k was sent with every codeword, and one codeword was sent for every microphone in a datablock.

5.4 LPC Python Implementation

5.4.1 Ability to recreate original values

For LPC to accurately recreate the original input signal, the coefficients used on the encoding side must be sent as metadata to the decoding side. To test the ideal case of recreating the data with the full size of the coefficients, LPC order 1 with Rice codes was examined. The results are displayed in the figure below:

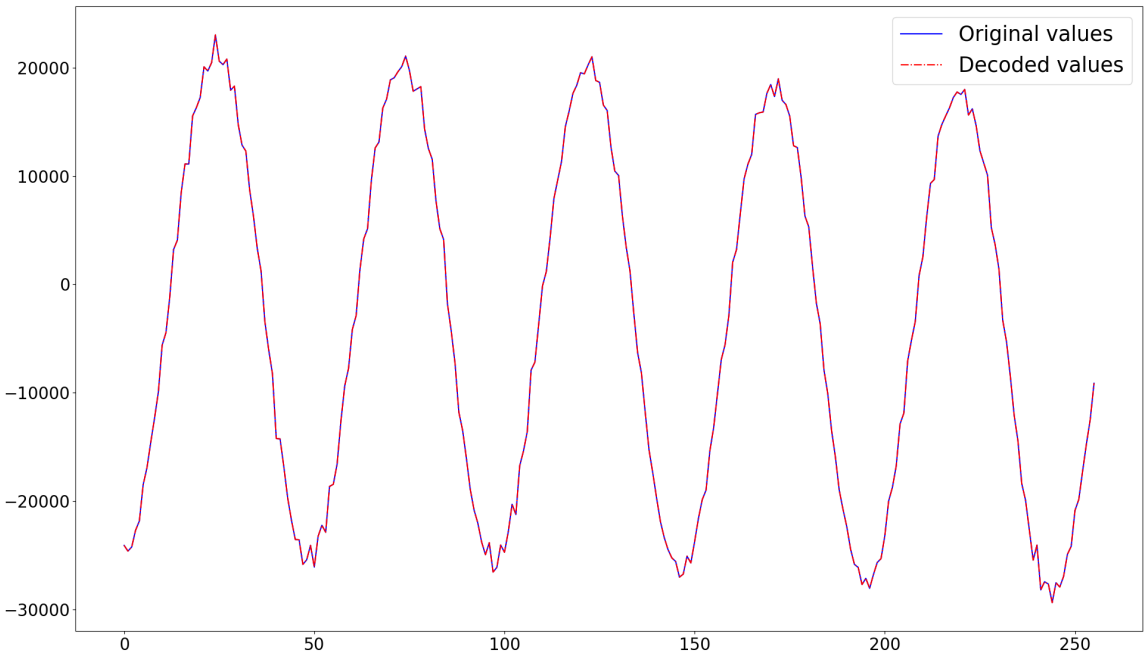


Figure 11: Recreate original values using LPC order 1 and Rice codes

In Figure 11, the original values and decoded values appear to be completely overlapping, suggesting successful recreation. However, when plotting the difference between them (original values - decoded values), it becomes evident that this is not the case. The results from this analysis are shown in the figure below:

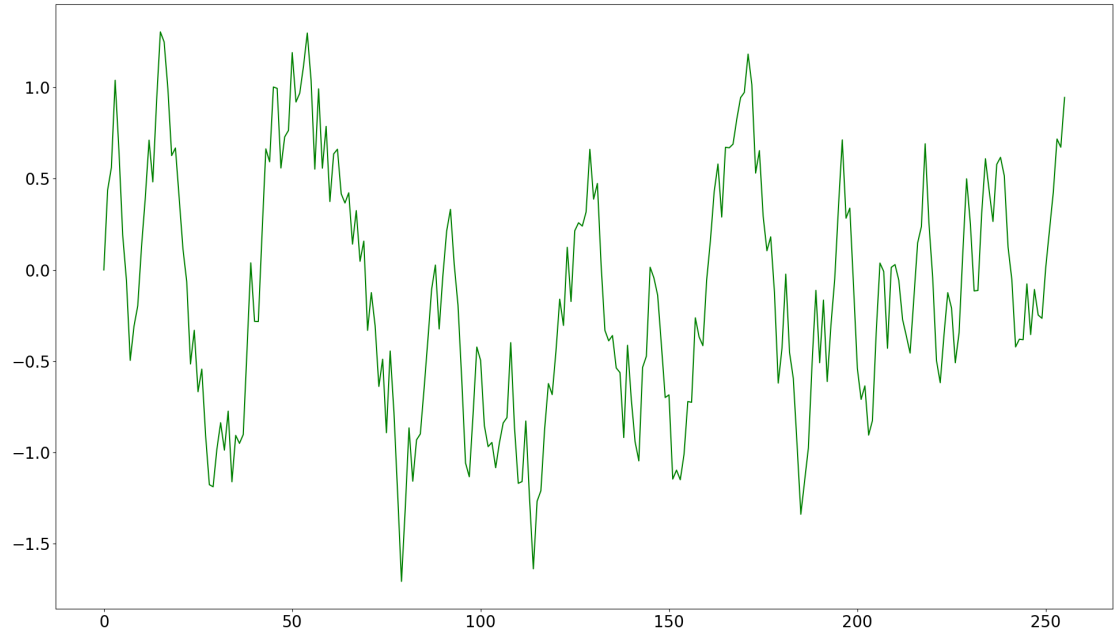
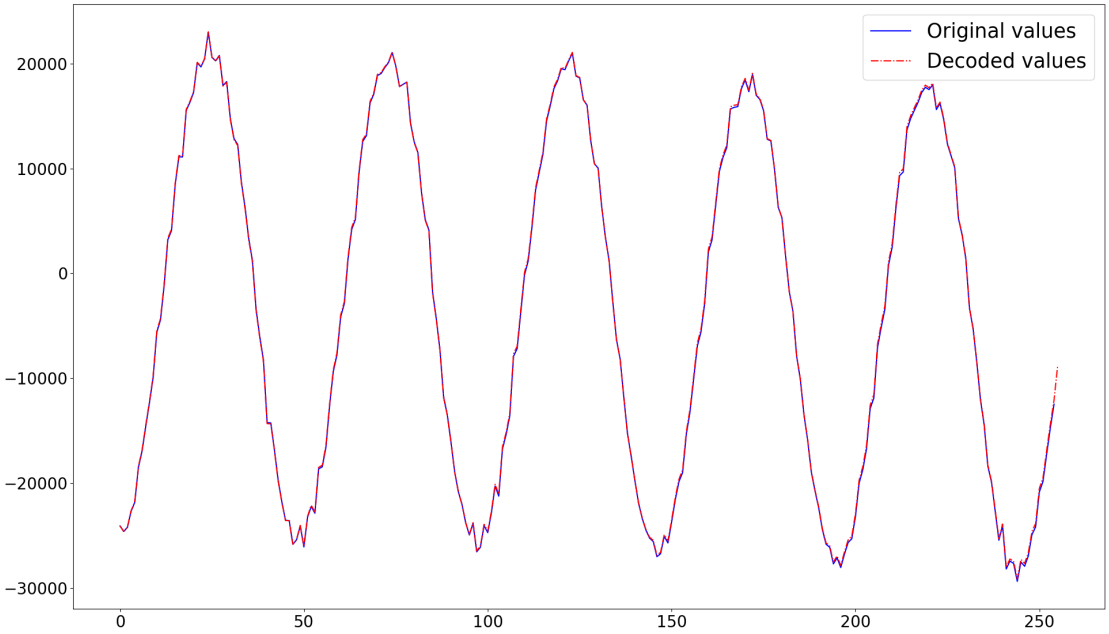


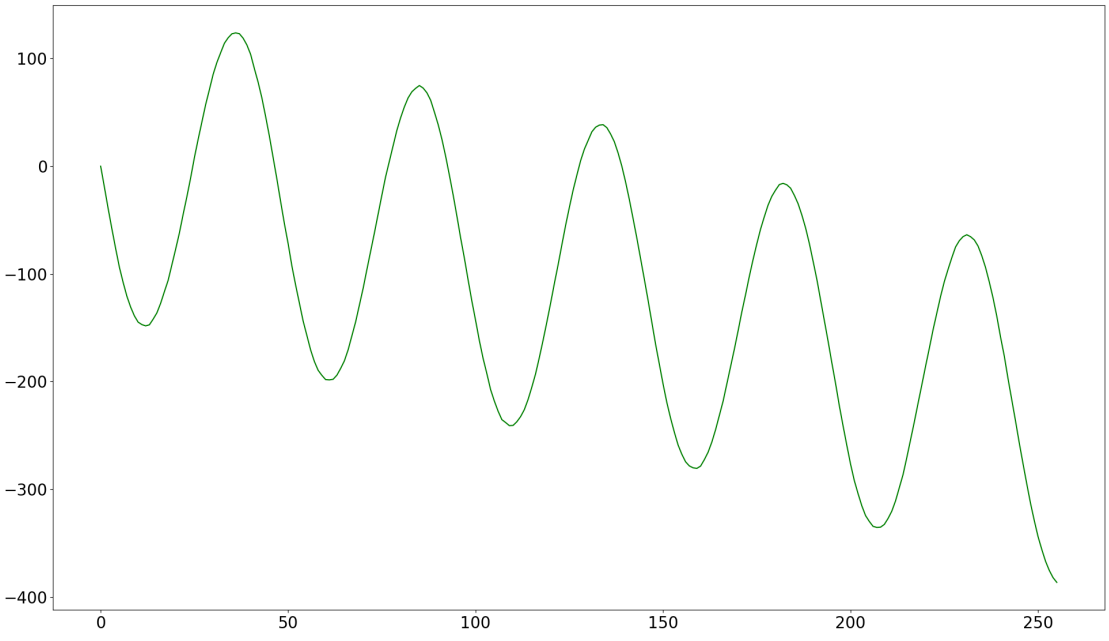
Figure 12: Plotting the difference between original values and decoded values

In Figure 12, the difference between the original values and decoded values seen in Figure 11 has been plotted. If the two plots in Figure 11 were completely overlapping, the plot in Figure 12 would always be zero. Therefore, Figure 12 indicates that LPC is unable to recreate the original values completely even with perfect coefficients. The results for LPC order 2-5 can be found in the appendix Figures 52, 53, 54, and 55, along with the results when using Golomb codes in Figures 56, 57, 58, 59, and 60.

When using a set number of bits to include decimals for the coefficients in the meta-data, the performance deteriorates further. The performance was tested for LPC orders 1 through 5 using Rice codes, with 10, 15, and 20 bits used to encode the decimals for each coefficient. The results for LPC order 1 can be seen in the figures below:

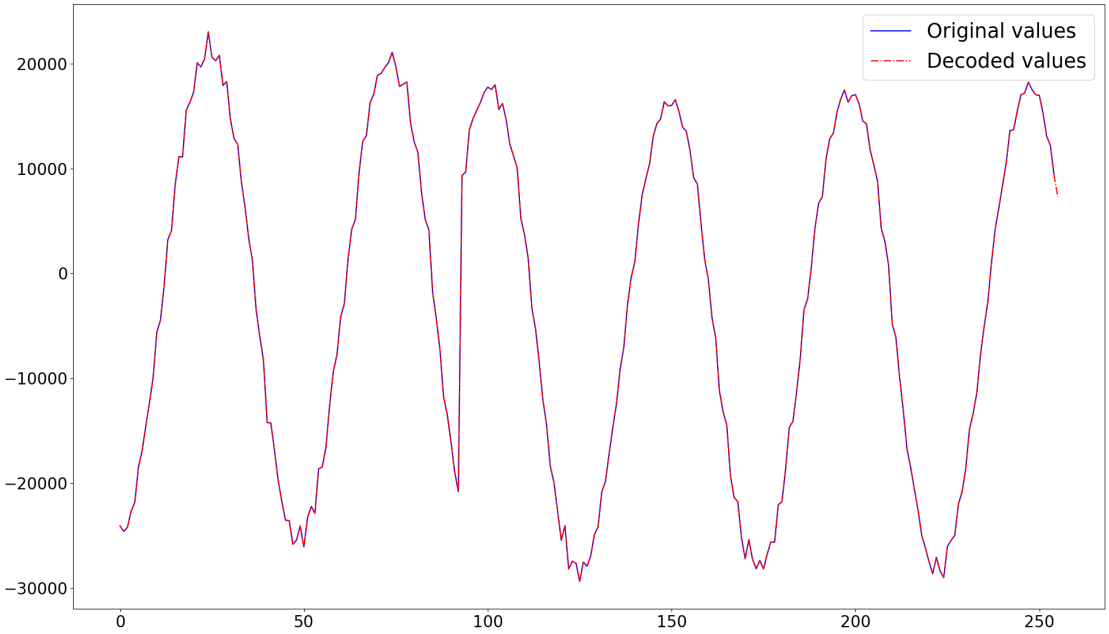


(a) Original and recreated signal

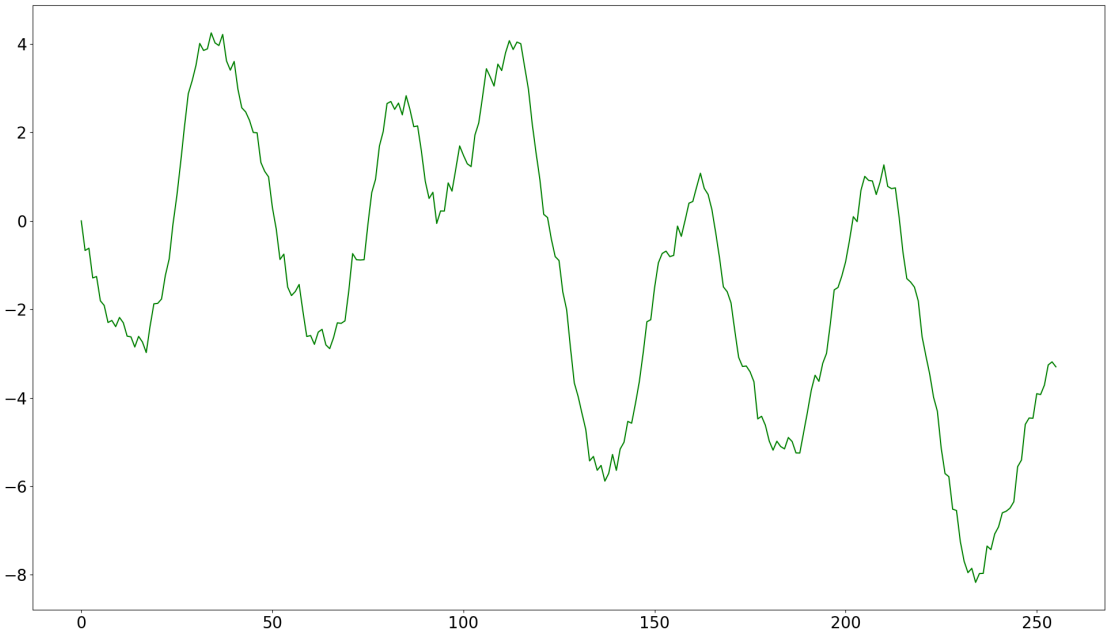


(b) Original signal - recreated signal

Figure 13: Recreate 1 k Hz tone using LPC order 1 and Rice codes, 10 bits metadata for decimals in coefficients (Note the difference in scale between Figure a and b)

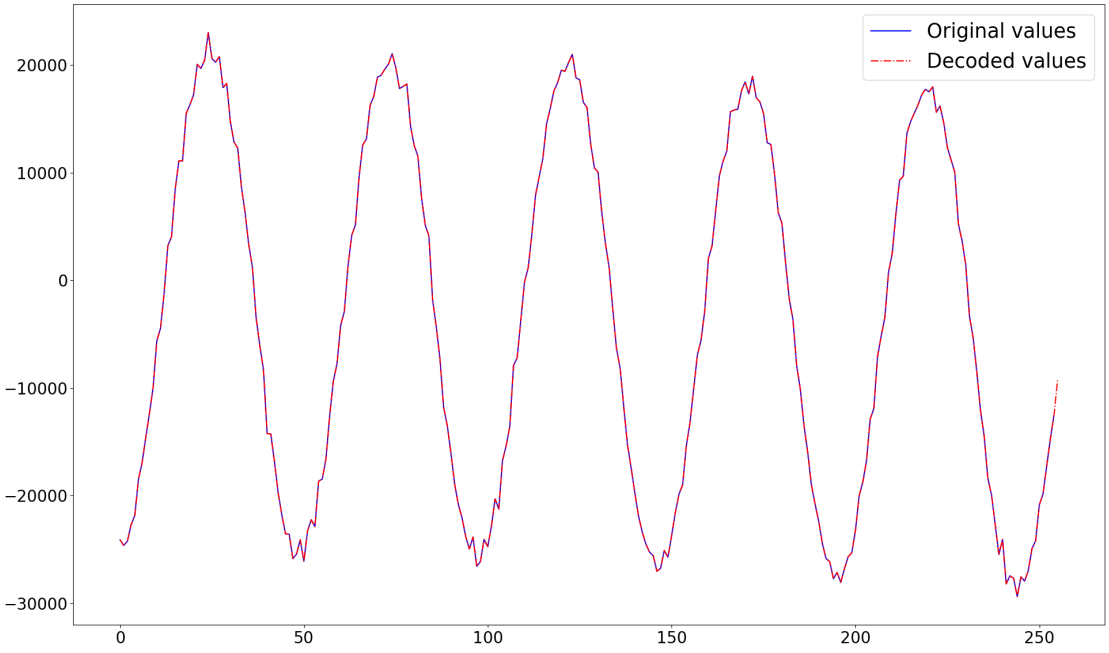


(a) Original and recreated signal

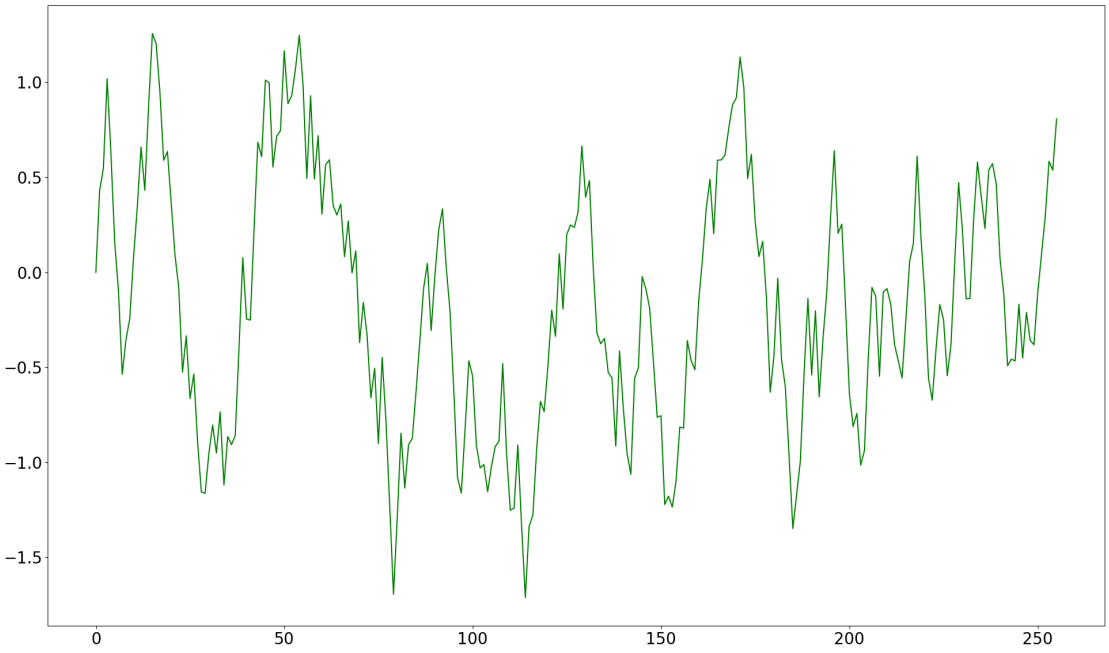


(b) Original signal - recreated signal

Figure 14: Recreate 1 k Hz tone using LPC order 1 and Rice codes, 15 bits metadata for decimals in coefficients (Note the difference in scale between Figure a and b)



(a) Original and recreated signal



(b) Original signal - recreated signal

Figure 15: Recreate 1 k Hz tone using LPC order 1 and Rice codes, 20 bits metadata for decimals in coefficients (Note the difference in scale between Figure a and b)

Comparing the plots in Figures 13b, 14b, and 15b, it can be seen how the error

shrinks when the amount of metadata is increased. Comparing Figure 12 and 15b, the fault is more or less the same. The results for the other orders of LPC can be seen in Appendix Figures 61, 65, 69, 62, 66, 70, 63, 67, 71, 64, 68, and 72.

5.4.2 Evalute k value for Rice codes

To evaluate Rice codes in combination with LPC, compression rates were tested for LPC orders 1 to 5 using varying k values to encode the residuals. The results are depicted in the figure below:

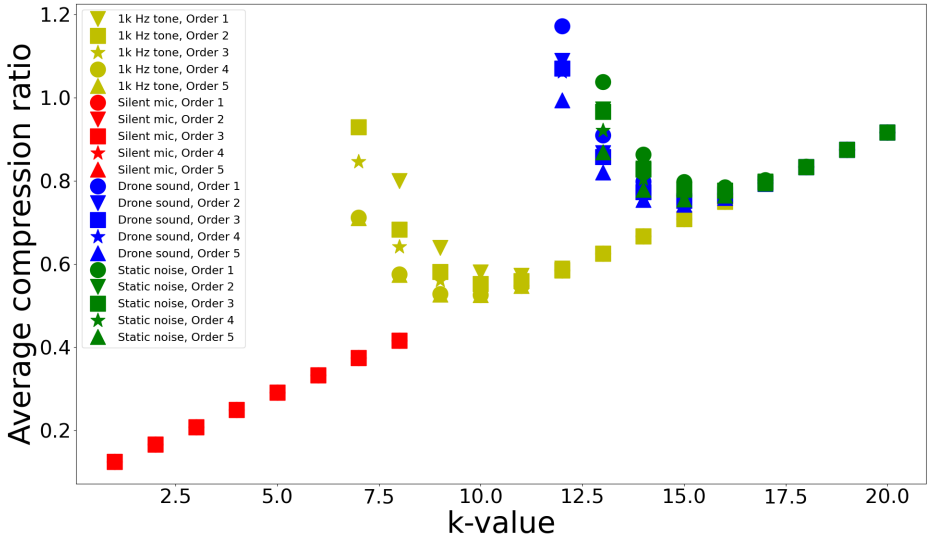


Figure 16: Compression rate for different orders of LPC using Rice codes

The result in Figure 16 shows that the compression rate follows a parabolic curve as k increases, in all cases except when k = 1 gives the best compression rate. For a more detailed overview of the compression rates using different k-values, more information can be found in the appendix in Figures 6, 7, 8, and 9.

To evaluate the performance of equation 10 in combination with LPC, the ideal k-value according to theory was compared to the k-value giving the best compression rate. The compression rate when using LPC in combination with Golomb codes was also taken into account, with the m-values giving the best compression rate. The result for LPC order 1 can be seen in the figure below:

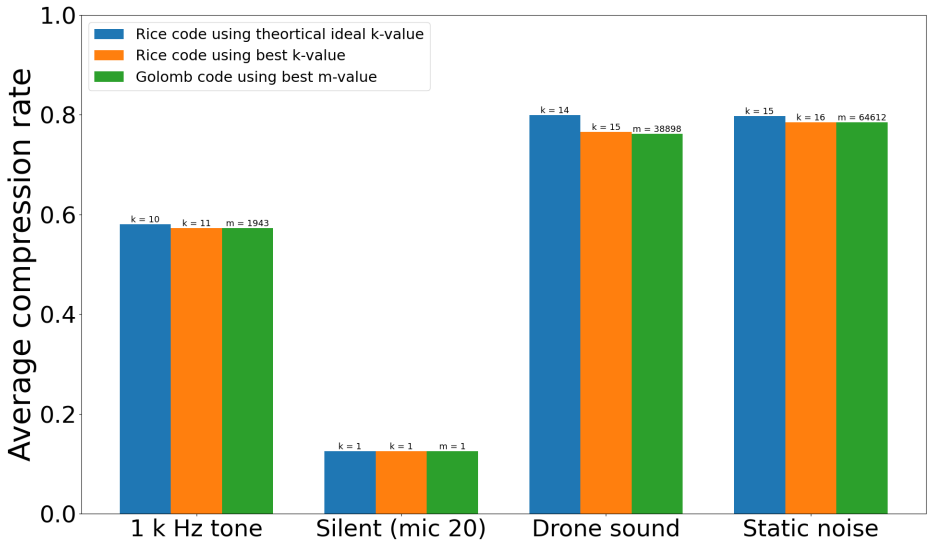
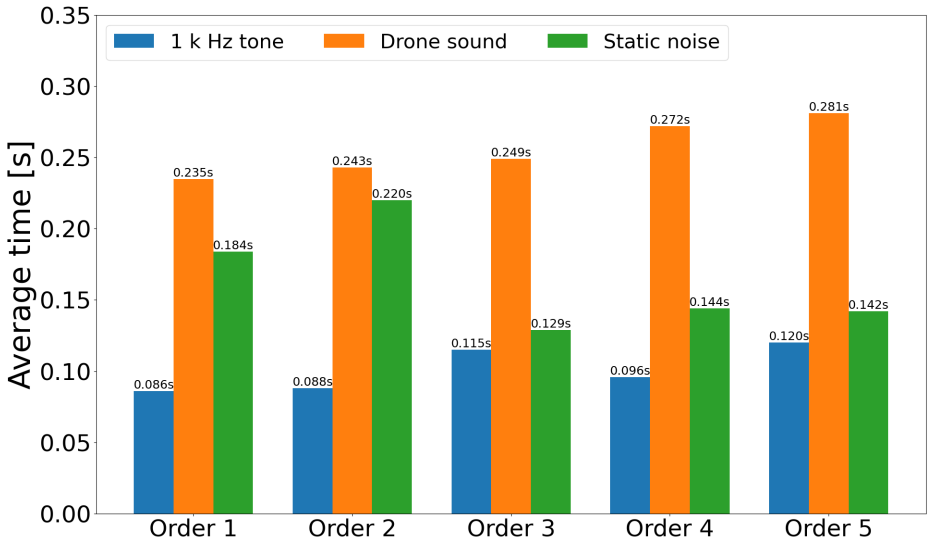


Figure 17: Average compression rates using LPC order 1

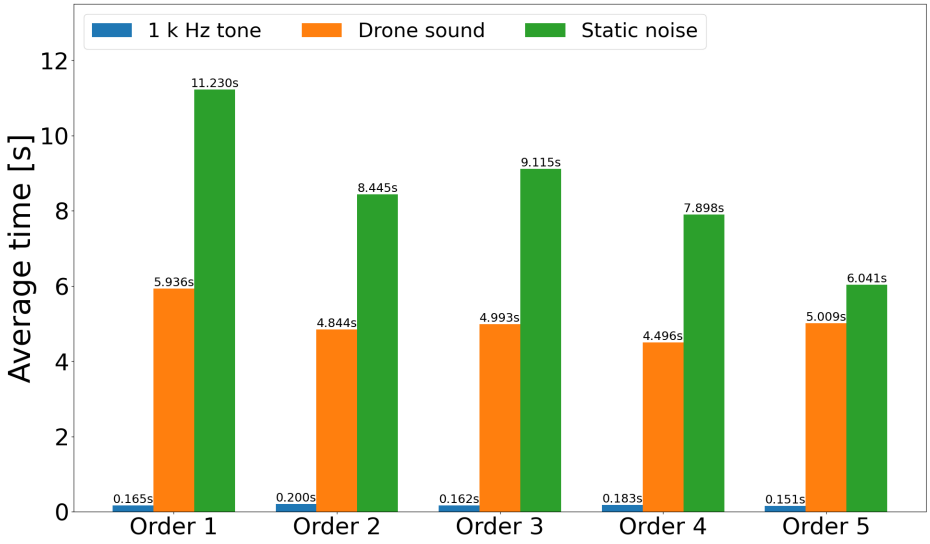
In Figure 17, the performance of Rice and Golomb codes are similar when the best k- and m-values are chosen. It can also be seen that the best k-value is consistently one or two values higher than the ideal k-value in theory, except when k = 1 is the best value. This holds true for all orders of LPC; figures for the other orders can be found in Appendix Figures 73, 74, 75, and 76. Therefore, equation 19 is to be used when calculating the k-value for Rice codes using LPC.

5.4.3 Time to recreate values from codewords

Comparing the timing performance of different orders of LPC with both Rice and Golomb codes is of interest when choosing an encoder for the model. The time it took for both Rice and Golomb codes to recreate values can be seen in the figures below:



(a) Using Rice codes



(b) Using Golomb codes

Figure 18: Time to recreate original values from codewords

The results in Figure 18 indicate that Rice codes outperform Golomb codes in all cases.

5.4.4 Compression rate

The performance of LPC was evaluated with taking metadata into account. The metadata consisted of 5 bits for the k-value and 20 additional bits for the coefficients. For the coefficients, 20 bits were used to ensure accurate recreation of the data. The metadata was sent with every codeword, and for a datablock one codeword was sent for every microphone. The result can be seen in the figure below:

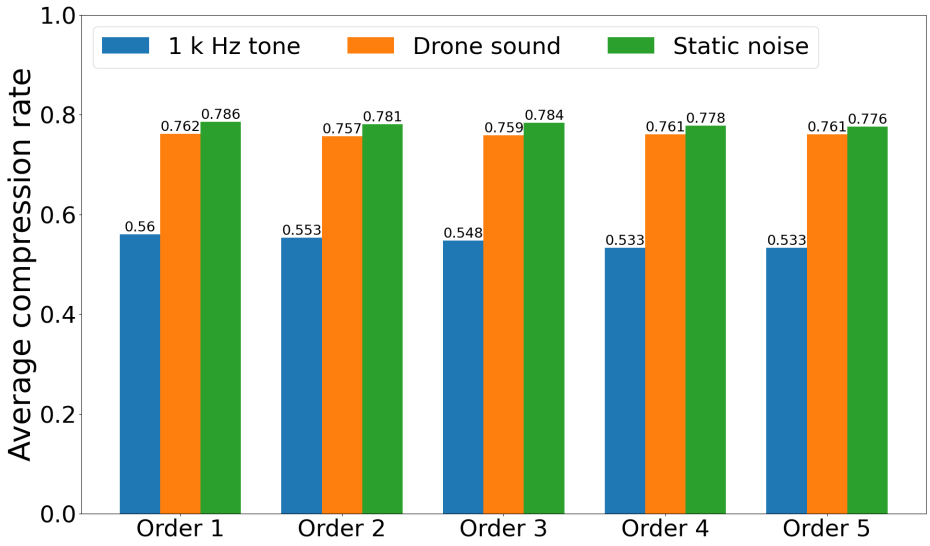


Figure 19: Compression rates using LPC

In Figure 19, it can be seen that all orders of LPC perform similarly regardless of the sound file, even though more metadata is sent for the higher orders. It is also clear that the algorithm's performance is better on the 1 kHz tone compared to the other two sound files, which exhibit similar performance.

5.5 FLAC Python Implementation

The FLAC algorithmen utilises the previously tested algorithms Shorten and LPC with Rice codes, along with RLE. The test done on Shorten and LPC indicated that equation 19 gave the best k values to encode the residuals using Rice codes. Equation 19 is therefore used in the FLAC algorithm. It is possible to limit the maximum LPC order that the FLAC algorithm uses. The standard is to limit the LPC order to 32, which was used for the test performed using FLAC.

5.5.1 Time to recreate values from codewords

The time it took to recreate the data from codewords using FLAC can be seen in the figure below:

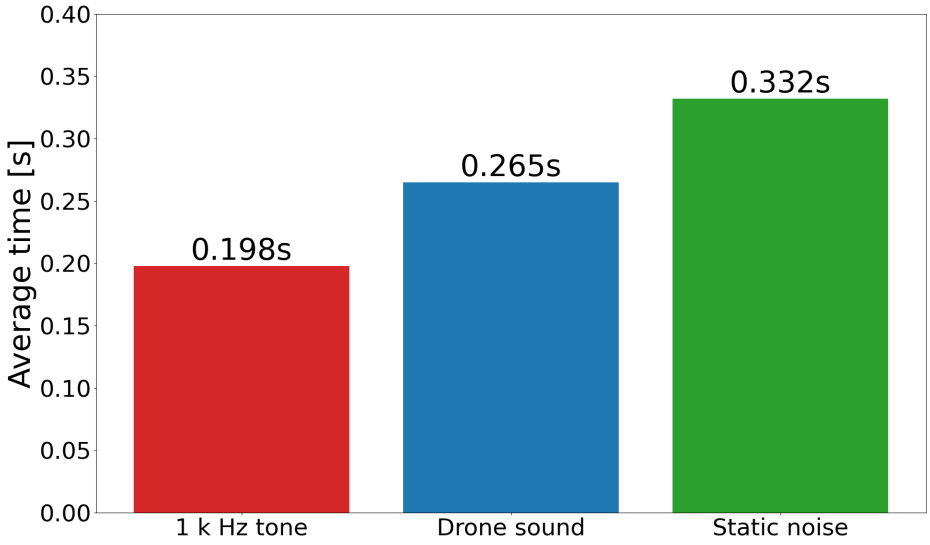


Figure 20: Time to recreate values using FLAC

5.5.2 Compression rate

The average compression rate for FLAC was tested for all 3 sound files. The results can be seen in the figure below:

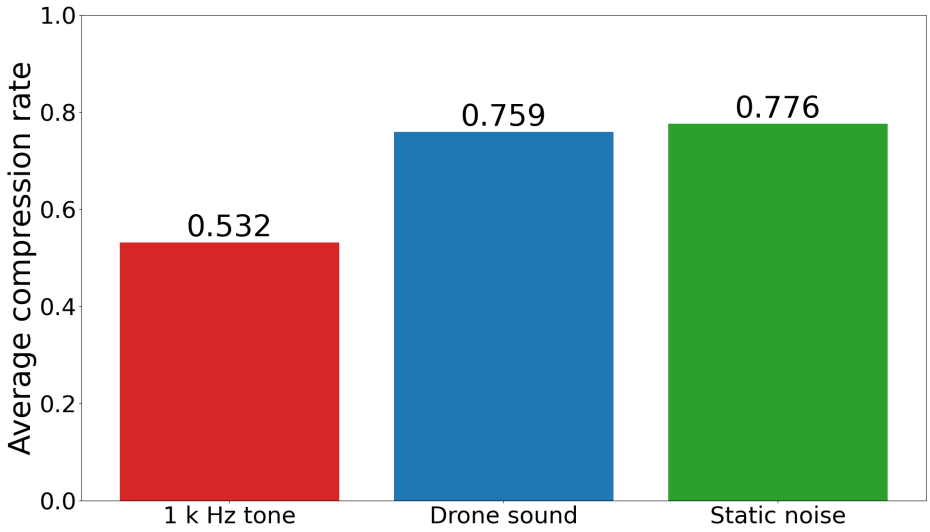


Figure 21: Compression rate using FLAC

In Figure 21 it can be seen that the FLAC algorithm perform best on the 1k Hz tone audio. For the other two sound files the performance is similar, with the Drone sound being slightly better.

The compression rate in Figure 21 takes the metadata needed to decode the codewords into account. The FLAC algorithm sends one codeword for every microphone in a datablock. All codewords contain metadata for what encoder that have been used, this is represented in 6 bits. It needs to be able to indicate RLE, Shorten order 0-4, and LPC order 1-32, totaling to 38 different values. If Shorten or LPC is used the codeword also contains metadata for the k-value used with the Rice codes, this is represented in 5 bits. For LPC the coefficients is also sent as metadata. The user is able to decide how many bits to use to represent the decimals for every coefficient, in the test results in Figure 21 20 bits is used.

5.6 Adjacent Python Implementation

5.6.1 Ability to recreate original values

The ability of the Adjacent algorithm to compress and recreate the original data was tested, and the result can be seen in the figure below:

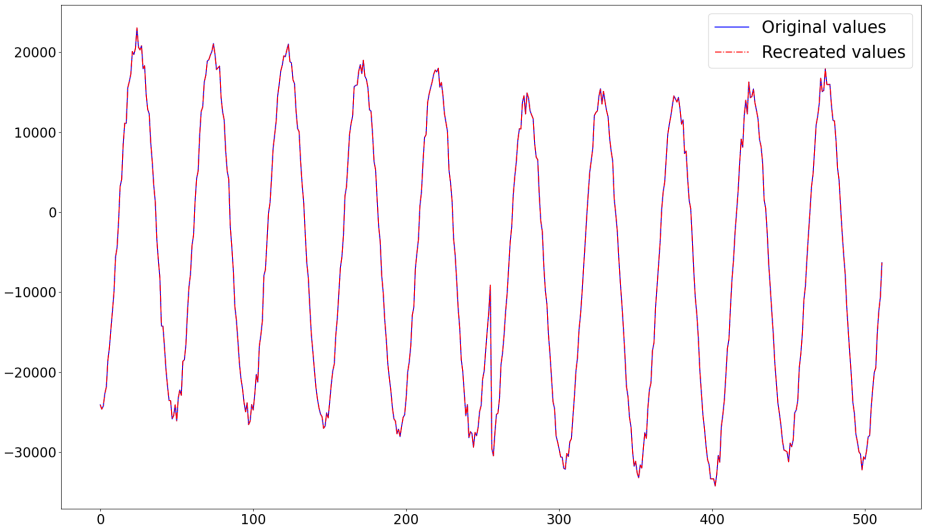


Figure 22: Using Adjacent to compress and recreate the data

In Figure 22, it can be seen that the original and recreated signals completely overlap, indicating that all values have been successfully recreated.

5.6.2 Evaluate k value for Rice codes

The best k-value when using Adjacent with Rice codes was tested by finding the compression rate for different k-values, the results can be seen in the figure below:

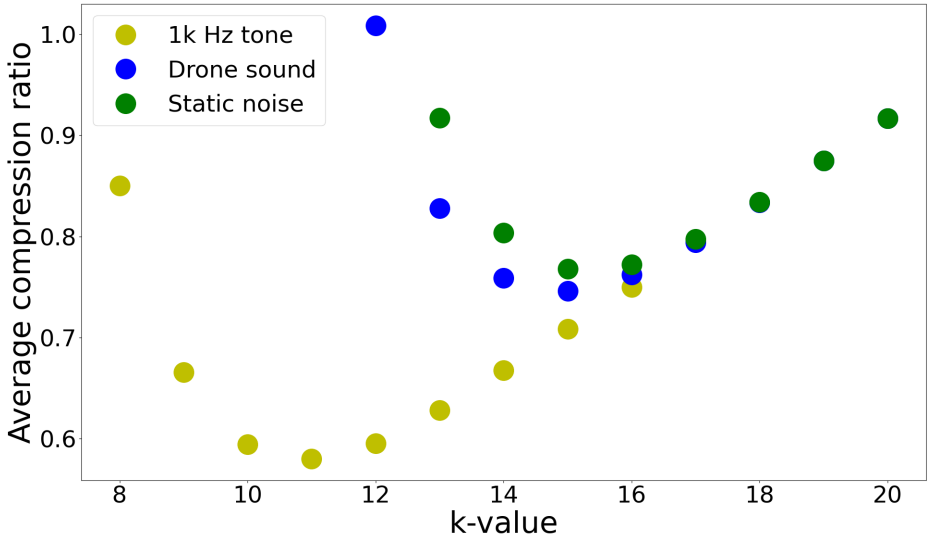


Figure 23: Compression rate for Adjacent using Rice codes at different k-values

In Figure 23, it can be seen that the compression rate follows a parabolic curve. More detailed information on the compression rate for different k-values can be found in the appendix table 10. The best k-value from Figure 23 was compared with the ideal k-value from Equation 10, alongside the compression rate when Golomb codes are used with the m-value that gives the best compression rate. The result can be seen in the figure below:

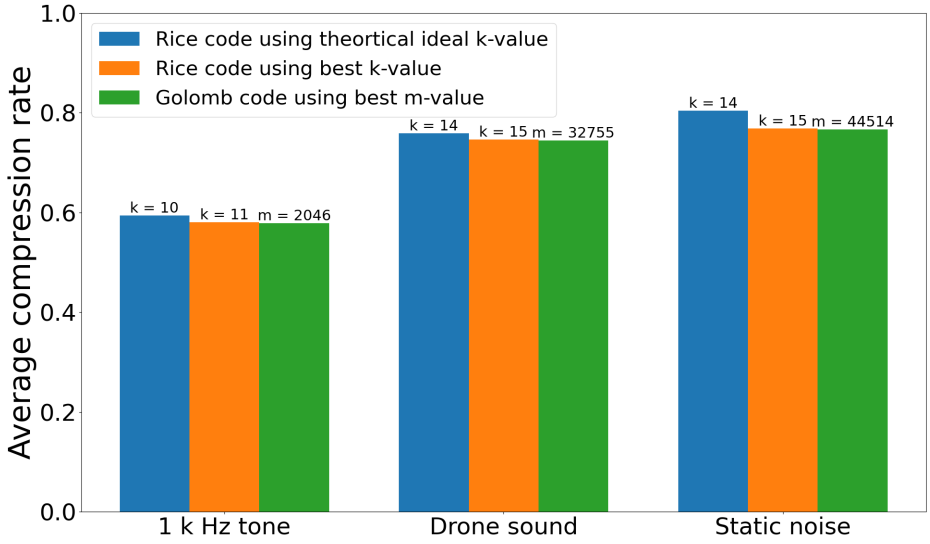


Figure 24: Compression rate using Adjacent with ideal k, best k, and best m values

In Figure 24, it can be seen that Rice and Golomb codes perform similarly when

the best k- and m- values are chosen. It can also be observed that the best k-value is always larger than the theoretically ideal k-value by 1. This indicates that it is better to use Equation 19 when calculating k-values for Adjacent with Rice codes. However, it is not proven from this test that when $\mathbb{E}[x] \leq 6.64$, k = 1 gives the best compression rate since this have not been tested.

5.6.3 Time to recreate values

The performance of Rice versus Golomb codes in time to recreate original values in combination with Adjacent can be seen in the figure below:

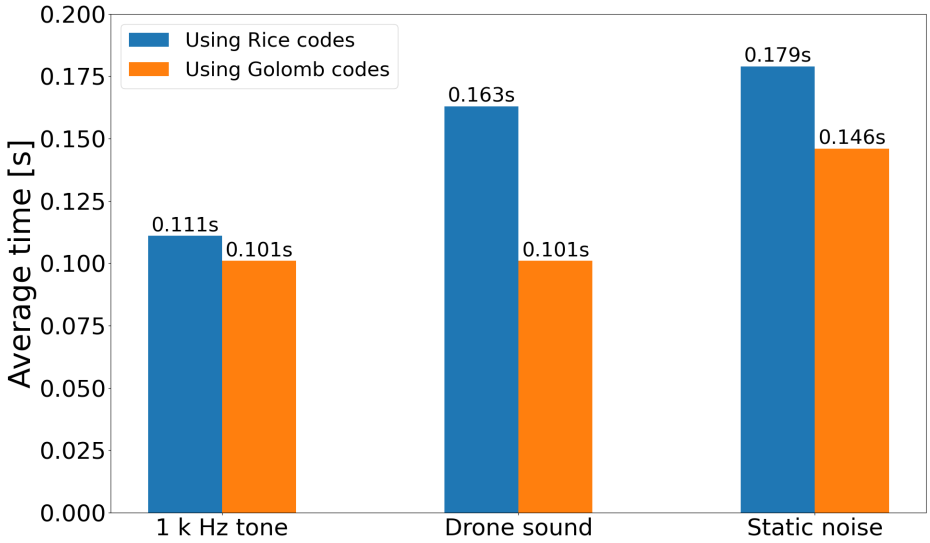


Figure 25: Time to recreate original values from codewords

It can be observed in Figure 25 that Rice and Golomb codes have similar performance, but Golomb codes slightly outperform Rice codes for all sound files.

5.6.4 Compression rate

The compression rate using Adjacent to model the input signal is evaluated using Rice codes with equation 19 to calculate k-values. The average compression rate for different sound files can be seen in the figure below:

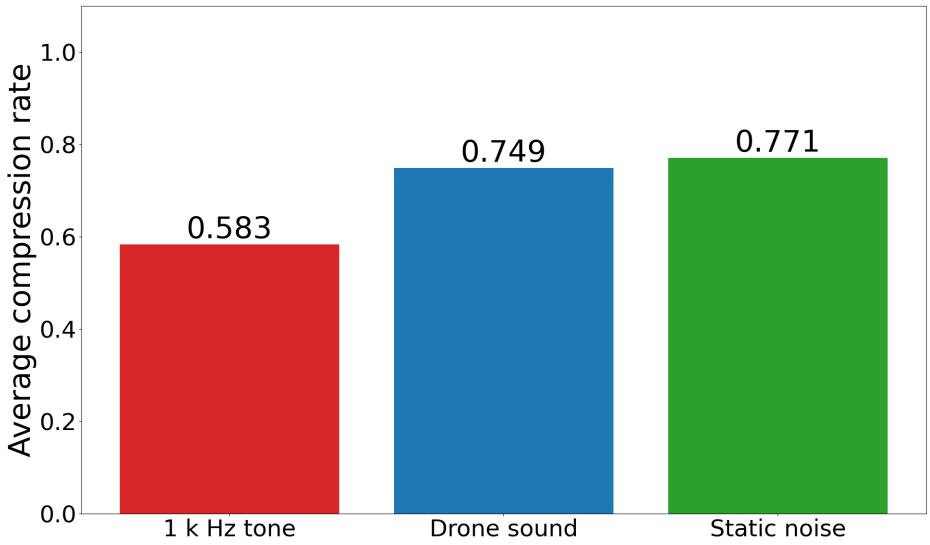


Figure 26: Average compression rate using Adjacent with Rice codes

In Figure 26, it can be seen that Adjacent performs best when compressing the audio from the 1 kHz tone. In the other cases, the performance is similar but slightly better when compressing the drone sound. The compression rate in Figure 26 takes into account the metadata needed to recreate the Adjacent values. The metadata needed for Adjacent is only the k-value used for the Rice codes, which is encoded in 5 bits and sent with every codeword. For Adjacent, one codeword is sent for every sample in the datablock, meaning 256 codewords are sent per datablock.

5.7 FLAC Modified Python Implementation

A modified version of FLAC using Adjacent instead of LPC has also been tested. The rationale behind this decision is that RLE, Shorten, and Adjacent are all easier to implement in an FPGA environment since none of them require handling floating-point numbers or division operations. The modified algorithm operates similarly to regular FLAC by calculating the compression rate for all available models and then selecting the best performer.

The algorithm was designed to read a full datablock of samples for a complete array of microphones and encode them. It allows the user to specify the number of microphones and samples. The algorithm automatically selects the best encoder for every codeword and can either send one codeword for every sample if the Adjacent model is the best performing model, or it sends one codeword for every microphone.

Additionally, the algorithm can group the residuals calculated with Adjacent by their microphones instead of by sample and encode them accordingly. This enables residuals calculated with Adjacent to be chosen as codewords even when the codewords are grouped by microphone. The residuals are encoded with Rice codes using Equation 19 to calculate the k value.

5.7.1 Time to recreate values from codewords

The performance of how fast FLAC modified can recreate the original input data was tested, and the result can be seen in the figure below:

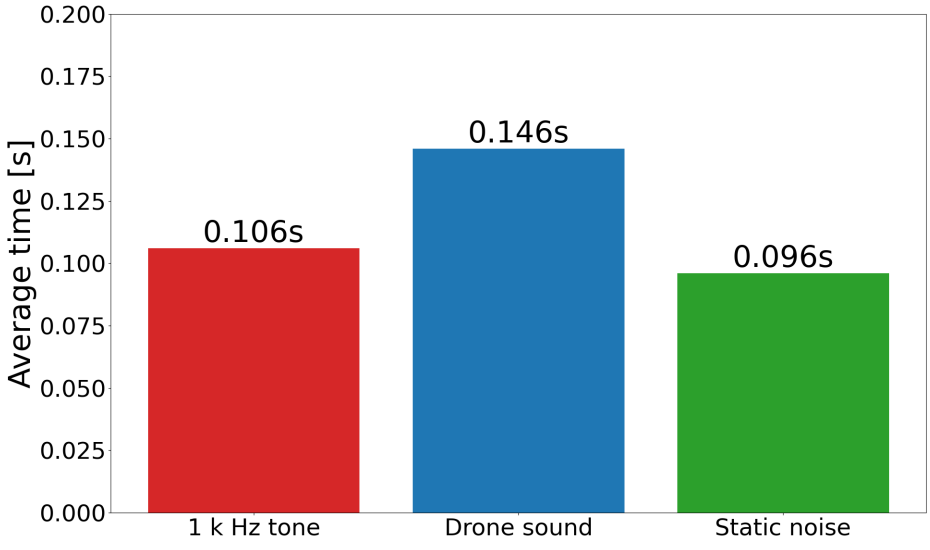


Figure 27: Time to recreate values using FLAC Modified

5.7.2 Compression rate

The compression rate using FLAC modified was also tested and can be seen in the figure below:

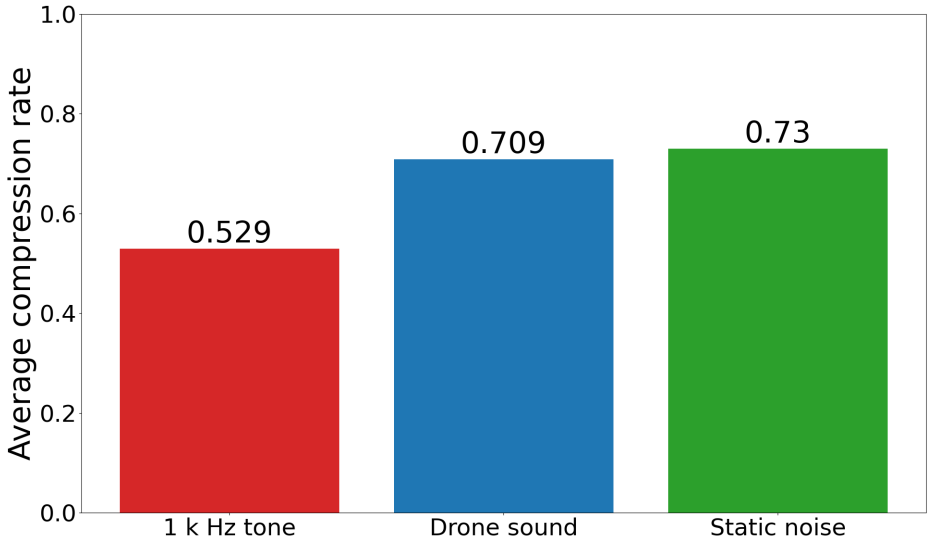


Figure 28: Average compression rate using FLAC Modified

In Figure 28, it can be seen that FLAC modified performs better on the 1 kHz tone audio, while the other two audio files have similar compression rates, with the Drone sound being slightly better. The compression rate displayed in Figure 28 takes the meta-data needed to decode the codewords into account. The meta-data needed for FLAC modified includes 5 bits for the k-value used for the Rice codes and 3 bits needed to indicate which encoders have been used. The encoders to choose from are RLE, Shorten order 0-4, and two versions of Adjacent. This totals to eight different encoders and matches the values 3 bits can take on perfectly. The meta-data needs to be sent for every codeword. FLAC modified can either choose to create codewords for every microphone or by sample, depending on which encoder is chosen. The meta-data needed to be sent is also taken into account when deciding on which encoder to choose.

5.8 Double Compression Python Implementation

The performance of Rice codes theoretically improves with smaller residuals. By combining two compression algorithms, the aim is to reduce these residuals. Initially, Adjacent is employed, where the previous microphone's value is used to predict the next microphone's value, and the difference becomes the residual. All the residuals obtained from using Adjacent are then grouped by microphones. Subsequently, Shorten is utilized to predict the next residual from the previous residuals, and the difference becomes the residual for Shorten. Finally, the Shorten residual is encoded using Rice codes.

5.8.1 Time to recreate values from Codewords

The time taken to recreate the original inputs when using double compression has been evaluated for all three sound files, and the result can be seen in the figure below:

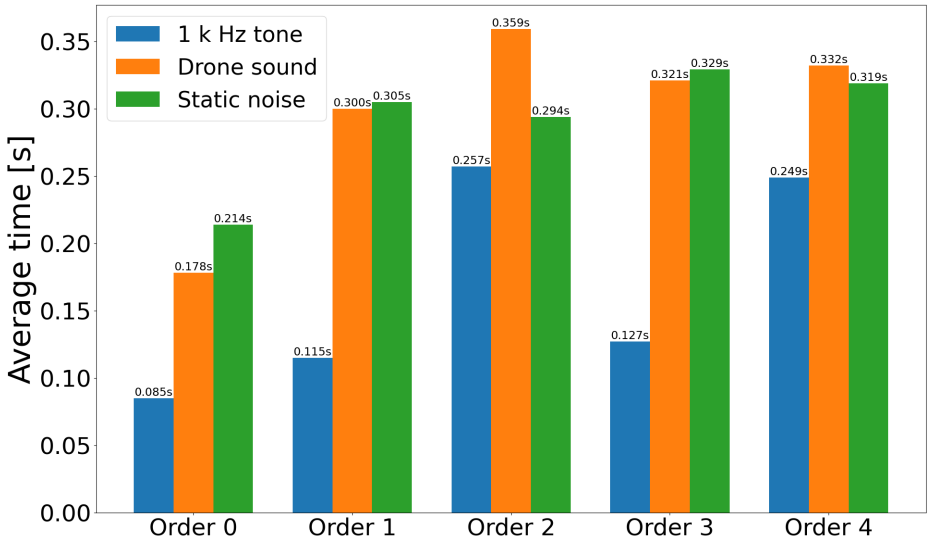


Figure 29: Average compression rate using Double Compression

5.8.2 Compression rate

The average compression rate for double compression can be seen in the figure below:

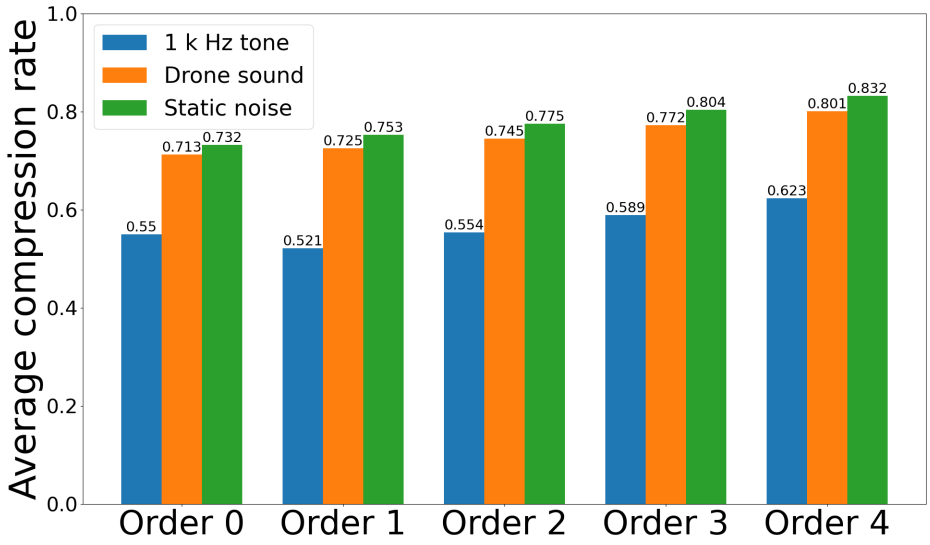


Figure 30: Average compression rate using Double Compression

In Figure 30, it can be observed that all orders performed similarly. Orders 0 and 1 exhibited slightly better performance than the rest, with the compression rate deteriorating as the order increased from 1 to 4. Additionally, it is evident that the algorithm performed best on the 1 kHz tone audio for all orders, while the other two sound files showed similar performance across all orders, with the drone sound being slightly better.

The compression rate in Figure 30 takes the metadata needed to decode the codeword into account. The only metadata required when using double compression is the k-value used for the Rice codes. This is represented by 5 bits, allowing k-values in the range of 1-33.

5.9 Adjacent Testbench Implementation

Connecting the entities described in Section 3.3 resulted in the following block diagram for the testbench:

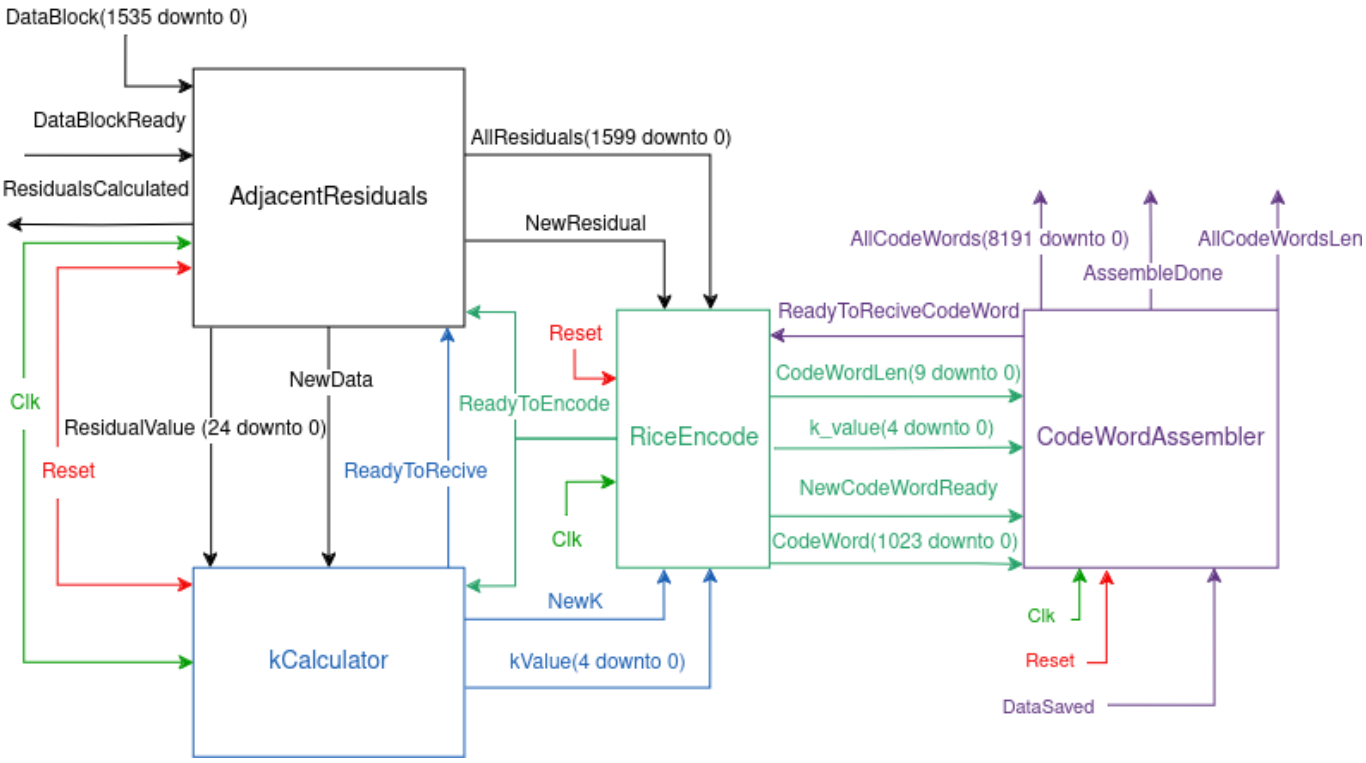


Figure 31: Testbench block diagram

In Figure 31, it can be seen how the entities communicate with each other, as well as the signals sent to receive and send data.

5.9.1 Timing

The testbench displayed in Figure 31 was run over 256 samples from 64 microphones to test various timing aspects of the VHDL code. The testbench uses a 125 MHz clock to mimic the clock used in the *Acoustic Warfare* system. The encoding time for the entire datablock of all 64 microphones is presented in the figure below:

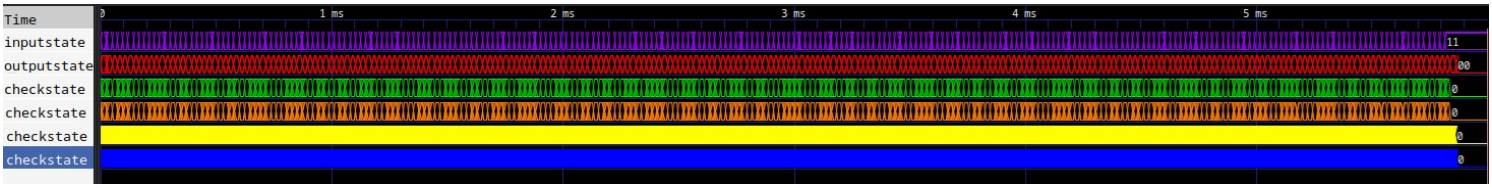
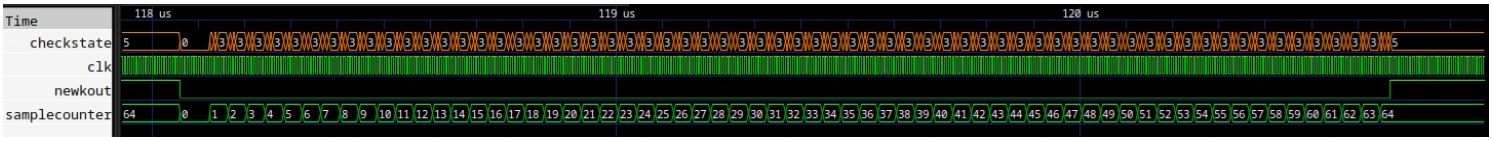


Figure 32: Testbench full run time

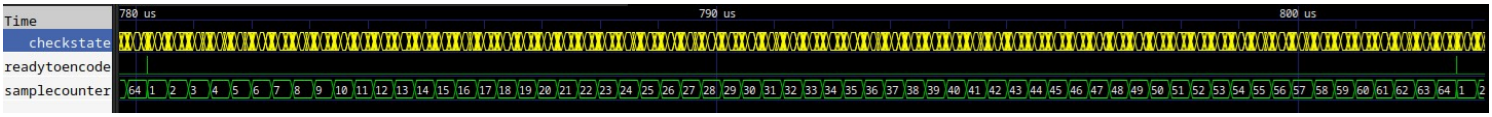
In Figure 32 input and output state is different states for reading and writing data in the testbench. checkstate rows in Figure 32 indicates what states the entities are in, green for AdjacenstResidual, orange for kCalculator, yellow for RiceEncode, and blue for CodeWordAseembler. Once *inputstate* reaches "11", all data have been read by the testbench, and when *outputstate* reaches state "00", all codewords have been written by the testbench. In Figure 32, it can be seen that the encoding time for the entire datablock is about 5858.712 μs.

Each entity was also individually timed to determine the duration from the receiving state to the sending state for each entity in the testbench.

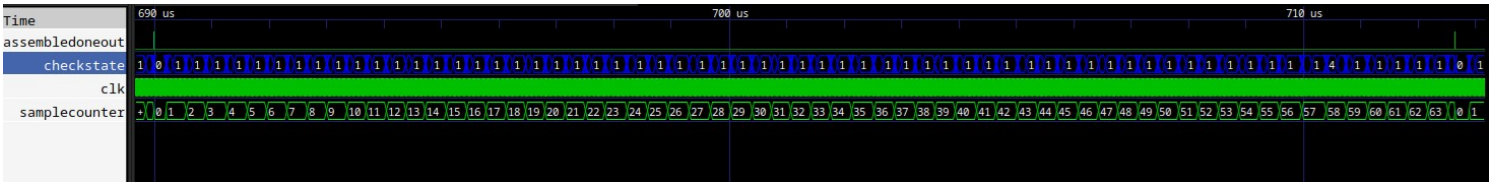
(a) AdjacentResidual



(b) kCalculator



(c) RiceEncode



(d) CodeWordAssembler

Figure 33: Time test for the state machines in each entity

In Figure 33, the time for each entity to go through all states in the state machine

can be observed, corresponding to handling a sample from the datablock for all 64 microphones. In Figure 33a, the signal newresidualout goes low when the AdjacentResidual entity starts calculating the residual for the first microphone and goes high when the residual for the last microphone has been calculated, which takes 2.58 μs. Figure 33b displays the time for the kCalculator entity to calculate a k-value to encode the residuals. The state machine is in the idle state (state 0) for some clock cycles before receiving the signal to start calculating. The time to calculate a k value is 2.552 μs. In Figure 33c, the time for the RiceEncode entity to encode all 64 microphones can be observed. The time it takes to encode all values can be seen from when readytoencode goes low to when it goes high again, taking 22.536 μs. The time it takes for the entity CodeWordAssembler to write the full codeword for all microphones and metadata can be observed in Figure 33d. It takes 22.464 μs to assemble all codewords with their metadata.

The full system is limited by the speed of the slower entities, which can be seen in the figure below:

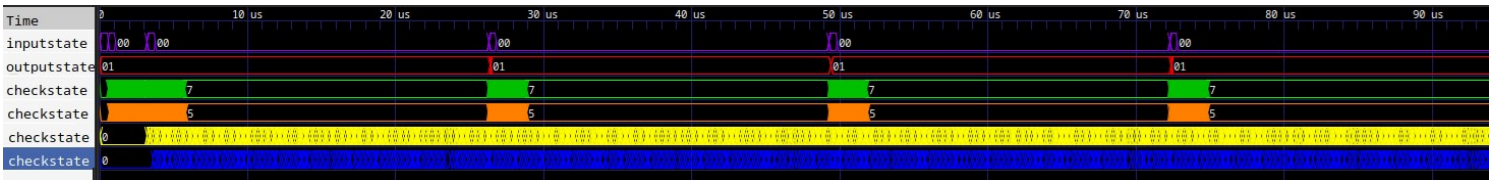


Figure 34: Zoomed in section of testbench GTKwave plot

In Figure 34, a zoomed-in section of the signals plotted in Figure 32 can be observed. Here, it is clear that the faster entities, AdjacentResidual and kCalculator, need to wait in their last state in the state machine to send the data before the next entity, RiceEncode, is ready to receive it.

5.9.2 Validation

To validate the codewords, they were loaded in and decoded in a Python script. The decoded values were compared to the original values and were found to be identical. Additionally, each microphone's original and decoded values were plotted to visually validate the results. The plot for microphone 79 can be seen in the figure below:

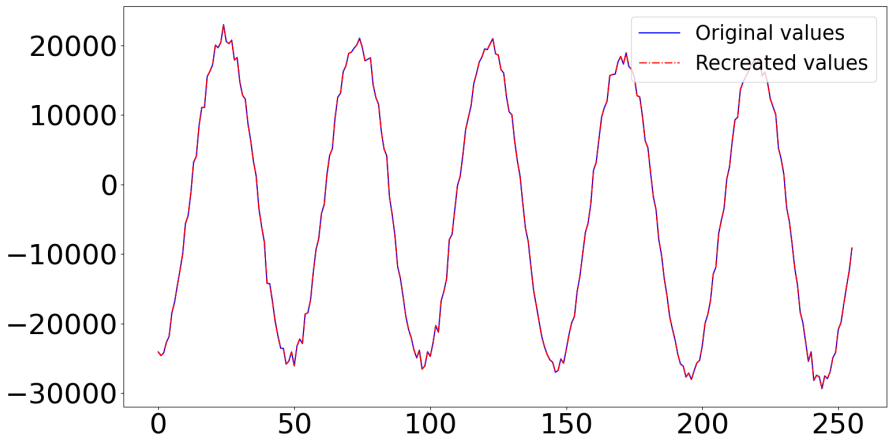


Figure 35: Plotting original and decoded values from microphone 79

In Figure 35, it can be observed that the original value and recreated value completely overlap, indicating that all values have been encoded and then decoded correctly. The resulting compression rate is at 0.577 for the data sent by all microphones in the datablock.

6 Analysis and discussion

6.1 Acoustic Warfare System

The *Acoustic Warfare* System has several faults that have affected this project. As can be seen in Figure 3, there are several microphones that do not work correctly. This affects the transmitted data. Modeling the signal becomes less effective when sudden spikes occur, such as those seen for microphone 233 in Figure 4a and 217 in Figure 4c, as it disrupts the sample-to-sample correlation. Microphones like 136 in Figure 4a, which do not capture the sound wave correctly, can also affect the sample-to-sample correlation. Additionally, there is a significant issue with muted microphones that do not capture any sound at all. Although this is not an issue when modeling the signal sample-to-sample, it becomes problematic when using the *Adjacent algorithm*, which relies on the correlation between microphones. Fortunately, microphone array 2, seen in Figure 3b, had mostly functional microphones. By limiting the tests to this array, data quality was assured while still utilizing a significant number of microphones during the tests.

Another factor that affected the results is the lack of pre-filtration. In the current system, no filtering is applied, meaning all recorded data is being sent. This includes frequencies above the Nyquist criterion that can distort the signal and data that may not be relevant for the end use. By applying a filter before encoding the data, it is possible to discard irrelevant data and avoid distortion. This could improve the modeling of the signal and reduce the amount of data that needs to be sent, thereby increasing the compression rate.

6.2 Coding choice

When analysing the results in figures 8, 17, and 24 it can be seen that the compression rate using both Rice and Golomb codes are close to identical in all cases. The same holds true when looking at the appendix figures 49, 50, 51, 73, 74, 75, and 76.

However, as mentioned in Section 4.2.3, the advantage of using Rice codes is the computing speed, especially when decoding. This is reflected in the results when comparing Shorten and LPC in combination with Golomb or Rice codes, as seen in Figures 9 and 18. For Shorten and LPC, Rice codes consistently outperform Golomb codes in decoding speed, especially for Drone sounds and Static noise. From this, it seems that for shorter codewords, they perform similarly, but as the codeword length grows, Rice codes start to outperform Golomb codes more and more in decoding speed. However, when looking at the time comparison using Adjacent modeling with either Golomb or Rice codes in Figure 25, the theory does not hold true. In this case, Golomb codes slightly outperform Rice codes, though they are very close in performance in all the test cases.

With the performance between Golomb and Rice codes being quite similar in terms of both compression rate and speed when used in combination with the Adjacent algorithm for modeling, the determining factor ultimately was which one was more suitable for implementation in an FPGA environment. Golomb codes involve several calculations, as shown in Equations 5, 6, and 7, as well as calculations for the length of the codeword. Rice codes need none of these calculations; instead, they follow the three steps outlined in Section 4.2.3. This makes it easier to implement Rice codes in the VHDL environment. Additionally, it results in faster-running code due to requiring fewer steps. Because of this, Rice codes were chosen over Golomb codes in the VHDL implementation

6.3 Modelling choice

When choosing modeling techniques, three aspects were considered: compression rate, whether the model loses any information, and how easy the model is to implement in VHDL.

Comparing the compression rates achieved by the different models in Figures 10, 19, 21, 26, 28, and 30, it is evident that the compression rates are quite close to each other. All models could achieve a compression rate around 0.55 for the 1 kHz tone, 0.75 for the drone sound, and slightly below 0.8 for the static noise. The best performance for each sound file was achieved by Double compression order 1 for the 1 kHz tone at a compression rate of 0.521, and FLAC modified for the other two sound files with a compression rate of 0.709 for the drone sound and 0.73 for the static noise, as seen in Figures 30 and 28. The worst performance was achieved by different orders of Shorten for all sound files. Shorten order 0 had a compression rate of 0.671 for the 1 kHz tone, and order 3 had compression rates of 0.808 for the drone sound and 0.842 for the static noise, as seen in Figure 10. However, it should be noted that by choosing the correct order of Shorten, it performs at similar levels to the other algorithms. Shorten order 1, for example, produces compression rates of 0.557 for the 1 kHz tone, 0.767 for the drone sound, and 0.795 for the static noise

When it comes to performing lossless compression, the only model that failed was LPC, and as an extension of this, FLAC also failed since it implements LPC. Since the coefficients produced by calculating the autocorrelation are practically always decimal numbers, the resulting residual will also be a decimal number. To encode this with Golomb or Rice codes, it has to be rounded off to an integer, which leads to loss of information. Another area where information is lost is the coefficients themselves. The coefficients need to be sent as metadata, and if all the decimals are to be included, the metadata will increase in size, negatively affecting the compression rates. Looking at the compression results for LPC in Figure 19, it can be seen that with 20 bits per coefficient, the compression rates are still in the same range as

the other models. However, when examining Figure 15 and Figures 52 to 72 in the Appendix, it can be seen that the missing data is small in comparison to the values sent, but it is still not completely lossless. The result is contradictory to what is stated in theory. In Section 4.3.3, it is stated that FLAC is a lossless compression algorithm; in fact, it says so in its name. This has the potential to be true when RLE or Shorten is utilized in the FLAC algorithm. However, since FLAC also includes LPC with Rice codes, the tests performed in this project show that losses could occur when implementing FLAC.

In the end, the most determining factor for modeling was how well the different models would fit into the *Acoustic Warfare* system. The recording device capturing the signal utilizes antenna arrays, in this case, several microphones. This makes adjacent a very fitting choice since it utilizes this aspect of the system. It is also quite easy to implement since the residual calculation only consists of a subtraction of the microphone value by an adjacent microphone value. Another advantage when it comes to the VHDL implementation is that it sends one set of codewords per sample. Meaning at every sample recorded by the system, it can be sent through the full block diagram seen in Figure 31. The other algorithms that have been tested all send one datablock per microphone as an assembly of codewords, meaning 256 samples have to be taken of the system and then the codewords from all microphones are assembled and ready to be sent at the same time. This could be parallelized in the FPGA for the different microphones so that it would not affect the speed of encoding the values, but it would still result in spikes and valleys of sending data.

Shorten, FLAC modified, and Double compression were also models that were up for consideration when it comes to implementation on the FPGA. Shorten calculates the residual by adding or subtracting multiples of previous samples, depending on what order is used following equations 13, 14, 15, and 16. These are all simple operations to implement on an FPGA. FLAC modified and Double compression both utilize Adjacent and Shorten. FLAC modified also utilizes RLE, which counts repeats of samples. This could also be easily implemented on the FPGA since it only requires incrementing a counter. FLAC modified could also take advantage of the fact that FPGAs are designed to run parallel processes. This could allow it to calculate the compression rate for all algorithms included in FLAC modified in parallel, speeding up the process of finding the best choice for the current datablock.

LPC was deemed to be the least fitting algorithm to implement on the FPGA, and as an extension, the FLAC algorithm was also ruled out since it utilizes LPC. Calculating MSE and the Levinson-Durbin algorithm both utilize division, which takes more time to perform and results in larger logic-blocks.

6.4 Soundfiles

The sound data being recorded has a significant effect on the resulting compression rate. Around 20-25% more compression is achieved for the 1 kHz tone compared to the other two sound files when comparing to the original data size. However, when applying Rice codes to the raw data samples, the performance is more similar between all sound files, as can be seen in Figure 5. The compression rate for the 1 kHz tone when using Rice codes on the raw data is at 0.739, while the drone sound and static noise are at 0.809 and 0.808, respectively. The reason for Rice codes to work better on the 1 kHz tone when applying it to the raw data can be attributed to the fact that Rice codes work better on smaller values. Looking at the amplitude of the soundwaves in Figure 5, it can be seen that the 1 kHz tone has a lower amplitude than the other two sound files. The fact that Rice codes have similar performance on the raw data implies that the modeling helps more when it comes to the 1 kHz sound compared to the other two sound files. The 1 kHz tone follows a 1 kHz sinusoidal waveform, while the other two sound files have more randomly distributed data points, as can be seen in Figure 5. It is therefore logical that the models have a greater effect on the 1 kHz tone since it has a larger sample-to-sample correlation. This can also be seen to a smaller extent when comparing drone sound to static noise. The drone sound compression rate has a larger gain when applying the models than the static noise. Although not evident when looking at the time domain on the waveforms, the drone sound is predominantly composed of frequencies concentrated within specific spectrums, while the static noise is an attempt at creating completely random sound data. This shows in the compression rate results as the drone sound consistently gets more compressed than the static noise, as seen in Figures 10, 19, 21, 26, 28, and 30. However, while consistent, this difference is very small.

6.5 Testbench Performance

By rewriting the Python application in a VHDL environment, it was possible to get an idea of how the Adjacent algorithm would perform in the *Acoustic Warfare* system. The codewords created when testing the entities with the testbench were then decoded with a Python script, and the recreated values matched the original values as displayed in Figure 35.

When analyzing the timing aspect of the VHDL code, it shows that it takes circa 5.86 ms to send 256 samples, giving it a sample rate of approximately 43.70 kHz. This means the current VHDL code is a bit slower than the microphone's sampling speed, which would lead to a pile-up of samples being sent to the encoder. However, implementing more than one array would not affect this problem since it is possible to run several Adjacent algorithms in parallel for every array that is plugged in.

6.6 Future Work

There are three main areas to focus on for future implementations: improving compression rate, optimizing VHDL code, and integration with the full system.

My primary recommendation for improving compression rate is to enhance the modeling aspect of the compression algorithm by refining the signal of interest. As discussed in Section 6.4, the compression rate improves by 20-25% when the sound waves follow a clear pattern and are easier to model. Applying prefiltration to isolate the soundwave of interest could facilitate better modeling. Alongside this project, another master's thesis has been written that filters the data recorded with the *Acoustic Warfare* system on the computer side to isolate drone sound[12]. The next step in improving the compression rate could involve using the filtered data from this master's thesis project and compressing it using the Python codes presented in this project to assess how the filter affects the compression rates. If the resulting compression rate on filtered data is promising, the next step would be to implement a filter on the FPGA board. The filter type implemented on the computer side is an FIR filter. I recommend exploring adaptive FIR filter implementations for FPGAs. By having an adaptive filter, the system is not limited to listening to drone sounds.

As mentioned in Section 6.5, the time performance of the VHDL code needs to be improved to match the sampling speed of the microphones. RiceEncoder and CodeWordAssembler are the entities that cause the largest delay in the current system, as can be seen in Figures 33 and 34. Optimization should be explored for these two entities. It might also be possible to combine these two entities into one with minor changes, which could potentially lead to some time savings in the system. Another possibility would be to divide each array into two or more sections and allow several Adjacent algorithms to run in parallel with different starting microphones. This would reduce the number of samples to be handled by each entity and thus speed up the encoding process.

In Section 6.3, other algorithms that would be suitable to implement in VHDL code were discussed, including the Shorten algorithm. The Shorten algorithm could potentially be faster to implement because it is more conducive to performing calculations in parallel on the FPGA. The Shorten algorithm looks at one microphone at a time and tries to predict the next sample from earlier samples. It could therefore be set up so that the residuals can be calculated for all microphones in parallel. Exploring the implementation of the Shorten algorithm in VHDL code could therefore also be a valid option in the search for a more optimized compression algorithm.

However, it should be noted that AdjacentResidual is about 10 times faster than the entities RiceEncode and CodeWordAssembler. In order to achieve larger time savings, the subsequent steps also need to be able to work in parallel. This might

be possible with some minor adjustments, but I leave that subject to be explored in future work.

Finally, the compression algorithm needs to be integrated into the rest of the *Acoustic Warfare* system. This requires the VHDL entities from this project to be implemented for encoding on the FPGA side and the Python functions to be implemented on the computer side for decoding. All data that needs to be encoded per package is gathered in the *full_sample* entity in the current system. From here, the VHDL code from this project could encode the data.

The next step would be to investigate any adaptations needed for the AXI protocol and UDP transmitter to accommodate the fact that the new data packages will vary in length. On the computer side, decoding should be implemented as the first operation after receiving the data via the UDP receiver.

7 Conclusion

In Section 2.4, the aim of this project was stated: to reduce the average transfer rate for Saab's *Acoustic Warfare* system below 90 Mbps per array, preferably down to 45 Mbps per array. The current transfer rate is 100.78 Mbps per array, but this includes the zero padding bits at each sample, which make up circa 25% of the bits sent. By simply removing these, the average transfer rate is reduced to 75.585 Mbps per array, achieving the first goal of going below 90 Mbps per array.

The compression algorithms explored in this project could further compress data between circa 20 to 50%, depending on the sound. The Adjacent algorithm, chosen for implementation, achieved a compression rate ranging from 0.583 at best to 0.771 at worst, depending on the sound being compressed. At the low end, this would further reduce the transfer rate by 22.9%, to circa 58.276 Mbps per array. At the high end, it would reduce it by 41.7%, bringing the transfer rate below the 45 Mbps threshold, to circa 44.066 Mbps per array.

In conclusion, this project has shown that it is feasible to implement a compression algorithm on the FPGA used in the *Acoustic Warfare* system. This has been accomplished by implementing the VHDL compression code and testing it in a testbench, followed by decompressing the codewords with Python code. The recommended compression algorithm to move forward with is the Adjacent algorithm, which easily achieves the goal of an average transfer rate below 90 Mbps per array. It can achieve compression rates that bring the average transfer rate below 45 Mbps, depending on the sound being compressed, all while being completely lossless.

The project leaves the implementation of the compression algorithm into the *Acoustic Warfare* system to future work. It also leaves several interesting subjects to be further explored, such as optimization of the algorithm and pre-filtering.

8 References

[1] *RECENT CONTRIBUTIONS TO THE MATHEMATICAL THEORY OF COMMUNICATION, Author: Warren Weaver, Published in: ETC: A Review of General Semantics, Vol. 10. No. 4, SPECIAL ISSUE ON INFOMRATION THEORY (SUMMER 1953), Published by: Institute of General Semantics, Year of Publication: 1953* Available at: `https://www.jstor.org/stable/42581364?seq=1&cid=pdf-reference#references_tab_contents`, Accessed: January 31, 2024.

[2] *Introduction to Data Compression, Author: Guy E. Blelloch, Date of Publication: 31 January 2013* Available at: `https://www.cs.cmu.edu/~15750/notes/compression_guy_blelloch_book_chapter_draft.pdf`, Accessed: February 1, 2024.

[3] *Acoutstic Warfare, Author: Marcus Allen, Tuva Björnberg, Isac Bruce, Ivar Nilsson, Mika Söderström, Jonas Teglund, Date of Publication: July 2023* Available at: `https://github.com/acoustic-warfare/Beamforming/blob/main/Ljudkriget.pdf`, Accessed: January 29, 2024.

[4] *A Mathematical Theory of Communication, Author: C. E. Shannon, Published in: The Bell System Technical Journal, Vol. 27, Published by: Nokia Bell Labs, Year of Publication: 1948* Available at: `https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf`, Accessed: January 31, 2024.

[5] *Information Security Science, Author: Carl S. Young, Published by: Elsecier Inc, Year of Publication: 2016* Available at: `https://www-sciencedirect-com.ezproxy.its.uu.se/book/9780128096437/information-security-science#book-description`, Accessed: February 1, 2024.

[6] *Handbook of Data Compression Fifth Edition, Author: David Salomon and Giovanni Motta with contributions by David Bryant, Published by: Springer London, Date of Publication: Januray 18 2010* Ebok available at: `https://link.springer.com/book/10.1007/978-1-84882-903-9`, Accessed: January 26, 2024.

[7] *Introduction to Data Compression, Author: Khalid Sayood, Published by: Elsevier Inc, Date of Publication: 2012* Ebok available at: `https://www.sciencedirect.com/book/9780124157965/introduction-to-data-compression#book-description`, Accessed: January 26, 2024.

[8] *Run-length encdoings, Author: S. Golomb, Published in: IEEE Transactions on Information Theory (Volume: 12, Issue: 3), Date of Publication:*

July 1996 Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1053907, Accessed: January 26, 2024.

[9] *SHORTEN: Simple lossless and near-lossless waveform compression, Author: Tony Robinson, Published by: Cambridge University, Date of Publication: December 1994* Available at: https://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/robinson_tr156.pdf, Accessed: January 26, 2024.

[10] *Introduction to Digital Speech Processing, Author: Lawrence R. Rabiner and Ronal W. Shafer, Published by: now Publishers Inc., Date of Publication: 2007* Available at: https://research.iaun.ac.ir/pd/mahmoodian/pdfs/UploadFile_2643.pdf, Accessed: January 30, 2024.

[11] *FLAC, Format, Author: Josh Coalson, Published by: Xiph Org Fundation, Date of Publication: December 1994* Available at: https://xiph.org/flac/format.html, Accessed: January 29, 2024.

[12] *Isolating Drone Frequencies In A Real-Time Drone Detection System, Author: Jonas Teglund, Master thesis at Uppsala University, Date of Publication: Yet to be published.*

A Appendix

A.1 Test results

A.1.1 Figures

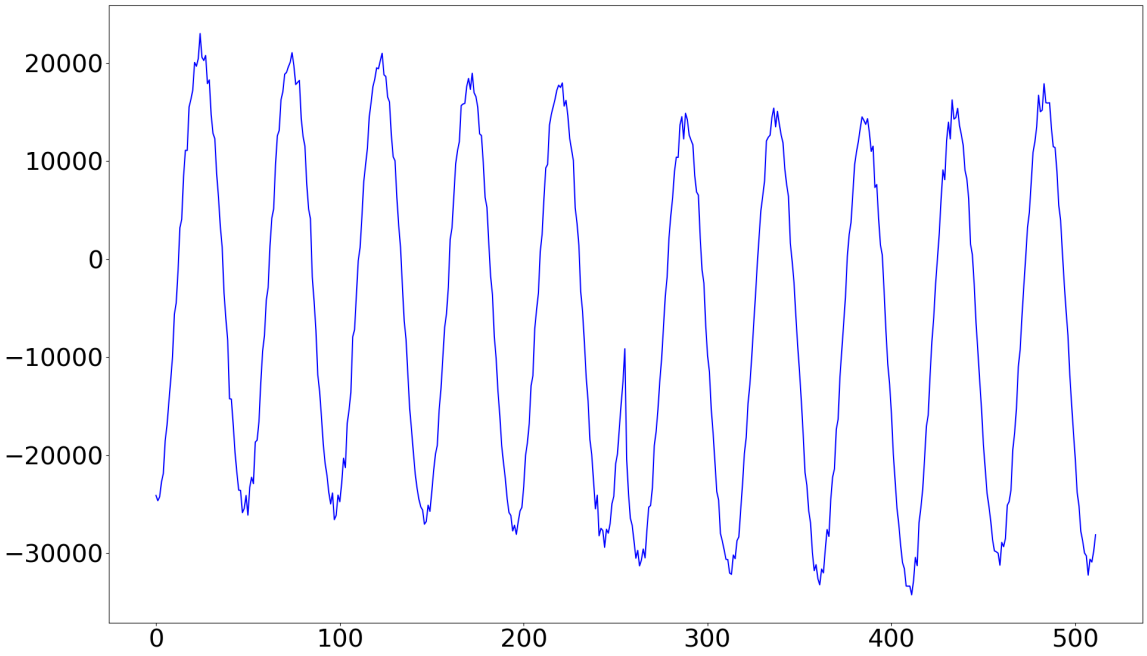


Figure 36: Plotting two datablocks at 256 samples each of 1 k Hz tone using mic 79

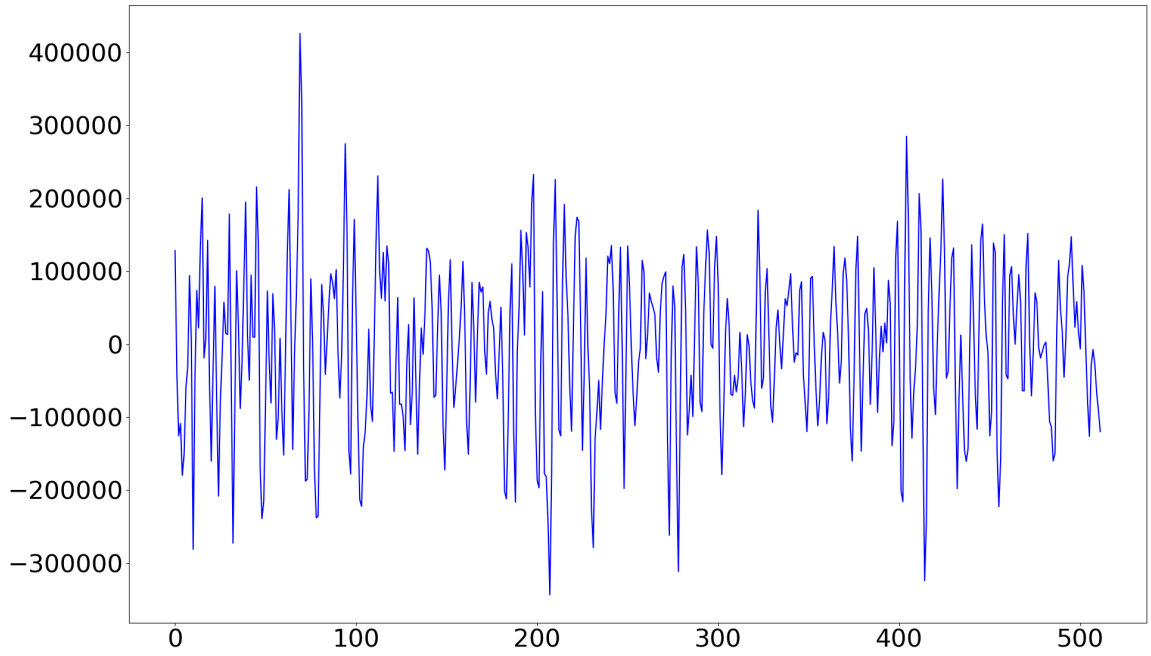


Figure 37: Plotting two datablocks at 256 samples each of drone sound using mic 79

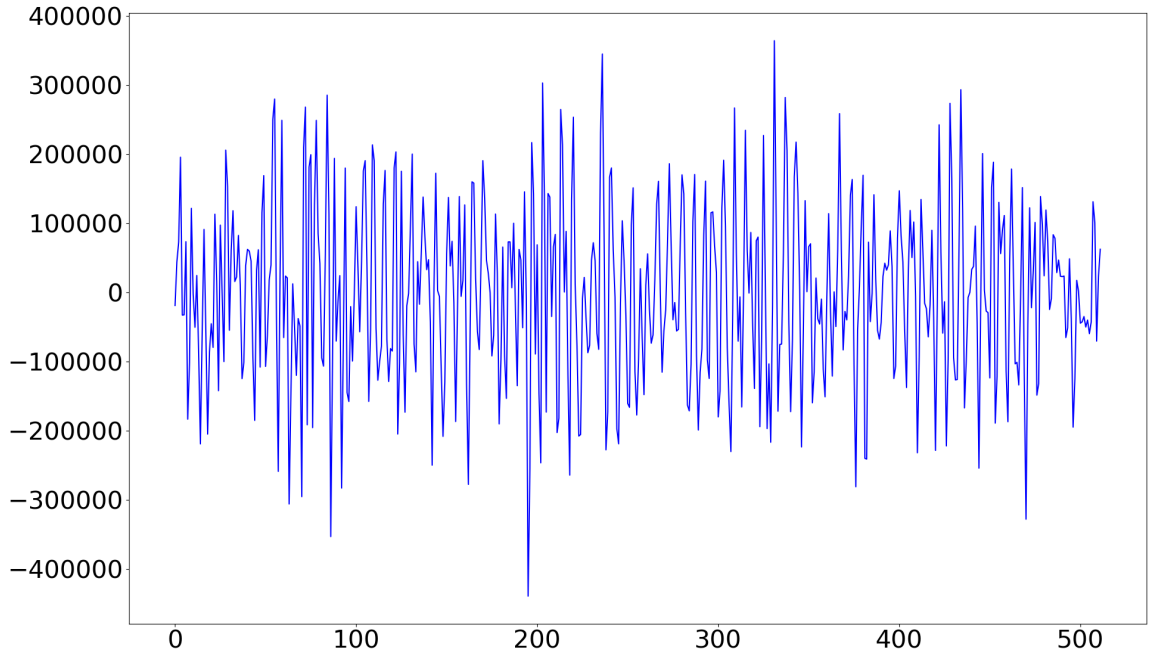


Figure 38: Plotting two datablocks at 256 samples each of static noise using mic 79

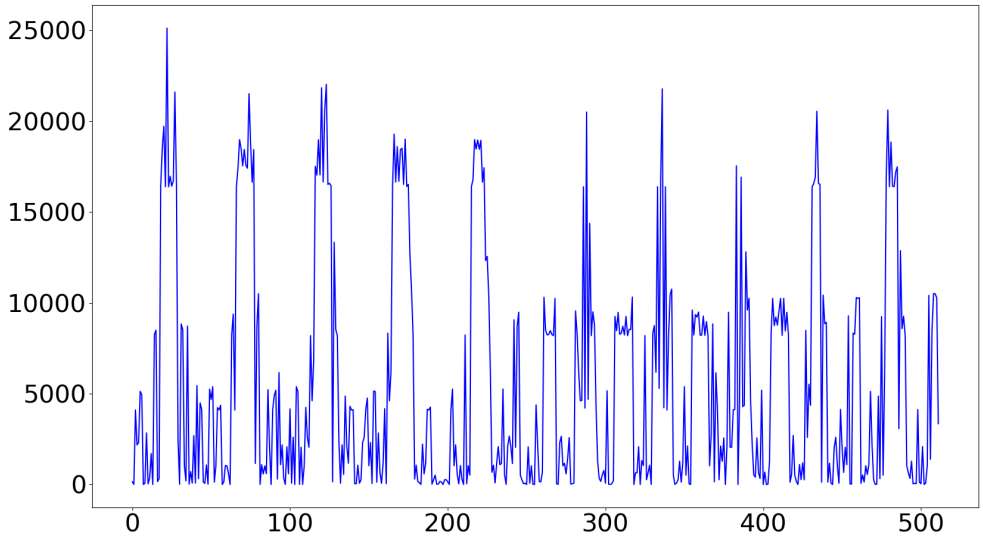


Figure 39: Plotting two datablocks at 256 samples each of 1 k Hz tone using mic 136

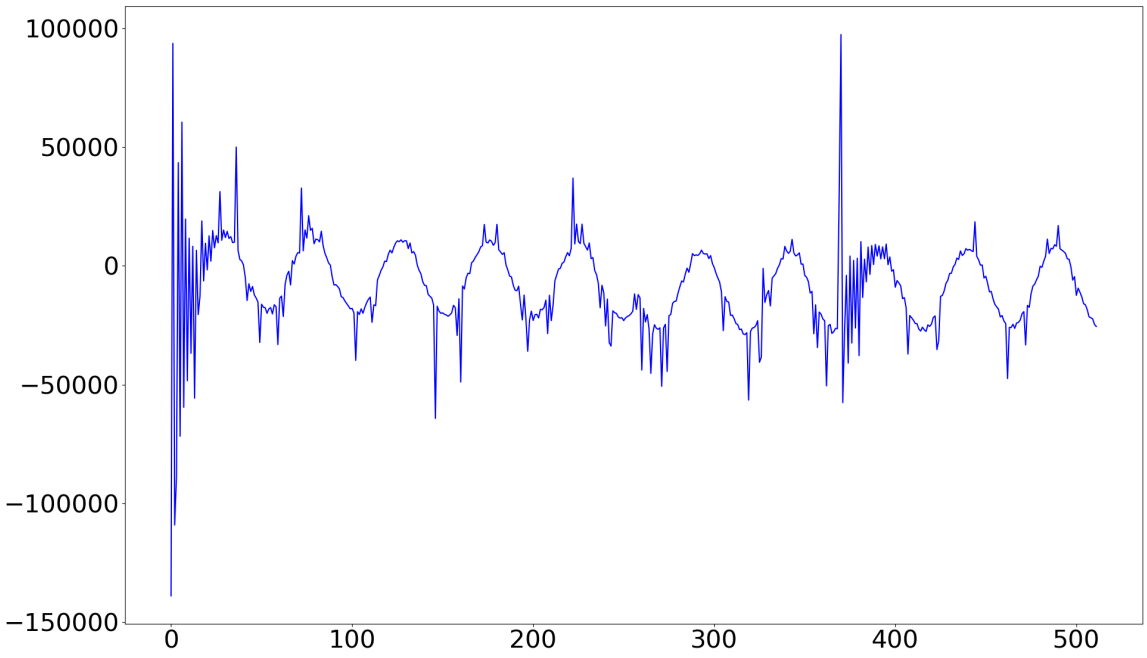


Figure 40: Plotting two datablocks at 256 samples each of 1 k Hz tone using mic 233

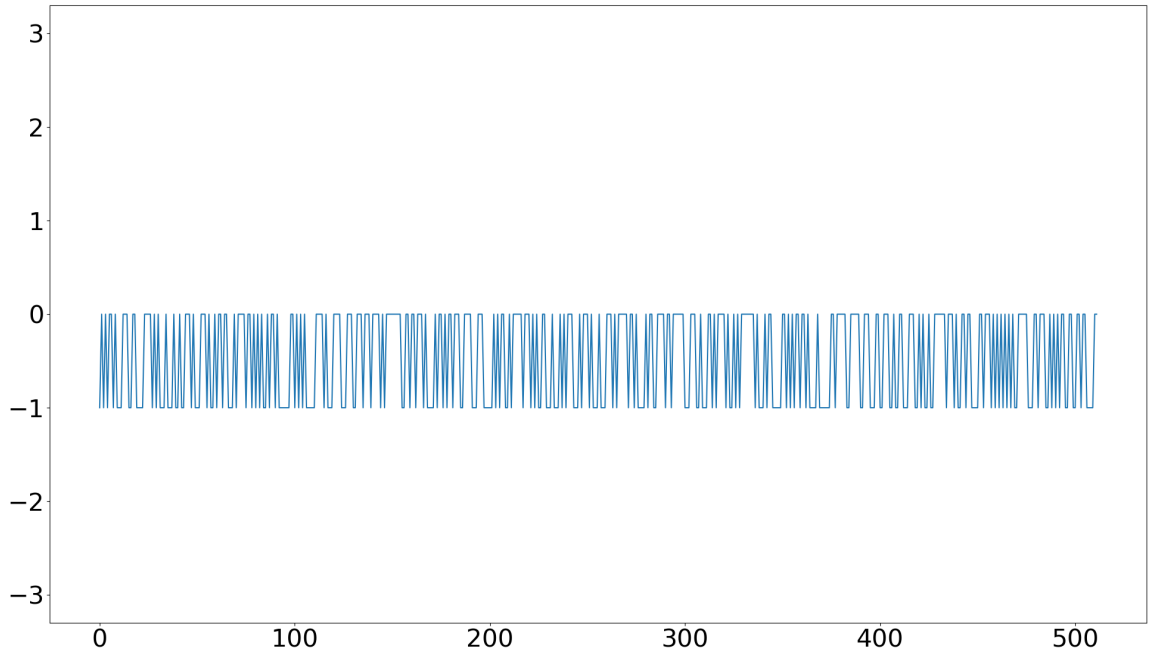


Figure 41: Plotting two datablocks at 256 samples each of 1 k Hz tone using mic 19

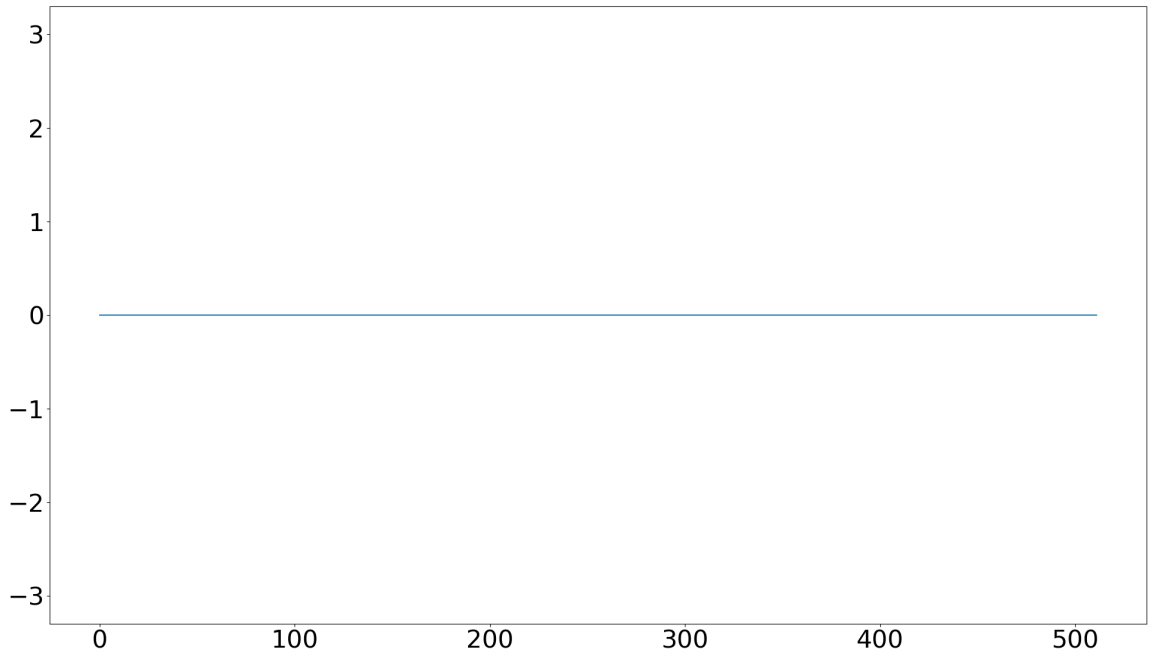


Figure 42: Plotting two datablocks at 256 samples each of 1 k Hz tone using mic 20

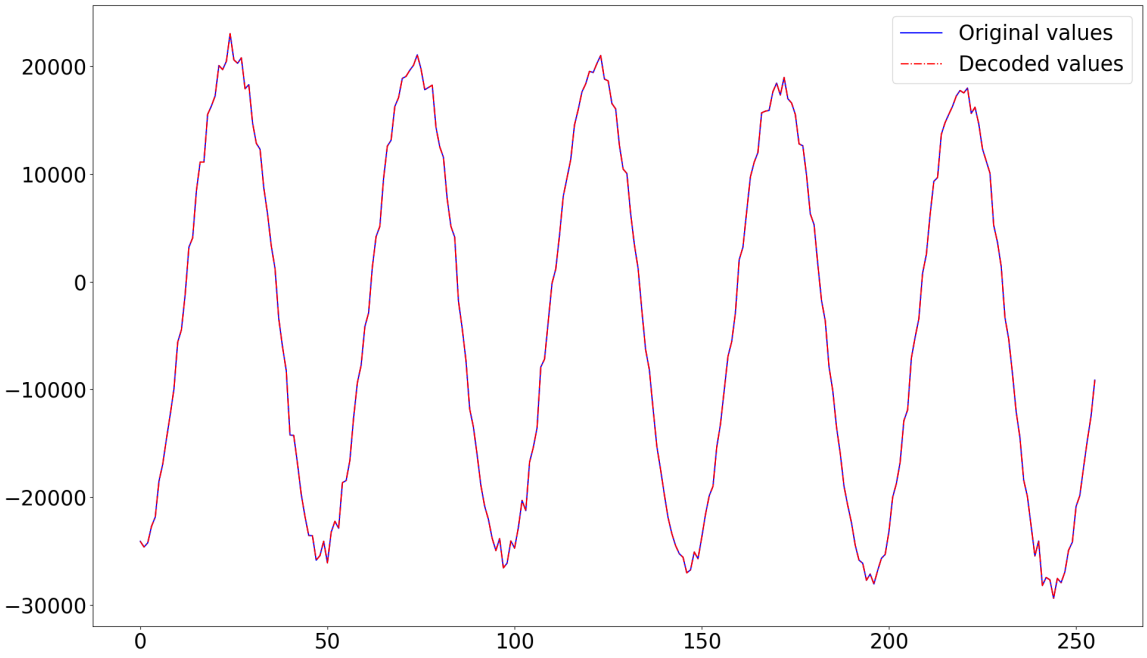


Figure 43: Original values and recreated values of 1 k Hz tone, using Shorten order 0 with Rice codes

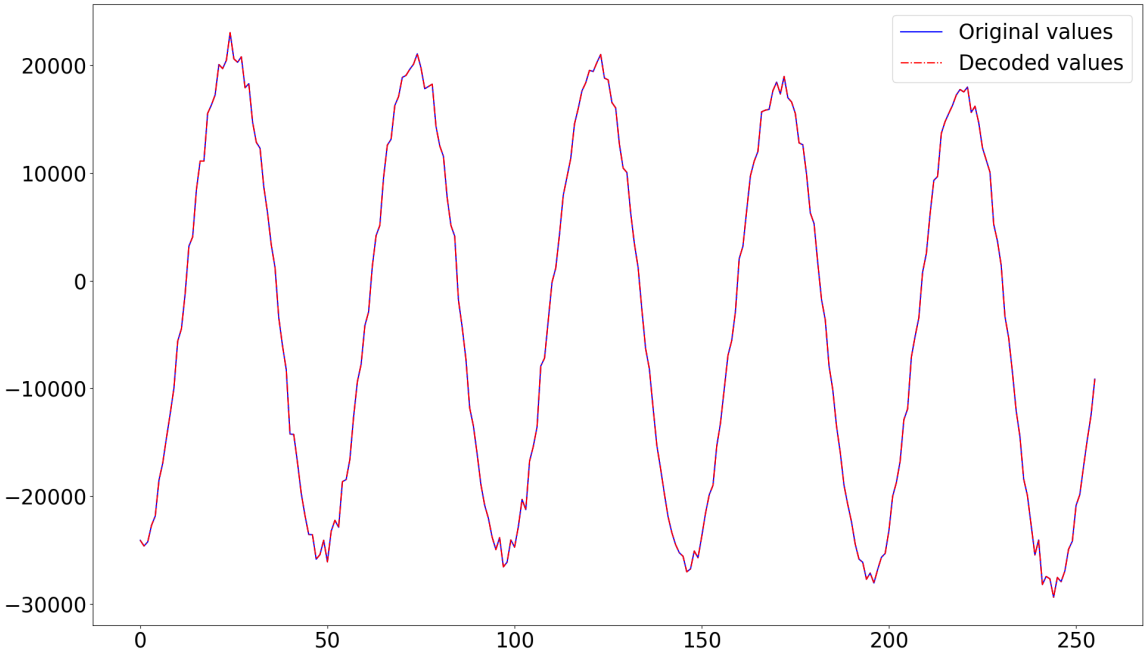


Figure 44: Original values and recreated values of 1 k Hz tone, using Shorten order 2 with Rice codes

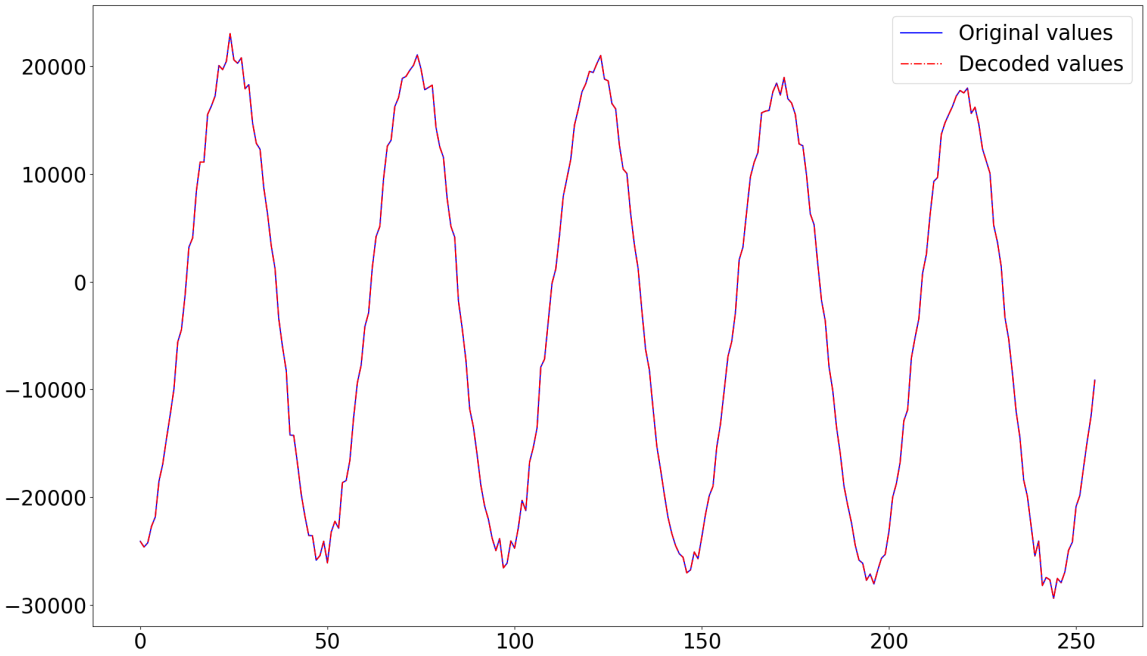


Figure 45: Original values and recreated values of 1 k Hz tone, using Shorten order 3 with Rice codes

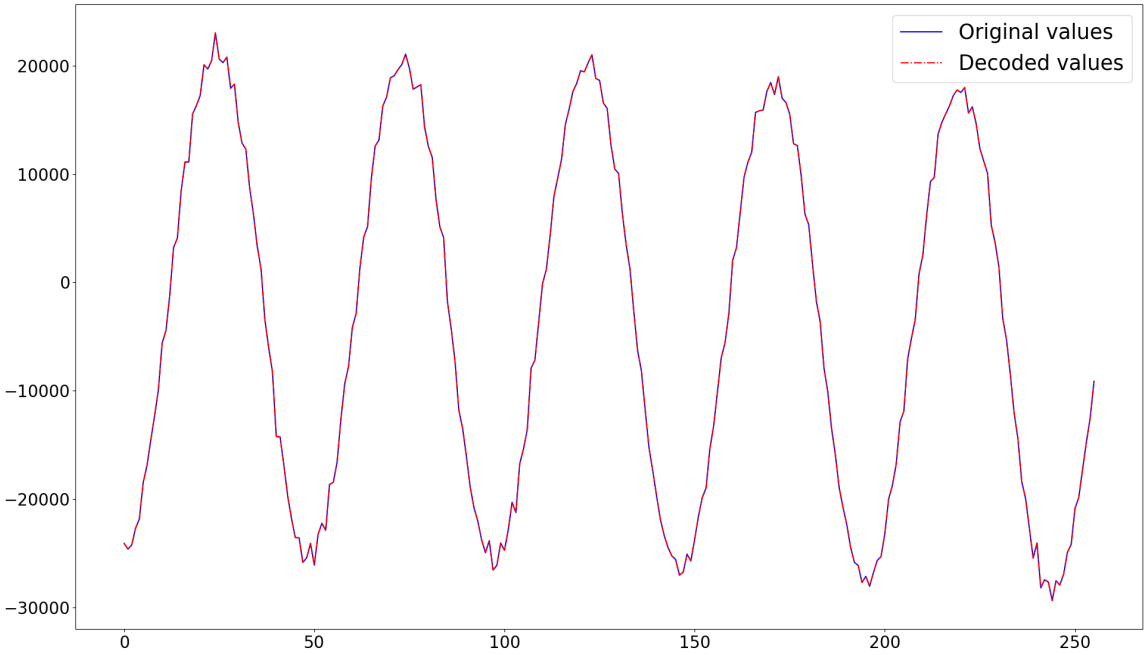


Figure 46: Original values and recreated values of 1 k Hz tone, using Shorten order 0 with Golomb codes

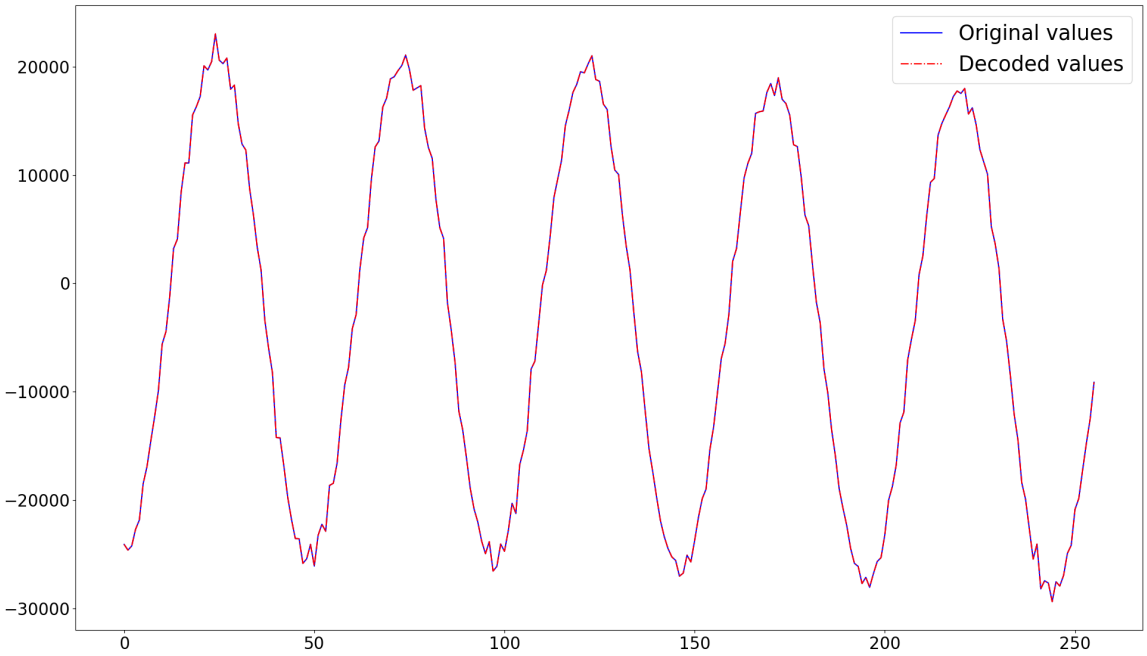


Figure 47: Original values and recreated values of 1 k Hz tone, using Shorten order 2 with Golomb codes

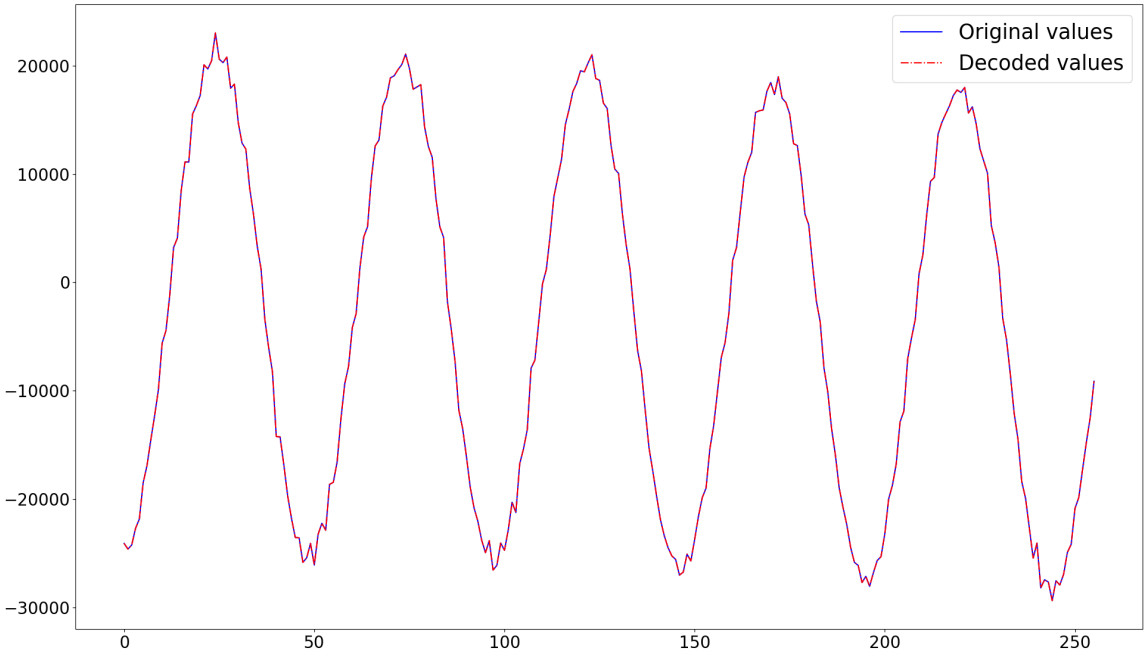


Figure 48: Original values and recreated values of 1 k Hz tone, using Shorten order 3 with Golomb codes

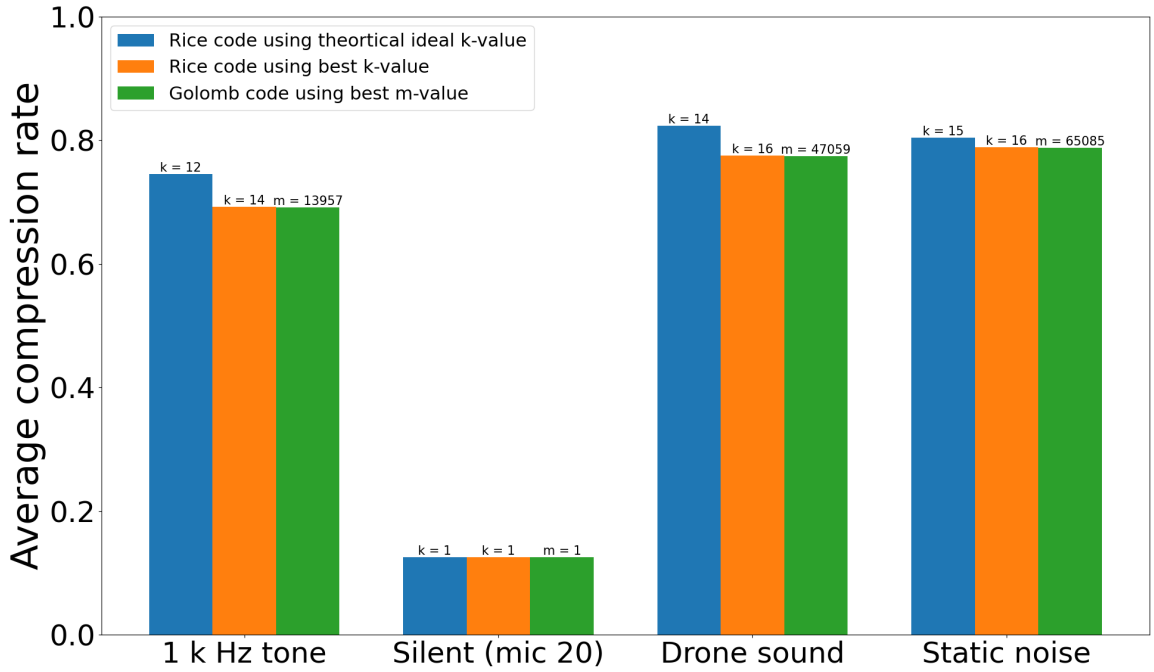


Figure 49: Average compression rate using Shorten order 0

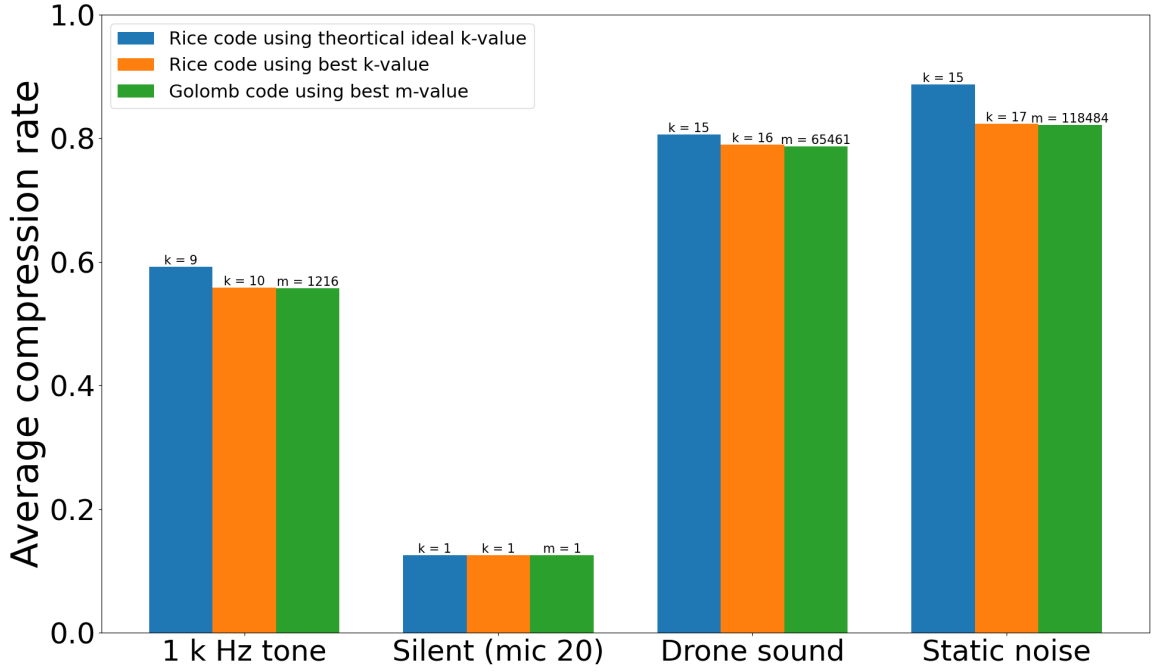


Figure 50: Average compression rate using Shorten order 2

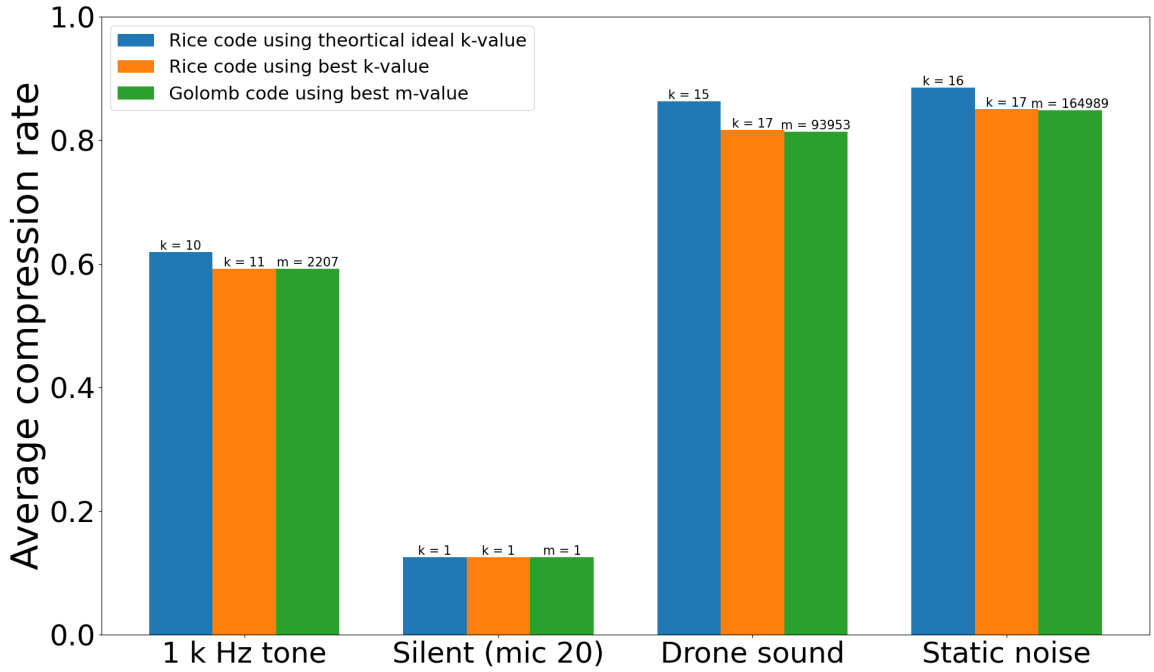
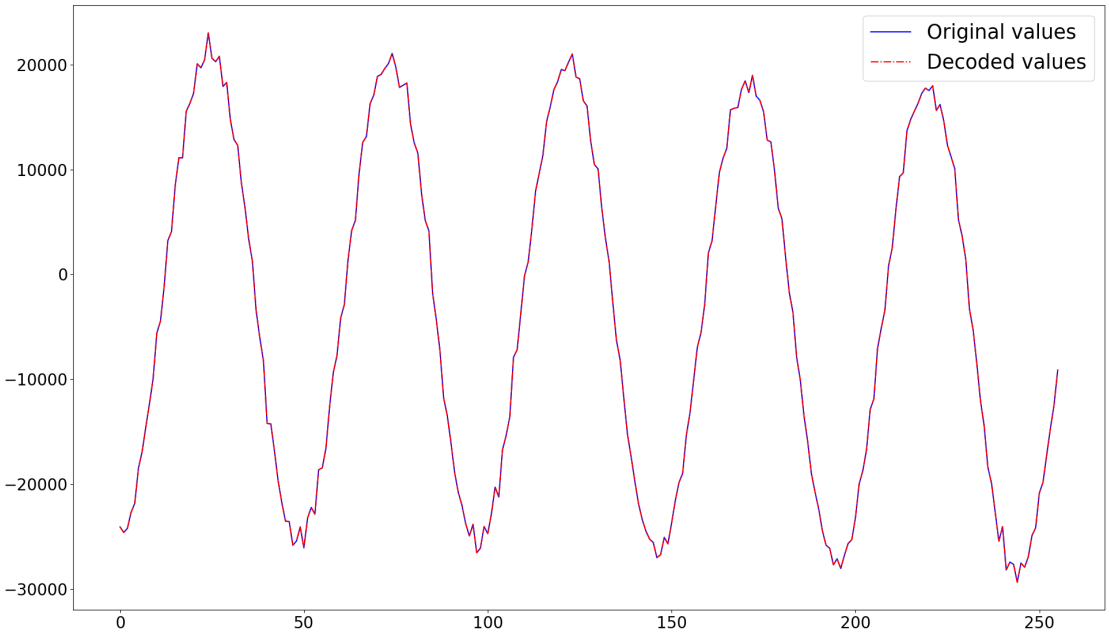
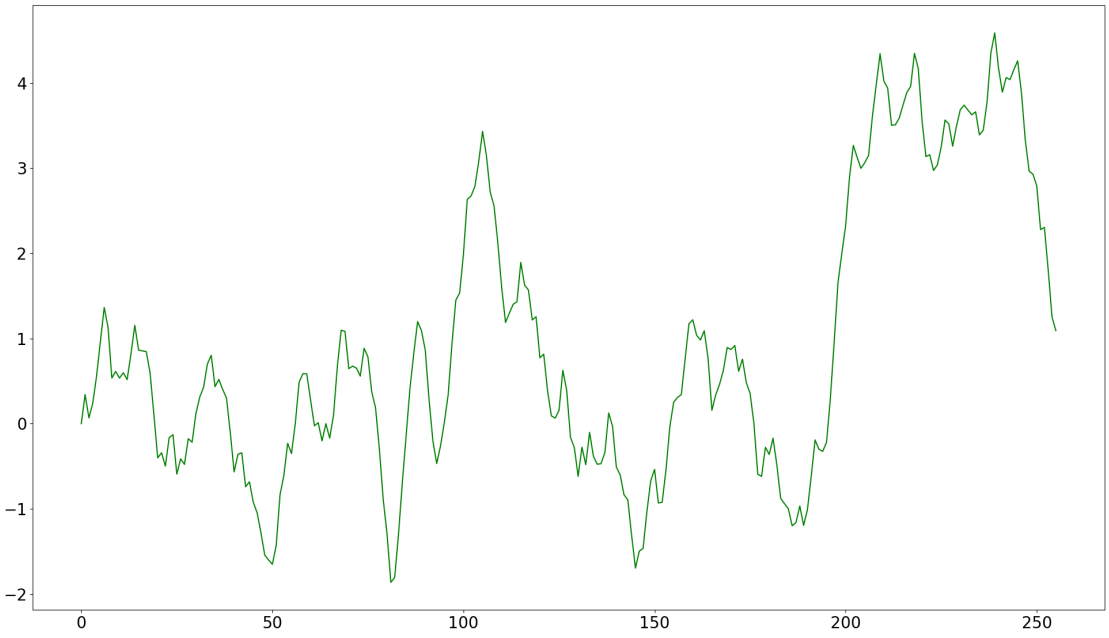


Figure 51: Average compression rate using Shorten order 3

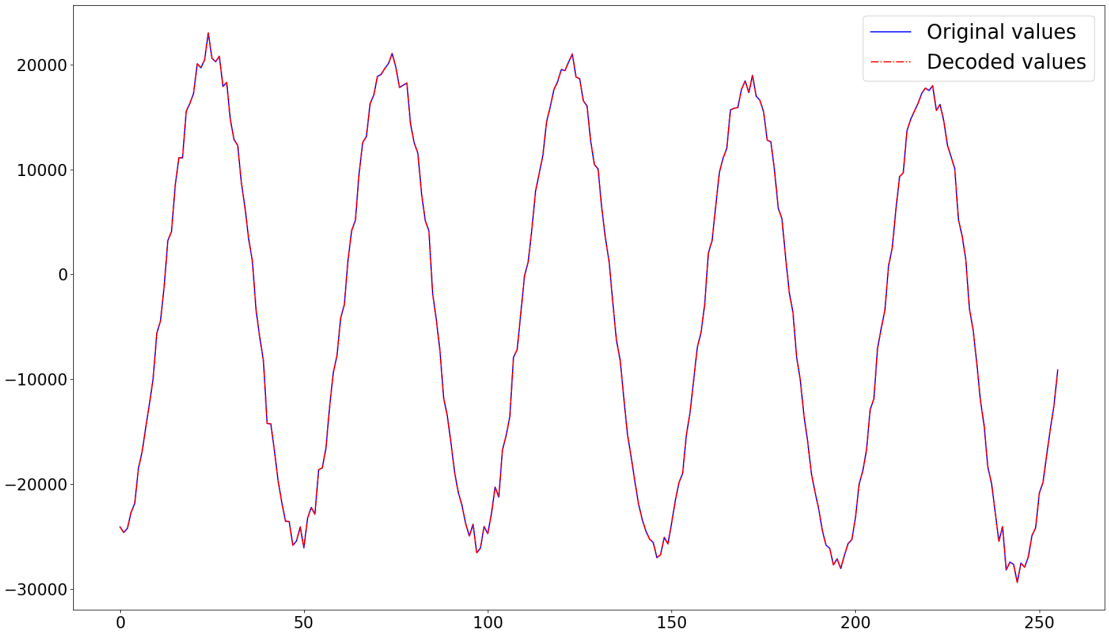


(a) Original and recreated signal

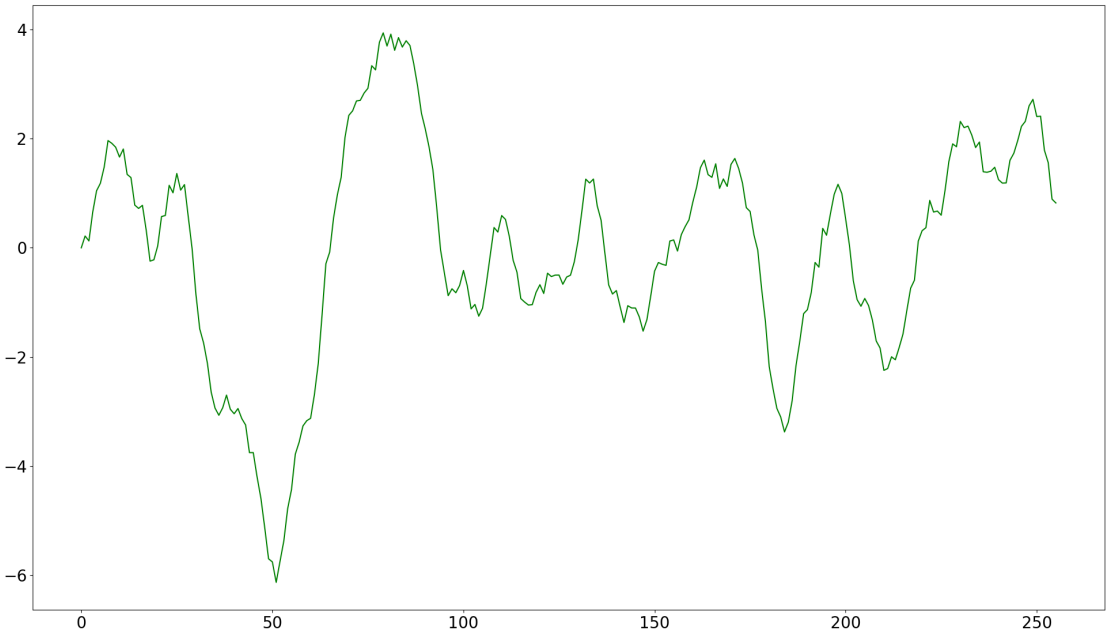


(b) Original signal - recreated signal

Figure 52: Recreate 1 k Hz tone using LPC order 2 and Rice codes, all coefficients recreated perfectly

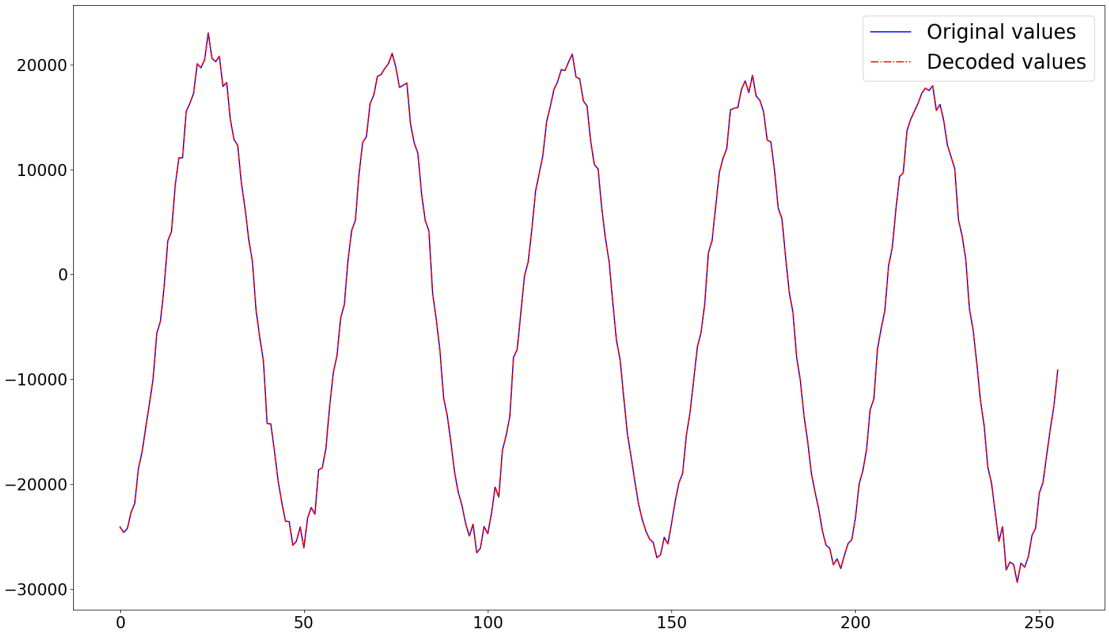


(a) Original and recreated signal

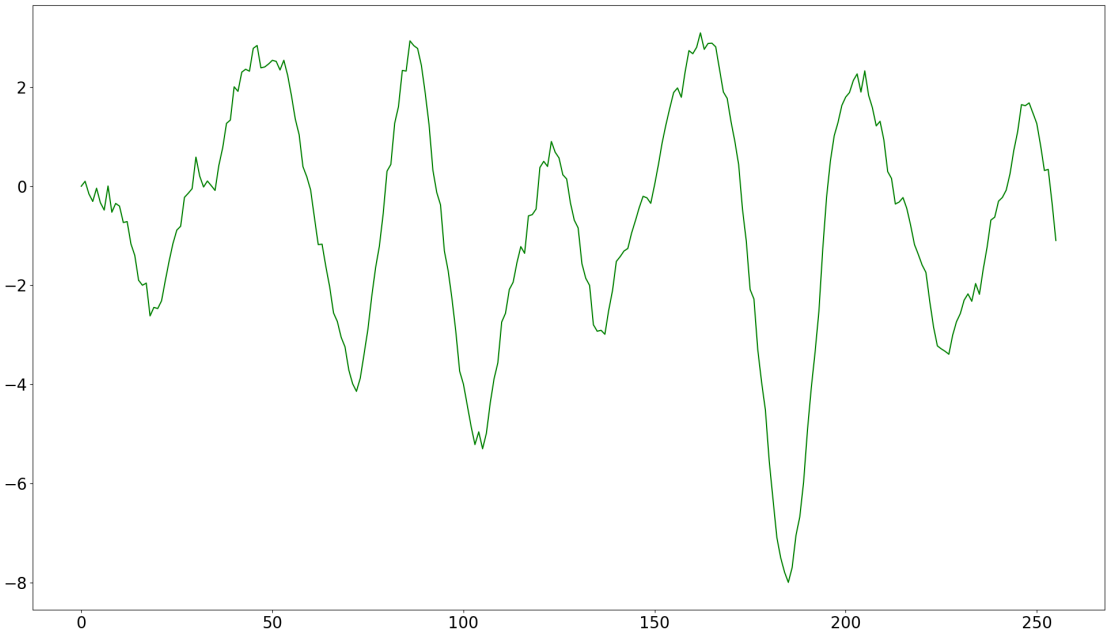


(b) Original signal - recreated signal

Figure 53: Recreate 1 k Hz tone using LPC order 3 and Rice codes, all coefficients recreated perfectly

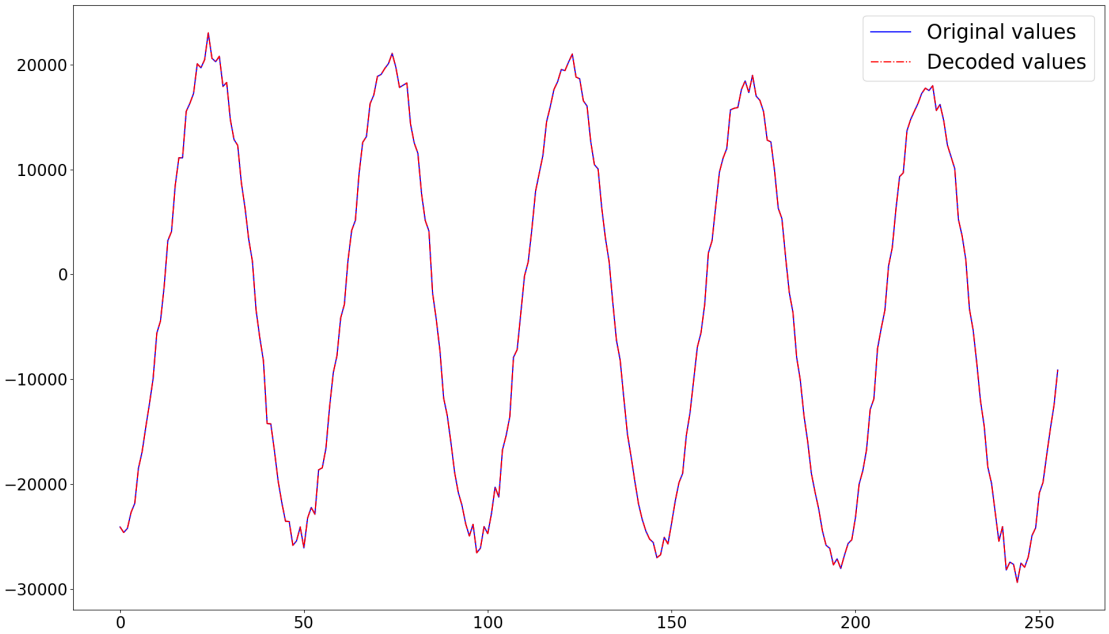


(a) Original and recreated signal

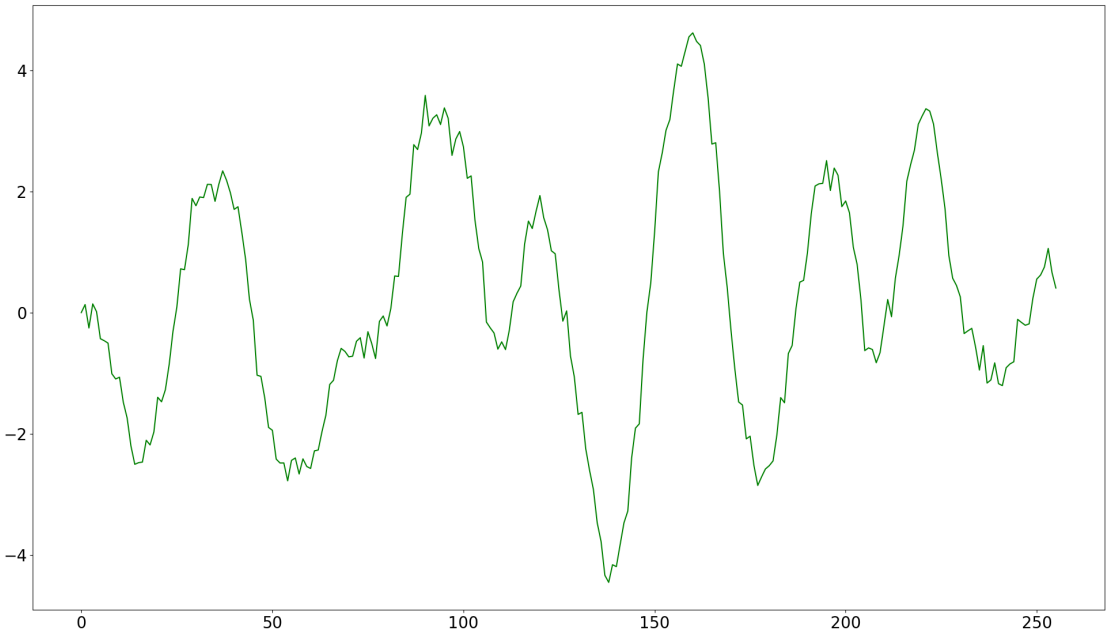


(b) Original signal - recreated signal

Figure 54: Recreate 1 k Hz tone using LPC order 4 and Rice codes, all coefficients recreated perfectly

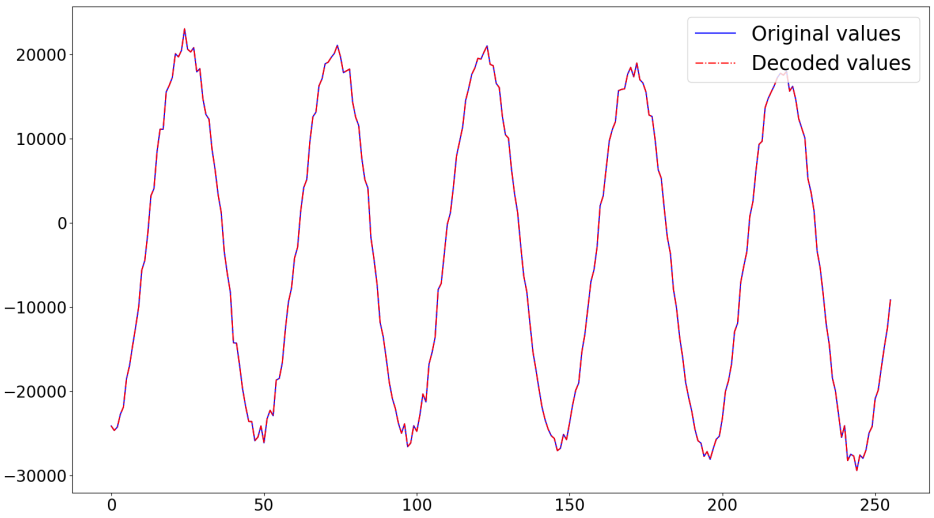


(a) Original and recreated signal

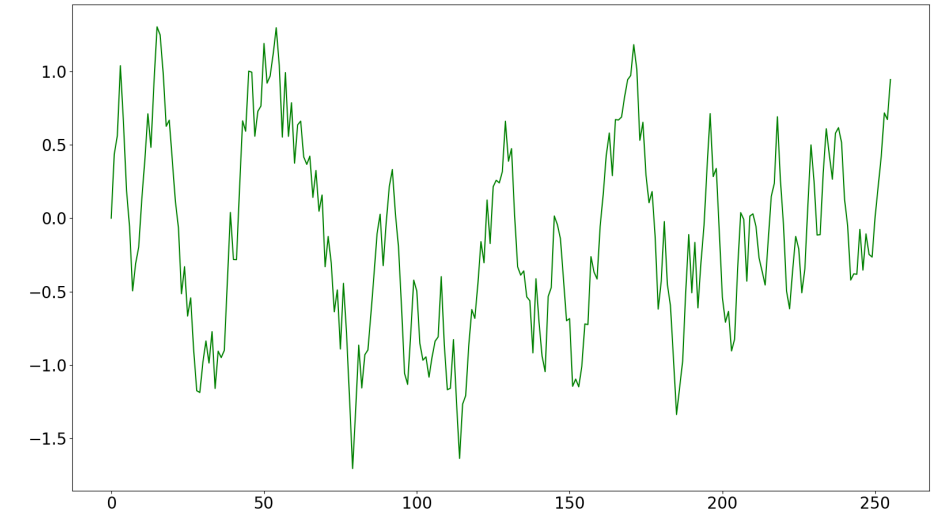


(b) Original signal - recreated signal

Figure 55: Recreate 1 k Hz tone using LPC order 5 and Rice codes, all coefficients recreated perfectly

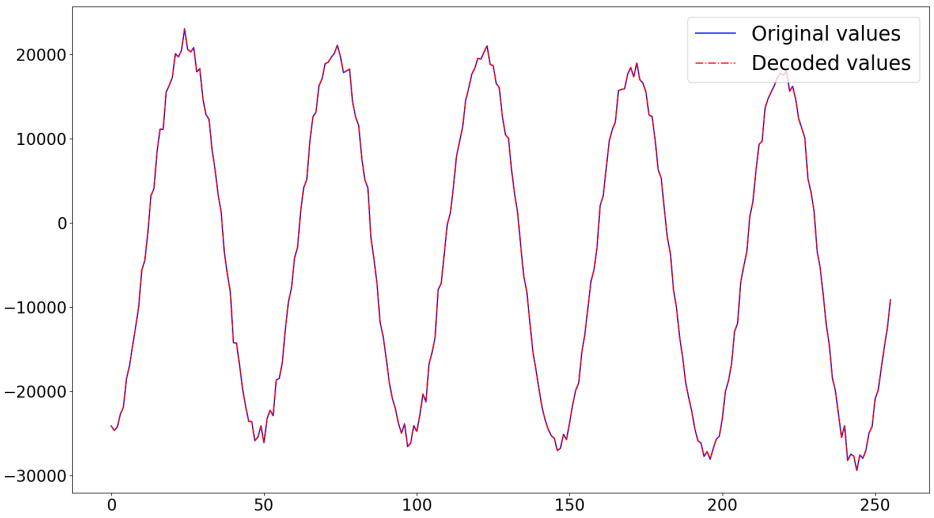


(a) Original and recreated signal

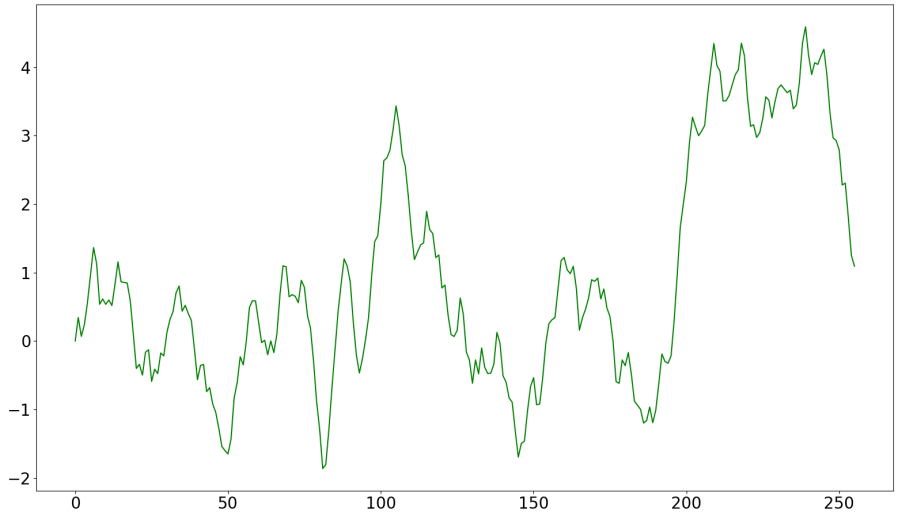


(b) Original signal - recreated signal

Figure 56: Recreate 1 k Hz tone using LPC order 1 and Golomb codes, all coefficients recreated perfectly

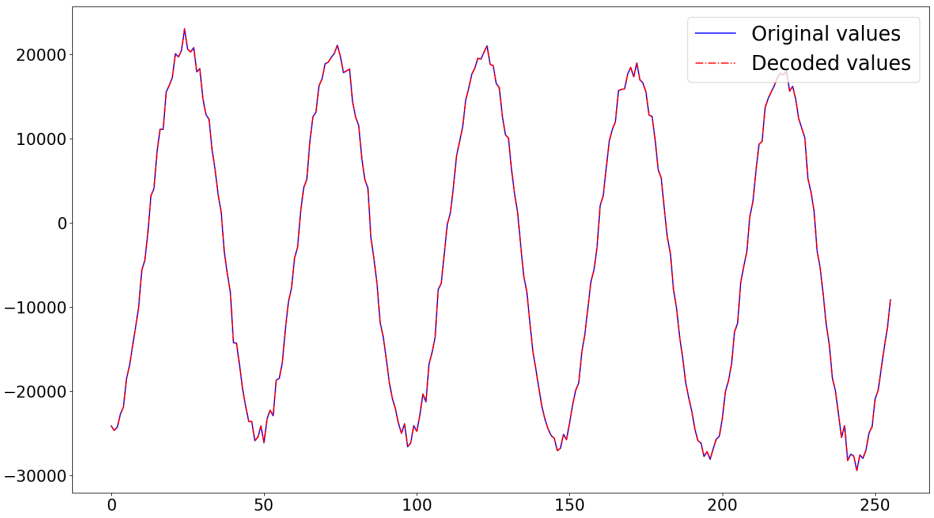


(a) Original and recreated signal

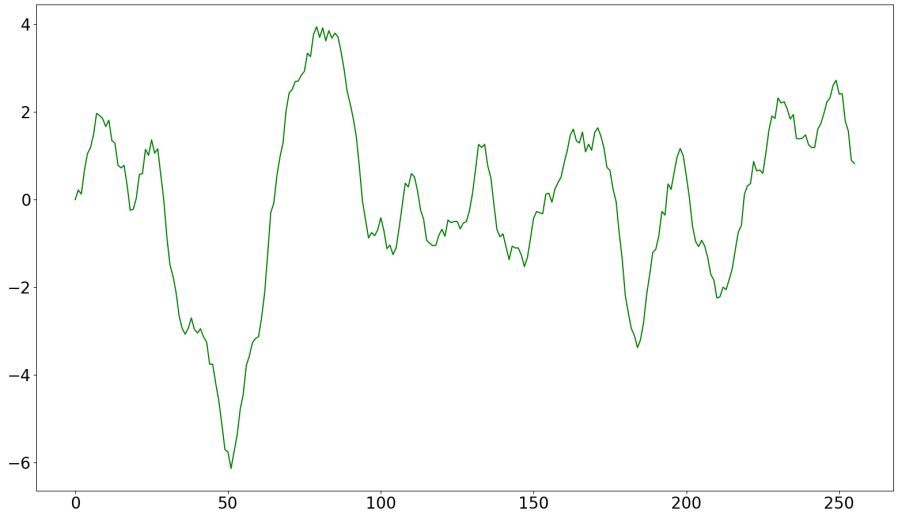


(b) Original signal - recreated signal

Figure 57: Recreate 1 k Hz tone using LPC order 2 and Golomb codes, all coefficients recreated perfectly

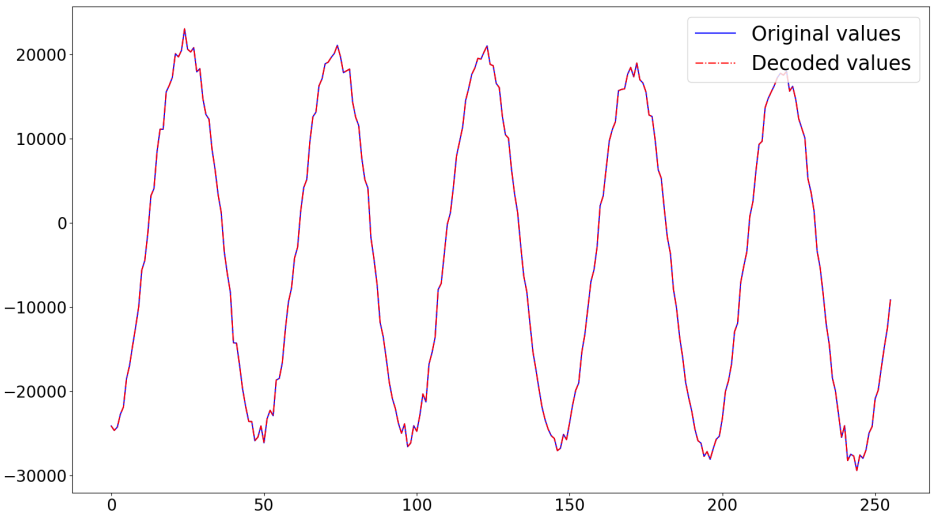


(a) Original and recreated signal

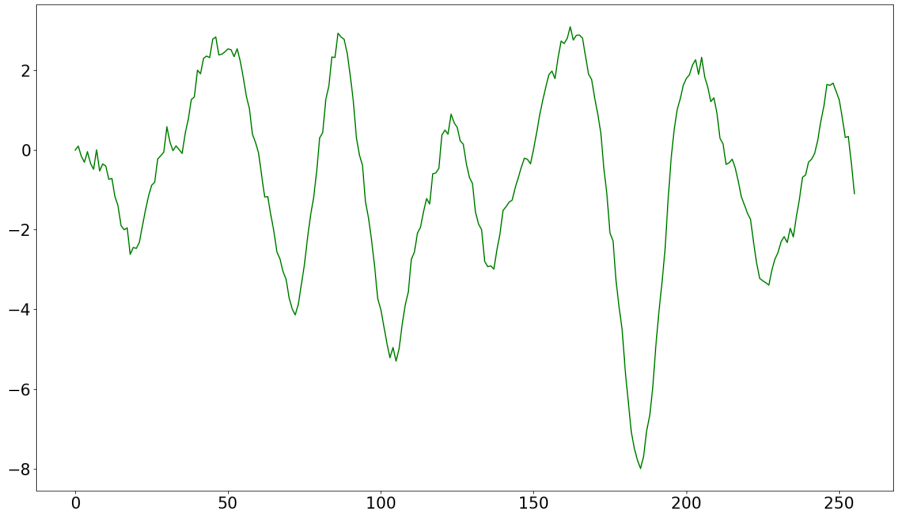


(b) Original signal - recreated signal

Figure 58: Recreate 1 k Hz tone using LPC order 3 and Golomb codes, all coefficients recreated perfectly

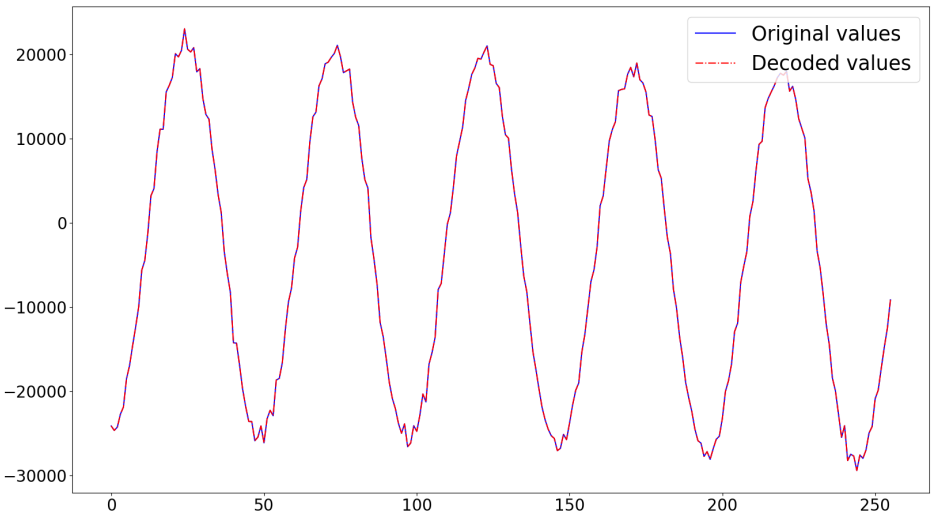


(a) Original and recreated signal

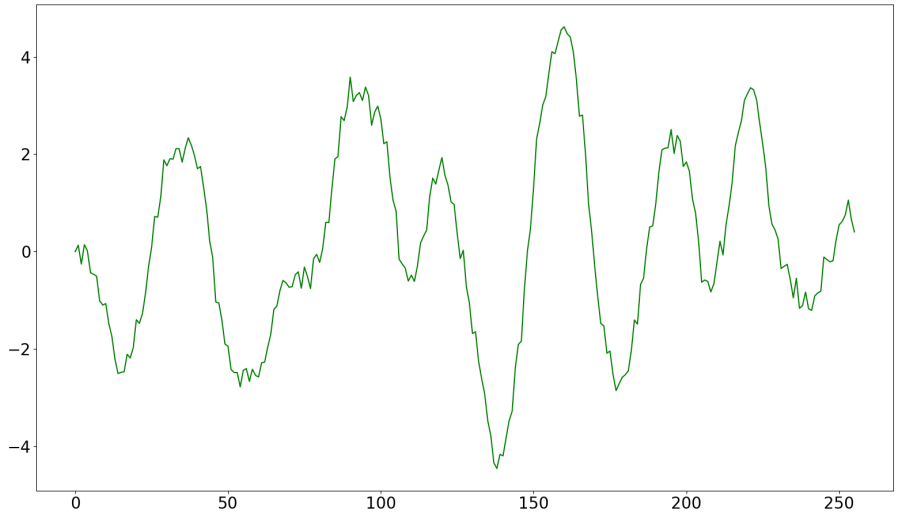


(b) Original signal - recreated signal

Figure 59: Recreate 1 k Hz tone using LPC order 4 and Golomb codes, all coefficients recreated perfectly

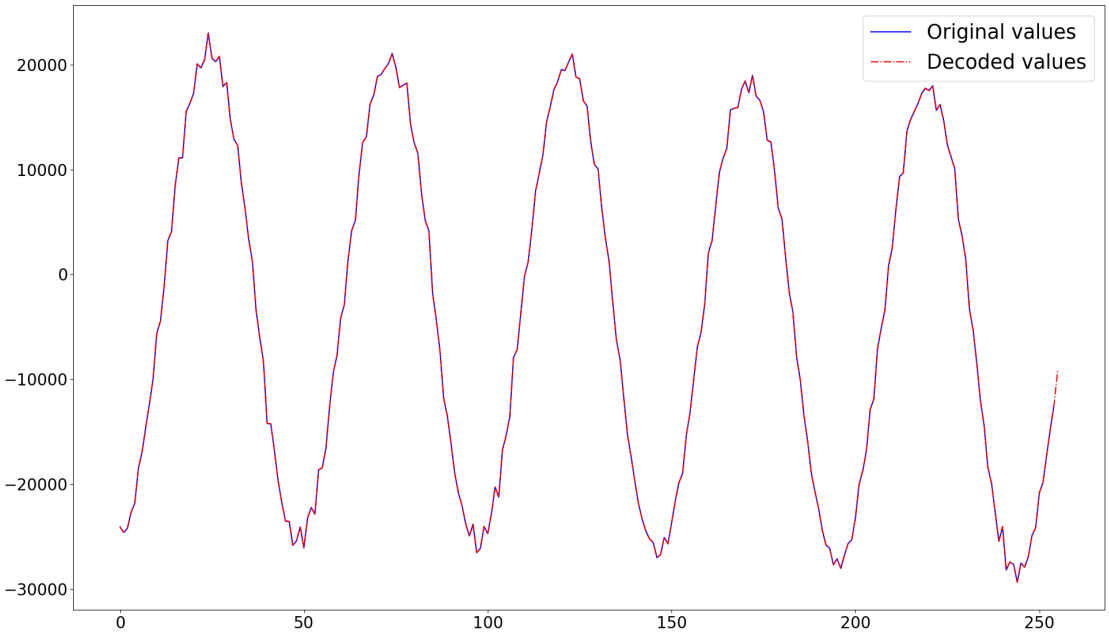


(a) Original and recreated signal

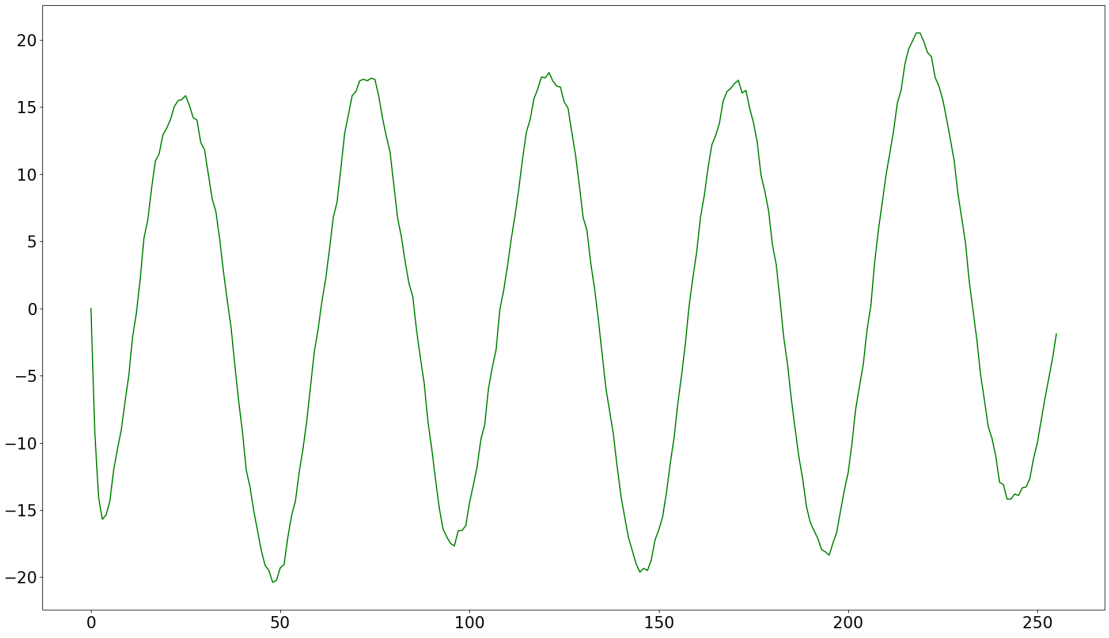


(b) Original signal - recreated signal

Figure 60: Recreate 1 k Hz tone using LPC order 5 and Golomb codes, all coefficients recreated perfectly

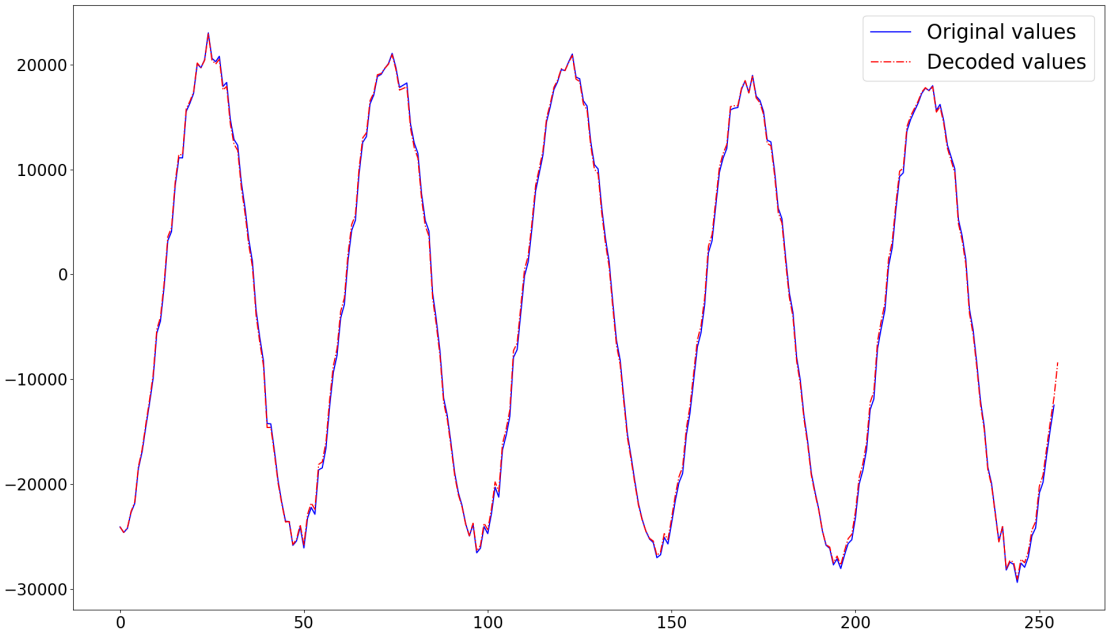


(a) Original and recreated signal

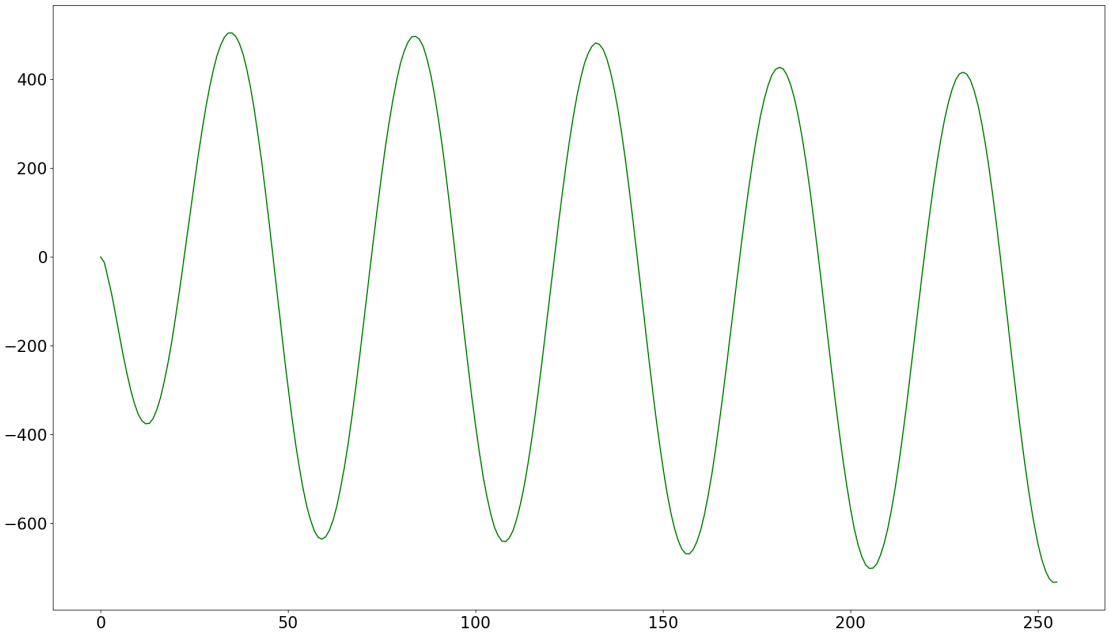


(b) Original signal - recreated signal

Figure 61: Recreate 1 k Hz tone using LPC order 2 and Rice codes, 10 bits metadata for decimals in coefficients

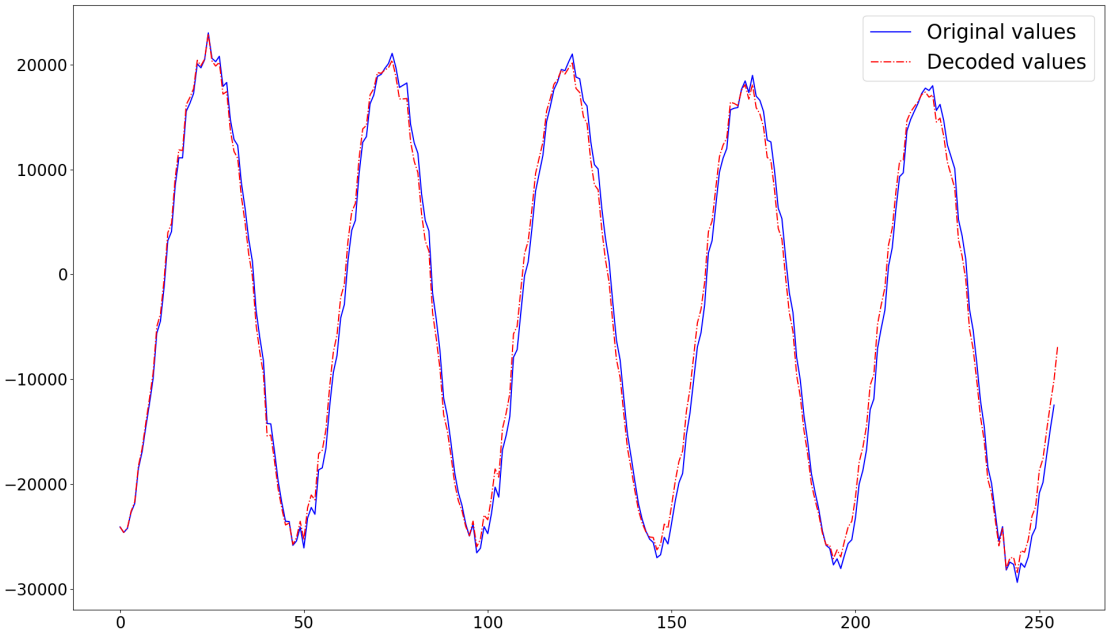


(a) Original and recreated signal

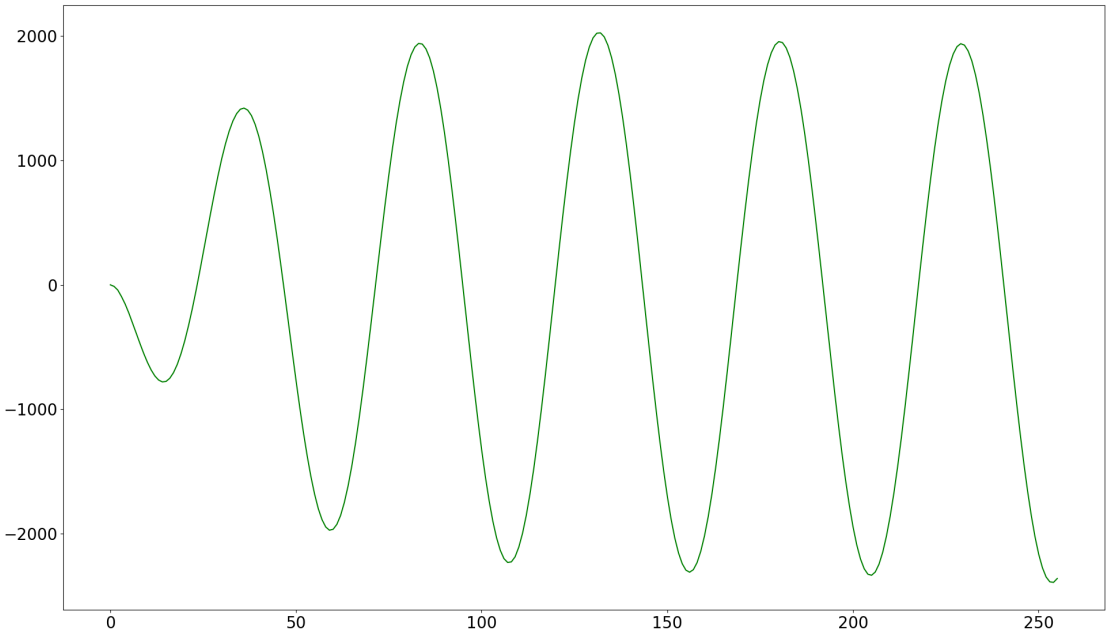


(b) Original signal - recreated signal

Figure 62: Recreate 1 k Hz tone using LPC order 3 and Rice codes, 10 bits metadata for decimals in coefficients

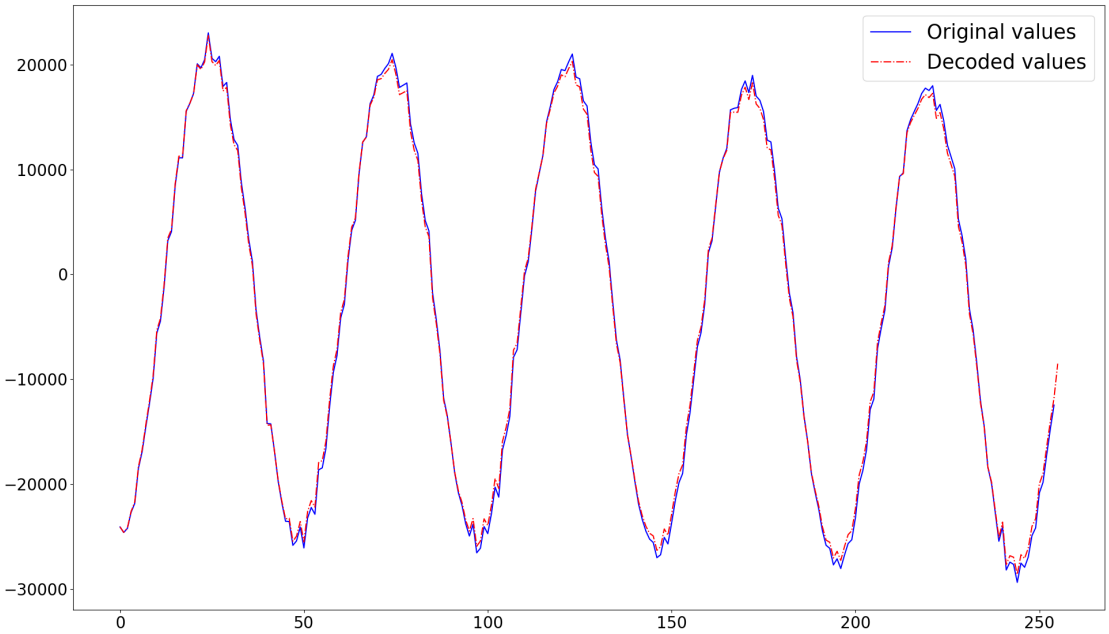


(a) Original and recreated signal

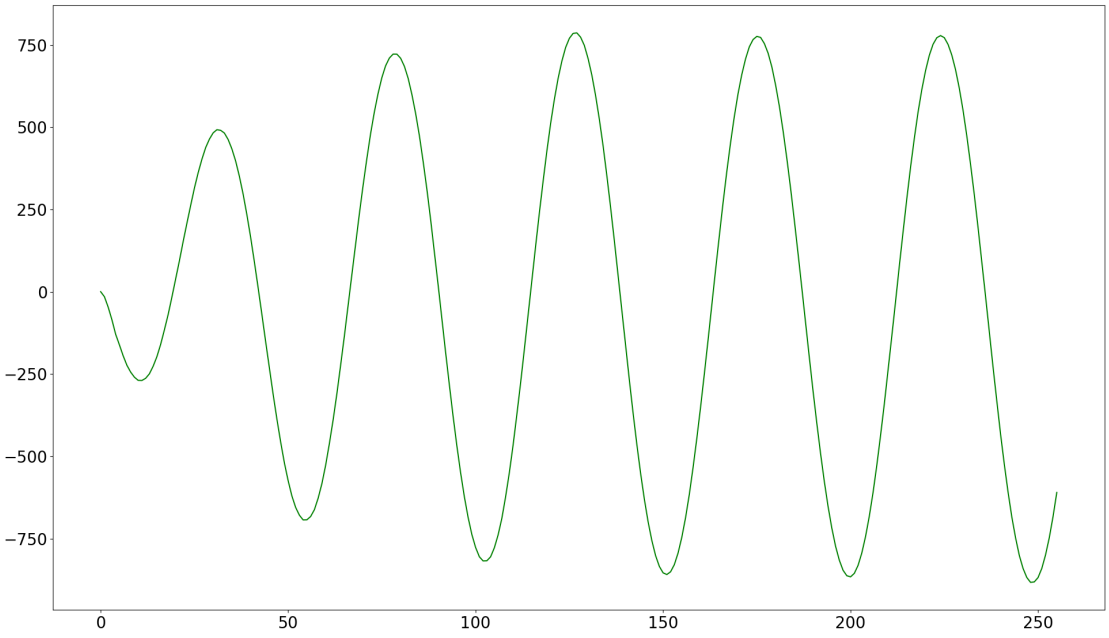


(b) Original signal - recreated signal

Figure 63: Recreate 1 k Hz tone using LPC order 4 and Rice codes, 10 bits metadata for decimals in coefficients

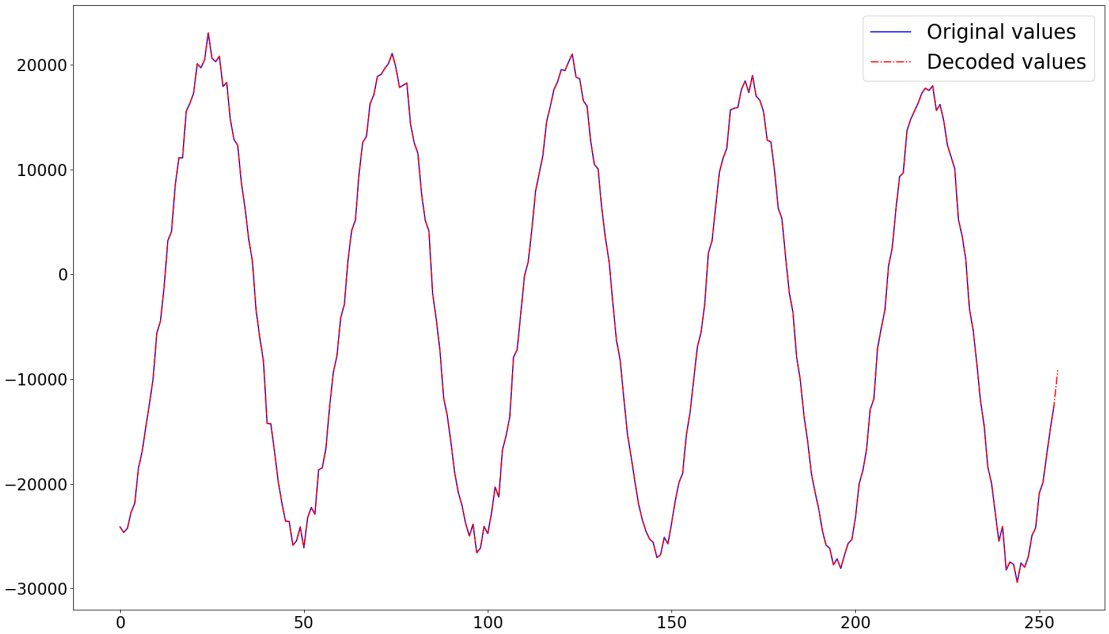


(a) Original and recreated signal

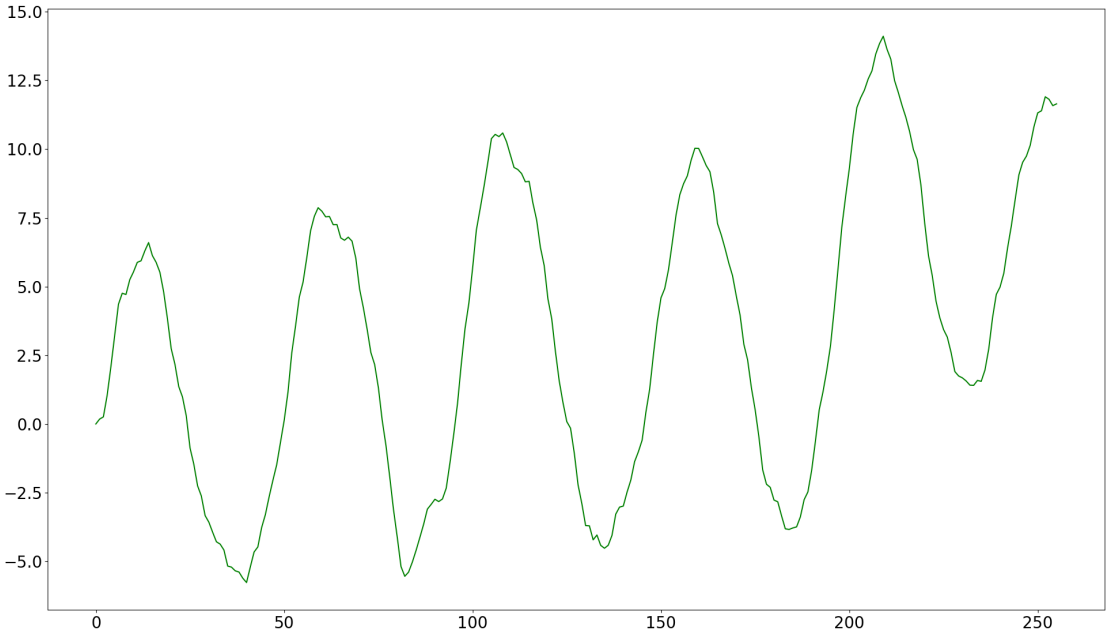


(b) Original signal - recreated signal

Figure 64: Recreate 1 k Hz tone using LPC order 5 and Rice codes, 10 bits metadata for decimals in coefficients

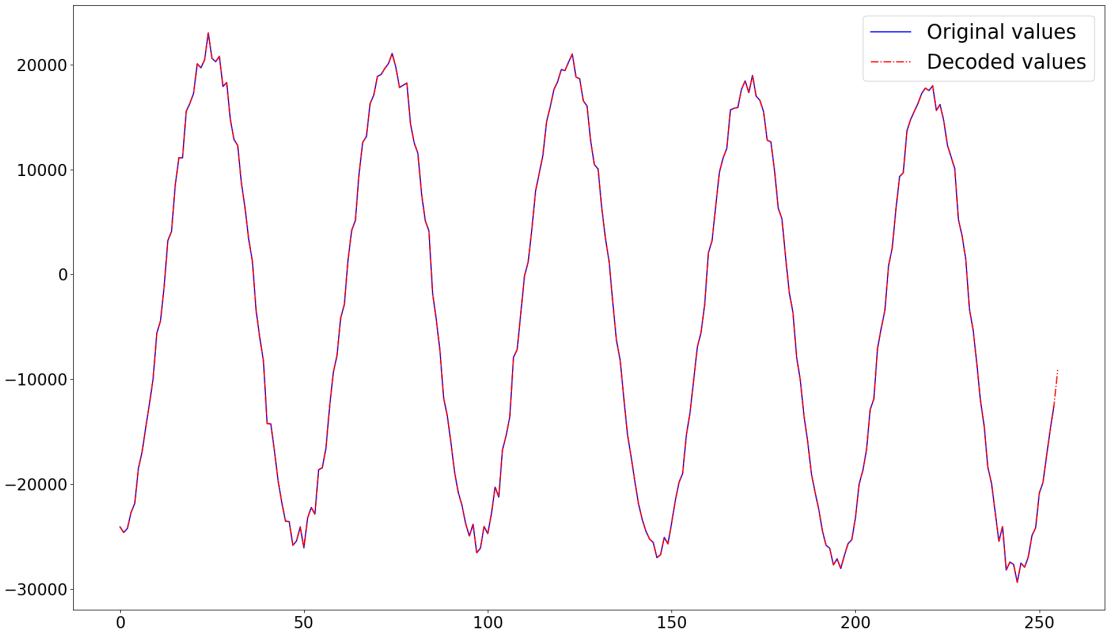


(a) Original and recreated signal

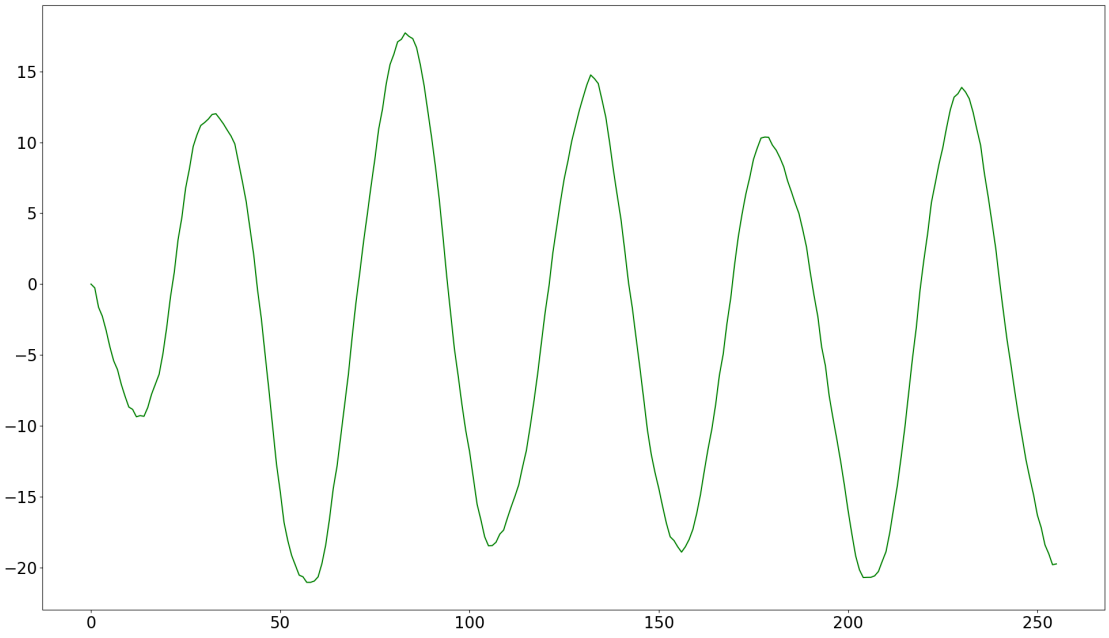


(b) Original signal - recreated signal

Figure 65: Recreate 1 k Hz tone using LPC order 2 and Rice codes, 15 bits metadata for decimals in coefficients

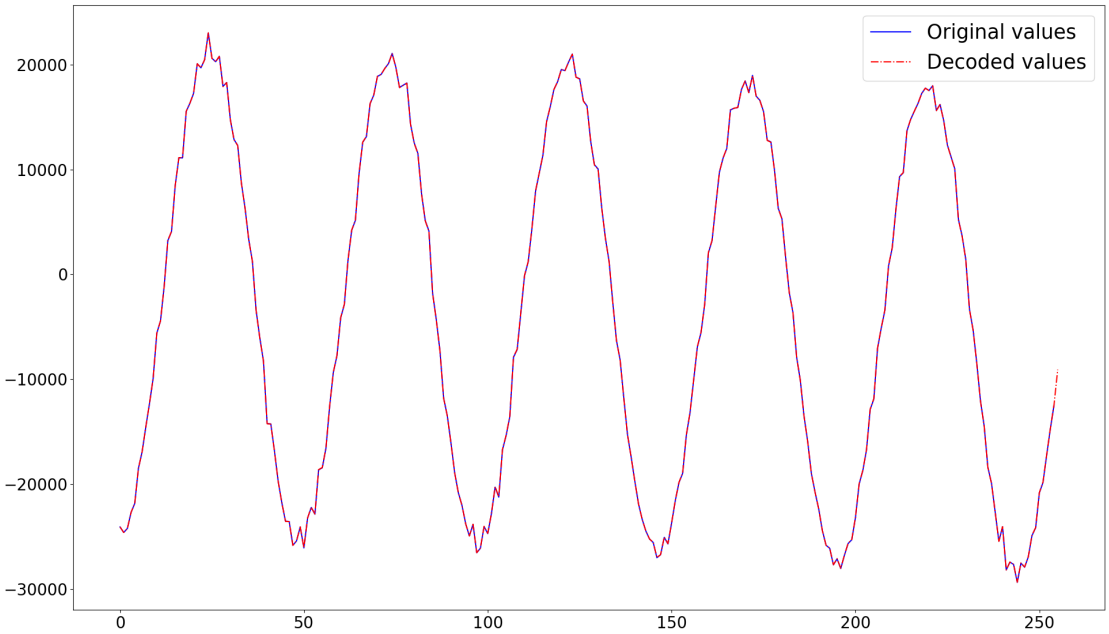


(a) Original and recreated signal

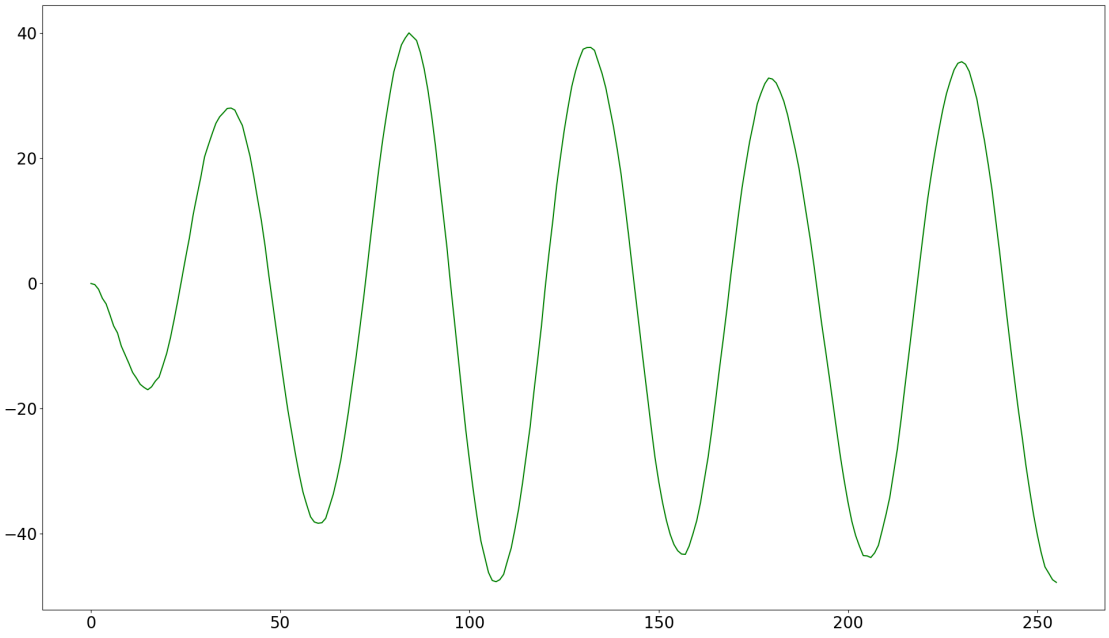


(b) Original signal - recreated signal

Figure 66: Recreate 1 k Hz tone using LPC order 3 and Rice codes, 15 bits metadata for decimals in coefficients

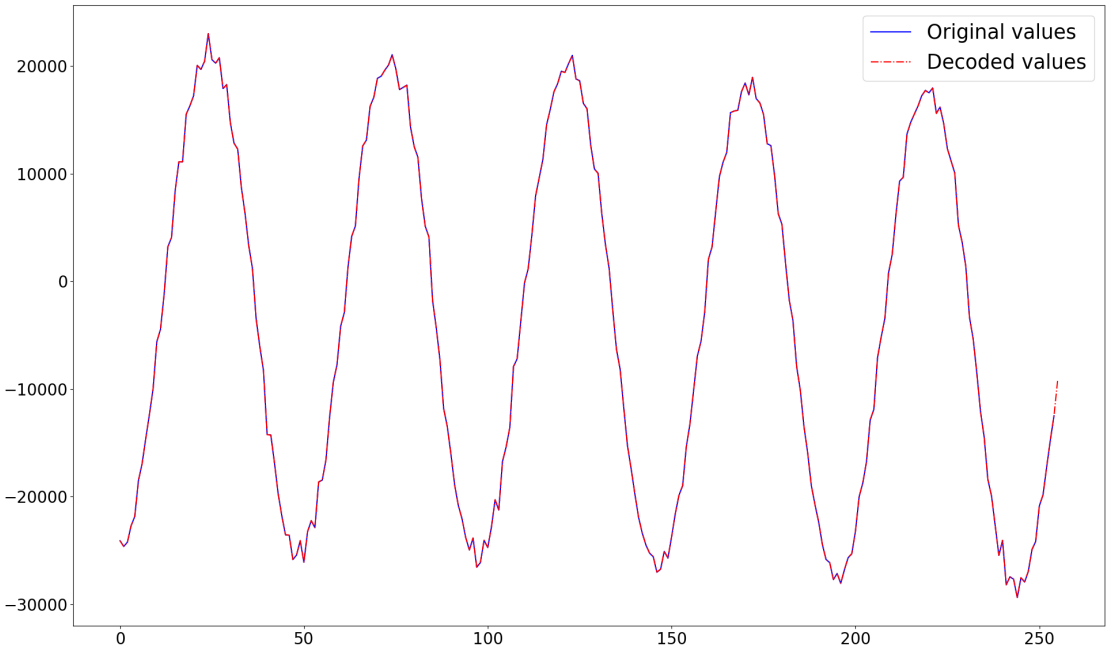


(a) Original and recreated signal

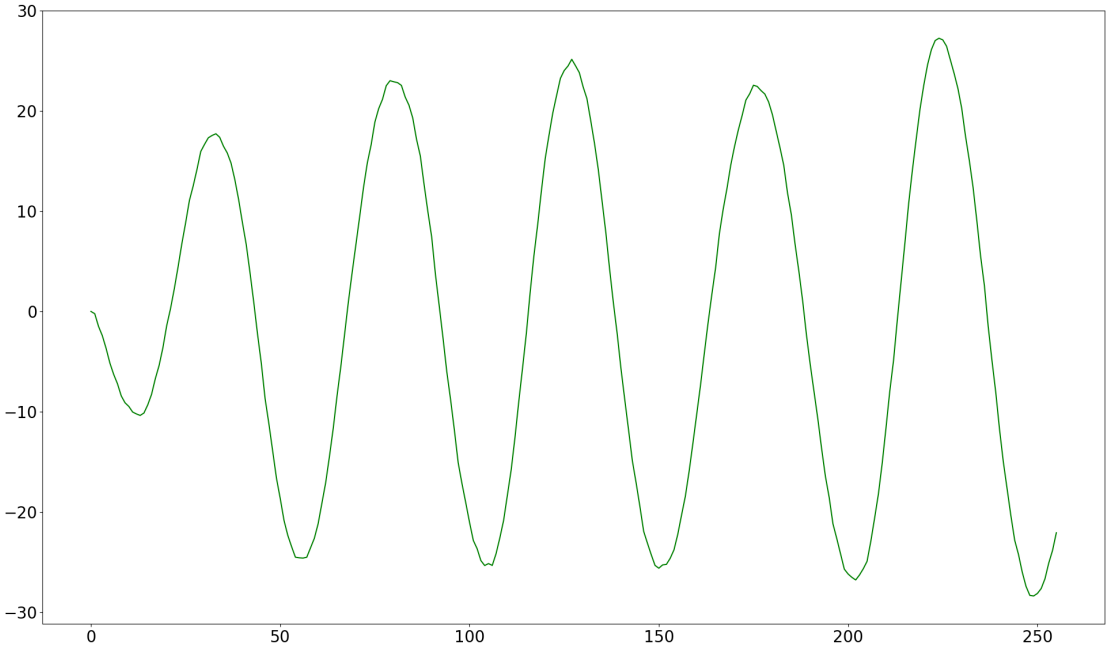


(b) Original signal - recreated signal

Figure 67: Recreate 1 k Hz tone using LPC order 4 and Rice codes, 15 bits metadata for decimals in coefficients

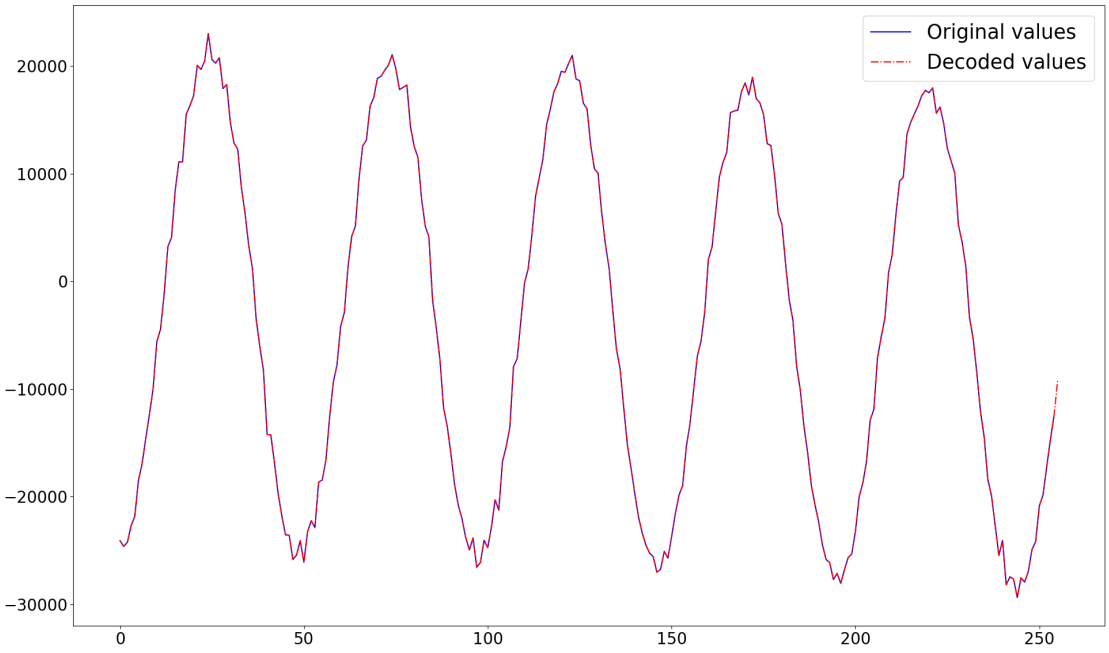


(a) Original and recreated signal

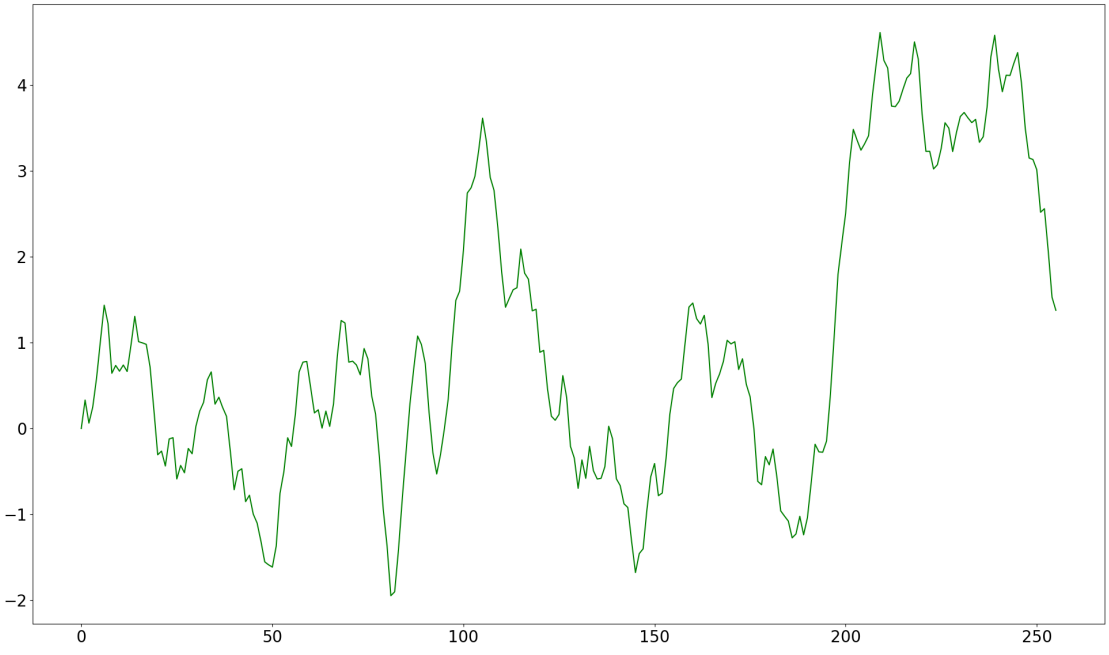


(b) Original signal - recreated signal

Figure 68: Recreate 1 k Hz tone using LPC order 5 and Rice codes, 15 bits metadata for decimals in coefficients

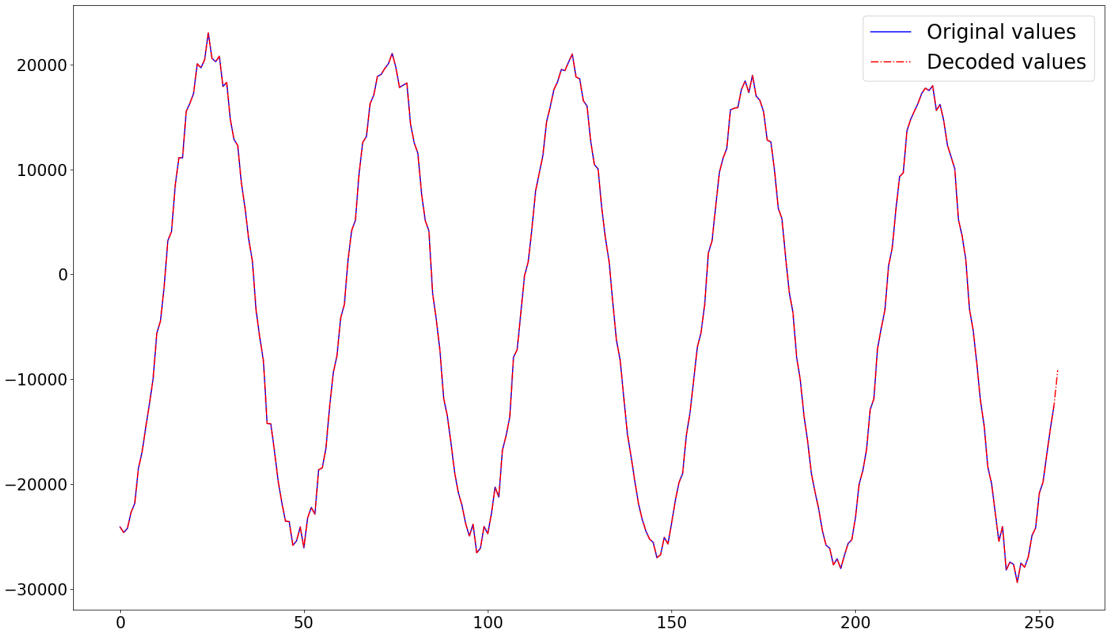


(a) Original and recreated signal

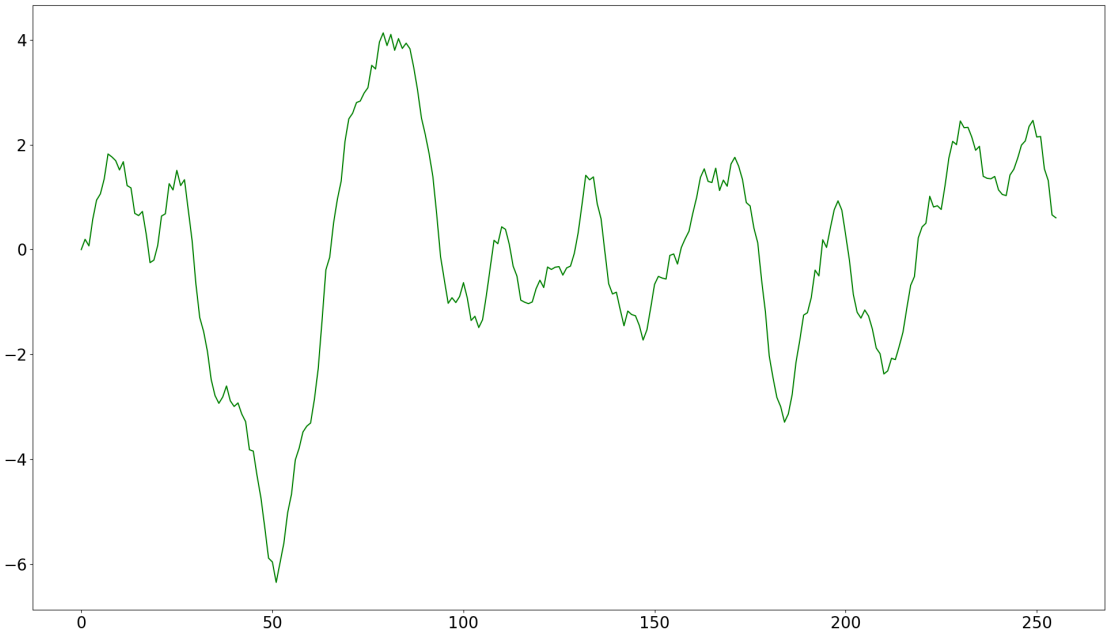


(b) Original signal - recreated signal

Figure 69: Recreate 1 k Hz tone using LPC order 2 and Rice codes, 10 bits metadata for decimals in coefficients

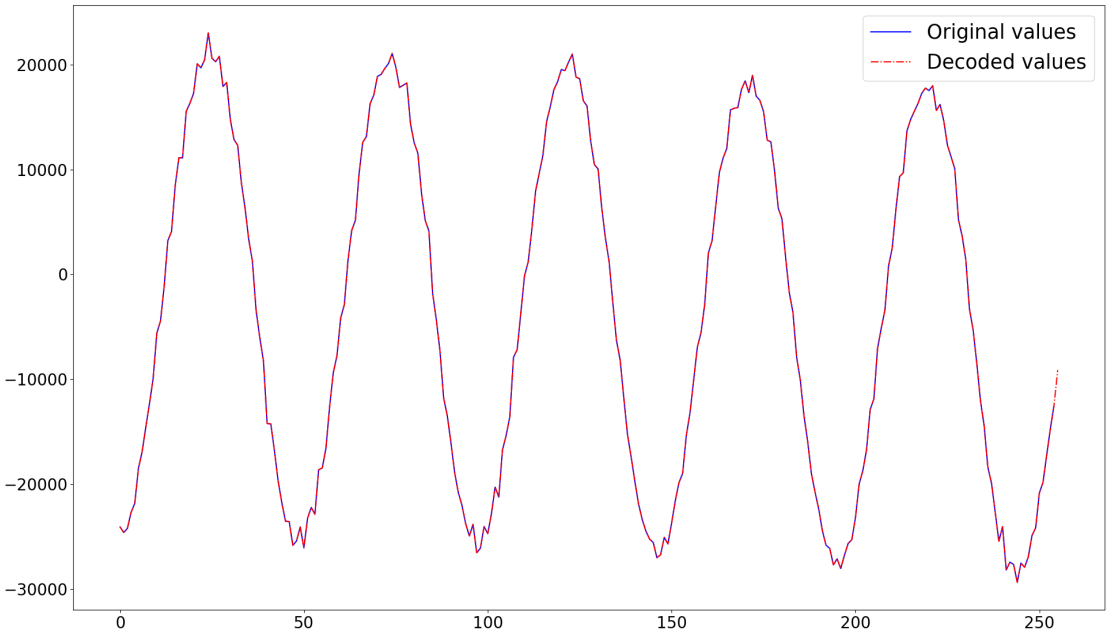


(a) Original and recreated signal

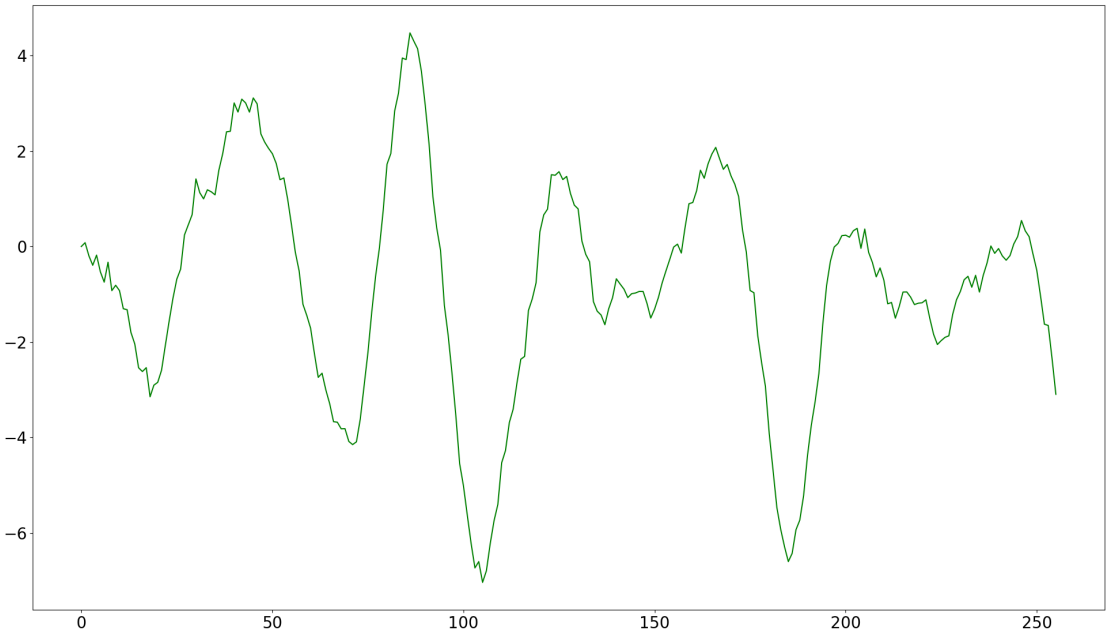


(b) Original signal - recreated signal

Figure 70: Recreate 1 k Hz tone using LPC order 3 and Rice codes, 20 bits metadata for decimals in coefficients

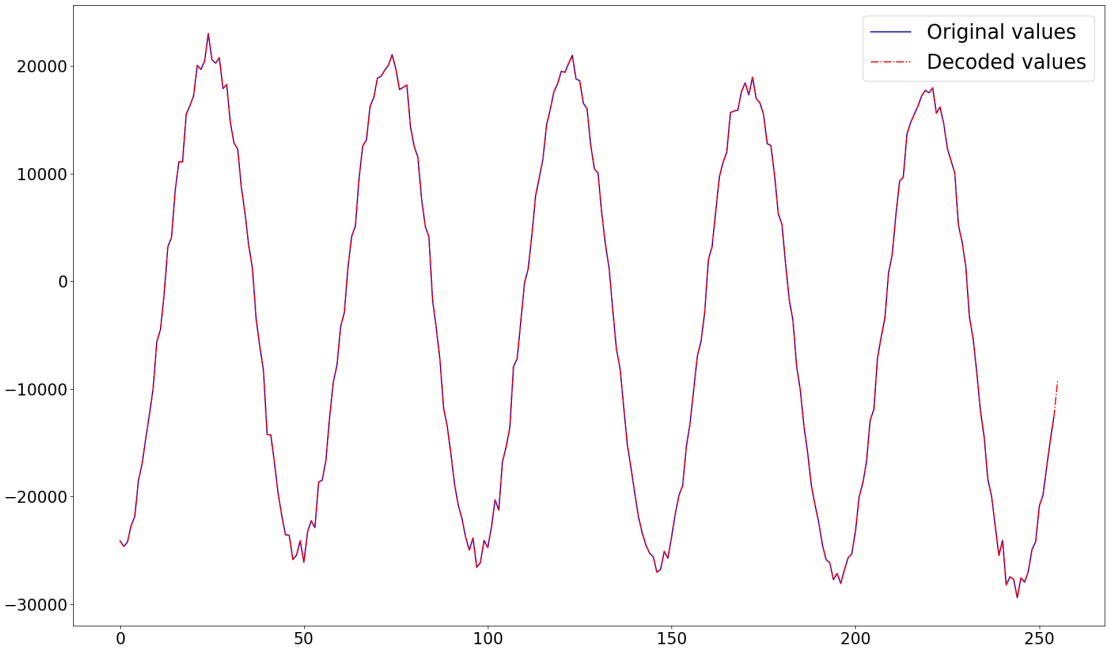


(a) Original and recreated signal

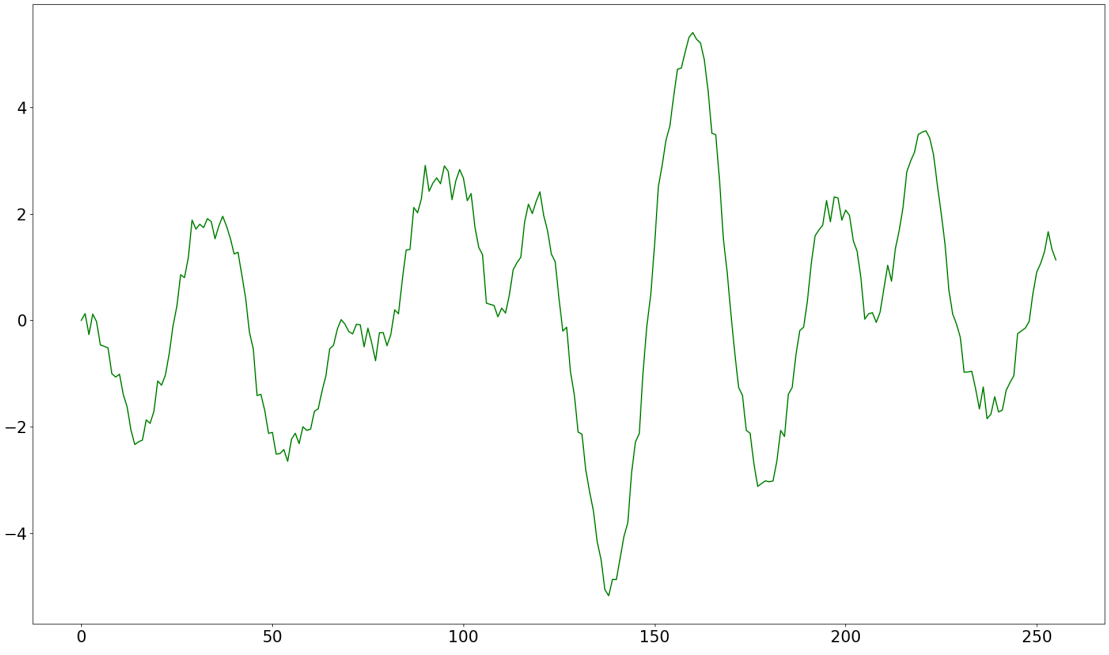


(b) Original signal - recreated signal

Figure 71: Recreate 1 k Hz tone using LPC order 4 and Rice codes, 20 bits metadata for decimals in coefficients



(a) Original and recreated signal



(b) Original signal - recreated signal

Figure 72: Recreate 1 k Hz tone using LPC order 5 and Rice codes, 20 bits metadata for decimals in coefficients

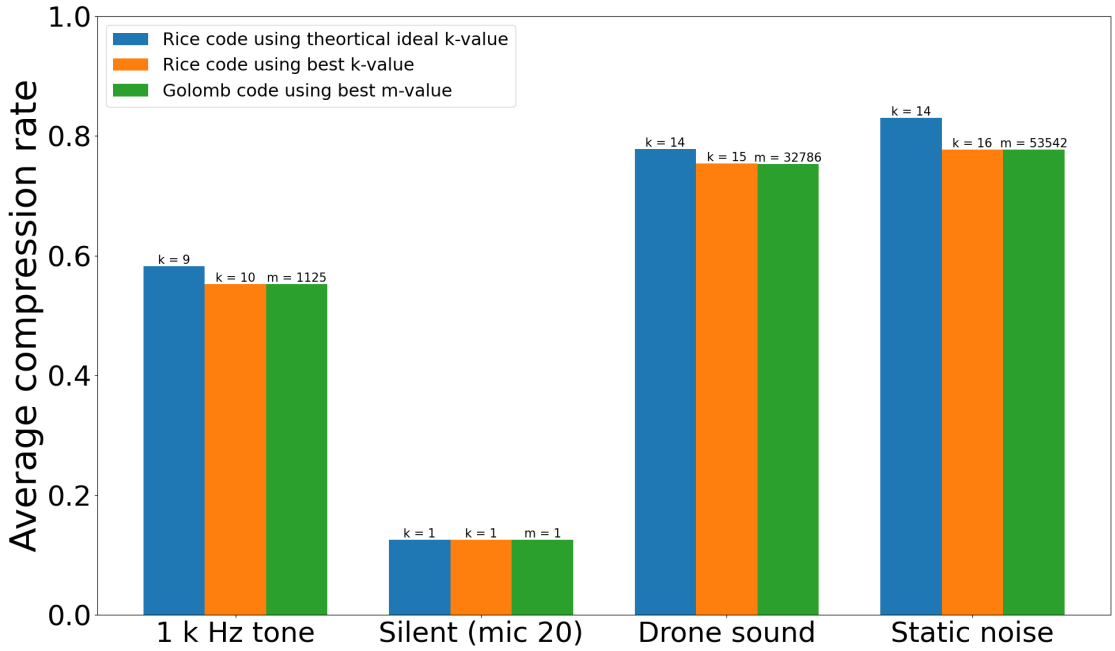


Figure 73: Average compression rate using LPC order 2

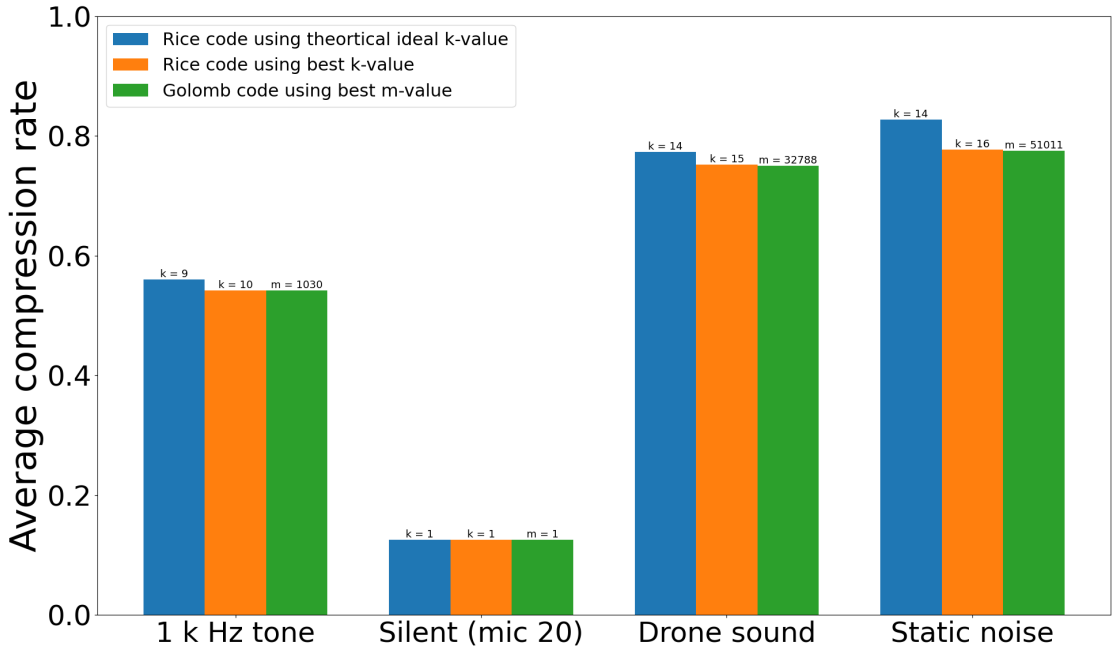


Figure 74: Average compression rate using LPC order 3

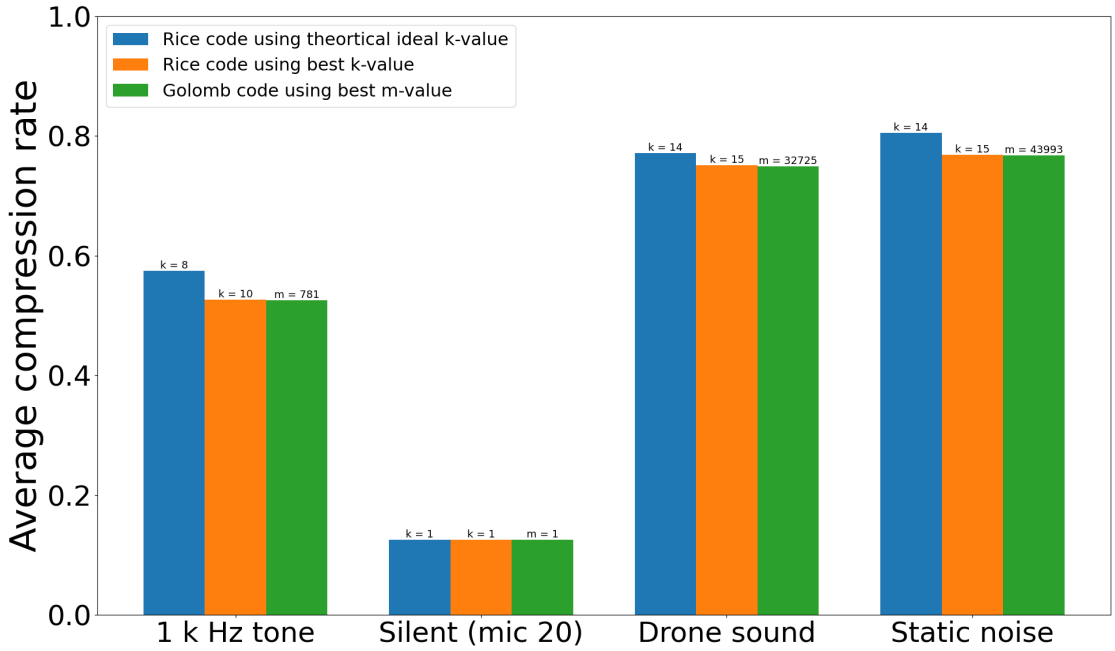


Figure 75: Average compression rate using LPC order 4

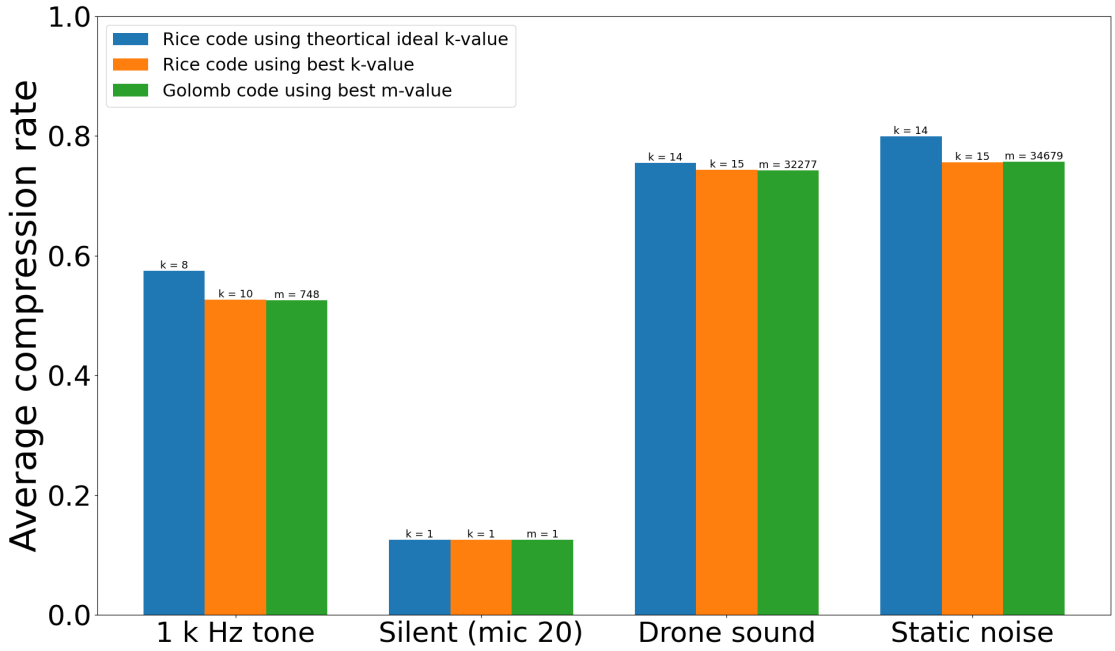


Figure 76: Average compression rate using LPC order 5

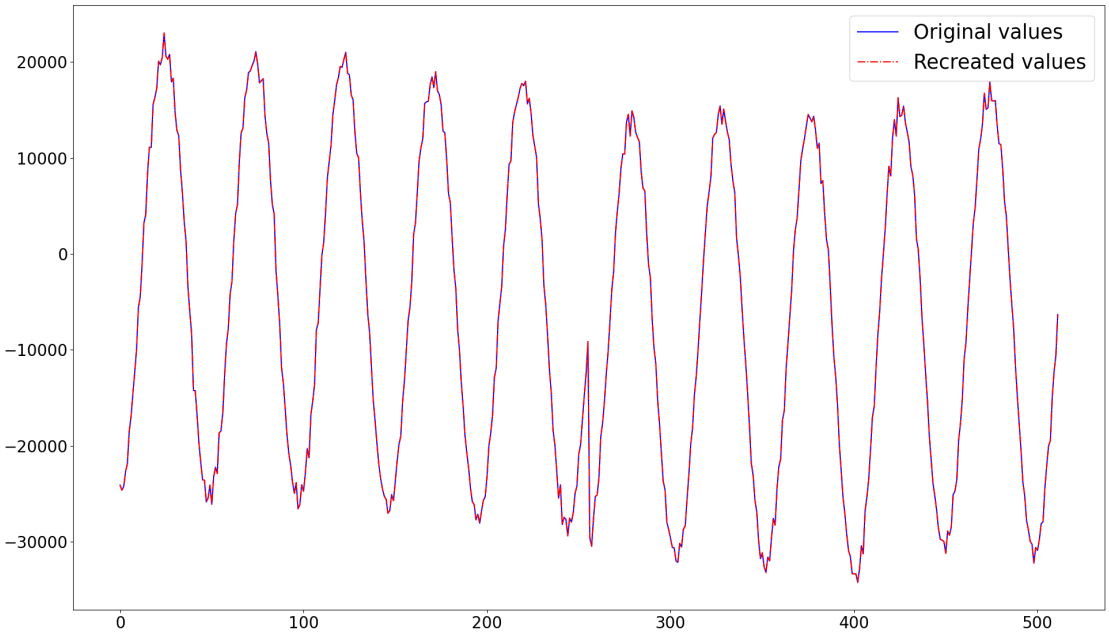


Figure 77: Recreate original signal using Adjacent algorithm with Rice codes

A.1.2 Tables

Order / k-value	Shorten order 0	Shorten order 1	Shorten order 2	Shorten order 3
8	cr = 3.309407552083333	cr = 0.7937744140625	cr = 0.6918619791666667	cr = 0.9262125651041668
9	cr = 1.8946533203125004	cr = 0.6391438802083333	cr = 0.5924886067708333	cr = 0.7160400390625001
10	cr = 1.2075927734375	cr = 0.5803059895833333	cr = 0.5577067057291667	cr = 0.6189046223958334
11	cr = 0.88515625	cr = 0.5723795572916666	cr = 0.5619140624999999	cr = 0.5917643229166667
12	cr = 0.7446940104166666	cr = 0.58974609375	cr = 0.5873860677083333	cr = 0.6001627604166668
13	cr = 0.6957600911458334	cr = 0.6253743489583334	cr = 0.6258138020833334	cr = 0.6279134114583333,
14	cr = 0.691748046875	cr = 0.6668212890625002	cr = 0.6670003255208333	cr = 0.6674235026041667
15	cr = 0.7091064453125	cr = 0.7083577473958335	cr = 0.7083984375000002	cr = 0.7085693359375

Table 2: Compression rate (not taking metadata into account) using different orders of Shorten with Rice codes at different k-values, with 1 k Hz tone as input using microphone 79

Order / k-value	Shorten order 0	Shorten order 1	Shorten order 2	Shorten order 3
1	cr = 0.125	cr = 0.125	cr = 0.125	cr = 0.125
2	cr = 0.16666666666666663	cr = 0.16666666666666663	cr = 0.16666666666666663	cr = 0.16666666666666663
3	cr = 0.20833333333333334	cr = 0.20833333333333334	cr = 0.20833333333333334	cr = 0.20833333333333334
4	cr = 0.25	cr = 0.25	cr = 0.25	cr = 0.25
5	cr = 0.29166666666666674	cr = 0.29166666666666674	cr = 0.29166666666666674	cr = 0.29166666666666674
6	cr = 0.33333333333333326	cr = 0.33333333333333326	cr = 0.33333333333333326	cr = 0.33333333333333326
7	cr = 0.375	cr = 0.375	cr = 0.375	cr = 0.375
8	cr = 0.4166666666666667	cr = 0.4166666666666667	cr = 0.4166666666666667	cr = 0.4166666666666667

Table 3: Compression rate (not taking metadata into account) using different orders of Shorten with Rice codes at different k-values, with silence as input using microphone 20

Order / k-value	Shorten order 0	Shorten order 1	Shorten order 2	Shorten order 3
10	cr = 3.3030843098958327	cr = 3.305574544270833	cr = 4.223356119791665	cr = 6.068538411458333
11	cr = 1.9330403645833332	cr = 1.9370198567708332	cr = 2.39814453125	cr = 3.3204671223958337
12	cr = 1.2686604817708333	cr = 1.2708251953125	cr = 1.5013020833333333	cr = 1.9620849609374997
13	cr = 0.9572916666666667	cr = 0.9583170572916666	cr = 1.0740071614583333	cr = 1.3038330078125
14	cr = 0.8228434244791666	cr = 0.8236083984375	cr = 0.8810791015625001	cr = 0.9959472656250001
15	cr = 0.7768636067708334	cr = 0.7773030598958333	cr = 0.8058430989583334	cr = 0.8629313151041668
16	cr = 0.77548828125	cr = 0.7756266276041667	cr = 0.7895426432291667	cr = 0.8175862630208334
17	cr = 0.7971842447916666	cr = 0.7973714192708334	cr = 0.8037353515625	cr = 0.8169677734375
18	cr = 0.8335286458333334	cr = 0.8336018880208333	cr = 0.8348551432291668	cr = 0.8393717447916668
19	cr = 0.875	cr = 0.875	cr = 0.875048828125	cr = 0.8754964192708332
20	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667

Table 4: Compression rate (not taking metadata into account) using different orders of Shorten with Rice codes at different k-values, with drone sound as input using microphone 79

Order / k-value	Shorten order 0	Shorten order 1	Shorten order 2	Shorten order 3
10	cr = 4.186490885416667	cr = 4.923518880208333	cr = 6.815055338541667	cr = 10.3636474609375
11	cr = 2.3744547526041666	cr = 2.744270833333333	cr = 3.696964518229167	cr = 5.479703776041667
12	cr = 1.4894938151041668	cr = 1.6743164062499996	cr = 2.1504638671875003	cr = 3.0420491536458334
13	cr = 1.0679524739583335	cr = 1.1603922526041666	cr = 1.3983723958333336	cr = 1.8439208984375
14	cr = 0.8781494140625	cr = 0.9243652343750002	cr = 1.0432942708333335	cr = 1.2656819661458334
15	cr = 0.8041178385416666	cr = 0.827392578125	cr = 0.8866048177083332	cr = 0.9977783203125001
16	cr = 0.7887207031250001	cr = 0.800244140625	cr = 0.8293619791666667	cr = 0.8848958333333332
17	cr = 0.8031575520833332	cr = 0.8084879557291668	cr = 0.8226725260416667	cr = 0.8495035807291667
18	cr = 0.8342936197916666	cr = 0.8358154296875	cr = 0.8415852864583332	cr = 0.853662109375
19	cr = 0.875	cr = 0.8750162760416667	cr = 0.8756266276041667	cr = 0.8788574218749998
20	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9167724609375

Table 5: Compression rate (not taking metadata into account) using different orders of Shorten with Rice codes at different k-values, with static noise as input, using microphone 79

Order / k-value	LPC order 1	LPC order 2	LPC order 3	LPC order 4	LPC order 5
5	cr = 3.4345377604166667	cr = 2.4585693359375	cr = 2.1077311197916666	1.562744140625	cr = 1.5526204427083334
6	cr = 1.9260335286458332	cr = 1.4617187500000002	cr = 1.2953450520833332	cr = 1.0278483072916667	cr = 1.0251302083333336
7	cr = 1.161181640625	cr = 0.9288818359375	cr = 0.8458658854166666	cr = 0.7119547526041666	cr = 0.7107747395833334
8	cr = 0.7995198567708334	cr = 0.6835774739583332	cr = 0.6416259765625	cr = 0.5751871744791667	cr = 0.5746744791666667
9	cr = 0.6397867838541667	cr = 0.5820475260416667	cr = 0.5608479817708333	cr = 0.5279947916666667	cr = 0.5276936848958333
10	cr = 0.58076171875	cr = 0.5524739583333333	cr = 0.5417968750000001	cr = 0.5262125651041667	cr = 0.5258382161458333
11	cr = 0.5725748697916667	cr = 0.5596923828125	cr = 0.5546793619791667	cr = 0.5483317057291667	cr = 0.5481526692708334
12	cr = 0.5897460937500001	cr = 0.586181640625	cr = 0.5852213541666667	cr = 0.5848063151041667	cr = 0.584814453125
13	cr = 0.6253417968749999	cr = 0.6255452473958333	cr = 0.6255777994791667	cr = 0.6255777994791666	cr = 0.6255289713541667
14	cr = 0.666796875	cr = 0.6668701171874999	cr = 0.6668375651041667	cr = 0.666845703125	cr = 0.6668294270833333
15	cr = 0.7083658854166668	cr = 0.7083740234375001	cr = 0.7083740234375001	cr = 0.7083740234375001	cr = 0.7083740234375001
16	cr = 0.75	cr = 0.75	cr = 0.75	cr = 0.75	cr = 0.75

Table 6: Compression rate (not taking metadata into account) using different orders of LPC with Rice codes at different k-values, with 1 k Hz tone as input, using microphone 79

Order / k-value	LPC order 1	LPC order 2	LPC order 3	LPC order 4	LPC order 5
1	cr = 0.125	cr = 0.125	cr = 0.125	cr = 0.125	cr = 0.125
2	cr = 0.16666666666666663	cr = 0.16666666666666663	cr = 0.16666666666666663	cr = 0.16666666666666663	cr = 0.16666666666666663
3	cr = 0.20833333333333334	cr = 0.20833333333333334	cr = 0.20833333333333334	cr = 0.20833333333333334	cr = 0.20833333333333334
4	cr = 0.25	cr = 0.25	cr = 0.25	cr = 0.25	cr = 0.25
5	cr = 0.29166666666666674	cr = 0.29166666666666674	cr = 0.29166666666666674	cr = 0.29166666666666674	cr = 0.29166666666666674
6	cr = 0.33333333333333326	cr = 0.33333333333333326	cr = 0.33333333333333326	cr = 0.33333333333333326	cr = 0.33333333333333326
7	cr = 0.375	cr = 0.375	cr = 0.375	cr = 0.375	cr = 0.375
8	cr = 0.4166666666666667	cr = 0.4166666666666667	cr = 0.4166666666666667	cr = 0.4166666666666667	cr = 0.4166666666666667

Table 7: Compression rate (not taking metadata into account) using different orders of LPC with Rice codes at different k-values, with silence as input, using microphone 20

Order / k-value	LPC order 1	LPC order 2	LPC order 3	LPC order 4	LPC order 5
11	cr = 1.7375813802083335	cr = 1.5669026692708332	cr = 1.5272542317708333	cr = 1.5131998697916667	cr = 1.3738525390625003
12	cr = 1.1723307291666667	cr = 1.0881510416666664	cr = 1.0695393880208333	cr = 1.06240234375	cr = 0.99404296875
13	cr = 0.9091634114583333	cr = 0.8672119140625	cr = 0.8576416015625	cr = 0.8544026692708332	cr = 0.8203938802083334
14	cr = 0.7989420572916666	cr = 0.7778727213541666	cr = 0.7733072916666667	cr = 0.7717692057291667	cr = 0.7546630859375
15	cr = 0.764892578125	cr = 0.7544514973958333	cr = 0.752490234375	cr = 0.7512125651041666	cr = 0.7431396484375
16	cr = 0.7698974609374999	cr = 0.7651448567708333	cr = 0.7640380859375	cr = 0.7636881510416667	cr = 0.7598225911458334
17	cr = 0.7954264322916667	cr = 0.7940022786458335	cr = 0.7938802083333334	cr = 0.7936686197916667	cr = 0.7926920572916666
18	cr = 0.8334147135416667	cr = 0.8333984375000002	cr = 0.8333984375000002	cr = 0.8333902994791668	cr = 0.8333577473958333
19	cr = 0.875	cr = 0.875	cr = 0.875	cr = 0.875	cr = 0.875
20	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667

Table 8: Compression rate (not taking metadata into account) using different orders of LPC with Rice codes at different k-values, with drone sound as input, using microphone 79

Order / k-value	LPC order 1	LPC order 2	LPC order 3	LPC order 4	LPC order 5
11	cr = 2.25615234375	cr = 1.9848063151041664	cr = 1.9643066406249996	cr = 1.7797526041666665	cr = 1.5676513671875
12	cr = 1.4300862630208333	cr = 1.2963297526041666	cr = 1.286376953125	cr = 1.1958740234375	cr = 1.0925048828125
13	cr = 1.038134765625	cr = 0.9714192708333332	cr = 0.9663818359375	cr = 0.9208984375	cr = 0.8694905598958332
14	cr = 0.8631022135416666	cr = 0.8300211588541668	cr = 0.8274332682291666	cr = 0.8046549479166665	cr = 0.7792236328125
15	cr = 0.7969401041666667	cr = 0.7802571614583333	cr = 0.7792073567708333	cr = 0.7679361979166667	cr = 0.7555094401041667
16	cr = 0.7851318359375	cr = 0.7771728515625	cr = 0.7765055338541667	cr = 0.7709879557291667	cr = 0.7656331380208332
17	cr = 0.8017008463541666	cr = 0.7982666015625	cr = 0.798046875	cr = 0.7954833984375	cr = 0.7939127604166666
18	cr = 0.8340087890625	cr = 0.8336344401041667	cr = 0.8336669921875	cr = 0.8334147135416667	cr = 0.8333902994791668
19	cr = 0.875	cr = 0.875	cr = 0.875	cr = 0.875	cr = 0.875
20	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667	cr = 0.9166666666666667

Table 9: Compression rate (not taking metadata into account) using different orders of LPC with Rice codes at different k-values, with static noise as input, using microphone 79

Audio file / k-value	1 k Hz tone	Drone sound	Static noise
8	cr = 0.8500504811604804	-	-
9	cr = 0.6653977711995446	-	-
10	cr = 0.5942123413085907	-	-
11	cr = 0.5801274617513004	-	-
12	cr = 0.5951808929443353	cr = 1.0085985819498702	cr = 1.1877508799234995
13	cr = 0.6281656901041665	cr = 0.8280771891276049	cr = 0.9175244649251315
14	cr = 0.667703755696605	cr = 0.7590602874755846	cr = 0.8035437266031901
15	cr = 0.708484268188506	cr = 0.7461541493733703	cr = 0.7679500579833974
16	cr = 0.75	cr = 0.7623184204101547	cr = 0.7721909840901684
17	-	cr = 0.7942623138427689	cr = 0.7975072224934917
18	-	cr = 0.8335726420085013	cr = 0.8339841206868762
19	-	cr = 0.8750035603841146	cr = 0.8750086466471352
20	-	cr = 0.9166666666666524	cr = 0.9166666666666524

Table 10: Compression rate (not taking metadata into account) using different orders of Adjacent with Rice codes at different k-values

A.2 Python code

A.2.1 Golomb class code

```
import math
import numpy as np

class GolombCoding:

    #m is a constant that affects the length of the code word.
    #For larger residuals m should be larger.
    #m have to be an int
    #sign is True if the input data can be signed

    def __init__(self, m, sign):
        self.m = m
        self.sign = sign

    def Encode(self, n):



        cc = math.log(self.m,2)
        c = math.ceil(cc)
        b = pow(2,c) - self.m

        #encoded is started with a binary "0".
        #This is to be able to add the unary coded 1's on the
   left side
        #and the bianry coded remainder on the right side
        encoded = bin(0)[2:]

        #Checks if the input data is signed in order to save a
    signed bit
        if self.sign == True:
            if n < 0:
                s = "1"
                #in cases where the input data is negative it
   is converted to positive before encoded.
                #The sign bit is saved as a "1" beffore to
   remember that the value should be negative
```

```
                n = -n

            else:
                s = "0"

        #q is caled the quotient
        #this value will be unary coded in "1's" followed by a
 0
        #r is called the remainder
        #this value will be binary coded.
        #in some cases the binary coding will differ (More
bellow)
        q = math.floor(n/ self.m)
        r = n % self.m



        #Unary codes the quotient.
        #The int value q is represented in a number of "1's"
followed by a 0
        #this is achived by assining a 1 as the MSB to encoded
 for every iteation of the loop
        #with length q
        for i in range(q):
            encoded = "1" + encoded

        #checks that the encoded length is correct
        if len(encoded) != q+1:
            raise ValueError(f"Unary code size does not match
q value. Current values are: encoded = {encoded}, q = {q}")


        #There are three differente cases for how r can be
encoded:

        #Case 1. if cc is an interger that means that m is a
power of 2.
        #in this case r can be encoded in binary in c bits
        if cc.is_integer():
            encoded = encoded + np.binary_repr(r, width = c)
        #Case 2. if r is less than  c   - m it can be encoded
```

```
in binary using c-1 bits
        elif r < b:
            encoded = encoded + np.binary_repr(r, width = c-1)
        #Case 3. if r is larger or equal to c   - m it will be
 encoded so that the largest residual
        #is encoded in all "1" with length c, for example if c
 = 2 then the largest residual will be "11".
        #The largest possible residual will allways be m-1.
        #To easily get the correct binary value it is possible
 to take r +  c   - m = r + b and then convert to binary
        else:
            encoded = encoded + np.binary_repr(r+b, width = c)


        #If the data is signed the signed bit is added
        if self.sign == True:
            encoded = s + encoded

        #returns the encoded value
        return encoded



    def Decode(self, code):

        cc = math.log(self.m,2)
        c = math.ceil(cc)
        b = pow(2,c) - self.m
        decoded_values = []

        while len(code) > 0:
            A = 0
            S = "0"
            R = ""



            #Checks if the data is signed
            if self.sign == True:
                #if the sign bit is "1" the result should be
```

```
negative,
                #to ensure this S is set to "1"
                if code[0] == "1":
                    S = "1"
                #The sign bit is removed and the rest of the
code word can be decoded as if it was unsigned
                code = code[1:]

            #Decodes the unary part of the codeword by staying
 in the while loop aslong as the MSB is "1"
            while code[0] == "1":
                #for every "1" in the code word A is
incremented by one
                A += 1
                #Removes the MSB of the codeword
                code = code[1:]
            #After the unary code is decoded the MSB will be a
 "0".
            #This zero exist to indicate that the unary code
is done and can be removed
            code = code[1:]



            #The c-1 last bits are converted to integer and
denoted R
            for i in range(c-1):
                R += code[0]
                code = code[1:]




            #There are 3 cases when decoding, reflecting the 3
 cases for encoding

            #Case 1. m is a power of 2, then cc will be an
interger.
            if (cc).is_integer():
                #In this case the remaning bits should be of
```

```
lenght c
                #One more bits is removed from code and added
to R
                R += code[0]
                code = code[1:]

                #The remaining bits are converted to int and
added to the unary decoded A times m
                value = self.m * A + int(R, 2)
            else:

                #Case 2. if R is less than c   - m
                if int(R, 2) < b:
                    #In these cases the remaning code should
be length c-1. Else and error will be raised
                    #The remaining bits are converted to int
and added to the unary decoded A times m
                    value = self.m * A + int(R, 2)
                #Case 3. if R is greater or equal to c   - m
                else:
                    #in this case the remaining code should be
 of length c.
                    #One more bits is removed from code and
added to R
                    R += code[0]
                    code = code[1:]

                    #to decode in case the unary decoded A is
multiplied with m and added to the interger valued code as
the previous cases.
                    #But in this case b is subtracted (b = c
 - m). This is the deconding of the encoding case where the
 largest remainder is encoded in "1" bits with length c.
                    value = self.m * A + int(R, 2) - b



            #Checks if the data is signed and if S ="1" the
decoded value should be negative
            if self.sign and S == "1":
                value = -value
```

```
            #Adds the decoded value to the array of the
    decoded values
            decoded_values.append(value)

        #Returns the decoded values in an array
        return decoded_values
```

A.2.2 Rice class code

```
import numpy as np
import math



class RiceCoding:

    #k should be an interger based on expected values
    calculated from previous sampel residuals.
    #larger k values for largerresiduals.
    # sign is if the input data is signed or unsigen
    def __init__(self, k, sign):
        self.k = k
        self.sign = sign

    #n is the value to encode in Rice code
    def Encode(self, n):
        #If the input data is signed the signed bit needs to
    be saved.
        #In case the input data is negative it is converted to
     positive beffore encoding.
        if self.sign:
            if n < 0:
                s = "1"
                n = -n
            else:
                s = "0"


        #If statments checks if n = 0, this should be handled
    the same way as if n < k (see below),
```

```
        #but since math-log(0,2) does not work an extra if
statement is needed
        if n == 0:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there are k
bits representing the value n
            r = np.binary_repr(n,self.k)
            #The loop constant will detemen how long the unary
 code will be.
            #When n is fully represented in binary by r there
is no need for unary code.
            loop = 0

        #If statement checks that n value is large enough to
be longer than k bits when converted to binary.
        #This is done so when separating the k last bits it
would create an error if there is not enoguh bits.
        elif math.log(n,2) < self.k:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there are k
bits representing the value n
            r = np.binary_repr(n,self.k)
            #The loop constant will detemen how long the unary
 code will be.
            #When n is fully represented in binary by r there
is no need for unary code.
            loop = 0
        else:
            #Convert n to a binary value
            n = bin(n)[2:]
            #Takes the k last bits of n and saves it in
variable r
            r = n[len(n)-self.k:]
            #The remaining bits in n are converted to their
interger value.
            #This interger value will be unary coded
            loop = int(n[:len(n)-self.k],2)

        #Append a "0" bit as MSB in the code word r.
```

```
        #This will later show where the unary code ends.
        r = "0" + r


        #for loop with "loop" number of iteration.
        #Each iteration adds a "1" as MSB to r,
        #This way the unary code is created.
        for i in range(loop):
            r = "1" + r

        #In the case for signed data the sign bit is added to
r.
        if self.sign:
            r = s + r

        #Returns the code word in r
        return r

    def Decode(self, code):


        decoded_values = []

        #Loops trough the code word to get all values in the
code word
        while len(code) > 0:
            A = 0
            #If S is true the output should be negative
            S = False
            value = ""

            #Checks if the data is signed
            if self.sign:
                #Checks if the MSB is "1", indicating that the
 value should be negative.
                if code[0] == "1":
                    #Sets S to True if the output should be
negative
                    S = True
                #Removes the MSB, signed bit, from the code
word.
```

```
                #This way the rest of the code word can be
handled as if it was unsigned
                code = code[1:]


            #Decodes the unary code.
            #Unary code represents a interger value as a set
of "1":s followed by a "0".
            #By incrementing A by 1 if the MSB is a one then
moving the codeword one step and checking again
            #all "1" will increment A until a "0" is MSB
indicating that the unary code have been decoded.
            while code[0] == "1":
                A += 1
                code = code[1:]


            #After the unary code have been uncoded the
remaining code will be
            #a "0" as MSB followed by the last k-bits from the
 original binary value as LSB
            #By looping thorugh the codeword and removing k
bits to the current value being decoded
            #The last bits in the code word for the current
value can be decoded
            for j in range(self.k):
                code = code[1:]
                value += code[0]
            #Since there is a 0 separating the MSB with the
LSB 1 more bit needs to be removed for the current value in
 the codeword
            code = code[1:]

            value = bin(A)[2:] + value

            #The original binary value is converted to an int
            value = int(value, 2)

            #In if the original value was negative sign and S
will be true,
            #code will then be converted to a negative value
```

```
            if self.sign and S:
                value = -value

            #appends the decoded value in an array

            decoded_values.append(value)

        #Returns array with decoded int values
        return decoded_values
```

A.2.3 Shorten class code

```
#Takes data as input
#order of predicuter as order
#memory as the last sampels beffore the input data
#memory should be the length of order
#if memory is the start of the data stream it should be an
   array of zeros
class Shorten:

    def __init__(self, order: int = 3):
        #Order can only be between 0 and 3 for Shorten, and
   have to be an int
        if not (0 <= order <= 3) or not isinstance(order, int)
   :
            raise ValueError(f"Order can only have integer
   values between 0 and 3. Current value of is {order}")

        self.order = order
        #Coeficents for differente orders, from 0 to 3
        all_coeficents = [[0],[1],[2, -1],[3, -3, 1]]
        #Assigns the coeificents to be used based on order
        self.coeficents = all_coeficents[order]

    def predict(self, memory):
        prediction = 0

        if self.order > 1:
            #this forloop is reversed so that memory values
   needed are not overwritten when values are shuffled one
```

```
index back
            for j in reversed(range(1, self.order)):
                prediction = prediction + self.coeficents[j] *
 memory[j]
                memory[j] = memory[j-1]

            prediction = prediction + self.coeficents[0] *
memory[0]

        elif self.order == 1:
            #in case of order 1 the current prediction is the
last input
            prediction = memory[0]

        elif self.order == 0:
            #in case of order zero the prediciton is allways
zero.
            #in this case the full current value is sent as
residual
            prediction = 0

        else:
            raise ValueError(f"Order can only have integer
values between 0 and 3. Current value of is {self.order}")
        return prediction, memory


    def In(self, input, memory: list = [0] * 3):




        #raises error if memory is not the length of order
        if self.order != len(memory):
            raise ValueError(f"Order and memory length should
match. Current values are: order = {self.order}, memory
length = {len(memory)}")

        predictions = []#only needed for testing?
        residuals = []
```

```
        #loops through the input array
        for i in range(len(input)):
            prediction, memory = self.predict(memory)


            #calculates residual by taking the current value
and subtract the predicted value
            residual = input[i] - prediction

            #update the first memory slot with the current
input value
            #This should not be done for order 0 since that
have an empty memory
            if self.order > 0:
                memory[0] = input[i]


            #saves residuals and prediction in their
respective array
            predictions.append(prediction)
            residuals.append(residual)

        #returns residual. memory. and prediction array
        #The memory array can then be used for the next set of
 data inputs if the come in blocks
        return residuals, memory, predictions


    def Out(self, residuals, memory: list = [0] * 3):

        predictions = []#only needed for testing?
        input = []

        #loops through the residuals array
        for i in range(len(residuals)):
            prediction, memory = self.predict(memory)


            #calculates original input by taking the current
```

```
    residual and adding the predicted value
            current_input = residuals[i] + prediction

            #update the first memory slot with the current
    input value
            #This should not be done for order 0 since that
    have an empty memory
            if self.order > 0:
                memory[0] = current_input

            #saves residuals and prediction in their
    respective array
            input.append(current_input)
            predictions.append(prediction)

        return input, memory, predictions
```

A.2.4 LPC class code

```
import numpy as np
import math


class MetaLPC:

    def __init__(self, order, sign = True,
    CoefficentDecimalBits: int = 10):
        #Order have to be an integer of size 1 or larger for
    LPC
        #No upperlimit exist other than when the order is
    larger than the number of inputs the lag when caluclating
    autocorrelation will raise an error
        if  order < 0 or not isinstance(order, int):
            raise ValueError(f"Order can only have integer
    values larger than 0. Current value of is {order}")

        self.order = order
        self.sign = sign
        self.CoefDecBit = CoefficentDecimalBits
```

```
        self.CoeffDivisionFactor = pow(2,CoefficentDecimalBits
) - 1



     #Function to calculate the autocorrelation of the input
values
     def autocorrelation(self, x, lag):

        if lag >= len(x):
            raise ValueError(f"Lag must be shorter than array
length")
        if not isinstance(lag, int) or lag < 0:
            raise ValueError(f"Lag must be an positive int")




        x_mean = np.mean(x)
        n = 0
        t = 0


        for i in range(len(x)):
            n += pow(x[i] - x_mean, 2)
            if i >= lag:
                t += (x[i] - x_mean) * (x[i-lag] - x_mean)


        #In the case where all x values are identical to each
other the equation for autocorrelation dont work
        #This is beceause n will be equal to 0 and this can
not be used to dicide t
        #In the case that all values are identical the
autocorrelation should be =1 regardless of lag-value
        if n == 0:
            if lag == 0:
                t = 1
            else:
                t = 0
```

```
            n = 1



        return t/n


    #Calculates the coefficents for LPC using the Levinson-
Durbin algorithm
    #The coefficents can be calculated with matrix
multiplications
    #How ever it is faster to use this algorithm
    def Coefficents(self, inputs):

        #If all inputs are constant a division by 0 error will
    occur
        #To handle this this if statement is created.
        #Since all values are exactly the same it is enough to
    predict the next value with the previous value
        #hence the first cofficent is a 1 and the rest are 0
        if all(element == inputs[0] for element in inputs):
            a = [1] + [0] * (self.order - 1)

        else:
            E = self.autocorrelation(inputs, 0)
            a = []

            for i in range(self.order):

                k = self.autocorrelation(inputs, i+1)
                if i > 0:
                    for j in range(i):
                        k -= a[j] * self.autocorrelation(
inputs, i-j)

                k = k / E
```

```
                a.append(k)

                if i > 0:
                    a_old = a.copy()
                    for j in range(i):
                        a[j] = a_old[j] - k * a_old[(i-1)-j]
                E = (1 - pow(k,2)) * E









        return a


    #Calculates the prediction of the current value using the
cofficents and earlier values
    def prediciton(self, memory, coef):
        prediction = 0

        for j in reversed(range(1, self.order)):
            prediction += coef[j] * memory[j]
            memory[j] = memory[j-1]#updates the memory with
the earlier values taking one step in the memory array


        prediction += coef[0] * memory[0]

        return prediction, memory


    def In(self, inputs, memory):

        coef = self.Coefficents(inputs)



        residuals = []
```

```
        for i in range(len(inputs)):
            predict, memory = self.prediciton(memory, coef)

            memory[0] = inputs[i]#Updates the first slot in
the memory array with the current value
            residual = inputs[i] - predict#caltulcates the
residual

            residual = round(residual)#Since Rice and Golomb
codes need to have interger the residual is rounded to
closest int


            #Append current prediction and residual in array
to returned
            residuals.append(residual)


        binaryCoefficents = self.CoefEncode(coef.copy())

        idealK, binaryIdealK = self.kCalculator(residuals.copy
())

        CodeWord = self.RiceEncode(residuals, idealK)

        FullCodeWord = binaryIdealK + binaryCoefficents +
CodeWord




        return FullCodeWord, memory, binaryCoefficents


    def kCalculator(self, residuals):
        #Calculates the ideal k_vaule
        abs_res = np.absolute(residuals.copy())
        abs_res_avg = np.mean(abs_res)
        #if abs_res_avg is less than 4.7 it would give a k
value less than 1.
        #k needs tobe a int > 1. All abs_res_avg values bellow
```

```
 6.64 will be set to 1 to avoid this issue
      if abs_res_avg > 6.64:
          #from testing it appears that the actual ideal k-
value is larger by +1 than theory suggest,
      #atleast for larger k-value. The exact limit is
unknown but it have been true for all test except for when
the lowest k, k =1 is best.
      #Therefore th formula have been modified to increment
k by 1 if abs_res_avg > 6.64.
          k = int(round(math.log(math.log(2,10) *
abs_res_avg,2))) + 1
      else:
          k = 1

      #Binary represent k in 5 bits for metadata to be sent
      #This allows for values 0-32, since k never is less
than 1 it is possible to shift down k by 1 beffore encoding
,
      #allowing for k values 1-33 to be encoded

      binaryK = np.binary_repr(k-1,5)

      return k, binaryK


  def RiceEncode(self, residuals, k):
      CodeWord = ""

      for n in residuals:
          #n is the value to encode in Rice code

          #If the input data is signed the signed bit needs
to be saved.
          #In case the input data is negative it is
converted to positive beffore encoding.
          if self.sign:
              if n < 0:
                  s = "1"
                  n = -n
              else:
                  s = "0"
```

```
        #If statments checks if n = 0, this should be
handled the same way as if n < k (see below),
        #but since math-log(0,2) does not work an extra if
 statement is needed
        if n == 0:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there
are k bits representing the value n
            r = np.binary_repr(n,k)
            #The loop constant will detemen how long the
unary code will be.
            #When n is fully represented in binary by r
there is no need for unary code.
            loop = 0

        #If statement checks that n value is large enough
to be longer than k bits when converted to binary.
        #This is done so when separating the k last bits
it would create an error if there is not enoguh bits.
        elif math.log(n,2) < k:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there
are k bits representing the value n
            r = np.binary_repr(n,k)
            #The loop constant will detemen how long the
unary code will be.
            #When n is fully represented in binary by r
there is no need for unary code.
            loop = 0
        else:
            #Convert n to a binary value
            n = bin(n)[2:]
            #Takes the k last bits of n and saves it in
variable r
            r = n[len(n)-k:]
            #The remaining bits in n are converted to
their interger value.
```

```
                #This interger value will be unary coded
                loop = int(n[:len(n)-k],2)

            #Append a "0" bit as MSB in the code word r.
            #This will later show where the unary code ends.
            r = "0" + r


            #for loop with "loop" number of iteration.
            #Each iteration adds a "1" as MSB to r,
            #This way the unary code is created.
            for i in range(loop):
                r = "1" + r

            #In the case for signed data the sign bit is added
 to r.
            if self.sign:
                r = s + r

            #add r to the total codeword
            CodeWord += r

        return CodeWord


    #Enode the coefficents in binary code
    def CoefEncode(self, coefficent):
        #The cofficents are float numbers, usualy between -1
and 1.
        #To encode them the cofficents are multiplied by 1023
and rounded down to closest int,
        #That int is then written in binary.
        #baseline is to write this int in 10 bits binary and
have 1 extra bit for sign bit.
        #This is enough when the coefficents are between -1
and 1, but if any coefficents are smaller/larger than this
extra bits are needed.
        #The first bits in the Binary coefficents are run
length encoded for how many extra bits are needed.
        #If the largest coefficents is 3.5 for example,
multiplying this with 1023 and round down to closest in
```

```
will get 3580 which needs 12 bits to be written,
     #that is 2 extra bits. It is the same amount of bits
needed to write 3.5 rounded down to closest int.


     #The basline for how many bits should be used for
decimal can be changed in the function by changing:
     #CoefficentDecimalBits in innit

     BinaryCoefficents = ""
     if np.max(np.absolute(coefficent.copy())) > 1:

         extraBits = np.max(np.absolute(coefficent.copy()))
         extraBits = math.ceil(math.log(extraBits,2))
     else:
         extraBits = 0



     #Run length encode how many extra bits are needed
     for RunLengthEncode in range(extraBits):
         BinaryCoefficents += "1"
     BinaryCoefficents += "0"



     #Encode each coefficent in binary
     #loop thorugh all coefficents
     for i in range(self.order):
         currentCoefficent = coefficent[i]
         #assign sign bit
         if currentCoefficent < 0:
             CurrentBinaryCoef = "1"
             currentCoefficent = -currentCoefficent
         else:
             CurrentBinaryCoef = "0"
         #Multipli coeffcient by 1023 and round down to
closest int (1023 is standard value)
         currentCoefficent = int(currentCoefficent * self.
CoeffDivisionFactor)
```

```
            #current coefficent should never be able to be
larger than 2 ^ (10 + extraBits)
            #If it is it will not be possible to encode it
            if currentCoefficent > pow(2,self.CoefDecBit +
extraBits):
                raise ValueError(f"current coefficent, {
currentCoefficent} is larger than 2^{self.CoefDecBit+
extraBits}")


            #Write the currentCoefficent value in binary, with
 10 + extraBits lenght
            CurrentBinaryCoef +=  np.binary_repr(
currentCoefficent, self.CoefDecBit + extraBits)

            #Add the current coefficent in binary to the full
code word
            BinaryCoefficents += CurrentBinaryCoef

        return BinaryCoefficents


    def coefDecode(self, codeword):
        extraBits = 0
        coefficents = []

        #Recreate the extra bits from the rle code
        while codeword[0] == "1":
            extraBits +=1

            codeword = codeword[1:]

        codeword = codeword[1:]

        #loop thorugh the codeword and take out the bits
representing a cofficents
        #this is done for every coefficents needed (number of
coefficents = order)
        for NumberOfCoefficents in range(self.order):
            if codeword[0] == "1":
                IsNegative = True
```

```
            else:
                IsNegative = False

            codeword = codeword[1:]

            CurrentCoefficentBinary = codeword[:self.
CoefDecBit+extraBits]
            codeword = codeword[self.CoefDecBit+extraBits:]



            CurrentCoefficent = int(CurrentCoefficentBinary,
2)

            if IsNegative:
                CurrentCoefficent = -CurrentCoefficent

            coefficents.append(CurrentCoefficent / self.
CoeffDivisionFactor)

        if len(coefficents) != self.order:
            raise ValueError(f"Number of coefficents should
match order, decoded coefficents are = {coefficents}")

        return coefficents, codeword


    def Out(self, codeword, memory):
        RecreatedValues = []

        kValue, codeword = self.kDecode(codeword)



        coef, codeword = self.coefDecode(codeword)



        residuals = self.RiceDecode(codeword, kValue)
```

```
        for i in range(len(residuals)):
            predict, memory = self.prediciton(memory, coef)


            #Recreates the current input
            #This maybe should also use round as in LPC.In
            #since the values in this project to be recreated
always are int
            currentValue = residuals[i] + predict

            memory[0] = currentValue#Uppdates the first memory
 slot with the recreated input


            #Appends the current recreated value to be
returned
            RecreatedValues.append(currentValue)

        return RecreatedValues, memory

    def kDecode(self, codeword):
        #5 first bits in the codeword represent the k value
used to encode the residuals
        kBinary = codeword[:5]
        codeword = codeword[5:]

        #Convert k from binary to int
        #Increment k by 1 because it is shifted down by 1 when
 encoded
        k = int(kBinary,2) + 1

        return k, codeword

    def RiceDecode(self, code, k):


        decoded_values = []

        #Loops trough the code word to get all values in the
```

```
code word
      while len(code) > 0:


          A = 0
          #If S is true the output should be negative
          S = False
          value = ""

          #Checks if the data is signed
          if self.sign:
              #Checks if the MSB is "1", indicating that the
 value should be negative.
              if code[0] == "1":
                  #Sets S to True if the output should be
negative
                  S = True
              #Removes the MSB, signed bit, from the code
word.
              #This way the rest of the code word can be
handled as if it was unsigned
              code = code[1:]


          #Decodes the unary code.
          #Unary code represents a interger value as a set
of "1":s followed by a "0".
          #By incrementing A by 1 if the MSB is a one then
moving the codeword one step and checking again
          #all "1" will increment A until a "0" is MSB
indicating that the unary code have been decoded.
          while code[0] == "1":
              A += 1
              code = code[1:]


          #After the unary code have been uncoded the
remaining code will be
          #a "0" as MSB followed by the last k-bits from the
 original binary value as LSB
          #By looping thorugh the codeword and removing k
```

```
   bits to the current value being decoded
            #The last bits in the code word for the current
   value can be decoded
            for j in range(k):
                code = code[1:]
                value += code[0]
            #Since there is a 0 separating the MSB with the
   LSB 1 more bit needs to be removed for the current value in
    the codeword
            code = code[1:]

            value = bin(A)[2:] + value

            #The original binary value is converted to an int
            value = int(value, 2)

            #In if the original value was negative sign and S
   will be true,
            #code will then be converted to a negative value
            if self.sign and S:
                value = -value

            #appends the decoded value in an array
            decoded_values.append(value)

        #Returns array with decoded int values
        return decoded_values
```

A.2.5 FLAC class code

```
import numpy as np
import math
from Rice import RiceCoding



class MetaFLAC:

    def __init__(self, LpcOrder: int = 32,
   CoefficentDecimalBits: int = 10, sign = True):
```

```
        self.LpcOrder = LpcOrder
        self.ShortCoff = [[0],[1],[2, -1],[3, -3,
1],[4,-6,4,-1]]
        self.CoefDecBit = CoefficentDecimalBits
        self.CoeffDivisionFactor = pow(2,CoefficentDecimalBits
) - 1
        self.sign = sign


     #Function to calculate the autocorrelation of the input
values
     def autocorrelation(self, x, lag):

        if lag >= len(x):
            raise ValueError(f"Lag must be shorter than array
length")
        if not isinstance(lag, int) or lag < 0:
            raise ValueError(f"Lag must be an positive int")

        x_mean = np.mean(x)
        n = 0
        t = 0

        for i in range(len(x)):
            n += pow(x[i] - x_mean, 2)
            if i >= lag:
                t += (x[i] - x_mean) * (x[i-lag] - x_mean)

        #In the case where all x values are identical to each
other the equation for autocorrelation dont work
        #This is beceause n will be equal to 0 and this can
not be used to dicide t
        #In the case that all values are identical the
autocorrelation should be =1 regardless of lag-value
        if n == 0:
            if lag == 0:
                t = 1
            else:
                t = 0
            n = 1
```

```
        return t/n


    #Calculates the coefficents for LPC using the Levinson-
Durbin algorithm
    #The coefficents can be calculated with matrix
multiplications
    #How ever it is faster to use this algorithm
    def LpcCoefficentsCalc(self, inputs):

        E = self.autocorrelation(inputs, 0)
        a = []
        CoefficentsArray = []

        #Check if all the inputs are the same, this will give
a divison by zero error when using levensin durbin
algorithm
        if all(element == inputs[0] for element in inputs):
            AllSame = True
        else:
            AllSame = False

        for i in range(self.LpcOrder):

            #To avoid division by zero this if statement is
implemented
            #Since this edge case only handels when all inputs
 are the same the first cofficents is set to a 1
            #and the rest is set to 0
            if AllSame == True:
                a = [1] + [0] * i

            else:
                k = self.autocorrelation(inputs, i+1)
                if i > 0:
                    for j in range(i):
                        k -= a[j] * self.autocorrelation(
inputs, i-j)

                k = k / E
                a.append(k)
```

```
            if i > 0:
                a_old = a.copy()
                for j in range(i):
                    a[j] = a_old[j] - k * a_old[(i-1)-j]
            E = (1 - pow(k,2)) * E
        CoefficentsArray.append(a.copy())

    return CoefficentsArray

def LpcCoefficentsBinaryIn(self, LpcCoefficentArray):

    BinaryCoefficentArray = []
    for c in range(len(LpcCoefficentArray)):

        coef = LpcCoefficentArray[c]
        BinaryCoefficents = ""
        if np.max(np.absolute(coef.copy())) > 1:

            extraBits = np.max(np.absolute(coef.copy()))
            extraBits = math.ceil(math.log(extraBits,2))
        else:
            extraBits = 0

        #Run length encode how many extra bits are needed
        for RunLengthEncode in range(extraBits):
            BinaryCoefficents += "1"
        BinaryCoefficents += "0"

        #Encode each coefficent in binary
        #loop thorugh all coefficents
        for i in range(len(coef)):
            currentCoefficent = coef[i]
            #assign sign bit
            if currentCoefficent < 0:
                CurrentBinaryCoef = "1"
                currentCoefficent = -currentCoefficent
            else:
                CurrentBinaryCoef = "0"
            #Multipli coeffcient by 1023 and round down to
closest int (1023 is standard value)
```

```
                currentCoefficent = int(currentCoefficent *
self.CoeffDivisionFactor)

                #current coefficent should never be able to be
 larger than 2 ^ (10 + extraBits)
                #If it is it will not be possible to encode it
                if currentCoefficent > pow(2,self.CoefDecBit +
 extraBits):
                    raise ValueError(f"current coefficent, {
currentCoefficent} is larger than 2^{self.CoefDecBit+
extraBits}")


                #Write the currentCoefficent value in binary,
with CoefDecBit + extraBits lenght
                CurrentBinaryCoef +=  np.binary_repr(
currentCoefficent, self.CoefDecBit + extraBits)

                #Add the current coefficent in binary to the
full code word
                BinaryCoefficents += CurrentBinaryCoef

            BinaryCoefficentArray.append(BinaryCoefficents)



        return BinaryCoefficentArray


    def ResidualCalculation(self, inputs, memory):
        LpcCoff = self.LpcCoefficentsCalc(inputs)

        ShortPredcitions = [[],[],[],[],[]]
        ShortResiduals = [[],[],[],[],[]]

        RleCode = ""
        RleCount = 0

        LpcPredictions = []
        LpcResiduals = []
        for i in range(self.LpcOrder):
```

```
            LpcPredictions.append([])
            LpcResiduals.append([])

        for i in range(len(inputs)):

            #RLE predicitons, should start once i > 0 so that
memory have been updated once
            if i > 0:
                #If the next imput is the same as the earlier
the RLE count increases by 1
                if inputs[i] == memory[0]:
                    RleCount += 1
                #If the next input is not the same as the
first one in memory it means that a new value have arrived
at input
                #The old value is then encoded with its RLE
count
                else:
                    #Save the encoded Rle value in the RleCode
 variable
                    RleCode += self.RleEnconder(memory[0],
RleCount)
                    #Reset the Rle Counter
                    RleCount = 0

            #calculates the current prediction for all orders
of shorten and LPC, aswell as steps the memory array
            shortCurrentPrediciton, LpcCurrentPrediction,
memory = self.prediction(memory, LpcCoff)

            #Replace the first spot in the memory array with
the current input
            memory[0] = inputs[i]

            #Calculates the current residual for all orders of
 shorten
            #By looping through all the calculated predictions
 by different orders for the current input
            #and subtracting the value from the current input
and saving the ressidual in an array
            for j in range(len(shortCurrentPrediciton)):
```

```
                ShortCurrentResidual = inputs[i] -
shortCurrentPrediciton[j]
                ShortResiduals[j].append(ShortCurrentResidual)
                ShortPredcitions[j].append(
shortCurrentPrediciton[j])

            #The same is done for all orders of LPC
            for j in range(len(LpcCurrentPrediction)):
                LpcCurrentResidual = round(inputs[i] -
LpcCurrentPrediction[j])
                LpcResiduals[j].append(LpcCurrentResidual)
                LpcPredictions[j].append(LpcCurrentPrediction[
j])

        #The last value needs to be RLE encoded
        #This is done by calling the RLE function
        #Save the encoded Rle value in the RleCode variable
        RleCode += self.RleEnconder(memory[0], RleCount)
        #Reset the Rle Counter
        RleCount = 0

        LpcBinaryCoefficents = self.LpcCoefficentsBinaryIn(
LpcCoff.copy())

        return RleCode, ShortResiduals, LpcResiduals, memory,
LpcBinaryCoefficents, LpcCoff, ShortPredcitions,
LpcPredictions


    def In(self, inputs, memory):

        RleCode, ShortResiduals, LpcResiduals, memory,
LpcBinaryCoefficents, LpcCoff, SPred, LPred = self.
ResidualCalculation(inputs, memory)


        #Crate an array to store all binary represented values
    for the residuals and RLE
        AllResidualsBinary = []
        k_array = [0]
```

```
        #Append RLE at the first place in the binary
    represntation array
        AllResidualsBinary.append(RleCode)

        #Crate a for - loop and use Rice codes to append all
    Shorten residuals
        for i in range(len(ShortResiduals)):
            #Calculates ideal k value for current residuals
    and RIce codes and saves them in array
            CurrentCodeWord, ideal_k = self.FlacRice(
    ShortResiduals[i])
            AllResidualsBinary.append(CurrentCodeWord)
            k_array.append(ideal_k)

        #Crate a for loop and use Rice codes to append all LPC
     residuals
        for i in range(len(LpcResiduals)):
            #Calculates ideal k value for current residuals
    and RIce codes and saves them in array
            CurrentCodeWord, ideal_k = self.FlacRice(
    LpcResiduals[i])

            #Add the coefficents for current lpc to the
    codeword
            CurrentCodeWord = LpcBinaryCoefficents[i] +
    CurrentCodeWord
            AllResidualsBinary.append(CurrentCodeWord)
            k_array.append(ideal_k)

        codeChoice = 0
        k_Choice = 0
        valueChoice = AllResidualsBinary[0]
        for i in range(1, len(AllResidualsBinary)):
            #If specific compression algorithm needs to be
    tested it can be done with an if statement here
            #Example, for test RLE only add the if statment "
    if i < 1:"
            if len(AllResidualsBinary[i]) < len(valueChoice):
                valueChoice = AllResidualsBinary[i]
                codeChoice = i
                k_Choice = k_array[i]
```

```
        #Only need to return the relevant LpcCofficents
        #If RLE or Shorten have been used the cofficents are
just an empty array
        LpcCoffChoosen = []
        #If LPC have been choosen the relevant cofficents are
saved in the LpcCoffChoosen variabel
        #LPC order 1 is equal to codeChoice = 6 therefore the
if statement checks if codeChoice is larger than 6
        if codeChoice > 5:
            LpcCoffChoosen = LpcCoff[codeChoice-6]



        #k is encoded in 5 bits, allowing for values from
0-32, since k can never be less than 1
        #k is decremented by 1 beffore enocding it. Allowing
for k values in between: 1 <= k <= 33
        if 33 < k_Choice :
            raise ValueError(f"k can not be represented in 5
bits binary, max value for k is 32 and current value is {
k_Choice}")

        k_Choice = k_Choice - 1


        #Add the code choice to get final codeword
        FinalCodeWord = np.binary_repr(codeChoice,6) +
valueChoice



        return FinalCodeWord, memory


    def LpcCoefDecode(self, codeword, CurrentLpcOrder):
        extraBits = 0
        coefficents = []

        #Recreate the extra bits from the rle code
        while codeword[0] == "1":
```

```
            extraBits +=1

            codeword = codeword[1:]

        codeword = codeword[1:]

        #loop thorugh the codeword and take out the bits
representing a cofficents
        #this is done for every coefficents needed (number of
coefficents = order)
        for NumberOfCoefficents in range(CurrentLpcOrder):
            if codeword[0] == "1":
                IsNegative = True
            else:
                IsNegative = False

            codeword = codeword[1:]

            CurrentCoefficentBinary = codeword[:self.
CoefDecBit+extraBits]
            codeword = codeword[self.CoefDecBit+extraBits:]



            CurrentCoefficent = int(CurrentCoefficentBinary,
2)

            if IsNegative:
                CurrentCoefficent = -CurrentCoefficent

            coefficents.append(CurrentCoefficent / self.
CoeffDivisionFactor)

        if len(coefficents) != CurrentLpcOrder:
            raise ValueError(f"Number of coefficents should
match order, decoded coefficents are = {coefficents}")

        return coefficents, codeword
```

```
 def Out(self, code_residuals, memory):
     DecodedValues = []#Array to save decoded values


     #the first 6 bits in the codeword indicate what
encoder have been used
     code_choose_binary = code_residuals[:6]
     code_residuals = code_residuals[6:]
     code_choose = int(code_choose_binary,2)

     #If code_chosse is 0 the residual is encoded using RLE
     if code_choose == 0:

         #Loops trough all the enocded binary values with a
 while llop
         while len(code_residuals) > 0:
             #Raise error if there is not enough length
left of code_residuals for 1 code_word
             if len(code_residuals) < 33:
                 raise ValueError(f"Code_word length is {
len(code_residuals)}, should  never be lower than 33")


             #The first bit in each code word is the sign
bit
             #This is saved as s and then the
code_residuals is stepped one step
             s = code_residuals[0]
             code_residuals = code_residuals[1:]

             #The next 24 bits represents the value that
have been encoded
             #Once they been saved 24 steps trough the
code_residuals are taken
             RleValue = code_residuals[:24]
             code_residuals = code_residuals[24:]

             #The next 8 bits represent how many times the
value was repeated
```

```
                #Once it been saved 8 steps is taken in the
code_residuals
                RleCount = code_residuals[:8]
                code_residuals = code_residuals[8:]

                #The value that have been encoded is converted
 to int
                IntRelValue = int(RleValue, 2)

                #If the sign bit is "1" the value is converted
 to negative
                if s == "1":
                    IntRelValue = -IntRelValue

                #A for loop is used to append as many values
as needed
                #The RleCount value is increased by 1 because
even though
                #a value never was reapeted it occured once
                for i in range(1 + int(RleCount, 2)):
                    DecodedValues.append(IntRelValue)

            #Uppdate memory
            #Copy the decoded values to assign them to the
memory array later
            RleCopy = DecodedValues.copy()

            #Assign the last values of the decoded values to
the memory array
            #Important that the size of the memory array stays
 the same
            if len(RleCopy) > len(memory):
                new_memory = RleCopy[len(RleCopy)-len(memory)
:]
            #In the case that there is to few values among the
 decoded values pad the memory array with 0:s
            else:
                new_memory = RleCopy
                while len(new_memory) < len(memory):
                    new_memory.append(0)
```

```
            #Assign the new_memory to memory
            memory = new_memory


        else:



            #If code choice is larger than 5 LPC have been
used to encode the residuals,
            #the first bits in the codewords are then the
coffiecnts used in LPC
            if code_choose > 5:
                #The LPC order will be 5 less than code_choose
 since code_choice 0 is for RLE and 1-5 is for Shorten
                CurrentLpcOrder = code_choose - 5
                #Decode the first bits that represent the LPC
cofficents,
                #The number of cofficents are equal to
CurrentLpcOrder
                LpcCoff, code_residuals = self.LpcCoefDecode(
code_residuals, CurrentLpcOrder)


            #Else if code_choose is not > 5 shorten have been
used to encode it
            else:
                #The Shorten order will be 1 less than
code_choose since code_choice 0 is for RLE
                ShortenOrder = code_choose - 1

            #The next 5 MSB in the codeword represnt the k-
value used to encode the residuals using Rice codes
            k_binary = code_residuals[:5]
            code_residuals = code_residuals[5:]
            k_value = int(k_binary,2) + 1




            #In cases where RLE encoding is not used, the
```

```
resiudals have been encoded using Rice codes
            #Decodes the Residuals with Rice.Decode
            Rice_decoder = RiceCoding(k_value, self.sign)
            Residuals = Rice_decoder.Decode(code_residuals)


            #loops thorugh all the residuals to calculate the
original input value
            for i in range(len(Residuals)):
                #If code_choose is 1 to 5 Shorten have been
used by FLAC
                if code_choose < 6:


                    #If shorten order does not match length of
 memeory array error should be raised


                    #If Shorten order is 0 the prediciton is
allways 0
                    if ShortenOrder == 0:
                        current_prediciton = 0
                    #In other cases the current predicition is
 calculated by multiplying the orders cofficents with the
memory array
                    else:
                        current_prediciton = sum( np.array(
self.ShortCoff[ShortenOrder]) * np.array(memory[:
ShortenOrder]) )

                #If the code_choose is larger than 5 LPC have
been used by FLAC
                else:

                    #Calculated the current_prediciton by
multiplying the LpcCoff with the memory
                    current_prediciton = sum( np.array(LpcCoff
) * np.array(memory[:CurrentLpcOrder]) )

                #Calculating current input by taking the
prediciton and adding the current residual
```

```
                current_input = current_prediciton + Residuals
[i]

                #Uppdating memory by stepping all slots one
step back and putting the current_input in the first slot
                memory = [current_input] + memory[:-1]

                #Append the decoded input value
                DecodedValues.append(current_input)

        #Return the decoded values and memory array
        return DecodedValues, memory, code_choose


    def RleEnconder(self, RleValue, RleCounter):
        #First the signed dignit is set, 0 for positive and 1
for negative
        if RleValue < 0:
            s = "1"
            #if the value was negative it is turned to
positive beffore saved as value to be RLE encoded
            Rle_value = -RleValue
        else:
            s = "0"
            Rle_value = RleValue

        #The RLE value is written as a 24 bit representation
since this is the largest that can be sampled by the system
        RleValueBinary = np.binary_repr(Rle_value, 24)
        #The RLE count is written as 8 bit value
        #The current block size is 256, which then is the max
amount of identical sampels that can come in a row for the
same block
        #It is important to note that what is RLE encoded is
the amount of repitions.
        #So that if RLE count is encoded as 0 that means that
the value occours once and then is repeated 0 times.
        RleCountBinary = np.binary_repr(RleCounter, 8)
```

```
        #Completes the current binary representation of the
RLE
        RleBinary = s + RleValueBinary + RleCountBinary

        #Return the binary representation of the RLE
        return RleBinary


    def FlacRice(self, current_residuals):
        #calculates the ideal k-value for the current
residuals
        abs_res = np.absolute(current_residuals)
        abs_res_avg = np.mean(abs_res)

        #k needs tobe a int > 1. abs_res_avg = 6.64 gives k =
1.
        #All k values for abs_res_avg above 6.64 is calculated
 using modified Rice Theory
        if abs_res_avg > 6.64:
            k_ideal = int(round(math.log(math.log(2,10) *
abs_res_avg,2))) + 1
        else:
            k_ideal = 1

        if k_ideal < 1 :
            raise ValueError(f"k can not be less than 1,
current value is {k_ideal}")


        kBinary = k_ideal - 1
        if 1 > kBinary or kBinary > 32:
            raise ValueError(f"kBinary have to be larger or
equal to 1 and less or equal to 32, current value is {
kBinary}")

        #Represent the k value in 5 bits
        kBinary = np.binary_repr(kBinary,5)

        #calculates the Rice code word for the residual
        code_word =kBinary
        for q in range(len(current_residuals)):
```

```
            Rice_coder = RiceCoding(k_ideal, self.sign)
            n = int(current_residuals[q])
            kodOrd = Rice_coder.Encode(n)
            code_word += kodOrd

        return code_word, k_ideal


    def prediction(self, memory, LpcCoff):
        ShortCurrentPredcition = [0]*5
        LpcCurrentPrediciton = [0]*self.LpcOrder
        new_memory = [0]


        for i in range(len(memory)):
            if i <= 3:
                ShortCurrentPredcition[i+1] = sum( np.array(
    memory[:i+1]) * np.array(self.ShortCoff[i+1]) )

            if i < self.LpcOrder:
                LpcCurrentPrediciton[i] = sum( np.array(memory
    [:i+1]) * np.array(LpcCoff[i]) )

        new_memory += memory[:-1]

        return ShortCurrentPredcition, LpcCurrentPrediciton,
    new_memory
```

A.2.6 Adjacent class code

```
import numpy as np

#Predcits mic value using value from previous mic

class Adjacent:

    def __init__(self, order):
        self.order = order
        #Coeficents for differente orders, from 0 to 4
```

```
        all_coeficents = [[0],[1],[2, -1],[3, -3,
1],[4,-6,4,-1]]
        #Assigns the coeificents to be used based on order
        self.coeficents = all_coeficents[order]


    def FirstIn(self, firstInput, memory):
        firstPrediction = 0

        if self.order > 0:
            firstPrediction = sum( np.array(self.coeficents) *
 np.array(memory) )

        firstResidual = firstInput - firstPrediction

        if self.order > 0:
            memory = [firstInput] + memory[:-1]


        return firstResidual, memory, firstPrediction

    def In(self, inputs, memory):

        #Use Shorten to calculate the residual for the first
microphone in the array
        firstResidual, memory, firstPrediction = self.FirstIn(
inputs[0], memory)

        #Save the first residual and prediciton in the
residual and prediciton arrays
        Residuals = [firstResidual]
        Predictions = [firstPrediction]


        #Save the first value as the previous value and
previous row value
        #Previous indicates the value of the previous mice
        #PreviousRow indicates the first value of the previous
 row
        #The rest of the resiudals are calculated by taking
the difference between the current mic value and the
```

```
previous mic value
      Previous = inputs[0]
      PreviousRow = inputs[0]
      for i in range(1, len(inputs)):
          #When i modulos 8 is equal to 0 it indicates that
a new row has started (8 mics per row)
          if i % 8 == 0:
              #The previous value is then taken from the
previous row and not the previous mic
              #This will give a mic that is closer to the
current mic
              CurrentResidual, CurrentPrediction = self.
PredicitorIn(inputs[i], PreviousRow)
              #A new PreviousRow value is calculculated
              #This is the first value of the new row
              PreviousRow = inputs[i]
          else:
              CurrentResidual, CurrentPrediction = self.
PredicitorIn(inputs[i], Previous)

          #Update the previous value
          Previous = inputs[i]

          #Save the residual and predicted value in their
respective array
          Residuals.append(CurrentResidual)
          Predictions.append(CurrentPrediction)

      #Return the residuals, memory, and predictions
      return Residuals, memory, Predictions


  def PredicitorIn(self, CurrentInput, PreviousInput):
      #Predicted value is the previous mic value
      Predcition = PreviousInput
      #The residual is the difference of the current mic
value and the previous mic value
      Residual = CurrentInput - Predcition
      #Return the residual and prediciton
      return Residual, Predcition
```

```
 def FirstOut(self, firstResidual, memory):
     firstPrediction = 0

     #If order is larger than 0 calculate prediciton, else
prediciton is always 0
     if self.order > 0:
         firstPrediction = sum( np.array(self.coeficents) *
 np.array(memory) )

     #Recreate the original input by adding the residual
and prediction value
     firstInput = firstResidual + firstPrediction

     #Update memory
     if self.order > 0:
         memory = [firstInput] + memory[:-1]


     return firstInput, memory, firstPrediction


 def Out(self, residuals, memory):
     #Use Shorten to calculate the original input for the
first microphone in the array
     firstInput, memory, firstPrediction = self.FirstOut(
residuals[0], memory)

     #Save the first residual and prediciton in the
residual and prediciton arrays
     Inputs = [firstInput]
     Predictions = [firstPrediction]


     #Save the first value as the previous value and
previous row value
     #Previous indicates the value of the previous mice
     #PreviousRow indicates the first value of the previous
 row
     #The original inputs are calculated unsing previous
mic values and residuals
```

```
        Previous = firstInput
        PreviousRow = firstInput
        for i in range(1, len(residuals)):

            #When i modulos 8 is equal to 0 it indicates that
a new row has started (8 mics per row)
            if i % 8 == 0:
                #The previous value is then taken from the
previous row and not the previous mic
                #This will give a mic that is closer to the
current mic
                CurrentInput, CurrentPrediction = self.
PredicitorOut(residuals[i], PreviousRow)
                #A new PreviousRow value is calculculated
                #This is the first value of the new row
                PreviousRow = CurrentInput
            else:
                CurrentInput, CurrentPrediction = self.
PredicitorOut(residuals[i], Previous)

            #Update previous value
            Previous = CurrentInput

            #Save the recreated input value and predicted
value in their respective array
            Inputs.append(CurrentInput)
            Predictions.append(CurrentPrediction)


        #Return recreated inputs, memory and predictions
        return Inputs, memory, Predictions


    def PredicitorOut(self, CurrentResidual, PreviousInput):
        #Predicted value is the previous mic value
        Predcition = PreviousInput
        #The original input value will be the residual +
previous mic value
        CurrentInput = CurrentResidual + Predcition
        #Return the residual and prediciton
        return CurrentInput, Predcition
```

A.2.7 FLAC Modifed class code

```
import numpy as np
import math

class FlacModified:

    def __init__(self, sign = True, mics: int = 64, samples:
   int = 256, AdjacentOrder: int = 2, ForceEncoder = "None"):
        self.ShortenCofficents = [[0],[1],[2, -1],[3, -3,
   1],[4,-6,4,-1]]
        self.mics = mics
        self.samples = samples
        self.AdjacentOrder = AdjacentOrder
        self.sign = sign
        EncoderForced = -1
        Encoders = ["RLE", "Shorten0", "Shorten1", "Shorten2",
    "Shorten3", "Shorten4", "AdjacentSamples", "AdjacentMics"]
        for i in range(len(Encoders)):
            encoder = Encoders[i]
            if encoder == ForceEncoder:
                EncoderForced = i

        self.EncoderForced = EncoderForced
        #An encoder that is not in the list cant be choosen
        if self.EncoderForced < 0 and ForceEncoder != "None":
            raise ValueError(ForceEncoder," is not a possible
   choice of encoder, options are: ", Encoders)



    def In(self, Inputs, memorysIn):
        AdjacentResiduals = []
        AdjacentFirstPredictions = []
        AdjacentPredictions = []
        CodeWords = []

        for microphone in range(self.mics):
            RleCode = ""
            RleCount = 0
```

```
            currentInputs = Inputs[microphone,:]
            ShortenResiduals = [[],[],[],[],[]]
            ShortenCodeWords = ["","","","",""]
            AdjacentSampleCodeWords = ""
            #ShortenK = [0,0,0,0,0]#Not needed?
            AdjacentResiduals.append([])


            for sample in range(self.samples):
                #RLE calculations:

                #RLE predicitons, should start once sample > 0
 so that memory have been updated once
                if sample > 0:
                    #If the next imput is the same as the
earlier the RLE count increases by 1
                    if currentInputs[sample] == memorysIn[
microphone][0]:
                        RleCount += 1
                    #If the next input is not the same as the
first one in memory it means that a new value have arrived
at input
                    #The old value is then encoded with its
RLE count
                    else:
                        #Save the encoded Rle value in the
RleCode variable
                        RleCode += self.RleEnconder(memorysIn[
microphone][0], RleCount)
                        #Reset the Rle Counter
                        RleCount = 0

                #For the last sample in the datablock the RLE
value needs to be encoded
                if sample == self.samples - 1:
                    RleCode += self.RleEnconder(currentInputs[
sample], RleCount)



                #Shorten calculations:
```

```
            #calculate the predicitons for the current
input for all roders of Shorten, update the memory for all
slots except the first one
            ShortenPredictions, new_memory = self.
ShortenPredictIn(memorysIn[microphone])

            #Calculate the residuals for all orders of
shorten
            for order in range(5):
                ShortenResiduals[order].append(
currentInputs[sample] - ShortenPredictions[order])

            #Uppdate the first slot in the new_memory to
the current sample
            new_memory[0] = currentInputs[sample]
            memorysIn[microphone] = new_memory


            #Adjacent calculations:

            #First prediction for adjacent is saved as the
 first value in the begining of a new row
            #First residual for adjacent in the residual
given by tge choosen order of shorten prediction
            if microphone == 0:
                #CurrentAdjacentResidual = currentInputs[
sample] - ShortenPredictions[order]
                AdjacentResiduals[microphone].append(
currentInputs[sample] - ShortenPredictions[self.
AdjacentOrder])
                AdjacentFirstPredictions.append(
currentInputs[sample])
                AdjacentPredictions.append(currentInputs[
sample])

            else:
                #If microphone % 8 == 0 a new row for
microphones have begun
                if microphone % 8 == 0:
                    #the first value in the row is
```

```
predicted using the previous row first value
                    CurrentAdjacentResidual,
NewAdjacentPrediction = self.AdjacentPredictIn(
currentInputs[sample], AdjacentFirstPredictions[sample])
                    #The first value in the row is then
updated to the value of the first mic in the new row
                    AdjacentFirstPredictions[sample] =
NewAdjacentPrediction

                else:
                    #Predict the current mic value with
the previous mic value
                    CurrentAdjacentResidual,
NewAdjacentPrediction = self.AdjacentPredictIn(
currentInputs[sample], AdjacentPredictions[sample])

                #Save the resiudla
                AdjacentResiduals[microphone].append(
CurrentAdjacentResidual)
                #Update the prediciton for the next mic
                AdjacentPredictions[sample] =
NewAdjacentPrediction

        #Encode the residuals for shorten using RiceCodes
        for order in range(5):
            CurrentShortenResiduals = ShortenResiduals[
order]

            #Calculate the k value
            k = self.kCalculator(CurrentShortenResiduals)
            #ShortenK[order] = k#Not needed?
            if k > 33:
                raise ValueError(f"k can not be
represented in 5 bits binary, max value for k is 32 and
current value is {k}")
            #The k-value is represented in 5 bits, which
allows for up to k = 33 to be represented.
            #This is becuase of k being subtracted by 1 so
 that k = 33 is represented by k = 32 which is max with 5
bits
            #k can never be less than 1 so k = 1 is
```

```
shifted down to be represented as 0 in binary

                #The first part of the codeword will be the k
value
                ShortenCodeWords[order] = np.binary_repr(k
-1,5)

                #rice code the residuals and add them to the
code word
                for i in range(len(CurrentShortenResiduals)):
                    ShortenCodeWords[order] += self.RiceEncode
(CurrentShortenResiduals[i], k)

            CurrentAdjacentSampleResidual = AdjacentResiduals[
microphone]

             #Calculate the k value
            k = self.kCalculator(CurrentAdjacentSampleResidual
)
            #ShortenK[order] = k#Not needed?
            if k > 33:
                raise ValueError(f"k can not be represented in
 5 bits binary, max value for k is 32 and current value is
{k}")
            #The k-value is represented in 5 bits, which
allows for up to k = 33 to be represented.
            #This is becuase of k being subtracted by 1 so
that k = 33 is represented by k = 32 which is max with 5
bits
            #k can never be less than 1 so k = 1 is shifted
down to be represented as 0 in binary

            #The first part of the codeword will be the k
value
            AdjacentSampleCodeWords = np.binary_repr(k-1,5)

            #rice code the residuals and add them to the code
word
            for i in range(len(CurrentAdjacentSampleResidual))
:
                AdjacentSampleCodeWords += self.RiceEncode(
```

```
CurrentAdjacentSampleResidual[i], k)

            #If EncoderForced is: 0 <= EncoderForced <= 6,
            if 0 <= self.EncoderForced <= 6:
                ChoosenEncoder = self.EncoderForced

            #Else the shortest codeword is calculated and
choosen as encoder
            else:
                #Find wich code word is shortest of RLE and
the orders of shorten
                ChoosenEncoder = 0
                EncoderLength = len(RleCode)
                for order in range(5):
                    if EncoderLength > len(ShortenCodeWords[
order]):
                        EncoderLength = len(ShortenCodeWords[
order])
                        ChoosenEncoder = order + 1

                #Compare the shortest code word of RLE or
SHorten to adjacent codeword
                if EncoderLength > len(AdjacentSampleCodeWords
):
                    ChoosenEncoder = 6

            #Binary represent the choosen code word, this will
 be the first bits in the code word
            CurrentCodeWord = np.binary_repr(ChoosenEncoder,3)
            #If Choosen encoder is 0 the best code word is
from RLE code
            if ChoosenEncoder == 0:
                CurrentCodeWord += RleCode
            #If choosen encoder is 6 the best code word is
from adjacent when residuals are sorted by samples
            elif ChoosenEncoder == 6:
                CurrentCodeWord += AdjacentSampleCodeWords
            #else it is from shorten
            else:
                CurrentCodeWord += ShortenCodeWords[
ChoosenEncoder-1]
```

```
            CodeWords.append(CurrentCodeWord)

        #Encode the adjacent residuals by sorting the
resiudals based on mic
        #The theory behind this is that the reisudals based on
 mic will be closer together and therefore have better k-
value when encoding
        #instead of encoding adjacent based on samples from
one mic
        #The first slot in the AdjacentCodeWords array will
represent the enocding choice for using adjacent
        AdjacentCodeWords = [np.binary_repr(7,3)]
        for sample in range(self.samples):

            #Save all residuals from all mics corresponding to
 one sample in the currentResidual array
            currentResidual = []
            for microphone in range(self.mics):
                AdjacentMicResiduals = AdjacentResiduals[
microphone]
                currentResidual.append(AdjacentMicResiduals[
sample])#Not sure about this line?

            #Calculate k-value for the currentResidual array
            k = self.kCalculator(currentResidual)
            if k > 33:
                raise ValueError(f"k can not be represented in
 5 bits binary, max value for k is 32 and current value is
{k}")
            #The k-value is represented in 5 bits, which
allows for up to k = 33 to be represented.
            #This is becuase of k being subtracted by 1 so
that k = 33 is represented by k = 32 which is max with 5
bits
            #k can never be less than 1 so k = 1 is shifted
down to be represented as 0 in binary

            #The first part of the codeword will be the k
```

```
value
            CurrentAdjacentCodeWord = np.binary_repr(k-1,5)

            #Rice code the residuals and add them to the code
word
            for i in range(len(currentResidual)):
                CurrentAdjacentCodeWord += self.RiceEncode(
currentResidual[i],k)

            AdjacentCodeWords.append(CurrentAdjacentCodeWord)

        #if EncoderForced is less than 0 the shortest code
words will be choosen
        if self.EncoderForced < 0:

            #Compare the total length of the adjacent
codewords based on sorting residuals by mics to the other
codewords
            samplesLen = 0
            micsLen = 0
            for i in range(len(CodeWords)):
                samplesLen += len(CodeWords[i])

            for i in range(len(AdjacentCodeWords)):
                micsLen += len(AdjacentCodeWords[i])

            if samplesLen < micsLen:
                ChoosenCodeWords = CodeWords

            else:
                ChoosenCodeWords = AdjacentCodeWords
        #If 0 <= EncoderForced < 7 the codeword is forced to
take a value previously calculated and Choosen codewords is
 set to CodeWords
        elif self.EncoderForced < 7:
            ChoosenCodeWords = CodeWords

        #In other cases for EncoderForced the ChoosenCodeWords
 will be for Adjacent when residuals are grouped by samples
        else:
            ChoosenCodeWords = AdjacentCodeWords
```

```
        return ChoosenCodeWords, memorysIn









    def Out(self, CodeWords, MemorysOut):
        AdjacentMics = self.CheckFirst(CodeWords[0])
        DecodedValues = []
        for microphone in range(self.mics):
            DecodedValues.append([])

        if AdjacentMics:


            #For Adjacent Mics the first
            for sample in range(1, len(CodeWords)):
                #Recreate the residuals from the Rice codes
                CodeWord = CodeWords[sample]
                RecreatedResiduals = self.RiceDecode(CodeWord)

                for i in range(len(RecreatedResiduals)):
                    CurrentRecreatedResidual =
RecreatedResiduals[i]
                    #Recreate the first mic value from Shorten
                    if i == 0:
                        FirstInRowValue, MemorysOut[i] =  self
.FirstAdjacentDecoder(CurrentRecreatedResidual, MemorysOut[
i])
                        AdjacentValue = FirstInRowValue
                    #If i%8 == 0 then a new row have been
started
                    #The mic value will be predicted from the
previous row and the decoded value will be saved as the new
```

```
 FirstInRowValue
                elif i % 8 == 0:
                    FirstInRowValue, MemorysOut[i] = self.
AdjacentDecoder(FirstInRowValue, CurrentRecreatedResidual,
MemorysOut[i])
                    AdjacentValue = FirstInRowValue
                #The value is predicted from the previous
mic value
                else:
                    AdjacentValue, MemorysOut[i] = self.
AdjacentDecoder(AdjacentValue, CurrentRecreatedResidual,
MemorysOut[i])

                DecodedValues[i].append(AdjacentValue)


    else:
        for i in range(len(CodeWords)):
            CodeWord = CodeWords[i]
            #Check what encoder was used to encode the
resiudals
            ChoosenEncoder, CodeWord = self.FindEncoder(
CodeWord)
            #If ChoosenEncoder is 0 RLE have been used to
encode the resiudals
            if ChoosenEncoder == 0:
                #Decode the values when RLE is used to
encode
                DecodedValues[i], MemorysOut[i] = self.
RleDecoder(CodeWord)

            else:
                #Decode the residuals from Rice codes
                RecreatedResiduals = self.RiceDecode(
CodeWord)

                #If ChossenEncoder is 6 Adjacent grouped
by sampels have been encoded
                if ChoosenEncoder == 6:
                    if i == 0:
                        DecodedValues[i], MemorysOut[i] =
```

```
self.ShortenDecoder(RecreatedResiduals, self.AdjacentOrder,
 MemorysOut[i])

                    else:
                        CurrentMicDecodedValues = []
                        if i % 8 == 0:
                            prediction = DecodedValues[i
-8].copy()

                        else:
                            prediction = DecodedValues[i
-1].copy()

                        for j in range(len(prediction)):
                            CurrentPrediction = prediction
[j]
                            CurrentResidual =
RecreatedResiduals[j]

                            CurrentDecodedValue,
MemorysOut[i] = self.AdjacentDecoder(CurrentPrediction,
CurrentResidual, MemorysOut[i])
                            CurrentMicDecodedValues.append
(CurrentDecodedValue)

                        DecodedValues[i] =
CurrentMicDecodedValues

                #In other cases Shorten have been used
                else:
                    #The shorten order that have been used
 is ChoosenEncoder - 1
                    order = ChoosenEncoder - 1

                    DecodedValues[i], MemorysOut[i] = self
.ShortenDecoder(RecreatedResiduals, order, MemorysOut[i])


    #Length of DecodedValues should match number of mics
    if len(DecodedValues) != self.mics:
        raise ValueError(f"DecodedValues length does not
```

```
match number of mics, current length is: {len(DecodedValues
)}")

        #Each saved slot in decodedValues should have a length
 that matches number of samples
        for microphone in range(len(DecodedValues)):
            if len(DecodedValues[microphone]) != self.samples:
                raise ValueError(f"Only {len(DecodedValues[
microphone])} samples decoded for microphone #{microphone}"
)

        return DecodedValues, MemorysOut




    def CheckFirst(self, CodeWord):
        #The first 3 binary value
        BinaryEncoder = CodeWord[:3]
        #Change their binary value to int
        ChoosenEncoder = int(BinaryEncoder,2)

        #Checks if the encoder used is adjacent mics
        if ChoosenEncoder == 7:
            AdjacentMicsUsed = True

        else:
            AdjacentMicsUsed = False

        return AdjacentMicsUsed


    def FindEncoder(self, CodeWord):
        #The first 3 binary value
        BinaryEncoder = CodeWord[:3]
        #Change their binary value to int
        ChoosenEncoder = int(BinaryEncoder,2)

        #Return the CodeWord without the binary digits
representing the Choosene encoder
```

```
        RemainingCodeWord = CodeWord[3:]

        return ChoosenEncoder, RemainingCodeWord


    #Calculates the ideal k-value according to modified Rice
theory
    #Take the Rice Theory value and if it is larger than 1
increment it by 1 (Best from test results)
    def kCalculator(self, residuals):

        #Convert all values to positive, since the will be
positive when encoding
        abs_res = np.absolute(residuals)
        #Calculate the mean value
        abs_res_avg = np.mean(abs_res)

        #k needs tobe a int > 1. abs_res_avg = 6.64 gives k =
1.
        #All k values for abs_res_avg above 6.64 is calculated
    using modified Rice Theory
        if abs_res_avg > 6.64:
            k = int(round(math.log(math.log(2,10) *
abs_res_avg,2))) + 1
        else:
            k = 1

        return k


    def RiceDecode(self, CodeWord):
        #The 5 first binary digits in the codeword represnt
the k value used to Rice code the residuals
        kBinary = CodeWord[:5]
        #When encoding k value it is decremented by 1 so when
deocing k it have to be incremented by 1
        k = int(kBinary,2) + 1
        #k should only be able to be in value between 1 and 33
        if not (1 <= k <= 33) or not isinstance(k, int):
            raise ValueError(f"Order can only have integer
values between 1 and 33. Current value of is {k}")
```

```
        CodeWord = CodeWord[5:]

        decoded_values = []

        #Loops trough the code word to get all values in the
code word
        while len(CodeWord) > 0:
            A = 0
            #If S is true the output should be negative
            S = False
            value = ""

            #Checks if the data is signed
            if self.sign:
                #Checks if the MSB is "1", indicating that the
 value should be negative.
                if CodeWord[0] == "1":
                    #Sets S to True if the output should be
negative
                    S = True
                #Removes the MSB, signed bit, from the code
word.
                #This way the rest of the code word can be
handled as if it was unsigned
                CodeWord = CodeWord[1:]


            #Decodes the unary code.
            #Unary code represents a interger value as a set
of "1":s followed by a "0".
            #By incrementing A by 1 if the MSB is a one then
moving the codeword one step and checking again
            #all "1" will increment A until a "0" is MSB
indicating that the unary code have been decoded.
            while CodeWord[0] == "1":
                A += 1
                CodeWord = CodeWord[1:]
```

```
            #After the unary code have been uncoded the
remaining code will be
            #a "0" as MSB followed by the last k-bits from the
 original binary value as LSB
            #By looping thorugh the codeword and removing k
bits to the current value being decoded
            #The last bits in the code word for the current
value can be decoded
            for j in range(k):
                CodeWord = CodeWord[1:]
                value += CodeWord[0]
            #Since there is a 0 separating the MSB with the
LSB 1 more bit needs to be removed for the current value in
 the codeword
            CodeWord = CodeWord[1:]

            value = bin(A)[2:] + value

            #The original binary value is converted to an int
            value = int(value, 2)

            #In if the original value was negative sign and S
will be true,
            #code will then be converted to a negative value
            if self.sign and S:
                value = -value

            #appends the decoded value in an array
            decoded_values.append(value)

        #Returns array with decoded int values
        return decoded_values


    #n is the value to encode in Rice code
    def RiceEncode(self, n, k):
        #If the input data is signed the signed bit needs to
be saved.
        #In case the input data is negative it is converted to
 positive beffore encoding.
        if self.sign:
```

```
            if n < 0:
                s = "1"
                n = -n
            else:
                s = "0"


        #If statments checks if n = 0, this should be handled
the same way as if n < k (see below),
        #but since math-log(0,2) does not work an extra if
statement is needed
        if n == 0:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there are k
bits representing the value n
            r = np.binary_repr(n,k)
            #The loop constant will detemen how long the unary
 code will be.
            #When n is fully represented in binary by r there
is no need for unary code.
            loop = 0

        #If statement checks that n value is large enough to
be longer than k bits when converted to binary.
        #This is done so when separating the k last bits it
would create an error if there is not enoguh bits.
        elif math.log(n,2) < k:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there are k
bits representing the value n
            r = np.binary_repr(n,k)
            #The loop constant will detemen how long the unary
 code will be.
            #When n is fully represented in binary by r there
is no need for unary code.
            loop = 0
        else:
            #Convert n to a binary value
            n = bin(n)[2:]
```

```
            #Takes the k last bits of n and saves it in
variable r
            r = n[len(n)-k:]
            #The remaining bits in n are converted to their
interger value.
            #This interger value will be unary coded
            loop = int(n[:len(n)-k],2)

        #Append a "0" bit as MSB in the code word r.
        #This will later show where the unary code ends.
        r = "0" + r


        #for loop with "loop" number of iteration.
        #Each iteration adds a "1" as MSB to r,
        #This way the unary code is created.
        for i in range(loop):
            r = "1" + r

        #In the case for signed data the sign bit is added to
r.
        if self.sign:
            r = s + r

        #Returns the code word in r
        return r


    def RleDecoder(self, CodeWord):

        DecodedValues = []
        #Loops trough all the enocded binary values with a
while llop
        while len(CodeWord) > 0:
            #Raise error if there is not enough length left of
 code_residuals for 1 code_word
            if len(CodeWord) < 33:
                raise ValueError(f"Code_word length is {len(
CodeWord)}, should  never be lower than 33")
```

```
            #The first bit in each code word is the sign bit
            #This is saved as s and then the code_residuals is
 stepped one step
            s = CodeWord[0]
            CodeWord = CodeWord[1:]

            #The next 24 bits represents the value that have
been encoded
            #Once they been saved 24 steps trough the
code_residuals are taken
            RleValue = CodeWord[:24]
            CodeWord = CodeWord[24:]

            #The next 8 bits represent how many times the
value was repeated
            #Once it been saved 8 steps is taken in the
code_residuals
            RleCount = CodeWord[:8]
            CodeWord = CodeWord[8:]

            #The value that have been encoded is converted to
int
            IntRelValue = int(RleValue, 2)

            #If the sign bit is "1" the value is converted to
negative
            if s == "1":
                IntRelValue = -IntRelValue

            #A for loop is used to append as many values as
needed
            #The RleCount value is increased by 1 because even
 though
            #a value never was reapeted it occured once

            for reapeats in range(1 + int(RleCount, 2)):
                DecodedValues.append(IntRelValue)

        #Uppdate memory
        #Copy the decoded values to assign them to the memory
array later
```

```
        RleCopy = DecodedValues.copy()
        #The last for values will be the memory array in
reverse order

        new_memory = []
        if len(RleCopy) > 3:
            reversed_memory = RleCopy[len(RleCopy)-4:]
        #In the case that there is to few values among the
decoded values pad the memory array with 0:s
        else:
            reversed_memory = RleCopy
            while len(new_memory) < 3:
                reversed_memory.insert(0,0)
        for i in reversed(reversed_memory):
            new_memory.append(i)


        #Assign the new_memory to memory
        return DecodedValues, new_memory


    def RleEnconder(self, RleValue, RleCounter):
        #First the signed dignit is set, 0 for positive and 1
for negative
        if RleValue < 0:
            s = "1"
            #if the value was negative it is turned to
positive beffore saved as value to be RLE encoded
            Rle_value = -RleValue
        else:
            s = "0"
            Rle_value = RleValue

        #The RLE value is written as a 24 bit representation
since this is the largest that can be sampled by the system
        RleValueBinary = np.binary_repr(Rle_value, 24)

        #The amount of bits that RLE count i written in is
dependent on self.samples
        #The maximum value for count is sample -1, the reason
for -1 is that RLE count is countign how many times a value
```

```
 is reapeted
        #So for 256 values you can have 1 initial value and
then reapeat it 255 times
        #It is important to round the value upwards
        BinaryRepresentation = math.ceil( math.log(self.
samples, 2) )
        RleCountBinary = np.binary_repr(RleCounter,
BinaryRepresentation)

        #Completes the current binary representation of the
RLE
        RleBinary = s + RleValueBinary + RleCountBinary

        #Return the binary representation of the RLE
        return RleBinary


    def ShortenDecoder(self, Residuals, Order, memoryOut):
        RecreatedValues = []
        for residual in Residuals:
            if Order == 0:
                #If order = 0 preditiction is allways 0
                prediciton = 0

            else:
                #Calculate the prediciton from memory and
cofficents
                prediciton = sum( np.array(memoryOut[:Order])
* np.array(self.ShortenCofficents[Order]) )

            #Recreate the original value by adding prediciton
and residual value
            RecreatedValue = prediciton + residual

            #Update the memory by stepping all values one step
 and setting the first memory slot to the recreated value
            new_memory = [RecreatedValue]
            memoryOut = new_memory + memoryOut[:-1]

            #Save the recreated value
            RecreatedValues.append(RecreatedValue)
```

```
        return RecreatedValues, memoryOut


    def ShortenPredictIn(self, memoryIn):
        #Predict shorten for order 0-4
        ShortenPredictions = [0]
        for i in range(1,5):

            ShortenPredictions.append(sum( np.array(memoryIn[:
i]) * np.array(self.ShortenCofficents[i]) ))

        #All memory slots are stepped one step back and the
oldest memory is dropped
        new_memory = [0]
        new_memory += memoryIn[:-1]

        return ShortenPredictions, new_memory


    def AdjacentPredictIn(self, CurrentInput,
CurrentPrediction):
        #Calculate the residual by taking the current input
and subtracting the residual
        CurrentResidual = CurrentInput - CurrentPrediction

        #Crate a new prediction by taking the current mic
value, the next mic will be predicted with the current mic
value
        NewPrediction = CurrentInput

        return CurrentResidual, NewPrediction


    def AdjacentDecoder(self, prediction, residual, memory):


        #Recreate the original input value by adding
prediciton and residual
        DecodedValue = prediction + residual
```

```
        #Update memory
        new_memory = [DecodedValue] + memory[:-1]


        return DecodedValue, new_memory


    def FirstAdjacentDecoder(self, residual, memory):
        Firstpredict = (sum( np.array(memory[:self.
    AdjacentOrder]) * np.array(self.ShortenCofficents[self.
    AdjacentOrder]) ))
        FirstDecodedValue = Firstpredict + residual
        new_memory = [FirstDecodedValue] + memory[:-1]

        return FirstDecodedValue, new_memory
```

A.2.8 DoubleCompression class code

```
import numpy as np
import math

class DoubleCompression:

    def __init__(self, ShortenOrder, sign = True, mics: int =
    64, samples: int = 256, AdjacentOrder: int = 2):
        self.ShortenCofficents = [[0],[1],[2, -1],[3, -3,
    1],[4,-6,4,-1]]
        self.mics = mics
        self.samples = samples
        self.AdjacentOrder = AdjacentOrder
        self.sign = sign
        self.ShortenOrder = ShortenOrder

    def In(self, Inputs, MemoryShorten, MemoryAdjacent):
        UnsortedResiduals = []
        AllCodeWords = []
        AllSortedResiduals = []#only for testing
        #loop thorugh all inputs and calculate their residuals
    using Adjacentpredictor
        for sample in range(self.samples):
```

```
            CurrentInputs = Inputs[:,sample]

            AdjacentResiduals, MemoryAdjacent = self.
AdjacentResiduals(CurrentInputs, MemoryAdjacent)
            UnsortedResiduals.append(AdjacentResiduals)

        #Sort the residuals so that they are grouped by mic
and not by sample
        SortedResiduals = self.SortResiduals(UnsortedResiduals
)
        AllSortedResiduals.append(SortedResiduals)#only for
testing

        #Loop though each residual and compress it again by
using shorten
        #The shorten predictor now predict the residual
between two mic for all sampels
        for mic in range(self.mics):

            ShortenResiduals, MemoryShorten[mic] = self.
ShortenResiduals(SortedResiduals[mic], MemoryShorten[mic])

            #Calculate what k-value is best to encode the
Shorten residuals using Rice Codes.
            #Modified k equation is used where k-values that
are not rounded up to one are incremented by 1

            k = self.kCalculator(ShortenResiduals)

            #The first value in the CodeWord will be the k-
value for the residual in binary code
            #k is encoded in 5 bits, allowing for values from
0-32, since k can never be less than 1
            #k is decremented by 1 beffore enocding it.
Allowing for k values in between: 1 <= k <= 33
            if 33 < k :
                raise ValueError(f"k can not be represented in
 5 bits binary, max value for k is 32 and current value is
{k}")

            kBinary = np.binary_repr(k-1,5)
```

```
            CodeWord = kBinary
            #encode all residuals using RiceCode
            for residual in ShortenResiduals:
                CodeWord += self.RiceEncode(residual, k)

            AllCodeWords.append(CodeWord)

        return AllCodeWords, MemoryShorten, MemoryAdjacent,
AllSortedResiduals#AllsortedResiuduals is only for testing

    def Out(self, AllCodeWords, MemoryShorten, MemoryAdjcent):
        AllDecodedValues = []
        for mic in range(self.mics):
            #Grab the codewords for each encoded mic
            CodeWords = AllCodeWords[mic]

            #Decode the Rice coded codewords to get the
Shorten Residuals
            ShortenResiduals = self.RiceDecode(CodeWords)

            #Recreated the adjacent residual from the shorten
residuals
            AdjacentResiduals, MemoryShorten[mic] = self.
ShortenRecreate(ShortenResiduals, MemoryShorten[mic], self.
ShortenOrder)

            #If it is the first mic the Adjacent residuals
have been created using Shorten
            if mic == 0:
                DecodedValues, MemoryAdjcent = self.
ShortenRecreate(AdjacentResiduals, MemoryAdjcent, self.
AdjacentOrder)
            #If it is any of the other mics they have been
created using an adjacent mic
            #If mic % 8 == 0 they have been predicted using
the first mic in the previous row
            elif mic % 8 == 0:
                DecodedValues = self.AdjacentRecreate(
AdjacentResiduals, AllDecodedValues[mic-8])
            #Else they have been predicted using the previous
mic value
```

```
            else:
                DecodedValues = self.AdjacentRecreate(
AdjacentResiduals, AllDecodedValues[mic-1])


            AllDecodedValues.append(DecodedValues)


        return AllDecodedValues, MemoryShorten, MemoryAdjcent




    def RiceDecode(self, CodeWord):
        #The 5 first binary digits in the codeword represnt
the k value used to Rice code the residuals
        kBinary = CodeWord[:5]
        #When encoding k value it is decremented by 1 so when
deocing k it have to be incremented by 1
        k = int(kBinary,2) + 1
        #k should only be able to be in value between 1 and 33
        if not (1 <= k <= 33) or not isinstance(k, int):
            raise ValueError(f"Order can only have integer
values between 1 and 33. Current value of is {k}")


        CodeWord = CodeWord[5:]

        decoded_values = []

        #Loops trough the code word to get all values in the
code word
        while len(CodeWord) > 0:
            A = 0
            #If S is true the output should be negative
            S = False
            value = ""

            #Checks if the data is signed
            if self.sign:
```

```
            #Checks if the MSB is "1", indicating that the
 value should be negative.
            if CodeWord[0] == "1":
                #Sets S to True if the output should be
negative
                S = True
            #Removes the MSB, signed bit, from the code
word.
            #This way the rest of the code word can be
handled as if it was unsigned
            CodeWord = CodeWord[1:]


        #Decodes the unary code.
        #Unary code represents a interger value as a set
of "1":s followed by a "0".
        #By incrementing A by 1 if the MSB is a one then
moving the codeword one step and checking again
        #all "1" will increment A until a "0" is MSB
indicating that the unary code have been decoded.
        while CodeWord[0] == "1":
            A += 1
            CodeWord = CodeWord[1:]


        #After the unary code have been uncoded the
remaining code will be
        #a "0" as MSB followed by the last k-bits from the
 original binary value as LSB
        #By looping thorugh the codeword and removing k
bits to the current value being decoded
        #The last bits in the code word for the current
value can be decoded
        for j in range(k):
            CodeWord = CodeWord[1:]
            value += CodeWord[0]
        #Since there is a 0 separating the MSB with the
LSB 1 more bit needs to be removed for the current value in
 the codeword
        CodeWord = CodeWord[1:]
```

```
            value = bin(A)[2:] + value

            #The original binary value is converted to an int
            value = int(value, 2)

            #In if the original value was negative sign and S
will be true,
            #code will then be converted to a negative value
            if self.sign and S:
                value = -value

            #appends the decoded value in an array
            decoded_values.append(value)

        #Returns array with decoded int values
        return decoded_values


    #n is the value to encode in Rice code
    def RiceEncode(self, n, k):
        #If the input data is signed the signed bit needs to
be saved.
        #In case the input data is negative it is converted to
 positive beffore encoding.
        if self.sign:
            if n < 0:
                s = "1"
                n = -n
            else:
                s = "0"


        #If statments checks if n = 0, this should be handled
the same way as if n < k (see below),
        #but since math-log(0,2) does not work an extra if
statement is needed
        if n == 0:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there are k
bits representing the value n
```

```
            r = np.binary_repr(n,k)
            #The loop constant will detemen how long the unary
 code will be.
            #When n is fully represented in binary by r there
is no need for unary code.
            loop = 0

        #If statement checks that n value is large enough to
be longer than k bits when converted to binary.
        #This is done so when separating the k last bits it
would create an error if there is not enoguh bits.
        elif math.log(n,2) < k:
            #If n represent fewer than k bits it will be
converted to a bianry value with padding.
            #The padding are zeros as MSB so that there are k
bits representing the value n
            r = np.binary_repr(n,k)
            #The loop constant will detemen how long the unary
 code will be.
            #When n is fully represented in binary by r there
is no need for unary code.
            loop = 0
        else:
            #Convert n to a binary value
            n = bin(n)[2:]
            #Takes the k last bits of n and saves it in
variable r
            r = n[len(n)-k:]
            #The remaining bits in n are converted to their
interger value.
            #This interger value will be unary coded
            loop = int(n[:len(n)-k],2)

        #Append a "0" bit as MSB in the code word r.
        #This will later show where the unary code ends.
        r = "0" + r


        #for loop with "loop" number of iteration.
        #Each iteration adds a "1" as MSB to r,
        #This way the unary code is created.
```

```
        for i in range(loop):
            r = "1" + r

        #In the case for signed data the sign bit is added to
r.
        if self.sign:
            r = s + r

        #Returns the code word in r
        return r

    #Calculates the ideal k-value according to modified Rice
theory
    #Take the Rice Theory value and if it is larger than 1
increment it by 1 (Best from test results)
    def kCalculator(self, residuals):

        #Convert all values to positive, since the will be
positive when encoding
        abs_res = np.absolute(residuals)
        #Calculate the mean value
        abs_res_avg = np.mean(abs_res)

        #k needs tobe a int > 1. abs_res_avg = 6.64 gives k =
1.
        #All k values for abs_res_avg above 6.64 is calculated
 using modified Rice Theory
        if abs_res_avg > 6.64:
            k = int(round(math.log(math.log(2,10) *
abs_res_avg,2))) + 1
        else:
            k = 1

        if k < 1 :
            raise ValueError(f"k can not be less than 1,
current value is {k}")

        return k

    #Sort the residuals so that they are grouped by mic and
not sample
```

```
    def SortResiduals(self, UnsortedResiduals):
        SortedResiduals = []
        for mic in range(self.mics):
            SortedResiduals.append([])
            for sample in range(self.samples):
                SampleResiduals = UnsortedResiduals[sample]
                SortedResiduals[mic].append(SampleResiduals[
mic])

        return SortedResiduals

    def AdjacentRecreate(self, residuals, adjacentValues):
        OriginalValues = []
        #recreate the original values by adding the residual
to the adjacent mic values
        for i in range(len(residuals)):
            originalValue = residuals[i] + adjacentValues[i]
            OriginalValues.append(originalValue)
        return OriginalValues


    #Predict the next mic using the previous mic, save the
residual
    def AdjacentResiduals(self, Inputs, memory):
        #Use Shorten to calculate the residual for the first
microphone in the array
        firstPrediction = self.ShortenPredictor(memory, self.
AdjacentOrder)
        firstResidual = Inputs[0] - firstPrediction
        #Save the first residual and prediciton in the
residual and prediciton arrays
        Residuals = [firstResidual]
        new_memory = [Inputs[0]] + memory[:-1].copy()

        #Save the first value as the previous value and
previous row value
        #Previous indicates the value of the previous mice
        #PreviousRow indicates the first value of the previous
 row
        #The rest of the resiudals are calculated by taking
the difference between the current mic value and the
```

```
previous mic value
        Previous = Inputs[0]
        PreviousRow = Inputs[0]
        for mic in range(1, len(Inputs)):
            #When i modulos 8 is equal to 0 it indicates that
a new row has started (8 mics per row)
            if mic % 8 == 0:
                #The previous value is then taken from the
previous row and not the previous mic
                #This will give a mic that is closer to the
current mic
                CurrentResidual = self.AdjacentPredictor(
Inputs[mic], PreviousRow)
                #A new PreviousRow value is calculculated
                #This is the first value of the new row
                PreviousRow = Inputs[mic]
            else:
                CurrentResidual = self.AdjacentPredictor(
Inputs[mic], Previous)

            #Update the previous value
            Previous = Inputs[mic]

            #Save the residual
            Residuals.append(CurrentResidual)


        #Return the residuals, memory
        return Residuals, new_memory

    def AdjacentPredictor(self, input, Prediction):
        #The residual is the difference of the current mic
value and the previous mic value
        Residual = input - Prediction
        #Return the residual
        return Residual

    def ShortenRecreate(self, ShortenResiduals, memory, order)
:
        RecreatedValues = []
        for residual in ShortenResiduals:
```

```
            prediction = self.ShortenPredictor(memory, order)

            recreatedValue = residual + prediction
            memory = [recreatedValue] + memory[:-1]
            RecreatedValues.append(recreatedValue)

        return RecreatedValues, memory

    def ShortenResiduals(self, inputs, memory):
        residuals = []
        for i in range(len(inputs)):
            currentPrediction = self.ShortenPredictor(memory,
    self.ShortenOrder)
            currentResidual = inputs[i] - currentPrediction


            residuals.append(currentResidual)
            memory = [inputs[i]] + memory[:-1]

        return residuals, memory

    #Prediction using Shorten, predict the value using Shorten
    cofficents of the correct order and
    def ShortenPredictor(self, memory, order):
        if order == 0:
            prediction = 0

        else:
            prediction = sum( np.array(memory[:order]) * np.
    array(self.ShortenCofficents[order]) )



        return prediction
```

A.3 VHDL code

A.3.1 AdjacentResidual code

```
-- AdjacentResiduals
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- Import the numeric_std package
   for signed type

entity AdjacentResiduals is

   port (
      clk   : in std_logic;
      reset : in std_logic;

      DataBlockIn      : in std_logic_vector(1535 downto 0);
      ReadyToEncodeIn  : in std_logic;
      DataBlockReadyIn : in std_logic;
      ReadyToReciveIn  : in std_logic;

      ResidualsCalculatedOut : out std_logic;
      ResidualValueOut       : out std_logic_vector(24 downto
         0);
      NewDataOut             : out std_logic;
      AllResidualsOut        : out std_logic_vector(1599
         downto 0);
      NewResidualOut         : out std_logic;

      ErrorOut : out std_logic
   );

end entity;

architecture Behavioral of AdjacentResiduals is
   type state_type is (Idle, Reciving, GrabNextValue,
      CalculateResidual, SetResidualValueOut,
      SetAllResidualsValue, SendCurrentResidual,
      SendAllResiduals, ErrorState); -- States for the
      statemachine
   signal state : state_type;

   signal CurrentDataBlock    : std_logic_vector(1535 downto
      0);
   signal mic                 : integer range 0 to 64;
```

```
   signal CurrentValue        : std_logic_vector(23 downto 0);
   signal predict_first       : signed(23 downto 0);
   signal prediction          : signed(23 downto 0);
   signal prediction_row      : signed(23 downto 0);
   signal CurrentResidual     : std_logic_vector(24 downto 0);
   signal CheckState          : integer range 0 to 10;
   signal ResidualCounter     : integer range 0 to 1535;
   signal AllResidualsCounter : integer range 0 to 1599;

begin
   process (clk)
   begin
      if rising_edge(clk) then

         case state is
            when Idle =>
               CheckState <= 0;

               --reset all to deafult values
               ResidualsCalculatedOut <= '1';--This is set to
                  '1' to indicate that the code is ready to
                  calculate residuals for a datablock
               ResidualValueOut       <= (others => '0');
               NewDataOut             <= '0';
               AllResidualsOut        <= (others => '0');
               NewResidualOut         <= '0';
               ErrorOut               <= '0';

               mic                 <= 0;
               CurrentValue        <= (others => '0');
               prediction          <= (others => '0');
               prediction_row      <= (others => '0');
               CurrentResidual     <= (others => '0');
               ResidualCounter     <= 0;
               AllResidualsCounter <= 0;

               --Once a new datablock is available go to the
                  next state
               if DataBlockReadyIn = '1' then
                  state <= Reciving;
               end if;
```

```
            when Reciving =>
               CheckState <= 1;

               ResidualsCalculatedOut <= '0';
               --Store the datablock
               CurrentDataBlock <= DataBlockIn;

               state <= GrabNextValue;

            when GrabNextValue =>
               CheckState <= 2;
               NewDataOut <= '0';

               --Grab the current value from the datablock
               CurrentValue(23) <= CurrentDataBlock(1535 -
                  ResidualCounter);
               CurrentValue(22) <= CurrentDataBlock(1534 -
                  ResidualCounter);
               CurrentValue(21) <= CurrentDataBlock(1533 -
                  ResidualCounter);
               CurrentValue(20) <= CurrentDataBlock(1532 -
                  ResidualCounter);
               CurrentValue(19) <= CurrentDataBlock(1531 -
                  ResidualCounter);
               CurrentValue(18) <= CurrentDataBlock(1530 -
                  ResidualCounter);
               CurrentValue(17) <= CurrentDataBlock(1529 -
                  ResidualCounter);
               CurrentValue(16) <= CurrentDataBlock(1528 -
                  ResidualCounter);
               CurrentValue(15) <= CurrentDataBlock(1527 -
                  ResidualCounter);
               CurrentValue(14) <= CurrentDataBlock(1526 -
                  ResidualCounter);
               CurrentValue(13) <= CurrentDataBlock(1525 -
                  ResidualCounter);
               CurrentValue(12) <= CurrentDataBlock(1524 -
                  ResidualCounter);
               CurrentValue(11) <= CurrentDataBlock(1523 -
                  ResidualCounter);
```

```
            CurrentValue(10) <= CurrentDataBlock(1522 -
               ResidualCounter);
            CurrentValue(9)  <= CurrentDataBlock(1521 -
               ResidualCounter);
            CurrentValue(8)  <= CurrentDataBlock(1520 -
               ResidualCounter);
            CurrentValue(7)  <= CurrentDataBlock(1519 -
               ResidualCounter);
            CurrentValue(6)  <= CurrentDataBlock(1518 -
               ResidualCounter);
            CurrentValue(5)  <= CurrentDataBlock(1517 -
               ResidualCounter);
            CurrentValue(4)  <= CurrentDataBlock(1516 -
               ResidualCounter);
            CurrentValue(3)  <= CurrentDataBlock(1515 -
               ResidualCounter);
            CurrentValue(2)  <= CurrentDataBlock(1514 -
               ResidualCounter);
            CurrentValue(1)  <= CurrentDataBlock(1513 -
               ResidualCounter);
            CurrentValue(0)  <= CurrentDataBlock(1512 -
               ResidualCounter);

            -- increment ResidualCounter by 24 so that the
               next value can be grabbed
            if ResidualCounter > 1488 then
               state <= ErrorState;

            else
               ResidualCounter <= ResidualCounter + 24;

            end if;

            state <= CalculateResidual;

         when CalculateResidual =>
            CheckState <= 3;

            --Check if its the first mic
            --if it is use the memory from the previous
               datablock to predict the value
```

```
            if mic = 0 then
               CurrentResidual <= std_logic_vector(
                  to_signed(to_integer(signed(CurrentValue)
                   - predict_first), 25));
               --update prediction row to the currentvalue
                  (first mic is also the first in a row)
               prediction_row <= signed(CurrentValue);
               --update the first mic value
               predict_first <= signed(CurrentValue);

            --Check if the current mic is the first mic in
               a row,
            -- if mic % 8 = 0 it is the first mic in a row
            elsif mic mod 8 = 0 then
               --Calculate the residual,
               --if it is the first mic in the row
                  calculate from the first mic in the
                  previous row
               CurrentResidual <= std_logic_vector(
                  to_signed(to_integer(signed(CurrentValue)
                   - prediction_row), 25));
               --update prediction row to the currentvalue
               prediction_row <= signed(CurrentValue);

            else
               --Calculate the residual
               CurrentResidual <= std_logic_vector(
                  to_signed(to_integer(signed(CurrentValue)
                   - prediction), 25));

            end if;

            --Update prediction to the current value
            prediction <= signed(CurrentValue);

            state <= SetResidualValueOut;

         when SetResidualValueOut =>
            CheckState <= 4;

            ResidualValueOut <= CurrentResidual;
```

```
            state <= SetAllResidualsValue;

         when SetAllResidualsValue =>
            CheckState <= 5;

            --Set the bits in all Resiudal value
            AllResidualsOut(1599 - AllResidualsCounter) <=
               CurrentResidual(24);
            AllResidualsOut(1598 - AllResidualsCounter) <=
               CurrentResidual(23);
            AllResidualsOut(1597 - AllResidualsCounter) <=
               CurrentResidual(22);
            AllResidualsOut(1596 - AllResidualsCounter) <=
               CurrentResidual(21);
            AllResidualsOut(1595 - AllResidualsCounter) <=
               CurrentResidual(20);
            AllResidualsOut(1594 - AllResidualsCounter) <=
               CurrentResidual(19);
            AllResidualsOut(1593 - AllResidualsCounter) <=
               CurrentResidual(18);
            AllResidualsOut(1592 - AllResidualsCounter) <=
               CurrentResidual(17);
            AllResidualsOut(1591 - AllResidualsCounter) <=
               CurrentResidual(16);
            AllResidualsOut(1590 - AllResidualsCounter) <=
               CurrentResidual(15);
            AllResidualsOut(1589 - AllResidualsCounter) <=
               CurrentResidual(14);
            AllResidualsOut(1588 - AllResidualsCounter) <=
               CurrentResidual(13);
            AllResidualsOut(1587 - AllResidualsCounter) <=
               CurrentResidual(12);
            AllResidualsOut(1586 - AllResidualsCounter) <=
               CurrentResidual(11);
            AllResidualsOut(1585 - AllResidualsCounter) <=
               CurrentResidual(10);
            AllResidualsOut(1584 - AllResidualsCounter) <=
               CurrentResidual(9);
            AllResidualsOut(1583 - AllResidualsCounter) <=
               CurrentResidual(8);
```

```
            AllResidualsOut(1582 - AllResidualsCounter) <=
               CurrentResidual(7);
            AllResidualsOut(1581 - AllResidualsCounter) <=
               CurrentResidual(6);
            AllResidualsOut(1580 - AllResidualsCounter) <=
               CurrentResidual(5);
            AllResidualsOut(1579 - AllResidualsCounter) <=
               CurrentResidual(4);
            AllResidualsOut(1578 - AllResidualsCounter) <=
               CurrentResidual(3);
            AllResidualsOut(1577 - AllResidualsCounter) <=
               CurrentResidual(2);
            AllResidualsOut(1576 - AllResidualsCounter) <=
               CurrentResidual(1);
            AllResidualsOut(1575 - AllResidualsCounter) <=
               CurrentResidual(0);

            --Increment AllResidualsCounter for the next
               residuals to be set
            if AllResidualsCounter > 1550 then
               state <= ErrorState;

            else
               AllResidualsCounter <= AllResidualsCounter +
                   25;

            end if;

            --increment mic
            if mic < 64 then
               mic <= mic + 1;

            else
               state <= ErrorState;

            end if;

            state <= SendCurrentResidual;

         when SendCurrentResidual =>
```

```
            CheckState <= 6;
            NewDataOut <= '1';

            if ReadyToReciveIn = '1' then
               --Check if all mics residuals have been
                  calculated
               if mic = 64 then
                  --If mic = 64 all mics residuals have
                     been calculated and AllResidualsOut
                     can be sent
                  state <= SendAllResiduals;

                  --if there are more residuals to be
                     calculated in the datablock grab the
                     next value
               else
                  state <= GrabNextValue;

               end if;

            end if;
         when SendAllResiduals =>
            CheckState     <= 7;
            NewResidualOut <= '1';

            NewDataOut <= '0';

            if ReadyToEncodeIn = '1' then
               state <= Idle;

            end if;

         when ErrorState =>
            CheckState <= 10;
            ErrorOut   <= '1';

      end case;

      if reset = '1' then
         state <= Idle;
```

```
            ResidualsCalculatedOut <= '0';
            ResidualValueOut       <= (others => '0');
            NewDataOut             <= '0';
            AllResidualsOut        <= (others => '0');
            NewResidualOut         <= '0';
            ErrorOut               <= '0';

            mic                 <= 0;
            CurrentValue        <= (others => '0');
            prediction          <= (others => '0');
            prediction_row      <= (others => '0');
            CurrentResidual     <= (others => '0');
            ResidualCounter     <= 0;
            AllResidualsCounter <= 0;

            predict_first <= (others => '0');

        end if;

    end if;
  end process;

end Behavioral;
```

A.3.2 kCalculator code

```
-- kCalculator

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity kCalculator is

   port (
      reset                      : in std_logic;
      clk                        : in std_logic;
      ResidualValueIn            : in std_logic_vector(24
         downto 0);--value in to be encoded
      NewDataIn                  : in std_logic;--New data
         available for input
      ReadyToEncodeInkCalculator : in std_logic;

      ReadyToReciveOut : out std_logic;
      NewKOut          : out std_logic;--New k-value to be
         sent
      kValueOut        : out std_logic_vector(4 downto 0);--k-
         value between 0 and 31
      ErrorOut         : out std_logic--Send an error signal
         when error state is reached

   );

end entity;

architecture Behavioral of kCalculator is
   type state_type is (Idle, Reciving, SumValues, WaitForValue
      , CalculateK, Sending, ErrorState); -- States for the
      statemachine
   signal state : state_type;

   signal CheckState    : integer range 0 to 10;--to see what
      state is currently used for the simulation
   signal CurrentValue  : signed(24 downto 0);
   signal SampleCounter : integer range 0 to 256;--count how
```

```
      many samples have been recived, assuming maximum 256
      samples per datablock
   signal TotalSum      : integer range 0 to 1300000000;
begin
   process (clk)
   begin
      if rising_edge(clk) then

         case state is
            when Idle =>
               CheckState <= 0;

               SampleCounter    <= 0;
               TotalSum         <= 0;
               NewKOut          <= '0';
               ErrorOut         <= '0';
               CurrentValue     <= (others => '0');

               if NewDataIn = '1' then
                  state <= Reciving;
                  ReadyToReciveOut <= '0';

               else
                  ReadyToReciveOut <= '1';

               end if;

            when Reciving =>
               CheckState <= 1;

               ReadyToReciveOut <= '0';
               CurrentValue     <= signed(ResidualValueIn);
               SampleCounter    <= SampleCounter + 1;

               state <= SumValues;

            when SumValues =>
               CheckState <= 2;
               --This state summarises the absolute value of
                  all input values
               TotalSum <= TotalSum + to_integer(abs(
```

```
                CurrentValue));

            --if TotalSum is equal or larger than
               1261097232 the best k-value is allway 22 (
               for all inputs of 24 bit signed)
            --This limit is set after "4 926 122*256",
               since if that value is reached for totalsum
               k will allways be 22.
            --This assumes that 256 samples will be used,
               however when looking at just one microphone
               array only 64 samples per datablock is used
            --The limit could therefore be lowered.
            if TotalSum >= 1261087232 then
               state <= CalculateK;

               --if 64 samples (assuming 64 samples per
                  datablock, if more samples per data block
                   is used this number needs to adapt after
                   it)
               --have been looked at the full datablock
                  have been sumarised and it is time to
                  find the ideal k-value,
               -- else another sample is grabbed
            elsif Samplecounter < 64 then
               state <= WaitForValue;

            else
               state <= CalculateK;

            end if;

         when WaitForValue =>
            CheckState       <= 3;

            if NewDataIn = '1' then
               state <= Reciving;
               ReadyToReciveOut <= '0';
            else
               ReadyToReciveOut <= '1';

            end if;
```

```
            when CalculateK =>
               CheckState <= 4;

               --if statement for differente TotalSum levels
                  to find best k-values
               --these levels have been derive from:
               -- k = log2(log10(2) * x /s ) + 1, where "x" is
                   the TotalSum and "s" is the total amount of
                   samples
               -- to avoid devision the limits for the total
                  sum have been pre calculated
               -- the limits assume 64 datasamples per
                  datablock, this is one microphone array.
               -- but the limits for all microphones, 256
                  samples per datablock have also been
                  calculated and can be found in the comment
               -- along with how the calculations was made so
                  that the limit can easily be adjusted in the
                   future by multiplying with a new amount of
                  samples

               if TotalSum < 38528 then--602*64 = 38 528,
                  602*256 = 154112
                  kValueOut <= "01000";--8

               elsif TotalSum < 76992 then--1203 * 64 = 76
                  992, 1203*256 = 307968
                  kValueOut <= "01001";--9

               elsif TotalSum < 153984 then--2406 * 64 = 153
                  984, 2406*256 = 615936
                  kValueOut <= "01010";--10

               elsif TotalSum < 307904 then--4811 * 64 = 307
                  904, 4811*256 = 1231616
                  kValueOut <= "01011";--11

               elsif TotalSum < 615808 then--9622 * 64 = 615
                  808, 9622*256 = 2463232
                  kValueOut <= "01100";--12
```

```
            elsif TotalSum < 1231552 then--19234 * 64 = 1
               231 552, 19243*256 = 4926208
               kValueOut <= "01101";--13

            elsif TotalSum < 2463104 then--38486 * 64 = 2
               463 104,38486*256 = 9852416
               kValueOut <= "01110";--14

            elsif TotalSum < 4926144 then--76971 * 64 = 4
               926 144, 76971*256 = 19704576
               kValueOut <= "01111";--15

            elsif TotalSum < 9852288 then--153942 * 64 = 9
               852 288, 153942*256 = 39409152
               kValueOut <= "10000";--16

            elsif TotalSum < 19704512 then--307 883 * 64 =
               19 704 512, 307883*256 = 78818048
               kValueOut <= "10001";--17

            elsif TotalSum < 39409024 then--615 766 * 64 =
               39 409 024, 615766*256 = 157636096
               kValueOut <= "10010";--18

            elsif TotalSum < 78817984 then--1 231 531 * 64
               = 78 817 984, 1231531*256 = 315271936
               kValueOut <= "10011";--19

            elsif TotalSum < 157635904 then--2 463 061 * 64
                = 157 635 904, 2463061*256 = 630543616
               kValueOut <= "10100";--20

            elsif TotalSum < 315271808 then--4 926 122 * 64
                = 315 271 808, 4926122*256 = 1261087232
               kValueOut <= "10101";--21

            else
               kValueOut <= "10110";--22

            end if;
```

```
            state <= Sending;

         when Sending =>
            CheckState <= 5;
            NewKOut <= '1';

            --New k value calculated, ready to be sent out


            --Once ready to encode is =1 RiceEncoder block
               is ready to recive the new k-value
            --When it have been sent return to idle state
               to wait for the next residuals to calculate
               new k-value
            if ReadyToEncodeInkCalculator = '1' then
               state <= Idle;

            end if;

         when ErrorState =>
            CheckState <= 10;
            ErrorOut   <= '1';

      end case;

      if reset = '1' then
         state             <= Idle;
         SampleCounter     <= 0;
         TotalSum          <= 0;
         ReadyToReciveOut  <= '0';
         NewKOut           <= '0';
         ErrorOut          <= '0';
         kValueOut         <= (others => '0');
         kValueOut         <= "01000";
         CurrentValue      <= (others => '0');

      end if;

   end if;
```

```
    end process;

end Behavioral;
```

A.3.3 RiceEncode code

```
-- RiceEncoder

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity NewRiceEncoder is

   port (
      reset                   : in std_logic;
      clk                     : in std_logic;
      AllResidualsIn          : in std_logic_vector(1599
         downto 0);--value in to be encoded
      kValueIn                : in std_logic_vector(4 downto
         0);
      NewKIn                  : in std_logic;--New data
         available for input
      NewResidualsIn          : in std_logic;
      ReadytoReciveCodeWordIn : in std_logic;

      NewCodeWordReady : out std_logic;--New CodeWord ready to
          be sent
      ReadyToEncode    : out std_logic;
      CodeWordLen      : out std_logic_vector(9 downto 0);--
         Bits used in the codeword
      CodeWord         : out std_logic_vector(1023 downto 0);
         --CodeWord out
      k_valueOut       : out std_logic_vector(4 downto 0);--
         Behvs den?

      ErrorOut : out std_logic--Send an error signal when
         error state is reached

   );

end entity;

architecture Behavioral of NewRiceEncoder is
   type state_type is (Idle, Reciving, NewResidual, SetValue,
```

```
      SignBit, FindKLastBits, FindRleCode, RleCoding,
      SetKLastBits, Sending, ErrorState); -- States for the
      statemachine
   signal state : state_type;
   --used in all states
   signal LenCounter          : integer range 0 to 1023;--
      count length of code word
   signal CurrentCodeWord     : std_logic_vector(1023 downto
      0);--set bits for current codeword
   signal AllCurrentResiduals : std_logic_vector(1599 downto
      0);
   signal CurrentResidual     : std_logic_vector(24 downto 0);
   signal CurrentValue        : signed(24 downto 0);--current
      value being encoded
   signal k_val_int           : integer range 0 to 32;
   signal k_pow               : integer range 0 to 1073741824;
      -- max val is 30^2, max val of k^2
   signal abs_bits_set        : integer range 0 to 32;--counts
       how many of the last bits are set
   signal CheckState          : integer range 0 to 10;--to see
       what state is currently used for the simulation
   signal ResidualBitCounter  : integer range 0 to 1600;
   signal SampleCounter       : integer range 0 to 64;
   --used in SignBit state
   signal AbsoluteValue : std_logic_vector(24 downto 0);

   --Used in **KLastBits
   signal kLastBits : std_logic_vector(24 downto 0);
   signal kBitsSet  : integer range 0 to 32;

   --Used in **RleCoding
   signal RleBits : std_logic_vector(24 downto 0);
   signal RleLoop : integer range 0 to 1023;



begin
   process (clk)
   begin
      if rising_edge(clk) then
```

```
        case state is
           when Idle =>
              --Indicate what the current state is in
                 simulation
              CheckState <= 0;

              -- default values for signals
              k_val_int           <= to_integer(unsigned(
                 kValueIn));
              k_pow               <= 2 ** to_integer(unsigned
                 (kValueIn));
              LenCounter          <= 0;
              CurrentCodeWord     <= (others => '0');
              CurrentValue        <= (others => '0');
              NewCodeWordReady    <= '0';
              AbsoluteValue       <= (others => '0');
              kLastBits           <= (others => '0');
              RleBits             <= (others => '0');
              RleLoop             <= 0;
              abs_bits_set        <= 0;
              kBitsSet            <= 0;
              ErrorOut            <= '0';
              CurrentResidual     <= (others => '0');
              AllCurrentResiduals <= (others => '0');
              ResidualBitCounter  <= 0;
              SampleCounter       <= 0;
              ReadyToEncode       <= '0';

              -- when new data is available go to reciving
                 state
              if (NewKIn = '1') and (NewResidualsIn = '1')
                 then
                 ReadyToEncode <= '1';
                 state <= Reciving;
                 k_valueOut <= kValueIn;
              end if;

           when Reciving =>
              --Indicate what the current state is in
                 simulation
              CheckState    <= 1;
```

```
            ReadyToEncode <= '0';

            -- load all residuals
            AllCurrentResiduals <= AllResidualsIn;

            state <= NewResidual;

         when NewResidual =>
            CheckState    <= 2;
            ReadyToEncode <= '0';

            CurrentValue     <= (others => '0');
            NewCodeWordReady <= '0';
            AbsoluteValue    <= (others => '0');
            kLastBits        <= (others => '0');
            RleBits          <= (others => '0');
            RleLoop          <= 0;
            abs_bits_set     <= 0;
            kBitsSet         <= 0;

            --testing
            LenCounter <= 0;
            CurrentCodeWord <= (others => '0');


            --Set the current residual from the datablock
               with all residuals
            CurrentResidual(24) <= AllCurrentResiduals(1599
                - ResidualBitCounter);
            CurrentResidual(23) <= AllCurrentResiduals(1598
                - ResidualBitCounter);
            CurrentResidual(22) <= AllCurrentResiduals(1597
                - ResidualBitCounter);
            CurrentResidual(21) <= AllCurrentResiduals(1596
                - ResidualBitCounter);
            CurrentResidual(20) <= AllCurrentResiduals(1595
                - ResidualBitCounter);
            CurrentResidual(19) <= AllCurrentResiduals(1594
                - ResidualBitCounter);
            CurrentResidual(18) <= AllCurrentResiduals(1593
                - ResidualBitCounter);
```

```
                CurrentResidual(17) <= AllCurrentResiduals(1592
                    - ResidualBitCounter);
                CurrentResidual(16) <= AllCurrentResiduals(1591
                    - ResidualBitCounter);
                CurrentResidual(15) <= AllCurrentResiduals(1590
                    - ResidualBitCounter);
                CurrentResidual(14) <= AllCurrentResiduals(1589
                    - ResidualBitCounter);
                CurrentResidual(13) <= AllCurrentResiduals(1588
                    - ResidualBitCounter);
                CurrentResidual(12) <= AllCurrentResiduals(1587
                    - ResidualBitCounter);
                CurrentResidual(11) <= AllCurrentResiduals(1586
                    - ResidualBitCounter);
                CurrentResidual(10) <= AllCurrentResiduals(1585
                    - ResidualBitCounter);
                CurrentResidual(9)  <= AllCurrentResiduals(1584
                    - ResidualBitCounter);
                CurrentResidual(8)  <= AllCurrentResiduals(1583
                    - ResidualBitCounter);
                CurrentResidual(7)  <= AllCurrentResiduals(1582
                    - ResidualBitCounter);
                CurrentResidual(6)  <= AllCurrentResiduals(1581
                    - ResidualBitCounter);
                CurrentResidual(5)  <= AllCurrentResiduals(1580
                    - ResidualBitCounter);
                CurrentResidual(4)  <= AllCurrentResiduals(1579
                    - ResidualBitCounter);
                CurrentResidual(3)  <= AllCurrentResiduals(1578
                    - ResidualBitCounter);
                CurrentResidual(2)  <= AllCurrentResiduals(1577
                    - ResidualBitCounter);
                CurrentResidual(1)  <= AllCurrentResiduals(1576
                    - ResidualBitCounter);
                CurrentResidual(0)  <= AllCurrentResiduals(1575
                    - ResidualBitCounter);
                -- Residual bits can never be larger than 1575,
                    since that gives a negative argument to
                   AllCurrentResiduals
                if ResidualBitCounter < 1551 then
                   --increment ResidualBitCounter so that the
```

```
                next residuals will be grabbed from
                allresiduals when this state is called
                next time
            ResidualBitCounter <= ResidualBitCounter +
                25;

        else
            state <= ErrorState;

        end if;

        --for adjacent each datablock should have 64
            residuals, one for every mic in the array
        if SampleCounter < 64 then
            --increment the sample counter since 1 more
                residual have been grabbed from all
                residuals
            SampleCounter <= SampleCounter + 1;

        else
            state <= ErrorState;

        end if;
        --when the current residual is grabbed set the
            value to encode to the current residual
        state <= SetValue;

    when SetValue =>
        CheckState    <= 3;
        ReadyToEncode <= '0';

        CurrentValue <= signed(CurrentResidual);

        state <= SignBit;

    when SignBit =>
        --Indicate what the current state is in
            simulation
        CheckState    <= 4;
        ReadyToEncode <= '0';
```

```
            --Check if CurrentValue is positive or negative
               ,
            --this decides the sign bit (MSB) for the
               codeword
            if (to_integer(CurrentValue)) < 0 then
               CurrentCodeWord(1023 - LenCounter) <= '1';

            else
               CurrentCodeWord(1023 - LenCounter) <= '0';

            end if;
            --LenCounter cant be larger than 1023, that is
               the maximum codeword length
            if LenCounter < 1023 then
               --Increment LenCounter by 1, because 1 bit
                  have been set in the codeword
               LenCounter <= LenCounter + 1;
            else
               state <= ErrorState;
            end if;

            --Store the absolute value of CurrentCodeWord
            AbsoluteValue <= std_logic_vector(abs(
               CurrentValue));
            state         <= FindKLastBits;

         when FindKLastBits =>
            --Indicate what the current state is in
               simulation
            CheckState    <= 5;
            ReadyToEncode <= '0';

            --set bits so that the k-last bits are saved to
                later be used in the codeword
            if abs_bits_set < k_val_int then
               --set the k last bits in the AbsoluteValeue
                  to the kLastBits variable, later to be
                  the k last bits of the codeword
               kLastBits(abs_bits_set) <= AbsoluteValue(
                  abs_bits_set);
               abs_bits_set            <= abs_bits_set + 1;
```

```
            else

               --once all bits have been stored go to the
                  next state
               state <= FindRleCode;
            end if;

         when FindRleCode =>
            --Indicate what the current state is in
               simulation
            CheckState    <= 6;
            ReadyToEncode <= '0';

            --set the remaining bits in Absolute value to
               the LSB:s in RleBits to later Run length
               encode the resulting value
            if abs_bits_set < 25 then
               RleBits(abs_bits_set - k_val_int) <=
                  AbsoluteValue(abs_bits_set);

               abs_bits_set <= abs_bits_set + 1;

            else

               if to_integer(unsigned(RleBits)) > 1023 then
                  --if the int value of RleBits is larger
                     than 1023 the codeword will be to long
                  state <= ErrorState;
               else

                  --Set the RleBits value as the integer
                     indicating how many ones needs to be
                     Run length encoded
                  RleLoop <= to_integer(unsigned(RleBits));
                  state   <= RleCoding;
               end if;
            end if;

         when RleCoding =>
            --Indicate what the current state is in
```

```
            simulation
        CheckState    <= 7;
        ReadyToEncode <= '0';

        --If RleLoop is larger than 0, append a 1 to
            the codeword and decrement RleLoop value
        if RleLoop > 0 then
            CurrentCodeWord(1023 - LenCounter) <= '1';
            RleLoop                            <=
                RleLoop - 1;

            --LenCounter cant be larger than 1023, that
                is the maximum codeword length
            if LenCounter < 1023 then
                --Increment LenCounter by 1, because 1
                    bit have been set in the codeword
                LenCounter <= LenCounter + 1;
            else
                state <= ErrorState;
            end if;

            --Else the RleCode is set,
            --set the next bit in the codeword to 0 to
                indicate that the RLE part of the code
                word is done
        else
            CurrentCodeWord(1023 - LenCounter) <= '0';

            --LenCounter cant be larger than 1023, that
                is the maximum codeword length
            if LenCounter < 1023 then
                --Increment LenCounter by 1, because 1
                    bit have been set in the codeword
                LenCounter <= LenCounter + 1;
            else
                state <= ErrorState;
            end if;
            --go to the next state once RLE coding is
                done

            state <= SetKLastBits;
```

```
            end if;

        when SetKLastBits =>
            --Indicate what the current state is in
               simulation
            CheckState    <= 8;
            ReadyToEncode <= '0';

            --set the k-last bits of the code word
            if kBitsSet < k_val_int then
               --set one bit for each loop until all k last
                   bits are set

               CurrentCodeWord(1023 - LenCounter) <=
                  kLastBits(k_val_int - 1 - kBitsSet);
               --LenCounter cant be larger than 1023, that
                  is the maximum codeword length
               if LenCounter < 1023 then
                  --Increment LenCounter by 1, because 1
                     bit have been set in the codeword
                  LenCounter <= LenCounter + 1;
                  -- if kBitsSet = k_val_int the codeword
                     is done and the length is precisly
                     enough to hold it,
                  -- but if it is not equal more bits need
                     to be set and the codeword wont be
                     able to fit it
               elsif kBitsSet /= k_val_int then
                  state <= ErrorState;
               end if;

               --increment kBitsSet to set the next bit
               kBitsSet <= kBitsSet + 1;
            else
               --The codeword is now finalised,
               --set the output variables for codeword and
                  codeword length
               CodeWord    <= CurrentCodeWord;
               CodeWordLen <= std_logic_vector(to_unsigned(
                  LenCounter, 10));
```

```
            state <= Sending;

         end if;

      when Sending =>
         --Indicate what the current state is in
            simulation
         CheckState    <= 9;
         ReadyToEncode <= '0';

         --set NewDataOut to 1 to indicate that new data
             is ready to be sent
         NewCodeWordReady <= '1';

         --wait until CodeWordAssembler is ready to
            recive the codeword beffore going to the
            next state
         if ReadytoReciveCodeWordIn = '1' then
            --If samplecounter = 64 all residuals have
               been looked at in the datablock, and it
               is ready to recive a new datablock
            if SampleCounter = 64 then
               state <= Idle;

               --Else the next residual in the current
                  datablock will be grabbed
            else
               state <= NewResidual;

            end if;
         end if;

      when ErrorState =>
         --added State to handle errors
         CHeckState <= 10;
         ErrorOut   <= '1';
   end case;
   if reset = '1' then

      if (to_integer(unsigned(kValueIn)) < 8) or (
```

```
                to_integer(unsigned(kValueIn)) > 22) then
                --k_value have to be atleast 8,
                --to limit max lenght of code words in niche
                   cases
                state <= ErrorState;

            else
                state     <= Idle;
                k_val_int <= to_integer(unsigned(kValueIn));
                k_pow     <= 2 ** to_integer(unsigned(kValueIn)
                   );

            end if;

            ErrorOut            <= '0';
            LenCounter          <= 0;
            CurrentCodeWord     <= (others => '0');
            CurrentValue        <= (others => '0');
            NewCodeWordReady    <= '0';
            abs_bits_set        <= 0;
            kBitsSet            <= 0;
            CurrentResidual     <= (others => '0');
            AllCurrentResiduals <= (others => '0');
            ResidualBitCounter  <= 0;
            SampleCounter       <= 0;
            ReadyToEncode       <= '0';
            k_valueOut <= "01000";

        end if;

    end if;

  end process;

end Behavioral;
```

A.3.4 CodeWordAssembler code

```
-- RiceEncoder

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity CodeWordAssembler is

   port (
      reset              : in std_logic;
      clk                : in std_logic;
      CodeWordIn         : in std_logic_vector(1023 downto 0);
         --CodeWord
      CodeWordLenIn      : in std_logic_vector(9 downto 0);
      k_valueIn          : in std_logic_vector(4 downto 0);--
         Saved as metadata
      NewCodeWordReadyIn : in std_logic;--New codeword
         available for assemble
      DataSavedIn        : std_logic;

      AllCodeWordsOut          : out std_logic_vector(8191
         downto 0);--All codewords together, length is 2^13 -
         1
      AssembleDoneOut          : out std_logic;--New CodeWord
         ready to be sent
      ReadyToReciveCodeWordOut : out std_logic;--Indicates
         that it is ready to recive the next codeword to
         assemble
      AllCodeWordsLenOut       : out std_logic_vector(12
         downto 0);


      ErrorOut                 : out std_logic--Send an error
         signal when error state is reached

   );

end entity;
```

```
architecture Behavioral of CodeWordAssembler is
   type state_type is (Idle, WaitForCodeWord, Reciving,
      LengthCheck, Assemble, SetMetaData, Sending, ErrorState)
      ; -- States for the statemachine
   signal state : state_type;
   --used in all states
   signal TotalLenCounter    : integer range 0 to 8191;--count
       the length off all the assembled codewords
   signal CurrentCodeWordLen : integer range 0 to 1023;--
      length of the current codeword
   signal AssembleLenCounter : integer range 0 to 1024;--count
       how many bits of the current codeword have been
      assembled
   signal SampleCounter      : integer range 0 to 256;--counts
       how many codewords have been recived
   signal SampleCounter_meta : std_logic_vector(7 downto 0);--
      SampleCounter as binary metadata
   signal CurrentCodeWord    : std_logic_vector(1023 downto 0)
      ;--set bits for current codeword
   signal k_meta             : std_logic_vector(4 downto 0);
   signal CheckState         : integer range 0 to 10;--to see
      what state is currently used for the simulation
begin
   process (clk)
   begin
      if rising_edge(clk) then

         case state is
            when Idle =>
               CheckState <= 0;
               --Reset all values
               AllCodeWordsOut    <= (others => '0');
               AllCodeWordsLenOut <= (others => '0');
               ErrorOut           <= '0';
               AssembleDoneOut    <= '0';
               TotalLenCounter    <= 0;
               CurrentCodeWordLen <= 0;
               AssembleLenCounter <= 0;
               CurrentCodeWord    <= (others => '0');
               k_meta             <= (others => '0');
               SampleCounter      <= 0;
```

```
            SampleCounter_meta <= (others => '0');
            if NewCodeWordReadyIn = '1' then
               state                    <= Reciving;


            end if;

            ReadyToReciveCodeWordOut <= '1';

         when WaitForCodeWord =>
            CheckState <= 1;

            if NewCodeWordReadyIn = '1' then
               state                    <= Reciving;

            end if;

            ReadyToReciveCodeWordOut <= '1';


         when Reciving =>
            CheckState               <= 2;
            ReadyToReciveCodeWordOut <= '0';
            --if it is the first codeword to be assembled
               the k_value is saved to later be used as
               meta data
            if TotalLenCounter = 0 then
               k_meta <= k_valueIn;
            end if;

            --Grab the current recived CodeWord
            CurrentCodeWord <= CodeWordIn;
            --Reset assembled bits counter
            AssembleLenCounter <= 0;
            --Grab the length of the current codeword
            CurrentCodeWordLen <= to_integer(unsigned(
               CodeWordLenIn));
            state              <= LengthCheck;

            --length check state to see that
               currentcodeword fits, totallen + currentlen
```

```
                < 6143

            when LengthCheck =>
                CheckState <= 3;
                --If adding the new codeword to the assembled
                    codewords creates a to long vector the
                    codeword can not be added
                --The metadata is instead added on and the
                    assembled codeword can be sent
                if (CurrentCodeWordLen + TotalLenCounter) >
                    8191 then
                    --SampleCounter is decremented by 1, so that
                        it can be saved as a 8 bit value (since
                        0 samples will never be sent anyway)
                    if SampleCounter = 0 then
                        state <= ErrorState;

                    else
                        SampleCounter_meta <= std_logic_vector(
                            to_unsigned(SampleCounter - 1, 8));
                        state              <= SetMetaData;

                    end if;

                else
                    --The codeword can be added and the
                        Samplecounter is incremented by 1
                    SampleCounter <= SampleCounter + 1;
                    state         <= Assemble;

                end if;

            when Assemble =>
                CheckState <= 4;
                --if AssembleLenCounter is larger than
                    CurrentCodeWordLen all bits in the codeword
                    have been assembled
                if AssembleLenCounter < CurrentCodeWordLen then
                    --Assemble the codewords with eachother in "
                        AllCodeWords"
                    --first 13 bits are for metadata
```

```
                AllCodeWordsOut(8178 - TotalLenCounter) <=
                   CurrentCodeWord(1023 - AssembleLenCounter
                   );
                --Increment the assembleLenCOunter by 1
                AssembleLenCounter <= AssembleLenCounter +
                   1;

                --Increment the total length of all
                   codewords by 1
                TotalLenCounter <= TotalLenCounter + 1;

                --once all the bits in the current codeword
                   have been assembled check if all 256
                   samples have been recived
                --if the SampleCounter is less than 256
                   recive another sample
             elsif SampleCounter < 64 then
                state <= WaitForCodeWord;

                --if all the samples have been assembled set
                    the metadata
             else
                --SampleCounter is decremented by 1, so that
                    it can be saved as a 8 bit value (since
                   0 samples will never be sent anyway)
                if SampleCounter = 0 then
                   state <= ErrorState;
                else
                   SampleCounter_meta <= std_logic_vector(
                      to_unsigned(SampleCounter - 1, 8));
                   TotalLenCounter    <= TotalLenCounter +
                      13;--13 bits of meta data
                   state              <= SetMetaData;
                end if;
             end if;

          when SetMetaData =>
             CheckState <= 5;

             --Set total len for codeword out
             AllCodeWordsLenOut <= std_logic_vector(
```

```
                to_unsigned(TotalLenCounter, 13));

            --Set the k-value meta data
            AllCodeWordsOut(8191) <= k_meta(4);
            AllCodeWordsOut(8190) <= k_meta(3);
            AllCodeWordsOut(8189) <= k_meta(2);
            AllCodeWordsOut(8188) <= k_meta(1);
            AllCodeWordsOut(8187) <= k_meta(0);

            --Set the SampleCounter metadata
            AllCodeWordsOut(8186) <= SampleCounter_meta(7);
            AllCodeWordsOut(8185) <= SampleCounter_meta(6);
            AllCodeWordsOut(8184) <= SampleCounter_meta(5);
            AllCodeWordsOut(8183) <= SampleCounter_meta(4);
            AllCodeWordsOut(8182) <= SampleCounter_meta(3);
            AllCodeWordsOut(8181) <= SampleCounter_meta(2);
            AllCodeWordsOut(8180) <= SampleCounter_meta(1);
            AllCodeWordsOut(8179) <= SampleCounter_meta(0);
            state                 <= Sending;

         when Sending =>
            CheckState      <= 6;
            AssembleDoneOut <= '1';
            if DataSavedIn = '1' then
               state <= Idle;
            end if;

         when ErrorState =>
            CheckState <= 10;
            ErrorOut   <= '1';
      end case;

      if reset = '1' then
         state                  <= Idle;
         AllCodeWordsOut        <= (others => '0');
         AllCodeWordsLenOut     <= (others => '0');
         ErrorOut               <= '0';
         AssembleDoneOut        <= '0';
         TotalLenCounter        <= 0;
         CurrentCodeWordLen     <= 0;
         AssembleLenCounter     <= 0;
```

```
            CurrentCodeWord           <= (others => '0');
            SampleCounter             <= 0;
            SampleCounter_meta        <= (others => '0');
            k_meta                    <= (others => '0');
            ReadyToReciveCodeWordOut  <= '0';

         end if;

      end if;

   end process;

end Behavioral;
```

A.3.5 Tentbench code

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

library vunit_lib;
context vunit_lib.vunit_context;

use work.MATRIX_TYPE.all;

entity tb_adjacent_combined is
   generic (
      runner_cfg : string
   );

end tb_adjacent_combined;

architecture adjacent_combined_arch of tb_adjacent_combined is

   --General signals used in all entitys
   constant C_SCK_CYKLE : time      := 8 ns; -- 125 MHz
   signal clk_tb        : std_logic := '0';
   signal reset_tb      : std_logic := '1';

   --Used for LoadData(Only testbench no entity)
   signal CurrentValue_tb  : std_logic_vector(23 downto 0);
   signal SampleCounter_tb : integer range 0 to 65535;
   signal MicCounter_tb    : integer range 0 to 65535;
   signal InputCounter_tb  : integer range 0 to 1535;

   signal EndOfFileReached_tb : std_logic;

   signal DatablockOut_tb          : std_logic_vector(1535
      downto 0);
   signal DataBlockReadyOut_tb     : std_logic;
   signal ResidualsCalculatedIn_tb : std_logic;

   --Used for SaveToFile(Only testbench no entity)
```

```
signal NewRow_tb       : std_logic;
signal BottomLimit_tb : integer range 0 to 8191;
signal RowCounter_tb   : integer range 0 to 65535;
signal AllDone         : std_logic;
signal LengthError     : std_logic;

--Used in AdjacentResiduals
signal DataBlockIn_tb       : std_logic_vector(1535 downto
   0);
signal ReadyToEncodeIn_tb   : std_logic;
signal DataBlockReadyIn_tb : std_logic;
signal ReadyToReciveIn_tb   : std_logic;

signal ResidualsCalculatedOut_tb : std_logic;
signal ResidualValueOut_tb       : std_logic_vector(24
   downto 0);
signal NewDataOut_tb             : std_logic;
signal AllResidualsOut_tb        : std_logic_vector(1599
   downto 0);
signal NewResidualOut_tb         : std_logic;

signal ErrorOut_Adjacent_tb : std_logic;

--Used in kCalculator
signal ResidualValueIn_tb            : std_logic_vector(24
   downto 0);--value in to be encoded
signal NewDataIn_tb                  : std_logic;--New data
    available for input
signal ReadyToEncodeInkCalculator_tb : std_logic;

signal ReadyToReciveOut_tb : std_logic;
signal NewKOut_tb          : std_logic;--New k-value to be
   sent
signal ErrorOut_tb         : std_logic;--Send an error
   signal when error state is reached
signal kValueOut_tb        : std_logic_vector(4 downto 0);

signal ErrorOut_kCalculator_tb : std_logic;

--Used in RiceEncode
signal AllResidualsIn_tb          : std_logic_vector(1599
```

```
    downto 0);
signal kValueIn_tb                : std_logic_vector(4
    downto 0);
signal NewKIn_tb                  : std_logic;--New data
    available for input
signal NewResidualsIn_tb          : std_logic;
signal ReadytoReciveCodeWordIn_tb : std_logic;

signal NewCodeWordReady_tb : std_logic;--New CodeWord ready
     to be sent
signal CodeWordLen_tb      : std_logic_vector(9 downto 0);
    --Bits used in the codeword
signal CodeWord_tb         : std_logic_vector(1023 downto
    0);--CodeWord out
signal ReadyToEncode_tb    : std_logic;
signal k_valueOut_tb       : std_logic_vector(4 downto 0);

signal ErrorOut_RiceEncode_tb : std_logic;

--Used in CodeWordAssembler
signal CodeWordIn_tb         : std_logic_vector(1023 downto
     0);--CodeWord
signal CodeWordLenIn_tb      : std_logic_vector(9 downto 0)
    ;
signal k_valueIn_tb          : std_logic_vector(4 downto 0)
    ;--Saved as metadata
signal NewCodeWordReadyIn_tb : std_logic;--New codeword
    available for assemble
signal DataSavedIn_tb        : std_logic;

signal AllCodeWordsOut_tb          : std_logic_vector(8191
    downto 0);--All codewords together, length is 2^13 - 1
signal AssembleDoneOut_tb          : std_logic;--New
    CodeWord ready to be sent
signal ReadyToReciveCodeWordOut_tb : std_logic;
signal AllCodeWordsLenOut_tb       : std_logic_vector(12
    downto 0);

signal ErrorOut_CodeWordAssmbler_tb : std_logic;

--Signals used in testbench
```

```
   signal InputState_tb  : std_logic_vector(1 downto 0);
   signal OutputState_tb : std_logic_vector(1 downto 0);
   signal CreateInput_tb : std_logic_vector(23 downto 0);

begin
   clk_tb <= not(clk_tb) after C_SCK_CYKLE/2;

   reset_tb <= '1', '0' after 35 ns;

   --AdjacentResiduals entity
   AdjacentResidualsTest : entity work.AdjacentResiduals
      port map(
         clk   => clk_tb,
         reset => reset_tb,

         DataBlockIn      => DataBlockIn_tb,
         ReadyToEncodeIn  => ReadyToEncodeIn_tb,
         DataBlockReadyIn => DataBlockReadyIn_tb,
         ReadyToReciveIn  => ReadyToReciveIn_tb,

         ResidualsCalculatedOut => ResidualsCalculatedOut_tb,
         ResidualValueOut       => ResidualValueOut_tb,
         NewDataOut             => NewDataOut_tb,
         AllResidualsOut        => AllResidualsOut_tb,
         NewResidualOut         => NewResidualOut_tb,

         ErrorOut => ErrorOut_Adjacent_tb
      );

   --kCalculator entity
   kCalculatorTest : entity work.kCalculator
      port map(
         reset                      => reset_tb,
         clk                        => clk_tb,
         ResidualValueIn            => ResidualValueIn_tb,
         NewDataIn                  => NewDataIn_tb,
         ReadyToEncodeInkCalculator =>
            ReadyToEncodeInkCalculator_tb,

         ReadyToReciveOut => ReadyToReciveOut_tb,
         NewKOut          => NewKOut_tb,
```

```
        kValueOut         => kValueOut_tb,

        ErrorOut => ErrorOut_kCalculator_tb
    );

--RiceEncode entity
RiceEncoderTest : entity work.NewRiceEncoder
    port map(
        reset                   => reset_tb,
        clk                     => clk_tb,
        AllResidualsIn          => AllResidualsIn_tb,
        kValueIn                => kValueIn_tb,
        NewResidualsIn          => NewResidualsIn_tb,
        NewKIn                  => NewKIn_tb,
        ReadytoReciveCodeWordIn => ReadytoReciveCodeWordIn_tb
            ,

        NewCodeWordReady => NewCodeWordReady_tb,
        CodeWordLen      => CodeWordLen_tb,
        CodeWord         => CodeWord_tb,
        ReadyToEncode    => ReadyToEncode_tb,
        k_valueOut       => k_valueOut_tb,

        ErrorOut => ErrorOut_RiceEncode_tb
    );

--CodeWordAssembler entity
TestCodeWordAssembler : entity work.CodeWordAssembler
    port map(
        reset => reset_tb,
        clk   => clk_tb,

        CodeWordIn         => CodeWordIn_tb,
        CodeWordLenIn      => CodeWordLenIn_tb,
        k_valueIn          => k_valueIn_tb,
        NewCodeWordReadyIn => NewCodeWordReadyIn_tb,
        DataSavedIn        => DataSavedIn_tb,

        AllCodeWordsOut          => AllCodeWordsOut_tb,
        AssembleDoneOut          => AssembleDoneOut_tb,
        ReadyToReciveCodeWordOut =>
```

```
            ReadyToReciveCodeWordOut_tb,
        AllCodeWordsLenOut       => AllCodeWordsLenOut_tb,

        ErrorOut => ErrorOut_CodeWordAssmbler_tb
    );
ws_process_12345 : process (clk_tb)

    --For reading data
    variable text_line : line;
    variable row_value : std_logic_vector(23 downto 0);
    --Make sure this directiory is correct!
    --file my_file : text open read_mode is "/home/toad/
       Projects/FPGA-sampling2/pl/test/first_sample_binary.
       txt";--First sample for mic 64-127
    --file my_file : text open read_mode is "/home/toad/
       Projects/FPGA-sampling2/pl/test/data_block_binary.txt
       ";--First datablock, all 256 samples for mic 64-12
    file my_file : text open read_mode is "C:\\Users\\axelo
       \\OneDrive\\Skrivbord\\Exjobb\\GIT\\SoundData\\VHDL\\
       data_block_binary.txt";--First datablock, all 256
       samples for mic 64-12

    --For saving data
    --Importent to check path /My/Path/MyFileName.txt
    --file file_handler     : text open write_mode is "/home
       /toad/Projects/FPGA-sampling2/pl/test/EncodedData.txt
       ";
    file file_handler     : text open write_mode is "C:\\
       Users\\axelo\\OneDrive\\Skrivbord\\Exjobb\\GIT\\
       SoundData\\VHDL\\EncodedData.txt";
    variable row          : line;
    variable v_data_write : std_logic_vector(8191 downto 0);
begin

    if falling_edge(clk_tb) then

        if InputState_tb = "01" then
            --This is the starting state when grabbing a
               datablock
            DatablockOut_tb      <= (others => '0');
            DataBlockReadyOut_tb <= '0';
```

```
        CurrentValue_tb <= (others => '0');
        MicCounter_tb   <= 0;
        InputCounter_tb <= 0;

        InputState_tb <= "10";

    elsif InputState_tb = "10" then
        --check that it is not end of file
        if not endfile(my_file) then
            --point to the next line
            readline(my_file, text_line);
            --read the current lint
            read(text_line, row_value);


            --assign CurrentValue_tb with the read value
               from text file
            CurrentValue_tb <= row_value;-- only to see
               current value

            --set input values for testbench
            DataBlockOut_tb(1535 - InputCounter_tb) <=
               row_value(23);
            DataBlockOut_tb(1534 - InputCounter_tb) <=
               row_value(22);
            DataBlockOut_tb(1533 - InputCounter_tb) <=
               row_value(21);
            DataBlockOut_tb(1532 - InputCounter_tb) <=
               row_value(20);
            DataBlockOut_tb(1531 - InputCounter_tb) <=
               row_value(19);
            DataBlockOut_tb(1530 - InputCounter_tb) <=
               row_value(18);
            DataBlockOut_tb(1529 - InputCounter_tb) <=
               row_value(17);
            DataBlockOut_tb(1528 - InputCounter_tb) <=
               row_value(16);
            DataBlockOut_tb(1527 - InputCounter_tb) <=
               row_value(15);
            DataBlockOut_tb(1526 - InputCounter_tb) <=
```

```
            row_value(14);
        DataBlockOut_tb(1525 - InputCounter_tb) <=
            row_value(13);
        DataBlockOut_tb(1524 - InputCounter_tb) <=
            row_value(12);
        DataBlockOut_tb(1523 - InputCounter_tb) <=
            row_value(11);
        DataBlockOut_tb(1522 - InputCounter_tb) <=
            row_value(10);
        DataBlockOut_tb(1521 - InputCounter_tb) <=
            row_value(9);
        DataBlockOut_tb(1520 - InputCounter_tb) <=
            row_value(8);
        DataBlockOut_tb(1519 - InputCounter_tb) <=
            row_value(7);
        DataBlockOut_tb(1518 - InputCounter_tb) <=
            row_value(6);
        DataBlockOut_tb(1517 - InputCounter_tb) <=
            row_value(5);
        DataBlockOut_tb(1516 - InputCounter_tb) <=
            row_value(4);
        DataBlockOut_tb(1515 - InputCounter_tb) <=
            row_value(3);
        DataBlockOut_tb(1514 - InputCounter_tb) <=
            row_value(2);
        DataBlockOut_tb(1513 - InputCounter_tb) <=
            row_value(1);
        DataBlockOut_tb(1512 - InputCounter_tb) <=
            row_value(0);

        if InputCounter_tb < 1489 then

            --increment InputCounter_tb to set the nxt
                24 values for the next mic
            InputCounter_tb <= InputCounter_tb + 24;
        end if;

        --increment mic
        if MicCounter_tb < 63 then
            --if all 63 mics have not been set grab the
                next mic value
```

```
                MicCounter_tb <= MicCounter_tb + 1;
                InputState_tb <= "10";

            else
                InputState_tb <= "00";

                --increment the sample counter
                SampleCounter_tb <= SampleCounter_tb + 1;

                --Set DataBlockReadyOut_tb to high to
                   indicate that the datablock is done
                DataBlockReadyOut_tb <= '1';

            end if;

        else
            EndOfFileReached_tb <= '1';
            InputState_tb <= "11";

        end if;

    elsif InputState_tb = "00" then

        if ResidualsCalculatedIn_tb = '1' then
            --once AdjacentResidual have recived the
               datablock (ResidualsCalculatedIn_tb is high)
            --start createing the next datablock
            InputState_tb <= "01";

        end if;

    end if;



    if OutputState_tb <= "01" then

        DataSavedIn_tb <= '1';

        --Once the full codeword is assembled and ready to
            be sent the next state is entered
```

```
            if AssembleDoneOut_tb = '1' then
               --store All CodeWords and the length of them
               v_data_write := AllCodeWordsOut_tb;
               if to_integer(unsigned(AllCodeWordsLenOut_tb))
                  > 0 then
                  BottomLimit_tb <= to_integer(unsigned(
                     AllCodeWordsLenOut_tb) - 1);

               else

                  BottomLimit_tb <= to_integer(unsigned(
                     AllCodeWordsLenOut_tb));
                  LengthError <= '1';
               end if;

               OutputState_tb <= "10";

            end if;
         elsif OutputState_tb <= "10" then

            --This indicated that the codeword can be saved
               and the next assemble can start
            DataSavedIn_tb <= '0';
            if NewRow_tb = '0' then
               write(row, v_data_write(8191 downto 8191 -
                  BottomLimit_tb));
               NewRow_tb     <= '1';
               RowCounter_tb <= RowCounter_tb + 1;

            else
               writeline(file_handler, row);
               NewRow_tb <= '0';

               OutputState_tb <= "11";

            end if;

         elsif OutputState_tb <= "11" then
            --Check if all 256 samples have been written
            if RowCounter_tb < 256 then
               OutputState_tb <= "01";
```

```
        else
            OutputState_tb <= "00";

        end if;

    elsif OutputState_tb = "00" then
        report "I am done!";
        AllDone <= '1';
        file_close(file_handler);


    end if;

    if reset_tb = '1' then

        --Tesbench signals initial values
        InputState_tb            <= "01";
        OutputState_tb           <= "01";
        CreateInput_tb           <= (others => '0');
        InputCounter_tb          <= 0;
        ResidualsCalculatedIn_tb <= '0';

        --LoadData initial values
        DatablockOut_tb      <= (others => '0');
        DataBlockReadyOut_tb <= '0';

        CurrentValue_tb  <= (others => '0');
        SampleCounter_tb <= 0;
        MicCounter_tb    <= 0;
        InputCounter_tb  <= 0;

        EndOfFileReached_tb <= '0';

        --SaveToFile initial values
        NewRow_tb      <= '0';
        AllDone        <= '0';
        RowCounter_tb  <= 0;
        BottomLimit_tb <= 0;
        DataSavedIn_tb <= '0';
        LengthError <= '0';
```

```
        --AdjacentResidual initial values
        DataBlockIn_tb      <= (others => '0');
        ReadyToEncodeIn_tb  <= '0';
        DataBlockReadyIn_tb <= '0';
        ReadyToReciveIn_tb  <= '0';

        --kCalculator initial values
        ResidualValueIn_tb            <= (others => '0');
        NewDataIn_tb                  <= '0';
        ReadyToEncodeInkCalculator_tb <= '0';

        --RiceEncode initial values
        AllResidualsIn_tb          <= (others => '0');
        kValueIn_tb                <= "01000";
        NewKIn_tb                  <= '0';
        NewResidualsIn_tb          <= '0';
        ReadytoReciveCodeWordIn_tb <= '0';

        --CodeWordAssembler initial values
        CodeWordIn_tb         <= (others => '0');
        CodeWordLenIn_tb      <= (others => '0');
        k_valueIn_tb          <= "01000";
        NewCodeWordReadyIn_tb <= '0';
        DataSavedIn_tb        <= '0';

        --in the else statement inputs for entitys that
           are driven by outputs from other entitys are
           specified
    else

        --LoadData
        ResidualsCalculatedIn_tb <=
           ResidualsCalculatedOut_tb;

        --AdjacentResidual
        ReadyToEncodeIn_tb  <= ReadyToEncode_tb;
        ReadyToReciveIn_tb  <= ReadyToReciveOut_tb;
        DataBlockIn_tb      <= DatablockOut_tb;
        DataBlockReadyIn_tb <= DataBlockReadyOut_tb;
```

```
            --kCalculator
            ResidualValueIn_tb             <=
               ResidualValueOut_tb;
            NewDataIn_tb                   <= NewDataOut_tb;
            ReadyToEncodeInkCalculator_tb <= ReadyToEncode_tb;

            --RiceEncode
            AllResidualsIn_tb          <= AllResidualsOut_tb;
            kValueIn_tb                <= kValueOut_tb;
            NewKIn_tb                  <= NewKOut_tb;
            NewResidualsIn_tb          <= NewResidualOut_tb;
            ReadytoReciveCodeWordIn_tb <=
               ReadyToReciveCodeWordOut_tb;

            --CodeWordAssembler
            CodeWordIn_tb         <= CodeWord_tb;
            CodeWordLenIn_tb      <= CodeWordLen_tb;
            k_valueIn_tb          <= k_valueOut_tb;
            NewCodeWordReadyIn_tb <= NewCodeWordReady_tb;

         end if;

      end if;
   end process;

   main_1234 : process
   begin
      test_runner_setup(runner, runner_cfg);
      while test_suite loop
         if run("wave") then
            -- test 1 is so far only meant for gktwave

            --wait for 2950000ns; -- duration for a full
               datablock
            --wait for 30000 ns;

            wait until AllDone = '1' for 6 ms;
            assert AllDone = '1' report"AllDone should be set
               HIGH" severity failure;

         elsif run("auto") then
```

```
            wait for 11 ns;

        end if;
    end loop;

    test_runner_cleanup(runner);
  end process;

  test_runner_watchdog(runner, 100 ms);
end architecture;
```

A.4 GIThub repository

GIThub repository link:
https://github.com/AxelOstfeldt/Examensarbete