Lossless compression and coding of audio signals

Author: Christos Konstantas

Supervising Professor: Professor George Karystinos

Examination Committee: Professor Aggelos Bletsas

Examination Committee: Professor Thrasyvoulos Spyropoulos

Technical University of Crete, Department of Electrical and Computer Engineering

July 28, 2022

Contents

Acknowledgments

First of all, I would like to thank my parents, George and Paschalina, for their continuous support, trust and for shaping me to who I am right now. I am most thankful for having them by my side and I would never try to change them in any way.

I would like to specially thank my supervising Professor George Karystinos for his guidance and for letting me pursue a thesis that interests me. Also, I would like to thank the examination commitees, Professor Aggelos Bletsas and Professor Thrasyvoulos Spyropoulos, for accepting to participate in my thesis and sacrifice their time to study it, it is truly an honor for me.

I am also thankful to the PhD student Ioannis Grypiotis, for the helpful discussion that we had on my thesis.

It is also very important for me to give a respectable and big thank you to my high school math teacher Thomas Podimatas, for showing me the beauty of mathematics with his brilliant lectures, his perfect sense of humor and for helping me believe in myself.

Last, but of course not least, I would like to thank my friends that were next to me, not only at the best but also at the difficult times too during the years of my studies. I would never forget our adventures and our battles for the better.

This thesis is dedicated to the memory of my grandfather Christos and his brother Athanasios.

Abstract

This thesis is on lossless audio compression and coding that is very important for studio applications, such as digital audio signal processing, and is prefered by many artists around the world. We study and present an implementation of the IEEE 1857.2 standard and explicitly its lossless audio coding extension, that typically implements a very similar process of compression and coding for the audio signal with many other famous lossless audio compression standards (FLAC [1], MPEG-4 ALS [2], [3],etc). Many of these techniques, are being used on medical applications too [4], where the preservation of the original information of the source is extremely important. Last but not least, similar processes are being used for compressing losslessly optical satellite images. Another discussion is that in order to have a greater rate of transmission and reduce bandwidth in a communication system, lossless compression of the source is crucial. In the thesis, we present in detail the theoretic foundation of the linear predictive model, Arithmetic, Golomb-Rice, and exponential-Golomb coding, and IEEE-1857.2 defined pre- and post-processing techniques. We implement all above techniques and illustrate by compressing and coding some input uncompressed (raw) CD quality audio files (.wav files), achieving minimization of the number of bits while maintaining quality.

Chapter 1

Introduction

To represent digital audio data with the least bits possible, it is important to apply compression, i.e. detect and remove any redundancies that are contained in the uncompressed input audio files. The software (or hardware) that does this process is often refered to as an audio codec. A codec applies compression of the input data in order to achieve faster rate of transmission and is capable to decompress the received data too. A lot of audio codecs are being used everyday by many people around the world, depending on their demandings. There are lossless and lossy audio formats. A famous lossy audio format is the well-known MP3. But with MP3 files, we get only an approximation of the input audio file that was originally compressed. In lossless audio formats though, the quality of the original input audio file is retained perfectly. The basic operations in most lossless audio compression algorithms (see [5]) are depicted in Figure 1.1.

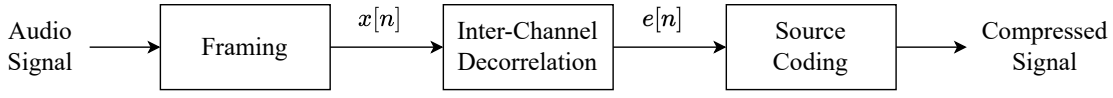


Figure 1.1: A general lossless audio compression scheme

- **Framing** is the process where the input audio file is being splitted into independent blocks that contain samples. This is done because we want to apply later a predictive model and in order to work with it, we must create frames of very small duration, so that we can consider them to be WSS stochastic processes (see Definition 2.1.6). In other words the nonstationary nature of audio signals does not help us to apply a predictive model, therefore we have to work somehow with WSS stochastic processes.

- **Inter-Channel Decorrelation** is the process where we detect and remove redundancies by decorrelating the samples that are stored inside a frame. In this stage we use a predictive model, usually a modified linear predictive model, for the input signal, resulting in a prediction error for every sample inside a frame and all of them form a sequence of prediction error samples. The redundancy that is removed from the signal is being represented by the predictor parameters and these predictor parameters together with the prediction error can represent the signal in each frame (or block).

- **Source Coding** as we will see later, it is used to efficiently encode the given prediction error sequence and represent it with the minimum number of bits.

Some lossless audio codecs that exist are the FLAC (Free Lossless Audio Codec) [1] that is the most famous one, the MPEG-4 ALS [3], Dolby TrueHD [6] and the most recent IEEE 1857.2 standard, with its lossless extension [7], on which we will primarily work on this thesis.

Thesis outline

In this thesis, we will start by some very important theoretical definitions that are useful in order to understand how the IEEE 1857.2 lossless audio coding extension works. We will start by explaining some basic definitions on stochastic processes and the Z-transform, that is the counterpart of Laplacian transform but for discrete-time signals, such as digital audio signals. Then we will proceed on a discussion about FIR and IIR filters and later on explaining a parametric signal modeling technique, that will be used as an autoregressive (AR) model, that is Linear Prediction and why it is important in order to remove redundancy of the source. We will also see, how to derive the Levinson-Durbin algorithm in order to achieve optimal Linear Prediction. After that, we will see some basic definitions on Information Theory that will be helpful in order to understand the optimality and unique decodability of the source coding techniques that we will implement using the IEEE 1857.2 lossless audio coding extension and will be applied in order to efficiently represent the output of the Linear Predictive Model. At last, we will see the basic components of the IEEE 1857.2 lossless audio coding extension and discuss the results on compression efficiency, that can be derived from finding the compression ratio for each type of souce coding.

Below, in Figure 1.2, we see a depiction of the processes of framing, prediction and entropy coding, i.e. source coding of the input audio signal:

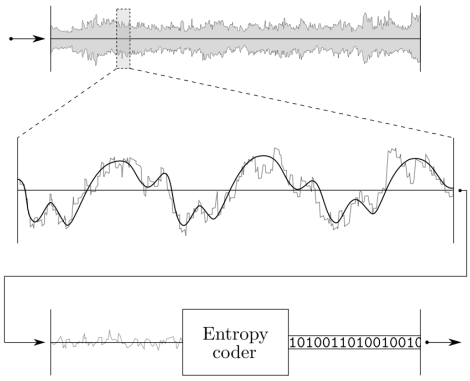


Figure 1.2: Framing, linear predictive modeling and entropy encoding processes. [1]

[1] Figure from [8]

Chapter 2

Theoretical prerequisites

This chapter contains some important definitions that are being used throughout this thesis. We start from scratch with some basic stochastic processes definitions. Afterwards we proceed to the elemental discrete-time signal and systems theory, the Z-Transform and we present some structures for discrete time systems that help a software or hardware designer to construct algorithms based on signal flow graphs and by using them we'll construct FIR and IIR filters that are useful for analysis and synthesis of a given signal.
If you are familiar with all of the above feel free to skip this chapter.

2.1 Stochastic Processes

Definition 2.1.1 (**Stochastic Process**)

In probability theory, a stochastic process (or a random process) is a mathematical concept that can be used as a system's mathematical model that seems to fluctuate randomly and it can be defined as a collection of many random variables. We can define a stochastic process as $\mathcal{X}(t) = \{X(t)\}_{t\in T}$ or as $\mathcal{X}(t) = \{X_t\}_{t\in T}$ ([9]) where , $X(t)$ or X_t are used in order to make a reference to the random variable that has an index t. There are two major categories in which a stochastic process can belong:

(i) Discrete-time stochastic process:

A countable collection of random variables can also be refered to as a discrete-time stochastic process. In this case we have a finite (or countable) number of elements that the index set of the stochastic process can take, specifically $t \in T \subseteq \mathbb{Z}$. If $T \subset \mathbb{Z}$ the stochastic process is finite and it can be fully defined by its joint probability density function, $f_{X[t_1],\ldots,X[t_n]}$ where $(t_1, \ldots, t_n) \in T^n$. When we refer to discrete-time stochastic processes we will use the symbolism $\mathcal{X}(t) = \{X[t]\}_{t\in T}$ with brackets instead of parenthesis.

(ii) Continuous-time stochastic process:

A noncountable infinite collection of random variables,one collection $\forall t$, can also be refered to as a continuous-time stochastic process with index t, where $t \in T \subseteq \mathbb{R}$. It is clear that we can not fully define a continuous-time stochastic process by its probability density function. The reason for this is that the number of random variables is a noncountable infinity. When we refer to continuous-time stochastic processes we will use the symbolism $\mathcal{X}(t) = \big(X(t)\big)_{t\in T}$.

Note: For this thesis we only consider the discrete-time case for stochastic processes. We do this because digital audio signals can be seen as random signals with discrete time instances, or equivalently, as discrete-time stochastic processes. In additions, the definitions used are strictly based on [10], [11] and [12].

Definition 2.1.2 (**Completely specified stochastic process**)

A stochastic process $\mathcal{X}(t)$ is completely specified if $\forall n \in \mathbb{N}$ where $(t_1, \ldots, t_n) \in T^n$ the joint probability density function of the random variables $X[t_1], \ldots, X[t_n]$ which is $f_{X[t_1],\ldots,X[t_n]}$ can be fully defined. For the statistical M-order description of a stochastic process one must know the underlying probability density function of the random variables $X[t_1], \ldots, X[t_n]$, $\forall n \leq M$ where $(t_1, \ldots, t_n) \in T^n$. Thus, the statistical M-order description may be enough for the complete specification of the stochastic process (if $n = M$) or may be not (if $n < M$).

Definition 2.1.3 (**Averaging Operator**)

We define an averaging operator of two random variables X and Y as:

$$\phi_{XY}(t_1, t_2) = \mathcal{E}\{X[t - t_1]Y[t - t_2]\} \tag{2.1.1}$$

If $Y = X$ it follows that

$$\phi_{XX}(t_1, t_2) = \mathcal{E}\{X[t]X[t - (t_1 - t_2)]\} \tag{2.1.2}$$

which can be proved by defining $t' = t - t_2$ and then $t = t'$.

Definition 2.1.4 (**Autocorrelation Function**)

The autocorrelation function (ACF) is one of the tools used to find patterns in the data and is defined as the statistical correlation between two time instances of a random variable. ACF can be written as:

$$R_{XX}(t_1, t_2) = \mathcal{E}\{X[t_1]X[t_2]\} \tag{2.1.3}$$

Definition 2.1.5 (**Strict Sense Stationarity**)

A stochastic process $\mathcal{X}(t)$ is called Strict Sense Stationary (SSS) if
$\forall n \in \mathbb{N}$, $(t_1, \ldots, t_n) \in T^n$ and $\forall a : t_i + a \in T$ where $i = 1, ..., n$:

$$f_{X[t_1],X[t_2],...,X[t_n]} = f_{X[t_{1+a}],X[t_{2+a}],...,X[t_{n+a}]} \tag{2.1.4}$$

This means that in order for a stochastic process to be Strict Sense Stationary (SSS) the joint probability density function of a finite subsequence of $\mathcal{X}(t)$ is time invariant. This statement is very restrictive and Strict Sense Stationarity is a strong property a stochastic process that is SSS implies that this process is completely specified, meaning that the probability density function of

the random variables of the process is fully defined. Also from (2.1.4) it follows that SSS stochastic processes have constant mean:

$$\mathcal{E}\{X[t_1], X[t_2], \ldots, X[t_n]\} = \mathcal{E}\{X[t_{1+a}], X[t_{2+a}], \ldots, X[t_{n+a}]\} \tag{2.1.5}$$

Definition 2.1.6 (**Wide Sense Stationarity**)

A stochastic process $\mathcal{X}(t)$ is called Wide Sense Stationary (WSS) if its autocorrelation function as long as its mean function do not change by time lags (or shifts in time). Therefore, the following conditions are true:

(i) The mean function is independent of the time instance $t \in T$. We can write this as $m_{\mathcal{X}}(t) = \mathcal{E}[\mathcal{X}(t)] = m_{\mathcal{X}}$. We can also say that (2.1.5) is true for this case.

(ii) The autocorrelation function (ACF) $R_{XX}(t_1, t_2) = \mathcal{E}\{X(t_1)X(t_2)\}$ (2.1.3) is only a function of $\tau = t_1 - t_2$, and not t_1 and t_2 individually. Thus, we can write

$$R_{XX}(t_1, t_2) = R_X(t_1 - t_2) = R_X(\tau). \tag{2.1.6}$$

And from (2.1.2) this is equivalent to:

$$\phi_{XX}(t_1, t_2) = R_X(t_1 - t_2) \tag{2.1.7}$$

Wide Sense Stationary (WSS) stochastic processes have more relaxed conditions than SSS stochastic processes. All stationary random processes are WSS but a WSS process is not always SSS. Intuitively, one can say that WSS processes take values close to their mean $m_{\mathcal{X}}$ and they are similar to their delayed versions. By convention, in audio processing literature ACF can be referred as the autocorrelation sequence.

Comments

- Later we will suppose that the audio signal is a collection of random variables also called a stochastic process, where the samples of the process that are close together are correlated in some way. Therefore, we can we can characterize short frames of the audio signal with this correlation.

- An audio signal can not be seen as a stochastic process that is SSS because we don't know its probability density function.

- Audio is a non-stationary signal, consequently its statistical characteristics are fluctuating over time. Therefore, its charachteristic properties and spectral attributes should only be derived from small duration frames of the signal ([13]) [14]. When we frame the audio signal ([14]), we consider each frame to be a WSS stochastic process because frames have a very small duration ($\approx$ 23.2 miliseconds), thus its statistical properties are constant within this time interval. For this reason we assume that our audio signal has constant mean, at least in each frame. This argument is intuitive and it does not contain any mathematical rigor.

2.2 Audio Signals

Definition 2.2.1 (**Sound Wave**)

A sound wave can be created by vibrations that travel through a transmission medium (sound travels through water, air, walls, steel etc; hence, all of them can be mediums for sound). The particles of sound waves can not be carried without a medium. Sound waves find an entrance from the outer ear and travel to the middle ear after passing the ear canal. Then, the sound waves are channeled to the eardrum by the auditory canal. The eardrum is a sensitive, attenuated membrane extended firmly over the way into the middle ear, and it vibrates when it receives a sound wave. Afterwards, these vibrations are passed from the eardrum on to the hammer, that is one of three little bones in the human ear. When the hammer vibrates causes the anvil to vibrate too, that is the small bone touching the hammer. These vibrations are passed from the anvil to the stirrup,

which is a tiny bone that touches the anvil, and from the stirrup into the inner ear. Also, the stirrup is connected with a liquid filled sack and the vibrations travel into a shell shaped part of the inner ear, the cochlea. The cochlea contains many special cells that are attached to nerve fibers and by them information can be transmitted through the auditory nerve to the brain. The brain manipulates the information that it gets from the ear and we can then distinguish many different types of sounds. An auditory perception in humans is elicited only by sound waves lying in the frequency range between 20 Hz and 20 kHz, .

Ultrasound waves are sound waves that lie above 20 kHz and are not audible to humans. Infrasound waves, are also not audible to humans, and lie below 20 Hz.

Definition 2.2.2 (**Analog audio signal**)

Analog audio signals are electrical voltages of alternating current (AC). Thus, we can express an analog audio signal as a function $f : \mathbb{R} \to \mathbb{R}$ which maps a time variable t to the voltage $f(t)$ at time t. The polarity of analog audio signals gets reversed from positive to negative and the number of these oscillations per second determines the frequency of the signal. The frequencies of audio signals have a range from 20Hz to 20kHz, which is the lower and upper limit respectively of human hearing and this happens after low-pass filtering (2.3). Though it's easiest to imagine a simple sine wave with smooth oscillations, most audio signals have more complex characteristics and they contain multiple frequencies and overtones. But frequency is only half of the equation, because we also consider the amplitude of the positive and negative peaks which is varying too. This property determines the level or volume (when amplified) of the signal, measured in decibels (dB). Frequency and amplitude allow analog systems to recreate complex sounds in the form of electrical signals with astonishing accuracy, and to convert them back again so we can hear them through speakers and headphones. Volume amplification is determined to be the amplitude of the analog audio signal peaks. Analog audio signals may be directly synthesized, or they may originate at a transducer such as a microphone, musical instrument, pickup, phonograph cartridge, or tape head. Loudspeakers or headphones can actually convert an electrical audio signal back into a sound. In computer arithmetic we can only do countable approximations of analog signals. This creates

the need to represent f in the digital domain. But how do we achieve that?

2.3 Pulse Code Modulation

Pulse Code Modulation, abbreviated as PCM, is a technique by which the analog audio signal gets converted into digital form (binary sequence) and is usually the standard representation format for digital audio signals (see [15], [16]). PCM allows the representation of the analog input signal as a sequence of binary coded pulses. The new digital signal represents the original signal where the set $\mathbb{Z}$ is the domain of the signal's function (discrete-time signal). PCM systems are basically signal coders also known as waveform coders, because they create a coded form of the original waveform.

The mission of a waveform coder is to allow the reproduction of the source's output waveform to its final destination (receiver) and simultaneously achieve minimum distortion of the waveform signal. Each digit in the binary sequence represents the approximate amplitude of the input signal, that is the amplitude of the quantized signal, at that instant. Using PCM, it is possible to digitize every kind of analog data, including voice, music, video, etc. PCM can be used for storage or transmission.

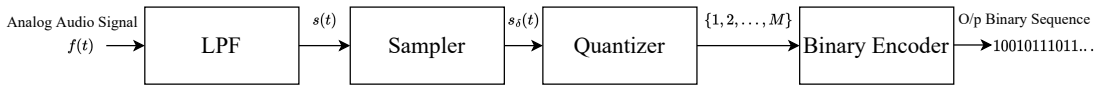


Figure 2.1: PCM encoder scheme

(1) Low Pass Filter (LPF)

At first the analog audio signal passes through a low-pass filter (LPF) (Figure 2.2). The LPF has a cutoff frequency f_c to eliminate the high-frequency components of the analog audio signal and allows only the frequency components that lie below f_c to pass. Since the human ear's perceptual range for pure tones is between 20 Hz and 20 kHz, the low-pass filter may be designed in such a way

that the cutoff frequency f_c starts at 20 kHz and a few kilohertz are allowed as the transition band before the stopband. This happens because we want to limit the input audio signal bandwidth.

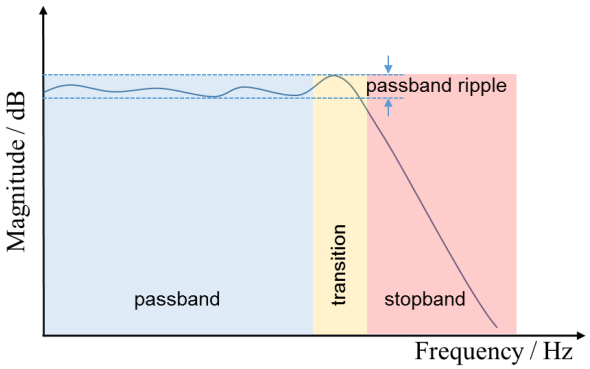


Figure 2.2: A typical low-pass filter (LPF) [1]

(2) Sampler

[18] The analog audio signal is sampled according to the Nyquist-Shannon sampling theorem, which states that if all frequencies in the signal are lower than $F_s/2$, where F_s is the sampling frequency, the continuous-time signal can be reconstructed from its discrete-time representation. To ensure this condition is satisfied, the input analog signal has been filtered with a low-pass filter whose stopband frequencies lie below than the half of the sample rate. For analog audio signals the sampling frequency is 44.1 kHz ($44.1 \cdot 10^3 \cdot \frac{samples}{second}$) which is greater than the Nyquist Frequency (40kHz= $2 \cdot 20$kHz).We choose this sampling rate of 44.1 kHz to prevent aliasing. Aliasing can happen if we choose $F_s < 2 \cdot Fmax$ or because the LPF is not ideal and there is a small range of frequencies in the stopband region implying that Fmax, which is the filtered signal's highest frequency, will be slightly greater than 20kHz. We sample the signal that is the output of the low-pass filter $s(t)$ instantaneously at a uniform rate, F_s, once every $T_s = 1/F_s$ seconds. Let $s_d[n] = s(nT_s)$

[1] Figure from [17]

and $s_\delta(t)$ be a continuous time signal. We want to construct:

$$s_\delta(t) = \sum_{n=-\infty}^{\infty} s_d[n]\delta(t - nT_s) \tag{2.3.1}$$

And in frequency domain (2.3.1) becomes:

$$S_\delta(F) = F_s \sum_{k=-\infty}^{\infty} S(F - kFs) \tag{2.3.2}$$

Proof: $s_\delta(t) \overset{\mathcal{F}}{\longleftrightarrow} S_\delta(F) \iff S_\delta(F) = \sum_{n=-\infty}^{\infty} s_d[n]\mathcal{F}\{\delta(t - nT_s)\} \iff$

$$S_\delta(F) = \sum_{n=-\infty}^{\infty} s_d[n]e^{-j2\pi(F/F_s)n} = S\left(\frac{F}{F_s}\right) \tag{2.3.3}$$

Where: $s_d(n) \overset{\mathcal{F}}{\longleftrightarrow} S_d(f)$ and $s(t) \overset{\mathcal{F}}{\longleftrightarrow} S(F)$. For a discrete-time signal we have:

$$S_d(f) = \sum_{n=-\infty}^{\infty} s_d[n]e^{-j2\pi fn} \tag{2.3.4}$$

$$s_d[n] = \int_{-1/2}^{1/2} S_d(f)e^{j2\pi fn}\, df \tag{2.3.5}$$

In addition $s_d[n] = s(nT_s) \iff s_d[n] = s\left(\frac{n}{F_s}\right) \iff \int_{-1/2}^{1/2} S_d(f)e^{j2\pi fn}\, df = \int_{-\infty}^{\infty} S(F)e^{j2\pi(F/F_s)n}\, dF$

$\overset{\text{f'=F/Fs}}{=} F_s \int_{-\infty}^{\infty} S(f'F_s)e^{j2\pi f'n}\, df'$

$\overset{\text{f=}f'}{=} F_s \int_{-\infty}^{\infty} S(fF_s)e^{j2\pi fn}\, df$

$= F_s \sum_{k=-\infty}^{\infty} \int_{k-1/2}^{k+1/2} S(fF_s)e^{j2\pi fn}\, df$

$\overset{\text{f'=}f-k}{=} F_s \sum_{k=-\infty}^{\infty} \int_{-1/2}^{1/2} S((f'+k)F_s)e^{j2\pi(f'+k)n}\, df'$

$\overset{\text{k'=}-k}{=} F_s \int_{-1/2}^{1/2} \sum_{k'=-\infty}^{\infty} S((f'-k')F_s)e^{j2\pi(f'-k')n}\, df'$

By noticing that $e^{-j2\pi k'n} = 1$ and defining $k' = k$, $f' = f$ we conclude to the equation:

$\int_{-1/2}^{1/2} S_d(f)e^{j2\pi fn}\, df = F_s \int_{-1/2}^{1/2} \sum_{k=-\infty}^{\infty} S((f-k)F_s)e^{j2\pi fn}\, df \iff$

$S_d(f) = F_s \sum_{k=-\infty}^{\infty} S((f-k)F_s)$ and if we define $f = \dfrac{F}{F_s}$ then:

$$S\left(\frac{F}{F_s}\right) = F_s \sum_{k=-\infty}^{\infty} S(F - kFs) \tag{2.3.6}$$

Now if we go backwards from (2.3.6) to (2.3.3) we can finally prove that the statement (2.3.2) is true. This is a very important equation because it proves that we can retrieve the spectrum $S(F)$ of the continuous-time signal $s(t)$ from the spectrum $S_\delta(f)$ of the discrete-time signal that we have constructed at (2.3.1), and if we sample at frequent time intervals ($F_s > 2{\cdot}F_{max}$) we can assume that there is no information loss during sampling.

Afterwards,in linear PCM or in non-linear PCM, the sampler's output signal must be quantized using a uniform quantization method or a non-uniform quantization method respectively.

(3) Quantizer

[19], [20] We define an M-point, one-dimension scalar quantizer Q as a mapping Q: $\mathbb{R} \to \mathcal{C}$, where the domain of Q is the real line, $\mathcal{C} = \{a_1, a_2, \ldots, a_M\} \subset \mathbb{Z}$ is a codebook with size M and a_j are the quantization region representation points, where $j \in \{1, \ldots, M\}$. The input value of the quantizer is the analog signal $s_\delta(t)$ that is the output of the sampler. A scalar quantizer partitions the set $\mathbb{R}$ into M subsets $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_M$ which are the quantization regions $\mathcal{R}_j = (b_{j-1}, b_j]$ and b_{j-1}, b_j are the boundary decision points for all quantization levels $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_M$. In order to say that this quantizer is regular one more condition must be satisfied which is $a_j \in (b_{j-1}, b_j)$.

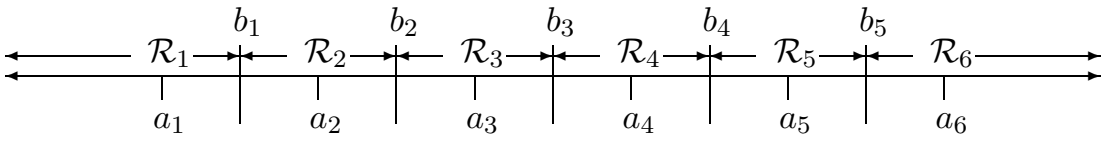


Figure 2.3: Quantization regions and representation points [2]

[2]This image is taken from [19]

One must define the conditions that must exist in order to create an optimal quantizer.

(i) Choice of $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_M$ for given $a_1, a_2, \ldots, a_M$:

Given any $u \in \mathbb{R}$ as a sample value of a random variable U, we want to minimize the squared error $(u - a_j)^2$. This is minimized (over the fixed set of representation points $\{a_1, a_2, \ldots, a_M\}$) by representing u by the closest representation point a_j. This means, for example, that if u is between a_j and a_{j+1}, then u is mapped into the closer of the two. Thus the boundary b_j between $\mathcal{R}_j$ and $\mathcal{R}_{j+1}$ must lie halfway between the representation points a_j and a_{j+1} where $j = 1, \ldots, M-1$. That is $b_j = \dfrac{a_j + a_{j+1}}{2}$. This specifies each quantization region, and also shows why each region should be an interval. In addition, for this minimization we didn't need a probabilistic model of the quantizer's input signal.

(ii) Choice of $a_1, a_2, \ldots, a_M$ for given $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_M$:

To answer this problem the probabilistic model for $U_1, U_2, \ldots$ (which are the discrete input random variables representing only one sample value of $s_\delta(t)$ in each interval) is important. Suppose that the random variables $\{U_k\}$ are iid analog random variables with probability density function $f_U(u)$. For a given set of points $\{a_j\}$, $Q(U)$ maps each sample value $U \in \mathbb{R}_j$ into a_j. The mean squared error (MSE) is then:

$$\text{MSE} = \mathcal{E}\{(U - Q(U))^2\} = \int_{-\infty}^{\infty} f_U(u)(u - Q(u))^2 \, du = \sum_{j=1}^{M} \int_{\mathcal{R}_j} f_U(u)(u - a_j)^2 \, du \tag{2.3.7}$$

and this quantity must be minimized.

Now let $A_j = P(U \in \mathcal{R}_j) = \int_{\mathcal{R}_j} f_U(u) \, du$ and $f_j(u) = f_j(u | u \in \mathcal{R}_j) \iff f_j(u) = \dfrac{f_j(u, u \in \mathcal{R}_j)}{A_j}$

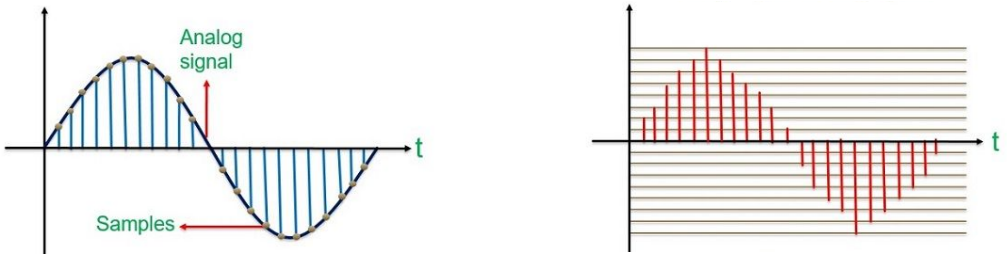
$$\iff f_j(u) = \left\{ \begin{array}{ll} \dfrac{f_U(u)}{A_j}, & u \in \mathcal{R}_j \\ 0, & u \notin \mathcal{R}_j \end{array} \right\} \tag{2.3.8}$$

From (2.3.7) and (2.3.12) it is true that:

$\text{MSE} = A_j \int_{\mathcal{R}_j} f_j(u)(u-a_j)^2\,du = A_j \int_{\mathcal{R}_j} f_j(u|u \in \mathcal{R}_|)(u-a_j)^2\,du \iff$

$$\text{MSE} = \mathcal{E}\{(U-a_j)^2 | U \in \mathcal{R}_j\} \tag{2.3.9}$$

$\mathcal{E}\{(U-a_j)^2\} = \mathcal{E}\{U^2\} - 2a_j\mathcal{E}\{U\} + {a_j}^2 = \mathcal{E}\{U^2\} - 2y\mathcal{E}\{U\} + y^2$ where $y = a_j$. The minimum value of the trinomial is $y = -(\frac{-2\mathcal{E}\{U\}}{2}) \iff y = \mathcal{E}\{U\} \iff a_j = \mathcal{E}\{U|U \in \mathcal{R}_j\}$.
From **(i)** and **(ii)** we have proved two conditions, which are $a_j = \mathcal{E}\{U|U \in \mathcal{R}_j\}$ and $b_j = \frac{a_j + a_{j+1}}{2}$. These conditions are necessary to minimize the MSE for a given number M of representation points, but they are not sufficient because the probability density function f_U may be way more "dense" in a specific interval $\mathcal{V}$, therefore it is more possible that $U(\omega) \in \mathcal{V}$, where $U : \Omega \to \mathbb{R}$ and $\omega \in \Omega$. If the quantizer is uniform then this problem can not be treated, but, audio signals tend to have a lot of repetitions and possibly there will be no such interval $\mathcal{V}$.



(a) Sampling of analog signal

(b) Quantization of samples

Figure 2.4: Quantization Process [3]

In addition the SNR of quantization is

$$\text{SQNR}_{\text{dB}} = 20log\left[\frac{V_{\text{rms}}}{q_e}\right] \tag{2.3.10}$$

where V_{rms} is the signal's $s_\delta(t)$ rms peak voltage and q_e is the noise rms voltage also called the quantized noise. The uniform quantization step size is defined as $S = \frac{s_{\delta_{max}} - s_{\delta_{min}}}{M}$ and it is fixed.

[3]Images from [15]

Now we can calculate the quantization noise as $q_e = \frac{1}{S}\int_{-S/2}^{S/2} q^2\, dq$ where q is the quantization error and this follows that $q_e = \frac{S}{\sqrt{3}}$. In addition the full scale of the signal is $V_{max} - V_{min} = V_{fs}$ where $V_{max} = s_{\delta_{max}}$ is the highest voltage of the input analog signal and $V_{min} = s_{\delta_{min}}$ the lowest one. The peak voltage is $\frac{V_{fs}}{2}$ and the rms of peak voltage is $V_{rms} = \frac{V_{fs}}{2\sqrt{2}}$. Now we write $S = \frac{s_{\delta_{max}} - s_{\delta_{min}}}{M} = \frac{V_{max} - V_{min}}{M} = \frac{V_{fs}}{M} \iff V_{fs} = S \cdot M$ and the signal rms peak voltage will be $V_{rms} = \frac{S \cdot M}{2 \cdot \sqrt{2}} = \frac{S \cdot 2^n}{2 \cdot \sqrt{2}}$, where n is the bit-depth of quantization. Knowing that $V_{rms} = \frac{S \cdot 2^n}{2 \cdot \sqrt{2}}$ and $q_e = \frac{S}{2 \cdot \sqrt{3}}$ from (2.3.10) we have $\text{SQNR}_{\text{dB}} = 20log\left(2^n \cdot \sqrt{\frac{3}{2}}\right) = 20log(2^n) + 20log\left(\sqrt{\frac{3}{2}}\right) \iff$

$$\text{SQNR}_{\text{dB}} = 6.02n + 1.76(\text{dB}) \tag{2.3.11}$$

The output of the quantizer is the set $\mathcal{Z} = \{1, 2, \ldots, M\}$ that contains all the indexes of the quantization levels. If we use more quantization levels it will be true that M is increasing, therefore more bits are being used in order to represent the approximate-quantized amplitudes of the input signal $s_\delta(t)$ (the quantizer's input). This is equivalent to saying that the bit-depth of the quantizer is increasing, therefore from (2.3.11) $SQNR_{dB}$ is increasing too and we can achieve a more accurate representation of the analog signal $s_\delta(t)$. However, this can have a great cost in terms of storage and rate of transmission.

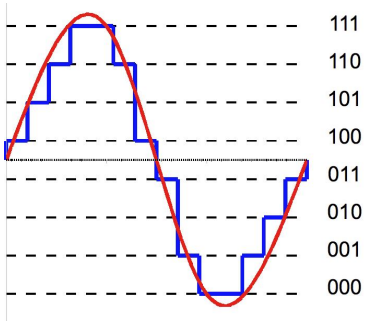


Figure 2.5: Quantizer example for M=8

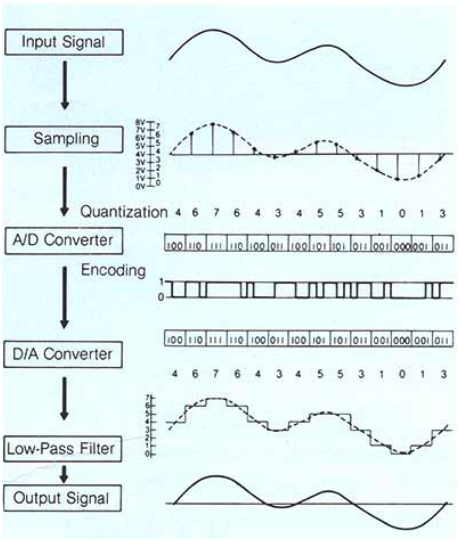
(4) Binary Encoder

In this stage we will encode the set $\mathcal{Z} = \{1, 2, \ldots, M\}$ that is the input of the binary encoder. To do so, we first notice that the length of the set is $|\mathcal{Z}| = M$, therefore we will use $\lceil log_2 M \rceil$ bits to represent each index j of the corresponding quantization level $\mathcal{R}_j = (b_{j-1}, b_j]$, where $a_j \in (b_{j-1}, b_j)$. The process is easy, because the code has a fixed length for every coded index.
For example if $M = 6 \implies \lceil log_2 M \rceil = 3$ meaning that we will need 3 bits for each codeword. Thus, $1 \to 000$, $2 \to 001$, $3 \to 010$, $4 \to 011$, $5 \to 100$, $6 \to 101$ and if we prefer gray coding then $1 \to 000$, $2 \to 001$, $3 \to 011$, $4 \to 010$, $5 \to 110$, $6 \to 111$. Another example is depicted in Figure 2.5 where the bitstream that will be stored in the computer is "100101110111110100011001000001010011".

The encoder mapping is $E : \mathcal{Z} \to \mathcal{I}$ where $\mathcal{I} = \{1, 2, \ldots, M\}_2$ which is a set of integers that take binary values. Hence, we can declare a decoder mapping $D : \mathcal{I} \to \mathcal{C}$, where $\mathcal{C} = \{a_1, a_2, \ldots, a_M\}$ is the codebook (equivalently the look-up table). Thus, if $Q(x) = a_j \implies E(x) = j \implies D(j) = a_j$ and x is the amplitude value of a sample. According to this statement, $Q(x) = D(E(x))$. The decoder will decode the bitstream that contains all the binary indexes by splitting it into many $\lceil log_2 M \rceil$ parts. Then it will convert each part from binary representation to integer representation,which is the index j, according to the encoding method (gray coding or not). In the end the decoder will search the j-th value of the codebook $\mathcal{C}$, $a_j \in \mathcal{C}$, and this is will be the output of the decoder. Sometimes the decoder is reffered to as an "inverse quantizer".
In Figure 2.6, PCM encoding and decoding processes for Sony PCM-F1 digital audio recorder ([21]) are depicted.

The A/D converter contains the quantizer and the binary encoder circuits.
The D/A converter contains the binary decoder circuit and it converts a fixed-point binary number into voltage.
The Low-Pass Filter is needed (the spectrum of the output of the sampler equal to (2.3.2)) for retrieving the original spectrum (that is the middle one for $k = 0$) and ignore all the other delayed

[4]Images from [21]



(a) PCM-F1 scheme.

(b) Sony PCM-F1 digital audio recorder.

Figure 2.6: [4]

spectrums. An ideal LPF will be like:

$$H(F) = \left\{ \begin{array}{ll} \frac{1}{F_s}, & |F| < F_s \\ 0, & otherwise \end{array} \right\} \tag{2.3.12}$$

In the PCM-F1 case, the LPF circuit is not ideal, therefore it will contain some extra frequencies in the stopband region (see figure 2.2) and aliasing may happen. This is the reason for choosing a sampling frequency of $F_s = 44.1$kHz that is greater than the Nyquist frequency (40kHz).
In the end the ouput of the Low-Pass Filter circuit will be fed into the amplifier circuit of the speakers.

Comments

- Our uncompressed audio signals that are the input signals of our compression and coding schemes are encoded using linear PCM, with bit-depth=16 bits and are stored in the computer

in a signed, little-endian binary representation (.wav files). In addition they've been sampled with a sampling rate of $44100\frac{samples}{second}$.

- The .wav files can be decoded from the software of the computer (for example foobar, winamp, windows media player, etc).
- In Matlab, using the function audioread() the .wav file gets decoded as an one-dimensional matrix (if the .wav contains a mono audio signal) or as a two-dimensional matrix (if the .wav contains a stereo audio signal). Thus we can define a stereo audio signal as $X = \begin{bmatrix} X^L & X^R \end{bmatrix}$.
- Sample Rate $\times$ bit-depth $\times$ channels = bit-rate, therefore the bit rate of optimal quality digital .wav stereo signals is $44100\frac{samples}{second}$ x $16\frac{bits}{sample}$ x $2 = 1411.2kbps$.

2.4 The Z-Transform

This section is based on [22] and [23]. The Z-transform for discrete-time signals is the counterpart of Laplace-transform for continuous-time signals and they each have a similar relationship to the corresponding Fourier-transform. We need the Z-transform because Fourier-transform does not converge for all sequences and it is useful to have a generalization of the Fourier-transform that encompasses a broader class of signals. Also, in analytical problems, the Z-Transform notation is often more convenient than the Fourier-Transform notation.

Definition 2.4.1 (**Discrete-time Fourier-transform (DTFT)**)

The discrete-time Fourier transform (DTFT) is a form of Fourier analysis that is applicable to a sequence x[n] and is defined as:

$$X(f) = \sum_{n=-\infty}^{\infty} x\,[n] \cdot e^{j2\pi fn} \tag{2.4.1}$$

or as:

$$X(e^{j\omega n}) = \sum_{n=-\infty}^{\infty} x\,[n] \cdot e^{-j\omega n}, \qquad \omega = 2\pi f \tag{2.4.2}$$

In this section, we will mainly use equation (2.4.2).

Definition 2.4.2 (**Bilateral Z-transform**)

The bilateral (or two-sided) Z-transform of a sequence $x[n]$ is defined as:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{2.4.3}$$

Equation (2.4.3) is an infinite sum of power series with z considered to be a complex variable.

Definition 2.4.3 (**Z-transform operator**)

A Z-Transform operator transforms a sequence into a function. That is the Z-transform operator $\mathcal{Z}\{\cdot\}$ and we can say that

$$\mathcal{Z}\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n]z^{-n} = X(z) \tag{2.4.4}$$

Its unique correspondence between a sequence and its Z-transform can be written as $x[n] \overset{\mathcal{Z}}{\longleftrightarrow} X(z)$.

Definition 2.4.4 (**Unilateral Z-transform**)

The unilateral (or one-sided) Z-transform of a sequence $x[n]$ is defined as:

$$X(z) = \sum_{n=0}^{\infty} x[n]z^{-n} \tag{2.4.5}$$

Relation of Fourier-transform with Z-transform

At first, if we state that $z = e^{j2\pi f}$ then from (2.4.3) it is true that: $X(z) = X(e^{j2\pi f}) \iff \iff X(e^{j2\pi f}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j2\pi fn} = X(f)$ that is the Fourier-transform of $x[n]$. When the Fourier-transform exists it is equivalent to $X(z)$, where $z = e^{j2\pi f} \iff z = e^{j2\omega}$, with $\omega = 2\pi f$. This corresponds to restricting z to have a unity magnitude, i.e , for $|z| = 1$ the Z-transform corresponds to the Fourier-transform.

Also, by expressing the complex variable z in polar form as $z = re^{j\omega}$, thus $X(re^{j\omega})$=$X(z)$ $\iff X(re^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n](re^{j\omega})^{-n} \iff X(re^{j\omega}) = \sum_{n=-\infty}^{\infty}(x[n]r^{-n}) \cdot (e^{-j\omega n})$ and we can interpret the Fourier-transform of the product of the original sequence $x[n]$ and the exponential sequence r^{-n}. If $r = 1$ this expression reduces to the Fourier-transform of $x[n]$.

Definition 2.4.5 (**Region Of Convergence (ROC)**)

For any given sequence $x[n]$, the set of values of z (or the set of points in the complex plane) for which the Z-transform power series converges is called the region of convergence (ROC), of the Z-Transform.

$$ROC = \left\{ z : \left| \sum_{n=-\infty}^{\infty} x[n]z^{-n} \right| < \infty \right\}. \tag{2.4.6}$$

The Z-transform does not converge for all sequences $x[n]$ or for all values of z. Similarly, the power series representing the Fourier transform does not converge for all sequences; i.e., the infinite sum may not always be finite. But if $x[n]$ is absolutely summable, the Fourier-transform converges to a continuous function of ω,i.e if

$$\sum_{n=-\infty}^{\infty} \left|x[n]\right| < \infty \implies \left|X(e^{j\omega})\right| \leq \sum_{n=-\infty}^{\infty} \left|x[n]\right| \cdot \left|e^{-j\omega n}\right| = \sum_{n=-\infty}^{\infty} \left|x[n]\right| < \infty \tag{2.4.7}$$

Convergence of the power series of (2.4.3) for a given sequence $x[n]$ depends only on $|z|$, since $\left|X(z)\right| < \infty$ if:

$$\sum_{n=-\infty}^{\infty} \left|x[n]\right| \left|z^{-n}\right| < \infty \tag{2.4.8}$$

i.e, the ROC of the power series in (2.4.3) consists of all values of z, such that the inequality in (2.4.8) holds. In addition, if the ROC includes the unit circle, then this implies convergence of the Z-transform of $x[n]$ for $|z| = 1$, or equivalently, the Fourier-transform of the sequence $x[n]$ converges. Inversely, if the ROC does not include the unit circle, the Fourier-transform does not converge absolutely. Some sequences are square summable (e.g $\frac{\sin \omega_c n}{\pi n}$, $n \in \mathbb{Z}$), meaning that the Fourier-transform converges in the mean-square sense to a discontinuous periodic function. Other sequences (e.g $\cos \omega_c n$, $n \in \mathbb{Z}$) are neither square summable nor absolutely summable, but a useful Fourier-transform can be defined using impulse functions. Therefore, it is not strictly correct to think of the Fourier-transform as being the Z-transform evaluated on the unit circle.

Definition 2.4.6 (**Rational Function**)

A function $f(x)$ is called a rational function if it can be written in the form: $f(x) = \frac{P(x)}{Q(x)}$, where P, Q are polynomial functions of x and $Q(x) \neq 0$. In addition, $f(x)$ can be called a proper rational function if $deg(P(x)) < deg(Q(x))$.

Definition 2.4.7 (**Rational Z-transform**)

Rational Z-transform is a very important and useful Z-transform notation for a rational function. We can say that if $x[n] \overset{\mathcal{Z}}{\longleftrightarrow} X(z)$ and $X(z) = \frac{P(z)}{Q(z)}$ then from (2.4.6) $X(z)$ is a rational function inside the ROC where $P(z), Q(z)$ are polynomial functions of z.

Poles: the points in z-plane where $X(z) = \pm\infty \implies Q(z) = 0$.

Zeros: the points in z-plane where $X(z) = 0 \implies P(z) = 0$.

There exist a number of important relationships between the location of poles of $X(z)$ and the ROC of the Z-transform.

Z-transforms and LTI systems

Since we shall rely on the Z-transform notation extensively in this thesis, it is worthwhile to illustrate how the Z-transform can be used in the representation and analysis of LTI systems. An

LTI system can be represented as the convolution $y[n] = x[n] * h[n]$ of the input $x[n]$ with $h[n]$, where $h[n]$ is the response of the system to the unit impulse sequence $\delta[n]$. From the convolution property of Z-transform it follows that the Z-transform of $y[n]$ is:

$$Y(z) = X(z) \cdot H(z) \tag{2.4.9}$$

where $x[n] \overset{\mathcal{Z}}{\longleftrightarrow} X(z)$, $h[n] \overset{\mathcal{Z}}{\longleftrightarrow} H(z)$. In this context, the Z-transform $H(z)$ is called the system function of the LTI system whose impulse response is $h[n]$. In addition, if $H(z)$ is a rational function it can be written as $H(z) = \dfrac{Y(z)}{X(z)}$.

Definition 2.4.8 (Stability)

In order to have a stable LTI system, the impulse response $h[n]$ must be absolutely summable, therefore $h[n]$ has a Fourier-transform, i.e the ROC must include the unit circle .

Definition 2.4.9 (Causality)

In order to have a causal LTI system, the system's output should only depend on present and past inputs at any given instance $n = n_0$. Causality implies that the system's impulse response $h[n]$ is a right-sided sequence, therefore the ROC of $H(z)$ must be outside the outermost pole.

2.5 FIR, IIR Lattice Filters

In this section, (see [24], [23]), we will start by presenting some basic block diagrams. Afterwards we will present signal flow graph descriptions of computational structures for LTI systems. Such flow graphs are also called "networks" in analogy to electrical circuit diagrams. Using a combination of algebraic manipulations and manipulations of block diagram representations, we'll derive some basic equivalent structures for implementing an LTI discrete-time system, the lattice structures. The basic elements required for the implementation of an LTI discrete-time system are adders, multipliers, and memory for storing delayed sequence values and coefficients. The intercon-

nection of these basic elements is conveniently depicted by block diagrams composed of the basic pictorial symbols shown in Figure 2.7 .

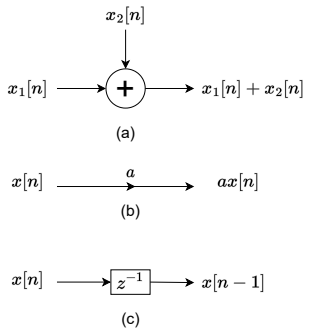


Figure 2.7: Block diagram symbols: (a) Addition of two sequences. (b) Multiplication of a sequence by a constant. (c) Unit delay.

In Figure 2.7(a) we represent the addition of two sequences. In general block diagram notation, an adder may have any number of inputs. However in almost all practical implementations, adders have only two inputs. Figure 2.7(b) depicts multiplication of a sequence by a constant and Figure 2.7(c) depicts delaying a sequence by one sample. In digital implementations, the delay operation can be implemented by providing a storage register for each unit delay that is required [24]. The unit delay system is represented by its system function, z^{-1}. Delays of more than one sample can be denoted as in Figure 2.7(c), with a system function of z^{-M}, where M is the number of samples of delay. In an integrated-circuit implementation, these unit delays might form a shift register that is clocked at the sampling rate of the input signal. In a software implementation, M cascaded unit delays would be implemented as M consecutive memory registers.

Definition 2.5.1 (Linear-constant coefficient difference equation system)

An important class of LTI systems consists of those systems for which the input sequence $x[n]$ and the output sequence $y[n]$ satisfy an N-th order linear constant-coefficient difference equation of the form:

$$\sum_{k=0}^{N} a_k \cdot y[n-k] = \sum_{m=0}^{M} b_m \cdot x[n-m] \tag{2.5.1}$$

A linear constant-coefficient difference equation for a discrete-time system, does not provide a unique specification of the output for a given input, therefore auxilary information or conditions are required. If an additional condition is that the system is initially at rest, then the system will be LTI and causal.

In addition, equation (2.5.1), using Definition 2.4.3 becomes: $\sum_{k=0}^{N} a_k z^{-k} Y(z) = \sum_{m=0}^{M} b_m z^{-m} X(z) \iff$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{m=0}^{M} b_m z^{-m}}{\sum_{k=0}^{N} a_k z^{-k}} \tag{2.5.2}$$

where $H(z)$ is the system function (or transfer function of the system) . Note that the impulse response of the system is $h[n]$ and $h[n] \overset{\mathcal{Z}}{\longleftrightarrow} H(z)$.

Setting $a_0 = 1$ we can write (2.5.1) as

$$y[n] = \sum_{m=0}^{M} b_m x[n-m] + \sum_{k=1}^{N} a_k y[n-k] \tag{2.5.3}$$

where the system function can be written as:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{m=0}^{M} b_m z^{-m}}{1 - \sum_{k=1}^{N} a_k z^{-k}} \tag{2.5.4}$$

Definition 2.5.2 (FIR system)

A system is called FIR if its impulse response is finite in duration (FIR $\rightarrow$ Finite Impulse Response). This reminds us of Definition 2.4.8 and we must state that FIR systems never become unstable ($\forall$ input $x[n]$), i.e FIR systems are always stable, meaning that the FIR system function $H(z)$ has an all-zero structure ($N = 0$ in equation (2.5.1)) meaning that we can characterize an FIR system by the following equation:

$$y[n] = \sum_{m=0}^{M} b_m x[n-m], \qquad a_0 = 1 \tag{2.5.5}$$

and from (2.5.2) and (2.5.5) it follows that

$$H(z) = \sum_{m=0}^{M} b_m z^{-m} \overset{\mathcal{Z}^{-1}}{\longleftrightarrow} h[n] = \sum_{m=0}^{M} b_m \delta(n-m) \tag{2.5.6}$$

From (2.5.6) $\Longrightarrow$

$$h[m] = \begin{cases} b_m, & m \leq M \\ 0, & otherwise \end{cases} \tag{2.5.7}$$

where we can clearly see that the impulse response has a finite duration. Furthermore, the output $y[n]$ of any FIR system can be computed non-recursively where the coefficients are the values of the impulse response sequence. We typically use FIR systems in applications where linear phase is important and they require a good amount of memory and computational performance.

Definition 2.5.3 (IIR system)

A system is called IIR if its impulse response has infinite duration (IIR $\rightarrow$ Infinite Impulse Response). IIR systems have data precedence relations and they are recursive in nature. If we say that the impulse response of the system is $h[n]$ and $h[n] \overset{\mathcal{Z}}{\longleftrightarrow} H(z)$ then $H(z)$ has at least one non-zero pole that is not cancelled by a zero. That means that $h[n]$ will have at least one term. The stability of an IIR filter can be studied using the zero-pole plot that concerns the transfer funtion $H(z)$ of the filter. A digital filter is stable if all the poles of the transfer function $H(z)$ are lying inside the unit circle in the z-plane. If at least a pole lies outside the unit circle, this introduces a component in the filter's impulse response that is exponentially increasing. This causes the filter to be unstable. IIR filters have low implementation cost, low latency and they are less numerically stable than the FIR filters due to feedback paths. In addition they have non-linear phase characteristics. At last, there is an analog equivalent for every IIR system, meaning that IIR systems may be used for mimicking an analog filter's characteristics .

2.5.1 Signal flow graphs for linear constant-coefficient difference equations

A signal flow graph,[24], is used for representing a difference equation and is fundamentally the same as a block diagram representation except for a few differences in notation.

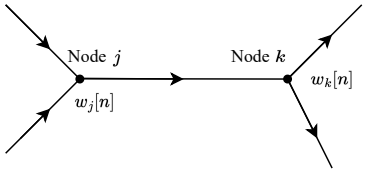


Figure 2.8: Nodes and branches in a signal flow graph.

Formally, a signal flow graph is a network of directed branches that connect at nodes. Node variables for digital filters mainly represent sequences and we frequently use the notation $w_k[n]$ to indicate this. The branch (j, k) denotes a branch with node j as its origin and node k as its termination and the direction from j to k is represented by an arrow on the branch. In Figure 2.8 $w_j[n]$ is the input signal and $w_k[n]$ is an intermediate node. In a signal flow graph that is linear, the output of a branch is a linear transformation of the input. Also, each node's value in a graph is the summation of all the outputs of the branches that enter the node. Multiplying with a coefficient can be represented in the same way as in block diagram representations. Finally, we define a source node as a node with no entering branches and a sink node as a node with entering branches only.

Figure 2.9 depicts an example that helps us clarify signal flow graphs. Having explained the basics of signal flow graphs we will present some basic structures for IIR and FIR systems in order to understand how to implement them using signal flow graphs.

2.5.2 Lattice structures

Lattice structures are based on a cascade (output to input) connection of the basic structure shown in Figure 2.10(a). In this case the basic building block system has two inputs and two outputs

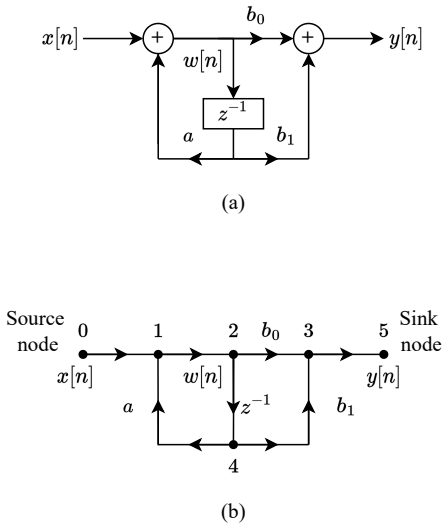


Figure 2.9: (a) Block diagram representation of a 1^{st} order digital filter. (b) Structure of the signal flow graph corresponding to the block diagram in (a).

and is called a two-port flow graph. Figure 2.10(b) shows the equivalent flow graph representation. If M two-port flow graphs are cascaded with a "termination" at each end of the cascade, the overall system will be a single-input-single-output (SISO) system. In this case, $x[n]$ is the input that feeds both inputs ($a^{(0)}[n]$ and $b^{(0)}[n]$) of the two-port flow graph (1) and $y[n] = a^{(M)}[n]$ the upper output of the last two-port flowgraph (M). The lower output of the two-port flow graph (M) is initially ignored. This is depicted in Figure 2.11. In this thesis, we will limit our attention to a widely used class of FIR and IIR filter structures whose shape motivates the name lattice filters.

2.5.3 FIR Lattice Filters

In this subsection we will focus on the use of lattice filters to implement FIR transfer functions.

The coefficients $k_1, k_2, \ldots, k_M$, are the k-coefficients of the lattice structure and the node variables $a^{(i)}[n]$, $b^{(i)}[n]$ are intermediate sequences that depend upon the input $x[n]$ through the set of difference equations:

$$x[n] = a^{(0)}[n] = b^{(0)}[n] \tag{2.5.8}$$

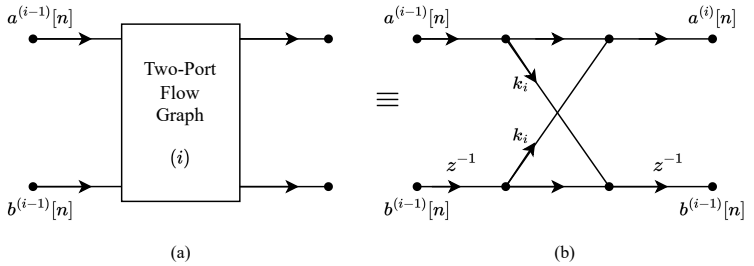


Figure 2.10: (a) Block diagram representation of a two-port building block. (b) Equivalent flow graph representation.

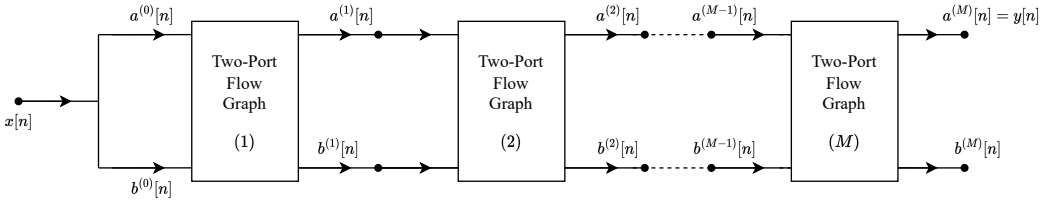


Figure 2.11: Cascade connection of M two-port building blocks.

$$a^{(i)}[n] = a^{(i-1)}[n] + k_i b^{(i-1)}[n-1], \qquad i = 1, 2, \ldots, M \tag{2.5.9}$$

$$b^{(i)}[n] = b^{(i-1)}[n-1] + k_i a^{(i-1)}[n], \qquad i = 1, 2, \ldots, M \tag{2.5.10}$$

$$y[n] = a^{(M)}[n] \tag{2.5.11}$$

These equations must be computed in the order shown ($i = 0, 1, \ldots, M$) since the output of the two-port building block $(i-1)$ is needed as input to the two-port building block (i). In addition, the lattice structure in Figure 2.12 is a linear signal flow graph with only delays and constant branch coefficients, therefore it is an LTI system and the impulse response from the input of the system to any internal node has finite length (note that there are no feedback loops in the FIR lattice structure). We can clarify this by saying that all paths to any node variable $a^{(i)}[n]$ or $b^{(i)}[n]$ pass through at least one delay (z^{-1}) with the greatest delay being along the bottom path and then up to node variable $a^{(i)}[n]$ through the coefficient k_i (information passes through a register and a multiplier). This will be the last impulse that arrives at $a^{(i)}[n]$ with a length of $i + 1$ samples.
It is convenient to say that $x[n] = \delta[n]$ so that $a^{(i)}[n]$ and $b^{(i)}[n]$ are the resulting impulse responses at

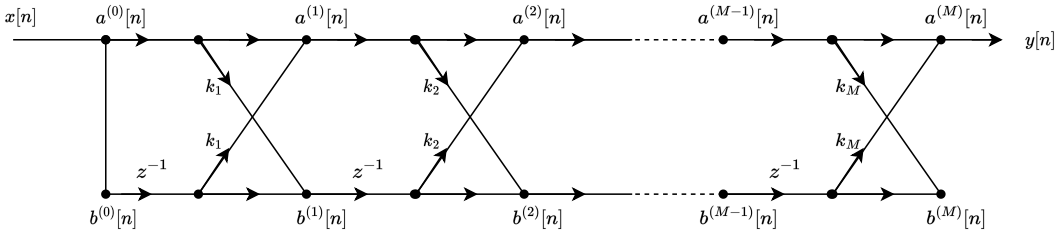


Figure 2.12: Lattice flow graph for an FIR system based on cascade of M two-port building block sections.

the associated nodes, and define their Z-transforms $A^{(i)}(z)$ and $B^{(i)}(z)$ respectively. Consequently, the transfer function between the input and the upper node (i) is:

$$A^{(i)}(z) = \sum_{n=0}^{i} a^{(i)}[n]z^{-n} = 1 + \sum_{n=1}^{i} a^{(i)}[n]z^{-n} \tag{2.5.12}$$

From (2.5.8) and because we stated that $x[n] = \delta[n]$:

$$A^{(0)}(z) = B^{(0)}(z) = 1 \tag{2.5.13}$$

From (2.5.9):

$$A^{(i)}(z) = A^{(i-1)}(z) + k_i z^{-1} B^{(i-1)}(z), \qquad i = 1, 2, \ldots, M \tag{2.5.14}$$

And from (2.5.10):

$$B^{(i)}(z) = k_i A^{(i-1)}(z) + z^{-1} B^{(i-1)}(z), \qquad i = 1, 2, \ldots, M \tag{2.5.15}$$

Using (2.5.13),(2.5.14) and (2.5.15) we can calculate $A^{(i)}(z)$ and $B^{(i)}(z)$ recursively up to any value of i . If we continue, the pattern that emerges in the relationship between $A^{(i)}(z)$ and $B^{(i)}(z)$ is:

$$B^{(i)}(z) = z^{-i} A^{(i)}(1/z) \tag{2.5.16}$$

$$A^{(i)}(z) = z^{-i} B^{(i)}(1/z) \tag{2.5.17}$$

Equation (2.5.17) can be proved by defining $z = \frac{1}{z}$ in equation (2.5.16), but now we need to prove (2.5.16). This will be done below by using the mathematical proof technique of induction:

Base case: for $i = 1$, using (2.5.14) and (2.5.15) it is true that:

$A^{(1)}(z) = A^{(0)}(z) + k_1 z^{-1} B^{(0)}(z) \overset{(2.5.13)}{\iff} A^{(1)}(z) = 1 + k_1 z^{-1}$

$B^{(1)}(z) = k_1 A^{(0)}(z) + z^{-1} B^{(0)}(z) \overset{(2.5.13)}{\iff} B^{(1)}(z) = k_1 + z^{-1} = z^{-1}(1 + k_1 z) = z^{-1} A^{(1)}(1/z)$

Now note that for $i = 2$, using (2.5.14) and (2.5.15) it is true that:

$A^{(2)}(z) = \underbrace{A^{(1)}(z)}_{1+k_1 z^{-1}} + k_2 z^{-1} \underbrace{B^{(1)}(z)}_{k_1 + z^{-1}} = 1 + k_1(1 + k_2) z^{-1} + k_2 z^{-2}$

$B^{(2)}(z) = k_2 \underbrace{A^{(1)}(z)}_{1+k_1 z^{-1}} + z^{-1} \underbrace{B^{(1)}(z)}_{k_1 + z^{-1}} = z^{-2}(1 + k_1(1 + k_2) z + k_2 z^2) = z^{-2} A^{(2)}(1/z)$

Induction step: We state that for $i - 1$ it is true that $B^{(i-1)}(z) = z^{-(i-1)} A^{(i-1)}(1/z) \overset{z=1/z}{\iff}$ $B^{(i-1)}(1/z) = z^{(i-1)} A^{(i-1)}(z) \iff A^{(i-1)}(z) = z^{-(i-1)} B^{(i-1)}(1/z)$. We must prove that the above statement is true for $(i - 1) + 1 = i$.

By using (2.5.15):

$B^{(i)}(z) = k_i z^{-(i-1)}(B^{(i-1)}(1/z)) + z^{-1} z^{-(i-1)} A^{(i-1)}(1/z) = z^{-i} \cdot \underbrace{[A^{(i-1)}(1/z) + k_i z B^{(i-1)}(1/z)]}_{(2.5.14) \overset{z=1/z}{=} A^{(i)}(1/z)} \iff$

$B^{(i)}(z) = z^{-i} A^{(i)}(1/z)$, therefore we have just proved (2.5.16) $\forall i \geq 0$.

The transfer functions $A^{(i)}(z)$ and $B^{(i)}(z)$ are i^{th} order polynomials and it is useful to obtain a direct relationship among the coefficients of the polynomials. From (2.5.12) by setting $i = i - 1$ it is true that:

$$A^{(i-1)}(z) = 1 + \sum_{n=1}^{i-1} a^{(i-1)}[n] z^{-n} \tag{2.5.18}$$

From (2.5.16) and (2.5.18) $\implies$

$$B^{(i-1)}(z) = z^{-(i-1)} A^{(i-1)}(1/z) = z^{-(i-1)} \left[1 + \sum_{n=1}^{i-1} a^{(i-1)}[n] z^{n} \right] \tag{2.5.19}$$

Substituting equations (2.5.18) and (2.5.19) into equation (2.5.14), $A^{(i)}(z)$ can also be expressed

as:

$$A^{(i)}(z) = \left(1 + \sum_{n=1}^{i-1} a^{(i-1)}[n] z^{-n}\right) + k_i z^{-1} \left(z^{-(i-1)} \left[1 + \sum_{n=1}^{i-1} a^{(i-1)}[n] z^{n}\right]\right) \tag{2.5.20}$$

And reindexing the second summation with replacing n by $i - n$ gives us:

$$A^{(i)}(z) = 1 + \sum_{n=1}^{i-1} \left[a^{(i-1)}[n] + k_i a^{i-1}[i-n]\right] z^{-n} + k_i z^{-i} \tag{2.5.21}$$

Comparing (2.5.21) with (2.5.18) and then with (2.5.12) shows that:

$$a^{(i)}[n] = \left[a^{(i-1)}[n] + k_i a^{(i-1)}[i-n]\right], \qquad n = 1, \ldots, i-1 \tag{2.5.22}$$

$$a^{(i)}[i] = k_i \tag{2.5.23}$$

We can use (2.5.22) and (2.5.23) recursively in order to compute the transfer functions of succesively higher order FIR filters until we come to the end of the cascade giving us:

$$A(z) = 1 + \sum_{n=1}^{M} a[n] z^{-n} = \frac{Y(z)}{X(z)} \tag{2.5.24}$$

Now we have a very important algorithm that we'll use in this thesis:

Algorithm 2.1 k-Parameters-to-Coefficients Algorithm

Given $k_1, k_2, \ldots, k_M$

for $i = 1, 2, \ldots, M$ **do**
  $a^{(i)}[i] = k_i$
  if $i > 1$ **then** $\forall j = 1, 2, \ldots, i-1$:
    $a^{(i)}[j] = a^{(i-1)} + k_i a^{(i-1)}[i-j]$
  end if
end for

$a[j] = a^{(M)}[j], \qquad j = 1, 2, \ldots, M$

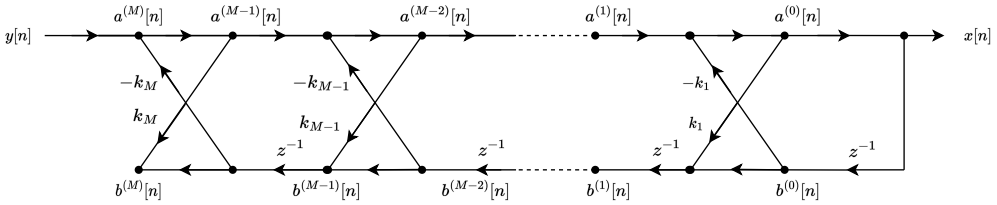


Figure 2.13: Lattice flow graph for an IIR all-pole system based on cascade of M two-port building block sections.

2.5.4 IIR Lattice Filters

In the previous subsection, in equation (2.5.24), we've found the transfer function of the FIR Lattice Filter, $A(z)$, that was an all-zero transfer function. Now we define the all-pole inverse transfer function as $H(z) = \dfrac{1}{A(z)}$. In order to create an all-pole lattice structure, assume that we are given $y[n] = a^{(M)}[n]$ and we wish to compute the input $a^{(0)}[n] = x[n]$. We can do this by inverting from right to left the computations in Figure 2.12. This will be done by solving equation (2.5.14) as $A^{(i-1)}(z) = g(A^{(i)}(z), B^{(i-1)}(z))$ and leave equation (2.5.15) as it is. Specifically:

$$A^{(i-1)}(z) = A^{(i)}(z) - k_i z^{-1} B^{(i-1)}(z), \qquad i = M, M-1, \ldots, 1 \tag{2.5.25}$$

$$B^{(i)}(z) = k_i A^{(i-1)}(z) + z^{-1} B^{(i-1)}(z), \qquad i = M, M-1, \ldots, 1 \tag{2.5.26}$$

In this case the signal flow graph is from $i = M$ to $i = 0$ along the top of the graph and from $i = 0$ to $i = M$ along the bottom. The input of the all-pole lattice filter will be $a^{(M)}[n]$ and the output

will be $a^{(0)}[n]$. At last, the condition $x[n] = a^{(0)}[n] = b^{(0)}[n]$ at the terminals of the last stage causes a feedback connection, that is necessary for an IIR system, which provides the sequences $b^{(i)}[n]$ to propagate in reverse direction. In addition, $a^{(M)}[n] = y[n]$ is the input of the first stage. This concept is depicted in Figure 2.13 and the set of difference equations represented will be:

$$a^{(i-1)}[n] = a^{(i)}[n] - k_i b^{(i-1)}[n-1], \qquad i = M, M-1, \ldots, 1 \tag{2.5.27}$$

$$b^{(i)}[n] = k_i a^{(i-1)}[n] + b^{(i-1)}[n-1], \qquad i = M, M-1, \ldots, 1 \tag{2.5.28}$$

Comments

- The k-coefficients have a special meaning in the context of all-pole and all-zero modeling of signals, and in the next chapter we will refer to them as the PARCOR coefficients (PARtial CORrelation $\rightarrow$ PARCOR).

- By using the FIR lattice filter implementation and specifically Algorithm 2.1, where if we know the k-parameters we can find the a-coefficients, we are going to use these coefficients later to perform linear prediction analysis and synthesis of the audio signal, i.e find $e[n]$ and $y[n]$ of the linear constant-coefficient difference equation system in (2.5.1) by setting $x[n] = e[n]$. Linear prediction will be explained in the next chapter.

Chapter 3

Linear Prediction

Prediction is a procedure concerning statistical estimation where one or more random variables can be estimated from observations of other random variables ([25]). We call it prediction because the variables that we will estimate can be affiliated with the "future" and the observable variables are affiliated with the "past" (or past and present). A very frequent use of prediction is to predict a sample of an SSS random process by observing several prior samples. Using linear prediction we eliminate redundancy so that there is less waste. As a consequence, we need fewer bits to represent each waveform time instant. In this chapter we consider the basic theory needed for compressing random signals using linear prediction, that is a parametric modeling technique.

3.1 Parametric Signal Modeling

In this section, that is based on [12], we define a parametric random signal modeling technique, where we can represent a random signal by using a mathematical model that has a preset structure involving a number of parameters that is limited. We represent a given signal $y[n]$ by choosing the particular selection of parameters that results in the model output $\hat{y}[n]$, being arbitrarily as close as possible to the given signal $y[n]$.

Definition 3.1.1 (Discrete-time random signal)

We consider a random signal to be a member of a group of discrete-time signals that is defined using a set of probability density functions. Precisely, for a particular signal at a particular time instant, the amplitude of the signal's sample at that time instant is supposed to be determined by an underlying probabilistic scheme. The probability density function may be known, or not. So, each and every sample $y_a[nT_s] = y[n]$ (see 2.3) of a particular signal can be seen as an outcome of a random variable Y_n which is predefined. Using a collection of such random variables, one for each sample time $n \in \mathbb{Z}$, we can represent the entire signal and consider it as a discrete-time stochastic process (Definition 2.1.1).

In our case, we will model a random signal that is the output of a discrete-time LTI system, with transfer function $H(z)$ that is assumed to be a rational function (Definition 2.4.6) of the form:

$$H(z) = \frac{\sum_{m=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}}, \qquad a_0 = 1 \tag{3.1.1}$$

In this case the signal is modeled by the values of a_k and b_m ($\forall k \in \{1, 2, \ldots, N\}, \forall m \in \{1, 2, \ldots, M\}$), or equivalently, by the poles and zeros of the system's transfer function $H(z)$, along with the knowledge of the input. When the model is appropriately chosen, it is possible to represent a large number of signal samples by a relatively small set of parameters. In data compression, the set of model parameters is transmitted or stored and the receiver then uses the model with those parameters to reconstruct the signal. Equation (3.1.1) is the transfer function of a linear-constant coefficient difference equation system (Definition 2.5.1). Therefore we can write:

$$y[n] + \sum_{k=1}^{N} a_k y[n-k] = b_0 x[n] + \sum_{m=1}^{M} b_m x[n-m] \tag{3.1.2}$$

Special Cases:

- All-zero model $\iff$ Moving Average (MA) model, where $a_k = 0 \ \forall k \in \{1, 2, \ldots, N\}$
- All-pole model $\iff$ Autoregressive (AR) model, where $b_m = 0 \ \forall m \in \{1, 2, \ldots, M\}$

Moving average (MA) models specify that the output signal $\{y[n]\}$ is linearly dependent on the current and various past values of the input random signal $\{x[n]\}$. In addition, autoregressive (AR) models specify that the output signal $\{y[n]\}$ depends linearly on its own previous values and the input $x[n]$. ARMA models are defined as in (3.1.2) with $N \neq 0$ and $M \neq 0$.
Parametric signal modeling is a powerful approach to signal representation and such models which are comprised of the input signal $x[n]$ and the transfer function $H(z)$ of the linear system, become useful with the addition of constraints that make it possible to solve for the parameters of $H(z)$ given the signal to be represented. In the next section, we'll explain what is a linear predictive model and later we will use this type of modeling for achieving intra-frame linear prediction that is a prediction for every sample inside a frame of samples of an audio signal.

3.2 Linear Predictive Model

A linear predictive model is generally an autoregressive (AR) model that is equivalent to an all-pole model where we assume that the random signal $\{y[n]\}$ is given as a linear combination of its past values and some input $\{x[n]\}$, (see [12], [26], [27], [28]). This can be expressed as:

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + Gx[n] \tag{3.2.1}$$

G is a gain factor and we will set $G = 1$ for our purposes. Equation (3.2.1) is a linear constant-coefficient difference equation where $b_0 = G = 1$, $x[n]$ is the input, $y[n]$ is the output and $h[n]$ is the impulse response, thus $h[n] = x[n] * y[n] \overset{\mathcal{Z}}{\longleftrightarrow} H(z) = X(z)Y(z) \iff H(z) = \dfrac{1}{1 + \sum_{k=1}^{N} a_k z^{-k}}$.
Given a particular random signal $y[n]$, the problem is to determine the predictor coefficients a_k, $k \in \{1, 2, \ldots, N\}$ in some manner. If we wish to approximate $y[n]$ using a linearly weighted

summation of its past samples we would write:

$$\hat{y}[n] = -\sum_{k=1}^{N} a_k y[n-k] \tag{3.2.2}$$

Where N is the prediction order. Consequently (3.2.1) $\overset{(3.2.2)}{\iff}$ $y[n] = \hat{y}[n] + x[n] \iff x[n] = y[n] - \hat{y}[n]$ which is the difference between the actual output $y[n]$ and the prediction $\hat{y}[n]$. This is the prediction error (sometimes it's called the estimation error or residual), $x[n] = e[n]$ and we can rewrite (3.2.1) as:

$$y[n] = \hat{y}[n] + e[n] \iff e[n] = y[n] - \hat{y}[n] \tag{3.2.3}$$

The Z-transform of (3.2.2) will be written as $\hat{Y}(z) = -\sum_{k=1}^{N} a_k z^{-k} Y(z) \iff \frac{\hat{Y}(z)}{Y(z)} = -\sum_{k=1}^{N} a_k z^{-k}$ where $\hat{y}[n] \overset{\mathcal{Z}}{\longleftrightarrow} \hat{Y}(z)$ and the transfer function of the prediction filter is:

$$A(z) = -\sum_{k=1}^{N} a_k z^{-k} \tag{3.2.4}$$

In (3.2.1) , with $G = 1$ it is true that the transfer function of the described system with input $x[n] = e[n]$ and output $y[n]$ is:

$$H(z) = \frac{1}{1 - A(z)} \tag{3.2.5}$$

The inverse system with input $y[n]$ and output $x[n] = e[n]$ has a transfer function:

$$H_i(z) = 1 - A(z) \tag{3.2.6}$$

Our task is to obtain the $\{a_k\}_{k=1}^{N}$ coefficients in order to find the optimal $\hat{y}[n]$ and significantly reduce the prediction error $e[n]$ $\forall n$. To assess the performance of this prediction error reduction, prediction gain is defined as:

$$PG = \frac{\sigma_y^2}{\sigma_e^2} \tag{3.2.7}$$

where σ_y^2 denotes the variance of the source signal $\{y[n]\}$ and σ_e^2 denotes the variance of the prediction error signal $\{e[n]\}$. Since the prediction error signal is likely to have a much smaller

variance than the source signal, the mean square quantization error (MSQE) could be significantly reduced if the prediction error signal is quantized in place of the source signal. This amounts to linear predictive analysis and synthesis of the audio signal where the linear predictive analysis filter has a transfer function equal to (3.2.6) and it can be seen as an FIR filter, whereas the linear predictive synthesis filter (or reconstruction filter) has a transfer function equal to (3.2.5) and it can be seen as an IIR filter (see definitions (2.5.2), (2.5.3)) and [29].

3.3 Optimal Linear Prediction

In order to achieve an optimal linear predictive model, we must first maximize the prediction gain (3.2.7) ([26]) or equivalently minimize σ_e^2 (see [12], [27]) that is the variance of the prediction error signal $\{e[n]\}$. Note that the average "AC" power quantity of the signal $\{e[n]\}$ is directly proportional to its variance. An approach to this problem is to use the mean square error (MSE) that is defined as:

$$\text{MSE} = \mathcal{E}\{e^2[n]\} = E_N \tag{3.3.1}$$

The optimization problem is to find the set of coefficients $\{a_k\}_{k=1}^N$ that minimizes the MSE, i.e maximizes the prediction gain and it can be written as:

$$\min_{\{a_k\}_{k=1}^N}\{\text{MSE}\} \overset{(3.2.3)}{=\!=\!=} \min_{\{a_k\}_{k=1}^N}\{\mathcal{E}\{(y[n]+\sum_{k=1}^{N} a_k y[n-k])^2\}\} \tag{3.3.2}$$

We must state that $\{e[n]\}$ is a discrete-time random signal, as well as $\{y[n]\}$. To find the parameters that minimize E_N, we differentiate equation (3.3.1) with respect to the i^{th} coefficient a_i and set the derivative equal to zero. We can assure that this assumption gives us a minimized MSE $= E_N$ because $\dfrac{d^2E_N}{d^2a_i} = y[n-i]^2 \geq 0$ therefore E_N is a convex function of a_i and if its first derivative with respect to a_i is zero (a horizontal tangent to the lowest point of a convex function) this will give us a global minimum: $\dfrac{dE_N}{da_i}=0 \iff \dfrac{d\mathcal{E}\{e^2[n]\}}{da_i}=0 \iff \dfrac{d\mathcal{E}\{(y[n]+\sum_{k=1}^{N} a_k y[n-k])^2\}}{da_i}=0 \iff$

$2\mathcal{E}\{(y[n]+\sum_{k=1}^{N}a_k y[n-k])\cdot y[n-i]\}=0 \iff \mathcal{E}\{y[n]y[n-i]\}+\mathcal{E}\{\sum_{k=1}^{N}a_k y[n-k]y[n-i]\}=0 \iff$
$\mathcal{E}\{y[n]y[n-i]\}+\sum_{k=1}^{N}a_k\mathcal{E}\{y[n-k]y[n-i]\}=0 \iff$

$$\sum_{k=1}^{N}a_k\mathcal{E}\{y[n-k]y[n-i]\}=-\mathcal{E}\{y[n]y[n-i]\} \tag{3.3.3}$$

In our case, $\{y[n]\}$ is a discrete-time random signal (Definition 3.1.1). Also, the values $y[1], y[2], \ldots, y[N]$ are samples inside a frame of an audio signal (2.1) (and this is why we fall in the case of intra-frame linear prediction) . This means that the discrete-time signal $\{y[n]\}$ is a discrete-time WSS stochastic process (Definition 2.1.6) and we can write (3.3.3) $\overset{(2.1.7)}{\iff}$

$$\sum_{k=1}^{N}a_k R_y[i-k]=-R_y[i], \qquad i\in\{1,\ldots,N\} \tag{3.3.4}$$

Using the averaging operator notation (Definition 2.1.3) the above equation can be written as:

$$\sum_{k=1}^{N}a_k\phi_{yy}[i,k]=-\phi_{yy}[i,0], \qquad i\in\{1,\ldots,N\} \tag{3.3.5}$$

$$\Updownarrow$$

$$\underbrace{\begin{bmatrix}\phi_{yy}[1,1] & \phi_{yy}[1,2] & \cdots & \phi_{yy}[1,N]\\ \phi_{yy}[2,1] & \phi_{yy}[2,2] & \cdots & \phi_{yy}[2,N]\\ \vdots & \vdots & \ddots & \vdots\\ \phi_{yy}[N,1] & \phi_{yy}[N,2] & \cdots & \phi_{yy}[N,N]\end{bmatrix}}_{\Phi}\cdot\underbrace{\begin{bmatrix}a_1\\ a_2\\ \vdots\\ a_N\end{bmatrix}}_{A}=-\underbrace{\begin{bmatrix}\phi_{yy}[1,0]\\ \phi_{yy}[2,0]\\ \vdots\\ \phi_{yy}[N,0]\end{bmatrix}}_{\Psi} \tag{3.3.6}$$

In matrix notation, the linear equation (3.3.5) has the representation $\Phi\cdot A=-\Psi$.

Since $\phi_{yy}[i,k]=\phi_{yy}[k,i] \implies$

$$\phi_{yy}[i,k]=R_y[|i-k|] \tag{3.3.7}$$

The matrix Φ is symmetric, and, because it arises in a least-squares problem, it is also positive-definite which guarantees that is invertible. Equations of this category are also refered to as Yule-Walker equations. Considering these statements we can also rewrite (3.3.6) as:

$$\begin{bmatrix} R_y[0] & R_y[1] & \cdots & R_y[N-1] \\ R_y[1] & R_y[0] & \cdots & R_y[N-2] \\ \vdots & \vdots & \ddots & \vdots \\ R_y[N-1] & R_y[N-2] & \cdots & R_y[0] \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = - \begin{bmatrix} R_y[1] \\ R_y[2] \\ \vdots \\ R_y[N] \end{bmatrix} \tag{3.3.8}$$

In this case, the matrix Φ is also a Toeplitz matrix, meaning that all its elements on each subdiagonal (from left to right) are equal. Since the coefficients $R_y[i-k]$ form an autocorrelation matrix, we shall call this method the autocorrelation method. An autocorrelation matrix is a symmetric, positive definite and Toeplitz matrix.

If we wish to prove equation (3.3.7) we shall write $\phi_{yy}[i,k] = \mathcal{E}\{y[n-i]y[n-k]\} \overset{m=n-i}{\Longleftrightarrow} \phi_{yy}[i,k] = \mathcal{E}\{y[m]y[m+i-k]\} = R_y[i-k]$ and $R_y[k-i] = \mathcal{E}\{y[m]y[m+k-i]\} = \mathcal{E}\{y[m+k-i]y[m]\} \overset{n=m+k-i}{\Longleftrightarrow} R_y[k-i] = \mathcal{E}\{y[n]y[n+i-k]\} = R_y[i-k]$ thus we can write $R_y[|i-k|] = \mathcal{E}\{y[m]y[m+i-k]\} = \phi_{yy}[i,k]$.

Now we must derive another important equation that is:

$$E_N = \phi_{yy}[0,0] + \sum_{k=1}^{N} a_k \phi_{yy}[0,k] \tag{3.3.9}$$

In order to prove that the above equation holds, we must use equation (3.3.1) and write that $E_N = \mathcal{E}\{\underbrace{(y[n] + \sum_{k=1}^{N} a_k y[n-k])^2}_{e^2[n]=(y[n]+\hat{y}[n])^2}\} = \mathcal{E}\{y^2[n] + 2y[n]\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=1}^{N} a_k y[n-k]\sum_{k=1}^{N} a_k y[n-k]\} =$
$\mathcal{E}\{y^2[n]\} + 2\mathcal{E}\{y[n]\sum_{k=1}^{N} a_k y[n-k]\} + \mathcal{E}\{\sum_{i=1}^{N} a_i y[n-k]\sum_{k=1}^{N} a_k y[n-k]\} =$
$\phi_{yy}[0,0] + 2\sum_{k=1}^{N} a_k \mathcal{E}\{y[n]y[n-k]\} + \sum_{i=1}^{N}\sum_{k=1}^{N} a_i a_k \mathcal{E}\{y[n-i]y[n-k]\} \iff$

$\iff E_N = \phi_{yy}[0,0] + 2\sum_{k=1}^{N} a_k\phi_{yy}[0,k] + \sum_{i=1}^{N} a_i \sum_{k=1}^{N} a_k\phi_{yy}[i,k]$. Now from (3.3.5) we know that $\sum_{k=1}^{N} a_k\phi_{yy}[i,k] = -\phi_{yy}[i,0], i \in \{1,\dots,N\} \rightarrow E_N = \phi_{yy}[0,0] + 2\sum_{k=1}^{N} a_k\phi_{yy}[0,k] - \sum_{i=1}^{N} a_i\phi_{yy}[i,0] \iff E_N = \phi_{yy}[0,0] + \sum_{k=1}^{N} a_k\phi_{yy}[0,k] + (\sum_{k=1}^{N} a_k\phi_{yy}[0,k] - \sum_{i=1}^{N} a_i\phi_{yy}[i,0])$. Since $\phi_{yy}[k,0] = \phi_{yy}[0,k]$ by setting $i = k$ we have $E_N = \phi_{yy}[0,0] + \sum_{k=1}^{N} a_k\phi_{yy}[0,k] + \underbrace{(\sum_{k=1}^{N} a_k\phi_{yy}[0,k] - \sum_{k=1}^{N} a_k\phi_{yy}[0,k])}_{=0} \iff$ $E_N = \phi_{yy}[0,0] + \sum_{k=1}^{N} a_k\phi_{yy}[0,k]$ that is identical to (3.3.9) and we've finally proved this equation. Another way to write (3.3.9) is:

$$E_N = R_y[0] + \sum_{k=1}^{N} a_k R_y[k] = \sum_{k=0}^{N} a_k R_y[k], \qquad a_0 = 1 \tag{3.3.10}$$

Equations (3.3.8) and (3.3.10) will be used in order to derive a very important algorithm for estimating the optimal set of coefficients $\{a_k\}_{k=1}^{N}$, namely, the Levinson-Durbin algorithm.

3.3.1 Derivation of the Levinson-Durbin algorithm

At first we can rewrite (3.3.8) and (3.3.10) with denoting the prediction order on the upper-right side of the coefficients $\{a_k\}_{k=1}^{N}$. We know that the optimum predictor coefficients satisfy the set of equations:

$$\begin{bmatrix} R_y[0] & R_y[1] & \cdots & R_y[N-1] \\ R_y[1] & R_y[0] & \cdots & R_y[N-2] \\ \vdots & \vdots & \ddots & \vdots \\ R_y[N-1] & R_y[N-2] & \cdots & R_y[0] \end{bmatrix} \cdot \begin{bmatrix} a_1^{(N)} \\ a_2^{(N)} \\ \vdots \\ a_N^{(N)} \end{bmatrix} = - \begin{bmatrix} R_y[1] \\ R_y[2] \\ \vdots \\ R_y[N] \end{bmatrix} \tag{3.3.11}$$

And the MSE is:

$$E_N = \sum_{k=0}^{N} a_k^{(N)} R_y[k], \qquad a_0 = 1 \tag{3.3.12}$$

From compacting (3.3.4) and (3.3.12), because they both contain the same autocorrelation values, we can derive the following set of linear equations:

$$\underbrace{\begin{bmatrix} R_y[0] & R_y[1] & R_y[2] & \cdots & R_y[N] \\ R_y[1] & R_y[0] & R_y[1] & \cdots & R_y[N-1] \\ R_y[2] & R_y[1] & R_y[0] & \cdots & R_y[N-2] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_y[N] & R_y[N-1] & R_y[N-2] & \cdots & R_y[0] \end{bmatrix}}_{\boldsymbol{R^{(N)}}} \cdot \underbrace{\begin{bmatrix} 1 \\ a_1^{(N)} \\ a_2^{(N)} \\ \vdots \\ a_N^{(N)} \end{bmatrix}}_{\boldsymbol{A^{(N)}}} = \underbrace{\begin{bmatrix} E_N \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\boldsymbol{e^{(N)}}} \tag{3.3.13}$$

This set of equations can be solved recursively by the Levinson-Durbin algorithm. This is done by successively incorporating a new autocorrelation value at each iteration and solving for the next higher-order predictor in terms of the new autocorrelation value and the previously found predictor. Thus, we state that for any order i, the set of equations in (3.3.13) can be represented in matrix notation as: $\boldsymbol{R^{(i)} \cdot A^{(i)} = e^{(i)}}$. We can show that the i^{th} solution can be derived from the $(i-1)^{th}$ solution, therefore if it is known that $\boldsymbol{R^{(i-1)} \cdot A^{(i-1)} = e^{(i-1)}}$ we shall derive the solution to $\boldsymbol{R^{(i)} \cdot A^{(i)} = e^{(i)}}$:

(I)

$$\underbrace{\begin{bmatrix} R_y[0] & R_y[1] & R_y[2] & \cdots & R_y[i-1] \\ R_y[1] & R_y[0] & R_y[1] & \cdots & R_y[i-2] \\ R_y[2] & R_y[1] & R_y[0] & \cdots & R_y[i-3] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_y[i-1] & R_y[i-2] & R_y[i-3] & \cdots & R_y[0] \end{bmatrix}}_{\boldsymbol{R^{(i-1)}}} \cdot \underbrace{\begin{bmatrix} 1 \\ a_1^{(i-1)} \\ a_2^{(i-1)} \\ \vdots \\ a_{i-1}^{(i-1)} \end{bmatrix}}_{\boldsymbol{A^{(i-1)}}} = \underbrace{\begin{bmatrix} E_{i-1} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\boldsymbol{e^{(i-1)}}} \tag{3.3.14}$$

(II) Append a zero to the vector $\boldsymbol{A^{(i-1)}}$ and multiply by the matrix $\boldsymbol{R^{(i)}}$:

$$\underbrace{\begin{bmatrix} R_y[0] & R_y[1] & R_y[2] & \cdots & R_y[i] \\ R_y[1] & R_y[0] & R_y[1] & \cdots & R_y[i-1] \\ R_y[2] & R_y[1] & R_y[0] & \cdots & R_y[i-2] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_y[i-1] & R_y[i-2] & R_y[i-3] & \cdots & R_y[1] \\ R_y[i] & R_y[i-1] & R_y[i-2] & \cdots & R_y[0] \end{bmatrix}}_{\boldsymbol{R^{(i)}}} \cdot \underbrace{\begin{bmatrix} 1 \\ a_1^{(i-1)} \\ a_2^{(i-1)} \\ \vdots \\ a_{i-1}^{(i-1)} \\ 0 \end{bmatrix}}_{\tilde{\boldsymbol{A}}^{(i-1)}} = \underbrace{\begin{bmatrix} E_{i-1} \\ 0 \\ 0 \\ \vdots \\ 0 \\ \gamma^{(i-1)} \end{bmatrix}}_{\tilde{\boldsymbol{e}}^{(i-1)}} \tag{3.3.15}$$

From this equation it is true that:

$$\gamma^{i-1} = R_y[i] + \sum_{j=1}^{i-1} a_j{}^{(i-1)} R_y[i-j] \tag{3.3.16}$$

(III) Due to the special symmetry of $\boldsymbol{R^{(i)}}$ that is a Toeplitz matrix, we can write the set of linear equations (3.3.15) in reverse order:

$$\underbrace{\begin{bmatrix} R_y[0] & R_y[1] & \cdots & R_y[i] \\ R_y[1] & R_y[0] & \cdots & R_y[i-1] \\ R_y[2] & R_y[1] & \cdots & R_y[i-2] \\ \vdots & \vdots & \ddots & \vdots \\ R_y[i] & R_y[i-1] & \cdots & R_y[0] \end{bmatrix}}_{\boldsymbol{R^{(i)}}} \cdot \underbrace{\begin{bmatrix} 0 \\ a_{i-1}^{(i-1)} \\ \vdots \\ a_1^{(i-1)} \\ 1 \end{bmatrix}}_{\tilde{\boldsymbol{A}}'^{(i-1)}} = \underbrace{\begin{bmatrix} \gamma^{(i-1)} \\ 0 \\ \vdots \\ 0 \\ E_{i-1} \end{bmatrix}}_{\tilde{\boldsymbol{e}}'^{(i-1)}} \tag{3.3.17}$$

(IV) Now equation (3.3.17) and equation (3.3.15) is combined according to:

$$\boldsymbol{R^{(i)}} \cdot \underbrace{\left[\tilde{\boldsymbol{A}}^{(i-1)} + k_i \cdot \tilde{\boldsymbol{A}}'^{(i-1)}\right]}_{\boldsymbol{A^{(i)}}} = \underbrace{\left[\tilde{\boldsymbol{e}}^{(i-1)} + k_i \cdot \tilde{\boldsymbol{e}}'^{(i-1)}\right]}_{\boldsymbol{e^{(i)}}} \tag{3.3.18}$$

We see that we are closer to derive the desired equation in the form of $\boldsymbol{R^{(i)} \cdot A^{(i)} = e^{(i)}}$ remembering that we started from equation $\boldsymbol{R^{(i-1)} \cdot A^{(i-1)} = e^{(i-1)}}$. Also, we define the coefficients k_i for $i = 1, 2, \ldots, N$ in a way, such that $\boldsymbol{e^{(i)}}$ has only one non-zero entry:

$$k_i = -\frac{\gamma^{(i-1)}}{E_{i-1}} \tag{3.3.19}$$

If we take the right hand side of equation (3.3.18) and use equation (3.3.19) it is clear that:

$$\begin{aligned}
\boldsymbol{e^{(i)}} &= \left[\underbrace{\begin{bmatrix} E_{i-1} \\ 0 \\ \vdots \\ 0 \\ \gamma^{(i-1)} \end{bmatrix}}_{\tilde{\boldsymbol{e}}^{(i-1)}} + k_i \cdot \underbrace{\begin{bmatrix} \gamma^{(i-1)} \\ 0 \\ \vdots \\ 0 \\ E_{i-1} \end{bmatrix}}_{\tilde{\boldsymbol{e}}'^{(i-1)}} \right] \\
&= \left[\begin{bmatrix} E_{i-1} \\ 0 \\ \vdots \\ 0 \\ \gamma^{(i-1)} \end{bmatrix} + \begin{bmatrix} \gamma^{(i-1)} \cdot k_i \\ 0 \\ \vdots \\ 0 \\ -\gamma^{(i-1)} \end{bmatrix} \right] \\
&= \begin{bmatrix} E_{i-1}(1 - k_i^2) \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}
\end{aligned} \tag{3.3.20}$$

If we take the left hand side of equation (3.3.18):

$$\boldsymbol{A^{(i)}} = \begin{bmatrix} 1 \\ a_1^{(i)} \\ \vdots \\ a_{i-1}^{(i)} \\ a_i^{(i)} \end{bmatrix} = \begin{bmatrix} 1 \\ a_1^{(i-1)} + k_i a_{i-1}^{(i-1)} \\ \vdots \\ a_{i-1}^{(i-1)} + k_i a_1^{(i-1)} \\ k_i \end{bmatrix} \tag{3.3.21}$$

From (3.3.19) using (3.3.16) we can derive:

$$k_i = -\frac{R_y[i] + \sum_{j=1}^{i-1} {a_j}^{(i-1)} R_y[i-j]}{E_{i-1}}, \qquad i = 1, 2, \ldots, N \tag{3.3.22}$$

From (3.3.21), we can derive the set of equations for updating the prediction coefficients:

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)}, \qquad j = 1, 2, \ldots, i-1 \tag{3.3.23}$$

$$a_i^{(i)} = k_i, \qquad i = 1, 2, \ldots, N \tag{3.3.24}$$

And from (3.3.20):

$$E_i = E_{i-1}(1 - k_i^2), \qquad i = 1, 2, \ldots, N \tag{3.3.25}$$

The Levinson-Durbin algorithm

We've already found the basic equations needed for the Levinson-Durbin algorithm. The next step is to be implemented as a pseudocode [30]. After this we can clearly see that if we already know the set of the PARCOR (PARtial CORrelation) coefficients, $\{k_i\}_{i=1}^{N}$,thus we don't need Levinson-Durbin to calculate the values of the set $\{k_i\}_{i=1}^{N}$, we conclude to the exact same

algorithm we've already implemented that is the k-parameters to coefficients algorithm (Algorithm 2.1) for converting from k-parameters of a lattice structure to the FIR impulse response coefficients using the equations (3.3.23) and (3.3.24).

Algorithm 3.1 Levinson-Durbin Algorithm

-Initialization-

$a_0^{(i)} = 1, \quad \forall i = 1, 2, \ldots, N$

$E_0 = R_y[0]$

$k_0 = -\dfrac{R_y[1]}{R_y[0]}$

for $i = 1, 2, \ldots, N$ **do**

$$k_i = -\frac{R_y[i] + \sum_{j=1}^{i-1} a_j{}^{(i-1)} R_y[i-j]}{E_{i-1}} \qquad (3.3.22)$$

$$a_i^{(i)} = k_i \qquad (3.3.24)$$

if $i > 1$ **then** $\forall j = 1, 2, \ldots, i-1$

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)} \qquad (3.3.23)$$

end if

$$E_i = E_{i-1}(1 - k_i^2) \qquad (3.3.25)$$

end for

$a_j = a_j^{(N)}, \quad \forall j = 1, 2, \ldots, N$

Now, after defining the Levinson-Durbin algorithm, we must state that (3.3.2) has been solved.

- It is important to clarify that $a_j^{(N)} = a^{(N)}[j]$ is the j^{th} prediction coefficient that is being multiplied by sample $y[n-j]$, where $j = 1, 2, \ldots, N$. Also, $i = 1, 2, \ldots, j+1$ is the order of prediction that we used in order to calculate each $a_j^{(i)}$ every time using $a_j^{(i-1)}$ and k_i, until index $i = N$.

- If we collect $\forall j$ all the $a_j^{(N)}$ coefficients and perform subtraction operation on all the elements of the set $\{a_j^{(N)} \cdot y[n-j]\}_{j=1}^{N}$ we can finally derive $\hat{y}[n] \approx y[n]$ (3.2.2) that is the optimal

prediction according to the linear predictive model, with prediction error $e[n] = \hat{y}[n] - y[n]$.

- The Levinson-Durbin algorithm has time complexity of $\mathcal{O}(N^2)$ which can significantly improved by Itakura's method ([31]).

- The linear predictive model is the reason we can achieve an intra-frame linear prediction. As we've already stated, each frame will have a significantly small duration because we want it to be a WSS stochastic process.

- It is important to say that our system is initially at rest. Therefore the prediction, that is $\hat{y}[n] = -\sum_{k=1}^{N} a_k^{(N)} \cdot y[n-k]$ where N is the prediction order, must always satisfy that $n \geq N$. Thus for the first $N+1$ samples we can assume prediction orders $\tilde{N} = 1, 2, \ldots, N$. That is, for sample 1 we assume prediction order of 0, for sample 2 we assume prediction order of 1, for sample 3 we assume prediction order of 2 etc. When we find a prediction for all the first N samples, then for every sample with $n > N$ we will predict normally using the pre-defined order. Hence, for the first $N+1$ samples we have $\hat{y}[n] = -\sum_{k=1}^{\tilde{N}} a_k^{(\tilde{N})} \cdot y[n-k]$. If the total number of samples is T then for the rest $T - (N+1)$ samples we will have $\hat{y}[n] = -\sum_{k=1}^{N} a_k^{(N)} \cdot y[n-k]$.

- From (3.3.25) we found that $E_i = E_{i-1}(1 - k_i^2)$. Notice that $E_i \geq E_{i-1} \geq E_N > 0$, so the greater the prediction order the less is the MSE (3.3.1). Also the MSE is always non-negative, thus $|k_i| < 1 \iff -1 < k_i < 1$. Also $k_i \neq \pm 1$ because MSE $= 0$ only for a deterministic process. If $k_i = 0$ then we have no performance improvement going from order i to $i+1$.

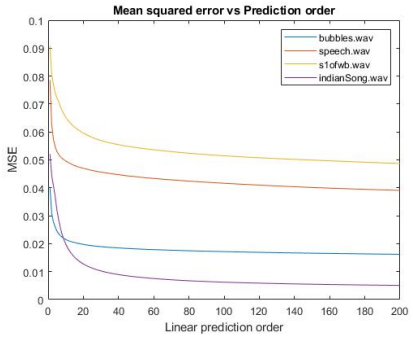


Figure 3.1: MSE versus prediction order for 4 audio signals.

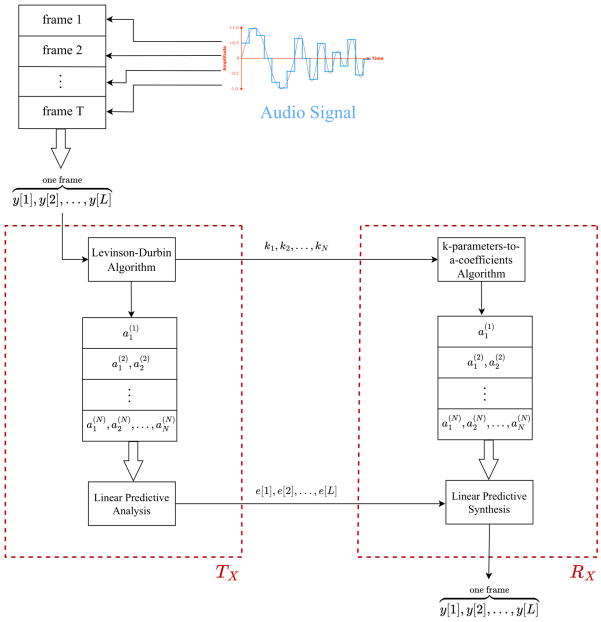


Figure 3.2: Transmitter vs Receiver for linear predictive modeling.

- In Figure 3.1 we can see that as the prediction order increases, the MSE decreases logarithmically.
- In Figure 3.2 we can see a transmitter and a receiver for a digital audio signal, that is the dequantized signal after applying the Pulse Code Modulation (Section 2.3) and it is framed in short time intervals. An underlying linear predictive modeling for each frame is implied.

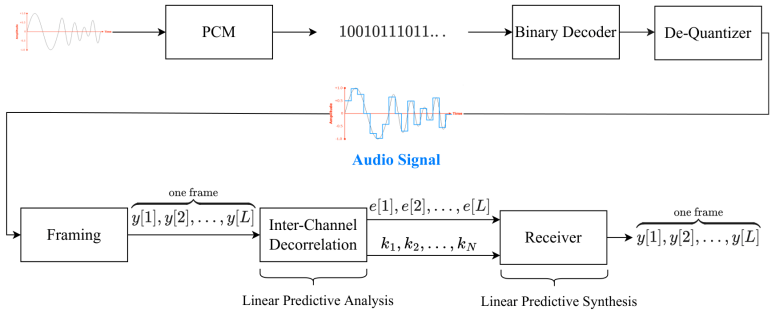


Figure 3.3: "Sending" the prediction error uncoded.

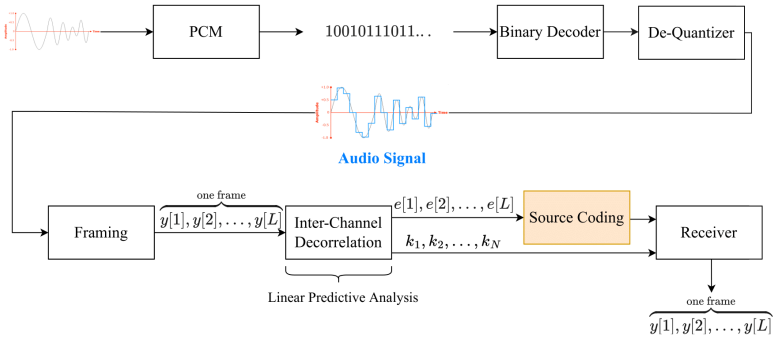


Figure 3.4: "Sending" the prediction error using source coding.

- In Figure 3.3 we depict the whole process of "sending" (storing in the computer) the dequantized audio signal, where the PCM block is the Pulse Code Modulation from Section 2.3, where we store the prediction error uncoded.

- In Figure 3.4 we again depict the same process but with an extra block: the Source Coding block. In Chapter 4 we will see what is Source Coding and why it is important in order to achieve compression of the signal. In communication systems compression is important for acheiving a faster rate of transmission.

Chapter 4

Source coding techniques

In this chapter we introduce some basic information theory definitions ([32], [33] , [34]) that are useful in order to understand the coding (encoding and decoding) process of the residual, also known as the prediction error. Afterwards, we'll proceed to the explanation of Huffman, Arithmetic and Golomb Codes which are very useful coding techniques and prove why (and when) they are optimal codes.

4.1 Basic definitions on information theory

Information theory is a very creative scientific subject, that is being employed in many applications in real life. We can see this subject being applied in many different fields such as telecommunications for achieving faster rates of either analog or digital signal transmission (source coding-entropy coding), error resilience of a communication system (channel coding), cryptography, quantum computing, biomedical applications that require lossless compression of images, audio compression and coding and many other significant fields. In this section we'll start by the definition of a source code, see all the kinds of different source codes that exist, introduce the Kraft-McMillan inequality and the definitions of entropy.

Definition 4.1.1 (**Source Code**)

A source code is a mapping $C : \mathcal{X} \to \mathcal{D}^*$. Hence $\forall x \in \mathcal{X}$ we can define a codeword $C(x) \in \mathcal{D}^*$ with length $l(x)$ measured in units/symbol. In addition $\mathcal{D}^*$ is a set of D-ary alphabet symbols. For example, suppose that we have a source $\mathcal{X} = \{x_1, x_2\}$ and we want to encode the source's values x_1 and x_2. If we assign their corresponding codewords as $C(x_1) = 11$ and $C(x_2) = 0$ we have a valid source code. In addition $l(x_1) = 2$ and $l(x_2) = 1$ coded bits per source symbol.

A source code can be singular, non-singular, uniquely decodable or prefix-free. Notice that a prefix-free code is also a uniquely decodable code but a uniquely decodable code is not always a prefix-free code. This can be clearly seen in Figure 4.1.

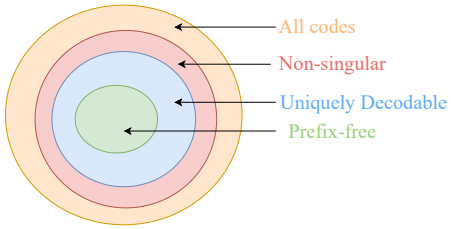


Figure 4.1: Graph representation of all possible source codes.

- **Singular code** is every source code that can perform a mapping from a source to a codebook. All codes are singular.
- **Non-singular code** is a source code that can assign a unique codeword to each source symbol, but when we decode we may get different codewords than those in the encoding procedure.
- **Uniquely decodable code** is a source code where if $\boldsymbol{x} \in \mathcal{X}^M$ and $\boldsymbol{x} = x_1 x_2 \dots x_M$ is the message that we want to send from a source, then $C(\boldsymbol{x}) = C(x_1 x_2 \dots x_M) = C(x_1)C(x_2) \dots C(x_M)$. The way we decode this type of source code is unique, but it has the disadvantage that before the decoding of each codeword, we might need to see a big portion of $\boldsymbol{x} = x_1 x_2 \dots x_M$.
- **Prefix-free code** is a type of source code that no codeword is a prefix of another codeword. It is also called instantaneous code because every codeword can be decoded instantly i.e., at the time it is received. This is a great advantage over uniquely decodable codes. Using a

mathematic notation, if $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ then the mapping $C : \mathcal{X} \to \mathcal{D}^*$ is prefix-free if $\forall i \in \{1, 2, \ldots, n\}$ and $\forall j \in \{1, 2, \ldots, n\}$, where $i \neq j$, $C(x_i)$ is not a prefix of $C(x_j)$ and vice versa.

Definition 4.1.2 (**Kraft-McMillan inequality**)

Let $\mathcal{X} = \{a_1, a_2, \ldots, a_M\}$ be the alphabet of a discrete source with codeword lengths $l(a_1), \ldots, l(a_M) = \{l(a_i)\}$ where $l(a_i) = l[C(a_i)]$ with $C(a_i)$ being the codeword of the symbol a_i, $i = 1, 2, \ldots, M$. Then it is true that:

- Every prefix-free code satisfies the Kraft-McMillan inequality:

$$\sum_{i=1}^{M} D^{-l(a_i)} \leq 1$$

  where $D = 2$ if the codewords are written in binary representation.

- If the Kraft-inequality is satisfied for a given set of codeword lengths $\{l(a_i)\}$, $i = 1, 2, \ldots, M$ then a prefix-free code can be constructed using the codeword lengths of the set $\{l(a_i)\}$.

Definition 4.1.3 (**Shannon's entropy**)

Let's consider a discrete source X that takes values from an alphabet $\mathcal{X}$, with probability mass function $p_X(x) = P(X = x)$. The D-ary entropy of the discrete source X, for representing the codewords in $D-$ary alphabet can be defined as:

$$H_D(X) = -\sum_{x \in \mathcal{X}} p_X(x) \log_D p_X(x)$$

To be more specific, we can replace x with a_i given $i = 1, 2, \ldots, |\mathcal{X}|$.

Definition 4.1.4 (**Expected codeword length**)

The expected codeword length is simply the sum of all the possible codeword lengths that

exist in a codebook $\mathcal{D}^*$ weighted by the probabilities of each length. Hence, if we have a source X that takes values from an alphabet $\mathcal{X} = \{a_1, a_2, \ldots, a_M\}$, a source probability mass function that its probability masses are in the set $\mathcal{A} = \{p_X(a_i)\}_{i=1,2,\ldots,M}$ and the respective codeword lengths $l(a_1), l(a_2), \ldots, l(a_M) = l[C(a_1)], l[C(a_2)], \ldots, l[C(a_M)]$ then we can define the expected codeword length as:

$$\bar{L} = \sum_{i=1}^{M} p_X(a_i) \cdot l(a_i) = \mathcal{E}_{p_X}\{l(X)\}$$

Theorem 4.1.1 (**Jensen Inequality**)

If we have a convex function $f(X)$ where X is a random variable and $E_{p_X}\{f(X)\}$ is the expected value of $f(X)$ with underlying probability mass function $p_X(x_i)$ with $i = 1, \ldots, k$ then the Jensen Inequality states that:

$$\mathcal{E}_{p_X}\{f(X)\} \geq f(\mathcal{E}_{p_X}\{X\}) \iff \sum_{i=1}^{k} p_X(x_i) \cdot f(x_i) \geq f(\sum_{i=1}^{k} p_X(x_i) \cdot x_i)$$

Proof: The above statement can be proved by induction. The first step is to suppose that for $k = 2$ probability mass points the above statement is true, thus:

$$p_X(x_1)f(x_1) + p_X(x_2)f(x_2) \geq f(p_X(x_1)x_1 + p_X(x_2)x_2)$$

The second step is the induction step in which we say that if the desired inequality is true for $k-1$ mass points then it must be true for k mass points also. To prove this step it is very helpful to define $\tilde{p}_X(x_i) = \dfrac{p_X(x_i)}{(1 - p_X(x_k))}$ for $i = 1, 2, \ldots, k-1$. Thus we have:

$$\sum_{i=1}^{k} p_X(x_i)f(x_i) = p_X(x_k)f(x_k) + (1 - \tilde{p}_X(x_k)) \sum_{i=1}^{k-1} \tilde{p}_X(x_i)f(x_i) \geq$$

$$p_X(x_k)f(x_k) + (1 - p_X(x_k))f(\sum_{i=1}^{k-1} \tilde{p}_X(x_i)x_i) \geq$$

$$\geq f(p_X(x_k)x_k + (1-p_X(x_k))\sum_{i=1}^{k-1}\tilde{p}_X(x_i)x_i) = f(\sum_{i=1}^{k}p_X(x_i)\cdot x_i)$$

therefore the theorem has been proved. Notice that the second inequality is true because we have supposed that $f(\cdot)$ is a convex function.

On the other hand, if we have a concave function $g(X)$ it is true that:

$$\mathcal{E}_{p_X}\{g(X)\} \leq g(\mathcal{E}_{p_X}\{X\}) \iff \sum_{i=1}^{k}p_X(x_i)\cdot g(x_i) \leq g(\sum_{i=1}^{k}p_X(x_i)\cdot x_i)$$

Having the proof of the Jensen inequality for a convex function, it is easy to prove the above statement that is about a concave function $g(\cdot)$. This can be achieved by saying that if $g(X)$ is a concave function then $-g(X)$ is a convex one (notice that the second derivative of $-g(X)$ has a different sign than the second derivative of $g(X)$). Therefore we can replace $f(X) = -g(X)$ and then multiply by -1 to the Jensen inequality for a convex function. This clearly gives the Jensen inequality for concave functions. We can also use the generalization of Jensen Inequality considering that $f(\cdot)$ is a convex function: $\mathcal{E}\{f(g(X))\} \geq f(\mathcal{E}\{g(X)\})$ or $f(\cdot)$ is a concave function: $\mathcal{E}\{f(g(X))\} \leq f(\mathcal{E}\{g(X)\})$.

Theorem 4.1.2 (**Entropy is the lowest bound of the expected codeword length**)

In this theorem we want to prove that the entropy of a source is always the lowest bound of the expected codeword length and this condition can not be relaxed. The best we can achieve in source coding is a codeword length that is equal to the entropy of the source but never less than it. The proof can be done easily by using the Jensen inequality (Theorem 4.1.1). At first using Definition 4.1.3 and Definition 4.1.4 we can write:

$$H_D(X) - \bar{L} = \underbrace{-\sum_{i=1}^{M}p_X(a_i)\log_D p_X(a_i)}_{H_D(X)}\underbrace{-\sum_{i=1}^{M}p_X(a_i)l(a_i)}_{\bar{L}} = -\sum_{i=1}^{M}p_X(a_i)(\log_D p_X(a_i) + l(a_i)) =$$

$$= \sum_{i=1}^{M} p_X(a_i)(-\log_D p_X(a_i) + \log_D D^{-l(a_i)}) = \sum_{i=1}^{M} p_X(a_i) \log_D \left(\frac{D^{-l(a_i)}}{p_X(a_i)} \right) \ (i)$$

It is important to note that $f(x) = \log_D(x)$ is a concave function therefore we can use Jensen's Inequality for this case (Theorem 4.1.1). Hence $(i) \iff H_D(X) - \bar{L} = \sum_{i=1}^{M} p_X(a_i) \log_D \left(\frac{D^{-l(a_i)}}{p_X(a_i)} \right) \leq \log_D \left(\sum_{i=1}^{M} p_X(a_i) \left(\frac{D^{-l(a_i)}}{p_X(a_i)} \right) \right) = \log_D \left(\sum_{i=1}^{M} D^{-l(a_i)} \right) \leq \log_D(1) = 0$ and the last inequality is true because of Kraft-McMillan inequality (Definition 4.1.2). As we can see using Jensen's Inequality and Kraft-McMillan inequality we've proved one of the most significant theorems in information theory which is that the expected codeword length is always greater (or equal) to the entropy of the source: $H_D(X) \leq \bar{L}$.

Definition 4.1.5 (**Kullback-Leibler distance**)

The Kullback-Leibler distance (or relative entropy) for discrete sources is defined for two different probability mass functions $q(\cdot)$ and $p(\cdot)$ as:

$$\mathfrak{D}(p||q) \triangleq \sum_{x} p(x) \log_D \frac{p(x)}{q(x)} = \mathcal{E}_p \left\{ \log_D \left(\frac{p(X)}{q(X)} \right) \right\}$$

The Kullback-Leibler distance is always non-negative ($\mathfrak{D}(p||q) \geq 0$):

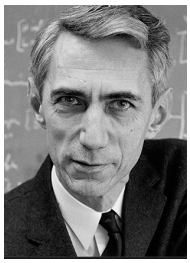
$$\mathfrak{D}(p||q) = \mathcal{E}_p \left\{ \log_D \left(\frac{p(X)}{q(X)} \right) \right\} = \mathcal{E}_p \left\{ -\log_D \left(\frac{q(X)}{p(X)} \right) \right\} \geq -\log_D \left(\sum_{x} p(x) \frac{q(x)}{p(x)} \right) =$$

$$= -\log_D(1) = 0$$

and we used the Jensen Inequality (Theorem 4.1.1) for the convex function case (because $f(x) = -\log(x)$ is a convex function) to prove the above.

4.2 Shannon-Fano code

In this section we'll explain the Shannon-Fano code ([35]) and when this type of code is optimal. At last we'll apply the Shannon-Fano code using a wrong source probability distribution and discuss the results.



(a) Claude Shannon

(b) Robert Fano

Figure 4.2

Before explaining the Shannon-Fano code it is sufficient to prove Theorem 4.1.2 using a different proof and find when $H_D(X) = \bar{L}$, i.e. when the expected codeword length is equal to its lowest bound that is the entropy of the source X that takes values from the alphabet $\mathcal{X} = \{x_1, x_2, \ldots\}$. It is helpful to define a probability mass function $q_X(x_i) = \dfrac{D^{-l(x_i)}}{c}$ where $c = \sum\limits_{\forall i} D^{-l(x_i)} \leq 1$ (see Kraft-McMillan inequality in Definition 4.1.2). We must note that this is not the underlying probability mass function of the source. Therefore we can define a different probability mass function for the discrete source that is written as $p_X(x_i)$. If one wants to prove that $q_X(x_i)$ is a legit probability distribution, then the following expression is true: $\sum\limits_{\forall i} q_X(x_i) = \sum\limits_{\forall i} \dfrac{D^{-l(x_i)}}{\sum\limits_{\forall i} D^{-l(x_i)}} = \dfrac{1}{\sum\limits_{\forall i} D^{-l(x_i)}} \cdot \sum\limits_{\forall i} D^{-l(x_i)} = 1$. In addition:

$$\bar{L} - H_D(X) = \sum_{\forall i} p_X(x_i) l(x_i) - \sum_{\forall i} p_X(x_i) \log_D \left(\frac{1}{p_X(x_i)} \right) =$$

$$= \sum_{\forall i} p_X(x_i) \log_D \left(\frac{p_X(x_i)}{D^{-l(x_i)}} \right) = \sum_{\forall i} p_X(x_i) \log_D \left(\frac{p_X(x_i)}{q_X(x_i)} \right) + \sum_{\forall i} p_X(x_i) \log_D \left(\frac{1}{c} \right)$$

$$= \mathfrak{D}(p_X||q_X) + log_D\left(\frac{1}{c}\right) \geq 0,$$

where $\mathfrak{D}(p_X||q_X) \geq 0$ and $c \leq 1$. Hence, $\bar{L} \geq H_D(X)$ and $\bar{L} = H_D(X)$ if and only if $c = 1$ (Kraft McMillan Inequality is satisfied with equality thus the code is complete) and $\mathfrak{D}(p_X||q_X) = 0$ meaning that $q_X(x_i) = p_X(x_i) \forall x_i \in \mathcal{X}$. This means that $q_X(x_i) = D^{-l(x_i)} = p_X(x_i)$. By this, one can understand that we can achieve the lowest bound of the expected codeword length , i.e $\bar{L} = H_D(X)$ when $p_X(x_i) = D^{-l(x_i)} \iff l(x_i) = \dfrac{1}{\log_D(p_X(x_i))}$, $\forall x_i \in \mathcal{X}$.

This corollary implies that $l(x_i) \in \mathbb{N}$ if and only if $p_X(x_i) = D^{-l(x_i)}$. But if we can not control the probability distribution of the source how can we define a code that will have integer lengths? This question can be answered by using the Shannon-Fano code which states that every source symbol x_i will have a codeword $C(x_i)$ assigned, such that $l(x_i) = \left\lceil \dfrac{1}{\log_D(p_X(x_i))} \right\rceil = l_{SF}(x_i)$. We can clearly see that $\sum\limits_{\forall i} D^{-l_{SF}(x_i)} = \sum\limits_{\forall i} D^{-\left\lceil \frac{1}{\log_D(p_X(x_i))} \right\rceil} \leq \sum\limits_{\forall i} D^{-\log_D\left(\frac{1}{p_X(x_i)}\right)} = \sum\limits_{\forall i} p_X(x_i) = 1$. Hence, the Kraft-McMillan inequality is true (Definition 4.1.2) and we can construct an instantaneous,i.e prefix-free code encoding our source symbols with lengths $l_{SF}(x_i) = \left\lceil \log_D\left(\dfrac{1}{p_X(x_i)}\right) \right\rceil$.

4.2.1 Optimality of the Shannon-Fano code

In order to discuss the optimality of the Shannon-Fano code we start by stating that for a source symbol x_i:

$$l_{SF}(x_i) = \left\lceil \log_D\left(\frac{1}{p_X(x_i)}\right) \right\rceil$$

$$\implies \log_D\left(\frac{1}{p_X(x_i)}\right) \leq l_{SF}(x_i) < \log_D\left(\frac{1}{p_X(x_i)}\right) + 1$$

$$\implies \sum_{\forall i} p_X(x_i) \log_D\left(\frac{1}{p_X(x_i)}\right) \leq \sum_{\forall i} p_X(x_i) l_{SF}(x_i) < \sum_{\forall i} p_X(x_i) \log_D\left(\frac{1}{p_X(x_i)}\right) + \underbrace{\sum_{\forall i} p_X(x_i)}_{=1}$$

$$\implies \quad H_D(X) \leq \bar{L}_{SF} < H_D(X) + 1 \text{ (D-ary digits / source symbol)} \tag{4.2.1}$$

We can optimize this code, with block coding and assuming that the symbols of the source are i.i.d (independent identically distributed). If a random variable X and a random variable Y are independent then the D-ary entropy can be defined as:

$$H_D(X,Y) = -\sum_{\forall x,y} \underbrace{p_{X,Y}(x,y)}_{p_X(x)\cdot p_Y(y)} \log_D(p_{X,Y}(x,y)) = -\sum_{\forall x} p_X(x)\log_D(p_X(x)) - \sum_{\forall y} p_Y(y)\log_D(p_Y(y))$$

$$\iff H_D(X,Y) = H_D(X) + H_D(Y) \tag{4.2.2}$$

This can be further generalized if $X_1, X_2, \ldots, X_n$ are i.i.d random variables as:

$$H_D(X_1, X_2, \ldots, X_n) = H_D(X_1) + H_D(X_2) + \ldots + H_D(X_n) = n \cdot H_D(X) \tag{4.2.3}$$

where X is a random variable with the same underlying probability distribution as $X_1, \ldots, X_n$. Then, if we apply block coding for tuples of n elements then the expected codeword length will be decreased. So if we have a code $C^n_{SF} : \mathcal{X}^n \to \mathcal{D}^*$ where $C_{SF}(X_1, \ldots, X_n)$ is a codeword then:

$$H_D(X_1, X_2, \ldots, X_n) \le L_{(C^n_{SF})} < H_D(X_1, X_2, \ldots, X_n) + 1$$

$$\overset{(4.2.3)}{\iff} n \cdot H_D(X) \le L_{(C^n_{SF})} < n \cdot H_D(X) + 1 \text{ D-ary digits / n source symbols}$$

$$\iff H_D(X) \le \frac{L_{(C^n_{SF})}}{n} < H_D(X) + \frac{1}{n} \text{ D-ary digits / source symbol} \tag{4.2.4}$$

This means that as $n \to \infty$ then $\frac{L_{(C^n_{SF})}}{n} \overset{n\to\infty}{\longrightarrow} H_D(X)$.

In realistic applications, block coding with big values of n is not prefered because then we should create very large codebooks. For example if we have a source which takes values from an alphabet that contains M different symbols then we have M^n different combinations (M^n tuples) in the codebook, meaning that as n increases the spatial complexity of the code increases exponentially.

4.2.2 Coding with a wrong probability distribution

Suppose that have a discrete source X that takes values from an alphabet $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ with probability mass function $p_X(x_i)$ where $x_i \in \mathcal{X}$. If we decide to apply Shannon-Fano coding using another probability mass function $q_X(x_i)$ then we have codeword lengths of $l(x_i) = \left\lceil \log_D \left(\frac{1}{log_D(q_X(x_i))} \right) \right\rceil$. Then:

$$\begin{aligned}
&\sum_{\forall i} p_X(x_i) \log_D \left(\frac{1}{q(x_i)} \right) \leq \sum_{\forall i} p_X(x_i) \left\lceil \log_D \left(\frac{1}{q(x_i)} \right) \right\rceil < \sum_{\forall i} p_X(x_i) \log_D \left(\frac{1}{q(x_i)} \right) + 1 \\
&\iff \sum_{\forall i} p_X(x_i) \log_D \left(\frac{p_X(x_i)}{q(x_i)} \right) + \sum_{\forall i} p_X(x_i) \log_D \left(p_X(x_i) \right) \leq \sum_{\forall i} p_X(x_i) \log_D \left(l(x_i) \right) \\
&\qquad < \sum_{\forall i} p_X(x_i) \log_D \left(\frac{p_X(x_i)}{q(x_i)} \right) + \sum_{\forall i} p_X(x_i) \log_D \left(p_X(x_i) \right) + 1 \\
&\iff H_D(X) + \mathfrak{D}(p_X || q_X) \leq \bar{L}_{SF} < H_D(X) + \mathfrak{D}(p_X || q_X) + 1 \text{ (D-ary digits / source symbol)}
\end{aligned} \tag{4.2.5}$$

If we compare (4.2.1) and (4.2.5) it is clear that the Kullback Leibler distance that is added to the lowest and highest bound of the expected codeword length for the optimal Shannon-Fano code makes this type of code "less optimal" if we use a probability mass function that is not equal to the original, i.e using a different probability distribution than the probability distribution of the source. If both probability mass functions are equal ($q_X(x_i) = p_X(x_i)$, $\forall x_i \in \mathcal{X}$) then the Kullback Leibler distance is zero so we have an optimal Shannon-Fano code. We can also notice that the "further away" the probability mass function $q_X(\cdot)$ is from $p_X(\cdot)$ the greater the expected codeword length will be compared to the entropy of the source, i.e its lowest bound.

4.3 Huffman Coding

The main idea that was developed by David A. Huffman ([36]) for coding a source is that we can assign small codeword lengths to the source symbols with the highest probabilities and bigger codeword lengths to the source symbol with lesser probabilities by adding the two least probable symbol probabilities recursively and finally create D-ary tree. In this way the expected codeword length can be minimized. Huffman is an optimal symbol code.

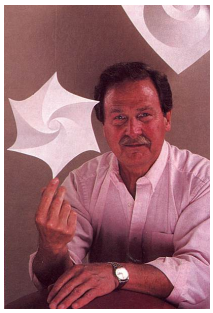


Figure 4.3: David Albert Huffman

Lemma 1

If we have a source X that takes values from an alphabet $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and $i \in \{1, 2, \ldots, n\}$. If $i \neq j$, for achieving a minimum expected codeword length:

$$p_X(x_i) > p_X(x_j) \implies l(x_i) \leq l(x_j)$$

To prove this, we will start by assuming that the code C_1 for the source X is optimal. Suppose also that C_2 is the same as C_1 but the only thing that changes is that the codewords x_i and x_j will be permuted-swapped. For this reason:

$$\bar{L}_{C_2} - \bar{L}_{C_1} =$$

$$\sum_k p_X(x_k) l'(x_k) - \sum_k p_X(x_k) l(x_k) = p_X(x_i) \underbrace{l(x_j)}_{l'(x_i)} + p_X(x_j) \underbrace{l(x_i)}_{l'(x_j)} - (p_X(x_i) l(x_i) + p_X(x_j) l(x_j)) =$$

$$= p_X(x_i)(l(x_j) - l(x_i)) - p_X(x_j)(l(x_j) - l(x_i))$$

$$= (p_X(x_i) - p_X(x_j))(l(x_j) - l(x_i)) \geq 0 \implies l(x_i) \leq l(x_j)$$

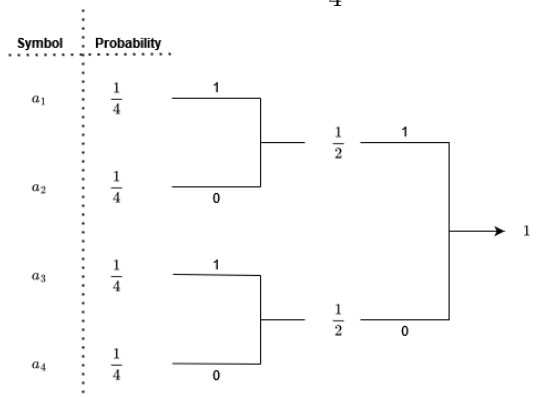
Lemma 2

The prefix-free code with the minimum expected codeword length is complete otherwise it is not optimal because we can delete a leaf from the tree and have a lesser expected codeword length. In addition the codewords with the greatest codeword lengths have equal sized lengths because if they don't one of the two would be an intermediate node therefore the other could have a lesser codeword length. We should state here that these codewords only differ in the last bit. This lemma will be clarified in the examples that we'll present below.

If we want to encode in binary the Huffman algorithm first chooses the two least probable symbols and assigns to them arbitrarily a "0" and a "1" respectively. These two symbols can be called "siblings" and they can be merged to a new symbol with a probability equal to the summation of their two respective probabilities. At the next step of the recursion, the comparison of the probabilities will be done for one symbol less because the previous two symbols have been merged into one. Huffman coding is a "greedy" algorithm yielding optimal solutions in each step of the recursion (at each step of the recursion merge the two least probable symbols into one). This method will lead to greater codeword lengths for lesser symbol probabilities and vice versa.

Example 4.3.1

Consider a source X that takes values from an alphabet $\mathcal{X} = \{1, 2, 3, 4\}$ and $P_X(1) = P_X(2) = P_X(3) = P_X(4) = \frac{1}{4}$.



$a_1 \to 11$

$a_2 \to 10$

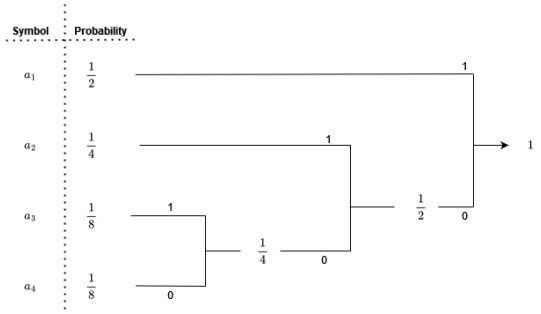
$a_3 \to 01$

$a_4 \to 00$

The entropy of the binary source is $H_2(X) = -\sum_{x\in\mathcal{X}} p_X(x)\log_2 p_X(x) = -4\cdot\log_2\left(\frac{1}{4}\right) = 2$ (bits/symbol) and the expected codeword length is $\bar{L} = \sum_{x\in\mathcal{X}} p_X(x)\cdot l(x) = (2\cdot\frac{1}{4})\cdot 4 = 2$ (bits/symbol). In this case $\bar{L} = H_2(X)$ and the resulting code is optimal.

Example 4.3.2

Consider a source X that takes values from an alphabet $\mathcal{X} = \{1,2,3,4\}$ and $P_X(1) = \frac{1}{2}$, $P_X(2) = \frac{1}{4}$, $P_X(3) = \frac{1}{8}$, $P_X(4) = \frac{1}{8}$.



$a_1 \to 1$

$a_2 \to 01$

$a_3 \to 001$

$a_4 \to 000$

The entropy of the binary source is $H_2(X) = -\sum_{x\in\mathcal{X}} p_X(x)\log_2 p_X(x) = 1.75$ (bits/symbol) and the expected codeword length is $\bar{L} = \sum_{x\in\mathcal{X}} p_X(x)\cdot l(x) = 2.5$ (bits/symbol). In this case $H_2(X) \le \bar{L} < H_2(X)+1$ and the resulting code is achieving a minimum expected codeword length. We would have the exact same results if we used a Shannon-Fano code given that the probabilities are negative powers of 2.

Of course the Huffman code can be extended using D-ary codebooks. Then in every recursion we merge the D symbols that are the least probable ones. The choice of D depends on the application. If we want to transmit a message using a 3PSK (or 3FSK) modulation we must use a ternary code ($D = 3$) and merge in every iteration the 3 least probable symbols. In order for this type of coding to be optimal we must keep in mind that we may have to fill the huffman tree with some extra (idle) symbols of zero probability. To do this we can say that the total number of symbols to encode and idle symbols is n and x respectively and the number of steps of the code is k. Therefore, we must find the minimum x such that $k \in \mathbb{N}$ using the formula $(x+n)-(D-1)\cdot k = D$.

4.3.1 Huffman Code Optimality

Since knowing that Huffman is optimal for every source X in the sense that:

$$H(X) \leq \bar{L}_H < H(X) + 1 \quad \text{(bits / symbol)} \tag{4.3.1}$$

we can apply block coding, as we did with the Shannon-Fano code, meaning that we can create all the n-tuples $(x_1, x_2, \ldots, x_n) \in \mathcal{X}^n$ where $\mathcal{X} = \{x_1, x_2, \ldots, x_M\}$, calculate the probability for each one of them and encode the M^n total n-tuples. Then for a source $X \in \mathcal{X}^n$:

$$\begin{aligned} H(X_1, \ldots, X_M) \leq \bar{L}_H &< H(X_1, \ldots, X_M) + 1 \\ \iff nH(X) \leq \bar{L}_H &< nH(X) + 1 \\ \iff H(X) \leq \frac{\bar{L}_H}{n} &< H(X) + \frac{1}{n} \quad \text{(bits / symbol)} \end{aligned} \tag{4.3.2}$$

This is a very good improvement over (4.3.1) and the more we increase n the closer the expected codeword length is to the entropy of the source. But unfortunately, it is a lot of times impractical, as the codebook needed for decoding increases exponentially. For example if we have a source of 10 symbols then for the simple Huffman Coding we exactly 10 entries in the codebook. Only with $n = 2$ we will have $10^2 = 100$ entries in the codebook, and with $n = 6$ we will need 1 million entries in the codebook, so every time in the decoding stage, we must search the encoded value inside this huge codebook. This is not only slow, but increases spatial complexity so much and for this reason, in real life applications it is really rare to see Huffman Block coding. Although this result is important, in order to compare with more coding techniques, as we will see in the rest of this chapter.

4.4 Arithmetic Coding

In this section a powerful source coding technique will be explained, named Arithmetic Coding([37],[38]), that is a computationally simple algorithm which can compute the range $[p, q)$ for a particular message given only a table of probabilities for each symbol. The initial range is $[0, 1)$ and in each iteration of the algorithm, the initial range gets narrowed depending on the next symbol's probability. After processing the last symbol, a value $x \in [p, q)$ is chosen as a representative for encoding the message. In other words, we encode the source's sequence of symbols by using their cumulative probabilities in order to narrow down the probability range of all the symbols in the sequence. By applying typical linear scaling to the narrowed range that each iteration of the algorithm gives, we can scale the cumulative probability of the present symbol that we wish to encode. The example below will be used to clarify the above [39].

Example 4.4.1

Suppose that we wish to encode the message "$aabc$" using the table below:

Symbol	Probability	Cumulative Probability
a	.100	.000
b	.010	.100
c	.001	.110
d	.001	.111

In this example we want to subdivide (narrow) the unit interval $[0, 1)$ for the arithmetic code using the table above keeping in mind that the input data string is "$aabc$" . Our arithmetic code is a type of FIFO arithmetic code. The first symbol "a" is implied as an arbitrary fractional number inside the interval $[0, .100)$. After that, we subdivide in the exact same proportions the interval $[0, .100)$ that are defined by the cumulative probabilities of each symbol. This is done by typical linear scaling. The same procedure is being done for the second symbol "a" and its subinterval is $[0, .010)$ so in the third step our current interval $[0, .100)$ will be narrowed into $[0, .010)$. Afterwards,

the third symbol is b and the subinterval that belongs to it is $[.001, .0011)$ so in the fourth step the current interval $[0, .010)$ will be narrowed into $[.001, .0011)$. At last, the fourth symbol c defines the subinterval $[.0010110, .0010111)$. This process is depicted clearly in Figure 4.4. We also must not forget to find a representative value $x \in [.0010110, .0010111)$ and this is the value that we'll store or transmit. If we choose $x = .0010110$ then we will store or transmit 0010110 that is 7-bits. Also notice that if the first symbol is "a" and the second symbol is "a" too, then the length of the

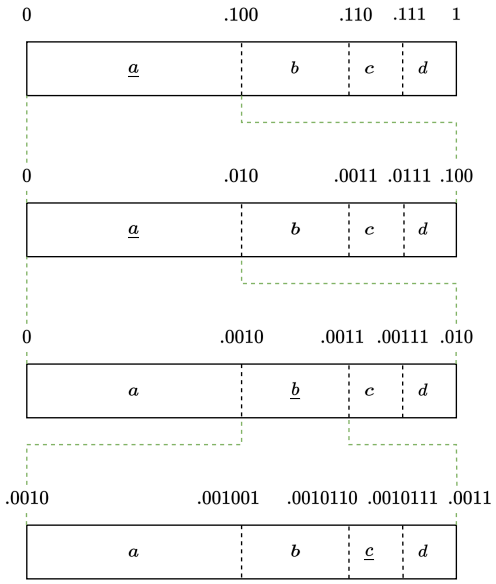


Figure 4.4: Arithmetic encoding of the sequence "$aabc$" .

second subinterval that belongs to the second a is $p(a) \times p(a) = .1 \times .1 = .01$. In addition the length of the second subinterval if the second symbol was "b" would be $p(a) \times p(b) = .1 \times .01 = .001$ and to verify, $.0011 - .0010 = .001$. This is an alternative way to see how the subintervals are formed if you don't prefer to apply linear scaling. Keep in mind that we chose the representative $x = 0010110$ that is about to be decoded. The decoder will read this value as a fractional one (.0010110) and then it will make comparisons to find the original message. In our case $0 < x \leq .1$, therefore the first symbol is "a". Then it will mimic the encoder procedure for linear scaling and will compare again, this time $0 < x \leq .010$ therefore the second symbol is "a" . For the third step, the comparison will be $.0010 < x \leq .0011$, hence the third symbol is b . Notice that in every step

x lies in one and only one interval of the four possible adjoint interval and after the comparison has been done we perform linear scaling to narrow the subintervals. At last, at the fourth step, the decoder will understand that $.0010110 < x \leq .0010111$. But how do we know when the decoder will stop? For this there are two solutions. The first one is to add to our alphabet one more symbol that is the EOF (End Of File) symbol and concatenate it at the rightmost side of the message. Another solution is send the message length in along with the entire message, so when the decoder sees that the given message had a length of 4 (length("$aabc$") = 4) then it will stop decoding.

Using this method to encode a sequence of symbols, the encoding does not give fixed symbol lengths and does not assign codewords for each symbol in a codebook (as in Huffman coding or in Shannon-Fano coding), but it assigns only one single codeword for the whole input sequence. Although this may be attractive, arithmetic coding in its primary form presents a precision problem. In theory we can easily imply infinite precision but computers have a limited amount of memory. Therefore, from a practical point of view, if we wish to encode a very large sequence using Arithmetic Coding it may be needed to import libraries in our software that support the implementation of really large integers. Nevertheless, this is not a great way to solve this problem efficiently in order to store really large representative values because we must either tether a big portion of memory which leads to a long time for every arithmetic operation to be done, or truncate some bits of the representative. This truncation may lead to incorrect decoding, if the magnitude of the difference between the original bit string and the truncated is greater (or equal) than the magnitude of any particular symbol's probability, otherwise the decoding will be correct. If we have a correct decoding though this doesn't mean that the compression efficiency of the scheme will not be negatively affected.

4.4.1 Implementation of Arithmetic Coding

Suppose that we have a discrete source X that takes values from an alphabet of integers $\mathcal{X} = \{0, 1, \ldots, n\}$. If we see X as a random variable then $X \sim \big(p_X(0), p_X(1), \ldots, p_X(n)\big)$. Without loss of generality, suppose that the input message is $x_1 x_2 \cdots x_k$ where $x_1, x_2, \ldots, x_k \in \mathcal{X}$.

The cumulative probability range of each symbol in the message will be:

$$[F_X(x_{i-1}), F_X(x_i)) \ \ \forall i \in \{1, 2, \ldots, k\} \tag{4.4.1}$$

where: $F_X(x_{i-1}) = \sum_{j=1}^{i-1} P_X(x_j)$, $F_X(x_i) = F_X(x_{i-1}) + P_X(x_i)$.
Alternatively by setting $F_X(x_{i-1}) = CP_{low}(x_i)$ and $F_X(x_i) = CP_{high}(x_i)$:

$$CP_{low}(x_i) = \begin{cases} 0 & , \text{ i=1} \\ CP_{low}(x_{i-1}) + p_X(x_{i-1}) & , \text{ otherwise} \end{cases} \tag{4.4.2}$$

$$CP_{high}(x_i) = \begin{cases} p_X(x_1) & , \text{ i=1} \\ CP_{low}(x_i) + p_X(x_i) & , \text{ otherwise} \end{cases} \tag{4.4.3}$$

It is important to remember that the lowest value of the x_i symbol's subinterval is $CP_{low}(x_i)$ and the highest is $CP_{high}(x_i)$. Also notice that $CP_{high}(x_i) = CP_{low}(x_{i+1})$.

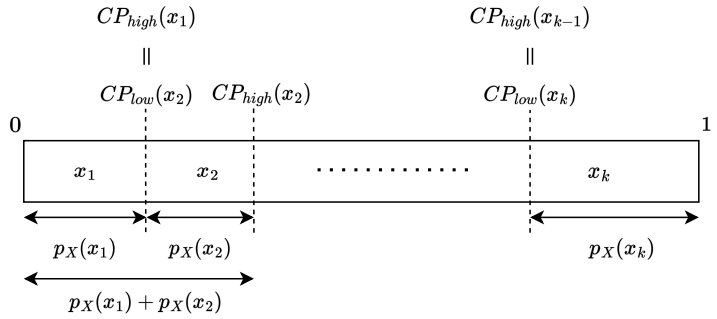


Figure 4.5: Initial cumulative probability ranges for the symbols $x_1, x_2, \ldots, x_k$.

Until now, we've completed the implementation for the initialization of the algorithm. To move forward, it is crucial to define somehow every possible message that we might get as an input to

our arithmetic encoder. At first we must define $EOF = 0$, where EOF is the $EndOfFile$ symbol. We also define the set $\boldsymbol{X}^{(k)}$ that contains all finite length strings, such that:

$$\boldsymbol{X}^{(k)} = \{x_1 \cdots x_k 0 : x_i \in \mathcal{X} - \{0\},\ k \in \mathbb{N}\} \tag{4.4.4}$$

The joint probability mass function of a random finite length string in $\boldsymbol{X}^{(k)}$, where $\boldsymbol{x} = x_1 \cdots x_k 0$ is the input sequence, is defined as:

$$\tilde{p}(\boldsymbol{x}) = p_{X_1,\ldots,X_k,X_{k+1}}(x_1,\ldots,x_k,0) = p_{X_1}(x_1)\cdots p_{X_k}(x_k)\cdot p_{X_{k+1}}(0) \tag{4.4.5}$$

The above equation is true because the random variables $X_1,\ldots,X_k,X_{k+1}$ are independent and identically distributed (iid). The joint probability mass function $p_{X_1,\ldots,X_k,X_{k+1}}(\cdot)$ is a probabilistic model that we can define on the probability mass function $\big(p_X(0),p_X(1),\ldots,p_X(n)\big)$. In order to be sure that $p_{X_1,\ldots,X_k,X_{k+1}}(\cdot)$ is an actual probability mass function:
Let $X_1,\ldots,X_K,X_{K+1}\sim\big(p_X(0),p_X(1),\ldots,p_X(n)\big)$ be some sequence of iid random variables. The sequence we get by concatenating these random variables will be: $X_1X_2\cdots X_K0$, where $EOF = 0$ and let K be a random variable too, such that:

$$K+1 = min\{i : X_i = 0\} \implies K\sim Geometric(p_X(0)) \tag{4.4.6}$$

The probability of a sequence $x_1\cdots x_K0$ to appear is:

$$P(X_1 = x_1,\ldots,X_K = x_k,X_{K+1} = 0) = P(X_1 = x_1)\cdot\cdots P(X_K = x_K)\cdot P(X_{K+1} = 0) \tag{4.4.7}$$

In addition, it is indispensable to declare that if $\boldsymbol{x} = x_1\cdots x_k0$ is a given sequence, then:

$$\sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x}) = 1 \tag{4.4.8}$$

Example 4.4.2

Suppose we have an input sequence to the Arithmetic Encoder that is $\boldsymbol{x} = x_1\ldots x_k0$. So in this case

we will se clearly that, beginning from the initial range of the message that is $[0, 1)$, as each symbol is processed their respective subintervals (and generally the probable subintervals) are getting more and more narrow to a portion allocated to each symbol. This is a helpful example to understand the probabilistic model of (4.4.5). This process is depicted in Figure 4.6 .

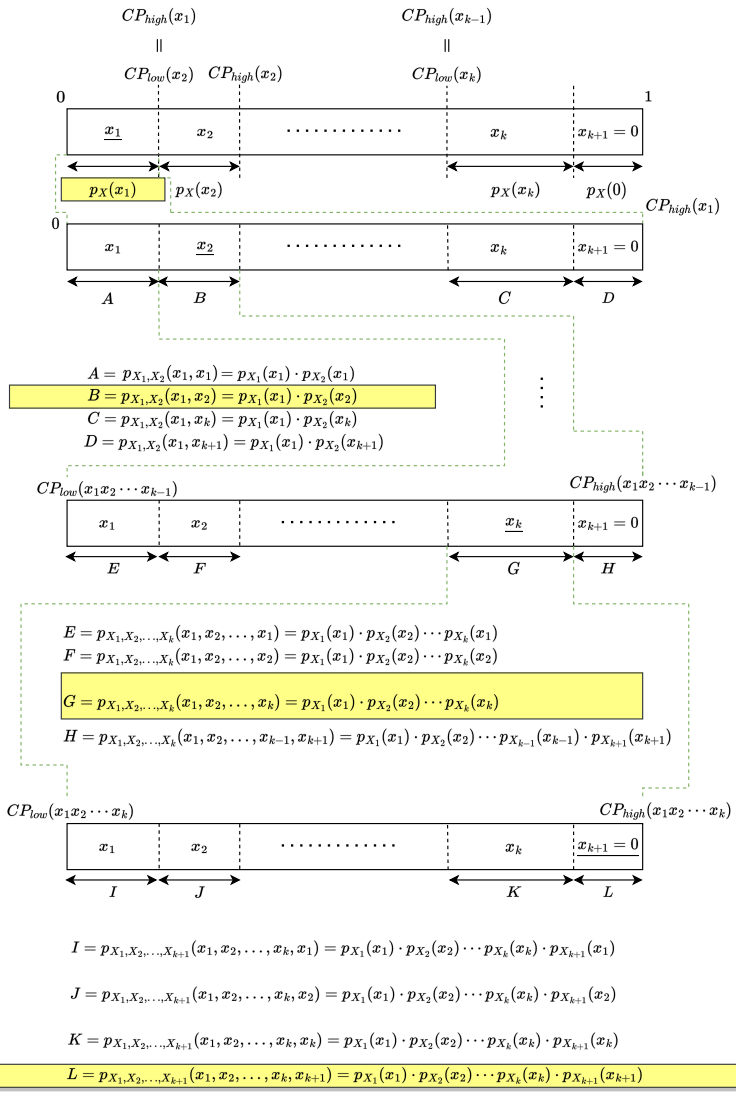


Figure 4.6: Subdividing the intervals to encode the input sequence $\boldsymbol{x} = x_1x_2\cdots x_k0$.

As we can see in Figure 4.6 the final interval for the whole sequence $x_1x_2\cdots x_k0$ will be $[low, high)$ where $high = CP_{high}(x_1x_2\cdots x_k)$ and $low = high - L$. The only thing missing is the definition of $CP_{high}(x_1x_2\cdots)$ and of $CP_{low}(x_1x_2\cdots)$ respectively that will be shown after this example. At last, we must not forget to find a representative, i.e. a tag value u such that $u \in [low, high)$. This is the value that we will consider as the encoded value by our Arithmetic Encoder. Of course, instead of the $EOF = 0$ symbol, we could transmit (or store) the length of the sequence.

Derivation of the sub-intervals and the representative

When the encoder has read the first symbol x_n then the $n-$th sub-interval that concerns this symbol is chosen as the current base interval. As we stated in (4.4.2) and in (4.4.3) the current sub-interval is $[low, high) = \big[CP_{low}(x_n), CP_{high}(x_n)\big)$. The other subintervals for the next step of the algorithm will change in every iteration. Therefore, if we state that the second symbol of the sequence is x_m, by setting $w = CP_{high}(x_n) - CP_{low}(x_n)$ the new interval will be:

$$\begin{aligned}[low, high) &= \\ &= \big[CP_{low}(x_n) + w \cdot CP_{low}(x_m),\ CP_{low}(x_n) + w \cdot CP_{high}(x_m)\big) \\ &= \big[CP_{low}(x_nx_m),\ CP_{high}(x_nx_m)\big)\end{aligned} \tag{4.4.9}$$

where:

$$CP_{low}(x_nx_m) = CP_{low}(x_n) + w \cdot CP_{low}(x_m) \tag{4.4.10}$$

$$CP_{high}(x_nx_m) = CP_{low}(x_n) + w \cdot CP_{high}(x_m) \tag{4.4.11}$$

Of course now, $a = CP_{low}(x_nx_m)$ and $b = CP_{high}(x_nx_m)$ as the new interval will be $[low, high)$. If we had a third symbol x_q then it is imperative to calculate $CP_{low}(x_nx_mx_q)$ and $CP_{high}(x_nx_mx_q)$.

The new sub-interval $[low, high)$ will have low and $high$ as:

$$low = CP_{low}(x_n x_m x_q) = CP_{low}(x_n x_m) + w \cdot CP_{low}(x_q) \tag{4.4.12}$$

$$high = CP_{high}(x_n x_m x_q) = CP_{low}(x_n x_m) + w \cdot CP_{high}(x_q) \tag{4.4.13}$$

but this time $w = CP_{high}(x_n x_m) - CP_{low}(x_n x_m)$. This process is repeated until the whole input sequence is encoded. We can also see this process, as repeatedly linear scaling the values $CP_{low}(\cdot)$ and $CP_{high}(\cdot)$ according to every new symbol's corresponding cumulative probabilities. We notice that as we are subdividing, i.e. narrowing the interval $[low, high)$, low and $high$ are getting closer and closer together.

Furthermore the representative, i.e. the tag for the final sub-interval can be defined as the lowest value of the final sub-interval $[low, high)$, but the middle value should suffice too. Hence we can define the tag for a sequence $\boldsymbol{x} = x_1 \cdots x_k 0$ in two possible ways:

$$\bar{T}_X(\boldsymbol{x}) = \underbrace{\sum_{\boldsymbol{y}<\boldsymbol{x}} \tilde{p}(\boldsymbol{y})}_{CP_{low}(\boldsymbol{x})} + \frac{1}{2} \cdot \tilde{p}(\boldsymbol{x}) \tag{4.4.14}$$

or:

$$\bar{T}_X(\boldsymbol{x}) = CP_{low}(\boldsymbol{x}) \tag{4.4.15}$$

Notice that the final subinterval is $[low,\ high) = \big[CP_{low}(\boldsymbol{x}), CP_{high}(\boldsymbol{x})\big)$.

Infinite precision algorithms

Here we will represent the infinite precision case, that we have already explained and see some pseudocodes in order to understand it better, before moving to the finite precision case.

Infinite Precision Arithmetic Encoder :

The encoding procedure has been already explained in the subsections above. Below we can see an implementation of the infinite precision encoder. Notice that this pseudocode is not always practical but it will serve us to understand better the integer arithmetic coding scheme that will follow.

Algorithm 4.1 Infinite precision algorithm for the Arithmetic Encoder.

function ARITHMETICENCODERINF($\boldsymbol{x},\ \tilde{p}(\cdot)$):
    Compute $CP_{low}(u)$ and $CP_{high}(u)$ for each symbol u in $\boldsymbol{x}$
    $EOF = \boldsymbol{x}(end)$
    $low = 0$
    $high = 1$
    $x = \boldsymbol{x}(1)$
    while $(x \neq EOF)$ **do**
        $w = high - low$
        $low = low + w \cdot CP_{low}(x)$
        $high = low + w \cdot CP_{high}(x) - 1$
        $x = next(x \text{ in } \boldsymbol{x})$
    end while
    Choose a representative $\bar{T}_X(\boldsymbol{x})$ using (4.4.14) or (4.4.15).
    return $\bar{T}_X(\boldsymbol{x})$
end function

As the interval $[low, high)$ is being sub-divided the top bits of low and $high$ become the same and for this reason they can be immediately stored or transmitted, since they will not be affected by further sub-division. By that, we mean that instead of returning $\bar{T}_X(\boldsymbol{x})$ we would achieve greater compression if we would return $\left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})} \leq \bar{T}_X(\boldsymbol{x})$ that is a truncated version of $\bar{T}_X(\boldsymbol{x})$ where its length is now $l(\boldsymbol{x})$. In [37] we see a proposed $l(\boldsymbol{x}) = \left\lceil log\left(\frac{1}{\tilde{p}(\boldsymbol{x})}\right)\right\rceil + 1$ that is 1 bit longer than a Shannon-Fano code (see Section 4.2) that is defined for a sequence $\boldsymbol{x} \in \boldsymbol{X}^{(k)}$ (4.4.4). We will see in the upcoming subsection were we will discuss the uniqueness requirements of the Arithmetic Code that this length is suitable for a tag like (4.4.14).

Infinite Precision Arithmetic Decoder :

In order to correctly decode the representative $\bar{T}_X(\boldsymbol{x})$ it is crucial for the decoder to mimic the

encoding procedure and scale in the same way the sub-intervals $[low, high)$. It is also required to scale the representative at every iteration, using the inverse procedure of the encoding. At the encoder the representative was getting smaller and smaller in every iteration of the algorithm, but at the decoder the representative gets larger and larger in every iteration of the algorithm. The scaling method of the representative in the decoder side will be the inverse scaling method that the encoder uses. Therefore, at the first iteration if $w = high - low$ then the scaled representative will be $s\_\bar{T}_x = \dfrac{\bar{T}_X(\boldsymbol{x}) - low}{w}$. At all times, the value $s\_\bar{T}_x$ being decoded, will obey the inequality $low \leq s\_\bar{T}_x < high$.

Below follows a pseudocode for the decoding procedure:

Algorithm 4.2 Infinite precision algorithm for the Arithmetic Decoder.

function ARITHMETICDECODERINF($\bar{T}_X(\boldsymbol{x})$, $\tilde{p}(\cdot)$, EOF):
    $low = 0$
    $high = 1$
    $decoded = [\,]$
    $i = 0$
    while *true* **do**
        $w = high - low$
        $s\_\bar{T}_X = \dfrac{\bar{T}_X(\boldsymbol{x}) - low}{w}$
        $u =$ decoded symbol such that $CP_{low}(u) \leq s\_\bar{T}_X < CP_{high}(u)$
        if $u == EOF$ **then**
            break
        end if
        $decoded[i] = u$
        $i = i + 1$
        $low = low + w \cdot CP_{low}(u)$
        $high = low + w \cdot CP_{high}(u)$
    end while
    return *decoded*
end function

4.4.2 Uniqueness and optimality of the Arithmetic Code

Unique Decodability

We have already seen that we can choose the midpoint (or the lower end) of the final subinterval $[low,\ high)$ to transmit or store the arithmetic code and we named it as the representative (or tag) and defined it in (4.4.14) and (4.4.15). But how are we sure that we have represented the final tag uniquely and efficiently?

We have already proposed that instead of storing (or transmitting) the binary value of the tag $\bar{T}_X(\boldsymbol{x})$ we would prefer a truncated version of the tag that is $\left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})}$, meaning that we can truncate the tag to $l(\boldsymbol{x}) = \left\lceil log\left(\frac{1}{\tilde{p}(\boldsymbol{x})}\right)\right\rceil + 1$ bits. Assuming the midpoint tag in (4.4.14) we must show that the truncated version of the code is unique, hence the code

$$\left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})} \in \left[\underbrace{CP_{high}(\boldsymbol{x}-1)}_{CP_{low}(\boldsymbol{x})},\ CP_{high}(\boldsymbol{x})\right) \tag{4.4.16}$$

that is the final sub-interval $[low,\ high)$. In order to be strict with our definitions we can say that if $\boldsymbol{x}$ is the sequence $x_1 \cdots x_{k+1}$, then $\boldsymbol{x}-1$ is the sequence $x_1 \cdots x_k$ ((4.4.2),(4.4.3),(4.4.10),(4.4.11), (4.4.12),(4.4.13)). At first it is true that:

$$\left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})} \leq \bar{T}_X(\boldsymbol{x}) < CP_{high}(\boldsymbol{x}) \tag{4.4.17}$$

Secondly, it is true that $2^{-l(\boldsymbol{x})} = 2^{-A} \leq 2^{-B} = \frac{1}{2}\tilde{p}(\boldsymbol{x}) \iff$

$$2^{-l(\boldsymbol{x})} \leq \frac{1}{2}\tilde{p}(\boldsymbol{x}) \tag{4.4.18}$$

where $A = \left\lceil log\left(\frac{1}{\tilde{p}(\boldsymbol{x})}\right)\right\rceil + 1$ and $B = log\left(\frac{1}{\tilde{p}(\boldsymbol{x})}\right) + 1$. From (4.4.14) and (4.4.18) it is true that:

$$\bar{T}_X(\boldsymbol{x}) - CP_{low}(\boldsymbol{x}) = \frac{1}{2}\tilde{p}(\boldsymbol{x}) \geq 2^{-l(\boldsymbol{x})} \tag{4.4.19}$$

It is also true that for binary fractions, if we want a binary splitted interval, this will be contained in $\left[CP_{low}(\boldsymbol{x}), CP_{high}(\boldsymbol{x})\right)$ and it has a length of $\frac{1}{2^{l(\boldsymbol{x})}} = 2^{-l(\boldsymbol{x})}$. $\left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})}$ is a representation value inside the final interval and $\bar{T}_X(\boldsymbol{x})$ too. So it is true that:

$$\bar{T}_X(\boldsymbol{x}) - \left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})} < 2^{-l(\boldsymbol{x})} \tag{4.4.20}$$

From (4.4.20), (4.4.19) and (4.4.14) it is also true that:

$$\underbrace{CP_{low}(\boldsymbol{x}) + \frac{1}{2}\tilde{p}(\boldsymbol{x})}_{\bar{T}_X(\boldsymbol{x})} - \left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})} < 2^{-l(\boldsymbol{x})} \leq \frac{1}{2}\tilde{p}(\boldsymbol{x}) \iff$$

$$\left\lfloor \bar{T}_X(\boldsymbol{x}) \right\rfloor_{l(\boldsymbol{x})} \geq CP_{low}(\boldsymbol{x}) \tag{4.4.21}$$

From (4.4.21) and (4.4.17) we can finally prove (4.4.16) that is required in order to prove the uniqueness of the code.

Prefix-free

Before we discuss why the Arithmetic Code is prefix-free, we shall begin with the proof of a theorem that is highly important.

Theorem 4.4.1

Given a number $i \in [0, 1)$ with an n-bit binary representation $b_1 b_2 \cdots b_n$ then for any other number j to have a binary representation with $b_1 b_2 \cdots b_n$ as the prefix, j has to lie in $\left[i, i + \frac{1}{2^n}\right)$.

Proof:

The number i has a fractional binary representation that is $i = .b_1 b_2 \cdots b_n$ therefore $i = b_1 2^{-1} + b_2 2^{-2} + \ldots + b_n 2^{-n}$ and if j has $b_1 b_2 \cdots b_n$ as a prefix then $j = b_1 2^{-1} + b_2 2^{-2} + \ldots + b_n 2^{-n} + b_{n+1} 2^{-(n+1)} + \ldots$. Thus, $j - i = b_{n+1} 2^{-(n+1)} + \ldots$ and of course $j - i \geq 0 \iff j \geq i$. To show that $j < i + \frac{1}{2^n}$ we observe that $j - i = b_{n+1} 2^{-(n+1)} + b_{n+2} 2^{-(n+2)} + \ldots \leq 2^{-(n+1)} + 2^{-(n+2)} + \ldots < 2^{-n} = \frac{1}{2^n}$. Also notice that n is the length of $b_1 b_2 \cdots b_n$.

If we can show that for any sequence $\boldsymbol{x}$, using Theorem 4.4.1 , that the interval $\left[\left\lfloor \bar{T}_X(\boldsymbol{x})\right\rfloor_{l(\boldsymbol{x})},\ \left\lfloor \bar{T}_X(\boldsymbol{x})\right\rfloor_{l(\boldsymbol{x})} + \frac{1}{2^{l(\boldsymbol{x})}}\right) \subset \left[CP_{low}(\boldsymbol{x}),\ CP_{high}(\boldsymbol{x})\right)$ then the code for one sequence is not possible to be the prefix for the code of another sequence. To prove this we do the following:
We have already shown that $\left\lfloor \bar{T}_X(\boldsymbol{x})\right\rfloor_{l(\boldsymbol{x})} \geq CP_{low}(\boldsymbol{x})$.
Therefore, we only require to prove that: $CP_{high}(\boldsymbol{x}) - \left\lfloor \bar{T}_X(\boldsymbol{x})\right\rfloor_{l(\boldsymbol{x})} > \frac{1}{2^{l(\boldsymbol{x})}}$. This is true because:

$$CP_{high}(\boldsymbol{x}) - \left\lfloor \bar{T}_X(\boldsymbol{x})\right\rfloor_{l(\boldsymbol{x})} > CP_{high}(\boldsymbol{x}) - \bar{T}_X(\boldsymbol{x}) = \frac{\tilde{p}(\boldsymbol{x})}{2} > \frac{1}{2^{l(\boldsymbol{x})}}$$

where the last inequality was derived from (4.4.19). Consequently, this code is prefix-free.

Optimality

Suppose that $\boldsymbol{x}$ is a random sequence of length m. We have already shown that using a code length of $l(\boldsymbol{x}) = \left\lceil log\left(\frac{1}{\tilde{p}(\boldsymbol{x})}\right)\right\rceil + 1$ bits, to encode the entire sequence we can have a prefix-free Arithmetic Code with the values of $\boldsymbol{x}$ being distinct. Using Definition 4.1.4 we can declare the expected codeword length for this type of code as
$\bar{L}_m = \sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x}) l(\boldsymbol{x}) = \sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x}) \left(\left\lceil log\left(\frac{1}{\tilde{p}(\boldsymbol{x})}\right)\right\rceil + 1\right) < \sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x}) \left(log\left(\frac{1}{\tilde{p}(\boldsymbol{x})}\right) + 1 + 1\right) =$
$= -\sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x}) \log\left(\tilde{p}(\boldsymbol{x})\right) + 2\sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x}) = H(X_1, \ldots, X_m) + 2$. From Theorem 4.1.2, the expected codeword length is always greater than the entropy so the bounds on $\bar{L}_m$ are:

$$H(X_1, \ldots, X_m) \leq \bar{L}_m < H(X_1, \ldots, X_m) + 2 \quad \text{(bits / } m \text{ source symbols)} \tag{4.4.22}$$

$$\frac{H(X_1, \ldots, X_m)}{m} \leq \bar{L} < \frac{H(X_1, \ldots, X_m)}{m} + \frac{2}{m} \quad \text{(bits / source symbol)} \tag{4.4.23}$$

But for iid sources (see (4.2.3)) , $H(X_1, \ldots, X_m) = mH(X)$ where X has the same probability distribution as $X_1 \ldots X_m$, hence:

$$H(X) \leq \bar{L} < H(X) + \frac{2}{m} \quad \text{(bits / source symbol)} \tag{4.4.24}$$

We see from (4.4.22) that the expected length of the arithmetic code is within 2 bits of the entropy for the entire message, therefore if we compress a file, then the size of the file is within 2 bits of the ideal compressed length. This result is very good. In addition from (4.4.24) we see a slightly worse result that Huffman block coding but at cost of an easier and more practical algorithm, that also has the benefit of requiring absolutely no codebook. Also, this type of code achieves greater compression than Huffman Coding and it is widely used in many state-of-the-art data compression techniques.

Notice :

As we have already commented under Algorithm 4.1, we know that for a range $[low,\ high)$ if we have low and $high$ sharing n bits at the beginning of a loop, any value between low and $high$ also shares those n bits. But if we scale low and $high$ at each step, how can we guarantee that every scale sits between low and $high$?
If we examine (4.4.9) to (4.4.13) we have an equation of the form:

$$new\_low = low + \underbrace{(high - low)}_{w} \cdot CP_{low}(x_i) = \underbrace{CP_{low}(x_i) \cdot high}_{t \cdot f(x_1)} + \underbrace{(1 - CP_{low}(x_i)) \cdot low}_{(1-t) \cdot f(x_2)}$$

A convex combination can be written as $t \cdot f(x_1) + (1-t) \cdot f(x_2)$ with $t \in [0,\ 1] \supset [0,\ 1)$. It is true that any convex combination on $f(x_1) \leq f(x_2)$ will lie in the interval $\big[f(x_1),\ f(x_2)\big] \supset \big[f(x_1),\ f(x_2)\big)$.

4.4.3 Finite precision Arithmetic Coding using integer representation

We have already discussed the infinite precision case for implementing Arithmetic Coding but this procedure is impractical for very large sequences because our software gives us access to libraries that contain data types that only have finite precision. In this subsection we are going to implement a practical algorithm for the Arithmetic Encoder and Decoder where we can finally implement the Arithmetic Coding scheme in realistic applications [37]. In order to do this, at first instead of considering the cumulative probability ranges for a specific symbol that give us floating point representations, we can consider their respective cumulative frequencies. By cumulative

frequencies we mean the cumulative times of appearance of every symbol that is in the input sequence $\boldsymbol{x} = \underbrace{x_1 \cdots x_k}_{T}$ where $x_1, \ldots, x_k \in \mathbb{N}$. If we define n_i as the number of times the symbol x_i occurs in the sequence $\boldsymbol{x}$ of length T, then the cumulative frequencies can be defined as:

$$CF_{high}(x_i) = \sum_{j=1}^{i} n_j \tag{4.4.25}$$

$$CF_{low}(x_i) = \sum_{j=1}^{i-1} n_j \tag{4.4.26}$$

We can now estimate the range $[low,\ high) = \big[CP_{low},\ CP_{high}\big)$ for a symbol x_i this time as:

$$low = CP_{low}(x_i) = \frac{CF_{low}(x_i)}{T} \tag{4.4.27}$$

$$high = CP_{high}(x_i) = \frac{CF_{high}(x_i)}{T} \tag{4.4.28}$$

But the way that a half open interval has to work in real numbers doesn't map on well to the integers because if we choose $low,\ high \in \mathbb{R}$ with $low < high$ there are always infinite numbers inside the interval $[low,\ high)$. Thus, in order to have an integer representation of the interval $[low,\ high)$ we can say the following:
Given a codeword length of m, we expand the initialization interval $[0,\ 1)$ to a range of 2^m binary words. Therefore the mapping is as follows:

$$0 \rightarrow \underbrace{00\ldots0}_{m \text{ times}} \tag{4.4.29}$$

$$1 \rightarrow \underbrace{11\ldots1}_{m\ times} \tag{4.4.30}$$

$$0.5 \rightarrow 1 \quad \& \quad \underbrace{00\ldots0}_{m-1\ \ times} \tag{4.4.31}$$

By & we mean concatenation. In addition we will define $w = high - low + 1$, thus the updated intervals for a new symbol x_i considering an input sequece $\boldsymbol{x} = x_1 \cdots x_M$, $\boldsymbol{x_i} = x_1 \cdots x_i$ and

$i \in \{1, \ldots, M\}$ can be calculated as:

$$low = CP_{low}(\boldsymbol{x}_i) = CP_{low}(\boldsymbol{x}_{i-1}) + \left\lfloor \frac{w \cdot CF_{low}(x_i)}{T} \right\rfloor \tag{4.4.32}$$

$$high = CP_{high}(\boldsymbol{x}_i) = CP_{low}(\boldsymbol{x}_{i-1}) + \left\lfloor \frac{w \cdot CF_{high}(x_i)}{T} \right\rfloor - 1 \tag{4.4.33}$$

Where, $CP_{low}(x_0) = 0$ (4.4.29) and $CP_{high}(x_0) = 2^m - 1$ (4.4.30) and T is the greatest cumulative frequency. The representation above will always give us a finite and countable interval .

If the interval $[low,\ high)$ is entirely confined to the lower half of the unit interval that is $[0,\ 0.5) = [00\ldots0,\ 10\ldots0) = [00\ldots0,\ 011\ldots1]$, then it is forever confined to that half of the unit interval. Furthermore, in this case the MSB of both *low* and *high* and obviously for any value in between will be 0. Hence, we can store 0 to the tag and perform E_1 mapping that is:

$$E_1 :\ [0,\ 0.5) \rightarrow [0,\ 1)\ ;\ E_1(x) = 2x \tag{4.4.34}$$

If the interval $[low,\ high)$ is entirely confined to the upper half of the unit interval that is $[0.5,\ 1) = [10\ldots0,\ 11\ldots1]$ then it is forever confined to that half of the unit interval and this time the MSB of both *low* and *high* will be 1, so we can store 1 to the tag and perform E_2 mapping that is:

$$E_2 :\ [0.5,\ 1) \rightarrow [0,\ 1)\ ;\ E_2(x) = 2(x - 0.5) \tag{4.4.35}$$

The mappings from $[low,\ high)$ to $[0,\ 1)$ are a major reason that our arithmetic is of finite precision. As soon as we perform these mappings, we store the MSB to the tag and we lose all information about it. The loss of information does not matter though, because as we said it is stored. We can observe that E_1 and E_2 mappings give us the same results as if we perform left shift operation to *low* and *high* but for *low* we have $low = (low << 1)\&0$ and for *high* we have $high = (high << 1)\&1$, i.e. we left shift *low* and *high* by 1 bit and then we concatenate 0 to *low* and 1 to *high*. The concatenation of 1 to high happens after we perform the mappings in order to ensure that always $low < high$. We use the term concatenation because we assume that when we shift left by one bit

we lose one bit too, that is the LSB. Otherwise we can replace concatenation with addition, if the left shifts always come up with a zero at the end. Now, we can continue with the next iteration of the algorithm. This process is called incremental encoding, and the reason is that we don't wait to see the entire sequence in order to generate the bits of the tag.

Critical conditions

Underflow :

We are not done yet, because we have a critical condition that is called underflow ([38]) and this can happen when $[low,\ high)$ is not entirely confined to either half of the unit interval while low and $high$ are approaching the midpoint of the unit interval. It is true that an underflow can occur after the arithmetic operations of scaling (4.4.32),(4.4.33) and low and $high$ start getting so small that our data types can no longer represent it properly. If this problem persists for long enough we will run out of bits and encoding will fail as we are losing precision because some bits are solidifying. If we are unlucky then, we could end up converging to 0.5 that is the midpoint of the unit interval. In other words this is the case where the tag interval in fact straddles the midpoint of the unit interval, so in order to make a mapping we need to check if $low \geq 0.25$ and $high < 0.75$ and if $[low,\ high)$ is not entirely confined to either half of the unit interval. Then the mapping is:

$$E_3 :\ [0.25,\ 0.75) \rightarrow [0,\ 1)\ ;\ E_3(x) = 2(x - 0.25) \tag{4.4.36}$$

E_3 mapping is applied when in fact $low \geq .010\ldots0$ and $high < .110\ldots0$ and for integer representation $low \geq 010\ldots0$ and $high < 110\ldots0 \iff high \leq 101\ldots1$. We can observe that we have two opposite most significant bits (MSBs) and second most significant bits for low and high. We need this type of mapping because it will help us not to lose precision as the algorithm proceeds. In order to remember how many times this condition has occured, what we need to do is use a variable $Scale3$ that is actually a counter and is incremented by one every time an underflow happens. Also this mapping is identical with saying $low = (low << 1)\&0$ and $high = (high << 1)\&1$ as in E_1 or

in E_2 mapping and then complement the new MSB of low and $high$ respectively. Instead of storing the new MSB immediately we wait for the next iteration to happen. If in the next iteration the tag interval is entirely confined to the upper half of the unit interval, then an E_2 mapping occurs and we know that we have to store a 1 to the tag , or if the tag interval is entirely confined to the lower half of the unit interval, then E_1 mapping occurs and we have to store a 0. In order to be precise and to show how many times E_3 mapping was applied, by keeping in mind that for each scaling-up the encoder expects the tag interval to be closer to the midpoint of the unit interval and by knowing that $Scale3$ is incremented by "1" every time E_3 mapping occurs, when the tag interval is entirely confined to either half of the unit interval (upper or lower) we store the common MSB of low and $high$ and $Scale3$ complements of this MSB.

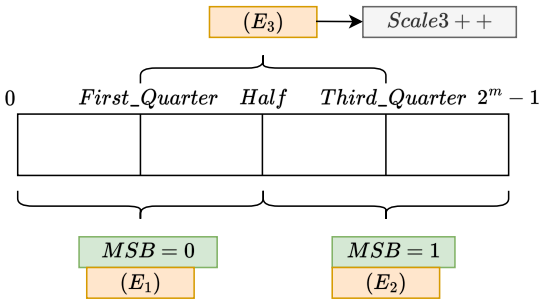


Figure 4.7: E_1, E_2 and E_3 mappings

Using E_3 mapping, the encoder can guarantee that, after the shifting operations either:

$$low < First\_Quarter < Half \leq high \tag{4.4.37}$$

or

$$low < Half < Third\_Quarter \leq high \tag{4.4.38}$$

The interval is smallest when, for equation (4.4.37) $high = Half$ and low is slightly lower than the $First\_Quarter$ (see [38]). Supposing that low is equal to the $First\_Quarter$ we have an even smaller range (by 1) so if m that is the length of code is large enough to accomodate uniquely the

set of values between 0 and the greatest cumulative frequency T in this range, then underflow will not happen. Also remember that we defined the interval range as $w = high + 1 - low$. So we have that $low = \frac{(2^m - 1 + 1)}{4} = 2^{m-2}$ and $high = \frac{2^m}{2} = 2^{m-1}$ then $high - low = 2^{m-1} - 2^{m-2} = 2^{m-2} \cdot (2 - 1) = 2^{m-2} = \frac{2^m}{4}$.
Using the same logic for equation (4.4.38) $high = Third\_Quarter$ and $low = Half$. In this case $low = \frac{(2^m - 1 + 1)}{2} = 2^{m-1}$ and $high = \frac{3 \cdot (2^m - 1 + 1)}{4} = 1.5 \cdot 2^{m-1}$ so $high - low = \frac{2^m}{4}$.
Therefore, as long as the integer range $[low,\ high)$ that is scaled-up by the cumulative frequencies, fits into a quarter of that range based on the code values ($0\ to\ 2^m - 1$) the underflow problem can not occur.
Hence, the requirement that $T \leq \frac{2^m}{4}$ will prevent underflow to happen and for this reason we can choose a word length of $m = \lceil \log_2(T) \rceil + 2$.

Overflow and Encoding Termination:

We may not show how we handle it on Algorithm 4.3 but what we must do is very straightforward. When an iteration is completed and we calculate the new values for low and $high$ that are the values after the shifts, if we have a number that needs $N + m > m$ bits, we simply store only the m most significant bits.

In order to be sure that the encoding procedure will be terminated, it must be true that the value of the tag, T_X, will be greater than the value of low. Therefore, we must find the m-bit representation of low and concatenate it next to the tag T_X (after the LSB). Also, if Scale3 > 0, we must store Scale3 times the MSB of low and the rest $m - 1$ bits of low next to the binary representation of T_X.

Finite precision algorithms

Here we will represent the finite precision case, using integer arithmetic coding, that we have already explained and see the pseudocodes of the finite precision arithmetic encoder and the finite precision arithmetic decoder. We will start with the encoder algorithm and then we will proceed to the decoder algorithm [37] [38].

Algorithm 4.3 Finite precision algorithm for the Arithmetic Encoder.

function ARITHMETICENCODER($\boldsymbol{x}$, COUNTS):
  $\boldsymbol{x} = x_1 \dots x_M$
  Compute $CF_{\text{low}}(u)$ and $CF_{\text{high}}(u)$ for each symbol u in $\boldsymbol{x}$
  $T = CF_{\text{high}}(\text{end})$
  $m = \lceil \log_2(T) \rceil + 2$
  low $= 0$
  high $= 2^m - 1$
  for $u \in \boldsymbol{x}$ **do**
    $w = \text{high} + 1 - \text{low}$
    $\text{high} = \text{high} + \left\lfloor \dfrac{w \cdot CF_{\text{high}}(u)}{T} \right\rfloor$
    $\text{low} = \text{low} + \left\lfloor \dfrac{w \cdot CF_{\text{low}}(u)}{T} \right\rfloor$
    while True **do**
      if MSB(low) == MSB(high) **then**
        store MSB to tag T_X
        low << 1 + 0
        high << 1 + 1
        while Scale3 > 0 **do**
          store $\overline{\text{MSB}}$ to tag T_X
          Scale3 = Scale3 − 1
        end while
      else if 2ndMSB(low) == 1 && 2ndMSB(high) == 0 **then**
        Scale3 = Scale3 + 1
        low << 1 + 0
        high << 1 + 1
        MSB(low) = $\overline{\text{MSB(low)}}$
        MSB(high) = $\overline{\text{MSB(high)}}$
      else
        break
      end if
    end while
  end for
  Terminate Encoding
  return T_X
end function

Algorithm 4.4 Finite precision algorithm for the Arithmetic Decoder.

function ARITHMETICDECODER(ENCODED, COUNTS, LEN):
  $T = CF_{\text{high}}(end)$
  $m = \lceil \log_2(T) \rceil + 2$
  low = 0
  high = $2^m - 1$
  $Scale3 = 0$
  tag ← first m bits of ENCODED
  decoded = [ ]
  $i = 1$
  $k = m$
  while $i \leq$ LEN **do**
    w = high + 1 − low
    scaled_symbol = $\left\lfloor \frac{T \cdot (\text{tag} - \text{low} + 1) - 1}{w} \right\rfloor$
    j index such that $CF_{\text{low}}(x_j) \leq$ scaled_symbol $< CF_{\text{high}}(x_j)$
    decoded$[i] = x_j$
    $i = i + 1$
    high = low + $\left\lfloor \frac{w \cdot CF_{\text{high}}(x_j)}{T} \right\rfloor - 1$
    low = low + $\left\lfloor \frac{w \cdot CF_{\text{low}}(x_j)}{T} \right\rfloor$
    while $E_1 || E_2 || E_3$ **do**
      if $k ==$ length(ENCODED) **then**
        break
      end if
      k=k+1
      if MSB(low) == MSB(high) **then**
        low << 1 + 0
        high << 1 + 1
        tag << 1 + ENCODED(k)
      else if 2ndMSB(low) == 1 & 2ndMSB(high) == 0 **then**
        low << 1 + 0
        high << 1 + 1
        tag << 1 + ENCODED(k)
        MSB(low) = $\overline{\text{MSB(low)}}$
        MSB(high) = $\overline{\text{MSB(high)}}$
        MSB(tag) = $\overline{\text{MSB(tag)}}$
      end if
    end while
  end while
  return decoded
end function

4.5 Golomb Codes

Golomb Codes, were first developed by Solomon Wolf Golomb ([40]) and they are suitable variable-length binary codes for encoding lengths of runs of non-negative integers. The basic idea, is to encode the run lengths between successive unfavorable events, by keeping in mind that the probability of a run length of m is geometrically distributed and it can be expressed as $p^m(1-p)$ for $m \in \mathbb{N}$. We observe that the probability table can be infinite so we can't use Huffman coding for this type of source, because it will have the consequence of generating a codebook with infinite entries.

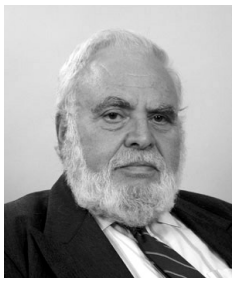


Figure 4.8: Solomon Wolf Golomb

At first [41], let's define an index function $f : \mathbb{N} \to \mathbb{N}$ that dissects the non-negative integers of a discrete source into indexed sets $S_0, S_1, \ldots$ where S_q is the finite set of integers that map to index q, and it can be written as:

$$S_q = \{n : f(n) = q\} \tag{4.5.1}$$

The length of the set S_q is $|S_q| = m \in \mathbb{N}^+$, therefore it is fixed $\forall q \in \mathcal{X} = \{0, 1, \ldots\}$. The code for the non-negative integer n will be: $C(n) = q\&r$ where r is the rank of n among all the integers of the set S_q and '&' means concatenation. The rank, can be written as:

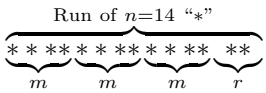
$$r = \left|\{j : \ j \in S_q, \ , j < n\}\right|, \quad r = 0, 1, \ldots |S_q| - 1 \tag{4.5.2}$$

The Golomb code of a non-negative integer $n = qm + r \iff n = \left\lfloor \frac{n}{m} \right\rfloor m + r$ can be defined as $G_c^{(m)}(n) = UnaryCode(q)\&BinaryCode(r)$, where $UnaryCode(q)$ is the quotient encoding using unary code and $BinaryCode(r)$ is the remainder that is a fixed length binary code. As we have already stated, m is the size of the set S_q. Golomb code corresponds to the case where $f(n) = q = \left\lfloor \frac{n}{m} \right\rfloor$ for $m \in \mathbb{N}^+$.

Figure 4.9: Golomb code $G_c^{(m)}(n) = UnaryCode(q)\&BinaryCode(r)$.

Example 4.5.1

In this example we will see the case where $n = 14$ (run length of 14) and we need to find $G_c^{(m)}(n) = UnaryCode(q)\&BinaryCode(r)$ giving $m = 4$. Then we have $n = qm + r \iff 14 = q \cdot 4 + r$ and this gives $q = 3$ and $r = 2$. For this reason $UnaryCode(q) = 1110$ and $BinaryCode(r) = 10$ so $G_c^{(4)}(14) = 111010$. Another way to see how we derive this code is to write n ones in a row and see how runs of m occur. When there is no more space for a run of m to occur then the rest of the run-length of $n = 14$ will be represented as a binary number of $\lceil \log_2(m) \rceil$ bits. Another way to see this ([42]) is to say:



and for every successful run of m we store 1 in the code, thus for this example we start by storing "111" to the code. When it is impossible for a new run of m to fit inside the rest of run of n we store a separator that is "0" to the code. At last, we write in binary how many remaining binary digits do we have in our run. In this case we store "10". Therefore the code will be "111010" that is equal to $G_c^{(4)}(14)$ that we have already found.

We can now understand that the construction of Golomb codes is based on the assumption that

the smaller integers of an infinite alphabet are more probable to occur than the bigger ones.

4.5.1 Uniqueness and optimality of the Golomb code

Prefix Free

Golomb Codes are prefix-free codes. This is based on the assumption that the index function $f(n)$ is surjective, meaning that $\forall q$ there is an n such that $f(n) = q$. In addition, because $n = q \cdot m + r$ and m is fixed, stating that q is determined $\forall n \in \mathcal{X}$ then, under these conditions, r has a different value corresponding to each index q. Therefore, Golomb codes are efficient in the sense that they satisfy Kraft-inequality with equality (Definition 4.1.2). Consequently, Golomb codes are prefix-free and it is impossible to add a codeword without violating the prefix-free condition. Golomb codes are also uniquely decodable for this reason (see Figure 4.1).

Optimality

Consider a reduced alphabet $\mathcal{X} = \{0, 1, \ldots, m + M\}$, where $M, m \in \mathbb{N}^+$ and the probability of a run of m ones is defined as $P(m) = (1 - p) \cdot p^m$. We are going to prove, based on [43], that Golomb's code is optimal for p satisfying

$$p^m + p^{m+1} \leq 1 < p^m + p^{m-1} \tag{4.5.3}$$

where $p \in (0,\ 1)$. We can see that $\forall p \in (0,\ 1)$ there is a unique m satisfying (4.5.3).

Consider that m is fixed and is determined by (4.5.3). For this p and m we define a probability mass function that is $P_M(n)$, where we have an M-reduced source that takes values from the reduced alphabet $\mathcal{X}$ which contains $m + M + 1$ symbols.

$$P_M(n) = \begin{cases} (1-p) \cdot p^n & ,\ 0 \leq n \leq M \\ \dfrac{(1-p) \cdot p^n}{1 - p^m} & ,\ M < n \leq M + m \end{cases} \tag{4.5.4}$$

Notice that this is an actual probability mass function because:

$$\begin{aligned}
\sum_{n=0}^{M+m} P_M(n) &= \sum_{n=0}^{M}(1-p)\cdot p^n + \sum_{n=M+1}^{M+m}\frac{(1-p)\cdot p^n}{1-p^m} \\
&= (1-p)\sum_{n=0}^{M} p^n + \frac{(1-p)}{1-p^m}\sum_{n=M+1}^{M+m} p^n \\
&= \frac{(1-p)}{(1-p)}\cdot(p^0 - p^{M+1}) + \frac{(1-p)}{(1-p^m)(1-p)}\cdot(p^{M+1} - p^{M+m+1}) \\
&= 1 - p^{M+1} + \frac{(1-p^m)}{(1-p^m)}\cdot p^{M+1} = 1 - p^{M+1} + p^{M+1} \\
&= 1
\end{aligned}$$

In addition, each of the final m probabilities, as we are going to see are the accumulated probabilities $\forall n > M$ in the probability mass function of the source. Also they are in the same equivalence class modulo m. Notice that, if $r = n$:

$$\begin{aligned}
\sum_{q=0}^{\infty}(1-p)\cdot p^{qm+r} &= \sum_{q=0}^{\infty}(1-p)\cdot(\underbrace{p^m}_{=a<1})^q\cdot p^r \\
&= (1-p)\cdot p^r\cdot\sum_{q=0}^{\infty} a^q \\
&= (1-p)\cdot p^r\cdot\frac{1}{1-a} \\
&= \frac{(1-p)\cdot p^n}{1-p^m}
\end{aligned} \tag{4.5.5}$$

where in the last equality we used the fact that $n = r$ and $p^m = a$.

For the next step, we will use binary Huffman coding on the M-reduced source, but we need to find a way first to sort its probabilities.

If we multiply the left side of (4.5.3) by p^M we have:

$$p^{M+m} + p^{M+m+1} \leq p^M \tag{4.5.6}$$

If we multiply the right side of (4.5.3) by p^M we have:

$$p^{M+m} + p^{M+m-1} > p^M \tag{4.5.7}$$

Now we can say using (4.5.6) that:

$$p^{M+m} + p^{M+m+1} \leq p^M \overset{\div p}{\iff} p^{M+m-1} + p^{M+m} \leq p^{M-1} \iff p^{M+m} \leq \underbrace{p^{M-1} - p^{M+m-1}}_{p^{M-1}(1-p^m)} \overset{\div(1-p^m)}{\iff}$$

$$\frac{p^{M+m}}{1-p^m} \leq p^{M-1} \overset{\cdot(1-p)}{\iff}$$

$$\frac{(1-p)\cdot p^{M+m}}{1-p^m} \leq (1-p)\cdot p^{M-1} \tag{4.5.8}$$

Also, by using (4.5.7) we have:

$$p^{M+m-1} > \underbrace{p^M - p^{M+m}}_{p^M(1-p^m)} \iff \frac{p^{M+m-1}}{1-p^m} > p^M \overset{\cdot(1-p)}{\iff}$$

$$\frac{(1-p)\cdot p^{M+m-1}}{1-p^m} > (1-p)\cdot p^M \tag{4.5.9}$$

We have just proved that the probability of the $(M+m)th$ symbol to appear is no more probable than the probability of the $(M-1)th$ symbol (4.5.8). Also the probability of the $(M+m-1)th$ symbol to appear is more probable than the $(M-1)th$ (4.5.9). Furthermore, notice that the probabilities of the first $M+1$ values of the source are monotonically decreasing, and this is true for the probabilities of the last m values too. Thus, for the first step of the algorithm for Huffman coding we have:

$$\frac{(1-p)\cdot p^{M+m}}{1-p^m} \leq (1-p)\cdot p^{M-1} < \frac{(1-p)\cdot p^{M+m-1}}{1-p^m} \tag{4.5.10}$$

and

$$(1-p)\cdot p^M < (1-p)\cdot p^{M-1} \tag{4.5.11}$$

so we will accumulate the probabilities of the Mth and the $(M+m)th$ symbol. After we accumulate these probabilites we will proceed considering an $M-1$ reduced source which will of course contain

the accumulation of the Mth and the $(M+m)th$ symbol, with a new probability equal to A, into one symbol. It is also important to note that:

$$A = \frac{(1-p)\cdot p^{M+m}}{(1-p^m)} + (1-p)\cdot p^M = \frac{(1-p)\cdot p^M}{1-p^m}$$

This is depicted in Figure 4.10. Now, for the $M-1$ reduced source, we will use the equations

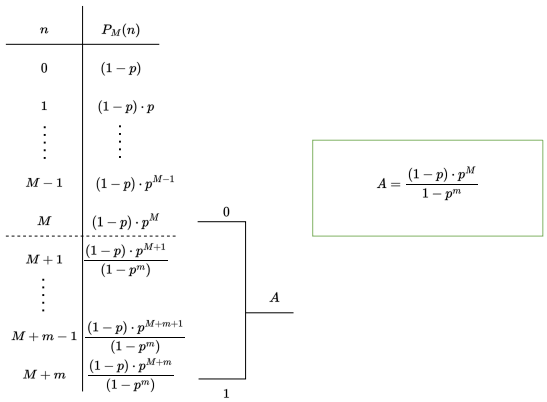


Figure 4.10: First step of Huffman Coding the M-reduced source.

(4.5.10) and (4.5.11) but this time for $M = M-1$ so:

$$\frac{(1-p)\cdot p^{M+m-1}}{1-p^m} \leq (1-p)\cdot p^{M-2} < \frac{(1-p)\cdot p^{M+m-2}}{1-p^m}$$

and

$$(1-p)\cdot p^{M-1} < (1-p)\cdot p^{M-2}$$

so we will accumulate the $(M-1)th$ and the $(M+m-1)th$ symbol, with a new probability equal to B, into one symbol, where:

$$B = \frac{(1-p)\cdot p^{M+m-1}}{(1-p^m)} + (1-p)\cdot p^{M-1} = \frac{(1-p)\cdot p^{M-1}}{1-p^m}$$

This process is depicted in Figure 4.11

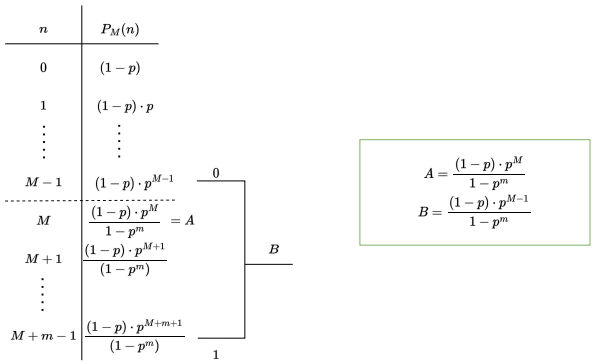


Figure 4.11: Second step of Huffman Coding the M-reduced source $\longleftrightarrow$ Huffman Coding the $M-1$ reduced source.

This process is repeated up to and including $M = 0$, where after accumulating the least likely symbols, we are going to have the $M = -1$ reduced source with:

$$P_{-1}(n) = \frac{(1-p)\cdot p^n}{(1-p)^m}, \quad 0 \leq n \leq m-1 \tag{4.5.12}$$

In figure 4.12 we see the whole process (all the steps) of the Huffman Code, using the source with probabilities as defined in (4.5.4).

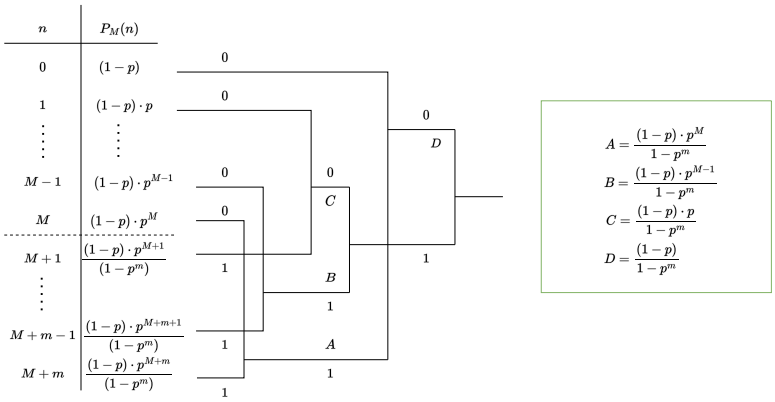


Figure 4.12: Huffman Coding process for the whole source

Also in Figure 4.13 we show all the possible ways our last step can be using the probabilities in (4.5.12).

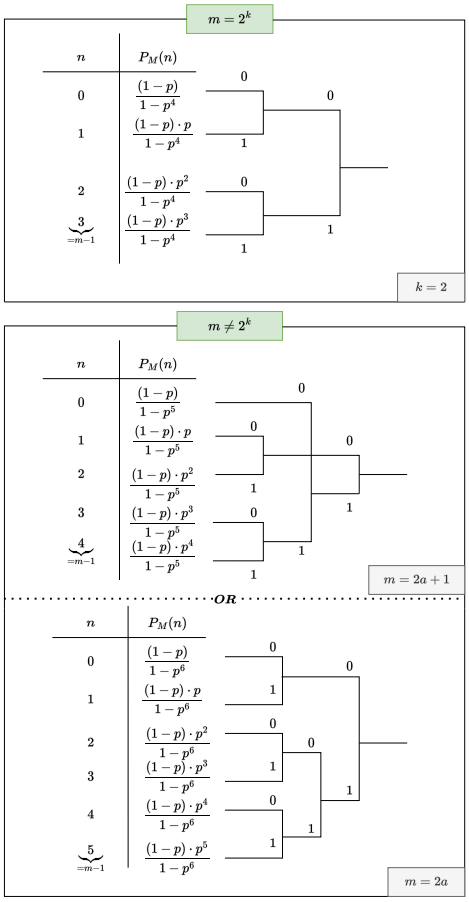


Figure 4.13: Huffman Coding process for the $M = -1$ reduced source, for $m = 4 = 2^2$, $m = 5 \neq 2^k$, $m = 6 \neq 2^k$.

For the $M = -1$ reduced source, notice that the sum of the two least likely symbols has a probability that is greater than the one of the most likely symbol. This is true because:

$$\begin{aligned} &\frac{(1-p)\cdot p^{m-1}}{1-p^m} + \frac{(1-p)\cdot p^{m-2}}{1-p^m} > \frac{(1-p)}{1-p^m} \\ \iff\ & p^{m-1} + p^{m-2} > 1 \iff p^m + p^{m-1} > p^m \end{aligned} \tag{4.5.13}$$

and from the right side of (4.5.3) we can easily prove that (4.5.13) is true. Also don't forget that the probabilities of the $M = -1$ reduced source are monotonically decreasing.

As we can see in Figure 4.13, we have two distinct cases for m:

- If $m = 2^k$, the optimal codewords have the same length.
- If $m \neq 2^k$, the optimal codewords differ by 1.

These codewords are for the rank r (see 4.5.2) and concern the remainder of the equation $n = qm+r$. To understand this, observe their probabilities in Figure 4.13 and then see (4.5.5). Working on a proof of concept we can see that the optimal encoding for (4.5.12) is to use codewords of length $\lfloor \log_2(m) \rfloor$ for $r < 2^{\lfloor \log_2 m+1 \rfloor} - m$ and codewords of length $\lfloor \log_2 m \rfloor + 1$ otherwise.

In this type of code, if one observes Figure 4.12 and replaces M and m with the some fixed values it will be clear that for $n \leq M$ each symbol will be encoded into the optimal codeword representation for $r = n \mathrm{mod} m$, concatenated with the unary code for $q = \left\lfloor \frac{n}{m} \right\rfloor$. Hence, if we want to show that for an infinite alphabet then we say that $M \to \infty$. This can be left as an exercise to the reader. Also, since the resulting code is a concatenation of two prefix-free codes, we can easily reverse the order of the codes, meaning that we are going to have and encoded codeword for n that will be the unary code for $q = \left\lfloor \frac{n}{m} \right\rfloor$ concatenated to its right with the binary code for $r = n \mathrm{mod} m$.

Before proving that Golomb's Code is optimal, we will start by some definitions. So, let:

- $\bar{L}$ be the infimum of the expected codeword length over all uniquely decodable codes for source with probabilities $P(n) = (1-p) \cdot p^n$.
- $\overline{L_G}$ be the expected codeword length for code $\tilde{G}_c^{(m)}$ which maps every integer n into a codeword that is the concatenation of $r = n \mathrm{mod} m$ with $q = \left\lfloor \frac{n}{m} \right\rfloor$ as $BinaryCode(r)\&UnaryCode(q)$.
- $\overline{L_M}$ be the expected codeword length for the optimal code for an M-reduced source.

At first we notice that given any uniquely decodable code for the source with probabilities $P(n) = (1-p) \cdot p^n$, where we have an infinite alphabet, its infimum expected codeword length will exceed $\overline{L_M}$ that it is for a finite alphabet with $M + m$ codewords. This is because if we have an optimum uniquely decodable code for the source $P(n)$ we can derive a code for the M-reduced source by using exactly the same codewords for $n \leq M$ and the shortest codewords remaining for

$M+1 \leq n \leq M+m$. Therefore, $\overline{L_M} \leq \bar{L}$. Also, given that$\tilde{G}_c^{(m)}$ is just a type of a uniquely decodable code we could use for $P(n) = (1-p) \cdot p^n$ it is true that $\bar{L} \leq \overline{L_G}$. So until now $\overline{L_M} \leq \bar{L} \leq \overline{L_G}$. Observe that $\overline{L_M}$ is increasing with M up to a point where $\lim_{M\to\infty} \overline{L_M} = \overline{L_G}$ In addition, because for $0 \leq n \leq M$ the M-reduced source follows a geometric distribution equal to $P(n) = (1-p) \cdot p^n$ then $\lim_{M\to\infty} \overline{L_M} = \bar{L}$. This means, that $\overline{L_G} = \bar{L}$, so the code $\tilde{G}_c^{(m)}$ is optimal and because it is a concatenation of two different prefix-free codes we can reverse the order and by this we can see that we have a Golomb Code $G_c^{(m)}$ that is optimal too, because the codeword lengths will not change.

All this process to prove optimality for the Golomb Code can be studied with less details [43] where the authors proved that Golomb Code is optimal not only for $p^m = \dfrac{1}{2}$ as Golomb stated in his original paper [40], but for a more general case where $p^m + p^{m+1} \leq 1 < p^m + p^{m-1}$. At last, we will present and example so the reader can understand this subsection better.

Example 4.5.2

In this example we will encode an M-reduced source where $M = m = 3$ using the probabilities in (4.5.4). From what we have already explained what we want is to see the Huffman binary tree that contains all the codewords, as long as the $M = -1$ reduced source tree, just to have different perspectives and understand the code better. So the Huffman tree will be:

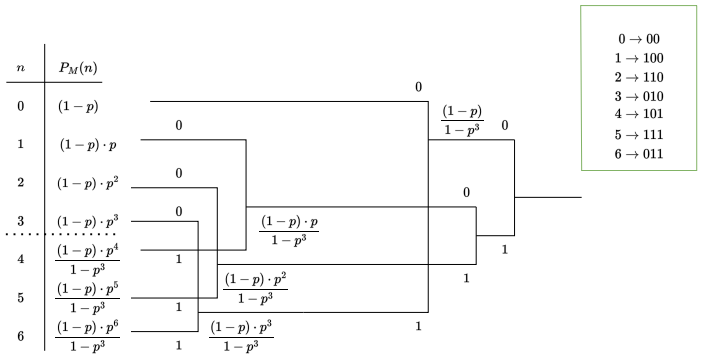


Figure 4.14: Huffman Coding process for the whole source, supposing that $m = 3$.

In addition for the $M = -1$ reduced source we will eventually have:

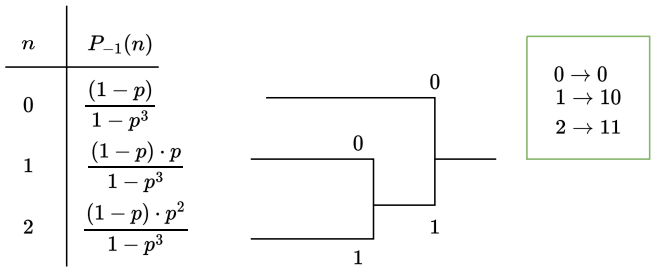


Figure 4.15: Huffman Coding process for the $M = -1$ reduced source, supposing that $m = 3$.

Notice that $r = n \text{mod} m$ and $0\text{mod}3 = 0$, $1\text{mod}3 = 1$, $2\text{mod}3 = 2$, $3\text{mod}3 = 0$, $4\text{mod}3 = 1$, $5\text{mod}3 = 2$, $6\text{mod}3 = 0$. Therefore $r \in \{0, 1, 2\}$ and using the probabilities $\frac{(1-p)\cdot p^n}{1-p^m}$ where $n = r$ we will have the binary codewords for the remainders that are represented in Figure 4.15. This is an optimal encoding as we use codewords of size $\lfloor \log_2(3) \rfloor = 1$ for $n < 2^{\lfloor \log_2(3)+1 \rfloor} - 3 = 1 \implies n = 0$ and codewords of size $\lfloor \log_2(3) \rfloor + 1 = 2$ otherwise for representing in binary $n = n + 2^{\lfloor \log_2(m)+1 \rfloor} - m$. So we have some binary representations for the remainder. From Figure 4.14 we can see that the codewords contain these binary representations for remainders concatenated with the unary code for the quotient, claiming that $n = qm + r$. This happens though only untill $n = M = 3$, and in this case the symbols follow a geometric distribution.

If $M \to \infty$ then we would have some optimal representations for the codewords of an infinite alphabet. We should not forget also to reverse the order of the code when $n \leq M$ in order to derive a Golomb Code.

Golomb Encoder

Given the integers n and m, and $A = \lfloor \log_2(m) + 1 \rfloor$, encode $n = q \cdot m + r \iff n = \left\lfloor \frac{n}{m} \right\rfloor + r$. For q find the $Unary\_Code(q)$ and for r:

- if $r < 2^A - m \to Binary\_Code(r)$ has length of $\lfloor \log_2(m) \rfloor$-bits
- if $r \geq 2^A - m \to Binary\_Code(r)$ has length of $\lfloor \log_2(m) \rfloor + 1$-bits, where $r = r + 2^A - m$.

Algorithm 4.5 Golomb Encoder

Given n, m such that $n = q \cdot m + r$ and $A = \lfloor \log_2(m) + 1 \rfloor$
$q = \left\lfloor \frac{n}{m} \right\rfloor$
$r = n - q \cdot m$
Calculate $Unary\_Code(q)$
if $m = 2^k$ **then**
    Calculate $Binary\_Code(r)$ using $\log_2(m)$-bits
else
    if $r < 2^A - m$ **then**
        Calculate $Binary\_Code(r)$ using $\lfloor \log_2(m) \rfloor$-bits
    else
        $r = r + 2^A - m$
        Calculate $Binary\_Code(r)$ using $\lfloor \log_2(m) \rfloor + 1$-bits
    end if
end if

$G_c^{(m)}(n) = UnaryCode(q) \& BinaryCode(r)$

Golomb Decoder

To decode $G_c^{(m)}(n)$, we have to do the inverse process of the encoder.

Algorithm 4.6 Golomb Decoder

Given $G_c^{(m)}(n), m$ and $A = \lfloor \log_2(m) + 1 \rfloor$
Calculate q by counting the "1" bits in $G_c^{(m)}(n)$ until you find the first "0"
Ignore the first "0", read the next $\lfloor \log_2(m) \rfloor$ bits in $G_c^{(m)}(n)$, and store them in r'
if $r' < 2^A - m$ **then**
    $r = r'$
else
    Ignore the first "0", read the next $\lfloor \log_2(m) \rfloor + 1$ bits in $G_c^{(m)}(n)$, and store them in r'
    $r = r' - (2^A - m)$
end if
$n = q \cdot m + r$

- Notice that, if $m \neq 2^k$ a representation of $\lfloor \log_2 m \rfloor$ bits for the remainder is not always enough. That's the reason that we have defined a threshold, that is $2^A - m$ ($A = \lfloor \log_2(m) + 1 \rfloor$) and when the remainder, r, is not less than the threshold we have to write the binary representation of $r = r + 2^A - m$.

- Notice that we have m codewords in each group S_q. In addition, $q = \left\lfloor \frac{n}{m} \right\rfloor$ therefore the unary code of q will change only when $n = t \cdot m$, for $t = 0, 1, 2, \ldots$. Therefore, $\forall\ n = t \cdot m$ Golomb Code is prefix-free. Now we have to concetrate at the m codewords inside a group, hence we focus on the case that $t \cdot m \leq n < (t+1) \cdot m$. To represent r in an optimal way when we are in a group of m symbols we can use Huffman Coding. The Huffman tree, as we saw in Figure 4.13, will have codeword representations for r of $\log_2(m)$ bits, if $m = 2^k$. But if $m \neq 2^k$ the codewords will have $\lfloor \log_2(m) \rfloor$ bits for the binary representation of $r < 2^A - m$ and $\lfloor \log_2(m) + 1 \rfloor$ bits otherwise. In addition, they will differ by at most "1" bit. From Figure 4.15 we can also observe that for $r < 2^A - m$ we have a binary representation of r but for $r \geq 2^A - m$ we have binary representations of $r = r + 2^A - m$. This addition is done in order to have a truncated binary tree of a prefix-free code for r.

Symbol	Codeword	Bits	Codeword	Bits	Codeword	Bits	Codeword	Bits
n	$m = 1,\ k = 0$		$m = 2,\ k = 1$		$m = 3$		$m = 4,\ k = 2$	
0	0	1	00	2	00	2	000	3
1	10	2	01	2	010	3	001	3
2	110	3	100	3	011	3	010	3
3	1110	4	101	3	100	3	011	3
4	11110	5	1100	4	1010	4	1000	4
5	111110	6	1101	4	1011	4	1001	4
6	1111110	7	11100	5	1100	4	1010	4
7	11111110	8	11101	5	11010	5	1011	4
8	111111110	9	111100	6	11011	5	11000	5
9	1111111110	10	111101	6	11100	5	11001	5
10	11111111110	11	1111100	7	111010	6	11010	5
11	111111111110	12	1111101	7	111011	6	11011	5
12	1111111111110	13	11111100	8	111100	6	111000	6
13	11111111111110	14	11111101	8	1111010	7	111001	6
14	111111111111110	15	111111100	9	1111011	7	111010	6
15	1111111111111110	16	111111101	9	1111100	7	111011	6

Figure 4.16: Binary codewords for $G_c^{(1)}$, $G_c^{(2)}$, $G_c^{(3)}$ and $G_c^{(4)}$. [1]

[1] Figure from [44]

Golomb Codes and their binary trees :

To close the section we will also present the binary trees for the first four Golomb Codes from Figure 4.5.2.

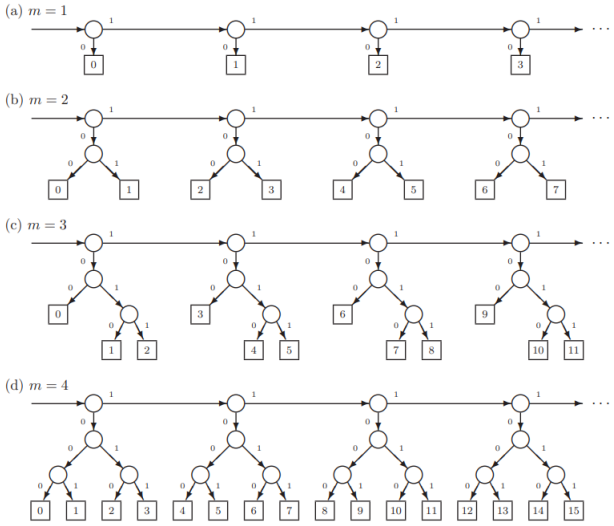


Figure 4.17: Binary trees for $G_c^{(1)}$, $G_c^{(2)}$, $G_c^{(3)}$ and $G_c^{(4)}$. [2]

- $G_c^{(m)}(n) = UnaryCode(q)\&BinaryCode(r)$.
- The source integer value $n = qm + r$.

4.5.2 Golomb-Rice Codes

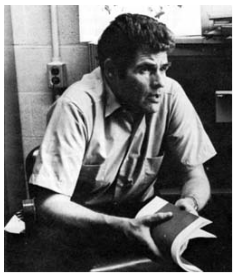
The Golomb-Rice Code is simply, a Golomb Code for $m = 2^k$. Robert Rice used a code that is exactly the same as $G_c^{(2^k)}$ in an adaptive scheme, as we can see in [45].

[2] Figure from [44]

4.6 Exponential-Golomb Codes

The Exponential-Golomb Codes, where first proposed by Jukka Teuhola [42], as a compression technique for integers following a geometric distribution. The main idea is to encode runs of 2^k bits, where k is incremented by one after one run is encoded. Some people also refer to them as Elias-Teuhola codes, because of their close resemblance to Elias gamma coding. In this approach, Peter Elias a code that augments the natural binary representation of an integer using a length indicator. The reader can study [46] for further information, as for this thesis we will not use Elias gamma coding

.



(a) Peter Elias

(b) Jukka Teuhola

Figure 4.18

Also the first idea for Arithmetic Coding (4.4), is the Shannon-Fano-Elias code where Peter Elias contributed to this research too [32].

In order though to continue with the explanation of Exponential-Golomb Codes, we must first explain how they work. The basic principle of the Exponential-Golomb Code scheme is that when a run is encoded, we try to separate the successive sub-vectors of 2^k, 2^{k+1}, 2^{k+2},...("$*$"). When this does not succeed any more then the end of the run will be encountered with a zero bit ("0") and the rest of the run will simply be encoded into a binary number. This is a method with potential because it encodes both short and long runs efficiently, as the successive sub-vectors grow exponentially. Also, we will symbolize the Exponential-Golomb Code of parameter k for an integer

n as $EG_c^{(k)}(n)$.

In order to understand this concept better, a simple example will suffice:

Example 4.6.1

Encode the integer $n = 19$, with a parameter $k = 1$. So we are looking for $EG_c^{(1)}(19)$.

$$\overbrace{\underbrace{**}_{2^k}\underbrace{****}_{2^{k+1}}\underbrace{**********}_{2^{k+2}}\underbrace{******}_{r}}^{\text{Run of } n=19 \text{ “}*\text{”}}$$

For all the successful runs of $2^k, 2^{k+1}$ etc, we will store bit 1 to $EG_c^{(1)}(19)$. For the rest of the run (r) that does not fit at the final run of 2^{k+i} (in this example $i = 3$) we will store a separator, that is a zero (”0”) and then we will encode r in $k + i$ bits using a simple binary code. The binary code for r will be 0101 (we want to represent the decimal number 5 to $k + i = 4$-bits. Therefore, $EG_c^{(1)}(19) = 11100101$

Generally, if the length of the last sub-vector was 2^{k+i}, then $k + i + 1$ bits are needed to represent the rest of the run. Consider though, that this method is not very sensitive to the value of k. Unfortunately, there is not an exact optimization rule, because the sub-vectors aren't fixed as they are in the Golomb Code case that we encode successful runs of m (or sub-vectors of fixed length m).

To summarize, we can say that: $EG_c^{(k)}(n) = UnaryCode(q)\&\underbrace{BinaryCode(r)}_{k+q\ bits}$, where:

- $q = \left\lfloor \log_2(1 + \frac{n}{2^k}) \right\rfloor$
- $r = n - \sum_{j=0}^{q-1} 2^{j+k}$

So for Example 4.6.1 $q = \left\lfloor \log_2\left(1 + \frac{19}{2^1}\right) \right\rfloor = 3$ and $r = 19 - \sum_{j=0}^{2} 2^{j+k} = 19 - 2^1 + 2^2 + 2^3 = 5$. This means that $UnaryCode(q) = 1110$ and $BinaryCode(r) = 0101$. We have described the encoding procedure.

The decoding procedure is really simple. Because k is given, we can find the $UnaryCode(q)$ until we encounter a separator ("0"-bit). When we find q then the input number n that was encoded before is $n = r + \sum_{j=0}^{q-1} 2^{j+k}$.

A faster implementation is to say that $r = n - 2^k(2^q - 1)$.

To close the section we will also present the binary tree for the $k = 0$-order Exponential-Golomb Code, $EG_c^{(0)}(n)$ [44], and we will understand better that we call it "exponential" because the children nodes of the binary trees grow exponentially, as the integer n gets bigger. .

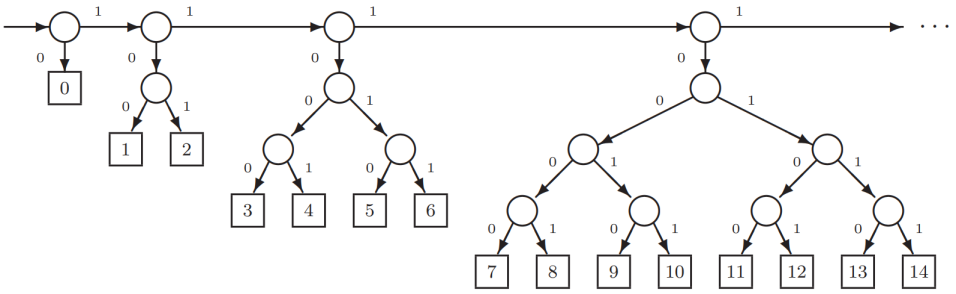


Figure 4.19: Binary tree for $EG_c^{(0)}(n)$. [3]

[3] Figure from [44]

Chapter 5

Lossless Audio Compression

In this chapter we will work on lossless audio compression and see how we can combine all the previous chapters in order to achieve this. Lossless audio files need more storage space in contrast with the lossy audio files. The good thing though, is that all the original information is preserved, hence we can achieve high quality audio, even though it is compressed, that is exactly the same as in the original waveform coded audio .wav file (the original PCM signal from Section 2.3). In the sections below we will work at most with IEEE 1857.2 audio standard and precisely with its lossless extension [7].

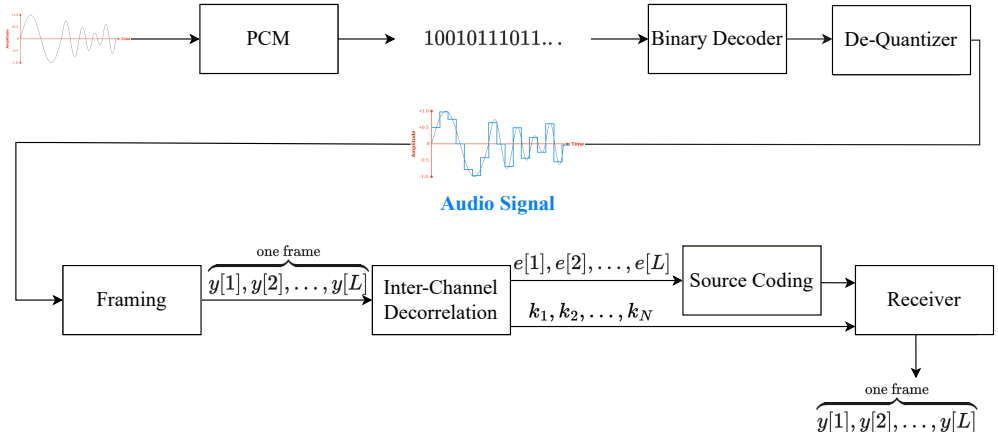


Figure 5.1: Lossless Audio Coding general scheme for a single audio frame.

5.1 The IEEE 1857.2 Lossless Audio Codec

In this section we will mainly follow [7], [47], [48],[49] and see how we implement a code that is based on IEEE 1857.2 lossless audio coding extension.

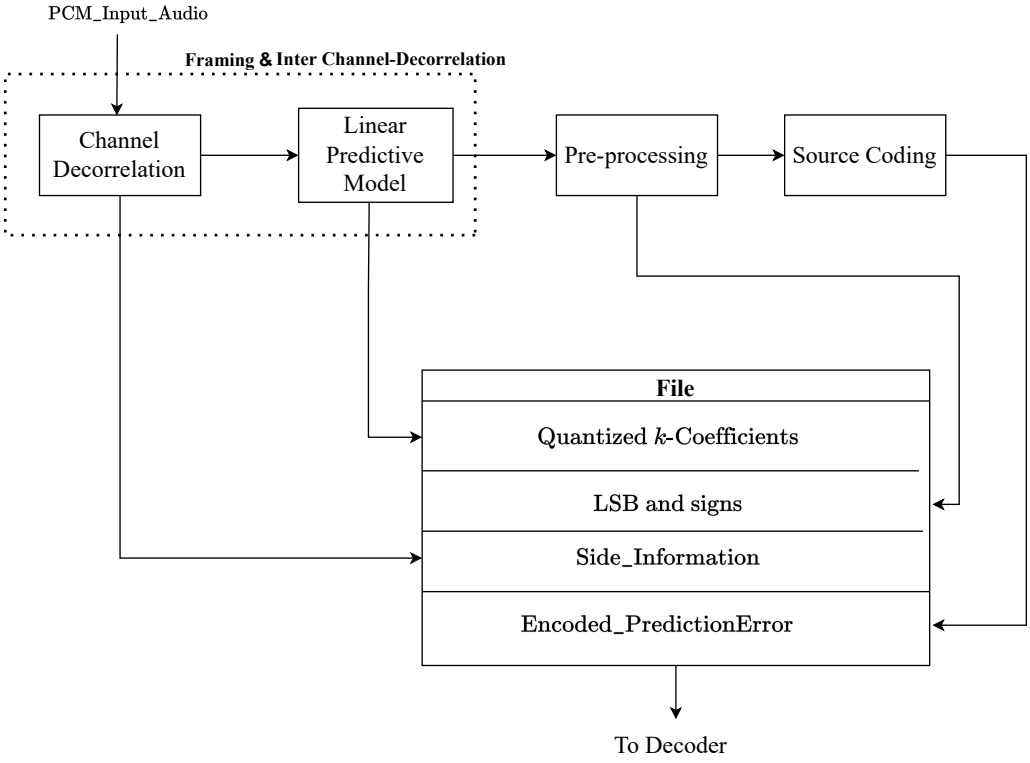


Figure 5.2: The IEEE 1857.2 encoder.

The **File** contains the bitstreams that we are going to store, so that the IEEE 1857.2 Decoder can retrieve the original information, that is the PCM_Input_Audio, which is an audio signal that is modulated using the PCM waveform coder that we explained in Section 2.3, that is a universal technique for the digitization of an analog audio signal , but keep in mind that it is uncompressed. So, let's dive deep into the codec in order to understand how compression is achieved.

5.1.1 Channel Decorrelation

This block is activated only when the input signal is a stereo signal, i.e when it has two channels. That means that when the signal is mono (one channel) we deactivate this block. So we can say that PCM_Input_Audio $= \left[X^{(L)} \;\; X^{(R)}\right]$. In IEEE 1857.2 lossless audio codec we can see that the Channel Decorrelation is performed as:

$$Mid = \frac{X^{(L)} + X^{(R)}}{2} \tag{5.1.1}$$

$$Side = X^{(L)} - X^{(R)} \tag{5.1.2}$$

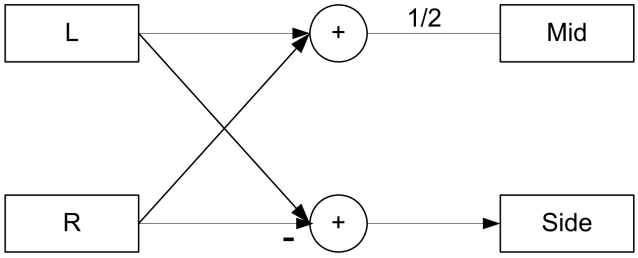


Figure 5.3: Diagram of Channel Decorrelation.

In Figure 5.3 we can see that the left and right channels, are converted to the Mid and Side pair. Mid will be processed independently and Side will be stored uncompressed.

5.1.2 Linear Predictive Model

The purpose of this stage combined with the Channel Decorrelation stage is to remove redundancy by decorrelating the left and right channel samples (for stereo input only) and then we use a modified linear predictive model for the signal. We will start by framing (or blocking) the input signal into frames (or blocks) in a way that each block is a WSS stochastic process (see Definition 2.1.6). In [5] we see that a lot of audio codecs use a 13 to 26 ms block duration. As we have already discussed in the comments of Section 2.1 a 23.2 ms block duration will suffice. If we choose a block

length of 1024 samples then we have a block duration equal to $\frac{1024}{44100} \cdot \frac{samples}{samples/second} \approx 23.2\ ms$. We also define a prediction order strictly less than the block length, because we apply an intra-frame linear prediction, i.e $N < L$.

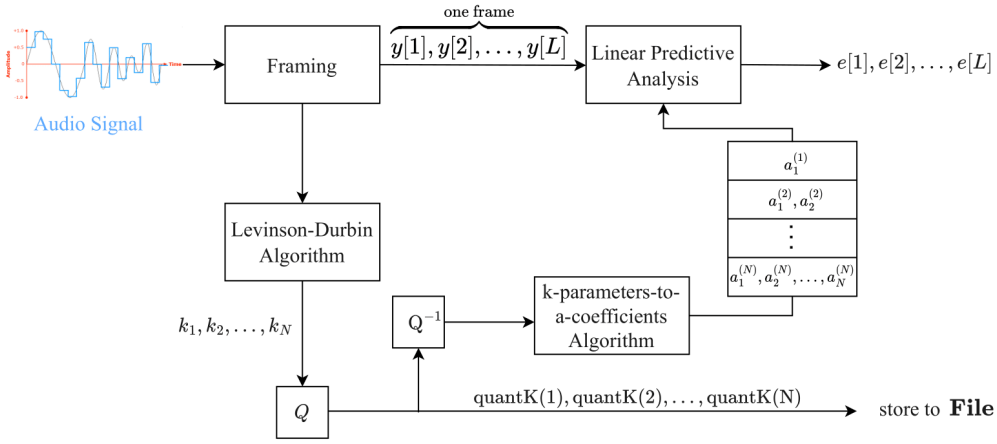


Figure 5.4: Diagram of Linear Predictive Modeling in IEEE 1857.2.

After the Framing step, that we have already explained, we will use the Levinson-Durbin Algorithm (see Algorithm 3.1), where we will obtain the PARCOR (PARtialCORrelation) coefficients that are the k-coefficients (or reflection coefficients) that we see in the Levinson-Durbin Algorithm. Afterwards, we will use a compound quantization technique that is already defined by the standard [7] and we will restrict the k-coefficients to the interval $[-64,\ 63]$ by locally quantizing them. The companding quantization for the k-coefficients is defined by (5.1.3) that we see below:

$$\text{quantK}[i] = \begin{cases} \left\lfloor 64 \left(\ln\left(\frac{2}{3} + \frac{5}{6}\sqrt{\frac{1+k_1}{2}} \right) \Big/ \ln\left(\frac{3}{2}\right) \right) \right\rfloor, & i = 1 \\ \left\lfloor 64 \left(\ln\left(\frac{2}{3} + \frac{5}{6}\sqrt{\frac{1-k_2}{2}} \right) \Big/ \ln\left(\frac{3}{2}\right) \right) \right\rfloor, & i = 2 \\ \lfloor 64 \cdot k_i \rfloor, & i = 3, \ldots, N \end{cases} \tag{5.1.3}$$

- N: linear prediction order.
- L: frame length.

Then the dequantization function for the quantized k-coefficients is defined as:

$$\text{dequantK}[i] = \begin{cases} \left(2 \cdot \left(\left(\exp\left(\frac{\text{quantK}(1)}{64 \cdot \ln\left(\frac{3}{2}\right)}\right) - \frac{2}{3}\right) \cdot \frac{6}{5}\right)^2\right) - 1, & i = 1 \\ \left(2 \cdot \left(\left(\exp\left(\frac{\text{quantK}(2)}{64 \cdot \ln\left(\frac{3}{2}\right)}\right) - \frac{2}{3}\right) \cdot \frac{6}{5}\right)^2\right) + 1, & i = 2 \\ \frac{\text{quantK}(i)}{64}, & i = 3, \ldots, N \end{cases} \tag{5.1.4}$$

The implementation of the k-parameters to a-coefficients algorithm has been defined in this thesis, using Algorithm 2.1, that was derived by implementing a digital FIR lattice filter. The k-parameters are actually the dequantized k-coefficients (dequantK()). So for these k-coefficients we derive their respective a-coefficients that we saw in Chapter 3, where we discussed Linear Prediction. Now that we have our a-coefficients as an output from the k-parameters-to-a-coefficients algorithm, we can now use Linear Prediction to find the prediction error $e[n]$ from (3.2.3). This means, using (3.2.2) and (3.2.1), for G=1 that:

$$e[n] = y[n] - \underbrace{(-\sum_{k=1}^{N} a_k y[n-k])}_{\hat{y}[n]} \iff e[n] = y[n] + \sum_{k=1}^{N} a_k y[n-k] \tag{5.1.5}$$

Remind that, we perform an intra-frame linear prediction so the prediction errors will not propagate

beyond frame boundaries. Another way that we can write this expression, is by using the IEEE 1857.2 standard implementation of linear prediction using the equation below:

$$\hat{y}(i) = \begin{cases} -y(i), & i = 1 \\ -\sum_{j=1}^{i} a_j^{(i)} y(i-j), & 2 \leq i \leq N-1 \\ -\sum_{j=1}^{N} a_j^{(i)} y(i-j), & N \leq i \leq L \end{cases} \tag{5.1.6}$$

In this case i is the time index for the samples inside the frame, $y(i)$ denotes the input sample i inside the frame and $\hat{y}(i)$ is the prediction for the sample i inside the frame, where the frame length is L. Having the prediction $\hat{y}(i)$ $\forall i \leq L$ we can calculate the corresponding prediction errors using equation (3.2.3). From the prediction errors (or prediction residues), as we can see in Figure 5.4, one can construct the output signal of the Linear predictor block. Also $\hat{y}$ is rounded.

5.1.3 Pre-Processing

The Pre-Processing block has the prediction errors $e[n]$ (or prediction residues) and the quantized k-coefficients (quantK) as its input (see Figure 5.1.3 and [48]). The task of this block is to normalize the large prediction residues at the beginning of each audio frame, because they increase the dynamic range of the prediction error in a considerable manner. This means that later, our source coder (that performs an encoding of the source) adopts a large alphabet size. We want to encounter this situation because it will increase the computational complexity of source coding, as long as the compression efficiency.

The number of samples to be normalized, is determined by $M = \min(N, 16)$ and each of these samples is downshifted by shift$[n]$ (the number of shifts for each sample) (see (5.1.7)).

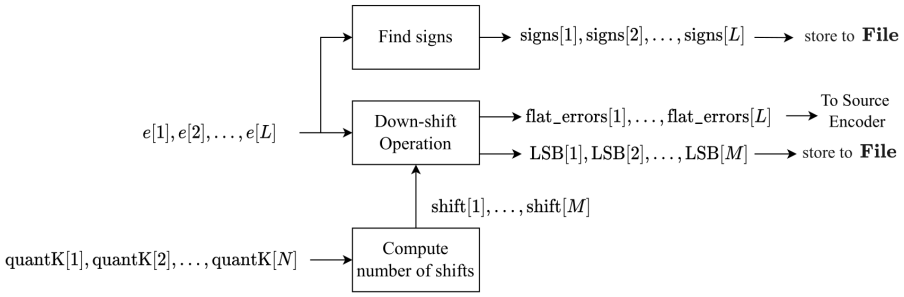


Figure 5.5: Diagram of the Pre-Processing block in IEEE 1857.2.

$$\text{shift}[n] = \begin{cases} \left\lfloor \frac{2^{12} + \sum_{k=1}^{n+1} \text{RAshift12}[\text{quantK}[n]]}{2^{13}} \right\rfloor, & n = 1, 2 \\ \left\lfloor \frac{2^{12} + \sum_{k=1}^{2} \text{RAshift12}[\text{quantK}[n]] + \sum_{k=3}^{n+1} \text{RAshift}[|\text{quantK}[n]|]}{2^{13}} \right\rfloor, & 3 \leq n \leq M \end{cases} \tag{5.1.7}$$

The RAshift12 and RAshift tables are defined by the IEEE 1857.2 standard (see Figure A.1 and Figure A.2), so they are fixed and can be used by the IEEE 1857.2 encoder and decoder. The normalization process is described in the following pseudo-code where the LSB (Less Significant Bits) that have been shifted out are directly stored in the output bitstream (or output File).

Algorithm 5.1 Down-shift Operation for the Pre-processing block

1: **for** $n = 1, \ldots, M$ **do**
2: mask $= (1 << \text{shift}[n]) - 1$
3: LSB$[n] = |e[n]|$ &&mask //output in shift[n] bits
4: flat_errors$[n] = |e[n]| >> \text{shift}[n]$
5: **end for**
6: **for** $n = M + 1, \ldots, L$ **do**
7: flat_errors$[n] = |e[n]|$
8: **end for**

An example for Algorithm 5.1 is the following:

Example 5.1.1

Suppose that $e[n] = (689)_{10} = (1010110001)_2$ and $\text{shift}[n] = (5)_{10}$, therefore

mask = (1 << shift[n]) − 1 $\iff$ mask = $(2^{\text{shift}[n]} - 1)_{10} = (2^5 - 1)_{10} = (31)_{10} = (11111)_2$. This means that LSB = $(1010110001\&\&0000011111)_2 = (0000010001)_2$ in shift[n] = 5-bits, hence LSB = $(10001)_2$ and this is stored to the file (bitstream). Now the rest information of the binary representation of $e[n]$ will be written in flat_errors[n] by right shifting the absolute value of $e[n]$ for shift[n] times: flat_errors[n] = $|e[n]|$ >> shift[n] = $(1010110001)_2 >> 5 = (0000010101)_2$. In the post-processing block (5.1.5), we will se that we can retrieve the prediction error $e[n]$ by applying: $e[n]$ = flat_errors[n] << shift[n] + LSB = $(1010100000)_2 + (10001)_2 = (1010110001)_2$.

Notice that in Example 5.1.3 the masking operation of the LSB can be written as LSB = $689 \bmod 2^{\text{shift}[n]} = 689 \bmod 32 = 17_{10} = (10001)_2$.
Also the signs of the prediction errors are outputed from this block, so we don't lose the sign information.

5.1.4 Source Coding and Source Decoding

In the source coding block of Figure 5.1, we will find the encoded representation of the flattened residue using three types of source coding techniques:

- Finite precision Arithmetic Coding using integer representation (see 4.4.3).
- Adaptive Golomb-Rice Codes.
- Exponential Golomb Codes.

The Arithmetic Coding procedure is considered to be ideal and non-realistic in our scenario because it is not adaptive and the IEEE 1857.2 decoder can not estimate the source probabilities for every prediction residue. So we suppose that these probabilities are known and we will use it only as a benchmark to see how "close" are the adaptive Golomb-Rice and the Exponential Golomb codes, in terms of compression efficiency. It is important to state that none of these coding techniques use a codebook, and that the Golomb-Rice and the Exponential Golomb codes (adaptive or not) can be seen in many audio codecs [50], [2], [1] as long as in many video [51] and image [52] codecs. Also we can see that adaptive Golomb-Rice codes are being used in ECG signal compression for

medical applications [4]. We will use the same approach as in [2], [4] and [53] in order to make the Golomb-Rice Codes adaptive. So for $m = 2^k$ we will choose a good value for k such that:

$$k = \left\lfloor \log_2 \left(\frac{\sum_{n=1}^{\text{Total Samples}} \text{flat\_errors}[n]}{\text{Total Samples}} \right) \right\rfloor \tag{5.1.8}$$

In equation (5.1.8), we can see that we estimate k by measuring the average of the $\{\text{flat\_errors}[n]\}$ signal. Using some logic, it is true that k is analogous to the expected amplitude of the $\{\text{flat\_errors}[n]\}$ signal, so that Golomb-Rice Coding is more efficient and Exponential Golomb Coding (from Section 4.5.2 and Section 4.6 respectively). As we have already mentioned, the source coding techniques that we are going to use are depicted in Figure 5.6.

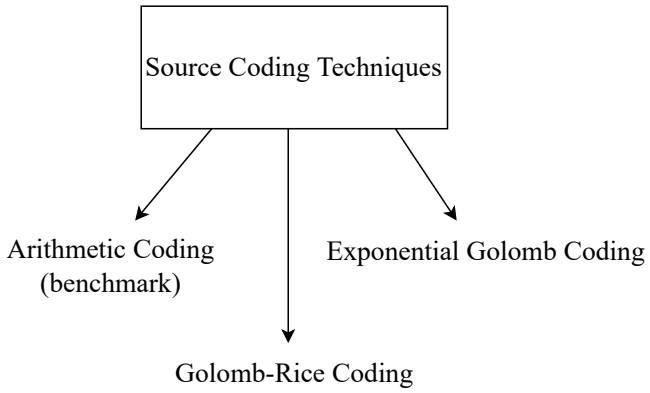


Figure 5.6: Source Coding techniques that we are going to use.

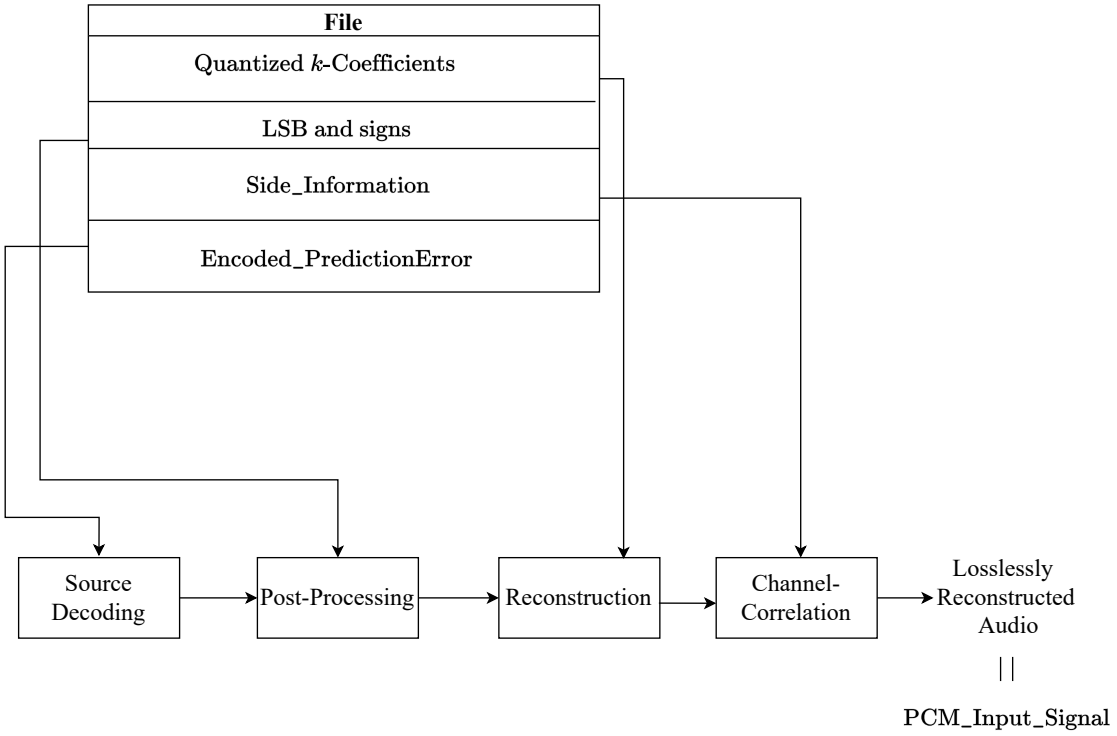


Figure 5.7: Diagram of the IEEE 1857.2 Decoder.

5.1.5 Post-processing

In the post-processing block (see Figure 5.8 and [48]) we de-normalize (de-flatten) the flattened prediction error signal's $\{\text{flat\_errors}[n]\}$ first $M = \min(N, 16)$ samples and we recover the signs of the prediction error signal $\{e[n]\}$.

At first we use equation (5.1.7) for the quantized k-coefficients. Then, in order to de-flatten the prediction errors and recover their signs we perform Algorithm 5.2.

Note that this algorithm must be performed for every frame of the audio signal. In order to understand Algorithm 5.2, see Example 5.1.3.

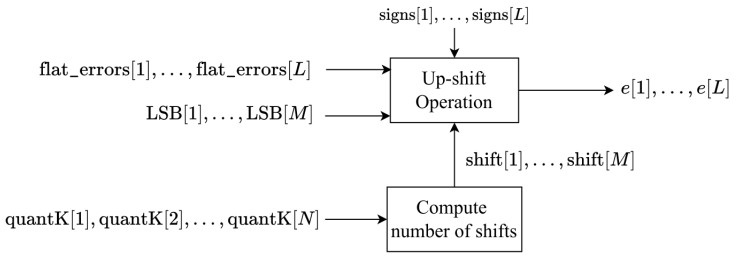


Figure 5.8: Diagram of the Post-Processing block in IEEE 1857.2.

Algorithm 5.2 Up-shift operation for the Post-processing block

for $n = 1, \ldots, M$ **do**
    $e[n] = (\text{flat\_errors}[n] << \text{shift}[n]) + \text{LSB}[n]$ //De-normalization and recovery $e[n]$
    if $\text{signs}[n] == 1$ **then**
        $e[n] = -e[n]$
    end if
end for
for $n = M + 1, \ldots, L$ **do**
    $e[n] = \text{flat\_errors}[n]$
    if $\text{signs}[n] == 1$ **then**
        $e[n] = -e[n]$
    end if
end for

5.1.6 Reconstruction

In Reconstruction block, the quantized k-coefficients (quantK()) are extracted from the stored **File** and then they are locally dequantized (dequantK()) by using equations (5.1.3) and (5.1.4) respectively, and converted to the linear prediction k-coefficients, which are exactly the same with those used in the IEEE 1857.2 encoder. The linear predictor generates a prediction signal $\hat{y}[n]$, which is added to the recovered prediction error signal $\{e[n]\}$ that is the output of the Post-processing block to losslessly reconstruct the original input audio (see Figure 5.9).

Again, as in the Linear Predictive Model block in 5.1.2, we perform the k-parameters-to-a-coefficients algorithm (see Algorithm 2.1). Having the a-coefficients and the prediction errors $e[1], \ldots, e[L]$ we can create $\hat{y}[n]$ (using (5.1.6)) and knowing that $e[n] + \hat{y}[n] = y[n]$ where $y[n]$ is the original input sample n we can recover the input PCM signal's frame where with the total number of frames we have the losslessly reconstructed audio (see Figure 5.9).

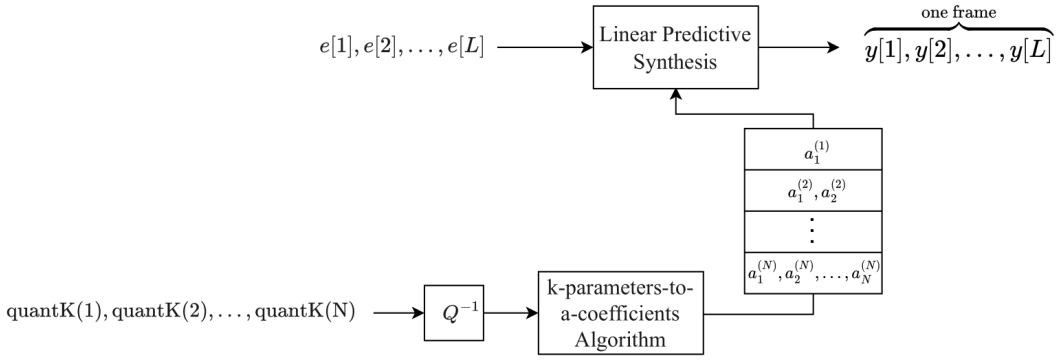


Figure 5.9: Diagram of the Reconstruction block in IEEE 1857.2.

5.1.7 Channel Correlation

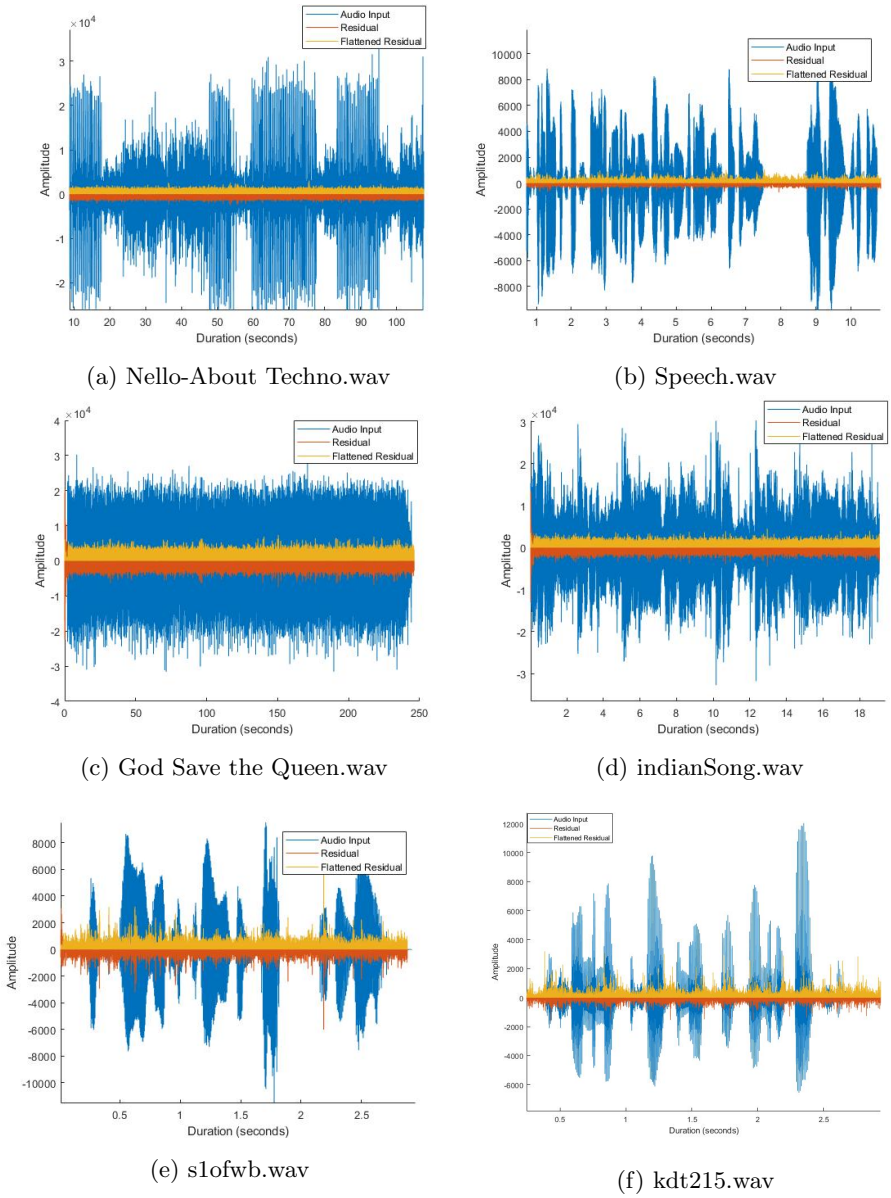
The Channel Correlation block is activated only if the input audio signal is a stereo signal with two channels. Remind that the input audio signal was PCM_Input_Audio $= \left[X^{(L)} \;\; X^{(R)}\right]$ and we only have the Mid signal and the Side information that is stored in the File. To originally reconstruct the input audio we need to perform the inverse of (5.1.1) and (5.1.2). That is:

$$X^{(L)} = \text{Mid} + \frac{\text{Side}}{2} \tag{5.1.9}$$

$$X^{(R)} = \text{Mid} - \frac{\text{Side}}{2} \tag{5.1.10}$$

5.2 Results

At first we will start by seeing some plots, using different audio signals, that depict the input audio signal, the error signal and the flattened error signal. Some of them were created by my friend Nikolaos Marinelis (you can go to [54] to listen to his music). As we can see the audio signal will have a significantly larger alphabet if it is sent to the source encoder directly. In the IEEE 1857.2 lossless audio extension, we send the flattened prediction error signal's frames (flattened residual's frames) instead of the prediction error signal's frames (residual's frames) to the source coder. In Figure 5.10 we present some sub-figures to clarify this point.



(a) Nello-About Techno.wav

(b) Speech.wav

(c) God Save the Queen.wav

(d) indianSong.wav

(e) s1ofwb.wav

(f) kdt215.wav

Figure 5.10: Residuals and flattened residuals of the input audio signal.

Before we see the underlying probability distribution of the residual and the flattened residual, i.e. the prediction error signal $\{e[n]\}$ and the flattened prediction error signal $\{\text{flat\_errors}[n]\}$ respectively, it is important to say that these signals always follow a geometric distribution, that its counterpart for continuous time signals is the Laplacian distribution. A random variable X has a Laplace(μ,b) distribution where $\mu = \mathcal{E}\{X\}$, $\sigma = \mathcal{E}\left\{(X-\mu)^2\right\}$ is the variance and b is the scale parameter where $\sigma^2 = 2b^2$ and its probability density function is:

$$f_X(x) = \frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right) \tag{5.2.1}$$

In bibliography [53], we can see the above probability distribution being defined as:

$$f_X(x) = \frac{1}{\sqrt{2}\sigma}\exp\left(-\frac{\sqrt{2}}{\sigma}|x-\mu|\right) \tag{5.2.2}$$

We will call the above probability distribution as the alternative definition for Laplacian. Below we can see some figures, to illustrate the probability density functions of (5.2.1) and (5.2.2), where $\mu = 0$.

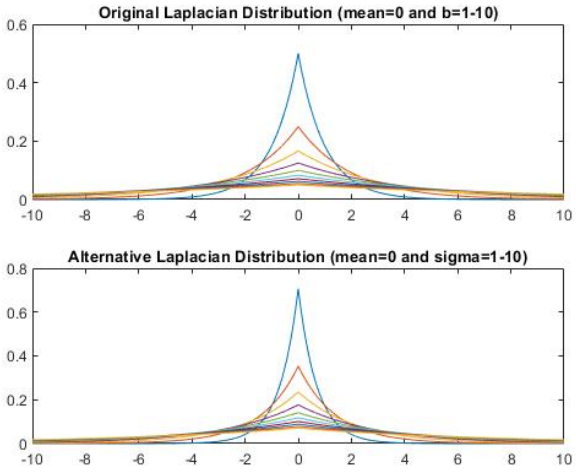
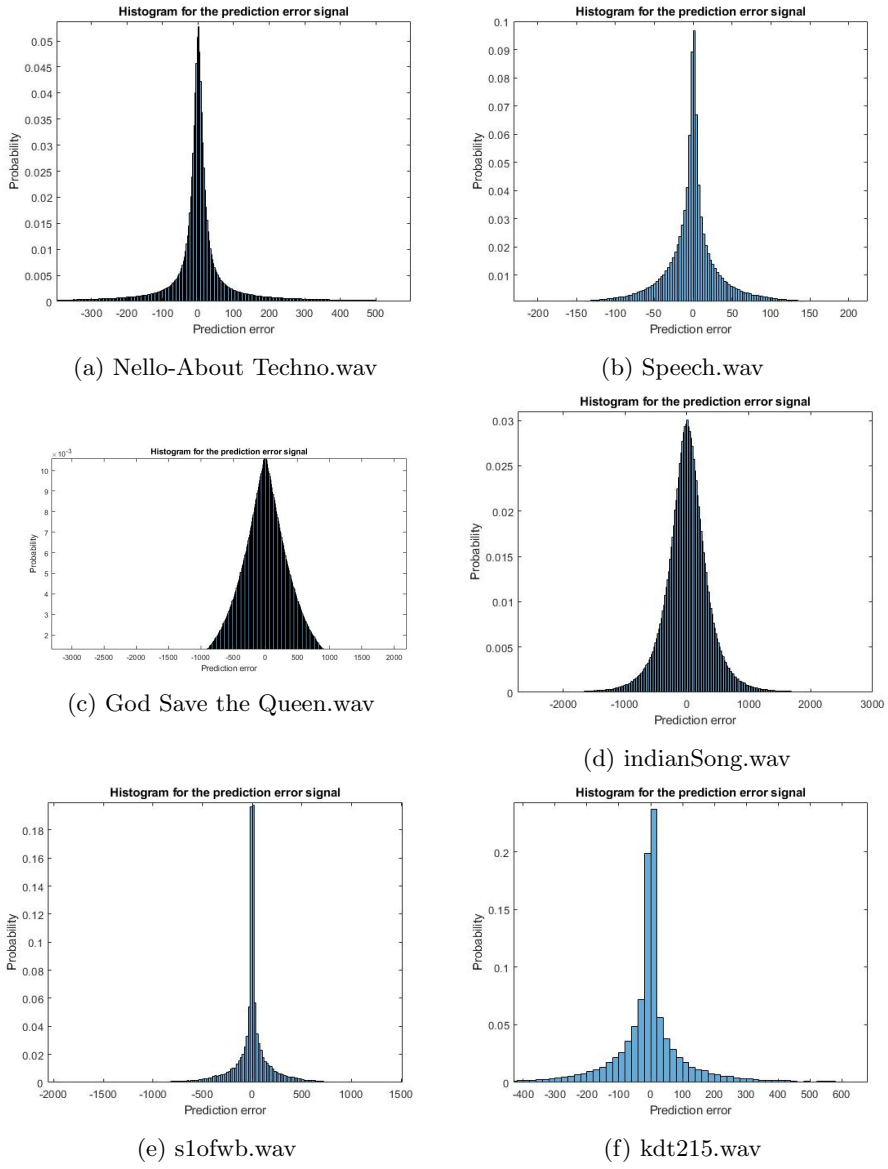


Figure 5.11: Laplacian and alternative definition for Laplacian distribution for $\mu = 0$

Using (5.2.2) we will se how well this distribution "fits" to the probabilities of the prediction

error signal $\{e[n]\}$, i.e the residual.



(a) Nello-About Techno.wav

(b) Speech.wav

(c) God Save the Queen.wav

(d) indianSong.wav

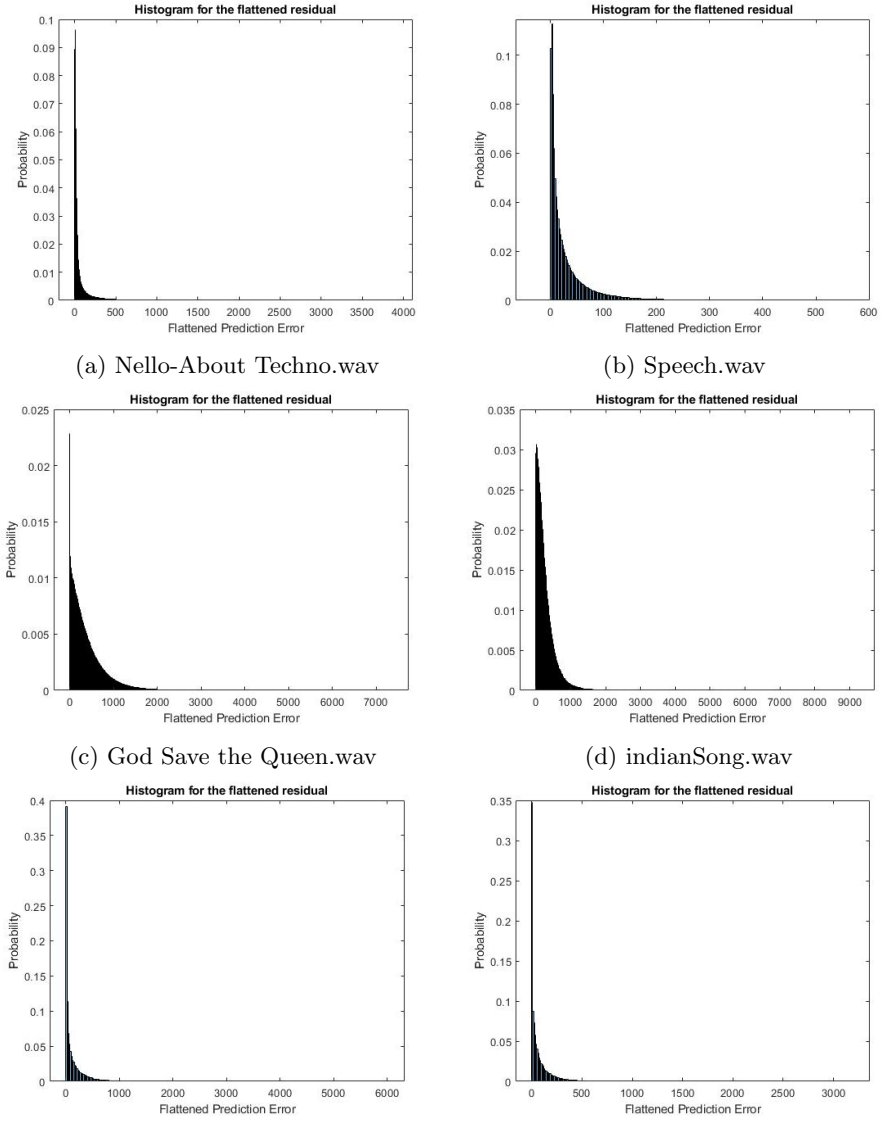
(e) s1ofwb.wav

(f) kdt215.wav

Figure 5.12: Prediction error signal distribution vs alternative Laplacian distribution

To derive Figure 5.12 we normalized the Laplacian distribution of (5.2.2) by experimenting with several values for σ. Each subfigure depicts the probability distribution of $\{e[n]\}$.

Afterwards we proceed to the pre-processing block (see 5.1.3). We can clearly see that the flattened prediction errors, i.e. the flattened residuals, are non-negative (it is clear if we observe the plots on Figure 5.10 and Figure 5.13).



(a) Nello-About Techno.wav

(b) Speech.wav

(c) God Save the Queen.wav

(d) indianSong.wav

(e) s1ofwb.wav

(f) kdt215.wav

Figure 5.13: Flattened prediction error signal distribution.

We have seen that the output of the pre-processing block, that is the flattened prediction error signal {flat_errors[n]}, will be the input of the source encoder. In order to measure compression efficiency (how much we have compressed the audio signal), it is important to define compression ratio as:

$$\text{Compression Ratio} = \frac{\text{Input File bits}}{\text{Output File bits}} \tag{5.2.3}$$

We also define Redundancy as the proportion of wasted "space" after applying compression. In other words it is defined as the percentage of the distance between the expected codeword length and the entropy of the source X:

$$\text{Redundancy} = \frac{\bar{L} - H(X)}{H(X)} \cdot 100\% \tag{5.2.4}$$

By Theorem 4.1.2 we can see that Redundancy is strictly non-negative.
For encoding the prediction error signal, we will use the source coding techniques mentioned below:

- Arithmetic Coding (ideal an used as benchmark)
- Adaptive Golomb Coding ([4], [53], [2]).
- Exponential Golomb Coding by exhaustively searching for the best k (the k that gives us the best compression performance).

5.2.1 Compression efficiency and Redundancy of Source Coding.

To measure Compression Ratio correctly, we must state which values we will have to store as bits in the computer and how many bits we will use for each one of them. We have already seen, when we were discussing about the IEEE 1857.2 blocks, that to have a correct reconstruction we are going to need:

- The flattened prediction error signal {flat_errors[n]} in the form of an AxB sized array where $A = 1024$ (fixed frame length of 1024 samples) and B is the number of total frames. The number of bits of the flattened prediction error signal will be the total number of bits at the output of the Source Encoder.

- The total number of frames B (negligible as 20 bits are already enough because if we have $2^{19}-1$ total frames then with $A = 1024$ samples in each frame and a sampling frequency of the input signal that is $44100\frac{\text{samples}}{\text{second}}$ we can have an input audio file of $\frac{1024 \cdot (2^{19}-1)}{44100} \approx 200$ minutes maximum).

- The quantized k-coefficients (or quantized PARCOR coefficients) that we defined them as quantK in subsection 5.1.2 that are restricted to the interval $[-64, 63]$ and are stored in a NxB array where N =prediction_order and B is the total number of frames. Therefore we need $\underline{N \cdot B \cdot 7}$ bits.

- The signs of the prediction error signal, as an AxB array, where we encode $+1$ to 0 and -1 to 1. So we just need one bit for every sample, therefore $\underline{A \cdot B \text{ bits}}$ in total.

- The LSB data with size MxB, where $M = \min\{N, 16\}$ that needs totally max{shift[n]} bits (see (5.1.7) and Algorithm 5.1) for every sample, therefore we need $\underline{\max\{\text{shift[n]}\} \cdot B \cdot M}$ bits in total.

- The Side Information (only for stereo audio signals) 16 bits for each sample because we send it uncoded. There will be a discussion for this at the last chapter.

- The prediction order (negligible number of bits).

- Parameter m for adaptive Golomb-Rice Coding (negligible).

- Parameter k for exponential Golomb Coding (negligible).

To measure compression ratio for this case we will consider all of the bullets above and add the extra bit lengths to the denominator of equation (5.2.3). Every audio signal will be of a certain genre.

Stereo signals were converted to mono, using Audacity software [55]. In the next page we will see a table with the results.

Compression ratio results for mono audio signals:

			Compression Ratio		
Audio File	**Genre**	**Order**	Arithmetic Coding	Adaptive Golomb Coding	Exponential Golomb Coding
Nello-Crystal Guitar.wav	Electronic	10	1.9231	1.5983	1.6187
Nello-Break.wav	Hip-Hop	10	1.9459	1.7203	1.7453
Nello-About Techno.wav	Techno	5	2.1739	1.8107	1.8987
Sex Pistols - God Save the Queen.wav	Punk	10	1.605	1.4307	1.3967
indianSong.wav	Folk	20	1.6382	1.5013	1.4713
s1ofwb.wav	Speech	20	2.018	1.7145	1.7657
speech.wav	Speech	10	2.3858	2.0984	2.122
kdt215.wav	Speech	20	2.2603	1.8516	1.9184
racing.wav	Sound Effect	5	1.8417	1.6922	1.6799
bubbles.wav	Sound Effect	5	2.1203	2.0483	1.9772

Redundancies for mono audio signals:

			Redundancy		
Audio File	**Genre**	**Order**	Arithmetic Coding	Adaptive Golomb Coding	Exponential Golomb Coding
Nello-Crystal Guitar.wav	Electronic	10	9.68%	35.3%	33%
Nello-Break.wav	Hip-Hop	10	14.86%	32.24%	30.1%
Nello-About Techno.wav	Techno	5	10.34%	36.15%	28.9%
Sex Pistols - God Save the Queen.wav	Punk	10	15.57%	31.43%	34.9%
indianSong.wav	Folk	20	18.7%	31.21%	33.79%
s1ofwb.wav	Speech	20	13.15%	36.6%	32.12%
speech.wav	Speech	10	11.26%	29.27%	27.56%
kdt215	Speech	20	9.56%	38.4%	32.89%
racing.wav	Sound Effect	5	8.67%	30.6%	31.67%
bubbles.wav	Sound Effect	5	21.76%	26.77%	32.05%

As we can see, we use small prediction orders. This happens because as the prediction order increases, the more k-coefficients we have for each frame. The LSB data might increase too if the prediction order is greater than 16. To illustrate this, we have compressed kdt215.wav file using a lot of prediction orders and see the compression ratio results for different prediction orders, using Arithmetic Coding. This is depicted in Figure 5.14 where we notice that we have a peak for prediction order $N \approx 20$, i.e the highest compression ratio is achieved when the prediction order $N \approx 20$.

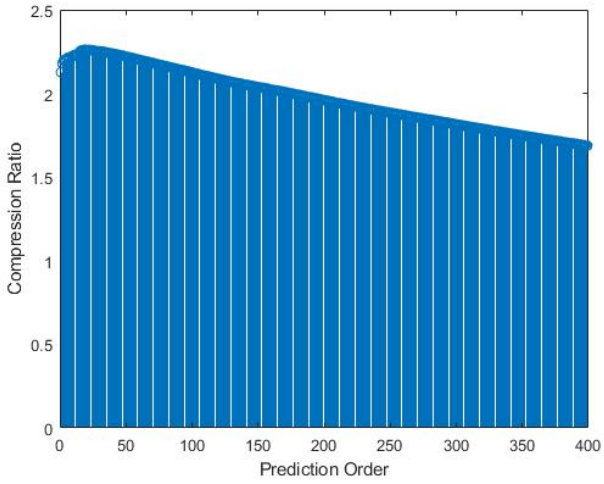


Figure 5.14: Compression Ratio vs Prediction Order for kdt215.wav

Compression ratio results for stereo audio signals:

			Compression Ratio		
Audio File	**Genre**	**Order**	Arithmetic Coding	Adaptive Golomb Coding	Exponential Golomb Coding
Nello-Crystal Guitar.wav	Electronic	10	1.2837	1.1858	1.1948
Nello-Break.wav	Hip-Hop	10	1.2810	1.2122	1.2223
Nello-About Techno.wav	Techno	5	1.3307	1.2375	1.2655
Sex Pistols - God Save the Queen.wav	Punk	10	1.2214	1.1422	1.1316

We notice that the compression ratio gets significantly smaller, because we save the Side Information uncoded, at the cost of 16-bits for each sample.

Considering our encoding and decoding algorithms, the fastest encoding is achieved by the adaptive Golomb Code. The exponential Golomb Code has a similar encoding speed, but in our case, because we have done exhaustive search for the optimum k it is the slowest one. Arithmetic Coding has a similar encoding speed with the adaptive Golomb Code, but the decoding speed is slower. Exponential Golomb Code and the adaptive Golomb code have approximately the same decoding speeds, which are much faster than the Arithmetic decoding speed.

Optimal m and Optimal k for adaptive Golomb-Rice Coding and Exponential Golomb Coding respectively for mono audio signals:

			$m = 2^k$	k
Audio File	**Genre**	**Order**	Adaptive Golomb Coding	Exponential Golomb Coding
Nello-Crystal Guitar.wav	Electronic	10	128	6
Nello-Break.wav	Hip-Hop	10	64	5
Nello-About Techno.wav	Techno	5	64	4
Sex Pistols - God Save the Queen.wav	Punk	10	256	8
indianSong.wav	Folk	20	128	7
speech.wav	Speech	10	16	3
s1ofwb.wav	Speech	20	64	5
kdt215.wav	Speech	20	32	4
racing.wav	Sound Effect	5	64	6
bubbles.wav	Sound Effect	5	32	5

Notice from Figure 5.12 that m and k is analogous to variance of the probability distribution of the source. For this reason when we have a greater variance, m and k get greater too, and vice versa. This happens because for smaller variances, we expect to assign smaller codewords for the signal values flat_errors[n] that are equal to zero (or close to zero) and have a high probability of appearance at the cost of bigger codewords for the signal values that are futher away from zero and have a low probability of apperance. This is because the source always follows a geometric distribution considering that we have a discrete-time digital signal at the input of the source encoder, whose equivalent distribution for continuous-time signals is the Laplacian (5.2.1), (5.2.2).

Chapter 6

Conclusion and Future Work

In Chapter 5 we have seen a method to losslessly compress and decompress digital audio signals. This method uses a pre-processing and a post-processing block to reduce the dynamic range of the prediction error signal. Other methods, like [3], [1] use algorithms to find adaptively the optimal block sizes, which we considered to be fixed in this thesis. So this is something that the author wants to study further. In addition, it is very important to do our experiments faster and Matlab programming language, that was used exclusively in this thesis, is not the best for digital audio compression. The author would prefer to implement an audio codec to a lower-level programming language (C++, assembly) in order to achieve faster encodings and decodings. In addition, C++ allows for the implementation of multithreading, that would be really helpful because we could encode more information without losing time, because the encodings (and decodings) for the Side Information would be done in parallel. In this thesis we saved some information without coding it (we saw in Chapter 5 and in Section 5.2 that we encode the prediction error signal $e[n]$ only, and all the other information is being stored uncoded). This would increase the complexity of the scheme, but it would make it more efficient in terms of compression. Notice that the linear predictive model has a very broad use [4], [1], [3], [51], [6] and the general scheme that was described is not limited to digital audio signals.

The author is very pleasant and thankful that you have studied his thesis. Best wishes to you!

Bibliography

[1] "Free lossless audio codec." `https://xiph.org/flac/`.

[2] Y. A. Reznik, "Coding of prediction residual in MPEG-4 standard for lossless audio coding (MPEG-4 ALS)," in *2004 IEEE International conference on acoustics, speech, and signal processing*, vol. 3, pp. iii–1024, IEEE, 2004.

[3] T. Liebchen and Y. A. Reznik, "MPEG-4 ALS: An emerging standard for lossless audio coding," in *Data Compression Conference, 2004. Proceedings. DCC 2004*, pp. 439–448, IEEE, 2004.

[4] T.-H. Tsai and W.-T. Kuo, "An efficient ECG Lossless Compression System for Embedded Platforms with Telemedicine Applications," *IEEE Access*, vol. 6, pp. 42207–42215, 2018.

[5] M. Hans and R. W. Schafer, "Lossless compression of digital audio," *IEEE Signal processing magazine*, vol. 18, no. 4, pp. 21–32, 2001.

[6] V. Melkote, M. Law, and R. Wilson, "Hierarchical and Lossless Coding of audio objects in Dolby TrueHD," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 384–388, IEEE, 2015.

[7] "IEEE Standard for Advanced Audio Coding," *IEEE Std 1857.2-2013*, pp. 1–343, 2013.

[8] V. D. B. Maxim, "Linear predictive coding and Golomb-Rice codes in the FLAC lossless audio compression codec." `https://scripties.uba.uva.nl/download?fid=679884`, 2020.

[9] Wikipedia, "Stochastic process." https://en.wikipedia.org/wiki/Stochastic_process.

[10] A. Liavas, "Telecommunications I, Lecture Notes," *Technical University of Crete*.

[11] M. Paterakis, "Probability Theory and Random Signals, Lecture Notes," *Technical University of Crete*.

[12] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing*, pp. 890–941. Upper Saddle River, N.J.: Pearson, 3rd ed., 2010.

[13] J. Johansen, A. J. Fuglsig, K. Stern, and K. Ramsgaard-Jensen, "Improving the quality of lpc-encoding," tech. rep., Aalborg University, 2016.

[14] A. Malek, "Signal framing." https://superkogito.github.io/blog/2020/01/25/signal_framing.html.

[15] ElectronicsCoach, "Pulse code modulation." https://electronicscoach.com/pulse-code-modulation.html.

[16] J. G. Proakis and M. Salehi, *Fundamentals of Communication Systems*, pp. 313–320. Upper Saddle River, N.J.: Pearson, 2014.

[17] "Beckhoff information system." https://infosys.beckhoff.com/english.php?content=../content/1033/tf3680_tc3_filter/5843792395.html&id=.

[18] G. Karystinos and A. Liavas, "Signals and Systems, Lecture Notes," *Technical University of Crete*.

[19] R. Gallager, "Quantization, Lecture Notes," *Massachusetts Institute of Technology*, 2006.

[20] A. Bletsas, "Telecommunications II, Lecture Notes," *Technical University of Crete*.

[21] SONY, "Digital audio processor pcm-f1." https://www.kenrockwell.com/audio/sony/pcm-f1.htm, 1981.

[22] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing*, pp. 99–152. Upper Saddle River, N.J.: Pearson, 3rd ed., 2010.

[23] A. V. Oppenheim, "MIT, Digital Signal Processing Lectures." `https://www.youtube.com/watch?v=rkvEM5Y3N60&list=PL8157CA8884571BA2`, 1975.

[24] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing*, pp. 374–492. Upper Saddle River, N.J.: Pearson, 3rd ed., 2010.

[25] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, pp. 83–112. New York: Springer Science + Business Media, LLC, 3rd ed., 1992.

[26] Y. You, *Audio coding. Theory and applications*, pp. 53–65. New York Dordrecht Heidelberg London: Springer, 2010.

[27] J. Makhoul, "Linear prediction: A tutorial review," *Proceedings of the IEEE*, vol. 63, no. 4, pp. 561–580, 1975.

[28] D. O'Shaughnessy, "Linear predictive coding," *IEEE potentials*, vol. 7, no. 1, pp. 29–32, 1988.

[29] J. Makhoul, "Stable and efficient lattice methods for linear prediction," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 5, pp. 423–428, 1977.

[30] C. Margina and B. Costinescu, "Implementing the Levinson-Durbin Algorithm on the StarCore™ SC140/SC1400 Cores," 2001.

[31] F. Itakura, "Minimum prediction residual principle applied to speech recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 23, no. 1, pp. 67–72, 1975.

[32] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* Wiley-Interscience, 2nd ed., 2006.

[33] R. Gallager, *Information Theory and Reliable Communication.* John Wiley and Sons, 1968.

[34] G. Karystinos, "Information Theory and Coding, Lecture Notes," *Technical University of Crete.*

[35] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[36] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[37] K. Sayood, *Introduction to Data Compression, Third Edition (Morgan Kaufmann Series in Multimedia Information and Systems)*, pp. 81–115. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[38] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[39] G. G. Langdon, "An introduction to Arithmetic Coding," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 135–149, 1984.

[40] S. Golomb, "Run-length encodings," *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, 1966.

[41] A. Kiely and M. Klimesh, "Generalized Golomb codes and adaptive coding of wavelet-transformed image subbands," *Interplanetary Network Progress Report*, vol. 42, pp. 1–14, 2003.

[42] J. Teuhola, "A compression method for clustered bit-vectors," *Information processing letters*, vol. 7, no. 6, pp. 308–311, 1978.

[43] R. Gallager and D. Van Voorhis, "Optimal source codes for geometrically distributed integer alphabets," *IEEE Transactions on Information theory*, vol. 21, no. 2, pp. 228–230, 1975.

[44] A. Said, "On the determination of optimal parameterized prefix codes for adaptive entropy coding," *HP Labs Report*, pp. 11–24, 2006.

[45] R. F. Rice, "Some practical universal noiseless coding techniques," tech. rep., 1979.

[46] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[47] H. Huang, H. Shu, and R. Yu, "Lossless audio compression in the new IEEE standard for advanced audio coding," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6934–6938, IEEE, 2014.

[48] A. R. Lopez, "Lossless audio compression in IEEE 1857.2." `https://github.com/adriaromero/lossless-audio-compression`.

[49] F. A. Muin, T. S. Gunawan, E. M. Elsheikh, and M. Kartiwi, "Performance analysis of IEEE 1857.2 lossless audio compression linear predictor algorithm," in *2017 IEEE 4th International Conference on Smart Instrumentation, Measurement and Application (ICSIMA)*, pp. 1–6, 2017.

[50] V. D. B. Maxim, "Adaptive Flac." `https://github.com/amsqi/adaptive-flac`, 2020.

[51] S. Nargundmath and A. Nandibewoor, "Entropy coding of H.264/AVC using Exp-Golomb coding and CAVLC coding," in *International Conference on Advanced Nanomaterials Emerging Engineering Technologies*, pp. 607–612, 2013.

[52] M. J. Weinberger, G. Seroussi, and G. Sapiro, "LOCO-I: A low complexity, context-based, lossless image compression algorithm," in *Proceedings of Data Compression Conference-DCC'96*, pp. 140–149, IEEE, 1996.

[53] T. Robinson, "Shorten: Simple lossless and near-lossless waveform compression," 1994.

[54] N. Marinelis, "Spotify channel." `https://open.spotify.com/artist/5RnFZhnod15UpgMGw9rHMu?si=1ZKaPrDzQu6LPRoK3Iq4Ug`.

[55] "Audacity software." `https://www.audacityteam.org/`.

Appendix A

Supplementaries for the IEEE 1857.2 Standard

A.1 RA_shift and RA_shift12 tables

Here we present the RAshift and RAshift12 tables for the IEEE 1857.2 standard (see [7]). Note that a_k are the quantized k-coefficients from (5.1.3), so $a_k = \text{quantK}(k)$ and must not be confused with the a-coefficients that are used for linear prediction synthesis and linear prediction analysis. In the standard we do not see a clear explanation of how these tables were derived. These tables are depicted in Figure A.1 and in Figure A.2.

[1] Figure from [7].

Table 4.9—*RA_shift*[a_k], k>2

a_k	*RA_shift*[a_k]	a_k	*RA_shift*[a_k]	a_k	*RA_shift*[a_k]	a_k	*RA_shift*[a_k]
0	0	16	381	32	1700	48	4885
1	1	17	432	33	1826	49	5214
2	6	18	487	34	1960	50	5570
3	13	19	545	35	2100	51	5956
4	23	20	607	36	2248	52	6378
5	36	21	673	37	2404	53	6841
6	52	22	743	38	2569	54	7354
7	71	23	817	39	2743	55	7927
8	93	24	896	40	2927	56	8573
9	118	25	978	41	3122	57	9313
10	146	26	1066	42	3329	58	10176
11	177	27	1158	43	3548	59	11205
12	211	28	1255	44	3781	60	12476
13	249	29	1358	45	4030	61	14128
14	290	30	1466	46	4296	62	16477
15	334	31	1580	47	4580	63	20526
						64	23147

Figure A.1: RAshift table for the pre-processing and post-processing blocks. [1]

[2]Figure from [7].

Table 4.8—*RA_shift12*[a_k], k = 1,2

a_k	*RA_shift*12[a_k]	a_k	*RA_shift*12[a_k]	a_k	*RA_shift*12[a_k]	a_k	*RA_shift*12[a_k]
–64	58348	–32	12084	0	3577	32	49
–63	48794	–31	11702	1	3399	33	25
–62	43108	–30	11330	2	3226	34	9
–61	39207	–29	10969	3	3056	35	1
–60	36249	–28	10618	4	2891	36	1
–59	33866	–27	10277	5	2730	37	10
–58	31870	–26	9944	6	2573	38	29
–57	30151	–25	9620	7	2420	39	58
–56	28643	–24	9305	8	2271	40	98
–55	27298	–23	8997	9	2127	41	149
–54	26083	–22	8697	10	1986	42	213
–53	24977	–21	8404	11	1849	43	291
–52	23959	–20	8118	12	1717	44	384
–51	23018	–19	7839	13	1589	45	493
–50	22141	–18	7566	14	1465	46	621
–49	21321	–17	7300	15	1345	47	769
–48	20551	–16	7039	16	1229	48	939
–47	19824	–15	6785	17	1118	49	1135
–46	19136	–14	6536	18	1012	50	1360
–45	18483	–13	6293	19	909	51	1618
–44	17862	–12	6055	20	812	52	1915
–43	17269	–11	5822	21	719	53	2258
–42	16702	–10	5594	22	631	54	2655
–41	16159	–9	5372	23	548	55	3119
–40	15638	–8	5154	24	469	56	3667
–39	15136	–7	4941	25	397	57	4320
–38	14654	–6	4733	26	329	58	5117
–37	14189	–5	4529	27	267	59	6113
–36	13740	–4	4330	28	211	60	7409
–35	13305	–3	4135	29	161	61	9209
–34	12885	–2	3944	30	117	62	12043
–33	12479	–1	3758	31	79	63	18365

Figure A.2: RAshift table for the pre-processing and post-processing blocks. [2]

Appendix B

Supplementaries on Source Coding

B.1 Golomb Code examples

At first we will se some tables representing some Golomb codewords for $m = 13$, $m = 11$ and $m = 7$:

$\boldsymbol{G_c^{(13)}(n)}$

n	Codeword
0	0|000
1	0|001
2	0|010
3	0|0110
4	0|0111
5	0|1000
6	0|1001
7	0|1010
8	0|1011
9	0|1100
10	0|1101
11	0|1110
12	0|1111
13	10|000
14	10|001
15	10|010

n	Codeword
16	10|0110
17	10|0111
18	10|1000
19	10|1001
20	10|1010
21	10|1011
22	10|1100
23	10|1101
24	10|1110
25	10|1111
26	110|000
27	110|001
28	110|010
29	110|0110
30	110|0111
31	110|1000

$\boldsymbol{G_c^{(11)}(n)}$

n	Codeword	n	Codeword
0	0|000	16	10|1010
1	0|001	17	10|1011
2	0|010	18	10|1100
3	0|011	19	10|1101
4	0|100	20	10|1110
5	0|1010	21	10|1111
6	0|1011	22	110|000
7	0|1100	23	110|001
8	0|1101	24	110|010
9	0|1110	25	110|011
10	0|1111	26	110|100
11	10|000	27	110|1010
12	10|001	28	110|1011
13	10|010	29	110|1100
14	10|011	30	110|1101
15	10|100	31	110|1110

$G_c^{(7)}(n)$

n	Codeword	n	Codeword
0	0|00	16	110|011
1	0|010	17	110|100
2	0|011	18	110|101
3	0|100	19	110|110
4	0|101	20	110|111
5	0|110	21	1110|00
6	0|111	22	1110|010
7	10|00	23	1110|011
8	10|010	24	1110|100
9	10|011	25	1110|101
10	10|100	26	1110|110
11	10|101	27	1110|111
12	10|110	28	11110|00
13	10|111	29	11110|010
14	110|00	30	11110|011
15	110|010	31	11110|100

Notice that in this thesis we have encoded discrete sources that have an underlying probability distribution, whose continuous-time counterpart is similar to the Laplacian (see (5.2.1), (5.2.2)), that is the geometric probability distribution. This has been done in a way, such that for greater values of the variance σ^2, we need greater values for m and vice versa.

Now for $m = 11$ we are going to show a binary tree using a code where we first perform unary coding of the quotient q and we concatenate it to its right with a binary code for the remainder r, considering that $n = qm + r$.

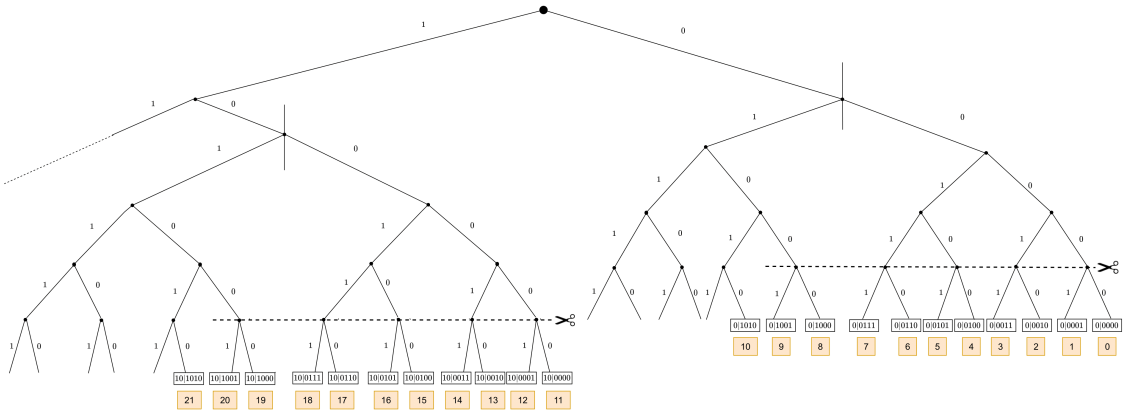


Figure B.1: Representation of r using $\lfloor \log_2(m) + 1 \rfloor$ bits.

Notice that in the binary tree of Figure B.1 we can truncate the binary tree as the code we have applied, even though is prefix-free, it is not complete. Therefore Kraft-Inequality (Definition (4.1.2) is not satisfied with equality. When we truncate the binary tree for this type of source code we will see that the code becomes complete, in the sense that Kraf-Inequality is satisfied with equality, thus we can not add a codeword without violating it. Also the code still retains its prefix-free property. To represent the new code we have created the truncated binary tree in Figure B.2. The encoding that is performed in Figure B.2, considering that $A = \lfloor \log_2(m) + 1 \rfloor$, is to represent the quotient q using an unary code and the code for the remainder r as explained below:

- if $r < 2^A - m \rightarrow$ the binary code of r has length of $\lfloor \log_2(m) \rfloor$-bits
- if $r \geq 2^A - m \rightarrow$ the binary code of r has length of $\lfloor \log_2(m) \rfloor + 1$-bits, where $r = r + 2^A - m$.

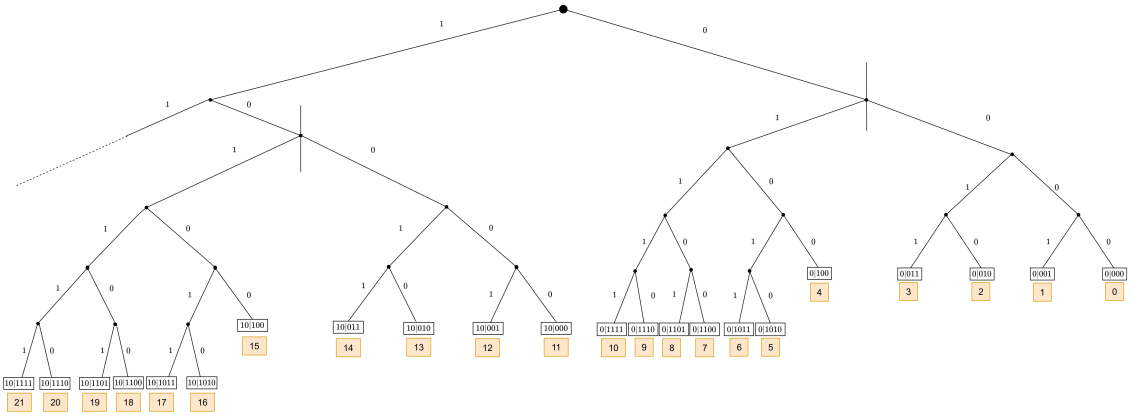


Figure B.2: Representation of r using $\lfloor \log_2(m) \rfloor$ bits for the smallest integers and $\lfloor \log_2(m) \rfloor + 1$ bits for the greatest.

The code represented in Figure B.2 is a Golomb Code for $m = 11$, or as we represented it in Section 4.5, it is the binary tree for $G^{(11)}(n)$ where $0 \leq n \leq 21$ and $n \in \mathbb{N}$.