UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

A Comparison between RS+TCM and LDPC FEC schemes for the G.fast standard

Marcos Yuichi Takeda

DM 04/2016

UFPA / ITEC / PPGEE

Campus Universitário do Guamá

Belém - Pará - Brasil

2016

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

A Comparison between RS+TCM and LDPC FEC schemes for the G.fast standard

Autor: Marcos Yuichi Takeda

Orientador: Aldebaro Barreto da Rocha Klautau Júnior

Dissertação submetida à Banca Examinadora do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Pará para obtenção do Grau de Mestre em Engenharia Elétrica. Área de concentração: **Telecomunicações**.

UFPA / ITEC / PPGEE
Campus Universitário do Guamá
Belém, PA
2016

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

A Comparison between RS+TCM and LDPC FEC schemes for the G.fast standard

AUTOR(A): MARCOS YUICHI TAKEDA

DISSERTAÇÃO DE MESTRADO SUBMETIDA À AVALIAÇÃO DA BANCA EXAMINADORA APROVADA PELO COLEGIADO DO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA, DA UNIVERSIDADE FEDERAL DO PARÁ E JULGADA ADEQUADA PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA NA ÁREA DE TELECOMUNICAÇÕES.

Prof. Dr. Aldebaro Barreto da Rocha Klautau Júnior
(Orientador - UFPA / ITEC)

Prof. Dr. Evaldo Gonçalves Pelaes
(Membro - UFPA / PPGEE)

Profa. Dra. Valquíria Gusmão Macedo
(Membro - UFPA / ITEC)

Acknowledgement

I would like to thank every colleague, friend, professor that I had contact at this 3 years of master degree program. The year of 2015 was a life changer year for me, as I was diagnosed with cancer. This result in almost a year of delay due to cancer treatment, namely chemotherapy and radiotherapy. Fortunately, the treatment worked well and now I could finally write this disseratation thesis.

Thus I want to truly thank everyone who gave me support during the heavy cancer treatment. My family, Kazuo, Maria and Akira Takeda, in first place, and friends from LaPS, and in special my advisor Prof. Aldebaro Klautau, which was very comprehensive, my former coadvisor Fernanda Smith, which took my place at a paper presentation when I was at treatment. Special thanks to Pedro Batista, Bruno Haick, Francisco Muller, Ana Carolina, Silvia Lins, Igor Almeida, Leonardo Ramalho, Joary Fortuna, Igor Freire, Kelly Souza, and many others from LaPS.

Also, I would want to thank friends outside UFPA and LaPS, that were invaluable during my treatment.

I would like to thank Prof. Evaldo Pelaes and Prof. Valquíria Macedo for reading and evaluating this dissertation.

Abstract

The need for increasing internet connection speeds motivated the creation of a new standard, G.fast, for digital subscriber lines (DSL), in order to provide fiber-like performance using short twisted pair copper wires. This dissertation focuses in the two former candidate codes for the forward error correction (FEC) scheme, namely Reed-Solomon (RS) with trellis-coded modulation (TCM), from previous DSL standards, and low density parity check (LDPC) codes, from the G.hn standard for home networking. The main objective is to compare both schemes, in order to evaluate which of them performs better, considering metrics such as error correction performance and computational cost. Even though the selected code for G.fast was RS with TCM, it is believed that this choice was biased, as a more recent iterative code like LDPC is known to have a better error correction performance.

Resumo

A necessidade de velocidades de conexão a internet cada vez maiores motivou a criação de um novo padrão, G.fast, para DSL (linha digital do assinante), de modo a prover performance semelhante a fibra usando par trançado de fios de cobre. Esta dissertação foca em dois prévios candidatos a esquemas de correção de erro, sendo eles codigos Reed-Solomon com modulaçao codificada em treliça (TCM), de padrões DSL anteriores, e códigos de verificação de paridade de baixa densidade (LDPC), do padrão G.hn para redes residenciais. O objetivo principal é comparar os dois esquemas, de modo a avaliar qual deles tem melhor performance, considerando métricas como poder de correção de erro e custo computacional. Embora o código selecionado para o G.fast já tenha sido RS+TCM, acredita-se que esta decisão foi tendenciosa, já que é sabido que códigos mais recentes como LDPC possuem maior poder correção.

Contents

List of Figures

List of Tables

List of Symbols

ACS Add-compare-select

API Application Programming Interface

AWGN Additive White Gaussian Noise

BCH Bose-Chaudhuri-Hocquenghem

BER Bit error rate

BLER Block error rate

BMU Branch metric unit

CD Compact disc

DMT Discrete multitone

DSL Digital subscriber line

DVB Digital video broadcasting

GF Galois Field

LDPC Low-density parity-check

LLR Log-likelihood ratio

LSB Least significant bit

MSB Most significant bit

OFDM Orthogonal frequency division multiplexing

PAM Pulse amplitude modulation

PMU Path metric unit

QAM Quadrature amplitude modulation

QC-LDPC-BC Quasi-cylic low-density parity-check block code

RS Reed-Solomon

SNR Signal-to-noise ratio

TBU Traceback unit

TCM Trellis-coded modulatin

VDSL2 Very-high-bit-rate digital subscriber line 2

XOR Exclusive or

Introduction

In light of the new G.fast standard for fiber-like speeds over twisted pair copper wires, there was a discussion at which forward error correction scheme would be used in the physical layer. The two main candidate schemes were Reed-Solomon codes with trellis-coded modulation, repeated from previous DSL standards like VDSL2, and LDPC codes, from the G.hn standard for home networks.

G.fast

The G.fast [1] standard was devised to provide up to 1 Gbps aggregated (uplink and downlink) rate over short twister pair copper wires. The motivation is to provide cabled infrastructure at a lower cost than using optical fiber. A deployment scenario example is a building, where the fiber reaches the base level and the network is further spread up using copper.

Technology

G.fast uses discrete multitone modulation (DMT), which can be viewed as a special case of orthogonal frequency division multiplexing (OFDM). DMT divides the channel into smaller non-overlapping sub-channels, transmitting a variable amount of bits depending on the conditions of each sub-channel. G.fast uses QAM in each sub-channel, with each sub-carrier using up to 4096-QAM (12 bits).

The use of DMT incur in a bitloading algorithm with channel estimation. This calculates precisely how many bits can be transmitted at each sub-channel, resulting in a efficient adaptation to the channel. As seen later, the use of a bitloading table plays a important role on using trellis-coded modulation on DMT systems.

Reed-Solomon with trellis-coded modulation

Reed-Solomon code [2] is a classical code with its main characteristic of having absolute control over its error correction properties. Also, they are used in a variety of applications as CDs, Blu-rays, barcodes, QR codes and secret sharing schemes as well. Basically, a Reed-Solomon code can correct half the number of the redundancy it inserts at a system.

Trellis-coded modulation [3] is also a classical scheme developed first for old telephone modems. Trellis-coded modulation introduction was revolutionary at the time, as it provided a way to jump from rates of 14 kbps to 33 kbps. It improved systems by increasing the number of bits per symbol instead of using more bandwidth.

LDPC

In the other hand, LDPC codes were discovered at 1963 [4], but remained forgotten for more than 30 years, because at the time the computational cost was not practical. With its rediscovery 30 years later, it gained rapid popularity as it managed to surpass the recent discovered Turbo codes. LDPC and Turbo codes belong to a class of codes called capacity approaching. In fact, LDPC codes, when used with very long codeword sizes, could achieve error correction curves within $0.0045dB$ in E_b/N_0 of the Shannon limit on AWGN channels, which is the theoretical limit of reliable transmission [5]. LDPC works by using long codewords and sparse parity check matrices. Its design can be pseudo-random, and its exact error correction performance is difficult to calculate. Generally, we measure the average behaviour of a LDPC code, as opposed to Reed-Solomon code, where we do not even need to measure, as the error correction performance is fixed at the design. Also, LDPC decoders use probabilities instead of bits, avoiding early loss of information caused by the hard decisions caused by demodulation.

State of art

A comparison between these codes was already made, but with another purpose. Neckebroek [6] showed that when considering impulse noise, Gaussian noise, with retransmissions. They showed that under these conditions, retransmissions result in better performance than FEC due to shorter symbol durations in G.fast, and thus impulse noise affecting an entire symbol at once. Retransmissions are better with short symbol durations as a rapid retransmission is possible, incurring on low latency.

Comparison motivation

In the end, the scheme of Reed-Solomon with trellis-coded modulation was chosen for the G.fast 100 MHz profile. Although this decision was already made, it is believed that it was biased to remain unchanged, and thus favoring existing implementations. At AWGN channel tests, the error correcting performance of Reed-Solomon with trellis-coded modulation was lower than that of LDPC. One may argue with computational cost and complexity, but this remains as a very difficult question, as specialized hardware exist for both schemes.

Outline

This work is divided as follows: Chapter 1 gives an quick overview of each code, Chapter 2 gives a detailed explanation of the codes used for each standard, Chapter 3 shows the comparison results for both schemes and Chapter 4 concludes with some final remarks.

Chapter 1

Fundamentals of channel coding

This chapter contains a brief review of the fundamentals of channel coding, with emphasis on the codes used on this works, namely Reed-Solomon codes, trellis-coded modulation and LDPC. This work will not assess all the details of coding theory, as there are a number of references which contain them [7–9].

1.1 Channel Coding

Channel coding is a way to improve efficiency in telecommunications systems. We can only get close to the so-called Shannon's channel capacity by using it. The channel can be anything: a cable, air, vacuum or even a media CD or hard drive. When information is sent/written to, or received/read from a channel, noise may corrupt data. The main concept of channel coding is to add controlled redundancy data to the information data in order to protect this information from such noise. Noise is anything undesirable, and out of our control that corrupts data such as thermal noise on cables or scratches on the CD surface.

A code can be defined as a set of codewords. A codeword is a representation of an array of symbols. A symbol can be one or more bit. When we use a bit as a symbol the code is said binary. The main motivation behind using a set of codewords to represent data is that we can expand the "space" between valid sequences. An analogy can be made with digital modulation: When dealing with modulation, the values are continuous, so to "open space" we just scale the constellation by a factor greater than 1, increasing the distance between points. In a code, the values are discrete and finite, so we need to map these values to a "bigger space" in order to make room for possible errors that may occur. The mapping procedure is called encoding, and the resulting values on which the original are mapped into form the code. The decoding procedure can be also seen as an analogy to modulation: the decoder receives a value, which is not necessarily valid, so we need to find the "nearest" valid value to the received value. The process of finding this nearest is called decoding.

The main objective of channel coding in telecommunications is to achieve transmission rates near the channel capacity. The channel capacity [10] for AWGNis calculated by

$$C = \log_2(1 + \mathrm{SNR}), \tag{1.1}$$

where C is the channel capacity in bits per channel use. Shannon proved in the noisy-channel coding theorem that we can get arbitrarily close to C using an appropriate code. The main problem is finding such a code. Also, for practical reasons, it is preferred that this code has manageable computational complexity for encoding and decoding. The word "redundancy" may sound as something that is not entirely necessary, but in channel coding it is the key to achieve optimal rates of transmissions in noisy channels.

Channel coding may be divided in two categories: Block codes and Convolutional codes.

1.1.1 Block codes

Block codes use fixed lengths for both data and redundancy. A Hamming(7,4) code is an example of a block code. It takes 4 information bits and creates more 3 bits of redundancy, comprising a codeword with 7 bits. Note that the new codeword may not resemble the original information at first glance, as we can further classify a code which maintain the original information in the codeword as systematic, and a code which does not as non-systematic. But this does not mean that non-systematic codes are not usable, as the original information can be recovered by using proper decoding algorithms. Nevertheless, the codes in this work are all from telecommunications standards, which use mostly systematic codes, justified by the fact that if a codeword is not corrupted, then we can just remove the redundancy and retrieve information more easily.

A linear block code is defined by a $k \times n$ matrix G, called generator matrix, or $(n-k) \times n$ matrix H, called parity-check matrix. We need just one of them, as given one, it is possible to calculate the other. The rows of G define a base of a k-dimensional vectorial subspace. Any vector in this subspace is called a codeword. The rows of H define the null space of the rows of G, such that $GH^T = 0$, meaning that any codeword $\hat{c}$ multiplied with H^T results in 0. The code rate of a linear block code is $R_{\text{block}} = k/n$.

Figure 1.1 shows how a systematic codeword is generated by a block code. The information of size k is concatenated to $n - k$ bits in order to generate a n bit codeword. In

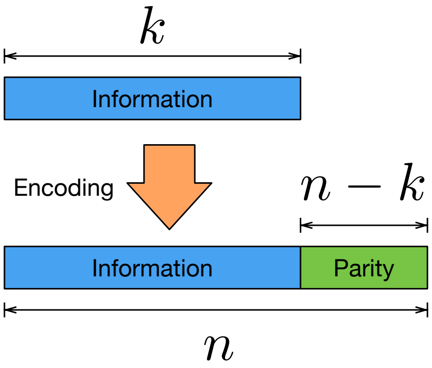


Figure 1.1: Systematic codeword generation by a block code.

this case the left side of G is an identity matrix, as the code is systematic. To return to the original information, we simply drop the parity the bits.

In order to decode a codeword, usually we use H first to verify if the received codeword is a valid codeword. Considering the received vector as $\hat{r} = \hat{c} + \hat{e}$, where $\hat{e}$ represents the error caused by the channel, we calculate the so called syndrome $\hat{s} = \hat{r}H^T$. If $\hat{s}$ is an all-zero vector, then $\hat{r}$ is a codeword. This does not directly imply that $\hat{r}$ is correct, but in general we assume this is true. This is because for $\hat{r}$ to be a codeword different of $\hat{c}$, the transmitted codeword, the error vector $\hat{e}$ must be a codeword, as $\hat{r}H^T = \hat{c}H^T + \hat{e}H^T = 0$, and $\hat{c}H^T = 0$, so we have that $\hat{e}$ is a codeword indeed. This case can be neglected as in practice we adopt codes where the minimum codeword weight is high, preventing that a low number of errors could confuse the decoder. When a non-zero syndrome is produced, we detect that an error has occurred, and try to correct these errors, if possible. This is where different types of codes differ. It is desirable that codes have efficient decoding algorithms, and that these algorithms can correct more errors with less redundancy.

1.1.2 Convolutional codes

Convolutional codes use a potentially infinite length of data as their input and produce an output with length larger than the input. A convolutional code acts similarly to a digital filter from signal processing. In other words, it has a combination of coefficients and memories that given an input, produce an output. The main differences are that, in a binary code, the coefficients can only be 0 or 1, which reduces to indicating only the presence or not of a connection, and that there is not only one input signal and one output signal, at each time step, the code receives k inputs and produces n outputs, from which we can calculate the code rate $R = k/n$.

The convolutional encoder can be seen as a sliding window calculating n different parities in parallel. The sliding window receives a stream of bits as its input and calculates the outputs, as shown in Figure 1.2. In this example, there are 3 red squares, where the first reads the actual value $x[n]$, and the next 2 squares read the delayed values $x[n-1]$ and $x[n-2]$. Thus, there are two binary memories.

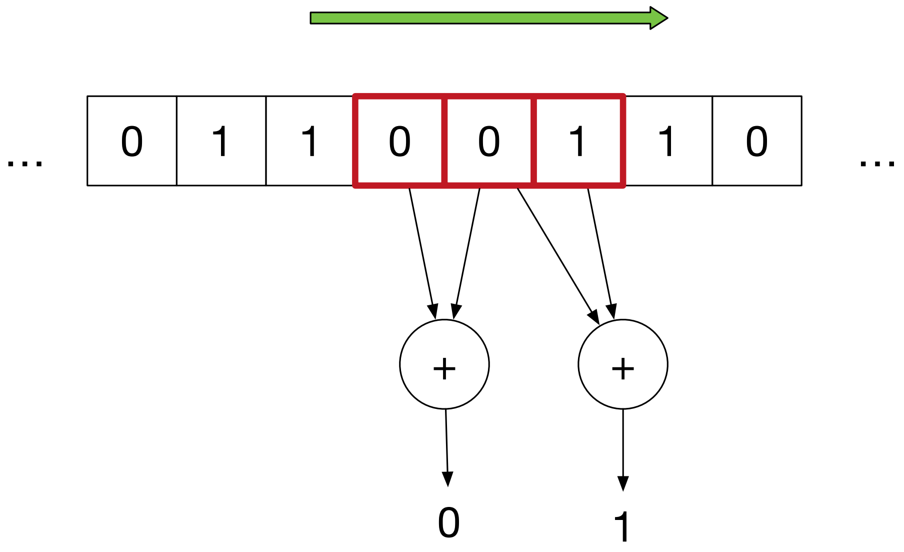


Figure 1.2: Representation of a convolutional code as parities on a sliding window.

As the memory elements inside a binary convolutional code are limited only to the values 0 and 1, the code can be described as a finite state machine with 2^m states, m being the number of memory registers. This representation leads to a trellis diagram that can be used at the decoding procedure. The finite state machine can be described by a state diagram as shown in Figure 1.3. This diagram represents the same code of Figure 1.2, where each state is represented by the bits of the first memory and the second, in this order. A dashed line is an input bit 0 and a solid line is an 1. The outputs are represented by the two bits $y_1[n] = x[n] + x[n-1]$ and $y_2[n] = x[n-1] + x[n-2]$, labeled next to the transitions. In this diagram, it is hard to read a sequence of states along the evolution of time. A trellis diagram can represent time better, as it shows both all previous and next states at once. Figure 1.4 shows the same code as a trellis diagram.

One of the advantages of convolutional codes is that one can encode a large amount of

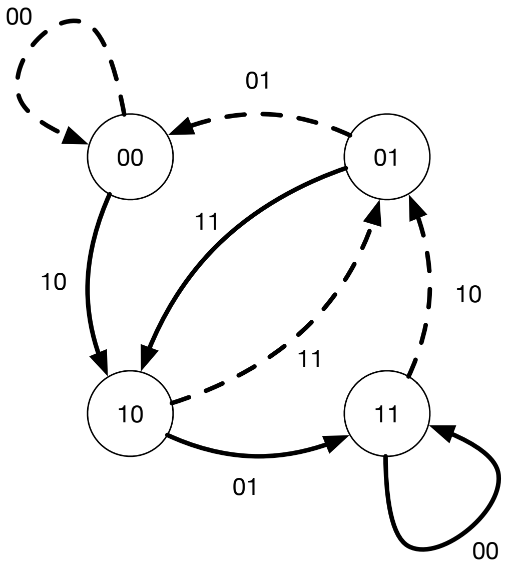


Figure 1.3: State diagram.

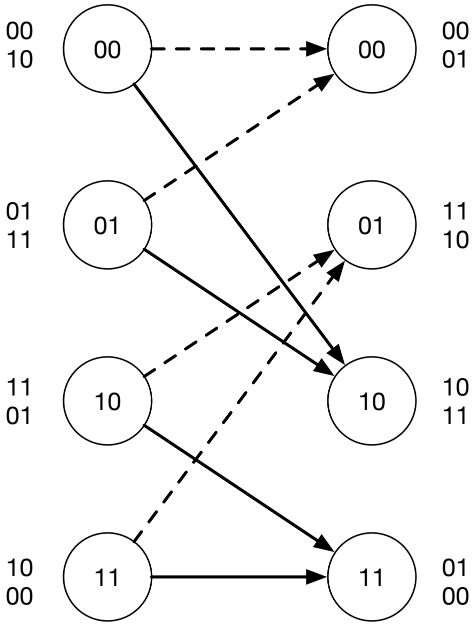


Figure 1.4: Trellis diagram.

data in a stream-like fashion, generating just one big codeword. Furthermore, the decoding can also be performed in the same fashion, which can reduce the latency caused by waiting the full codeword to be received.

1.2 Reed-Solomon codes

Reed-Solomon codes are a powerful class of error correction and detection non-binary codes. The main focus of Reed-Solomon is to have control over the error correction capabilities of the code. It operates on polynomials over finite fields.

In order to generate a Reed-Solomon code, one must define a finite field. A finite field can only have a prime power as its number of elements, so typically 2 is used as the base prime, resulting in fields of order $q = 2^m$. In this work, $m = 8$ is conveniently used, in order to make a field with 256 elements, which we can represent using 8 bits, a byte.

After setting the finite field, the codeword length of the code is $N_{RS} = q-1$ symbols. A symbol can be any element from the field. Now we must set the number of parity symbols M_{RS}. By construction, a Reed-Solomon code can correct up to half the number of parity symbols in a codeword. The number of information symbols is $K_{\mathrm{RS}} = N_{\mathrm{RS}} - M_{\mathrm{RS}}$. For example, we can use $M_{\mathrm{RS}} = 32$ parity symbols resulting in a codeword of $K_{\mathrm{RS}} = 255 - 32 = 223$, so that we have a (255,223) Reed-Solomon code.

A Reed-Solomon code can be viewed as an interpolation problem, as follows: given two different polynomials with K_{RS} coefficients, we evaluate them at the same N_{RS} points. Now these points agree in at most $K_{\mathrm{RS}} - 1$ points. This is because if they agree in K_{RS} points, one could interpolate them and generate the same polynomial, contradicting the premise that they are different. If they agree in at most $K_{\mathrm{RS}} - 1$ points, they disagree in at least $N_{\mathrm{RS}} - (K_{RS} - 1)$, resulting in a minimum distance of $d = N_{\mathrm{RS}} - K_{\mathrm{RS}} + 1$, so that we can correct up to $t = \lfloor \frac{d-1}{2} \rfloor = \lfloor \frac{N_{\mathrm{RS}} - K_{\mathrm{RS}}}{2} \rfloor$ erroneous symbols. In this case, the decoding algorithm must find a way to find the errors such that the interpolation of the values can return the polynomial. The evaluation points usually are taken as consecutive powers of a primitive element α: α^0, α^1, α^2, $\ldots$, because this leads to interesting mathematical properties.

Another view, adopted from BCH codes, is that a Reed-Solomon codeword itself is a polynomial of N_{RS} coefficients. The requirement is that this codeword is divisible by the generator polynomial $G(X)$ of $N_{\mathrm{RS}} - K_{\mathrm{RS}} + 1$ coefficients. The generator polynomial is defined by having consecutive powers of a primitive element α as its roots, as follows

$$G(X) = \prod_{i=1}^{N_{\mathrm{RS}}-K_{\mathrm{RS}}} (X - \alpha^i) = \sum_{i=0}^{N_{\mathrm{RS}}-K_{\mathrm{RS}}} g_i X^i \ , \tag{1.2}$$

where a codeword is any polynomial with N_{RS} coefficients divisible by $G(X)$. In other words, a codewords has at least the same roots of $G(X)$. A systematic encoding algorithm finds a codeword that starts by the given message and calculates the rest by making the codeword divisible by $G(X)$. In order to do this, one should fill the coefficients of the codeword from the highest order coefficients, leaving the lower ones as zeros, then after calculation of the remainder of the polynomial division, we subtract this remainder and thus generate a polynomial divisible by $G(X)$.

(1.4) shows the equation for this method, where $M(X)$ is the message polynomial, with its coefficients conveying the information, $R(X)$ is the parity polynomial, and $C(X)$ is the codeword constructed by the concatenation of $M(X)$ and $-R(X)$.

$$R(X) = M(X)X^{N_{\mathrm{RS}}-K_{\mathrm{RS}}} \bmod G(X) \tag{1.3}$$

$$C(X) = M(X)X^{N_{\mathrm{RS}}-K_{\mathrm{RS}}} - R(X) \tag{1.4}$$

Both views seems very different as in the first we evaluate a polynomial to obtain the values as a codeword, and in the second the codeword is a polynomial itself. If we consider that we evaluate the polynomial of the first view at consecutive powers of α,α^i, $i = 0, 1, 2, \ldots, N_{\mathrm{RS}} - 1$, we can relate both views by the finite field Fourier transform. From the first view, we have a polynomial

$$c(X) = c_0 + c_1 X + c_2 X^2 + \cdots + c_{K_{\mathrm{RS}}-1} X^{K_{\mathrm{RS}}-1} \tag{1.5}$$

and then we evaluate $c(X)$ at α^j, in order to calculate the codeword $C_j = c(\alpha^j)$, resulting in a familiar formula

$$C_j = \sum_{i=0}^{N_{\mathrm{RS}}-1} c_i \alpha^{ij}, \tag{1.6}$$

also, the inverse transform is valid, $c_i = \frac{1}{N_{\mathrm{RS}}} C(\alpha^{-i}) = \frac{1}{N_{\mathrm{RS}}} C(\alpha^{N_{\mathrm{RS}}-i})$, resulting in

$$c_i = \frac{1}{N_{\mathrm{RS}}} \sum_{j=0}^{N_{\mathrm{RS}}-1} C_j \alpha^{-ij} = \frac{1}{N_{\mathrm{RS}}} \sum_{j=0}^{N_{\mathrm{RS}}-1} C_j (\alpha^{N_{\mathrm{RS}}-i})^j, \tag{1.7}$$

where $1/N_{\mathrm{RS}}$ is the reciprocal of $\overbrace{1 + 1 + \cdots + 1}^{N_{\mathrm{RS}}}$, which, in the binary case, equals to 1, and also$\alpha^N_{\mathrm{RS}} = 1$. From (1.5) and (1.7) we can note that $c_i = 0$ for $i = K_{\mathrm{RS}}, K_{\mathrm{RS}} + 1, \ldots, N_{\mathrm{RS}} - 1$, so that $C(\alpha^{N_{\mathrm{RS}}-i}) = 0$ for the same i's, or equivalently, $C(\alpha^k) = 0$ for $k = 1, 2, \ldots, N_{\mathrm{RS}} - K_{\mathrm{RS}}$. This last part indicates that $C(X)$ is also a codeword according to the second view, having consecutive powers of α as its roots.

1.2.1 Finite fields

The definition of a finite field, in a few words, is a kind of “interface” (from programming languages), that is, it is a set that satisfies some requirements (methods or function, in a programming language), so that these requirements are sufficient for us to use other tools. Basically, a finite field is a finite set with two operations, namely addition and multiplication, and both are invertible inside the field. The most notable (non-finite) field would be the rationals $\mathbb{Q}$ representing fractions of integers, on which can be performed the four basic arithmetic operations.

The requirements of a field enable us to use all the mathematical knowledge that already works on sets such as $\mathbb{C}$, $\mathbb{R}$ and $\mathbb{Q}$ on other sets, even sets we define ourselves. For example, algebra tools used for system of equations, polynomials, matrices and vectors can be used on finite fields with minimal or no changes.

Formally a finite field is a set of elements, equipped with two operations, namely "addition" and "multiplication". The quotation marks indicates that they are not the conventional addition and multiplication. They can be any operation, usually defined conveniently to serve a purpose. A finite field requires closure of both operations, in other words, the "addition" and "multiplication" of two elements must result in another element in the set. Also, both operations have a identity element, namely "0" and "1" (note the quotation marks again), where "0" is an element such that $a + 0 = a$ and "1" is an element such that $a \cdot 1 = a$. Also, both operations have the associativity and commutative properties, meaning that the order each operation is carried does not matter. Another required property is that the "multiplication" is distributive over "addition", $a \cdot (b + c) = a\dot{b} + a\dot{c}$. And the last requirement is what differs a field from a commutative ring, which is the presence of inverses for both operations, as a commutative ring requires inverses only for "addition". An inverse of a element for an operation is another element which when applied the operation gives a identity element. For example, for "addition" b is the additive inverse of a when $a + b = 0$, usually we explicitly name b as $b = -a$, such that $a + (-a) = 0$. The same goes for "multiplication", $c \cdot d = 1$, and we express d as $d = \frac{1}{c}$, such that $c \cdot \frac{1}{c} = 1$. An exception for the inverse requirement is "0", which has no multiplicative inverse. The field we usually learn first is the set of rationals $\mathbb{Q}$, where we can divide any number. The reals $\mathbb{R}$ and complexes $\mathbb{C}$ are also fields.

A finite field is a field with a finite number of elements. Considering that the "0" and "1" are different, $0 \neq 1$, the smallest finite field we can construct is using addition and multiplication modulo 2, making a field with 2 elements, 0 and 1. Addition modulo 2 is equivalent to a binary XOR operation, and multiplication modulo 2 is binary AND, as shown in Table 1.1. We can note from the addition table that, for this binary field, addition and

Table 1.1: Addition and multiplication tables.

$+$	0	1
0	0	1
1	1	0

$\times$	0	1
0	0	0
1	0	1

subtraction are equal, as $1 + 1 = 0$.

It is possible to construct any finite field with a prime number of elements using the same method of considering addition and multiplication modulo p, where p is prime. Furthermore, it is possible to construct finite fields with its number of elements equals to a prime power p^n, but using a different method. This method uses primitive polynomials over base prime fields instead of prime integers, such that the field elements are now represented by polynomials. Primitive polynomials are the minimal polynomials of primitive elements. Primitive elements are generators of the field multiplicative group, which is cyclic (excluding 0). In other words, if we take one root of a primitive polynomial and apply consecutive powers to it, we obtain the entire multiplicative group, generating the entire field.

As an example, to generate the finite field of 4 elements, we use polynomials with two coefficients from the binary finite field. The primitive polynomial is given as $X^2 + X + 1$, and the elements of the field are calculated modulo this polynomial. This results in four possible polynomials, with degree up to 1. The polynomials are 0, 1, X and $X + 1$. Considering α as one of the roots of $X^2 + X + 1$, then

$$\alpha^2 + \alpha + 1 = 0 \tag{1.8}$$

$$\alpha^2 = \alpha + 1 \tag{1.9}$$

and the field is $\{0, 1, \alpha, \alpha^2\}$, or equivalently, $\{0, 1, \alpha, \alpha + 1\}$, or in binary form $\{00, 01, 10, 11\}$. One can note that α can be both X or $X + 1$, but this does not matter, the field operations will behave the same. We showed three different views of this field, resumed in Table 1.2

To perform addition, we just use the polynomial view and add coefficients of same degree. This is also equivalent to a XOR operation between their vector representations. In the other side, for multiplication, we use the power representation and just add the exponents modulo 3 (the exponents can be only 0, 1 or 2, in this example). This said, Table 1.3 shows both addition and multiplication tables, proving that this construction is indeed a field, as this set is closed under both operations, and each line and column has the respective identity element, indicating the presence of inverses.

The same procedures can be used to construct a field with more elements, just by

Table 1.2: Three different views of GF(4).

Index	Power	Polynomial	Vector
0		0	00
1	α^0	1	01
2	α^1	α	10
3	α^2	$\alpha + 1$	11

Table 1.3: Addition and multiplication tables for GF(4).

$+$	0	1	α	α^2
0	0	1	α	α^2
1	1	0	α^2	α
α	α	α^2	0	1
α^2	α^2	α	1	0

$\times$	0	1	α	α^2
0	0	0	0	0
1	0	1	α	α^2
α	0	α	α^2	1
α^2	0	α^2	1	α

selecting a higher degree primitive polynomial.

1.2.2 Encoding

Reed-Solomon [2] encoding can be performed in numerous ways. Using the interpolation view, we assume that the information to be encoded is a polynomial and evaluate this polynomial of degree up to $K_{\text{RS}} - 1$ at N_{RS} different points. The evaluation points usually are taken at consecutive powers $\{\alpha^1, \alpha^2, \dots\}$ of a primitive element α as this leads to interesting mathematical properties. This leads to a non-systematical encoding, where the information is not directly available on the codeword.

A way to encode systematically, using the same interpolation view, is to directly set the first values of the codeword as the information to be encoded, in other words, forcing the codeword to be systematic. To calculate the remaining parity values, we interpolate the K_{RS} information symbols in order to make a polynomial and evaluate this polynomial at the remaining $N_{\text{RS}} - K_{\text{RS}}$ positions. This method implies more cost due to the interpolation, but on the other hand it results in a systematic encoding procedure.

Using the BCH [11] view, we have a generator polynomial $G(X)$, and a codeword is any polynomial with at least the same roots of $G(X)$. By this definition a simple encoding method is to do

$$C(X) = M(X)G(X), \tag{1.10}$$

where $C(X)$ is guaranteed to be a multiple of $G(X)$. Again, this does not yield a systematic encoding.

For a systematic encoding using the BCH view, one must force the first coefficients of $C(X)$ to be the information symbols by doing $M(X)X^{(N_{\mathrm{RS}}-K_{\mathrm{RS}})}$, this leaves the coefficients of smaller degree set to 0. Now, to make a codeword, $C(X)$ must be a multiple of $G(X)$, so we have to subtract the amount it needs to be a multiple, thus we subtract the remainder of the polynomial division of $M(X)X^{(N_{\mathrm{RS}}-K_{\mathrm{RS}})}$ by $G(X)$. As a result we have that $C(X) = M(X)X^{(N_{\mathrm{RS}}-K_{\mathrm{RS}})} - (M(X)X^{(N_{\mathrm{RS}}-K_{\mathrm{RS}})} \bmod G(X))$. One can make an analogy to integers, for example, if we want a multiple of 7, we could take any number, say 52, and then take $52 \bmod 7 = 3$, then we just subtract $52 - 3 = 49$, which is a multiple of 7. Note that this is the very definition of remainder of division.

1.2.3 Decoding

Reed-Solomon decoding, ideally, could be made using interpolation, as one can interpolate all the combinations of the received N_{RS} values, taken K_{RS} at a time, and then create a ranking counting which polynomial appears more frequently, this would be the decoded polynomial. While the idea is simple, it is costly due to the combinatorics explosion even on smaller codeword lengths.

In order to decode efficiently, a syndrome decoding method was devised. This greatly reduces the cost, as we work only with syndromes, ignoring the codeword entirely. The syndrome sizes are as big as the redundancy sizes, so usually their lengths are only small fractions of the codeword lengths.

Assuming $T(X)$ is the transmitted polynomial, $R(X)$ is the received polynomial, and $E(X)$ is the error polynomial, such that

$$R(X) = T(X) + E(X), \tag{1.11}$$

we have that $T(X)$ is codeword, thus $T(X)$ have at least the same roots as the generator polynomial $G(X)$, resulting that $T(\alpha^i) = 0$ for $i = 1, 2, \ldots, N_{\mathrm{RS}} - K_{\mathrm{RS}}$. Knowing this, we have that

$$S_i = R(\alpha^i) = T(\alpha^i) + E(\alpha^i) \text{ for } i = 1, 2, N_{\mathrm{RS}} - K_{\mathrm{RS}} \tag{1.12}$$

$$S_i = R(\alpha^i) = E(\alpha^i) \tag{1.13}$$

where S_i represents the syndrome for each value of i. Now, considering polynomial $E(X)$ having only t errors, we have

$$S_j = E(\alpha^j) = \sum_{k=1}^{t} e_{i_k} (\alpha^j)^{i_k}, \tag{1.14}$$

where i_k is a index at $E(X)$ where coefficient e_{i_k} is non-zero. Considering $Z_k = \alpha^{i_k}$ as the error locations and $Y_k = e_{i_k}$ as the error values, we have $S_j = \sum\limits_{k=1}^{t} Y_k Z_k^j$, which can be seen as a system of equations:

$$\begin{bmatrix} Z_1 & Z_2 & \dots & Z_t \\ Z_1^2 & Z_2^2 & \dots & Z_t^2 \\ \vdots & \vdots & \ddots & \vdots \\ Z_1^{n-k} & Z_2^{n-k} & \dots & Z_t^{n-k} \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_t \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_{n-k} \end{bmatrix}. \tag{1.15}$$

The main decoding problem is to find the Z_k, corresponding to the leftmost matrix. Note that this is a overdetermined system, which can lead to multiple solutions. The true solution is the one which minimizes t, in other words, which assume a minimal number of errors.

Finding the error locations Z_k are by far the most complex part of decoding. This can be done by using the Berlekamp-Massey [12,13] algorithm. This algorithm is used to calculate the error locator polynomial, which is defined as polynomial $\Lambda(X)$ which has the reciprocals Z_k^{-1} as its roots:

$$\Lambda(X) = \prod_{k=1}^{t} (1 - XZ^k), \tag{1.16}$$

and $\Lambda(Z_k^{-1}) = 0$. Using the algorithm, we find $\Lambda(X)$, and then we can calculate its roots by exhaustive search, which is feasible as it is a polynomial over a finite field. This exhaustive search is done efficiently by an algorithm called Chien search. With Z_k calculated one can finally calculate Y_k by solving (1.15). Again, there is a efficient algorithm called Forney algorithm which calculates Y_k. With both Z_k and Y_k, we construct $E(X)$ and recover $T(X)$ by just subtracting $T(X) = R(X) - E(X)$.

1.3 Trellis-coded modulation

Trellis-coded modulation is a technique that mixes both a convolutional code and a modulation scheme. The key concept of this technique is to replace conventional Hamming distances between sequences of bits by Euclidean distances between sequences of constellation points. A typical trellis-coded modulation is done by using a convolutional code of rate $\frac{k}{k+1}$.

This can be viewed as expanding the modulation order by one bit, for example, if a code with rate 2/3 is used with PAM modulation, then it means that a 4-PAM constellation is expanded to a 8-PAM constellation. Considering an uncoded modulation, this expansion would reduce the distance between any two neighboring points, assuming the same transmission power. The advantage of coded modulation is that we do not consider the neighborhood anymore as the code design ensures that the only possible points are farther than the closest neighbors, thus increasing the distance.

In practice, trellis-coded modulation uses uncoded and coded bits. For example, using a 256-QAM modulation, each symbol contains 8 bits. Using a hierarchical constellation mapping, we could assign different importances to each bit. In other words, it means that some bits are more susceptible to noise than others. As an example if a bit is the only difference between two neighbor points, then this bit is very fragile to noise, as opposed to the situation where a bit defines the quadrant or the sign of a constellation point. The strategy used by trellis-coded modulation is to protect fragile bits using convolutional codes, and leave more robust bits uncoded. This leads to a very efficient scheme where we can produce coding gains by sacrificing just one bit as redundancy. For a concrete example, Chapter 2 has a detailed implementation of a trellis-coded modulation scheme.

1.3.1 Encoding/Mapping

The encoding of trellis-coded modulation is pretty straightforward: we just feed the input bits to the convolutional encoder to generate the output bits, then these bits are mapped to the respective points in the constellation. These steps have no difference from a non-coded modulation scheme, as the difference lies in the fact that the design of the code is fully aware of the mapping of the constellation points, thus taking in account which output bit pattern generates each point. The design is made by maximizing the (Hamming) distance between sequences of bits, and then each bit pattern is mapped accordingly into the constellation in order to maximize the (Euclidean) distance between points, in other words a good convolutional code may not have the same performance when used for trellis-coded modulation, because it depends on the modulation scheme used.

1.3.2 Decoding/Demapping

The decoding of trellis-coded modulation is basically the same as the decoding of a convolutional code, but with distance metrics being distances between constellation points as opposed to distance between bit sequences. The optimal algorithm in the sense of maximum likelihood decoding is well-known as the Viterbi algorithm. The Viterbi algorithm finds the best sequence of constellation points that minimizes the overall Euclidean distance checking all possible sequences in a clever way that greatly reduces the complexity. The main decoding scheme is made in a similar way as of a conventional convolutional code, which is composed by three parts. The first one is the branch metric unit, which, in the trellis-coded modulation, calculates the branch metrics for each output pattern by looking at the constellation and returning the Euclidean distance metric. Then, the second part is the path metric unit, which uses the Viterbi algorithm to determine the sequence of states and branches that minimize the path total Euclidean distance. Finally, the last part is the traceback unit, which just follow the trellis backwards from the last state, backtracking the path with minimum distance and returning the input that generated this path. The input is then the decoded message.

1.4 Low-Density Parity-Check codes

Low-Density Parity-Check (LDPC) [4, 14, 15] codes are iterative block codes, used as an error correcting code. LDPC codes are defined by a parity check matrix, which represents a bipartite graph. The main characteristic is that the parity check matrix has to be sparse, that is, it contains a low number of non-zero values.

As a linear code, an LDPC code has message length k and codeword length n, defining a (n, k) LDPC code. The parity-check matrix H is a $(n-k) \times n$ sparse matrix. A codeword is any vector $\hat{v}$ that satisfies $\hat{v}H^T = 0$.

The design of a LDPC code is generally done using random or pseudo-random techniques to generate H.

1.4.1 Encoding

LDPC codes can be encoded, in general, as a linear code, multiplying the message vector by a generator matrix. To find a systematic encoding procedure, one can apply Gaussian elimination on H in order to obtain an identity matrix at the right side of H. With H in the form $[\ P \quad I_{n-k}\ ]$, we find generator matrix G using $G = [\ I_k \quad -P^T\ ]$. Although this method works for all full rank matrices, the storage of G, and the multiplication $\hat{u}G$ are problems as G is not guaranteed to be sparse.

Another method, proposed by Richardson-Urbanke [16], tries to reorganize the matrix in a way to produce a lower triangular matrix at the right side of H using only row and column swaps (no additions between rows). This method achieve almost linear encoding complexity with respect to n.

1.4.2 Decoding

Decoding is the difference between classical codes and iterative codes. LDPC decoding is done by receiving information from the channel and improving this information iteratively, until convergence to a valid codeword.

Decoding represents H with a Tanner graph, where each row is check-node, and each column is a bit-node. Initial information from the channel enters the bit-nodes and flows through the connections of the graph to the check-nodes. Then the check node calculates new messages that flow back over the connections to the bit-nodes. The bit-nodes gather these messages and produces new messages again. This process is done until the messages converge to a valid codeword or the number of iterations is exceeded. This algorithm is called message passing, and it has a number of variants.

The most popular variant is the sum-product algorithm, which uses a probabilistic setting to exchange information between nodes. The information is represented by probabilities instead of bits, so that the decoder works on soft information, which provides much more information on the channel than hard information. The sum-product algorithm has a variant which reduces complexity by sacrificing some error correction performance called min-sum algorithm. Also, when dealing with hard information, one can use the bit-flip algorithm variant.

Chapter 2

Standards Overview

This chapter provides a more detailed view of each code's implementation on this work based on their respective standards.

2.1 G.fast forward error correction

The G.fast standard establishes new directives for broadband transmission on copper wires, allowing for connections speeds of up to 1Gbps. The FEC scheme for G.fast was chosen to be a combination of Reed-Solomon and trellis-coded modulation, which is a similar scheme to that of previous DSL standards [17].

2.1.1 Reed-Solomon

G.fast uses a Reed-Solomon code over GF(256). The codeword size N_{RS} has a maximum value of 255 bytes, which represents the code without shortening. With the use of shortening, the minimum N_{RS} is 32 bytes. The redundancy size R_{RS} can assume any even values from 2 to 16. The standard uses the primitive binary polynomial $x^8 + x^4 + x^3 + x^2 + 1$ to perform the arithmetic operations on GF(256).

2.1.1.1 Encoding

The encoding procedure seeks to implement

$$C(X) = M(X)X^{R_{RS}} \bmod G(X), \tag{2.1}$$

where $G(X)$ is the generator polynomial, $M(X)$ is the message polynomial and $C(X)$ is the check polynomial.

The multiplication by $X^{R_{RS}}$ is used only to 'make room' for the R_{RS} redundancy symbols. As any codeword is divisible by $G(X)$, we take $M(X)X^{R_{RS}}$ and calculate the remainder of the polynomial division $C(X)$. Then $M(X)X^{R_{RS}} + C(X)$ is divisible by $G(X)$, which guarantees that it is also a codeword. Note that the codeword is just a concatenation of $M(X)$ and $C(X)$, resulting in a systematic encoding procedure. Systematic codes are preferred as, after the syndrome calculation at the decoder, if there are no errors, one should just drop $C(X)$ to recover the transmitted message.

The implementation basically is focused on the calculation of the polynomial division remainder. This can be done by using a digital filter over GF(256) in direct form II transposed, on which the remainder will be located at the filter memory. The division $B(X)/A(X)$ can be calculated by finding the impulse response of a digital filter $H(X) = B(X)/A(X)$. In this implementation, we exchange the positions of the impulse and the numerator, so we use $H(X) = 1/A(X)$ and the input are the coefficients of $B(X)$. Also, each polynomial is read from the greatest order coefficient, so that the trailing zeros of $B(X)$ can be ignored and $A(X)$ first coefficient is 1. This can be done considering a temporary change of variables like $X = Y^{-1}$. The resulting encoding filter is shown in Figure 2.1. The message polynomial is the input $x[n]$, the generator polynomial coefficients are multipliers, and the result is found at the memories of the delay line represented by the square blocks. Also, normally, the coefficients of $G(X)$ should have changed their signs, but for finite fields with characteristic 2 we know that $a = -a$ for any a.

2.1.1.2 Decoding

The decoding procedure has, by far, the most computational cost. It is the decoder that has the intelligence of the code. The decoder has to decide whether errors are present or not, and, if there are errors, decide again whether the codeword can be corrected or not. If the number of errors exceeds the error correction capabilities of the code, then the decoder does not try to correct anything and return an error code.

The decoder is implemented by the following main parts: syndrome calculation, Berlekamp-Massey algorithm, Chien search, error evaluator polynomial and error correction, as shown in Figure 2.2.

The decoding algorithm starts with syndrome calculation, which simply verifies if the received polynomial is divisible by the generator polynomial $G(X)$. This is done by evaluating

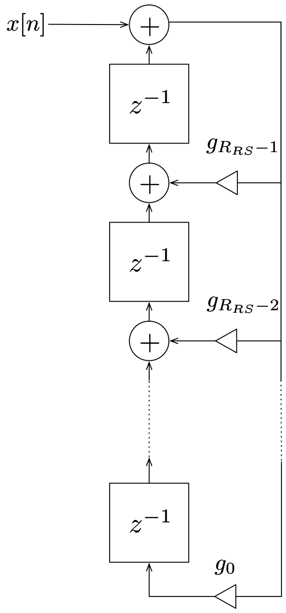


Figure 2.1: Reed-Solomon Encoding Filter.

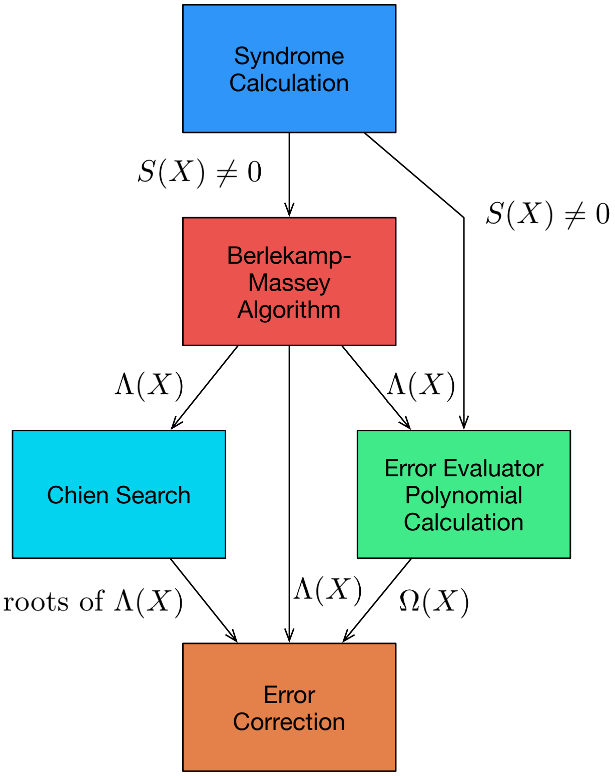


Figure 2.2: Decoder blocks.

the received polynomial at each root of $G(X)$. $G(X)$ is defined in a way that its roots are consecutive powers of the primitive element α of the Galois field. All the polynomial evaluations are made at the same time, using previous results in the calculations of the new ones, reducing the number of calculations in a fashion similar to Chien search. As the result we have the syndrome polynomial $S(X)$, on which each coefficient is the value obtained for each root. If all coefficients are zero, $S(X) = 0$, then the received polynomial is a valid codeword and the algorithm stops, returning it as the decoded codeword. If $S(X) \neq 0$, then at least one coefficient is non-zero, which means that at least an error occurred somewhere.

After knowing that at least an error exists, we must determine three things: the quantity of errors, the location of errors and the value of errors. The Berlekamp-Massey algorithm calculates the error locator polynomial $\Lambda(X)$ which has as its roots the reciprocal of the errors locations, so that finding the roots of the polynomial results in finding the error locations.

To solve the $\Lambda(X)$ polynomial, we must find all possible roots. This is accomplished by testing all possible values and looking for the ones that results in 0. This testing procedure can be implemented using Chien search to reduce the number of calculations. Chien search then returns the locations of errors.

As this code is not binary, one also needs to calculate the values of the errors, which is done by firstly calculating the error evaluator polynomial $\Omega(X)$. $\Omega(X)$ is calculated by $\Omega(X) = S(X)\Lambda(X) \bmod X^{R_{RS}}$, which means that each coefficient from $\Omega(X)$ is obtained by convolution of the coefficients of $\Lambda(X)$ and $S(X)$, up to the $(R_{RS} - 1)$-th degree coefficient.

Having access to $\Lambda(X)$, to its roots and to $\Omega(X)$, we can use Forney algorithm to find the values of the errors using

$$e_j = \frac{{r_j}^{-1}\Omega(r_j)}{\Lambda'(r_j)} \tag{2.2}$$

where e_j is error value, r_j is the root associated to e_j and $\Lambda'(X)$ is the formal derivative of $\Lambda(X)$. The formal derivative in a field of characteristic 2 is quite simple, as the coefficients λ'_i are determined by:

$$\lambda'_i = \begin{cases} \lambda_{i+1} & \text{if } i \text{ is even} \\ 0 & \text{if } i \text{ is odd} \end{cases} \tag{2.3}$$

then, finally, to apply the correction we just add each error value e_j on its respective location ${r_j}^{-1}$. This completes the decoding process returning a corrected message as its result.

2.1.1.3 Shortening

The shortening procedure is used to reduce the fixed codeword size of 255 by inserting zeros as information bytes. For N_{RS} bytes we have an amount of zeros $A_0 = 255 - N_{RS}$. The redundancy size R_{RS} is not modified, so effectively the code rate drops as we are reducing the information part. The shortening process sizes are shown in Figure 2.3. The zeros part is not transmitted, as the receiver already know that this part is zeros. The main advantage of shortening is that we can change the code rate using exactly the same Reed-Solomon code.

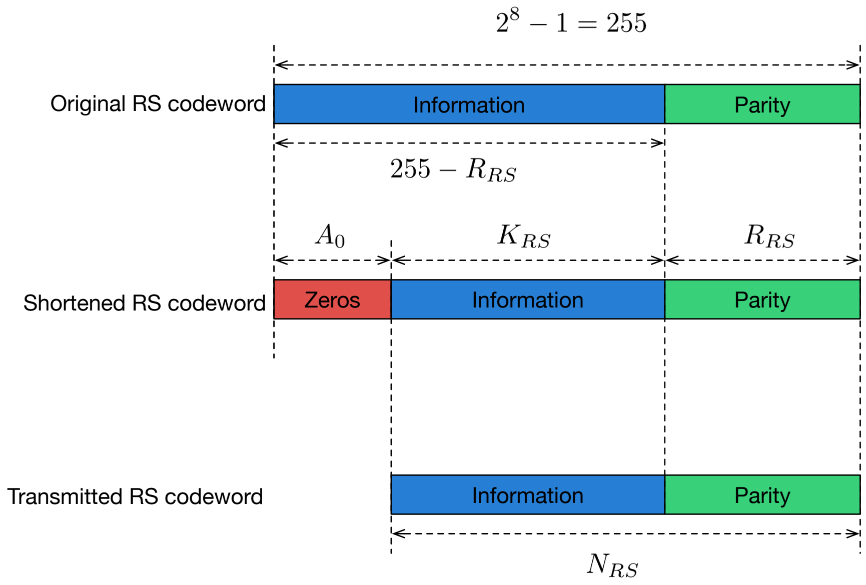


Figure 2.3: Shortening process.

Figure 2.4 shows each step of shortening. At first, we receive K_{RS} bytes to be encoded. As the Reed-Solomon encoder uses $255 - R_{RS}$ bytes, we have to fill the difference with zeros. The encoding process then calculates R_{RS} parity bytes to form a full Reed-Solomon codeword with 255 bytes. At the last step, we remove the zeros to transmit only relevant information.

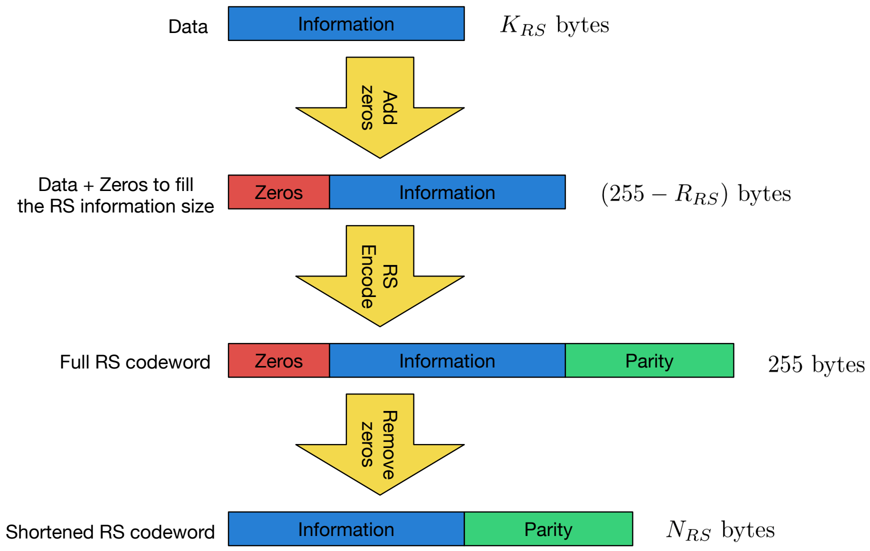


Figure 2.4: Each step of Reed-Solomon encoding.

2.1.1.4 Unshortening

Unshortening does the reverse process of shortening, adding the zeros to reconstruct a full codeword. This procedure turns shortening invisible to the decoder, which interprets the zeros as information as well. The steps are shown in Figure 2.5.

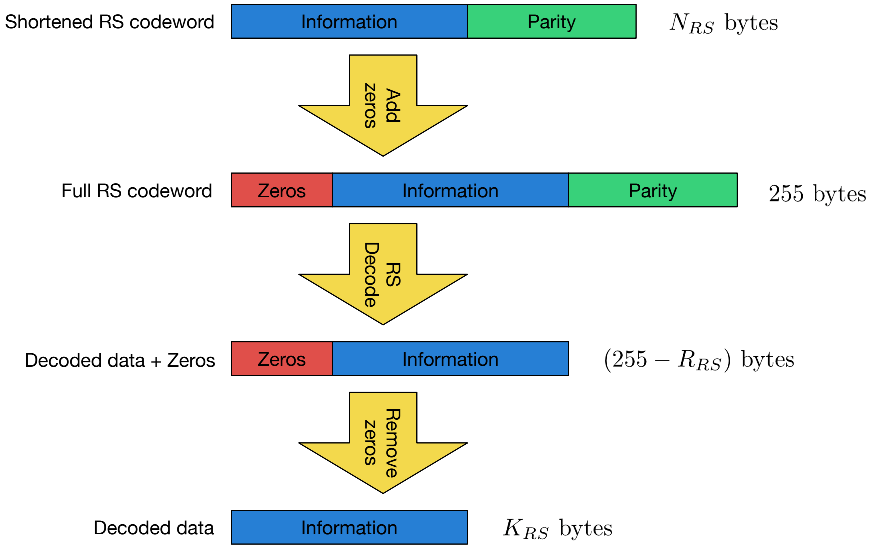


Figure 2.5: Each step of Reed-Solomon decoding.

2.1.2 Interleaving

Interleaving is used to control error bursts. Error bursts are a sequence of errors occurring next to each other that easily overwhelm the error correcting capacity of the Reed-Solomon code. One of the main sources of error bursts is the trellis-coded modulation, given that the Viterbi decoder often produce this kind of errors. Interleaving is done by shuffling data in a way that bytes located near to each other before shuffling are located in different Reed-Solomon codewords after shuffling. The effect is that the errors are spread across several codewords, decreasing the probability of decoding failure on the Reed-Solomon decoder.

G.fast uses a block interleaver which is implemented using a matrix. This matrix has D rows, representing the codewords, and N_{RS} columns, representing the bytes of each codeword. D is called interleaver depth and $D = 1$ means no interleaving.

The interleaver operates writing the input row-wise and reading the output column-wise in the matrix. If we use indexes i from 0 to $D - 1$ and j from 0 to $N_{RS} - 1$, we can calculate the output position l using

$$l = j \times D + i, \tag{2.4}$$

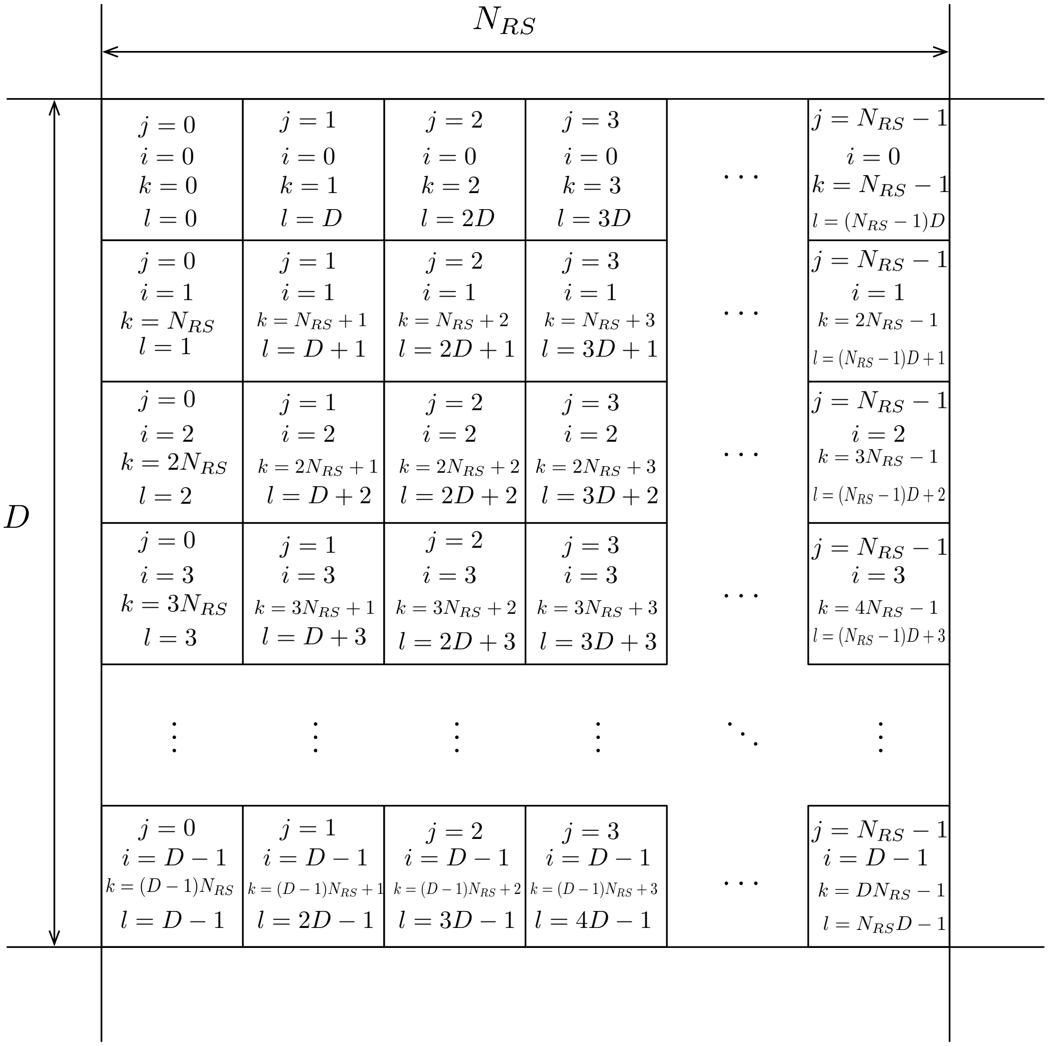
where l vary from 0 to $N_{RS} \times D - 1$. i and j can be calculated from the input position k using

$$i = k/N_{RS}, \tag{2.5}$$

where / is integer division, and

$$j = k \bmod N_{RS}, \tag{2.6}$$

such that $k = i \times N_RS + j$. Figure 2.6 shows the interleaving matrix with values for i , j , k and l. We can easily note that k increases following a row-wise fashion and, on the other hand l increases following a column-wise fashion.



$j=0$ $i=0$ $k=0$ $l=0$	$j=1$ $i=0$ $k=1$ $l=D$	$j=2$ $i=0$ $k=2$ $l=2D$	$j=3$ $i=0$ $k=3$ $l=3D$	$\cdots$	$j=N_{RS}-1$ $i=0$ $k=N_{RS}-1$ $l=(N_{RS}-1)D$
$j=0$ $i=1$ $k=N_{RS}$ $l=1$	$j=1$ $i=1$ $k=N_{RS}+1$ $l=D+1$	$j=2$ $i=1$ $k=N_{RS}+2$ $l=2D+1$	$j=3$ $i=1$ $k=N_{RS}+3$ $l=3D+1$	$\cdots$	$j=N_{RS}-1$ $i=1$ $k=2N_{RS}-1$ $l=(N_{RS}-1)D+1$
$j=0$ $i=2$ $k=2N_{RS}$ $l=2$	$j=1$ $i=2$ $k=2N_{RS}+1$ $l=D+2$	$j=2$ $i=2$ $k=2N_{RS}+2$ $l=2D+2$	$j=3$ $i=2$ $k=2N_{RS}+3$ $l=3D+2$	$\cdots$	$j=N_{RS}-1$ $i=2$ $k=3N_{RS}-1$ $l=(N_{RS}-1)D+2$
$j=0$ $i=3$ $k=3N_{RS}$ $l=3$	$j=1$ $i=3$ $k=3N_{RS}+1$ $l=D+3$	$j=2$ $i=3$ $k=3N_{RS}+2$ $l=2D+3$	$j=3$ $i=3$ $k=3N_{RS}+3$ $l=3D+3$	$\cdots$	$j=N_{RS}-1$ $i=3$ $k=4N_{RS}-1$ $l=(N_{RS}-1)D+3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$j=0$ $i=D-1$ $k=(D-1)N_{RS}$ $l=D-1$	$j=1$ $i=D-1$ $k=(D-1)N_{RS}+1$ $l=2D-1$	$j=2$ $i=D-1$ $k=(D-1)N_{RS}+2$ $l=3D-1$	$j=3$ $i=D-1$ $k=(D-1)N_{RS}+3$ $l=4D-1$	$\cdots$	$j=N_{RS}-1$ $i=D-1$ $k=DN_{RS}-1$ $l=N_{RS}D-1$

Figure 2.6: Interleaver matrix and the values for each index.

2.1.2.1 Deinterleaving

Deinterleaving just does the inverse process of interleaving using the same matrix, but this time we write the input bytes column-wise and read them row-wise. The output position l can be calculated into a similar way as the interleaver, using indexes i and j:

$$l = j \times N_{RS} + i \tag{2.7}$$

and also, i and j can be calculated from the input index k using

$$i = k/D, \tag{2.8}$$

where / is integer division, and

$$j = k \bmod D \tag{2.9}$$

such that $k = i \times D + j$.

2.1.3 Trellis-coded modulation

G.fast trellis-coded modulation is the same as previous DSL standards, which use a Wei's 16-state 4-dimensional trellis code.

2.1.3.1 Bit extraction

Bit extraction is the way we take bits in order to apply trellis-coded modulation given a bit allocation table (bitload). In this work, we used only modulations with an even number of bits, so that the bitload is limited to even values. This is because these constellations mapping algorithms are easier. G.fast uses a 4-dimensional trellis code, which in turn uses 2 2-dimensional (QAM) constellations, meaning that the trellis code sees two sub-carriers as one. This has the effect that the bitload is read in pairs formed by consecutive entries in the bitload.

In order to fit the bitload exactly, for each sub-carrier pair (x, y), we extract $z = x+y-1$ bits. The 1 bit difference accounts for the redundancy bit introduced later by the encoder. For the last two pairs, only $z = x + y - 3$ bits are extracted. These 2 extra bits are used to force the encoder to return to its initial all-zero state.

The z bits form a binary word $\hat{u}$, which can be represented in two forms, according to the two cases above: if the pair is not one of the last two pairs, then $\hat{u} = \{d_1, d_2, \ldots, d_z\}$. If it is one of the last two, then $\hat{u} = \{b_1, b_2, d_1, d_2, \ldots, d_z\}$, where d_k are the extracted data bits, and b_1 and b_2 are the extra bits mentioned above. Note that the length (len) of $\hat{u}$ changes in each case, being $\text{len}(\hat{u}) = z$ in the first case and $\text{len}(\hat{u}) = z + 2$ in the second case.

There is a special case when the bitload has a odd number of entries. In this case we insert a dummy entry with value 0 to make the number of entries even. Also, $\hat{u} = \{0, d_0, 0, d_1, d_2, \ldots, d_z - 1\}$ to fill the dummy sub-carrier with zeros.

2.1.3.2 Encoding

G.fast uses a convolutional code described in the finite state machine of Figure 2.7. This representation is useful for the encoding operations, as we can just implement the state machine with XOR and AND operations on the input bits.

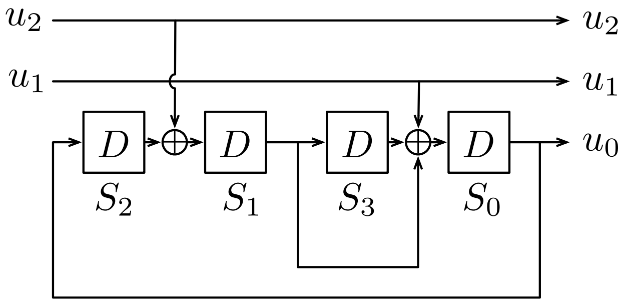


Figure 2.7: Finite state machine form of the trellis code.

The rate 2/3 systematic recursive convolutional code takes 2 inputs u_1 and u_2 and generates a parity bit u_0 based on the inputs and its internal memory. As the code is systematic, the input bits are repeated at the output as well.

The encoder receives the binary word $\hat{u}$, adds 1 bit at the beginning (u_0) and calculates two binary words $\hat{v}$ and $\hat{w}$, with lengths x and y, respectively. To calculate u_0, we take only the first two bits u_1 and u_2 and input them on the convolutional encoder. Then, to calculate $\hat{v}$ and $\hat{w}$, we take the three bits from the output of the convolutional encoder plus an extra uncoded bit u_3 and input them to the bit converter, which produces four bits (v_0, v_1) and (w_0, w_1). These are the first two bits for each binary word, the other bits are uncoded bits taken directly from $\hat{u}$. The bits from v_2 to $v_{\text{len}(\hat{u})-y}$ are taken from u_4 to $u_{\text{len}(\hat{u})-y+2}$. The bits from w_2 to w_{y-1} are taken from $u_{\text{len}(\hat{u})-y+3}$ to $u_{\text{len}(\hat{u})}$. These relations are shown in Figure 2.8.

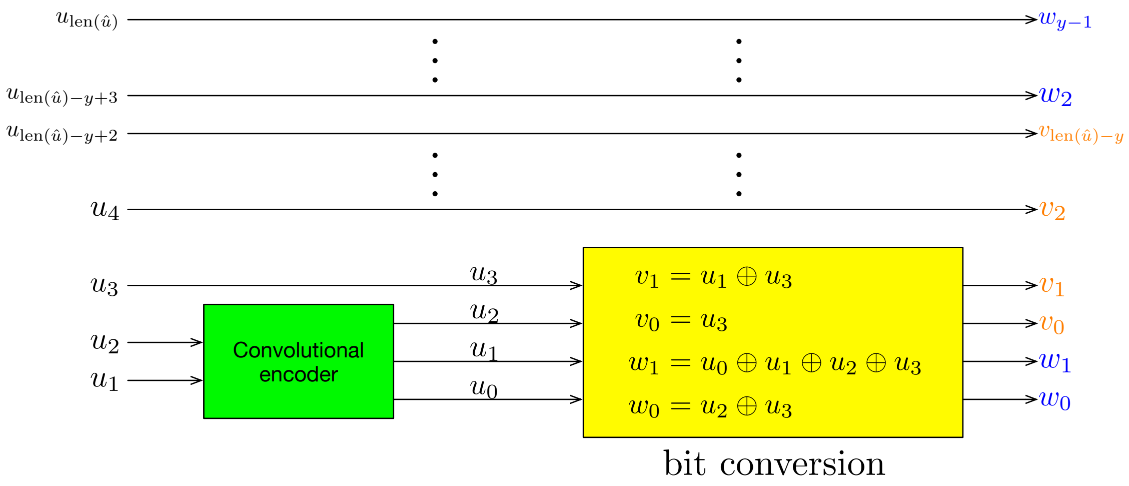


Figure 2.8: Relation of $\hat{u}$ with $\hat{v}$ and $\hat{w}$.

2.1.3.3 QAM modulation

Although QAM modulation is not an error correction code, we had to implement it in order to have a baseline system. G.fast QAM design has trellis-coded modulation taken in consideration, so that it is not Gray-coded, for example. In other words, these constellations are best used with trellis-coded modulation, and have suboptimal bit error rates otherwise.

The modulation algorithm is simple for even number of bits $b = \log_2(M)$, where M is the modulation order. To calculate integer coordinates x and y one should just assign bits alternately to x and y. The MSBs are interpreted as the sign bit, and the other bits are interpreted assuming a two's complement form. Assuming $\hat{v} = \{v_{b-1}, v_{b-2}, \ldots, v_0\}$ are the bits to be modulated, $\hat{x} = \{v_{b-1}, v_{b-3}, \ldots, v_1, 1\}$ and $\hat{y} = \{v_{b-2}, v_{b-4}, \ldots, v_0, 1\}$ are the binary representations in two's complement form of x and y, respectively. For example, if we are considering 16-QAM, then $b = 4$ bits, supposing these 4 bits are $\hat{v} = \{1, 0, 1, 1\}$, we have that $\hat{x} = \{1, 1, 1\} = -1$ and $\hat{y} = \{0, 1, 1\} = 3$, forming the point $(-1, 3)$. Figure 2.9 shows an example for $b = 8$.

This algorithm makes the two LSBs of each point form 4 grids with minimal distance of 4 instead of 2 if we are measuring distances only from the same grid. Figure 2.10 shows these grids. Each grid is called a 2-dimensional coset. These 2D cosets are controlled by the two LSBs, which in turn, are controlled by the bit conversion, according to Figure 2.8, as both pairs (v_0, v_1) and (w_0, w_1) are generated there.

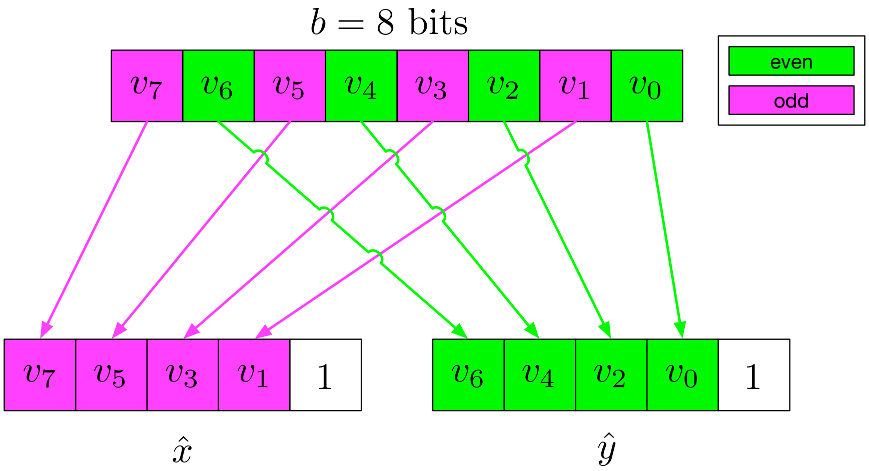
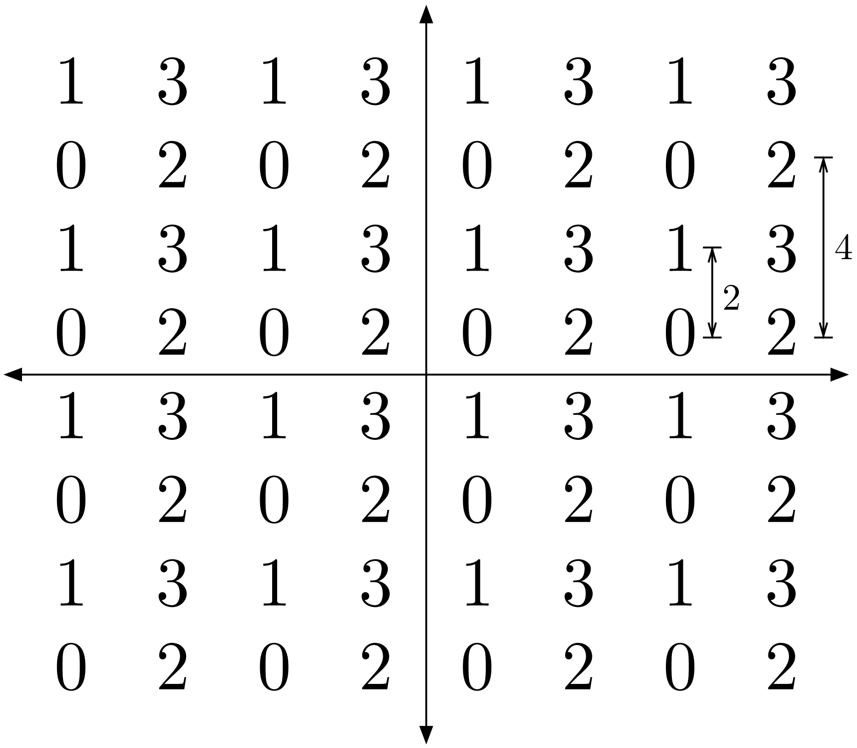

$b = 8$ bits
v_7 v_6 v_5 v_4 v_3 v_2 v_1 v_0
even
odd
v_7 v_5 v_3 v_1 1
v_6 v_4 v_2 v_0 1
$\hat{x}$
$\hat{y}$

Figure 2.9: Modulation algorithm.



1	3	1	3	1	3	1	3
0	2	0	2	0	2	0	2
1	3	1	3	1	3	1	3
0	2	0	2	0	2	0	2
1	3	1	3	1	3	1	3
0	2	0	2	0	2	0	2
1	3	1	3	1	3	1	3
0	2	0	2	0	2	0	2

Figure 2.10: 2-dimensional cosets in a QAM constellation.

2.1.3.4 QAM demodulation

QAM demodulation is made differently, as the constellations are trellis-coded. For each received coordinate, we calculate the distance for each 2D coset, resulting in 4 distances. For each pair combination of 2D coset we calculate the 4D cosets distance, resulting in 16 distances. As we have 8 4D cosets, we choose only the smaller distance of the two for each coset. This choice occurs because each 4D coset is formed by the union of two Cartesian product between two 2D cosets. For example, $C^0_{\text{4D}} = (C^0_{\text{2D}} \times C^0_{\text{2D}}) \cup (C^3_{\text{2D}} \times C^3_{\text{2D}})$, which means that the 4D coset with index 0 C^0_{4D} is formed by the union two Cartesian products. The first is between the 2D coset of index 0 for the first subcarrier with the 2D coset of index 0 for the second subcarrier. The second product is between the 2D coset of index 3 for the first subcarrier and the 2D coset of index 3 for the second subcarrier. To calculate the distance metric for this 4D coset, we have to do the following. Calculate the 2D distance from the received coordinates of the first subcarrier to the nearest points that belongs to cosets 0 and 3. The same is done for the second subcarrier coordinates. Then, to calculate the 4D distances we sum the 2D distances from coset 0 for both subcarriers, and the same for coset 3. Finally, we choose the smaller 4D distance as the candidate. Note that the distances are squared euclidean distances, which allows us to sum 2D distances to obtain 4D distances. Also, we should mention that the Cartesian product corresponds to a sum of distances, while the union corresponds to a choice for the minimum. We must pair each distance with its respective bits. These bits are obtained by reversing the process described in Figure 2.9. This is basically "shuffling" the bits from the binary representation of the coordinates, putting the bits of x coordinate in the odd positions and bits of y in the even positions.

Given that a received point has coordinates $R = (x_r, y_r)$, then to calculate the 2D squared euclidean distances to each 2D coset we just have to calculate the distance for the 4 nearest points, as shown in Figure 2.11. (x_k, y_k) is a point in 2D coset k and d_k is the 2D squared distance to 2D coset k, with $k = 0, 1, 2, 3$. To calculate the 4D metrics, we need a pair of received points, thus the superscript $< i >$ indicates which element of the pair we are taking, 1 or 2. This generates 8 different distances, 4 for the first point, 4 for the second. Then we have to add distances from the first point with distance from the second, resulting in a 4D distance metric. These 4D metrics, with their respective possible candidate points, are passed to the Viterbi path metric unit.

The 4D cosets are formed according to the convolutional code, summarized in Figure 2.12.

From the above, we can summarize the demodulation steps as:

1. Receive coordinates as inputs
2. Find the 4 nearest points of each input, one for each 2D coset.
3. Calculate 2D square euclidean distances to 4 nearest constellation points
4. Considering pairs, calculate 4D distances by adding 2D distance from each pair, resulting in $4 \times 4 = 16$ possible combinations.

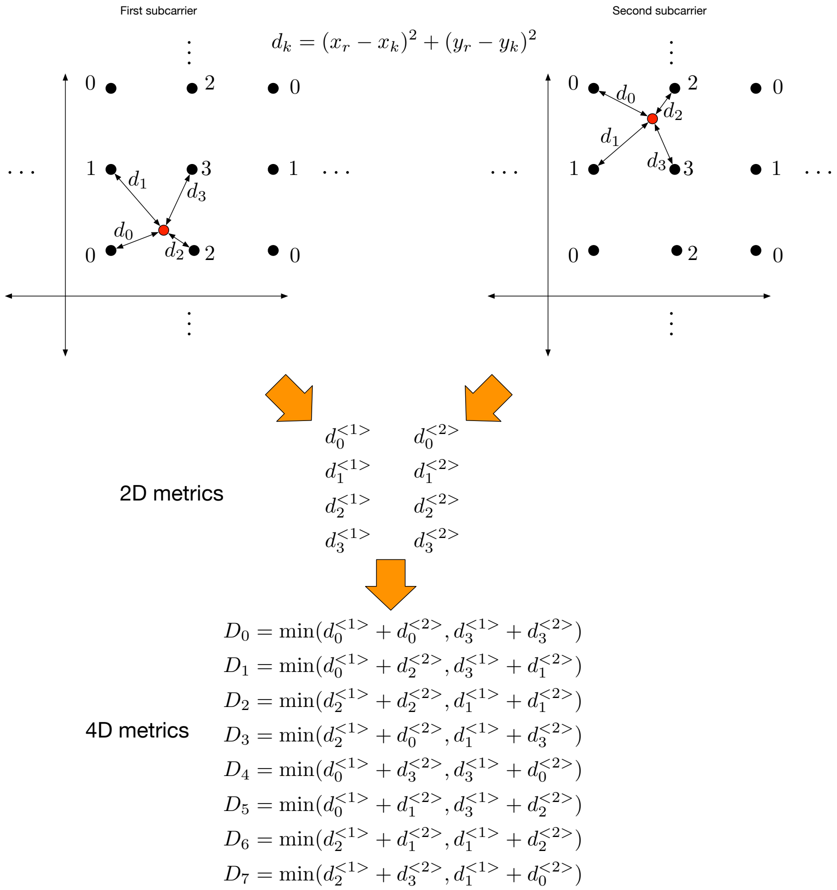

First subcarrier
Second subcarrier
$d_k = (x_r - x_k)^2 + (y_r - y_k)^2$
0
1
2
3
d_0
d_1
d_2
d_3
. . .
2D metrics
$d_0^{<1>}$ $d_0^{<2>}$
$d_1^{<1>}$ $d_1^{<2>}$
$d_2^{<1>}$ $d_2^{<2>}$
$d_3^{<1>}$ $d_3^{<2>}$
4D metrics
$D_0 = \min(d_0^{<1>} + d_0^{<2>}, d_3^{<1>} + d_3^{<2>})$
$D_1 = \min(d_0^{<1>} + d_2^{<2>}, d_3^{<1>} + d_1^{<2>})$
$D_2 = \min(d_2^{<1>} + d_2^{<2>}, d_1^{<1>} + d_1^{<2>})$
$D_3 = \min(d_2^{<1>} + d_0^{<2>}, d_1^{<1>} + d_3^{<2>})$
$D_4 = \min(d_0^{<1>} + d_3^{<2>}, d_3^{<1>} + d_0^{<2>})$
$D_5 = \min(d_0^{<1>} + d_1^{<2>}, d_3^{<1>} + d_2^{<2>})$
$D_6 = \min(d_2^{<1>} + d_1^{<2>}, d_1^{<1>} + d_2^{<2>})$
$D_7 = \min(d_2^{<1>} + d_3^{<2>}, d_1^{<1>} + d_0^{<2>})$

Figure 2.11: 2D and 4D distance metrics.

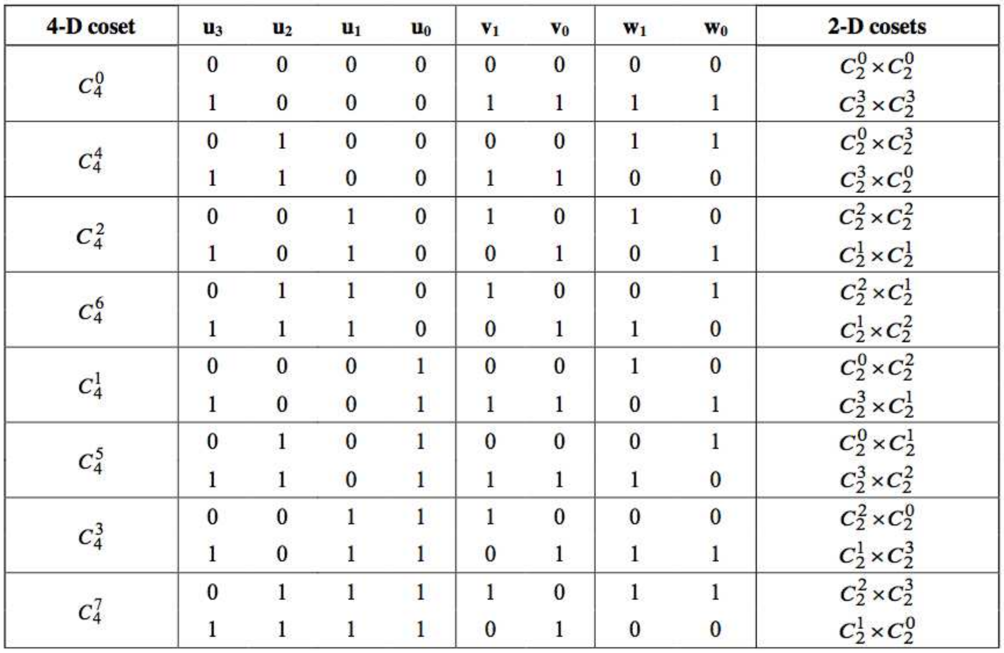
5. Take the minimum, according to the relations between 2D and 4D cosets summarized in Figure 2.12, also choosing the respective point from step 2.

6. The demodulator returns the possible pairs, with theirs respective 4D metrics, as its output.

The demodulator receives the coordinates for each subcarrier as its input and, with the bitload, calculates the demodulation candidates for each pair of subcarriers along with its distance from the received coordinate. The information that is forwarded to the Viterbi decoder is the sequence of distances, in order to calculate the which of the candidate point of the constellation should be used, and their respective candidate points.

2.1.3.5 Decoding

To decode this trellis code, the optimal algorithm, given that the code has only 16 states and it always starts and ends in the all-zero state, is the Viterbi algorithm, as it provides the



4-D coset	u_3	u_2	u_1	u_0	v_1	v_0	w_1	w_0	2-D cosets
C_4^0	0	0	0	0	0	0	0	0	$C_2^0 \times C_2^0$
	1	0	0	0	1	1	1	1	$C_2^3 \times C_2^3$
C_4^4	0	1	0	0	0	0	1	1	$C_2^0 \times C_2^3$
	1	1	0	0	1	1	0	0	$C_2^3 \times C_2^0$
C_4^2	0	0	1	0	1	0	1	0	$C_2^2 \times C_2^2$
	1	0	1	0	0	1	0	1	$C_2^1 \times C_2^1$
C_4^6	0	1	1	0	1	0	0	1	$C_2^2 \times C_2^1$
	1	1	1	0	0	1	1	0	$C_2^1 \times C_2^2$
C_4^1	0	0	0	1	0	0	1	0	$C_2^0 \times C_2^2$
	1	0	0	1	1	1	0	1	$C_2^3 \times C_2^1$
C_4^5	0	1	0	1	0	0	0	1	$C_2^0 \times C_2^1$
	1	1	0	1	1	1	1	0	$C_2^3 \times C_2^2$
C_4^3	0	0	1	1	1	0	0	0	$C_2^2 \times C_2^0$
	1	0	1	1	0	1	1	1	$C_2^1 \times C_2^3$
C_4^7	0	1	1	1	1	0	1	1	$C_2^2 \times C_2^3$
	1	1	1	1	0	1	0	0	$C_2^1 \times C_2^0$

Figure 2.12: Relation between 2D and 4D cosets, from G.fast standard.

best result in a manageable complexity.

The Viterbi decoder is divided in three units: the branch metric unit (BMU), the path metric unit (PMU) and the traceback unit (TBU). The BMU is simply the demodulator, which calculates the squared Euclidean distances (metrics). The PMU searches for the best path in a trellis, calculating the path metrics by summing and comparing the branch metrics provided by the BMU, and also saving each of the decisions in the TBU. The TBU calculates the best path by using the best path metric and then backtracking each decision in order to form the best sequence of outputs (cosets) of the convolutional code. With the coset sequence, we can finally choose the correct candidates to obtain the binary sequence.

The trellis formed by the convolutional code is described in Figure 2.13. It has 16 states, labeled by a single hexadecimal digit. The trellis diagram represents an before-after relationship, with previous states on the column on the left, and next states on the right. A straight line represents a transition. Each state has 4 transitions, one for each possible input (two binary digits gives 4 different possibilities), resulting in a total of 64 transitions. The labels near each states are the output cosets for each transitions. They could be written near their respective lines, but this would result in a bad visualization, so the leftmost coset label represents the uppermost transition. The cosets are represented using a decimal representation of three binary numbers, thus they go from 0 to 7. Also, as the code is systematic, one can retrieve the inputs by just dropping the last bit, or, equivalently, by performing integer division by 2, so a label 6(110) or 7(111) was generated both by input 3(11), with the last bit taken from the encoder's state machine.

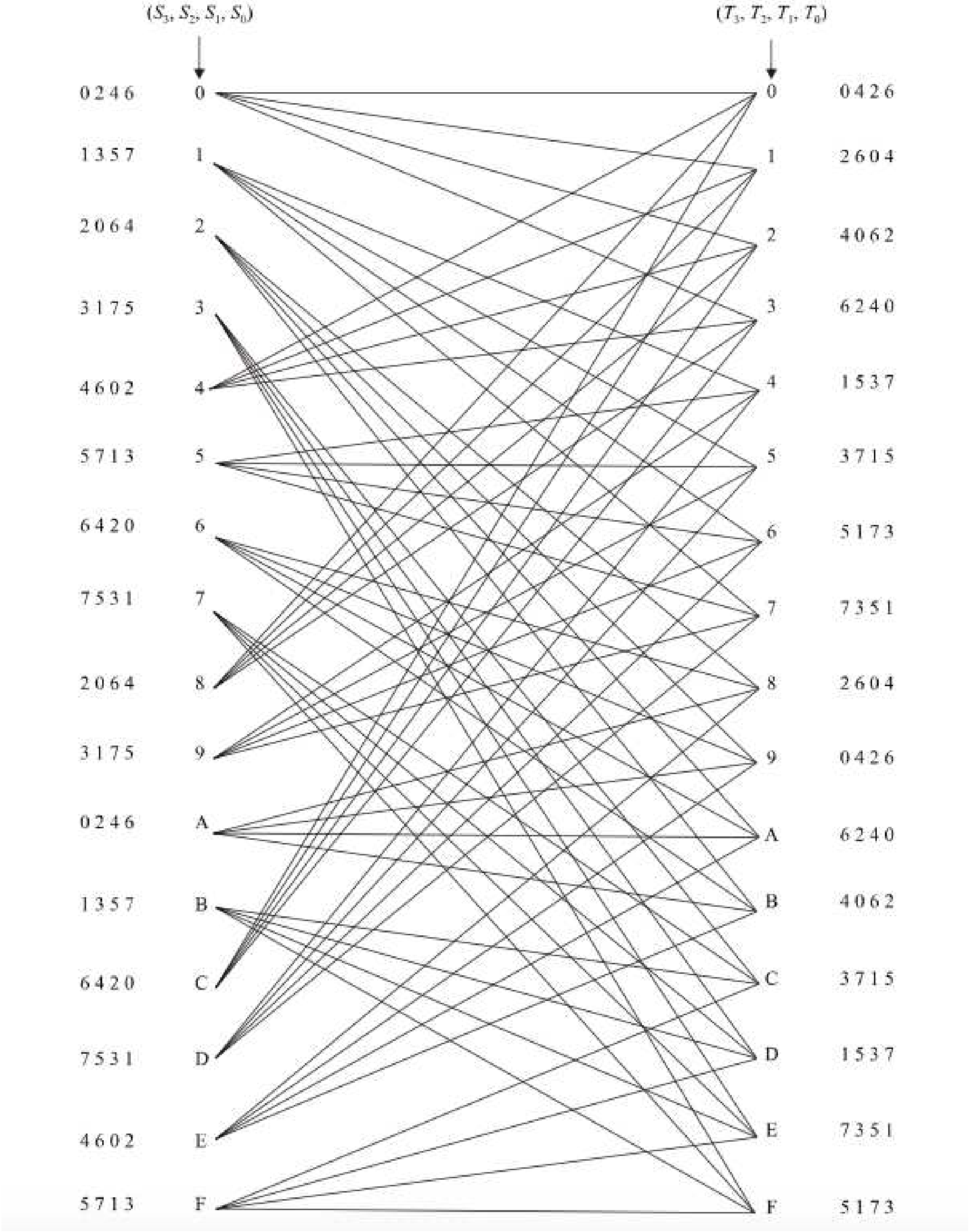


Figure 2.13: Trellis diagram from G.fast.

The viterbi algorithm operates over the trellis to obtain the optimal path. Naively, one could search all possible paths and select one with minimal summed metrics, but this would result in a complexity exponential with the number of pairs. The viterbi algorithm solves this by applying dynamic programming in order to reduce the number of paths to be analyzed. The main concept of dynamic programming is to divide the problem into smaller, recursive sub-problems. Note that this differs from traditional divide-and-conquer strategies, as the recursion makes the sub-problems somewhat dependent on each other.

In order to implement a Viterbi decoder, one must calculate the path with the minimal accumulated metric, so we define a state metric which represents the best metric until this state. These metrics are initialized with zeros. Then, recursively, these state metrics are updated at each step by the recursion formula

$$S_j[t+1] = \min_i (S_{p(j,i)}[t] + B_{j,i}[t]), \tag{2.10}$$

where $S_j[t]$ is the state metric for state j at step t, $B_{j,i}[t]$ is the branch metric calculated at the demodulator for state j and input i at step t and $p(j,i)$ is a state such that when we are at state $p(j,i)$ and receive input i, we end in state j, in other words, state $p(j,i)$ transitions to state j when given an input i. Clearly, function $p(j,i)$ is defined by the design the convolutional code, and can be taken from Figure 2.13 as well. For example, $p(2,1) = C$, as when on state C, and receive input 1, result in state 2. To see the input, we just take the integer division by 2 of the labels, so that input 1 is label 2, which is the fourth transition from top, connecting C to 2.

With just the state metrics, we can only calculate the best path metric, but we can not reconstruct the path itself. So we need to store the decisions at each step as well. This is done by

$$D_j[t+1] = \operatorname*{argmin}_i (S_{p(j,i)}[t] + Bj, i[t]), \tag{2.11}$$

where $D_j[t]$ is the decision at step t which lead to state j. Note that (2.11) is similar with (2.10), with the only difference being a argmin instead of a min.

The Viterbi algorithm runs for all step, and for each step it runs for all states. In the end we will have a state metric S_j for each final step j and a table of decisions $D_j[t]$ along each step t. With the best final state metric we can finally find the best sequence by calculating it backwards. This is done by the traceback unit, which sets the final state by taking the minimum state metric. However this is not necessary in our case, as the encoding procedure guarantees that the sequence always start and finish at state zero. From the final state we just recursively traceback the decisions in order to find previous states and inputs by

$$s[t-1] = p(s[t], D_{s[t]}[t]), \tag{2.12}$$

where sequence $D_{s[t]}[t]$ will retrieve the inputs, and sequence $s[t]$ is the sequence of states. With this information one can finally choose the best sequence of constellation points and truly demodulate points into bits, thus removing the redundancy of convolutional coding.

For example, supposing we have Table 2.1 as the algorithm input. In Figure 2.14 we have a Viterbi algorithm trellis diagram illustrating how the algorithm works. In this

Table 2.1: Distance by 4-D cosets.

	P_0		P_1		P_2		P_3	
Coset	Dist.	Labels	Dist.	Labels	Dist.	Labels	Dist.	Labels
0	13	$(11, 3)$	15	$(52, 4)$	7	$(15, 19)$	5	$(4, 0)$
1	7	$(11, 1)$	5	$(52, 6)$	5	$(8, 18)$	3	$(15, 1)$
2	1	$(9, 1)$	15	$(54, 6)$	3	$(10, 18)$	5	$(5, 1)$
3	7	$(9, 3)$	9	$(49, 7)$	5	$(10, 24)$	3	$(14, 0)$
4	13	$(11, 0)$	7	$(52, 7)$	3	$(15, 24)$	1	$(15, 0)$
5	7	$(8, 1)$	17	$(51, 6)$	1	$(15, 18)$	7	$(4, 1)$
6	13	$(9, 2)$	7	$(49, 6)$	3	$(13, 18)$	5	$(5, 2)$
7	7	$(9, 0)$	17	$(49, 4)$	5	$(13, 24)$	3	$(5, 0)$

example, only the survivors paths are shown for simplicity. A survivor path is a path that was not discarded in the min/argmin process. The red path shows the traceback from final state 0. This diagram was constructed for a sequence of four pairs, thus there are four columns of transitions and five columns of states. The numbers above each state are the states metrics for each state at each step. We can note that possibly an error occurred in the second pair, as there is a sudden increase in all state metrics. By returning to state zero we could avoid an error at the final state, as without this information we would choose for final state 2 instead of 0, as the former has smaller accumulated state metric.

We can see directly that the sequence state is $\{0, 1, 5, 4, 0\}$, and with help of Figure 2.13, we can find the respective 4D cosets as $\{2, 3, 5, 4\}$ and thus find the nearest pair of points to these 4D cosets, thus completing the demodulation/decoding.

2.2 G.hn forward error correction

G.hn [18] is a standard for home network technology which aims for rates of up to 1 Gbps over power lines, telephone copper wires and coaxial cables. G.hn FEC uses LDPC as its code.

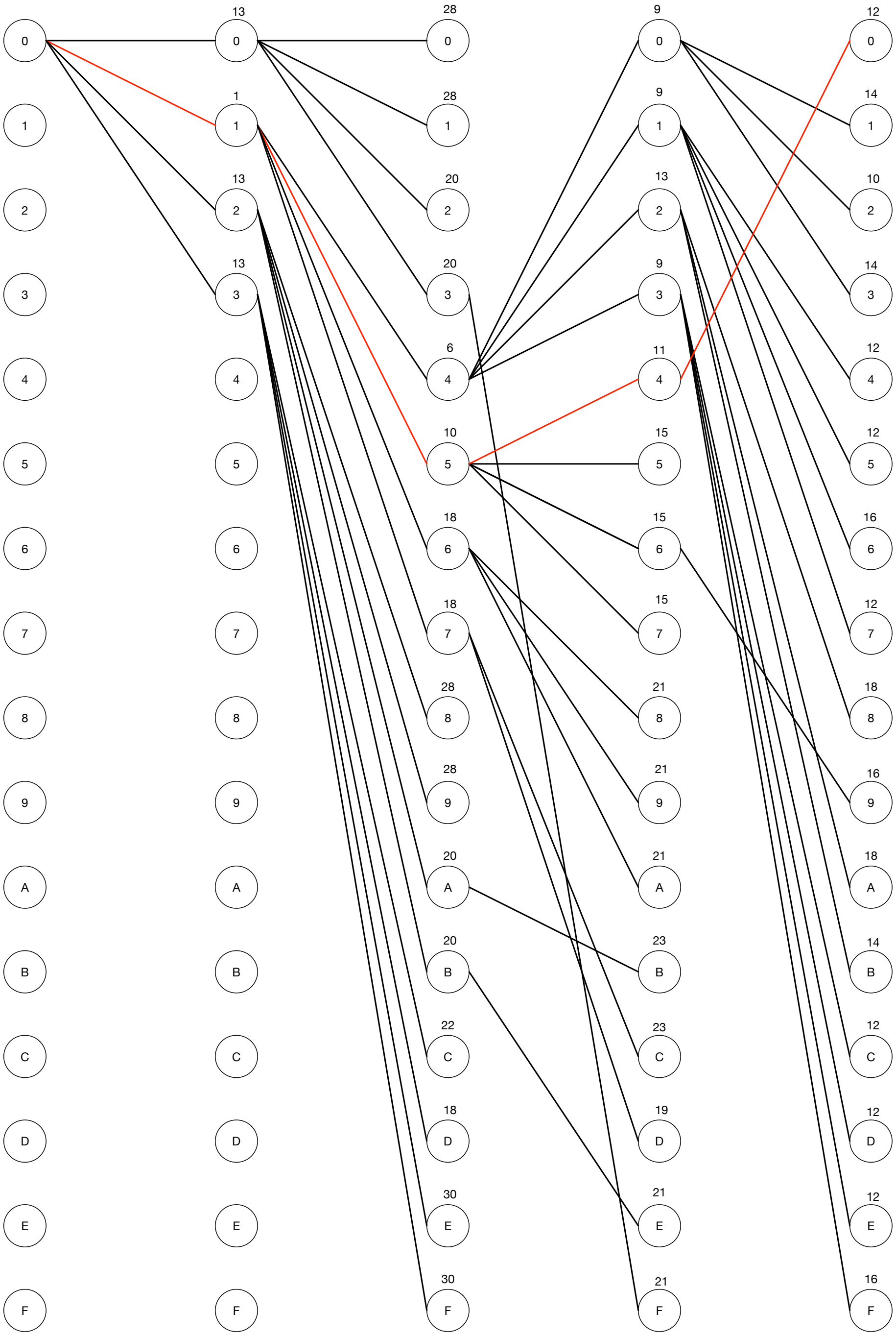


Figure 2.14: Viterbi algorith diagram example.

2.2.1 LDPC

G.hn LDPC uses a quasi-cyclic LDPC block code (QC-LDPC-BC). The standard use 6 codes for data transmission: 3 “short” codes with information size of 120 bytes and 3 “long” codes with information size of 540 bytes. Three rates are used for each size: 1/2, 2/3, 5/6.

Also, rates of 8/9 and 20/21 are achieved by means of puncturing. Puncturing erase both information and redundancy bits at pre-determined positions in order to change the code rate without the need to change the decoder.

The codes are defined by their parity check matrix. Each matrix is composed by an array of $r \times c$ square sub-matrices $A_{i,j}$.

$$H = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & \dots & A_{1,c} \\ A_{2,1} & A_{2,2} & A_{2,3} & \dots & A_{2,c} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{r,1} & A_{r,2} & A_{r,3} & \dots & A_{r,c} \end{bmatrix} \tag{2.13}$$

Each $A_{i,j}$ has dimensions $b \times b$ and is either a identity matrix with a right column shift or an all-zero matrix. This property ensures that we have a sparse matrix. Each parity check matrix is represented in a compact form using an integer $a_{i,j}$ for each sub-matrix. $a_{i,j} = -1$ means a zero matrix, $a_{i,j} = 0$ means an identity matrix and when $a_{i,j}$ is a positive integer means a right column shifted identity matrix.

For example, the parity check matrix in compact form for code $(5/6)_S$, with $r = 4$, $c = 24$, rate 5/6 and $b = 48$ is:

−1	13	32	47	41	24	−1	25	22	40	1	31	8	15	20	15	42	30	13	3	−1	0	−1	−1
25	46	15	43	45	29	39	47	23	38	39	12	−1	21	−1	38	33	0	0	−1	39	0	0	−1
35	45	45	38	14	16	6	11	−1	18	7	41	35	17	32	45	41	−1	18	17	0	−1	0	0
9	32	6	22	26	31	9	8	22	32	40	4	18	40	36	−1	−1	23	31	41	39	20	−1	0

We can note that the code rate R can be calculated as $R = \frac{c-r}{r}$, and the codeword size is $N = c \times b$, which in this case is $N = 24 \times 48 = 1152$ bits. Also, the information size is $K = (c - r) \times b$, resulting in $K = (24 - 4) \times 48 = 960$ bits.

Table 2.2 shows all parameters for each parity check matrix. Note that rate 8/9 and 20/21 are not shown because they are obtained via puncturing of rate 5/6 codes.

2.2.1.1 Encoding

The encoding procedure makes use of the quasi-cyclic structure to reduce complexity. This is a variant of the method described in [19]. H is further sub-divided in blocks:

$$H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}, \tag{2.14}$$

Table 2.2: Summary of G.hn parity check matrices

Code	r	c	b	N	K
$(1/2)_S$	12	24	80	1920	960
$(1/2)_L$	12	24	360	8640	4320
$(2/3)_S$	8	24	60	1440	960
$(2/3)_L$	8	24	270	6480	4320
$(5/6)_S$	4	24	48	1152	960
$(5/6)_L$	4	24	216	5184	4320

Table 2.3: Dimensions of sub-blocks (in compact form).

Sub-matrix	Dimensions
A	$(r-1)\times(c-r)$
B	$(r-1)\times 1$
T	$(r-1)\times(r-1)$
C	$1\times(c-r)$
D	1×1
E	$1\times(r-1)$

with dimensions given in Table 2.3.

With this format we can use these two equations to calculate parity:

$$p_1^T = ET^{-1}As^T + Cs^T \tag{2.15a}$$

$$Tp_2^T = As^T + Bp_1^T, \tag{2.15b}$$

where the codeword was divided as a row-vector $\begin{bmatrix} s & p_1 & p_2 \end{bmatrix}$, with p_1 and p_2 having lengths of b and $(r-1)b$, respectively. The codeword is divided as follows: :s is the systematic part, $p1$ and p_2 are the parity parts.

The encoding algorithm calculates both p_1 and p_2 as follows:

1. Calculate As^T

2. Calculate Cs^T

3. Calculate $ET^{-1}As^T = \begin{bmatrix} (X+I) & I & I & \dots & I \end{bmatrix} As^T = \begin{bmatrix} X & 0 & 0 & \dots & 0 \end{bmatrix} As^T +$

$\begin{bmatrix} I & I & \dots & I \end{bmatrix} As^T$, where X is the matrix at the beginning of E and I is an identity matrix.

4. Calculate p_1^T from (2.15a)

5. Calculate p_2^T from (2.15b) using back substitution

Items 1 and 2 are multiplications of a permutation matrix and a column vector, thus we have just to cyclically rotate blocks to left inside the vector and then apply the XOR operation to sum. Item 3 is done by block-wise sum of As^T plus a cyclically rotated version of the first block. Item 4 is a direct sum. Item 5 can be seen as a system of equations that we can solve easily by back substitution due to the double diagonal structure.

Figure 2.15 shows the relations between the sub-matrices used in the encoding process. The double diagonal form of T is given emphasis. Also, T^{-1} is a lower triangular matrix composed only by identity matrices.

With both p_1^T and p_2^T, we can form the parity and concatenate it with the information, thus forming a codeword.

2.2.1.2 Puncturing

Puncturing of LDPC codes is used to obtain codes with higher rates from existing codes by omitting some bits of a codeword. Puncturing in G.hn is applied only to the codes with rate 5/6, in both sizes: S and L. It is done using a pre-defined pattern, removing always bits from the same positions from a codeword.

There are 4 patterns shown in Table 2.4. The $\mathrm{pp}_T^{(i)}$ notation denotes that the length of the input is T and i bits are erased in total. The overall procedure of puncturing is just a copy of predetermined parts of the codeword to the output. These parts are the positions where the puncturing pattern is equal to 1.

2.2.1.3 Constellation Mapper

G.hn uses gray-coded QAM constellations. The algorithm for calculating the integers coordinates (I, Q) uses a recursive definition. In this work we used only constellations with order m being an even power of 2, in other words, only square constellations were used. The mapper receives b bits and generates the coordinates (I, Q). The data is represented as $d = [\ d_0 \quad d_1 \quad \dots \quad d_{b-1}\ ]$. The algorithm is: if $b = 0$, then $I = 0$ and $Q = 0$, else $I = \mathrm{sgn}_I \times \mathrm{val}_I$ and $Q = \mathrm{sgn}_Q \times \mathrm{val}_Q$, where $\mathrm{sgn}_I = 2d_0 - 1$, $\mathrm{val}_I = |I_{b-2} - 2^{b/2-1}|$, $\mathrm{sgn}_Q = 2d_{b/2} - 1$,

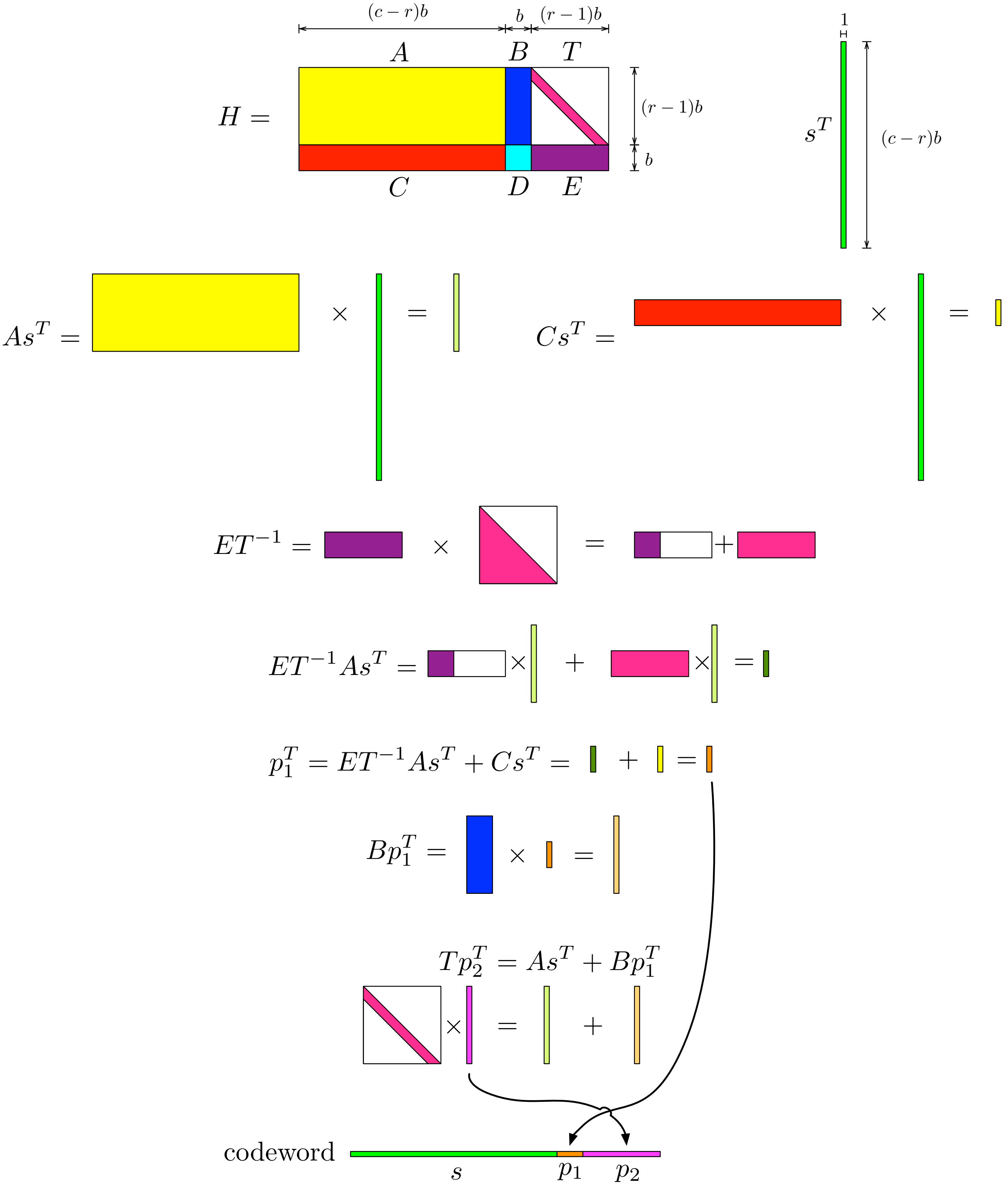


Figure 2.15: Encoding process.

$\text{val}_Q = |Q_{b-2} - 2^{b/2-1}|$. The notation I_{b-2} and Q_{b-2} means calculating I and Q removing d_0 and $d_{b/2}$, using a new $d = [\ d_1 \quad d_2 \quad \ldots \quad d_{b/2-1} \quad d_{b/2+1} \quad \ldots \quad d_{b-1}\ ]$. This algorithm provides gray-coded QAM constellations, ensuring minimal bit errors between neighbor points.

Table 2.4: Puncturing Patterns.

	Puncturing Pattern
$\mathrm{pp}_{1152}^{(72)}$	$[\underbrace{1\ 1\ \cdots\ 1}_{720}\ \underbrace{0\ 0\ \cdots\ 0}_{36}\ \underbrace{1\ 1\ \cdots\ 1}_{360}\ \underbrace{0\ 0\ \cdots\ 0}_{36}]$
$\mathrm{pp}_{5184}^{(324)}$	$[\underbrace{1\ 1\ \cdots\ 1}_{3240}\ \underbrace{0\ 0\ \cdots\ 0}_{162}\ \underbrace{1\ 1\ \cdots\ 1}_{972}\ \underbrace{0\ 0\ \cdots\ 0}_{162}\ \underbrace{1\ 1\ \cdots\ 1}_{648}]$
$\mathrm{pp}_{1152}^{(144)}$	$[\underbrace{1\ 1\ \cdots\ 1}_{720}\ \underbrace{0\ 0\ \cdots\ 0}_{48}\ \underbrace{1\ 1\ \cdots\ 1}_{240}\ \underbrace{0\ 0\ \cdots\ 0}_{96}\ \underbrace{1\ 1\ \cdots\ 1}_{48}]$
$\mathrm{pp}_{5184}^{(648)}$	$[\underbrace{0\ 0\ \cdots\ 0}_{216}\ \underbrace{1\ 1\ \cdots\ 1}_{4320}\ \underbrace{0\ 0\ \cdots\ 0}_{432}\ \underbrace{1\ 1\ \cdots\ 1}_{216}]$

2.2.1.4 Demapper

The constellation demapper receives coordinates $R = (x, y)$ which were subject to noise from the channel. A common QAM demapper would calculate the nearest constellation coordinate and return the bits associated with it, but as we are using LDPC, we need that the demapper returns more information in the form of log-likelihood ratios (LLRs), representing the probability of each bit being 0 or 1 in a different way. The formula for calculating the exact LLR is

$$\mathrm{LLR}_{b_i} = \ln\left(\frac{\Pr(b_i = 0 | R = (x, y))}{\Pr(b_i = 1 | R = (x, y))}\right) = \ln\left(\frac{\sum\limits_{s \in S_0} \exp\left(-\frac{1}{\sigma^2}((x - s_x)^2 + (y - s_y)^2)\right)}{\sum\limits_{s \in S_1} \exp\left(-\frac{1}{\sigma^2}((x - s_x)^2 + (y - s_y)^2)\right)}\right), \quad (2.16)$$

where S_0 and S_1 represent subsets where b_i is equal to 0 and 1, respectively. Although (2.16) provides the exact result, it is computationally expensive as it requires summing through all the exponentials of each distance from constellation points. An alternative to this is recognizing that $\log \sum \exp(-a) \approx -\min(a)$, resulting in

$$\mathrm{LLR}_{b_i} \approx -\frac{1}{\sigma^2}\left(\min_{s \in S_0}((x - s_x)^2 + (y - s_y)^2) - \min_{s \in S_1}((x - s_x)^2 + (y - s_y)^2)\right), \quad (2.17)$$

which have reduced complexity, as we need only to look for the two nearest coordinates, one for $b_i = 0$ and one for $b_i = 1$. Also, this approach eliminates ln and exp functions, avoiding the possibility of numeric overflow and underflow.

2.2.1.5 Depuncturing

Depuncturing accomplishes the reverse process of puncturing, but it takes in account the fact that the input from the demapper are LLRs, not bits. Actually, this fact turns depucturing much more easier, as an erasure is easily represented by using a LLR with value 0, meaning that there is no information whether the bit is 0 or 1 (remember that a positive value indicates 0 and a negative value indicates 1). Figure 2.16 shows the depuncturing process. The zigzag pattern represents that the codeword was subject to noise and data is represented with LLRs instead of bits. A solid fill represents decoded data free of noise, with data represented in bits.

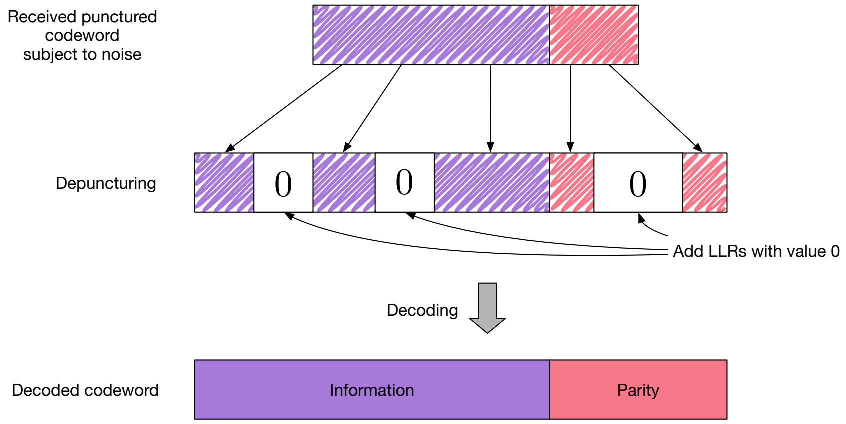


Figure 2.16: Depuncturing process.

2.2.1.6 Decoding

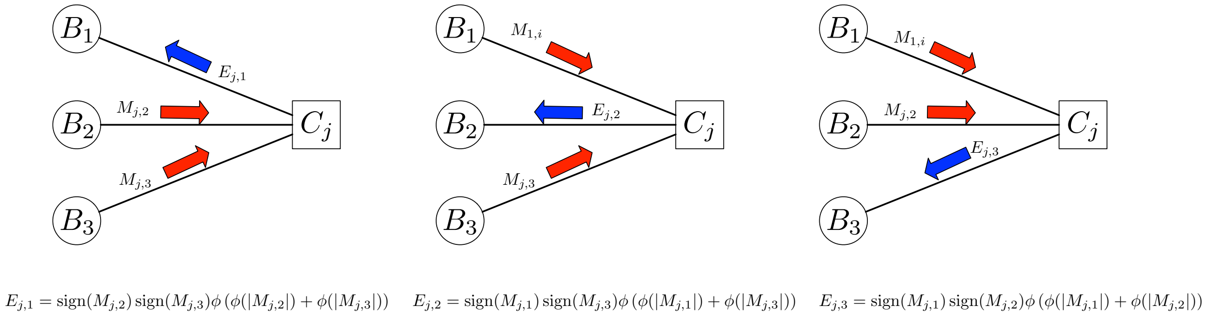
LDPC decoding is, by far, the most computational expensive part. LDPC decoding method is not specified by the standard, so we use the sum-product algorithm. The sum-product algorithm is composed by 2 main parts: the check-node calculation and the bit node calculation [15].

Check-node calculation receives LLRs from the bit-nodes and calculates the check-to-bit messages using

$$E_{j,i} = \left(\prod_{i'} \operatorname{sgn}(M_{j,i}) \right) \phi \left(\sum_{i'} \phi(|M_{j,i}|) \right), \tag{2.18}$$

where $E_{j,i}$ is the message from check-node j to bit-node i, the index i' means every bit-node connected except the bit-node i, $\phi(x) = -\ln \tanh \frac{x}{2} = \frac{e^x+1}{e^x-1}$. Figure 2.17 shows how a check node generates messages for each bit node connected to it.

Bit-node calculation receives LLRs from both the demapper and check-nodes in order



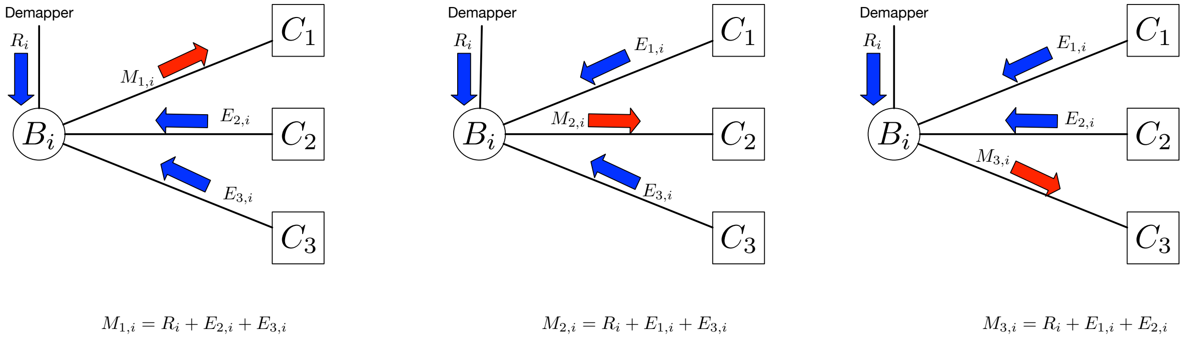
$E_{j,1} = \text{sign}(M_{j,2})\,\text{sign}(M_{j,3})\phi\left(\phi(|M_{j,2}|) + \phi(|M_{j,3}|)\right)$ $E_{j,2} = \text{sign}(M_{j,1})\,\text{sign}(M_{j,3})\phi\left(\phi(|M_{j,1}|) + \phi(|M_{j,3}|)\right)$ $E_{j,3} = \text{sign}(M_{j,1})\,\text{sign}(M_{j,2})\phi\left(\phi(|M_{j,1}|) + \phi(|M_{j,2}|)\right)$

Figure 2.17: Check-node processing.

to calculate bit-to-check messages using

$$M_{j,i} = R_i + \sum_{j'} E_{j',i}, \tag{2.19}$$

where R_i is the a priori LLR from the demapper, the index j' means every check-node connected except check-node j and $E_{j,i}$ the bit-to-check message. Figure 2.18 shows an example of how a bit-node calculates each message.

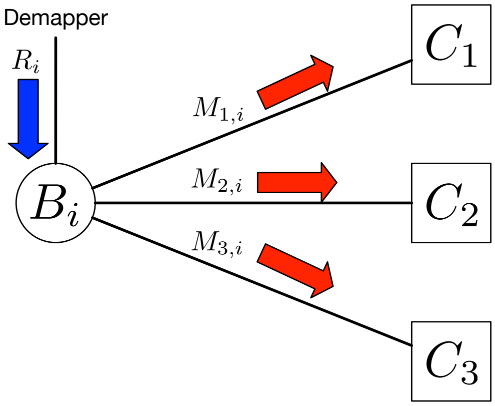


$M_{1,i} = R_i + E_{2,i} + E_{3,i}$ $M_{2,i} = R_i + E_{1,i} + E_{3,i}$ $M_{3,i} = R_i + E_{1,i} + E_{2,i}$

Figure 2.18: Bit-node processing.

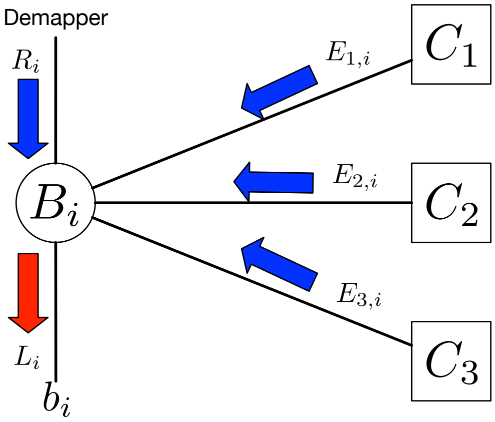
Besides the two main parts, there are initialization and verification calculations. Initialization can be seen as a special case of the bit-to-check message, when there are no previous check-to-bit messages, so that each $M_{j,i} = R_i$ as is shown in Figure 2.19.

Verification calculates a posteriori LLRs from a priori LLRs plus check-to-bit LLRs, in a similar fashion of Figure 2.18 and (2.19). Figure 2.20 shows how a bit-node generates a posteriori LLR L_i.



$$M_{1,i} = R_i$$
$$M_{2,i} = R_i$$
$$M_{3,i} = R_i$$

Figure 2.19: Initialization.



$$L_i = R_i + E_{1,i} + E_{2,i} + E_{3,i}$$

Figure 2.20: A posteriori LLR calculation.

From L_i we can update the values for each bit b_i using

$$b_i = \begin{cases} 0 & \text{if } L_i \geq 0 \\ 1 & \text{if } L_i < 0 \end{cases}, \tag{2.20}$$

which is used to verify the parity check equations. This final step is a sparse binary matrix-binary vector multiplication, which is implemented easily using just XOR functions.

The decoding algorithm stops if a valid codeword is reached or if the maximum number of iterations is reached. Either way the decoder stops and returns whatever improvements it could make. In the case when a valid codeword was not achieved, it signals that there was a decoder failure and thus the data is certainly wrong and must be discarded by its receiver.

Chapter 3

Results

This chapter contains the comparison between the RS+TCM scheme from G.fast and the LDPC scheme from G.hn. Comparison is made taking in account both error correction performance and implementation complexity.

3.1 Error correction performance

Error correction performance is measured using metrics related to error rates. An error correction code reduce the error rates by using redundancy bits. Furthermore, the error rates depend on noise levels of the channel. The more noisy a channel is, the bigger are the error rates.

3.1.1 Metrics

The error metric used in this work is the block error rate (BLER), instead of the traditional bit error rate (BER). This is done because BLER is more easily calculated as

$$\text{BLER} = \frac{\text{number of blocks with error}}{\text{total number of blocks}} \tag{3.1}$$

and

$$\text{BER} = \frac{\text{number of bit errors}}{\text{total number of bits}}, \tag{3.2}$$

from which we can see that BER calculation requires bit-by-bit verification by comparison of transmitted and received sequences, whereas BLER calculation can be done simply by verifying the decoder results, in other words, if the decoder declares a decoding error, the entire block is already in error so that it is not necessary to compare it with the transmitted block. A block is defined as a sequence of N bits.

Also, if we assume the bits are statistically independent one can convert BLER into BER and vice-versa by using

$$\mathrm{BLER} = 1 - (1 - \mathrm{BER})^N \tag{3.3}$$

$$\mathrm{BER} = 1 - \sqrt[N]{1 - \mathrm{BLER}}. \tag{3.4}$$

For simulations using complex symbols (QAM) we have

$$\frac{E_s}{N_0} = \mathrm{SNR}L \tag{3.5}$$

where L is the oversampling factor, which is given in samples per symbol. Considering a symbol-based simulation we have that $L = 1$ (a sample is a symbol) so $\frac{E_s}{N_0} = \mathrm{SNR}$. Also we have that

$$E_s = RE_b, \tag{3.6}$$

where R is given in information bits per symbol. R is calculated as $R = R_m R_c$, where $R_m = \log_2 M$ is the number of bits per QAM symbol and R_c is the code rate from the channel coding scheme used, giving the proportion of information bits by total bits.

Joining (3.5) and (3.6), we have

$$\frac{E_s}{N_0} = \mathrm{SNR} = \frac{E_b}{N_0}R \tag{3.7}$$

and then, converting to dB, we have

$$\frac{E_b}{N_0}(\mathrm{dB}) = -10\log_1 0(R) + \mathrm{SNR}(\mathrm{dB}) \tag{3.8}$$

where the schemes with the highest $R = R_m R_c$ are favored, as their curves are moved to the left the most.

In order to simulate, we must set the SNR at the channel in each simulation, this is done by setting E_s to a fixed value, and, knowing that $\sigma^2_{\text{complex}} = N_0$, we have that

$$\sigma^2_{\text{complex}} = \frac{E_s}{\mathrm{SNR}} \tag{3.9}$$

where $\sigma^2_{\text{complex}}$ is the variance of the complex Gaussian noise. Note that the noise level on each in-phase and quadrature component is still $\frac{\sigma^2_{\text{complex}}}{2} = \sigma_I^2 = \sigma_Q^2 = \frac{N_0}{2}$. After the simulation we can just convert the SNR axis to a $\frac{E_b}{N_0}$ axis using (3.8).

3.1.2 Simulation

Simulation was performed for both standards, G.hn and G.fast. The main focus of simulation is to assess the error correction performance for each code, without resorting to direct evaluation of error probabilities from the code structure. Instead we take a Monte Carlo approach in order to simplify this evaluation, as evaluating LDPC codes can be troublesome due to its longer codewords and its pseudo-random design. Monte Carlo method disadvantage is that it evaluates only average behaviour of codes, ignoring worst and best cases.

Simulation is done by following these steps:

Generate random data This is done by a linear feedback shift register pseudo random number generation. A 48-bit register is used.

Encode data To encode data, we use one of interleaved RS or LDPC. Random data is encoded with redundancy and sent to the slicer block. Interleaved RS encodes multiple codewords and then interleave them before sending. LDPC just encodes a codeword.

Slice encoded data according to bitload Slicing has two options. A common slicer, which slices the bitstream according to the bitload, and a TCM slicer, which convolutionally encodes the bitstream while slicing the bits. The TCM slicer still respects the bitload by calculating the number of redundancy bits in prior and taking less bits from the input, so that the resulting number of bits complies with the bitload.

Map sliced data according to bitload Mapping is done in two ways, G.fast and G.hn. G.fast mapping takes in account the TCM scheme, resulting in a non-Gray coded constellation. G.hn uses a common Gray-coded scheme to map data to constellations.

Send symbols through AWGN channel We add Gaussian noise to the constellation points in order to emulate a AWGN channel. Noise is also pseudo randomly generated by the same linear feedback shift register to create uniformly distributed samples, then using a Box-Muller transform we generate two Gaussian distributed samples from two uniform ones.

Demap data Demapping can be LLR demapping, for soft-decoding of LDPC codes, which calculates the probabilities of each bit being 0 or 1 inside a constellation point. Or it can be Viterbi demapping, which return the most probable point by taking in account the finite state machine created by the convolutional code.

Deslice data Deslicing, as its slicing counterpart, also is done in two ways. A common deslicer, which combines separated bits (LLRs) into a bitstream again, according to the bitload, while a TCM deslicer performs a similar task, but it drops the parity bits added by the convolutional encoder.

Decode data Interleaved RS decoder deinterleaves data prior to RS decoding, spreading possible error bursts. LDPC decoder receives LLRs as its inputs and decodes iteratively, resulting in a codeword formed by bits.

Generate statistics After decoding, errors are counted using the data at the decoder output.

The simulator generates BLER for each SNR value inputted, given a configuration of coding and modulation scheme. Also, we can set parameters like a range of SNRs, number of points to simulate, maximum number of simulations and maximum number of errors.

The range of SNRs is crucial to define where the plotted curve will lay, and it is necessary to plot the most important part of a error versus noise curve called waterfall region, which shows at which SNR levels a scheme starts to produce errors.

The number of points to simulate is decided after we manage to discover where the waterfall region lies, as an increased number of points results in long simulation times. A low number of points may not show properly the waterfall region, so there is a trade-off. These points are in a equally spaced assuming that SNR is in decibels.

The maximum number of simulation limits the simulation time, as when we try to estimate a very low BLER (say, 10^{-20}) would require an enormous simulation time. In other words, we limit how low a BLER can be estimated in order to save time.

The maximum number of errors is used for stopping the simulation early and reduce simulation time. In the estimation of BLER we divide the number of errors by the total number, so that a low number of errors results in a poor estimation. In th other hand, if the simulation already reached a good number of errors, we stop the simulation and starts the next.

The simulator and all the blocks were implemented in C programming language. The simulator uses OpenMP to parallelize simulation of different points with a dynamic schedule, which implements a task queue to perform the simulation, where each task is the simulation of BLER for a given SNR.

Simulations were run on a cluster with 8 computers with 8 cores each. The configuration used was 16 points, 100 errors and 100000 simulations.

3.1.3 Results

To make the comparison, we must first set similar conditions for both codes, so we used RS with 135 bytes of information plus 16 of redundancy and a interleaver with depth $D = 4$, and the longer (L) matrices for LDPC. This is because we end with the same block size for both codes, so that we can compare their BLER, as the BLER depends on the block size. In this case, the information block size is 540 bytes for both codes. At first, this may seem unfair for RS by considering four codewords as one, but one must recall that after interleaving there is the TCM encoder, which will encode bits from different codewords together, due to the interleaving.

The comparison is made here with 64-QAM in all sub-carriers. A (135, 151) Reed-Solomon code with interleaver depth $D = 4$ and trellis-coded modulation on 2048 sub-carriers is used, resulting in a overall rate of $\frac{135}{151}\frac{2048\times6-(2048/2+4)}{2048\times6} = 0.8192$. The LDPC codes used are the larger codes with 540 bytes of information, with rate varying from 1/2 to 20/21, according to its standard.

Error correction curves generally have the same trend, exhibiting a non convergence region in low signal to noise ratios, and a waterfall region where the error probability suddenly drops. The main objective is to make curves to

Figure 3.1 shows that, if we consider only the sheer amount of error correction performance, all LDPC codes outperform the RS+TCM scheme, excepting the LDPC with rate 20/21. This is somewhat unfair, as for example, LDPC with rate 1/2 uses 540 bytes of parity along with 540 bytes of information, while the RS+TCM scheme uses less than 120 bytes of redundancy. Comparing codes with similar rates can give a better view, for example, LDPC with rate $5/6 = 0.8333$, thus a higher rate (less redundancy) still outperforms RS+TCM scheme by about 2 dB in E_b/N_0. Another comparison is possible with LDPC codes with rate $16/18 = 0.8889$ as it performs slightly better, about 0.75 dB,than RS+TCM and uses less than 68 bytes of redundancy, against 120 of RS+TCM. In other words, RS+TCM uses 76% more redundancy than LDPC and it is still outperformed.

Another case is shown in Figure 3.2, now for 1024-QAM on all sub-carriers. Now the RS+TCM has the code rate of 0.8492, using less than 96 bytes of redundancy. As the same of the 64-QAM case, LDPC with rate 16/18 outperforms the RS+TCM scheme by more than 1 dB, with RS+TCM using 41% more redundancy.

Figure 3.3 shows the Shannon capacity for the BLER of 10^{-3} with different modulation orders. Each marker in a curve represents a QAM of order 4, 16, 64, 256, 1024. We can see directly that the LDPC curves are closer than the RS+TCM, with the difference becoming

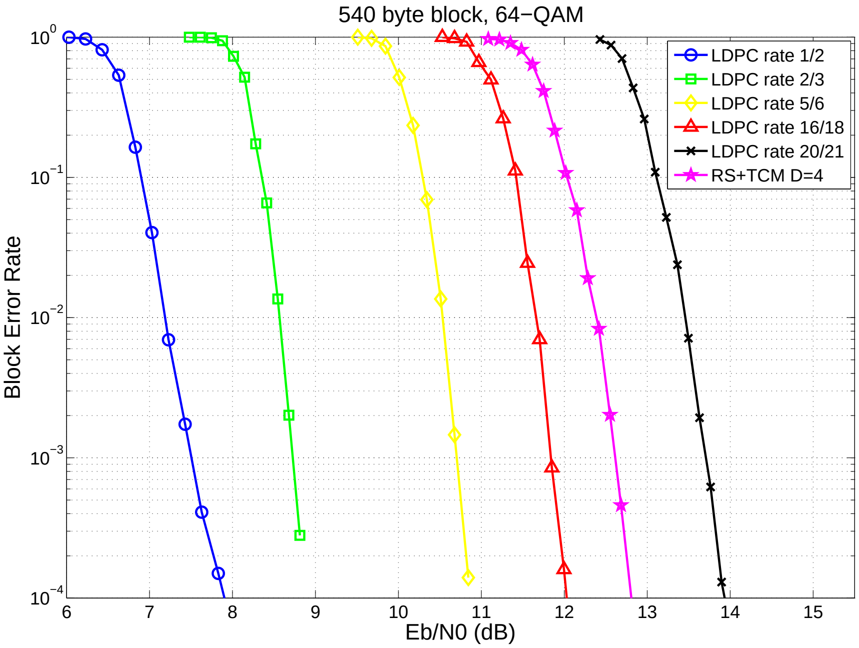


Figure 3.1: RS+TCM vs LDPC on 64-QAM.

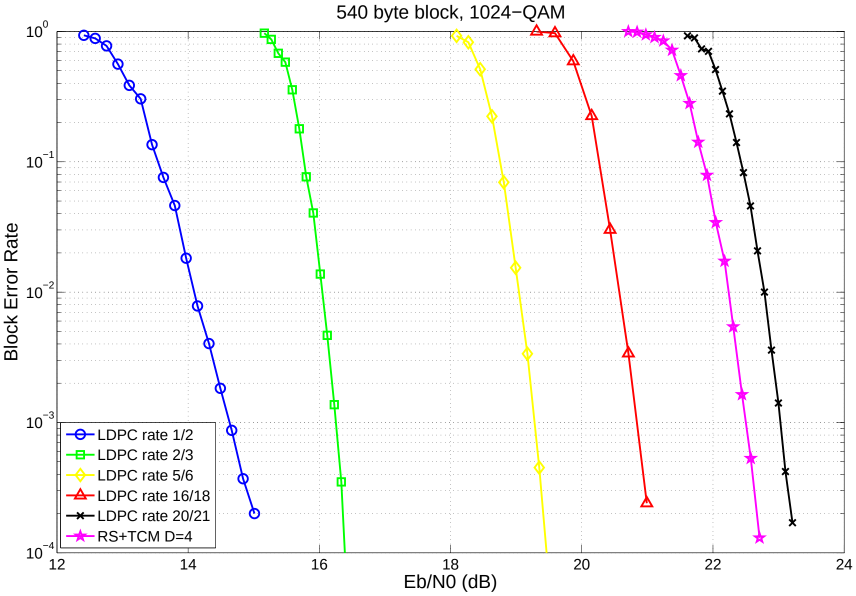


Figure 3.2: RS+TCM vs LDPC on 1024-QAM.

more apparent at higher SNRs. G.fast is expected to deal with SNRs higher than 30dB, in which LDPC is more effective than RS+TCM. For example, LDPC 5/6 1024-QAM and RS+TCM 1024-QAM have similar information rates, but LDPC has an advantage of 3 dB in SNR.

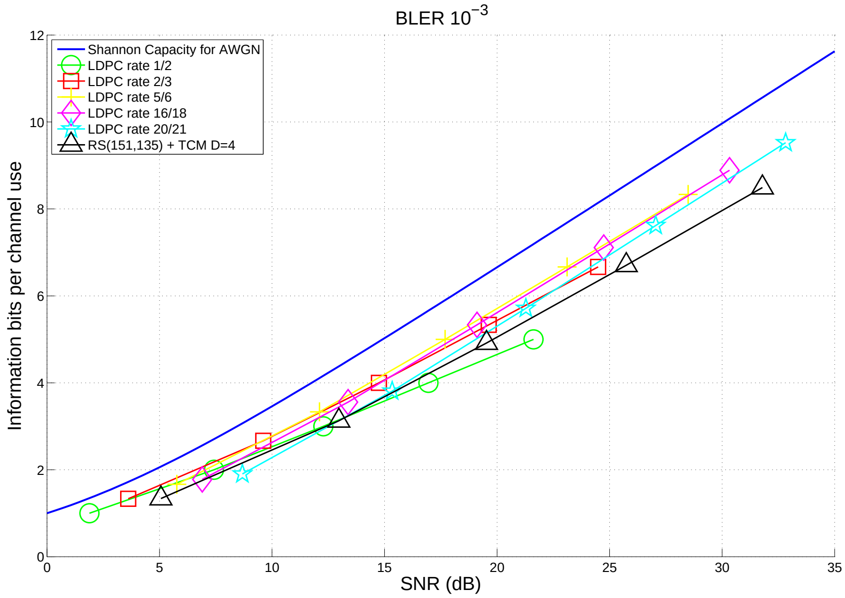


Figure 3.3: Comparison to Shannon capacity.

3.2 Complexity

Another important point of view is the computational complexity of both schemes. Computational complexity may represent a lot of concepts. For example, when designing hardware, a very complex function may require lots of circuitry, increasing the cost of the project. Another example is when dealing with a software implementation, a complex function may consume resources like cache, memory and processor cycles.

3.2.1 Metrics

Measuring computational complexity a in fair way is not trivial. One must bring all operations in a algorithm to a common denominator, for example, number of additions and multiplications, or number of cycles spent on processing. But none of them is perfect, as an operation can be implemented directly in some hardwares, and is unavailable in others, thus benefiting the former in number of cycles, for example. Also, there are subtleties that are ignored at first, then prove themselves to be troublesome, like memory and cache accesses in a software implementation. In this work we assume different operations for different schemes, thus resulting in a somewhat heterogeneous comparison.

3.2.2 Results

Here we calculate the complexity costs for each scheme. We present the complexity by counting operations and memory required for each algorithm. In order to simplify, we do not count memory accesses and we also assume worst case situations.

3.2.2.1 G.fast complexity

Reed-Solomon RS complexity is concentrated at the decoder. The decoder only has complexity comparable to the encoder when there are no errors.

Encoder The encoder implements a digital filter in direct form II transposed. This results in a time complexity of:

$$K_{RS}((N_{RS} - K_{RS} - 1)(\mathrm{GF}_{\mathrm{MULT}} + \mathrm{GF}_{\mathrm{ADD}}) + \mathrm{GF}_{\mathrm{MULT}})$$

where $\mathrm{GF}_{\mathrm{MULT}}$ is Galois Field multiplication, $\mathrm{GF}_{\mathrm{ADD}}$ is Galois Field addition, which is simply a XOR operation. The encoding algorithm uses $(N_{RS} - K_{RS})$ bytes as the filter memory.

Decoder The RS decoder has a lot of steps, as follows:

Syndrome Syndrome calculation is done by evaluating $N_{RS} - K_{RS}$ polynomials of N_{RS} coefficients. This is can be done using Horner's method repeatedly, with a time complexity of:

$$(N_{\mathrm{RS}} - K_{\mathrm{RS}})N_{\mathrm{RS}}(\mathrm{GF}_{\mathrm{MULT}} + \mathrm{GF}_{\mathrm{ADD}})$$

using $(N_{\mathrm{RS}} - K_{\mathrm{RS}})$ bytes of memory.

Berlekamp-Massey In order to find the error locator polynomial, the Berlekamp-Massey algorithm works by constructing a polynomial incrementally, increasing its degree at each step. In the worst case, we have $(N_{\mathrm{RS}} - K_{\mathrm{RS}})/2$ errors, resulting in a polynomial of degree $(N_{\mathrm{RS}} - K_{\mathrm{RS}})/2$. The time complexity is:

$$\begin{aligned} \frac{(N_{\mathrm{RS}} - K_{\mathrm{RS}})}{2}(\frac{(\frac{N_{\mathrm{RS}}-K_{\mathrm{RS}}}{2} - 1)}{2}&(\mathrm{GF}_{\mathrm{MULT}} + \mathrm{GF}_{\mathrm{ADD}})+ \\ \frac{(N_{\mathrm{RS}} - K_{\mathrm{RS}})}{2}&(\mathrm{GF}_{\mathrm{MULT}} + \mathrm{GF}_{\mathrm{ADD}})+ \\ \frac{(N_{\mathrm{RS}} - K_{\mathrm{RS}})}{2}&(\mathrm{GF}_{\mathrm{MULT}} + \mathrm{GF}_{\mathrm{RECIP}})) \end{aligned}$$

and the space complexity is $\frac{3(N_{\mathrm{RS}}-K_{\mathrm{RS}})}{2}$.

Chien Search Chien search finds the roots of the error locator polynomial by exhaustive search. This results in the evaluation of N_{RS} polynomials of degree $\frac{(N_{\text{RS}}-K_{\text{RS}})}{2}$ in the worst case. This leads to a time complexity of

$$N_{\text{RS}}\frac{(N_{\text{RS}}-K_{\text{RS}})}{2}(\text{GF}_{\text{MULT}}+\text{GF}_{\text{ADD}})$$

using a space complexity of $(N-K)$ bytes.

Error evaluator polynomial The error evaluator polynomial is calculated by a partial convolution of the syndrome polynomial and the error locator polynomial up to $\frac{(N-K)}{2}$ coefficients. The time complexity is:

$$\frac{N-K}{2}\frac{(\frac{(N-K)}{2}-1)}{2}(\text{GF}_{\text{MULT}}+\text{GF}_{\text{ADD}})$$

The space complexity is $\frac{(N-K)}{2}$ bytes.

Error correction The error correction step uses the Forney algorithm to correct errors. It requires a evaluation of the error evaluator polynomial, a evaluation of the formal derivative of the error evaluator polynomial, two multiplications and a division. Thus the time complexity is:

$$\begin{aligned}\frac{N_{\text{RS}}-K_{\text{RS}}}{2}(&\frac{N_{\text{RS}}-K_{\text{RS}}}{2}(\text{GF}_{\text{MULT}}+\text{GF}_{\text{ADD}})+\\ &\frac{N_{\text{RS}}-K_{\text{RS}}}{4}(\text{GF}_{\text{MULT}}+\text{GF}_{\text{ADD}})+\\ &2\text{GF}_{\text{RECIP}}+2\text{GF}_{\text{MULT}}+\text{GF}_{\text{ADD}})\end{aligned}$$

and the space complexity is 0, as the error corrections are applied directly.

Interleaving Interleaving has only two parts, one in transmission, and one in reception, namely interleaver and deinterleaver.

Interleaver The interleaver expends most of its computational costs on memory, as it has store D entire codewords at each transmission. If done in software, the interleaver time complexity is just the calculation of the interleaved indices from the non-interleaved indices, resulting in:

$$N_{\text{RS}}D(\text{INT}_{\text{DIV}}+\text{INT}_{\text{MOD}})$$

where INT_{DIV} is the complexity of an integer division and INT_{MOD} is the complexity of an integer division remainder. In some systems both operations can be accomplished together at once. The space complexity is $N_{\text{RS}}D$ bytes.

Deinterleaver The deinterleaver performs the reverse of the interleaver, resulting in same time complexity:

$$N_{\mathrm{RS}}D(\mathrm{INT}_{\mathrm{DIV}} + \mathrm{INT}_{\mathrm{MOD}})$$

and same space complexity $N_{\mathrm{RS}}D$ bytes.

Trellis-coded modulation Trellis coded modulation has several parts, as follows:

Convolutional Encoder The convolutional encoder is a binary digital filter, which is implemented simply only using XOR on bits. This results in a time complexity of

$$N_{\mathrm{Pairs}}3\mathrm{XOR}$$

where N_{Pairs} is half the number of subcarriers and XOR is bit exclusive OR. The convolutional encoder have 4 binary delay units, resulting in a space complexity of 4 bits.

Bit conversion The bit-conversion maps 8 4-D cosets on 4 × 4 2-D cosets. This is done by 4 binary equations which uses only XOR, resulting in a time complexity of

$$N_{\mathrm{Pairs}}5\mathrm{XOR}$$

and 4 bits space complexity only to store the results.

Constellation mapper The constellation mapper can use pre-calculated values, resulting in just a read from memory. This causes the time complexity to be just the memory read complexity and the space complexity is the order of the QAM scheme. If implemented without pre-calculated values, the time complexity is approximately

$$B_{\mathrm{length}}(E[b_x]2(\mathrm{SHIFT} + \mathrm{ADD} + \mathrm{OR}) + \mathrm{SUB} + \mathrm{SHIFT} + \mathrm{MULT}),$$

where B_{length} is the bitload length omitting subcarriers with zeros, $E[b_x]$ is the expected value of the number of bits per sub-carrier b_x. SHIFT, ADD, SUB, OR and MULT, represent the complexity of bit shift, addition, subtraction, logical OR, and multiplication operations. The advantage is that this approach does not need to store any values, thus having no memory complexity.

Constellation demapper The constellation demapper calculates the 2D euclidean distance metrics to the four nearest points. This means a squared euclidean distance calculation repeated 4 times for each sub-carrier. This results in a time complexity of

$$B_{\mathrm{length}}(2\mathrm{FLOOR} + \mathrm{ADD} + \mathrm{SHIFT} + 4(2\mathrm{SUB} + 2\mathrm{MULT} + \mathrm{ADD})),$$

and memory complex of the order of the QAM constellation, as the demodulator has supply the respective labels for each constellation point.

Viterbi decoder The Viterbi decoder is divided in three parts:

BMU To calculate the branch metrics one should find the 4D cosets distance metrics. Basically this is accomplished by add two 2D coset metrics and then calculating the minimum. Also, we want to know which labels results in the minimum metrics, so instead of a minimum, we calculate argmin. This results in a time complexity of

$$N_{\text{pairs}} 8(2\text{ADD} + \text{ARGMIN})$$

and no memory complexity.

PMU Path metric calculation is done by the so called Add-Compare-Select (ACS) butterfly. As its name suggests, it accumulates the branch metrics (add), find the minimum (compare), and also find the argmin(select). This is done for every state, resulting in

$$N_{\text{pairs}} 16(4\text{ADD} + \text{ARGMIN}[4]),$$

where ARGMIN[4] is the complexity of calculating argmin of 4 numbers at a time. The memory complexity of PMU is 16 integers for state metrics and $4N_{\text{pairs}}$ decisions to be stored for the TBU.

TBU Traceback is done simply and has time complexity of

$$(N_{\text{pairs}} - 1)(\text{OR} + \text{SHIFT} + \text{AND})$$

Bit deconversion This is step is inverse process of bit conversion and has the same computational costs. The time complexity is

$$N_{\text{pairs}} 5\text{XOR}$$

and no memory complexity.

3.2.2.2 G.hn complexity

LDPC The LDPC is divided in two parts:

Encoder The encoder uses the QC-LDPC-BC structure to efficiently encode data. The main operation is the right cyclic shift. We can further divide the encoding process in several parts:

As^T Direct multiplication by matrix A, resulting in a time complexity of

$$(M-1)K(\mathrm{RCS}[L]+\mathrm{XOR}[L]),$$

where M is the parity size, K is the information size, $\mathrm{RCS}[L]$ is a right column shift of L bits and $\mathrm{XOR}[L]$ is a XOR operation on L bits. The memory complexity is zero as the algorithm can run in place.

Cs^T Direct multiplication by matrix C, resulting in a time complexity of

$$K(\mathrm{RCS}[L]+\mathrm{XOR}[L]),$$

again with no memory complexity.

$ET^{-1}As+Cs^T$ This calculates the first part of the parity p_1^T. Its time complexity is

$$M\mathrm{XOR}[L]+RCS[L],$$

with no memory complexity.

$Tp_2^T=As^T+Bp_1^T$ Solving this system of equation returns the other part of the parity. Its time complexity is

$$(M-1)\mathrm{RCS}[L]+(2M-3)\mathrm{XOR}[L],,$$

with operations that can be calculated in place.

Decoder The LDPC decoder runs a sum-product algorithm in order to decode corrupted codewords, divided in three parts: bit-to-check messages, check-to-bit messages and parity check.

Check-to-bit message Check-to-bit messages are the most computational expensive, this is because the calculation of the $\phi()$ function can be challenging as $\phi(x)=-\log(\tanh(x/2))$. The time complexity of check-to-bit message is

$$\mathrm{ITER}ME[C_{\mathrm{links}}](2\mathrm{PHI}+\mathrm{ABS}+\mathrm{ADD}+\mathrm{SIGN}+2\mathrm{XOR}+\mathrm{SUB}),$$

where PHI is the complexity of the $\phi()$ function, SIGN is the signum function, and $E[C_{\mathrm{links}}]$ is the average number of links from any check nodes to their respective bit nodes. ITER is the number of iterations used in the decoder. The memory complexity is $ME[C_{\mathrm{links}}]$ LLRs storage size.

Bit-to-Check message Bit-to-check messages are simpler, but as bit nodes exist in bigger numbers, they have modest complexity. The time complexity is

$$\mathrm{ITER}N(E[B_{\mathrm{links}}](\mathrm{ADD}+\mathrm{SUB})+\mathrm{SIGN}),$$

where $E[B_{links}]$ is the average number of links from any bit nodes to their respective check nodes. The memory complexity is $NE[B_{\text{links}}]$ LLRs storage sizes.

Parity check The parity check simply implements Hv^T, which is a sparse binary matrix by binary vector multiplication. The time complexity is

$$\text{ITER}ME[C_{\text{links}}]XOR,$$

and the memory complexity is M syndrome bits.

Constellation mapper **Gray-coded QAM mapper** The constellation mapper for G.hn has time complexity of

$$B_{\text{length}}E[b_x](2(\text{AND}+\text{SHIFT}+\text{SUB})+2(\text{SHIFT}+\text{SUB}+\text{ABS}+\text{MULT})), \quad (3.10)$$

and requires no memory.

LLR constellation demapper The LLR calculation can be very costly, having a time complexity of

$$B_{\text{length}}(2^{E}[b_x](2\text{SUB}+2\text{MULT}+ADD)+E[b_x]*(2*MIN[2^{(}E[b_x]-1)]),, \quad (3.11)$$

this is because we have to compare every point in the constellation to find the minimum for both 0 and 1. This could be alleviated if an algorithm could find these minimum in a clever way without testing all the points. Such an algorithm exists, but it was not implemented. Although there is a term which is exponential to $E[b_x]$, b_x can be at most 12, limiting the exponential growth of complexity.

3.2.2.3 Remarks

A fair comparison is not well defined for complexity as we have suppose weights for each function. For example, when doing Galois Field multiplications in software, a processor may have a instruction to implement Galois Field multiplications directly, or when the instruction is not present, one may implement it in terms of another functions, for example using logarithm tables for finite fields. Thus the cost could be one instruction or many, depending on the hardware. This is also valid for different functions. In some systems, a addition may have lower cost than a multiplication, in other they may have the same cost, and so on.

Chapter 4

Conclusion

This work provides a comparison of two standards forward error correction schemes, RS+TCM and LDPC, as former candidates for the G.fast standard. It was shown that considering sheer error correction, RS+TCM falls behind most LDPC codes proposed in G.hn. Even when considering LDPC codes with similar codes rates, RS+TCM is outperformed in all cases, with the difference getting larger at higher SNRs, where G.fast is expected to function.

LDPC is also more flexible in terms of decoding, with a variety of decoding algorithms. Basically the difference among different algorithms is that they trade-off complexity for error correction performance. More complex algorithms produces better results. Even maintaining the same decoding algorithm, one can adjust the decoder complexity and performance by just adjust the maximum number of decoder iterations. The more the iterations, the more complexity and performance. This freedom of choice has to be considered as an advantage, as vendors have to compete to offer the best trade-off.

In the other hand, the classical RS+TCM has some adjustable parameters, like parity size, shortening size and interleaver depth, but the decoding algorithm is just the same. For example, shortening decreases the code rate by inserting zeros, but this does not reduce the decoder complexity. The error correcting performance of RS is the same despite the decoding algorithm. Trellis-coded modulation is fixed, and the optimum algorithm is already the Viterbi algorithm, leaving no opening for any trade-off. The advantage of this lack of freedom is that it is more easy to develop specialized hardware or instructions to implement them.

One may argue that LDPC does not include interleaving and thus is more vulnerable to noise bursts than RS+TCM. This is not a great problem as G.fast foresees the use of retransmissions. As G.fast symbol rate is 12 times faster than VDSL2 (48k symbols/s vs 4k symbols/s), its symbols have the shorter duration of approximately 20 μs against 250 μs of VDSL2. This incur in more vulnerability to impulse noise, with the possibility of losing entire symbols at once. With higher symbol rates, the retransmission rates are higher too, resulting in lower latencies, which is the major concern of retransmissions, as described in [6].

As technology advanced, LDPC, that was once considered too computationally expensive, is now affordable and it is used in a vast amount of today standards, including Wi-Fi's 802.11 family of standards, digital video broadcasting DVB-S2 and more. In these standards, LDPC are implemented to efficiently to run on rates of over 1 Gbps, thus there is no complexity concern on LDPC for G.fast.

Additionally, this work can be used educationally in channel coding teaching, as its is a major overview of practical of both block codes and convolutional codes, and also both classical and modern approaches. Reed-Solomon are classical block codes, LDPC is a modern capacity-approaching block code. Trellis-coded modulation is the concept of convolutional coding mixed with modulation. But of course this does not mean that this is a complete overview of channel coding, it just provide some concrete examples and implementations of some of the major codes. The implementation was written in a accessible programming language, C, although it is not fully optimized, it could run simulations without major concerns. The program is even capable of multi-thread simulation using several cores at once.

4.1 Future works

In order to fully evaluate the complexity of each scheme, an optimized implementation in hardware would be necessary. Then we could evaluate the trade-off of complexity versus performance, accounting for hardware costs and if an implementation could reach the desired rates. A good approach would be an implementation in FPGA, on which we can develop hardware in a flexible manner.

Bibliography

[1] G.9701, "ITU-T:Fast Access to Subscriber Terminals(FAST)," 2014.

[2] I. Reed and G. Solomom, "Polynomial codes over certain finite fields," *SIAM Journal of Applied Math*, vol. 8, pp. 300–304, 1960.

[3] G. Ungerboeck, "Channel coding with multilevel/phase signals," *Information Theory, IEEE Transactions on*, vol. 28, no. 1, pp. 55–67, Jan 1982.

[4] R. G. Gallager, "Low-Density Parity-Check Codes," Ph.D. dissertation, Cambridge, MA: MIT Press, 1963.

[5] S.-Y. Chung, G. D. Forney, T. J. Richardson, and R. Urbanke, "On the Design of Low-Density Parity-Check Codes within 0.0045 dB of the Shannon Limit," *IEEE Communications Letters*, vol. 5, no. 2, pp. 58–60, Feb. 2001.

[6] J. Neckebroek, M. Moeneclaey, M. Guenach, M. Timmers, and J. Maes, "Comparison of error-control schemes for high-rate communication over short DSL loops affected by impulsive noise," in *Communications (ICC), 2013 IEEE International Conference on*, June 2013, pp. 4014–4019.

[7] S. Lin and D. C. Jr., *Error Control Coding: Fundamentals and Applications.* Prentice-Hall, 1983.

[8] R. H. Morelos-Zaragoza, *The Art of Error correcting Coding.* John Wiley e Sons, Ltd., 2002.

[9] J. C. Moreira and P. G. Farrell, *Essentials of Error-Control Coding.* John Wiley e Sons, Ltd., 2006.

[10] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 28, 1948.

[11] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, pp. 68–79, March 1960.

[12] E. R. Berlekamp, "On decoding binary Bose-Chaudhuri-Hocquenghem codes," *IEEE Trans. Inf. Theory*, vol. IT-11, pp. 577–580, October 1965.

[13] J. L. Massey, "Step-by-step decoding of the Bose-Chaudhuri-Hocquenghem codes," *IEEE Trans. Inf. Theory*, vol. IT-11, pp. 580–585, October 1965.

[14] D. MacKay, "Good error-correcting codes based on very sparse matrices," *Information Theory, IEEE Transactions on*, vol. 45, no. 2, pp. 399 –431, mar 1999.

[15] S. J. Johnson, *Iterative Error Correction Turbo, Low-Density Parity-Check and Repeat-Accumulate Codes.* Cambridge, 2009.

[16] T. J. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 638–656, Feb. 2001.

[17] G.993.2, "ITU-T:Very high speed digital subscriber line transceivers 2 (VDSL2)," 2010.

[18] G.9960, "ITU-T:Unified high-speed wire-line based home networking transceivers - System architecture and physical layer specification," 2010.

[19] S. Myung, K. Yang, and J. Kim, "Quasi-Cyclic LDPC Codes for Fast Encoding," *IEEE Trans. Inf. Theory*, vol. 51, no. 8, pp. 2894–2900, Aug. 2005.