Improving the convergence rate of the iterative Parity-check Transformation Algorithm decoder for Reed–Solomon codes

Peter C. Brookstein

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, May 2018

Declaration

I declare that this dissertation is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other university.

Signed this ____ day of _____ 20__

_____

Peter C. Brookstein

Abstract

This masters by research dissertation contributes to research in the field of Telecommunications, with a focus on forward error correction and improving an iterative Reed-Solomon decoder known as the Parity-check Transformation Algorithm (PTA). Previous work in this field has focused on improving the runtime parameters and stopping conditions of the algorithm in order to reduce its computational complexity. In this dissertation, a different approach is taken by modifying the algorithm to more effectively utilise the soft-decision channel information provided by the demodulator. Modifications drawing inspiration from the Belief Propagation (BP) algorithm used to decode Low-Density Parity-Check (LDPC) codes are successfully implemented and tested. In addition to the selection of potential codeword symbols, these changes make use of soft channel information to calculate dynamic weighting values. These dynamic weights are further used to modify the intrinsic reliability of the selected symbols after each iteration.

Improvements to both the Symbol Error Rate (SER) performance and the rate of convergence of the decoder are quantified using computer simulations implemented in MATLAB and GNU Octave. A deterministic framework for executing these simulations is created and utilised to ensure that all results are reproducible and can be easily audited. Comparative simulations are performed between the modified algorithm and the PTA in its most effective known configuration (with $\delta = 0.001$). Results of simulations decoding half-rate RS(15,7) codewords over a 16-QAM AWGN channel show a more than 50-fold reduction in the number of operations required by the modified algorithm to converge on a valid codeword. This is achieved while simultaneously observing a coding gain of 1dB for symbol error rates between 10^{-2} and 10^{-4}.

Acknowledgements

I would like to acknowledge and thank Dr. D.J Versfeld for his supervision, and many valuable discussions along the way, even after accepting a new position at another university.

I would also like to thank and acknowledge Prof. Ken Nixon for stepping up to provide invaluable mentorship and supervision.

I am also grateful to Prof. Ling Cheng for providing his time and key advice when needed.

Finally, to my family & friends for helping, motivating and supporting me along the way, with a special thank you to Lauren and my mother, Jenny, for taking the time to help proofread the final product.

Contents

List of Figures

List of Tables

Nomenclature

ABP Adaptive Belief Propogation

APP a-posteriori Probability

AWGN Additive White Gaussian Noise

BCH Bose-Chaudhuri-Hocquenghem

BEC Binary Erasure Channel

BER Bit Error Rate

BPSK Binary Phase Shift Keying

BP Belief Propagation

FEC Forward Error Correction

HDPC High-Density Parity-Check

HD Hard-Decision

KV Koetter-Vardy

LDPC Low-Density Parity-Check

LLR Log Likelihood Ratio

MDS Maximum Distance Seperable

MPI Message Passing Interface

PRNG Pseudo Random Number Generator

PSD Power Spectral Density

PSK Phase Shift Keying

PTA Parity-check Transformation Algorithm

QAM Quadrature Amplitude Modulation

RS Reed-Solomon

SD Soft-Decision

SER Symbol Error Rate

SISO Soft-Input-Soft-Output

SNR Signal-to-Noise Ratio

SPA Sum Product Algorithm

SSH Secure Shell

Chapter 1

Introduction

Error correcting codes are an important component of any modern communications or data-storage system. When utilised correctly they can help ensure data resiliency across unreliable channels and increase the effective data throughput of a system [1, 2].

Of particular interest are Reed-Solomon (RS) codes [3], a non-binary cyclic code with an optimal minimum Hamming distance d_{min} between codewords. RS codes, discovered by Irving Reed and Gustave Solomon in 1960, are extensively deployed in many real-world communications and storage systems [2, 4, 5], and, while they are considered a *classical* error correction code, their popularity and optimal performance at short code lengths continues to make them an interesting target for research.

The problem of effectively decoding RS codes has been studied for decades, and it has been shown that improvements to RS decoders allow for errors to be corrected beyond the theoretical capability of the code [6, 7]. Additionally, decoders designed to make use of Soft-Decision (SD) channel information from the demodulator have shown even greater performance gains [8, 9] over traditional Hard-Decision (HD) algorithms such as the Berlekamp-Massey algorithm [10].

Research efforts into techniques related to improving the performance of Forward Error Correction (FEC) decoders are not limited to error-correction capability. Methods of reducing the computational complexity of decoders have allowed for their practical use in lower power environments, or with longer, more efficient code lengths [7, 11, 12]. As a result, improved techniques for decoding RS codes is an interesting and active research topic in the realm of FEC codes.

1.1 Research Problem

The Parity-check Transformation Algorithm (PTA) as described in [13] is a modern, iterative, soft-decision RS decoder, capable of surpassing the error-correction performance of other the soft-decision decoders such as the Koetter-Vardy (KV) algorithm [9] for short code lengths.

The impressive error-correction performance of the PTA comes at the cost of high computational complexity, manifesting in the significant number of iterations required for the algorithm to converge at low Signal-to-Noise Ratio (SNR) values [14]. Research into improving the computational performance of the PTA has been attempted before in [14], yielding positive results.

It is shown in Chapter 3 of this dissertation that the PTA is analogous to the Low-Density Parity-Check (LDPC) Bit–Flipping algorithm [15], a simple binary hard-decision decoder. It is also known that soft-decision decoders can perform significantly better than hard-decision decoders [8, 16], and thus, the limited use of soft channel information by the PTA conceivably limits its potential performance.

1.1.1 Research Question

With this potential limitation of the PTA in mind, the research question addressed in this dissertation is:

To what extent does improving the utilisation of available soft-decision information in the iterative Parity Transformation Algorithm decoder for Reed-Solomon codes have an effect on the error correction and computational performance of the algorithm.

In order to answer this question, algorithmic modifications to the existing PTA decoder are proposed to maximise the decoder's use of soft-decision information during each iteration. Computer simulations are used to study and quantify the relative performance of the modified decoder and the original PTA.

The resulting decoder draws inspiration from the Sum Product Algorithm (SPA) (also known as Belief Propagation (BP)) used in the decoding of LDPC codes [15]. However, unlike the SPA, the modified algorithm is adapted for use with non-binary RS codes and utilises the symbol level *distance metric* reliability matrix [17] as its source of soft channel information instead of bit-level Log Likelihood Ratios (LLRs). This makes the modified decoder a drop-in replacement for the PTA.

1.1.2 Gaps in Existing Research

It is known how effective message-passing decoders are for the efficient decoding of LDPC codes [18], and algorithms such as the Adaptive Belief Propogation (ABP) [19] decoder are a good example of successfully applying the concept to High-Density Parity-Check (HDPC) codes such as RS codes.

Existing research into improving the performance of message passing decoders for HDPC codes has focused techniques to reduce the density of the parity-check matrix [20], or to take advantage of the cyclic nature of RS codes [19]. Both techniques utilise bit-level LLRs for soft channel information instead of operating at the symbol level like the PTA.

1.1.3 Research Relevance

To summarise, the research being conducted is relevant because:

1. Reed-Solomon codes are an important fixture in modern communication systems.
2. The PTA shows good potential as an iterative RS decoder, but is hampered by high decoding complexity.
3. Belief Propagation is a powerful, modern technique used in decoding the near Shannon limit LDPC codes, but struggles with HDPC codes such as RS, due to rapid error propagation. Techniques to work around that could result in powerful RS decoders, as shown with the ABP algorithm.
4. Algorithms such as the ABP operate on the bit level, potentially increasing the number of operations required to decode a codeword [14].

1.1.4 Research Contribution

The primary goal is to compare the relative performance of the original PTA [13] against a modified version of the algorithm introduced in this dissertation in order to quantify any performance penalty incurred by the PTA for ignoring some of the soft-decision information available to it.

This research contributes the following:

1. A new iterative soft-decision Reed-Solomon decoding algorithm.
2. A thorough performance analysis of both the PTA and the resultant modified decoder.
3. A set of possible future work around further optimisation of the algorithm.

1.2 Dissertation Outline

The remainder of this dissertation is broken up into several chapters:

Chapter 2 - *Background.* This chapter focuses on understanding the principles behind forward error-correction techniques, and, in particular, RS codes. The PTA is introduced, along with other modern soft-decision and iterative decoders for RS codes. Additional background information is covered in preparation for later chapters.

Chapter 3 - *Research Methodology.* This chapter covers the process used to answer the research question. The experimental set-up used for reproducibly comparing the performance of the algorithms is discussed in detail. The deterministic data-generation pipeline, and simulation environment, developed to quickly iterate on the algorithm while utilising the *CeTAS Compute Cluster* are also discussed.

Chapter 4 - *Parity-check Transformation Algorithm Analysis.* A thorough analysis of the PTA is performed. The PTA is modelled as a message-passing algorithm, highlighting inefficiencies in its use of available soft-decision information. The time complexity of each iteration of the algorithm is derived in order to understand its overall computational complexity, providing a solid base for interpreting experimental results.

Chapter 5 - *Modified Algorithm / Improvements.* The modifications made to the PTA are introduced, motivated and discussed in this chapter. A detailed example of the modified algorithm decoding a codeword is provided, along with a detailed analysis of the algorithm's time complexity.

Chapter 6 - *Results.* This chapter reveals the results of the computer simulations and experiments comparing the performance of the PTA to the new modified version of the algorithm. Additionally, a simple optimisation of a new tunable parameter introduced by the modified algorithm is performed.

Chapter 7 - *Conclusion.* This chapter presents the conclusion of the dissertation, along with potential future work related to the modified algorithm.

Chapter 2

Background

This chapter covers the fundamentals of Reed-Solomon (RS) error-correction codes along with the existing literature around decoding them. The principles behind the Belief Propagation algorithm used to decode Low-Density Parity-Check (LDPC) codes are introduced along with the Parity-check Transformation Algorithm (PTA) and Adaptive Belief Propogation (ABP) decoders for decoding non-binary RS codes.

Forward Error Correction (FEC) coding is a technique whereby additional information, called parity, is appended to a message before transmitting it over a noisy channel. This parity information can help the receiver correct errors induced by the channel without requiring the sender retransmit the data.

There are two major categories of FEC codes, *convolutional* codes and *block* codes. Convolutional codes operate on arbitrary length streams of information symbols, while block codes work on fixed-size blocks of k information symbols at a time.

Furthermore, codes can be classified by the symbols they use. *Binary* codes are defined over the alphabet of $\mathcal{A} \in \{0, 1\}$, while *non-binary* (or q-ary codes) are defined over an alphabet of q symbols. Binary codes can be seen as a simple sub-type of q-ary code, with $q = 2$.

RS codes are a type of *non-binary block code*, and, before understanding their construction, the fundamentals of linear block codes along with several basic concepts and definitions are first introduced.

2.1 Channel Coding Fundamentals

A (n, k) linear block code maps k message symbols to n (encoded) codeword symbols. The number of parity symbols per codeword introduced by a code is therefore given by

$$h = n - k, \tag{2.1}$$

and the the *rate* of a code is defined as the ratio:

$$R = \frac{k}{n}. \tag{2.2}$$

By decreasing the code rate R defined in (2.2), the error-detection and correction capability of a code is improved, however this is done at the expense of additional overhead caused by the addition of extra parity symbols.

Definition 2.1. The *Hamming weight* of a vector is the number of elements in the vector which are non-zero.

Definition 2.2. The *Hamming distance* between any two vectors $\mathbf{v}$ and $\mathbf{u}$, is the number of corresponding positions in which the two vector's symbols differ, and is denoted $d(\mathbf{v}, \mathbf{u})$.

Definition 2.3. The *minimum distance* of a code, denoted d_{min} is the smallest Hamming distance between any two valid codeword vectors $\mathbf{v}$ and $\mathbf{u}$ within the code $\mathcal{C}$, where $\mathbf{v}, \mathbf{u} \in \mathcal{C}$.

The minimum distance of a code is an important property, as it defines the error-detection and correction power of the code. Because any two valid codewords within the code $\mathcal{C}$ must differ in at least d_{min} positions, any number of errors up to $d_{min} - 1$ is guaranteed to produce an invalid codeword, and are thus easily detectable.

It follows geometrically that it is possible to *correct* up to t errors, where

$$t = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor, \tag{2.3}$$

by finding the codeword with minimum Hamming distance from the received vector.

The notation $\lfloor x \rfloor$ introduced in (2.3) is the *floor* function. The floor function rounds a real number input x down to its nearest integer value.

Definition 2.4. The *Singleton bound* is the upper bound on the size of a q-ary code, with block length n, and minimum distance d_{min}. It is defined as:

$$A_q(n, d_{min}) \leq q^{n-d_{min}+1}. \tag{2.4}$$

For a q-ary code, with a message length of k, this can be written as:

$$q^k \leq q^{n-d_{min}+1}, \tag{2.5}$$

which can in turn be simplified to:

$$k \leq n - d_{min} + 1. \tag{2.6}$$

Definition 2.5. A *Maximum Distance Separable* (MDS) code is able to fulfil the Singleton bound with equality [21]. This means (2.6) can be rewritten to express minimum distance of an MDS code as:

$$d_{min} = n - k + 1. \tag{2.7}$$

Combining (2.3) and (2.7), it can be seen that MDS codes can detect up to $n - k$ errors, and correct up to t errors where:

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor. \tag{2.8}$$

As a result MDS codes are capable of correcting the largest number of errors for a given code rate R.

Reed-Solomon codes, as defined in Section 2.3 are MDS codes [2], making them optimal for a given code rate.

2.2 Algebraic Coding Theory

Several algebraic coding concepts critical to the understanding of RS codes are introduced in this section.

Definition 2.6. A *field* $\mathbb{F}$ is a set of elements on which the operations of *multiplication* and *addition* are defined along with their inverses *division* and *subtraction*, and which behave analogously to the equivalent operations on real numbers [22].

Additionally, a field must satisfy the following properties:

1. Closure under addition, $a + b \in \mathbb{F}$ and multiplication $a \cdot b \in \mathbb{F}$ where $a, b \in \mathbb{F}$
2. The multiplicative identity denoted 1 must exist within $\mathbb{F}$ such that $a \cdot 1 = a$ for all elements $a \in \mathbb{F}$.
3. The additive identity denoted 0 must exist such that $a + 0 = a$ for all elements $a \in \mathbb{F}$.
4. All elements have an additive inverse such that for an element $a \in \mathbb{F}$ there exists another element b also in $\mathbb{F}$ where $a + b = 0$. The element b is denoted as $-a$.
5. All elements, except 0 have a multiplicative inverse where for $a \in \mathbb{F}$ there exists another element b also in $\mathbb{F}$ where $a \cdot b = 1$. The element b is denoted as a^{-1}.
6. Operations must be associative $(a \cdot b) \cdot c = a \cdot (b \cdot c)$, and commutative $a \cdot b = b \cdot a$ for all $a, b, c \in \mathbb{F}$.

Definition 2.7. A *Galois field*, or *finite field* is a field which contains only a finite number of elements and is denoted $\mathbb{F}_{p^m}$ or $GF(p^m)$. Where p is a prime number.

Definition 2.8. An *irreducible polynomial* is a polynomial of degree m over the field $\mathbb{F}$ which is not divisible by any other polynomial in $\mathbb{F}$ with a degree less than m and greater than 0 [23].

Definition 2.9. A *primitive polynomial* is an irreducible polynomial $p(x)$ of degree m over $\mathbb{F}_q$, where the smallest positive integer for which $p(x)$ divides $(x^n - 1)$ is $n = q^m - 1$ [23].

Primitive polynomials are fundamental in construction Galois Fields, where multiplication and addition over a field $GF(q)$ is performed modulo the field's primitive polynomial.

Let α be a primitive element in $GF(2^m)$.

An example of a Galois Field, $GF(2^3)$, represented in index, polynomial, binary and decimal form is shown in Table 2.1.

Table 2.1: Galois Field for Primitive Polynomial $\alpha^4 + \alpha + 1 = 0$

Index	Polynomial	Binary	Decimal
0	0	0 0 0 0	0
α^0	1	0 0 0 1	1
α^1	α	0 0 1 0	2
α^2	α^2	0 1 0 0	4
α^3	α^3	1 0 0 0	8
α^4	$\alpha + 1$	0 0 1 1	3
α^5	$\alpha^2 + \alpha$	0 1 1 0	6
α^6	$\alpha^3 + \alpha^2$	1 1 0 0	12
α^7	$\alpha^3 + \alpha + 1$	1 0 1 1	11
α^8	$\alpha^2 + 1$	0 1 0 1	5
α^9	$\alpha^3 + x$	1 0 1 0	10
α^{10}	$\alpha^2 + \alpha + 1$	0 1 1 1	7
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	1 1 1 0	14
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	1 1 1 1	15
α^{13}	$\alpha^3 + \alpha^2 + 1$	1 1 0 1	13
α^{14}	$\alpha^3 + 1$	1 0 0 1	9
α^{15}	1	0 0 0 1	1

2.2.1 Generator Matrix

The generator matrix $\mathbf{G}$ of a (n, k) linear block code consists of the k linearly independent codewords that form a basis of the code $\mathcal{C}$ as its rows. As a result of this, all codewords defined in $\mathcal{C}$ can be represented as a linear combination of the rows in $\mathbf{G}$ [22].

It is often desirable to use the *systematic* form of a code, where the message symbols are left unaltered by encoding, and parity symbols are appended to form a codeword.

Given the message vector:

$$\mathbf{m} = \begin{bmatrix} m_1 & m_2 & m_3 & \cdots & m_k \end{bmatrix}, \tag{2.9}$$

the encoded *systematic* codeword is of the form:

$$\mathbf{c} = \Big[\underbrace{p_1 \quad p_2 \quad \cdots \quad p_{n-k}}_{\textbf{parity symbols}} \quad \underbrace{m_1 \quad m_2 \quad m_3 \quad \cdots \quad m_k}_{\textbf{message symbols}}\Big]. \tag{2.10}$$

The general form of a *systematic* generator matrix is:

$$\mathbf{G} = [P|I_k], \tag{2.11}$$

where I_k is the k dimensional identity sub-matrix, and P is the *parity* sub-matrix, shown in expanded form below:

$$\mathbf{G} = \Big[\underbrace{\begin{matrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n-k} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n-k} \\ \vdots & \vdots & \ddots & \vdots \\ p_{k,1} & p_{k,2} & \cdots & p_{k,n-k} \end{matrix}}_{P} \quad \underbrace{\begin{matrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{matrix}}_{I_k}\Big]. \tag{2.12}$$

Encoding

Encoding a linear block code using the systematic generator matrix is performed by simple matrix multiplication of the message vector $\mathbf{m}$ and generator matrix $\mathbf{G}$ as shown below:

$$\mathbf{c} = \mathbf{m} \cdot \mathbf{G}. \tag{2.13}$$

2.2.2 Parity-check Matrix

A linear block code can also be described by its parity-check matrix $\mathbf{H}$. The parity-check matrix is the *dual* code of the generator matrix $\mathbf{G}$ [2], and defines the set of linear parity-check equations a vector must satisfy to be considered valid codeword as seen in:

$$\mathbf{c} \cdot \mathbf{H}^T = \mathbf{0}. \tag{2.14}$$

Given the systematic generator matrix in (2.11), the *parity-check matrix* for the code is given as:

$$\mathbf{H} = [I_{n-k}|P^T]. \tag{2.15}$$

2.2.3 Tanner Graphs

Tanner graphs, a type of bipartite graph named after Michael Tanner [22], are a useful way to represent the parity-check matrix of a linear block code. The graphs consist of two types of node, *check nodes* and *variable nodes*. As a bipartite graph, edges exist only between nodes of different types, and never amongst nodes of the same type.

Tanner graphs are drawn using a simple rule, wherever a non-zero element in a matrix appears, the edge between the check node i and variable node j is drawn. For non-binary systems, this edge includes a magnitude/co-efficient in position i, j in the matrix.

For example, given the parity-check matrix:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 6 & 4 & 3 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 6 & 5 & 2 \\ 0 & 0 & 0 & 1 & 7 & 5 & 3 \end{bmatrix}, \tag{2.16}$$

a non-binary Tanner graph can be constructed as seen in Figure 2.1.

2.3 Reed-Solomon Codes

RS codes are a popular class of non-binary linear block code, invented in 1960 by Irving S. Reed and Gustave Solomon [3]. It was later shown that RS codes are a non-binary sub-class of Bose-Chaudhuri-Hocquenghem (BCH) codes [10].

RS codes are Maximum Distance Seperable (MDS) codes which enjoy wide real-world usage, due to their optimal error-correction capacity. They are particularly good at correcting bursts of errors, and perform well as erasure codes [23].

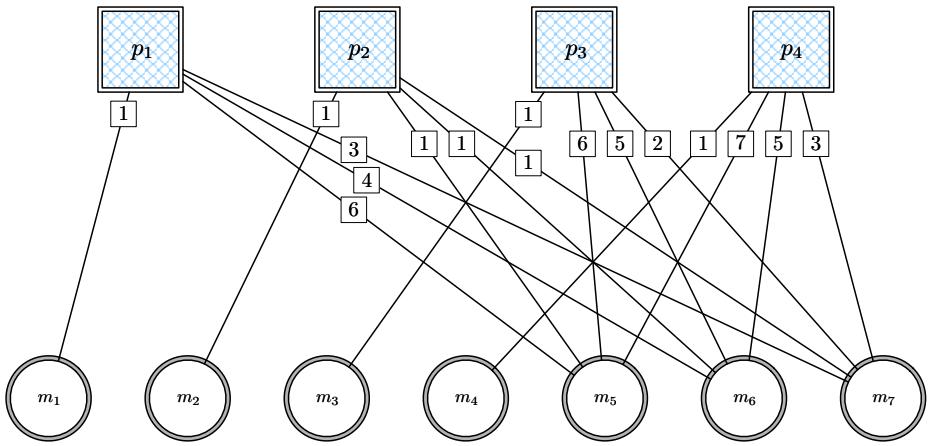


Figure 2.1: Tanner graph representing the parity-check matrix described in (2.16). The square nodes shown at the top (p_1 - p_4) are the graph's *check nodes*, while the round *variable nodes* (m_1 - m_7) can be seen at the bottom.

2.3.1 Properties

As a non-binary q-ary code, operations are performed on an alphabet $\mathcal{A}$ of multi-bit symbols defined over $\mathcal{A} = GF(q)$, where $q = 2^m$ and m is the number of bits used to represent a symbol. A result of this is the fixed length n of a RS codeword is defined as:

$$n = p^m - 1. \tag{2.17}$$

As an MDS code, RS codes are capable of correcting up to t errors, where:

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor, \tag{2.18}$$

$$k = n - 2t, \tag{2.19}$$

$$d_{min} = 2t + 1. \tag{2.20}$$

2.3.2 Code Construction

RS codes are defined over a Galois Field, of m-bit symbols, i.e. $GF(2^m)$. The codes are constructed from a generator polynomial:

$$g(x) = (x + \alpha^b)(x + \alpha^{b+1}) \dots (x + \alpha^{b+2t-1}), \tag{2.21}$$

consisting of $n - k = 2t$ factors, with roots which are *consecutive* elements in the GF. This ensures the maximum d_{min} property of the code [2].

The choice of b can be arbitrary, however careful selection can have an impact on the complexity of hardware design for encoding and decoding RS codes [12].

Example 2.1. Find a generator polynomial for a Reed-Solomon code using $m = 4$ bit symbols and capable of correcting $t = 2$ errors.

For $m = 4$ bit symbols, the code is defined over the field $GF(2^4)$, with a codeword length of:

$$\begin{aligned} n &= 2^m - 1 \\ &= 15. \end{aligned}$$

Given these requirements, a potential code is defined by the generator polynomial:

$$\begin{aligned} g(x) &= (x + \alpha^0)(x + \alpha^1)(x + \alpha^2)(x + \alpha^3) \\ &= \alpha^0 x^4 + \alpha^{12} x^3 + \alpha^4 x^2 + \alpha^0 x + \alpha^6. \end{aligned} \tag{2.22}$$

Using Table 2.1, (2.22) can be written in decimal form as:

$$\begin{aligned} g(x) &= (x + 1)(x + 2)(x + 4)(x + 8) \\ &= x^4 + 15x^3 + 3x^2 + x + 12. \end{aligned} \tag{2.23}$$

2.3.3 Encoding

As a linear block code, the encoding procedure for RS codes envisioned by Reed and Solomon is straightforward [3].

Given the message vector:

$$\mathbf{m} = [m_0, m_1, m_2 \cdots m_{k-1}],$$

the message polynomial is defined as:

$$m(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \ldots m_1 x + m_0. \tag{2.24}$$

The parity information is calculated by multiplying the message polynomial $m(x)$ by x^{n-k}, and dividing the result by the generator polynomial $g(x)$ over the field $GF(q^m)$. This produces a quotient $s(x)$ and the parity information $r(x)$.

A systematically encoded codeword can be seen as the equation:

$$C(x) = m(x) \cdot x^{n-k} + r(x), \tag{2.25}$$

where $r(x)$ is the parity information added to the end of the message in order to form the codeword. Multiplication by x_{n-k} has the effect of shifting the message by $(n-k)$ positions to make way for the parity information.

2.3.4 Decoding

The study of RS decoders has been a field of active research for decades, and has led to the discovery of many algorithms and techniques for efficiently decoding codewords. Decoders can be broadly broken down into either Hard-Decision (HD) decoders, or Soft-Decision (SD) decoders based on the decoder's ability to utilise soft channel information provided by the demodulator.

Hard-Decision Decoders

Classical examples of HD decoders include the Berlekamp-Massey algorithm and the Euclidean algorithm [2]. HD decoders are provided symbol values from the demodulator in a static form, with no additional information about the relative reliability or quality of the received symbols. As a result, traditional RS decoders are limited in the sense that they are only capable of correcting $t = \frac{n-k}{2}$ errors.

Some more modern HD algorithms allow for *beyond minimum distance* decoding of received codewords. One example, the Guruswami-Sudan decoder [6], an algebraic list decoder capable of decoding up to $t = n - 1 - \lfloor \sqrt{k-1} \cdot n \rfloor$ errors. It achieves this improvement by returning a list of *potential* codewords within the new extended decoding radius.

Soft-Decision Decoders

It has been shown that utilising soft-decision or channel-measurement information provided by the demodulator can double the error-correction capabilities of a code [8].

Soft-decision decoding of RS codes can be extremely effective. Decoders such as the Adaptive Belief Propagation (ABP) algorithm [19], Koetter-Vardy (KV) [9], Parity-check Transform Algorithm (PTA) [13] and others described in [20] and [24] can show a significant gain in error-rate performance over regular HD decoders.

While these decoders improve the error-correction abilities of a code, the cost in terms of computational complexity can be significant [9, 25, 13].

2.4 Low-Density Parity-Check Codes

LDPC codes [15], invented by Robert Gallager in 1960, are a type of binary linear block code constructed using large sparse bipartite graphs [18]. LDPC codes are interesting due to their ability to provide near Shannon capacity performance [1, 22].

The low-density nature of the LDPC parity-check matrix facilitates iterative message passing decoders such as the Sum Product Algorithm (SPA), also known as Belief Propagation (BP).

Unlike RS codes, which can be effective at short code lengths, LDPC codes can require extremely long codewords in order to achieve their impressive error-correction gains [11].

2.5 Iterative Message-Passing Decoders

Iterative message-passing algorithms such as the Sum Product Algorithm (SPA) or Bit-Flipping algorithm [18] are popular methods used to decode modern LDPC codes.

These decoders operate through rounds of exchanging messages over a bipartite graph [26]. During each round, messages are sent from *variable nodes* to their connected *check nodes* where they are processed before the resulting outgoing messages are sent back to the connected *variable nodes*. Completely decoding a codeword generally requires many of these rounds, or iterations.

2.5.1 Log Likelihood Ratios

The bit probabilities used in the SPA described in the next section are represented as Log Likelihood Ratios (LLRs). LLRs are useful for combining both the bit value and probability into a single value. Additionally, LLRs help reduce computational complexity as probabilities can be added instead of multiplied together. The LLR of bit j in a message vector $\boldsymbol{x}$ is:

$$L(x_j) = log_e\left(\frac{p(x_j = 0)}{p(x_j = 1)}\right). \tag{2.26}$$

The sign of $L(x_j)$ represents the state of the bit. If $p(x_j = 0) > p(x_j = 1)$ then $L(x_j)$ is positive (a hard-decision value of $\mathbf{0}$), and if $p(x_j = 0) < p(x_j = 1)$ it will be negative (a hard-decision value of $\mathbf{1}$). As the difference between $p(x_j = 0)$ and $p(x_j = 1)$ grows, so does the magnitude of $L(x_j)$, and with it the confidence in the current sate of the bit.

Working backwards, it is possible to find the probability that a given bit j is a 1 using:

$$\begin{aligned} p(x_j = 1) &= \frac{p(x_j = 1)/p(x_j = 0)}{1 + p(x_j = 1)/p(x_j = 0)} \\ &= \frac{e^{-L(x_j)}}{1 + e^{-L(x_j)}}, \end{aligned} \tag{2.27}$$

or a 0 using:

$$p(x_j = 0) = \frac{p(x_j = 0)/p(x_j = 1)}{1 + p(x_j = 0)/p(x_j = 1)} = \frac{e^{L(x_j)}}{1 + e^{L(x_j)}}. \tag{2.28}$$

2.5.2 Sum Product Algorithm

The SPA or Belief Propagation (BP) algorithm was first introduced by Gallager as a near-optimal graph-based algorithm to decode LDPC codes [15]. The algorithm makes use of soft-decision information in the form of bit probabilities represented as LLRs. Operations within the SPA occur locally at the nodes within the graph, allowing the SPA to be applied to graphs that contain cycles [22].

The input bits from the channel are known as the *a priori* probabilities, and form the initial state of the variable nodes on the graph.

The goal of the SPA is to calculate the maximum *a posteriori* probability (MAP) for each bit j in the received codeword vector x as shown in:

$$P_i = P\{x_i = 1|N\}, \tag{2.29}$$

where N is the event that x is a valid codeword.

This is achieved by iteratively computing an approximation of the MAP for each bit at each of its connected check nodes in the graph. This approximation is known as the *extrinsic* information for the bit.

The *extrinsic* information calculated by check node i and sent to bit (or variable node) j in iteration t is:

$$E_{j,i}^{(t)} = log\left(\frac{1 + \Pi_{i' \in B_j, i' \neq i} \tanh(L_{j,i'}^{(t)}/2)}{1 - \Pi_{i' \in B_j, i' \neq i} \tanh(L_{j,i'}^{(t)}/2)}\right). \tag{2.30}$$

The notation of B_j, introduced in (2.30), is the set of all variable nodes connected to check node j, and L is the variable node vector initialised with the *intrinsic* information for each bit as LLRs.

After each iteration, the *extrinsic* information from all connected check nodes is added to the current *intrinsic* information for each bit:

$$L_{j,i}^{(t+1)} = \sum_{j' \in A_i, j' \neq j} E_{j',i}^{(t)} + L_{j,i}^{(t)}, \tag{2.31}$$

where A_i is the set of all check nodes connected to variable node i.

There is no guarantee that the algorithm will converge, and so an upper limit on the number of iterations allowed is defined.

The algorithm continues until either *all* of the check nodes are simultaneously satisfied, or a maximum number of iterations is reached.

2.5.3 Issues with HDPC Codes

Due to their good performance operating on LDPC codes, interest in the field of iterative message-passing decoders for decoding RS codes has led to several algorithms making use of the technique. Research into the application of iterative binary block codes such as in [16] includes several lessons on implementing iterative algorithms.

As indicated in the previous section, the effectiveness of a message-passing decoder such as the SPA is diminished when cycles exist within the graph.

Similar to LDPC codes, the parity-check matrix of a RS code can be visualised as a Tanner Graph [27] as shown in Section 2.2.3. RS codes are by nature HDPC codes, and thus do not naturally lend themselves to iterative message-passing decoders. This is due to the rapid error propagation, which occurs as a result of the increased number of cycles in their graphs [25, 20, 22].

Techniques to reduce the ill effects of the HDPC matrix have been successfully implemented for RS codes. These include the binary expansion step in the ABP algorithm [19], stochastic shifting of message symbols [20] or the re-ordering of highly dense columns within the matrix [13, 28], and can all be used to improve the performance of iterative message-passing decoders. These steps can, however, have a negative impact on the computational complexity of the decoders [14].

2.6 The Parity-check Transformation Algorithm

The PTA is an iterative soft-decision RS decoder defined in [13]. The decoder operates at the symbol level and was shown to outperform both the hard-decision Berlekamp-Massey [10] and soft-decision Koetter–Vardy [9] decoders.

The algorithm works by utilising soft symbol information to find the $\mathcal{R} = n - k$ most and $\mathcal{U} = k$ least reliable symbols within the received codeword $\mathbf{x}$. This knowledge is then used to transform the systematic parity-check matrix $\mathbf{H}$ of the code as shown in [29]. This transformation results in the high-density columns of $\mathbf{H}$ in the positions of the most reliable codeword symbols, and has the effect of prioritising the information contained in the more reliable symbols during decoding steps.

The PTA then attempts to check if each of the parity equations, defined in the new transformed $\mathbf{H}^{\gamma}$ matrix, are satisfied by the current HD symbols from the codeword. When a parity equation is satisfied, all involved symbols from the codeword have their reliability score incremented by a fixed amount. If an equation doesn't 'check' then the corresponding symbol's reliability score instead gets decremented.

The magnitude of the alteration to a symbol's reliability score is a function of a pre-defined fixed value δ.

Once all parity-check equations have been tested, the next iteration begins by finding (a potentially new) transformed parity-check matrix. The algorithm continues iterating until all parity-checks are satisfied, or a pre-defined maximum number of iterations is reached.

As a core focal point of this research, the PTA is discussed and analysed in more detail in Chapter 4.

2.6.1 The Reliability Matrix

The PTA expects each codeword to be provided as a $2^m \times n$ reliability matrix $\mathbf{R}$ from the demodulator. The index of each row in $\mathbf{R}$ corresponds to each of the symbols in $GF(2^m)$ over which the code is defined, while the columns represent each symbol in the codeword. Thus the row index of the maximum value in each column j is the hard-decision value for symbol S_j of the received codeword.

The elements in $\mathbf{R}$ consist of *reliabilities* calculated using the distance metric technique

as defined in [17], and are normalised in order for each column to sum to **1**.

The **R** matrix is defined as:

$$\mathbf{R} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ \vdots \\ 2^m-1 \end{array} \begin{array}{c} \begin{array}{cccc} s_1 & s_2 & \cdots & s_n \end{array} \\ \begin{bmatrix} \mu_{0,1} & \mu_{0,2} & \cdots & \mu_{0,n} \\ \mu_{1,1} & \mu_{1,2} & \cdots & \mu_{1,n} \\ \mu_{2,1} & \mu_{2,2} & \cdots & \mu_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{2^m-1,1} & \mu_{2^m-1,2} & \cdots & \mu_{2^m-1,n} \end{bmatrix} \end{array}, \tag{2.32}$$

where $\mu_{i,j}$ is the normalised Euclidean distance metric calculated using:

$$\mu_{i,j} = \frac{e^{-d_{i,j}}}{\sum_{s=1}^{2^m} e^{-d_{i,s}}}, \tag{2.33}$$

and d is the Euclidean distance between symbol S_j and the scaled q-QAM constellation point corresponding to the symbol value i in $GF(2^m)$.

The function to calculate $\mu_{i,j}$ can be seen as a modified version of the Softmax function used to represent a categorical distribution [30].

This is shown graphically for a single symbol in Figure 2.2.

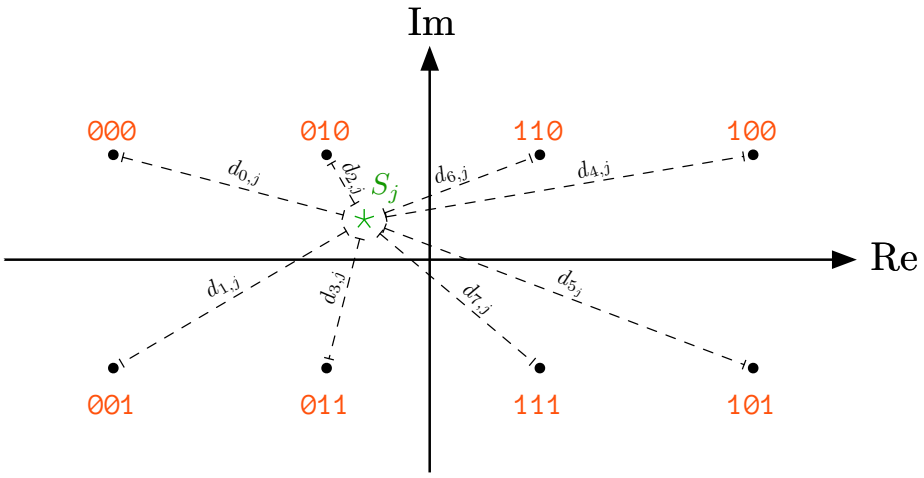


Figure 2.2: Diagram showing the distance metric calculation of a received symbol S_j within a Gray coded 8-QAM constellation.

Example 2.2. A simple example of a received matrix $\mathbf{R}'$, for a codeword of length $n = 7$ over $GF(2^3)$ is shown below:

$$\mathbf{R}' = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 \end{array} \\ \left[\begin{array}{ccccccc} 0.0309 & 0.0397 & 0.0361 & \mathbf{0.3475} & 0.0714 & 0.1542 & 0.0992 \\ 0.0275 & 0.0421 & 0.0460 & 0.1536 & 0.0614 & 0.1185 & 0.1162 \\ 0.0699 & 0.0856 & 0.0720 & 0.1789 & 0.1604 & \mathbf{0.2560} & 0.1621 \\ 0.0592 & 0.0943 & 0.1030 & 0.1144 & 0.1195 & 0.1558 & \mathbf{0.2372} \\ \mathbf{0.3555} & 0.1651 & 0.1301 & 0.0354 & 0.1096 & 0.0528 & 0.0583 \\ 0.1803 & \mathbf{0.2048} & \mathbf{0.2679} & 0.0294 & 0.0892 & 0.0469 & 0.0646 \\ 0.1581 & 0.1646 & 0.1207 & 0.0798 & \mathbf{0.2407} & 0.1190 & 0.1184 \\ 0.1188 & 0.2038 & 0.2242 & 0.0611 & 0.1479 & 0.0968 & 0.1439 \end{array}\right] \end{array}. \tag{2.34}$$

From the received matrix $\mathbf{R}'$ in (2.34), by selecting each symbol value as the row index of the maximum reliability value in each column, the following hard-decision codeword vector is extracted:

$$\begin{aligned} \mathbf{c} &= \begin{bmatrix} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 5 & 5 & 0 & 6 & 2 & 3 \end{bmatrix}, \end{aligned} \tag{2.35}$$

along with the corresponding reliability values:

$$\mathbf{r} = \begin{bmatrix} 0.3555 & 0.2048 & 0.2679 & 0.3475 & 0.2407 & 0.2560 & 0.23725 \end{bmatrix}. \tag{2.36}$$

Distance Metric vs LLR

While Log Likelihood Ratios (LLRs) are generally used for binary symbols, non-binary/multi-dimensional LLRs can be calculated. The number of LLRs required for a q-ary code increases exponentially with alphabet size [31], and thus quickly become infeasible for use in non-binary codes.

2.7 Algorithmic Complexity

The complexity of an algorithm can be expressed in terms of both time and space complexity. Space complexity gives a worst-case indication of how much memory an algorithm could require at any point during its operation [32].

The time complexity of an algorithm is an expression of the *worst-case* number of operations required by an algorithm to perform its job on a set of data [33]. It is a good proxy for understanding the amount of time an algorithm will take to compute an answer and gives an indication of the *order* of the rate at which the number of operations will grow as an algorithm's input size gets larger.

As an indicator of the *worst-case* complexity for an algorithm, it is often simplified to include only the largest contributing terms. For example, $\mathcal{O}(n^3 + n + 4)$ can be simplified to $\mathcal{O}(n^3)$, the dominating term, as n gets larger.

2.7.1 Big O Notation

The $\mathcal{O}$ of an algorithm can be found by combining the number of operations required for each step in the algorithm. Because the notation is interested in the *order* of the number of operations required, the rules for simplifying $\mathcal{O}$ are straightforward:

1. **Addition** - For all terms added together, the *dominating* factor as the input grows large is kept, all other terms are removed.
2. **Multiplication** - All *constant* co-efficients are removed as their impact is insignificant compared to the order of the terms.

Example 2.3. Simplify $\mathcal{O}(3x^3 + 5x + 4)$.

$$\begin{aligned} \text{let } f(x) &= \mathcal{O}(g(x)) \\ \therefore f(x) &= 3x^3 + 5x + 4 \end{aligned}$$

Of the three terms in the polynomial $f(x)$, as $x \to \infty$ the term with the highest power ($3x^3$) will dominate, and by *Rule 1*, the other two terms can be eliminated:

$$\therefore f(x) = 3x^3$$

Finally, using *Rule 2*, the multiplication constant can be removed, yielding:

$$\therefore f(x) = x^3$$
$$\therefore \mathcal{O}(3x^3 + 5x + 4) \approx \mathcal{O}(x^3)$$

2.8 AWGN Channel

The Additive White Gaussian Noise (AWGN) channel is used to simulate a natural noisy communications channel in this research. As the name implies, it is a linear memoryless channel, where Gaussian white noise is added to the transmitted signal.

White noise is a random signal with a constant power spectral density, and is useful for simulating natural processes such as thermal noise.

The channel is simple and does not take into account environment-specific oddities such as interference, fading, frequency selectivity, or any other non-linearities. Its simplicity allows it to be applied broadly to get an understanding of the base performance of a system before more advanced channel characteristics are simulated [34].

2.8.1 Channel Capacity

The AWGN channel has a capacity defined as:

$$C = \frac{1}{2} \log 2 \left(1 + \frac{S}{N}\right), \tag{2.37}$$

where $\frac{S}{N}$ is the Signal-to-Noise Ratio (SNR), and defined as:

$$SNR = \frac{E_b}{N_0}, \tag{2.38}$$

where E_b is the energy per bit, and N_0 is the noise spectral density (in Watts/Hz).

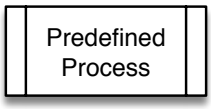
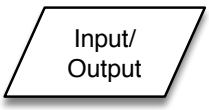
If the energy per symbol is E_s, then the energy per information bit is $E_b = \frac{E_s}{R}$, where R is the code rate $\frac{k}{n}$, introduced in Section 2.1.

An AWGN channel consists of white noise, which is defined as noise with a Power Spectral Density (PSD) independent of frequency.

2.9 Flow-Chart Notation

Flow charts are used in this dissertation to describe high-level algorithmic and data flow. The conventions adopted are summarised in Table 2.2.

Table 2.2: Description of flow-chart components and conventions.

Type	Example	Description
Start / End		Indicates the start or end of the algorithm.
Process		A single operation or process defined in the flow chart.
Sub-process		Abstracts another more complex process which could consist of multiple operations.
Decision		A Boolean decision branch.
Data		Describes data used at the input of a process, or produced as a result of a process.
Input / Output		External input or output to or from the flow chart.

Chapter 3

Research Methodology

The approach taken, success criteria and experimental setup designed to test improvements to the algorithm are described.

The research presented attempts to quantify the potential impact that sub-optimal use of Soft-Decision (SD) information has on the performance of the Parity-check Transformation Algorithm (PTA) decoder.

To achieve this, the PTA is first analysed in order to understand the extent to which the algorithm makes use of soft-decision information. Using techniques learned from other iterative soft-decision decoders identified in the literature, changes to the algorithm are then proposed. Finally, the affects of these modifications are quantified experimentally through computer simulations.

3.1 Performance Metrics and Success Criteria

The performance metrics studied in order to quantify the effects of various modifications are:

- Symbol Error Rate (SER) at different Signal-to-Noise Ratio (SNR) values between 0dB and 16dB.
- Average number of iterations required to converge on a valid codeword.
- The impact on the per-iteration time complexity introduced by modifications to the algorithm.

The overall computational complexity of the algorithm can be quantified using a combination of the per-iteration time complexity and the average number of iterations required to decode a codeword. If the modifications to the algorithm do not drastically affect the time complexity of a single iteration, a simple average of the number of iterations required to converge can be used as an effective proxy for quickly comparing the decoding complexity between the algorithms under test.

Modifications to the algorithm are considered successful if they are able to improve on any one of the performance metrics listed above.

3.2 Experimental Setup

This research makes use of computer simulations written in `MATLAB` and `GNU OCTAVE` to carry out experiments.

Various modifications to the PTA are evaluated by decoding a set of pre-generated test codewords. The effective SER and resulting decoding complexity are then compared to the unmodified PTA decoding an identical set of test data. The comparative results are plotted on graphs measuring the Symbol Error Rate vs. Signal to Noise Ratio $\frac{E_s}{N_o}$, along with various metrics to highlight effects on the convergence rate of the decoders.

A software testing framework for running rapid, reproducible experiments via simulations was developed to ensure that changes to the existing PTA could be prototyped, tested and measured quickly.

3.2.1 System Model

The system model utilised for this research, and implemented in the simulation environment, can be seen in Figure 3.1. The major components are a random information source, systematic Reed–Solomon encoder, 16-QAM modulator, Additive White Gaussian Noise (AWGN) channel, distance metric QAM-demodulator, the decoders under test and an information sink. Simulation parameters selected for each of these components will be discussed in the following sections.

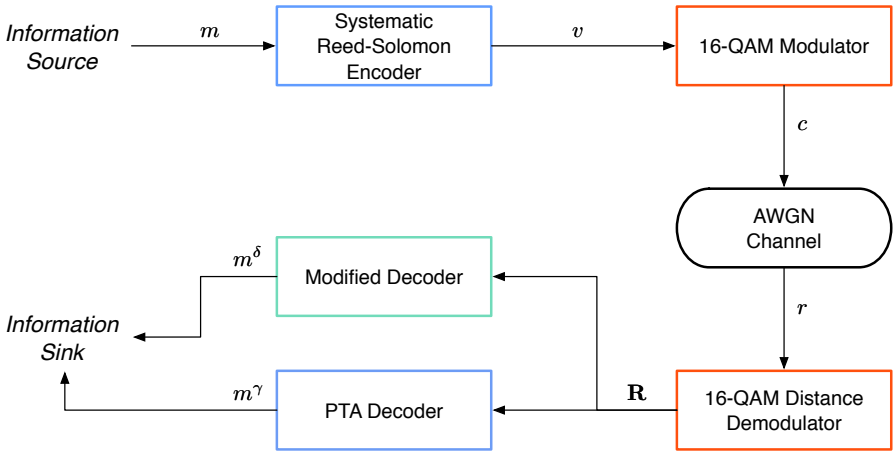


Figure 3.1: System overview

3.2.2 Reed-Solomon Encoding Parameters

Due to the aforementioned computational complexity of the PTA when operating on long codewords, and to ensure practical bounds on the total simulation time required by each experiment, the maximum codeword length is limited to $n = 15$.

From Chapter 2, it is known that the length of a Reed–Solomon codeword and the number of bits per symbol are related by:

$$n = 2^m - 1 \tag{3.1}$$

$$\therefore m = \log_2(n+1), \tag{3.2}$$

and as a result, $m = 4$ bit symbols are used with the code defined over $GF(16)$.

The PTA has been shown to perform well for *half-rate* and lower code rates [14]. As a result, the message length will be fixed at $k = 7$, producing a systematic $RS(15, 7)$ code. Fixing the code rate is useful to limit the number of simulation variables when testing modifications to the algorithm.

The *systematic* form of the code is useful for implementing an iterative algorithm, as it allows for the decoder to simply return the first k symbols of the received codeword should the algorithm terminate, due to a maximum number of iterations being reached. Additionally, the partially sparse nature of the systematic parity-check matrix helps reduce error propagation by message-passing decoders.

3.2.3 Modulation and Channel Model

The 16-QAM modulation scheme is chosen to allow for a one-to-one mapping of Reed-Solomon (RS) codeword symbols to modulation symbols. This keeps the simulation environment simple, as it allows for a direct implementation of the distance metric demodulator for generating the $\mathbf{R}$ matrix utilised by the PTA decoder.

Other modulation schemes such as Binary Phase Shift Keying (BPSK), 8-QAM, or 32-QAM can be used, however they require additional steps to generate a valid $\mathbf{R}$ matrix for the selected RS(15,7) code, and as a result are not considered in this dissertation.

The AWGN channel is chosen to provide a simple, memoryless channel model with a single tunable parameter. This ensures the performance analysis of various modifications to the decoder can focus exclusively on the relative performance of the decoder and not the underlying RS code's effectiveness in different environments.

3.2.4 Parity-check Transformation Algorithm Parameters

The PTA has two main tunable parameters: `max_iterations` - the maximum number of iterations allowed per codeword before terminating, and δ - the amount to increment or decrement the reliability scores per iteration.

The PTA has been shown to have good symbol error-rate performance when using a small value of δ. Simulations are performed with a value of $\delta = 0.001$, known to ensure the PTA's best symbol error-rate performance [13]. Based on the results shown in the literature, the number of iterations required to converge on a codeword is inversely proportional to the size of δ, and for small values can require up to 1,000 iterations for some codewords to decode. To ensure the overwhelming majority of codewords decode fully, and the stopping condition is used only to prevent runaway computation, a `max_iterations` value of 5,000 is chosen. This will allow for a complete understanding of the worst-case performance of the decoder.

3.2.5 Modified Algorithm Testing Parameters

Several variations of the modified algorithm are tested against the PTA. The number of times per iteration that the parity-check matrix is transformed is varied in order

to understand the transformation's impact on the performance of the decoder. The following variations are tested:

Single Transformation
The parity-check matrix is transformed once at the start of decoding a codeword. This modified matrix is used for all subsequent iterations.

Multi Transformation
The parity-check matrix is transformed *once per iteration* based on the modified **R** matrix from the previous iteration.

No Transformation
The parity-check matrix is never transformed, the default systematic **H** matrix is used for every iteration.

3.2.6 CeTAS Cluster

Due to the high computational complexity of the PTA, simulations were performed using the **CeTAS** compute cluster, located in the Telecom's Laboratory at the School of Electrical & Information Engineering. Use of the cluster required that simulations be run using GNU OCTAVE while making use of MPI for parallel computation [35, 36].

In order to make effective use of both the compute cluster running OCTAVE, and the toolboxes available exclusively in MATLAB, the simulation is split into multiple phases. The first phase, described in Section 3.2.7 generates all required testing data, including random source data, RS-encoded, modulated, received and distance demodulated symbol reliabilities. This data is saved, and copied to the cluster for further testing. The second phase of the simulation is run on the cluster, and is used to rapidly test different variations of the algorithms. The decoded data from each test is then persisted to disk, and finally utilised for analysis. This division of work can be seen in Figure 3.2.

3.2.7 Simulation Test Data

The simulation utilises a large set of 10,000 pre-generated codewords, modulated and transmitted at SNR's ranging between 0dB and 16dB. The test data is generated once and persisted, allowing for all experiments and modifications to be tested against

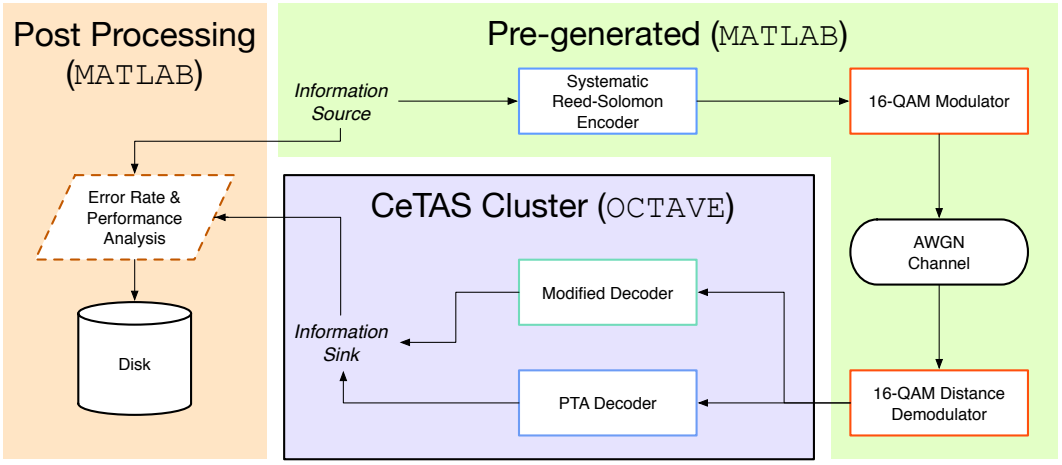


Figure 3.2: Work division between local and remote cluster environment.

identical data sets and channel conditions. This has the added benefit of significantly reducing the amount of redundant computation required per simulation run. A graphical representation of the data generated and persisted in this phase can be seen in Figure 3.3.

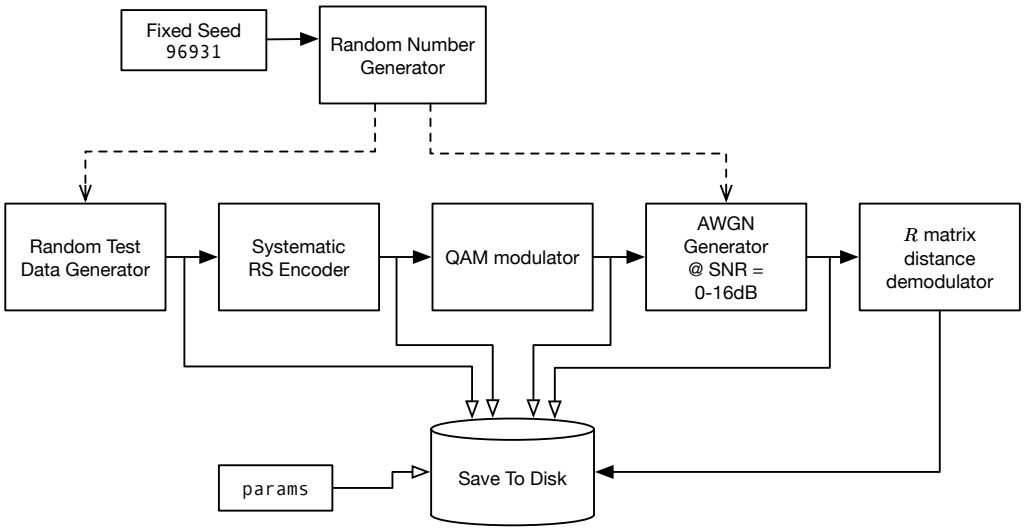


Figure 3.3: Test data generation and persistence showing the full evidence chain and all parameters saved.

All parameters and intermediate data generated are also saved to disk. During the data generation phase, a *Mersenne Twister* Pseudo Random Number Generator (PRNG) [37] is seeded with a known value and used to generate both the test data and channel noise. This ensures that the experimental environment can be reliably recreated, allowing for easy auditing of both test data and results. The fixed PRNG seed value was initially created using the MATLAB `randseed` function [38].

Encoding and Modulation

Test data is encoded using the RS encoder provided by the MATLAB *Telecommunications Toolbox* [39].

Encoded codewords are then modulated using the *Telecommunications Toolbox* QAM modulator [40]. The modulator is set to use a Gray-coded, rectangular 16-QAM constellation, with the average transmit power normalised to 1. This helps eliminate the effects of constellation size and transmit power on the results. For a regular rectangular 16-QAM system, the normalisation step has the effect of dividing all signal magnitudes by $\sqrt{10}$ [40].

Channel Simulation

Channel simulation is achieved using the `awgn` function provided by the MATLAB *Telecommunications Toolbox* [41]. The channel's SNR, measured as $\frac{E_s}{N_o}$, is varied in 0.5dB increments from 0dB to 16dB. While not strictly required due to the power normalisation performed in the modulation step, the `awgn` function is set to measure the average power of the input before applying noise.

Distance Demodulation

The received symbols at each SNR are finally run through a distance demodulator as described in Chapter 2. This produces the **R** matrix for each codeword as required by the PTA.

The demodulator generates a reference 16-QAM constellation of all valid symbols using identical parameters to the modulation step. This reference constellation is then used to calculate Euclidean distance measurements for each received symbol.

3.2.8 Version Control

All test data, simulation code and results are stored in a set of Git repositories [42]. Simulation parameters including the git commit ID of the code used to run the simulation are all persisted alongside every set of results generated by various experiments.

Experiments were created on unique git branches (prefixed with `experiment/`) in order to easily highlight what code changes made for each experiment. This also ensures that the state of the code base for every experiment can be easily retrieved for easy reproducibility and verification of results.

Results are also persisted to a Git repository, allowing for a complete chain of evidence, including all code, simulation parameters and input data to be produced for every experiment and simulation from data generation through to final results.

Chapter 4

Parity-check Transformation Algorithm Analysis

This chapter describes and analyses the Parity-check Transformation Algorithm (PTA) in detail, with particular attention paid to the extent to which it uses soft decision information. The time complexity of the algorithm is found, and potential areas for improvements to the algorithm's use of soft decision information are discussed.

As a precondition to being able to answer the research question introduced in Chapter 1, an understanding around *how* the PTA utilises soft channel information must be formed. Further analysis related to the computational complexity of the algorithm is also required in order to accurately quantify the effects of modifications to the algorithm.

4.1 Parity-check Transformation Algorithm Overview

The PTA is an iterative Reed–Solomon decoder, which operates on a matrix of received symbol reliabilities $\mathbf{R}$.

As described in Chapter 2, the $\mathbf{R}$ matrix consists of 2^m rows and n columns. Each row i represents a symbol value within the finite field $GF(2^m)$, while each column j corresponds to the jth symbol in the received codeword. The value in each element represents the "reliability" assigned by the demodulator, and can be thought of as a pseudo-probability that a symbol has the value indicated by the row index.

At a high level, the PTA performs the following steps in iteration t when decoding a codeword:

1. Using the received reliability matrix $\mathbf{R}^{(t)}$, the *current* Hard-Decision (HD) vector $\mathbf{x}^{(t)}$ is found by selecting the row index with the highest reliability score for each symbol position/column. The reliability score of each HD symbol is stored in a separate vector $\mathbf{r}^{(t)}$.
2. $\mathbf{r}^{(t)}$ is sorted to find the indices of the $n-k$ least reliable symbols, represented as $\mathcal{U}^{(t)}$, and the k most reliable symbols shown as $\mathcal{R}^{(t)}$.
3. The systematic $\mathbf{H}$ matrix is *transformed* using Gaussian row operations such that the k high density columns are in the positions of the highest reliability symbols, producing $\mathbf{H}^{\gamma(t)}$.
4. Each parity-check equation in $\mathbf{H}^{\gamma(t)}$ is then tested using the current HD symbol values. If an equation is satisfied (or equals 0), all contributing symbols have the reliabilities of their current selected symbol value in $\mathbf{R}^{(t)}$ *incremented*, if not, the reliabilities are *decremented*. The magnitude of this change is a fixed value of δ for the $n-k$ least reliable symbols in $\mathcal{U}^{(t)}$, and $\frac{\delta}{2}$ for the most reliable symbols in $\mathcal{R}^{(t)}$.
5. If all of the parity equations are simultaneously satisfied, or the maximum number of iterations is reached, the first k symbols of the HD vector $\mathbf{x}^{(t)}$ are returned as the decoded message. Otherwise, the process begins again using the updated $\mathbf{R}^{(t+1)}$ matrix.

A flow diagram of these steps can be seen in Figure 4.1.

4.2 PTA as a message-passing algorithm

While the PTA is described in terms of matrix operations and update rules in [13], it can be beneficial to visualise it as a message passing algorithm with the help of Tanner graphs [27]. Messages sent from variable nodes indicate their current state, while check nodes send messages indicating to how the connected variable nodes should adjust their corresponding reliabilities.

The parity-check matrix can be represented as a Tanner graph, with check nodes representing the rows of the $\mathbf{H}$ matrix, and variable nodes representing the columns. For non-binary codes the edges between nodes include a scaling-factor/co-efficient from that position in the $\mathbf{H}$ matrix. Any message passed along that edge is multiplied

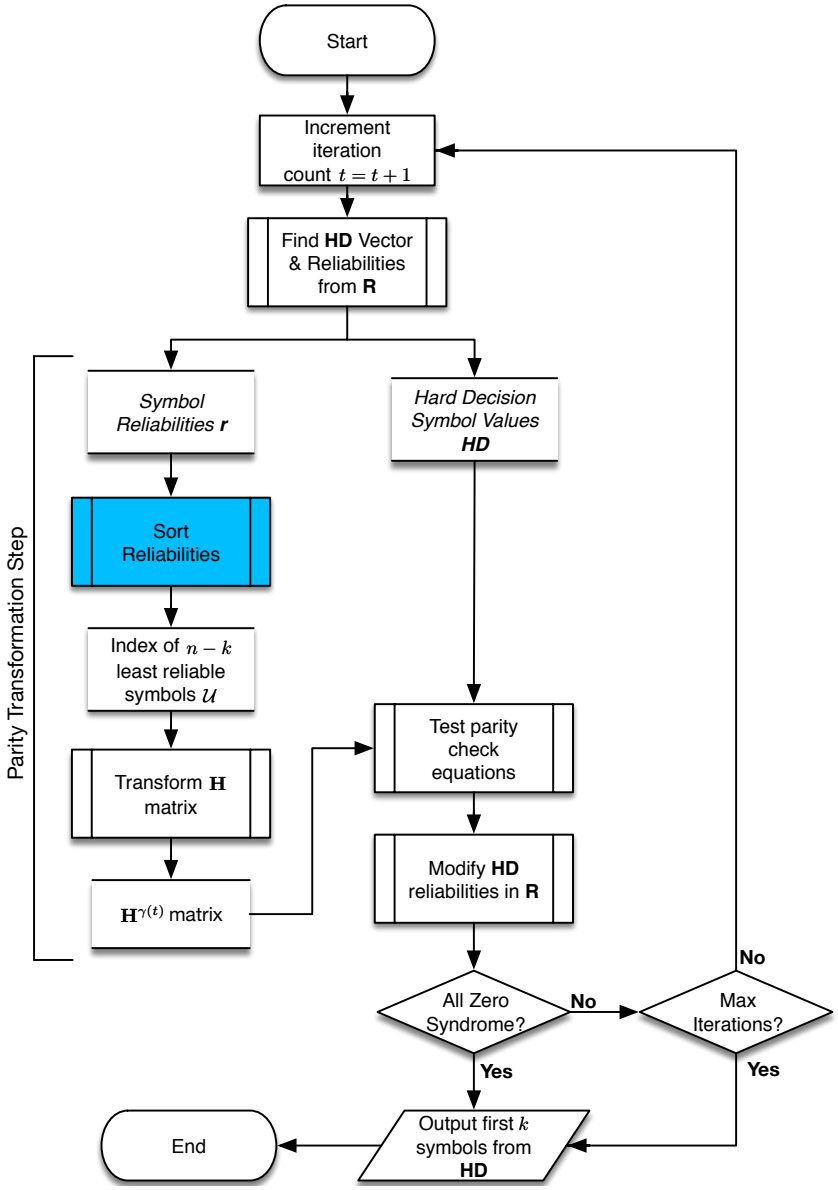


Figure 4.1: High-level flow chart of the Parity Transformation Algorithm. The highlighted reliability sorting stage is the only part of the algorithm which makes use of soft-decision information.

by this co-efficient on the way into a check node, and divided by it on the way back to a variable node.

To further facilitate understanding of how the PTA operates as a message-passing algorithm, an example of the algorithm decoding a received codeword is presented.

This example makes use of a systematic RS(7,3) code defined by the parity-check matrix:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 6 & 4 & 3 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 6 & 5 & 2 \\ 0 & 0 & 0 & 1 & 7 & 5 & 3 \end{bmatrix}, \tag{4.1}$$

and illustrated as the Tanner graph shown in Figure 4.2. The decoder's δ value is set to **0.2** for illustrative purposes.

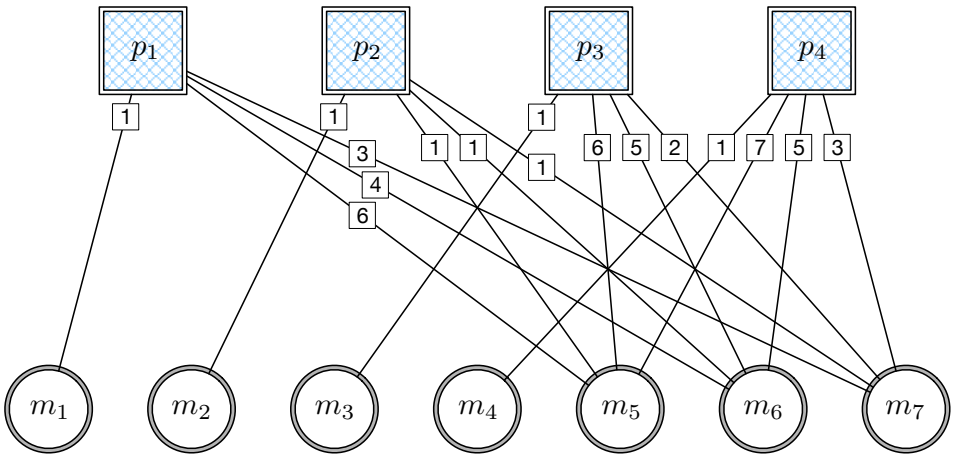


Figure 4.2: A Tanner graph visualising the parity-check matrix defined in (4.1).

4.2.1 Step 1: Hard-Decision Symbol and Reliability Extraction

Given a received reliability matrix from a noisy channel:

$$\mathbf{R} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 \end{array} \\ \left[\begin{array}{ccccccc} 0.1499 & 0.0987 & 0.1612 & \mathbf{0.3297} & \mathbf{0.2830} & 0.1909 & 0.0995 \\ \mathbf{0.3265} & 0.0788 & \mathbf{0.2143} & 0.1665 & 0.1332 & \mathbf{0.3495} & 0.1771 \\ 0.1192 & \mathbf{0.2224} & 0.1580 & 0.1739 & 0.2166 & 0.1195 & 0.1197 \\ 0.1882 & 0.1360 & 0.2076 & 0.1249 & 0.1188 & 0.1552 & \mathbf{0.2665} \\ 0.0310 & 0.0831 & 0.0389 & 0.0340 & 0.0444 & 0.0272 & 0.0467 \\ 0.0370 & 0.0681 & 0.0421 & 0.0301 & 0.0349 & 0.0303 & 0.0638 \\ 0.0645 & 0.1876 & 0.0834 & 0.0769 & 0.0994 & 0.0587 & 0.0862 \\ 0.0836 & 0.1253 & 0.0945 & 0.0641 & 0.0698 & 0.0686 & 0.1406 \end{array}\right] \end{array}, \tag{4.2}$$

the algorithm begins by identifying the maximum value in each column, highlighted in **bold**. The corresponding row indices represent the hard-decision values for each symbol in the codeword.

For the first iteration of the algorithm, the resulting hard-decision symbol vector extracted from (4.2) is:

$$\mathbf{x}^{(1)} = \begin{bmatrix} 1 & 2 & 1 & 0 & 0 & 1 & 3 \end{bmatrix}, \tag{4.3}$$

and the vector containing each symbol's corresponding reliability value is:

$$\mathbf{r}^{(1)} = \begin{bmatrix} 0.3265 & 0.2224 & 0.2143 & 0.3297 & 0.2830 & 0.3495 & 0.2665 \end{bmatrix}. \tag{4.4}$$

4.2.2 Step 2: Parity-check Matrix Transformation

The reliability vector $\mathbf{r}$ is sorted to find the *index positions* of the $(n-k)$ most and k least reliable symbols in the hard-decision vector. These are shown as the most reliable positions $\mathcal{R}$:

$$\mathcal{R}^{(1)} = \begin{bmatrix} 1 & 4 & 6 \end{bmatrix}, \tag{4.5}$$

and least reliable positions $\mathcal{U}$ in:

$$\mathcal{U}^{(1)} = \begin{bmatrix} 3 & 2 & 7 & 5 \end{bmatrix}. \tag{4.6}$$

It is critical to note that this ordering step is the *only time* the PTA makes use of the soft symbol information provided by the demodulator.

The parity-check matrix $\mathbf{H}$ is then *transformed* using Gaussian row operations in order to position the high-density parity columns in the positions of the most reliable symbols (listed in $\mathcal{R}$), and the identity columns in the positions shown in $\mathcal{U}$. The result of the first transformation is:

$$\mathbf{H}^{\gamma(1)} = \begin{bmatrix} 5 & 1 & 0 & 3 & 0 & 7 & 0 \\ 5 & 0 & 1 & 2 & 0 & 6 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 4 & 0 & 0 & 2 & 0 & 7 & 1 \end{bmatrix}, \tag{4.7}$$

and can be visualised as a Tanner graph in Figure 4.3.

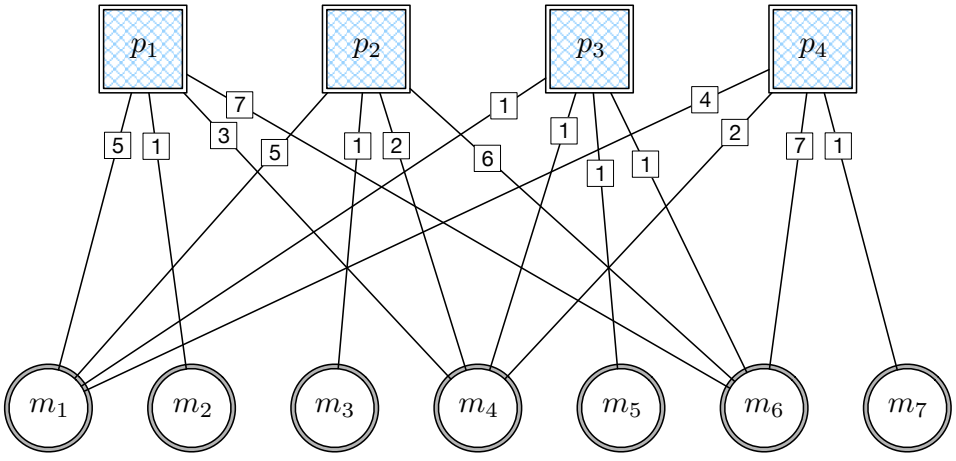


Figure 4.3: A Tanner graph visualising the transformed parity-check matrix shown in (4.7).

4.2.3 Step 3: Syndrome / Check-node Test

The transformed parity-check matrix is then used to calculate the syndrome of the current hard-decision vector.

Described as a message-passing algorithm, this step consists of each check node receiving the symbol values of all of its connected variable nodes (each multiplied over $GF(8)$ by the value of their respective connecting edges). If the sum of all inputs over the code's GF is zero, then that the node is satisfied.

For check nodes which are satisfied, an **increment** ($\oplus$) message is then broadcast to all connected variable nodes for having participated in a valid parity-check equation. If, however, a check node is not satisfied, a **decrement** ($\ominus$) message is sent to its connected variable nodes. As a result of these messages being broadcast equally to all participating variable nodes in a check, the system can unfairly 'punish' correct symbols simply because another symbol connected to the same check node is incorrect.

The increment and decrement messages have an effect on the connected variable node's corresponding symbol reliability. The magnitude of this change is the static parameter δ for sparsely connected nodes, and $\frac{\delta}{2}$ for densely connected nodes in order to compensate for the additional messages received by those nodes in every iteration.

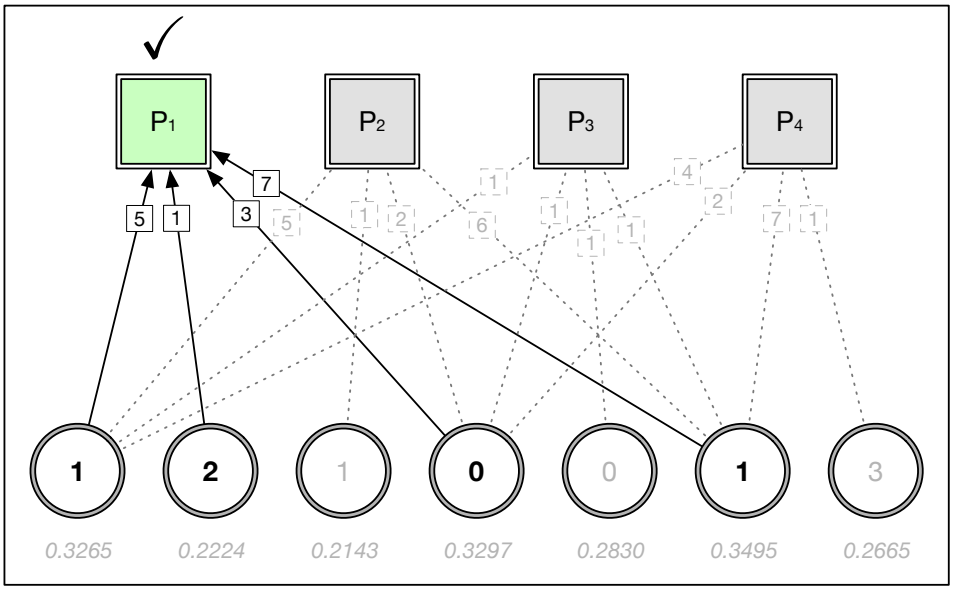
This process is shown graphically using a value of $\delta = 0.2$ in Figures 4.4 to 4.7.

4.2.4 Step 4: Reliability Update

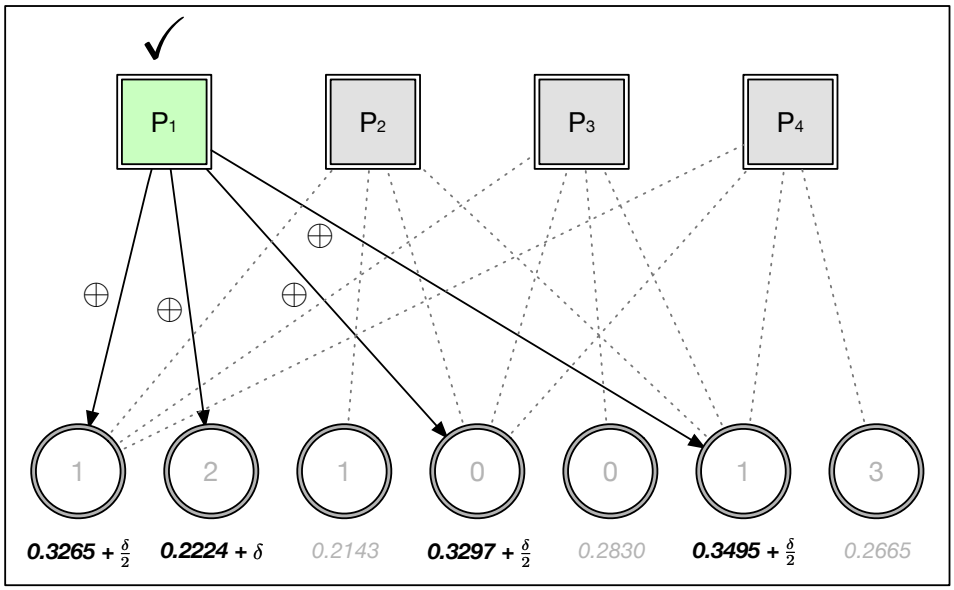
After the first round of check nodes sending their increment/decrement messages, detailed in Figures 4.4 to 4.7, the resulting reliability values for the currently selected symbols are:

$$\mathbf{r}^{(2)} = \begin{bmatrix} 0.5265 & 0.4224 & 0.4143 & 0.5297 & 0.0830 & 0.5495 & 0.4665 \end{bmatrix}. \tag{4.8}$$

These values are inserted back into the $\mathbf{R}$ matrix at the positions corresponding to their symbol values.

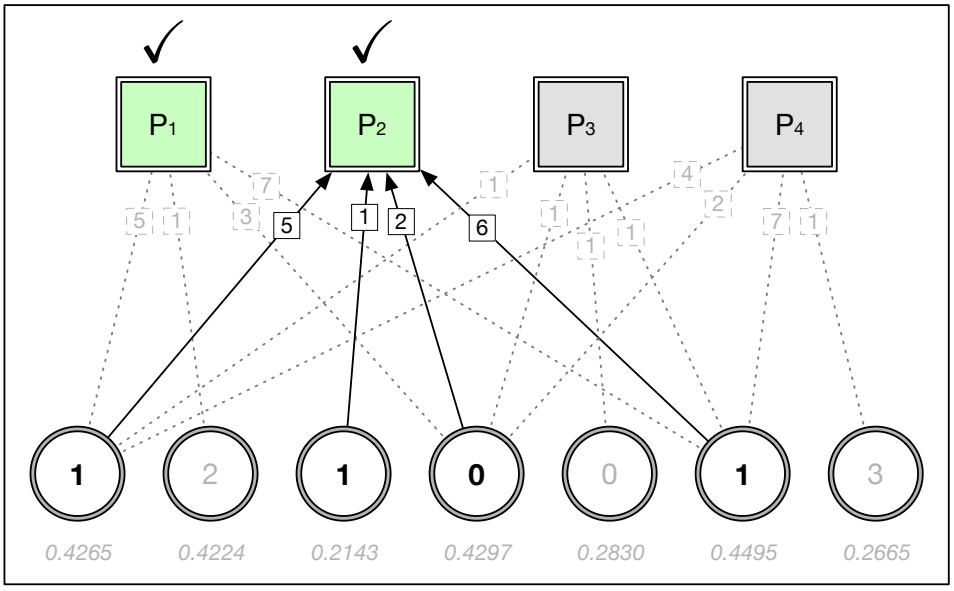


(a) Message symbol values are passed to the first check node p_1. The values are first multiplied by the values on their connecting edges and summed at the check node. All operations are performed over $GF(8)$

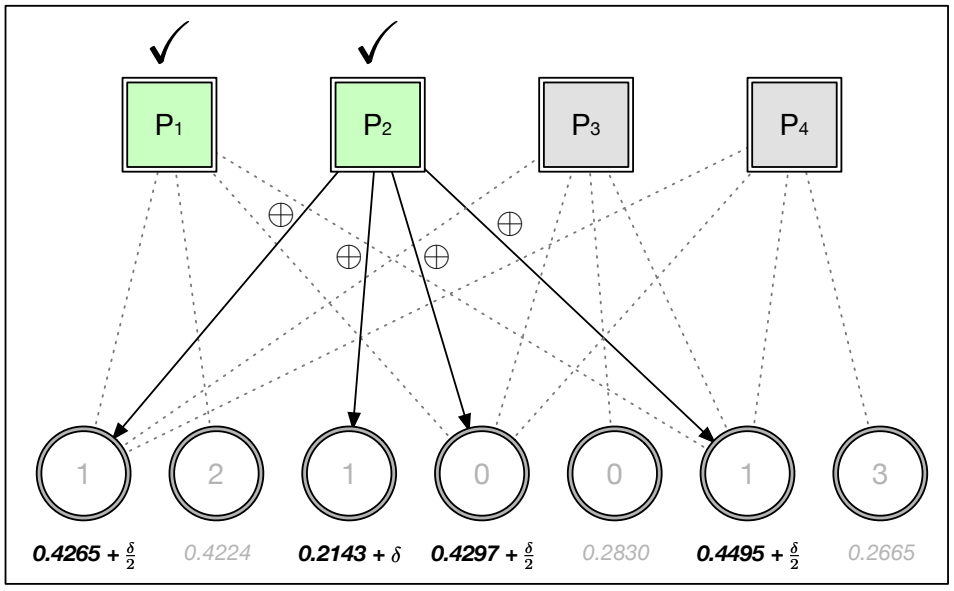


(b) The first check node is satisfied, therefore a reliability *increment* message $\oplus$ is passed back to the variable nodes. The magnitude of the increment applied is δ for sparsely connected variable nodes, and $\frac{\delta}{2}$ for densely connected nodes, where $\delta = 0.2$.

Figure 4.4: A set of messages passed between variable nodes and check node p_1.

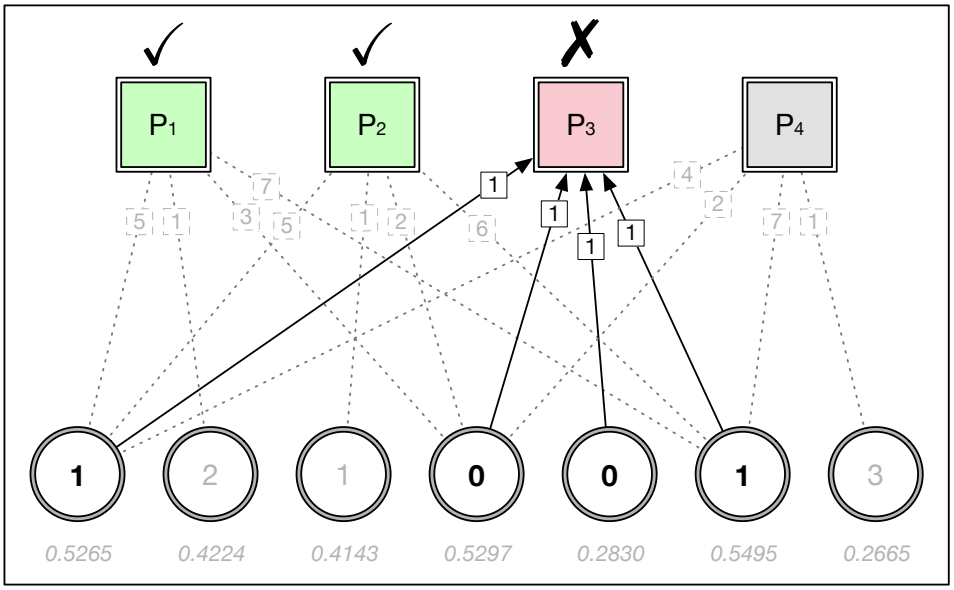


(a) Message symbol values are passed to the second check node p_2 in the same manor as for p_1.

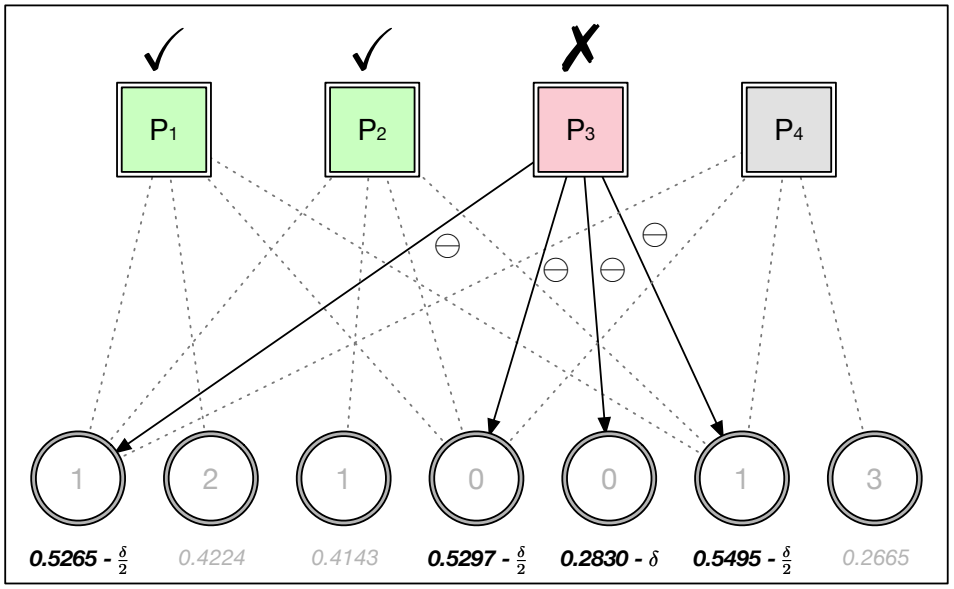


(b) The reliability *increment* message $\oplus$ is passed back to the connected variable nodes.

Figure 4.5: A set of messages passed between variable nodes and the second check node p_2. The check node is satisfied and as a result connected variable nodes have their reliabilities incremented.

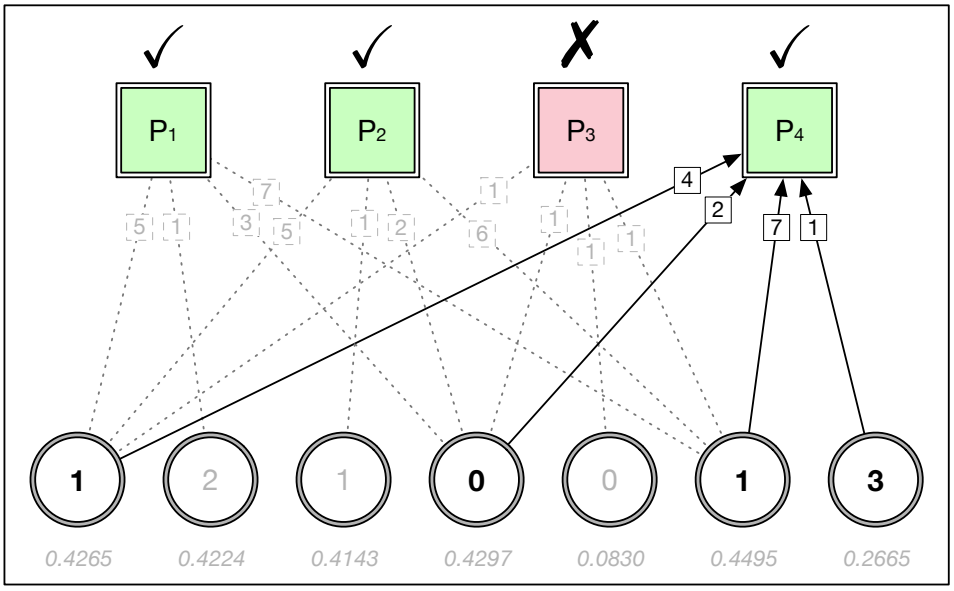


(a) Message symbol values are passed to the third check node p_3. The sum of all inputs over $GF(8)$ at this node is not 0, and so, therefore, its check is not satisfied.

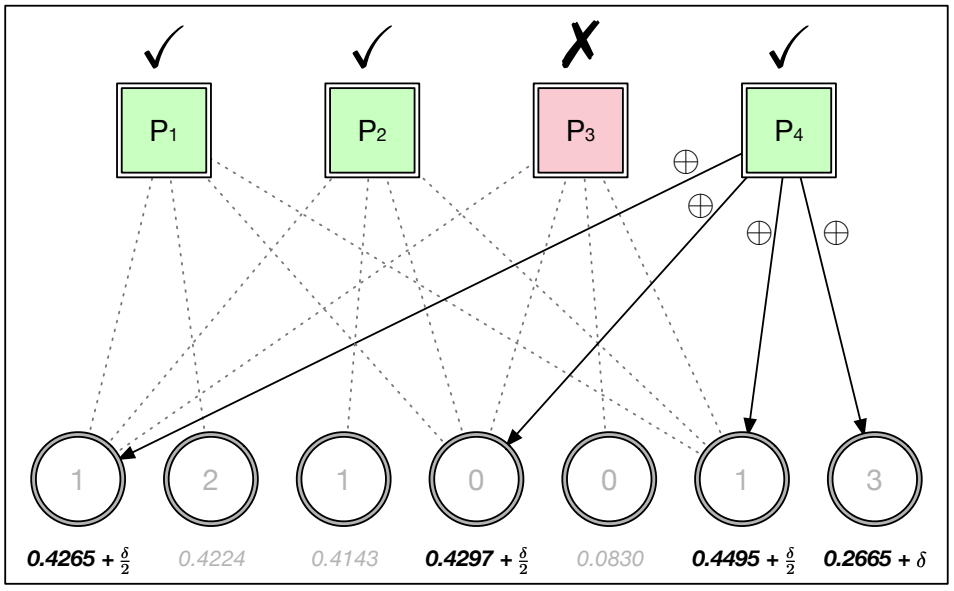


(b) As a result of the check node not being satisfied, the reliability *decrement* message $\ominus$ is passed back to the connected variable nodes.

Figure 4.6: The set of messages passed between variable nodes and the third check node p_3, resulting in the connected variable nodes have their symbol state reliabilities decremented.



(a) Message symbol values are finally passed to the fourth check node p_4, resulting in the check being satisfied.



(b) The reliability *increment* message $\oplus$ is passed back to the connected variable nodes.

Figure 4.7: The final set of messages passed between variable nodes and the last check node p_4. The check node is satisfied and, as a result, connected variable nodes have their reliabilities incremented.

The resulting $\mathbf{R}$ matrix can be seen with new hard decision values highlighted below:

$$\mathbf{R}^{(2)} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 \end{array} \\ \left[\begin{array}{ccccccc} 0.1499 & 0.0987 & 0.1612 & \mathbf{0.5297} & \mathbf{0.0830} & 0.1909 & 0.0995 \\ \mathbf{0.5265} & 0.0788 & 0.0143 & 0.1665 & 0.1332 & \mathbf{0.5495} & 0.1771 \\ 0.1192 & \mathbf{0.4224} & 0.1580 & 0.1739 & 0.2166 & 0.1195 & 0.1197 \\ 0.1882 & 0.1360 & \mathbf{0.4143} & 0.1249 & 0.1188 & 0.1552 & \mathbf{0.4665} \\ 0.0310 & 0.0831 & 0.0389 & 0.0340 & 0.0444 & 0.0272 & 0.0467 \\ 0.0370 & 0.0681 & 0.0421 & 0.0301 & 0.0349 & 0.0303 & 0.0638 \\ 0.0645 & 0.1876 & 0.0834 & 0.0769 & 0.0994 & 0.0587 & 0.0862 \\ 0.0836 & 0.1253 & 0.0945 & 0.0641 & 0.0698 & 0.0686 & 0.1406 \end{array}\right] \end{array}. \tag{4.9}$$

4.2.5 Step 5: Stopping Condition Check

The algorithm then checks to see if any of the stopping criteria have been met, including:

- If *all* of the parity equations successfully 'checked'.
- The maximum number of iterations is reached.
- A negative value is found in the new $\mathbf{R}$ matrix.

If any of the stopping conditions are met, the first k symbols from the current iteration's hard-decision symbol vector are output as the decoded codeword.

4.2.6 Step 6: Repeat

If none of the above cause the algorithm to terminate, the iterative process begins again, using the updated state of the $\mathbf{R}$ matrix as seen in (4.8) as the starting state for finding the next HD vector $\mathbf{x}^{(t+1)}$ and reliabilities.

In this example, after the first iteration *none* of the above are true, and, as a result, the algorithm will repeat.

4.2.7 Step 7: Convergence

After **1** additional iteration, the PTA successfully converges on a valid codeword:

$$\mathbf{c} = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 1 & 3 \end{bmatrix}, \tag{4.10}$$

with the final extracted message output:

$$\mathbf{m} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}. \tag{4.11}$$

4.3 Time Complexity Analysis

The computational complexity of an iterative algorithm such as the PTA can be expressed as a function of the average number of iterations before the algorithm converges and the number of operations required per iteration.

Additionally, the per-iteration time complexity of the algorithm is an important characteristic in order to be able to understand how modifications to the PTA affect its computational complexity.

Each iteration in the PTA can be broken down into several discreet phases, each will be analysed individually, and combined to assemble the overall time complexity of each iteration.

The time complexity of each phase will be presented using Big-O notation, and, as discussed in Chapter 2, will take into account the *worst-case* performance of the algorithm.

4.3.1 Initialisation Stage

This stage occurs only once at the start of the algorithm, and involves copying the $\mathbf{R}$ matrix into memory.

4.3.2 Hard-Decision and Reliability Extraction

The first phase of the algorithm determines the hard-decision symbol values, along with their corresponding reliabilities. This requires 2^m operations per symbol to find the symbol index with the highest value, and is repeated for each of the n columns in the $\mathbf{R}$ matrix:

$$\mathcal{O}(2^m \cdot n). \tag{4.12}$$

4.3.3 Reliability Ordering

The next phase of each iteration orders the message symbols by reliability in order to find the $n-k$ most and k least-reliable symbols. This is achieved by sorting the symbol reliability vector $\mathbf{r}$, and noting the original index of each item. This requires at worst n^2 comparisons:

$$\mathcal{O}(n^2). \tag{4.13}$$

4.3.4 Parity-check Matrix Transformation

This phase of the iteration requires the newly transformed $\mathbf{H}^\gamma$ be found using Gauss Row Reduction. As shown in [28], this requires a total of $(n-k)^3$ operations:

$$\mathcal{O}((n-k)^3). \tag{4.14}$$

It is important to note the complexity introduced by the parity transformation step limits its effectiveness at longer codeword lengths, and lower code rates.

The transformation step accounts for a large component of the per-iteration time complexity, and thus for short code lengths it may be advantageous to pre-compute all possible transformed parity-check matrices. This would reduce the complexity of this phase into a simple constant time lookup.

Memory Requirements for the pre-computation of transformed H matrices

In order to assess the feasibility of using a set of pre-computed **H** matrix transforms, the overall memory requirements for storing the resulting matrices (and lookup index) must first be calculated. To accomplish this, the total number of uniquely transformed **H** matrices must first be found.

During the transformation, the algorithm discriminates only between *reliable* symbols indexed in $\mathcal{R}$ and unreliable symbols in $\mathcal{U}$, with any sub-ordering within these classes being completely ignored. The total number of transformed matrices is therefore equal to the number of unique permutations of reliable/unreliable symbols in the received codeword.

As shown in [43], the number of unique permutations of a list, consisting of two sets of indistinguishable items, can be calculated using:

$$\text{unique permutations} = \frac{\text{total permutations}}{\text{permutations of A} * \text{permutations of B}}. \tag{4.15}$$

Given that there are n symbols in the received vector, with $(n-k)$ symbols classified as *reliable* and k as *unreliable*, the total number of unique permutations P is:

$$P = \frac{n!}{k!(n-k)!}. \tag{4.16}$$

The total number of permuted matrices, along with the overall memory requirements for various short Reed-Solomon code lengths are detailed in Table 4.1.

For each code rate, the number of bits required to hold every permutation of the parity-check matrix, along with index vectors used to look up the correct matrix are calculated. The results show that it is completely feasible to rely on pre-computed $\mathbf{H}^{\gamma}$ matrices for short code lengths.

Table 4.1: Memory required to pre-calculate all permutations of H.

			Index	$\mathbf{H}^{\gamma}$ Matrix		Total Storage	
Code	m	**Permutations**	**Bits**	**Elements**	**Bits**	**Bits**	**Bytes**
RS(7,3)	3	35	21	28	84	3675	459.3 B
RS(15,3)	4	455	60	180	720	354900	43.3 KB
RS(15,7)	4	6 435	60	120	480	3474900	424.2 KB
RS(15,11)	4	1 365	60	60	240	409500	50.0 KB
RS(31,15)	5	300 540 195	155	496	2480	7.92E11	92.2 GB
RS(31,25)	5	736 281	155	186	930	7.99E8	95.2 MB

4.3.5 Syndrome Calculation

The next phase of the algorithm calculates the syndrome of the current hard-decision vector in order to test which parity-check nodes were satisfied. This involves finding the vector-matrix product between HD and $\mathbf{H}^{\gamma}$ as shown below:

$$S = \mathbf{x}^T \cdot \mathbf{H}^{\gamma}. \tag{4.17}$$

The number of operations required to find the syndrome is on the order of:

$$\mathcal{O}(1 \cdot n \cdot (n-k)) = \mathcal{O}(n \cdot (n-k)). \tag{4.18}$$

4.3.6 Update reliabilities in the R matrix

The result of computing the syndrome is then utilised to increment or decrement the involved hard-decision symbol reliabilities in R.

This requires adding or subtracting the static δ value from the reliability of each of the n selected hard-decision symbols, once for each of the $(n-k)$ parity-check equations. The number of operations required for this step is therefore on the order of:

$$\mathcal{O}((n-k) \cdot n). \tag{4.19}$$

However, due to the partially sparse nature of the systematic parity-check matrix, a more accurate representation can be calculated by taking into account the average row weight $\mathcal{W}_{\mathbf{H}}$ of the transformed parity-check matrix. Thus the average number of updates performed by each check node is reduced to:

$$\mathcal{O}((n-k)\cdot\mathcal{W}_{\mathbf{H}}). \tag{4.20}$$

4.3.7 Test Stopping Conditions

The final phase of each iteration tests to see if any of the algorithm's stopping conditions have been met.

The first test checks to see a valid codeword has been found by testing if all $(n-k)$ values within the syndrome vector are 0:

$$\mathcal{O}(n-k). \tag{4.21}$$

The updated $\mathbf{R}$ matrix is then scanned for negative values, incurring a cost of:

$$\mathcal{O}(n\cdot 2^m). \tag{4.22}$$

Finally, the iteration count is tested to see if the maximum has been hit, this requiring a single operation:

$$\mathcal{O}(1). \tag{4.23}$$

4.3.8 Total Operations

Combining the results for each phase of the algorithm within an iteration, and assuming the transformed parity-check matrices are *not* pre-computed the overall complexity is expressed as:

$$\mathcal{O}(2\cdot 2^m\cdot n + n^2 + (n-k)^3 + (n-k)(n+\mathcal{W}_{\mathbf{H}}+1)+1). \tag{4.24}$$

As discussed in Chapter 2, the average weight of a systematic Reed-Solomon (RS) parity-check matrix row is $\mathcal{W}_{\mathbf{H}} = (k+1)$ and $2^m = n+1$. Using this the overall per-iteration complexity can be written in terms of only n and k:

$$\begin{aligned} \text{total} &= \mathcal{O}(2(n+1)\cdot n + n^2 + (n-k)^3 + (n-k)(n+k+2) + 1) \\ &= \mathcal{O}(n^3 + 4n^2 + 4n + 3k^2n - k^3 - k^2 - 3kn^2 - 2k + 1). \end{aligned} \tag{4.25}$$

Using the rules defined in Section 2.7 of Chapter 2, the worst-case time complexity of the PTA can be simplified to:

$$\mathcal{O}(n^3 - k^3). \tag{4.26}$$

Pre-computed Parity-check Matrix

If the memory requirements allow for all transformed $\mathbf{H}^{\gamma}$ matrices to be pre-calculated, the $\mathcal{O}((n-k)^3)$ term in (4.25) can be replaced with a constant time lookup operation of $\mathcal{O}(1)$ as seen below:

$$\begin{aligned} &= \mathcal{O}(2(n+1)\cdot n + n^2 + 1 + (n-k)(n+k+2) + 1) \\ &= \mathcal{O}(4n^2 + 4n - k^2 - 2k + 2). \end{aligned} \tag{4.27}$$

Using the simplification rules for worst-case time complexity, this simplifies to:

$$\mathcal{O}(n^2 - k^2), \tag{4.28}$$

a significant reduction in per-iteration complexity compared to (4.26), for code lengths short enough to support this optimisation.

4.4 Conclusion and Potential Improvements

Thoroughly examining the PTA hints at several potential enhancements which could improve performance.

As discussed in Section 4.2.2, the PTA utilises only the available soft symbol reliabilities for the *relative* ranking and classification of each symbol as either *reliable* or *unreliable*. This results in potentially useful information such as the actual reliability magnitudes being ignored and discarded prematurely.

While the PTA makes use of fixed values δ and $\frac{\delta}{2}$ to promote or punish the reliability of a currently selected symbol, a dynamic value could enable the algorithm to make larger changes to the reliability of a symbol based on its confidence in the state of the rest of the system. An improved algorithm could make use of these discarded reliability magnitudes to influence the value of δ.

Furthermore, the algorithm is limited to modifying only the reliability values of the *currently selected* hard-decision symbols. The effect of this is iterations wasted while *waiting* for an incorrect symbol's value to be replaced in the HD vector before its reliability can be incremented, resulting in a negative impact on the rate of convergence.

Additionally, as discussed in Section 4.2.3, the algorithm will unfairly punish all symbols connected to a check node that is not satisfied, even when only a single incorrect symbol caused the check to fail.

Finally, the algorithm's per-iteration computational complexity was quantified as a useful metric for comparative performance analysis with any modified versions of the algorithm. This analysis also shows the feasibility of reducing the algorithm's complexity through the use of pre-calculated parity matrix transformations. However, the memory requirements make this impractical for longer code lengths.

Chapter 5

Modified Algorithm

This chapter focuses on implementing modifications to the Parity-check Transformation Algorithm (PTA) in order to make more effective use of the soft channel information available to the decoder. Several changes are motivated and introduced. A full description of the resulting modified algorithm is presented, along with a worked example of it decoding a codeword. Finally, the per-iteration time complexity of the algorithm is analysed.

The analysis of the PTA performed in Chapter 4 highlights several potential avenues for the decoder to make better use of available soft symbol information. Based on this, modifications to the PTA are introduced, resulting in a new symbol-level belief-propagation-like decoder for Reed-Solomon (RS) codes.

5.1 Modifications and Motivation

As noted in Chapter 4, the PTA makes limited use of the soft-decision channel information contained in the $\mathbf{R}$ matrix. This information is used by the PTA only to perform a relative classification of each symbol position as either one of the $n-k$ most or k least-reliable symbols.

In order to make more effective use of the channel information, several changes to the PTA are introduced, primarily drawing inspiration from the Belief Propagation (BP) algorithm introduced in Chapter 2. The final changes described in this chapter were arrived at after extensive experimentation using the simulation environment described in Chapter 3.

Unlike the Adaptive Belief Propogation (ABP) algorithm [19], which performs a binary expansion step on the parity-check matrix in order to perform traditional bit-level belief propagation, the goal of the modifications introduced in this chapter is to keep the resulting decoder completely compatible with the existing PTA, and continue to operate at the symbol level.

5.1.1 Check-node Messaging

As discussed in the previous chapter, with the regular PTA modelled as a message-passing algorithm, each check node instructs its connected variable nodes to increment or decrement the reliabilities of their corresponding hard-decision symbol states, based on whether the symbols it received summed to 0 over the finite field $GF(2^m)$. These messages are *broadcast* by each check node to all connected variable nodes indiscriminately, conceivably reducing the overall efficiency of the algorithm. This is in contrast to the BP algorithm, where the check nodes dispatch individual messages to each of their connected variable nodes.

While the BP algorithm performs a simple addition on the input Log Likelihood Ratios (LLRs), a different approach is required to function with the non-binary symbols used by RS codes.

The chosen method involves check nodes 'voting' on the preferred symbol states for each of their connected variable nodes. These votes are the result of calculating the symbol value, which would result in that check equation being equal to 0, given the current state of all other connected variable nodes. Votes are implemented by adding an amount, δ, to the selected symbol's reliability value in the $\mathbf{R}$ matrix. A worked example of this can be seen in Section 5.3.

While calculating individual votes for each connected variable node requires additional work per iteration compared to the simple binary broadcast used by the PTA, it introduces two major benefits.

First it allows the check node to directly indicate the preferred state of a variable node from its perspective, which means that check nodes can modify a reliability in the $\mathbf{R}$ matrix for symbol values other than the currently selected Hard-Decision (HD) symbol. This has the direct benefit of allowing it to more rapidly increase the reliability of a correct (but not yet selected) symbol state without first wasting iterations decreasing the currently selected hard-decision symbol's reliability.

Secondly, this change helps prevent a check node from erroneously reducing the reliability of correct symbol values when another incorrect variable node causes its check to fail.

The equation describing the symbol value which check node i believes variable node j should have is:

$$V_{i,j} = \frac{\sum_{j' \in B_i, j' \neq j} E_{i,j}}{H^{\gamma}_{i,j}}, \tag{5.1}$$

with all operations performed over the code's $GF(2^m)$.

The notation B_i introduced in (5.1) is the set encompassing the indices of the variable nodes connected to check node i:

$$B_i = \{j \ \forall H^{\gamma}_{i,j} \neq 0\}. \tag{5.2}$$

Additionally, the matrix $E_{i,j}$ introduced in (5.1), is defined as:

$$E_{i,j} = x_i \times H^{\gamma}_{i,j}, \tag{5.3}$$

and contains the results of an element-wise multiplication of the current hard-decision vector with each row of the parity-check matrix. This represents the current "*check node state*" of the system.

5.1.2 Dynamic δ Values

As shown in Chapter 4, each iteration of the PTA updates reliabilities in the $\mathbf{R}$ matrix using a pre-defined, constant value for δ.

Generating dynamic values for δ is a compelling use case for the soft-decision channel information ignored by the PTA.

If variable nodes include their *a priori* reliability values along with their current symbol state in the messages sent to their connected check nodes, their *extrinsic* reliability can be calculated. This can, in turn, be used as a weighting, or δ for each of the check node's votes. This weighting ensures that the votes generated by

check nodes connected to highly reliable input symbols have a greater impact on the system than votes calculated using messages from less reliable variable nodes. This weighting is independent of a node being classified as '*reliable*' or '*unreliable*' by the parity-check transformation step, and can be used to further reduce the negative effects of the high density **H** matrix.

Reliability Combination Function

During the development of the modified algorithm, several functions were tested to generate the value for a dynamic δ. The best performing among the test set was a simple scaled average of the participating node reliabilities (that is all connected variable-node reliabilities, excluding the node where the vote is currently being sent).

This function defined as:

$$\delta_{i,j} = \frac{1}{\lambda} \cdot \frac{\sum_{j' \in B_i, j' \neq j} r_{j'}}{|B_i| - 1}, \tag{5.4}$$

is the δ value used by check node i voting on the state of variable node j. The static scaling variable λ is introduced to ensure that a single vote does not completely drown out the existing state of the system. $|B_i|$ is the cardinality, or number of elements in the set B_i, introduced previously in (5.2).

All votes are added back into the R matrix, and **boost** the reliability of symbol states which received votes.

5.1.3 Normalisation Step

With the removal of negative votes from the check nodes, a normalisation step is introduced to implicitly punish the reliability scores of symbol values which do not receive votes in a given iteration.

After all of the weighted votes have been added to the $\mathbf{R}^{(t)}$ matrix, producing $\mathbf{R}'^{(t)}$, each of its columns is re-normalised to ensure that the sum of all elements in each column add up to 1:

$$R_{i,j}^{(t+1)} = \frac{R'_{i,j}}{\sum_{i'=1}^{2^m} R'_{i',j}}, \tag{5.5}$$

resulting in a new $\mathbf{R}^{(t+1)}$ matrix to be used by the next iteration.

This change to the algorithm solves the PTA's issue of negative reliability values occurring in the $\mathbf{R}$ matrix, and allows for the relative magnitudes of each check node's vote weightings to vary without risking unbalancing the system.

This modification also ensures that the $\mathbf{R}$ matrix remains in a valid state after each iteration, allowing for its final values to be used as soft-output, similar to other algorithms such as [19]. The soft-output can be utilised by other components within the system such as the demodulator for equalisation or improved channel estimation.

5.2 Modified Algorithm Overview

The changes described in the previous section were applied to the PTA, resulting in a new symbol-level belief-propagation-like decoder for Reed-Solomon codes.

The modified algorithm performs the following steps when decoding a codeword:

1. The first 3 steps of the algorithm are identical to the PTA:
   (a) Find the current HD vector $\mathbf{x}^{(t)}$ and corresponding reliabilities $\mathbf{r}^{(t)}$ from $\mathbf{R}^{(t)}$.
   (b) Sort the reliabilities into two classes of *reliable* or *unreliable*.
   (c) Transform the systematic $\mathbf{H}$ matrix through Gaussian row operations into $\mathbf{H}^{\gamma(t)}$ such that the high-density columns are in the positions of the most-reliable symbols.
2. The syndrome is calculated for the current HD vector $\mathbf{x}^{(t)}$, if it is 0, the first k symbols of the $\mathbf{x}^{(t)}$ vector are returned along with the current state of the $\mathbf{R}^{(t)}$ matrix.
3. Variable nodes send their current symbol value, along with their reliability magnitude, to all connected check nodes in $\mathbf{H}^{\gamma(t)}$.
4. Check nodes generate 'votes' for each connected variable node by calculating what symbol value would be required for the node's parity-check equation to be satisfied, assuming the state of all other contributing variable nodes is correct.
5. Each output vote is *weighted* by calculating the average of the contributing input symbol reliabilities, and scaled by a constant, λ.
6. Votes are added to the current $\mathbf{R}^{(t)}$ matrix, producing $\mathbf{R}'^{(t)}$. This is done by

adding the vote weighting to the row at the index corresponding to the vote value for a given symbol. Once all votes are added, each column in the $\mathbf{R}'^{(t)}$ matrix is normalised to produce $\mathbf{R}^{(t+1)}$.

7. If the maximum number of iterations has been reached, the first k symbols are returned along with the current state of $\mathbf{R}^{(t)}$. Otherwise, the process begins again with the $\mathbf{R}^{(t+1)}$ matrix as an input.

A high-level flow diagram of the algorithm is shown in Figure 5.1, highlighting the section of the PTA which is modified in order to produce the new algorithm. The additional use of soft-decision information over the standard PTA is also highlighted.

5.3 Worked Example

A worked example of the modified algorithm decoding a systematic RS(7,3) codeword is presented in order to further enhance understanding. This builds on the worked example of the original PTA presented in Chapter 4, using the received matrix:

$$\mathbf{R}^{(1)} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 \end{array} \\ \left[\begin{array}{ccccccc} 0.1499 & 0.1014 & 0.1612 & \mathbf{0.3297} & \mathbf{0.2830} & 0.1720 & 0.0995 \\ \mathbf{0.3265} & 0.0809 & \mathbf{0.2143} & 0.1665 & 0.1332 & \mathbf{0.4140} & 0.1771 \\ 0.1192 & \mathbf{0.2012} & 0.1580 & 0.1739 & 0.2166 & 0.1077 & 0.1197 \\ 0.1882 & 0.1397 & 0.2076 & 0.1249 & 0.1188 & 0.1398 & \mathbf{0.2665} \\ 0.0310 & 0.0854 & 0.0389 & 0.0340 & 0.0444 & 0.0245 & 0.0467 \\ 0.0370 & 0.0700 & 0.0421 & 0.0301 & 0.0349 & 0.0273 & 0.0638 \\ 0.0645 & 0.1927 & 0.0834 & 0.0769 & 0.0994 & 0.0529 & 0.0862 \\ 0.0836 & 0.1287 & 0.0945 & 0.0641 & 0.0698 & 0.0618 & 0.1406 \end{array}\right] \end{array}. \tag{5.6}$$

5.3.1 Decoding - Iteration 1

Given the received $\mathbf{R}$ matrix shown in (5.6), the hard-decision vector:

$$\mathbf{x}^{(1)} = \begin{bmatrix} 1 & 2 & 1 & 0 & 0 & 1 & 3 \end{bmatrix}, \tag{5.7}$$

its corresponding reliabilities:

$$\mathbf{r}^{(1)} = \begin{bmatrix} 0.3265 & 0.2012 & 0.2143 & 0.3297 & 0.2830 & 0.4140 & 0.2665 \end{bmatrix}, \tag{5.8}$$

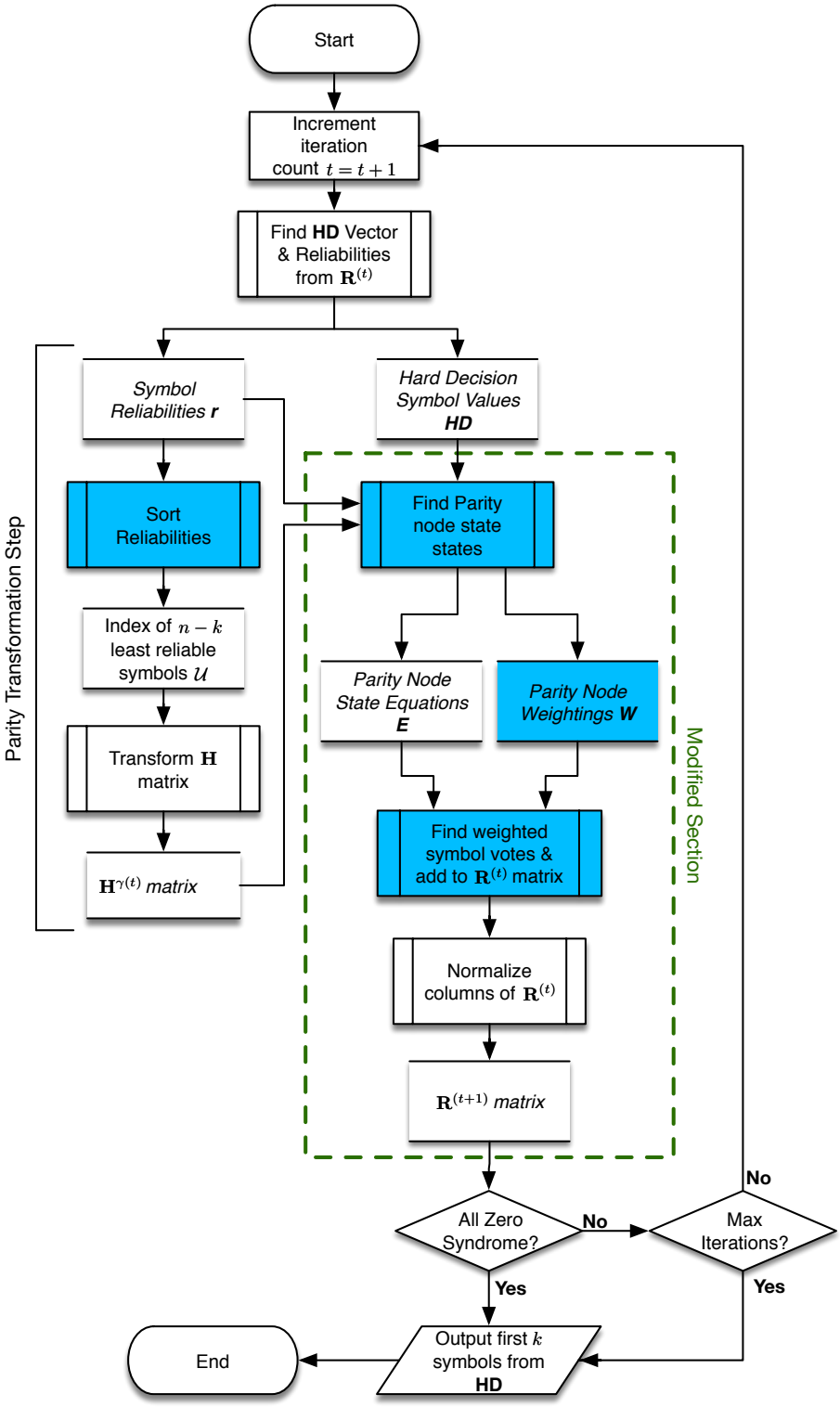


Figure 5.1: High-level overview of the modifications made to the PTA. Highlighted blocks indicate additional usage of soft-decision information by the algorithm.

and transformed $\mathbf{H}$ matrix:

$$\mathbf{H}^{\gamma(1)} = \begin{bmatrix} 5 & 1 & 0 & 3 & 0 & 7 & 0 \\ 5 & 0 & 1 & 2 & 0 & 6 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 4 & 0 & 0 & 2 & 0 & 7 & 1 \end{bmatrix}, \tag{5.9}$$

are all found in the same way as shown in Section 4.2 of Chapter 4.

The syndrome is calculated as:

$$\mathbf{s}^{(1)} = \begin{bmatrix} 0 & 0 & 2 & 0 \end{bmatrix}, \tag{5.10}$$

indicating that the codeword is not valid and requires additional iterations of the algorithm to decode.

Calculate Votes

The transformed $\mathbf{H}^{\gamma(1)}$ matrix is represented visually as a Tanner graph in Figure 5.2.

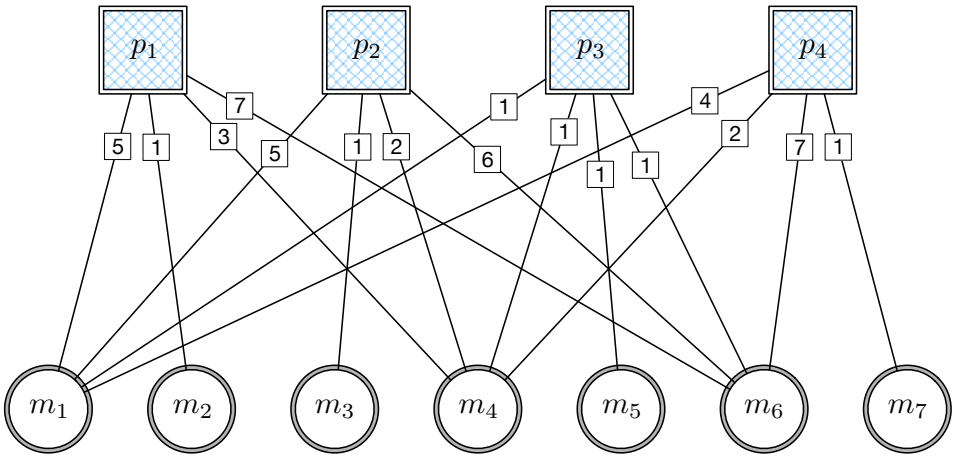


Figure 5.2: A Tanner graph visualising the transformed parity-check matrix $\mathbf{H}^{\gamma(1)}$ shown in (5.9). This graph will be used to route messages sent between variable and check nodes during the first iteration of the algorithm.

For each check node in the system, the symbol state votes (and corresponding vote weightings) are now calculated for each of their connected variable nodes.

Using a simple vote scaling value of $\lambda = 10$, a visual representation of the calculations occurring at check node p_1 can be seen in Figure 5.3.

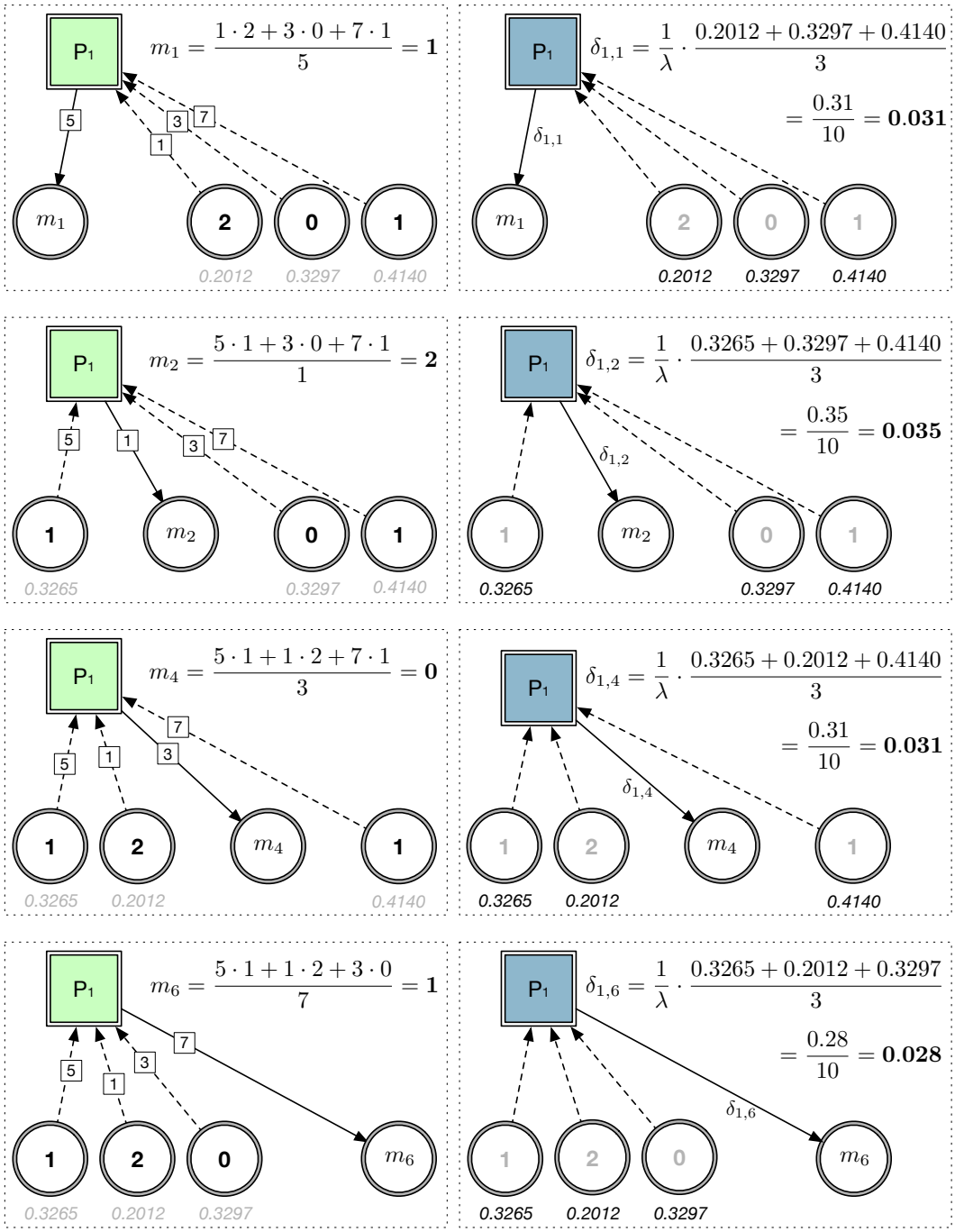


Figure 5.3: Check node p_1 is shown calculating symbol values over $GF(8)$ (left) and corresponding weightings (right) for the votes it contributes to its connected variable nodes during the first iteration of the algorithm. The resultant votes for the connected variable nodes $\{m_1, m_2, m_4, m_6\}$ are the symbol values {1, 2, 0, 1} with vote weightings of {0.031, 0.035, 0.031, 0.043}.

The remaining three check nodes function similarly, resulting in the aggregated votes for the first iteration seen below:

$$\mathbf{V}^{(1)} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 0 & 0 & +0.0991 & +0.0357 & 0 & 0 \\ +0.0994 & 0 & 0 & +0.0318 & 0 & +0.0906 & 0 \\ 0 & +0.0357 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & +0.0357 & 0 & 0 & 0 & +0.0357 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ +0.0319 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & +0.0290 & 0 \end{array}\right] \end{array}. \tag{5.11}$$

The $\mathbf{V}$ matrix, shown purely for illustrative purposes in (5.11), is structured identically to the $\mathbf{R}$ matrix, with columns representing each symbol position within the codeword, and the row index representing a possible symbol value.

The vote magnitudes in $\mathbf{V}^{(1)}$ are added to the existing symbol reliability magnitudes in $\mathbf{R}^{(1)}$. The result is then normalised column wise to ensure that the sum of all values in each column remains 1, producing:

$$\mathbf{R}^{(2)} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 \end{array} \\ \left[\begin{array}{ccccccc} 0.1325 & 0.0979 & 0.1556 & \mathbf{0.3791} & \mathbf{0.3077} & 0.1536 & 0.0961 \\ \mathbf{0.3765} & 0.0781 & 0.2069 & 0.1754 & 0.1286 & \mathbf{0.4507} & 0.1710 \\ 0.1054 & \mathbf{0.2287} & 0.1526 & 0.1538 & 0.2091 & 0.0962 & 0.1156 \\ 0.1664 & 0.1349 & \mathbf{0.2349} & 0.1104 & 0.1147 & 0.1249 & \mathbf{0.2917} \\ 0.0274 & 0.0825 & 0.0376 & 0.0301 & 0.0429 & 0.0219 & 0.0451 \\ 0.0609 & 0.0676 & 0.0406 & 0.0266 & 0.0337 & 0.0244 & 0.0616 \\ 0.0570 & 0.1861 & 0.0805 & 0.0680 & 0.0960 & 0.0472 & 0.0832 \\ 0.0739 & 0.1243 & 0.0912 & 0.0567 & 0.0674 & 0.0811 & 0.1357 \end{array}\right] \end{array}, \tag{5.12}$$

to be used as an input to the next iteration of the algorithm.

After the first iteration of the algorithm, the resulting change to each of the symbol reliabilities is:

$$\Delta\mathbf{R} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 \end{array} \\ \begin{bmatrix} -0.0174 & -0.0035 & -0.0056 & +0.0494 & +0.0247 & -0.0184 & -0.0034 \\ +0.0500 & -0.0028 & -0.0074 & +0.0089 & -0.0046 & +0.0367 & -0.0061 \\ -0.0138 & +0.0275 & -0.0054 & -0.0201 & -0.0075 & -0.0115 & -0.0041 \\ -0.0218 & -0.0048 & +0.0273 & -0.0145 & -0.0041 & -0.0149 & +0.0252 \\ -0.0036 & -0.0029 & -0.0013 & -0.0039 & -0.0015 & -0.0026 & -0.0016 \\ +0.0239 & -0.0024 & -0.0015 & -0.0035 & -0.0012 & -0.0029 & -0.0022 \\ -0.0075 & -0.0066 & -0.0029 & -0.0089 & -0.0034 & -0.0057 & -0.0030 \\ -0.0097 & -0.0044 & -0.0033 & -0.0074 & -0.0024 & +0.0193 & -0.0049 \end{bmatrix} \end{array}, \tag{5.13}$$

where $\Delta\mathbf{R} = \mathbf{R}^{(2)}$ - $\mathbf{R}^{(1)}$.

5.3.2 Decoding - Iteration 2

Beginning the second iteration, the identical process is followed. The hard decision vector:

$$\mathbf{x}^{(2)} = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 1 & 3 \end{bmatrix}, \tag{5.14}$$

and its corresponding reliabilities are extracted from $\mathbf{R}^{(2)}$:

$$\mathbf{r}^{(2)} = \begin{bmatrix} 0.3765 & 0.2287 & 0.2349 & 0.3791 & 0.3077 & 0.4507 & 0.2917 \end{bmatrix}. \tag{5.15}$$

The relative positions of the *most* and *least* reliable symbols are the same as the previous iteration, and as a result the transformed parity check matrix:

$$\mathbf{H}^{\gamma(2)} = \begin{bmatrix} 5 & 1 & 0 & 3 & 0 & 7 & 0 \\ 5 & 0 & 1 & 2 & 0 & 6 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 4 & 0 & 0 & 2 & 0 & 7 & 1 \end{bmatrix}, \tag{5.16}$$

is the same as $\mathbf{H}^{\gamma(1)}$ in (5.9).

The HD vector's syndrome is then calculated as:

$$\mathbf{s}^{(2)} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}, \tag{5.17}$$

which is the *zero*-syndrome, indicating that a valid codeword has been found.

The first k symbols of $\mathbf{x}^{(2)}$ are therefore returned as the decoded message:

$$\mathbf{m} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}, \tag{5.18}$$

along with the contents of $\mathbf{R}^{(2)}$ shown in (5.12) as the decoder's soft output information.

5.4 Time Complexity Analysis

Like the analysis of the PTA performed in Chapter 4, the modified algorithm will be analysed in order to determine its per-iteration time complexity, allowing for a fair comparison of overall computational complexity between the algorithms.

5.4.1 Hard-decision Extraction and Parity-check Matrix Transformation

The first phase of the modified algorithm is identical to the steps followed by the PTA and outlined in Sections 4.3.2 to 4.3.4 of Chapter 4.

In this phase, the HD vector, along with corresponding reliabilities, are extracted from the current $\mathbf{R}^{(t)}$ matrix. The reliabilities are sorted and the transformed parity-check matrix $\mathbf{H}^{\gamma(t)}$ is found.

The worst-case time complexity to perform this step is:

$$\mathcal{O}(2^m \cdot n + n^2 + (n-k)^3). \tag{5.19}$$

As with the PTA, should the required set of transformed parity-check matrices be pre-calculated, then the time complexity of this phase reduces to:

$$\mathcal{O}(2^m \cdot n + n^2 + 1). \tag{5.20}$$

5.4.2 Syndrome Check

As a stopping condition of the algorithm, the current HD vector's syndrome must be calculated in order to check if a valid codeword has been found. This involves finding the vector-matrix product between the HD vector and the current $\mathbf{H}^{\gamma(t)}$ matrix, with a time complexity of:

$$\mathcal{O}(n \cdot (n-k) + (n-k)). \tag{5.21}$$

5.4.3 Calculate Parity-node Symbol State Matrix

The overall check-node state, or $\mathbf{E}$ matrix described in Section 5.1.1, is used to represent the input symbol states at each check node, once scaled by the graph edges defined in the parity-check matrix. To calculate this, the n symbols of the current hard-decision vector are multiplied by each of the n elements in the $(n-k)$ rows in the parity-check matrix. This results in a time complexity of:

$$\mathcal{O}(n \cdot n(n-k)). \tag{5.22}$$

5.4.4 Calculate Parity Reliability Matrix

The check-node reliability matrix is a representation of the reliability magnitudes (or weights) to be emitted with each vote sent to the variable nodes connected to a given check node.

To reduce the number of redundant computations, the total sum of the input reliabilities at a check node is first found. The output weight is then calculated by subtracting the variable node reliability vector from the total sum, dividing the resulting vector by the number of connected variable nodes, and finally masking off the positions in the vector not connected to the check node. This is performed for each of the $(n-k)$ check nodes, requiring a worst case complexity of:

$$\mathcal{O}(n \cdot (n-k) + n \cdot (n-k) + 5n \cdot (n-k)), \tag{5.23}$$

which simplifies to:

$$\mathcal{O}(7n \cdot (n-k)). \tag{5.24}$$

5.4.5 Calculate Votes / Update Reliability

Using the results of the previous two phases, each of the $(n-k)$ check nodes now 'votes' on the state of its connected variable nodes.

For each check-node, the sum of all input values is found using the check-node state matrix $\mathbf{E}$. An individual vote value sent to variable node j is calculated by subtracting the current state of j from this total. The number of operations required to find the sum for each check node is:

$$\mathcal{O}(n \cdot (n-k)). \tag{5.25}$$

The implemented algorithm requires 3 constant time operations to calculate the individual vote value from the sum calculated above, divide it by the co-efficient on the edge connecting the parity and variable nodes, and, finally, add the vote weighting to the $\mathbf{R}$ matrix at the resulting index. If the number of connected variable nodes at each check node is represented by the average weight of each column in the $\mathbf{H}$ matrix, $\mathcal{W}_{\mathbf{H}}$, the number of operations required is:

$$\mathcal{O}(3\mathcal{W}_{\mathbf{H}} \cdot (n-k)). \tag{5.26}$$

Finally, each of the n columns in the $\mathbf{R}$ matrix have their 2^m elements normalised. This first requires a pass to calculate the sum of all elements in a column, and a second pass to normalise them:

$$\mathcal{O}(2n \cdot 2^m). \tag{5.27}$$

5.4.6 Total Operations

Combining the complexity of all of the above phases, the worst-case total number of operations performed in a single iteration of the modified algorithm is represented as:

$$\mathcal{O}(2^m + n^2 + (n-k)^3 + n(n-k) + 7n(n-k) + 3\mathcal{W}_{\mathbf{H}}(n-k) + n(n-k) + 2^m \cdot 2n).$$

By using the known relationships of $\mathcal{W}_{\mathbf{H}} = k + 1$ and $2^m = n + 1$, introduced in Chapter 2 this can be simplified to:

$$\mathcal{O}(n^3 + 4n^2 + 16n + 3k^2n + 3kn - k^3 - 3k^2 - 3kn^2 - 13k), \tag{5.28}$$

and with codes short enough to pre-compute all required parity-check matrix transforms, (i.e. the $\mathcal{O}(n-k)^3$) component becomes a linear time lookup of $\mathcal{O}(1)$), further reducing to:

$$\mathcal{O}(4n^2 + 16n + 3kn - 3k^2 - 13k + 1). \tag{5.29}$$

In both cases, the modified algorithm requires more operations per iteration than the PTA. However, using the Big-$\mathcal{O}$ simplification rules introduced in Section 2.7 of Chapter 2, the worst case per iteration complexity of the algorithm can be approximated as $\mathcal{O}(n^3 - k^3)$, and $\mathcal{O}(n^2 - k^2)$ when utilising pre-computed parity-check matrix transforms. These are of the same order of per-iteration complexity shown in Chapter 4 for the PTA for large values of n and k.

5.5 Conclusion

Modifications to the PTA decoder, introduced in order to improve the algorithm's use of soft symbol information, resulted in a new symbol-level, belief-propagation-like, iterative decoder for Reed-Solomon codes.

The changes and resulting decoder have been described in detail, and the process of decoding a received codeword demonstrated through a worked example.

The time complexity of the new decoder was analysed. While its per-iteration complexity is of the same order as the PTA, for small values of n and k the modified decoder requires additional operations per iteration.

Chapter 6

Results and Analysis

This chapter presents the results of simulations comparing the relative performance of the Parity-check Transformation Algorithm (PTA) decoder and variations of the modified belief-propagation-like decoder introduced in Chapter 5. The primary results are concerned with both the effective symbol error rates and overall computational complexity of each decoder. A simple analysis into optimising the parity-check vote-weighting function of the modified decoder is also performed.

The research presented is primarily interested in understanding the performance impact of modifications introduced to the PTA in an attempt to make more effective use of available soft-decision information. The deterministic, reproducible simulation environment, as described in Chapter 3, is used to compare the various decoders under identical test conditions.

In the interest of achieving a fair test, the simulation parameters are chosen in order to ensure that the PTA produces its best-known error-correction performance results. Specifically, by using a half-rate RS(15,7) code, with the PTA's δ parameter set to 0.001 [13].

Additionally, in order to understand the effects of the parity-check matrix transformation step, variations of the modified algorithm are presented. These include a configuration that, like the PTA, performs a parity transformation step during every iteration, a simplified mode that performs the transformation only once at the start of decoding a codeword, and, finally, a version that forgoes the transformation step altogether. These different versions are labelled **Multi Transform**, **Single Transform** and **No Transform** respectively.

6.1 Symbol Error-rate Performance Analysis

The results of simulations comparing the PTA and multiple versions of the modified algorithm, seen in Figure 6.1, show an improvement in symbol error-rate performance so long as the parity-check matrix is transformed at least once. This highlights the effectiveness of the parity-check matrix transformation stage in reducing error propagation in iterative High-Density Parity-Check (HDPC) decoders and further re-enforces the value of the PTA's matrix transform step.

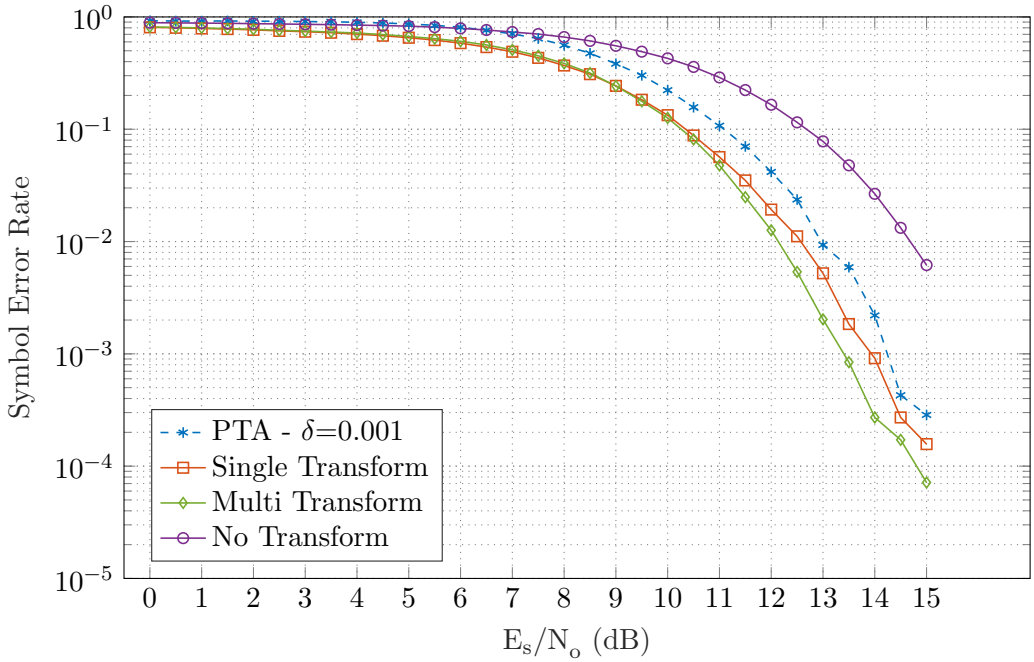


Figure 6.1: Comparative Symbol Error-Rate performance over a 16-QAM AWGN channel between the PTA decoder and variations of the modified algorithm at different SNR values.

At $\frac{E_s}{N_o}$ greater than 10dB, the benefits of transforming the **H** matrix after every iteration become clear. While performing the parity-check matrix transformation, every iteration is more computationally expensive; it also results in an approximately 0.5dB gain in error-correction capability.

For symbol error rates between 10^{-2} and 10^{-4}, the modified algorithm achieves a relatively uniform coding gain of 1dB over the PTA.

6.2 Computational Complexity Analysis

The time complexity analysis performed on both the PTA (in Chapter 4) and the modified algorithm in (Chapter 5) indicate that both decoders have the same approximate per-iteration time complexity on the order of $\mathcal{O}(n^3 - k^3)$. This result makes it feasible to compare the decoders using the *number of iterations* required to converge on a valid codeword as a good proxy for the overall computational complexity of the decoders. Comparing the convergence rate of the algorithms using the number of iterations also allows for a good abstracted understanding of the high-level performance characteristics of each algorithm at different $\frac{E_s}{N_o}$ ratios.

It is critical to note that the time complexity of each algorithm is shown to be approximately $\mathcal{O}(n^3 - k^3)$ only by assuming large values of both n and k. As a result, there are material differences in the actual number of operations required by each decoder with the simulation parameters used to generate the results presented here. Section 6.2.3 discusses the critical results presented here as an average number of *operations* required to decode a codeword in order to compensate for the varying per-iteration complexity of each algorithm.

6.2.1 Average Iteration Count

The average number of iterations required for each version of the algorithm to converge on a valid codeword can be seen in Figure 6.2.

The PTA performs as expected for a small value of δ based on existing results found in the literature [14, 13], often requiring several hundred iterations in order to converge on a codeword.

All versions of the modified algorithm show a *significant* decrease in the required number of iterations per codeword, with near constant performance at all simulated $\frac{E_s}{N_o}$ values.

In order to better visualise the performance gap between decoders, the same results are plotted using a log scale y-axis in Figure 6.3. This further highlights the scale of improvement across all simulated Signal-to-Noise Ratio (SNR) values.

The average number of required iterations to decode a codeword at different SNR bands is shown in Table 6.1, along with an overall average performance improvement for each decoder configuration.

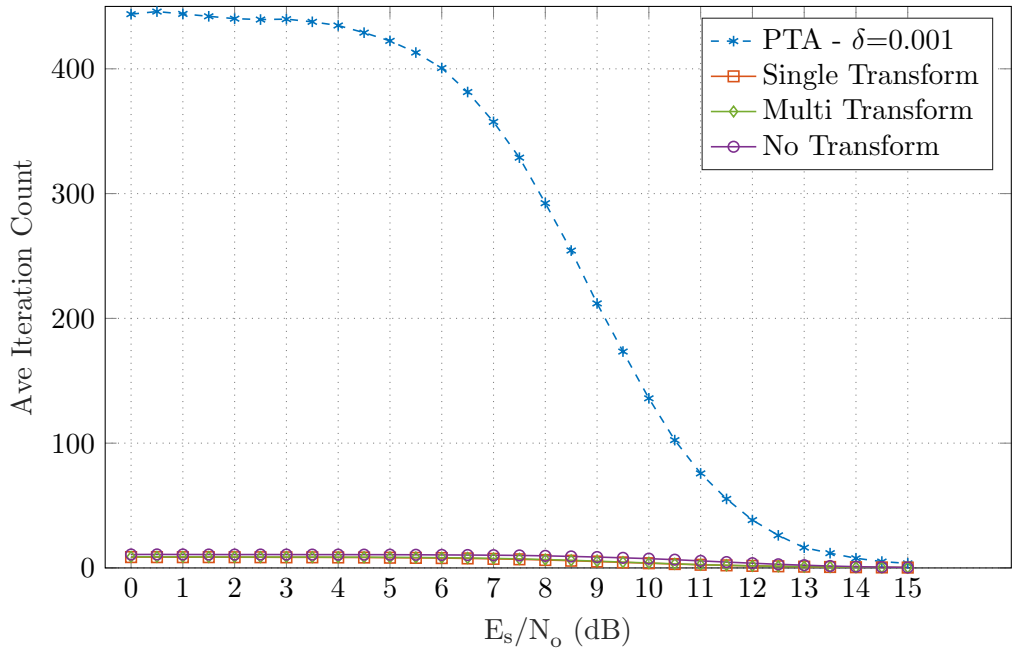


Figure 6.2: Average number of iterations required to converge on a valid codeword.

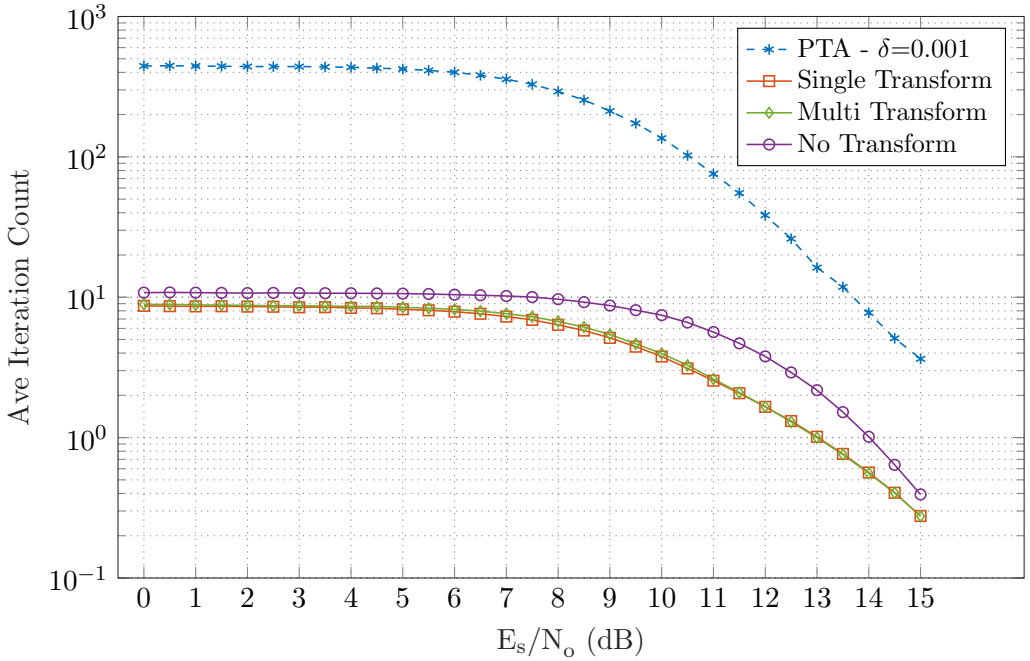


Figure 6.3: Log scale graph showing average number of iterations required to decode a codeword.

Table 6.1: Average number of iterations required by each decoder configuration at various SNR bands. The gain relative to the number of iterations required by the PTA is also shown.

	Average Iteration Count							
	0 → 5dB		*5 → 10dB*		*10 → 15dB*		*Overall*	
Decoder	Itr.	Gain	Iter.	Gain	Iter.	Gain	Iter.	Gain
PTA $\delta = 0.001$	438.09	-	306.53	-	43.50	-	261.64	-
Single Permute	8.51	51.48	6.50	47.16	1.59	27.36	5.50	47.57
Multi Permute	8.72	50.24	6.80	45.07	1.62	28.85	5.68	46.06
No Permute	10.72	40.87	9.58	31.99	3.35	12.99	7.80	33.54

The *multi-transform* configuration of the modified algorithm, shown to have the best Symbol Error Rate (SER) performance among the decoders tested in Section 6.1, achieved its error-correction gain, while simultaneously showing a **46.06x reduction** in the average number of iterations required to converge when compared to the PTA.

The fastest decoder under test to converge is the *single-transform* version of the modified algorithm, requiring on average **47.57x** fewer iterations than the PTA, however this comes at the expense of a slight loss in coding gain when compared to the *multi-transform* decoder.

Furthermore, in addition to improving in the decoder's SER performance, as shown in Section 6.1, it can be seen that performing the parity-check transformation step *also* improves the modified decoder's rate of convergence.

Transforming the **H** matrix every iteration, compared to only once per codeword, has a small effect on the decoder's convergence rate. In order to visualise the difference, the results must be plotted without the PTA results scaling the y-axis, as seen in Figure 6.4.

The results show a fractional reduction in the average iteration count when the **H** matrix is transformed only once. However, at higher $\frac{E_s}{N_o}$ values, where performing the transformation every iteration results in an improved symbol error rate, this difference collapses.

With an effectively identical number of iterations required per codeword, and a significantly reduced per-iteration complexity, transforming the parity-check matrix only once per codeword is the most computationally efficient configuration for the

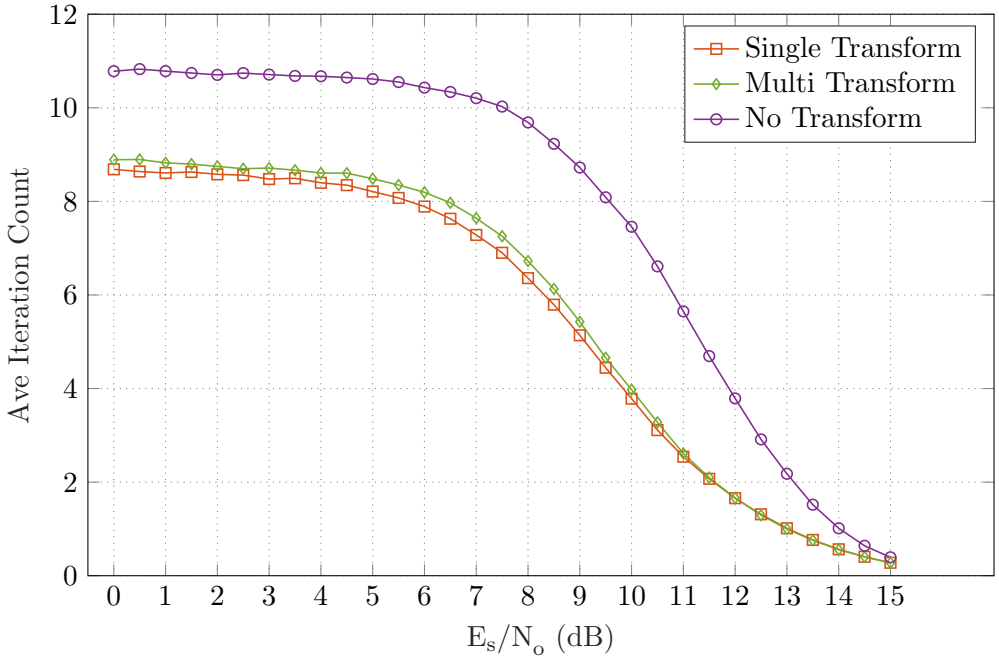


Figure 6.4: Average number of iterations required by variants of the modified algorithm.

modified algorithm. If, however, the **H** matrix transformations are pre-computed, as discussed in Section 4.3.4 of Chapter 4, the resulting complexity of both configurations is nearly identical, and the additional error-correction gain achieved by the *multi-transform* version of the algorithm, makes it the preferable configuration.

6.2.2 Maximum Iteration Count

The worst-case computational performance of each decoder is visualised in Figure 6.5 using the *maximum* number of iterations required to converge at each $\frac{E_s}{N_o}$ over all decoded codewords.

The modified algorithm again shows significant improvement over the PTA decoder.

Without the PTA affecting the scale of the results in Figure 6.6, the value of the parity-check transformation step is further highlighted. It can be seen that performing the parity-check transformation reduces the maximum number of iterations required by the algorithm. Performing the transformation at every iteration additionally reduces the variability of the maximum number of iterations required across all SNR values. This reduced variability ensures that the decoder's worst-case number of iterations is far more bounded and predictable, a useful property for an iterative

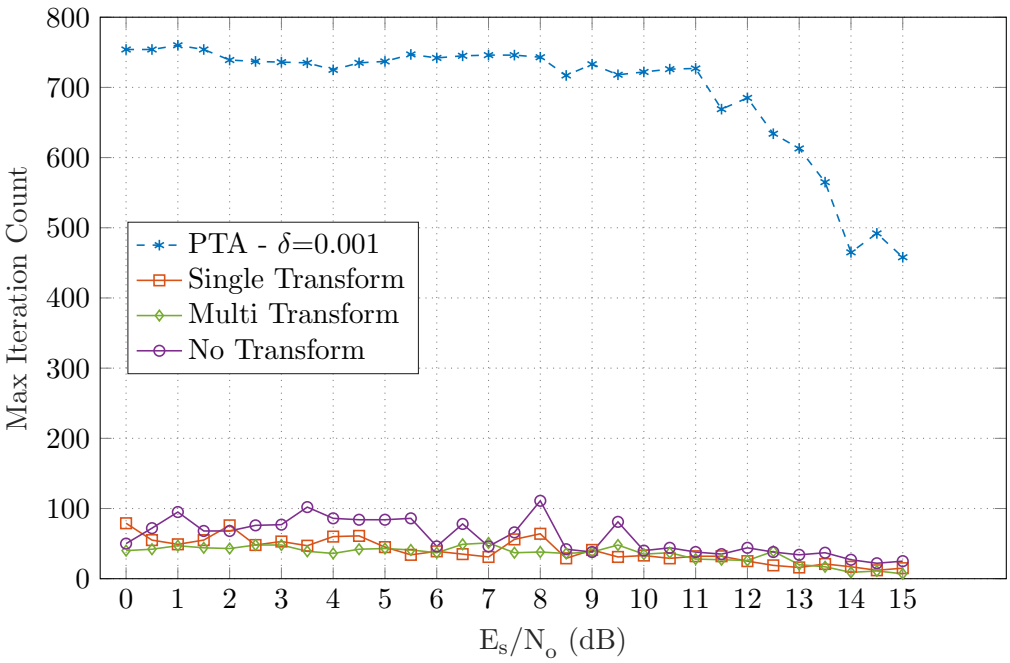


Figure 6.5: Maximum number of iterations required to decode a codeword between the PTA decoder and variations of the modified algorithm.

algorithm.

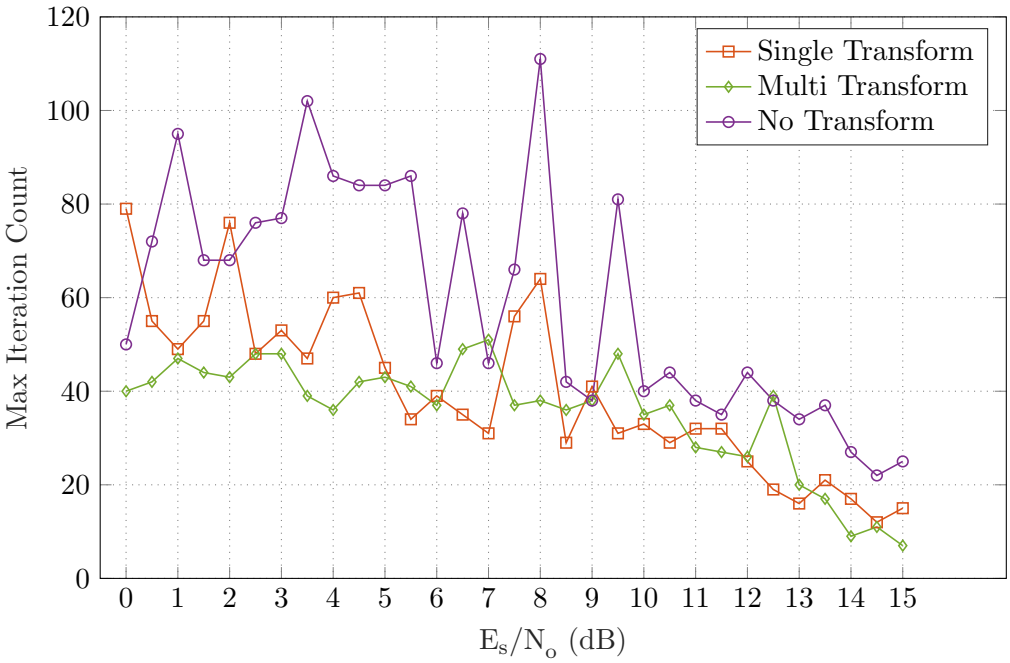


Figure 6.6: Maximum number of iterations required by the modified algorithm to decode a codeword.

6.2.3 Operations per Iteration

As discussed at the start of the chapter, while the worst-case time complexity of the PTA and modified algorithm are both shown to be of the order $\mathcal{O}(n^3 - k^3)$m, that approximation is valid only for large values of n and k.

Due to the relatively small values of $n = 15$ and $k = 7$ used in the simulations to produce the results discussed in this chapter, it is feasible to calculate the exact number of operations per iteration for the configurations under test.

The possibility of pre-computing the required set of transformed parity-check matrices and its effect on the number of operations required per iteration, and overall per codeword, are also taken into account.

Parity Transformation Algorithm

From the analysis performed in Chapter 4, the worst-case number of operations required per-iteration of the PTA is calculated in Example 6.1. Additionally, the reduced complexity form, calculated by assuming the required parity-check transformations have been pre-computed, is shown in Example 6.2

Example 6.1 (PTA Full Complexity)**.** Using the per-iteration time complexity of the PTA from (4.25) calculated in Chapter 4:

$$\mathcal{O}(n^3 + 4n^2 + 4n + 3k^2n - k^3 - k^2 - 3kn^2 - 2k + 1),$$

the number of operations required per iteration, for $n = 15$ and $k = 7$, is calculated as:

$$\begin{aligned} &= 15^3 + 4 \cdot 15^2 + 4 \cdot 15 + 3 \cdot 7^2 \cdot 15 - 7^3 - 7^2 - 3 \cdot 7 \cdot 15^2 - 2 \cdot 7 + 1 \\ &= 1410 \text{ operations per iteration.} \end{aligned}$$

Example 6.2 (PTA Pre-calculated Parity-check Matrix Transformations)**.** The reduced per-iteration complexity of the PTA when using pre-computed parity-check matrix transformations is shown to be:

$$\mathcal{O}(4n^2 + 4n - k^2 - 2k + 2),$$

and as a result, for $n = 15$ and $k = 7$, the decoder requires:

$$= 4 \cdot 15^2 + 4 \cdot 15 - 7^2 - 2 \cdot 7 + 2$$
$$= 899 \text{ operations per iteration.}$$

Modified algorithm

The number of operations required for each iteration of the *multi-transformation* configuration of the modified algorithm is calculated in Example 6.3, and the number of operations required per iteration when using pre-computed parity-check matrix transformations is shown in Example 6.4.

As discussed in Chapter 5, the parity-check matrix transformation step requires $\mathcal{O}((n-k)^3)$, or **512** operations under the current test conditions. Knowing this, the number of operations required by the other configurations of the decoder can be calculated.

Example 6.3 (Modified Algorithm, full complexity with parity-check matrix transformation in every iteration)**.** From (5.28), using the known full time complexity:

$$\mathcal{O}(n^3 + 4n^2 + 16n + 3k^2n + 3kn - k^3 - 3k^2 - 3kn^2 - 13k), \tag{6.1}$$

the number of operations required is calculated as:

$$= 15^3 + 4 \cdot 15^2 + 16 \cdot 15 + 3 \cdot 7^2 \cdot 15 + 3 \cdot 7 \cdot 15 - 7^3 - 3 \cdot 7^2 - 3 \cdot 7 \cdot 15^2 - 13 \cdot 7$$
$$= 1729 \text{ operations per iteration.}$$

Finally, the number of operations required per iteration when using pre-computed parity-check matrix transformations is shown in Example 6.4.

Example 6.4 (Modified Algorithm, reduced complexity with pre-computed parity-check matrix transformations)**.** As shown in (5.29) from Chapter 5, when pre-computing the transformed parity-check matrices, the per-iteration time complexity reduces to:

$$\mathcal{O}(4n^2 + 16n + 3kn - 3k^2 - 13k + 1). \tag{6.2}$$

Under the current test conditions, with $n = 15$ and $k = 7$, this equates to:

$$= 4 \cdot 15^2 + 16 \cdot 15 + 3 \cdot 7 \cdot 15 - 3 \cdot 7^2 - 13 \cdot 7 + 1$$
$$= 1218 \text{ operations per iteration.}$$

6.2.4 Overall Complexity Analysis

The worst-case number of operations required per iteration of each algorithm is summarised in Table 6.2.

Table 6.2: Summary of the worst-case number of operations required per-iteration by each decoder.

Algorithm	**Ops. Per Iteration (*Regular*)**	**Ops. Per Iteration (*Pre-calculated*)**
Regular PTA	1410	899
Modified - Single Transform	1217 (+512)	1217 (+1)
Modified - Multi Transform	1729	1218
Modified - Zero Transform	1217	1217

Combined with the results presented in Section 6.2.1, the average number of *operations* required to decode a RS(15,7) codeword over a 16-QAM AWGN channel at different $\frac{E_s}{N_o}$ values can be seen in Table 6.3.

The results presented show the overall computational complexity of the modified decoder is significantly improved over the PTA, even with the additional per-iteration overhead introduced by the modifications.

The modified algorithm is capable of converging up to **51.19x** faster than the PTA when the parity-check matrix is transformed only once. The *multi-transform* mode of the modified algorithm, requires **37.5x** fewer operations than the PTA (while simultaneously achieving a 1dB coding gain as shown in Section 6.1).

6.3 Parity-node Weighting Scaling Factor

The extreme difference in the number of iterations required by the PTA and modified algorithm is partially due to the magnitude of each algorithm's vote or δ value

Table 6.3: Average number of operations required by each decoder to converge.

	Average Operations Required			Overall Ave.	
Decoder	*0 → 5dB*	*5 → 10dB*	*10 → 15dB*	Ops.	Gain
Regular					
PTA $\delta = 0.001$	617,707	432,207	61,335	368,912	-
Single Permute	10,869	8,423	2,447	7,206	51.19
Multi Permute	15,077	11,757	2,800	9,821	37.56
No Permute	13,046	11,658	4077	9,492	38.86
Pre-calculated					
PTA $\delta = 0.001$	393,842	275,570	39,106	235,214	-
Single Permute	10,358	7,912	1,936	6,695	35.13
Multi Permute	10,621	8,282	1,973	6,918	34.00
No Permute	13,046	11,659	4,077	9,493	24.77

used. Larger δ values should allow the decoder to make bigger *steps* toward a valid codeword during each iteration.

In the case of the PTA, increasing the value of δ *reduces* the algorithm's error-correction capability [13], with the best error-correction performance achieved using a small δ on the order of 0.001.

As shown in Chapter 5, the modified decoder makes use of dynamic values for δ calculated using:

$$\delta_{i,j} = \frac{\Delta_{i,j}}{\lambda}, \tag{6.3}$$

where the static value λ is used to scale the resulting intrinsic reliability vote $\Delta_{i,j}$ from check node i to variable node j.

Across the entire simulation dataset used for this research, the average intrinsic check node reliability value is in the range $0.1541 \leq \Delta_{i,j} \leq 0.1650$.

For all results presented earlier in this chapter, a static value of $\lambda = 10$ is used, producing an effective range of δ values between 0.01541 and 0.01650. These values are a full order of magnitude larger than the δ used by the PTA, revealing one of the underlying reasons of how the modified algorithm converges on a valid codeword so rapidly compared to the PTA.

6.3.1 Symbol Error Rate vs. λ

In order to understand how the modified decoder performs for different values of δ, the tuning parameter λ is now varied.

The results of simulations comparing the relative Symbol Error Rate (SER) for different values of λ can be seen in Figure 6.7. The modified decoder is capable of matching the SER performance of the PTA at $\lambda = 7$ (or an effective average $\delta = 0.02$ across the simulated $\frac{E_s}{N_o}$ values). Using this value as the divisor would further improve the computational efficiency of the decoder while still maintaining SER parity with the PTA.

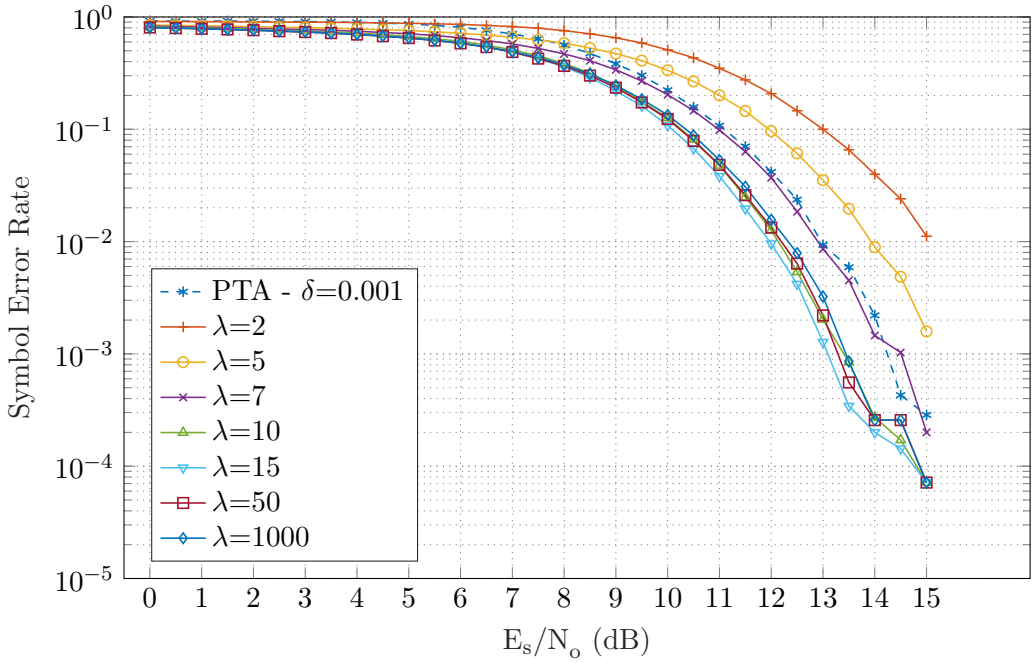


Figure 6.7: Symbol error rate of the modified decoder at various values of λ.

Reducing the size of δ was shown to always improve the symbol error-rate performance of the PTA. However, as seen in Figure 6.7, the modified algorithm appears to achieve its best SER performance with a value of $\lambda = 15$ (for the code rate tested in this research). Reducing the value of δ any further has a *negative* impact on the decoders SER performance.

This effect can be seen visually in Figure 6.8, where each line plotted shows the performance of the decoder at a different $\frac{E_s}{N_o}$ value. Here it is easy to see how increasing the size of λ improves the performance of the decoder, up until a local minima is reached at $\lambda = 15$. After this point, increasing the value of λ further (and

reducing the size of δ to values comparable to the PTA) has negative effects on both the symbol error rate *and* the number of iterations required to decode a codeword. This effect is clearly visible across all $\frac{E_s}{N_o}$ values tested.

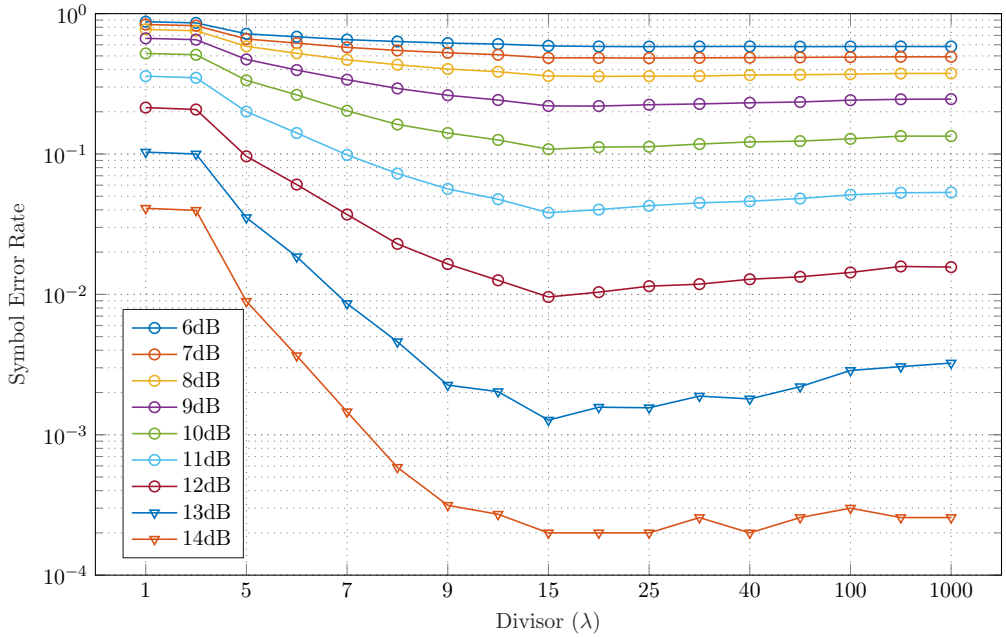


Figure 6.8: The symbol error rate of the modified decoder using different λ values. Each line is drawn for a different $\frac{E_s}{N_o}$ value in the simulation.

6.3.2 Iteration Count vs λ

The average number of iterations required to decode a codeword for different values of λ can be seen in Figure 6.9.

Intuitively, increasing the size of δ (by using a smaller λ value) *reduces* the number of iterations required by the decoder to converge on a codeword. However, similar to the effect seen in Section 6.3.1, reducing the size of λ beyond a certain point has negative effects on the number of iterations required by the decoder (while simultaneously worsening it's SER performance).

As seen in Figure 6.9, for the chosen code rate, the minimum number of iterations required by the decoder is seen for λ values between 7 and 10.

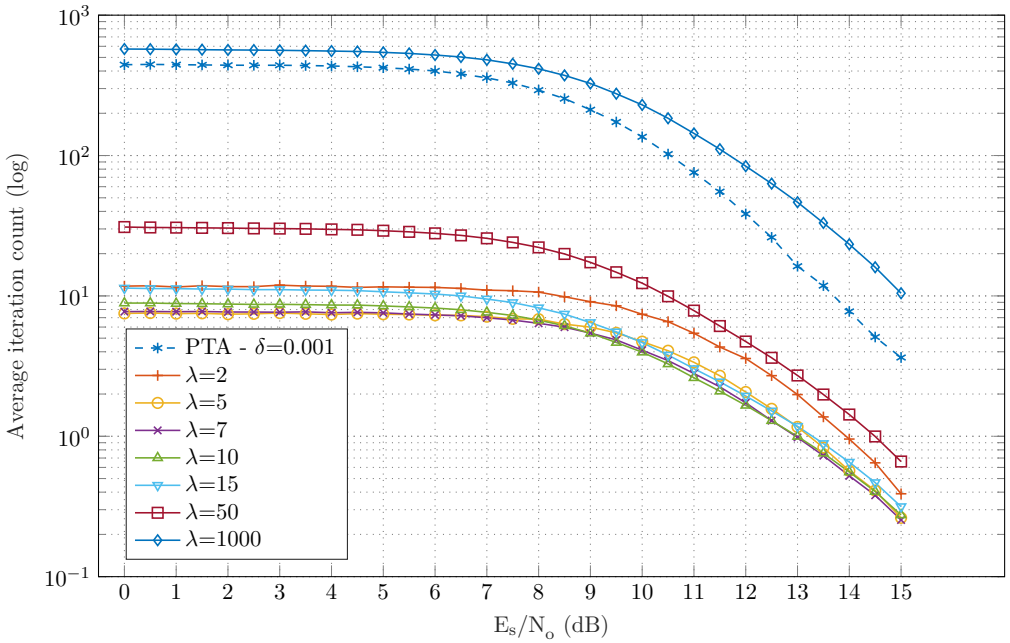


Figure 6.9: Average number of iterations required by the modified algorithm to decode a codeword using different values of the λ scaling factor.

6.3.3 Optimal λ value

In choosing an *optimal* λ value for the modified decoder, the trade-off between SER performance and iteration count must be considered.

From Figure 6.7, a decoder configured using $\lambda = 15$, ensuring optimal error correction capability, can achieve a **1dB** gain in SER performance over a decoder configured using the most computationally efficient value of $\lambda = 7$. However, the difference in the average number of iterations required between these λ values is almost negligible.

As a result, tuning the decoder's λ value to favour its optimal SER performance will result in the best balance in overall performance, with minimal compromise.

6.4 Conclusion

The results of simulations comparing the relative performance between the original PTA and the modified version, introduced in Chapter 5, were presented in this chapter.

Simulations consisting of *10,000* RS(15,7) codewords, sent over a 16-QAM AWGN

channel show an *improvement* in the error-correction capability of the modified decoder, with a **1dB** gain for symbol error rates between 10^{-2} and 10^{-3}. This is achieved while simultaneously reducing the number of iterations required to decode a codeword by an average factor of **46.06x**.

While the modified algorithm increases the computational complexity of each iteration, it was shown that it achieves an overall reduction in the average number of *operations* required to decode a codeword by a factor of **37.56x**.

Finally, a simple optimisation of the tunable λ parameter introduced by the modified algorithm was performed, and an optimal value of $\lambda = 15$ was found for the RS(15,7) codes used in this research.

Chapter 7

Conclusion and Future Work

The primary goal of this research was to determine the degree to which the performance of the iterative Parity-check Transformation Algorithm (PTA) decoder could be improved through more effective utilisation of available soft-decision channel information.

An analysis of the PTA (in Chapter 4) revealed potentially useful modifications that could be made to the algorithm in pursuit of this goal. The modifications (discussed in Chapter 5) resulted in a new symbol-level belief-propagation-like decoder for Reed-Solomon codes.

The modified algorithm improves upon the PTA through the use of dynamic δ values for symbol reliability updates. This is achieved by replacing the PTA's static broadcast-style reliability updates with a weighted symbol-level voting system based on current soft channel information. These changes, combined with the existing parity-check matrix transformation step, help accelerate decoding and further reduce the effects of error propagation in a message-passing decoder for non-binary High-Density Parity-Check (HDPC) codes such as Reed-Solomon codes.

The decoder is implemented and a relative performance study comparing it and the unmodified PTA through simulations show performance gains in error-correction capability, as well as a significant reduction in the overall computational complexity of the algorithm.

Over a 16-QAM AWGN channel, Chapter 6 shows that the new decoder achieves a 1dB error correction gain over the PTA for Symbol Error Rates between 10^{-2} and 10^{-4}, while simultaneously reducing the average number of iterations required to converge on a valid codeword by a factor of 46.06x for half-rate RS(15,7) codes.

These improvements come at the cost of a slight increase in per-iteration computational complexity. However, the reduction in the average number of iterations required by the algorithm to decode a codeword significantly outweighs this overhead, leading to an overall average reduction in complexity by a factor of 37.56x being observed.

7.1 Future Work

While the algorithm presented drastically improves on the performance of the PTA, it has not been thoroughly optimised or mathematically analysed. Additionally, a comparative performance study between the modified algorithm and other similar algorithms, such as the bit-level Adaptive Belief Propogation (ABP) algorithm, may also yield interesting results.

There are also several other avenues for future work on the algorithm including investigations into alternative functions for calculating the intrinsic reliabilities / vote weightings at the check nodes, or investigating additional vote scaling based on the connection density of the receiving variable nodes (in a similar way to how the current PTA compensates for some variable nodes receiving more votes per iteration than others).

Finally, with the speed gains achieved when using the modified algorithm, it may also be feasible to investigate its behaviour at longer code lengths and at more varied code rates.

Bibliography

[1] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.

[2] S. Lin and D. J. Costello, *Error control coding*. Prentice Hall Englewood Cliffs, 2004, vol. 2.

[3] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[4] J. S. Plank *et al.*, "A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems," *Software - Practise and Experience*, vol. 27, no. 9, pp. 995–1012, 1997.

[5] S. B. Wicker and V. K. Bhargava, *Reed–Solomon codes and their applications*. John Wiley & Sons, 1999.

[6] V. Guruswami and M. Sudan, "Improved decoding of Reed–Solomon and algebraic-geometric codes," *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pp. 28–37, 1998.

[7] R. M. Roth and G. Ruckenstein, "Efficient decoding of Reed–Solomon codes beyond half the minimum distance," *IEEE Transactions on Information Theory*, vol. 46, no. 1, pp. 246–257, 2000.

[8] D. Chase, "Class of algorithms for decoding block codes with channel measurement information," *IEEE Transactions on Information theory*, vol. 18, no. 1, pp. 170–182, 1972.

[9] R. Koetter and A. Vardy, "Algebraic soft-decision decoding of Reed–Solomon codes," *IEEE Transactions on Information Theory*, vol. 49, no. 11, pp. 2809–2825, 2003.

[10] E. R. Berlekamp, *Algebraic coding theory*. McGraw-Hill, 1968.

[11] Li, Xuemei and Zhang, Wei and Liu, Yanyan, "Efficient architecture for algebraic soft-decision decoding of reed–solomon codes," *IET Communications*, vol. 9, no. 1, pp. 10–16, 2014.

[12] C. Clarke, "Reed–Solomon error correction," *BBC R&D White Paper, WHP*, vol. 31, 2002.

[13] O. Ogundile, Y. Genga, and D. Versfeld, "Symbol level iterative soft decision decoder for Reed–Solomon codes based on parity-check equations," *Electronics Letters*, vol. 51, no. 17, pp. 1332–1333, 2015.

[14] Y. Genga and D. Versfeld, "The modified soft input parity check transformation algorithm for reed solomon codes," *SAIEE Africa Research Journal*, vol. 108, no. 1, pp. 24–30, 2017.

[15] R. Gallager, "Low-density parity-check codes," *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[16] R. Lucas, M. Bossert, and M. Breitbach, "On iterative soft-decision decoding of linear binary block codes and product codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 276–296, Feb 1998.

[17] O. Ogundile and D. Versfeld, "Improved reliability information for rectangular 16-QAM over flat Rayleigh fading channels," in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*. IEEE, 2014, pp. 345–349.

[18] S. J. Johnson, "Introducing Low-Density Parity-Check Codes," School of Electrical Engineering and Computer Science, The University of Newcastle.

[19] J. Jiang and K. R. Narayanan, "Iterative soft-input soft-output decoding of Reed–Solomon codes by adapting the parity-check matrix," *IEEE Transactions on Information Theory*, vol. 52, no. 8, pp. 3746–3756, 2006.

[20] J. Jiang and K. R. Narayanan, "Iterative soft decoding of Reed–Solomon codes," *IEEE Communications Letters*, vol. 8, no. 4, pp. 244–246, 2004.

[21] A. Neubauer, J. Freudenberger, and V. Kuhn, *Coding theory: algorithms, architectures and applications*. John Wiley & Sons, 2007.

[22] W. Ryan and S. Lin, *Channel codes: classical and modern*. Cambridge University Press, 2009.

[23] T. K. Moon, *Error correction coding*. John Wiley & Sons, Inc, 2005.

[24] O. Ur-Rehman and N. Zivic, "Soft decision iterative error and erasure decoder for Reed–Solomon codes," *IET Communications*, vol. 8, no. 16, pp. 2863–2870, 2014.

[25] V. Guruswami and A. Vardy, "Maximum-likelihood decoding of Reed–Solomon codes is NP-hard," pp. 470–478, 2005.

[26] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on information theory*, vol. 47, no. 2, pp. 498–519, 2001.

[27] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Information Theory*, vol. 27, no. 9, pp. 533–547, September 1981.

[28] D. Versfeld, H. Ferreira, and A. Helberg, "Efficient packet erasure decoding by transforming the systematic generator matrix of an RS code," in *Information Theory Workshop, 2008. ITW'08. IEEE.* IEEE, 2008, pp. 401–405.

[29] D. Versfeld, J. N. Ridley, H. C. Ferreira, and A. S. Helberg, "On Systematic Generator Matrices for Reed–Solomon Codes," *IEEE Transactions on Information Theory*, vol. 56, no. 6, pp. 2549–2550, 2010.

[30] E. Alpaydin, *Introduction to Machine Learning, Third Edition.* MIT Press, 2014.

[31] R. A. Carrasco and M. Johnston, *Non-binary error control coding for wireless communication and data storage.* John Wiley & Sons, 2008.

[32] M. Goodrich, R. Tamassia, and D. Mount, *DATA STRUCTURES AND ALOGORITHMS IN C++.* John Wiley & Sons, 2007.

[33] S. Arora and B. Barak, *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[34] J. G. Proakis, *Fundamentals of Communication Systems.* Pearson Education Ltd, 2015.

[35] J. W. Eaton. About - GNU Octave. GNU. Last Accessed: 10 November 2017. [Online]. Available: https://www.gnu.org/software/octave/about.html

[36] A. N. Laboratory. MPICH Overview. MPICH. Last Accessed: 25 June 2017. [Online]. Available: https://www.mpich.org/about/overview/

[37] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[38] Mathworks, "MATLAB randseed," last Accessed: 15 October 2017. [Online]. Available: https://www.mathworks.com/help/comm/ref/randseed.html

[39] Mathworks, "Communications System Toolbox - MATLAB," last Accessed: 15 October 2017. [Online]. Available: https://www.mathworks.com/products/communications.html

[40] Mathworks, "MATLAB qammod," last Accessed: 15 October 2017. [Online]. Available: https://www.mathworks.com/help/comm/ref/qammod.html

[41] Mathworks, "MATLAB awgn," last Accessed: 15 October 2017. [Online]. Available: https://www.mathworks.com/help/comm/ref/awgn.html

[42] L. Torvalds and J. Hamano. Git Overview. Git. Last Accessed: 12 June 2017. [Online]. Available: https://git-scm.com/about

[43] N. A. Weiss and C. A. Weiss, *Introductory Statistics*. Pearson Education Limited, 2017.