The Original View of Reed-Solomon Coding and the Welch-Berlekamp Decoding Algorithm

The Original View of Reed-Solomon Coding and the Welch-Berlekamp Decoding Algorithm

by

Sarah Edge Mann

A Dissertation Submitted to the Faculty of the

Graduate Interdisciplinary Program in Applied Mathematics

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

July 19, 2013

The University of Arizona
Graduate College

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by **Sarah Edge Mann** entitled **The Original View of Reed-Solomon Coding and the Welch-Berlekamp Decoding Algorithm** and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

_____ Date: July 19, 2013
Marek Rychlik

_____ Date: July 19, 2013
Kevin Lin

_____ Date: July 19, 2013
Don Wang

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____ Date: July 19, 2013
Dissertation Director: **Marek Rychlik**

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

Signed: _____ Sarah Edge Mann_____

Acknowledgments

I would like to thank my advisor, Dr. Marek Rychlik, for his patient guidance over the last four years, and Michael Anderson for getting me started with this project and for reviewing this dissertation. I would also like to thank Dr. Kevin Lin and Dr. Don Wang for serving on my committee and providing feedback on this dissertation. Thanks also to Dr. Michael Tabor for heading an excellent Applied Math program, and to Anne Keyl and Stacey LaBorde for keeping the program running smoothly and knowing all the answers.

Table of Contents

Table of Contents—*Continued*

TABLE OF CONTENTS—*Continued*

List of Figures

List of Tables

List of Algorithms

Abstract

Reed-Solomon codes are a class of maximum distance separable error correcting codes with known fast error correction algorithms. They have been widely used to assure data integrity for stored data on compact discs, DVDs, and in RAID storage systems, for digital communications channels such as DSL internet connections, and for deep space communications on the Voyager mission. The recent explosion of storage needs for "Big Data" has generated renewed interest in large storage systems with extended error correction capacity. Reed-Solomon codes have been suggested as one potential solution. This dissertation reviews the theory of Reed-Solomon codes from the perspective taken in Reed and Solomon's original paper on them. It then derives the Welch-Berlekamp algorithm for solving certain polynomial equations, and connects this algorithm to the problem of error correction. The discussion is mathematically rigorous, and provides a complete and consistent discussion of the error correction process. Numerous algorithms for encoding, decoding, erasure recovery, error detection, and error correction are provided and their computational cost is analyzed and discussed thus allowing this dissertation to serve as a manual for engineers interested in implementing Reed-Solomon coding.

Chapter 1
Introduction

When digital data is stored or transmitted it is often corrupted. The cause may be a faulty hard drive in the case of data storage, or atmospheric disturbances in the case of data transmissions from deep space. To address this pervasive problem, error correcting codes (ECC) were developed. In a typical error correcting code, the data is first grouped into *messages* of a prescribed length. These messages are then *encoded* to form a *codeword* and this codeword is then transmitted or stored. It my be later *decoded* to regain the original message. The codeword is typically longer then the message and thus contains some redundant information. This redundancy allows the original codeword to be fully recovered after a few errors have accumulated, thus providing the desired protection against corruption.

There are two distinct types of corruption with which we are concerned. In the *erasure* case, some data is lost from the codeword in known locations. In the *error* case, some of the data in the codeword is in error, but it is not known which data is correct and which is not. In both cases, the goal is to recover the original message from the corrupted codeword (*received word*). In the error case, the first step of this process is to first *detect* that such errors exist. The recovery process is typically far easier in the erasure case than the error case. A wide variety of error correcting codes have been developed that are capable of erasure correction, error detection, and error correction. These codes vary in the the number of erasures they can correct, the number of errors they can detect and correct, and the complexity of the encoding, decoding, erasure correcting, and error correcting algorithms. This dissertation will focus on Reed-Solomon codes.

Reed-Solomon codes were first introduced by Irving S. Reed and Gustave Solomon in their 1960 paper "Polynomial codes over certain finite fields" [16]. Reed-Solomon codes are

an optimal class of codes in that they are *maximum distance separable* (MDS) and therefore have the best possible error correction capability for their redundancy. Reed-Solomon codes are the best-known, non-trivial MDS codes. They are therefore of considerable interest and have been used in numerous applications. They provide error correction for compact discs and DVDs, deep space communication on the Voyager mission, and DSL internet connections.

In Reed and Solomon's original paper, they proposed viewing the message as defining the coefficients of a polynomial and the codeword as values of that polynomial. Methods for message encoding and decoding follow directly from this definition, as do algorithms for erasure correction and error detection. They showed that these codes are MDS, and therefore capable of powerful error correction, but did not provide an efficient algorithm for this important task.

In the years since Reed and Solomon's paper first appeared, it has become popular to view Reed-Solomon codes in a different light from what they first proposed. Whereas they viewed the codewords as being values of a polynomial, researchers later considered the codewords as being coefficients of polynomials with some prescribed roots. The view presented from Reed and Solomon's paper came to be known as the "original view", and this new perspective as the "classical view"[18]. Most texts on the subject present the classical view. The two perspectives are not equivalent, but the codes they generate share some common properties[1]. In particular, all Reed-Solomon codes are maximum distance separable.

In 1968-1969, Elwyn Berlekamp and James Massey introduced the Berlekamp-Massey algorithm which efficiently solved the error correction problem for Reed-Solomon codes. In 1986, Berlekamp and Lloyd R. Welch patented ([19]) the Berlekamp-Welch algorithm which also efficiently solved the error correction problem. These algorithms paved the way for the wide spread adoption of Reed-Solomon codes in a variety of ECC applications.

[1]Original view and classical view Reed-Solomon codes are actually dual codes of one another; the generator matrix of one is the parity check matrix of the other.

Reed-Solomon Codes

Original View

- Appears in Reed and Solomon's original paper
- Codewords are the values of a polynomial at prescribed points

Classical View

- Appears in most standard text on Reed-Solomon codes such as [1, 11, 4]
- Codewords are the coefficients of a polynomial with some prescribed roots

Most expositions of these algorithms take the classical view point of Reed-Solomon codes.

1.1 Connection to Data Storage Systems

In 1988, Patterson, Gibson, and Katz published their groundbreaking paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)"[13]. In this paper, they suggested that a reliable data storage system could be constructed from inexpensive, and not-so-reliable hard disks through the use of error correction. The essential idea was to group together some number of hard drives with one extra "check" drive. The extra drive is used to store redundant data in such a way that if any one drive in the group fails, all of its data may be recovered from the data on the remaining drives. They proposed a number of data layouts to this end which came to be known as RAID-1 through RAID-5. RAID-5 systems soon became a popular design solution, and this idea was later extended to RAID-6 systems which offered protection against up to two drives failing simultaneously. Since this paper was originally published, data storage systems have grown larger and larger to meet the ever increasing demand to store vast quantitates of digital data. The rise of "cloud computing" has contributed to this trend by centralizing many individuals' data into a few data warehouses. Protection against two drive failures is no longer sufficient,

and so researchers looked for new methods to extend greater data protection to the RAID paradigm.

In 1997 and 2003, James S. Plank published the paper "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems" and "Note: Correction to the 1997 Tutorial on Reed-Solomon Coding"[14, 15]. These papers proposed a method of extending the RAID idea beyond RAID-6 based on the original view of Reed-Solomon codes. These codes can be customized to accommodate an essentially arbitrary number of check drives and thus allow engineers to design RAID systems that protect against as many simultaneous drive failures as desired. Due to Plank's papers, data storage system designers have begun to use Reed-Solomon codes as the mathematical basis for extended RAID-like systems. Plank's paper detailed methods for encoding, decoding, and erasure recovery. However, he did not cover error correction algorithms as error correction is not the primary purpose of RAID systems. However, undetected errors can occur in any storage system and become more and more likely as these systems grow larger. It would be advantageous to use the error-correcting capability of Reed-Solomon codes to both detect and correct such errors.

1.2 Contributions

The goal of this dissertation is to provide an exposition of the original view of Reed-Solomon codes that is thorough, mathematically rigorous, and accessible to system designers. I begin in Section 1.3 with an overview of the mechanics of error correcting codes. In Chapters 2 and 3, I discuss encoding and decoding in Reed-Solomon codes from two different perspectives: the polynomial perspective and the linear algebra perspective respectively. Although these two perspectives yield identical codes, they also allow different proofs of the main results and suggest different encoding, decoding, and erasure correction algorithms. For this reason, I chose to discuss both perspectives and highlight their relative strengths. The polynomial perspective follows the spirit of Reed and Solomon's original paper; the linear algebra perspective is more closely aligned with Plank's exposition. I substantially

expand on the linear algebra perspective as compared to Plank's exposition, providing a more general theory as well as a more efficient algorithm for computing the desired generating matrix (Algorithm 3.4.2). In Chapter 4, I discuss erasure recovery, the conditions under which error detection and correction is possible, and the error detection algorithm. I provide some more general definitions and results in coding theory related to distance, but do not treat this topic exhaustively. Some of the results of Chapters 1-4 can also be found in [2], a patent application arising from work I performed related to the application of Reed-Solomon codes to RAID storage systems.

Chapters 5 - 7 comprise the heart of the dissertation in which I explore various error correcting algorithms. Error correcting algorithms attempt to determine the most likely original codeword corresponding to a given received word using properties of the code. Success in not guaranteed in general, but is assured if the error process meets some reasonable assumptions.

Error correcting algorithms for Reed-Solomon codes may be split into three phases:

- In **phase 1**, a "key equation" is derived from the received word such that the solution to the key equation gives information about the most probable codeword, the locations of the errors, and/or their magnitudes.
- In **phase 2**, a solution to this key equation is computed.
- In **phase 3**, the solution of the key equation is used to recover the original message or codeword.

Most of the literature in this area focuses on phase 2, as this was historically the most computationally expensive portion. Unfortunately, most of these resources fail to connect the key equation and its solution to the problem of decoding a Reed-Solomon code. When they do, they almost always consider Reed-Solomon codes from the classical perspective. The major contribution of this dissertation is to link all three phases of decoding together with consistent notation for Reed-Solomon codes in their original view. I offer multiple

algorithms for each phase, each of which is compatible with the original view Reed-Solomon codes.

I start in Chapter 5 with a conceptually simple algorithm that has a reasonable but sub-optimal run time. I refer to this algorithm as the modified Welch-Berlekamp algorithm as it bears some relation to the Welch-Berlekamp algorithm but is less computationally efficient. This exposition follows [9], but I provide a more detailed derivation and an exposition of relevant techniques in linear algebra and detailed pseudo code for the algorithm.

In Chapter 6, I derive a key equation for decoding that lends itself well to known algorithms to compute its solution. I then provide a proof that the solution to this key equation is unique, and that this unique solution corresponds to the desired solution to the decoding problem. This proof is an essential bridge between available derivations of this key equation for original-view Reed-Solomon codes, and methods for solving this type of equation available elsewhere. Although this proof shares some common ideas with available proofs of uniqueness for the classical-view case, it is original and, to my knowledge, not available elsewhere.

I continue in Chapter 7 by deriving the Welch-Berlekamp algorithm for solving the key equation. This derivation follows [8] and [12] which justify the algorithm using Gröbner basis techniques. My exposition expands upon that which is offered in those papers, filling in the details of many proofs that were previously left to the reader or simply ignored. I also provide a discussion of two different methods for phase 3, as well as pseudo code for all proposed algorithms and a detailed analysis of their computational complexity. Finally, I note that the key equation may be generalized, and show that the Welch-Berlekamp algorithm may be readily adapted to solve any instance of this general equation.

I provide three appendices with supplemental materials. In Appendix A, I discuss various algorithms for operations with polynomials and linear algebra on which the encoding, decoding, erasure correcting, error detecting, and error correcting algorithms depend. These are all well known algorithms, and are provided here for clarity and to support

detailed analyses of the computational complexity of the other algorithms. Appendix B provides a brief introduction to modules and Gröbner bases. This discussion is certainly not exhaustive, but provides the necessary definitions and results to support the derivation of the Welch-Berlekamp algorithm in Chapter 7. Appendix C provides a list of mathematical notation used in this text, as well as a table of symbols used and there meaning.

1.3 Overview of Error Correcting Codes

In Reed-Solomon error correcting codes (ECC), data originates as a string of bits which are first grouped into *characters* consisting of l bits of data; $l = 8$ is typical and corresponds to byte-sized characters. These characters are then grouped into k character *messages*. By adding an additional $r = n - k$ carefully chosen characters to a message, a group of n characters (the *codeword*) is created; this process is referred to as *encoding*. The codeword is somewhat redundant, containing more characters than the message contains. The set of all possible codewords constitutes the *code*; this set is dependent on the manner in which the redundant characters are chosen.

l	number of bits in a character
k	number of characters in a message
n	number of characters in a codeword
r	$n - k$, number of characters in a codeword which are redundant

TABLE 1.1. A Brief Table of Symbols

The codeword is then transmitted or stored for later use and in the process is subjected to random errors. We refer to this corrupted codeword as the *received word*. Due to the redundancy, these errors may be detected and the original codeword may be recovered from the received word if the number of errors is not too great. This process is called *error detection and correction*. Unfortunately, upon detecting errors in a received word it is impossible to know exactly what error process has occurred. Perhaps only a few errors occurred suggesting that the received word should be corrected to some codeword c. On

the other hand, many errors may have occurred implying that the original codeword was actually c'. Since there is no way of knowing which event occurred, the best we may do is make some assumptions about the error process and pick a codeword that is most likely under those assumptions for the given received word. In this dissertation, we assume that every single character error is equally likely, and that it is more likely that fewer errors have occurred than many errors. We therefore choose the codeword that corresponds to the least number of accumulated errors in the received word. We may adjust the values of k and n such that the probability of incorrectly correcting a received word is within an acceptable bound. Figure 1.1 depicts the lifecycle of a message under ECC protection.

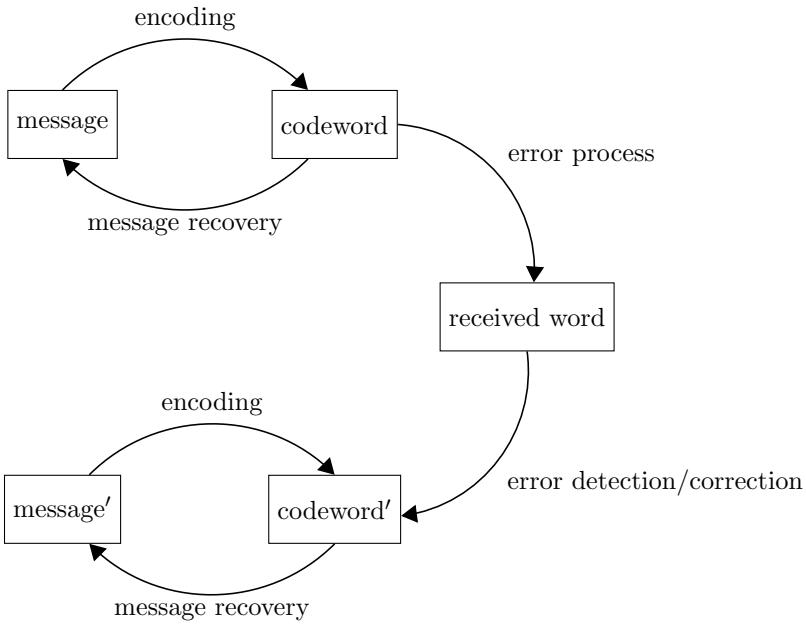

FIGURE 1.1. A depiction of the processes involved in ECC. If error correction is succesfull, codeword = codeword' and message = message', but this is not guaranteed.

The essence of the problem of ECC is how to choose the redundant characters in the codeword such that a codeword may be correctly recovered from a received word in the presence of a large number of errors in a computationally efficient manner. Reed-Solomon

codes are one such solution to this problem.

In what follows, a general background in coding theory, particularly linear block codes, is assumed, as is some familiarity with arithmetic over finite fields. These subjects are well covered in the standard texts such as [11, 1]. All arithmetic is computed over the field $\mathbb{F}$. For concreteness and relevance to applications, $\mathbb{F}$ will be taken to be the finite field $GF(2^l)$ in all examples. It might be more intuitive to think of $\mathbb{F}$ as $\mathbb{R}$ at first, to understand some of the proofs.

1.4 Characters, Messages, and Finite Fields

Before data can be protected, it must be preprocessed into a format appropriate to ECC. For Reed-Solomon codes, this means that existing data must be first broken into l bit characters. These l bit characters may be viewed as integers between 0 and $2^l - 1$. Thus our *alphabet*, the set of all characters, is the set of integers between 0 and $2^l - 1$ inclusive.

These characters may be viewed as elements of the finite Galois field GF(2^l) , and we may perform arithmetic on our characters using the field operations. Messages may be viewed as vectors in GF$(2^l)^k$ and we may perform linear algebra on these vectors. This is the view that we will take for the rest of this paper: characters are elements in GF(2^l). See [11, Chapter 5] and [1, Chapter 11] for details on Galois field arithmetic. In all examples for $l = 8$, we use the representation of GF(256) generated by the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$.

Example 1.4.1. For $l = 8$ and $k = 5$, a bit stream may be represented as follows:

Bit Stream: 11101001110100110000000000000111000100101011010111
11110100001111000100100101110101001110111001000000
10011100011111000011 . . .

Split into characters: 11101001 11010011 00000000 00000111 00010010 10110101
11111101 00001111 00010010 01011101 01001110 11100100
00001001 11000111 11000011 . . .

Integer representation: 233 211 0 7 18 181 126 15 18 93 78 228 9 199 195 . . .

Split into messages: 233 211 0 7 18 181 126 15 18 93 78 228 9 199 195 . . .

Chapter 2
Message Encoding: The Polynomial Perspective

We now examine how messages are encoded in original-view Reed-Solomon codes, how the code, the set of all codewords, is defined, and how the message may be recovered from the codeword. We start with the polynomial perspective which follows the spirit of Reed and Solomon's paper [16]. We introduce algorithms for encoding and decoding here. These algorithms rely on some of the algorithms for computation with polynomials discussed in Appendix A.3. The reader is encouraged to review that appendix before continuing or as needed.

2.1 Message Encoding

Per the original view of Reed-Solomon codes [16, 18], we may view the message characters as the coefficients of a polynomial. Thus we interpret the message $\vec{m} = (m_0, m_1, \ldots, m_{k-1})^T$ as a polynomial $m(x) = \sum_{i=0}^{k-1} m_i x^i$. Let $\vec{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})^T$ be n distinct characters in GF(2^l). To encode the message, we simply evaluate the polynomial m at the points $\vec{\alpha}$ to obtain a codeword $\vec{c} = (c_0, c_1, \ldots, c_{n-1})^T$:

$$\vec{c} = m(\vec{\alpha}), \quad \iff \quad c_i = m(\alpha_i)\ 0 \leq i < n.$$

Example 2.1.1. Consider the first message from Example 1.4.1, $\vec{m} = (233, 211, 0, 7, 18)^T$, with $l = 8$ and $k = 5$. We have the message polynomial

$$m(x) = 233 + 211x + 7x^3 + 18x^4$$

To create a length $n = 8$ codeword we chose $\vec{\alpha} = (0, 1, 2, 3, 4, 5, 6, 7)^T$. Thus the codeword

$\vec{c}$ is given by

$$c_0 = m(0) = 233$$
$$c_1 = m(1) = 47$$
$$c_2 = m(2) = 87$$
$$c_3 = m(3) = 131$$
$$c_4 = m(4) = 168$$
$$c_5 = m(5) = 2$$
$$c_6 = m(6) = 134$$
$$c_7 = m(7) = 62$$

i.e. $\vec{c} = (233, 47, 87, 131, 168, 2, 134, 62)^T$.

This encoding scheme yields some immediate restriction on the code parameters. First, in order for the codeword to contain redundancy, we must have $k < n$. Second, since the values in $\vec{\alpha}$ must be distinct and $|\vec{\alpha}| = |\vec{c}| = n$[1], we must have $n \leq |\mathrm{GF}(2^l)| = 2^l$. In summary:

$$k < n \leq 2^l.$$

Message encoding may be achieved computationally by calling PolyEval$(\vec{m}, \vec{\alpha})$, defined in Algorithm A.3.1 in Appendix A. Since $\vec{m}$ has degree $k-1$, it costs $2(k-1)n$ operations to encode a message in this manner.

2.2 Recovering the Message from the Codeword

It is imperative that given an uncorrupted codeword, we may recover the intended message exactly. Our codewords are n points on a degree $k-1$ polynomial. Since any k points

[1] $|\vec{x}|$ denotes the length of the vector $\vec{x}$. See Appendix C for a complete list of mathematical notation used in this text.

uniquely defines a degree $k-1$ polynomial, to recover the message we may simply choose k of the codeword characters along with the corresponding values from $\vec{\alpha}$ and perform Lagrange interpolation to recover $m(x)$ and thus the message.

In particular, if $\mathcal{I}$ is the set of indices of the k points chosen for Lagrange interpolation, then we seek the polynomial $m(x)$ such that $m(\vec{\alpha}_{\mathcal{I}}) = \vec{c}_{\mathcal{I}}$. $m(x)$ is given explicitly by via the Lagrange interpolation formula

$$m(x) = \sum_{i \in \mathcal{I}} c_i \prod_{\substack{j \in \mathcal{I} \\ j \neq i}} \frac{x - \alpha_j}{\alpha_i - \alpha_j}.$$

Algorithm 2.2.1 implements message recovery via Lagrange interpolation and requires at most $6k^2 - k$ operations.

Algorithm 2.2.1 Message recovery via Lagrange interpolation

Dependencies: LagrangeInterpolate defined in Algorithm A.3.8.

```
1: procedure MessageRecoverLI(c⃗, α⃗, ℐ)      ⋆Recover the message m⃗ from c⃗ at the
                                               ⋆indices ℐ via Lagrange interpolation
2:     m⃗ ← LagrangeInterpolate(α⃗_ℐ, c⃗_ℐ)
3:     return m⃗
4: end procedure
```

Computational complexity: No more than $6k^2 - k$ operations required.

Example 2.2.1. Continuing with Example 2.1.1, we may chose $\mathcal{I} = \{0, 1, 2, 3, 4\}$. Then we must interpolate the points $(0, 233)$, $(1, 47)$, $(2, 87)$, $(3, 131)$, and $(4, 168)$. This means that

$$\begin{aligned} m(x) &= 233\left(\frac{x-1}{0-1}\right)\left(\frac{x-2}{0-2}\right)\left(\frac{x-3}{0-3}\right)\left(\frac{x-4}{0-4}\right) + 47\left(\frac{x-0}{1-1}\right)\left(\frac{x-2}{1-2}\right)\left(\frac{x-3}{1-3}\right)\left(\frac{x-4}{1-4}\right) \\ &\quad + 87\left(\frac{x-0}{2-1}\right)\left(\frac{x-1}{2-1}\right)\left(\frac{x-3}{2-3}\right)\left(\frac{x-4}{2-4}\right) + 131\left(\frac{x-0}{3-1}\right)\left(\frac{x-1}{3-1}\right)\left(\frac{x-2}{3-2}\right)\left(\frac{x-4}{3-4}\right) \\ &\quad + 168\left(\frac{x-0}{4-1}\right)\left(\frac{x-1}{4-1}\right)\left(\frac{x-2}{4-2}\right)\left(\frac{x-3}{4-3}\right) \\ &= 18x^4 + 7x^3 + 211x + 233 \end{aligned}$$

and so the message was $\vec{m} = (233, 211, 0, 7, 18)^T$.

2.3 Systematic Encoding

It would be nice if the message appeared as a subset of the codeword, *i.e.*

$$\vec{c} = (m_0, \ldots m_{k-1}, c_k, \ldots, c_{n-1})^T.$$

In this case, recovering a message from the codeword is trivial; one simply reads the message from the first k characters of the codeword. This type of encoding is termed *systematic*, and may be achieved by altering the encoding algorithm.

Previously we had encoded using the rule $\vec{c} = m(\vec{\alpha})$. We will now introduce a new polynomial $\mu(x)$ that will generate the encoding systematically via $\vec{c} = \mu(\vec{\alpha})$. $\mu(x)$ is the unique degree $k-1$ or less polynomial such that $\mu(\alpha_i) = m_i$ for $0 \leq i < k$. We can find $\mu(x)$ explicitly using Lagrange interpolation :

$$\mu(x) = \sum_{i=0}^{k-1} m_i \prod_{0 \leq j < k, j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j}$$

Given this definition, it is easy to see that $\vec{c}$ is indeed systematic.

Systematic message encoding is implemented in Algorithm 2.3.1 and costs $2nk + 4k^2 - 2n + k$ operations. Recovering the message from the codeword is trivial when using a systematic encoding as it is simply read from the first k characters.

Example 2.3.1. Consider the first message from Example 1.4.1, $\vec{m} = (233, 211, 0, 7, 18)^T$, with $l = 8$ and $k = 5$. We have the message polynomial

$$\mu(x) = 65x^4 + 112x^3 + 10x^2 + x + 233$$

and systematic codeword $\vec{c} = (233, 211, 0, 7, 18, 166, 14, 135)^T$.

[2]Computational complexity of Algorithm 2.2.1: The call to LagrangeInterpolate costs $6k^2 - k$ operations. The call to PolyEval costs $2r(k-1)$ operations. This yields a total of $6k^2 - k + 2r(k-1) = 6k^2 + k(2r-1) - 2r$ operations.

Algorithm 2.3.1 Systematic message encoding: polynomial perspective

Dependencies: LagrangeInterpolate defined in Algorithm A.3.8, PolyEval defined in Algorithm A.3.1.

```
1: procedure MessageEncodeSys(m⃗, α⃗)                    ⋆Encode m⃗ systematically
2:     μ⃗ ←LagrangeInterpolate(α⃗_{0:k−1}, m⃗)            ⋆μ(α_i) = m_i for 0 ≤ i < k
3:     Initialize c⃗ as a length n vector
4:     c⃗_{0:k−1} ← m⃗
5:     c⃗_{k:n−1} ← PolyEval(μ⃗, α⃗_{k:n−1})              ⋆c_i = μ(α_i)
6:     return c⃗
7: end procedure
```

Computational complexity: No more than $6k^2 + k(2r-1) - 2r$ operations required[2].

2.4 Codes

Formally, a *code* C is the set of all possible codewords, *i.e.* the image of the set of all possible messages under the chosen encoding. Reed-Solomon codes are a class of codes that share some properties; in particular they are maximum distance separable[3]. The choice of k, n, $\vec{\alpha}$ and the encoding method (non-systematic as in Section 2.1 vs. systematic as in Section 2.3) define a particular Reed-Solomon code.

2.5 Historical Notes

The non-systematic code and encoding procedure presented in Section 2.1 is exactly the code and encoding procedure proposed by Reed and Solomon in [16] in 1960 with one deviation. Reed and Solomon required $n = 2^l$ and thus $\vec{\alpha}$ was simply an ordered list of all the elements in GF(2^l). This view on Reed-Solomon codes is sometimes called the *original view*, [18].

Modern textbooks on coding theory such as [1] and [11] view Reed-Solomon codes from a different perspective sometimes called the *classical view*. In this framework, we chose $\vec{\beta}$,

[3] See Chapter 4.3.1 and Definition 4.3.12 for more on maximum distance separable codes.

a length $n-k$ vector of distinct, non-zero elements in $GF(2^l)$. We construct the generator polynomial as

$$g(x) = \prod_{i=0}^{n-k-1} (x - \beta_i)$$

and, given the message $\vec{m}$, we construct the message polynomial as $m(x) = \sum_{i=0}^{k-1} m_i x^i$. We encode by $c(x) = g(x)m(x)$ with $\vec{c}$ defined by $c(x) = \sum_{i=0}^{n} c_i x^i$. Thus $\vec{c}$ is a codeword if and only if $c(\beta_i) = 0$ for $0 \leq i < n-k$. While this produces a different code than the original view, both codes share similar properties such as being MDS. In fact, they are actually dual codes of each other.

Chapter 3
Message Encoding: The Linear Algebra Perspective

The encoding algorithms presented in Chapter 2 are actually linear, and we can view encoding and message recovery as matrix operations. In this section we will develop the necessary framework for this perspective. This discussion follows the spirit of Plank's exposition on Reed-Solomon codes for extended RAID systems in [14, 15]. Many of the algorithms in this chapter rely on algorithms for linear algebra presented in Appendix A.4. The reader is encouraged to review that appendix before continuing, or as needed.

3.1 The Generalized Vandermonde Matrix

We start with some theory regarding Vandermonde matrices that will be crucial to defining the encoding and decoding algorithms from a linear algebra perspective.s

Definition 3.1.1. Let $\vec{\alpha}$ be a vector in $\mathbb{F}^a$, and α_i be the ith element of $\vec{\alpha}$. The $a \times b$ *Vandermonde Matrix* generated by $\vec{\alpha}$ is defined[1] to be

$$V^{a,b}(\vec{\alpha}) = \begin{bmatrix} \alpha_0^0 & \alpha_0^1 & \alpha_0^2 & \dots & \alpha_0^{b-1} \\ \alpha_1^0 & \alpha_1^1 & \alpha_1^2 & \dots & \alpha_1^{b-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{a-1}^0 & \alpha_{a-1}^1 & \alpha_{a-1}^2 & \dots & \alpha_{a-1}^{b-1} \end{bmatrix} = \left[\vec{\alpha}^0 | \vec{\alpha}^1 | \vec{\alpha}^2 | \dots | \vec{\alpha}^{b-1}\right].$$

Algorithm 3.1.1 generates the Vandermonde matrix for a given $\vec{\alpha}, a$, and b.

Definition 3.1.2. Let $\vec{\alpha}$ be a vector in $\mathbb{F}^a$, and let $\{f_i\}_{i=0}^{b-1} \subset \mathbb{F}[x]$ be a set of b linearly independent polynomials with $\deg f_i < b$ for all i. Define the $a \times b$ *generalized Vandermonde Matrix* by

$$G(\{f_i\}, \vec{\alpha}) = \left[f_0(\vec{\alpha}) | f_1(\vec{\alpha}) | \dots | f_{b-1}(\vec{\alpha})\right].$$

[1] Some authors define the Vandermonde matrix as the transpose of the matrix given here.

Algorithm 3.1.1 Vandermonde matrix constriction

```
1: procedure VAND(a, b, α⃗)          ⋆Construct the Vandermonde matrix V^{a,b}(α⃗)
2:     create a matrix V of size a × b
3:     for i ← 0 . . . a − 1 do
4:         for j ← 0 . . . b − 1 do
5:             V_{i,j} ← α_i^j
6:         end for
7:     end for
8:     return V
9: end procedure
```

Computational complexity: ab operations required.

Remark 3.1.3. Notice that the Vandermonde matrix is the special case of the generalized Vandermonde matrix where $f_i(x) = x^i$.

Theorem 3.1.4. *The square generalized Vandermonde matrix is invertible if and only if α_i are distinct.*

Proof. Consider a $b \times b$ generalized Vandermonde matrix $G(\{f_i\}, \vec{\alpha})$. Trivially, if α_i are not distinct then $G(\{f_i\}, \vec{\alpha})$ will contain repeated rows, and will not be invertible.

Assume α_i are distinct, and, in search of contradiction, assume that $G(\{f_i\}, \vec{\alpha})$ is not invertible. Then there exists $\vec{c} \in \mathbb{F}^b$ such that $\vec{c} \neq \vec{0}$ and $G(\{f_i\}, \vec{\alpha})\vec{c} = \vec{0}$, *i.e.*

$$\sum_{i=0}^{b-1} c_i f_i(\alpha_j) = 0$$

for any $0 \leq j \leq b-1$. $\sum_{i=0}^{b-1} c_i f_i(x)$ is a degree at most $b-1$ polynomial that vanishes at b points, and is thus uniformly 0. This contradicts the assumed linear independence of $\{f_i\}$, so $G(\{f_i\}, \vec{\alpha})$ must be invertible. □

Corollary 3.1.5. *The square Vandermonde matrix is invertible if and only if α_i are distinct.*

Corollary 3.1.6. *Any square submatrix of a generalized Vandermonde matrix is a square generalized Vandermonde matrix and is thus invertible.*

Theorem 3.1.7. *Let $\vec{\alpha}$ be a vector in $\mathbb{F}^b$ with α_i distinct, and $\{f_i\}_{i=0}^{b-1}$ be a set of degree at most $b-1$ polynomials over $\mathbb{F}$. If the matrix $A = [f_0(\vec{\alpha})|f_1(\vec{\alpha})|\ldots|f_{b-1}(\vec{\alpha})]$ is invertible, then $\{f_i\}_{i=0}^{b-1}$ are linearly independent and $G(\{f_i\}, \vec{\gamma})$ is a generalized Vandermonde matrix for any $\gamma \in \mathbb{F}^a$.*

Proof. In search of contradiction, assume that A is invertible and $\{f_i\}$ are not linearly independent. Then there exists $\vec{c} \neq \vec{0} \in \mathbb{F}^b$ such that

$$\sum_{i=0}^{b-1} c_i f_i(x) = 0, \qquad \forall x,$$

and, in particular,

$$\sum_{i=0}^{b-1} c_i f_i(\alpha_j) = 0, \qquad 0 \leq j \leq b-1.$$

Thus the columns of A are not linearly independent and A is not invertible. This is a contradiction, so $\{f_i\}$ must be linearly independent. □

Corollary 3.1.8. *For $a > b$, let $\vec{\alpha}$ be a vector in $\mathbb{F}^a$ with α_i distinct, and $\{f_i\}_{i=0}^{b-1}$ be a set of degree $b-1$ polynomials over $\mathbb{F}$. If*

$$A = [f_0(\vec{\alpha})|f_1(\vec{\alpha})|\ldots|f_{b-1}(\vec{\alpha})] = \begin{bmatrix} I_b \\ \star \end{bmatrix}$$

where I_b is the $b \times b$ identity matrix, and $\star$ is some $(a-b) \times b$ matrix, then A is a generalized Vandermonde matrix.

Proof. Notice that I_b is an invertible matrix generated by the first b elements of $\vec{\alpha}$, and thus, by Theorem 3.1.7, $\{f_i\}$ are linearly independent and $A = G(\{f_i\}, \vec{\alpha})$ is a generalized Vandermonde matrix. □

3.2 Encoding and Message Recovery as Linear Operations

Let $V^{n,k}(\vec{\alpha})$ be the Vandermonde matrix generated by $\vec{\alpha}$ with α_i distinct. Then the non-systematic encoding of Section 2.1 is equivalent to

$$\vec{c} = V^{n,k}(\vec{\alpha})\vec{m}.$$

Recovering the message from the codeword is equivalent to solving $\vec{c} = V^{n,k}(\vec{\alpha})\vec{m}$ for $\vec{m}$. Since by Corollary 3.1.6 any square submatrix of the Vandermonde matrix is invertible, this equation is solvable. We simply select a set of k indices $\mathcal{I}$ to use for recovery, and consider only the rows in this equation corresponding to those indices: $\vec{c}_{\mathcal{I}} = V^{n,k}_{\mathcal{I},:}(\vec{\alpha})\vec{m}$. We then solve this system using standard techniques from linear algebra. Algorithm 3.2.1 provides one such method for recovery the message.

Algorithm 3.2.1 Message recovery via linear algebra

Dependencies: LU defined in Algorithm A.4.1, SolveLU defined in Algorithm A.4.2.

```
1: procedure MessageRecoverLA(c⃗, G, I)      ⋆Recover the message m⃗ from c⃗ at the
                                                ⋆indices I by solving G_{I,:}m⃗ = c⃗_I
2:     L, U ← LU(G_{I,:})                    ⋆Compute the LU factorization of G_I
3:     m⃗ ← SolveLU(L, U, c⃗_I)
4:     return m⃗
5: end procedure
```

Computational complexity: $\frac{2}{3}k^3 + \frac{7}{2}k^2 + \frac{11}{6}k - 6$ operations required.

Example 3.2.1. (Analogue of Examples 2.1.1 and 2.2.1) Consider the first message from Example 1.4.1, $\vec{m} = (233, 211, 0, 7, 18)^T$, with $l = 8$, $k = 5$, $n = 8$ and $\vec{\alpha} = (0, 1, 2, 3, 4, 5, 6, 7)^T$. Then

$$V^{8,5}(\vec{\alpha}) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 5 & 15 & 17 \\ 1 & 4 & 16 & 64 & 29 \\ 1 & 5 & 17 & 85 & 28 \\ 1 & 6 & 20 & 120 & 13 \\ 1 & 7 & 21 & 107 & 12 \end{bmatrix}$$

and $\vec{c} = V^{8,5}(\vec{\alpha})\vec{m} = (233, 47, 87, 131, 168, 2, 134, 62)^T$.

Given the codeword $\vec{c} = (233, 47, 87, 131, 168, 2, 134, 62)^T$, to recover the message we

consider the following linear system:

$$\begin{bmatrix} 233 \\ 47 \\ 87 \\ 131 \\ 168 \\ 2 \\ 134 \\ 62 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 5 & 15 & 17 \\ 1 & 4 & 16 & 64 & 29 \\ 1 & 5 & 17 & 85 & 28 \\ 1 & 6 & 20 & 120 & 13 \\ 1 & 7 & 21 & 107 & 12 \end{bmatrix} \vec{m}$$

This system is overdetermined but consistent, so we discard the last three rows reducing the problems to solving

$$\begin{bmatrix} 233 \\ 47 \\ 87 \\ 131 \\ 168 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 5 & 15 & 17 \\ 1 & 4 & 16 & 64 & 29 \end{bmatrix} \vec{m}.$$

and find that $\vec{m} = (233, 211, 0, 7, 18)^T$.

3.3 Codes as Linear Spaces

For G full rank and an encoding scheme of the form $\vec{c} = G\vec{m}$, G is called the *generator* of the code. Equivalently, G is said to *generate* the code. The message space, the vector space of all possible messages, is $\mathrm{GF}(2^l)^k$ and has dimension k. The code C is a subspace of $\mathrm{GF}(2^l)^n$ and also has dimension k. The generator G defines the code; $\vec{c}$ is a codeword if and only if it is in the image of G. The columns of G form a basis for the code.

Exercise 3.3.1. Show that C is a vector space, *i.e.* that $c_1, c_2 \in C$ implies $c_1 + c_2 \in C$ and that for any $\gamma \in \mathrm{GF}(2^l)$ and $c \in C$, $\gamma c \in C$.

Exercise 3.3.2. Show that the columns of G form a basis for the code C generated by G.

A matrix $F \in \mathrm{GF}(2^l)^{n \times k}$ such that any $k \times k$ submatrix of F is invertible is called an *information dispersal matrix* (IDM),[14, 15] . The generalized Vandermonde matrix is an

example of an IDM since any square submatrix of generalized Vandermonde matrix is a square generalized Vandermonde matrix and thus invertible. Since IDMs are necessarily full rank, any IDM may be used to generate a code. That any $k \times k$ submatrix of an IDM is invertible ensures easy recovery of the message from a codeword per the technique of Section 3.2. If the top k rows of F form a $k \times k$ identity matrix, then F is said to be a systematic matrix and the code defined by F will be systematic.

3.4 Systematic Encoding as a Linear Operation

The systematic encoding of Section 2.3 is also linear and its generator may be constructed as follows, as proposed by Plank in [15] . Start with $V^{n,k}(\vec{\alpha})$ with α_i distinct, then perform elementary column operations on this matrix such that the top k rows are transformed into the identity matrix[2] . This is accomplished via Gaussian elimination on the columns of $V^{n,k}(\vec{\alpha})$. The result is a systematic matrix $F^{n,k}(\vec{\alpha})$, and we encode the message via $\vec{c} = F^{n,k}(\vec{\alpha})\vec{m}$. Algorithm 3.4.1 provides an explicit construction for $F^{n,k}(\vec{\alpha})$.

Theorem 3.4.1. *For $\vec{\alpha}$ with α_i distinct, the matrix $F^{n,k}(\vec{\alpha})$ described above is a systematic information dispersal matrix.*

Proof. We obtain $F^{n,k}(\vec{\alpha})$ by transforming $V^{n,k}(\vec{\alpha}) = \left[\vec{\alpha}^0|\vec{\alpha}^1|\vec{\alpha}^2|\ldots|\vec{\alpha}^{k-1}\right]$ by performing elementary column operations. Thus each column of $F^{n,k}(\vec{\alpha})$ is a linear combination of the columns of $V^{n,k}(\vec{\alpha})$, *i.e.* a polynomial of degree $k-1$ over GF(2^l). So, $F^{n,k}(\vec{\alpha}) = [f_0(\vec{\alpha})|f_1(\vec{\alpha})|\ldots|f_{k-1}(\vec{\alpha})]$, where f_i are polynomials of degree $k-1$. Moreover, the top k rows of $F^{n,k}(\alpha)$ form the identity matrix. By Corollary 3.1.8, $F^{n,k}(\alpha)$ is a generalized Vandermonde matrix and thus an information dispersal matrix. By construction, $F^{n,k}(\vec{\alpha})$ is systematic. □

Corollary 3.4.2. *The message encoding generated by $F^{n,k}(\vec{\alpha})$ is equivalent to the systematic polynomial encoding discussed in Section 2.3.*

[2] In [15] , Plank used $\alpha_i = i$. This restriction is unnecessary; it suffices that $\vec{\alpha}$ has no repeat entries.

Proof. We have $F^{n,k}(\vec{\alpha}) = [f_0(\vec{\alpha})|f_1(\vec{\alpha})|\dots|f_{k-1}(\vec{\alpha})]$. Define $\mu(x) = \sum_{i=0}^{k-1} m_i f_i(x)$. Notice that $\mu(x)$ has degree at most $k-1$ and $\mu(\alpha_i) = m_i$ for $0 \leq i < k$ by construction of $F^{n,k}(\vec{\alpha})$. Thus $\mu(x)$ is exactly the message polynomial from Section 2.3. Then $\vec{c} = F^{n,k}(\vec{\alpha})\vec{m} = \mu(\vec{\alpha})$, and the encoding is exactly the encoding of Section 2.3. □

Exercise 3.4.3. In a typical implementation of Gaussian elimination, partial pivoting is used to ensure that there is no division by zero and to promote numerical stability. Algorithm 3.4.1, however, relies on Gaussian elimination but does not use partial pivoting. In all applications, arithmetic is performed over a finite field and therefore the only reason pivoting would ever be needed would be to prevent division by zero. This would only be be necessary if in the ith step of Gaussian elimination the ith diagonal entry were zero. Show that this will never be case in Algorithm 3.4.1. Hint: use the fact that Gaussian elimination is being performed on a Vandermonde matrix, and Corollary 3.1.6.

Example 3.4.4. For $l = 8$, $k = 5$, $n = 8$ and $\vec{\alpha} = (0, 1, 2, 3, 4, 5, 6, 7)^T$, the systematic

[3]Operation count for Algorithm 3.4.1: The call to VAND in line 2 costs nk operations. The for loop in the remaining lines costs at most

$$\begin{aligned}
\sum_{c=0}^{k-1}\left[\left(\sum_{r=c}^{n-1} 1\right) + \left(\sum_{i=0,i\neq c}^{k-1}\sum_{r=c}^{n-1} 2\right)\right] &= \sum_{c=0}^{k-1}\left[n-c+2\left(\sum_{i=0,i\neq c}^{k-1} n-c\right)\right] \\
&= nk + \sum_{c=0}^{k-1}[-c+2(k-1)(n-c)] \\
&= nk + \sum_{c=0}^{k-1}[2nk-2ck-2n+c] \\
&= nk + 2nk^2 - 2nk + (1-2k)\sum_{c=0}^{k-1} c \\
&= 2nk^2 - nk + (1-2k)\frac{1}{2}k(k-1) \\
&= 2nk^2 - nk + \frac{3}{2}k^2 - \frac{1}{2}k - k^3
\end{aligned}$$

operations. This yields a total or $2nk^2 - k^3 + \frac{3}{2}k^2 - \frac{1}{2}k$ operations.

Algorithm 3.4.1 Constructing a systematic IDM via Gaussian elimination

Dependencies: VAND defined in Algorithm 3.1.1.

1: **procedure** SYSIDMG($n, k, \vec{\alpha}$) ⋆*Construct the systematic IDM* $F^{n,k}(\vec{\alpha})$
2: $F \leftarrow$ VAND($n, k, \vec{\alpha}$) ⋆*Initialize* F *as the Vandermonde matrix generated by* $\vec{\alpha}$
3: **for** $c \leftarrow 0 \ldots k-1$ **do** ⋆*Perform Gaussian elimination on the columns* $V^{n,k}(\vec{\alpha})$ ⋆*until the first* k *rows of the matrix are the identity.*
4: **if** $F_{c,c} \neq 1$ **then** ⋆*Rescale the column so the entry on the diagonal is 1*
5: **for** $r \leftarrow c \ldots n-1$ **do**
6: $F_{r,c} \leftarrow \frac{F_{r,c}}{F_{c,c}}$
7: **end for**
8: **end if**
9: **for** $i \leftarrow 0 \ldots k-1, i \neq c$ **do** ⋆*Zero out the entry in row* c *of* ⋆*each column except for column* c
10: **for** $r \leftarrow c \ldots n-1$ **do**
11: $F_{r,i} \leftarrow F_{r,i} - F_{r,c}F_{c,i}$
12: **end for**
13: **end for**
14: **end for**
15: **return** F
16: **end procedure**

Computational complexity: No more than $2k^2n - k^3 + \frac{3}{2}k^2 - \frac{1}{2}k$ operations required[3].

IDM obtained from $V^{8,5}(\vec{\alpha})$ shown in Example 3.2.1 is

$$F^{8,5}(\vec{\alpha}) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 7 & 7 & 6 & 6 & 1 \\ 9 & 8 & 9 & 8 & 1 \\ 15 & 14 & 14 & 15 & 1 \end{bmatrix}. \tag{3.1}$$

For $\vec{m} = (233, 211, 0, 7, 18)^T$ we obtain $\vec{c} = F^{8,5}(\vec{\alpha}) = (233, 211, 0, 7, 18, 166, 14, 135)^T$ as in Example 2.3.1.

As noted in the proof of Theorem 3.4.1, the columns of $F^{n,k}(\vec{\alpha})$ are generated by the degree $k-1$ polynomials $\{f_i\}_{i=0}^{k-1}$. Using the constraint $f_i(\alpha_j) = \delta_{ij}$, we can write an explicit formula for these functions using Lagrange polynomials, as shown in Theorem 3.4.5. Algorithm 3.4.2 provides a new way to construct $F^{n,k}(\vec{\alpha})$ based on this perspective.

Theorem 3.4.5. *The information dispersal matrix $F^{n,k}(\vec{\alpha})$ has the form*

$$F^{n,k}(\vec{\alpha}) = [f_0(\vec{\alpha})|f_1(\vec{\alpha})|\ldots|f_{k-1}(\vec{\alpha})] \tag{3.2}$$

where

$$f_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{k-1} \frac{x-\alpha_j}{\alpha_i - \alpha_j}. \tag{3.3}$$

Proof. From the proof of Theorem 3.4.1, $F^{n,k}(\vec{\alpha}) = [f_0(\vec{\alpha})|f_1(\vec{\alpha})|\ldots|f_{k-1}(\vec{\alpha})]$, where $\{f_i\}_{i=0}^{k-1}$ are degree $k-1$ polynomials. The constraint $f_i(\alpha_j) = \delta_{ij}$ for $0 \leq i, j < k$ defines these polynomials uniquely. Using Lagrange polynomials yields the form of $f_i(x)$ given. □

Algorithm 3.4.2 Constructing a systematic IDM via Lagrange polynomials

1: **procedure** SysIDML$(n, k, \vec{\alpha})$ ⋆*Construct the systematic IDM* $F^{n,k}(\vec{\alpha})$
2: Initalize F as an $n \times k$ matrix of zeros
3: **for** $c \leftarrow 0 \ldots k-1$ **do** ⋆*Loop across the columns*
4: $a \leftarrow 1$ ⋆*Construct* $a = \left(\prod_{\substack{j=0 \\ j \neq c}}^{k-1} (\alpha_c - \alpha_j) \right)^{-1}$
5: **for** $j = 0 \ldots k-1, j \neq c$ **do**
6: $a \leftarrow a/(\alpha_c - \alpha_j)$
7: **end for**
8: $F_{c,c} \leftarrow 1$ ⋆*Set* $F_{c,c} = 1$ *and leave the rest of the first* k *rows of column* c *as zeros to create the identity block*
9: **for** $r \leftarrow k \ldots n-1$ **do** ⋆*Construct* $F_{r,c} = f_c(\alpha_r)$ *for the last* $n-k$ *rows*
10: $F_{r,c} \leftarrow a$
11: **for** $j = 0 \ldots k-1, j \neq c$ **do**
12: $F_{r,c} \leftarrow F_{r,c} \cdot (\alpha_r - \alpha_j)$
13: **end for**
14: **end for**
15: **end for**
16: **return** $\vec{F}$
17: **end procedure**

Computational complexity: $2k(k-1)(r+1)$ operations required[4].

Remark 3.4.6. Algorithms 3.4.1 and 3.4.2 both produce the same matrix $F^{n,k}(\vec{\alpha})$ using different methods. Algorithm 3.4.1 costs $2k^2n - k^3 + \frac{3}{2}k^2 - \frac{1}{2}k$ operations, where as Algorithm 3.4.2 costs $2k(k-1)(r+1) = 2(k^2n - k^3 + 2k^2 - kn - k)$ operations and is thus somewhat faster. Algorithm 3.4.2 is thus recommended as the preferred construction for $F^{n,k}(\vec{\alpha})$.

Remark 3.4.7. Message encoding from the linear algebra perspective consists of two steps. First, the generating matrix G must be constructed. In the non-systematic case, $G = V^{k,n}(\vec{\alpha})$ is required and is constructed via Algorithm 3.1.1; this costs kn operations. In the systematic case, $G = F^{n,k}(\vec{\alpha})$ is required and is constructed via Algorithm 3.4.2; this costs $2k(k-1)(r+1)$ operations. Next $\vec{c} = G\vec{m}$ must be computed via matrix multiplication; this costs $(2k-1)n$ operations in the non-systematic case, and $(2k-1)r$ operations in the systematic case. Typically, the generating matrix is precomputed once, then used many times to encode many different codewords. With this in mind, we will count the computational cost of encoding in the linear algebra perspective as only the cost of the matrix multiplication and not the cost of generating G.

3.5 The Computational Cost of Message Encoding and Message Recovery

We have now seen two different types of encoding (systematic and non), each from two different perspectives (polynomial and linear algebra). How do we choose which type of encoding to use and how to implement it? Each perspective yields a different algorithm for

[4]Operation count for Algorithm 3.4.2:

$$\begin{aligned}\sum_{c=0}^{k-1}\left[\left(\sum_{\substack{j=0\\j\neq c}}^{k-1}2\right)+\left(\sum_{r=k}^{n-1}\sum_{\substack{j=0\\j\neq c}}^{k-1}2\right)\right] &= \sum_{c=0}^{k-1}\left[2(k-1)+2(k-1)(n-k)\right]\\ &= 2k\left[(k-1)+(k-1)r\right]\\ &= 2k(k-1)(r+1).\end{aligned}$$

Encoding:	**Non-systematic**		**Systematic**	
Perspective:	**Polynomial**	**Linear Algebra**	**Polynomial**	**Linear Algebra**
Encode Alg.	Algorithm A.3.1	Remark 3.4.7	Algorithm 2.3.1	Remark 3.4.7
Encode Cost	$2(k-1)n$	$(2k-1)n + [kn]$	$6k^2+k(2r-1)-2r$	$(2k-1)r+[2k(k-1)(r+1)]$
Recovery Alg.	Algorithm 2.2.1	Algorithm 3.2.1	Read $\vec{m}$ from first k characters of $\vec{c}$	
Recovery Cost	$6k^2-k$	$2k^2+k-3+[\frac{2}{3}k^3+\frac{3}{2}k^2+\frac{5}{6}k-3]$	0	0

TABLE 3.1. Computational cost of message encoding and message recovery: The cost of creating the generating matrix for encoding in the linear algebra perspective is listed in square brackets as [cost]. This cost can be ignored in the typical case where the generating matrix is computed once then used many times to encode various codewords. Similarly, the cost of computing the LU factorization for message recovery in the linear algebra perspective is listed as [cost]; in the typical case the first k indices are used for message recovery, and this LU factorization may be precomputed and reused.

message encoding and decoding. A major factor in deciding on an encoding scheme is the computational cost of encoding and message recovery in that scheme. Table 3.1 summarizes the cost of message encoding and decoding in each of the four cases discussed. It shows that systematic encoding in the linear algebra perspective is the most computationally efficient. Although computing the generating matrix is somewhat expensive, this may be considered as a precomputed matrix and this cost should not be included in the cost of encoding. Given that, encoding costs a mere $2(k-1)r$ operations, and message recovery is free. For most applications, systematic encoding using linear algebra will be the best choice, and we will often assume this choice in the remainder of this text.

Remark 3.5.1. Although systematic encoding is the best choice for most code applications, there are some situations when non-systematic encoding would make sense and be worth the additional computational cost for message recovery. One such example is distributive storage. Imagine a system in which data is encoded non-systematically via a Reed-Solomon code and then each character in the codeword is stored in a different data warehouse. To accurately reconstruct the message, access to k warehouse would be needed. Access to any fewer revels little information about the message. This provides some amount

of data security. Since the data is encoded non-systematically, the warehouses are symmetric; it doesn't matter which k warehouses are accessed. On the other hand, loss of one or a few warehouses due to loss of electricity, hurricanes or earthquakes, or malicious attack would not compromise the integrity of the data in the system. This could form the basis for a secure distributive storage system.

Remark 3.5.2. The astute reader will recall that is possible to transform from the coefficients of a polynomial to its values and vice versa using the discrete fourier transform, see [5, Chapter 30.2]. When the discrete fourier transform is implemented using the fast fourier transform, this may be accomplished in $\Theta(n \log n)$ operations where n is the degree of the polynomial. This is a substantial improvement over the encoding and decoding algorithms so far presented. Unfortunately, to achieve this improved computational complexity the points at which the polynomial is evaluated must be the n nth roots of unity. For an arbitrary $n < 2^l$, these nth roots of unity (the solutions to the equation $x^n = 1$) are not necessarily all contained in the base field $\mathrm{GF}(2^l)$. Therefore, the Fast Fourier Transform approach is not easily adapted for the purpose of encoding and decoding messages.

Chapter 4
Erasures, Errors, and Distance

After the message has been encoded as a code word, it is typically stored for later use or transmitted for use elsewhere. During storage and transmission, errors may accumulate which corrupt the codeword. Thus the *received word* may be different from the codeword. The purpose of encoding the message before transmitting the codeword was to allow us to recover the message from the received word in spite of accumulated errors. In this section we examine two different types of errors that might occur, how to detect these errors, then how to correct them and fully recover the intended message. We use $\vec{r} = (r_0, r_1, \ldots, r_{n-1})^T$ to denote the received word. Note that if no errors have occurred, $\vec{r} = \vec{c}$.

4.1 Erasure Decoding

The simplest type of data corruption is called an *erasure*. This occurs when one or more characters in the received word $\vec{r}$ was corrupted or deleted at known locations, *i.e.* for certain values of i, r_i is known to be in error. Let $\mathcal{F} = \{i : r_i \neq c_i\}$ be the set of indices of the received word that are known to be in error, and define $\mathcal{I}$ to be a size k subset of $\overline{\mathcal{F}} = \{0, 1, \ldots, n-1\} \backslash \mathcal{F}$, *i.e.* $\mathcal{I}$ is a size k set of indices that are not in error. Although it may seem unrealistic that we would know the location of errors in the received word, there are several applications for Reed-Solomon codes in which this is indeed the case. Data storage systems are one such example and are discussed in Section 4.1.1.

From the perspective of polynomial encoding of Section 2, so long as there no more than $n-k$ erasures ($|\mathcal{F}| \leq n-k$), we still know at least k valid points on the message polynomial and can interpolate any k of these points to recover the message, just as we did to recover the message from the codeword when there were no erasures. If the encoding was non-systematic, erasure decoding may be accomplished by calling MessageRecoverLI($\vec{r}, \vec{\alpha}, \mathcal{I}$)

from Algorithm 2.2.1 and costs $6k^2 - k$ operations. If the encoding was systematic, the message polynomial $\mu(x)$ must first be recovered via Lagrange interpolation, then the values at erased message indices recovered by evaluating this polynomial. Algorithm 4.1.1 implements such an erasure decoding algorithm, and costs $6k^2 - k + f_m 2(k-1)$ operations where $f_m = |\{0, 1, \ldots k-1\} \cap \mathcal{F}|$.

Algorithm 4.1.1 Erasure decoding of a systematic codeword using Lagrange Interpolation

Dependencies: LagrangeInterpolate defined in Algorithm A.3.8, PolyEval defined in Algorithm A.3.1.

```
1: procedure ErasureDecodeSysLI(r⃗, α⃗, F)   ⋆Recover the message m⃗ from r⃗ using
                                            ⋆Lagrange interpolation if the indices F were erased
2:     I ← first k values in {0, 1, ... n − 1}\F
3:     μ⃗ ←LagrangeInterpolate(α⃗_I, r⃗_I)
4:     m⃗ ← r⃗_{0:k−1}                                  ⋆Recover m⃗ from μ(x)
5:     for i ∈ {0, 1, ... k − 1} ∩ F do    ⋆Only recover message indices that were erased
6:         m_i ←PolyEval(μ⃗, α_i)                      ⋆m_i = μ(α_i)
7:     end for
8:     return m⃗
9: end procedure
```

Computational complexity: No more than $6k^2 - k + 2f_m(k-1)$ operations required[1], where $f_m = |\{0, 1, \ldots k-1\} \cap \mathcal{F}|$.

From the linear algebra perspective with generating matrix G, so long as there are no more than $n-k$ erasures the system of equations $\vec{r} = G\vec{m}$ contains at least k valid equations. We can solve any k of the valid equations to recover $\vec{m}$. If the encoding was non-systematic, erasure decoding may be accomplished by calling MessageRecoverLA$(\vec{r}, G, \mathcal{I})$ from Algorithm 3.2.1 and costs $\frac{2}{3}k^3 + \frac{7}{2}k^2 + \frac{11}{6}k - 6$ operations. For a systematic encoding, the same technique could be used to decode, but it is actually possible to decode from erasures more cheaply by exploiting the known structure of a systematic generator.

[1]Computational complexity of Algorithm 4.1.1: The call to LagrangeInterpolate costs $6k^2 - k$ operations. Each call to PolyEval costs $2(k-1)$ operations. This yields a total of $6k^2 - k + f_m 2(k-1)$ operations where $f_m = |\{0, 1, \ldots k-1\} \cap \mathcal{F}|$.

If G is systematic, it has the form

$$G = \begin{bmatrix} I_k \\ \star \end{bmatrix},$$

where I_k is the $k \times k$ identity matrix, and $\star$ some matrix. Let $G_{\mathcal{I}}$ denote the matrix constructed from the rows of G whose indices are in $\mathcal{I}$; $G_{\mathcal{I}}$ is a $k \times k$ matrix. Construct $\vec{r}_{\mathcal{I}}$ similarly. Then

$$G_{\mathcal{I}}\vec{m} = \vec{r}_{\mathcal{I}}.$$

It will also be convenient to reorder $\vec{m}$ so that the elements of $\vec{m}$ that were not erased from $\vec{r}$ appear before the elements that were erased. Denote this rearranged message by $\hat{m}$. Mathematically, there exists a permutation matrix P such that $\hat{m} = P\vec{m}$. Define $\hat{G} = G_{\mathcal{I}}P^T$. Then

$$\hat{G}\hat{m} = G_{\mathcal{I}}P^T P\vec{m} = G_{\mathcal{I}}\vec{m} = \vec{r}_{\mathcal{I}}$$

since $P^T = P^{-1}$ for permutation matrices. Let f_m be as defined above. This equation has the block structure

$$\begin{matrix} \hat{G} \\ \end{matrix} \quad \begin{matrix} \hat{m} \\ \end{matrix} \quad \begin{matrix} \vec{r}_{\mathcal{I}} \\ \end{matrix}$$
$$\begin{bmatrix} I_{k-f_m} & 0 \\ A & B \end{bmatrix} \begin{bmatrix} \vec{x} \\ \vec{y} \end{bmatrix} = \begin{bmatrix} \vec{x} \\ \vec{w} \end{bmatrix}.$$

Notice that $\vec{x} = \vec{r}_{\{i\in\mathcal{I}, i<k\}}$, $\vec{w} = \vec{r}_{\{i\in\mathcal{I}, i\geq k\}}$, $A = G_{\{r\in\mathcal{I}, i\geq k\},\{c\in\mathcal{I}, c<k\}}$ is size $f_m \times (k - f_m)$, $B = G_{\{r\in\mathcal{I}, i\geq k\},\{c\in\mathcal{F}, c<k\}}$ is size $f_m \times f_m$, and $\vec{y}$ is the unknown message symbols at positions that were erased[2]. This block structure yields the reduced equation

$$B\vec{y} = \vec{w} - A\vec{x}.$$

Since B is a square submatrix of G, it is guaranteed to be invertible and this equation is solvable. Once $\vec{y}$ is found, $\vec{m}$ may be reconstructed by simply rearranging the elements of $\vec{x}$ and $\vec{y}$. Algorithm 4.1.2 implements erasure decoding using this method and costs $\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right)f_m - 6$ operations.

[2]See Appendix C for notes on notation used here.

[3]Computational complexity of Algorithm 4.1.2: Computation of $\vec{w} - A\vec{x}$ on line 6 requires $[2(k - f_m) -$

Algorithm 4.1.2 Erasure decoding of a systematic codeword using linear algebra

Dependencies: LU defined in Algorithm A.4.1, SolveLU defined in Algorithm A.4.2.

1: **procedure** ErasureDecodeSysLA($\vec{r}, G, \mathcal{F},$) ⋆*Recover the message* $\vec{m}$ *from* $\vec{r}$ *using ⋆linear algebra if the indices* $\mathcal{F}$ *were erased*
2: $A \leftarrow G_{\{r\in\mathcal{I}, i\geq k\},\{c\in\mathcal{I}, c<k\}}$
3: $B \leftarrow G_{\{r\in\mathcal{I}, i\geq k\},\{c\in\mathcal{F}, c<k\}}$
4: $\vec{x} \leftarrow \vec{r}_{\{i\in\mathcal{I}, i<k\}}$
5: $\vec{w} \leftarrow \vec{r}_{\{i\in\mathcal{I}, i\geq k\}}$
6: $\vec{z} \leftarrow \vec{w} - A\vec{x}$
7: $L, U \leftarrow \mathrm{LU}(B)$ ⋆*Compute the LU decomposition of* B
8: $\vec{y} \leftarrow \mathrm{SolveLU}(L, U, \vec{z})$ ⋆*Solve* $B\vec{y} = \vec{c}$
9: Reconstruct $\vec{m}$ from $\vec{x}$ and $\vec{y}$ ⋆*Reconstruct the message*
10: **return** $\vec{m}$
11: **end procedure**

Computational complexity: $\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m - 6$ operations required[3].

4.1.1 Application to Data Storage Systems

Reed-Solomon codes can be used in data storage systems to protect stored data from corruption or device failure. RAID systems are often used for this task. A RAID (Redundant Array of Independent Disks) is a collection of n storage devices with the capacity to hold k devices worth of data and $r = n - k$ devices worth of redundant data. As always with Reed-Solomon codes, the stored data is broken into l bit characters, and the l bit characters are grouped into length k messages. The message is then systematically encoded as a length n codeword as in Section 3.4. One character of each codeword is stored on each device.

1]$f_m + f_m = 2f_m(k - f_m)$ operations. Computation of the LU decomposition on line 7 costs $\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \frac{5}{6}f_m - 3$ operations. Solving the linear system on line 8 costs $2f_m^2 + f_m - 3$ operations. Construction of $A, B, \vec{x}, \vec{w}, \vec{z}$, and reconstruction of $\vec{m}$ will not be counted in the cost of the algorithm as they involve no finite field arithmetic and can be circumvented through clever referencing of G and $\vec{r}$. Therefore, the total operation count for Algorithm 4.1.2 is

$$2f_m(k - f_m) + \frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \frac{5}{6}f_m - 3 + 2f_m^2 + f_m - 3 = \frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m - 6$$

The failure of a device in the RAID corresponds to an erasure in the codewords. So long as no more than r devices fail simultaneously, full data recovery is possible using erasure decoding, per Section 4.1.

4.2 Error Detection

It is convenient when we know the locations of all errors in a codeword, but it is not generally the case. When presented with a received word $\vec{r}$, we may have no outside information to tell us if the received word was corrupted or not but will have to determine its status based solely on the contents of $\vec{r}$ itself. We refer to this situation broadly as the errors case as opposed to the erasures case. Determining whether $\vec{r}$ was corrupted is called *error detection*, whereas recovering the message and/or the codeword from $\vec{r}$ is called *error correction*.

Assume there was some original message $\vec{m}$ which was encoded as $\vec{c}$. After storage or transmission, we obtain the received word $\vec{r}$. The goal of error detection is to determine if $\vec{c} = \vec{r}$. This cannot always be done, but may be achieved in practice with high probability. The proxy we use is to check if $\vec{r}$ is a codeword in the code in use. If it is, assume that $\vec{r} = \vec{c}$, and there were no errors. If it is not, it is certain that $\vec{r}$ is a corrupted version of $\vec{c}$. This method can fail when multiple errors serendipitously transform $\vec{c}$ into some other codeword $\vec{c}'$ (see Example 4.2.2).

To determine if $\vec{r}$ is a codeword, assume that it is and recover the corresponding message $\vec{m}'$. Next, reencode $\vec{m}'$ to obtain the codeword $\vec{c}'$. Compare $\vec{r}$ and $\vec{c}'$. If they are the same, assume that there are no errors and $\vec{r} = \vec{c} = \vec{c}'$ and $\vec{m} = \vec{m}'$. If $\vec{r}$ and $\vec{c}'$ are different, then errors have occurred.

Example 4.2.1. Let $l = 8$, $k = 5$, $n = 8$, $\vec{\alpha} = (0, 1, 2, 3, 4, 5, 6, 7)^T$, and

$$\vec{m} = (233, 211, 0, 7, 18)^T.$$

Encoding systematically yields

$$\vec{c} = (233, 211, 0, 7, 18, 166, 14, 135)^T.$$

$\vec{c}$ is then transmitted and an error occurs resulting in a received word

$$\vec{r} = (233, 117, 0, 7, 18, 166, 14, 135)^T.$$

Since this encoding is systematic, we can recover a message from $\vec{r}$ by simply selecting the first k characters, so we have

$$\vec{m}' = (233, 117, 0, 7, 18)^T.$$

Encoding $\vec{m}'$ yeilds

$$\vec{c}' = (233, 117, 0, 7, 18, 243, 87, 45)^T.$$

Clearly $\vec{r} \neq \vec{c}'$ so we correctly conclude that $\vec{r} \neq \vec{c}$ and an error has occurred.

Example 4.2.2. This example illustrates how error detection can fail. Let $l = 8$, $k = 5$, $n = 6$, $\vec{\alpha} = (0, 1, 2, 3, 4, 5)^T$, and

$$\vec{m} = (233, 211, 0, 7, 18)^T.$$

Encoding systematically yields

$$\vec{c} = (233, 211, 0, 7, 18, 166)^T.$$

$\vec{c}$ is then transmitted and two errors occur resulting in a received word

$$\vec{r} = (233, 117, 0, 7, 18, 243)^T.$$

Since this encoding is systematic, we can recover a message from $\vec{r}$ by simply selecting the first k characters, so we have

$$\vec{m}' = (233, 117, 0, 7, 18)^T.$$

Encoding $\vec{m}'$ yeilds

$$\vec{c}' = (233, 117, 0, 7, 18, 243)^T.$$

$\vec{r} = \vec{c}'$ so we incorrectly conclude that $\vec{r} = \vec{c}$ and there have been no errors.

4.3 Conditions for Error Detection and Correction

As we have just seen, in most cases we can detect errors in the received word, but it is not always possible. We will now develop a framework that will allow us to state precisely under which conditions errors will be detectable. Moreover, we will derive conditions under which we can recover the correct message from a corrupted received word.

4.3.1 Distance

The notion of distance between words is important in coding theory, and will provide an appropriate tool to determine when errors are detectable and correctable. In this section we define the *Hamming distance*, and prove an important result about the distance between two codewords in a Reed-Solomon code. For generality, we assume that all arithmetic is performed over some field $\mathbb{F}$, although in applications in this paper we will always have $\mathbb{F} = \text{GF}(2^l)$.

Definition 4.3.1. Let $\vec{a}$ and $\vec{b}$ be vectors in $\mathbb{F}^n$. Then the *Hamming distance* or just *distance* between $\vec{a}$ and $\vec{b}$ is

$$d(\vec{a}, \vec{b}) = n - \sum_{i=1}^{n} \delta_{a_i, b_i} = \sum_{i=1}^{n} (1 - \delta_{a_i, b_i}).$$

$d(\vec{a}, \vec{b})$ is the count of the indices where $\vec{a}$ and $\vec{b}$ differ. The Hamming distance was introduced by Hamming in [10] where he noted that d is a metric on $\mathbb{F}^n$.

Exercise 4.3.2. Show that the Hamming distance is a metric on $\mathbb{F}^n$, *i.e.* that for all $\vec{a}, \vec{b}, \vec{c} \in \mathbb{F}^n$, d is

1. Positive definite: $d(\vec{a}, \vec{b}) \geq 0$, and $d(\vec{a}, \vec{b}) = 0$ if and only if $\vec{a} = \vec{b}$,
2. Symmetric: $d(\vec{a}, \vec{b}) = d(\vec{b}, \vec{a})$,
3. has the Triangle inequality: $d(\vec{a}, \vec{c}) \leq d(\vec{a}, \vec{b}) + d(\vec{b}, \vec{c})$.

Example 4.3.3. The Hamming distance between the vectors $\vec{a} = (3, 7, 2, 4, 5)$ and $\vec{b} = (3, 1, 2, 8, 5)$ is two since they differ in two locations.

Remark 4.3.4. Let $\vec{c}$ be a codeword is a Reed-Solomon code, and $\vec{r}$ be a received word. Then $d(\vec{c}, \vec{r})$ gives the number of errors in the received word.

Definition 4.3.5. The *distance d of a code* is the minimum distance between any two codewords in the code C:

$$d = \min_{\vec{c}_1, \vec{c}_1 \in C} d(\vec{c}_1, \vec{c}_2).$$

Theorem 4.3.6. *Let $\vec{c_1}$ and $\vec{c_2}$ be two codewords in a Reed-Solomon code. Then $d(\vec{c_1}, \vec{c_2}) \geq n - k + 1$ and so Reed-Solomon codes have distance $d = n - k + 1$.*

Proof. We'll prove this from the linear algebra perspective of encoding, without loss of generality, and assume the generating matrix G was used. In search of contradiction assume there exists distinct codewords $\vec{c_1}$ and $\vec{c_2}$ with $d(\vec{c_1}, \vec{c_2}) \leq n - k$. Then $\vec{c_1}$ and $\vec{c_2}$ agree at at least k indices. There exists $\vec{m_1}$ and $\vec{m_2}$ such that $\vec{c_1} = G\vec{m_1}$ and $\vec{c_2} = G\vec{m_1}$. Construct $\hat{G}$ from k rows of G corresponding to indices where $\vec{c_1}$ and $\vec{c_2}$ agree. Then $\hat{G}\vec{m_1} = \hat{G}\vec{m_2}$. Since the generators for Reed-Solomon codes are IDMs, $\hat{G}$ is invertible, and thus $\vec{m_1} = \vec{m_2}$ and $\vec{c_1} = \vec{c_2}$. This is a contradiction, and so no such c_1 and c_2 may exist. □

Exercise 4.3.7. Prove Theorem 4.3.6 from the polynomial perspective of encoding of Chapter 2.

Remark 4.3.8. $n - k + 1$ errors must occur to transform one codeword into another. Therefore, we are guaranteed to be able to detect $n - k$ or fewer errors in a received word. If $n-k+1$ or more errors occurred, they *might* go undetected. We make a base assumption that the probability of an error affecting any particular character in a codeword is smaller than one half, and therefore it is more probable that a small number of errors accrue in a received word than that a large number of errors have accrued. By making n sufficiently larger than k, we can design the code so that an undetected error is a sufficiently low

probability event. Of course, this requires greater redundancy in the code which makes the code less efficient.

Definition 4.3.9. A code of length n (number of characters in a codeword), dimension k (number of characters in the message), and distance d is called an (n, k, d) code.

Corollary 4.3.10. *Reed-Solomon codes are* $(n, k, n-k+1)$ *codes.*

Lemma 4.3.11. *(Singleton bound) Let* C *be an* (n, k, d) *code over a field* $\mathbb{F}$ *with* $|\mathbb{F}| = q$. *Then*

$$|C| \leq q^{n-d+1}.$$

As well, $d \leq n - k + 1$. *These bounds are both referred to as the* Singleton bound.

Proof. Since any two codewords in C have distance at least d, they must differ in at least d positions. Let C' be the code of length $n-d+1$ constructed by truncating the last $d-1$ characters from each codeword in C:

$$C' = \{\vec{a} \in \mathbb{F}^{n-d+1} : \exists \vec{c} \in C \text{ such that } \vec{c} = (a_0, \ldots, a_{n-d}, c_{n-d+1}, \ldots, c_{n-1}\}.$$

Clearly, $|C| = |C'| \leq q^{n-d+1}$.

On the other hand, since the code is of dimension k, $|C| = q^k$ and thus

$$q^k = |C| \leq q^{n-d+1},$$

so $k \leq n - d + 1$, and $d \leq n - k + 1$. □

Definition 4.3.12. An (n, k, d) code C that achieves equality in the Singleton bound is called a *maximum distance separable* (*MDS*) code.

Corollary 4.3.13. *Reed-Solomon codes are MDS codes.*

Proof. This follows directly from definition and Corollary 4.3.10. □

Definition 4.3.14. Let $\vec{a}$ and $\vec{b}$ be vectors in $\mathbb{F}^n$ and $\mathcal{I} \subset \{0, 1, \ldots, n-1\}$ be a set of indices. The the *Hamming distance restricted* by $\mathcal{I}$ between $\vec{a}$ and $\vec{b}$ is

$$d_{\mathcal{I}}(\vec{a}, \vec{b}) = |\mathcal{I}| - \sum_{i \in \mathcal{I}} \delta_{a_i, b_i} = \sum_{i \in \mathcal{I}} (1 - \delta_{a_i, b_i}).$$

$d_{\mathcal{I}}(\vec{a}, \vec{b})$ is the count of the number of indices where $\vec{a}$ and $\vec{b}$ differ, ignoring those indices not in $\mathcal{I}$.

Exercise 4.3.15. Show that there exists $\vec{a}$, $\vec{b}$, and $\mathcal{I}$ such that $d_{\mathcal{I}}(\vec{a}, \vec{b}) = 0$ but $\vec{a} \neq \vec{b}$ and therefore the restricted Hamming distance is not a metric.

Exercise 4.3.16. Show that the restricted Hamming distance is a pseudometric (or semi-metric) on $\mathbb{F}^n$, *i.e.* that for all $\vec{a}, \vec{b}, \vec{c} \in \mathbb{F}^n$, d is

1. Positive: $d_{\mathcal{I}}(\vec{a}, \vec{b}) \geq 0$
2. Symmetric: $d_{\mathcal{I}}(\vec{a}, \vec{b}) = d_{\mathcal{I}}(\vec{b}, \vec{a})$
3. Triangle inequality: $d_{\mathcal{I}}(\vec{a}, \vec{c}) \leq d_{\mathcal{I}}(\vec{a}, \vec{b}) + d_{\mathcal{I}}(\vec{b}, \vec{c})$

Exercise 4.3.17. Show that $d(\vec{a}, \vec{b}) \leq d_{\mathcal{I}}(\vec{a}, \vec{b}) + n - |\mathcal{I}|$.

Example 4.3.18. The Hamming distance between the vectors $\vec{a} = (3, 7, 2, 4, 5)$ and $\vec{b} = (3, 1, 2, 8, 5)$ restricted by $\mathcal{I} = \{0, 1, 2\}$ is $d_{\mathcal{I}}(\vec{a}, \vec{b}) = 1$ since they differ in one of the first three locations.

4.3.2 Assumptions and Conditions for Error Correction

Given a received word $\vec{r}$ in which errors have been detected and no outside information about the nature of the errors accumulated, it is impossible to recover the intended message $\vec{m}$ with absolute certainty. Two different codewords could be transformed into the same received word through the accumulation of various errors, and by examining only the received word, it is impossible to tell which of the codewords we started with. Error

correction is necessarily an uncertain business and the best we can do is to seek to recover $\vec{m}$ with high probability. To do so, we always assume that the codeword closest in Hamming distance to the received word is the correct codeword. This is equivalent to choosing the codeword corresponding the fewest possible accumulated errors in the received word. This assumption is called *maximum likelihood decoding*. The problem of *decoding* (error correction) will be to identify the closest codeword to a given received word as efficiently as possible. Maximum likelihood decoding will occasionally result in an incorrect decoding. In this section we derive conditions under which maximum likelihood decoding is guaranteed to succeed and see that success can be achieved with high probability.

Theorem 4.3.19. *(Error bound) In a Reed-Solomon code, if t errors have accumulated in the received word $\vec{r}$, and $t < \frac{d}{2}$, then the closest codeword to $\vec{r}$ is the correct codeword.*

Proof. Let $\vec{c}$ be the original codeword and $\vec{r}$ be the received word in which t errors have accumulated. Then $d(\vec{c}, \vec{r}) = t$. In search of contradiction, assume there exits some other codeword $\vec{c}'$ that is as close as or closer to $\vec{r}$ than $\vec{c}$. Then $d(\vec{r}, \vec{c}') \leq t$ and $d(\vec{c}, \vec{c}') \leq d(\vec{c}, \vec{r}) + d(\vec{r}, \vec{c}') \leq 2t < d$. This violates the distance bound for Reed-Solomon codes, thus no such $\vec{c}'$ may exist. □

If $t \geq \frac{d}{2}$ errors are present, then some other codeword may be closer to or as close to the received word as the original codeword. If so, the decoding may be incorrect.

It is also sometimes possible to recover a message from a received word in the presence of both erasures (in known locations) and errors (in unknown locations). In this case we seek the closest codeword to the received word not counting errors in the known erasure locations.

Corollary 4.3.20. *(Error bound with erasures) In a Reed-Solomon, if f erasures in the known locations $\mathcal{F} \subset \{0, 1, \ldots, n-1\}$ and t errors (in unknown locations) have accumulated in the received word $\vec{r}$, and $2t + f < d$, then the closest codeword to $\vec{r}$ in Hamming distance*

restricted by $\overline{\mathcal{F}}$ is the correct codeword. Here $\overline{\mathcal{F}}$ is the complement of $\mathcal{F}$: $\overline{\mathcal{F}} = \{0, 1, \ldots n - 1\} \backslash \mathcal{F}$.

Proof. By assumption, there exists a codeword $\vec{c}$ such that $d_{\overline{\mathcal{F}}}(\vec{c}, \vec{r}) = t$. In search of contradiction, assume there exists some other codeword $\vec{c}'$ such that $d_{\overline{\mathcal{F}}}(\vec{c}', \vec{r}) \leq t$. Then, by Exercise 4.3.16, $d_{\overline{\mathcal{F}}}(\vec{c}, \vec{c}') \leq 2t$, and, by Exercise 4.3.17, $d(\vec{c}, \vec{c}') \leq 2t + f < d$. This violates the distance bound for Reed-Solomon codes, thus no such $\vec{c}'$ may exist. □

Chapter 5
Error Correction

The most challenging aspect to error correcting codes is efficient error correction when the errors are in unknown locations. In the remainder of this dissertation, we explore a few algorithms for error correction for Reed-Solomon codes. These error correction algorithms all follow a common three-phase structure:

- In **phase 1**, a "key equation" is derived based on properties of the code. A particular instance of this key equation depends on the received word, and the various encoding parameters. The solution of the key equation will give information about the most probable codeword corresponding to the given received word, the locations of the errors, and/or their magnitudes.
- In **phase 2**, a solution to this key equation is computed.
- In **phase 3**, the solution of the key equation is used to recover the original message or codeword.

In the following chapters, I will present multiple approaches to each of these phases, and compare the efficiency of each.

5.1 Inputs, Assumptions, and Notation for ECC Algorithms

Each of these algorithms will accept the following inputs:

- the parameters, k, n, and $\vec{\alpha}$,
- a received word $\vec{r}$ in which errors have been detected (Section 4.2),
- the number of erasures f in $\vec{r}$ along with the set of indices $\mathcal{F}$ at which those erasures occurred,

- the type of encoding (non-systematic, or systematic),
- (optional) the generating matrix for the code, F.

l, the size of a character, will be treated as a system wide constant, and it will often be assumed that the encoding was systematic. These inputs are subject to the constraints

$$k < n \leq 2^l, \quad f < n - k, \quad |\mathcal{F}| = f, \quad |\vec{\alpha}| = n,$$

and all α_i unique. Given these inputs, define the following intermediary items:

- $r = n - k$, the number of redundant characters in a codeword,
- $d = r + 1 = n - k + 1$, the distance of the code,
- $\mathcal{W} = \overline{\mathcal{F}} = \{0, 1, \ldots, n-1\} \backslash \mathcal{F}$, the set of indices of characters in $\vec{r}$ that were not erased,
- $\hat{t} = \lfloor \frac{n-k-f}{2} \rfloor$, the maximum number of errors within the error bound

In addition, in deriving the algorithms it will be useful to have notation for the following unknown items:

- $\vec{c}$, the unique codeword such that $d_{\mathcal{W}}(\vec{r}, \vec{c}) \leq \hat{t}$,
- $\mu(x)$, the message polynomial under systematic encoding, $\mu(\vec{\alpha}) = \vec{c}$, $\deg(\mu) = k - 1$,
- $\vec{m}$, the message corresponding to the codeword $\vec{c}$ under the given encoding,
- t, the number of errors present in $\vec{r}$, $t \leq \hat{t}$,
- $\delta = \hat{t} - t$
- $\vec{e}$, the error vector, $\vec{r} = \vec{c} + \vec{e}$,
- $\mathcal{E}$, the set of indices of errors, $\mathcal{E} = \{i : e_i \neq 0\}$, $|\mathcal{E}| = t$,

- $g(x)$, the *error locator polynomial*, $g(x) = \prod_{i \in \mathcal{E}}(x - \alpha_i)$.

We assume that $t \leq \hat{t}$ and therefore the number of errors is within the error bounds of Section 4.3.2. Under this assumption, maximum likelihood decoding is guaranteed to succeed. We may think of the problem of decoding equivalently as identifying $\vec{c}$, $\vec{m}$, $m(x)$, $\vec{e}$, $\mathcal{E}$, or $g(x)$ as given any one of these items we can easily calculate all the others.

Exercise 5.1.1. Show that, given the inputs listed above, finding any one of $\vec{c}$, $\vec{m}$, $m(x)$, $\vec{e}$, $\mathcal{E}$, and $g(x)$ is equivalent to finding them all.

5.2 The Modified Welch-Berlekamp Algorithm

We start our discussion of error correcting codes by considering a conceptually simple decoding algorithm with suboptimal computational efficiency. This algorithm is based on polynomial interpolation, and the presentation follows the work of Gemmel and Sudan in [9]. I will to refer to this algorithm as the *modified Welch-Berlekamp algorithm*.

5.2.1 Algorithm Outline

The problem of decoding is to find the unique degree $k-1$ or less polynomial $m(x)$, such that

$$m(\alpha_i) = r_i$$

for all but t values of $i \in \mathcal{W}$, where $t < \frac{d-f}{2}$. There exists a non-zero, degree t polynomial q such that

$$q(\alpha_i)m(\alpha_i) = q(\alpha_i)r_i, \qquad \text{for all } i \in \mathcal{W}.$$

In particular, if $\mathcal{E}$ is the set of the (unknown) indices of the errors in $\vec{r}$, and $q(x)$ is the monic error locator polynomial $g(x) = \prod_{i \in \mathcal{E}}(x - \alpha_i)$, then the above equation will hold for $q(x) = cg(x)$ for any constant c. Let $s(x) = q(x)m(x)$. $\deg(s) \leq k + t - 1$, and

$$s(\alpha_i) = q(\alpha_i)r_i, \qquad \text{for all } i \in \mathcal{W}. \tag{5.1}$$

Let $s(x) = \sum_{i=0}^{k+t-1} s_i x^i$ and $q(x) = \sum_{i=0}^{t} q_i x^i$. Then there are $k + 2t + 1$ unknowns that define $s(x)$ and $q(x)$, and (5.1) yields $n - f$ independent equations. Since $2t < d - f = n - k - f + 1$, the number of unknowns is at most $k + n - k - f + 1 = n - f + 1$ and there is at most one more unknown than there are equations. By requiring that q is non-zero, we are assured a solution that is unique up to a constant factor.

5.2.2 Computing $m(x)$ and Recovering $\vec{m}$

Once a solution $s(x), q(x)$ is found, we can recover $m(x)$ by $m(x) = \frac{s(x)}{q(x)}$. If the encoding was non-systematic, then the coefficients of $m(x)$ form the message vector $\vec{m}$. If the encoding was systematic, then we must compute $\vec{m} = m(\vec{\alpha}_{0:k-1})$. Notice that in previous sections we have referred to the polynomial $m(x)$ found here as $\mu(x)$ in the systematic encoding case; in the algorithm outlined in Section 5.2.6 we will use this notation to emphasize the assumption of systematic encoding.

5.2.3 t is Unknown

The technique just outlined would lead immediately to an algorithm for finding $m(x)$ if t were known *a priori*. However, we generally do not know how many errors are present in $\vec{r}$ before decoding, just that there are some errors. We will assume that $t < \frac{d-f}{2}$ and therefore that maximum likelihood decoding will succeed. Let $\hat{t}$ be the maximum allowable value of t, $\hat{t} = \lfloor \frac{d-f-1}{2} \rfloor = \lfloor \frac{n-k-f}{2} \rfloor$. We can rewrite the above technique in terms of the assumption that $t \leq \hat{t}$ without assuming that we know t:

We seek the unique degree k-1 or less polynomial $m(x)$, such that

$$m(\alpha_i) = r_i$$

for all but at most $\hat{t}$ values of $i \in \mathcal{W}$. There exists a non-zero polynomial q such that $\deg(q) \leq \hat{t}$, and

$$q(\alpha_i)m(\alpha_i) = q(\alpha_i)r_i, \qquad \text{for all } i \in \mathcal{W}.$$

Let $s(x) = q(x)m(x)$. Then

$$\forall i \in \mathcal{W},\ s(\alpha_i) = q(\alpha_i)r_i \ \text{ with } \ \deg(s) \leq k + \hat{t} - 1 \text{ and } \deg q \leq \hat{t}. \tag{5.2}$$

Let $s(x) = \sum_{i=0}^{k+\hat{t}-1} s_i x^i$ and $q(x) = \sum_{i=0}^{\hat{t}} q_i x^i$. Then there are $k+2\hat{t}+1 = n-f+1$ unknowns that define $s(x)$ and $q(x)$, and (5.2) yields $n - f$ equations which are not necessarily independent. This system is underdetermined and thus many solutions exist. Nonetheless, we shall see that any solution to (5.2) yields the same value for $m(x)$, and that we may use standard linear algebra techniques to find such a solution.

Theorem 5.2.1. *Let s, q be a solution to (5.2). Then $\frac{s(x)}{q(x)} = m(x)$ where $m(x)$ is the unique polynomial such that $\deg m \leq k - 1$ and $m(\alpha_i) = r_i$ for all but at most $\hat{t}$ values of $i \in \mathcal{W}$.*

Proof. By assumption, there exists $t \leq \hat{t}$ such that there are t errors in $\vec{r}$, and thus there exists a polynomial $m(x)$ with $\deg m \leq k - 1$ such that $m(\alpha_i) = r_i$ for all but t values of $i \in \mathcal{W}$. Let $g(x)$ be the error locator polynomial

$$g(x) = \prod_{i \in \mathcal{E}} (x - \alpha_i)$$

and $s(x) = m(x)g(x)$. Then $\deg(g) = t$, $\deg(s) \leq k + t - 1$, and s, g satisfy (5.2).

Let $l(x), u(x)$ be another solution to (5.2). For all $i \in \mathcal{W}$ we have

$$s(\alpha_i) = g(\alpha_i)r_i \qquad \text{and} \qquad l(\alpha_i) = u(\alpha_i)r_i.$$

Thus

$$\begin{aligned} s(\alpha_i)u(\alpha_i)r_i &= l(\alpha_i)g(\alpha_i)r_i \\ s(\alpha_i)u(\alpha_i) &= l(\alpha_i)g(\alpha_i) \end{aligned}$$

for $i \in \mathcal{W}$. Notice that if $r_i = 0$ then we must have $s(\alpha_i) = l(\alpha_i) = 0$, and thus the cancellation above applies in both the $r_i = 0$ and the $r_i \neq 0$ cases. $\deg(s \cdot u) \leq k+t+\hat{t}-1 \leq$

$n - f - 1$ and similarly $\deg(l \cdot g) \leq n - f - 1$. $s(x)u(x)$ and $l(x)g(x)$ agree at $n - f$ points but are only at most degree $n - f - 1$ polynomials and thus must be identically equal:

$$s(x)u(x) = l(x)g(x).$$

Thus

$$\frac{l(x)}{u(x)} = \frac{s(x)}{g(x)} = m(x)$$

is unique. □

Corollary 5.2.2. *For any q that satisfies (5.2), $g(x) \mid q(x)$.*

Proof. Suppose s, q is a solution to (5.2). By Theorem 5.2.1, $\frac{s(x)}{q(x)} = m(x)$ with $\deg m \leq k - 1$ and $m(\alpha_i) = r_i$ for all but at most $\hat{t}$ values of $i \in \mathcal{W}$. Therefore, we may rewrite (5.2) as

$$q(\alpha_i)m(\alpha_i) = q(\alpha_i)r_i \quad \forall i \in \mathcal{W}.$$

In search of contradiction, assume that there exists some $j \in \mathcal{W}$ such that $m(\alpha_j) \neq r_k$ and $(x - \alpha_j) \nmid g(x)$. Then

$$q(\alpha_j)m(\alpha_j) = q(\alpha_j)r_j,$$

and dividing by $q(\alpha_j)$ yields $m(\alpha_j) = r_j$. This is a contradiction, so $m(\alpha_j) \neq r_j$ implies $(x - \alpha_j) \mid q(x)$, which proves the corollary. □

Theorem 5.2.1 allows us freedom to choose easy to identify solutions to (5.2) with the assurance that any choice will lead to the same correct decoding. In order to solve (5.2) computationally, we will first translate it into a linear system, then apply standard techniques from linear algebra.

5.2.4 Translation to a Linear System

If $\mathcal{I} \subset \{0, \ldots n - 1\}$ is a set of indices, then we define $\vec{\alpha}_\mathcal{I}$ to be the vector of elements of $\vec{\alpha}$ with indices in $\mathcal{I}$. Thinking of $\vec{\alpha}_\mathcal{I}$ as a set, $\vec{\alpha}_\mathcal{I} = \{\alpha_i : i \in \mathcal{I}\}$, but as a vector the items in $\vec{\alpha}_\mathcal{I}$ maintain the same ordering as in $\vec{\alpha}$.

Let $\vec{s} = (s_0, \ldots, s_{k+\hat{t}-1})^T$ and $\vec{q} = (q_0, \ldots, q_{\hat{t}})^T$. Then we may rewrite (5.2) as

$$V^{n-f,k+\hat{t}}(\vec{\alpha}_{\mathcal{W}})\vec{s} = \text{diag}(\vec{r}_{\mathcal{W}})V^{n-f,\hat{t}+1}(\vec{\alpha}_{\mathcal{W}})\vec{q},$$

where $V^{a,b}(\vec{\alpha})$ is the Vandermonde matrix per Definition 3.1.1, and

$$\text{diag}(\vec{x}) = \begin{bmatrix} x_1 & 0 & 0 & \ldots & 0 \\ 0 & x_2 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & x_m \end{bmatrix}$$

for any $\vec{x}$ of length m. We may rewrite this as

$$\begin{bmatrix} \text{diag}(\vec{r}_{\mathcal{W}})V^{n-f,\hat{t}+1}(\vec{\alpha}_{\mathcal{W}}) & -V^{n-f,k+\hat{t}}(\vec{\alpha}_{\mathcal{W}}) \end{bmatrix} \begin{bmatrix} \vec{q} \\ \vec{s} \end{bmatrix} = \vec{0}.$$

Let

$$A = \begin{bmatrix} \text{diag}(\vec{r}_{\mathcal{W}})V^{n-f,\hat{t}+1}(\vec{\alpha}_{\mathcal{W}}) & -V^{n-f,k+\hat{t}}(\vec{\alpha}_{\mathcal{W}}) \end{bmatrix}. \tag{5.3}$$

Then decoding $\vec{r}$ reduces to finding a vector in the null space of the $(n-f) \times (k+2\hat{t}+1)$ matrix A.

5.2.5 The Rank of A and the Number of Errors

We have constructed A assuming there are $\hat{t}$ errors. Let t be the actual number of errors in $\vec{r}$ and $\delta = \hat{t} - t$. The rank of A, and thus the dimension of the null space of A, is determined by δ.

Theorem 5.2.3. *The dimension of the null space of A is $\delta + 1$.*

Proof. Let s, g be a solution to (5.2). Then, by Theorem 5.2.1 and Corollary 5.2.2, $g(x) \mid q(x) \mid s(x)$. Thus there exists a polynomial $a(x)$ such that $q(x) = g(x)a(x)$ and $s(x) = m(x)g(x)a(x)$. Since m and g are both unique, the solution space is completely characterized by choice of $a(x)$. $a(x)$ has degree at most δ and thus may be parameterized by its $\delta + 1$ coefficients. Thus the null space of A which is the solution space of (5.2) has dimension $\delta + 1$. □

Consider the simple case where $n-k-f$ is even and $t=\hat{t}$. Then A is an $(n-f)\times(n-f+1)$ matrix and its null space has dimension one. If we transform A into $\hat{A}$, the reduced-echelon form of A, via elementary row operations, then

$$\hat{A}=\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & x_0 \\ 0 & 1 & 0 & \dots & 0 & x_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & x_{n-f-1} \end{bmatrix}$$

and

$$\hat{A}\begin{bmatrix} \vec{g} \\ \vec{s} \end{bmatrix}=\vec{0}.$$

$\vec{q}=-(x_0,\dots,x_t)^T$ and $\vec{s}=-(x_{t+1},\dots,x_{n-f-1},1)^T$ is a solution to this linear system[1]. $\hat{A}$ and A share the same null space, so this is also a solution to 5.3.

If $n-k-f$ is even and $t<\hat{t}$, then the rank of the null space of A is $\delta+1$. Any solution in this null space will yield the same solution for $m(x)$; we proceed by choosing an element in the null space corresponding to $q(x)$ and $s(x)$ of lowest possible degree. In this case, the reduced-echelon form of A will be

$$\hat{A}=\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & x_{0,0} & \dots & x_{0,\delta} \\ 0 & 1 & 0 & \dots & 0 & x_{1,1} & \dots & x_{1,\delta} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & x_{n-f-\delta-1,1} & \dots & x_{n-f-\delta-1,\delta} \\ 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{bmatrix}=\begin{bmatrix} I^{n-f-\delta} & X \\ 0^{\delta,n-f-\delta} & 0^{\delta,\delta+1} \end{bmatrix},$$

where I^a is the $a\times a$ identity matrix, $0^{a,b}$ is the $a\times b$ matrix of zeros, and X is size $(n-f-\delta)\times(\delta+1)$. $\vec{q}=-(x_{0,0},\dots,x_{\hat{t},0})$ and $\vec{s}=-(x_{\hat{t}+1,0},\dots,x_{n-f-\delta-1,0},1,0,\dots 0)$ is a solution[2] in the null space of A corresponding to $q(x)$ and $s(x)$ of lowest possible degree. Notice that since $s(x)$ is degree $k+t-1$, $q(x)$ must be degree t. Thus we are guaranteed that $x_{t+1,0}=x_{t+2,0}=\dots=x_{\hat{t},0}=0$, and $\vec{q}=(x_{0,0},\dots,x_{t,0},0,\dots,0)$

[1] Note that the negative signs may be omitted when $\mathbb{F}$ has characteristic two such as $\mathbb{F}=\mathrm{GF}(2^l)$.

[2] Again, negative signs may be discarded in fields of characteristic two.

Exercise 5.2.4. Verify that if $\vec{s} = (x_{\hat{t}+1,0}, \ldots, x_{n-f-\delta-1,0}, 1, 0, \ldots 0)$ then $\deg s = k+t-1$.

Exercise 5.2.5. Show that if s, q is a solution to (5.2) and $\deg s = k+t-1$, then $\deg g = t$.

In the case where $n-k-f$ is odd, $\hat{t} = \frac{n-k-f-1}{2}$. We will have the same error correcting capability and assumption for $\hat{t}$ if we label one further location i as erased, adding i to $\mathcal{F}$, removing i from $\mathcal{W}$, and incrementing f. This reduces to the $n-k-f$ odd case to the previously discussed $n-k-f$ even case without compromising error correcting capability.

5.2.6 The Algorithm

Algorithm 5.2.1 below provides an implementation of the modified Welch-Berlekamp algorithm developed above. It decodes a received word $\vec{r}$ to the message $\vec{m}$ corresponding to the codeword $\vec{c}$ nearest to $\vec{r}$. It is assumed that the encoding was systematic and that the number of errors in $\vec{r}$ is within the error bounds. An outline of the algorithm is provided below with a discussion of its computational costs.

1. **Lines**: 4 - 10

   Purpose: Let $\hat{f} = f$. If $n-k-f$ is odd, add the last non-erased index in $\vec{r}$ to $\mathcal{F}$ and increment $\hat{f}$. $\hat{f}$ is the number "psychological" number of erasures. Let $\hat{t} = \frac{n-k-\hat{f}}{2}$, $\hat{n} = n - \hat{f}$, and $\mathcal{W} = \overline{\mathcal{F}}$.

   Computational complexity: No Galois field computation required.

2. **Lines**: 12 - 14

   Purpose: Construct the matrix A from (5.3).

   Computational complexity: The calls to VAND cost $\hat{n}(\hat{t}+1)$ and $\hat{n}(k+\hat{t})$. The

multiplication $\text{diag}(\vec{r}_{\mathcal{W}})B$ costs $\hat{n}(\hat{t}+1)$ operations. This yields a total of

$$\begin{aligned}\hat{n}(3\hat{t}+k+2) &= \hat{n}\left(3\left(\frac{\hat{n}-k}{2}\right)+k+2\right)\\ &= \frac{1}{2}\hat{n}(3\hat{n}-3k+2k+4)\\ &= \frac{1}{2}\hat{n}(3\hat{n}-k+4)\end{aligned}$$

operations required to construct A.

3. **Lines**: 16

   Purpose: Compute $\hat{A}$ the reduced row echelon form of A via elementary row operations using Gaussian elimination with pivoting.

   Computational complexity: A is size $\hat{n} \times (\hat{n}+1)$, so this call to RREF costs

$$\begin{aligned}(2\hat{n}-1)\hat{n}(\hat{n}+1) - \hat{n}(\hat{n}-1)\left(\hat{n}-\frac{1}{2}\right) &= \hat{n}\left(2\hat{n}^2+2\hat{n}-\hat{n}-1-\left(\hat{n}^2-\frac{3}{2}\hat{n}+\frac{1}{2}\right)\right)\\ &= \hat{n}\left(\hat{n}^2-\frac{1}{2}\hat{n}-\frac{1}{2}\right)\end{aligned}$$

   operations.

4. **Lines**: 17 - 20

   Purpose: Compute δ = number of rows in $\hat{A}$ - $\text{rank}(\hat{A})$, and $t = \hat{t} - \delta$. δ will be the number of rows at the bottom of A that are zero; we compute δ by counting these rows.

   Computational complexity: No Galois field arithmetic required.

5. **Lines**: 22 - 23

   Purpose: $\hat{A}$ is a $\hat{n} \times (\hat{n}+1)$ matrix. Set $\vec{q} = (\hat{A}_{0,\hat{n}-\delta}, \ldots, \hat{A}_{t,\hat{n}-\delta})$ and $\vec{s} = (\hat{A}_{\hat{t}+1,\hat{n}-\delta}, \ldots, \hat{A}_{\hat{n}-\delta-1,\hat{n}-\delta}, 1)$.

   Computational complexity: No computation required.

6. **Lines**: 24

   Purpose: Compute $\vec{\mu}$ as $\frac{s(x)}{q(x)}$.

Computational complexity: s has degree at most $k + \hat{t} - 1$ and g has degree at most $\hat{t}$, so the call to PolyDiv costs

$$\begin{aligned} 2\hat{t}(k + \hat{t} - 1 - \hat{t}) + (k + \hat{t} - 1) + \hat{t} + 1 &= 2\hat{t}(k-1) + k + 2\hat{t} \\ &= (\hat{n} - k)(k-1) + k + \hat{n} - k \\ &= k\hat{n} - \hat{n} - k^2 + k + \hat{n} \\ &= k(\hat{n} - k + 1) \end{aligned}$$

operations.

7. **Lines**: 27

   Purpose: Return $\vec{m}$, the decoded message as $m_j = \mu(\alpha_j)$ for $0 \leq j < k$.

   Computational complexity: The call to PolyEval costs at most $2k(k-1)$ operations.

The total operation count for Algorithm 5.2.1 is

$$\begin{aligned} &\frac{1}{2}\hat{n}(3\hat{n} - k + 4) + \hat{n}\left(\hat{n}^2 - \frac{1}{2}\hat{n} - \frac{1}{2}\right) + k(\hat{n} - k + 1) + 2k(k-1) \\ &= \frac{3}{2}\hat{n}^2 + 2\hat{n} + \hat{n}^3 - \frac{1}{2}\hat{n}^2 - \frac{1}{2}\hat{n} + k\left(\frac{1}{2}\hat{n} - k + 1 + 2k - 2\right) \\ &= \frac{3}{2}\hat{n}^2 + 2\hat{n} + \hat{n}^3 - \frac{1}{2}\hat{n}^2 - \frac{1}{2}\hat{n} + k\left(\frac{1}{2}\hat{n} + k - 1\right) \\ &= \hat{n}^3 + \hat{n}^2 + \frac{1}{2}\hat{n}\,(k+3) + k\,(k-1) \\ &= \hat{n}^3 + \hat{n}^2 + \frac{1}{2}\hat{n}k + k^2 + \Theta(\hat{n} + k). \end{aligned}$$

Exercise 5.2.6. How must Algorithm 5.2.1 be modified when the encoding is non-systematic?

Example 5.2.7. (Continued from example 4.2.1) Let $l = 8$, $k = 5$, $n = 8$, $\vec{\alpha} = (0, 1, 2, 3, 4, 5, 6, 7)^T$, $\vec{r} = (233, 117, 0, 7, 18, 166, 14, 135)^T$, and assume systematic encoding and that there are no erasures. As we showed in example 4.2.1, $\vec{r}$ has some errors. We will now

Algorithm 5.2.1 Modified Welch-Berlekamp

Dependencies: VAND defined in Algorithm 3.1.1, RREF defined in Algorithm A.4.3, POLYDIV defined in Algorithm A.3.7.

procedure MWB($\vec{\alpha}, \vec{r}, \mathcal{F}, F$) ⋆*Recover the message $\vec{m}$ from $\vec{r}$*
  $n, k \leftarrow$size(F) ⋆*F is the generating matrix for the code and is size $n \times k$*
  $\hat{f} \leftarrow$length($\mathcal{F}$)
  if $n - k - f$ odd **then**
    Add the last non-erased index in $\vec{r}$ to $\mathcal{F}$
    $\hat{f} \leftarrow \hat{f} + 1$
  end if
  $\hat{t} \leftarrow \frac{n-k-\hat{f}}{2}$
  $\hat{n} \leftarrow n - \hat{f}$
  $\mathcal{W} \leftarrow \overline{\mathcal{F}}$
  ⋆*Construct the matrix A from (5.3)*
  $B \leftarrow \text{VAND}(\hat{n}, \hat{t}+1, \vec{\alpha}_{\mathcal{W}})$ ⋆$B = V^{\hat{n},\hat{t}+1}(\vec{\alpha}_{\mathcal{W}})$
  $C \leftarrow \text{VAND}(\hat{n}, k+\hat{t}, \vec{\alpha}_{\mathcal{W}})$ ⋆$C = V^{\hat{n},k+\hat{t}}(\vec{\alpha}_{\mathcal{W}})$
  $A \leftarrow [\text{diag}(\vec{r}_{\mathcal{W}})B \quad -C]$ ⋆*Construct the $\hat{n} \times (\hat{n}+1)$ matrix A*

  $\hat{A} \leftarrow \text{RREF}(A)$ ⋆*Compute $\hat{A}$ the row reduced echelon form of A*
  $\delta \leftarrow 0$ ⋆*determine δ as the number of rows at the end of A that are zero*
  while $A_{\hat{n}-\delta,\hat{n}-\delta} = 0$ **do**
    $\delta \leftarrow \delta + 1$
  end while

  $\vec{q} \leftarrow (\hat{A}_{0,\hat{n}-\delta}, \ldots, \hat{A}_{\hat{t},\hat{n}-\delta})$
  $\vec{s} \leftarrow (\hat{A}_{\hat{t}+1,\hat{n}-\delta}, \ldots, \hat{A}_{\hat{n}-\delta-1,\hat{n}-\delta}, 1)$.
  $\vec{\mu}, \vec{R} \leftarrow \text{POLYDIV}(\vec{s}, \vec{q})$
  Require: $\vec{R} = 0$ ⋆*The remainder from division ought to be zero*

  $\vec{m} \leftarrow \text{POLYEVAL}(\vec{\mu}, \vec{\alpha}_{0:k-1})$ ⋆*Recover $\vec{m}$*
  return $\vec{m}$
end procedure

Computational complexity: $\hat{n}^3 + \hat{n}^2 + \frac{1}{2}\hat{n}\,(k+3) + k(k-1)$ operations required.

try to recover the original message from $\vec{r}$ using the modified Welch-Berlekamp algorithm (5.2.1).

1. $n-k-f=3$, so we set $\mathcal{F}=\{7\}$ and $f=1$. $\hat{t}=1$, $\vec{r}_{\mathcal{W}}=(233,117,0,7,18,166,14)^T$ and $\vec{\alpha}_{\mathcal{W}}=(0,1,2,3,4,5,6)^T$.

2. $$A=\begin{bmatrix} 233 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 117 & 117 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 4 & 8 & 16 & 32 \\ 7 & 9 & 1 & 3 & 5 & 15 & 17 & 51 \\ 18 & 72 & 1 & 4 & 16 & 64 & 29 & 116 \\ 166 & 4 & 1 & 5 & 17 & 85 & 28 & 108 \\ 14 & 36 & 1 & 6 & 20 & 120 & 13 & 46 \end{bmatrix}.$$

3. $$\hat{A}=\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 95 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 95 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 80 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 15 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 35 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 148 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 233 \end{bmatrix}.$$

4. $\delta=0$, $t=1$.

5. $\vec{q}=(95,95)^T$, $\vec{s}=(80,15,35,148,233,1)^T$.

6. $\vec{\mu}=(233,1,10,112,65)^T$.

7. $\vec{m}=(233,211,0,7,18)^T$.

Referring back to example 4.2.1, we see that this did indeed produce the correct decoding.

Chapter 6
The Key Equation

In order to improve on the computational complexity of the modified Welch-Berlekamp algorithm presented in Chapter 5.2, we must first derive a "key equation" that is amenable to efficient solution by available algorithms. We will find that this key equation is similar in structure to (5.2), but involves polynomials of lower degree. In this sense, it is a "smaller" problem. In Chapter 7, we will then develop a method for solving this type of equation substantially faster than the linear algebra techniques discussed previously. We consider the set up of this key equation to be "phase 1" of an error correction algorithm. The derivation of this key equation follows [18].

6.1 Deriviation

Let $\vec{c}$ be the original codeword, $\vec{m}$ be the original message, and assume the message was encoded via $\vec{c} = \mu(\vec{\alpha})$. $\vec{c}$, $\vec{m}$, and $\mu(x)$ are unknown and we would like to find them. Let $\mathcal{I}$ be an arbitrary, fixed set of k indices, $\mathcal{I} \subset \{0, 1, \ldots, n-1\}$, with $|\mathcal{I}| = k$. We may recover a message polynomial $\hat{m}(x)$ from the characters in the received word $\vec{r}$ at the indices $\mathcal{I}$ as in Section 2.2. Probably, we will have $\mu(x) \neq \hat{m}(x)$ unless we happen to exclude all indices from $\mathcal{I}$ that are in error. $\hat{m}(x)$ is uniquely defined by

$$\hat{m}(\alpha_i) = r_i \quad \text{for} \quad i \in \mathcal{I}.$$

For $i \in \mathcal{I}$, define

$$\phi_i(x) = \prod_{j \in \mathcal{I}, j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j}.$$

Then $\deg(\phi_i) = k-1$, and $\phi_i(\alpha_j) = \delta_{i,j}$ for all $i, j \in \mathcal{I}$. Define $G_{\mathcal{I}}(x) = \prod_{j\in\mathcal{I}}(x-\alpha_j)$, and let $G'_{\mathcal{I}}(x)$ be the formal derivative:

$$G'_{\mathcal{I}}(x) = \sum_{i\in\mathcal{I}} \prod_{j\neq i, j\in\mathcal{I}} (x-\alpha_j).$$

Then

$$\phi_i(x) = \frac{G_{\mathcal{I}}(x)}{(x-\alpha_i)G'_{\mathcal{I}}(\alpha_i)}. \tag{6.1}$$

Exercise 6.1.1. Prove equation (6.1).

Let $\vec{e}$ be the vector of errors in the received word: $\vec{r} = \vec{c} + \vec{e}$ and $r_i = c_i + e_i = \mu(\alpha_i) + e_i$. Then Lagrange interpolations states that

$$\begin{aligned}
\hat{m}(x) &= \sum_{i\in\mathcal{I}} r_i\phi_i(x) \\
&= \sum_{i\in\mathcal{I}} (\mu(\alpha_i) + e_i)\phi_i(x) \\
&= \sum_{i\in\mathcal{I}} \mu(\alpha_i)\phi_i(x) + \sum_{i\in\mathcal{I}} e_i\phi_i(x) \\
\hat{m}(x) &= \mu(x) + \sum_{i\in\mathcal{I}} e_i\phi_i(x).
\end{aligned}$$

Exercise 6.1.2. Show that $\mu(x) = \sum_{i\in\mathcal{I}} \mu(\alpha_i)\phi_i(x)$.

Define σ_j as the difference between the received values and the interpolated values at location j: $\sigma_j = r_j - \hat{m}(\alpha_j)$. Notice that for $j \in \mathcal{I}$, $\sigma_j = 0$ and for $j \in \overline{\mathcal{I}}, \sigma_j$ may be easily computed from $\vec{r}$ and $\vec{\alpha}$. On the other hand,

$$\begin{aligned}
\sigma_j &= r_j - \hat{m}(\alpha_j) \\
&= \mu(\alpha_j) + e_j - \mu(\alpha_j) - \sum_{i\in\mathcal{I}} e_i\phi_i(\alpha_j) \\
&= e_j - \sum_{i\in\mathcal{I}} e_i\phi_i(\alpha_j).
\end{aligned}$$

The goal is find the errors given the "syndromes" σ_j. Using (6.1), we may rewrite this further:

$$\sigma_j = e_j - \sum_{i\in\mathcal{I}} \frac{e_i G_{\mathcal{I}}(\alpha_j)}{(\alpha_j-\alpha_i)G'_{\mathcal{I}}(\alpha_i)}. \tag{6.2}$$

For $j \in \overline{\mathcal{I}}$, define

$$\sigma_j^* = \frac{\sigma_j}{G_{\mathcal{I}}(\alpha_j)}, \quad e_j^* = \frac{e_j}{G_{\mathcal{I}}(\alpha_j)}, \quad \text{and} \quad e_j^\dagger = \frac{e_j}{G'(\alpha_j)}.$$

Then

$$\sigma_j^* = e_j^* - \sum_{i \in \mathcal{I}} \frac{e_i^\dagger}{(\alpha_j - \alpha_i)}. \tag{6.3}$$

Let $\mathcal{E}$ bet the set of indices with errors, $\mathcal{E} = \{i : e_i \neq 0\}$, and define

$$g_{\mathcal{I}}(x) = \prod_{i \in \mathcal{I} \cap \mathcal{E}} (x - \alpha_i), \quad g_{\overline{\mathcal{I}}}(x) = \prod_{i \in \mathcal{I} \cap \overline{\mathcal{E}}} (x - \alpha_i),$$

and

$$g(x) = g_{\mathcal{I}}(x) g_{\overline{\mathcal{I}}}(x) = \prod_{i \in \mathcal{E}} (x - \alpha_i).$$

$g(x)$ is the error locator polynomial; α_i is a root of $g(x)$ if and only if $e_i \neq 0$. Notice that

$$\begin{aligned} \sum_{i \in \mathcal{I}} \frac{e_i^\dagger}{(\alpha_j - \alpha_i)} &= \sum_{i \in \mathcal{I} \cap \mathcal{E}} \frac{e_i^\dagger}{(\alpha_j - \alpha_i)} \\ &= \frac{\sum_{i \in \mathcal{I} \cap \mathcal{E}} e_i^\dagger \prod_{h \neq i, h \in \mathcal{I} \cap \mathcal{E}} (\alpha_j - \alpha_h)}{\prod_{i \in I \cap \mathcal{E}} (\alpha_j - \alpha_i)} \\ \sum_{i \in \mathcal{I}} \frac{e_i^\dagger}{(\alpha_j - \alpha_i)} &= \frac{A(\alpha_j)}{g_{\mathcal{I}}(\alpha_j)} \end{aligned} \tag{6.4}$$

where we take the above equation as a definition of $A(x)$, *i.e.*

$$A(x) = \sum_{i \in \mathcal{I} \cap \mathcal{E}} e_i^\dagger \prod_{h \neq i, h \in \mathcal{I} \cap \mathcal{E}} (x - \alpha_h),$$

and $\deg(A) \leq |\mathcal{I} \cap \mathcal{E}| - 1$. We may rewrite (6.3) in terms of these polynomials:

$$\sigma_j^* = e_j^* - \frac{A(\alpha_j)}{g_{\mathcal{I}}(\alpha_j)} \tag{6.5}$$

and therefore

$$\sigma_j^* g(\alpha_j) = e_j^* g(\alpha_j) - A(\alpha_j) g_{\overline{\mathcal{I}}}(x_j).$$

Notice that for all $j \in \overline{\mathcal{I}}$, either $e_j = 0$ or $g(\alpha_j) = 0$ and thus $e_j^* g(\alpha_j) = 0$. Therefore,

$$\sigma_j^* g(\alpha_j) = -A(\alpha_j) g_{\overline{\mathcal{I}}}(\alpha_j), \quad \forall j \in \overline{\mathcal{I}}.$$

Let $s(x) = -A(x) g_{\overline{\mathcal{I}}}(x)$; $\deg(s) \leq |\mathcal{I} \cap \mathcal{E}| - 1 + |\overline{\mathcal{I}} \cap \mathcal{E}| = |\mathcal{E}| - 1$. So

$$\sigma_j^* g(\alpha_j) = s(\alpha_j), \quad \forall j \in \overline{\mathcal{I}}. \tag{6.6}$$

Notice the structural similarities between (6.6) and (5.2). $\deg(g) = t$ and $\deg(s) \leq t-1$, so (6.6) consists of $2t+1 \leq n-k$ unknowns in the coefficients of s and g, and $n-k$ equations. This system may be solved via linear algebra in a manner similar to Section 5.2.4 and Algorithm 5.2.1 in $\Theta((n-k)^3) = \Theta(\hat{t}^3)$ operations[1]. However, a solution may be obtained in $\Theta((n-k)^2)$ operations using the Welch-Berlekamp algorithm developed in Chapter 7.

Let $\Sigma(x)$ be the unique polynomial of degree $\leq r-1$ such that $\Sigma(\alpha_j) = \sigma_j^*$ for all $j \in \overline{\mathcal{I}}$. Then

$$\Sigma(\alpha_j) g(\alpha_j) = s(\alpha_j), \quad \forall j \in \overline{\mathcal{I}},$$

and, equivalently,

$$s(x) \equiv \Sigma(x) g(x) \mod G_{\overline{\mathcal{I}}}(x). \tag{6.7}$$

The coefficients of Σ and $G_{\overline{\mathcal{I}}}$ are known, whereas the coefficients of g and s are unknown. $\deg(g) = t$ and $\deg(s) = t-1$. The goal of decoding is thus reduced to finding a solution s, g to this equation. Given such a solution, one can easily find the roots of g which gives the locations of the errors and solves the decoding problem. (6.7) is referred to as the *key equation* for decoding Reed-Solomon codes.

In general, the value of t will not be known *a priori*, so the above formulation of the key equation is of limited use. Instead, we use $\hat{t} = \lfloor \frac{r}{2} \rfloor$ as abound on the degrees of g and s and state the key equation as follows:

[1] See Definition A.2.1 for more on Θ notation.

Key Equation KE6.1.1

Given $\Sigma(x)$ and $G_{\overline{\mathcal{I}}}(x) = \prod_{j\in\overline{\mathcal{I}}}(x-\alpha_j)$ with $\deg(\Sigma) < \deg(G_{\overline{\mathcal{I}}}) = r$, find $s(x), g(x) \in \mathbb{F}[x]$ such that

- $s(x) = \Sigma(x)g(x) \mod G_{\overline{\mathcal{I}}}(x)$,
- $\deg(s) < \deg(g) \leq \hat{t} = \lfloor \frac{r}{2} \rfloor$,
- g is monic and of minimum degree.

It is clear from the derivation of the key equation, that the error locator polynomial g and s as defined in the derivation satisfy the relationship $s(x) = \Sigma(x)g(x) \mod G_{\overline{\mathcal{I}}}(x)$ with $\deg(s) < \deg(g) \leq \hat{t}$ and g monic. However, it is not clear that this is the solution of minimum degree, or that it is unique. The following lemmas and theorem establish that fact and as well as some other useful properties related to these polynomials.

Lemma 6.1.3. *For $\sigma_j, e_j, \mathcal{I}, \mathcal{E}, G_{\mathcal{I}\setminus\mathcal{E}}(x)$, and $A(x)$ as defined above,*

$$e_j = \sigma_j - G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j), \quad \forall j.$$

Proof. By (6.4),

$$\frac{A(\alpha_j)}{g_{\mathcal{I}}(\alpha_j)} = \sum_{i\in\mathcal{I}} \frac{e_i^\dagger}{(\alpha_j - \alpha_i)}, \quad \forall j \in \overline{I}.$$

Therefore, $\forall j$

$$\frac{G_{\mathcal{I}}(\alpha_j)A(\alpha_j)}{g_{\mathcal{I}}(\alpha_j)} = \sum_{i\in\mathcal{I}} \frac{e_i^\dagger G_{\mathcal{I}}(\alpha_j)}{(\alpha_j - \alpha_i)}$$

$$G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j) = \sum_{i\in\mathcal{I}} \frac{e_i G_{\mathcal{I}}(\alpha_j)}{(\alpha_j - \alpha_i)G'_{\mathcal{I}}(\alpha_i)}.$$

Substituting into (6.2), we see that

$$\sigma_j = e_j - G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j), \quad \forall j$$

which proves the desired result. □

Lemma 6.1.4. *For $g_{\mathcal{I}}, A$ as defined above,* $\gcd(A, g_{\mathcal{I}}) = 1$.

Proof. We seek to show that $A(\alpha_i) \neq 0$ for $i \in \mathcal{I} \cap \mathcal{E}$. In search of contradiction, assume that there exists some $i \in \mathcal{I} \cap \mathcal{E}$ such that $A(\alpha_i) \neq 0$. Since $i \in \mathcal{E}$, $e_i \neq 0$. Since $i \in \mathcal{I}$, $\sigma_i = 0$. From Lemma 6.1.3,

$$\begin{aligned} e_j &= \sigma_j - G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j) \\ e_j &= 0, \end{aligned}$$

providing the desired contradiction. □

Theorem 6.1.5. *Let $\Sigma(x)$ and $G_{\overline{I}}(x)$ be as defined in the above derivation, and assume that $\vec{c} \neq \hat{m}(\vec{\alpha})$. Then the unique solution to the key equation KE6.1.1 is s, g, for s and g also as defined above. In particular, g is the error locator polynomial and Lemma 6.1.3 holds for $A(x) = -s(x)/g_{\overline{I}}(x)$.*

Proof. Let g be the error locator polynomial and $s(x) = A(x)g_{\overline{I}}(x)$. Let a, b be another solution to the key equation KE6.1.1. Then we must have $\deg b \leq \deg g$, since b is a solution of minimal degree by assumption, and s, g also satisfies the key equation. We seek to show that $a = s$ and $b = g$ therefore the unique solution to the key equation is the desired solution.

Claim 6.1.5.1. *$A|a$ and $g_{\mathcal{I}}|b$ and thus there exists $\hat{a}, \hat{b} \in \mathbb{F}[x]$ such that $a = A\hat{a}$, $b = g_{\mathcal{I}}\hat{b}$.*

Proof. For all $i \in \overline{\mathcal{I}}$,

$$a(\alpha_i) = \sigma_i^* b(\alpha_i).$$

Using (6.5) we may rewrite this as

$$\begin{aligned} a(\alpha_i) &= \left(e_i^* - \frac{A(\alpha_i)}{g_{\mathcal{I}}(\alpha_i)}\right) b(\alpha_i) \\ g_{\mathcal{I}}(\alpha_i)a(\alpha_i) &= \left(g_{\mathcal{I}}(\alpha_i)e_i^* - A(\alpha_i)\right) b(\alpha_i). \end{aligned}$$

For $i \in \overline{\mathcal{I}} \backslash \mathcal{E}$, $e_i^* = 0$ and so

$$g_{\mathcal{I}}(\alpha_i) a(\alpha_i) = -A(\alpha_i) b(\alpha_i), \quad \forall i \in \overline{\mathcal{I}} \backslash \mathcal{E}$$

Let $t_0 = |\mathcal{I} \cap \mathcal{E}|$ and $t_1 = |\overline{\mathcal{I}} \cap \mathcal{E}|$, so $t_0 + t_1 = t$ and $t + t_0 \leq r - t_1$ since $2t \leq r$. Hence, $\deg g_{\mathcal{I}} a = \deg g_{\mathcal{I}} + \deg a < t_0 + t$ and $\deg Ab = \deg A + \deg b < t_0 + t$. $g_{\mathcal{I}} a$ and Ab are both degree at $t_0 + t - 1$ polynomials that agree on $r - t_1 > t_0 + t - 1$ points, so

$$g_{\mathcal{I}}(x) a(x) = -A(x) b(x).$$

By Lemma 6.1.4, $\gcd(A, g_{\mathcal{I}}) = 1$ so $A | a$ and $g_{\mathcal{I}} | b$ thus proving the claim. ✠

Claim 6.1.5.2. $\hat{a} = -\hat{b}$.

Proof. From (6.5), for $i \in \overline{\mathcal{I}} \backslash \mathcal{E}$, $e_i^* = 0$ and so

$$g_{\mathcal{I}}(\alpha_i) \sigma_i^* = -A(\alpha_i).$$

As well, since a, b satisfies the key equation, $\forall i \in \overline{\mathcal{I}} \backslash \mathcal{E}$

$$A(\alpha_i) \hat{a}(\alpha_i) = \sigma_i^* g_{\mathcal{I}}(\alpha_i) \hat{b}(\alpha_i)$$
$$A(\alpha_i) \hat{a}(\alpha_i) = -A(\alpha_i) \hat{b}(\alpha_i).$$

If $\sigma_i \neq 0$ for $i \in \overline{\mathcal{I}} \backslash \mathcal{E}$, it follows that $A(\alpha_i) \neq 0$, and we may divide by $A(\alpha_i)$ in the above equation. Let $S = \{i : \sigma_i = 0\}$. Then

$$\hat{a}(\alpha_i) = -\hat{b}(\alpha_i), \quad \forall i \in (\overline{\mathcal{I}} \backslash \mathcal{E}) \backslash S = \overline{\mathcal{I}} \cap \overline{\mathcal{E}} \cap \overline{\mathcal{S}}.$$

$\deg \hat{a} < t$ and $\deg \hat{b} < t$. We seek to show that $|(\overline{\mathcal{I}} \backslash \mathcal{E}) \backslash S| \geq t$ as this would imply that $\hat{a}$ and $-\hat{b}$ are polynomials of degree less than t that agree on at least t points and thus $\hat{a}(x) = -\hat{b}(x)$. $|(\overline{\mathcal{I}} \backslash \mathcal{E}) \backslash S| = |\overline{\mathcal{I}} \cap \overline{\mathcal{E}} \cap \overline{\mathcal{S}}| = r - t_1 - |\overline{I} \cap \overline{\mathcal{E}} \cap S|$, where $t_0 = |\mathcal{I} \cap \mathcal{E}|$ and $t_1 = |\overline{\mathcal{I}} \cap \mathcal{E}|$ and $t_0 + t_1 = t$. We therefore seek an upper bound on $|\overline{I} \cap \overline{\mathcal{E}} \cap S|$.

For $i \notin \mathcal{E}$, $\sigma_i = 0$ implies that $r_j = \hat{m}(\alpha_j)$ and since $e_j = 0$, $c_j = \hat{m}(\alpha_j)$. Therefore, $|\overline{\mathcal{E}} \cap S| \leq n - d(\vec{c}, \hat{m}(\vec{\alpha})) \leq n - (r+1)$ due to the distance bound on Reed-Solomon codes (Theorem 4.3.6) and by the assumption that $\vec{c} \neq \hat{m}(\vec{\alpha})$. As well, $\mathcal{I} \cap \overline{\mathcal{E}} \cap S = \mathcal{I} \cap \overline{\mathcal{E}}$, since $\sigma_i = 0$ for all $i \in \mathcal{I}$, so $|\mathcal{I} \cap \overline{\mathcal{E}} \cap S| = k - t_0$. Since $\overline{\mathcal{E}} \cap S = (\mathcal{I} \cap \overline{\mathcal{E}} \cap S) \cup (\overline{\mathcal{I}} \cap \overline{\mathcal{E}} \cap S)$ and $\mathcal{I} \cap \overline{\mathcal{E}} \cap S$ and $\overline{\mathcal{I}} \cap \overline{\mathcal{E}} \cap S$ are disjoint,

$$\begin{aligned} |\overline{I} \cap \overline{\mathcal{E}} \cap S| &= |\overline{\mathcal{E}} \cap S| - |\mathcal{I} \cap \overline{\mathcal{E}} \cap S| \\ &\leq n - (r+1) - (k - t_0) = t_0 - 1 \end{aligned}$$

Therefore $|(\overline{\mathcal{I}} \backslash \mathcal{E}) \backslash S| = r - t_1 - |\overline{I} \cap \overline{\mathcal{E}} \cap S| \geq r - t_1 - t_0 + 1 = r - t + 1 \geq r - \hat{t} + 1 = \hat{t} + 1$ and $|(\overline{\mathcal{I}} \backslash \mathcal{E}) \backslash S| > \hat{t} > t$ thus proving the claim. ✠

Claim 6.1.5.3. $g_{\overline{\mathcal{I}}} | \hat{b}$.

Proof. As shown above,

$$g_{\mathcal{I}}(\alpha_i) a(\alpha_i) = \left(g_{\mathcal{I}}(\alpha_i) e_i^* - A(\alpha_i)\right) b(\alpha_i), \quad \forall i \in \overline{\mathcal{I}}$$

and so $\forall i \in \overline{\mathcal{I}}$

$$\begin{aligned} g_{\mathcal{I}}(\alpha_i) A(\alpha_i) \hat{a}(\alpha_i) &= \left(g_{\mathcal{I}}(\alpha_i) e_i^* - A(\alpha_i)\right) g_{\mathcal{I}}(\alpha_i) \hat{b}(\alpha_i) \\ e_i^* (g_{\mathcal{I}}(\alpha_i))^2 \hat{b}(\alpha_i) &= g_{\mathcal{I}}(\alpha_i) A(\alpha_i) \hat{a}(\alpha_i) + g_{\mathcal{I}}(\alpha_i) A(\alpha_i) \hat{b}(\alpha_i) \\ &= -g_{\mathcal{I}}(\alpha_i) A(\alpha_i) \hat{b}(\alpha_i) + g_{\mathcal{I}}(\alpha_i) A(\alpha_i) \hat{b}(\alpha_i) \\ &= 0. \end{aligned}$$

For $i \in \overline{\mathcal{I}} \cap \mathcal{E}$, $e_i^* \neq 0$ and $g_{\mathcal{I}}(\alpha_i) \neq 0$, so we must have $\hat{b}(\alpha_i) = 0$ and thus $g_{\overline{\mathcal{I}}} | \hat{b}$. ✠

From the above claims, we see that for any a, b that solve the key equation except perhaps for b being of minimal degree and monic, there exists $c(x) \in \mathbb{F}[x]$ such that $a(x) = c(x)s(x)$ and $b(x) = c(x)g(x)$. If b is also of minimal degree and monic, then it follows that $c(x) = 1$.

Then $b(x) = g(x)$ and b is the error locator polynomial, and $a = s(x)$ and $A(x) = \frac{-s(x)}{g_{\overline{\mathcal{I}}}(x)}$, and the solution to the key equation is unique. □

6.2 Recovering the Codeword From a Solution to the Key Equation

Given a solution s, g to the key equation, one may find the roots of g by computing $g(\alpha_i)$ for $0 \leq i < n$. This yields the error locations; we may then recover the codeword treating the error location as erasures. This is a first, obvious approach to phase 3 of an error correcting algorithm. However, there is another way to recover the codeword by exploiting s.

Recall that $s(x) = -A(x)g_{\overline{I}}$. Given g, we first compute $\mathcal{E} = \{i : g(\alpha_i) = 0\}$. We may then easily compute $g_{\overline{\mathcal{I}}}$, and thus find $A(x) = \frac{-s(x)}{g_{\overline{\mathcal{I}}}(x)}$. From Lemma 6.1.3 we know that

$$\begin{aligned} e_j &= \sigma_j + G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j) \\ &= r_j - \hat{m}(\alpha_j) + G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j) \\ r_j - e_j &= \hat{m}(\alpha_j) - G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j) \\ c_j &= \hat{m}(\alpha_j) - G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j)A(\alpha_j). \end{aligned} \tag{6.8}$$

We may use this equation to directly recover the values of the codeword at error locations. This yields an alternate version of phase 3 of an error correcting algorithm.

Chapter 7
The Welch-Berlekamp Algorithm

We now seek an efficient algorithm to solve the key equation KE6.1.1. This algorithm will correspond to phase 2 or a decoding algorithm. In this chapter we present the Welch-Berlekamp algorithm which may be derived using techniques from algebraic geometry, particularly the theory of Gröbner bases. This algorithm was first suggested by Welch and Berlekamp in [19]; the derivation presented here follows [8] and [12]. The reader is strongly encouraged to read Appendix B on Modules and Gröbner bases before continuing with this chapter.

7.1 The Solution Module

In Section 6.1 we derived the key equation KE6.1.1. Let $\mathcal{I}$, $\overline{\mathcal{I}}$ be as in Chapter 6. $\overline{\mathcal{I}}$ is a set of r indices from $\{0, \ldots, n\}$. If we consider $\overline{\mathcal{I}}$ as an ordered list, then $\overline{\mathcal{I}} = \{i_0, \ldots, i_{r-1}\}$ where $i_u < i_v$ iff $u < v$. We may define $\overline{\mathcal{I}}_j$ as the first j elements of $\overline{\mathcal{I}}$: $\overline{\mathcal{I}}_j = \{i_0, \ldots, i_{j-1}\}$. From this, we may define the sequence of modules

$$M_j = \{(a, b) : a(x) \equiv \Sigma(x)b(x) \mod G_j(x)\} \subset \mathbb{F}[x] \times \mathbb{F}[x], \qquad 0 \leq j \leq r,$$

where $G_0(x) = 1$ and

$$G_j(x) = \prod_{i \in \overline{\mathcal{I}}_j} (x - \alpha_i) = \prod_{a=0}^{j-1} (x - \alpha_{i_a}).$$

Notice that $G_{j+1} = (x - \alpha_{i_j})G_j$. $M = M_r$ is the solution module; the solution we seek to the key equation, (s, g), is an element of M. Define $\Sigma_j(x) := \Sigma(x) \mod G_j(x)$. Then we may equivalently define M_j by

$$M_j = \{(a, b) : a(x) \equiv \Sigma_j(x)b(x) \mod G_j(x)\}.$$

Exercise 7.1.1. Show that M_j is a module per Definition B.1.1.

Given $\Sigma(x)$ and $G_{\overline{\mathcal{I}}}(X)$, KE6.1.1 asks us to find (s, g) such that the polynomials $s(x)$ and $\Sigma(x)g(x)$ agree at the r points $\vec{\alpha}_{\overline{\mathcal{I}}}$, subject to some degree constraints on s and g. The solution module M_j is the set of all polynomials (a, b) such that $a(x)$ and $\Sigma(x)b(x)$ agree at j prescribed points. We may think of M_j as the set of candidate solutions (a, b) to the key equation where equality of $a(x)$ and $\Sigma(x)b(x)$ is assured at j points and without restrictions on the degrees of a and b. The essence of the Welch-Berlekamp algorithm is to first characterize possible solutions that agree at no points (the elements of M_0), then use this to characterize possible solutions that agree at one point (elements of M_1), then two points (elements of M_2), until we arrive at a characterization of solutions that agree at r points (elements of M_r). We will then use this characterization to identify the unique desired solution to the full key equation. The remainder of this section establishes some useful properties of the polynomials G_j and the solution modules M_j which will be needed later. In 7.2 we discuss an appropriate characterization of the modules M_j, namely that of Gröbner bases. In 7.3 discuss how to use a characterization (Gröbner base) of M_j to find a characterization of M_{j+1}. In 7.4, we consolidate this discussion into an algorithm for solving the key equation.

Theorem 7.1.2. *If $f \in \mathbb{F}[x]$ then there exists $p \in \mathbb{F}[x]$ and $\gamma_0, \ldots, \gamma_{r-1} \in \mathbb{F}$ such that*

$$f(x) = p(x)G_{\overline{\mathcal{I}}}(x) + \sum_{j=0}^{r-1} \gamma_j G_j(x).$$

Proof. We note that $\deg(G_i) = i$.

Let $h \in \mathbb{F}[x]$ be the remainder of division of f by $G_{\overline{\mathcal{I}}}$. Thus $f(x) = p(x)G_{\overline{\mathcal{I}}}(x) + h(x)$. By definition, $\deg(h) < \deg(G_{\overline{\mathcal{I}}}) = r$. We need to show that

$$h(x) = \sum_{j=0}^{r-1} \gamma_j G_j(x).$$

Formally, we proceed by induction with respect to r. The existence of the above representation is obtained by a greedy algorithm which divides h by G_{r-1}, where $k = \deg(h)$.

The quotient is a constant γ_{r-1} (which could be 0) and the remainder is a polynomial $\tilde{h}$ of degree $< r-1$. Thus, by the induction hypothesis

$$\tilde{h}(x) = \sum_{j=0}^{r-2} \gamma_j G_j(x).$$

Hence, as claimed,

$$h(x) = \gamma_{r-1} G_{r-1}(x) + \tilde{h}(x) = \sum_{j=0}^{r-1} \gamma_j G_j(x).$$

□

Remark 7.1.3. Theorem 7.1.2 can be understood in terms of the well-known divided difference scheme for finding the Lagrange interpolating polynomial. Here we briefly summarize this approach.

For every polynomial $h \in \mathbb{F}[x]$ of degree r and arbitrary $\beta_1, \beta_2, \ldots, \beta_r \in \mathbb{F}$ we have a finite difference representation:

$$\begin{aligned} f(x) = f(\beta_1) + f[\beta_1, \beta_2](x - \beta_1) + \cdots \\ + f[\beta_1, \beta_2 \ldots, \beta_r](x - \beta_1)(x - \beta_2) \cdots (x - \beta_{r-1}). \end{aligned} \tag{7.1}$$

The coefficients $f[\beta_1, \beta_2, \ldots, \beta_j]$ are known as the *divided differences* and are computed according to these rules:

$$\begin{aligned} f[\beta_j] &= f(\beta_j), \\ f[\beta_j, \beta_{j+1}, \ldots, \beta_{j+u}] &= \frac{f[\beta_{j+1}, \beta_{j+2}, \ldots, \beta_{j+u}] - f[\beta_j, \beta_{j+1}, \ldots, \beta_{j+u-1}]}{\beta_{j+u} - \beta_j}. \end{aligned}$$

If f is an arbitrary function then the right-hand side yields an alternative representation of the Lagrange interpolating polynomial, with computational complexity $O(r^2)$.

Corollary 7.1.4. *Let $(a, b) \in M_j$ for some $j < r$. Define $f \in \mathbb{F}[x]$ by $\Sigma b - a \equiv f G_j \mod G_{j+1}$. Then $f \in \mathbb{F}$,* i.e. $\deg f \leq 0$. *Moreover, $f = (\Sigma(\alpha_{i_j}) b(\alpha_{i_j}) - a(\alpha_{i_j}))/G_j(\alpha_{i_j})$.*

Proof. By Theorem 7.1.2, there exists $p \in \mathbb{F}[x]$ and $\gamma_0, \ldots, \gamma_{r-1} \in \mathbb{F}$ such that

$$\Sigma b - a = p(x)G_{\overline{\mathcal{I}}}(x) + \sum_{i=0}^{r-1} \gamma_i G_i(x).$$

Since $(a, b) \in M_j$, $\Sigma b - a = 0 \mod G_j$ and

$$\Sigma b - a = p(x)G_{\overline{\mathcal{I}}}(x) + \sum_{i=j}^{r-1} \gamma_i G_i(x).$$

Thus

$$\Sigma b - a \equiv \gamma_j G(x) \mod G_{j+1},$$

and $f = \gamma_j \in \mathbb{F}$.

As well,

$$\begin{aligned} \Sigma(\alpha_{i_j})b(\alpha_{i_j}) - a(\alpha_{i_j}) &= p(\alpha_{i_j})G_{\overline{\mathcal{I}}}(\alpha_{i_j}) + \sum_{i=j}^{r-1} \gamma_i G_i(\alpha_{i_j}) \\ &= \gamma_j G_j(\alpha_{i_j}) \\ &= f G_j(\alpha_{i_j}), \end{aligned}$$

so $f = (\Sigma(\alpha_{i_j})b(\alpha_{i_j}) - a(\alpha_{i_j}))/G_j(\alpha_{i_j})$. □

Exercise 7.1.5. Assume $(a, b) \in M_j$. Show that $(a, b) \in M_{j+1}$ if and only if

$$\Sigma(\alpha_{i_j})b(\alpha_{i_j}) - a(\alpha_{i_j}) = 0.$$

7.2 Gröbner Bases for M_j

We seek a way of describing or characterizing the solutions modules M_j. Gröbner bases, defined in Appendix B.3, are an appropriate tool for accomplishing this. In this section, we consider Gröbner bases for M_j with respect to two different term orders (Appendix B.2). First, in Theorem 7.2.1, we explicitly give a Gröbner basis for M_j with respect to the term order $<_{\deg(\Sigma_j)}$. We use this basis to show that $\{M_j\}_{j=0}^{r}$ is a strictly decreasing sequence of modules in Theorem 7.2.2. In Theorem 7.2.3 and Corollary 7.2.4, we show that the unique

solution to the key equation is an element of a Gröbner basis for M_r with respect to the term order $<_{-1}$. This suggests that we should find a Gröbner basis for M_j with respect to the term order $<_{-1}$. We will use the known Gröbner basis with respect to $<_{\deg(\Sigma_j)}$ to establish that candidate Gröbner bases for M_j with respect to $<_{-1}$ are generating sets for M_j which is crucial in showing that they are in fact Gröbner bases. This task is completed in 7.3.

Theorem 7.2.1. *([12, Section 4]) $\beta = \{(\Sigma_j, 1), (G_j, 0)\}$ is a Gröbner basis for M_j with respect to the term ordering $<_{\deg(\Sigma_j)}$.*

Proof. First, observe that $(\Sigma_j, 1), (G_j, 0) \in M_j$. As well, for any $(a, b) \in M_j$, $a(x) - \Sigma(x)b(x) \equiv 0 \mod G_j(x)$ and thus $a(x) - \Sigma_j(x)b(x) = f(x)G_j(x)$ for some $f \in \mathbb{F}[x]$. Thus $(a, b) = (\Sigma_j b, b) + (a - \Sigma_j b, 0) = b(\Sigma_j, 1) + f(G_j, 0)$, and thus β forms a generating set of M_j.

With respect to $<_{\deg(\Sigma_j)}$, $LT((\Sigma_j, 1)) = (0, 1)$ and $LT((G_j, 0)) = (x^j, 0)$. β is a two element generating set of M_j with leading terms on opposite sides, and thus, by Theorem B.3.3, forms a Gröbner basis for M_j. □

Theorem 7.2.2. *$\{M_j\}_{j=0}^r$ is a strictly decreasing sequence of modules:*

$$M_0 \supset M_1 \ldots \supset M_r.$$

Proof. If $(a, b) \in M_{j+1}$, then $a - \Sigma b = fG_{j+1}$ for some $f \in \mathbb{F}[x]$. Then

$$a - \Sigma b = fG_{j+1} \equiv 0 \mod G_j,$$

so $(a, b) \in M_j$ and $M_{j+1} \subseteq M_j$.

By Theorem 7.2.1, $(G_j, 0) \in M_j$. However,

$$G_j - \Sigma \cdot 0 = G_j \not\equiv 0 \mod G_{j+1},$$

so $G_j \notin M_{j+1}$ and thus $M_j \supset M_{j+1}$. □

Theorem 7.2.3. *([12, Lemma 1]) (s, g), the sought solution to the key equation KE6.1.1, is a minimal element of M with respect to the term order $<_{-1}$.*

Proof. Recall that $\deg(s) \leq t-1$ and $\deg(g) = t$. Relative to the term order $<_{-1}$, the leading term of (s, g) is $(0, x^t)$. In search of contradiction, assume that (s, g) is not minimal. Then there exists some other $(a, b) \in M$ such that $LT(a, b) <_{-1} LT(s, g)$.

Case 1: $LT(a, b) = (0, x^{\deg b})$ with $\deg a < \deg b < \deg g$. This contradicts (s, g) as the solution with $\deg g$ minimal.

Case 2: $LT(a, b) = (x^{\deg a}, 0)$ with $\deg b \leq \deg a < \deg g = t$. Then $s(x) - \Sigma(x)g(x) \equiv 0 \mod G_{\overline{\mathcal{I}}}(x)$ and $a(x) - \Sigma(x)b(x) \equiv 0 \mod G_{\overline{\mathcal{I}}}(x)$. Thus

$$\begin{aligned} b(x)\left[s(x) - \Sigma(x)g(x)\right] - g(x)\left[a(x) - \Sigma(x)b(x)\right] &= b(x)s(x) - g(x)a(x) \\ &\equiv 0 \mod G_{\overline{\mathcal{I}}}(x). \end{aligned}$$

Hence, $b(x)s(x) - g(x)a(x)$ has degree at most $2t-1$ and is identically zero at $r > 2t-1$ points, so $b(x)s(x) = g(x)a(x)$. However, $\deg s < \deg g$ and, by assumption for this case, $\deg b \leq \deg a$, thus $\deg(b \cdot s) < \deg(g \cdot a)$, and thus $b(x)s(x) \neq g(x)a(x)$ providing a contradiction.

□

Corollary 7.2.4. *If β is a Gröbner basis for M with respect to the term order $<_{-1}$, then (s, g), the sought solution to KE6.1.1, is a scalar multiple of the minimal element of β.*

Proof. This follows directly from Theorem 7.2.3 and Lemma B.3.4. □

7.3 Solution by Successive Approximation

We showed in Corollary 7.2.4 above that the unique solution to the key equation is a scalar multiple of the minimal element of any Gröbner basis for M_r with respect to $<_{-1}$.

Therefore we seek to construct such a Gröbner basis. The Welch-Berlekamp algorithm accomplishes this iteratively starting with an obvious Gröbner basis for M_0 with respect to $<_{-1}$, and building towards a Gröbner basis for M_r. The remainder of this section details how to construct a Gröbner basis with respect to $<_{-1}$ for M_{j+1} given a Gröbner basis for M_j. This is the main iterative step to the Welch-Berlekamp algotihm.

Lemma 7.3.1. *$\{(0,1),(1,0)\}$ form a Gröbner basis for M_0 with respect to $<_{-1}$ and $(0,1)$ is the minimal element of this basis.*

Proof. Recall that $G_0(x) = 1$, and thus $M_0 = \{(a,b) : a(x) \equiv \Sigma(x)b(x) \mod 1\} = \{(a,b)\} = \mathcal{M} = \mathbb{F}[x] \times \mathbb{F}[x]$. Clearly, $\{(0,1),(1,0)\}$ is a generating set for M_0. Since the two elements have leading terms on opposite sides (Defintion B.2.6), this is a Gröbner basis (Theorem B.3.3). □

Corollary 7.3.2. *If $\beta_j = \{(a_1,b_1),(a_2,b_2)\}$ is a generating set for M_j and $(a_1,b_1) \in M_{j+1}$ then $(a_2,b_2) \notin M_{j+1}$.*

Proof. This follows directly from Theorem 7.2.2. □

Lemma 7.3.3. *If $(a,b) \in M_j$, then $(x-\alpha_{i_j})(a,b) \in M_{j+1}$. As well, $LT((x-\alpha_{i_j})(a,b)) = x\,LT((a,b))$ and is therefore on the same side (Defintion B.2.6) as $LT((a,b))$.*

Proof. Let $(u,v) = (x-\alpha_{i_j})(a,b)$. We seek to show that $u - v\Sigma \equiv 0 \mod G_{j+1}$. Notice that

$$u - \Sigma v = (x-\alpha_{i_j})(a - \Sigma b).$$

Since $(a,b) \in M_j$, $a - \Sigma b \equiv 0 \mod G_j$ and thus there exists $f \in \mathbb{F}[x]$ such that $a - \Sigma b = fG_j$. Therefore,

$$\begin{aligned} u - \Sigma v &= (x-\alpha_{i_j}) f G_j \\ &= f G_{j+1} \\ &\equiv 0 \mod G_{j+1}. \end{aligned}$$

□

Lemma 7.3.4. *([8, Lemma 4.3]) Let $\beta_j = \{(a_1, b_1), (a_2, b_2)\}$ be a Gröbner basis for M_j with respect to $<_{-1}$. Define the constants $f_1, f_2 \in \mathbb{F}$ by $\Sigma b_1 - a_1 \equiv f_1 G_j \mod G_{j+1}$ and $\Sigma b_2 - a_2 \equiv f_2 G_j \mod G_{j+1}$[1]. Then there exists $\phi, u_1, u_2, v_1, v_2 \in \mathbb{F}[x]$ such that $\Sigma_{j+1} = \Sigma_j + \phi G_j$,*

$$(\Sigma_j, 1) = u_1(a_1, b_1) + u_2(a_2, b_2) \quad \textit{and} \quad (G_j, 0) = v_1(a_1, b_1) + v_2(a_2, b_2).$$

As well, $(x - \alpha_{i_j}) \mid (f_1(u_1 + \phi v_1) + f_2(u_2 + \phi v_2))$.

Proof. Notice that $\Sigma_{j+1} = \Sigma_j + \phi G_j$ for some $\phi \in \mathbb{F}[x]$. Thus

$$(\Sigma_{j+1}, 1) = (\Sigma_j, 1) + \phi(G_j, 0).$$

Since $(\Sigma_j, 1), (G_j, 0) \in M_j$, it must be possible to express them in the β_j basis and the desired u_1, u_2, v_1, v_2 exist. Therefore,

$$\begin{aligned}(\Sigma_{j+1}, 1) &= u_1(a_1, b_1) + u_2(a_2, b_2) + \phi\left[v_1(a_1, b_1) + v_2(a_2, b_2)\right] \\ &= (u_1 + \phi v_1)(a_1, b_1) + (u_2 + \phi v_2)(a_2, b_2) \qquad (7.2)\end{aligned}$$

We may break (7.2) into two pieces:

$$\Sigma_{j+1} = (u_1 + \phi v_1)a_1 + (u_2 + \phi v_2)a_2 \quad \text{and} \qquad (7.3)$$

$$1 = (u_1 + \phi v_1)b_1 + (u_2 + \phi v_2)b_2 \qquad (7.4)$$

Multiplying (7.4) by Σ_{j+1} yields

$$\Sigma_{j+1} = (u_1 + \phi v_1)\Sigma_{j+1} b_1 + (u_2 + \phi v_2)\Sigma_{j+1} b_2.$$

[1] $f_1, f_2 \in \mathbb{F}$ by Corollary 7.1.4.

Subtracting (7.3) from this gives

$$\begin{aligned}
0 &= (u_1 + \phi v_1)(\Sigma_{j+1} b_1 - a_1) + (u_2 + \phi v_2)(\Sigma_{j+1} b_2 - a_2) \\
&\equiv (u_1 + \phi v_1)(\Sigma b_1 - a_1) + (u_2 + \phi v_2)(\Sigma b_2 - a_2) \mod G_{j+1} \\
&\equiv (u_1 + \phi v_1) f_1 G_j + (u_2 + \phi v_2) f_2 G_j \mod G_{j+1} \\
&\equiv [f_1(u_1 + \phi v_1) + f_2(u_2 + \phi v_2)]\, G_j \mod G_{j+1}
\end{aligned}$$

Thus $(x - \alpha_{i_j}) | [f_1(u_1 + \phi v_1) + f_2(u_2 + \phi v_2)]$. □

Lemmas 7.3.1 gives a Gröbner basis for the M_0 base case. Let $\beta_j = \{(a_1, b_1), (a_2, b_2)\}$ be a Gröbner basis for M_j with respect to $<_{-1}$. By Corollary 7.1.4, there exist constants $f_1, f_2 \in \mathbb{F}$ such that $\Sigma b_1 - a_1 \equiv f_1 G_j \mod G_{j+1}$ and $\Sigma b_2 - a_2 \equiv f_2 G_j \mod G_{j+1}$. $f_i = 0$ indicates that $(a_i, b_i) \in M_{j+1}$. We would like to leverage β_j as much as possible in constructing a Gröbner basis β_{j+1} for M_{j+1}. However, Corollary 7.3.2 indicates that $\beta_j \not\subset M_{j+1}$, thus constructing β_{j+1} is non-trivial even in the relatively easy case where one of the basis elements for M_j is also in M_{j+1}. Lemma 7.3.3 suggests a possible element in M_{j+1} that could be used to form the new Gröbner basis.

Theorem 7.3.5. *Let $\beta_j = \{(a_1, b_1), (a_2, b_2)\}$ be a Gröbner basis for M_j with respect to $<_{-1}$. If $(a_1, b_1) \in M_{j+1}$, then $\beta_{j+1} = \{(a_1, b_1), (x - \alpha_{i_j})(a_2, b_2)\}$ is a Gröbner basis for M_{j+1} with respect to $<_{-1}$.*

Proof. By Lemma 7.3.3, $(x - \alpha_{i_j})(a_2, b_2) \in M_{j+1}$. Since β_j is a Gröbner basis for M_j, (a_1, b_1) and (a_2, b_2) must have leading terms on opposite sides with respect to $<_{-1}$. By Lemma 7.3.3, $LT((x - \alpha_{i_j})(a_2, b_2))$ is on the same side as $LT((a_2, b_2))$. Thus the elements of β_{j+1} have leading terms on opposite sides.

It remains to be shown that β_{j+1} forms a generating set for M_{j+1} and thus is a Gröbner basis for M_{j+1} by Theorem B.3.3. By Theorem 7.2.1, $\{(\Sigma_{j+1}, 1), (G_{j+1}, 0)\}$ is a generating set for M_{j+1}. We will show that β_{j+1} also forms a generating set for M_{j+1} by showing

that both $(\Sigma_{j+1}, 1)$ and $(G_{j+1}, 0)$ can be written as a linear combination of the elements of β_{j+1}.

$(G_j, 0) \in M_j$, so there exists $u, v \in \mathbb{F}[x]$ such that

$$(G_j, 0) = u(a_1, b_1) + v(a_2, b_2).$$

Thus

$$\begin{aligned}(G_{j+1}, 0) &= (x - \alpha_{i_j})(G_j, 0) \\ &= u(x - \alpha_{i_j})(a_1, b_1) + v(x - \alpha_{i_j})(a_2, b_2)\end{aligned}$$

and $(G_{j+1}, 0)$ may be written as a linear combination of elements of β_{j+1}.

Define $f_1, f_2 \in \mathbb{F}$ and $\phi, u_1, u_2, v_1, v_2 \in \mathbb{F}[x]$ as in Lemma 7.3.4. Then from (7.2) in the proof of Lemma 7.3.4,

$$\begin{aligned}(\Sigma_{j+1}, 1) &= (u_1 + \phi v_1)(a_1, b_1) + (u_2 + \phi v_2)(a_2, b_2) \\ &= (u_1 + \phi v_1)(a_1, b_1) + \frac{(u_2 + \phi v_2)}{x - \alpha_{i_j}}(x - \alpha_{i_j})(a_2, b_2). \qquad (7.5)\end{aligned}$$

Lemma 7.3.4 states that $(x - \alpha_{i_j}) \mid (f_1(u_1 + \phi v_1) + f_2(u_2 + \phi v_2))$. Since $(a_1, b_1) \in M_{j+1}$, $f_1 = 0$ and $(x - \alpha_{i_j}) \mid f_2(u_2 + \phi v_2)$ and, since $f_2 \in \mathbb{F}$, $(x - \alpha_{i_j}) \mid (u_2 + \phi v_2)$. Thus (7.5) gives $(\Sigma_{j+1}, 1)$ as a linear combination of the elements of β_{j+1}. □

Theorem 7.3.6. *Let $\beta_j = \{(a_1, b_1), (a_2, b_2)\}$ be a Gröbner basis for M_j with respect to $<_{-1}$, and assume that (a_1, b_1) is the minimal element of β_j,* i.e. *$(a_1, b_1) <_{-1} (a_2, b_2)$. Define $f_1, f_2 \in \mathbb{F}$ by $\Sigma b_1 - a_1 \equiv f_1 G_j \mod G_{j+1}$ and $\Sigma b_2 - a_2 \equiv f_2 G_j \mod G_{j+1}$. Assume that $f_1 \neq 0$ and thus $(a_1, b_1) \notin M_{j+1}$. Then $\beta_{j+1} = \left\{(x - \alpha_{i_j})(a_1, b_1), \left(a_2 - \frac{f_2}{f_1} a_1, b_2 - \frac{f_2}{f_1} b_1\right)\right\}$ is a Gröbner basis for M_{j+1} with respect to $<_{-1}$.*

Proof. This is a hefty theorem, so I will break it into a number of pieces:

Claim 7.3.6.1. $\frac{f_2}{f_1} \in \mathbb{F}$ *and thus* $\left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right) \in \mathbb{F}[x] \times \mathbb{F}[x]$.

Proof. It follows from Corollary 7.1.4 that $f_1, f_2 \in \mathbb{F}$. By assumption $f_1 \neq 0$, thus $\frac{f_2}{f_1} \in \mathbb{F}$ and the rest of the claim follows. ✠

Claim 7.3.6.2. $(x - \alpha_{i_j})(a_1, b_1) \in M_{j+1}$.

Proof. This follows from Lemma 7.3.3. ✠

Claim 7.3.6.3. $\left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right) \in M_{j+1}$.

Proof. Notice that

$$\begin{aligned}\Sigma\left(b_2 - \frac{f_2}{f_1}b_1\right) - \left(a_2 - \frac{f_2}{f_1}a_1\right) &= (\Sigma b_2 - a_2) - \frac{f_2}{f_1}(\Sigma b_1 - a_1) \\ &\equiv f_2 G_j - \frac{f_2}{f_1} f_1 G_j \mod G_{j+1} \\ &\equiv 0 \mod G_{j+1}.\end{aligned}$$

✠

Claim 7.3.6.4. *β_{j+1} has leading terms on opposite sides, and is therefore a Gröbner basis by Theorem B.3.3.*

Proof. By Lemma 7.3.3, $LT((x - \alpha_{i_j})(a_1, b_1))$ is on the same side as $LT((a_1, b_1))$. As well,

$$LT\left(\left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right)\right) = LT\left((a_2, b_2) - \frac{f_2}{f_1}(a_1, b_1))\right) = LT(a_2, b_2)$$

since (a_1, b_1) is minimal. Since (a_1, b_1) and (a_2, b_2) having leading terms on opposite sides, β_{j+1} must also. By Theorem B.3.3, β_{j+1} is a Gröbner basis, although it remains to be shown that it is a Gröbner basis for M_{j+1}. ✠

We have yet to show that β_{j+1} forms a generating set for M_{j+1}. By Theorem 7.2.1, $\{(\Sigma_{j+1}, 1),\ (G_{j+1}, 0)\}$ is a generating set for M_{j+1}. We will show that β_{j+1} also forms a generating set for M_{j+1} by showing that both $(\Sigma_{j+1}, 1)$ and $(G_{j+1}, 0)$ can be written as a linear combination of the elements of β_{j+1}. Let $\phi, u_1, u_2, v_1, v_2 \in \mathbb{F}[x]$ be defined as in Lemma 7.3.4.

Claim 7.3.6.5. $(G_{j+1}, 0)$ *can be written as a linear combination of the elements of* β_{j+1}.

Proof.

$$\begin{aligned}
(G_{j+1}, 0) &= (x - \alpha_{i_j})(G_j, 0) \\
&= (x - \alpha_{i_j})\left[v_1(a_1, b_1) + v_2(a_2, b_2)\right] \\
&= (x - \alpha_{i_j})\left[v_1(a_1, b_1) + v_2\frac{f_2}{f_1}(a_1, b_1) - v_2\frac{f_2}{f_1}(a_1, b_1) + v_2(a_2, b_2)\right] \\
&= (x - \alpha_{i_j})\left[\left(v_1 + v_2\frac{f_2}{f_1}\right)(a_1, b_1) + v_2\left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right)\right] \\
&= \left(v_1 + v_2\frac{f_2}{f_1}\right)(x - \alpha_{i_j})(a_1, b_1) - (x - \alpha_{i_j})v_2\left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right).
\end{aligned}$$

By Claim 7.3.6.1, $\frac{f_2}{f_1} \in \mathbb{F}$ so $\left(v_1 + v_2\frac{f_2}{f_1}\right)$ and $(x - \alpha_{i_j})v_2$ are elements in $\mathbb{F}[x]$ and the above expression gives $(G_{j+1}, 0)$ in terms of the elements of β_{j+1}. ✠

Claim 7.3.6.6. $(\Sigma_{j+1}, 1)$ *can be written as a linear combination of the elements of* β_{j+1}.

Proof. From (7.2) in the proof of Lemma 7.3.4,

$$\begin{aligned}(\Sigma_{j+1}, 1) &= (u_1 + \phi v_1)(a_1, b_1) + (u_2 + \phi v_2)(a_2, b_2) \\ &= (u_1 + \phi v_1)(a_1, b_1) + (u_2 + \phi v_2)\left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right) \\ &\quad + \frac{f_2}{f_1}(u_2 + \phi v_2)(a_1, b_1) \\ &= \frac{(u_1 + \phi v_1) + \frac{f_2}{f_1}(u_2 + \phi v_2)}{x - \alpha_{i_j}}(x - \alpha_{i_j})(a_1, b_1) \qquad (7.6) \\ &\quad + (u_2 + \phi v_2)\left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right).\end{aligned}$$

Lemma 7.3.4 states that $(x - \alpha_{i_j}) \mid (f_1(u_1 + \phi v_1) + f_2(u_2 + \phi v_2))$. Since $f_1 \in \mathbb{F}$, it follows that $(x - \alpha_{i_j}) \mid \left((u_1 + \phi v_1) + \frac{f_2}{f_1}(u_2 + \phi v_2)\right)$, and (7.6) gives $(\Sigma_{j+1}, 1)$ as a linear combination of the elements of β_{j+1}. ✠

□

7.4 The Welch-Berlekamp Algorithm

We have now established the necessary background to present the Welch-Berlekamp decoding algorithm. As previously discussed, this algorithm may be broken into three phases. In phase one, the "key equation" is derived from the received word, as discusses in Section 6.1. In phase two, this key equation is solved by successive approximation per the techniques of Section 7.3. In phase three, the message or codeword is recovered from the solution to the key equation. Each of these phases is described in detail below.

7.4.1 Phase 1: Set Up the Key Equation

Given a received word $\vec{r}$ we wish to translate the problem of decoding $\vec{r}$ to solving the key equation KE6.1.1. In order to solve the key equation in phase two, it will suffice to

know the values of σ_j^* for $j \in \overline{\mathcal{I}}$; it is not necessary to construct $\Sigma(x)$ or $G_{\overline{\mathcal{I}}}(x)$ explicitly. Algorithms 7.4.1 computes $\vec{\sigma}^*$ in the following steps:

(i) **Lines**: 3

Purpose: Recover the message $\vec{m}'$ from the indices $\mathcal{I}$ in $\vec{r}$, per the methods of erasure decoding in Section 4.1. F is the generating matrix for the code and is assumed to be systematic.

Computational complexity: If $f_m = |\{i \notin \mathcal{I}, i < k\}|$, then ErasureDecodeSysLA costs $\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m - 6$ operations. If $f_m = 0$, the typical case, then this is free.

(ii) **Lines**: 4

Purpose: $\hat{m}(x)$ per the notation in Section 6.1 is the message polynomial corresponding to $\vec{m}'$. Construct $\hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}})$, the vector of the polynomial $\hat{m}(x)$ evaluated at the points α_i for $i \in \overline{\mathcal{I}}$. This may be accomplished by encoding $\vec{m}'$ at the indices $\overline{\mathcal{I}}$ using the generating matrix F.

Computational complexity: $F_{\overline{\mathcal{I}},:}$ is size $r \times k$ and $\vec{m}'$ is size $k \times 1$ so this matrix multiplication costs $r(2k-1)$ operations.

(iii) **Lines**: 5

Purpose: Construct $G_{\mathcal{I}}(\vec{\alpha}_{\overline{\mathcal{I}}})$, the vector of the polynomial $G_{\mathcal{I}}$ evaluated at the points α_i for $i \in \overline{\mathcal{I}}$.

Computational complexity: $2kr$

(iv) **Lines**: 7 - 10

Purpose: Construct the vector $\vec{\sigma}^*$ as $\sigma_j^* = \frac{r_j - \hat{m}(\alpha_j)}{G_{\mathcal{I}}(\alpha_j)}$ for $j \in \overline{\mathcal{I}}$.

Computational complexity: $2r$

Overall,

$$\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m - 6 + r(2k-1) + 2kr + 2r$$
$$= \frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m + r(4k+1) - 6$$

operations are required for phase 1 if $f_m > 0$. If $f_m = 0$, then $r(4k+1)$ operations are required.

Algorithm 7.4.1 Welch-Berlekamp Phase 1: Set up the key equation

Dependencies: ErasureDecodeSysLA defined in Algorithm 4.1.2, PolyEvalRoots defined in Algorithm A.3.5

1: **procedure** WBPhase1($k, n, \vec{\alpha}, \vec{r}, \mathcal{I}, \overline{\mathcal{I}}, F$) ⋆*Phase 1: Set up the key equation*
2: $r \leftarrow n - k$
3: $\vec{m}' \leftarrow$ ErasureDecodeSysLA($\vec{r}, F, \overline{\mathcal{I}}$) ⋆*Recover $\vec{m}'$ from $\vec{r}$ treating $\overline{\mathcal{I}}$ as the erasure set*
4: $\hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}}) \leftarrow F_{\overline{\mathcal{I}},:}\vec{m}'$ ⋆*Evaluate $\hat{m}$ at the points α_i for $i \in \overline{\mathcal{I}}$*
5: $G_{\mathcal{I}}(\vec{\alpha}_{\overline{\mathcal{I}}}) \leftarrow$ PolyEvalRoots($\vec{\alpha}_{\mathcal{I}}, \vec{\alpha}_{\overline{\mathcal{I}}}$)
6:
7: create an array $\vec{\sigma}^*$ of length r
8: **for** $j \leftarrow 0 \ldots r-1$ **do**
9: $\sigma_j^* = \frac{r_{i_j} - \hat{m}(\alpha_{i_j})}{G_{\mathcal{I}}(\alpha_{i_j})}$
10: **end for**
11: **return** $\vec{\sigma}^*, \hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}})$
12: **end procedure**

Computational complexity: $\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m + 4rk - 6$ operations required for $f_m > 0$, and $r(4k+1)$ operations required for $f_m = 0$, where $f_m = |\{i \notin \mathcal{I}, i < k\}|$.

7.4.2 Phase 2: Solve the Key Equation Using the Welch-Berlekamp Algorithm

In phase 2, we seek solutions to the key equation (6.7) using the Welch-Berlekamp algorithm developed in Section 7.3. The solution is found iteratively; in each iteration, $\{(a_1, b_1), (a_2, b_2)\}$ is a Gröbner basis for M_j with respect to $<_{-1}$ and (a_1, b_1) is its minimal element. We use this Gröbner basis to compute a Gröbner basis for M_{j+1}. By

Lemma 7.2.4, (s, g), the sought solution to (6.7), is a scalar multiple of the minimal element of the Gröbner basis of $M = M_r$ with respect to $<_{-1}$. Therefore, we iterate until we have a Gröbner basis for M_r with (a_1, b_1). Lemma 7.4.1 shows the b_1 must be monic, so we return $(s, g) = (a_1, b_1)$ as the unique solution to the key equation.

For ease of notation in this phase, we define $\vec{\gamma} = \vec{\alpha}_{\overline{\mathcal{I}}}$ and denote $\gamma_j = \alpha_{i_j}$. Phase 2 will accept $\vec{\gamma}$ as an input in lieu of $\vec{\alpha}$ and $\mathcal{I}$. Algorithm 7.4.2 provides pseudocode for phase 2; its functionality, proof of correctness, and computational complexity are outlined below.

(i) **Lines**: 2 - 4

Purpose: Initialize $(a_1, b_1) = (0, 1)$ and $(a_2, b_2) = (1, 0)$. By Lemma 7.3.1, this forms a Gröbner basis for M_0 with respect to $<_{-1}$. Initialize $d_1 = d_2 = 0$. d_1 and d_2 will hold the degree of the leading term of (a_1, b_1) and (a_2, b_2) respectively. d_1 and d_2 will be useful in enforcing the minimality of (a_1, b_1) at the end of each iteration.

Computational complexity: No computation required.

(ii) **Lines**: 5 - 28

Purpose: Loop from $j = 0$ to $r-1$. At the beginning of each loop, $\{(a_1, b_1), (a_2, b_2)\}$ is Gröbner basis for M_j with respect to $<_{-1}$ and (a_1, b_1) is the minimal element. Over the course of the loop, a Gröbner basis for M_{j+1} is computed.

(a) **Lines**: 6

Purpose: Compute $\hat{f}_1 = \sigma_j^* b_1(\gamma_j) - a_1(\gamma_j)$. Note that by Corollary 7.1.4, $\hat{f}_1 = G_j(\gamma_j) f_1$, for f_1 defined as in Theorem 7.3.6. By Exercise 7.1.5, $(a_1, b_1) \in M_{j+1}$ if and only if $\hat{f}_1 = 0$. As well, $\frac{f_2}{f_1} = \frac{\hat{f}_2}{\hat{f}_1}$. It is more computationally efficient to compute $\hat{f}_1$ and $\hat{f}_2$ than f_1 and f_2 and functionally equivalent.

(b) **Lines**: 7 - 10

Purpose: If $\hat{f}_1 = 0$, then $(a_1, b_1) \in M_{j+1}$. By Theorem 7.3.5, $\{(a_1, b_1), (x - \gamma_j)(a_2, b_2)\}$ is a Gröbner basis for M_{j+1} with respect to $<_{-1}$. Set $(a_2, b_2) =$

$(x-\gamma_j)(a_2,b_2)$ and do not change (a_1,b_1). This increases the degree of both a_2 and b_2 and thus the degree of the leading term of (a_2,b_2), so d_2 must also be incremented.

(c) **Lines**: 11 - 18

Purpose: $(a_1,b_1) \notin M_{j+1}$. By Theorem 7.3.6,

$$\left\{(x-\gamma_j)(a_1,b_1), \left(a_2 - \frac{f_2}{f_1}a_1, b_2 - \frac{f_2}{f_1}b_1\right)\right\}$$

is a Gröbner basis for M_{j+1} with respect to $<_{-1}$. Compute $\hat{f}_2 = \sigma_j^* b_2(\gamma_j) - a_2(\gamma_j)$ and update the basis accordingly. This increases the degree of both a_1 and b_1 and thus the degree of the leading term of (a_1,b_1), so d_1 must also be incremented. The degree of the leading term of (a_2,b_2) does not change (see the proof of Claim 7.3.6.4).

(d) **Lines**: 19 - 27

Purpose: If $(a_2,b_2) <_{-1} (a_1,b_1)$ switch (a_1,b_1) and (a_2,b_2) so that (a_1,b_1) is always the minimal element of the Gröbner basis. We may determine whether $(a_2,b_2) <_{-1} (a_1,b_1)$ by examining d_1 and d_2. These polynomials are represented by the vector of their coefficients, so $\text{length}(a_1) - 1 = \deg(a_1)$. If $\text{length}(a_1) - 1 = d_1$, then the leading term of (a_1,b_1) must be on the left and the leading term of (a_2,b_2) must be on the right. If we also have $d_1 > d_2 - 1$ then $(a_2,b_2) <_{-1} (a_1,b_1)$ so we must swap (a_1,b_1) with (a_2,b_2) and d_1 with d_2. If $\text{length}(a_1) - 1 \neq d_1$, then the leading term of (a_1,b_1) must be on the right and the leading term of (a_2,b_2) must be on the left. If we also have $d_2 \leq d_1 - 1$ then $(a_2,b_2) <_{-1} (a_1,b_1)$ so we must swap (a_1,b_1) with (a_2,b_2) and d_1 with d_2.

Computational complexity: In the worst case in iteration j $\hat{f}_1 \neq 0$, so both $\hat{f}_1$ and

$\hat{f}_2$ must be computed. By Lemma 7.4.3, $\deg(a_1)+\deg(b_1)+\deg(a_2)+\deg(b_2) < 2j$ so the four necessary calls to PolyEval cost in total no more than $2(2j-1)$ operations plus 4 more operations to construct $\hat{f}_1$ and $\hat{f}_2$. The two calls to PolyMultLin cost at most $2\deg(a_1)+1+2\deg(b_1)+1 \leq 2j$ operations, by Lemma 7.4.3. Lines 13 and 14 contribute at most an additional $2(\deg(a_1)+1)+2(\deg(b_1)+1)+1 \leq 2j+3$ operations. This yields a total operation count across all iterations of no more than

$$\begin{aligned}\sum_{j=0}^{r-1} 4j+2+2j+2j+3 &= \sum_{j=0}^{r-1} 8j+5 \\ &= 4r(r-1)+5r = 4r^2+r.\end{aligned}$$

In other texts, the term "Welch-Berlekamp algorithm" refers only to Phase 2 of the "Welch-Berlekamp" algorithm presented here. Phase 2 closely resembles the algorithm presented in [12], and Lemmas 7.4.2 and 7.4.3 which allow for a tight analysis of the computational cost were inspired by comments in that paper. To leading term, the operation count is the same as the operation count found in [12].

Lemma 7.4.1. *Let $\beta_j = \{(a_1,b_1),(a_2,b_2)\}$ be the Gröbner basis for M_j with respect to $<_{-1}$ at the beginning of the jth iteration in Algorithm 7.4.2. Let (a_{r_j},b_{r_j}) be the element in this basis with leading term on the right. Then b_{r_j} is monic, and the solution to the key equation KE6.1.1 found by this algorithm is the unique solution.*

Proof. We prove this inductively. In the base case $j=0$ with $\beta_0 = \{(0,1),(1,0)\}$, $(a_r,b_r) = (0,1)$ and b_{r_0} is clearly monic. Let $\beta_j = \{(a_1,b_1),(a_2,b_2)\}$ be the Gröbner basis for M_j with respect to $<_{-1}$ at the beginning of the jth iteration in Algorithm 7.4.2, and assume b_{r_j} is monic. Let (a_{l_j},b_{l_j}) denote the element of β_j with leading term on the left.

Case 1: $\hat{f}=0$. Then $\beta_{j+1} = \{(a_1,b_1),(x-\gamma_j)(a_2,b_2)\}$. Clearly, $b_{r_{j+1}}$ is also monic since b_{r_j} is, regardless of whether $r_j = 1$ or 2.

Case 2: $\hat{f} \neq 0$, and

$$\beta_{j+1} = \left\{ (x - \gamma_j)(a_1, b_1), \left(a_2 - \frac{f_2}{f_1} a_1, b_2 - \frac{f_2}{f_1} b_1 \right) \right\}.$$

If $r_j = 1$, then $b_{r_{j+1}} = (x - \gamma_j) b_{r_j}$ is also monic. If $r_j = 2$, then $b_{r_{j+1}} = b_{r_j} - \frac{f_2}{f_1} b_{l_j}$, $\deg a_{l_j} < \deg b_{r_j}$ (since $l_j = 1$ and therefore (a_{l_j}, b_{l_j}) is the minimal element of β_j), and $\deg a_{l_j} \geq \deg b_{l_j}$ (since the leading term of (a_{l_j}, b_{l_j}) is on the left), so $\deg b_{l_j} < \deg_{r_j}$. Therefore the leading monomial of $b_{r_{j+1}}$ is the leading monomial of b_{r_j} and $b_{r_{j+1}}$ is monic.

By Corollary 7.2.4, the unique solution to KE6.1.1 is a scalar multiple of the minimum element of β_r. Since (s, g) is the unique solution to KE6.1.1, and it has leading term on the right, the minimal element of β_j, (a_1, b_1), must also have leading term on the right and thus b_1 must be monic. Thus $(a_1, b_1) = (s, g)$. □

Lemma 7.4.2. *Let $\beta_j = \{(a_1, b_1), (a_2, b_2)\}$ be the Gröbner basis for M_j with respect to $<_{-1}$ at the beginning of the jth iteration in Algorithm 7.4.2. Then*

$$\deg(LT((a_1, b_1)) + \deg(LT((a_2, b_2)) = j.$$

Proof. Notice that this is trivially true for the base case $j = 0$. I will prove it in the general case inductively. Assume the lemma holds for some value of j. Let $\beta_{j+1} = \{(a_1', b_1'), (a_2', b_2')\}$.

Case 1: $\hat{f} = 0$. Then $\beta_{j+1} = \{(a_1, b_1), (x - \gamma_j)(a_2, b_2)\}$, and $\deg(a_1) = \deg(a_1)$, $\deg(b_1) = \deg(b_1)$, $\deg(a_2') = \deg(a_2) + 1$, and $\deg(b_2) = \deg(b_2) + 1$. Therefore,

$$\deg(LT((a_1', b_1')) + \deg(LT((a_2', b_2')) = j + 1,$$

and the lemma holds.

Case 2: $\hat{f} \neq 0$, and

$$\beta_{j+1} = \left\{ (x - \gamma_j)(a_1, b_1), \left(a_2 - \frac{f_2}{f_1} a_1, b_2 - \frac{f_2}{f_1} b_1 \right) \right\}.$$

$\deg(LT(a_1', b_1'))) = \deg(LT(a_1, b_1)) + 1$, and $\deg(LT(a_2', b_2'))) = \deg(LT(a_2, b_2))$ (see the proof of Claim 7.3.6.4). Therefore,

$$\deg(LT((a_1', b_1')) + \deg(LT((a_2', b_2')) = j + 1,$$

and the lemma holds.

Notice that in some iterations (a_1, b_1) and (a_2, b_2) are swapped, but this does not effect the sum of their leading terms. □

Lemma 7.4.3. *Let $\beta_j = \{(a_1, b_1), (a_2, b_2)\}$ be the Gröbner basis for M_j with respect to $<_{-1}$ at the beginning of the jth iteration in Algorithm 7.4.2. Then*

$$\deg(a_1) + \deg(b_2) \leq j \quad \textit{and} \quad \deg(a_2) + \deg(b_1) \leq j,$$

$$\deg(a_1) + \deg(b_2) + \deg(a_2) + \deg(b_1) < 2j,$$

and

$$\deg(a_1) + \deg(b_1) < j.$$

Proof. The leading terms of β_j are on opposite sides, and thus are of the form $(x^p, 0)$ and $(0, x^q)$. From Lemma 7.4.2,

$$\deg(LT((a_1, b_1)) + \deg(LT((a_2, b_2)) = p + q = j.$$

Case 1: The leading term of (a_1, b_1) is on the left. Then

(i) $\deg(a_1) = p$.

(ii) $\deg(b_1) \leq p$ since $(0, b_1) <_{-1} (a_1, 0)$.

(iii) $\deg(b_2) = q$ since the leading term of (a_2, b_2) is on the right.

(iv) $\deg(a_2) < q$ since $(a_2, 0) <_{-1} (0, b_2)$.

Therefore,

$$\deg(a_1) + \deg(b_2) = p + q = j \quad \text{and} \quad \deg(a_2) + \deg(b_1) < q + p = j.$$

Since (a_1, b_1) is minimal, we must have $(x^p, 0) <_{-1} (0, x^q)$, so $p < q$, and $2p < p+q = j$. Therefore,

$$\deg(a_1) + \deg(b_1) \leq 2p < j.$$

Case 2: The leading term of (a_1, b_1) is on the right. Then

(i) $\deg(b_1) = q$.

(ii) $\deg(a_1) < q$ since $(a_1, 0) <_{-1} (0, b_1)$.

(iii) $\deg(a_2) = p$ since the leading term of (a_2, b_2) is on the left.

(iv) $\deg(b_2) \leq p$ since $(0, b_1) <_{-1} (a_1, 0)$.

Therefore,

$$\deg(a_1) + \deg(b_2) < q + p = j \quad \text{and} \quad \deg(a_2) + \deg(b_1) = p + q = j.$$

Since (a_1, b_1) is minimal, we must have $(0, x^q) <_{-1} (x^p, 0)$, so $q \leq p$, and $2q \leq p+q = j$. Therefore,

$$\deg(a_1) + \deg(b_1) < 2q \leq j.$$

□

7.4.3 Phase 3: Reconstruct the Message $\vec{m}$ via Erasure Recovery

In phase 3, we use the error locator polynomial g and the received word $\vec{r}$ to recover the original message $\vec{m}$. Algorithm 7.4.3 provides pseudocode for phase 3; its functionality, proof of correctness, and computational complexity are outlined below. Notice that Algorithm 7.4.3 accepts $\mathcal{F}$ as its final argument. $\mathcal{F}$ is the set of known erasures in the received word $\vec{r}$. The Welch-Berlekamp algorithm is adapted for the errors and erasures setting in Algorithm 7.4.7 discussed in Section 7.4.6.

Algorithm 7.4.2 Welch-Berlekamp Phase 2: Solve the key equation

Dependencies: PolyEval defined in Algorithm A.3.1, PolyMultLin defined in Algorithm A.3.2

1: **procedure** WBPhase2($r, \vec{\gamma}, \vec{\sigma}^*$) ⋆*Phase 2: Solve the key equation*
⋆*Inputs:* $r = n - k, \vec{\gamma} = \vec{\alpha}_{\overline{\mathcal{I}}}$, $\vec{\sigma}^*$ *defines the key equation*
2: $(a_1, b_1) \leftarrow (0, 1)$ ⋆*initialize basis elements*
3: $(a_2, b_2) \leftarrow (1, 0)$
4: $d_1 \leftarrow 0, \quad d_2 \leftarrow 0$ ⋆*Initialize degree of leading terms*
5: **for** $j \leftarrow 0 \ldots r-1$ **do** ⋆$\{(a_1, b_1), (a_2, b_2)\}$ *is a basis for* M_j,
⋆*construct a basis for* M_{j+1}
6: $\hat{f}_1 \leftarrow \sigma_j^*$ PolyEval(b_1, γ_j) - PolyEval(a_1, γ_j) ⋆$\hat{f}_1 = \sigma_j^* b_1(\gamma_j) - a_1(\gamma_j)$
7: **if** $\hat{f}_1 = 0$ **then** ⋆$(a_1, b_1) \in M_{j+1}$
8: $a_2 \leftarrow$PolyMultLin(a_2, γ_j) ⋆*Set* $(a_2, b_2) = (x - \gamma_j)(a_2, b_2)$
9: $b_2 \leftarrow$PolyMultLin(b_2, γ_j)
10: $d_2 \leftarrow d_2 + 1$ ⋆*Increment the degree of the leading term of* (a_2, b_2)
11: **else** ⋆$(a_1, b_1) \notin M_{j+1}$
12: $\hat{f}_2 \leftarrow \sigma_j^*$ PolyEval(b_2, γ_j) - PolyEval(a_2, γ_j) ⋆$\hat{f}_2 = \sigma_j^* b_2(\gamma_j) - a_2(\gamma_j)$
13: $a_2 \leftarrow a_2 - \frac{\hat{f}_2}{\hat{f}_1} a_1$ ⋆*Set* $(a_2, b_2) = \left(a_2 - \frac{\hat{f}_2}{\hat{f}_1} a_1, b_2 - \frac{\hat{f}_2}{\hat{f}_1} b_1\right)$
14: $b_2 \leftarrow b_2 - \frac{\hat{f}_2}{\hat{f}_1} b_1$
15: $a_1 \leftarrow$PolyMultLin(a_1, γ_j) ⋆*Set* $(a_1, b_1) = (x - \gamma_j)(a_1, b_1)$
16: $b_1 \leftarrow$PolyMultLin(b_1, γ_j)
17: $d_1 \leftarrow d_1 + 1$ ⋆*Increment the degree of the leading term of* (a_1, b_1)
18: **end if**
19: **if** length$(a_1) - 1 = d_1$ **then** ⋆*Leading term of* (a_1, b_1) *is on the left*
20: **if** $d_1 > d_2 - 1$ **then** ⋆$(a_2, b_2) <_{-1} (a_1, b_1)$
21: $(a_2, b_2) \longleftrightarrow (a_1, b_1), \quad d_1 \longleftrightarrow d_2$ ⋆*swap* (a_1, b_1) *and* (a_2, b_2)
22: **end if**
23: **else** ⋆*Leading term of* (a_1, b_1) *is on the right*
24: **if** $d_2 \leq d_1 - 1$ **then** ⋆$(a_2, b_2) <_{-1} (a_1, b_1)$
25: $(a_2, b_2) \longleftrightarrow (a_1, b_1), \quad d_1 \longleftrightarrow d_2$ ⋆*swap* (a_1, b_1) *and* (a_2, b_2)
26: **end if**
27: **end if**
28: **end for**
29: **return** a_1, b_1
30: **end procedure**

Computational complexity: At most $4r^2 + r$ operations required.

(i) **Lines**: 2 - 8

Purpose: Construct $\mathcal{E} = \{i : g(\alpha_i) = 0\}$ by checking whether $g(\alpha_i) = 0$ for $1 \leq i \leq n$.

Computational complexity: g has degree at most $\hat{t} = \lfloor \frac{r-f}{2} \rfloor$, so each call to PolyEval costs $2\hat{t} \leq r - f$ operations. This yields a total of no more than $n(r-f)$ operations.

(ii) **Lines**: 9

Purpose: Recover $\vec{m}$ from $\vec{r}$ per Section 4.1 using $\mathcal{E}$ as the erasure set.

Computational complexity: If $t_m = |\{i \in \mathcal{E} \cup \mathcal{F}, i < k\}|$ is the number of message indices that are either erased or in error, then this costs $\frac{2}{3}t_m^3 + \frac{3}{2}t_m^2 + (2k + \frac{11}{6})t_m - 6$ operations.

Overall, no more than $n(r-f) + \frac{2}{3}t_m^3 + \frac{3}{2}t_m^2 + (2k + \frac{11}{6})t_m - 6$ operations are required for phase 3. We can obtain a more concrete bound on this cost by noting that $t_m \leq \frac{r-f}{2}$, so we have no more than

$$\begin{aligned} & n(r-f) + \frac{2}{3}\left(\frac{r-f}{2}\right)^3 + \frac{3}{2}\left(\frac{r-f}{2}\right)^2 + \left(2k + \frac{11}{6}\right)\left(\frac{r-f}{2}\right) - 6 \\ &= \frac{1}{12}(r-f)^3 + \frac{3}{8}(r-f)^2 + \left(n + k + \frac{11}{12}\right)(r-f) - 6 \end{aligned}$$

operations.

7.4.4 Phase 3: Reconstruct the Codeword $\vec{c}$ From $A(x)$

As suggested in Section 6.2, there is an alternate method for finding the codeword given s, g using (6.8). Notice that this version returns the original codeword not the message. A decision about which version of phase 3 to use in an application should be made largely based on whether the message or the codeword is desired as output, and whether or not the code is systematic. For a systematic code, this version is faster in either case. Pseudocode for this method is given by Algorithm 7.4.4 which is outlined below.

Algorithm 7.4.3 Welch-Berlekamp Phase 3: Reconstruct the message $\vec{m}$

Dependencies: PolyEval defined in Algorithm A.3.1, ErasureDecodeSysLA defined in Algorithm 4.1.2.

```
1: procedure WBPhase3(α⃗, r⃗, g⃗, F, F)        ⋆Phase 3: Reconstruct the message m⃗.
2:     Initialize E as empty list               ⋆E is the set of error indices
3:     n ←length(α⃗)
4:     for j ← 0 ... n − 1 do                   ⋆Determine error locations
5:         if PolyEval(g⃗, α_j) = 0 then         ⋆g(α_j) =PolyEval(g, α_j)
6:             Add j to E
7:         end if
8:     end for
9:     m⃗ ← ErasureDecodeSysLA(r⃗, F, E ∪ F)     ⋆Recover m⃗ from r⃗ using E ∪ F
                                                ⋆as the erasure set.
10:    return m⃗
11: end procedure
```

Computational complexity: $n(r-f) + \frac{2}{3}t_m^3 + \frac{3}{2}t_m^2 + (2k + \frac{11}{6})t_m - 6 \leq \frac{1}{12}(r-f)^3 + \frac{3}{8}(r-f)^2 + \left(n + k + \frac{11}{12}\right)(r-f) - 6$ operations required, where $t_m = |\{i \in \mathcal{E} \cup \mathcal{F}, i < k\}|$.

(i) **Lines**: 3 - 9

Purpose: Construct $\mathcal{E} = \{i : g(\alpha_i) = 0\}$ by checking whether $g(\alpha_i) = 0$ for $1 \leq i \leq n$.

Computational complexity: g has degree at most $\hat{t} = \lfloor \frac{r}{2} \rfloor$, so each call to PolyEval costs $2\hat{t} \leq r$ operations. This yields a total of no more than nr operations.

(ii) **Lines**: 10

Purpose: Construct $g_{\overline{\mathcal{I}}}(x)$.

Computational complexity: Let $t_0 = |\mathcal{I} \cap \mathcal{E}|$. Then $\deg g_{\overline{\mathcal{I}}} = t_0$ and the call to PolyRoots costs t_0^2 operations.

(iii) **Lines**: 11

Purpose: Compute $A(x) = \frac{-s(x)}{g_{\overline{\mathcal{I}}}(x)}$.

Computational complexity: $\deg s < t$ and $\deg g_{\overline{\mathcal{I}}} = t_0$, so the call to PolyDiv

costs at most

$$2t_0(t - t_0) + t + t_0 + 1 = 2t_0t_1 + t + t_0 + 1$$

operations where $t_1 = |\overline{\mathcal{I}} \cap \mathcal{E}| = t - t_0$.

(iv) **Lines**: 13

Purpose:Construct $G_{\mathcal{I}\setminus\mathcal{E}}(\vec{\alpha}_{\mathcal{E}})$.

Computational complexity: The call to PolyEvalRoots costs $2(k - t_0)t$ operations.

(v) **Lines**: 14

Purpose: Evaluate A at the error locations.

Computational complexity: $\deg A < t_0$, so the call to PolyEval costs at most $2t_0t$ operations.

(vi) **Lines**: 15 - 21

Purpose: Correct the errors per Section 6.2.

Computational complexity: $2t$ operations are required.

Overall, no more than $nr + t_0^2 + 2t_0t_1 + t + t_0 + 1 + 2(k - t_0)t + 2t_0t + 2t = nr + kt + t_0^2 + 2t_0t_1 + 3t + t_0 + 1 \leq nr + kt + 3t^2 + 4t + 1$ operations are required for this version of phase 3.

7.4.5 Two Error Correction Algorithms

The full error correcting algorithm algorithm consists of simply executing the three phases in order and carefully passing the results of one phase on to the next. It accepts as inputs the dimension of the code k, the length of the code n, the interpolating points $\vec{\alpha}$, the systematic generating matrix F, and the received word $\vec{r}$, and returns the message $\vec{m}$ corresponding to the closest code word to $\vec{r}$. By default, it will use $\mathcal{I} = \{0, \ldots, k - 1\}$ as this choice reduces the computational complexity of the message recovery step in phase

Algorithm 7.4.4 Welch-Berlekamp Phase 3v2: Reconstruct the message $\vec{c}$

Dependencies: PolyEval defined in Algorithm A.3.1, PolyRoots defined in Algorithm A.3.4, PolyEvalRoots defined in Algorithm A.3.5.

procedure WBPhase3v2($\vec{\alpha}, \vec{r}, \hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}}), \vec{g}, \vec{s}, \mathcal{I}, \overline{\mathcal{I}}$) ⋆*Phase 3: Reconstruct the*
⋆*codeword* $\vec{c}$.
  Initialize $\mathcal{E}$ as empty list ⋆$\mathcal{E}$ *is the set of error indices*
  $n \leftarrow$length($\vec{\alpha}$)
  for $j \leftarrow 0 \ldots n-1$ **do** ⋆*Determine error locations*
    if PolyEval($\vec{g}, \alpha_j$) $= 0$ **then** ⋆$g(\alpha_j) =$PolyEval(g, α_j)
      Add j to $\mathcal{E}$
    end if
  end for
  $\vec{g}_{\overline{\mathcal{I}}} \leftarrow$ PolyRoots($\vec{\alpha}_{\overline{\mathcal{I}} \cap \mathcal{E}}$) ⋆*Construct* $g_{\overline{\mathcal{I}}}(x)$
  $\vec{A}, \vec{p} \leftarrow$ PolyDiv($-\vec{s}, \vec{g}_{\overline{\mathcal{I}}}$) ⋆*Construct* $A(x) = \frac{-s(x)}{g_{\overline{\mathcal{I}}}(x)}$
  Require: $\vec{p} = 0$ ⋆*The remainder from division ought to be zero*
  $G_{\mathcal{I}\setminus\mathcal{E}}(\vec{\alpha}_{\mathcal{E}}) \leftarrow$ PolyEvalRoots($\vec{\alpha}_{\mathcal{I}\setminus\mathcal{E}}, \vec{\alpha}_{\mathcal{E}}$) ⋆*Construct* $G_{\mathcal{I}\setminus\mathcal{E}}(\vec{\alpha}_{\mathcal{E}})$
  $A(\vec{\alpha}_{\mathcal{E}}) \leftarrow$ PolyEval($\vec{A}, \vec{\alpha}_{\mathcal{E}}$) ⋆*Evaluate* A *at the error locations*
  $\vec{c} \leftarrow \vec{r}$ ⋆*Initialize* $\vec{c}$ *as the received word*
  for $j \in \mathcal{E} \cap \mathcal{I}$ **do** ⋆*Correct errors*
    $c_j \leftarrow r_j - G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j) A(\alpha_j)$ ⋆*Use (6.8), note* $\hat{m}(\alpha_j) = r_j$ *for* $j \in \mathcal{I}$
  end for
  for $j \in \mathcal{E} \cap \overline{\mathcal{I}}$ **do**
    $c_j \leftarrow \hat{m}(\alpha_j) - G_{\mathcal{I}\setminus\mathcal{E}}(\alpha_j) A(\alpha_j)$ ⋆*Use (6.8)*
  end for
  return $\vec{c}$
end procedure

Computational complexity: No more than $nr + kt + t_0^2 + 2t_0t_1 + 3t + t_0 + 1 \leq nr + kt + 3t^2 + 4t + 1$ operations required.

1 for systematic codes. It is given by Algorithm 7.4.5 below. With this choice of $\mathcal{I}$, phase 1 costs $r(4k+1)$ operations, phase 2 costs $4r^2+r$ operations, and phase 3 costs $rn+\frac{2}{3}t_m^3+\frac{3}{2}t_m^2+(2k+\frac{11}{6})t_m-6$ operations where $t_m=|\{i\in\mathcal{E},i<k\}|$. The total cost is

$$\begin{aligned}
&r(4k+1)+4r^2+r+rn+\frac{2}{3}t_m^3+\frac{3}{2}t_m^2+\left(2k+\frac{11}{6}\right)t_m-6\\
&=r(4(n-r)+1)+4r^2+r+rn+\frac{2}{3}t_m^3+\frac{3}{2}t_m^2+\left(2k+\frac{11}{6}\right)t_m-6\\
&=5rn+2r+\frac{2}{3}t_m^3+\frac{3}{2}t_m^2+\left(2k+\frac{11}{6}\right)t_m-6.
\end{aligned}$$

Using the upper bound for phase 3, we see that we can bound the total operation count by

$$\begin{aligned}
&r(4k+1)+4r^2+r+\frac{1}{12}r^3+\frac{3}{8}r^2+\left(n+k+\frac{11}{12}\right)r-6\\
&r(4(n-r)+1)+4r^2+r+\frac{1}{12}r^3+\frac{3}{8}r^2+\left(n+n-r+\frac{11}{12}\right)r-6\\
&=6rn+\frac{1}{12}r^3-\frac{5}{8}r^2+\frac{35}{12}r-6.
\end{aligned}$$

Algorithm 7.4.5 Welch-Berlekamp

Dependencies: WBPhase1 defined in Algorithm 7.4.1, WBPhase2 defined in Algorithm 7.4.2, WBPhase3 defined in Algorithm 7.4.3

1: **procedure** WB($k,n,\vec{\alpha},F,\vec{r}$)
2: $\quad\mathcal{I}\leftarrow\{0,\ldots,k-1\}$
3: $\quad\overline{\mathcal{I}}\leftarrow\{k,\ldots,n-1\}$
4: $\quad\vec{\sigma}^*,\hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}})\leftarrow$WBPhase1$(k,n,\vec{\alpha},\vec{r},\mathcal{I},\overline{\mathcal{I}},F)$
5: $\quad\vec{s},\vec{g}\leftarrow$WBPhase2$(n-k,\vec{\alpha}_{\overline{I}},\vec{\sigma}^*)$
6: $\quad\vec{m}\leftarrow$WBPhase3$(\vec{\alpha},\vec{r},\vec{g},\emptyset,F)$
7: $\quad$**return** $\vec{m}$
8: **end procedure**

Computational complexity: No more than $5rn+r+\frac{2}{3}t_m^3+\frac{3}{2}t_m^2+\left(2k+\frac{11}{6}\right)t_m-6\leq 6rn+\frac{1}{12}r^3-\frac{5}{8}r^2+\frac{35}{12}r-6$ operations required.

Alternately, we could use the second version of phase 3. Pseudo code for this is given

in Algorithm 7.4.6. If the encoding was systematic, then this costs no more than

$$\begin{aligned} r(4k+1) + 4r^2 + r + nr + kt + 3t^2 + 4t + 1 &= 4rk + 4r^2 + 3t^2 + nr + kt + 2r + 4t + 1 \\ &= 4rk + 4r^2 + 3t^2 + (n+2)r + (k+4)t + 1 \\ &\leq 4r(n-r) + 4r^2 + \frac{3}{4}r^2 + (n+2)r + (n-r+4)\frac{r}{2} + 1 \\ &= \frac{1}{4}r^2 + \left(\frac{11}{2}n + 4\right)r + 1 \end{aligned}$$

operations.

Algorithm 7.4.6 Welch-Berlekamp version 2

Dependencies: WBPhase1 defined in Algorithm 7.4.1, WBPhase2 defined in Algorithm 7.4.2, WBPhase3v2 defined in Algorithm 7.4.4

1: **procedure** WBv2($k, n, \vec{\alpha}, F, \vec{r}$)
2: $\mathcal{I} \leftarrow \{0, \ldots, k-1\}$
3: $\overline{\mathcal{I}} \leftarrow \{k, \ldots, n-1\}$
4: $\vec{\sigma}^*, \hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}}) \leftarrow$WBPhase1$(k, n, \vec{\alpha}, \vec{r}, \mathcal{I}, \overline{\mathcal{I}}, F)$
5: $\vec{s}, \vec{g} \leftarrow$WBPhase2$(n-k, \vec{\alpha}_{\overline{I}}, \vec{\sigma}^*)$
6: $\vec{c} \leftarrow$WBPhase3v2$(\vec{\alpha}, \vec{r}, \hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}}), \vec{g}, \vec{s}, \mathcal{I}, \overline{\mathcal{I}})$
7: **return** $\vec{c}$
8: **end procedure**

Computational complexity: No more than $4rk + 4r^2 + 3t^2 + (n+2)r + (k+4)t + 1 \leq \frac{1}{4}r^2 + \left(\frac{11}{2}n + 4\right)r + 1$ operations required.

7.4.6 An Error and Erasure Correcting Algorithm

Only a few modifications are needed to adapt Algorithm 7.4.5 for use in the presence of erasures. In the erasure case, the algorithm accepts one additional input: the set of indices of erasures $\mathcal{F}$.

We take $\mathcal{I}$ to be the first k values of $0 \leq i < n$ such that $i \notin \mathcal{F}$, and redefine $\overline{\mathcal{I}} = \{1, \ldots, n\} \backslash (\mathcal{I} \cup \mathcal{F})$, that is $\overline{\mathcal{I}}$ is the indices that are neither in $\mathcal{I}$ nor are erased. With this redefinition, Phases 1 and 2 proceed as in Algorithm 7.4.5. In Phase 3 step (c), use $\mathcal{E} \cup \mathcal{F}$

as the erasure set. Phase 1 costs $\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m + 4rk - 6$ operations where $f_m = |\{i \in \mathcal{F}, i < k\}|$; Phase 2 costs $4r^2 + r$ operations; Phase 3 costs $n(r-f) + \frac{2}{3}t_m^3 + \frac{3}{2}t_m^2 + (2k + \frac{11}{6})t_m - 6 \leq \frac{1}{12}(r-f)^3 + \frac{3}{8}(r-f)^2 + \left(n + k + \frac{11}{12}\right)(r-f) - 6$ operations where $t_m = |\{i \in \mathcal{E} \cup \mathcal{F}, i < k\}|$. The total operation count is

$$\begin{aligned}
&\frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m + 4rk - 6 + 4r^2 + r + n(r-f) + \frac{2}{3}t_m^3 \\
&\qquad + \frac{3}{2}t_m^2 + \left(2k + \frac{11}{6}\right) t_m - 6 \\
&= \frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right) f_m + 4r(n-r) + 4r^2 + r + n(r-f) + \frac{2}{3}t_m^3 \\
&\qquad + \frac{3}{2}t_m^2 + \left(2k + \frac{11}{6}\right) t_m - 12 \\
&= \frac{2}{3}f_m^3 + \frac{3}{2}f_m^2 + \left(2k + \frac{11}{6}\right)(f_m + t_m) + (5r - f)n + r + \frac{2}{3}t_m^3 + \frac{3}{2}t_m^2 - 12.
\end{aligned}$$

We can bound this count using $f_m \leq f$ and $t_m \leq \frac{r-f}{2}$:

$$\begin{aligned}
&\leq \frac{2}{3}f^3 + \frac{3}{2}f^2 + \left(2k + \frac{11}{6}\right)\left(f + \frac{r-f}{2}\right) + (5r - f)n + r + \frac{1}{12}(r-f)^3 + \frac{3}{8}(r-f)^2 - 12 \\
&= \frac{2}{3}f^3 + \frac{3}{2}f^2 + \left(n - r + \frac{11}{12}\right)(r + f) + (5r - f)n + r + \frac{1}{12}(r-f)^3 + \frac{3}{8}(r-f)^2 - 12 \\
&= \frac{2}{3}f^3 + \frac{3}{2}f^2 + \left(-r + \frac{11}{12}\right)(r + f) + r(6n + 1) + \frac{1}{12}(r-f)^3 + \frac{3}{8}(r-f)^2 - 12 \\
&= \Theta(f^3 + (r-f)^3 + rn).
\end{aligned}$$

It is not recommended to use Algorithm 7.4.4, the alternate version of phase 3, in the erasures case. This version would return the codeword with the error corrected, but with the erasures still missing. An additional erasure recovery step would still be necessary, so the overall cost would likely be lower using the first version of phase 3.

Example 7.4.4. Consider the systematic code defined by $l = 8$, $n = 10$, $k = 6$, and $\vec{\alpha} = (0, 1, 2, \ldots, 9)^T$. We would like to decode the received word

$$\vec{r} = (177, 44, 243, 8, 112, 97, 161, 96, 138, 204)^T$$

Algorithm 7.4.7 Welch-Berlekamp with erasures

Dependencies: WBPhase1 defined in Algorithm 7.4.1, WBPhase2 defined in Algorithm 7.4.2, WBPhase3 defined in Algorithm 7.4.3

1: **procedure** WBe$(k, n, \vec{\alpha}, \vec{r}, \mathcal{F}\ )$
2: $f \leftarrow |\mathcal{F}|$
3: $\mathcal{I} \leftarrow$ first k values of $0 \leq i < n$ such that $i \notin \mathcal{F}$
4: $\overline{\mathcal{I}} \leftarrow \{1, \ldots, n\} \backslash (\mathcal{I} \cup \mathcal{F})$
5: $\vec{\sigma}^* \leftarrow$WBPhase1$(k, n, \vec{\alpha}, \vec{r}, \mathcal{I}, \overline{\mathcal{I}}, F\ )$
6: $\vec{g} \leftarrow$WBPhase2$(n - k - f, \vec{\alpha}_{\overline{I}}, \vec{\sigma}^*)$
7: $\vec{m} \leftarrow$WBPhase3$(k, n, \vec{\alpha}, \vec{r}, \vec{g}, \mathcal{F})$
8: **return** $\vec{m}$
9: **end procedure**

Computational complexity: $\Theta(f^3 + (r - f)^3 + rn)$.

using Algorithm 7.4.5; there are no erasures. We have

$$\mathcal{I} = \{0, 1, 2, 3, 4, 5\} \quad \text{and} \quad \overline{\mathcal{I}} = \{6, 7, 8, 9\},$$

and

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 7 & 6 & 5 & 4 & 3 & 2 \\ 6 & 7 & 4 & 5 & 2 & 3 \\ 160 & 223 & 223 & 183 & 254 & 232 \\ 223 & 160 & 183 & 223 & 232 & 254 \end{bmatrix}.$$

In phase 1 we compute:

$$\vec{m}' = (177, 44, 243, 8, 112, 97)^T, \quad \hat{m}(\vec{\alpha}_{\overline{\mathcal{I}}}) = (178, 197, 242, 187)^T, \quad G_{\mathcal{I}}(\vec{\alpha}_{\overline{\mathcal{I}}}) = (35, 35, 79, 79)^T$$

and

$$\vec{\sigma}^* = (138, 21, 141, 191)^T.$$

In phase 2 we compute the following Gröbner bases for M_j:

- $\beta_0 = \{(0,1),(1,0)\}$ is a Gröbner basis for M_0.
- $\beta_1 = \{(1,39),(0,x+6)\}$ is a Gröbner basis for M_1.
- $\beta_2 = \{(222,x+228),(x+7,39x+245)\}$ is a Gröbner basis for M_2.
- $\beta_3 = \{(x+5,21x+23),(222x+190,x+236x+115)\}$ is a Gröbner basis for M_3.
- $\beta_4 = \{(145x+208,x^2+6x+7),(x^2+12x+45,21x^2+170x+175)\}$ is a Gröbner basis for M_4.

Therefore, $g(x) = x^2 + 6x + 7$. In phase 3 we find that the errors are in the locations $\mathcal{E} = \{1,7\}$ and so the original message was

$$\vec{m} = (177,81,243,8,112,97)^T.$$

The codeword corresponding to this message is

$$\vec{c} = (177,81,243,8,112,97,161,171,138,204)^T.$$

So $d(\vec{r},\vec{c}) = 2$ which is within the error bounds for this code so we assume this decoding is correct.

7.5 A Generalized Key Equation and Its Solution

It is possible to generalize the key equation somewhat and adapt the Welch-Berlekamp algorithm to solve this generalized equation. In Chapter 7.6, we will see that (5.2) may be viewed as an instance on this gneralized key equation, and thus can be solved using the Welch-Berlekamp algorithm faster than by the linear algebra techniques discussed in Chapter 5.2. Instances of this generalized key equation may also be found in other applications, and thus it is worth noting that they can be solved efficiently. This generalization is also discussed by Fitzpatrick in [8], but Jennings does not explore this generalization in [12] when she adapts Fitzpatrick's arguments to the key equation we use here.

Generalized Key Equation KE7.5.1

Given $\Sigma(x)$ and $G(x) = \prod_{i=0}^{r-1}(x-\alpha_i)$ with $\deg(\Sigma) < \deg(G) = r$, find $s(x), g(x) \in \mathbb{F}[x]$ such that

- $s(x) = \Sigma(x)g(x) \mod G(x)$,
- $\deg(s) \leq \tau$, $\deg(g) \leq t$ and $\tau + t < r$,
- g is of minimum degree and monic.

Notice that this generalized key equation reduces to the key equation of Chapter 6.1 when $\tau - t = -1$. To solve this generalized key equation, we again consider the solution modules

$$M_j = \{(a, b) : a(x) \equiv \Sigma_j(x)b(x) \mod G_j(x)\}.$$

Most of the rest of the development of the Welch-Berlekamp algorithm proceeds as in Sections 7.1 and 7.3 with some minor adjustments of notation. We no longer use the term order $<_{-1}$. Instead, define $\phi = \tau - t$. We replace $<_{-1}$ with $<_\phi$. In most of the ensuing statements, we may simply make this replacement and the theorems, lemmas, corollaries, and their proofs follow. However, we must modify Lemma 7.3.1 somewhat, and substantially modify the proof of Theorem 7.2.3.

Lemma 7.5.1. *(Generalization of Lemma 7.3.1)* $\{(0,1),(1,0)\}$ *form a Gröbner basis for* M_0 *with respect to* $<_\phi$*. For* $\phi < 0$, $(0,1)$ *is the minimal element of this basis; for* $\phi \geq 0$, $(1,0)$ *is the minimal element of this basis.*

Proof. That $\{(0,1),(1,0)\}$ is a Gröbner basis for M_0 follows exactly as in the proof of Lemma 7.3.1. Which element is minimal follows directly from the definition of term orderings. □

Theorem 7.5.2. *(Generalization of Theorem 7.2.3)* (s, g)*, the sought solution to the generalized key equation KE7.5.1, is a minimal element of* M *with respect to the term order* $<_\phi$*.*

Proof. Recall that $\deg s \leq \tau$ and $\deg g = t$ and $\tau + t \leq r$. Relative to the term order $<_\phi$, the leading term of (s, g) is $(0, x^t)$. In search of contradiction, assume that (s, g) is not minimal. Then there exists some other $(a, b) \in M$ such that $LT(a, b) <_\phi LT(s, g)$.

Case 1: $LT(a, b) = (0, x^{\deg b})$. Then $LT(a, b) <_\phi LT(s, g)$ implies $\deg b < \deg g$ which contradicts (s, g) as the solution with $\deg g$ minimal.

Case 2: $LT(a, b) = (x^{\deg a}, 0)$. Then $\deg b < \deg a - \phi \leq \deg g = t$, and $\deg b < t$ and $\deg a \leq \tau$. Then $s(x) - \Sigma(x)g(x) \equiv 0 \mod G(x)$ and $a(x) - \Sigma(x)b(x) \equiv 0 \mod G(x)$. Thus

$$\begin{aligned} b(x)\left[s(x) - \Sigma(x)g(x)\right] - g(x)\left[a(x) - \Sigma(x)b(x)\right] &= b(x)s(x) - g(x)a(x) \\ &\equiv 0 \mod G(x). \end{aligned}$$

$\deg(bs) < t + \tau - 1$ and $\deg(ga) \leq t + \tau$, so $b(x)s(x) - g(x)a(x)$ has degree at most $t + \tau$ and is identically zero at $r > t + \tau$ points, so $b(x)s(x) = g(x)a(x)$. However, $\deg s \leq \deg g + \phi$ and, by assumption for this case, $\deg b < \deg a - \phi$, thus $\deg(bs) < \deg g + \phi + \deg a - \phi = \deg(ga)$ and thus $b(x)s(x) \neq g(x)a(x)$ providing a contradiction.

□

Exercise 7.5.3. Adapt the rest of Sections 7.1 and 7.3 to solve the generalized key equation.

Remark 7.5.4. In the original statement of the key equation in Section 6.1, it is not necessary to require $\deg s < \deg g \leq \hat{t}$. It is sufficient to require $\deg s < \hat{t}$, and $\deg g \leq \hat{t}$. The uniqueness of the solution (Theorem 6.1.5) holds under this looser condition, and this unique solution is found by the Welch-Berlekamp algorithm using $\phi = -1$.

Algorithm 7.5.1 gives the Welch-Berlekamp algorithm for solving the generalized key equation. Notice that this algorithm is identical to Algorithm 7.4.2 except that it

- accepts an extra input ϕ,
- initializes the Gröbner basis for M_0 depending on ϕ per Lemma 7.5.1,
- and uses modified conditions for swapping (a_1, b_1) and (a_2, b_2) to reflect the use of the term order $<_\phi$ instead of $<_{-1}$.

Its run time is identical to that of Algorithm 7.4.2. The proof of this follows the proof for Algorithm 7.4.2 without modification, except for Lemma 7.4.3. This Lemma still holds for the generalized algorithm, but the proof must be altered slighty. This alteration is straightforward and is left as an exercise for the reader.

7.6 Application of the Generalized Key Equation: An Alternate Decoding Algorithm

In Chapter 5.2, we derived the modified Welch-Berlekamp algorithm, in which we phrased the error correction problem as solving (5.2). This may be rewritten as a generalized key equation as follows with $\phi = k - 1$:

A Generalized Key Equation for Error Correction KE7.6.2

Given $k, \vec{\alpha}$, $\mathcal{W}$, and $\vec{r}$, define $\hat{n} = |\mathcal{W}|$, $\hat{t} = \lfloor \frac{\hat{n}-k}{2} \rfloor$, $G_{\mathcal{W}}(x) = \prod_{i \in \mathcal{W}}(x - \alpha_i)$ and $R(x) \in \mathbb{F}[x]$ such that $R(\alpha_i) = r_i$ for $i \in \mathcal{W}$. Then $\deg(R) < \deg(G_{\overline{\mathcal{I}}}) = \hat{n}$. Find $s(x), g(x) \in \mathbb{F}[x]$ such that

- $s(x) = R(x)g(x) \mod G(x)$,
- $\deg(s) \leq k + \hat{t} - 1$, $\deg(g) \leq \hat{t}$ and $k + 2\hat{t} - 1 < n - f$,
- g is of minimum degree and monic.

This suggests a new decoding algorithm that requires essentially no compution in phase 1 to set up KE7.6.2, relies on the generalized Welch-Berelekamp algorithm in phase 2 to solve KE7.6.2, and requires very little post-processing in phase 3. This new algorithm is described in pseudo-code by Algorithm 7.6.1, and outlined below. Let $\hat{n} = n - f$ where f is the number of erasures.

Algorithm 7.5.1 Welch-Berlekamp Phase 2: Solve the generalized key equation

Dependencies: PolyEval defined in Algorithm A.3.1, PolyMultLin defined in Algorithm A.3.2

procedure WBPhase2Gen($r, \vec{\gamma}, \vec{\sigma}^*, \phi$) ⋆*Phase 2: Solve the key equation*
⋆*Inputs:* $r = n - k, \vec{\gamma} = \vec{\alpha}_{\overline{\mathcal{I}}}$, $\vec{\sigma}^*$ *defines the key equation*
  if $\phi < 0$ **then** $(a_1, b_1) \leftarrow (0, 1)$ **else** $(a_1, b_1) \leftarrow (1, 0)$ ⋆*initialize basis elements*
  if $\phi < 0$ **then** $(a_2, b_2) \leftarrow (1, 0)$ **else** $(a_2, b_2) \leftarrow (0, 1)$
  $d_1 \leftarrow 0, \quad d_2 \leftarrow 0$ ⋆*Initialize degree of leading terms*
  for $j \leftarrow 0 \ldots r-1$ **do** ⋆$\{(a_1, b_1), (a_2, b_2)\}$ *is a basis for* M_j,
⋆*construct a basis for* M_{j+1}
    $\hat{f}_1 \leftarrow \sigma_j^*$ PolyEval(b_1, γ_j) - PolyEval(a_1, γ_j) ⋆$\hat{f}_1 = \sigma_j^* b_1(\gamma_j) - a_1(\gamma_j)$
    if $\hat{f}_1 = 0$ **then** ⋆$(a_1, b_1) \in M_{j+1}$
      $a_2 \leftarrow$PolyMultLin(a_2, γ_j) ⋆*Set* $(a_2, b_2) = (x - \gamma_j)(a_2, b_2)$
      $b_2 \leftarrow$PolyMultLin(b_2, γ_j)
      $d_2 \leftarrow d_2 + 1$ ⋆*Increment the degree of the leading term of* (a_2, b_2)
    else ⋆$(a_1, b_1) \notin M_{j+1}$
      $\hat{f}_2 \leftarrow \sigma_j^*$ PolyEval(b_2, γ_j) - PolyEval(a_2, γ_j) ⋆$\hat{f}_2 = \sigma_j^* b_2(\gamma_j) - a_2(\gamma_j)$
      $a_2 \leftarrow a_2 - \frac{\hat{f}_2}{\hat{f}_1} a_1$ ⋆*Set* $(a_2, b_2) = \left(a_2 - \frac{\hat{f}_2}{\hat{f}_1} a_1, b_2 - \frac{\hat{f}_2}{\hat{f}_1} b_1\right)$
      $b_2 \leftarrow b_2 - \frac{\hat{f}_2}{\hat{f}_1} b_1$
      $a_1 \leftarrow$PolyMultLin(a_1, γ_j) ⋆*Set* $(a_1, b_1) = (x - \gamma_j)(a_1, b_1)$
      $b_1 \leftarrow$PolyMultLin(b_1, γ_j)
      $d_1 \leftarrow d_1 + 1$ ⋆*Increment the degree of the leading term of* (a_1, b_1)
    end if
    if length$(a_1) - 1 = d_1$ **then** ⋆*Leading term of* (a_1, b_1) *is on the left*
      if $d_1 > d_2 + \phi$ **then** ⋆$(a_2, b_2) <_\phi (a_1, b_1)$
        $(a_2, b_2) \longleftrightarrow (a_1, b_1), \quad d_1 \longleftrightarrow d_2$ ⋆*swap* (a_1, b_1) *and* (a_2, b_2)
      end if
    else ⋆*Leading term of* (a_1, b_1) *is on the right*
      if $d_2 \leq d_1 + \phi$ **then** ⋆$(a_2, b_2) <_\phi (a_1, b_1)$
        $(a_2, b_2) \longleftrightarrow (a_1, b_1), \quad d_1 \longleftrightarrow d_2$ ⋆*swap* (a_1, b_1) *and* (a_2, b_2)
      end if
    end if
  end for
  return a_1, b_1
end procedure

Computational complexity: At most $4r^2 + r$ operations required.

(i) **Lines**: 2 - 3

Purpose: Define $\mathcal{W}$ as all indices that are not erased, f as the number of erasures.

Computational complexity: No Galois field computation required.

(ii) **Lines**: 4

Purpose: Solve the generalized key equation for error correction using the Welch-Berlekamp algorithm.

Computational complexity: $4\hat{n}^2 + \hat{n}$ operations required.

(iii) **Lines**: 5

Purpose: Compute $\vec{\mu}$ as $\frac{s(x)}{g(x)}$.

Computational complexity: s has degree at most $k + \hat{t} - 1$ and g has degree at most $\hat{t}$, so the call to PolyDiv costs

$$\begin{aligned} 2\hat{t}(k + \hat{t} - 1 - \hat{t}) + (k + \hat{t} - 1) + \hat{t} + 1 &= 2\hat{t}(k-1) + k + 2\hat{t} \\ &= (\hat{n} - k)(k-1) + k + \hat{n} - k \\ &= k\hat{n} - \hat{n} - k^2 + k + \hat{n} \\ &= k(\hat{n} - k + 1) \end{aligned}$$

operations.

(iv) **Lines**: 10

Purpose: Return $\vec{m}$, the decoded message as $m_j = \mu(\alpha_j)$ for $0 \leq j < k$.

Computational complexity: The call to PolyEval costs at most $2k(k-1)$ operations.

Overall, no more than $4\hat{n}^2 + \hat{n} + k(\hat{n} - k + 1) + 2k(k-1) = 4\hat{n}^2 + (k+1)\hat{n} + k^2 + k$ operations are required for this algorithm.

Remark 7.6.1. We have largely focused on systematic codes in our discussion of error correction algorithms because message recovery is far cheaper in systematic codes than

Algorithm 7.6.1 Error correction with trivial phase 1, using the generalized Welch-Berlekamp algorithm

Dependencies: WBPhase2Gen defined in Algorithm 7.5.1, WBPhase3 defined in Algorithm 7.4.3

1: **procedure** WBG($k, n, \vec{\alpha}, \mathcal{F}, \vec{r}$)
2: $\mathcal{W} \leftarrow \{0, \ldots, n-1\} \backslash \mathcal{F}$
3: $f \leftarrow |\mathcal{F}|$
4: $\vec{s}, \vec{g} \leftarrow$WBPhase2Gen$(n-f, \vec{\alpha}_{\mathcal{W}}, \vec{r}_{\mathcal{W}}, k-1)$
5: $\vec{\mu}, \vec{R} \leftarrow$ PolyDiv$(\vec{s}, \vec{g})$
6: **Require:** $\vec{R} = 0$ ⋆*The remainder from division ought to be zero*
7:
8: $\vec{m} \leftarrow$ PolyEval$(\vec{\mu}, \vec{\alpha}_{0:k-1})$ ⋆*Recover* $\vec{m}$
9: **return** $\vec{m}$
10: **end procedure**

Computational complexity: No more than $4\hat{n}^2 + (k+1)\hat{n} + k^2 + k$ operations required.

non-systematic codes, and most algorithms discussed include a message recovery step. Algorithm 7.6.1 is the exception to this rule, and thus is particularly well suited for non-sytematic codes. In fact, for non-systematic codes, $\vec{\mu} = \vec{m}$, so there is no need to execute line 8 in Algorithm 7.6.1. Therefore, Algorithm 7.6.1 requires only $4\hat{n}^2 + (k+1)\hat{n} - k^2 + k$ operations, and is the only quadratic-time error correcting algorithm for non-systematic codes discussed.

Chapter 8
Conclusion

In the preceding chapters, I have defined Reed-Solomon codes from the original view, and discussed techniques for erasure correction, error detection, and error correction. I provided two different perspectives from which to view Reed-Solomon codes, the polynomial perspective and the linear algebra perspective, and derived encoding and decoding algorithms based on each perspective. I proposed a new algorithm (Algorithm 3.4.2) for computing the systematic generating matrix for Reed-Solomon codes that is original and more computationally efficient that the method proposed by Plank in [15]. I compared the computational efficiency of the discussed encoding and decoding algorithms in Table 3.1. I then considered theoretical properties of Reed-Solomon codes in Chapter 4, showing that Reed-Solomon codes are maximum distance separable and therefore have the best possible error correction capacity for their length and dimension.

In Chapter 5, I set up a 3-phase framework for error correction algorithms, then introduced the modified Welch-Berlekamp algorithm for error correction. This algorithm was conceptually simple, and thus served as a gentle introduction to error correction algorithms. I provide a detailed derivation of how to frame the error correction problem as finding polynomial solutions to a key equation, proved the uniqueness of the solution to the equation and that the unique solution solved the error correction problem, and showed that this problem may be translated to a linear system. I then showed that standard techniques from linear algebra may be used to find the desired solution to the key equation, provided psuedo-code for this error correction algorithm, and showed that it requires $\Theta(n^3)$ operations.

In Chapter 6, I began my derivation of the Welch-Berlekamp algorithm for error correction which is known to have only a quadratic run time. The first step was to derive a key

equation of lower degree than the one found in Chapter 5. I then showed that the unique solution to this key equation solves the error correction problem. This is a key result that is missing from the literature. In 6.2, I also discussed how the solution to this key equation can be translated back to a solution to the error correction problem.

In Chapter 7, I provided a complete derivation of the Welch-Berlekamp algorithm to solve this key equation. I followed the methodology of [8] and [12], expanding on their work by providing complete and thorough proofs throughout. I then provided pseudo-code for all three phases of the suggested error correction, connected the proof of correctness for these algorithms to the preceding discussions, and perform a detailed and tight analysis of their computational costs. I succeeded in producing an error correction algorithm for systematic codes that requires $\Theta(n^2)$ operations, Algorithm 7.4.6. I concluded Chapter 7 by generalizing the key equation from Chapter 6, and showing the Welch-Berlekamp algorithm can be easily modified to solve this generalized key equation without any increase in computational cost. I then used this generalization to create a new decoding algorithm, that is slower than the ones previously discussed in this chapter for systematic codes, but may be faster for nonsystematic codes.

8.1 Future Work

There are many potential future projects related to this dissertation. What follows is a discussion of a few of them.

8.1.1 Parallel Processing and Computer Architecture

The discussion of computational complexity presented here considered solely the number of Galois Field operations required to execute each algorithm. Modern computer architectures typically feature parallel processors. Some of the algorithms are easily parallelized, such as message encoding, whereas others, such as solving the key equation via the Welch-Berlekamp algorithm, would be difficult to parallelize. As well, issues related to memory

management can have a substantial impact on the real execution time of an algorithm, and have been ignored in the discussion so far. An investigation into efficient parallelization of the algorithms discussed here and which are well suited to the memory architecture in modern computers would be of great practical use to system engineers.

8.1.2 Implement Algorithms as a C package

The development of a package of algorithms written in C (or some other appropriate language) would be a significant and substantial follow-on project to this dissertation. Each of the algorithms here could be implemented as functions in such a package. This would allow more ready access of the discussed ECC technology to developers everywhere.

Appendix A
Algorithms

Many of the algorithms discussed elsewhere in this dissertation rely on common operations with polynomial or matrix operations. Algorithms for these operations are provided in A.3 and A.4 below. As well, A.1 provides some background on arithmetic over finite fields, and A.2 discusses how the computational cost of algorithms is computed in this dissertation.

A.1 Computation Over Finite Fields

Throughout this dissertation, it is assumed that all arithmetic is performed over some field $\mathbb{F}$. In all examples presented and in practical application of ECC algorithms, $\mathbb{F}$ is the finite field (Galois field) GF(2^l). The construction of Galois fields including the definition of $+, -, \cdot$ and $\div$ are well covered in the literature; see [7, 1, 4, 11], for example. Here are a few reminders about finite fields:

- For any prime p and integer l, there exists a unique finite field with p^l elements. This field is often denoted GF(p^l).
- GF(p^l) has two well defined operations, $+$ and $\cdot$, and is closed under these operations.
- 0 is the additive identity and 1 is the multiplicative identity in GF(p^l).
- All elements in GF(p^l) have a unique additive inverse. Addition by the additive inverse if often denoted by $-$.
- All elements in GF(p^l) except 0 have a unique multiplicative inverse. Multiplication by the multiplicative inverse is often denoted by $\div$ or $/$.

In particular, in GF(2^l),

- addition of two elements is equivalent to bitwise addition of the binary representation of those two numbers;
- each element is its own additive inverse ($a + a = 0$ for $a \in \mathrm{GF}(2^l)$) and thus addition and subtraction are equivalent operations;
- multiplication and division are more complicated to define but are often implemented as lookup tables.

Throughout this text, I have typically been careful to keep track of negative signs to preserve the generality of my arguments, but have occasionally dropped them noting that they are not needed in our applications where $\mathbb{F} = \mathrm{GF}(2^l)$. For the purposes of quantifying the computational complexity of various algorithms, I consider all Galois field operations to be equivalent. This is not strictly true in practice, however this assumption simplifies the analyses while still providing a useful measure of the computational cost of the algorithms. The real execution time of all algorithms can be improved through improved implementation of Galois field arithmetic; this is often accomplished in hardware.

A.2 Computational Complexity

In order to compare the efficiency of various algorithms for ECC, it is necessary to carefully count the number of operations required by each algorithm. I have made the following simplifying assumptions in the analysis of algorithms in this dissertation:

1. All Galois field operations ($+, -, \cdot, \div$, exponentiation) have the same computational cost. I therefore provide a count of Galois field operations (or simply "operations"), but do not break these counts down by type of operation.
2. I ignore the costs of:
   (a) incrementing counters

(b) memory allocation

(c) copying data or updating pointers

In any reasonable implementation of the algorithms in this dissertation, the cost of the above tasks is of the same order as or significantly less than the cost of the Galois Field operations required, so I use the number of Galois field operations as a convenient proxy.

In most cases, I give a precise count of the Galois field operations required to execute a given algorithm. However, these counts are often quite complex so I occasionally use Θ notation to discuss algorithms in terms of their asymptotic computational complexity.

Definition A.2.1. For two functions $f(\vec{x}), g(\vec{x})$, we say that $f(\vec{x}) \in \Theta(g(\vec{x}))$ if and only if there exists constants $c_1, c_2, \vec{N} > 0$ such that

$$0 \leq c_1 g(\vec{x}) \leq f(\vec{x}) \leq c_2 g(\vec{x}) \quad \text{when} \quad x_i > N_i \text{ for all } i.$$

Essentially, $f(\vec{x}) \in \Theta(g(\vec{x}))$ if $f(\vec{x})$ behaves like $g(\vec{x})$ asymptotically. For example $x^2 - 5x + 23 \in \Theta(x^2)$ and $xy^2 + 2xy + 3y \in \Theta(xy^2)$.

A.3 Operations with Polynomials

Let $p(x) = \sum_{i=0}^{n} p_i x^i$ be a degree n polynomial. p is completely described by the length $n+1$ vector of its coefficients, $\vec{p} = (p_0, p_1, \ldots, p_n)$; p will typically be represented by $\vec{p}$ for computational purposes. Evaluation of a polynomial p at the point α is a common polynomial operation and may be implemented efficiently by appealing to Horner's rule, as in [5, Chapter 30.1]:

$$p(\alpha) = \sum_{i=0}^{n} p_i \alpha^i = p_0 + \alpha(p_1 + \alpha(p_2 + \ldots (p_{n-1} + \alpha p_n) \ldots)).$$

Algorithm A.3.1 implements polynomial evaluation of a vector of m points via Horner's rule and executes in $2mn$ operations.

Algorithm A.3.1 Polynomial evaluation (Horner's Algorithm)

```
1:  procedure PolyEval(p⃗, α⃗)          ⋆Evaluate the polynomial p⃗ at the points α⃗
2:      n ←length(p⃗) − 1              ⋆Determine the degree of p(x) based on the length of p⃗
3:      m ←length(α⃗)                  ⋆m is the number of points to evaluate
4:      for j ← 0 . . . m − 1 do       ⋆Loop across all of the points
5:          y_j ← p_n
6:          for i ← n − 1 . . . 0 do
7:              y_j ← p_i + α_j y_j
8:          end for
9:      end for
10:     return y⃗
11: end procedure
```

Computational complexity: $2mn$ operations required.

Given a polynomial $p(x)$ and some constant α, computation of the coefficients of $f(x) = (x-\alpha)p(x)$ may be achieved through application of the formula

$$(f_0, f_1, f_2, \ldots, f_n, f_{n+1}) = \begin{array}{c} (0, \quad p_0, \quad p_1, \quad \ldots, \quad p_{n-1}, \quad p_n) \\ - \\ (\alpha p_0, \quad \alpha p_1, \quad \alpha p_2, \quad \ldots, \quad \alpha p_n, \quad 0) \end{array}$$

which implies that $f_i = p_{i-1} - \alpha p_i$ for $1 \leq i \leq n$, $f_0 = \alpha p_0$ and $f_{n+1} = p_n$. This is implemented in Algorithm A.3.2 which executes in $2n+1$ operations.

Algorithm A.3.2 Polynomial multiplication by $x - \alpha$

```
1:  procedure PolyMultLin(p⃗, α)      ⋆Compute the coefficients of f(x) = (x − α)p(x)
2:      n ←length(p⃗) − 1             ⋆Determine the degree of p(x) based on the length of p⃗
3:      create an array f of length n + 2
4:      f_0 ← αp_0
5:      for i ← 1 . . . n do
6:          f_i = p_{i−1} − αp_i
7:      end for
8:      f_{n+1} = p_n
9:      return f
10: end procedure
```

Computational complexity: $2n+1$ operations required.

To multiply two polynomial $p(x)$ and $q(x)$ in the general case, where $\deg p = d_p$ and

$\deg q = d_q$, we use the formula

$$f(x) = p(x)q(x) = \left(\sum_{i=0}^{d_p} p_i x^i\right)\left(\sum_{j=0}^{d_q} q_j x^j\right)$$
$$= \sum_{i=0}^{d_p}\sum_{j=0}^{d_q} p_i q_j x^{i+j}.$$

This is implemented as Algorithm A.3.3 and requires $2d_p d_q$ operations.

Algorithm A.3.3 Polynomial multiplication

```
1:  procedure PolyMult(p⃗, q⃗)          ⋆Compute the coefficients of f(x) = p(x)q(x)
2:      d_p ←length(p⃗) − 1
3:      d_q ←length(q⃗) − 1
4:      Initialize f⃗ as a length d_p + d_q + 1 vector of zeros
5:      for i ← 0 ... d_p do
6:          for j ← 0 ... d_q do
7:              f_{i+j} ← f_{i+j} + p_i q_j
8:          end for
9:      end for
10:     return p⃗
11: end procedure
```

Computational complexity: $2d_p d_q$ operations required.

To construct a polynomial $p(x)$ whose roots are the elements of a vector $\vec{\alpha}$,

$$p(x) = \prod_{\alpha_i \in \vec{\alpha}} (x - \alpha_i),$$

start with $p(x) = 1$, then multiply by each of the linear factors $(x - \alpha_i)$ successively using Algorithm A.3.2. This is implemented in Algorithm A.3.4. In line 5 of this algorithm, $p(x)$ has degree i, thus the call to PolyMultLin requires $2i + 1$ operations. Algorithm A.3.4 thus executes in a total of $\sum_{i=0}^{n-1} 2i + 1 = n + 2\sum_{i=0}^{n-1} i = n + 2\frac{n(n-1)}{2} = n^2$ operations.

If $p(x)$ is defined as the polynomial whose roots are the elements of the vector $\vec{\alpha}$, and $\vec{\beta}$ is some other vector, we may evaluate $p(\vec{\beta})$ without explicitly constructing the coefficients

Algorithm A.3.4 Constructing a polynomial from its roots

Dependencies: PolyMultLin defined in Algorithm A.3.2.

```
1: procedure PolyRoots(α⃗)        ⋆Compute the coefficients of p(x) = ∏_{α_i∈α⃗}(x − α_i)
2:     n ←length(α⃗)
3:     p⃗ ← 1
4:     for i ← 0...n − 1 do
5:         p⃗ ←PolyMultLin(p⃗, α_i)
6:     end for
7:     return p⃗
8: end procedure
```

Computational complexity: n^2 operations required.

Algorithm A.3.5 Polynomial evaluation when the polynomial is defined by its roots

```
1: procedure PolyEvalRoots(α⃗, β⃗)   ⋆Compute p(β⃗) when p is defined by its roots α⃗
2:     n ← length(α)                ⋆Determine the degree of p(x) from the number of roots
3:     m ← length(β)                                          ⋆Number of points to evaluate
4:     for i ← 0...m − 1 do
5:         p(β_i) ← 1
6:         for j ← 0...n − 1 do
7:             p(β_i) ← p(β_i)(β_i − α_j)
8:         end for
9:     end for
10:    return p(β⃗)
11: end procedure
```

Computational complexity: $2nm$ operations required.

of $p(x)$ via the formula

$$p(\beta_i) = \prod_{\alpha_j \in \vec{\alpha}} (\beta_i - \alpha_j).$$

This is implemented as Algorithm A.3.5. If $|\vec{\alpha}| = n$ and $|\vec{\beta}| = m$, then the $2nm$ operations are required.

Given a degree n polynomial $p(x)$ and some constant α, there exists a polynomial q and constant r such that

$$\frac{p(x)}{x-\alpha} = q(x) + \frac{r}{x-\alpha}.$$

$r = 0$ if and only if $(x-\alpha)|p(x)$. Notice that the first few steps of long division look like this:

$$\begin{array}{rl|llll} & & p_n x^{n-1} & & & \\ \hline x-\alpha & \Big| \; p_n x^n & +p_{n-1}x^{n-1} & +p_{n-2}x^{n-2} & +\ldots & p_0 \\ & -(p_n x^n & -\alpha p_n) & & & \\ \cline{3-3} & & (p_{n-1}+\alpha p_n)x^{n-1} & +p_{n-2}x^{n-2} & +\ldots & p_0 \end{array}$$

We may perform the division recursively using the fact that $\frac{p(x)}{x-\alpha} = p_n x^{n-1} + \frac{f_i(x)}{x-\alpha}$ where

$$f_i(x) = (p_{n-1} + \alpha p_n)x^{n-1} + p_{n-2}x^{n-2} + \ldots + p + 0.$$

Algorithm A.3.6 implements polynomial division by a linear term in this manner and costs $2n$ operations.

Dividing a degree n polynomial $p(x)$ by an (arbitrary) degree m polynomial $d(x)$ is a similar process. In this case, there exists a degree $n-m$ polynomial $q(x)$ and a degree $m-1$ or less polynomial $r(x)$ such that

$$\frac{p(x)}{d(x)} = q(x) + \frac{r(x)}{d(x)}.$$

Algorithm A.3.6 Polynomial division by $x - \alpha$

```
1: procedure PolyDivLin(p⃗, α)        ⋆Determine q⃗ and r such that p(x)/(x−α) = q(x) + r/(x−α)
2:     n ←length(p⃗) − 1              ⋆Determine the degree of p(x) based on the length of p⃗
3:     q⃗ ← p⃗
4:     for i ← 1 . . . n do
5:         q_i = q_i + α q_{i−1}
6:     end for
7:     r ← q_n
8:     q⃗ ← q⃗_{0:n−1}
9:     return q⃗, r
10: end procedure
```

Computational complexity: $2n$ operations required.

If $n < m$ then $r(x) = p(x)$ and $q(x) = 0$. This computation may be done recursively noting that

$$\frac{p(x)}{d(x)} = \frac{p_n}{d_m} x^{n-m} + \frac{R(x)}{d(x)},$$

where $R(x)$ is the degree $n-1$ polynomial that results from one step of the standard polynomial division algorithm. Thus, $q_{n-m} = \frac{p_n}{d_m}$, the remaining coefficients in $q(x)$ are found by completing the division $R(x)/d(x)$, and we stop when $\deg(R) < m$ and return this R as the remainder. Algorithm A.3.7 implements polynomial long division in this manner. Each call to PolyDiv costs $1 + 2m$ operations in lines 8 and 9 if $d(x)$ is of degree m. From the recursions, PolyDiv is called $n - m + 1$ times (one time to compute each of the $n - m + 1$ coefficients of $q(x)$, and once more in the base case which is essentially free), so the total operation count is $(n - m + 1)(1 + 2m) = 2m(n - m) + n + m + 1$.

Algorithm A.3.7 Polynomial division

1: **procedure** PolyDiv($\vec{p}, \vec{d}$) ⋆*Determine* $\vec{q}$ *and* r *such that* $\frac{p(x)}{d(x)} = q(x) + \frac{r(x)}{d(x)}$
2: $n \leftarrow$length($\vec{p}$) $- 1$ ⋆*Determine the degree of* $p(x)$ *based on the length of* $\vec{p}$
3: $m \leftarrow$length($\vec{d}$) $- 1$ ⋆*Determine the degree of* $d(x)$ *based on the length of* $\vec{d}$
4: **if** $n < m$ **then** ⋆*Base Case*
5: $\vec{r} \leftarrow \vec{p}, \quad \vec{q} \leftarrow 0$
6: **else** ⋆*Do one step of division*
7: $\vec{R} \leftarrow \vec{p}_{0:n-1}$
8: $q_0 \leftarrow p_n / d_m$
9: $\vec{R}_{n-1-m:n-1} \leftarrow \vec{R}_{n-1-m:n-1} - q_0 d_{0:m-1}$
10: $\vec{q}_1, \vec{r} \leftarrow$ PolyDiv($\vec{R}, \vec{d}$) ⋆*Recurse*
11: $\vec{q} \leftarrow (\vec{q}_1, q_0)$
12: **end if**
13: **return** $\vec{q}, \vec{r}$
14: **end procedure**

Computational complexity: $2m(n-m) + n + m + 1$ operations required.

A.3.1 Lagrange Interpolation

The set of points $\{(\alpha_i, \beta_i)\}_{i=0}^{n-1}$ uniquely defines a degree $n-1$ polynomial $p(x)$ such that $p(\vec{\alpha}) = \vec{\beta}$. Define

$$d(x) = \prod_{j=0}^{n-1}(x - \alpha_j) \quad \text{and} \quad d_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n-1}(x - \alpha_j) = \frac{d(x)}{(x - \alpha_i)},$$

for $0 \leq i < n$. Then $d_i(\alpha_j) = 0$ for $i \neq j$, and $p(x)$ may be written explicitly as

$$p(x) = \sum_{i=0}^{n-1} \beta_i \frac{d_i(x)}{d_i(\alpha_i)}.$$

This techniques is called *Lagrange interpolation* and is implemented in Algorithm A.3.8 via the above formula. The call to PolyRoots on line 4 costs n^2 operations. The call to PolyDivLin on line 7 costs $2n$ operations. The call to PolyEval on line 9 costs $2(n-1)$, and the division in that line costs one more. Line costs $2n$ operations. This yields a total operation count of $n^2 + n(2n + 2(n-1) + 1 + 2n) = 6n^2 - n.$ operations.

Algorithm A.3.8 Lagrange interpolation

Dependencies: PolyRoots defined in Algorithm A.3.4, PolyDivLin defined in Algorithm A.3.6, PolyEval defined in Algorithm A.3.1.

1: **procedure** LagrangeInterpolate($\vec{\alpha}, \vec{\beta}$) ⋆*Construct the coefficients of the ⋆polynomial $p(x)$ such that $p(\vec{\alpha}) = \vec{\beta}$*
2: **Require:** $|\vec{\alpha}| = |\vec{\beta}|$
3: $n \leftarrow$length($\vec{\alpha}$) ⋆*$p(x)$ will have degree $n-1$*
4: $\vec{d} \leftarrow$ PolyRoots($\vec{\alpha}$) ⋆*Construct $d(x) = \prod_{j=0}^{n-1}(x - \alpha_j)$*
5: Initialize $\vec{p}$ as a length n vector of zeros
6: **for** $i \leftarrow 0 \ldots n-1$ **do**
7: $\vec{d_i}, r \leftarrow$PolyDivLin($\vec{d}, \alpha_i$) ⋆*Construct $d_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1}(x - \alpha_j) = \frac{d(x)}{(x-\alpha_i)}$*
8: **Require:** $r = 0$ ⋆*Division should have a zero remainder*
9: $a \leftarrow \beta_i /$PolyEval($\vec{d_i}, \alpha_i$) ⋆*Construct $a = \beta_i / d_i(\alpha_i)$*
10: $\vec{p} \leftarrow \vec{p} + a\vec{d_i}$
11: **end for**
12: **return** $\vec{p}$
13: **end procedure**

Computational complexity: $6n^2 - n$ operations required.

A.4 Linear Algebra

A.4.1 Solving Linear Systems via LU Decomposition

A key component to many of the algorithms presented here is computing the solution $\vec{x}$ to the system of linear equation $A\vec{x} = \vec{b}$. There are a number of methods available to solve this problem; we will rely on solution via LU factorization. For the invertible square matrix A of size m, it is always possible to factor A as $A = LU$ where L is a lower-triangular matrix with ones on the diagonal, and U is an upper-triangular matrix. Then we may equivalently solve the system

$$LU\vec{x} = \vec{b}$$

by first solving

$$L\vec{y} = \vec{b}$$

for $\vec{y}$ via forward substitution, then solving

$$U\vec{x} = \vec{y}$$

for $\vec{x}$ via bask substitution.

Algorithm A.4.1 LU decomposition, [17, Algortihm 20.1]

```
1:  procedure LU(A)                          ⋆Compute the LU decomposition of A
2:      m ←size(A)                                    ⋆A is an m × m matrix
3:      U ← A
4:      L ← I_m                              ⋆Initialize L as the m × m identity matrix
5:      for c ← 0 ... m − 2 do
6:          for r ← c ... m − 1 do
7:              L_{r,c} ← U_{r,c}/U_{c,c}
8:              U_{r,c:m−1} ← U_{r,c:m−1} − L_{r,c} U_{c,c:m−1}
9:          end for
10:     end for
11:     return L, U
12: end procedure
```

Computational complexity: $\frac{2}{3}m^3 + \frac{3}{2}m^2 + \frac{5}{6}m - 3$ operations required.

The LU factorization is implemented in Algorithm A.4.1. Notice that the LU factorization presented here is done without partial pivoting. When computing over $\mathbb{R}$, the LU factorization is numerically unstable and is thus typically implemented using partial pivoting to control this instability. Since we assume that arithmetic is performed over a finite field, numerical instability is not a concern and we omit pivoting for the sake of simplicity. In Algorithm A.4.1, line 7 costs one operation each iteration, and line 8 costs $2(m-c)$ operations each iteration. This yields a total cost of

$$\begin{aligned}
\sum_{c=0}^{m-2}\sum_{r=c}^{m-1}(1+2(m-c)) &= \sum_{c=0}^{m-2}\left((m-c)+2(m-c)^2\right) \\
&= \sum_{j=2}^{m} j+2j^2 \\
&= \frac{m(m+1)}{2}-1+2\left(\frac{m(m+1)(2m+1)}{6}-1\right) \\
&= \frac{1}{6}m\left(3m+3+4m^2+6m+2\right)-3 \\
&= \frac{1}{6}m(4m^2+9m+5)-3 \\
&= \frac{2}{3}m^3+\frac{3}{2}m^2+\frac{5}{6}m-3
\end{aligned}$$

operations, where Exercise A.4.1 was used in evaluating the summation. For a detailed discussion of the LU factorization, see [17, Lecture 20].

Exercise A.4.1. Show that $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$. Hint: Use induction.

Algorithm A.4.2 takes L, U and $\vec{b}$ and solves the linear system $A\vec{x} = LU\vec{x} = \vec{b}$ as

Algorithm A.4.2 Solving a linear system with LU factorization

1: **procedure** SolveLU($L, U, \vec{b}$) ⋆*Solve $A\vec{x} = \vec{b}$, where A has the ⋆LU factorization $A = LU$*
2: **Require**: L square, lower triangular with ones on the diagonal
3: **Require**: U square, upper triangular
4: $m \leftarrow$size(L) ⋆*L and U are $m \times m$ matrices*
5: $\vec{y} \leftarrow \vec{b}$ ⋆*Solve $L\vec{y} = \vec{b}$ via forward substitution*
6: **for** $c \leftarrow 0 \ldots m-2$ **do**
7: **for** $r \leftarrow c \ldots m-1$ **do**
8: $y_r \leftarrow y_r - y_c L_{r,c}$
9: **end for**
10: **end for**
11: $\vec{x} \leftarrow \vec{y}$ ⋆*Solve $U\vec{x} = \vec{y}$*
12: **for** $c \leftarrow m-1 \ldots 1$ **do**
13: $x_c \leftarrow x_c / U_{c,c}$
14: **for** $r \leftarrow 0 \ldots c-2$ **do**
15: $x_r = x_r - x_c U_{r,c}$
16: **end for**
17: **end for**
18: **return** $\vec{x}$
19: **end procedure**

Computational complexity: $2m^2 + m - 3$ operations required.

discussed above. It costs

$$\begin{aligned}
\sum_{c=0}^{m-2}\sum_{r=c}^{m-1} 2 + \sum_{c=m-1}^{1}\left(1+\sum_{r=0}^{c-1}2\right) &= \left(\sum_{c=0}^{m-2} 2(m-c)\right) + (m-1) + \sum_{c=m-1}^{1} 2c \\
&= \left(2\sum_{j=2}^{m} j\right) + (m-1) + 2\sum_{c=1}^{m-1} c \\
&= 2\left(\frac{m(m+1)}{2} - 1\right) + (m-1) + 2\left(\frac{(m-1)m}{2}\right) \\
&= m^2 + m - 2 + m - 1 + m^2 - m \\
&= 2m^2 + m - 3
\end{aligned}$$

operations.

A.4.2 Computing the Row Reduced Echelon Form of a Matrix

Determining the null space of a matrix A is another frequent task from linear algebra. This may be accomplished by computing its row reduced echelon form, $\hat{A}$. A and $\hat{A}$ share the same null space, and it is easy to identify the null space of a matrix in row reduced echelon form. If A is size $m \times n$ with $m \leq n$ and A is full rank, then the row reduced echelon form of A has the form

$$\hat{A} = [I_m \quad \star\,]$$

where I_m is the $m \times m$ identity matrix, and $\star$ is some non-trivial matrix. The null space can be easily constructed from the columns of $\star$. If A is not full rank, the last few rows will be zero. Algorithm A.4.3 computes the row reduced echelon form of a given matrix A.

Algorithm A.4.3 costs

$$\begin{aligned}
\sum_{c=0}^{m-1}\left((n-c)+\sum_{r=0,r\neq c}^{m-1}2(n-c)\right) &= \sum_{c=0}^{m-1}\left((n-c)+2(n-c)(m-1)\right)\\
&= (2(m-1)+1)\sum_{c=0}^{m-1}(n-c)\\
&= (2m-1)\sum_{i=n-m+1}^{n} i\\
&= (2m-1)\left(\frac{n(n+1)}{2}-\frac{(n-m)(n-m+1)}{2}\right)\\
&= (2m-1)\left(\frac{n^2+n-(n^2-mn+n-mn+m^2-m)}{2}\right)\\
&= (2m-1)\left(\frac{2mn-m^2+m}{2}\right)\\
&= m\left(2mn-m^2+m-n+\frac{m}{2}-\frac{1}{2}\right)\\
&= (2m-1)mn-m(m-1)\left(m-\frac{1}{2}\right)
\end{aligned}$$

operations.

Algorithm A.4.3 Row reduced echelon form of a matrix

```
1: procedure RREF(A)                          ⋆Compute the row reduced echelon form of A
2:     m, n ←size(A)                                  ⋆A is an m × n matrix, m ≤ n
3:     for c ← 0 . . . m − 1 do
4:         k ← c + 1                                  ⋆k is a row to switch with if necessary
5:         while A_{c,c} == 0, k < m do               ⋆Swap rows so that A_{c,c} is non-zero
6:             if A_{k,c} == 0 then                   ⋆Look for row that is non-zero in column c
7:                 k ← k + 1
8:             else
9:                 Swap rows c and k in A
10:            end if
11:        end while
12:        if A_{c,c} = 0 then                        ⋆Swapping failed, return A
13:            return A
14:        end if
15:                                                   ⋆Zero out column c
16:        A_{c,c:n−1} ← A_{c,c:n−1}/A_{c,c}
17:        for r ← 0 . . . m − 1, r ≠ c do
18:            A_{r,c:n−1} ← A_{r,c:n−1} − A_{r,c} ∗ A_{c,c:n−1}
19:        end for
20:    end for
21:    return A
22: end procedure
```

Computational complexity: $(2m-1)mn - m(m-1)\left(m - \frac{1}{2}\right)$ operations required.

Appendix B
Modules and Gröbner Bases

What follows is a brief introduction to Modules and Gröbner bases as they are relevant to the rest of this document. This discussion is certainly not exhaustive. For further discussion see [6, 3, 8, 12].

B.1 Modules

Definition B.1.1. ([3, Definition 3.27]) Let R be a ring. A **module** M is an additive Abelian group with an additional operation $\circ : R \times M \to M$, called scaler multiplication , such that for all $\alpha, \beta \in R$ and $a, b \in M$, the following hold:

i. $\alpha \circ (a + b) = \alpha \circ a + \alpha \circ b$,

ii. $(\alpha + \beta) \circ a = \alpha \circ a + \beta \circ a$,

iii. $(\alpha \cdot \beta) \circ a = \alpha \circ (\beta \circ a)$, and

iv. $1 \circ a = a$

We will often denote scalar multiplication by $\alpha a = \alpha \circ a$.

Definition B.1.2. Let $\mathbb{F}$ be a finite field. We will often be interested in the module $\mathcal{M} = \mathbb{F}[x] \times \mathbb{F}[x]$ over the ring $\mathbb{F}[x]$. We denote an element in $\mathcal{M}$ by (f, g) and for any $p \in \mathbb{F}[x]$ define scalar multiplication by $p(f, g) = (pf, pg)$.

Exercise B.1.3. Show that $\mathcal{M}$ is a module, as claimed.

B.2 Term Orders

An element in $\mathcal{M}$ of the form $(x^i, 0)$ or $(0, x^i)$ for $i \in \mathbb{N}$ is called a **term**. Every element in $\mathcal{M}$ may be written as a linear combination of terms over $\mathbb{F}$. Let $\mathcal{T} = \{(0, x^i), i \geq 0\} \cup \{(x^i, 0), i \geq 0\}$ be the set of terms in $\mathcal{M}$. It will be useful to have a strict well-ordering on the elements of $\mathcal{T}$.

Definition B.2.1. ([8, Part II]) For $a \in \mathbb{Z}$, define the **term order** $<_a$ on $\mathcal{T}$ by

1. $(x^i, 0) <_a (x^j, 0) \iff i < j$
2. $(0, x^i) <_a (0, x^j) \iff i < j$
3. $(x^i, 0) <_a (0, x^j) \iff i \leq j + a$
4. $(0, x^i) <_a (x^j, 0) \iff j + a < i$.

Theorem B.2.2. *$<_a$ is a strict well-ordering of $\mathcal{T}$: for $s, t, u \in \mathcal{T}$,*

(a) (Transitive) If $s <_a t$ and $t <_a u$ then $s <_a u$.

(b) (Trichotomous) Either $s <_a t$, $t <_a s$, or $s = t$.

(c) (Well-founded) If $\mathcal{S} \subseteq \mathcal{T}$, then $\mathcal{S}$ has a least element.

As well, $<_a$ is compatible with multiplication by x^i: $s <_a t$ if and only if $x^i s <_a x^i t$.

Proof. Left as an exercise for the reader. □

Example B.2.3. $a = -1$ imposes the following ordering on $\mathcal{T}$:

$$(0,1) <_{-1} (1,0) <_{-1} (0,x) <_{-1} (x,0) <_{-1} (0,x^2) <_{-1} \dots$$

Theorem B.2.4. *For any term ordering $<_a$, every element $f \in \mathcal{M}$ may be represented uniquely as $f = \sum_{i=0}^{b} d_i f_i$, $d_i \in \mathbb{F}$ and f_i a term in $\mathcal{M}$, such that $f_i <_a f_j$ for $i < j$. In this representation, the **leading term** of f with respect to $<_a$ is f_b and is denoted $LT(f)$.*

Example B.2.5. Let $\mathbb{F} = GF(2^8)$ and consider $f = (5x^2 + 182x^8 + 217x^5, 42x^7 + x^2) \in \mathcal{M}$. With respect to the term ordering $<_0$, f is uniquely represented as

$$f = 5(x^2, 0) + (0, x^2) + 217(x^5, 0) + 42(0, x^7) + 182(x^8, 0),$$

and $LT(f) = (x^8, 0)$. With respect to the term ordering $<_2$, f is uniquely represented as

$$f = 5(x^2, 0) + (0, x^2) + 217(x^5, 0) + 182(x^8, 0) + 42(0, x^7),$$

and $LT(f) = (0, x^7)$. With respect to the term ordering $<_{-1}$, f is uniquely represented as

$$f = (0, x^2) + 5(x^2, 0) + 217(x^5, 0) + 42(0, x^7) + 182(x^8, 0),$$

and $LT(f) = (x^8, 0)$.

Definition B.2.6. We say that $f, g \in \mathcal{M}$ have **leading terms on opposite sides** if $LT(f) = (x^i, 0)$ and $LT(g) = (0, x^j)$, or equivalently if $LT(f) = (0, x^i)$ and $LT(g) = (x^j, 0)$. We say that $f, g \in \mathcal{M}$ have **leading terms on the same sides** if $LT(f) = (x^i, 0)$ and $LT(g) = (x^j, 0)$, or equivalently if $LT(f) = (0, x^i)$ and $LT(g) = (0, x^j)$.

Definition B.2.7. For $\mathcal{S} \subset \mathcal{M}$, let $\langle LT(\mathcal{S}) \rangle$ denote the submodule generated by the leading terms of the elements of $\mathcal{S}$. That is, $f \in \langle LT(S) \rangle$ iff $f = \sum_{i=0}^{b} c_i LT(s_i)$ for some $c_i \in \mathbb{F}[x]$, $s_i \in \mathcal{S}$, and $b \in \mathbb{N}$.

Definition B.2.8. Let M be a submodule of $\mathcal{M}$. We say that $f \in M$ is a **minimal element** of M if $LT(f)$ is minimal amongst $LT(M)$ with respect to the term order $<_a$. That is, f is a minimal element of M iff for any $g \in LT(M)$, $LT(f) \leq_a LT(g)$.

B.3 Gröbner Bases

Definition B.3.1. ([3, Definition 3.5]) Let M be a submodule of $\mathcal{M}$, and $G \subseteq M$. G is called a **generating set** for M if for all $f \in M$, there exists $n \in \mathbb{N}$, $\lambda_1, \ldots, \lambda_n \in \mathbb{F}$, and $g_1, \ldots g_n \in G$ such that

$$f = \sum_{i=1}^{n} \lambda_i g_i.$$

Definition B.3.2. Let M be a submodule of $\mathcal{M}$, and $G = \{g_1, \ldots, g_a\} \subset M$ be a finite generating set for M. If $\langle LT(G)\rangle = \langle LT(M)\rangle$, then G is said to be a **Gröbner basis** of M.

Theorem B.3.3. *Let M be a submodule of $\mathcal{M}$, and $G = \{g_1, g_2\} \subset M$ be a generating set for M. G is a Gröbner basis for M iff g_1 and g_2 have leading terms on opposite sides.*

Proof. See [8, Corollary 2.3]. □

Lemma B.3.4. *Let M be a submodule of $\mathcal{M}$, $\{g_1, g_2\}$ be a Gröbner basis for M, and assume $g_1 <_a g_2$. Let f be a minimal element of M. Then f is a scalar multiple of g_1.*

Proof. Since $\{g_1, g_2\}$ is a basis for M, there exists $c, d \in \mathbb{F}[x]$ such that $f = cg_1 + dg_2$. If $d \neq 0$, then $LT(f) \geq_a x^{\deg(d)} LT(g_2)$ and $LT(g_1) <_a LT(f)$ violating the minimality of f. Thus $f = cg_1$, and $LT(f) = x^{\deg(c)} LT(g_1)$. If $\deg(c) \neq 0$, then $LT(g_1) <_a LT(f)$ violating the minimality of f. Thus $\deg(c) = 0$ and f is a scalar multiple of g_1. □

Appendix C
Notation

Notation used in this text:

- $a \mid b$: a divides b
- $a \nmid b$: a does not divide b
- $\vec{x}^T$ = the transpose of the vector $\vec{x}$
- A^T = the transpose of the matrix A
- $|\mathcal{S}|$ = cardinality of the set $\mathcal{S}$
- $|\vec{x}|$ = the length of the vector $\vec{x}$
- $\overline{\mathcal{S}}$ = the complement of the set $\mathcal{S}$, typically $\overline{\mathcal{S}} = \{0, 1, \ldots, n\} \backslash \mathcal{S}$
- $\vec{x}_{\mathcal{S}} = (x_{s_0}, \ldots, x_{s_{n-1}})$ if $\mathcal{S} = \{s_0, s_1, \ldots s_{n-1}\}$
- $\vec{x}_{a:b} = (x_a, x_{a+1}, \ldots, x_b)$
- $M_{\mathcal{A},\mathcal{B}}$ = the matrix composed of the rows of M at indices in $\mathcal{A}$ and the columns of M at the indices in $\mathcal{B}$, where M is a matrix and $\mathcal{A}$ and $\mathcal{B}$ are ordered lists
- $M_{\mathcal{A},:}$ = the matrix composed of the rows of M at indices in $\mathcal{A}$ and all columns of M, where M is a matrix and $\mathcal{A}$ is an ordered list
- $M_{:,\mathcal{B}}$ = the matrix composed of all rows of M and the columns of M at the indices in $\mathcal{B}$, where M is a matrix and $\mathcal{B}$ is an ordered list
- $\deg(f)$ = degree of the polynomial $f(x)$

- $\deg(\beta) = \max_{(a,b)\in\beta} \max(\deg(a), \deg(b))$, for $\beta \subset \mathbb{F}[x] \times \mathbb{F}[x]$
- $d(\vec{a}, \vec{b})$ = Hamming distance between $\vec{a}$ and $\vec{b}$
- $d_{\mathcal{I}}(\vec{a}, \vec{b})$ = Hamming distance between $\vec{a}$ and $\vec{b}$ restricted by the set of indices $\mathcal{I}$
- $\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$

	Integers
l	number of bits in a character
k	number of characters in a message
n	number of characters in a codeword
r	$n-k$, number of characters in a codeword which are redundant
d	$n-k+1$, distance of code
f	number of erasures in $\vec{r}$
f_m	number of erasures in $\vec{r}$ that occur in message indices, $f_m = \lvert\{0,1,\ldots k-1\} \cap \mathcal{F}\rvert$
t	number of errors in $\vec{r}$
$\hat{t}$	$\lfloor \frac{n-k-f}{2} \rfloor$, maximum number of errors within the error bound
δ	$\hat{t}-t$, the difference between the maximum number of errors correctable and the actual number of errors
	Vectors
$\vec{\alpha}$	$(\alpha_0, \ldots, \alpha_{n-1})^T$, generating vector for the code
$\vec{m}$	$(m_0, \ldots, m_k)^T$, message
$\vec{c}$	$(c_0, \ldots, c_{n-1})^T$, codeword
$\vec{r}$	$(r_0, \ldots, r_{n-1})^T$, received word
$\vec{e}$	$(e_0, \ldots, e_{n-1})^T$, error: $\vec{r} = \vec{c} + \vec{e}$
	Matrices
$V^{a,b}(\vec{\alpha})$	The $a \times b$ Vandermonde matrix defined by $\vec{\alpha}$, see Section 3.1
	Sets
C	the code, *i.e.* the set of all possible codewords
$\mathcal{F}$	set of indices of erasures in $\vec{r}$, $\lvert\mathcal{F}\rvert = f$
$\mathcal{W}$	set of indices that are "working" in $\vec{r}$, $\mathcal{W} = \overline{\mathcal{F}}$, $\lvert\mathcal{W}\rvert = n-f$
$\mathcal{E}$	set of indices of errors, $\mathcal{E} = \{i : e_i \neq 0\}$, $\lvert\mathcal{E}\rvert = t$
	Polynomials
$m(x)$	encoding polynomial for non-systematic codes, $m(x) = \sum_{i=0}^{k-1} m_i x^i$, and $c_i = m(\alpha_i)$.
$\mu(x)$	encoding polynomial for systematic codes defined by $\mu(\alpha_i) = m_i$ for $0 \leq i < k$. Then $c_i = m(\alpha_i)$
$g(x)$	monic error locator polynomial, $g(x) = \prod_{i \in \mathcal{E}} (x - \alpha_i)$
$q(x)$	error locator polynomial, $q(x) = cg(x)$ for some constant c, $g(\alpha_i) = 0 \Leftrightarrow e_i \neq 0 \Leftrightarrow i \in \mathcal{E}$.
	Fields
$GF(2^a)$	The Galois field (finite field) of size 2^a
$\mathbb{F}$	A field, typically $GF(2^l)$

TABLE C.1. Table of Symbols

Index

References

[1] J. Adámek, *Foundations of coding: theory and applications of error-correcting codes, with an introduction to cryptography and information theory.* Wiley, 1991.

[2] M. Anderson and S. E. Mann, "Accelerated Erasure Coding System and Method," U.S. Patent 20 130 173 996, 12 30, 2011.

[3] T. Becker and V. Weispfenning, *Gröbner Bases: A Computational Approach to Commutative Algebra.* Springer-Verlag, 1993.

[4] E. R. Berlekamp, *Algebraic Coding Theory.* McGraw-Hill, Inc., 1968.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw Hill, 2001.

[6] D. Cox, J. Little, and D. O'shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*, 2nd ed. New York: Springer-Verlag, 1997.

[7] D. S. Dummit and R. M. Foote, *Abstract Algebra*, 3rd ed. John Wiley & Sons, Inc., 2004.

[8] P. Fitzpatrick, "On the key equation," *Information Theory, IEEE Transactions on*, vol. 41, no. 5, pp. 1290–1302, 1995.

[9] P. Gemmell and M. Sudan, "Highly resilient correctors for polynomials," *Information Processing Letters*, vol. 43, no. 4, pp. 169–174, Sep. 1992.

[10] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, pp. 147–160, Apr. 1950.

[11] D. G. Hoffman, D. A. Leonard, C. C. Linder, K. T. Phelps, C. A. Rodger, and J. R. Wall, *Coding Theory: The Essentials.* Marcel Dekker, 1991.

[12] S. Jennings, "Grobner basis view of Welch-Berlekamp algorithm for Reed-Solomon codes," *Communications, IEE Proceedings-*, vol. 142, no. 6, pp. 349–351, 1995.

[13] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *SIGMOD Rec.*, vol. 17, no. 3, pp. 109–116, 1988.

[14] J. S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems," *Software – Practice & Experience*, vol. 27, no. 9, pp. 995–1012, Sep. 1997.

[15] J. S. Plank and Y. Ding, "Note: Correction to the 1997 Tutorial on Reed-Solomon Coding," *Software – Practice & Experience*, vol. 35, no. 2, pp. 189–194, Feb. 2005.

[16] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Indust. Appl. Math.*, vol. 8, pp. 300–304, 1960.

[17] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, Jun. 1997.

[18] L. R. Welch. (1997) The Original View of Reed-Solomon Codes. [Online]. Available: http://csi.usc.edu/PDF/RSoriginal.pdf

[19] L. R. Welch and E. R. Berlekamp, "Error correction for algebraic block codes," U.S. Patent 4 633 470, Dec. 30, 1986.