2008

MULTIVARIATE LIST DECODING OF EVALUATION CODES WITH A GRÖBNER BASIS PERSPECTIVE

Philip Busse
University of Kentucky, pbusse@ms.uky.edu

ABSTRACT OF DISSERTATION

Philip Busse

The Graduate School
University of Kentucky
2008

MULTIVARIATE LIST DECODING OF EVALUATION CODES WITH A GRÖBNER BASIS PERSPECTIVE

ABSTRACT OF DISSERTATION

A dissertation submitted in partial
fulfillment of the requirements for
the degree of Doctor of Philosophy
in the College of Arts and Sciences
at the University of Kentucky

By
Philip Busse
Lexington, Kentucky

Director: Dr. Uwe Nagel, Professor of Mathematics
Lexington, Kentucky 2008

ABSTRACT OF DISSERTATION

MULTIVARIATE LIST DECODING OF EVALUATION CODES WITH A GRÖBNER BASIS PERSPECTIVE

This dissertation presents several new aspects of list decoding evaluation codes beyond the Guruswami-Sudan error-correction radius, $1 - \sqrt{\kappa}$. We present an overview of the development of list decoding from Sudan's archetypal interpolation-factorization algorithm up to cutting edge results on capacity achieving codes. A recent result in this field is an interpolation algorithm ([25]) that efficiently computes a Gröbner basis. Another important result is a multivariate interpolation algorithm [18] that yields codes with error rates approaching list decoding capacity. Our primary contribution is producing a hybrid of the two methods and obtaining an explicit multivariate interpolation algorithm with proven complexity for list decoding folded Reed-Solomon codes. It was known that the multivariate interpolation step in w variables for a v-folded Reed-Solomon code of block length N could be performed in time $(Nv)^{\mathcal{O}(w)}$ ([17]), but an explicit algorithm for which precise complexity bounds could be measured was lacking. Our work establishes constructively that multivariate list decoding can be performed using $\mathcal{O}((Nv)^2 m\ell(e\ell)^{3w})$ field operations, where m is the chosen multiplicity for the interpolation step, κ is the rate of the code, and $\ell \in \mathcal{O}((m+w)\kappa^{-1/(w+1)})$ is a $\boldsymbol{y}$-degree bound on the interpolation polynomial. This is the most specific complexity estimate given so far on the interpolation problem. Thus, our algorithm sets a benchmark on the complexity of multivariate list decoding for Reed-Solomon-based evaluation codes.

KEYWORDS: list decoding, multivariate interpolation, Gröbner bases, Reed-Solomon codes, evaluation codes

Author's signature: Philip Busse

Date: June 26, 2008

MULTIVARIATE LIST DECODING OF EVALUATION CODES WITH A GRÖBNER BASIS PERSPECTIVE

By
Philip Busse

Director of Dissertation: Uwe Nagel

Director of Graduate Studies: Qiang Ye

Date: June 26, 2008

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Doctor's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this dissertation for use by its patrons is expected to secure the signature of each user.

Name	Date

DISSERTATION

Philip Busse

The Graduate School
University of Kentucky
2008

MULTIVARIATE LIST DECODING OF EVALUATION CODES WITH A GRÖBNER BASIS PERSPECTIVE

DISSERTATION

A dissertation submitted in partial
fulfillment of the requirements for
the degree of Doctor of Philosophy
in the College of Arts and Sciences
at the University of Kentucky

By
Philip Busse
Lexington, Kentucky

Director: Dr. Uwe Nagel, Professor of Mathematics
Lexington, Kentucky 2008

This work is dedicated with love to my family and friends for their endless support and encouragement throughout this endeavor, especially Mom and Dad. Above all, I dedicate this to God, who makes my paths straight.

ACKNOWLEDGMENTS

I thank my adviser, Prof. Uwe Nagel, for his time and invaluable guidance throughout the course of my graduate studies at the University of Kentucky. I thank his wife, Prof. Heide Gluesing-Luerssen for instructing me in coding theory and providing much useful feedback on the dissertation. I thank my other professors for their tutelage, especially professors Richard Ehrenborg and Andrew Klapper for their useful comments on this thesis and service on my defense committee (in addition to being excellent instructors). I thank Prof. Richard Poss for introducing me to coding theory. I thank Kwankyu Lee for his correspondence on list decoding. I thank Prof. Joachim Rosenthal for his sage advice to explore multivariate list decoding schemes, which turned out to be an essential part of this work. I thank Prof. Arny Stromberg for persistently reminding me to write-up more results. I thank the congregation of Good Shepherd Lutheran Church for their support throughout this venture. I thank my family for their love and encouragement. Finally, I thank God for His timely inspirations and divine wisdom.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Symbols

$\in$	is an element of
$\notin$	is not an element of
$\subseteq$	is a subset of
$\subset$	is a proper subset of
$A \backslash B$	set difference
$A \cup B$	set union
$A \cap B$	set intersection
$\lvert A \rvert$	the number of elements in set A
A^n	the set of n-tuples of A
$f : A \to B$	f is a function from set A to set B
$f(x)$	the image of x under f
$H_q(p)$	the q-ary entropy function; $H_q(p) = -p \log_q p - (1-p) \log_q (1-p) + p \log_q (q-1)$
$\mathbb{N}$	the set of natural numbers (positive integers)
$\mathbb{N}_0$	the set of nonnegative integers
$\mathbb{Z}$	the set of integers
$\mathbb{N}_k$	the set of integers greater than or equal to $k \in \mathbb{Z}$
$\mathbb{Z}_n$	the set of integers modulo n
$[n]$	the set of integers $\{1, \ldots, n\}$
$\mathbb{Q}$	the set of rational numbers
$\mathbb{R}$	the set of real numbers
$\mathbb{R}_{\geq 0}$	the set of nonnegative real numbers
$\mathbb{R}_+$	the set of positive real numbers
$B_\tau(\boldsymbol{r})$	the Hamming sphere of radius τ about $\boldsymbol{r} \in \mathbb{F}^n$
$T_=(m)$	the set of pairs of tuples $(a, b) \in \mathbb{N}_0^2$ such that $a + b = m$. Similarly, for $T_\leq(m)$, $a + b \leq m$, etc.
$W_=(m)$	the set of w-tuples $(a_1, \ldots, a_w) \in \mathbb{N}_0^w$ for some $w \in \mathbb{N}$ such that $\sum a_i = m$. Similarly, for $W_\leq(m)$, $\sum a_i \leq m$, etc.
$\binom{n}{k}$	binomial coefficient; the number of combinations of k objects selected from n choices without replacement
$\left(\!\binom{n}{k}\!\right)$	multinomial coefficient; the number of combinations of k objects selected from n choices with replacement
$n!$	n factorial
$(n)_k$	n falling factorial k; the product $n(n-1)(n-2)\cdots(n-k+1)$
$S(n)$	the symmetric group on $[n]$; the set of permutations of $\{1, \ldots, n\}$.
$S(A)$	the symmetric group on the set A; the set of all permutations on the elements of A.
$a \mid b$	a divides b
$\gcd(a, b)$	the greatest common divisor of a and b
$a \equiv_n b$	a is congruent to b modulo n

$\lvert(a_1,\ldots,a_n)\rvert$	the 1-norm of $(a_1,\ldots,a_n)$; $\sum_{i=1}^{n}\lvert a_i\rvert$
$\lvert(a_1,\ldots,a_n)\rvert_{\boldsymbol{u}}$	the $\boldsymbol{u}$-weighted 1-norm of $(a_1,\ldots,a_n)$; $\sum_{i=1}^{n} u_i\lvert a_i\rvert$
$\mathrm{wt}(a_1,\ldots,a_n)$	Hamming weight (0-norm) of $(a_1,\ldots,a_n)$, $\lvert\{i\in[n] \mid a_i\neq 0\}\rvert$
$\mathbb{F}$	a finite field
$\mathbb{F}^\times$	the multiplicative group of $\mathbb{F}$
$\mathbb{F}_q$	the Galois field of q elements
$\mathbb{F}[x]$	the polynomial ring over $\mathbb{F}$ in x
$\mathbb{F}[x]_k$	the set of polynomials in x of degree less than k
$\mathbb{F}[x_1,\ldots,x_w]$	the set of polynomials in $x_1,\ldots,x_w$
$\boldsymbol{y}^{\boldsymbol{a}}$	the monomial $y_1^{a_1}\cdots y_w^{a_w}$
$\mathbb{F}[x,\boldsymbol{y}]$	the set of polynomials in x and $y_1,\ldots,y_w$ for some $w\in\mathbb{N}$
$\mathbb{F}[x,\boldsymbol{y}]_k$	the polynomials in $\mathbb{F}[x,\boldsymbol{y}]$ that have no term with total degree in $y_1,\ldots,y_w$ greater than k [1]
$[x_1,\ldots,x_n]$	the monomials in $x_1,\ldots,x_n$
$\langle g_1,\ldots,g_s\rangle$,	the ideal or module (depending on context) generated by $\{g_1,\ldots,g_s\}$
$\deg_{k-1} f$	the $(1,k-1)$-weighted degree of $f(x,y)\in\mathbb{F}[x,y]$; $\max\{i+(k-1)j\}$ taken over all monomials x^iy^j in f.
$\deg_{\boldsymbol{u}} f$	the $\boldsymbol{u}$-weighted degree of $f(x,\boldsymbol{y})\in\mathbb{F}[x,\boldsymbol{y}]$ ($\boldsymbol{u}\in\mathbb{R}^{w+1}$); $\max\{u_0\cdot i+u_1\cdot j_1+\cdots+u_w\cdot j_w\}$ taken over all monomials $x^iy_1^{j_1}\cdots y_w^{j_w}$ in f.
$\deg_x f$	the x-degree of $f(x,\boldsymbol{y})\in\mathbb{F}[x,\boldsymbol{y}]$; $\max\{i\}$ taken over all monomials $x^iy_1^{j_1}\cdots y_w^{j_w}$ in f.
$\deg_{\boldsymbol{y}} f$	the $\boldsymbol{y}$-degree of $f(x,\boldsymbol{y})\in\mathbb{F}[x,\boldsymbol{y}]$; $\max\{j_1+\cdots+j_w\}$ taken over all monomials $x^iy_1^{j_1}\cdots y_w^{j_w}$ in f.
$\mathcal{O}(f(n))$	the set of all functions $g:\mathbb{N}\to\mathbb{R}$ that are bounded above by some constant multiple of $f(n)$ for all but finitely many values of n
$\Omega(f(n))$	the set of all functions $g:\mathbb{N}\to\mathbb{R}$ that are bounded below by some constant multiple of $f(n)$ for all but finitely many values of n
$\Theta(f(n))$	$\mathcal{O}(f(n))\cap\Omega(f(n))$
$o(f(n))$	$\mathcal{O}(f(n))\backslash\Theta(f(n))$
$\omega(f(n))$	$\Omega(f(n))\backslash\Theta(f(n))$

[1] It is unfortunate that the notation $\mathbb{F}[x,\boldsymbol{y}]_k$ allows for equality to k whereas the definition of $\mathbb{F}[x]_k$ does not, but that is how we use them so please forgive and tolerate the discrepancy.

Chapter 1
Introduction

The intent of this disquisition is to combine some of the recent results on the problem of list decoding evaluation codes in order to enhance and clarify the progress that these ideas have begotten. A secondary objective will be to ameliorate the practical aspects of these results by providing an explicit framework for implementing them.

1.1 List Decoding Synopsis

List decoding differs from conventional minimum distance decoding in that the decoder does not need to return a unique closest code word to a given received word. Rather, it returns a list of all codewords within some bounded distance. Indeed, for a code of minimum distance d, a minimum distance decoder will always fail to correct some error of weight $\lceil d/2 \rceil$ for some codeword, limiting the fraction of errors a decoder can correct in codes of rate κ to approximately $\frac{1-\kappa}{2}$ errors. A list decoder, on the other hand, is not bounded by the unique decoding radius of the code because an output of several codewords is still considered correct as long as the codeword that was sent is among them. The limitations on list decoding derive instead from the time complexity of producing the list of codewords within distance τ of the received word. It is known that for codes of rate κ, the length of the worst-case list size grows exponentially in the length n of the code for any decoding radius $\tau > n(1-\kappa)$. Conversely, the list-size is polynomially bounded for $\tau < n(1-\kappa)$. For this reason, $1-\kappa$ is deemed the list decoding capacity for the fraction of errors that we can hope to 'correct' (list-wise) up to in codes of rate κ. The capacity is only attainable in the limit of large alphabets. For a fixed alphabet size, there exist codes of any rate $\kappa < 1 - H_q(p)$ (where $H_q(p)$ is the q-ary entropy function) for which we can list decode up to an error rate of p. It is interesting that this capacity, which is essentially a complexity bound, corresponds to the channel capacity, which is a probabilistic bound, for a q-ary channel where the probability of error on any given symbol is p. There is some equivocation retained in the output of a list decoder since, in general, it returns a list and not a single codeword. However, based on the way that Hamming spheres pack in n-dimensional space for radii up to $n(1-\kappa)$, it has been shown that there exist codes of rate κ (Reed-Solomon codes among them) for which *most* errors of weight less than $n(1-\kappa)$ yield a received word with a unique codeword within distance $n(1-\kappa)$. For such codes a list decoder will output a unique closest codeword within radius $\tau < n(1-\kappa)$ of the received word with high probability, which suggests that list decoding has the potential to elicit real coding schemes approaching the Shannon limit for channel capacity.

1.2 Previous Results

A crucial ingredient for list decoding is having the means to decode a received word in a runtime that is polynomial in the length of the code. List decoding was first used for theoretical purposes by Peter Elias and Reiffen Wozenkraft (separately) in the late 1950's to prove matching upper and lower bounds on the probability of a decoding error occurring over a binary symmetric channel. However, no practical method for performing it was given until 1996 when Madhu Sudan proposed a two-step algorithm for list decoding Reed-Solomon codes. The first step involved interpolating a bivariate polynomial through points determined by the received word and the code locators of the RS code. The second step was to factorize the interpolated polynomial and obtain the list of nearest codewords from its roots (corresponding to polynomials in one variable). The algorithm ran efficiently in time $\mathcal{O}(n^3)$ and could correct a fraction of up to $1 - \sqrt{2\kappa}$ errors. A few years later, Sudan and his student, Venkatesan Guruswami, improved the error rate to $1 - \sqrt{\kappa}$ errors by interpolating the same points as before but with some positive multiplicity. The next asymptotic leap in list decoding performance came in 2005 when Alexander Vardy and his student, Farzad Parvaresh, gave a trivariate interpolation-based algorithm for so-called 'interleaved' Reed-Solomon codes in which the codewords near the received word corresponded to algebraically related roots of the interpolation polynomial $Q(x, y, z)$ that were polynomials in x. More specifically, the messages were pairs $(f(x), g(x))$ that satisfied a certain algebraic constraint, such as $f(x)^d = g(x) \bmod h(x)$, and the root-finding step was to find pairs $(f(x), g(x))$ satisfying the algebraic constraint and also the constraint $Q(x, f(x), g(x)) = 0$. This improved the best-known list decoding error rates for low rate codes ($\kappa < 1/16$), but the modification of sending two polynomials necessitates sending twice as much information and precludes the possibility of constructing codes with an information rate of more than one half in this scheme. However, the idea of raising the number of interpolation variables was novel and if the code and the message pairs are chosen more carefully (namely, the message polynomials are chosen to be horizontal stretches of each other, so their evaluations are cyclic shifts of each other) then the loss in information rate can be avoided.

A recent result that we will build upon is an explicit construction of codes that allow for error rates that asymptotically approach the list decoding capacity. These codes are known as folded Reed-Solomon codes and a polynomial time algorithm for list decoding them was presented by Guruswami and his student, Atri Rudra, in 2007 that was based on the trivariate interpolation algorithm of Parvaresh and Vardy but avoided the rate-loss factor mentioned in the previous paragraph. The key idea they came up with was to use the fact that for a primitive element $a \in \mathbb{F}_q$, any polynomial $f \in \mathbb{F}_q[x]$ of degree less than $q-1$ satisfies $f(ax) = f(x)^q \bmod x^{q-1} - a$ and have the algebraic constraint of the Parvaresh and Vardy method be $f(x)^q = g(x) \bmod x^{q-1} - a$. Then a message may take the form $(f(x), f(ax))$. If a is used to define the code locators of the underlying RS code, then the codeword given by $f(ax)$ is just the cyclic shift left of the codeword produced by $f(x)$. The additional information needed for the trivariate interpolation problem is therefore inherent within the codeword of the first polynomial. Thus the information rate loss factor is redressed.

Another result that we will utilize is a Gröbner basis based algorithm for performing the interpolation step of Sudan's list decoding method for Reed-Solomon codes. The algorithm, presented by Kwankyu Lee and Michael O'Sullivan in 2006, looks at the interpolation step as a problem of finding a polynomial in an ideal of small degree (the conventional approach is to solve a linear system of homogeneous equations) by computing a Gröbner basis. Computing such a basis can be a costly task in general but Lee and O'Sullivan were able to polynomially bound the complexity by restricting the ideal to a finitely generated submodule and computing a Gröbner basis there. Lee and O'Sullivan furthermore showed that their algorithm was an efficient generalization of the Berlekamp-Massey algorithm, which has been the preeminent method for minimum-distance decoding of Reed-Solomon codes since its inception in 1969.

1.3 Contribution of this Dissertation

The goal of this dissertation was to see how Lee and O'Sullivan's Gröbner basis based interpolation algorithm could be adapted to list decode folded Reed-Solomon codes and other extensions of RS codes. Particularly for the folded version of RS codes, but also for the other kinds, it was desirable to modify the algorithm to solve a multivariate interpolation algorithm. Multivariate interpolation seems to be necessary at this point in order to achieve error rates beyond the Guruswami-Sudan bound of $1 - \sqrt{\kappa}$ and, moreover, in order to approach list decoding capacity. The Gröbner basis algorithm was successfully adapted to this end by following the method of Guruswami and Rudra. It may be noted that in their presentation they only give the trivariate formulation of the interpolation problem and then observe at the end that it can be extended to more variables. We, on the other hand, present the multivariate interpolation problem in its most general form. Furthermore, we provide new and more explicit algorithms for performing both the interpolation and the root-finding steps of Guruswami and Rudra's algorithm, and so we are able to obtain more specific complexity bounds on the procedures. These new results are developed in Chapter 4, the zenith of this thesis. The chapter covers most of what is new in this work and is highlighted by a new algorithm for list decoding capacity-achieving codes and a precise complexity analysis of the algorithm. Source code for an implementation of the algorithm on the SINGULAR Computer Algebra System platform in all of its glorious generality is provided in the appendix.

1.4 Structure of the Discourse

There are three chapters partitioning the content of this dissertation. The first chapter naturally reviews the background topics that are germane to our multivariate list decoding algorithm. First and foremost, a short survey of coding theory is undertaken to establish important definitions and prepare some motivation for list decoding. With the exception of folded Reed-Solomon codes (introduced in Chapter 4) all of the codes that will be relevant to this paper are introduced in Section 2.1.2, including Reed-Solomon codes, alternant codes (which, for simplicity, we define a little differently than usual), and BCH codes. For BCH codes, we present in Lemma 2.1.6 an

explicit representation for them as alternant codes which will be crucial for applying our decoding algorithm to them. Chapter 2 also covers the fundamentals of Gröbner bases - monomial orders, division algorithms, S-polynomials, Buchberger's Criterion, and Buchberger's Algorithm - to establish a base for the algebraic side of the Gröbner basis-based interpolation algorithm. Gröbner bases here are presented in the more general context of submodules rather than the more specific setting of polynomial ideals because we will actually be seeking a Gröbner basis of a module, not a ring. Lastly for background, we present in Section 2.3 a bivariate root-finding algorithm based on the one in [33] but modified slightly to suit the root-finding problem for multivariate interpolation.

Chapter 3 lays the groundwork for list decoding. The introduction gives some motivation for it and sets some definitions peculiar to the field. Section 3.2 thoroughly analyzes the asymptotic complexity bounds of list decoding and establishes the list decoding capacity as the bound on rates of codes that are to be efficiently list decoded up to a certain radius. Section 3.3 presents Sudan's original list decoding algorithm and the following two sections analyze its performance. A discussion ensues regarding the concept of multiplicity. It is followed by a description and short analysis of the Guruswami-Sudan algorithm.

The last and most important chapter, Chapter 4, begins with an introduction to folded Reed-Solomon codes, which are really the same as Reed-Solomon codes except that they are viewed over a larger alphabet. In Section 4.3, multi-degree definitions and notations are established and some preparatory analysis is done for the multivariate interpolation algorithm. The actual algorithm is presented at the end of the section. In the following section, 4.4, an algebraic relation between the polynomials corresponding to the roots of the interpolation polynomial is shown and a simple algorithm is given for finding them that utilizes the root-finding algorithm of Section 2.3. The heart of our work culminates in Section 4.5. Here we take a more specific approach to the interpolation step by viewing it as a problem of locating a small polynomial of a particular ideal. We then restrict this ideal (a la Lee and O'Sullivan) to a submodule of the polynomial ring where all variables except x have some bounded degree. In order to find a Gröbner basis of this submodule, we need an initial basis. We start with a basis of the ideal, derived in Proposition 4.5.2, and from this we derive several generating sets for the submodule, analyzing each one's size and usefulness along the way. In Section 4.6, we present an algorithm (based on Buchberger's and Lee and O'Sullivan's algorithms) for computing a Gröbner basis of the submodule from some of the initial bases we found. To top it off, the correctness and complexity of the algorithm are proved in the final two subsections.

For the interested reader, the appendix contains source code for performing our multivariate list decoding algorithm in the SINGULAR Computer Algebra System. The code can easily be modified to operate on any generalized Reed Solomon code. The running time is polynomial but it is not 'small' polynomial so the performance is very slow even for normal size codes (unless done with only two variables). Nonetheless, the program is available and several small libraries (for fast multi-point interpolation, bivariate root-finding, and miscellaneous tools) are also included.

Chapter 2
Basic Tools and Concepts

2.1 Error-Correcting Codes

Data are abundant in the age of the information superhighway. The reliability of a datum can be no greater than that of the source who produced it. Moreover, for an electronic datum, its reliability cannot exceed that of the wires that transport it or the semiconductors, optical disks, or magnetic tapes that store it. Or can it?... By itself, a bit of information is what it is, and no further determination can be made as to the reliability of its content beyond what is known about the reliability of its source. In many systems, though, the whole is greater than the sum of the parts. So it is with information when we judiciously choose what sets of information are to be considered valid. If we frivolously say that all patterns of information may be transmitted or stored then we can make no determination that any information has been corrupted. But if we restrict the set of valid information patterns, which collectively we call a *code*, then we will be able to make a judgment upon seeing an invalid pattern that the data is valid or that it has somehow been corrupted and is not to be trusted. If the data pattern is unblemished, on the other hand, it will be seen as valid and its authenticity can be confirmed to some degree. Admittedly, it is possible that a valid pattern may be corrupted to another valid pattern and evade the check system we have established. This problem would be particularly relevant in a scenario where an opponent was attempting to corrupt our information. But we will refrain from dealing with the realm of adversarial attacks and focus our attention on more natural errors, such as a scratch on a CD or a short burst of electromagnetic interference. Our interest will lie in detecting when errors are sure to have occurred. What is more, using well-designed codes called *error-correcting codes*, we will look for ways to not only detect that the pattern is invalid, but to actually correct the mistake by systematically searching for the valid pattern (or patterns) that most likely corresponds to what the information we witnessed was before it became corrupted.

Over the last 60 years, check systems for various data transmission and storage channels have been studied extensively in the field of *information theory*. Basic systems for improving information reliability, such as check digit codes (e.g. ISBN and UPC) have been around for some time. But the advent of the modern incarnation of this field, which has developed almost synchronously with microprocessing, was marked by the publication in 1948 of the influential paper of Claude Shannon [35] in which he laid out the mathematical foundations of reliable communication. In it, he showed that for any reasonable (though not necessarily perfect) channel, there exists error-correcting codes which can be use to transmit data over the channel with arbitrarily high reliability. This is not to say errors will not happen, for they are inevitable on most channels. What this does say is that although errors may occur, our code, if wisely chosen (and a good choice always exists), will allow us to correct all but less than epsilon percent of the errors we encounter. The construction,

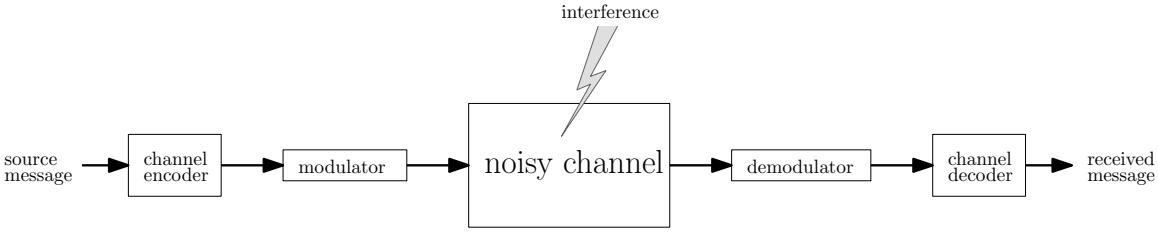

Figure 2.1: A standard model of a noisy channel

analysis, and implementation of these codes falls under the category of what is now called *coding theory*, which may be considered a branch of information theory. While information theory also encompasses other pursuits such as data compression (source coding) and quantification, coding theory focuses specifically on what is known as *channel coding*: efficiently correcting (or at least detecting) errors in data that are received from across a noisy (that is, error-prone) channel. In its early days, coding theory also incorporated the task of information security: making sure that a malefactor cannot read or alter your data. But problems related to information secrecy and authentication protocols now fall under the scope of cryptography (a fascinating field that is intimately related to coding theory but not especially pertinent to the task at hand).

The noisy channel scenario that we just described is illustrated in Figure 2.1. There are typically four components to communicating over a noisy channel, three of which are based on the main branches of information theory that were already touched upon. The first stage in data transmission (before the message is sent) is source encoding, where the data is compressed so that it can be sent using as few symbols as possible. The second stage is encryption, where the source encoded message is reversibly altered so that (hopefully) no one without sufficient knowledge of the cryptographic scheme can decipher its meaning. The third stage is channel encoding, where redundancy is added to the encrypted message (sort of the opposite of the first stage, but necessary) to ensure that the information can still be determined even if a small number of the symbols should be altered. The fourth stage is modulation, where the channel encoded message is reshaped (for example, from digital to analog waveform) into a form that can be conveyed by the channel. The message is then relayed through the noisy channel or storage medium and during this time errors may be introduced to the data. At the receiving end, the operations applied in each of the four stages are undone in reverse order and, ideally, the message that was sent is received. The first two stages and their reversions are omitted from Figure 2.1 because we will not be concerned with their applications in this study, nor those of the modulation stage, which is left in the figure only because it lies between channel encoding and decoding in the communication process. For now, we continue our discourse on coding theory by formalizing the notion of error-correcting codes.

2.1.1 Basic Definitions for Block Coding

A **probabilistic channel** S is a pair $(\Sigma, \Pr)$ where Σ is the **alphabet** of symbols that are ostensibly transmitted trough the channel and Pr is a conditional probability distribution $\Pr(\boldsymbol{y}$ is received $\mid \boldsymbol{x}$ is sent) for all pairs $(\boldsymbol{x}, \boldsymbol{y}) \in \Sigma^n \times \Sigma^n$ for any $n \in \mathbb{N}$. We assume here that the channel never deletes or inserts symbols into a message (thereby altering its length), but only changes what symbols are present. An **error correcting code** is a subset $\mathcal{C} \subseteq \bigcup_{n \geq 0} \Sigma^n$ of permissible patterns which we allow to be sent across our noisy channel. An element $\boldsymbol{c} \in \mathcal{C}$ is generally referred to as a **codeword**. The **channel encoder** is an injective mapping $E : \mathsf{M} \to \mathcal{C}$ from the **message space** M of **information words** (or message words) to the code $\mathcal{C}$. The **channel decoder** is a surjective mapping $D : \Sigma^n \to \mathsf{M}$ from the set of possible received words back to the message space. The duty of the channel decoder is two-fold. When the channel outputs an n-tuple, the decoder must first determine which codeword was most likely input to the channel and then it must invert the encoding to determine which message in M corresponds (uniquely) to this codeword. If each codeword in $\mathcal{C}$ has the same length n, then we say that $\mathcal{C}$ is a **block code** of **length** n. If $|\Sigma|$ is finite then the **alphabet size** is $q = |\Sigma|$. Henceforth we will assume that $\mathcal{C} \subseteq \Sigma^n$ is a block code of length n and $q = |\Sigma|$ is finite. In this case we say that $\mathcal{C}$ is a **q-ary (error-correcting) code**. The **dimension** of $\mathcal{C}$ is $k = \log_q |\mathcal{C}|$ (the reason for this name will be more evident shortly). The **information rate** (or sometimes just **rate**) of the code $\mathcal{C}$ is $\kappa = \frac{\log_q |\mathsf{M}|}{n}$. Since $|\mathsf{M}| = |\mathcal{C}| \leq q^n$, $\kappa \leq 1$.

It is often helpful to assume that Σ is an Abelian group (for example, $\mathbb{Z}_q$), in which case we say that S is an **additive channel** and $\mathcal{C}$ is an **additive code**. When a codeword $\boldsymbol{c} \in \mathcal{C}$ is sent, the noise of the channel can then be associated to a particular **error vector** $\boldsymbol{e} \in \Sigma^n$ where the received word $\boldsymbol{r} \in \Sigma^n$ is the sum of the codeword and the error vector, as depicted in Figure 2.2. The **(Hamming)**

$$\begin{array}{ccccc} & & \boldsymbol{e} & & \\ & & \downarrow & & \\ \boldsymbol{c} & \longrightarrow & \oplus & \longrightarrow & \boldsymbol{r} = \boldsymbol{c} + \boldsymbol{e} \end{array}$$

Figure 2.2: Symbolic representation of a noisy additive channel

weight of an n-tuple $\boldsymbol{x} = (x_1, \ldots, x_n)$ over an additive group Σ is the number of nonzero components of $\boldsymbol{x}$, denoted $\mathrm{wt}(\boldsymbol{x}) = |\{i \in [n] \mid x_i \neq 0\}|$. The **(Hamming) distance** $\mathrm{dist} : \Sigma^n \times \Sigma^n \to \mathbb{N}_0$ is the metric given by $\mathrm{dist}(\boldsymbol{x}, \boldsymbol{y}) = |\{i \in [n] \mid x_i \neq y_i\}|$, which counts the number of distinct components between any two n-tuples. The **distance** of the code $\mathcal{C}$ is the minimum distance between two distinct codewords of $\mathcal{C}$, denoted $\mathrm{dist}(\mathcal{C}) = \min_{\boldsymbol{c}, \boldsymbol{c'} \in \mathcal{C}} \mathrm{dist}(\boldsymbol{c}, \boldsymbol{c'})$. Letting $\ell = \lceil \log_q |\mathcal{C}| \rceil - 1$, we have that $q^\ell < |\mathcal{C}|$, so on the basis of the pigeonhole principle there must be at least two codewords with the same first ℓ symbols. Thus, $\mathrm{dist}(\mathcal{C})$ can be no greater than $n - \ell$, so $\mathrm{dist}(\mathcal{C}) \leq n - \lceil \log_q |\mathcal{C}| \rceil + 1 \leq n - k + 1$. This bound on the $\mathrm{dist}(\mathcal{C})$ is called the *Singleton bound*. A code whose distance achieves the Singleton bound is called a **maximum distance separable** (or **MDS**) code.

An **error** is any instance where the output of the channel differs from the input of the channel. The term error also is used to refer to the error vector that is introduced during a transmission, even if it happens to be the zero vector (and hence there is no actual error). We say that a channel decoder **corrects** an error vector $\boldsymbol{e}$ if for any codeword $\boldsymbol{c} \in \mathcal{C}$, upon input $\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{e}$ the decoder correctly outputs the message in M corresponding to $\boldsymbol{c}$. We then say that a decoder **corrects up to τ errors** (or is **τ-error correcting**) if it corrects all error vectors $\boldsymbol{e}$ whose weight satisfies $\mathrm{wt}(\boldsymbol{e}) \leq \tau$. If $\tau \in \mathbb{N}_0$ is the largest integer such that the decoder is τ-error correcting, then $p = \frac{\tau}{n}$ is called the **error rate** (of the decoder). When the decoder returns a message other than the one corresponding to the sent codeword, we call it a **decoding failure**.

Some popular decoding strategies are **maximum-likelihood decoding**, **minimum-distance decoding**, and **list decoding**. A maximum-likelihood decoder D_{MLD} decodes a received word $\boldsymbol{r} \in \Sigma^n$ to the message corresponding to the codeword $\boldsymbol{c} \in \mathcal{C}$ that maximizes the probability $\Pr(\boldsymbol{r} \text{ is received} \mid \boldsymbol{c} \text{ is sent})$. A minimum-distance decoder D_{MDD} returns the message corresponding to the codeword $\boldsymbol{c}$ that minimizes $\mathrm{dist}(\boldsymbol{c}, \boldsymbol{r})$. Each of these decoders faces the possibility of there not being a unique codeword optimizing their objective function, in which case a tiebreaker is necessary (perhaps choosing the smallest codeword lexicographically). A list decoder D_{LD} is a function that returns a *list* of all codewords within a prescribed distance to $\boldsymbol{r}$. Such a map is technically not a channel decoder because it returns a list of codewords instead of a unique unencoded message. However, D_{LD} can be thought of as a component of a channel decoder which finds all close codewords to $\boldsymbol{r}$ and from this (generally short) list a codeword of minimal distance away from the received word can be selected and its corresponding message returned to complete the decoding process. So list decoding is in some sense the same as minimum distance decoding, but there more methods available and more problems that may be solved when we take this more dynamic approach. We will focus extensively on list decoding starting in Chapter 3. For now, we consider the limitations of minimum-distance decoding.

The **(Hamming) sphere** of radius τ centered at $\boldsymbol{r} \in \Sigma^n$ is the set of words within distance τ of $\boldsymbol{r}$, denoted $B_\tau(\boldsymbol{r}) = \{\boldsymbol{y} \in \Sigma^n \mid \mathrm{dist}(\boldsymbol{r}, \boldsymbol{y}) \leq \tau\}$. Let $d = \mathrm{dist}(\mathcal{C})$. If the decoder D_{MDD} can correct up to τ errors, then it must be that the spheres $\{B_\tau(\boldsymbol{c})\}_{\boldsymbol{c} \in \mathcal{C}}$ are disjoint, for if $\boldsymbol{r} \in B_\tau(\boldsymbol{c_1}) \cap B_\tau(\boldsymbol{c_2})$ then there exist error vectors $\boldsymbol{e_1} = \boldsymbol{r} - \boldsymbol{c_1}$ and $\boldsymbol{e_2} = \boldsymbol{r} - \boldsymbol{c_2}$ of weight at most τ that corrupt two different codewords to the same received word, so the decoder cannot possibly correct both error vectors. Moreover, if $\boldsymbol{c_1}, \boldsymbol{c_2} \in \mathcal{C}$ are distance d apart, then changing $\lfloor d/2 \rfloor$ of the coordinates where they differ in $\boldsymbol{c}_1$ to the corresponding entries in $\boldsymbol{c}_2$ yields an n-tuple that lies in $B_\tau(\boldsymbol{c_1}) \cap B_\tau(\boldsymbol{c_2})$. It follows that any minimum distance decoder may correct up to $\tau = \left\lfloor \frac{d-1}{2} \right\rfloor \leq \frac{n-k}{2}$ errors, but not beyond (it may correct some larger error vectors, but it will fail on some errors of weight as small as $\lfloor d/2 \rfloor$). Since every vector in $B_{\lfloor (d-1)/2 \rfloor}(\boldsymbol{c}) \subseteq \Sigma^n$ is strictly closer to $\boldsymbol{c} \in \mathcal{C}$ than to any other codeword, it follows that the Hamming spheres of radius $\left\lfloor \frac{d-1}{2} \right\rfloor$ centered about all of the codewords in $\mathcal{C}$ are disjoint. Taking into account that $|B_{\lfloor (d-1)/2 \rfloor}(\boldsymbol{c})| = \sum_{i=0}^{\lfloor (d-1)/2 \rfloor} \binom{\lfloor (d-1)/2 \rfloor}{i} (q-1)^i$

it follows that

$$|\mathcal{C}| \leq \frac{q^n}{\sum_{i=0}^{\lfloor (d-1)/2 \rfloor} \binom{\lfloor (d-1)/2 \rfloor}{i} (q-1)^i}.$$

This inequality is called the *sphere-packing bound* (or the *Hamming bound*) for a code $\mathcal{C}$ of distance d over an alphabet of size q. It restricts the size of the code according to the alphabet size and the code's distance. Codes that meet this bound are called **perfect**.

2.1.2 Special Types of Codes

For most commonly studied codes, the alphabet is taken to be a finite field $\mathbb{F}_q$. The code $\mathcal{C} \subseteq \mathbb{F}_q^n$ is then a subset of an n-dimensional vector space over $\mathbb{F}_q$. When it is moreover a *subspace* of $\mathbb{F}_q^n$, we say that $\mathcal{C}$ is a **linear** q-ary code. In this case, the dimension $k = \log_q |\mathcal{C}|$ of the code agrees with its vector space dimension as an $\mathbb{F}_q$-subspace of $\mathbb{F}_q^n$. A linear code of length n and dimension k is often called an $[n, k]$ code (or an $[n, k, d]$ code, if one wishes to specify the distance, too). Such a code $\mathcal{C}$ can be described by a **generator matrix** G, which is any (full rank) $k \times n$ matrix over $\mathbb{F}_q$ whose rows form a basis of $\mathcal{C}$. The message space is usually presumed to be $\mathbb{F}_q^k$ and the channel encoder is an isomorphism $E : \mathbb{F}_q^k \to \mathcal{C}$. Encoding can then be described by matrix multiplication. Letting $\mathcal{B} = \{e_1, \ldots, e_k\}$ be the standard basis of $\mathbb{F}_q^k$ and $\mathcal{B}' = \{E(e_1), \ldots, E(e_k)\}$ the image of $\mathcal{B}$ under E, which is a basis of $\mathcal{C}$ (since E is an isomorphism), we have that the transpose of the generator matrix

$$G_E = \begin{bmatrix} -E(e_1)- \\ \vdots \\ -E(e_k)- \end{bmatrix}$$

of $\mathcal{C}$ is a matrix representation of E and we can encode a message $\boldsymbol{m} \in \mathbb{F}_q^k$ simply as $E(\boldsymbol{m}) = \boldsymbol{m} G_E$. The (right) nullspace of G_E is an $(n-k)$-dimensional subspace of $\mathbb{F}_q^n$, which can also be thought of as q-ary code of length n. This code is called the **dual code** of $\mathcal{C}$, denoted $\mathcal{C}^\perp$ (note: this is different from the dual space of $\mathcal{C}$). If we take an $(n-k) \times n$ generator matrix H of $\mathcal{C}^\perp$, then $\mathcal{C}$ is a subspace of the nullspace of H, but both spaces have dimension k, so $\mathcal{C} = \ker(H)$. Thus, $\mathcal{C} = (\mathcal{C}^\perp)^\perp$. We have now that $GH^t = \mathbf{0} = HG^t$. Since $\boldsymbol{r} \in \mathcal{C} = \ker(H) \iff H\boldsymbol{r}^t = \mathbf{0}$, computing $H\boldsymbol{r}^t$ gives us an efficient means of checking weather $\boldsymbol{r}$ is a codeword not. This motivates us to call H (or any generator matrix of the dual code) a **parity check matrix** for $\mathcal{C}$. G is likewise a parity check matrix for $\mathcal{C}^\perp$. One important property of linear codes derives from the fact that if $\boldsymbol{c_1}, \boldsymbol{c_2} \in \mathcal{C}$ are two codewords of distance $d = \text{dist}(\mathcal{C})$ apart, then $\boldsymbol{c_1} - \boldsymbol{c_2} \in \mathcal{C}$ is a codeword of weight d. Moreover, if $\boldsymbol{c} \in \mathcal{C}$ has weight less than d, then $\text{dist}(\mathbf{0}, \boldsymbol{c}) < d$ implies $\boldsymbol{c} = \mathbf{0}$ (since $\mathbf{0} \in \mathcal{C}$). Thus, the distance of a linear code is equal to the minimum weight of any nonzero codeword.

There are many popular families of linear codes that are studied, but we refer the reader to [1], [28], or [32] for a wider array and more thorough treatment of families of error-correcting codes. For our purposes we will focus on a broad class of codes

that fall under the category of *evaluation codes*. The constitutional class of codes among the families that the results herein will pertain to is the ubiquitous family of Reed-Solomon codes.

Definition 2.1.1 (RS Codes) *Let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{F}_q^n$ where $\alpha_1, \ldots, \alpha_n$ are distinct. The **Reed-Solomon code** of length n and dimension k generated by $\boldsymbol{\alpha}$ is the vector space obtained by evaluating all polynomials over $\mathbb{F} = \mathbb{F}_q$ of degree less than k at $\alpha_1, \ldots, \alpha_n$. It is denoted*

$$\mathrm{RS}_q(n, k, \boldsymbol{\alpha}) = \{(f(\alpha_1), \ldots, f(\alpha_n)) \mid f \in \mathbb{F}_q[x]_k\}.$$

The values $\alpha_1, \ldots, \alpha_n$ are called the **code locators**. The message space is regarded as $\mathbb{F}[x]_k \cong \mathbb{F}_q^k$. We call $f \in \mathbb{F}[x]_k$ a **message polynomial** and it is encoded as the codeword $\boldsymbol{c}_f = (f(\alpha_1), \ldots, f(\alpha_n))$. $\mathrm{RS}_q(n, k, \boldsymbol{\alpha})$ is the image of the **evaluation transformation** (which is also the channel encoder) $\mathrm{ev}_{\boldsymbol{\alpha}} : \mathbb{F}[x]_n \to \mathbb{F}^n$ restricted to $\mathbb{F}[x]_k$, where $\mathrm{ev}_{\boldsymbol{\alpha}}(f) = (f(\alpha_1), \ldots, f(\alpha_n))$. The map $\mathrm{ev}_{\boldsymbol{\alpha}}$ is an isomorphism whose inverse is given by the **interpolation map** $h_{\boldsymbol{\alpha}} : \mathbb{F}^n \to \mathbb{F}[x]_n$, where $h_{\boldsymbol{\alpha}}(r_1, \ldots, r_n)$ is the unique polynomial of degree less than n passing through (α_i, r_i) for $i = 1, \ldots, n$. We denote the polynomial $h_{\boldsymbol{\alpha}}(r_1, \ldots, r_n)$ by $h_{\boldsymbol{\alpha}, \boldsymbol{r}}$ or just $h_{\boldsymbol{r}}$ when $\boldsymbol{\alpha}$ is understood. As the image of a k-dimensional vector space under an isomorphism, $\mathrm{RS}_q(n, k, \alpha)$ is also a k-dimensional vector space. Also, since a nonzero polynomial of degree less than k can have at most $k - 1$ zeros, every nonzero codeword will have at least $n - k + 1$ nonzero components. Thus, $\mathrm{dist}(\mathrm{RS}_q(n, k, \boldsymbol{\alpha})) = n - k + 1$, so Reed-Solomon codes are maximum distance separable. Taking the basis $\{1, x, x^2, \ldots, x^{k-1}\}$ of $\mathbb{F}[x]_k$ and applying $\mathrm{ev}_{\boldsymbol{\alpha}}$ we see that

$$G_{\mathrm{RS}} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{k-1} & \alpha_2^{k-1} & \alpha_3^{k-1} & \cdots & \alpha_n^{k-1} \end{bmatrix}$$

is a generator matrix for $\mathrm{RS}_q(n, k, \boldsymbol{\alpha})$.

Generalizing Definition 2.1.1 a little bit yields a broad class of codes whose subcodes have been well studied.

Definition 2.1.2 (GRS Codes) *Let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n), \boldsymbol{v} = (v_1, \ldots, v_n) \in \mathbb{F}_q^n$ where $\alpha_1, \ldots, \alpha_n$ are distinct and each v_i is nonzero. The **generalized Reed-Solomon code** of length n and dimension k generated by $\boldsymbol{\alpha}$ and twisted by $\boldsymbol{v}$ is the image of $\mathrm{ev}_{\boldsymbol{\alpha}}|_{\mathbb{F}[x]_k}$ multiplied by $\boldsymbol{v}$ (that is, all vectors are multiplied componentwise by $\boldsymbol{v}$). It is denoted*

$$\mathrm{GRS}_q(n, k, \boldsymbol{\alpha}, \boldsymbol{v}) = \{(f(\alpha_1) \cdot v_1, \ldots, f(\alpha_n) \cdot v_n) \mid f \in \mathbb{F}_q[x]_k\}.$$

We obtain $\mathrm{GRS}_q(n, k, \boldsymbol{\alpha}, \boldsymbol{v})$ from $\mathrm{RS}_q(n, k, \boldsymbol{\alpha})$ by multiplying each codeword componentwise by $\boldsymbol{v}$, which is called the **multiplier** of the code and its components are

called the **column multipliers**. It is then easily seen to be linear with dimension k and distance $n-k+1$. A generator matrix for it is given by

$$G_{\mathrm{GRS}} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{k-1} & \alpha_2^{k-1} & \alpha_3^{k-1} & \cdots & \alpha_n^{k-1} \end{bmatrix} \begin{bmatrix} v_1 & & & \mathbf{0} \\ & v_2 & & \\ & & \ddots & \\ \mathbf{0} & & & v_n \end{bmatrix}. \tag{2.1}$$

There are several special categories of GRS codes that we make note of here. When $n = q-1$ the code is called **primitive**. When $\boldsymbol{v}$ is the all-1 vector, it is said to be **normalized**. Lastly, if each code locator is equal to its respective column multiplier (each $\alpha_i = v_i$) then it is called a **narrow sense** GRS code.

An important property of a GRS code is that its dual is also a GRS code. In fact, the dual can be defined using the same code locators.

Proposition 2.1.3 ([28], [32]) *Let $\mathcal{C} = \mathrm{GRS}_q(n, k, \boldsymbol{\alpha}, \boldsymbol{v})$ (as in Definition 2.1.2). Then $\mathcal{C}^\perp = \mathrm{GRS}_q(n, n-k, \boldsymbol{\alpha}, \boldsymbol{v}')$ for some multiplier $\boldsymbol{v}' \in \mathbb{F}_q^n$.*

Proof: Let $G = G_{\mathrm{GRS}}$ from equation (2.1). Consider the matrix

$$H = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{n-k-1} & \alpha_2^{n-k-1} & \alpha_3^{n-k-1} & \cdots & \alpha_n^{n-k-1} \end{bmatrix} \begin{bmatrix} x_1 & & & \mathbf{0} \\ & x_2 & & \\ & & \ddots & \\ \mathbf{0} & & & x_n \end{bmatrix}. \tag{2.2}$$

We claim that for a suitable choice of $\boldsymbol{x} = (x_1 \ldots, x_n)$, H is a parity check matrix for $\mathcal{C}$. To justify this, we just have to show that each entry of GH^t is zero since H is clearly a full rank matrix (since it is a (columnwise) scaled submatrix of the representation matrix for $\mathrm{ev}_{\boldsymbol{\alpha}}$, which is invertible). We need to find $\boldsymbol{x}$ such that for $1 \le i \le k$ and $1 \le j \le n-k$,

$$(GH^t)_{ij} = \sum_{m=1}^{n} v_m \alpha_m^{i+j-2} x_m = 0.$$

Since $i+j$ ranges from 2 to n, all we need is a nonzero (componentwise) solution to the system

$$G'V\boldsymbol{x}^t = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{n-2} & \alpha_2^{n-2} & \alpha_3^{n-2} & \cdots & \alpha_n^{n-2} \end{bmatrix} \begin{bmatrix} v_1 & & & \mathbf{0} \\ & v_2 & & \\ & & \ddots & \\ \mathbf{0} & & & v_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \mathbf{0}, \tag{2.3}$$

where G', V, and $\boldsymbol{x}^t$ are defined as the three matrices shown. Since V is invertible, this is equivalent to the system $G'\boldsymbol{x}^t = \mathbf{0}$. Since it has more unknowns that constraints, it must have a nonzero solution $\boldsymbol{v}' = (v'_1, \ldots, v'_n)$. If any component v'_i is zero, then taking the constraints of $G'\boldsymbol{x}^t = \mathbf{0}$ corresponding to the other $n-1$ coordinates yields a homogeneous system of equations satisfied by $\boldsymbol{x} = \boldsymbol{v}'$. But the coefficient matrix of this is Vandermonde and hence nonsingular (see [28, Ch. 4 Lemma 17]) so the system has no nonzero solution, a contradiction. Thus, $\boldsymbol{v}'$ has no zero components and setting $\boldsymbol{x} = V^{-1}\boldsymbol{v}'$ makes H a parity check matrix for $\mathcal{C}$ and a generator matrix for $\mathrm{GRS}_q(n, n-k, \boldsymbol{\alpha}, \boldsymbol{v}')$, which is therefore the dual of $\mathcal{C}$. $\square$

From the proof of Proposition 2.1.3, we see that we can find a multiplier $\boldsymbol{v}'$ for the dual code of $\mathrm{GRS}_q(n, n-k, \boldsymbol{\alpha}, \boldsymbol{v})$ by solving the system of equations in (2.3). Writing down an explicit formula for $\boldsymbol{v}'$, however, seems to be beyond our grasp. It turns out, though, that for BCH codes, which we will define shortly, writing an explicit formula for the multiplier is pretty easy. But before we consider these codes we examine an even broader class of codes.

Definition 2.1.4 (Alternant Codes) *Let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n), \boldsymbol{v} = (v_1, \ldots, v_n) \in \mathbb{F}_q^n$ where $\alpha_1, \ldots, \alpha_n$ are distinct and $v_1, \ldots, v_n \neq 0$ and consider the field extension $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$ of degree m. The **alternant code** of designed distance δ and length n generated by $\boldsymbol{\alpha}$ and twisted by $\boldsymbol{v}$ is the restriction of $\mathrm{GRS}_{q^m}(n, n-\delta+1, \boldsymbol{\alpha}, \boldsymbol{v})$ to $\mathbb{F}_q^n$. It is denoted*

$$\mathcal{A}_{q^m}(n, \delta, \boldsymbol{\alpha}, \boldsymbol{v}) = \{(f(\alpha_1) \cdot v_1, \ldots, f(\alpha_n) \cdot v_n) \mid f \in \mathbb{F}_{q^m}[x]_{n-\delta+1}\} \bigcap \mathbb{F}_q^n.$$

It should be noted that in older sources (like [28]), an alternant code is defined according the code locators of the *dual* of its defining GRS code. We have presented the alternant code in this more direct form because our interest in it will lie in its presentation as an evaluation code, in which case it is easier to have it defined by its generator matrix than by its parity check matrix. There are other sources (such as [32]) that define alternate codes as we do. The difference does not change the class because by Proposition 2.1.3 both a $\mathrm{GRS}_{\mathbb{F}_{q^m}}$ code and its dual are GRS codes, hence both of their restrictions to $\mathbb{F}_q^n$ are alternant codes by either definition.

An alternant code is referred to as a *subfield subcode* of a generalized Reed-Solomon code. Because it is a subset of a GRS code whose distance is δ, its minimum distance must be at least as great as δ. Its dimension (as an $\mathbb{F}_q$-vector space) can be no higher than $n - \delta + 1$ by the Singleton bound, but in general it will be less than that. One particularly popular group of alternant codes are the *Bose-Chaudhuri-Hocquenghem (BCH) codes*. That these are alternant codes, however, will not quite be clear from their definition.

Definition 2.1.5 (BCH Codes) *Let m be the smallest integer such that $n | q^m - 1$, let β be a primitive n^{th} root of unity in $\mathbb{F}_{q^m}$, and let $b \geq 0$. The **BCH code** of designed distance δ and length n is the set of coefficient vectors of the polynomials in*

$\mathbb{F}_q[x]/(x^n-1)$ *that have roots at* $\beta^b, \beta^{b+1}, \ldots, \beta^{b+\delta-2}$. *It is denoted*

$$\mathrm{BCH}_{q^m}(n,\delta,\beta,b) = \left\{ (c_0,\ldots,c_{n-1}) \,\middle|\, \sum_{i=0}^{n-1} c_i x^{i-1} = 0 \text{ for } x = \beta^i, b \le i \le b+\delta-2 \right\}.$$

By its definition (see also [28, Ch. 7, equation (19)]), a parity check matrix of $\mathcal{C} = \mathrm{BCH}_{q^m}(n,\delta,\beta,b)$ is given by

$$\begin{aligned} H_{\mathrm{BCH}} &= \begin{bmatrix} 1 & \beta^b & \beta^{b\cdot 2} & \cdots & \beta^{b(n-1)} \\ 1 & \beta^{b+1} & \beta^{(b+1)\cdot 2} & \cdots & \beta^{(b+1)(n-1)} \\ 1 & \beta^{b+2} & \beta^{(b+2)\cdot 2} & \cdots & \beta^{(b+2)(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta^{b+\delta-2} & \beta^{(b+\delta-2)\cdot 2} & \cdots & \beta^{(b+\delta-2)(n-1)} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \beta & \beta^2 & \cdots & \beta^{n-1} \\ 1 & \beta^2 & \beta^4 & \cdots & \beta^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta^{\delta-2} & \beta^{(\delta-2)\cdot 2} & \cdots & \beta^{(\delta-2)(n-1)} \end{bmatrix} \begin{bmatrix} 1 & & & & \\ & \beta^b & & & \mathbf{0} \\ & & \beta^{b\cdot 2} & & \\ & & & \ddots & \\ & \mathbf{0} & & & \beta^{b(n-1)} \end{bmatrix} \end{aligned} \tag{2.4}$$

Notice that H_{BCH} is a generator matrix for the code $\mathrm{GRS}_{q^m}(n,\delta-1,\boldsymbol{\beta},\boldsymbol{v})$ where $\boldsymbol{\beta} = (1,\beta,\beta^2,\ldots,\beta^{n-1})$ and $\boldsymbol{v} = (1,\beta^b,\beta^{b\cdot 2},\ldots,\beta^{b(n-1)})$. By Proposition 2.1.3, the dual of $\mathrm{GRS}_{q^m}(n,\delta-1,\boldsymbol{\beta},\boldsymbol{v})$ is another GRS code, $\mathrm{GRS}_{q^m}(n,n-\delta+1,\boldsymbol{\beta},\boldsymbol{v}')$ which has H_{BCH} as a parity check matrix. H_{BCH} is therefore a parity check matrix for the alternant code $\mathcal{A}_{q^m}(n,\delta,\boldsymbol{\beta},\boldsymbol{v}')$, too, which is thus the same as $\mathcal{C}$. The question that remains is what is the vector $\boldsymbol{v}'$ that defines $\mathrm{GRS}_{q^m}(n,n-\delta+1,\boldsymbol{\beta},\boldsymbol{v}')$, the parent GRS code of the BCH code $\mathcal{C}$?

Lemma 2.1.6 *The BCH code* $\mathrm{BCH}_{q^m}(n,\delta,\boldsymbol{\beta},b)$ *is identical to* $\mathcal{A}_{q^m}(n,\delta,\boldsymbol{\beta},\boldsymbol{v}')$ *where* $\boldsymbol{\beta} = (1,\beta,\ldots,\beta^{n-1})$ *and* $\boldsymbol{v}' = (1,\beta^{1-b},\beta^{(1-b)\cdot 2},\ldots,\beta^{(1-b)\cdot(n-1)})$.

Proof: Consider $\mathrm{GRS}_{q^m}(n,n-\delta+1,\boldsymbol{\beta},\boldsymbol{v}')$. A generator matrix for it is given by

$$G' = \begin{bmatrix} 1 & \beta^l & \beta^{l\cdot 2} & \cdots & \beta^{l(n-1)} \\ 1 & \beta^{l+1} & \beta^{(l+1)\cdot 2} & \cdots & \beta^{(l+1)(n-1)} \\ 1 & \beta^{l+2} & \beta^{(l+2)\cdot 2} & \cdots & \beta^{(l+2)(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta^{l+n-\delta} & \beta^{(1+n-\delta)\cdot 2} & \cdots & \beta^{(l+n-\delta)(n-1)} \end{bmatrix}. \tag{2.5}$$

The $(i,j)^{\mathrm{th}}$ entry in $G'H_{\mathrm{BCH}}^t$ for $1 \le i \le n-\delta+1$ and $1 \le j \le \delta-1$ (note: $2 \le i+j \le n$) is

$$(GH_{\mathrm{BCH}}^t)_{ij} = \sum_{k=1}^{n} (\beta^{b+1-b+i+j-2})^{k-1}$$

$$
\begin{aligned}
&= \sum_{k=1}^{n} (\beta^{i+j-1})^{k-1} \\
&= \frac{(\beta^{i+j-1})^n - 1}{\beta^{i+j-1} - 1} \\
&= 0.
\end{aligned}
$$

The second to last equality is justified because $\beta^{i+j-1} \neq 1$ for $1 \leq i + j - 1 \leq n - 1$. Thus, $G' H_{\text{BCH}}^t = 0$. Since H_{BCH} has full (column) rank it follows that H_{BCH} is a parity check matrix for $\text{GRS}_{q^m}(n, n - \delta + 1, \boldsymbol{\beta}, \boldsymbol{v}')$. Likewise, G' is a parity check matrix for $\text{GRS}_{q^m}(n, n - \delta + 1, \boldsymbol{\beta}, \boldsymbol{v})$. Hence, $\text{GRS}_{q^m}(n, n - \delta + 1, \boldsymbol{\beta}, \boldsymbol{v}') = \text{GRS}_{q^m}(n, n - \delta + 1, \boldsymbol{\beta}, \boldsymbol{v})^{\perp}$. So the vector $\boldsymbol{v}$ we sought (or at least one choice for it) is $\boldsymbol{v} = (1, \beta^{1-b}, \beta^{(1-b)\cdot 2}, \ldots, \beta^{(1-b)\cdot(n-1)})$. $\square$

2.2 Gröbner Bases

The presence of this section is warranted by our need to locate a particular element of a polynomial submodule in Chapter 4. Let $R = \mathbb{F}[x, \ldots, x_n]$. The **(polynomial) ideal membership problem** can be stated like this:

> Given a polynomial $f \in R$ and a generating set $\{g_1, \ldots, g_r\}$ of an ideal $I \subseteq R$, determine whether $f \in I$ or not. That is, determine if $f = \sum_{i=1}^{r} h_i g_i$ for some $h_1, \ldots, h_r \in R$.

The solution to the membership problem in the 1-variable case is easily obtained via the division algorithm and the fact that $\mathbb{F}[x]$ is a principal ideal domain.

Theorem 2.2.1 ((Univariate) Division Algorithm) *Let f and g be polynomials in $\mathbb{F}[x]$. Then there exist unique polynomials $q, r \in \mathbb{F}[x]$ (called the **quotient** and **remainder**, respectively) such that*

$$f = q \cdot g + r,$$

where either $r = 0$ or $\deg(r) < \deg(g)$.

It follows that $f(x)$ is an element of the ideal $I = \langle g(x) \rangle \subseteq \mathbb{F}[x]$ if and only if the remainder of f is zero upon division by g. However, R is not a principal ideal ring in general and the remainder in the multivariate division algorithm (which we describe later) is not uniquely determined. So the problem of determining when an arbitrary polynomial is an element of an ideal remained unsolved until 1965, when Bruno Buchberger firmly established the notion of *Gröbner bases* (named after his thesis adviser, Wolfgang Gröbner, who had previously studied them). Since their inception, Gröbner bases have become the workhorse of computational algebra with an extensive range of applications in mathematics, engineering, and science. Some examples include applications to elimination theory, robotics, geometric proof systems, solving systems of polynomial equations, reverse-engineering gene networks in

computational biology, and most importantly, solving the ideal membership problem. Our treatment of Gröbner bases shall be relatively brief; just enough to get us what we need to find an element of small degree within a submodule. For the interested reader, we refer to [5], [8], or [12] for comprehensive treatments of the subject.

To begin with, we need to define an ordering on the monomials of R, during the course of which we also define an ordering on $\mathbb{N}_0^n$. For any $\boldsymbol{a} \in \mathbb{N}_0^n$, we denote by $|\boldsymbol{a}|$ the 1-norm of $\boldsymbol{a} = (a_1, \ldots, a_n)$; that is, $|\boldsymbol{a}| = \sum_{i=1}^n a_i$. Let $[\boldsymbol{x}] = [x_1, \ldots, x_n] = \{x^{a_1} \cdots x^{a_n} \mid (a_1, \ldots, a_n) \in \mathbb{N}_0^n\}$ denote the set of monomials in the variables $x_1, \ldots, x_n$. Using multiindices, we will denote the monomial $x^{a_1} \cdots x^{a_n}$ by $\boldsymbol{x}^{\boldsymbol{a}}$. When we mention here the degree of a monomial $\boldsymbol{x}^{\boldsymbol{a}}$, we will mean its total degree (in Chapter 4, we will make sundry definitions for *degree*, but for now we only use this one). We denote it $\deg(\boldsymbol{x}^{\boldsymbol{a}}) = |\boldsymbol{a}|$. Naturally, for $f \in R$, $\deg(f)$ denotes the maximum degree of any (nonzero) term of f. Since our use of Gröbner bases will involve finding them for modules over a univariate polynomial ring, we will actually define them a bit more abstractly than is often done. Throughout this section, we let M be a finitely generated free R-module with basis $\{e_i\}$. A **monomial** in M is an element of the form $m = \boldsymbol{x}^{\boldsymbol{a}} e_i$. A **term** in M is a monomial in M multiplied by an element in $\mathbb{F}$. We say that the term ame_i divides the term bne_j if $i = j$ and m divides n in R.

Definition 2.2.2 *Let M be a free R-module with basis $\{e_i\}$. A **monomial ordering** $<$ on M is a relation on M that satisfies the following properties.*

(1) $<$ is a total order. *That is, for any $m_1, m_2 \in M$, either $m_1 < m_2$, $m_1 > m_2$, or $m_1 = m_2$.*

(2) $<$ is preserved under multiplication. That is, for all $m_1, m_2, m_3 \in M$, if $m_1 < m_2$ then $m_1 m_3 < m_2 m_3$.

(3) Every nonempty subset of M has a minimum element with respect to $<$.

If $M = R$, then the monomial order $<$ on R yields a corresponding order on $\mathbb{N}_0^n$, which we denote as $\prec$, that is given by $\boldsymbol{a} \prec \boldsymbol{b}$ if and only if $\boldsymbol{x}^{\boldsymbol{a}} < \boldsymbol{x}^{\boldsymbol{b}}$. Conditions (1) and (3) of Definition 2.2.2 are equivalent to saying that $<$ is a well-ordering. Moreover, condition (3) in itself is equivalent to the condition that every strictly decreasing sequence of monomials in $[\boldsymbol{x}]$ terminates. Also, conditions (2) and (3) together force the identity 1 to be the minimum monomial with respect to any monomial order. Interestingly, if we assume that 1 is the minimum monomial with respect to some order $<$, then conditions (1) and (2) are sufficient to imply (3).

Example 2.2.3 (lex & rlex Orders) *If $M = R$, then the **lexicographic order** on R is defined by $\boldsymbol{x}^{\boldsymbol{a}} >_{lex} \boldsymbol{x}^{\boldsymbol{b}}$ if the left-most nonzero entry of $\boldsymbol{a} - \boldsymbol{b} \in \mathbb{Z}^n$ is positive. The **(graded) reverse-lexicographic order** on R is defined by $\boldsymbol{x}^{\boldsymbol{a}} >_{rlex} \boldsymbol{x}^{\boldsymbol{b}}$ if either $\deg(\boldsymbol{x}^{\boldsymbol{a}}) > \deg(\boldsymbol{x}^{\boldsymbol{b}})$ or $\deg(\boldsymbol{x}^{\boldsymbol{a}}) = \deg(\boldsymbol{x}^{\boldsymbol{b}})$ and the right-most nonzero entry of $\boldsymbol{a} - \boldsymbol{b} \in \mathbb{Z}^n$ is negative. $>_{lex}$ and $>_{rlex}$ are monomial orders on R. Their corresponding orders on $\mathbb{N}_0^n$ are denoted $\succ_{lex}$ and $\succ_{rlex}$.*

We will use a weighted version of $<_{rlex}$ later on. For now, we fix more notation. For any $f \in M$, the **leading term** of f is defined to be the greatest term of f with respect to the order $<$, denoted $\mathrm{lt}_<(f)$ (or just $\mathrm{lt}(f)$ if the order is clear). If $\mathrm{lt}_<(f) = am$ where $a \in \mathbb{F}$ and m is a monomial of M, then we call a the leading coefficient of f and m the leading monomial of f, denoted $\mathrm{lc}_<(f)$ and $\mathrm{lm}_<(f)$, respectively. For a submodule $L \subseteq M$, the **initial submodule** of L is the submodule given by all of the leading terms of L, denoted $\mathrm{lt}_<(L) = \{\mathrm{lt}_<(f) \mid f \in L\}$.

Algorithm 2.2.4 ((Multivariate) Division Algorithm) *Let M be a free module over R with basis e_i and monomial order $<$. For any $f, g_1, \ldots, g_t \in M$, there exist $q_1, \ldots, q_t \in R$ and $r \in M$ such that*

$$f = \sum q_i g_i + r, \tag{2.6}$$

where none of the monomials in r is divisible by $\mathrm{lm}(g_i)$ and $\mathrm{lt}(f) \geq \mathrm{lt}(q_i g_i)$ for $i = 1, \ldots, t$.

We may produce such an expression as (2.6) iteratively as follows. Having chosen $s_1, \ldots, s_p$ and $m_1, \ldots, m_p$, if $r_p = f - \sum_{i=1}^{p} m_i g_{s_i} \neq 0$ and m is the maximal term of r_p that is divisible by some $\mathrm{lt}(g_k)$, then we choose $s_{p+1} = k$ and $m_{p+1} = m/\mathrm{lt}(g_k)$. The algorithm terminates when either $r_p = 0$ or no leading term of any g_k divides a monomial of r. The algorithm halts after a finite number of iterations because the maximal term of r_p that is divisible by some $\mathrm{lt}(g_k)$ decreases each time and the set of all these terms has a minimal element (by Definition 2.2.2(3)), which corresponds to the final iteration. We get the expression (2.6) by taking r to be the last r_p produced and taking $q_i = \sum_{j:s_j=i} m_j$. The expression produced in (2.6) is far from unique as we may have choices for which k to take for s_{p+1} in each step. However, the algorithm can be made deterministic if we take, for example, the smallest k for which $\mathrm{lt}(g_k)$ divides r_p each time. The element r produced by the Division Algorithm is called a **remainder** of f with respect to $G = \{g_1, \ldots, g_t\}$. Since the algorithm is nondeterministic, r is not unique, in general. However, for Gröbner bases, the remainder upon division is the key to solving the submodule (or ideal) membership problem.

Definition 2.2.5 (Gröbner Basis) *Fix a monomial order $<$ on a free module M. A finite basis $G = \{g_1, \ldots, g_t\}$ of a submodule $L \subseteq M$ is called a **Gröbner basis** (or **standard basis**) if*

$$\mathrm{lt}(L) = \langle \mathrm{lt}(G) \rangle = \langle \mathrm{lt}(g_1), \ldots, \mathrm{lt}(g_t) \rangle.$$

Example 2.2.6 ([12]) *In the case where L is an n-dimensional vector space over $\mathbb{F}$ with basis $\{e_i\}$, the elements of L can be identified by column vectors of height n. The monomials of L are $\{e_i\}$. Assume $e_1 > \cdots > e_n$. We can extend this to a monomial ordering on M by saying $\boldsymbol{x}^{\boldsymbol{a}} e_i > \boldsymbol{x}^{\boldsymbol{b}} e_j$ if $i < j$ or if $i = j$ and $\boldsymbol{x}^{\boldsymbol{a}} > \boldsymbol{x}^{\boldsymbol{b}}$ with respect to a fixed monomial ordering on R (such as $<_{lex}$). A subset $G = \{g_1, \ldots, g_t\}$ of L corresponds to an $n \times t$ matrix over $\mathbb{F}$. Then G is a Gröbner basis if and only*

if it contains a maximal independent set in 'echelon form', that is if some maximal independent subset of the column vectors g_i have their first nonzero entries in distinct rows.

If $M = R$, then the fact that every ideal has a Gröbner basis follows from Hilbert's Basis Theorem (which states that every ideal of R is finitely generated) by successively adding elements to the basis until its leading terms generate the whole initial ideal. We can do this with a finite number of additions because the rising chain of ideals formed by the initial ideals of the basis as we grow it must eventually stabilize since R is Noetherian. An essential property of Gröbner bases is that they provide a solution to the membership problem when applied in the Division Algorithm.

Lemma 2.2.7 *Let G be a Gröbner basis (with respect to some order $<$) of the submodule L of M. Let $f \in M$. Then $f \in L$ if and only if 0 is the remainder of f with respect to G.*

Proof: Suppose $f \in L$. Upon applying the Division Algorithm to f and G, we get an expression like (2.6), where the remainder $r = \sum q_i g_i - f$ has no monomial divisible by a leading term of G. But $r \in L$ (since f is) so $\mathrm{lt}(r) \in \mathrm{lt}(L) = \langle g_1, \ldots, g_2 \rangle$, therefore $\mathrm{lt}(r)$ is divisible by some $\mathrm{lt}(g_k)$, since any monomial in a monomial submodule is a multiple of some generator. Thus we get a contradiction unless $r = 0$. The converse of the lemma follows immediately from (2.6) when $r = 0$. $\square$

Buchberger gave a useful criterion for testing whether a basis G is a Gröbner basis or not. Assuming yet that $\{e_i\}$ is a basis of our free module M, for each $g, h \in G$ whose leading terms involve the same basis element, say $\mathrm{lt}(g) = ame_i$ and $\mathrm{lt}(h) = bne_i$, we define the **S-polynomial** of g and h by

$$S(g,h) = \frac{\ell}{\mathrm{lt}(g)} \cdot g - \frac{\ell}{\mathrm{lt}(h)} \cdot h,$$

where the monomial $\ell \in M$ is the least common multiple of $\mathrm{lt}(g)$ and $\mathrm{lt}(h)$; that is, $\ell = \mathrm{lcm}(m,n) \in R$ as monomials in R. For completeness, we will say that $S(g,h) = 0$ if the leading terms of g and h involve different basis elements of $\{e_i\}$.

Theorem 2.2.8 (Buchberger's Criterion) *Let L be a submodule of M generated by the finite basis G. G is a Gröbner basis of L if and only if 0 is the remainder of $S(g,h)$ with respect to G for all $g, h \in G$.*

A proof of Theorem 2.2.8 can be found in [12].

The idea of Gröbner bases would not be nearly so useful if there was not a clear method for finding them. In his thesis, Buchberger gave the following simple but useful algorithm for computing a Gröbner basis.

Algorithm 2.2.9 (Buchberger's Algorithm) *Let $G = \{g_1, \ldots, g_n\}$ be a finite basis of the submodule L of M. Compute the remainder r_{ij} of $S(g_i, g_j)$ with respect to G for all $g_i, g_j \in G$. If some $r_{ij} \neq 0$ then adjoin r_{ij} to G, and repeat the process. If all $r_{ij} = 0$, then G is a Gröbner basis of L.*

Since the submodule generated by the leading terms of $G \cup \{r_{ij}\}$ is strictly larger than the one generated by the leading terms of G, the process of Buchberger's Algorithm must eventually stop. Its exact complexity is not really clear, but analysis of Gröbner bases has shown that in worst-case situations the size of the Gröbner basis grows exponentially in the size of the initial generating set, and the best known upper bounds on the degrees of the generators of a Gröbner basis are doubly exponential in the number of variables in R. To be more precise, these bounds are $\mathcal{O}(d^{2^n})$ where n is the number of variables and d is the degree bound for the initial generators. Also, it has been shown that there are ideals for which every Gröbner basis contains $\Omega(2^{2^n})$ elements and moreover contains elements whose degrees are of the same order. So it seems computing Gröbner bases may not be the wisest course of action when speed or space matter. However, in practice there are many applications where the algorithm actually terminates fairly quickly and there are partial understandings for why it does so. Our application for instance, will not see the size of the basis nor the degrees of the generators grow at all, so the complexity will remain polynomially bounded in both.

2.3 Root-Finding

One of the essential steps for list decoding evaluation codes will involve factoring bivariate polynomials over a finite field. This step can actually be reduced from a factoring problem to a more tractable root-finding problem. The results of these areas are already well-established so we present them here with the other background information and refer back to them later in the paper when needed.

2.3.1 Complexity of Bivariate Polynomial Factorization

The field of polynomial factorization has seen substantial growth during the past few decades. Some of the highlights of this area's progress include the LLL basis reduction algorithm and Hensel lifting techniques. The LLL algorithm for finding short vectors in a lattice was introduced by Lenstra, Lenstra, and Lovász in 1982 and it yielded the first polynomial time univariate factorization algorithm for polynomials over $\mathbb{Q}$. Subsequently, it has become popular in many applications besides polynomial factorization such as integer programming, diophantine approximation, and cryptanalysis (using lattice approximation techniques to break cryptosystems). Hensel lifting is an elegant and powerful generalization of the Newton-Raphson method which solves polynomial congruences modulo a prime power. It has been shown that the average time for a bivariate polynomial factorization algorithm based on Hensel liftings is almost linear. However, it has also been shown (in [15]) that there are infinitely many polynomials that require exponential time to factor using Hensel liftings, so other methods are necessary for constructing an absolute polynomial time bivariate factoring algorithm. In [14], Shuhong Gao gave a polynomial time factorization algorithm for polynomials over $\mathbb{Q}$ or a finite field of large characteristic that is based on solving a system of linear equations derived from a partial differential equation. For bivariate polynomials over $\mathbb{F}_q$ with degree bounds m and n and at most r absolutely

irreducible factors (factors that are irreducible over the algebraic closure of $\mathbb{F}_q$) his algorithm finds all r factors using $\mathcal{O}(r(mn)^2 \log^2(mn) + r^2 \log q)$ operations in $\mathbb{F}_q$, as long as the characteristic of $\mathbb{F}_q$ is greater than $6mn$. This complexity is roughly $\mathcal{O}(N^{2.5})$ (ignoring logarithmic factors) where N is the input size. Other polynomial time factorization algorithms for bivariate polynomials over finite fields have been presented by Lenstra ($\mathcal{O}(N^4)$ field operations, [26]) and by von zur Gathen and Kaltofen ($\mathcal{O}(N^6)$ field operations, [39]). For the purposes of list decoding, however, we will not need to have a complete factorization of the polynomials we work with. Rather, for an operative polynomial $Q(x, y) \in \mathbb{F}[x, y]$ it will be sufficient to know what the *y-roots* of Q are, by which we mean the polynomials $f(x) \in \mathbb{F}[x]$ such that substituting f for y in Q yields a polynomial in x that is uniformly 0, i.e. $Q(x, f(x)) = 0$. To this end we utilize a root-finding algorithm that was first presented by Roth and Ruckenstein in [33].

2.3.2 A Recursive Bivariate Root-Finding Algorithm

The bivariate root-finding problem, as it concerns us, may be formally stated the following way.

> **Bivariate Root-Finding Problem:** Given a polynomial $Q(x, y) \in \mathbb{F}[x, y]$, find all $f(x) \in \mathbb{F}[x]$ such that $Q(x, f(x)) = 0$.

Our method of solving this problem will rely on a recursive algorithm that reduces the bivariate root-finding problem to a univariate root-finding problem. In [20], Kaltofen and Shoup showed that a univariate polynomial over $\mathbb{F}_q$ of degree d could be factored probabilistically with $\mathcal{O}(d^{1.815} \log q)$ arithmetic operations in $\mathbb{F}_q$, so root-finding should be manageable in sub-quadratic time, too. A simple way to find the roots in $\mathbb{F}_q$ of a univariate polynomial $f(x) \in \mathbb{F}_q[x]$ of degree d is to compute $g = \gcd(f(x), x^q - x)$ and then use an equal-degree factorization algorithm[1] to factor g. Such an algorithm is described in [38, Algorithm 14.15], where probabilistic methods are used to find all roots of $f(x)$ in $\mathbb{F}_q$ using an expected number of $\mathcal{O}(d \log^2 d \log\log d \log(dq))$ operations in $\mathbb{F}_q$, or roughly $\mathcal{O}(d \log q)$, ignoring small logarithmic factors. Deterministically, though, by [27] we can still find the roots of $f(x)$ using $\mathcal{O}((d^2 \log^2 d) \log q)$ operations in $\mathbb{F}_q$.

The bivariate root-finding algorithm of Roth and Ruckenstein ([33]) is based on a simple lemma that retracts the problem to a univariate root-finding matter. In the lemma, we wish to discard factors of x from the polynomials we produce so we make the following definition.

Definition 2.3.1 *For any nonzero $f \in \mathbb{F}[x, y]$, the **reduction** of f is f divided by the largest power of x that is a factor of f. We denote this by*

$$\mathrm{red}(f) = \frac{f}{x^{\max\{i \,:\, x^i | f\}}}.$$

[1] Equal-degree factorization algorithms have been studied as far back as Gauss, who studied the simplest case where all the factors are linear.

Lemma 2.3.2 *Suppose* $g(x) = \sum_{i \geq 0} g_i x^i$ *is a* y*-root of some nonzero polynomial* $Q(x, y) \in \mathbb{F}[x, y]$. *Iteratively define* Q_i, *for* $i \geq 0$, *as follows:*

$$Q_0 := \text{red}(Q)$$

$$Q_{i+1} := \text{red}(Q_i(x, xy + g_i)), \text{ for } i \geq 0.$$

Then for each $i \geq 0$, $Q_i(x, g^{[i]}(x)) = 0$, *where* $g^{[i]} = \sum_{s \geq i} g_s x^{s-i}$, *and consequently* $Q_i(0, g_i) = 0$. *Moreover,* $Q_i(0, y) \neq 0$ *so* g_i *is a nontrivial root of* $Q_i(0, y)$.

Proof: To start with, since $Q \neq 0$ and it has a y-root, $\deg_y(Q)$ must be positive. Let $\ell = \deg_y(Q) > 0$. Then inductively we have that $\deg_y(Q_i) = \ell > 0$ for all $i \geq 0$ because Q_0 is obtained from Q by removing x factors, which leaves the y-degree unchanged, and from there each Q_{i+1} is obtained from Q_i by substituting for y an expression that is still linear in y and discarding x factors. So the y-degree remains constant as we iteratively define these Q_i's. (For now, we only use this fact to establish that each Q_i is nonzero, but the constancy of the y-degree shall be more important later on when we analyze the complexity of the root-finding algorithm.) Since $Q_i \neq 0$, Q_{i+1} is well-defined by Definition 2.3.1. This is sufficient to show that $Q_i(0, y) \neq 0$ for $i \geq 0$ because $Q_i(0, y) = 0$ if and only if x divides Q_i but x cannot divide Q_i by the definition of Q_i.

It remains to show that $Q_i(x, g^{[i]}(x)) = 0$. Notice that $g^{[i]}(x) = xg^{[i+1]}(x) + g_i$ for $i \geq 0$. To begin our main induction argument for the lemma, observe that since $Q_0(x, y) = \frac{Q(x,y)}{x^s}$ for some $s \geq 0$, $Q_0(x, g^{[0]}(x)) = \frac{Q(x,g(x))}{x^s} = \frac{0}{x^s} = 0$. Assume that $Q_i(x, g^{[i]}(x)) = 0$ for some $i \geq 0$. Then for some $s \geq 0$,

$$Q_{i+1}(x, g^{[i+1]}(x)) = \frac{Q_i(x, xg^{[i+1]}(x) + g_i)}{x^s} = \frac{Q_i(x, g^{[i]}(x))}{x^s} = \frac{0}{x^s} = 0.$$

Thus, for every $i \geq 0$, $Q_i(x, g^{[i]}(x)) = 0$. Setting $x = 0$ in the this formula yields $Q_i(0, g_i) = 0$ for all $i \geq 0$. □

By Lemma 2.3.2, which is a slightly condensed form of Lemma 5.1 in [33], the coefficients of the y-roots of $Q(x, y)$ correspond to the roots of the nonzero polynomials $Q_i(0, y) \in \mathbb{F}[y]$, so by finding these we can reconstruct all of the y-roots of Q. We now describe a recursive algorithm for doing just that.

Algorithm 2.3.3
Description: finds the y*-roots of* $Q(x, y)$ *that have degree less than* d
Input: a nonzero, bivariate polynomial Q, *a degree bound* d *on the* y*-roots sought, and a recursive depth counter* i *that should be* 0 *for the initial call*
Output: y*-roots of* Q *of degree less than* d
Global: an array $g[0, \ldots, d-1]$ *is maintained for storing the coefficients*

`Rootfind(`$Q(x, y)$`, int` d`, int` i`)`

`R1.` $Q_i := \text{red}(Q)$

R2. Find all distinct roots s of $Q_i(0,y) \in \mathbb{F}[y]$

R3. For each s do

R4. $\quad g[i] := s$

R5. $\quad$ if $(i = d-1)$

R6. $\quad\quad$ output $g(x) = \sum_{j=0}^{d-1} g[j]x^j$ if $Q_i(x,s) = 0$

R7. $\quad$ else

R8. $\quad\quad$ Rootfind$(Q_i(x, xy+s), d, i+1)$

Algorithm 2.3.3 is a slightly condensed version of the algorithm in [33, Figure 2]. There is a slight difference in our algorithm in that step R6 checks if the polynomial found is in fact a y-root before outputting it, whereas the algorithm of [33] simply outputs all sequences found when it gets to step R6, including (possibly) erroneous y-roots. The input of the degree bound d is not essential to the algorithm's functionality. If the user simply wanted all of Q's y-roots and did not care about their degrees the algorithm could be modified to calculate an upper bound on the degree of any y-root and use this value for d. However, for our purposes inputting a bound is handy because we will only be interested in finding y-roots with sufficiently small degrees. Also, specifying d will permit tighter complexity bounds in our application. First, though, we verify the correctness of the algorithm.

Correctness

Throughout this and the next subsection we will use ℓ to denote $\deg_y(Q(x,y))$. Then, as discussed in the proof of Lemma 2.3.2, $\ell = \deg_y(Q_i)$ for all $i \geq 0$.

Proposition 2.3.4 *If Algorithm 2.3.3 is called with $Q \in \mathbb{F}[x,y]$, $d \in \mathbb{N}$, and $i = 0$, then it will eventually halt and correctly output exactly the y-roots of $Q(x,y)$ of degree less than d.*

Proof: To see that Algorithm 2.3.3 halts properly, notice that the maximum number of roots of $Q_i(0,y)$ is ℓ, which is also a bound on the number of iterations of the `for` loop (step R3) and since there are no other control loops there cannot be any infinite loops. ℓ is also a bound here on the number of recursive calls to Rootfind. Moreover, the recursive depth of the algorithm is bounded by d, so the recursion tree for Rootfind is a rooted tree of depth d where each node has degree at most ℓ. Hence there are no more than $\sum_{i=0}^{d-1} \ell^i = \frac{\ell^d - 1}{\ell - 1}$ recursive calls to Rootfind, so the algorithm must terminate after a finite number of steps.

Now we have to prove that the algorithm outputs $g(x)$ if and only if $g(x)$ is a y-root of $Q(x,y)$ of degree less than d. If $g(x) = \sum_{s=0}^{d-1} g_s x^s$ is such a y-root, then by Lemma 2.3.2 the algorithm will eventually reach a recursive depth level of d where the roots s it has recursed on are the successive coefficients of $g(x)$. At this point, the coefficients of $g(x)$ are in the array $g[0, \ldots, d-1]$ and $Q_i = Q_{d-1}$ in the algorithm

has been defined the same as Q_{d-1} in the lemma. Let r_j be the power of x such that $Q_j = \mathrm{red}(Q_{j-1}(x, xy + g_{j-1})) = \frac{Q_{j-1}(x,xy+g_{j-1})}{x^{r_j}}$ for $j \geq 1$ and let r_0 be such that $Q_0 = \frac{Q(x,y)}{x^{r_{-1}}}$. Then

$$\begin{aligned} Q_d - 1(x, g[d-1]) &= Q_{d-1}(x, g^{[d-1]}) \\ &= \frac{Q_{d-2}(x, g^{[d-2]})}{x^{r_{d-1}}} \\ &\vdots \\ &= \frac{Q_0(x, g^{[0]})}{x^{r_{d-1}+r_{d-2}+\cdots+r_1}} \\ &= \frac{Q(x, g(x))}{x^{r_{d-1}+\cdots+r_0}} \\ &= 0. \end{aligned} \tag{2.7}$$

Therefore, the algorithm will output $g(x)$ in step R6. Now suppose, that the polynomial $g(x)$ defined by the array at some point when step R6 is executed is outputted by the algorithm. In this case, equation 2.7 still holds and it must be that $Q(x, g(x)) = 0$, so $g(x)$ is a y-root of $Q(x, y)$. □

Complexity

In the proof of Proposition 2.3.4 we described the structure of the recursion tree for Algorithm 2.3.3 and from this it is clear that the maximum number of recursive calls from a depth level of i is ℓ^i and so the number of recursive descent paths the algorithm may follow is ℓ^{d-1}. This is a power of $d-1$ more than the possible number of y-roots, ℓ, that $Q(x, y)$ may have, so it seems the the algorithm may be doing a lot of hopeless searching during its execution. However, it turns out that the number of recursive calls to Rootfind is generally much less than the simple bound we stated before and that the total number of recursive descent paths is actually at most ℓ, so it is really possible that all paths could be fruitful in producing a y-root at the end.

Lemma 2.3.5 *Let $Q_i = \sum_{s=0}^{\ell} q_s(x)y^s \in \mathbb{F}[x, y]$ be a nonzero bivariate polynomial, let $a \in \mathbb{F}$ be a root of $Q_i(0, y)$ of multiplicity m, and let $Q_{i+1} = \mathrm{red}(Q_i(x, xy + a)) = \frac{Q_i(x,xy+a)}{x^r}$ where $r = \max\{i \ : \ x^i \mid Q_i(x, xy + a)\}$. Then $\deg(Q_{i+1}(0, y)) \leq m$.*

Proof: Let $Q' = Q_i(x, y + a) = \sum_{s=0}^{\ell} q'_s(x)y^s$. Since a is a root of $Q_i(0, y)$ of multiplicity m, 0 is a root of $Q'(0, y)$ of multiplicity m. Therefore, for $0 \leq s < m$, $q'_s(0) = 0$ and so $q'_s(x)$ is divisible by x but $q'_m(0) \neq 0$ so $q'_m(x)$ is not divisible by x. Now let $Q'' = Q_i(x, xy + a) = Q'(x, xy) = \sum_{s=0}^{\ell} q'_s(x)x^s y^s$. Then Q'' is divisible by x but not by x^{m+1}, so $1 \leq r \leq m$. Then $Q_{i+1} = Q''/x^r = \sum_{s=0}^{r} \frac{q'_s(x)x^s}{x^r} y^s + \sum_{s=r+1}^{\ell} q'_s(x)x^{s-r}y^s$. From this expression we see that $\deg(Q_{i+1}(0, y)) \leq r \leq m$ since the terms in the second summand all become zero when $x = 0$. □

Notice that Lemma 2.3.5 bounds the number of recursive subbranches for each recursive call in the algorithm by the multiplicity of the root that is recursed upon. This bounds the overall number of descent paths in the recursion tree to at most $\ell = \deg_y(Q)$.

Lemma 2.3.6 *If Algorithm 2.3.3 is called with $Q \in \mathbb{F}[x, y]$, $d \in \mathbb{N}$, and $i = 0$, then the overall number of recursive calls made to Rootfind is at most $\ell(d-1)$.*

Proof: At depth level $i = 0$, the degree of $Q_0(0, y)$ is at most ℓ and hence there are at most ℓ recursive calls to Rootfind. By Lemma 2.3.5, the sum of the degrees of the polynomials $Q_i(0, y)$ processed at depth level i is greater than or equal to the sum of the degrees of the polynomials $Q_{i+1}(0, y)$ processed at depth level $i + 1$ for $i = 0, \ldots, d-2$. Hence, inductively, there are at most ℓ recursive calls to Rootfind at each depth level i for $i = 0, \ldots, d-2$ and no recursive calls at higher depth levels because there the degree bound has been met. Thus, there are at most $\ell(d-1)$ recursive calls made to Rootfind before the algorithm halts. □

Proposition 2.3.7 *If Algorithm 2.3.3 is called with $Q \in \mathbb{F}_q[x, y]$ where $\deg_x(Q) = m$ and $\deg_y(Q) = \ell$, $d \in \mathbb{N}$, and $i = 0$, then the number of arithmetic operations in $\mathbb{F}_q$ used during the execution of the algorithm to its full recursive depth is*

$$\mathcal{O}(\ell^2(m + d\ell) + d\ell \log^2 \ell \log\log \ell(m + d^2\ell + \log q)), \text{ if } \ell < q$$

and it is

$$\mathcal{O}(d\ell^2(m + d^2\ell)), \text{ if } \ell \geq q.$$

Proof: Step R1 would require a number of steps proportional to the number of terms in Q to find the largest power of x that divides Q and subtract that power from x in each term, but it does not require any operations in $\mathbb{F} = \mathbb{F}_q$. Step R2 requires finding roots of univariate polynomials and by Lemma 2.3.5 the sum of the degrees of the operating polynomials $Q_i(0, y)$ at recursive depth level i is at most ℓ. By Corollary 14.16 in [38], the roots of a degree n univariate polynomial over $\mathbb{F}$ can be found using an expected number of $\mathcal{O}(n \log^2 n \log\log n \log qn)$ operations in $\mathbb{F}$. The greatest complexity we can incur at the i^{th} level in step R2 comes when there is just one polynomial of degree ℓ. In this case step R2 costs $\mathcal{O}(\ell \log^2 \ell \log\log \ell \log q\ell)$ field operations at level i and overall R2 requires an expected number of $\mathcal{O}(d\ell \log^2 \ell \log\log \ell \log q\ell)$ arithmetic operations in $\mathbb{F}$ during the entire recursive descent of the algorithm.

Let $m_i = m + i(\ell - 1)$ for $0 \leq i < d$. Then m_i is an upper bound on the x-degree of Q_i at depth level i in the recursive descent of the algorithm. The expense of step R6 comes from evaluating $Q_{d-1}(x, y)$ at $y = g_{d-1}$. The evaluation can be performed by finding $g_{d-1}^2, \ldots, g_{d-i}^\ell$ at a cost of $\ell - 1$ multiplications in $\mathbb{F}$, then multiplying each term of Q_{d-1} by the appropriate power of g_{d-1} and combining terms, which may require up to $(m_{d-1} + 1)\ell$ multiplications and the same number of additions. Hence, one execution of step R6 requires $\mathcal{O}(m_{d-1}\ell)$ operations in $\mathbb{F}[x]$. By Lemma 2.3.5 step

R6 can be run at most ℓ times so the cost of $R6$ through the entire recursive descent is $\mathcal{O}(\ell^2(m+d\ell))$.

In step R8, we must compute $Q_i(x, xy+s)$ for the recursive call to Rootfind. This can be done in two steps: first we compute the shifted polynomial $Q_i(x, y+s)$, and second we substitute xy for y. The second step simply requires increasing the x-degree of every term by the value of its y-degree and does not require any operations in $\mathbb{F}_q$, so we only need to assess the cost of computing $Q_i(x, y+s)$. If $\ell < q$, then we can proceed as follows. Expressing Q_i as $Q_i = \sum_{t=0}^{m_i} r_t(y)x^t$ where $r_t \in \mathbb{F}[y]$ and $\deg(r_t) \leq \ell$, we have that $Q_i(x, y+s) = \sum_{t=0}^{m_i} r_t(y+s)x^t$ and we can compute $r_t(y+s)$ by evaluating $r_t(y)$ at $\ell+1$ distinct values $a_0, \ldots, a_\ell \in \mathbb{F}$ and then obtain $r_t(y+s)$ by interpolating through the ℓ points $(a_j - s, r_t(a_j))$ for $j = 0, \ldots, t$. By Corollaries 10.8 and 10.10 of [38], both the evaluation and interpolation steps take $\mathcal{O}(\ell \log^2 \ell \log\log \ell)$ operations in $\mathbb{F}$. By Lemma 2.3.5, we may have to run step R8 up to ℓ times at each recursive depth level $i = 0, \ldots, d-2$ (step R8 will never be executed when $i = d-1$), so the net cost of step R8 is $\mathcal{O}(d\ell \log^2 \ell \log\log \ell(m + d^2\ell))$ operations in $\mathbb{F}$. If, however, $\ell \geq q$, then we will not have enough points to interpolate through to find $r_t(y+s)$, so we must take another approach. Express Q_i as $Q_i = \sum_{t=0}^{\ell} q_t(x)y^t$, where $q_t \in \mathbb{F}[x]$ and $\deg(q_t) \leq m_i$. Then $Q_i(x, y+s) = \sum_{t=0}^{\ell} q_t(x)(y+s)^t$. To compute $(y+s)^t = \sum_{j=0}^{t} \binom{t}{j} s^{t-j}y^t$ for $t = 0, \ldots, \ell$ requires finding $s^2, \ldots, s^\ell$, which can be done with $\ell - 1$ multiplications in $\mathbb{F}$, and then combining these values with the appropriate binomial coefficients and powers of y (no additional operations in $\mathbb{F}$ are needed here). Then we have to multiply $q_t(x) \cdot (y+s)^t$ for $t = 0, \ldots, \ell$, which may require as many as $\sum_{t=0}^{\ell}(m_i+1)(t+1) = (m+1)\binom{\ell+1}{2} \in \mathcal{O}(m\ell^2)$ multiplications, and then adding these products together may require up to $(m_i+1)(\ell+1)\ell$ additions in $\mathbb{F}$. So the cost of computing $Q_i(x, y+s)$ this way is $\mathcal{O}(m_i\ell^2)$ operations in $\mathbb{F}$ and the cost of step R8 throughout the entire recursive descent of the algorithm is $\mathcal{O}(d\ell^2(m + d^2\ell))$ operations in $\mathbb{F}$.

Combining the field arithmetic costs of steps R2, R6, and R8 we have that the complexity of Algorithm 2.3.3 is $\mathcal{O}(\ell^2(m+d\ell) + d\ell \log^2 \ell \log\log \ell(m + d^2\ell + \log q))$ if $\ell < q$ and it is $\mathcal{O}(d\ell^2(m + d^2\ell))$ if $\ell \geq q$. $\square$

Ignoring most logarithmic factors, the cost of Algorithm 2.3.3 is $\mathcal{O}(\ell^2(m+d\ell) + d\ell(m + d^2\ell + \log q))$ if $\ell < q$ and it is $\mathcal{O}(d\ell^2(m + d^2\ell))$ if $\ell \geq q$. Our proof followed most of the structure of the proof of Proposition 6.6 in [33] except their proof did not consider the case where $\ell \geq q$ and they made special assumptions on the weighted degree of the polynomial Q according to their purpose for using the algorithm. We now make a slight modification to the complexity analysis that will apply it more specifically to our needs.

Corollary 2.3.8 *Suppose $n \leq q-1$ and $Q \in \mathbb{F}_q[x, y]$ is a nonzero polynomial whose y-degree is at most $\ell < q$ and whose $(1, d)$-weighted degree is at most $N = m(n - \tau)$ (for some $m, \tau \in \mathbb{N}$). Then the number of field operations needed by Algorithm 2.3.3 to find all of the y-roots of Q is*

$$\mathcal{O}(\ell \log^2 \ell \log\log \ell(d \log q + N\ell)).$$

Proof: Again, steps R2, R6, and R8 are the only ones that require operations in $\mathbb{F}$. Step R2 costs the same as in the proof of Proposition 2.3.7, which is $\mathcal{O}(d\ell \log^2 \ell \log\log \ell \log q\ell)$ arithmetic operations in $\mathbb{F} = \mathbb{F}_q$ over the entire recursive descent of the algorithm.

Let $N_t = N - dt$ for $t = 0, \ldots, \ell$. Then N_t is an upper bound on the maximum x-degree of any term in Q with y-degree t. Because y is replaced by $xy + s$ in Q_i in each recursive call to Rootfind, at recursive depth level i we have that the maximum x-degree of any term with y-degree t in Q_i is $N_{i,t} := N_t + i(t-1) = N - (d-i)t - i \leq N$ since $i < d$ throughout the algorithm. (Note: in the proof of Lemma 2.3.5 it was shown that at least one factor of x is removed every time we reduce in step R2 when $i > 1$.)

The cost of step R6 comes from evaluating $Q_{d-1}(x, y)$ at $y = g_{d-1}$. The evaluation can be performed by finding $g_{d-1}^2, \ldots, g_{d-i}^\ell$ at a cost of $\ell - 1$ multiplications in $\mathbb{F}$, then multiplying each term of Q_{d-1} by the appropriate power of g_{d-1} and combining terms, which may require up to $\sum_{t=1}^{\ell} N_{d-1,t} = N\ell - \binom{\ell}{2} - (d-1)\ell$ multiplications and about the same number of additions. Hence, one execution of step R6 requires $\mathcal{O}(N\ell)$ operations in $\mathbb{F}[x]$. By Lemma 2.3.5 step R6 can be run at most ℓ times so the cost of $R6$ through the entire recursive descent is $\mathcal{O}(N\ell^2)$.

Consider the polynomial $Q_i = \sum_{t=0}^{N} q_t(y)x^t$. Let $\ell_t = \deg(q_t)$ for $0 \leq t \leq N$. Then $\ell_t \leq \ell$. Following the analysis of the proof of the cost of step R8 in Proposition 2.3.7, we find that computing $Q_i(x, y + s) = \sum_{t=0}^{N} q_t(y + s)x^t$ by interpolating through the values $q_t(a_j + s)$ for $\ell_t + 1$ distinct points $a_j \in \mathbb{F}$ has a cost of $\mathcal{O}(\ell_t \log^2 \ell_t \log\log \ell_t)$ operations in $\mathbb{F}$. By counting the number of terms of Q_i two different ways, we see that

$$\sum_{t=0}^{N} \ell_t \leq \sum_{t=0}^{\ell} N_{i,t} \leq (\ell + 1)N.$$

Then each execution of step R8 takes at most $\mathcal{O}(N\ell \log^2 \ell \log\log \ell)$ field operations. Hence, the overall complexity of step R8 in Algorithm 2.3.3 is $\mathcal{O}(dN\ell^2 \log^2 \ell \log\log \ell)$ on the specified input.

Combining the complexities of steps R2, R6, and R8, we have that the algorithm requires $\mathcal{O}(d\ell \log^2 \ell \log\log \ell(\log q + N\ell))$ operations in $\mathbb{F}$. $\square$

The result of Corollary 2.3.8 is similar to the time complexity bound given by Roth and Ruckenstein in Proposition 6.6 of [33] except that they had an additional factor of ℓ in their cost analysis of step R2 (R3 in [33]) because they cited univariate root-finding algorithms with such an extra factor. Also, they replaced $N\ell$ by n in their asymptotic bound with the aid of preconditions they had assumed on the relationship of n, N, and ℓ. We have not enforced the same constraints here because we will be using the root-finding algorithm in a more general setting. Also, in [33] a space complexity of $\mathcal{O}(n)$ is given for their algorithm. Our algorithm should achieve the same space complexity as the only significant difference between the algorithms is that ours checks whether the polynomial $g(x)$ is actually a y-root of Q before outputting it instead of always outputting the polynomial $g(x)$ (and possibly producing erroneous y-roots) when step R6 is reached. Evaluating $Q_{d-1}(x, s)$ in step R6 is no more complex

than the evaluating and interpolating that occurs in step R8, so the test in R6 does not increase the asymptotic space complexity of our algorithm. However, in the tasks that we are about to undertake our attention will be focused on improving the time complexity of the algorithms we utilize for list decoding, so we will not be concerned about analyzing the space complexity of our algorithms.

2.3.3 Modular Extension of RootFind

Algorithm 2.3.3 will be useful for performing the root-finding step of the bivariate list decoding algorithm that we present in Chapter 3, but for the multivariate list decoding algorithm in Chapter 4, we will need to adapt the root-finding algorithm to search for y-roots modulo a polynomial in $\mathbb{F}[x]$. This modified root-finding problem can be precisely stated as follows.

> **Bivariate Modular Root-Finding Problem:** Given polynomials $Q(x, y) \in \mathbb{F}[x, y]$ and $h(x) \in \mathbb{F}[x]$ such that Q is not a multiple of h, find all $f(x) \in \mathbb{F}[x]$ such that $Q(x, f(x)) \bmod h(x) = 0$.

When we write $Q(x, y) \bmod h(x)$ we shall be referring to the canonical representative of the residue class of Q in $\mathbb{F}[x, y]/\langle h(x)\rangle$, not to the residue class itself (because x may be a unit in the quotient ring and that would cause problems for our reduction definition). We can easily adapt Lemma 2.3.2 to apply to this modular root-finding problem.

Lemma 2.3.9 *Suppose $g(x) = \sum_{i\geq 0} g_i x^i$ is a y-root of $Q(x, y)$ modulo $h(x)$ (that is, $Q(x, g(x)) \bmod h(x) = 0$) of some polynomial $Q(x, y) \in \mathbb{F}[x, y]$ that is not divisible by $h(x)$, where $h(x)$ is a reduced polynomial, meaning $\mathrm{red}(h) = h$. Iteratively define Q_i, for $i \geq 0$, as follows:*

$$Q_0 := \mathrm{red}(Q \bmod h(x))$$

$$Q_{i+1} := \mathrm{red}(Q_i(x, xy + g_i) \bmod h(x)), \text{ for } i \geq 0.$$

Then for each $i \geq 0$, $Q_i(x, g^{[i]}(x)) = 0$, where $g^{[i]} = \sum_{s\geq i} g_s x^{s-i}$, and consequently $Q_i(0, g_i) = 0 \in \mathbb{F}_q[x, y]$. Moreover, $Q_i(0, y) \neq 0 \in \mathbb{F}_q[x, y]$ so g_i is a nontrivial root of $Q_i(0, y)$.

Proof: We first show the Q_i's all have the same y-degree. Let $t = \deg_y(Q \bmod h(x))$ and let $f(x)y^t \neq 0$ be the leading term of $Q \bmod h(x)$. Then $\deg(f) < \deg(h)$. The leading term of Q_0 is $f(x)y^t/x^s \neq 0$ for some $s \in \mathbb{N}_0$. Hence, the leading term of $Q_0(x, xy + g_0) \bmod h(x)$ is $f(x)x^t y^t/x^s \neq 0$ (since $\deg(f) < \deg(h)$ and h is not divisible by x) and so the leading term of Q_1 is $f(x)x^{s_1}y^t$ for some $s_1 \in \mathbb{N}_0$. Then the leading term of $Q_1(x, xy + g_1) \bmod h(x)$ is $f(x)x^{s_1+t}y^t$ and so the leading term of Q_2 is $f(x)x^{s_2}y^t$ for some $s_2 \in \mathbb{N}_0$. Repeating this argument we get that the leading term of Q_i is $f(x)x^{s_i}y^t$ for some $s_i \in \mathbb{N}_0$ for all $i \geq 0$ and so $Q_i \neq 0$ (we actually need this fact, iteratively, to have Q_{i+1} be well-defined) and by its definition it is not divisible by x so $Q_i(0, y) \neq 0$.

Now, because g is a y-root of $Q \bmod h(x) \neq 0$ and $\deg(g) < \deg(h)$, $(y - g)$ is a factor of $Q \bmod h(x)$ and so also of Q_0. Then, $(xy + g_0 - g) = x(y - g^{[1]})$

is a factor of $Q_0(x, xy + g_0) \bmod h(x)$ and so $(y - g^{[1]})$ is a factor of Q_1. Then, $(xy + g_1 - g^{[1]}) = x(y - g^{[2]})$ is a factor[2] of $Q_1(x, xy + g_1) \bmod h(x)$ and so $(y - g^{[2]})$ is a factor of Q_2. Repeating this argument, we get that $(y - g^{[i]})$ is a factor of Q_i for all $i \geq 0$. Since $(y - g^{[i]})$ is a factor of Q_i, $Q_i(x, g^{[i]}(x)) = 0$. Substituting 0 for x yields $Q_i(0, g_i) = 0$. □

We now modify Algorithm 2.3.3 to handle the modular root-finding problem.

Algorithm 2.3.10
Description: finds the y-roots of $Q(x, y)$ modulo $h(x)$ that have degree less than d
Input: a nonzero, bivariate polynomial Q, a degree bound d on the y-roots sought, a recursive depth counter i that should be 0 for the initial call, and a reduced polynomial $h(x)$ (that is, $\mathrm{red}(h) = h$)
Output: y-roots of Q of degree less than d
Global: an array $g[0, \ldots, d-1]$ is maintained for storing the coefficients

`ModRootfind(`$Q(x, y)$`, int` d`, int` i`,` $h(x)$`)`

`R1.` $Q_i := \mathrm{red}(Q \bmod h(x))$

`R2. Find all distinct roots` s `of` $Q_i(0, y) \in \mathbb{F}[y]$

`R3. For each` s `do`

`R4.` $g[i] := s$

`R5.` `if` $(i = d - 1)$

`R6.` `output` $g(x) = \sum g[j]x^j$ `if` $Q_i(x, g[i]) = 0$

`R7.` `else`

`R8.` `ModRootfind`$(Q_i(x, xy + s), d, i + 1, h(x))$

Proposition 2.3.11 *If Algorithm 2.3.10 is called with $Q \in \mathbb{F}[x, y]$ not equal to zero, $d \in \mathbb{N}$, $i = 0$, and a reduced polynomial $h(x) \in \mathbb{F}[x]$ of degree d or higher, then it will eventually halt and correctly output exactly the y-roots of $Q(x, y)$ modulo $h(x)$ of degree less than d.*

Proof: To see that Algorithm 2.3.10 halts properly, notice that the maximum number of roots of $Q_i(0, y)$ is $\deg_y(Q)$, which is a bound on the number of iterations of the `for` loop (step R3) and since there are no other control loops there cannot be any infinite loops. $\deg_y(Q)$ is also a bound here on the number of recursive calls to `ModRootfind` for each call made. Moreover, the recursive depth of the algorithm is bounded by d, so the recursion tree for `ModRootfind` is a rooted tree of depth d where each node has

[2]Actually, if $\deg(h) = 1$ then x is not a factor of $Q_1(x, xy + g_1) \bmod h(x)$, but this is inconsequential for what we are trying to show.

degree at most $\deg_y(Q)$. Hence there are no more than $\sum_{i=0}^{d-1} \deg_y(Q)^s < \infty$ recursive calls to `ModRootfind`, so the algorithm must terminate after a finite number of steps.

Now we have to prove that the algorithm outputs $g(x)$ if and only if $g(x)$ is a y-root of $Q(x,y)$ modulo $h(x)$ of degree less than d. If $g(x) = \sum_{s=0}^{d-1} g_s x^s$ is such a y-root, then by Lemma 2.3.9 the algorithm will eventually reach a recursive depth level of d where the roots s it has recursed on are the successive coefficients of $g(x)$. At this point, the coefficients of $g(x)$ are in the array $g[0,\ldots,d-1]$ and $Q_i = Q_{d-1}$ in the algorithm has been defined the same as Q_{d-1} in the lemma. Let r_j be the power of x such that $Q_j = \mathrm{red}(Q_{j-1}(x, xy + g_{j-1}) \bmod h(x)) = \frac{Q_{j-1}(x,xy+g_{j-1}) \bmod h(x)}{x^{r_j}}$ for $j \geq 1$ and let r_0 be such that $Q_0 = \frac{Q(x,y)}{x^{r_0}}$. Then

$$\begin{aligned}
Q_i(x, g[i]) &= Q_{d-1}(x, g_{d-1}) \bmod h(x) \\
&= \frac{Q_{d-2}(x, xg_{d-1} + g_{d-2})}{x^{r_{d-1}}} \bmod h(x) \\
&\vdots \\
&= \frac{Q_0(x, x^{d-1}g_{d-1} + x^{d-2}g_{d-2} + \cdots + xg_1 + g_0)}{x^{r_{d-1}+r_{d-2}+\cdots+r_1}} \bmod h(x) \\
&= \frac{Q(x, g(x))}{x^{r_{d-1}+\cdots+r_0}} \bmod h(x) \\
&= 0 \bmod h(x). \qquad (2.8)
\end{aligned}$$

Therefore, the algorithm will output $g(x)$ in step R6. Now suppose, that the polynomial $g(x)$ defined by the array at some point when step R6 is executed is outputted by the algorithm. In this case, equation 2.8 still holds and it must be that $Q(x, g(x)) = 0 \bmod h(x)$, so $g(x)$ is a y-root of $Q(x,y)$ modulo h(x). □

Proposition 2.3.12 *If Algorithm 2.3.10 is called with $Q \in \mathbb{F}_q[x,y]$ where $\deg_x(Q) = m$ and $\deg_y(Q) = \ell$, $d \in \mathbb{N}$, and $i = 0$, then the number of arithmetic operations in $\mathbb{F}_q$ used during the execution of the algorithm to its full recursive depth is*

$$\mathcal{O}(d\ell \log^2 \ell \log\log \ell (m + d^2\ell + \log q)), \text{ if } \ell < q$$

and it is

$$\mathcal{O}(d\ell \log^3 \ell \log\log \ell + d\ell^2 (m \log^2 m \log\log m + d^2\ell)), \text{ if } \ell \geq q.$$

Proof: The only difference between Algorithms 2.3.3 and 2.3.10 is the reduction modulo $h(x)$ in step R1 (and thereafter, Algorithm 2.3.10 may have fewer roots to recurse on). Using a fast Euclidean algorithm (see, for example, [38, Ch. 11]) we can perform this reduction using $\mathcal{O}(\ell m \log^2 m \log\log m)$ operations in $\mathbb{F}$. Overall, this adds as many as $\mathcal{O}(d\ell^2 m \log^2 m \log\log m)$ operations to the complexity of Algorithm 2.3.3 given in Proposition 2.3.7, which yields the complexity stated in this proposition. □

2.3.4 A Slight Generalization

We may find it advantageous to look at Algorithm 2.3.3 in slightly more general settings. For example, suppose we have a multivariate polynomial $Q(x, y_1, \ldots, y_r) \in \mathbb{F}[x, y_1, \ldots, y_r]$ and we wish to find every y_i-root of it, by which we mean every polynomial $g(x) \in \mathbb{F}[x]$ such that $Q(x, y_1, \ldots, y_{i-1}, g(x), y_{i+1}, \ldots, y_r) = 0$. By division with remainder we have that $g(x)$ is a y_i-root of Q if and only if $y_i - g(x)$ is a factor of Q. In this case, $Q = P(x, y_1, \ldots, y_r)(y_i - g(x))$ for some nonzero $P \in \mathbb{F}[x, y_1, \ldots, y_r]$. For a sufficiently general choice of $\boldsymbol{a} \in \mathbb{F}^r$ (that is, if we choose $\boldsymbol{a}$ so that it does not correspond to a root of P), $P(x, a_1, \ldots, a_{i-1}, y_i, a_{i+1}, \ldots, a_r) = P'(x, y_i) \neq 0$, so we can find $g(x)$ by running Algorithm 2.3.3 on $Q(x, a_1, \ldots, a_{i-1}, y_i, a_{i+1}, \ldots, a_r) = P'(x, y_i)(y_i - g(x)) \neq 0$.

Chapter 3
List-Decoding Evaluation Codes

3.1 Introduction

The minimum distance decoding problem has been well-studied for Reed-Solomon, BCH, and other popular algebraic geometry codes (see, for example, [1], [28], or [32]). Efficient decoding algorithms, such as the Berlekamp-Massey algorithm, have been devised that accurately correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors in a received word. This is a significant achievement as it has allowed for quick and effective decoding of error-correcting codes in many of our favorite devices, such as cd/dvd players and cell phones. But what about those special cases when a word is received for which the closest codeword is a distance of more than $\left\lfloor \frac{d-1}{2} \right\rfloor$ away? Does this happen?

Example 3.1.1 (Correctable error beyond the unique decoding radius) *Consider the* $[5, 2]$ *Reed-Solomon code over* $\mathbb{F}_5$

$$\mathcal{C} = \{(f(0), f(1), f(2), f(3), f(4)) | f \in \mathbb{F}_5[x], \deg(f) < 2\}.$$

The distance of $\mathcal{C}$ *is* $d = 5-2+1 = 4$*, so a classical decoding algorithm can correct up to* $\left\lfloor \frac{4-1}{2} \right\rfloor = 1$ *error in it. Suppose the codeword* $\boldsymbol{c} = (2, 2, 2, 2, 2) \in \mathcal{C}$ *was transmitted and the word* $\boldsymbol{r} = (2, 2, 2, 0, 0)$ *was received at the end of the channel. Since two errors occurred, the Berlekamp-Massey algorithm (which only decodes errors up to weight* $\left\lfloor \frac{d-1}{2} \right\rfloor$*) would fail to find the closest codeword, although it would at least detect that some errors occurred. In theory, since* $d = 4$ *and our received word is distance* 2 *away from the sent codeword, it is possible that there are other codewords of distance two away from* $\boldsymbol{r}$*. However, by inspection we find that* $\boldsymbol{c}$ *is the* unique *closest codeword to* $\boldsymbol{r}$ *and the next closest codewords (there are seven of them) are distance* 3 *away.*

Example 3.1.1 demonstrates in a small MDS code that it is possible to receive a word with no codeword within the standard error-correcting bound, which confounds most classical decoding algorithms. We may wonder whether this is a phenomenon we will encounter in other codes like, for example, the longer Reed-Solomon codes utilized in practice.

Example 3.1.2 (Classical correction-bound spheres fail to cover space) *Consider a popular* $[255, 223]$ *Reed-Solomon code over* $\mathbb{F}_{2^8}$*,* $\mathcal{C}$*, whose distance is* $d = 33$*. The standard error-correcting radius of this code is* 16*. Now* $\mathcal{C}$ *is not close to being a perfect code, but if it were perfect then the Hamming sphere of radius* 16 *about each codeword would have to contain* $\frac{256^{255}}{256^{223}} = 256^{33} \approx 1.1579 \times 10^{77}$ *vectors. As it is, each such sphere only contains* $\sum_{i=0}^{16} \binom{255}{i} 255^i \approx 3.0209 \times 10^{63}$ *vectors, meaning the vast majority of possible received words in* $\mathbb{F}^n$ *are outside the standard error-correcting radius of any codeword in* $\mathcal{C}$*. Indeed, one must take a radius of* 20 *before the number of vectors covered (with multiplicity) by the codeword spheres (*3.4951×10^{77} *in this case) exceeds the number of vectors in the whole space.*

In fact, the only general case where this cover shortage will not happen is in the case of *perfect* codes because they achieve the sphere-packing bound. So it seems there are many words out there that could have a unique closest codeword and would be correctable except that they lie outside the classical $\lfloor \frac{d-1}{2} \rfloor$ decoding radius. But even if we encountered an isolated word that was uncorrectable by standard decoding algorithms and did not have a unique closest codeword, it would be beneficial to at least know what the nearest codewords were. Then we would have some idea of where to start and could make an educated guess, perhaps with the aid of other probabilistic information (such as the transmission frequency of individual codewords), as to which codeword was most likely sent.

With this in mind, we are encouraged to alter the classical decoding problem of finding the unique (if it exists) closest codeword to our received word. We instead focus our effort on finding a *list* of all of the codewords within some prescribed distance of the received word. More formally, the list decoding problem can be stated in two ways, depending on whether you prefer to bound the length of the list or the decoding radius. We begin with the latter, which is prevalent among the two definitions in our work.

Definition 3.1.3 (List Decoding by Distance) *Given a code $\mathcal{C} \subseteq \mathbb{F}^n$, a vector $\boldsymbol{r} \in \mathbb{F}^n$, and a distance threshold $\tau \in \mathbb{N}$, the **(bounded distance) list decoding problem** is to find the list*

$$\mathcal{L}_{\tau,\boldsymbol{r}} = \{\boldsymbol{c} \in \mathcal{C} \mid \operatorname{dist}(\boldsymbol{c}, \boldsymbol{r}) \leq \tau\} = B_\tau(\boldsymbol{r}) \cap \mathcal{C}$$

*of all codewords within the Hamming radius τ of r. A **radius-τ decoder** is an algorithm for computing the function $\delta_\tau : \mathbb{F}^n \to 2^{\mathcal{C}}$ such that $\delta_\tau(\boldsymbol{r}) = \mathcal{L}_{\tau,\boldsymbol{r}}$ for all $\boldsymbol{r} \in \mathbb{F}^n$. We refer to the function δ_τ itself as a radius-τ decoder when we are not concerned with the actual method used to compute it.*

Most classical decoding algorithms give rise to a radius-$\lfloor \frac{d-1}{2} \rfloor$ decoder. As mentioned, the only problem with these decoders is that generally the $\lfloor \frac{d-1}{2} \rfloor$-spheres about all of the codewords do not come close to covering $\mathbb{F}^n$, so there is much to be gained in finding radius-τ decoders with $\tau > \lfloor \frac{d-1}{2} \rfloor$. Ideally, we would like to find a radius-τ decoder such that the τ-spheres about $\mathcal{C}$ cover $\mathbb{F}^n$ but overlap as little as possible. This way, every τ-sphere in the space would contain a codeword so we could decode any received word $r \in \mathbb{F}^n$ but the lists would not be so long as to make choosing the best candidate a burden. Also, it is essential that this decoder can be computed efficiently.

The other approach to list decoding is more concerned about the size of the lists.

Definition 3.1.4 (List Decoding by List Length) *Given a code $\mathcal{C} \subseteq \mathbb{F}^n$, a vector $\boldsymbol{r} \in \mathbb{F}^n$, and a list length bound $\ell \in \mathbb{N}$, the **list-ℓ decoding problem** is to find a list $\mathcal{L}(\boldsymbol{r}) \subseteq \mathcal{C}$, $|\mathcal{L}(\boldsymbol{r})| \leq \ell$, of no more than ℓ candidate codewords for decoding $\boldsymbol{r}$. A **list-ℓ decoder** is a function $\delta_\ell : \mathbb{F}^n \to 2^{\mathcal{C}}$ such that $|\delta_\ell(\boldsymbol{r})| \leq \ell$ for all $\boldsymbol{r} \in \mathbb{F}^n$. The **decoding radius** of δ_ℓ,*

$$\tau = \max(\{t \in \mathbb{N}_0 \mid \forall \boldsymbol{r} \in \mathbb{F}^n : \mathcal{L}_{t,r} \subseteq \delta_l(r)\} \cup \{-1\})$$

is the maximum radius t for which $\delta_\ell(\boldsymbol{r})$ contains all codewords within Hamming distance t for every vector $\boldsymbol{r}$. If no such radius t exists then we say the decoding radius is -1.

Most classical decoding algorithms do not quite yield a list-1 decoder because for some words $\boldsymbol{r}$ they return the empty list. To improve upon the error rates of classical decoding algorithms it will be necessary to consider list-ℓ decoders with $\ell > 1$.

We shall discuss the advantages of both approaches to the list decoding problem as we go along. For now, the bounded distance list decoding approach will be awarded slight preference because it lends itself toward comparisons with classical decoding algorithms. Though, to be fair, we shall be keenly interested in both the list sizes and the decoding radii of our decoders. But since the decoding radius is identical to the number of errors it is possible to correct in the code we will tend to look at the maximum list size as a function of the decoding radius or the corresponding error rate, wherefore we make the following definitions.

Definition 3.1.5 *Let $\mathcal{C}$ be a q-ary code of length n. For any $p \in [0,1]$ and $\ell \in \mathbb{N}$, we say that $\mathcal{C}$ is (p,ℓ)-**decodable** if*

$$|\mathcal{L}_{pn,\boldsymbol{r}}| \leq \ell, \text{ for all } \boldsymbol{r} \in \mathbb{F}_q^n.$$

*For any $p \in [0,1]$ the **decoding list length (up to radius** pn**)** of $\mathcal{C}$ is $\ell = \min\{L \mid \mathcal{C}$ is (p,L)-decodable$\}$. For any list size $\ell \in \mathbb{N}$ the **decoding radius (up to list length** ℓ**)** of $\mathcal{C}$ is $p = \max\{P \mid \mathcal{C}$ is (P,ℓ)-decodable$\}$. For a function $L : \mathbb{N} \to \mathbb{N}$, we say that a family of codes is $(p, L(n))$-**list decodable** if each code in the family is $(p, L(n))$-list decodable where n is the length of the code. If the function $L(n)$ is a constant ℓ, then we say that the family is (p,ℓ)-list decodable.*

Equivalently, we could say that $\mathcal{C}$ is (p,ℓ)-decodable if there exists a function $\delta : \mathbb{F}^n \to 2^{\mathcal{C}}$ that is both a radius-(pn) decoder and a list-ℓ decoder.

3.2 Bounds on List Decoding

The bound we will be most interested in for list decoding will be for the relationship between the error rate and information rate of codes. This bound is closely related to the well-known theorem by Claude Shannon ([35], 1948) which shows (non-constructively) that for every channel there exist codes that can be transmitted with arbitrarily high reliability for all information rates up to the capacity of the channel.

Informally, the *capacity* of the channel is a measure of the maximum amount of mutual information shared by the sent and received signals at either end of the channel. It is based on the bit-wise reliability of the channel and the marginal distribution of messages that we choose to send over the channel. Mutual information is the amount of information that can be obtained about one random variable by observing another one. Explicitly, the *mutual information* of a joint probability distribution function $p(x,y) : \mathbb{X} \times \mathbb{Y} \to [0,1]$ for two discrete random variables X and

Y is

$$I(p) = \sum_{x,y:p(x,y)\neq 0} p(x,y) \log_{|\mathbb{X}|} \left(\frac{p(x,y)}{p_1(x)p_2(y)} \right),$$

where $p_1(x) = \sum_{y\in\mathbb{Y}} p(x,y)$ and $p_2(y) = \sum_{x\in\mathbb{X}} p(x,y)$ are the marginal distribution functions. In our context, $\mathbb{X}$ and $\mathbb{Y}$ will be the sets of messages sent and received, respectively, at the two ends of the channel. We will assume that the conditional probability distribution $p(y \mid x)$ is fixed according to the channel's properties. The joint distribution $p(x,y) = p(y \mid x)p_1(x)$ is then completely determined by the marginal distribution $p_1(x)$ of messages we choose to send across the channel. The capacity of a transmission channel T is formally defined as

$$\text{cap}(T) = \max_{p_1} I(p),$$

where the maximum is taken over all distributions $p_1 : \mathbb{X} \to [0,1]$. To have some grasp of the uncertainty attenuating the mutual information of each distribution scheme and restricting the capacity of the channel it is useful to talk about entropy, which is a measure of the uncertainty associated to the value of a random variable. Given a probability mass function $p : \mathbb{X} \to [0,1]$ for a discrete random variable X, the entropy of X is

$$H_p(X) = -\sum_{x\in\mathbb{X}:p(x)\neq 0} p(x) \log_{|\mathbb{X}|} p(x).$$

It is worth noting here that the entropy of X is maximized to 1 when X is uniformly distributed (all values are equally likely so the outcome of a trial is as uncertain as possible) and it is minimized to 0 when $p(x) = 1$ for some $x \in \mathbb{X}$ (X is always x in this case, so there is no uncertainty).

Example & Definition 3.2.1 *Let SC_q be the memoryless q-ary symmetric channel with crossover probability p. That means our message space has size $|\mathbb{X}| = q$ (note: $\mathbb{X} = \mathbb{Y}$), each message is an independent, identically-distributed random variable X, and the probability of error is p with all possible erroneous messages equally likely, so the probability of observing y if x is sent is*

$$p(y \mid x) = \begin{cases} 1-p, & \text{if } y = x, \\ \frac{p}{q-1}, & \text{if } y \neq x. \end{cases}$$

*The entropy function in this case is formally known as the **q-ary entropy function** and it is denoted as*

$$H_q(p) = -p \log_q p - (1-p) \log_q (1-p) + p \log_q (q-1). \tag{3.1}$$

It is a well-known result that the capacity of this channel is $\text{cap}(SC_q) = 1 - H_q(p)$ (see Example 1.11 of [32] for a proof).

In his seminal paper on information theory, Shannon introduced "the Fundamental Theorem for a Discrete Channel with Noise" ([35], Theorem 11, it is now known more

popularly as **Shannon's Coding Theorem**), which related the channel capacity directly to the highest rate attainable in codes that perform arbitrarily well in terms of reliability. His theorem states that for a channel of capacity C, for any $\kappa \leq C$ there exists codes (of sufficiently large length) with rates at least κ such that the probability of error in block decoding over the channel is arbitrarily small. However, for any $\kappa > C$, the probability of a decoding error for any code of rate κ or more is at least $\kappa - C$ (although for sufficiently large lengths, there exists codes of rate at least κ for which the probability of error is arbitrarily close to $\kappa - C$). For the q-ary symmetric channel, Shannon's Theorem says that as long as $p < \frac{q-1}{q}$ one can communicate over the channel with arbitrarily small probability of error using codes with rates as high as $1 - H_q(p)$, but not with codes with higher rates than $1 - H_q(p)$. The proof of this was nonconstructive and it remains an open challenge to find such codes and their decoding algorithms. In the realm of list decoding, in turns out that there is a similar bound on the rates of codes that can be decoded efficiently.

The maximum length of the codeword list $\mathcal{L}_{\tau,\boldsymbol{v}}$ over all $\boldsymbol{v} \in \mathbb{F}^n$ is an *a priori* lower bound on the complexity of any radius-τ decoding algorithm for $\mathcal{C}$. In order for a code $\mathcal{C}$ to admit a radius-τ decoder that is computable in polynomial time it is therefore necessary that the number of codewords in every Hamming sphere of radius τ in $\mathbb{F}^n$ be bounded by a polynomial in n. In other words, if a family of codes are to be list decoded up to an error rate of p in time polynomial the code length n, then the decoding list length alone must be bounded by some polynomial $L(n)$. It has been known since at least the early 1980's that for any error rate p the minimum decoding list length $L(n)$ of any q-ary code with an information rate $\kappa \geq 1 - H_q(p)$ is exponential in n. To see this, we first present a bound we will need on the volume (number of vectors) inside a Hamming sphere in $\mathbb{F}_q^n$. These results and their proofs are standard (see, for example, [32] or [34]).

Lemma 3.2.2 *For $q, n, t \in \mathbb{N}$ with $q \geq 2$ and $n > t$,*

$$\binom{n}{t}(q-1)^t > e^{-}\frac{19}{150}\sqrt{\frac{2}{\pi n}} \cdot q^{nH_q(t/n)}.$$

Proof: By Stirling's approximation ([7], equation (3.17)) we have

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

Equations (3.19) and (3.20) in [7] actually provide a more precise bound of

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n}, \text{ where } \frac{1}{12n+1} < \alpha_n < \frac{1}{12n}. \tag{3.2}$$

From this expression for $n!$ we obtain

$$\begin{aligned}\binom{n}{t}(q-1)^t &= \frac{n!}{(n-t)!t!}(q-1)^t \\ &= \frac{\sqrt{2\pi n}(n/e)^n e^{\alpha_n}}{\sqrt{2\pi(n-t)}((n-t)/e)^{n-t}e^{\alpha_{n-t}}\sqrt{2\pi t}(t/e)^t e^{\alpha_t}}(q-1)^t\end{aligned}$$

$$
\begin{aligned}
&= \left(\sqrt{\frac{n}{2\pi(n-t)t}}\right)\left(\frac{n^n}{(n-t)^{(}n-t)t^t}\right)e^{\alpha_n-\alpha_{n-t}-\alpha_t}(q-1)^t \\
&= \left(\sqrt{\frac{n}{2\pi(n-t)t}}\right)\left(\frac{q^{n\log_q n}}{q^{(n-t)\log_q(n-t)}q^{t\log_q t}}\right)e^{\alpha_n-\alpha_{n-t}-\alpha_t}q^{t\log_q(q-1)} \\
&= \left(\sqrt{\frac{n}{2\pi(n-t)t}}\right)e^{\alpha_n-\alpha_{n-t}-\alpha_t}q^{n\log_q n-(n-t)\log_q(n-t)-t\log_q t+t\log_q(q-1)} \\
&= \left(\sqrt{\frac{n}{2\pi(n-t)t}}\right)e^{\alpha_n-\alpha_{n-t}-\alpha_t}q^{n(-(1-\frac{t}{n})\log_q(1-\frac{t}{n})-\frac{t}{n}\log_q\frac{t}{n}+\frac{t}{n}\log_q(q-1))} \\
&= \left(\sqrt{\frac{n}{2\pi(n-t)t}}\right)e^{\alpha_n-\alpha_{n-t}-\alpha_t}q^{nH_q(t/n)}. \qquad (3.3)
\end{aligned}
$$

One can easily check that

$$\alpha_n-\alpha_{n-t}-\alpha_t > \frac{1}{12n+1}-\frac{1}{12(n-t)}-\frac{1}{12t} \geq -\frac{12n^2-11^n+12}{12(12n^2-11n-1)} \geq -\frac{19}{150}$$

since $\frac{1}{12n+1}-\frac{1}{12(n-t)}-\frac{1}{12t}$ (which is always negative) is minimized when $t=1$ or $t=n-1$ to $-\frac{12n^2-11n+12}{12(12n^2-11n-1)}$, which is minimized when $n=2$ to $-19/150$. On the other side, it is even easier to see that

$$\alpha_n-\alpha_{n-t}-\alpha_t < \frac{1}{12n}-\frac{1}{12(n-t)+1}-\frac{1}{12t+1} = \frac{12nt-12n^2-12t^2-n+\frac{1}{12}}{n(12n-12t+1)(12t+1)} < 0$$

since the denominator of the last fraction is always positive while its numerator is always negative. From this it follows that

$$0.88102 \approx e^{-19/150} < e^{\alpha_n-\alpha_{n-t}-\alpha_t} < e^0 = 1.$$

As for $\sqrt{\frac{n}{2\pi(n-t)t}}$, this expression is minimized when $t=\frac{n}{2}$ to $\sqrt{\frac{2}{\pi n}}$. Thus from equation 3.3 we have that

$$\binom{n}{t}(q-1)^t > \sqrt{\frac{2}{\pi n}}e^{-19/150}q^{nH_q(t/n)} > \frac{7}{10\sqrt{n}}\cdot q^{nH_q(t/n)} \in \Omega(n^{-0.5}q^{nH_q(t/n)}).$$

□

Lemma 3.2.2 is slightly stronger than the bound of $\frac{1}{n+1}\cdot q^{nH_q(t/n)}$ given in [32, Lemma 4.8] (although their bound works also in the cases of $n=t$[1] and $n=0$, but these cases will not be very interesting to us). More importantly, the lemma gives us a firm grasp of the number of vectors that are a distance of t away from a given point in $\mathbb{F}_q^n$. We use this result now to get a lower bound on the volume of a sphere about a point.

[1]For us, t will end up being the number of errors we want to correct (usually denoted τ).

Corollary 3.2.3 ($|B_t(\boldsymbol{v})|$ Lower Bound) *For $q, n, t \in \mathbb{N}$ with $q \geq 2$ and $n > t$, for any $\boldsymbol{v} \in \mathbb{F}_q^n$,*

$$|B_t(\boldsymbol{v})| > \frac{7}{10\sqrt{n}} \cdot q^{nH_q(t/n)} \in \omega(q^{n(H_q(t/n)-\delta)}),$$

for any $\delta > 0$. In fact, for any constant $c > 0$, $|B_t(\boldsymbol{v})| > cq^{n(H_q(t/n)-\delta)}$ whenever $n \geq \frac{1}{\delta}\log_q(10c/7)$.

Proof: The number of vectors in the Hamming sphere $B_t(\boldsymbol{v})$ of radius t about $\boldsymbol{v}$ is

$$\sum_{i=0}^{t} \binom{n}{i}(q-1)^i \geq \binom{n}{t}(q-1)^t,$$

so the inequality follows immediately from Lemma 3.2.2. The asymptotic classification follows because if $n \geq \frac{1}{\delta}\log_q(10c/7)$, then $\frac{c}{q^{n\delta}} \leq \frac{7}{10\sqrt{n}}$ and from this it follows that $cq^{n(H_q(t/n)-\delta)} \leq \frac{7}{10\sqrt{n}} \cdot q^{nH_q(t/n)}$. $\square$

The bound of Corollary 3.2.3 is actually fairly tight when $0 < t/n \leq \frac{q-1}{q}$. We demonstrate this with a lower bound from [32, Lemma 4.7].

Lemma 3.2.4 ($|B_t(\boldsymbol{v})|$ Upper Bound) *For $q, n, t \in \mathbb{N}$ with $q \geq 2$ and $\frac{t}{n} \leq \frac{q-1}{q}$, for any $\boldsymbol{v} \in \mathbb{F}_q^n$,*

$$|B_t(\boldsymbol{v})| < q^{nH_q(t/n)}.$$

Proof: Since $t/n \leq 1 - 1/q$, $\frac{t/n}{(1-t/n)(q-1)} \leq 1 < \frac{1}{(1/q)(q-1)}$. Armed with this fact we make the following computation.

$$\begin{aligned}
|B_t(\boldsymbol{v})| &= \left(\sum_{i=0}^{t}\binom{n}{i}(q-1)^i\right)\left(\frac{q^{nH_q(t/n)}}{q^{n(-(1-t/n)\log_q(1-t/n)-t/n\log_q t/n+t/n\log_q(q-1))}}\right) \\
&= \left(\sum_{i=0}^{t}\binom{n}{i}(q-1)^i\right)\left((1-t/n)^{n-t}(t/n)^t(q-1)^{-t}\right)q^{nH_q(t/n)} \\
&> \left(\sum_{i=0}^{n}\binom{n}{i}(q-1)^i\left(\frac{t/n}{(1-t/n)(q-1)}\right)^{i-t}\right)\cdot\frac{(1-t/n)^{n-t}(t/n)^t q^{nH_q(t/n)}}{(q-1)^t} \\
&= \left(\sum_{i=0}^{n}\binom{n}{i}(t/n)^i(1-t/n)^{n-i}\right)q^{nH_q(t/n)} \\
&= (t/n+(1-t/n))^n q^{nH_q(t/n)} \\
&= q^{nH_q(t/n)}.
\end{aligned}$$

$\square$

We now present a couple of theorems which in some sense relate the limit of the Shannon Coding Theorem to upper and lower bounds on list decoding by showing that for codes with rates above the capacity of the channel, the decoding list length

grows exponentially in the length of the code (in fact, not just the longest list length but even the average list size grows exponentially in the code length) and, conversely, codes with polynomial decoding list lengths exist for all rates up to the channel capacity.

Theorem 3.2.5 (Efficient List Decoding Upper Bound) *Let $q \geq 2$. Fix an error rate $p \in [0,1]$. Suppose $\mathcal{C}$ is a q-ary code with information rate $\kappa = 1 - H_q(p) + \delta$ for some $\delta > 0$. Then the decoding list length up to radius pn of $\mathcal{C}$ is greater than $q^{n\delta}$.*

Proof: Let ℓ be the decoding list length of $\mathcal{C}$ up to radius p. This means that no Hamming sphere of radius pn in $\mathbb{F}_q^n$ contains more than ℓ codewords of $\mathcal{C}$. Since $|\mathcal{C}| = q^{\kappa n}$ the density of $\mathcal{C}$ in $\mathbb{F}_q^n$ is $\frac{q^{\kappa n}}{q^n} = q^{n(\kappa-1)}$. Therefore, the average number of codewords in $B_{pn}(\boldsymbol{v})$ over all $\boldsymbol{v} \in \mathbb{F}_q^n$ is

$$\frac{|B_{pn}(\boldsymbol{v})| \cdot |\mathcal{C}|}{|\mathbb{F}^n|} > q^{nH_q(t/n)} \cdot q^{n(\kappa-1)} = q^{n\delta},$$

the inequality following from Corollary 3.2.3. Since the average is no larger than the maximum number of codewords in any sphere of radius pn in $\mathbb{F}^n$, we get that the decoding list length of $\mathcal{C}$ is $\ell > q^{n\delta}$. $\square$

Corollary 3.2.6 *Let $q \geq 2$. Fix an error rate $p \in [0,1]$. Let $\delta > 0$. If $\{\mathcal{C}_n\}$ is a family of $(p, L(n))$-list decodable q-ary codes with arbitrarily long block lengths and rates greater than $1 - H_q(p) + \delta$, then*

$$L(n) \in \Omega(q^{n\delta}).$$

Theorem 3.2.5 is important to us because it tells us a limit to the potential decoding radius for list decoding. It says that the decoding list length (up to radius pn) grows exponentially in the length of the code for codes of rates higher than the capacity of the q-ary symmetric channel. This means that a radius-(pn) decoder for the code would require an exponential amount of time just to output the list of nearest codewords to a received (and of course it must somehow find the list first) so there is no hope for efficiently decoding up to radius-pn in codes with rates higher than the q-ary channel capacity. To put it another way, the theorem can be reformed to say that there do not exist any q-ary, $(p, L(n))$-decodable codes of length n with rates higher than $1 - H_q(p)$ where $L(n)$ is any sub-exponential function. More simply, there are no codes that can be efficiently list decoded up to radius pn with rates above $1 - H_q(p)$. A remarkable fact is that we can find codes that can be efficiently decoded up to radius pn with rates arbitrarily close to $1 - H_q(p)$.

Theorem 3.2.7 (Efficient List Decoding Lower Bound) *For every $q, \ell \in \mathbb{N}_2$ and $p \in [0, 1 - 1/q]$, there exists a (p, ℓ)-list decodable q-ary code with rate at least $1 - H_q(p) - \frac{1}{\ell}$.*

Proof: We follow a course similar to the proof given in [17, Theorem 3.5]. Other proofs can be found in [34] or [11].

Pick a large code length $n \geq \ell(\ell+1)$ and assume pn is an integer for simplicity. Let $\mathcal{C}$ be a set of $M = q^{n(1-H_q(p)-\frac{1}{\ell+1})}$ codewords chosen uniformly at random *with replacement* from $\mathbb{F}_q^n$. For any $\boldsymbol{v} \in \mathbb{F}_q^n$, the probability that an arbitrary subset of $\ell+1$ codewords of $\mathcal{C}$ is contained in the the Hamming sphere $B_{pn}(\boldsymbol{v})$ is

$$\left(\frac{|B_{pn}(\boldsymbol{v})|}{q^n}\right)^{\ell+1} < q^{-n(1-H_q(p))(\ell+1)}, \tag{3.4}$$

the inequality following from Lemma 3.2.4. Hence, the probability that some subset of $\ell+1$ codewords is contained in $B_{pn}(\boldsymbol{v})$ for some $\boldsymbol{v} \in \mathbb{F}_q^n$ is less than

$$\binom{M}{\ell+1} \cdot q^n \cdot q^{-n(1-H_q(p))(\ell+1)} \leq \frac{M}{(\ell+1)!} \cdot q^{-n(1-H_q(p)-\frac{1}{\ell+1})} < \frac{1}{3}.$$

So the probability that $\mathcal{C}$ is (p,ℓ)-decodable is greater than $2/3$. Moreover, the probability that there are at least $M/2$ distinct codewords in $\mathcal{C}$ is

$$\frac{\binom{q^n}{M/2}\left(\!\!\binom{q^n}{M/2}\!\!\right)}{\left(\!\!\binom{q^n}{M}\!\!\right)} > \frac{1}{2}.$$

Therefore, it must be that at least $1/3$ of the possible codes we could have chosen for $\mathcal{C}$ are (p,ℓ)-decodable *and* have $M/2$ or more distinct codewords. Since $n \geq \ell(\ell+1)$ we have $q^{-n/(\ell+1)} \geq 2q^{-n/\ell}$ which implies

$$M/2 = q^{n(1-H_q(p)-\frac{1}{\ell+1})}/2 \geq q^{n(1-H_q(p)-\frac{1}{\ell})}.$$

Thus, there exists a q-ary code of rate at least $\log_q\left(\frac{M/2}{q^n}\right) = (1 - H_q(p) - \frac{1}{\ell})$ that is (p,ℓ)-decodable. □

Corollary 3.2.8 ([17]) *For every $q, \ell \in \mathbb{N}_2$ and $p \in [0, 1-1/q]$, there exists a family of (p,ℓ)-list decodable q-ary codes with rates of at least $1 - H_q(p) - \frac{1}{\ell}$.*

Theorems 3.2.5 and 3.2.7 establish $1 - H_q(p)$ as the benchmark for information rates in codes that we would like to list decode up to radius p. For this reason we may call $1 - H_q(p)$ the *list decoding capacity* (of a q-ary channel). It represents the amount of information (as opposed to redundancy) we may transmit per symbol in a code and still have a code that is sparse enough that the number of codewords within distance pn (which is the amount of misinformation we would expect at the receiving end of a q-ary channel with crossover probability p) of any particular codeword is polynomially bounded (by Theorem 3.2.7). The list decoding capacity is not entirely the same as the channel capacity in the Shannon-theoretic sense. For one thing, the concept of decoding failure is different. In Shannon's theorem, which is based on

minimum distance decoding, the notion of a decoding error is an instance where the codeword that was sent over the channel is not the unique closest codeword to the word that was received. The actual time it takes to find the nearest codeword is of no concern in this case. For list decoding, we view decoding failure as an instance where the decoder cannot determine all of the codewords within a prescribed distance from the received word in polynomial time. Theorem 3.2.5 precludes the possibility of producing such a list in codes with rates above the channel capacity because the list sizes grow exponentially long. Theorem 3.2.7, on the other hand, tells us that codes do exist with rates arbitrarily close to the capacity whose list lengths are polynomially small, yet it gives no assurance that there is any means of computing one of these lists in polynomial time. The question remains whether we can find a polynomial time decoding algorithm for the codes with rates approaching the list decoding capacity. In pursuit of the answer it is interesting to look at the asymptotic nature of the list decoding capacity in the limit of increasing alphabet size. Observe that as q goes to infinity, the capacity approaches a simple limit,

$$1 - H_q(p) = 1 + p \log_q p + (1-p) \log_q (1-p) - p \log_q (q-1) \stackrel{q \to \infty}{\longrightarrow} 1 - p.$$

The following proposition provides a better understanding of how quickly the capacity approaches $1 - p$.

Proposition 3.2.9 *For any rate $\kappa \in (0,1)$ and any real $\delta \in (0, 1-p)$, there exists a q-ary code of rate at least κ that is $(1-\kappa-\delta, \ell)$-list decodable with $\ell < \frac{1}{\delta}$ and $q < 2^{1/\delta}$.*

Proof: From the assumptions of the proposition, we have

$$\begin{aligned} 1 - H_q(p) &= 1 + \frac{p \log p + (1-p) \log (1-p)}{\log q} - p + p \log_q \left(\frac{q}{q-1} \right) \\ &= 1 - \frac{H_2(p)}{\log q} - p + p \log_q \left(\frac{q}{q-1} \right) \\ &\geq 1 - p - \frac{1}{\log q} \\ &> 1 - p - \delta \end{aligned}$$

since $q > 2^{1/\delta}$. The statement then follows from Theorem 3.2.7. $\Box$

From the proposition it is clear that for any κ we can find codes of rate no less than κ in which we can correct arbitrarily close to a fraction of $1 - \kappa$ errors through list decoding. One caveat is that in order to approach the capacity of $1 - \kappa$, the alphabet size of the codes in point must grow unbounded.

The bounds given here on the capacity of list decoding may be theoretically satisfying, although some questions remain about what exactly happens at the boundaries - what can be said of codes with rates equal to the capacity $1 - H_q(p)$? Moreover, from a practical point of view there are two important unfulfilled components to these capacity results: (1) an explicit description of codes that achieve (at least asymptotically) the list decoding capacity, and (2) an explicit and efficient algorithm for list

decoding such codes. Guruswami and Rudra settled task (1) by utilizing folded Reed-Solomon Codes (which we will describe in Chapter 4). They did give a description of an algorithm that works in a special case but no explicit description of the algorithm is given that works for the actual codes that approach capacity. We endeavor to provide a concrete description and analysis in Chapter 4 and hope to improve upon the complexity bound estimates given in [17]. First, though, we cover an essential list decoding algorithm upon which the algorithm we present in the next chapter is based.

3.3 Sudan's Algorithm

The idea of bounded-distance list decoding was first studied by Peter Elias [10] in 1957 when he utilized the notion to investigate the upper and lower bounds on the average probability of error for binary symmetric channels, specifically in relation to Shannon's Coding Theorem for those channels. He showed that the method could produce codes with low average error probabilities for all rates. However, he made no attempt to determine how this method could be implemented efficiently. Indeed, there were no significant results to this end until forty years later when Madhu Sudan published a seminal paper [37] describing the first polynomial time list decoding algorithm that achieved error rates asymptotically better (albeit only for low information rates) than classical decoding algorithms.

Sudan's algorithm consists of two stages; the first stage is interpolating a certain bivariate polynomial and the second is factoring it. The implementation of his algorithm is conceptually quite simple but certain degree constraints on the interpolation polynomial require some care to ascertain. The algorithm itself applies specifically to Reed-Solomon codes and arose in the context of finding all univariate polynomials of bounded degree that passed through some sufficiently large subset of n points in the plane so let us begin there.

First, recall from Section 2.1.2 that the Reed-Solomon code $\mathrm{RS}(n, k, \boldsymbol{\alpha})$ is the image of all polynomials of degree less than k evaluated at some distinct $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$. We will assume henceforth that $n, k > 1$. Recall, the bounded distance-τ list decoding problem for $\mathrm{RS}(n, k, \boldsymbol{\alpha})$ for a given vector $\boldsymbol{r} \in \mathbb{F}^n$ is to find all $\boldsymbol{c} \in \mathrm{RS}(n, k, \boldsymbol{\alpha})$ within distance τ of $\boldsymbol{r}$; that is, all codewords that agree with $\boldsymbol{r}$ in at least $n - \tau$ components. Equivalently, if we can find all polynomials of degree less than k that interpolate at least $n - \tau$ of the points $P_i = (\alpha_i, r_i)$ for $i = 1, \ldots, n$ then by simply evaluating these polynomials at $\alpha_1, \ldots, \alpha_n$ we can construct the list of codewords that agree with $\boldsymbol{r}$ in at least $n - \tau$ places. By doing this we obtain a radius-τ decoder for $\mathrm{RS}(n, k, \boldsymbol{\alpha})$.

To construct this list of polynomials, it turns out to be beneficial to broaden our scope to bivariate polynomials. When we speak of a y-root of $h(x, y) \in \mathbb{F}[x, y]$ we will mean a polynomial $s(x) \in \mathbb{F}[x]$ such that $h(x, s(x)) = 0$. It is important here to recognize that y-roots of a bivariate polynomial behave in much the same way as roots of a univariate polynomial do. Namely, they correspond to certain linear factors of the polynomial (in this case y-linear).

Lemma 3.3.1 *Let $h \in \mathbb{F}[x, y]$. Then $f \in \mathbb{F}[x]$ is a y-root of h if and only if $y - f(x)$ is a factor of h.*

Proof: The converse of the lemma is obvious so all that needs to be shown is that $y - f(x)$ divides h if f is a y-root. Assume $h(x, f(x)) = 0$. By the Division Algorithm, we may write $h(x, y) = q(x, y)(y - f(x)) + r(x, y)$ where $q, r \in \mathbb{F}[x, y]$ and $\deg_y(r) < \deg_y(y - f(x)) = 1$, so $r(x, y)$ is a constant with respect to y. That is, $r = r(x, 0) \in \mathbb{F}[x]$. Then $h(x, f(x)) = 0$ implies $q(x, f(x))(f(x) - f(x)) + r(x, f(x)) = r(x, 0) = 0$ for every value of x. Hence, $r = 0$ and $h = q \cdot (y - f(x))$. □

Next, we will look for a nonzero polynomial $Q \in \mathbb{F}[x, y]$ of small degree that passes through all of the points $P_1, \ldots, P_n$. If its degree is sufficiently small, then its roots (and $(y - f(x))$-form factors) will correspond to the polynomials in $\mathbb{F}_k[x]$ we seek – those interpolating $n - \tau$ of the points P_i. We will often want to consider the $(1, k - 1)$-weighted degree of bivariate polynomials. Therefore, we will denote this weighted degree for $Q(x, y) = \sum q_{ij} x^i y^j \in \mathbb{F}[x, y]$ by $\deg_{k-1}(Q) = \max\{i + (k - 1)j \mid i, j : q_{ij} \neq 0\}$.

Proposition 3.3.2 *Suppose that $Q \in \mathbb{F}[x, y]$ passes through the points $P_1, \ldots, P_n$ and that $\deg_{k-1}(Q) < n - \tau$. If $f \in \mathbb{F}_k[x]$ is a polynomial interpolating at least $n - \tau$ of the points P_i, then f is a y-root of Q.*

Proof: Consider the polynomial $Q(x, f(x)) \in \mathbb{F}[x]$. The degree constraint for Q means that every term $ax^i y^j$ of Q satisfies $i + (k - 1)j < n - \tau$. Therefore, when we substitute $f(x)$ for y in each term $ax^i y^j$ of Q, the resulting polynomial $ax^i (f(x))^j$ has degree at most $i + (k - 1)j < n - \tau$ since $\deg(f) \leq k - 1$. So $\deg Q(x, f(x)) < n - \tau$.

Now, $Q(\alpha_i, r_i) = 0$ for $i = 1, \ldots, n$. Since $f(\alpha_i) = r_i$ for at least $n - \tau$ points α_i we have at these points that $Q(\alpha_i, f(\alpha_i)) = Q(\alpha_i, r_i) = 0$. So $Q(x, f(x))$ is a polynomial of degree less than $n - \tau$ with at least $n - \tau$ roots. Therefore, $Q(x, f(x))$ is uniformly 0 so f is a y-root of Q. □

We could easily find a nonzero polynomial Q that vanishes on all n of the points P_i (take, for example, $\prod_{i=1}^{n}(x - \alpha_i)$). However, for Q to be useful to us it should satisfy the weighted degree bound of Proposition 3.3.2. Our next step is to determine how large this bound $n - \tau$ has to be in order for a nonzero polynomial satisfying the premise of the proposition to exist. While the previous proposition told us how small $\deg_{k-1}(Q)$ has to be in order to possess the polynomials we seek as roots, the following proposition will tell us how large it should be if it is to interpolate our n points P_i. The balance between these two opposing constraints will be an important and delicate issue in constructing an algorithm that extends the error rate beyond classical bounds. We will exert considerable effort on finding a modus vivendi for them.

Proposition 3.3.3 *For any $\ell \in \mathbb{N}_0$, there exists a nonzero polynomial $Q \in \mathbb{F}[x,y]$ with y-degree at most ℓ that vanishes at $P_1, \ldots, P_n$ and satisfies*

$$\deg_{k-1}(Q) \leq \left\lceil \frac{n+1+(k-1)\binom{\ell+1}{2}}{\ell+1} \right\rceil - 1$$

Proof: Fix $\ell \in \mathbb{N}_0$. Let $d = \frac{n+1+(k-1)\binom{\ell+1}{2}}{\ell+1}$. We must show that there is a polynomial Q with $\deg_y(Q) \leq \ell$ and $\deg_{k-1}(Q) < d$ that is zero at each P_i. So we want $Q(\alpha_i, r_i) = 0$ for $i = 1, \ldots, n$. This condition gives us n homogeneous linear equations where the unknowns are the coefficients of Q. The number of possible nonzero coefficients in Q, whose $(1, k-1)$-degree is strictly bounded above by d, is

$$\begin{aligned}
\sum_{i=0}^{\ell}(d - i(k-1)) &= d(\ell+1) - (k-1)\binom{\ell+1}{2} \\
&= \frac{n+1+(k-1)\binom{\ell+1}{2}}{\ell+1}(\ell+1) - (k-1)\binom{\ell+1}{2} \\
&= n+1.
\end{aligned}$$

Since the system of equations under consideration has more unknowns than constraints, we are assured of a nontrivial solution which supplies the polynomial Q we desire and concludes the proof. $\square$

At this point, we assume no control over the values of n and k but we may choose ℓ however we like. Viewing the degree bound for Q above as a function of ℓ we have $d(\ell) = \frac{n+1}{\ell+1} + \frac{k-1}{2} \cdot \ell$. In Proposition 3.3.2 we want the degree of Q to be low, so we seek to minimize $d(\ell)$. Viewing it as a real-valued function, $d(\ell)$ is always positive and its second derivative, $d''(\ell) = \frac{2(n+1)}{(\ell+1)^3}$, shows that it is concave up. Therefore $\ell = \sqrt{\frac{2(n+1)}{k-1}} - 1$, which is the only zero of its first derivative, $d'(\ell) = \frac{-(n+1)}{(\ell+1)^2} + \frac{k-1}{2}$, minimizes the function d to the value

$$\begin{aligned}
d\left(\sqrt{\frac{2(n+1)}{k-1}} - 1\right) &= \frac{n+1}{\sqrt{\frac{2(n+1)}{k-1}}} + \frac{k-1}{2}\left(\sqrt{\frac{2(n+1)}{k-1}} - 1\right) \\
&= \sqrt{2(k-1)(n+1)} - \frac{k-1}{2}. \qquad (3.5)
\end{aligned}$$

Of course, when we perform the algorithm we will need to choose an integer value for ℓ. The critical point we found for $d(\ell)$ above is the absolute and only minimum point of d so either $\lfloor \ell_0 \rfloor$ or $\lceil \ell_0 \rceil$ for $\ell_0 = \sqrt{\frac{2(n+1)}{k-1}} - 1$ will yield the minimal output of $d(\ell)$ for an integer input value of ℓ. Which of the two choices is better will depend on the parameters n and k, though, as Table 3.1 shows. In fact, considering the row for $k = 43$ in the table, we cannot even say in general that rounding ℓ_0 produces the minimal value of d on integral inputs. So we will leave this dilemma open to case by case examination for now.

n	k	$\ell_0 = \sqrt{\frac{2(n+1)}{k-1}} - 1$	$d(\ell_0)$	$d(\lfloor \ell_0 \rfloor)$	$d(\lceil \ell_0 \rceil)$
255	31	3.131	108.935	109	111.2
255	33	3	112	112	112
255	35	2.881	114.939	$119.\bar{3}$	115
255	43	2.491	125.642	$127.\bar{3}$	127
255	45	2.411	128.093	$129.\bar{3}$	130
255	47	2.336	130.466	$131.\bar{3}$	133

Table 3.1: Minimizing values of $d(\ell)$ for some typical parameters n and k

One other interesting fact regarding the minimizing value ℓ_0 of the degree bound function $d(\ell)$ is that there are infinitely many code parameters $[n, k]$ for which $\lfloor \ell_0 \rfloor$ and $\lceil \ell_0 \rceil$ produce the same value of d. The proof of this fact uses a simple lemma which we prove first.

Lemma 3.3.4 *If x is an odd integer, then $x^2 - 9$ is divisible by 8.*

Proof: Since x is odd, both $x + 3$ and $x - 3$ are even and, moreover, because they differ by 6 exactly one of them is divisible by four. Therefore, their product, $x^2 - 9$, is divisible by 8. □

Proposition 3.3.5 *For any code dimension $k \geq 2$, there are infinitely many code lengths n for which the degree bound function $d(\ell)$ satisfies $d(\lfloor \ell_0 \rfloor) = d(\lceil \ell_0 \rceil)$ where ℓ_0 is the critical number of $d(\ell)$.*

Proof: We look for integers ℓ such that $d(\ell) = d(\ell + 1)$. Since $d(\ell)$ is concave up, there may be at most one such ℓ for any fixed $[n, k]$ and this ℓ is necessarily $\lfloor \ell_0 \rfloor$.

$$\begin{aligned} d(\ell) = d(\ell+1) &\iff \frac{n+1}{\ell+1} + \frac{k-1}{2} \cdot \ell = \frac{n+1}{\ell+2} + \frac{k-1}{2}(\ell+1) \\ &\iff (n+1)(-1) + \frac{k-1}{2}(\ell+1)(\ell+2)(-1) = 0 \\ &\iff \ell^2 + 3\ell + 2 + \frac{2(n+1)}{k-1} = 0 \end{aligned} \tag{3.6}$$

the solutions to equation (3.6) are

$$\ell = \frac{-3 \pm \sqrt{9 - \frac{8k-8n-16}{k-1}}}{2},$$

of which we care only about the positive possibility (since negative degree bounds are useless to us). To have ℓ be a positive integer, we need $x := \sqrt{9 - \frac{8k-8n-16}{k-1}}$ to be an odd integer greater than 3. After choosing such a value for x, we have

$$x^2 = 9 - \frac{8k - 8n - 16}{k-1} \Longrightarrow n = \frac{(x^2-9)(k-1)}{8} + k - 2.$$

Since x is odd, n is an integer by Lemma 3.3.4 and we obtain a pair $[n, k]$ for which $d(\ell) = d(\ell+1)$. It may be noted that $\ell = \frac{x-3}{2}$ here. Since ℓ_0 is the critical number of $d(\ell)$, it must lie between ℓ and $\ell+1$. Therefore, $\ell = \lfloor \ell_0 \rfloor$ and $\ell + 1 = \lceil \ell_0 \rceil$. Thus, for any code dimension $k \geq 2$, the lengths n for which the degree bound $d(\ell)$ is minimized at two consecutive integers $\lfloor \ell_0 \rfloor$ and $\lceil \ell_0 \rceil$ are

$$\left\{ n = \frac{(x^2-9)(k-1)}{8} + k - 2 \in \mathbb{N} \;\middle|\; x \geq 5 \text{ is an odd integer} \right\},$$

which is clearly an infinite set. □

Example 3.3.6 *Take dimension $k = 17$. Letting $x = 11$, we have $n = \frac{(11^2-9)(17-1)}{8} + 17 - 2 = 239$ and $\ell = \lfloor \ell_0 \rfloor = \frac{11-3}{2} = 4$. Thus, for a $[239, 17]$ Reed-Solomon code the minimal degree bound for Q will be $d(4) = d(5) = 80$, which is obtained with a y-degree bound of 4 or 5.*

We will conclude this section with a formal description of Sudan's Algorithm. The coding gains of this algorithm over classical achievements as well as optimal choices of the parameter ℓ will be examined more in the following sections. The algorithm is similar to the original one given by Sudan in [37] except that Sudan had an additional parameter m incorporated into the degree bound $d(\ell)$ that is unnecessary and his algorithm output the polynomials corresponding to the codewords that were within the prescribed distance to the received word rather than the codewords themselves.

Algorithm 3.3.7 (Sudan's List Decoding Algorithm for $\mathrm{RS}(n, k, \boldsymbol{\alpha})$**)**
Input: $\boldsymbol{r} = (r_1, \ldots, r_n)$, τ
Output: list of codewords, $\mathcal{L}_{\tau,\boldsymbol{r}}$, within Hamming distance τ of $\boldsymbol{r}$ if $d(\ell) \leq n - \tau$ for ℓ as computed in the algorithm

S1. Precompute $\ell_0 = \sqrt{\frac{2(n+1)}{k-1}} - 1$. Let ℓ be whichever of $\{\lfloor \ell_0 \rfloor, \lceil \ell_0 \rceil\}$ minimizes $d(s) = \frac{n+1}{s+1} + \frac{k-1}{2} \cdot s$. If $n - \tau < d(\ell)$, display "Warning: list may be incomplete!".

S2. Find a nonzero polynomial $Q \in \mathbb{F}[x, y]$ satisfying

(a) $Q(\alpha_i, r_i) = 0$ for $i = 1, \ldots, n$

(b) $\deg_{k-1}(Q) < d(\ell)$.

This step can be done by solving the system of equations given by condition (a) where the unknowns are the coefficients of Q for terms with $(1, k-1)$-weighted degree less than $d(\ell)$.

S3. Find all y-roots of $Q(x, y)$ of degree less than k using Algorithm 2.3.3.

S4. For each y-root $f(x)$ of Q from step S3, output the codeword $(f(\alpha_1), \ldots, f(\alpha_n))$ if its Hamming distance from $\boldsymbol{r}$ is no more than τ.

3.4 Selecting an Optimal List Size

The most crucial step of Algorithm 3.3.7 is the interpolation step: finding a polynomial Q interpolating the points P_i whose $(1, k-1)$-weighted degree is less than the degree bound $d(\ell)$ given in Proposition 3.3.3. Here, $\ell \in \mathbb{N}$ represents an upper bound on the y-degree of the polynomial $Q(x, y)$ that we seek and if an interpolation polynomial Q exists satisfying the degree constraints when $\ell = \ell_0$, then certainly one exists with no greater weighted degree for every $\ell \geq \ell_0$. To increase the error rate, we want $\deg_{k-1}(Q)$ as small as possible and while conceivably we might find better interpolation polynomials if we relaxed the y-degree constraint on Q, the fact is that the larger we allow the y-degree to be the greater the time complexity of the algorithm will be and, generally, the greater $\deg_{k-1}(Q)$ will be. It behooves us to know what ℓ minimizes the degree bound $d(\ell)$ and thereby maximizes the number of errors, τ, we can correct.

Now Propositions 3.3.2 and 3.3.3 told us that Algorithm 3.3.7 will correct τ errors if we can find a polynomial $Q \in \mathbb{F}[x, y]$ such that $\deg_{k-1}(Q) < n - \tau$ and that such a polynomial exists if $d(\ell) \leq n - \tau$ for some $\ell \in \mathbb{N}$. So we can correct up to $\tau = n - \min\{d(\ell) \mid \ell \in \mathbb{N}\}$ errors, which means the algorithm attains an error rate of $p = \frac{\tau}{n} = 1 - \frac{\min\{d(\ell)\}}{n}$. However, to understand this error rate we need to know which value $\ell_0 \in \mathbb{N}$ of ℓ minimizes $\frac{d(\ell)}{n}$. We discussed how to find ℓ_0 in Section 3.3 and saw that its value will vary according to the values of n and k. In fact, even for fixed rates $\kappa = \frac{k}{n}$, the minimizing value of ℓ will change as k and n vary, as shown in Table 3.2. However, asymptotically, the minimizing value of ℓ depends only on the rate $\kappa = \frac{k}{n}$.

Lemma 3.4.1 *For a fixed rate $\kappa = \frac{k}{n}$ define ℓ_0 as the unique integer such that*

$$\kappa \in \left(\binom{\ell_0 + 2}{2}^{-1}, \binom{\ell_0 + 1}{2}^{-1} \right].$$

Let $d_\ell(n) = \frac{n+1}{\ell+1} + \frac{k-1}{2} \cdot \ell$ for all $\ell \in \mathbb{N}$. Then the minimum of

$$\left\{ \lim_{n \to \infty} \frac{d_\ell(n)}{n} \;\middle|\; \ell \in \mathbb{N} \right\}$$

is attained when $\ell = \ell_0$.

n	k	$\min\{d(\ell) \mid \ell \in \mathbb{N}\}$	$\ell_0 \in \mathbb{N}$ minimizing $d(\ell)$
18	2	$5.\bar{6}$	5
90	10	36.2	4
180	20	73.75	3
$\to \infty$	$= \frac{1}{9}n$	$\to \infty$	$\to 3$

Table 3.2: Minimizing values ℓ_0 of $d(\ell)$ for $\kappa = \frac{1}{9}$

Proof: To begin with, we note that $\left\{\binom{i+1}{2}^{-1}\right\}_{i=1,2,\ldots}$ is a strictly decreasing sequence in $(0,1]$ and so the intervals $\left(\binom{\ell_0+2}{2}^{-1}, \binom{\ell_0+1}{2}^{-1}\right]$ in the statement form a partition of the positive unit interval.

Analyzing $\frac{d_\ell(n)}{n}$ as $n \to \infty$, we have

$$\frac{d_\ell(n)}{n} = \frac{n+1}{n(\ell+1)} + \frac{k-1}{2n} \cdot \ell = \frac{1}{\ell+1} + \frac{\ell\kappa}{2} + \frac{1}{n(\ell+1)} - \frac{\ell}{2n} \to \frac{1}{\ell+1} + \frac{\ell\kappa}{2}.$$

Let $f_\ell(\kappa) = \frac{1}{\ell+1} + \frac{\ell\kappa}{2}$ for $\ell \in \mathbb{N}$. Then $f_\ell(\kappa)$ is the asymptotic value of $\frac{d_\ell(n)}{n}$ for the fixed rate κ and so we must show that for any $\kappa \in \left(\binom{\ell_0+2}{2}^{-1}, \binom{\ell_0+1}{2}^{-1}\right]$, $f_{\ell_0}(\kappa)$ is minimal amongst all $f_\ell(\kappa)$. Now $\{f_\ell(\kappa)\}_{\ell=1,2,\ldots}$ is a sequence of functions on $(0,1]$ with increasing slopes and decreasing y-intercepts. To find which f_ℓ is minimal at each $\kappa \in (0,1]$ we start by comparing consecutive functions. For any $\ell \in \mathbb{N}_0$, the lines $f_\ell(\kappa) = \frac{1}{\ell+2} + \frac{\ell\kappa}{2}$ and $f_{\ell+1}(\kappa) = \frac{1}{\ell+2} + \frac{(\ell+1)\kappa}{2}$ intersect when

$$\begin{aligned}
\frac{1}{\ell+1} + \frac{\ell\kappa}{2} &= \frac{1}{\ell+2} + \frac{(\ell+1)\kappa}{2} \\
2(\ell+2) + \ell(\ell+1)(\ell+2)\kappa &= 2(\ell+1)(\ell+2)\kappa \\
\kappa &= \frac{2(\ell+1) - 2(\ell+2)}{\ell(\ell+1)(\ell+2) - (\ell+1)^2(\ell+2)} = \frac{2}{(\ell+1)(\ell+2)} \\
\kappa &= \binom{\ell+2}{2}^{-1}
\end{aligned}$$

at the point $X := \left(\binom{\ell+2}{2}^{-1}, \frac{2}{\ell+2}\right)$. Since $f_{\ell+1}$ has the greater slope, it is less than f_ℓ everywhere before X and more than f_ℓ after X. Inductively, we can see now that f_1 is minimal amongst $\{f_\ell\}$ on the interval $\left(\frac{1}{3}, 1\right]$ (actually, f_1 is minimal for all $\kappa \in \left(\frac{1}{3}, \infty\right)$, but rates beyond 1 are irrelevant to us) and that, in general, f_{ℓ_0} is minimal on the interval $\left(\binom{\ell_0+2}{2}^{-1}, \binom{\ell_0+1}{2}^{-1}\right]$ (see Figure 3.1). Thus, ℓ_0 minimizes $\frac{d_\ell(n)}{n}$ as $n \to \infty$ for any $\kappa \in \left(\binom{\ell_0+2}{2}^{-1}, \binom{\ell_0+1}{2}^{-1}\right]$. □

Theorem 3.4.2 *For any fixed information rate $\kappa = \frac{k}{n} \in (0,1]$, the integer*

$$\ell_0 = \left\lfloor -\frac{1}{2} + \frac{1}{2}\sqrt{1 + \frac{8}{\kappa}} \right\rfloor$$

maximizes the error rate $p = 1 - \frac{\lceil \min\{d_\ell(n) | \ell \in \mathbb{N}\} \rceil - 1}{n}$ asymptotically.

Proof: Recall from the proof of Lemma 3.4.1 that $f_\ell(\kappa)$ is just $\lim_{n\to\infty} \frac{d_\ell(n)}{n}$ for a fixed rate κ. The value of ℓ for which $f_\ell(\kappa)$ is minimal is therefore the same value that maximizes $1 - \frac{d_\ell(n)}{n}$ asymptotically. This value is the integer ℓ such that

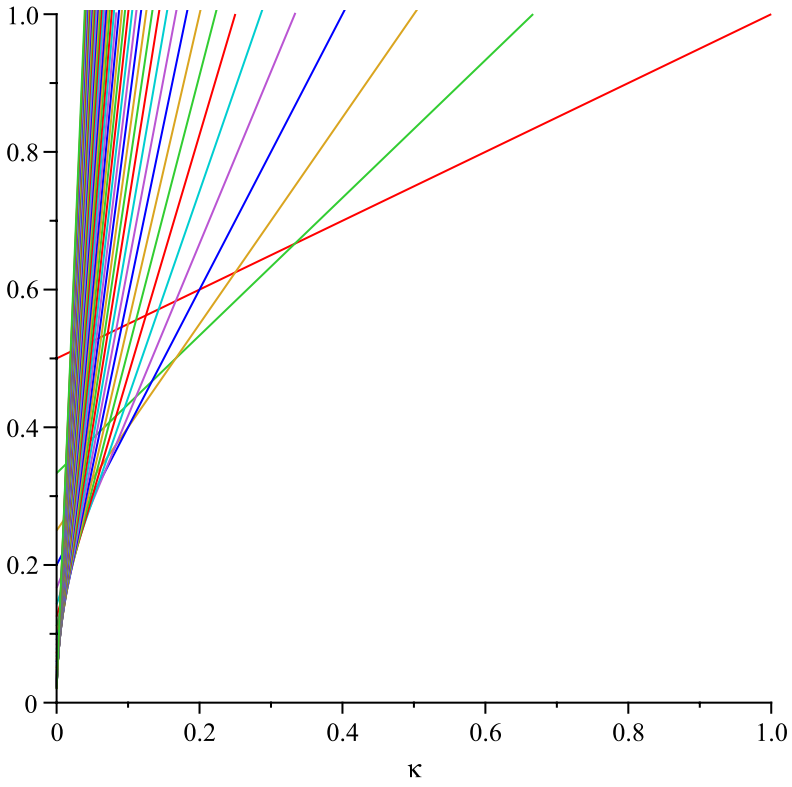

Figure 3.1: Sequence of lines $f_\ell(\kappa) = \frac{1}{\ell+1} + \frac{\ell}{2}\kappa$ for $\ell = 1, \ldots, 50$.

$\binom{\ell+2}{2}^{-1} < \kappa \leq \binom{\ell+1}{2}^{-1}$. To compute this ℓ from κ, we have that ℓ is the largest integer such that $\kappa \leq \frac{2}{\ell(\ell+1)}$, or equivalently

$$\ell^2 + \ell - \frac{2}{\kappa} \leq 0. \tag{3.7}$$

The roots of the quadratic expression on the left side of the inequality are $\ell = \frac{-1 \pm \sqrt{1+\frac{8}{\kappa}}}{2}$. Hence

$$\ell = \left\lfloor \frac{-1 + \sqrt{1 + \frac{8}{\kappa}}}{2} \right\rfloor$$

is the largest integer that satisfies inequality (3.7).

Now we know for all $\kappa \in (0, 1]$ which $\ell \in \mathbb{N}$ asymptotically minimizes $\frac{d(\ell)}{n}$ as $n \to \infty$. This same value, $\ell = \left\lfloor -\frac{1}{2} + \frac{1}{2}\sqrt{1 + \frac{8}{\kappa}} \right\rfloor$, therefore maximizes the error rate $p = 1 - \frac{\lceil \min\{d(\ell)\} \rceil - 1}{n}$ as $n \to \infty$ for a fixed information rate, κ. □

Corollary 3.4.3 *For a fixed information rate $\kappa = \frac{k}{n} \in (0, 1]$, we have*

$$\lim_{n\to\infty} \inf_{\ell \in \mathbb{N}} \left\{ \frac{d_\ell(n)}{n} \right\} = \inf_{\ell \in \mathbb{N}} \left\{ \lim_{n\to\infty} \frac{d_\ell(n)}{n} \right\}.$$

Proof: By Lemma 3.4.1,

$$\inf_{\ell\in\mathbb{N}}\left\{\lim_{n\to\infty}\frac{d_\ell(n)}{n}\right\}=\lim_{n\to\infty}\frac{d_{\ell_0}(n)}{n}=\frac{1}{\ell_0+1}+\frac{\ell_0\kappa}{2}$$

where $\ell_0\in\mathbb{N}$ is such that $\binom{\ell_0+2}{2}^{-1}<\kappa\leq\binom{\ell_0+1}{2}^{-1}$. If $\ell>\frac{4}{\kappa}$, then

$$\frac{d_\ell(n)}{n}=\frac{1}{\ell+1}+\frac{\ell\kappa}{2}+\frac{1}{n(\ell+1)}-\frac{\ell}{2n}>2-\frac{2}{\kappa n}\geq 1$$

if we assume $k\geq 2$, which we must. Observe, however, that

$$\frac{d_1(n)}{n}=\frac{1}{2}+\frac{\kappa}{2}+\frac{1}{2n}-\frac{1}{2n}\leq 1.$$

It follows that $\inf_{\ell\in\mathbb{N}}\left\{\frac{d_\ell(n)}{n}\right\}=\min_{\ell\in\mathbb{N}:\ell\leq 4\kappa^{-1}}\left\{\frac{d_\ell(n)}{n}\right\}$. Hence,

$$\lim_{n\to\infty}\inf_{\ell\in\mathbb{N}}\left\{\frac{d_\ell(n)}{n}\right\}=\lim_{n\to\infty}\min_{\ell\in\mathbb{N}:\ell\leq 4\kappa^{-1}}\left\{\frac{d_\ell(n)}{n}\right\}=\min_{\ell\in\mathbb{N}:\ell\leq 4\kappa^{-1}}\left\{\lim_{n\to\infty}\frac{d_\ell(n)}{n}\right\}$$

since each limit in the last expression is well-defined. Moreover, by Lemma 3.4.1 this last expression is equal to $\frac{1}{\ell_0+1}+\frac{\ell_0\kappa}{2}$ because $\ell_0<\frac{4}{\kappa}$ by Theorem 3.4.2, which shows the claim. □

In conclusion, we know that for list decoding $\mathrm{RS}(n,k,\boldsymbol{\alpha})$, the optimal choice for the y-degree bound ℓ of the interpolation polynomial Q in Algorithm 3.3.7 is whichever of $\left\lfloor\sqrt{\frac{2(n+1)}{k-1}}-1\right\rfloor$ and $\left\lceil\sqrt{\frac{2(n+1)}{k-1}}-1\right\rceil$ minimizes $d(\ell)=\frac{n+1}{\ell+1}+\frac{k-1}{2}\cdot\ell$. While there is a slight taste of unsavory ambiguity in this solution, at least we have narrowed down the best value of l to only two possibilities, which are easy to check. Moreover, we also know that for a fixed rate $\kappa=\frac{k}{n}$, as $n\to\infty$, the asymptotic optimal value of ℓ is $\left\lfloor-\frac{1}{2}+\frac{1}{2}\sqrt{1+\frac{8}{\kappa}}\right\rfloor$. Next we will examine closely the error rates the algorithm achieves and compare those to the error rates of classical decoding algorithms.

3.5 Improvement in Error Rate over Minimum Distance Decoding

Our main objective in the pursuit of list decoding is to handle decoding cases where the nearest codeword to the received word is equal to or more than half the distance of the code away. In these cases, there may or may not be a unique closest codeword so our algorithm might return multiple candidates from which we must choose one to decode the received word as. Therefore, we are not assured of decoding a word correctly even if few enough errors occurred so that the sent word appears in the list of closest codewords. However, we will still call such an error **correctable** because we can determine as much information as possible about the received word to make an informed decision in the decoding process. This is a much looser notion of error-correction than what it meant before list decoding, where a correctable error, $\boldsymbol{e}$, was

one for which a decoding algorithm always returned the correct codeword $\boldsymbol{c}$ if it was given the received word $\boldsymbol{c}+\boldsymbol{e}$. To distinguish between these ideas we shall say that the error $\boldsymbol{e}$ is **absolutely correctable** for a codeword $\boldsymbol{c}$ if $\boldsymbol{c}$ is the unique closest codeword to $\boldsymbol{c}+\boldsymbol{e}$ among those on the list returned by the algorithm given input $\boldsymbol{c}+\boldsymbol{e}$.

Note that whether an error is correctable or not depends on both the error and the codeword to which it is applied. When counting the total number errors that can be corrected, it is convenient to group errors according to weight and not worry about which coordinates are affected or which codeword is being corrupted. If an algorithm (absolutely) corrects all error vectors $\boldsymbol{e}$ with $\mathrm{wt}(\boldsymbol{e}) \leq \tau$ applied to any codeword, then we say it **(absolutely) corrects** up to τ errors or is **(absolutely) τ-error correcting**. Classical decoding algorithms absolutely correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor$. List decoding algorithms do that and more, though not all of the errors they correct will be absolutely correctable.

When comparing two decoding algorithms, one naturally looks for which algorithm corrects the most errors. This could be done case by case for varying code lengths and dimensions. However, when doing asymptotic analysis, we usually find that the error rate of an algorithm is determined just by the information rate of the code. So if we fix a rate $\kappa = \frac{k}{n}$ and let n go to infinity we get a well-defined value for the limit of the corresponding error rates of the algorithm. For example, the Berlekamp-Massey algorithm for decoding an $[n,k]$ Reed-Solomon code absolutely corrects up to $\left\lfloor \frac{n-k}{2} \right\rfloor = \left\lfloor n\frac{1-\kappa}{2} \right\rfloor$ errors. When we fix the rate and let $n \to \infty$ we see it has an asymptotic error rate of

$$\lim_{n\to\infty} \frac{\left\lfloor n\frac{1-\kappa}{2} \right\rfloor}{n} = \frac{1-\kappa}{2}.$$

By the Singleton bound, this is the best error rate one can achieve when considering how many errors are absolutely corrected. But counting non-absolutely corrected errors as well, Algorithm 3.3.7 asymptotically attains larger error rates than the Berlekamp-Massey algorithm for all information rates less than $\frac{1}{3}$. The proof of this fact uses the following simple lemma.

Lemma 3.5.1 *Let $f(x)$ be a function of x. Then $\lim_{x\to\infty} \left| \frac{\lceil f(x) \rceil}{x} - \frac{f(x)}{x} \right| = 0$. Therefore, if either limit exists,*

$$\lim_{x\to\infty} \frac{\lceil f(x) \rceil}{x} = \lim_{x\to\infty} \frac{f(x)}{x}.$$

Proof: Let $\epsilon > 0$. Then for any $x > \frac{1}{\epsilon}$,

$$\left| \frac{\lceil f(x) \rceil}{x} - \frac{f(x)}{x} \right| = \left| \frac{\lceil f(x) \rceil - f(x)}{x} \right| < \frac{1}{x} < \epsilon.$$

This shows the first claim. Since the limit of the difference of the two expressions is zero, the individual limits of the expressions must be equal if either exists. □

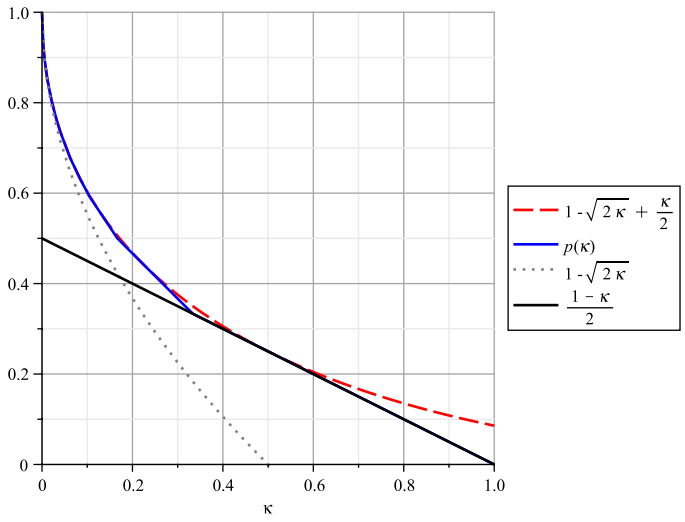

Figure 3.2: Error rate improvement of Sudan over Berlekamp-Massey

Proposition 3.5.2 *For any fixed information rate* $\kappa = \frac{k}{n} \in (0, 1]$*, the asymptotic error rate achieved by Algorithm 3.3.7 is*

$$p(\kappa) = 1 - \frac{1}{\ell_0 + 1} - \frac{\ell_0}{2}\kappa$$

where ℓ_0 *is as in Theorem 3.4.2.*

Proof: Theorem 3.4.2 tells us the y-degree bound ℓ_0 that maximizes $p(\kappa)$ as $n \to \infty$. To determine what value it is maximized to, recall that Algorithm 3.3.7 can correct up to $\tau = n - \lceil \min\{d_\ell(n) \mid \ell \in \mathbb{N}\} \rceil + 1$ errors. Hence, the asymptotic error rate it attains is

$$\begin{aligned}
\lim_{n\to\infty} \frac{n - \lceil \min\{d_\ell(n) \mid \ell \in \mathbb{N}\} \rceil + 1}{n} &= \lim_{n\to\infty} \frac{n - \lceil d_{\ell_0}(n) \rceil + 1}{n} \\
&= \lim_{n\to\infty} \frac{n - \left\lceil \frac{n+1}{\ell_0+1} + \frac{\kappa n - 1}{2} \cdot \ell_0 \right\rceil + 1}{n} \\
&= \lim_{n\to\infty} 1 - \left(\frac{1 + 1/n}{\ell_0 + 1} + \frac{\kappa - 1/n}{2} \cdot \ell_0 \right) + 1/n \\
&= 1 - \left(\frac{1}{\ell_0 + 1} + \frac{\ell_0}{2}\kappa \right).
\end{aligned}$$

Dropping the ceiling function after the second line is justified by Lemma 3.5.1 and the fact that $d_{\ell_0}(n)$ is a linear function in n and so $\lim_{n\to\infty} \frac{d_{\ell_0}(n)}{n}$ exists (we examined it before in Lemma 3.4.1). Thus we obtain our error rate $p(\kappa) = 1 - \frac{1}{\ell_0+1} - \frac{\ell_0}{2}\kappa$. □

It is evident from Proposition 3.3.3 that in our choice for the threshold τ of the number of errors we may allow and still be able to reconstruct the list $\mathcal{L}_{\tau,\boldsymbol{r}}$, we need

to make it small enough so that $n-\tau \geq \min\{d(\lfloor \ell_0 \rfloor), d(\lceil \ell_0 \rceil)\}$ where $\ell_0 = \sqrt{\frac{2(n+1)}{k-1}} - 1$ and $d(\ell) = \frac{n+1}{\ell+1} + \frac{k-1}{2} \cdot \ell$. Hence, we choose $\tau \leq n - \min\{d(\lfloor \ell_0 \rfloor), d(\lceil \ell_0 \rceil)\}$. Analyzing the value of $d(\ell)$ when its input is floored or ceilinged is complicated. Figure 3.2 depicts the asymptotic error rate $p(\kappa)$ attained by Sudan's algorithm by Proposition 3.5.2. A simple lower bound of $1 - \sqrt{2\kappa}$ is often stated to summarize this error rate, but as the dashed and dotted lines of the figure indicate, the expression $1 - \sqrt{2\kappa} + \frac{\kappa}{2}$ is actually a much closer bound (albeit, an upper bound) on $p(\kappa)$ and is a particularly good approximation of the error rate of Sudan's algorithm for information rates below $1/2$.

3.6 Modifying the Multiplicities of the Interpolation Polynomial

We saw in Figure 3.2 (based on Proposition 3.5.2) of the previous section that Sudan's Algorithm for decoding Reed-Solomon codes attains a higher asymptotic error rate than the Berlekamp-Massey algorithm for information rates less than one-third. However, for higher rates the algorithm performs no better than classical ones with respect to its asymptotic error rate. This is a somewhat unsatisfying result, especially considering that codes with higher information rates are typically more popular in practice. We know that the theoretical capacity remains much higher than what we have thus far achieved, which begs the question: how can we do better?

The idea of Sudan's algorithm is to look for a polynomial $Q(x, y)$ interpolating the points (α_i, r_i) and to compare it with factors of the form $y - f(x)$. If Q and $y - f(x)$ have enough points in common then the latter will divide the former. If we increase the number of points we have at which to compare these two polynomials then there will be more flexibility in which ones they can actually meet at and still necessarily have a common factor. This will potentially help us find more factors of the form $y - f(x)$ and thus correct more errors. Now, we do not want to constrain Q by forcing it to pass through arbitrary points that have no probabilistic connection to the factors corresponding to codewords near $r \in \mathbb{F}^n$. However, we can constrain Q further by specifying that it pass through each point (α_i, r_i) with some multiplicity

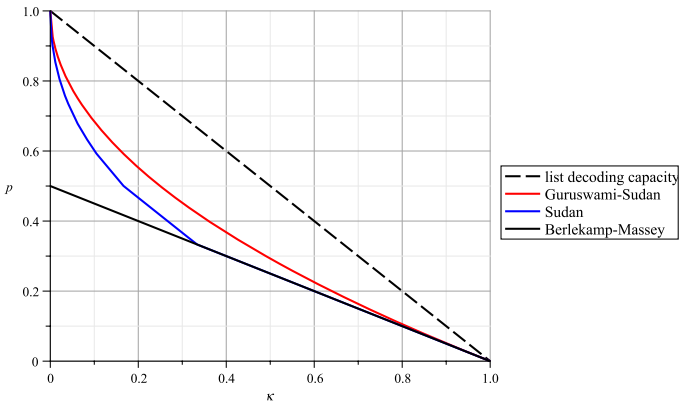

Figure 3.3: Error rates of RS decoding algorithms

$m_i \in \mathbb{N}$. This will increase the degree of Q and the complexity of the algorithm some but it will also increase the number of times $y - f(x)$ and Q intersect, causing them to have a common factor without $y - f(x)$ having to pass through as many of the interpolation points as it did when all of the multiplicities were implicitly 1. Guruswami and Sudan first proposed this modification in [19] and showed that it significantly improved the error rate when list decoding Reed-Solomon and other algebraic geometry codes.

3.6.1 Multiplicity of Bivariate Polynomials

There are different ways we can define the multiplicity of a polynomial at a point. If one is working over the real numbers, one typically begins in the realm of univariate polynomials by saying $c \in \mathbb{R}$ is a root of $f(x) = \sum_{i=0}^{t} f_i x^i \in \mathbb{R}[x]$ with multiplicity $m \in \mathbb{N}$ if one of the following equivalent conditions holds:

- $f(c) = 0$ and $\lim_{x \to c} \frac{f(x)}{(x-c)^m}$ exists and is nonzero,
- the derivatives of $f(x)$ satisfy the conditions $f^{(i)}(c) = 0$ for $0 \leq i < m$ and $f^{(m)}(c) \neq 0$,
- m is the largest power of $(x - c)$ that divides $f(x)$,
- the smallest nonzero term of the polynomial $f(x+c) = \sum_{i=0}^{t} \left(\sum_{j=i}^{t} f_j \binom{j}{i} c^{j-i} \right) x^i$ has degree m.

The last two are equally valid for defining multiplicity of roots of univariate polynomials over an arbitrary field $\mathbb{K}$, and the second condition is too as long as the derivative is understood to be the Hasse derivative. The first condition, however, does not extend so well because we have no clear notion of a limit in arbitrary fields.

We can extend these definitions to bivariate polynomials as follows. Let $T_<(m) = \{(s,t) \in \mathbb{N}_0^2 \mid s + t < m\}$ and $T_=(m) = \{(s,t) \in \mathbb{N}_0^2 \mid s + t = m\}$. We say that a point $(a,b) \in \mathbb{R}^2$ is a zero of $f(x,y) = \sum_{i,j} f_{i,j} x^i y^j \in \mathbb{R}[x,y] \backslash \{0\}$ (or f vanishes at (a,b)) with **multiplicity** $m > 0$ if one of the following conditions holds:

(a) m is the smallest number for which all partial derivatives of f of order less than m vanish at (a,b); that is, $f^{(s,t)}(a,b) = 0$ for all $(s,t) \in T_<(m)$ but not for all $(s,t) \in T_=(m)$,

(b) $f(a,b) = 0$ and f can be written as $f = \sum_{(s,t) \in T_=(m)} g_{s,t}$, where each $g_{s,t} \in \mathbb{R}[x,y]$ and for each $(s,t) \in T_=(m)$,

$$\lim_{(x,y) \to (a,b)} \frac{g_{s,t}(x,y)}{(x-a)^s (y-b)^t}$$

exists and at least one $g_{s,t}$ is nonzero.

(c) m is the largest integer such that f can be written as $f = \sum_{(s,t) \in T_=(m)} g_{s,t}$, $g_{s,t} \in \mathbb{R}[x,y]$, where $(x-a)^s (y-b)^t$ divides $g_{s,t}$ for each $(s,t) \in T_=(m)$,

(d) the smallest nonzero term of the shifted polynomial

$$f(x+a,y+b)=\sum_{s,t\geq 0}\left(\sum_{i\geq s,j\geq t} f_{i,j}\binom{i}{s}\binom{j}{t}a^{i-s}b^{j-t}\right)x^s y^t$$

has total degree m.

Lemma 3.6.1 *Conditions (a) through (d) above are equivalent.*

Proof: (a)⇒(b)
Assume $f^{(s,t)}(a,b)=0$ for all $(s,t)\in T_<(m)$ but $f^{(s',t')}(a,b)\neq 0$ for some $(s',t')\in T_=(m)$. Since $m>0$, $f(a,b)=f^{(0,0)}(a,b)=0$. Applying multivariate division with remainder (say with the order $<_{rlex}$), we can write f as

$$f=\sum_{(s,t)\in T_=(m)} q_{s,t}(x-a)^s(y-b)^t+r$$

where $q_{s,t},r\in\mathbb{R}[x,y]$ and no term of r is divisible by $\mathrm{lt}((x-a)^s(y-b)^t)=x^s y^t$. Let $g_{s,t}=q_{s,t}(x-a)^s(y-b)^t$ for each $(s,t)\in T_=(m)$. Now fix $(s,t)\in T_=(m)$. For any $(u,v)\in T_<(m)$, after repeated applications of the product rule for differentiation, we obtain

$$g_{s,t}^{(u,v)}=\sum_{\substack{0\leq i\leq u\\ 0\leq j\leq v}}\binom{u}{i}\binom{v}{j}q_{s,t}^{(i,j)}(x-a)^{s-i}(y-b)^{t-j}.$$

Then, since either $u<s$ or $v<t$ (or both) it is clear that $g_{s,t}^{(i,j)}(a,b)=0$. This holds for every $(s,t)\in T_=(m)$ and $(u,v)\in T_<(m)$. Therefore, since $r=f-\sum g_{s,t}$ it follows that $r^{(u,v)}=f^{(u,v)}-\sum g_{s,t}^{(u,v)}=0$ for all $(u,v)\in T_<(m)$. But since r has no term of total degree greater than or equal to m, r must be 0 (otherwise, for example, if $\mathrm{lt}(r)=cx^u y^v$ where $c\in\mathbb{R}\backslash\{0\}$ and $(u,v)\in T_<(m)$, then we would have $r^{(u,v)}=c\Rightarrow r^{(u,v)}(a,b)=c\neq 0$). Thus, $f=\sum_{(s,t)\in T_=(m)} g_{s,t}$. Since $f^{(s',t')}(a,b)\neq 0$, we must have

$$\begin{aligned}\sum_{(s,t)\in T_=(m)} g_{s,t}^{(s',t')}(a,b) &= \sum_{(s,t)\in T_=(m)}\left(\sum_{\substack{0\leq i\leq s'\\ 0\leq j\leq t'}}\binom{s'}{i}\binom{t'}{j}q_{s,t}^{(i,j)}(x-a)^{s-i}(y-b)^{t-j}\right)\Bigg|_{(a,b)}\\ &= q_{s',t'}(a,b)\\ &\neq 0.\end{aligned}$$

It follows that

$$\lim_{(x,y)\to(a,b)}\frac{g_{s',t'}(x,y)}{(x-a)^{s'}(y-b)^{t'}}=\lim_{(x,y)\to(a,b)} q_{s',t'}(x,y)=q_{s',t'}(a,b)\neq 0,$$

which shows that condition (b) holds.

(b)⇒(c)
Assume $f(a,b) = 0$ and $f = \sum_{(s,t)\in T_=(m)} g_{s,t}$ for some $g_{s,t} \in \mathbb{R}[x,y]$ and

$$\lim_{(x,y)\to(a,b)} \frac{g_{s,t}(x,y)}{(x-a)^s(y-b)^t}$$

exists for all $(s,t) \in T_=(m)$ and it is nonzero for some $(s',t') \in T_=(m)$. By Corollary 3.6.3, which follows this lemma[2], $g_{s,t}$ is divisible by $(x-a)^s(y-b)^t$ for each $(s,t) \in T_=(m)$. Thus, f is written in the form dictated by condition (c). To show that m is maximal for writing f this way, observe that $g_{s,t} = (x-a)^s(y-b)^t q_{s,t}$ for some $q_{s,t} \in \mathbb{R}[x,y]$. Additionally, we know that $q_{s',t'}(a,b) = \lim_{(x,y)\to(a,b)} \frac{g_{s',t'}}{(x-a)^{s'}(y-b)^{t'}}$ is not zero. Hence, the constant term of $q_{s',t'}(x+a,y+b)$ is nonzero, so $f(x+a,y+b) = \sum_{(s,t)\in T_=(m)} x^s y^t q_{s,t}(x+a,y+b)$ has a nonzero term with variable part $x^{s'}y^{t'}$ and total degree m. Now, suppose $f = \sum_{(s,t)\in T_=(m')} h_{s,t}$ for some $m' > m$ where each $h_{s,t} \in \mathbb{R}[x,y]$ is divisible by $(x-a)^s(y-b)^t$. Say $h_{s,t} = (x-a)^s(y-b)^t r_{s,t}$, where $r_{s,t} \in \mathbb{R}[x,y]$. Then every term of $f(x+a,y+b) = \sum_{(s,t)\in T_=(m')} x^s y^t r_{s,t}(x,y)$ has total degree at least $m' > m$, a contradiction. Thus, m is the largest integer for which f can be expressed in the desired form.

(c)⇒(d)
Suppose m is the largest integer such that f can be written as $f = \sum_{(s,t)\in T_=(m)} g_{s,t}$, $g_{s,t} \in \mathbb{R}[x,y]$, where $(x-a)^s(y-b)^t$ divides $g_{s,t}$ for each $(s,t) \in T_=(m)$. Since $(x-a)^s(y-b)^t$ divides $g_{s,t}$, shifting x to $x+a$ and y to $y+b$ provides $x^s y^t$ divides $g_{s,t}(x+a,y+b)$ for each $(s,t) \in T_=(m)$. Therefore, each term of $f(x+a,y+b) = \sum g_{s,t}(x+a,y+b)$ is divisible by $x^s y^t$ for some $(s,t) \in T_=(m)$, so its smallest nonzero term has degree at least m. If the degree d of $f(x+a,y+b)$'s smallest term is greater than m, then every term in f is divisible by $(x-a)^s(y-b)^t$ for some $(s,t) \in T_=(d)$ and therefore f can be written $f = \sum_{(s,t)\in T_=(d)} g_{s,t}$, which contradicts the supposition. Thus, the smallest degree of any term in f is m.

(d)⇒(a)
Suppose the smallest degree of a nonzero term of $f(x+a,y+b)$ is m. Then $f(x+a,y+b) = \sum_{(s,t)\in T_\geq(m)} c_{s,t} x^s y^t$ for some $c_{s,t} \in \mathbb{R}$. For any $(i,j) \in T_<(m)$, $f^{(i,j)}(x+a,y+b) = \sum_{(s,t)\in T_\geq(m)} (s)_i (t)_j c_{s,t} x^s y^t$ where $(s)_i = \prod_{k=0}^{i-1}(s-i)$ and $(t)_j$ is likewise a falling factorial. Notice that $(s)_i = 0$ if $i > s$ and $(t)_j = 0$ if $j > t$. Since either $i < s$ or $j < t$, it follows that every nonzero term of $f^{(i,j)}(x+a,y+b)$ has positive total degree and so $f^{(i,j)}(a,b) = f^{(i,j)}(0+a,0+b) = 0$. Thus every partial derivative of f of order less than m vanishes at (a,b). On the other hand, there is some $c_{s',t'} \neq 0$, so $f^{(s',t')}(a,b) = f^{(s',t')}(0+a,0+b) = s!t!c_{s,t} \neq 0$. Hence, m is the smallest integer for which all lesser order partial derivatives of f vanish at (a,b). □

[2]Corollary 3.6.3 follows from Lemma 3.6.2, but neither of them is based upon Lemma 3.6.1, so we are not committing a circular reasoning error.

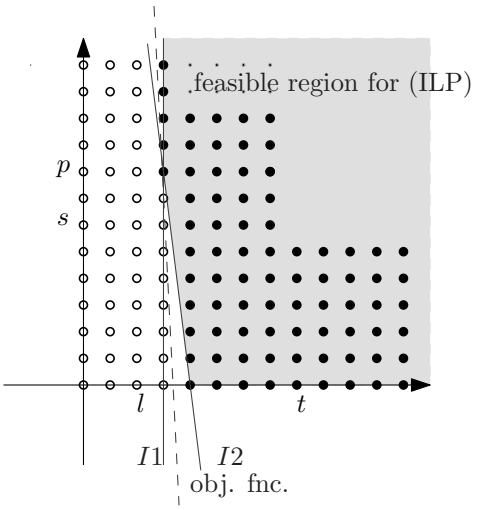

Figure 3.4: Feasible region and objective function for ILP

Lemma 3.6.2 *Let* $f(x,y) = \sum_{i,j\geq 0} f_{i,j}x^iy^j \in \mathbb{R}[x,y]$. *If* $\lim_{(x,y)\to(0,0)} \frac{f}{x^sy^t} = L \in \mathbb{R}$ *for some* $(s,t) \in T_=(m)$ $(m \geq 1)$, *then* $f_{i,j} = 0$ *whenever* $i < s$ *or* $j < t$.

Proof: We proceed by double induction on i and j with $i < t$ and j unbounded.

Since $\lim_{(x,y)\to(0,0)} \frac{f}{x^sy^t}$ exists and is real, $\lim_{(x,y)\to(0,0)} f = f(0,0) = f_{0,0}$ must be zero, otherwise the previous limit would blow up and could not be finite.

Suppose for some $\ell < t$ and p, $f_{i,j} = 0$ whenever $i < \ell$ or when $i = \ell$ and $j < p$. We need to show $f_{\ell,p} = 0$.

Letting (x,y) approach $(0,0)$ along the line $x = z^a$, $y = z^b$, where a and b are positive constants and z is a real variable, we obtain a univariate limit

$$\lim_{\substack{x=z^a,y=z^b\\ z\to 0}} \frac{f(z^a,z^b)}{z^{as+bt}} = \lim_{z\to 0} \frac{\sum\limits_{\substack{i,j:i>\ell \text{ or}\\ i=\ell \text{ and } j\geq p}} f_{i,j}z^{ai+bj}}{z^{as+bt}} = L.$$

Now we will try to choose a and b so as to minimize $a\ell + bp$ and thereby isolate the coefficient $f_{\ell,p}$. We want (ℓ,p) to be the solution to the following integer linear program.

$$\begin{array}{rll} \text{minimize} & ax_1 + bx_2 & \text{(ILP)} \\ \text{subject to} & x_1 \geq \ell & (I1) \\ & px_1 + x_2 \geq p(\ell+1) & (I2) \\ & x_1, x_2 \in \mathbb{N}_0 & \end{array}$$

In order for (ℓ,p) to be the unique optimal solution the objective function must lie between $(I1)$ and $(I2)$. Hence, a and b must satisfy $-\infty < -\frac{a}{b} < -p$. We may therefore choose $b = 1$ and $a = 2p$. Then $a\ell+bp = 2p\ell+p$ is less than $ai+bj = 2pi+j$ whenever $i > \ell$ or $i = \ell$ and $j > p$.

Now, multiplying the function in the limit above by $z^{as+bt-a\ell-bp}$ we have

$$\begin{aligned}
0 &= (0)(L) \\
&= \left(\lim_{z\to 0} z^{as+bt-a\ell-bp}\right)\left(\lim_{z\to 0}\frac{f(z^a,z^b)}{z^{as+bt}}\right) \\
&= \lim_{z\to 0}\frac{f(z^a,z^b)}{z^{a\ell-bp)}} \\
&= \lim_{z\to 0}\sum_{\substack{i,j:i>\ell \text{ or} \\ i=\ell \text{ and } j\geq p}} f_{i,j}z^{ai+bj-(a\ell+bp))} \\
&= \lim_{z\to 0}\left(f_{\ell,p}+\sum_{\substack{i,j:i>\ell \text{ or} \\ i=\ell \text{ and } j>p}} f_{i,j}z^{ai+bj-(a\ell+bp))}\right) \\
&= f_{\ell,p}.
\end{aligned} \tag{3.8}$$

Thus, $f_{\ell,p}=0$ and by induction on j we get $f_{\ell,j}=0$ for all $j\geq 0$.

Now we show that $f_{\ell+1,0}=0$ if $\ell+1<t$. Letting $a=b=1$ in the directional limit we used above we obtain

$$\begin{aligned}
0 &= (0)(L) \\
&= \left(\lim_{z\to 0} z^{s+t-(\ell+1)}\right)\left(\lim_{z\to 0}\frac{f(z,z)}{z^{s+t}}\right) \\
&= \lim_{z\to 0}\frac{f(z,z)}{z^{\ell+1)}} \\
&= \lim_{z\to 0}\sum_{i\geq \ell+1, j\geq 0} f_{i,j}z^{i+j-(\ell+1))} \\
&= \lim_{z\to 0}\left(f_{\ell+1,0}+\sum_{\substack{i,j:i\geq \ell+1 \text{ and } j\geq 0 \\ \text{or } i=\ell+1 \text{ and } j>0}} f_{i,j}z^{i+j-(\ell+1))}\right) \\
&= f_{\ell,p}.
\end{aligned} \tag{3.9}$$

Now, by double induction on i and j we see that $f_{i,j}=0$ for all $i<t$. By symmetry, we also get that $f_{i,j}=0$ whenever $j<s$, which concludes the proof. □

Corollary 3.6.3 *If* $\lim_{(x,y)\to(a,b)}\frac{f}{(x-a)^s(y-b)^t}=L\in\mathbb{R}$ *for some* $f\in\mathbb{R}[x,y]$ *then* f *is divisible by* $(x-a)^s(y-b)^t$.

Proof: Let $F(x,y)=f(x+a,y+b)=\sum_{i,j\geq 0}F_{i,j}x^iy^j\in\mathbb{R}[x,y]$. Then

$$\lim_{(x,y)\to(0,0)}\frac{F}{x^sy^t}=\lim_{(x,y)\to(a,b)}\frac{f}{(x-a)^s(y-b)^t}=L.$$

Therefore, by Lemma 3.6.2 $F_{i,j} = 0$ whenever $i < s$ or $j < t$ so every term of F has x-degree greater than or equal to s and y-degree greater than or equal to t. Thus, since $F(x, y) = f(x+a, y+b)$ is divisible by $x^s y^t$, $f(x, y)$ is divisible by $(x-a)^s(y-b)^t$. □

Conditions (c) and (d) can be easily adapted to define the multiplicity at a point of a bivariate polynomial over a finite field. However, conditions (a) and (b) are not so adaptable because the partial derivatives of polynomials over finite fields are not as well-behaved as those over fields of characteristic 0 and limits simply have no adequate conception in the finite field setting. However, one can adapt condition (a) to properly define multiplicity if Hasse derivatives are used in lieu of formal partial derivatives. The $(s,t)^{\text{th}}$ Hasse derivative of the bivariate polynomial $f(x, y) = \sum f_{i,j} x^i y^j$ is defined as

$$f^{[s,t]}(x, y) = \sum_{i \geq s, j \geq t} \binom{i}{s} \binom{j}{t} f_{i,j} x^{i-s} y^{j-t}.$$

Roth took such an approach in [32]. Note that the evaluation $f^{[s,t]}(x, y)$ at the point (a, b) is identical to the coefficient of $x^s y^t$ in the shifted polynomial $f(x + a, y + b)$, which yields what seems to be a preferable definition for multiplicity among list decoding researchers. To name a couple, Guruswami and Sudan in [19] as well as Lee and O'Sullivan in [24] used a change of coordinates to define multiplicity. They adapted condition (d) by defining the multiplicity of $f(x, y) \neq 0$ at the origin to be the smallest m such that f has a nonzero term of total degree m. Then the multiplicity of f at (a, b) is defined as the multiplicity of $f_{(a,b)} = f(x + a, y + b)$ at the origin. We will take an equivalent approach.

Definition 3.6.4 *For a nonzero polynomial $f \in \mathbb{F}[x, y]$, the **multiplicity** of f at the point $(a, b) \in \mathbb{F}^2$ is the smallest total degree m of any nonzero term of the shifted polynomial $f(x + a, y + b)$.*

3.7 The Guruswami-Sudan Algorithm

Next we will examine the consequences of strengthening the interpolation constraints in Sudan's algorithm by requiring the interpolation polynomial to vanish with some multiplicity at each point. We will see that although doing so increases the complexity of the algorithm, it also increases the error rate that we can achieve. The results of this section were first presented by Guruswami and Sudan in [19] and have since been studied extensively by many people.

Algorithm 3.7.1 (Guruswami-Sudan List Decoding for $\mathrm{RS}(n, k, \boldsymbol{\alpha})$**)**
Input: received word $\boldsymbol{r} = (r_1, \ldots, r_n)$, multiplicity parameter m, error threshold $\tau < n - \sqrt{kn}$
Output: list of codewords, $\mathcal{L}_{\tau,\boldsymbol{r}}$, within Hamming distance τ of $\boldsymbol{r}$

G1. Compute $m = 1 + \left\lfloor \dfrac{kn + \sqrt{k^2n^2 + 4((n-\tau)^2 - kn)}}{2((n-\tau)^2 - kn)} \right\rfloor$ and $d = m(n-\tau) - 1$.

G2. Find a nonzero polynomial $Q \in \mathbb{F}[x, y]$ satisfying:

a) $Q(\alpha_i, r_i) = 0$ with multiplicity m for $i = 1, \ldots, n$,

b) $\deg_{k-1}(Q) \leq d$.

G3. Find all polynomials $f \in \mathbb{F}[x]_k$ such that $Q(x, f(x))$ is uniformly 0.

G4. For each polynomial f found in G3, if $\mathrm{dist}(\mathrm{ev}_{\boldsymbol{\alpha}}(f), \boldsymbol{r}) \leq \tau$ then output the codeword $\mathrm{ev}_{\boldsymbol{\alpha}}(f) = (f(\alpha_1), \ldots, f(\alpha_n))$.

3.7.1 Correctness

We assume henceforth that $\tau < n - \sqrt{kn}$. The parameters m and d in step G1 are selected so that the two inequalities

$$n\binom{m+1}{2} < \frac{d(d+2)}{2k} \tag{3.10}$$

and

$$m(n-\tau) > d \tag{3.11}$$

will both be satisfied. The reason this is desirable will become more apparent during the course of proving the algorithm's correctness (which will consist of several lemmas and propositions similar to those in Section 3.3), but to summarize the reasons: the first inequality assures us that an interpolation polynomial Q such as what we seek in Algorithm 3.7.1 exists and the second inequality forces the polynomials corresponding to close codewords to $\boldsymbol{r}$ to be y-roots of Q. From the assignments of m and d in step G1, it is clear that (3.11) holds. To see that the other does as well, we use that fact that $d = m(n-\tau) - 1$ to simplify the right-hand side of (3.10) to $\frac{m^2(n-\tau)^2-1}{2k}$. From there the inequality reduces to a quadratic inequality (in m)

$$m^2((n-\tau)^2 - kn) - knm - 1 > 0,$$

which will be positive for all values of m greater than its largest root, which is

$$\gamma = \frac{kn + \sqrt{k^2n^2 + 4((n-\tau)^2 - kn)}}{2((n-\tau)^2 - kn)}$$

since $(n-\tau)^2 > kn$ by the assumption on τ. Then, because we want m to be an integer greater than γ, we set it to $\lfloor \gamma \rfloor + 1$ and thus both inequalities are satisfied.

We now show how the two inequalities (3.10) and (3.11) are useful to establish the correctness of the algorithm. Here we retain the notation $P_i = (\alpha_i, r_i)$ for $1 \leq i \leq n$.

Proposition 3.7.2 *Suppose $Q \in \mathbb{F}[x, y]$ passes through each of the points $P_1, \ldots, P_n$ with multiplicity m and $\deg_{k-1}(Q) \leq d = m(n-\tau) - 1$. If $f \in \mathbb{F}[x]_k$ is a polynomial interpolating at least $n - \tau$ of the points P_i, then f is a y-root of Q.*

Proof: Consider the polynomial $Q(x, f(x)) \in \mathbb{F}[x]$. The degree constraint for Q means that every term $ax^i y^j$ of Q satisfies $i + (k-1)j < m(n-\tau)$. Therefore, when we substitute $f(x)$ for y in each term $ax^i y^j$ of Q, the resulting polynomial $ax^i(f(x))^j$ has degree at most $i+(k-1)j < m(n-\tau)$ since $\deg(f) \leq k-1$. So $\deg Q(x, f(x)) \leq d$.

Now, $Q(\alpha_i, r_i) = 0$ for $i = 1, \ldots, n$ with multiplicity m. Since $f(\alpha_i) = r_i$ for at least $n - \tau$ points α_i we have at these points that $Q(\alpha_i, f(\alpha_i)) = Q(\alpha_i, r_i) = 0$ with multiplicity m (note that since $Q(x + \alpha_i, y + r_i)$ has no term with total degree less than m, $Q(x+\alpha_i, f(x+\alpha_i)+r_i)$ also has no term of degree less than m and so it has multiplicity m at α_i in the univariate sense). So $Q(x, f(x))$ is a polynomial of degree less than $m(n-\tau)$ with at least $m(n-\tau)$ roots (counting multiplicity). Therefore, $Q(x, f(x))$ is uniformly 0 so f is a y-root of Q. $\square$

Proposition 3.7.3 ([19]) *Let d and m be defined as in step G1. If $n\binom{m+1}{2} < \frac{d(d+2)}{2k}$ then there exists a nonzero polynomial $Q \in \mathbb{F}[x, y]$ that vanishes at each point $P_1, \ldots, P_n$ with multiplicity m and satisfies $\deg_{k-1}(Q) \leq d$.*

Proof: If we consider the Hasse derivative definition of multiplicity for a moment, then the condition that each P_i is a root of Q with multiplicity m means that $Q^{[s,t]}(\alpha_i, r_i) = 0$ for all $(s,t) \in T_<(m) = \{(s,t) \mid s,t \in \mathbb{N}_0 \text{ and } s + t < m\}$ for $i = 1, \ldots, n$. This gives us a homogeneous system of $n\binom{m+1}{2} = n|T_<(m)|$ equations in the variables corresponding to the coefficients of Q, a nontrivial solution to which is assured provided the number of equations is less than the number of unknowns, which we now count. The number of unknown coefficients in Q, whose $(1, k-1)$-weighted degree is strictly bounded above by $d = m(n-\tau)$, is

$$\begin{aligned}
\sum_{t=0}^{\lfloor d/k \rfloor} \sum_{s=0}^{d-kt} 1 &= \sum_{t=0}^{\lfloor d/k \rfloor} (d - kt + 1) \\
&= (d+1)\left(\left\lfloor \frac{d}{k} \right\rfloor + 1\right) - \frac{k}{2} \cdot \left(\left\lfloor \frac{d}{k} \right\rfloor\right)\left(\left\lfloor \frac{d}{k} \right\rfloor + 1\right) \\
&\geq \left(d + 1 - \frac{d}{2}\right)\left(\left\lfloor \frac{d}{k} \right\rfloor + 1\right) \\
&\geq \left(\frac{d+2}{2}\right)\left(\frac{d}{k}\right).
\end{aligned}$$

Therefore, from the proposition's assumption, a nonzero polynomial Q interpolating the n points with the multiplicity m exists. $\square$

Theorem 3.7.4 ([19]) *Assuming $\tau < n - \sqrt{kn}$, Algorithm 3.7.1 correctly outputs all of the codewords in $\mathrm{RS}(n, k, \boldsymbol{\alpha})$ that are up to Hamming distance τ away from $\boldsymbol{r}$.*

Proof: The theorem follows from Propositions 3.7.2 and 3.7.3 and the fact that the assignments to m and d in step G1 satisfy the inequalities (3.10) and (3.11). $\square$

The complexity of Algorithm 3.7.1 can be seen to be polynomial in n if we look at it in terms of the number of field operations required during execution. Step G1 requires no field arithmetic. Step G2 can be reduced to solving a system of homogeneous linear equations in at most $\frac{d(d+2)}{2k}$ variables, which can be solved by using Gaussian elimination with $\mathcal{O}(d^6/k^3)$ field operations. Step G3 has us find the y-roots of Q. To get a complexity bound on this step we need to consider the bound on $\deg_y(Q)$ - the possible number of y-roots of Q. Since $\deg_{k-1}(Q) \leq d$, we have $\ell = \deg_y(Q) \leq \left\lfloor \frac{d}{k-1} \right\rfloor$. Thus, by Corollary 2.3.8, step G3 can be done using

$$\mathcal{O}(\ell \log^2 \ell \log\log \ell(k \log q + d\ell)) = \mathcal{O}(d \log q + \frac{d^3}{k^2}(\log^2(d/k) \log\log(d/k))).$$

Lastly, step G4 can be performed efficiently with fast multipoint evaluation ([38, Chapter 10]) using $\mathcal{O}(n^2)$ operations in $\mathbb{F}_q$.

From the choice m and d made in step G1, we have $m \in \mathcal{O}(kn)$ and $d < mn \in \mathcal{O}(n^2 k)$, so Algorithm 3.7.1 can be done using $\mathcal{O}(n^{12}k^3)$. This is bound is excessively high but it is sufficient to verify that the algorithm correctly outputs a list of close codewords to $\boldsymbol{r}$ in a time that is polynomial in the code length n.

The best known bound for solving the radius-τ list decoding problem for $\tau < n - \sqrt{kn}$ is $\mathcal{O}(n^2 m^4)$. This bound is achieved by Gaborit and Ruatta in [13]. We will hold off on a more thorough complexity analysis until Chapter 4, when we do a careful analysis of a more general algorithm that reduces to the Guruswami-Sudan algorithm as a special case.

Remark 3.7.5 *The Guruswami-Sudan list decoding algorithm is presented as a decoding algorithm for Reed-Solomon codes in the ungeneralized sense, but it is easily adapted to list decoding generalized Reed-Solomon codes by rescaling a couple of the computations. Specifically, it works for* $\mathrm{GRS}_q(n, k, \boldsymbol{\alpha}, \boldsymbol{v})$ *just by changing the interpolation points from* $P_i = (\alpha_i, r_i)$ *to* $P_i = (\alpha_i, r_i/v_i)$ *and keeping in mind that the polynomials in* x *obtained in step G4 should be multiplied (compentwise) by* $\boldsymbol{v}$ *in step G4 when evaluating the polynomials to produce the codewords. From here, the algorithm immediately becomes serviceable as a list decoding algorithm for the alternant code* $\mathcal{C} = \mathcal{A}_{q^t}(n, \delta, \boldsymbol{\alpha}, \boldsymbol{v})$*. To operate it as such, when a received word is given one decodes it as if it is in the parent GRS code* $\mathrm{GRS}_{q^t}(n, n - \delta + 1, \boldsymbol{\alpha}, \boldsymbol{v})$ *and then restricts the list it returns to those words whose components lie exclusively in the base field* $\mathbb{F}$*. The algorithm still corrects up to* $\tau < n - \sqrt{kn}$ *errors and the time complexity is the same as before. The catch is that the complexity measures operations in the extension field* $\mathbb{F}_{q^t}$ *and not the base field* $\mathbb{F}_q$ *where the alternant code actually lives. An arithmetic operation in* $\mathbb{F}_{q^t}$ *can be carried out using* $\mathcal{O}(t \log t \log\log t)$ *operations in* $\mathbb{F}_q$ *via Schönhage and Strassen's multiplication algorithm [38, section 8.3]. This means the complexity of the Guruswami-Sudan algorithm adapted for alternant codes will be* $\mathcal{O}(n^2 m^4 \cdot t \log t \log\log t)$ *operations in* $\mathbb{F}_q$ *(using Gaborit and Ruatta's bound) where* t *is the degree of the field extension used to get the defining GRS code.*

Chapter 4
List-Decoding via Multivariate Interpolation and Gröbner Bases

4.1 Introduction

From around 1999 through 2005, the asymptotic error rate of $1 - \sqrt{\kappa}$ stood as the benchmark in list decoding performance for Reed-Solomon codes. However, as we saw in Section 3.2, the upper bound on efficiently list decoding up to radius p is $p = 1 - \kappa$, for beyond that error rate all codes would require an exponential amount of time to simply output the average-case list, let alone to find it. It could be possible, however, to efficiently list decode some codes at an error rate arbitrarily close to $1 - \kappa$, for we know that codes of length n (over large alphabets) which are $(1 - \kappa - \delta, 1/\delta)$-list decodable exist. However, we have not yet seen what these codes are, nor have we seen how (or even whether) it is possible to list decode them in polynomial time. A major breakthrough came in 2005 when Parvaresh and Vardy ([30]) made two essential changes to the foundation of the Guruswami-Sudan Algorithm. First, they branched off from the idea of decoding Reed-Solomon codes (for which numerous attempts to break the $1 - \mathcal{O}(\sqrt{\kappa})$-radius decoding barrier had failed) to a more general form of evaluation codes. Second, they extended the interpolation problem to a multivariate one. It has long been understood that decoding Reed-Solomon codes amounts to interpolating a univariate polynomial of small degree through as many of n given points as possible with the hope that it will match the generating polynomial $f(x)$ for the sent codeword (interpolating all n of the points with a polynomial of small degree, in general, will not be possible due to noise on the channel). Sudan's remarkable contribution (Section 3.3) to list decoding was to recognize that this matching could be extended to a bivariate interpolation problem wherein good candidates for the solution to the univariate problem will correspond to irreducible factors of the bivariate interpolation polynomial. Parvaresh and Vardy's clever idea was to generate related polynomials $g_1(x), \ldots, g_{w-1}(x)$ from $f(x)$ and send a codeword corresponding to the evaluation of all of these polynomials, instead of just f (a protraction of the Reed-Solomon code). The decoder is thereby provided extra information about the sent word by the way that the correlation of the g_i's to f is judiciously ingrained into the encoder. Their method yielded a new class of codes and an efficient algorithm for list decoding those codes with an error rate of $1 - \mathcal{O}(\kappa \log \frac{1}{\kappa})$, which yielded an asymptotic improvement over the $1 - \sqrt{\kappa}$ bound of the Guruswami-Sudan algorithm for all rates $\kappa < \frac{1}{16}$. Subsequently, Guruswami and Rudra ([18]) were able to adapt the scheme of Parvaresh and Vardy to achieve error rates near the list decoding channel capacity for all information rates by strengthening the algebraic correlation of the g_i polynomials to f in such a way that the codes of Parvaresh and Vardy could be sent with the same amount of information in basically a compressed form. The codes that Guruswami and Rudra utilized are called *folded* (or *interleaved*) Reed-Solomon codes, and we present an overview of them and a multivariate list decoding algorithm for them in this chapter.

$(f(\alpha^1), \ldots, f(\alpha^v))$	$(f(\alpha^{v+1}), \ldots, f(\alpha^{2v}))$	$\cdots$	$(f(\alpha^{n-v+1}), \ldots, f(\alpha^n))$

Table 4.1: 1-1 correspondence between codewords in the FRS code and the RS code

4.2 Folded Reed-Solomon Codes

The term "folded" Reed-Solomon code was first used in [23], where they were utilized in the pursuit of correcting *phase bursts*, which are error patterns in which the error positions are not random or uniform but instead come consecutively in periodic bursts. These codes are obtained directly from Reed-Solomon codes by simply batching together groups of symbols to obtain a code with the same rate but a fraction of the length. Essentially nothing is changed but the size of the alphabet and the length of the code; we transition from a small alphabet to a large one and from a long code length to a short one.

For simplicity, we will assume in this definition (and henceforth) that the folding factor v divides the underlying code length n. Also, we will need our underlying Reed-Solomon code to be a primitive code so we will assume the code evaluators are $\alpha, \ldots, \alpha^n$ for some primitive element $\alpha \in \mathbb{F}_q$, so we let $\boldsymbol{\alpha} = (\alpha, \ldots, \alpha^n)$ throughout this chapter.

Definition 4.2.1 *The v-**folded Reed-Solomon code** of length $N = n/v$ and dimension $K = k/v$ over $\mathbb{F}_q^v$, denoted $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$, is the set*

$$\{([f(\alpha), \ldots, f(\alpha^v)], [f(\alpha^{v+1}), \ldots, f(\alpha^{2v})], \ldots, [f(\alpha^{n-v+1}), \ldots, f(\alpha^n)]) \mid f \in \mathbb{F}_q[x]_k\}.$$

The v-folded Reed-Solomon code $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$ is almost identical to the ordinary Reed Solomon code $\mathrm{RS}_q(n, k, \boldsymbol{\alpha})$, the only difference being our interpretation of a symbol as a v-tuple over $\mathbb{F}_q$ rather than an element of $\mathbb{F}_q$. This decreases the length and dimension by a factor of v. The distance D of the folded code is at least $\lceil d/v \rceil$, where $d = n - k + 1$ is the distance of the original Reed-Solomon code. The rate κ of the folded code is the same as that of the original and the relative distance δ is greater than $1 - \kappa$. To encode a message polynomial $f \in \mathbb{F}[x]_k$, we just evaluate it at the coordinates of $\boldsymbol{\alpha}$ as we would for $\mathrm{RS}_q(n, k, \boldsymbol{\alpha})$, but we take the codeword symbols to be blocks of v consecutive evaluations of f rather than each individual one. The key difference to bear in mind here is that the j^{th} symbol of the codeword generated by f in $\mathrm{RS}_q(n, k, \boldsymbol{\alpha})$ is $f(\alpha^j)$, whereas in $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$, the j^{th} symbol is $[f(\alpha^{(j-1)v+1}), \ldots, f(\alpha^{jv})]$.

The main benefit of folding codes is that it reduces the number of error patterns that we have to correct in order to achieve a given error rate. For example, if we receive a word using $\mathrm{RS}_q(n, k, \boldsymbol{\alpha})$ where a fraction $p = \tau/n$ of the symbols have been corrupted by some error pattern, then to decode up to radius-p we would have to correctly decode the received word regardless of the distribution of the errors in the pattern. In the v-folded code, if the τ errors occurred within pn/v of the v-tuples, then we would still have to correct this error. If, on the other hand, the errors were spread throughout more than pn/v of the v-tuples, then even though only a fraction p of the individual symbols were corrupted, this error pattern would not have to

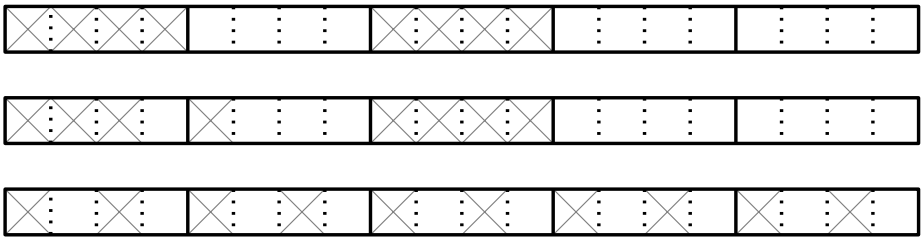

Figure 4.1: Error patterns

be corrected by a radius-(pN) decoder because more than pN of the v-tuples were corrupted. Figure 4.1 illustrates this scenario with three error patterns (with crossed-out boxes indicating nonzero error component) for a length 20 code and its 4-folded counterpart. A decoder achieving an error rate of $1/2$ would have to correct all three patterns in the unfolded code, but only the first pattern in the folded code. It seems like we are cheating by negating certain error patterns from our purview for being too evenly-distributed, but it is indeed a fair reinterpretation of the code and channel and it works very much to our favor. However, this abatement in error-correcting work comes at a cost. For a constant folding factor v, the alphabet size remains polynomial in the block length since we assume $n = q - 1$ for these folded codes. However, constructing a family of codes with error rates that approach capacity will call for raising the folding factor beyond a constant bound. At that point, extra care will be needed to reduce the alphabets to sizes polynomial in n.

4.3 Multivariate Based List Decoding

For the most part, multivariate interpolation-based list decoding will be a natural extension of the bivariate case. The key difference will be in the interpolation polynomial having roots that are algebraically related to each other. Before we begin, we need to make many simple definitions.

Definition 4.3.1 (Monomial/polynomial notation) *Let $w \in \mathbb{N}$. We denote the polynomial ring in $w+1$ variables over $\mathbb{F}$ by $\mathbb{F}[x, \boldsymbol{y}] = \mathbb{F}[x, y_1, \ldots, y_w]$. A monomial $m = x^{\rho_0} y_1^{\rho_1} \cdots y_w^{\rho_w}$ in $[x, \boldsymbol{y}]$ is written simply as $x^{\rho_0}\boldsymbol{y}^{\boldsymbol{\rho}}$ where $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_w)$. We denote by $|\boldsymbol{\rho}|$ the 1-norm of $\boldsymbol{\rho}$, so $|\boldsymbol{\rho}| = \sum_{i=1}^{w} \rho_i$. For any real $(w+1)$-tuple $\boldsymbol{u} = (u_0, u_1, \ldots, u_w)$, the **$\boldsymbol{u}$-weighted degree** of the monomial m is $\deg_{\boldsymbol{u}}(x^{\rho_0}\boldsymbol{y}^{\boldsymbol{\rho}}) := \sum_{i=0}^{w} u_i \rho_i$. The **$x$-part** of m is x^{ρ_0} and the **$\boldsymbol{y}$-part** of m is $\boldsymbol{y}^{\boldsymbol{\rho}}$. The **$x$-degree** of m is $\deg_x(m) = \rho_0$ and the **$\boldsymbol{y}$-degree** of m is $\deg_{\boldsymbol{y}}(m) = \boldsymbol{\rho}$. The **total degree** of m is $\deg_{\mathbf{1}}(m)$ where $\mathbf{1} \in \mathbb{R}^{w+1}$ is the all-1 vector. The **total $\boldsymbol{y}$-degree** of m is $\deg_{0,\mathbf{1}}(m) = \deg_{(0,1,\ldots,1)}(m)$. The **degree** or the **exponent** of m is $\deg_{x\boldsymbol{y}}(m) = (\rho_0, \rho_1, \ldots, \rho_w)$. The **$\boldsymbol{u}$-weighted degree** of a polynomial $f = \sum_{\rho_0, \boldsymbol{\rho}} f_{\rho_0, \boldsymbol{\rho}} x^{\rho_0}\boldsymbol{y}^{\boldsymbol{\rho}} \in \mathbb{F}[x, \boldsymbol{y}]$ is $\deg_{\boldsymbol{u}}(f) = \max_{(\rho_0, \boldsymbol{\rho})}\{\deg_{\boldsymbol{u}}(x^{\rho_0}\boldsymbol{y}^{\boldsymbol{\rho}})\}$. Likewise we extend the definitions of x-degree, total degree, and total $\boldsymbol{y}$-degree to the polynomial f in the natural way. Lastly, for any $a \in \mathbb{R}$ and $f \in \mathbb{F}[x, \boldsymbol{y}]$, we define $\deg_a(f) := \deg_{(1,a,\ldots,a)}(f)$.*

Definition 4.3.2 (Multivariate multiplicity and $\boldsymbol{y}$-roots) *A **root** (or **zero**) of a polynomial $f \in \mathbb{F}[x, \boldsymbol{y}]$ is a point $P \in \mathbb{F}^{w+1}$ such that $f(P) = 0$. The **multiplicity** of f at $P = (p_0, \ldots, p_w)$ is the smallest total degree of any nonzero term of the shifted polynomial $f(x + p_0, y_1 + p_1, \ldots, y_w + p_w)$. A **$\boldsymbol{y}$-root** of $f \in \mathbb{F}[x, \boldsymbol{y}]$ is a w-tuple $(g_1, \ldots, g_w) \in \mathbb{F}[x]^w$ such that $f(x, g_1, \ldots, g_w) = 0$.*

Sometimes we say f vanishes at P or f passes through P or f interpolates P if P is a root of f. If the multiplicity of f at P is greater than m, we generally consider f to still qualify as having multiplicity m. So when we refer to "all polynomials with multiplicity m at P," we really mean "all polynomials with multiplicity greater than or equal to m at P." We will wait to define the $\boldsymbol{y}$-degree of a polynomial in $\mathbb{F}[x, \boldsymbol{y}]$ until later when we fix our monomial order. Throughout this chapter we will use the degree weight vector $\boldsymbol{k} = (1, k-1, \ldots, k-1) \in \mathbb{N}^{w+1}$ to weigh our polynomials.

We proceed with generalizations of Propositions 3.7.2 and 3.7.3 that will be necessary to establish the correctness of the multivariate interpolation-based algorithm. In the following two propositions, we assume $P_i = (x_i, y_{i_1}, \ldots, y_{i_w})$ for $1 \leq i \leq n$ are n points in $\mathbb{F}^{w+1}$ with $x_1, \ldots, x_n$ distinct. These propositions are extensions of trivariate-based formulations given in [17] and [18] to the multivariate case.

Proposition 4.3.3 ($\boldsymbol{y}$-roots of a multivariate polynomial) *Suppose that $Q \in \mathbb{F}[x, \boldsymbol{y}]$ passes through each of the points $P_1, \ldots, P_n$ with multiplicity m and $\deg_{\boldsymbol{k}}(Q) \leq d = mt - 1$ for some $t \in \mathbb{N}$. If $f_1, \ldots, f_w \in \mathbb{F}[x]_k$ are polynomials such that each $f_j(x_i) = y_{i_j}$ for $j = 1, \ldots, w$ for at least t of the indices $i \in [n]$, then $(f_1, \ldots, f_w)$ is a $\boldsymbol{y}$-root of Q.*

Proof: Consider the polynomial $Q(x, f_1(x), \ldots, f_w(x)) \in \mathbb{F}[x]$. Since $\deg_{\boldsymbol{k}}(Q) < mt$, that means that every term $ax^{\rho_0}\boldsymbol{y}^{\boldsymbol{\rho}}$ of Q satisfies $\rho_i + (k-1)|\boldsymbol{\rho}| < mt$. Therefore, when we substitute $f_j(x)$ for y_j in each term $ax^{\rho_0}\boldsymbol{y}^{\boldsymbol{\rho}}$ of Q, the resulting polynomial $ax^{\rho_i} f_j(x)^{\rho_1} \cdots f(x)^{\rho_w}$ has degree at most $\rho_i + (k-1)|\boldsymbol{\rho}| < m(n - \tau)$ because $\deg(f_j(x)) \leq k - 1$. Thus we have that $\deg Q(x, f_1(x), \ldots, f_w(x)) \leq d$.

By assumption, $Q(x_i, y_{i_1}, \ldots, y_{i_w}) = 0$ for $i = 1, \ldots, n$ with multiplicity m. Since $(f_1(x_1), \ldots, f_w(x_i)) = (y_{i_1}, \ldots, y_{i_w})$ for at least t values of i we have at the corresponding points P_i that $Q(x_i, f_1(x_i), \ldots, f_w(x_i)) = Q(P_i) = 0$ with multiplicity m (note that since $Q(x+x_i, y_1+y_{i_1}, \ldots, y_w+y_{i_w})$ has no term with total degree less than m, $Q(x+x_i, f_1(x+x_i)+y_{i_1}, \ldots, f_w(x+x_i)+y_{i_w})$ also has no term of degree less than m and so it has multiplicity m at x_i in the univariate sense). So $Q(x, f_1(x), \ldots, f_w(x))$ is a polynomial of degree less than mt with at least mt roots (counting multiplicity). Therefore, $Q(x, f(x))$ is uniformly 0 so $(f_1, \ldots, f_w)$ is a $\boldsymbol{y}$-root of Q. $\square$

Proposition 4.3.4 (Existence of $Q(x, \boldsymbol{y})$) *Let $d, m \in \mathbb{N}$. If*

$$d > \sqrt[w+1]{(k-1)^w n (m+w)_{w+1}},$$

where $(m+w)_{w+1}$ denotes the falling factorial $(m+w)(m+w-1)\cdots(m+1)m$, then there exists a nonzero polynomial $Q \in \mathbb{F}[x, \boldsymbol{y}]$ that vanishes at each point $P_1, \ldots, P_n$ with multiplicity m and satisfies $\deg_{\boldsymbol{k}}(Q) \leq d$.

Proof: The condition that each P_i is a root of Q with multiplicity m means that no term of the shifted polynomial $Q(x+x_i, y_1+y_{i_1}, \ldots, y_w+y_{i_w})$ has total degree less than m. Then each term of the shifted polynomial with total degree less than m is a linear combination of the coefficients of Q. Since there are $\left(\!\!\binom{w+2}{m-1}\!\!\right) = \binom{m+w}{w+1}$ monomials in $[x, \boldsymbol{y}]$ of total degree less than m, this yields a homogeneous system of $n\binom{m+w}{w+1}$ equations in the variables corresponding to the coefficients of Q, a nontrivial solution to which is assured provided the number of equations is less than the number of unknowns, which we now count. Say the number of coefficients in Q, whose $\boldsymbol{k}$-degree is bounded above by d, is $N = |\{(\rho_0, \ldots, \rho_w) \in \mathbb{N}_0) \mid \rho_0+(k-1)\rho_1+\cdots+(k-1)\rho_w \leq d\}$. Geometrically, N is at least the volume of the $(w+1)$-dimensional simplex R in $\mathbb{R}^{w+1}_{\geq 0}$ bounded by the plane $x + (k-1)y_1 + \cdots + (k-1)y_w = d$. To justify this, observe that R is contained within the collection of $(w+1)$-dimensional unit cubes in $R^{w+1}_{\geq 0}$ with integral vertices, at least 1 of which lies within R; there are exactly N of these. The volume of the $(w+1)$-simplex R is

$$\frac{d^{w+1}}{(w+1)!(k-1)^w}. \tag{4.1}$$

This can be seen by starting with the edge of length d along the x-axis; call this the 1^{st} shape. Extending it as the base to the point $\frac{d}{k-1}$ on the y_1-axis forms a 2-dimensional cone (a triangle, to be conventional) of area $\frac{d^2}{2(k-1)}$; call this the 2^{nd} shape. Now, repeatedly extending the i^{th} shape into a new cone with its peak at $\frac{d}{k-1}$ on the y_i-axis to get the $(i+1)^{\text{st}}$ shape up to $i = w+1$, we get inductively that the i-dimensional volume of the i^{th} shape is $\frac{d^i}{i!(k-1)^{i-1}}$ because the volume of an i-dimensional cone is just $\frac{1}{i}$ times the product of its height and the $(i-1)$-dimensional volume of its base. The volume of R then follows because R is the $(w+1)^{\text{st}}$ shape. Alternatively, one could find the volume of R analytically by evaluating the integral

$$\int_0^{\frac{d}{u}} \cdots \int_0^{\frac{d}{u}-y_3-\cdots-y_w} \int_0^{\frac{d}{u}-y_2-\cdots-y_w} (d - uy_1 - uy_2 - \cdots - uy_w) dy_1 dy_2 \cdots dy_w,$$

where $u = k-1$. Either way, if equation (4.1) is more than the number of unknowns, $n\binom{m+w}{w+1}$, then a nonzero solution to the aforementioned system of equations, which corresponds to a nonzero polynomial Q interpolating each of the n points P_i with the multiplicity m, exists. Solving the inequality

$$\frac{d^{w+1}}{(w+1)!(k-1)^w} > n\binom{m+w}{w+1}$$

for d, we have $d > \sqrt[w+1]{(k-1)^w n \frac{(m+w)!}{(m-1)!}}$. $\square$

Propositions 4.3.3 and 4.3.4 tell us that if we find a multivariate polynomial of small weighted degree interpolating the n points with multiplicity m, then it will have $\boldsymbol{y}$-roots corresponding to polynomials in $\mathbb{F}[x]_k$ that also interpolate many (but

not all) of the points we choose. The question now is how do we most effectively choose those points so that we get the most out of the root-finding process in our effort to find all close codewords in $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$ to a received word $\boldsymbol{r} \in (\mathbb{F}_q^v)^N$? The answer to this lies in selecting the points so that the components of the $\boldsymbol{y}$-roots (polynomials in x) will be closely related. Parvaresh and Vardy ([30]) were among the pioneers of multivariate list decoding (others included Bleichenbacher, Kiayias, and Yung ([2]) as well as Coppersmith and Sudan ([6])) and were the first to enforce certain algebraic dependencies on the components of the $\boldsymbol{y}$-roots. Previous efforts had simply left off assumptions on their relations and attempted to use each variable as a means of running another instance of the decoder on a different codeword, but this yielded no improvement over the performance of previous decoding algorithms as there was a wide range in the possible values of the root components $f_1(x), \ldots, f_w(x)$. The only algebraic relation that was provided inherently through interpolation was $Q(f_1, \ldots, f_w) = 0$, but this was insufficient to make new coding gains. The key notion that Parvaresh and Vardy supplied that improved the $1-\sqrt{\kappa}$ decoding radius for rates below $1/16$ was to restrict the possibilities by enforcing that extra algebraic conditions be satisfied by the roots. Rather than having $f_1(x), \ldots, f_w(x)$ be w independent polynomials of degree less than k, they insisted that each $f_i(x) = f_1(x)^{j_i} \bmod h(x)$ for some irreducible polynomial $h \in \mathbb{F}_q[x]$. Then, after the interpolation polynomial is procured it is reduced modulo $h(x)$, producing a polynomial $R(\boldsymbol{y})$ in $y_1, \ldots, y_w$ with coefficients from the field $\mathbb{F}_q[x]/(h)$. The roots are then sought through the polynomial $R(z, z^{j_2}, \ldots, z^{j_w})$ via a univariate root-finding algorithm. The catch here is that with this technique, one must send w times as much information as was necessary for the underlying Reed-Solomon code, so a factor of w is lost from the rate of the original code and we cannot even construct codes with rates over $\frac{1}{w}$ this way (that is why Parvaresh and Vardy only yielded better error rates for low information rates). Guruswami and Rudra ([18]) overcame this rate loss by strengthening the root relations so that $f_i(x) = f_1(\alpha^{i-1}x)$. We make the following definition in light of these relations.

Definition 4.3.5 *Fix a primitive element $\alpha \in \mathbb{F}_q$. We say that a polynomial $f(x) \in \mathbb{F}_q[x]$ is a **folded root** (with respect to α) of the polynomial $Q \in \mathbb{F}_q[x, \boldsymbol{y}]$ if*

$$Q(x, f(x), f(\alpha), \ldots, f(\alpha^{w-1}x)) = 0.$$

By our particular code selection, it will be shown that each root coordinate is now just a cyclic shift of the previous one, so the evaluations of the f_i are almost identical and we no longer need to incorporate extra information for each root component into the code. Because the root components are to be cyclic shifts of each other, the prudent choice for our n points is to make $P_i = (\alpha^i, r_i, r_{i+1}, \ldots, r_{i+w-1})$ for $1 \leq i \leq n$, where the subscripts of $\boldsymbol{r}$ are taken modulo n and $\boldsymbol{r} = (r_1, \ldots, r_n) \in (\mathbb{F}_q^v)^N \cong \mathbb{F}_q^n$ is an element (ostensibly a received word) of the ambient space of $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$, although for simplicity we index it as an element of $\mathbb{F}_q^n$. Going forward, we take $\boldsymbol{r}$ and P_i to be so defined. Also, when we want to identify the folded symbols (the v-tuples that it is composed of as a folded codeword), we will refer to the j^{th} symbol as $\boldsymbol{r}_j$. That is, $\boldsymbol{r} = (\boldsymbol{r}_1, \ldots, \boldsymbol{r}_N) = (r_1, \ldots, r_n)$ where $N = n/v$, so $\boldsymbol{r}_j = (r_{(j-1)v+1}, \ldots, r_{jv}) \in \mathbb{F}_q^v$.

We now look at how low the degree of the interpolation polynomial Q must be to make the close codewords to $\boldsymbol{r}$ correspond to the folded roots of Q.

Lemma 4.3.6 ([17]) *Let $Q \in \mathbb{F}_q[x, \boldsymbol{y}]$ be a nonzero polynomial that interpolates each of the points P_i, $1 \leq i \leq n$, with multiplicity m. Let $\tau \in \mathbb{N}_0$ and let $d = \deg_{\boldsymbol{k}}(Q)$. If $d < m(v - w + 1)(N - \tau)$ then any $f(x) \in \mathbb{F}_q[x]_k$ that evaluates to a codeword $\boldsymbol{c}_f \in \mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$ which agrees with $\boldsymbol{r}$ on at least $N - \tau$ of the N locations is a folded root of Q.*

Proof: An essential point here is that since $\boldsymbol{r}$ and $\boldsymbol{c}_f$ agree on their i^{th} symbols,

$$(r_{(i-1)v+1}, \ldots, r_{iv}) = (f(\alpha^{(i-1)v+1}), \ldots, f(\alpha^{iv}))$$

for at least $N - \tau$ positions i. Hence, $f(\alpha^{i+j}) = r_{i+j}$ for $j = 0, \ldots, w-1$ for at least $(v - w + 1)(N - \tau)$ values of $i \in [n]$. Taking $f_j(x) = (\alpha^{j-1}x)$ for $j = 1, \ldots, w$, we get by Proposition 4.3.3 that $Q(x, f(x), f(\alpha x), \ldots, f(\alpha^{w-1}x)) = 0$. Thus, f is a folded root of Q. $\square$

Combining Lemma 4.3.6 with Proposition 4.3.4 we get a lower bound on the list decoding radius we can achieve with multivariate interpolation.

Proposition 4.3.7 (Multivariate List Decoding Radius) *For any error threshold $\tau \in \mathbb{N}_0$ and multiplicity parameter $m \in \mathbb{N}$, there exists a polynomial $Q \in \mathbb{F}_q[x, \boldsymbol{y}]$ with folded roots corresponding to all codewords in $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$ within distance τ of $\boldsymbol{r}$ if*

$$\tau \leq N - \left\lfloor N \frac{v}{v - w + 1} \sqrt[w+1]{\frac{(k-1)^w}{n^w} \left(1 + \frac{w}{m}\right) \cdots \left(1 + \frac{1}{m}\right)} \right\rfloor - 2. \tag{4.2}$$

Proof: By Proposition 4.3.4, we can find an interpolation polynomial Q of $\boldsymbol{k}$-weighted degree at most $d \in \mathbb{N}$ for d as small as

$$\left\lfloor \sqrt[m+1]{(k-1)^w n (m+w)_{w+1}} \right\rfloor + 1.$$

By Lemma 4.3.6, such a Q has the desired folded roots (polynomials corresponding to codewords within distance τ of $\boldsymbol{r}$) as long as $N - \tau > \frac{d}{m(v-w+1)}$. Given our bound on d and using the fact that $n = Nv$, we have

$$\begin{aligned}
\frac{d}{m(v-w+1)} &\leq \frac{\sqrt[w+1]{(k-1)^w n (m+w)_{w+1}} + 1}{m(v-w+1)} \\
&= \frac{Nv \sqrt[w+1]{\frac{(k-1)^w}{n^w}(1 + \frac{w}{m}) \cdots (1 + \frac{1}{m})}}{v - w + 1} + \frac{1}{m(v-w+1)} \\
&< \left\lfloor \frac{Nv \sqrt[w+1]{\frac{(k-1)^w}{n^w}\left(1 + \frac{w}{m}\right) \cdots \left(1 + \frac{1}{m}\right)}}{v - w + 1} \right\rfloor + 1 + \frac{1}{m(v-w+1)}
\end{aligned}$$

Hence, the desired polynomial Q exists when

$$N - \tau \geq \left\lfloor N \frac{v}{v-w+1} \sqrt[w+1]{\frac{(k-1)^w}{n^w}\left(1+\frac{w}{m}\right)\cdots\left(1+\frac{1}{m}\right)} \right\rfloor + 2,$$

which is equivalent to equation (4.2). $\square$

From Proposition 4.3.7, we have that multivariate list decoding of $[N, K]$ v-folded Reed-Solomon codes will yield an error rate of at least

$$\begin{aligned} p &= \frac{N - \left\lfloor N \frac{v}{v-w+1} \sqrt[w+1]{\frac{(k-1)^w}{n^w}\left(1+\frac{w}{m}\right)\cdots\left(1+\frac{1}{m}\right)} \right\rfloor - 2}{N} \\ &\geq 1 - \frac{v}{v-w+1} \cdot \frac{m+w}{m} \cdot \left(\frac{k-1}{n}\right)^{\frac{w}{w+1}} - \frac{2}{N}. \end{aligned} \tag{4.3}$$

Fixing the information rate $\kappa = \frac{k}{n} = \frac{K}{N}$ and taking the limit of p as N and w grow unbounded and v and m grow super-linear in w, say $v = m = w^2$, we find that

$$\lim_{\substack{N,w\to\infty \\ v=m=w^2}} p = 1 - \kappa.$$

Thus, in the limit of many variables and long folded lengths (and large alphabets, $q^v = q^{w^2}$ in this case) folded Reed Solomon codes offer the potential to decode a fraction of errors that approaches list decoding capacity. Of course, we need to have a way to efficiently decode them for this to be very useful.

We now present a multivariate list decoding algorithm for folded Reed-Solomon codes. The correctness of the algorithm has been established by the preceding discussion. Some more machinery is needed for actually performing the interpolation and root-finding steps, but these aspects will be treated thoroughly afterward.

Algorithm 4.3.8 (Multivariate List Decoding of $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$**)**
Input: received word $\boldsymbol{r} = (r_1, \ldots, r_n)$*, error threshold* $\tau \in \mathbb{N}_0$ *satisfying equation (4.2), number of variables* $w + 1$*, and multiplicity value* m
Output: list of codewords, $\mathcal{L}_{\tau,\boldsymbol{r}} \subseteq \mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$*, within Hamming distance* τ *of* $\boldsymbol{r}$

M1. Compute the degree bound $d = \left\lfloor \sqrt[w+1]{(k-1)^w n (m+w)_{w+1}} \right\rfloor + 1$ for Q.

M2. Find a nonzero polynomial $Q \in \mathbb{F}[x, \boldsymbol{y}]$ satisfying:

  a) $Q(\alpha^i, r_i, \ldots, r_{i+w-1}) = 0$ with multiplicity m for $i = 1, \ldots, n$,

  b) $\deg_{\boldsymbol{k}}(Q) \leq d$.

M3. Find all folded roots $f \in \mathbb{F}[x]_k$ of Q.

M4. For each polynomial f found in step M3, if $\mathrm{dist}(\boldsymbol{c}_f, \boldsymbol{r}) \leq \tau$ then output the codeword $\boldsymbol{c}_f$.

4.4 Folded Root-Finding

As we suggested before, the problem of finding $\boldsymbol{y}$-roots of the multivariate interpolation polynomial Q can be simplified by reducing Q modulo an irreducible $h \in \mathbb{F}_q[x]$ for which the coordinates of the $\boldsymbol{y}$-roots we seek are powers of each other (mod h). Then we can reduce the problem to a univariate root-finding problem by replacing the y_i variables of Q with appropriate powers of a single variable. The following lemma is from [17] and it indicates how we can make a prudent choice for $h(x)$.

Lemma 4.4.1 *Assuming $\alpha \in \mathbb{F}_q$ is a primitive element, the following hold:*

1. *$h(x) = x^{q-1} - \alpha \in \mathbb{F}_q[x]$ is irreducible over $\mathbb{F}_q$.*
2. *If $f(x) \in \mathbb{F}_q[x]$ has degree less than $q-1$, then $f(x)^q \bmod h(x) = f(\alpha x)$.*

Proof: See [17, Lemma 6.4]. □

We now show how to reduce the folded root-finding problem for Q down to a univariate root-finding problem which can be solved in time polynomial in q. The reduction hinges, however, on Q not having degree larger than $q-1$ in any y_i variable. For $1 \leq i \leq w$, $\deg_{y_i}(Q)$ can be at most $\deg_{\boldsymbol{k}}(Q)/(k-1)$. By step M1, $\deg_{\boldsymbol{k}}(Q)/(k-1) \leq \frac{\sqrt[w+1]{(k-1)^w n(m+w)_{w+1}}+1}{k-1} < (m+w)\sqrt[w+1]{n/(k-1)} + 1 < (m+w)\sqrt[w+1]{q} + 1$, since $n < q$. For this reduction to work, then, we should choose m and w so that $m + w < (q)^{w/(w+1)}$. This will probably not pose much of a hindrance.

Since $h(x) = x^{q-1} - \alpha$ is irreducible, the quotient ring $\mathbb{F}_q[x]/(h(x))$ is a field which we will denote by $F \cong \mathbb{F}_{q^{q-1}}$. The following lemma reduces the folded root-finding problem down to a univariate root-finding problem for polynomials over the quotient field F.

Lemma 4.4.2 *Let $Q \in \mathbb{F}_q[x, \boldsymbol{y}]$ such that $\deg_{y_i}(Q) < q$ for $i = 1, \ldots, w$ and suppose $f(x) \in \mathbb{F}_q[x]_k$ is a folded root of Q (with respect to $\alpha \in \mathbb{F}_q$). Let b be the largest power of $h(x) = x^{q-1} - \alpha$ that divides Q and set $Q'(x, \boldsymbol{y}) = \frac{Q(x,\boldsymbol{y})}{h(x)^b} \in \mathbb{F}_q[x, \boldsymbol{y}]$. Then*

$$R(z) := Q'(x, z, z^q, z^{q^2}, \ldots, z^{q^{w-1}}) \bmod h(x) \in F[z]$$

is nonzero and $f(x)$ is a root of $R(z)$.

Proof: Since f is a folded root of Q,

$$Q(x, f(x), f(\alpha x), \ldots, f(\alpha^{w-1}x)) = h(x)^b Q'(x, f(x), f(\alpha x), \ldots, f(\alpha^{w-1}x)) = 0,$$

so f is also a folded root of Q'. Now, $Q'(x, \boldsymbol{y}) \bmod h(x) \neq 0$ since Q' is not divisible by $h(x)$ (by its definition). Let $R(z) = Q'(x, z, z^q, \ldots, z^{q^{w-1}}) \bmod h(x) \in F[z]$. Since no variable y_i in Q has power greater than q in any term of Q, the same is true in Q' and so there can be no cancellation or combining of terms when we substitute $z^{q^{i-1}}$ for y_i. Hence, $R(z) = 0$ if and only if $Q'(x, z, z^q, \ldots, z^{q^{w-1}})$ is divisible by $h(x)$ if and only if $Q'(x, \boldsymbol{y})$ is divisible by $h(x)$, so $R(z) \neq 0$.

By Lemma 4.4.1,

$$\begin{aligned} R(f(x)) &= Q'(x, f(x), f(x)^q, \ldots, f(x)^{q^{w-1}}) \bmod h(x) \\ &= Q'(x, f(x), f(\alpha x), \ldots, f(\alpha^{w-1}x)) \bmod h(x) = 0, \end{aligned} \tag{4.4}$$

which shows f is a y-root of R. □

The folded root-finding problem can similarly be reduced to a bivariate root-finding problem, which allows us to complete the task using Algorithm 2.3.3. To guarantee success we must again assume that Q does not have y_i-degree as large as q in any y_i variable ($1 \le i \le w$).

Corollary 4.4.3 *Let $Q \in \mathbb{F}_q[x, \boldsymbol{y}]$ such that $\deg_{y_i}(Q) < q$ for $i = 1, \ldots, w$ and suppose $f(x) \in \mathbb{F}_q[x]_k$ is a folded root of Q (with respect to $\alpha \in \mathbb{F}_q$). Let b be the largest power of $h(x) = x^{q-1} - \alpha$ that divides Q and set $Q'(x, \boldsymbol{y}) = \frac{Q(x,\boldsymbol{y})}{h(x)^b} \in \mathbb{F}_q[x, \boldsymbol{y}]$. Then $f(x)$ is a y-root of the nonzero polynomial $R'(x, y) := Q'(x, y, y^q, \ldots, y^{q^{w-1}}) \bmod h(x)$ in $\mathbb{F}_q[x, y]/(h(x))$.*

Proof: As in Lemma 4.4.2, as f is a folded root of Q it is also a folded root of Q'. By the y_i-degree constraints on Q there cannot be cancellation or combining of terms when we substitute $y^{q^{i-1}}$ for y_i. Hence, $R'(x, y) = 0$ if and only if $Q'(x, \boldsymbol{y})$ is divisible by $h(x)$, which it is not. So R' is nonzero and has f as a y-root by equation (4.4) since $R'(x, z) = R(z)$ in that equation. □

With these results in hand, we may now give a more precise description of step M3 in Algorithm 4.3.8

Algorithm 4.4.4 (Folded Root Finding for Multivariate List Decoding)
Input: A polynomial $Q \in \mathbb{F}_q[x, \boldsymbol{y}]$ with $\deg_{y_i}(Q) < q$ for $i = 1, \ldots, w$, integer $k < q-1$, primitive element $\alpha \in \mathbb{F}_q$.
Output: a list of folded roots $f(x) \in \mathbb{F}_q[x]_k$ of Q.

`FoldedRootFind(`Q, k, α`,)`

F1. `compute` $h(x) = x^{q-1} - \alpha \in \mathbb{F}_q[x]$ `and set` $Q' = Q$

F2. `while` $h(x)$ `divides` $Q'(x, y)$ `in` $\mathbb{F}_q[x, \boldsymbol{y}]$

F3. `set` $Q' = Q'/h$

F4. `set` $R(x, y) = Q'(x, y, y^q, \ldots, y^{q^{w-1}}) \bmod h(x)$

F5. `return the results of the call ModRootFind(`$R, k-1, 0, h(x)$`) to Algorithm 2.3.10`

4.5 The Ideal Associated to a Code

The interpolation step of Guruswami and Rudra's v-folded Reed-Solomon list decoding algorithm calls for interpolating a nonzero polynomial of some bounded degree through each of the n points $P_i = (\alpha^i, r_i, r_i + 1, \ldots, r_{i+w-1})$, where subscripts of r are to be taken modulo n, with multiplicity $m > 1$. Such a polynomial (if one exists) is in the ideal

$$I_{r,m} = \{f \in \mathbb{F}[x, \boldsymbol{y}] \mid f \text{ vanishes with multiplicity } m \text{ at each } P_i\}.$$

By Proposition 4.3.3, our list decoding potential is maximized when the $\boldsymbol{k}$-weighted degree of the interpolation polynomial is minimized. Finding a solution then becomes a matter of finding a minimal polynomial of $I_{r,m}$ with respect to some graded monomial order $<$. For our order, we will take $<:=<_{k-1}$, which we define next.

Definition 4.5.1 *For any $d \in \mathbb{N}$, let $\boldsymbol{d} = (1, d, \ldots, d) \in \mathbb{N}^{w+1}$ and for any $\boldsymbol{\rho} \in \mathbb{N}^{w+1}$ let $|\boldsymbol{\rho}|_{\boldsymbol{d}}$ denote $\sum_{i=0}^{w} d_i \rho_i$. Define the relation $\prec_d$ on $\mathbb{N}_0^{w+1}$ by $\boldsymbol{\rho} \prec_d \boldsymbol{\lambda}$, for any $\boldsymbol{\rho} = (\rho_0, \ldots, \rho_w) \neq \boldsymbol{\lambda} = (\lambda_0, \ldots, \lambda_w) \in \mathbb{N}_0^{w+1}$, if*

> *$|\boldsymbol{\rho}|_{\boldsymbol{d}} < |\boldsymbol{\lambda}|_{\boldsymbol{d}}$, or $|\boldsymbol{\rho}|_{\boldsymbol{d}} = |\boldsymbol{\lambda}|_{\boldsymbol{d}}$ and the right-most nonzero entry in $(\rho_1, \ldots, \rho_w, \rho_0) - (\lambda_1, \ldots, \lambda_w, \lambda_0) \in \mathbb{Z}^{w+1}$ is positive.*[1]

Define the relation $<_d$ on the monomials of $\mathbb{F}[x, \boldsymbol{y}]$ by $x^{\rho_0} y^{\rho_1} \cdots y^{\rho_w} <_d x^{\lambda_0} y^{\lambda_1} \cdots y^{\lambda_w}$ if and only if $\boldsymbol{\rho} \prec_d \boldsymbol{\lambda}$.

In this chapter it shall often be the case that we wish to compare monomials or their exponent vectors with the x variable or component omitted. For these instances we make the following convention: the relation $\prec_d$ on $\mathbb{N}_0^w$ is defined to be the natural truncation of $\prec_d$ in Definition 4.5.1, given by $\boldsymbol{\rho} \prec_d \boldsymbol{\lambda}$, for any $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_w), \boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_1) \in \mathbb{N}_0^w$, if $(0, \rho_1, \ldots, \rho_w) \prec_d (0, \lambda_1, \ldots, \lambda_w)$. To abbreviate notation we will use $\boldsymbol{y}^{\boldsymbol{\rho}}$ to denote $y^{\rho_1} \cdots y^{\rho_w}$, even in the case where $\boldsymbol{\rho}$ is a $(w+1)$-tuple (the first coordinate, ρ_0, is disregarded in that case).

The orders $\prec_d$ and $<_d$ above are just the graded, weighted reverse lexicographic orders with the first coordinate (corresponding to x) moved to the end of the alphabet. We selected these orders so that for two monomials of $\mathbb{F}[x, \boldsymbol{y}]$ with equal weighted degrees, the one with larger total $\boldsymbol{y}$-degree will be the larger monomial with respect to $<_d$. This will be important when we verify the correctness and compute the complexity of the Algorithm 4.6.3, which will be presented later on. For now, we continue our discussion of $I_{r,m}$.

The minimal polynomial is necessarily an element in any Gröbner basis of $I_{r,m}$. Unfortunately, finding a Gröbner basis is generally a task of exponential complexity. For ideals in $\mathbb{F}[x, y]$, Buchberger showed in [3] and [4] that suitable adjustments could be made to his algorithm that ensured it finished in polynomial time. For ideals in $\mathbb{F}[x, y, z]$ the upper bound on run-time is singly-exponential in the input size and for more variables, with the exception of certain specialized cases, the best known bound

[1] $|\boldsymbol{a}|_{\boldsymbol{d}}$ here denotes the $\boldsymbol{d}$-weighted 1-norm of $\boldsymbol{a} \in \mathbb{N}_0^{w+1}$; that is $|\boldsymbol{a}|_{\boldsymbol{d}} = \sum_{i=1}^{w+1} d_i a_i$.

is doubly exponential in input size. However, by restricting the ideal $I_{r,m}$ to a certain submodule that retains the minimal polynomial we seek, we can hope to compute a Gröbner basis in considerably less than doubly exponential time (polynomial time, in fact), and thereby efficiently obtain our interpolation polynomial. This method was cleverly devised by Kwankyu Lee and Michael O'Sullivan in 2006 ([24]). They showed that it was an efficient generalization of the Berlekamp-Massey algorithm and it improved the time complexity of the bivariate interpolation step of the Guruswami-Sudan list decoding algorithm for Reed-Solomon codes from cubic to near-quadratic in the code length. We will adapt their method to the multivariate interpolation step for folded Reed-Solomon codes.

Let $\ell \geq m$ be an upper bound on the total $\boldsymbol{y}$-degree of the minimal polynomial of $I_{r,m}$ (recall this means that ℓ is a bound on the maximum total degree of the $\boldsymbol{y}$-part of any term in the minimal polynomial). Define the $\mathbb{F}[x]$-module $\mathbb{F}[x, \boldsymbol{y}]_\ell$ to be the restriction of $\mathbb{F}[x, \boldsymbol{y}]$ to those polynomials f satisfying $\deg_{0,\mathbf{1}}(f) \leq \ell$. Then we define $I_{r,m,\ell}$ to be the submodule

$$I_{r,m,\ell} = I_{r,m} \cap \mathbb{F}[x, \boldsymbol{y}]_\ell.$$

Our first task with $I_{r,m,\ell}$ is identifying a generating set. We obtain one by starting with a generating set of $I_{r,m}$. For several of the proofs that follow we use the notation $|\boldsymbol{a}| = \sum_{i=1}^{w} a_i$ to denote the 1-norm of $\boldsymbol{a} \in \mathbb{N}_0^w$ and $W_=(m) := \{\gamma \in \mathbb{N}_0^w : |\gamma| = m\}$ to denote the set of w-tuples that sum to m.

Proposition 4.5.2 *The ideal $I_{r,m}$ is generated by*

$$B = \{\eta^i \prod_{s=1}^{w} (y_k - h_{\sigma^{s-1}(\boldsymbol{r})})^{j_s} \mid i, j_1, \ldots, j_w \in \mathbb{N}_0, i + j_1 + \cdots + j_w = m\},$$

where $\eta = \prod_{s=1}^{n}(x - \alpha_s)$, $\sigma \in S_n$ is the cyclic shift left permutation, and $h_{\sigma^{s-1}(\boldsymbol{r})} \in \mathbb{F}[x]$ is the polynomial interpolated through the points $(\alpha^i, \sigma^{s-1}(r_i))$ for $1 \leq i \leq n$. By $\sigma^j(\boldsymbol{r})$ then we mean the n-tuple obtained by cyclically shifting the coordinates of r to the left j spaces (so $\sigma^j(\boldsymbol{r})_i = r_{i+j \bmod n}$).

Proof: To begin with, we note that $\langle B \rangle = \langle \eta, y_1 - h_r, \ldots, y_w - h_{\sigma^{w-1}(r)} \rangle^m$.

For each $\gamma \in W_{\leq}(m)$, let $g_\gamma = \eta^{m-|\gamma|} \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(\boldsymbol{r})})^{\gamma_s} \in B$. Since g has m factors that are zero at each point P_i it follows that g passes through each P_i with multiplicity m. Hence, $\langle B \rangle \subseteq I_{r,m}$. It remains to show that $\langle B \rangle \supseteq I_{r,m}$

Let $f \in I_{r,m}$. Using multivariate division with remainder, we can write

$$f = \sum_{\gamma \in W_{\leq}(m)} q_\gamma g_\gamma + R$$

where no term of R is divisible by the leading term of any generator in B. The leading terms of B are $\{x^{in} y_1^{j_1} \cdots y_w^{j_w} \mid i + j_1 + \cdots + j_w = m\}$. Hence, no term of R has $(1/n, 1, \ldots, 1)$-weighted degree as large as m. On the other hand, $R = f - \sum_{\gamma \in W_{\leq}(m)} q_\gamma g_\gamma$ is an element of $I_{r,m}$ since $f \in I_{r,m}$ and $B \subseteq I_{r,m}$. So R vanishes with multiplicity m at each P_i, meaning $R^{(i)} := R(x + \alpha^i, y_1 + r_i, \ldots, y_1 + r_{i+w-1})$ has

no term with total degree less than m. It must be then that $R^{(i)}$ is divisible by x, so R is divisible by $x-\alpha^i$. This is for $1 \leq i \leq n$, therefore η divides R. Then $R = \eta R_1$ for some $R_1 \in \mathbb{F}[x, y_1, \ldots, y_w]$ which has no term with $(1/n, 1, \ldots, 1)$-weighted degree as large as $m-1$. Moreover, since η has multiplicity 1 at P_i, R_1 must have multiplicity $m-1$ at P_i. So $R_1^{(i)} := R_1(x+\alpha^i, y_1+r_i, \ldots, y_1+r_{i+w-1})$ has no term with total degree less than $m-1$. This implies now that x divides $R_1^{(i)}$, so R_1 is divisible by η and hence R is divisible by η^2. Iterating this argument, we eventually obtain that R is divisible by η^m. But then $\mathrm{lt}(R)$ is divisible by $\mathrm{lt}(\eta^m) = x^{mn}$ unless $R = 0$ (and has no leading term). Therefore, since $\eta \in B$ we must have $R = 0$ and so $f \in \langle B \rangle$. Thus, $I_{r,m} = \langle B \rangle$. $\square$

A point worth noting is that the number of generators in the basis from Proposition 4.5.2 is $\left(\!\!\binom{w+1}{m}\!\!\right) = \binom{m+w}{m}$. An interesting observation here is that the number of generators of $I_{r,m} = \cap_{i=1}^{n} \langle x-\alpha^i, y_1-r_i, \ldots, y_m-r_{i+m-1}\rangle^m$, is independent of the number of points we interpolate through. This feature is probably unique to the affine case. If we were considering a projective ideal for this problem, we would expect that the number of basis generators would have to grow with the number of interpolation points.

From this basis B just described, we derive a generating set for $I_{r,m,\ell}$.

Corollary 4.5.3 *A generating set for $I_{r,m,\ell}$ as an $\mathbb{F}[x]$-submodule of $\mathbb{F}[x, \boldsymbol{y}]_\ell$ is*

$$\begin{aligned} G \;=\; & \left\{ \eta^i \prod_{s=1}^{w} (y_k - h_{\sigma^{s-1}(r)})^{j_s} \;\middle|\; i, j_s \in \mathbb{N}_0, i + j_1 + \cdots + j_w = m \right\} \cup \\ & \left\{ \prod_{s=1}^{w} \left[(y_s - h_{\sigma^{s-1}(r)})^{j_s} y_s^{J_s} \right] \;\middle|\; j_s, J_s \in \mathbb{N}_0, \Sigma j_s = m, \Sigma J_s \leq \ell - m \right\}. \quad (4.5) \end{aligned}$$

Proof: The condition $f \in I_{r,m,\ell}$ is equivalent to being able to write $f = \sum_{\gamma \in W_{\leq}(m)} h_\gamma g_\gamma$ as a sum of the generators of $I_{r,m}$ and $\deg_{0,\mathbf{1}}(f)$ being no more than ℓ. The latter condition just forces $\deg_{0,\mathbf{1}}(h_\gamma)$ to be no more than $d = \ell - \deg_{0,\mathbf{1}}(g_\gamma) = \ell - \sum_{i=1}^{w} \gamma_i$. Writing

$$h_\gamma = \sum_{\substack{i \in \mathbb{N}, a \in \mathbb{N}^w \\ |\boldsymbol{a}| \leq d}} h_{i,a} x^i \boldsymbol{y}^a,$$

we see that $f = \sum_{\gamma,i,a} h_{\gamma,i,a} x^i \boldsymbol{y}^a g_\gamma$ is an $\mathbb{F}[x]$-linear combination of the elements of G. $\square$

The basis G in Corollary 4.5.3 is handy for writing down an initial set of generators of $I_{r,m,\ell}$ but its size is a bit excessive. Fortunately, it can easily be reduced.

Corollary 4.5.4 *Another basis for $I_{r,m,\ell}$ as an $\mathbb{F}[x]$-submodule of $\mathbb{F}[x, \boldsymbol{y}]_\ell$ is*

$$G' \;=\; \left\{ \eta^i \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{j_s} \;\middle|\; i, j_s \in \mathbb{N}_0, i + j_1 + \cdots + j_w = m \right\} \cup$$

$$\left\{ \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{j_s} \,\middle|\, j_s \in \mathbb{N}_0, m < j_1 + \cdots + j_w \leq \ell \right\}. \tag{4.6}$$

Proof: Each polynomial of the form $g' = \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{j_s}$ (for some $j_s \in \mathbb{N}$ where $m < \sum j_s \leq \ell$) in G' can be written as an $\mathbb{F}[x]$-linear combination of elements of G because of Corollary 4.5.3 and the fact that $G' \subseteq I_{\boldsymbol{r},m} \cap \mathbb{F}[x, \boldsymbol{y}]_\ell$. Moreover, in this linear combination we may take the coefficient of the generator $g = \prod_{s=1}^{w} \left[(y_s - h_{\sigma^{s-1}(r)})^{j'_s} y_s^{j_s - j'_s} \right]$ (where $j'_s \in \mathbb{N}$ satisfy $\sum j'_s = m$ and $j'_s \leq j_s$ for all s) to be 1:

$$\begin{aligned}
g' &= \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{j_s} \\
&= \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{j'_s} \cdot \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{j_s - j'_s} \\
&= \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{j'_s} \cdot \left(\prod_{s=1}^{w} y_s^{j_s - j'_s} + \begin{array}{c} \text{lower order terms from the} \\ \text{product of binomial expansions} \end{array} \right) \\
&= g + \mathbb{F}[x]\text{-multiples of other elements of } G.
\end{aligned} \tag{4.7}$$

Therefore, these generators, g and g' may be swapped. (Note that g' can replace all g for any choice of j'_s's such that $\sum j'_s = m$ and $j'_s \leq j_s$ for all s.) $\square$

The basis G' has the advantage of having fewer generators than G. The size of G is

$$\begin{aligned}
& \left(\!\!\binom{w+1}{m}\!\!\right) + \left(\!\!\binom{w}{m}\!\!\right) \left(\sum_{i=1}^{\ell-m} \left(\!\!\binom{w}{i}\!\!\right) \right) \\
= \; & \left(\!\!\binom{w+1}{m}\!\!\right) + \left(\!\!\binom{w}{m}\!\!\right) \left(\left(\!\!\binom{w+1}{\ell-m}\!\!\right) - 1 \right) \\
= \; & \binom{w+m-1}{m-1} + \binom{m+w-1}{m} \binom{w+\ell-m}{\ell-m},
\end{aligned} \tag{4.8}$$

whereas the set G' only has

$$\begin{aligned}
\left(\!\!\binom{w+1}{m}\!\!\right) + \sum_{i=m+1}^{\ell} \left(\!\!\binom{w}{i}\!\!\right) &= \left(\!\!\binom{w+1}{m}\!\!\right) + \left(\!\!\binom{w+1}{\ell}\!\!\right) - \left(\!\!\binom{w+1}{m}\!\!\right) \\
&= \binom{w+\ell}{\ell}
\end{aligned} \tag{4.9}$$

generators. It may not be obvious that the expression in (4.9) is no larger than the one in (4.8), but in Corollary 4.6.9 we will see that G' has the same cardinality as a subset of G, making the point (that its size is smaller) clear. So it seems that G' will be the easier basis to manage at this time. For now, let us return to the task

of finding a Gröbner basis of $I_{r,m,\ell}$. The standard technique of testing whether a basis is a Gröbner basis is by Buchberger's S-pair criterion. However, under special circumstances there are other criteria we may use as well. The following proposition is a generalization of Proposition 12 in [24] from the bivariate to the multivariate case.

Proposition 4.5.5 *Let $<$ be a monomial order for $\mathbb{F}[x, \boldsymbol{y}]$ and let $S \subseteq \mathbb{F}[x, \boldsymbol{y}]_\ell$ be an $\mathbb{F}[x]$-submodule generated by $G = g_1, \ldots, g_t$. If $\deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_i))$ for $i = 1, \ldots, t$ are all distinct then G is a Gröbner basis of S with respect to $<$.*

Proof: Assume the premise of the proposition is true. Since there is no chance of confusion, we will denote $\mathrm{lt}_<(g)$ by $\mathrm{lt}(g)$. Let $f = f_1 g_1 + \cdots + f_t g_t \in S$, with $f_1, \ldots, f_t \in \mathbb{F}[x]$. Let $a_j = \deg f_j$ and $(b_{j0}, \ldots, b_{jw}) = \deg_{x\boldsymbol{y}} \mathrm{lt}(g_j)$ for all $j \in [t]$. We have that for some $j \in [t]$, $\mathrm{lt}(f_j g_j) \geq \mathrm{lt}(f_i g_i)$ for all $i \in [t]$. Suppose $\mathrm{lt}(f_j g_j) = \mathrm{lt}(f_i g_i)$ for some $i \neq j$ (note: this just means the leading monomials are equal, not necessarily the leading coefficients). Then $(a_j + b_{j0}, b_{j1}, \ldots, b_{jw}) = \deg_{x\boldsymbol{y}}(\mathrm{lt}(f_j g_j)) = \deg_{x\boldsymbol{y}}(\mathrm{lt}(f_i g_i)) = (a_i + b_{i0}, b_{i1}, \ldots, b_{iw})$, so $\deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_j)) = (b_{j1}, \ldots, b_{jw}) = (b_{i1}, \ldots, b_{iw}) = \deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_i))$, a contradiction of the premise that the generators' leading terms' $\boldsymbol{y}$-parts are distinct. Hence, $\mathrm{lt}(f_j g_j) > \mathrm{lt}(f_i g_i)$ for all $i \neq j$. Therefore, $\mathrm{lt}(f) = \mathrm{lt}(f_j g_j) = \mathrm{lt}(f_j) \cdot \mathrm{lt}(g_j)$. So the leading term of f is divisible by a leading term of one of the generators. Thus, G is a Gröbner basis of S. $\square$

Let us denote each generator of G' in (4.6) by $g_{\boldsymbol{\rho}} = \eta^{\rho_0} \prod_{s=1}^{w} (y_s - h_{\sigma^{s-1}(r)})^{\rho_s}$ where $\sum_{i=1}^{w} \rho_i \leq \ell$ and $\rho_0 = \max(0, m - \sum_{s=1}^{w} \rho_s)$. Notice that $g_{\boldsymbol{\rho}}$ has a term of the form $y_1^{\rho_1} \cdots y_w^{\rho_w} = \boldsymbol{y}^{\boldsymbol{\rho}}$ which would be the leading term if we had given the y_s variables weight $n-1$ instead of $k-1$. However, since $h_{\sigma^{s-1}(r)}$ has degree as large as $n-1$ in general, the leading term of $g_{\boldsymbol{\rho}}$ will generally be more x heavy than $\boldsymbol{y}$ heavy. Our goal, in order to obtain a Gröbner basis, will be to manipulate each $g_{\boldsymbol{\rho}}$ so that its leading term becomes $\boldsymbol{y}^{\boldsymbol{\rho}}$. Our strategy for achieving this shall have the flavor of a Euclidean algorithm. We will systematically take each $g_{\boldsymbol{\rho}}$ and subtract off multiples of an appropriate generator repeatedly to kill the leading term until it has $\boldsymbol{y}$-part $\boldsymbol{y}^{\boldsymbol{\rho}}$.

4.6 Gröbner Basis Algorithm

The algorithm in this subsection will rely on being given an $\mathbb{F}[x]$ submodule basis of $\mathbb{F}[x, \boldsymbol{y}]_\ell$ with the precondition that its generators can be ordered so that their leading $\boldsymbol{y}$-parts are strictly increasing. In this case, we could represent the basis with a row echelon form matrix over $\mathbb{F}[x]$ where the rows represent the generators (in descending order), the columns represent the different $\boldsymbol{y}$-parts present in the basis (again, in descending order), and the entries represent the corresponding $\mathbb{F}[x]$-coefficient of the $\boldsymbol{y}$-part of the generator. Given such a basis the algorithm will produce another basis satisfying the condition of Proposition 4.5.5, namely the $\boldsymbol{y}$-degrees of the leading terms will all be distinct. We will formalize these two basis conditions with the following definitions.

Definition 4.6.1 *Let G be a basis of an $\mathbb{F}[x]$-submodule of $\mathbb{F}[x, \boldsymbol{y}]_\ell$. If $\deg_{\boldsymbol{y}} \mathrm{lt}(g(1, \boldsymbol{y}))$ is different for each $g \in G$, then we say that it is an **echelon** (or pre-diagonal) basis. If $\deg_{\boldsymbol{y}} \mathrm{lt}(g(x, \boldsymbol{y}))$ is different for each $g \in G$, then we say that the basis is **diagonal**.*

Remark 4.6.2 *The basis G' in equation (4.6) is an echelon basis. However, the basis G in equation (4.5) is not echelon if $w > 1$, $\ell \geq m + 2$, and $w \geq 2$. In this case we could find $j_1, \ldots, j_w, J_1, \ldots, J_w \in \mathbb{N}_0$ such that $\sum j_s = m$, $\sum (j_k + J_k) \leq \ell$, $j_1 < m$, $j_2 > 0$, $J_1 = 1$, and $J_2 = 0$. Then $\prod_{s=1}^{w} [(y_s - h_{\sigma^{s-1}(r)})^{j_s} y_s^{J_s}]$ and $(y_1 - h_r)^{j_1+1}(y_2 - h_{\sigma(r)})^{j_2-1} y_2 \prod_{s=3}^{w} [(y_s - h_{\sigma^{s-1}(r)})^{j_s} y_s^{J_s}]$ are two different generators in G with the same maximal $\boldsymbol{y}$-part.*

The essence of the Gröbner basis computing algorithm is to do repeated S-polynomial-like computations, but we intentionally choose our S-pairs so that one of them will have a unit coefficient in the S-polynomial expression and can therefore be directly replaced by the S-polynomial, so we never have to increase the size of the basis. This algorithm is a generalization of Algorithm G in [25] from the bivariate to the multivariate case.

Algorithm 4.6.3 (Gröbner Basis Algorithm)
Description: converts a pre-conditioned basis to a Gröbner basis with the same number of generators.
Input: w (integer), ℓ (integer), and an echelon basis

$$G = \{g_{\boldsymbol{\rho}} \mid \boldsymbol{\rho} \in \mathbb{N}_0^w, |\boldsymbol{\rho}| < \ell\} \subseteq \mathbb{F}[x, \boldsymbol{y}]_\ell.$$

of a submodule of $\mathbb{F}[x, \boldsymbol{y}]_\ell$ of rank $\binom{w+\ell}{\ell}$ such that $\deg_{\boldsymbol{y}} \mathrm{lt}(g_{\boldsymbol{\rho}}(1, \boldsymbol{y})) = \boldsymbol{\rho}$ for all $\boldsymbol{\rho} \in \mathbb{N}_0^w$ with $|\boldsymbol{\rho}| < \ell$.
Output: a Gröbner basis of $\langle G \rangle$ with respect to $<=<_{k-1}$.

`GröbnerBasis(`w`, `ℓ`, `G`)`

1. `Let` $P = \{\boldsymbol{\rho} \mid \boldsymbol{\rho} \in \mathbb{N}_0^w, |\boldsymbol{\rho}| < \ell\}$ `and` $p = |P| = \binom{w+\ell}{\ell}$.
2. `Enumerate` $\boldsymbol{\rho}_1, \ldots, \boldsymbol{\rho}_p \in P$ `so that` $\boldsymbol{y}^{\boldsymbol{\rho}_1} < \cdots < \boldsymbol{y}^{\boldsymbol{\rho}_p}$.
3. `for` i `from` 2 `to` p
4. `while` $(\deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_{\boldsymbol{\rho}_i})) \neq \boldsymbol{\rho}_i)$
5. $\boldsymbol{\phi} := \deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_{\boldsymbol{\rho}_i}))$
   `Let` $a(x)$ `and` $b(x)$ `be the coefficients of the monomial` $\boldsymbol{y}^{\boldsymbol{\phi}}$ `in` $g_{\boldsymbol{\rho}_i}$ `and` $g_{\boldsymbol{\phi}}$`, respectively.`
   $d := \deg_x(a(x)) - \deg_x(b(x))$
   $c := \frac{\mathrm{lc}_<(a(x))}{\mathrm{lc}_<(b(x))}$
6. `if` $(d \geq 0)$ `then`
7. $g_{\boldsymbol{\rho}_i} := g_{\boldsymbol{\rho}_i} - c x^d g_{\boldsymbol{\phi}}$

8. else

9. swap $g_{\boldsymbol{\rho}_i} \longleftrightarrow g_{\boldsymbol{\phi}}$

10. $g_{\boldsymbol{\rho}_i} := g_{\boldsymbol{\rho}_i} - c^{-1}x^{-d}g_{\boldsymbol{\phi}}$

11. return $\{g_{\boldsymbol{\rho}}\}_{\boldsymbol{\rho}\in P}$.

4.6.1 Correctness

To prove the correctness of Algorithm 4.6.3 we must show two things: (1) the algorithm halts and (2) the set returned is a Gröbner Basis of the submodule generated by the input basis. To show the second part, we verify that the submodule generated by $G = \{g_{\boldsymbol{\rho}}\}$ remains the same as we update the generators during the course of the algorithm and that at the end the $\boldsymbol{y}$-degrees of the leading terms of the generators are distinct. To see that the algorithm terminates one simply needs to observe that each iteration of the while loop kills the leading term of $g_{\boldsymbol{\rho}_i}$ so that its new leading term is smaller than before in terms of the monomial order $<$. Since this can only happen finitely many times, termination follows. With this synopsis in mind, we proceed to prove the correctness with some invariants that will help our conceptual understanding of the algorithm.

Proposition 4.6.4 *On the input specified by the algorithm description, Algorithm 4.6.3 terminates and returns a set of generators for the same submodule as the input, but each of the returned generators $g_{\boldsymbol{\rho}}$ satisfies $\deg_{\boldsymbol{y}}(\mathrm{lt}(g_{\boldsymbol{\rho}})) = \boldsymbol{\rho}$. Thus, by Proposition 4.5.5, the algorithm returns a Gröbner basis of the submodule specified by the input.*

Proof: Assume the input conditions set forth by Algorithm 4.6.3. First of all, we verify that the basis returned by the algorithm generates the same submodule as the basis given. To see this, simply note that the only times we change the generators are in steps 7 and 10 and in either case the generator is only changed by subtracting off an $\mathbb{F}[x]$-multiple of another generator. Therefore, there is no change to the span of the generating set. Now we show it returns a Gröbner basis.

After the i^{th} iteration of the `for` loop, the following invariant holds for $1 \leq i \leq p$:

(I1) $\deg_{\boldsymbol{y}}(\mathrm{lt}(g_{\boldsymbol{\rho}})) = \boldsymbol{\rho}$ for $\boldsymbol{\rho}_0 \preceq \boldsymbol{\rho} \preceq \boldsymbol{\rho}_i$

We know that (I1) holds initially because $\boldsymbol{\rho}_1$ must be $(0, \ldots, 0)$ since $(0, \ldots, 0) \in P$, $<$ is a monomial order, and, by assumption, $\deg_{\boldsymbol{y}} \mathrm{lt}(g_{\boldsymbol{\rho}_1}(1, y_1, \ldots, y_w)) = \boldsymbol{y}^{\boldsymbol{\rho}_1} = 1$. It follows that $g_{\boldsymbol{\rho}_1}$ is a polynomial in just x, so $\deg_{\boldsymbol{y}}(\mathrm{lt}(g_{\boldsymbol{\rho}_1})) = (0, \ldots, 0) = \boldsymbol{\rho}_1$.

Suppose that (I1) holds after the i^{th} iteration for some i with $0 \leq i < p$. During the $(i+1)^{\text{st}}$ iteration, the only time any generator $g_{\boldsymbol{\rho}}$, $\boldsymbol{\rho} \preceq \boldsymbol{\rho}_i$, changes is when it is swapped with $g_{\boldsymbol{\rho}_{i+1}}$ in step 9. But in that case, $\deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_{\boldsymbol{\rho}})) = \deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_{\boldsymbol{\rho}_{i+1}}))$, so even then $\deg_{\boldsymbol{y}}(\mathrm{lt}(g_{\boldsymbol{\rho}}))$ remains unchanged, although the total degree of $g_{\boldsymbol{\rho}}$ decreases since $d < 0$ (this will be important for showing termination). Hence, we still have $\deg_{\boldsymbol{y}}(\mathrm{lt}(g_{\boldsymbol{\rho}})) = \boldsymbol{\rho}$ for $\boldsymbol{\rho}_1 \leq \boldsymbol{\rho} \leq \boldsymbol{\rho}_i$ after the $(i+1)^{\text{st}}$ iteration. As for $g_{\boldsymbol{\rho}_{i+1}}$, the $(i+1)^{\text{st}}$

iteration stops precisely when $\deg_{\boldsymbol{y}}(\mathrm{lt}_<(g_{\boldsymbol{\rho}_{i+1}})) = \boldsymbol{\rho}_{i+1}$. Therefore, (I1) holds after the $(i+1)^{\text{st}}$ iteration too.

Finally, to show the algorithm halts we must show the `while` loop terminates during each iteration i of the `for` loop. To see this, observe that $g_{\boldsymbol{\rho}_i}$ only changes in steps 7, 9, and 10. In step 9, as mentioned above, the total degree of one of the generators $g_{\boldsymbol{\rho}}$, $\boldsymbol{\rho} \prec \boldsymbol{\rho}_i$, strictly decreases and this is the only step where the generators preceding $g_{\boldsymbol{\rho}_i}$ change so step 9 can only be executed a finite number of times. Step 10 can only run finitely many times, too, then. So after finitely many iterations, the condition of step 6 must always be true and the algorithm only runs step 7 and never steps 9 or 10. In step 7, $g_{\boldsymbol{\rho}_i}$ is replaced by a polynomial whose leading term either has smaller $\boldsymbol{k}$-weighted degree or has the same weighted degree but its leading term is smaller with respect to $<$. Its weighted degree cannot increase because the weighted degree of $cx^d g_{\boldsymbol{\phi}}$ is the same as that of $g_{\boldsymbol{\rho}_i}$ (they have the same leading monomial), so it can only decrease or remain the same when we take their difference. If it remains the same, then the new leading monomial was also a monomial in either $g_{\boldsymbol{\rho}_i}$ or $x^d g_{\boldsymbol{\phi}}$ and so must have been smaller with respect to $<$ than the old leading term. The leading term can only decrease finitely many times before $g_{\boldsymbol{\rho}_i}$ becomes 0. But this cannot happen since $g_{\boldsymbol{\rho}_1}, \ldots, g_{\boldsymbol{\rho}_i}$ form a rank $i+1$ free $\mathbb{F}[x]$-module (note the generators all have distinct $\boldsymbol{y}$-degrees to begin with). So the `while` loop must terminate before that. □

4.6.2 Complexity

The complexity bound we obtain in this section will depend on which particular basis of the submodule we choose to start with when we call the procedure *Gröbner Basis(w, ℓ, G)*. We will discuss the costs and benefits of the different bases we might choose at the end of the section. In the meantime, the key to bounding the complexity of Algorithm 4.6.3 will be determining the number of times that the `while` loop (step 4) may be run during each iteration of the `for` loop (step 3). Proposition 4.6.4 showed that step 4 can only be run a finite number of times but it did not indicate a precise bound on that number. We know from the proof that the $\boldsymbol{k}$-weighted degrees of the generators never increase but during each iteration of the `while` loop either the weighted degree of one generator decreases or the $\boldsymbol{y}$-degree of the leading term of the target generator, $g_{\boldsymbol{\rho}_i}$ during the i^{th} iteration of the `for` loop, decreases with respect to $<$. At the start of the algorithm, we have that $\deg_{\boldsymbol{k}} g_{\boldsymbol{\rho}} \leq (n-1)^{\max\{m,|\boldsymbol{\rho}|\}} \leq (n-1)^{\ell}$ for all $\boldsymbol{\rho} \in P$. The maximum number of times the leading term of the target polynomial can jump before either the iteration terminates or the weighted degree of a generator falls is i, the iteration number for step 3. The worst case would be if all of the jumping occurred during the last (p^{th}) iteration of step 3. In this case, we get as a crude upper bound on the number of times step 4 is passed:

$$p(n-1)^{\ell} = \binom{w+\ell}{w}(n-1)^{\ell}. \tag{4.10}$$

We know from Proposition 4.3.4 that the minimal polynomial of $I_{\boldsymbol{r},m}$ has $\boldsymbol{k}$-weighted degree at most $d = \left\lfloor \sqrt[w+1]{(k-1)^w n(m+w)_{w+1}} \right\rfloor + 1$. Therefore, its total $\boldsymbol{y}$-degree bound ℓ satisfies the inequality

$$\begin{aligned} \ell &\leq \frac{d}{k-1} = \frac{\left\lfloor \sqrt[w+1]{(k-1)^w n(m+w)_{w+1}} \right\rfloor + 1}{k-1} \\ &\leq (m+w) \sqrt[w+1]{\frac{n}{k-1}} + 1. \end{aligned} \tag{4.11}$$

Combining 4.11 with 4.10 we have that the algorithm's `while` loop loops at most

$$\begin{aligned} \binom{w+\ell}{w}(n-1)^\ell &\leq \binom{w + \sqrt[w+1]{\frac{n}{k-1}}(m+w) + 1}{w} (n-1)^{(m+w)\sqrt[w+1]{\frac{n}{k-1}}+1} \\ &\in \mathcal{O}\left(\left(em \left(1 + \sqrt[w+1]{\frac{n}{k-1}} \right) \right)^w n^{(m+w)/\sqrt[w+1]{\kappa}+1} \right) \end{aligned} \tag{4.12}$$

times. This could be exponential in n if k were fixed and n were growing (whereby $1/\kappa$ would grow linearly in n). But we can actually achieve a smaller complexity than in (4.12). Obtaining a better grasp of the complexity requires establishing a tighter bound on the number of times step 4 is executed in the algorithm. First, it is important to notice that up through the i^{th} iteration of step 3, amongst all of the $\boldsymbol{y}$-parts occurring in the monomials of $g_{\boldsymbol{\rho}_1}, \ldots, g_{\boldsymbol{\rho}_i}$, $\boldsymbol{y}^{\boldsymbol{\rho}_i}$ is maximal with respect to $<$. This follows from the ordering we choose for $g_{\boldsymbol{\rho}_1}, ..., g_{\boldsymbol{\rho}_p}$. Furthermore, amongst all possible monomials in $g_{\boldsymbol{\rho}_1}, ..., g_{\boldsymbol{\rho}_p}$ of a fixed degree equal to $\deg_{\boldsymbol{k}}(x^u \boldsymbol{y}^{\boldsymbol{\rho}_i})$ for some $u \geq 0$, $x^u \boldsymbol{y}^{\boldsymbol{\rho}_i}$ is maximal. Now we use this and the fact that the `while` loop closes the gap between the weighted degree of the leading term of the target generator, $g_{\boldsymbol{\rho}_i}$, and that of the terms with the desired $\boldsymbol{y}$-part, $\boldsymbol{y}^{\boldsymbol{\rho}_i}$, by repeatedly killing off the leading term of $g_{\boldsymbol{\rho}_i}$. This is expressed more formally in the next proposition whose proof depends on the lemma that follows it (and the lemma is proved first).

Proposition 4.6.5 *Consider the i^{th} iteration of the `for` loop of Algorithm 4.6.3 ($2 \leq i \leq p$). After each run through the `while` loop, the difference between the $\boldsymbol{k}$-weighted degrees of the leading term of $g_{\boldsymbol{\rho}_i}$ and its maximum term (with respect to $<_{k-1}$) with $\boldsymbol{y}$-part $\boldsymbol{y}^{\boldsymbol{\rho}_i}$ either strictly decreases or the difference remains the same but the $\boldsymbol{y}$-part of the leading term strictly decreases (with respect to $<_{k-1}$).*

The proof of Proposition 4.6.5 relies on an interesting invariant of the algorithm whose long, technical proof is encapsulated in the following lemma. The lemma and its proof are based on Proposition 7 of [25], in which Lee and O'Sullivan proved the result for the bivariate case ($w = 1$).

Lemma 4.6.6 *Let $g_{\boldsymbol{\rho}_s} = \sum_{\boldsymbol{\rho} \in P} a_{\boldsymbol{\rho}_s \boldsymbol{\rho}} \boldsymbol{y}^{\boldsymbol{\rho}}$, $1 \leq s \leq p$, $a_{\boldsymbol{\rho}_s \boldsymbol{\rho}} \in \mathbb{F}[x]$, represent the basis G in Algorithm 4.6.3 during the i^{th} iteration of the `for` loop. Then before and after each run through the `while` loop (steps 5-10) the following invariant condition holds for any non-identity permutation $\pi \in S(P)$:*

(I2) $\displaystyle\sum_{\boldsymbol{\rho}\in P}\deg(a_{\boldsymbol{\rho}\boldsymbol{\rho}}) > \sum_{\boldsymbol{\rho}\in P}\deg(a_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})})$.

Proof: To start with, we must show that (I2) holds when the algorithm is initiated. For any $\pi \in S(P)$ and basis state $G = \{g_{\boldsymbol{\rho}} \mid \boldsymbol{\rho} \in P\}$ let $D_G(\pi) = \sum_{\boldsymbol{\rho}\in P}\deg(a_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})})$. Then condition (I2) just says that $\mathrm{id}_P \in S(P)$ maximizes $D_G(\pi)$ over all $\pi \in S(P)$.

Considering the left-hand side of (I2), initially we have $\deg(a_{\boldsymbol{\rho}\boldsymbol{\rho}}) \geq 0$ since the basis G is echelon, which implies $\deg_{\boldsymbol{y}}(g_{\boldsymbol{\rho}}) = \boldsymbol{\rho}$, so $a_{\boldsymbol{\rho}\boldsymbol{\rho}} \neq 0$. So the left-hand side of (I2) is non-negative. For the right-hand side of (I2), observe first that initially, since G is echelon, $a_{\boldsymbol{\rho}\boldsymbol{\lambda}} = 0$ for all $\boldsymbol{\lambda} \succ \boldsymbol{\rho}$. Since the permutation π considered in (I2) is not the identity, there is necessarily some $\boldsymbol{\rho} \in P$ such that $\pi(\boldsymbol{\rho}) \succ_{k-1} \boldsymbol{\rho}$, in which case $\deg(a_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})}) = -\infty$. This makes $D_G(\pi) = -\infty$, while $D_G(\mathrm{id}_P)$ is strictly non-negative, so (I2) is true initially.

It remains to show that after each run through the `while` loop (steps 5-10), which is the only place the basis G changes, property (I2) is preserved. Let $g_{\boldsymbol{\rho}} = \sum_{\boldsymbol{\phi}\in P} a_{\boldsymbol{\rho}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}$ and $g'_{\boldsymbol{\rho}} = \sum_{\boldsymbol{\phi}\in P} a'_{\boldsymbol{\rho}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}$, where $\boldsymbol{\rho} \in P$ and $a_{\boldsymbol{\rho}\boldsymbol{\phi}}, a'_{\boldsymbol{\rho}\boldsymbol{\phi}} \in \mathbb{F}[x]$, represent the states of the basis G before and after, respectively, an execution of the `while` loop during the i^{th} iteration of the `for` loop. Also, to simplify notation, we will let $\delta_{\boldsymbol{\rho}\boldsymbol{\phi}} = \deg(a_{\boldsymbol{\rho}\boldsymbol{\phi}})$ and $\delta'_{\boldsymbol{\rho}\boldsymbol{\phi}} = \deg_{k-1}(a'_{\boldsymbol{\rho}\boldsymbol{\phi}})$ for all $\boldsymbol{\rho}, \boldsymbol{\phi} \in P$.

First, consider the case when $d \geq 0$ in step 6. Since (I2) holds before the generator replacement of step 7, when we take the transposition $\pi' = (\boldsymbol{\rho}_i\ \boldsymbol{\phi})$ we have $D_G(\pi') < D_G(\mathrm{id}_P)$, which implies $\delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} - \delta_{\boldsymbol{\phi}\boldsymbol{\phi}} + \delta_{\boldsymbol{\phi}\boldsymbol{\rho}_i} = d + \delta_{\boldsymbol{\phi}\boldsymbol{\rho}_i} < \delta_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}$. Hence, $\delta'_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i} = \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i} - cx^d a_{\boldsymbol{\phi}\boldsymbol{\rho}_i}) = \delta_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}$. Since $g_{\boldsymbol{\rho}_i}$ is the only generator altered, it follows that $D_G(\mathrm{id}_P) = D_{G'}(\mathrm{id}_P)$. Now, $\delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} = \deg(a_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - cx^d a_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)}) \leq \max\{\delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)}, d + \delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)}\}$. If $\delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} \leq \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)}$ then $D_{G'}(\mathrm{id}_P) = D_G(\mathrm{id}_P) > D_G(\pi) \geq D_{G'}(\pi)$ for all $\pi \in S(P)\backslash\{\mathrm{id}_P\}$, and (I2) still holds. Otherwise, $\delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} = d + \delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)}$ and we treat two subcases depending on whether $\boldsymbol{\rho}_i$ and $\boldsymbol{\phi}$ are in the same cycle of π or not.

Suppose $\boldsymbol{\rho}_i$ and $\boldsymbol{\phi}$ are in the same cycle of π, say

$$\pi_1 = (\boldsymbol{\phi}\ \pi(\boldsymbol{\phi})\ \ldots\ \boldsymbol{\rho}_i\ \pi(\boldsymbol{\rho}_i)\ \ldots\ \pi^{-1}(\boldsymbol{\phi})).$$

Let $\Delta_{\boldsymbol{\rho}\boldsymbol{\phi}} = \deg_{k-1}(a_{\boldsymbol{\rho}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}})$ and $\Delta'_{\boldsymbol{\rho}\boldsymbol{\phi}} = \deg_{k-1}(a'_{\boldsymbol{\rho}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}})$ for all $\boldsymbol{\rho}, \boldsymbol{\phi} \in P$. From our supposition on $\delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)}$ we have $\Delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} = d + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)}$ and by invariant (I1) $\Delta_{\boldsymbol{\rho}\boldsymbol{\lambda}} \leq \Delta_{\boldsymbol{\rho}\boldsymbol{\rho}}$ for $\boldsymbol{\rho}_0 \preceq \boldsymbol{\rho} \prec \boldsymbol{\rho}_i$ and $\boldsymbol{\lambda} \in P$. Lastly, let $R = \{\pi(\boldsymbol{\rho}_i), \pi^2(\boldsymbol{\rho}_i), \ldots, \pi^{-1}(\boldsymbol{\phi})\}$ and let $\tau \in S(P)\backslash\{\mathrm{id}_P\}$ be the permutation

$$\tau(\boldsymbol{\rho}) = \begin{cases} \boldsymbol{\rho}, & \text{if } \boldsymbol{\rho} \in R \\ \pi(\boldsymbol{\rho}), & \text{if } \boldsymbol{\rho} \notin R \cup \{\boldsymbol{\rho}_i\} \\ \boldsymbol{\phi}, & \text{if } \boldsymbol{\rho} = \boldsymbol{\rho}_i \end{cases}$$

Then, using the facts here stated, we have

$$\begin{aligned} & \sum_{\boldsymbol{\rho}\in P}(\delta'_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\ = & \sum_{\boldsymbol{\rho}\in P}\Delta'_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} + \Delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} + d + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)} \\
&= \sum_{\boldsymbol{\rho}\in R} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R\cup\{\boldsymbol{\phi},\boldsymbol{\rho}_i\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} - \Delta_{\boldsymbol{\phi}\boldsymbol{\phi}} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)} \\
&\leq \sum_{\boldsymbol{\rho}\in R} \Delta_{\boldsymbol{\rho}\boldsymbol{\rho}} + \sum_{\boldsymbol{\rho}\notin R\cup\{\boldsymbol{\phi},\boldsymbol{\rho}_i\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} \\
&= \sum_{\boldsymbol{\rho}\in P} (\delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&< \sum_{\boldsymbol{\rho}\in P} (\delta_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&= \sum_{\boldsymbol{\rho}\in P} (\delta'_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})), \qquad (4.13)
\end{aligned}
$$

which implies (I2) still holds. If, on the other hand, $\boldsymbol{\rho}_i$ and $\boldsymbol{\phi}$ are in distinct cycles of π, let $\pi_1 = (\boldsymbol{\phi}\ \pi(\boldsymbol{\phi})\ \ldots\ \pi^{-1}(\boldsymbol{\phi}))$ be the cycle of π containing $\boldsymbol{\phi}$. We use the same notation as before, except that now we take $R = \{\boldsymbol{\lambda} \in \pi_1 \mid \boldsymbol{\lambda} \neq \boldsymbol{\phi}\}$ and $\tau \in S(P)\backslash\{\mathrm{id}_P\}$ to be the permutation

$$
\tau(\boldsymbol{\rho}) = \begin{cases} \boldsymbol{\rho}, & \text{if } \boldsymbol{\rho} \in R \\ \pi(\boldsymbol{\rho}), & \text{if } \boldsymbol{\rho} \notin R \cup \{\boldsymbol{\phi}, \boldsymbol{\rho}_i\} \\ \pi(\boldsymbol{\rho}_i), & \text{if } \boldsymbol{\rho} = \boldsymbol{\phi} \\ \boldsymbol{\phi}, & \text{if } \boldsymbol{\rho} = \boldsymbol{\rho}_i \end{cases}
$$

Then, using the same reasons as before, we have

$$
\begin{aligned}
&\sum_{\boldsymbol{\rho}\in P} (\delta'_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta'_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} + \Delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} + d + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)} \\
&= \sum_{\boldsymbol{\rho}\in R} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R\cup\{\boldsymbol{\phi},\boldsymbol{\rho}_i\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} - \Delta_{\boldsymbol{\phi}\boldsymbol{\phi}} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)} \\
&\leq \sum_{\boldsymbol{\rho}\in R} \Delta_{\boldsymbol{\rho}\boldsymbol{\rho}} + \sum_{\boldsymbol{\rho}\notin R\cup\{\boldsymbol{\phi},\boldsymbol{\rho}_i\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})}
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{\boldsymbol{\rho}\in P}(\delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&< \sum_{\boldsymbol{\rho}\in P}(\delta_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&= \sum_{\boldsymbol{\rho}\in P}(\delta'_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})), \qquad (4.14)
\end{aligned}
$$

so again, (I2) still holds after the replacement of $g_{\boldsymbol{\rho}_i}$ in step 7.

Next we use a similar treatment to handle the case when $d < 0$ in step 6. We will reuse all of the before and after notation (before and after steps 9-10 in this case) from the first case, but this time we will have that $a'_{\boldsymbol{\phi}\boldsymbol{\rho}} = a_{\boldsymbol{\rho}_i\boldsymbol{\rho}}$ and $a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}} = a_{\boldsymbol{\phi}\boldsymbol{\rho}} - c^{-1}x^{-d}a_{\boldsymbol{\rho}_i\boldsymbol{\rho}}$ since $g'_{\boldsymbol{\phi}} = g_{\boldsymbol{\rho}_i}$ and $g'_{\boldsymbol{\rho}_i} = g_{\boldsymbol{\phi}} - c^{-1}x^{-d}g_{\boldsymbol{\rho}_i}$. Because (I2) holds before the generator replacements of steps 9-10, applying it to the transposition $\pi' = (\boldsymbol{\rho}_i\ \boldsymbol{\phi})$ we have $D_G(\pi') < D_G(\mathrm{id}_P)$, which implies $\delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} - \delta_{\boldsymbol{\phi}\boldsymbol{\phi}} + \delta_{\boldsymbol{\phi}\boldsymbol{\rho}_i} = d + \delta_{\boldsymbol{\phi}\boldsymbol{\rho}_i} < \delta_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}$. Hence, $\delta'_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i} = \deg(a_{\boldsymbol{\phi}\boldsymbol{\rho}_i} - c^{-1}x^{-d}a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) = \delta_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i} - d$. But the increase of $(-d)$ in the diagonal summand here is counterbalanced by the fact that $\delta'_{\boldsymbol{\phi}\boldsymbol{\phi}} = \delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} = \delta_{\boldsymbol{\phi}\boldsymbol{\phi}} + d$. It follows now that $D_G(\mathrm{id}_P) = D_{G'}(\mathrm{id}_P)$. Now, $\delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} = \deg(a_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)} - c^{-1}x^{-d}a_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)}) \leq \max\{\delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)}, \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - d\}$ (strict inequality is possible only if both arguments are equal). If $\delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} \leq \delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)}$ then

$$
\begin{aligned}
D_{G'}(\pi) &= D_G(\pi) + \delta'_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} - \delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} \\
&= D_G(\pi) + \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} - \delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} \\
&\leq D_G(\pi) + \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} - \delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \delta_{\boldsymbol{\phi}\pi(\boldsymbol{\rho}_i)} - \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} \\
&= D_G(\pi) \geq D_{G'}((\boldsymbol{\rho}_i\ \boldsymbol{\phi}) \circ \pi) \\
&< D_G(\mathrm{id}_P) \\
&= D_{G'}(\mathrm{id}_P) \qquad (4.15)
\end{aligned}
$$

for all $\pi \in S(P)\backslash\{\mathrm{id}_P\}$, and (I2) still holds. Otherwise, $\delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} = \delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - d$ and we again treat two subcases depending on whether $\boldsymbol{\rho}_i$ and $\boldsymbol{\phi}$ are in the same cycle of π or not.

Suppose $\boldsymbol{\rho}_i$ and $\boldsymbol{\phi}$ are in the same cycle of π, say

$$\pi_1 = (\boldsymbol{\phi}\ \pi(\boldsymbol{\phi})\ \ldots\ \boldsymbol{\rho}_i\ \pi(\boldsymbol{\rho}_i)\ \ldots\ \pi^{-1}(\boldsymbol{\phi})).$$

Using the notation of the $d \geq 0$ case, we have $\Delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} = \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - d$ and by invariant (I1) $\Delta_{\boldsymbol{\rho}\boldsymbol{\lambda}} \leq \Delta_{\boldsymbol{\rho}\boldsymbol{\rho}}$ for $\boldsymbol{\rho}_0 \preceq \boldsymbol{\rho} \prec \boldsymbol{\rho}_i$ and $\boldsymbol{\lambda} \in P$. Let $R = \{\pi(\boldsymbol{\rho}_i), \pi^2(\boldsymbol{\rho}_i), \ldots, \pi^{-1}(\boldsymbol{\phi})\,\boldsymbol{\phi}\}$ and let $\tau \in S(P)\backslash\{\mathrm{id}_P\}$ be the permutation

$$\tau(\boldsymbol{\rho}) = \begin{cases} \boldsymbol{\rho}, & \text{if } \boldsymbol{\rho} \in R \\ \pi(\boldsymbol{\rho}), & \text{if } \boldsymbol{\rho} \notin R \cup \{\boldsymbol{\rho}_i\} \\ \pi(\boldsymbol{\phi}), & \text{if } \boldsymbol{\rho} = \boldsymbol{\rho}_i \end{cases}$$

Then, using the facts here stated, we have

$$\sum_{\boldsymbol{\rho}\in P}(\delta'_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}}))$$

$$
\begin{aligned}
&= \sum_{\boldsymbol{\rho}\in P} \Delta'_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} + \Delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta'_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - d - \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in R\setminus\{\boldsymbol{\phi}\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R\cup\{\boldsymbol{\rho}_i\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - d - \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in R\setminus\{\boldsymbol{\phi}\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R\cup\{\boldsymbol{\rho}_i\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - \Delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} + \Delta_{\boldsymbol{\phi}\boldsymbol{\phi}} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} \\
&\leq \sum_{\boldsymbol{\rho}\in R} \Delta_{\boldsymbol{\rho}(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R\cup\{\boldsymbol{\rho}_i\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} \\
&= \sum_{\boldsymbol{\rho}\in P} (\delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&< \sum_{\boldsymbol{\rho}\in P} (\delta_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&= \sum_{\boldsymbol{\rho}\in P} (\delta'_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})), \qquad (4.16)
\end{aligned}
$$

which implies (I2) still holds. If, on the other hand, $\boldsymbol{\rho}_i$ and $\boldsymbol{\phi}$ are in distinct cycles of π, let $\pi_1 = (\boldsymbol{\phi}\ \pi(\boldsymbol{\phi})\ \ldots\ \pi^{-1}(\boldsymbol{\phi}))$ be the cycle of π containing $\boldsymbol{\phi}$. We use the same notation as before except now we let $R = \{\boldsymbol{\lambda} \mid \boldsymbol{\lambda} \in \pi_1\}$ and let $\tau \in S(P)\setminus\{\mathrm{id}_P\}$ be the permutation

$$
\tau(\boldsymbol{\rho}) = \begin{cases} \boldsymbol{\rho}, & \text{if } \boldsymbol{\rho} \in R \\ \pi(\boldsymbol{\rho}), & \text{if } \boldsymbol{\rho} \notin R \end{cases}
$$

Then, by the same arguments as before, we have

$$
\begin{aligned}
&\sum_{\boldsymbol{\rho}\in P} (\delta'_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta'_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} + \Delta'_{\boldsymbol{\rho}_i\pi(\boldsymbol{\rho}_i)} - \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta'_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - d - \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in R\setminus\{\boldsymbol{\phi}\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} - d - \Delta_{\boldsymbol{\phi}\pi(\boldsymbol{\phi})} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})} \\
&= \sum_{\boldsymbol{\rho}\in R\setminus\{\boldsymbol{\phi}\}} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} - \Delta_{\boldsymbol{\rho}_i\boldsymbol{\phi}} + \Delta_{\boldsymbol{\phi}\boldsymbol{\phi}} + \Delta_{\boldsymbol{\rho}_i\pi(\boldsymbol{\phi})}
\end{aligned}
$$

$$
\begin{aligned}
&\leq \sum_{\boldsymbol{\rho}\in R} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} + \sum_{\boldsymbol{\rho}\notin R} \Delta_{\boldsymbol{\rho}\pi(\boldsymbol{\rho})} \\
&= \sum_{\boldsymbol{\rho}\in P} \Delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} \\
&= \sum_{\boldsymbol{\rho}\in P} (\delta_{\boldsymbol{\rho}\tau(\boldsymbol{\rho})} + \deg_{k-1}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&< \sum_{\boldsymbol{\rho}\in P} (\delta_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})) \\
&= \sum_{\boldsymbol{\rho}\in P} (\delta'_{\boldsymbol{\rho}\boldsymbol{\rho}} + \deg_{\boldsymbol{k}}(\boldsymbol{y}^{\boldsymbol{\rho}})), \qquad (4.17)
\end{aligned}
$$

so again, (I2) still holds after the replacement of $g_{\boldsymbol{\rho}_i}$ in steps 9 and 10. At last, we have shown that in any case the condition (I2) – if it holds before an iteration of the `while` loop – will still hold after executing steps 5 through 10. Thus (I2) is an invariant of Algorithm 4.6.3. □

Proof: [Proof of Proposition 4.6.5.] Let $g_{\boldsymbol{\rho}_s} = \sum_{\boldsymbol{\rho}\in P} a_{\boldsymbol{\rho}_s\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}$ and $g'_{\boldsymbol{\rho}_s} = \sum_{\boldsymbol{\rho}\in P} a'_{\boldsymbol{\rho}_s\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}$, with $1 \leq s \leq p$ and each $a_{\boldsymbol{\rho}_s\boldsymbol{\rho}}, a'_{\boldsymbol{\rho}_s\boldsymbol{\rho}} \in \mathbb{F}[x]$, represent the basis G in Algorithm 4.6.3 before and after, respectively, a run through the `while` loop (steps 5–10) during the i^{th} iteration of the `for` loop. We treat two similar cases now, depending on the outcome of step 6 (i.e. the sign of d).

Case I.
Suppose $d = a_{\boldsymbol{\rho}_i\boldsymbol{\phi}} - a_{\boldsymbol{\phi}\boldsymbol{\phi}} \geq 0$ in step 5. Then by step 7,

$$
a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}} = a_{\boldsymbol{\rho}_i\boldsymbol{\rho}} - cx^d a_{\boldsymbol{\phi}\boldsymbol{\rho}} \qquad (4.18)
$$

for all $\boldsymbol{\rho} \in P$. Taking $\pi \in S(P)$ to be the transposition of $\boldsymbol{\phi}$ and $\boldsymbol{\rho}_i$, (I2) reduces to $\deg(a_{\boldsymbol{\phi}\boldsymbol{\phi}}) + \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) > \deg(a_{\boldsymbol{\phi}\boldsymbol{\rho}_i}) + \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}})$. Equivalently, $\deg(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) > \deg(a_{\boldsymbol{\phi}\boldsymbol{\rho}_i}) + \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}) - \deg(a_{\boldsymbol{\phi}\boldsymbol{\phi}}) = \deg(a_{\boldsymbol{\phi}\boldsymbol{\rho}_i}) + d$, which together with (4.18) implies

$$
\deg(a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) = \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}). \qquad (4.19)
$$

By the definition of d, for each $\boldsymbol{\rho} \in P$ we have

$$
\begin{aligned}
\deg_{\boldsymbol{k}}(x^d a_{\boldsymbol{\phi}\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) &= \deg_{\boldsymbol{k}}(a_{\boldsymbol{\phi}\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) + d \\
&= \deg_{\boldsymbol{k}}(a_{\boldsymbol{\phi}\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) + \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}) - \deg_{\boldsymbol{k}}(a_{\boldsymbol{\phi}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}) \\
&\leq \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}})
\end{aligned}
$$

with strict inequality when $\boldsymbol{\rho} \succ \boldsymbol{\phi}$ since $a_{\boldsymbol{\phi}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}$, which contains the leading term of $g_{\boldsymbol{\phi}}$, must have no less weighted degree than $a_{\boldsymbol{\phi}\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}$ and equality is only possible if the later polynomial's $\boldsymbol{y}$-part is smaller with respect to $<_{k-1}$. Also, $\deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) \leq \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}})$ (since the later contains the leading term of $g_{\boldsymbol{\rho}_i}$) with strict inequality again holding for $\boldsymbol{\rho} \succ \boldsymbol{\phi}$. Thus, by (4.18) we obtain

$$
\deg_{\boldsymbol{k}}(a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) \leq \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}) \qquad (4.20)
$$

for all $\boldsymbol{\rho} \in P$. Moreover the inequality is strict when $\boldsymbol{\rho} \succeq \boldsymbol{\phi}$ due to the strictness of the previous inequalities and the choice of c and d in step 5.

Case II.
Now suppose $d = a_{\boldsymbol{\rho}_i\boldsymbol{\phi}} - a_{\boldsymbol{\phi}\boldsymbol{\phi}} < 0$. Then by steps 9 and 10,

$$a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}} = a_{\boldsymbol{\phi}\boldsymbol{\rho}} - c^{-1}x^{-d}a_{\boldsymbol{\rho}_i\boldsymbol{\rho}} \tag{4.21}$$

for all $\boldsymbol{\rho} \in P$. By applying the transposition of $\boldsymbol{\rho}_i$ and $\boldsymbol{\phi}$ to (I2) again we obtain $\deg(a_{\boldsymbol{\phi}\boldsymbol{\rho}_i}) < \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) + \deg(a_{\boldsymbol{\phi}\boldsymbol{\phi}}) - \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}) = \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) - d$, which together with (4.21) implies

$$\deg(a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) = \deg(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}) - d. \tag{4.22}$$

By the definition of d, for each $\boldsymbol{\rho} \in P$ we have

$$\begin{aligned} \deg_{\boldsymbol{k}}(x^{-d}a_{\boldsymbol{\rho}_i\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) &= \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) - d \\ &\leq \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}) - d \end{aligned}$$

and

$$\begin{aligned} \deg_{\boldsymbol{k}}(a_{\boldsymbol{\phi}\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) &= \deg_{\boldsymbol{k}}(a_{\boldsymbol{\phi}\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) - \deg_{\boldsymbol{k}}(a_{\boldsymbol{\phi}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}) + \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}) - d \\ &\leq \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\phi}}) - d \end{aligned}$$

with strict inequalities holding when $\boldsymbol{\rho} \succ \boldsymbol{\phi}$. Thus, by (4.21) we obtain

$$\deg_{\boldsymbol{k}}(a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}}\boldsymbol{y}^{\boldsymbol{\rho}}) \leq \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}) - d \tag{4.23}$$

for all $\boldsymbol{\rho} \in P$. Moreover the inequality is strict when $\boldsymbol{\rho} \succeq \boldsymbol{\phi}$ due to the strictness of the previous inequalities and the choice of c and d in step 5.

Now by (4.20) and (4.23), since $\deg_{\boldsymbol{k}}(\mathrm{lt}(g_{\boldsymbol{\rho}_i})) = \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\phi}}y^{\boldsymbol{\phi}})$, we see that

$$\deg_{\boldsymbol{k}}(\mathrm{lt}(g'_{\boldsymbol{\rho}_i})) - \deg_{\boldsymbol{k}}(a'_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}y^{\boldsymbol{\rho}_i}) \leq \deg_{\boldsymbol{k}}(\mathrm{lt}(g_{\boldsymbol{\rho}_i})) - \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i}y^{\boldsymbol{\rho}_i})$$

with equality holding only if $\deg_{\boldsymbol{y}}(\mathrm{lt}(g'_{\boldsymbol{\rho}_i})) \prec \deg_{\boldsymbol{y}}(\mathrm{lt}(g_{\boldsymbol{\rho}_i})) = \boldsymbol{\phi}$. Thus, following each iteration of the `while` loop, the difference between the weighted degrees of the leading term of $g_{\boldsymbol{\rho}_i}$ and its maximum term with $\boldsymbol{y}$-part $\boldsymbol{y}^{\boldsymbol{\rho}_i}$ either strictly decreases or the difference remains the same but the $\boldsymbol{y}$-part of the leading term strictly decreases with respect to $<_{k-1}$. □

Remark 4.6.7 *Proposition 4.6.5 says that after each basis update in the* `while` *loop of Algorithm 4.6.3, the gap between the weighted degrees of the leading term of $g_{\boldsymbol{\rho}_i}$ and its diagonal term is reduced with each iteration; either it strictly decreases or the value does not change but the leading term's $\boldsymbol{y}$-degree goes down which can only happen so many times before the value must decrease (at which point the $\boldsymbol{y}$-degree may jump up again). So every step in this algorithm makes progress toward the goal of attaining a diagonal basis.*

Now, to bound Algorithm 4.6.3's complexity we start with a bound on the number of times step 4 loops during the i^{th} iteration of step 3. Here, for all $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_w) \in P$ we define $\rho_0 = \max\{0, m - |\boldsymbol{\rho}|\}$ to be the exponent of η in the generator $g_{\boldsymbol{\rho}} = \eta^{\rho_0} \prod_{t=1}^{w} (y_t - h_{\sigma^t(\boldsymbol{r})})^{\rho_t}$. Again, we use the notation $g_{\boldsymbol{\rho}} = \sum_{\boldsymbol{\phi} \in P} a_{\boldsymbol{\rho}\boldsymbol{\phi}} \boldsymbol{y}^{\boldsymbol{\phi}}$, $a_{\boldsymbol{\rho}\boldsymbol{\phi}} \in \mathbb{F}[x]$, to represent each generator in terms of the standard $\mathbb{F}[x]$-basis $\{\boldsymbol{y}^{\boldsymbol{\rho}} \mid \boldsymbol{\rho} \in P\}$ of $\mathbb{F}[x, \boldsymbol{y}]_\ell$. At the start of the i^{th} iteration of step 3, the difference between the weighted degree of $g_{\boldsymbol{\rho}_i}$ and that of its target term is

$$\begin{aligned}
& \deg_{\boldsymbol{k}}(\mathrm{lt}(g_{\boldsymbol{\rho}_i})) - \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}_i\boldsymbol{\rho}_i} \boldsymbol{y}^{\boldsymbol{\rho}_i}) \\
= \; & \deg_{\boldsymbol{k}}(x^{(n-1)\rho_{i,0}} \prod_{t=1}^{w} \mathrm{lt}(y_t - h_{\sigma^t(\boldsymbol{r})})^{\rho_{i,t}}) - \deg_{\boldsymbol{k}}(x^{(n-1)\rho_{i,0}} \boldsymbol{y}^{\boldsymbol{\rho}_i}) \\
\leq \; & [(n-1)\rho_{i,0} + (n-1)|\boldsymbol{\rho}_i|] - [(n-1)\rho_{i,0} + (k-1)|\boldsymbol{\rho}_i|] \\
= \; & (n-k)|\boldsymbol{\rho}_i|. \qquad (4.24)
\end{aligned}$$

By Proposition 4.6.5 this difference either strictly decreases or the $\boldsymbol{y}$-degree of $g_{\boldsymbol{\rho}_i}$'s leading term strictly decreases. Because none of the generators have $\boldsymbol{y}$-degree greater than $\boldsymbol{\rho}_i$ up through the i^{th} iteration, the leading term can jump at most $i-1$ times before the difference above must strictly decrease. Since the terms in $g_{\boldsymbol{\rho}_i}$ with $\boldsymbol{y}$-degree $\boldsymbol{y}^{\boldsymbol{\rho}_i}$ are maximal amongst all terms of equal $\boldsymbol{y}$-degree in $g_{\boldsymbol{\rho}_i}$ (at least through the i^{th} iteration), the i^{th} iteration halts as soon as the aforementioned difference becomes zero. Therefore, there are at most $(n-k)|\boldsymbol{\rho}_i|(i-1)$ iterations of the `while` loop during the i^{th} iteration of the `for` loop.

The $\boldsymbol{k}$-weighted degree of any generator $g_{\boldsymbol{\rho}}$ with $\boldsymbol{\rho} \prec \boldsymbol{\rho}_i$ is at most $(n-1)|\boldsymbol{\rho}_i|$ and the $\boldsymbol{y}$-degree is at most $\boldsymbol{\rho}_i$ through the i^{th} iteration of step 3. Therefore, the number of possible terms in $g_{\boldsymbol{\rho}}$ is

$$\begin{aligned}
\sum_{\boldsymbol{\phi} \preceq \boldsymbol{\rho}_i} [(n-1)|\boldsymbol{\rho}_i| - (k-1)|\boldsymbol{\phi}| + 1] &= i(n-1)|\boldsymbol{\rho}_i| + i - (k-1) \sum_{\boldsymbol{\phi} \preceq \boldsymbol{\rho}_i} |\boldsymbol{\phi}| \\
&\leq i((n-1)|\boldsymbol{\rho}_i| + 1) \qquad (4.25)
\end{aligned}$$

The number of arithmetic operations in $\mathbb{F}$ needed for each iteration of step 4 is then at most $2i((n-1)|\boldsymbol{\rho}_i| + 1) + 1$. There is 1 division in step 5 to find $c = \frac{\mathrm{lc}(g_{\rho_i})}{\mathrm{lc}(g_{\boldsymbol{\phi}})}$ (we count this operation for the inversion of c in step 10, too, since we could just wait until after the test of the `if` statement to compute c or multiply $g_{\boldsymbol{\rho}_i}$ by c in step 10 instead of spending an extra inversion operation in the `else` block, but we chose to present the algorithm as it is for symmetry reasons and because the alternatives make little difference.) There are then $i((n-1)|\boldsymbol{\rho}_i| + 1)$ multiplications needed in step 7 or 10 to multiply $g_{\boldsymbol{\phi}}$ by the constant $c^{\pm 1}$ and $i((n-1)|\boldsymbol{\rho}_i| + 1)$ subtractions in the same step to find the difference of $g_{\boldsymbol{\rho}_i}$ and $c^{\pm 1} x^{\pm d} g_{\boldsymbol{\phi}}$ (multiplying $g_{\boldsymbol{\phi}}$ by $x^{\pm d}$ only requires shifting the coefficients (or their degrees) in $g_{\boldsymbol{\phi}}$ so it incurs no arithmetic cost). It is now a simple matter to calculate the cost of the Gröbner Basis algorithm.

Proposition 4.6.8 *Algorithm 4.6.3 requires $\mathcal{O}(n^2\ell^2(e(\ell+w)/w)^{3w})$ arithmetic operations in $\mathbb{F}$ to run given the basis G' of Corollary 4.5.4.*

Proof: Since the only arithmetic operations occur inside the `while` loop, which may loop up to $(n-k)|\boldsymbol{\rho}_i|(i-1)$ times during the i^{th} iteration of the `for` loop, the whole algorithm takes no more than

$$\begin{aligned}
&\sum_{i=2}^{p}(n-k)|\boldsymbol{\rho}_i|(i-1)\cdot(2i((n-1)|\boldsymbol{\rho}_i|+1)+1)\\
\leq\ & (n-k)\sum_{i=2}^{p}\ell[i^2(2(n-1)\ell+2)-i(2(n-1)\ell+2)-1]\\
<\ & (n-k)\ell\left[\frac{p(p+1)(2p+1)}{6}(2(n-1)\ell+2)-\frac{p(p+1)}{2}(2(n-1)\ell+2)-p\right]\\
=\ & (n-k)\ell\left[\frac{p(p+1)(p-1)}{3}(2(n-1)\ell+2)-p\right]\\
\in\ & \mathcal{O}(n^2\ell^2p^3)=\mathcal{O}\left(n^2\ell^2\binom{w+\ell}{\ell}^3\right) \qquad (4.26)
\end{aligned}$$

arithmetic operations in $\mathbb{F}$ to execute. $\square$

In the bivariate case ($w=1$), the complexity bound of Proposition 4.6.8 becomes $\mathcal{O}(n^2\ell^5)$, which is slightly worse than the bound of $\mathcal{O}(n^2m\ell^4)$ achieved in [24]. The reason for this discrepancy is the basis G' is not quite a natural extension of the basis used by Lee and O'Sullivan in [24]. The basis they use in handling the bivariate case is $\{(y_1-h_r)^i\eta^{m-i}, y_1^{i'-m}(y_1-h_r)^m \mid 0\leq i\leq m, m<i'\leq\ell_1\}$, which is equivalent to the basis G in equation (4.5). So G is an extension of their basis, but recall that G was not an echelon basis (Remark 4.6.2) and in general had more generators than G'. So G' reduced the size of our basis, but then the weighted degrees of the generators grew to be larger than optimal. Fortunately, we can find an echelon basis to start with that has the same number of generators as G' and smaller weighted degrees like G, so we get the best of both worlds. When we start with this basis instead of G' the bound in the previous proposition can be reduced to $\mathcal{O}\left(n^2m\ell(e(\ell+w)/w)^{3w}\right)$. We obtain this basis by choosing some $\boldsymbol{\rho}'\in\mathbb{N}_0^w$ for each $\boldsymbol{\rho}\in P$ such that $\rho'_t\leq\rho_t$ for $1\leq t\leq w$ and $\sum_{t=1}^{w}\rho'_t=\min\{m,\sum_{t=1}^{w}\rho_t\}$ and defining $\rho'_0=\rho_0=\max\{0,m-|\boldsymbol{\rho}|\}$. Then, as a corollary to Corollary 4.5.4 we can take for a basis of $I_{\boldsymbol{r},m,\ell}$ the set

$$G''=\left\{g'_{\boldsymbol{\rho}}:=\eta^{\rho'_0}\prod_{t=1}^{w}[(y_t-h_{\sigma^{t-1}(\boldsymbol{r})})^{\rho'_t}y_t^{\rho_t-\rho'_t}\ \middle|\ \boldsymbol{\rho}\in P\right\}. \qquad (4.27)$$

Corollary 4.6.9 *The set G'' in equation (4.27) is an echelon basis of the $\mathbb{F}[x]$-submodule $I_{\boldsymbol{r},m,\ell}\subseteq\mathbb{F}[x,\boldsymbol{y}]_\ell$.*

Proof: For each $\boldsymbol{\rho}\in P$, by our choice of $\boldsymbol{\rho}'$ we may take g' and g in equation (4.7) to be $g_{\boldsymbol{\rho}}\in G'$ and $g'_{\boldsymbol{\rho}}\in G''\subseteq G$, respectively. Then we see that $g'_{\boldsymbol{\rho}}$ is an $\mathbb{F}[x]$-linear combination of generators $g_{\boldsymbol{\rho}}$ and generators of G with smaller $\boldsymbol{y}$-degrees, which are in turn linear combinations of generators in G' (also, necessarily, having smaller $\boldsymbol{y}$-degrees than $g_{\boldsymbol{\rho}}$ and $g'_{\boldsymbol{\rho}}$) by Corollary 4.5.4. Since the coefficients of g and g' in

equation (4.7) are units, we may interchange them. Thus, swapping each $g'_{\boldsymbol{\rho}}$ for $g_{\boldsymbol{\rho}}$ in G' yields a new basis for $I_{\boldsymbol{r},m,\ell}$. This new basis, G'', is an echelon basis because G' is and the generators $g'_{\boldsymbol{\rho}}$ and $g_{\boldsymbol{\rho}}$ have equal $\boldsymbol{y}$-degrees, preserving the property. □

At the beginning of the chapter, we introduced the basis G, which was convenient for showing how the module $I_{\boldsymbol{r},m,\ell}$ is generated. But G had too many generators, so we found new basis G' that reduced the size of G but at a price of replacing some generators by polynomials with larger weighted degree. But now with G'', we have preserved the size of G' and regained the smaller degrees of G. Corollary 4.6.9 showed that G'' is obtained from G' of (4.6) by switching back each generator $g_{\boldsymbol{\rho}}$ with $|\boldsymbol{\rho}| > m$ to a generator $g_{\boldsymbol{\rho}'}$ from G of (4.5) with a unit coefficient in the basis expansion of equation (4.7). Using the notation $g'_{\boldsymbol{\rho}} = \sum_{\boldsymbol{\phi}\in P} a'_{\boldsymbol{\rho}\boldsymbol{\phi}}\boldsymbol{y}^{\boldsymbol{\phi}}$ this generator has smaller weighted degree than $g_{\boldsymbol{\rho}}$ so we will now have that at the start of the i^{th} iteration of step 3, the difference between the weighted degree of $g'_{\boldsymbol{\rho}_i}$ and that of its target term is

$$\begin{aligned}
& \deg_{\boldsymbol{k}}(\mathrm{lt}(g'_{\boldsymbol{\rho}_i})) - \deg_{\boldsymbol{k}}(a_{\boldsymbol{\rho}'_i\boldsymbol{\rho}_i}\boldsymbol{y}^{\boldsymbol{\rho}_i}) \\
= \;& \deg_{\boldsymbol{k}}(x^{(n-1)\rho'_{i,0}}\prod_{t=1}^{w}\mathrm{lt}(y_t - h_{\sigma^t(\boldsymbol{r})})^{\rho'_{i,t}}y_t^{\rho_t-\rho'_t}) - \deg_{\boldsymbol{k}}(x^{(n-1)\rho'_{i,0}}\boldsymbol{y}^{\boldsymbol{\rho}_i}) \\
\leq \;& [(n-1)\rho'_{i,0} + (n-1)|\boldsymbol{\rho}'_i| + (k-1)(|\boldsymbol{\rho}_i| - |\boldsymbol{\rho}'_i|)] - [(n-1)\rho'_{i,0} + (k-1)|\boldsymbol{\rho}_i|] \\
= \;& (n-k)|\boldsymbol{\rho}'_i| \\
\leq \;& (n-k)m. \qquad (4.28)
\end{aligned}$$

It follows that there are at most $(n-k)m(i-1)$ iterations of the `while` loop during the i^{th} iteration of the `for` loop. The $\boldsymbol{k}$-weighted degree of any generator $g'_{\boldsymbol{\rho}}$ with $\boldsymbol{\rho} \prec \boldsymbol{\rho}_i$ is at most $(n-1)m+(k-1)(|\boldsymbol{\rho}_i|-m)$ and the $\boldsymbol{y}$-degree is at most $\boldsymbol{\rho}_i$ through the i^{th} iteration of step 3. Therefore, the number of possible terms in $g'_{\boldsymbol{\rho}}$ is

$$\begin{aligned}
& \sum_{\boldsymbol{\phi}\preceq\boldsymbol{\rho}_i}[(n-1)m + (k-1)(|\boldsymbol{\rho}_i| - m - |\boldsymbol{\phi}|) + 1] \\
= \;& i(n-1)m + i + (k-1)\sum_{\boldsymbol{\phi}\preceq\boldsymbol{\rho}_i}(|\boldsymbol{\rho}_i| - m - |\boldsymbol{\phi}|) \\
\leq \;& i((n-1)|\boldsymbol{\rho}_i| + 1) \qquad (4.29)
\end{aligned}$$

The number of arithmetic operations in $\mathbb{F}$ needed for each iteration of step 4 is hence again no more than $2i((n-1)|\boldsymbol{\rho}_i| + 1) + 1$. With these bounds on the number of terms in each generator and the initial gap between the weighted degrees of the leading term and target term of each generator, we get the following improved bound on the complexity of the algorithm by doing, otherwise, the same analysis as for Proposition 4.6.8.

Proposition 4.6.10 *Algorithm 4.6.3 requires $\mathcal{O}(n^2m\ell^{3w+1}e^{3w})$ arithmetic operations in $\mathbb{F}$ to run given the basis G'' of equation (4.27).*

Proof: Similar to the proof of Proposition 4.6.8, we have that the whole algorithm takes no more than

$$\begin{aligned}
& \sum_{i=2}^{p}(n-k)m(i-1)\cdot(2i((n-1)|\boldsymbol{\rho}_i|+1)+1) \\
\leq\ & (n-k)m\sum_{i=2}^{p}[i^2(2(n-1)\ell+2)-i(2(n-1)\ell+2)-1] \\
<\ & (n-k)m\left[\frac{p(p+1)(2p+1)}{6}(2(n-1)\ell+2)-\frac{p(p+1)}{2}(2(n-1)\ell+2)-p\right] \\
=\ & (n-k)m\left[\frac{p(p+1)(p-1)}{3}(2(n-1)\ell+2)-p\right] \\
\in\ & \mathcal{O}(n^2m\ell p^3) \qquad (4.30)
\end{aligned}$$

arithmetic operations in $\mathbb{F}$. Eliminating p from equation (4.30), we get

$$\mathcal{O}(n^2m\ell p^3)=\mathcal{O}\left(n^2m\ell\binom{w+\ell}{\ell}^3\right)=\mathcal{O}(n^2m\ell(e(\ell/w+1))^{3w})\subseteq\mathcal{O}(n^2m\ell^{3w+1}e^{3w})$$

using the fact that $\binom{w+\ell}{w}\leq\left(\frac{e(w+\ell)}{w}\right)^w$ by Stirling's approximation (see [7, eq. C.5]). □

Remark 4.6.11 *By Proposition 4.6.10, our Algorithm 4.6.3 can now be viewed as an efficient generalization of Algorithm G in [24] to the multivariate case. In their paper, Lee and O'Sullivan show that Algorithm G has a complexity of $\mathcal{O}(n^2m\ell^4)$ multiplications in $\mathbb{F}$, where n is the code length, m is the multiplicity requirement for each interpolation point, and ℓ is the y-degree bound (there is only one y-variable in their case) of the minimal polynomial of $I_{\boldsymbol{r},m}\subseteq\mathbb{F}[x,y]$. Our algorithm reduces to their algorithm in the case where $w=1$, for then our complexity is just $\mathcal{O}(n^2m\ell^4)$ like theirs and the basis G'' we used is analogous the the basis Lee and O'Sullivan described. The complexity of Algorithm 4.6.3 remains low when $w=1$. However, increasing the number of variables or folds in the code has the benefit of yielding a family of codes whose error rates asymptotically achieve the capacity of the channel. This benefit is computationally expensive but theoretically exciting.*

4.7 Gröbner Basis-Based Multivariate List Decoding Algorithm

We now combine the Algorithms 4.6.3 and 4.4.4 to produce a more explicit formulation of Algorithm 4.3.8 for list decoding $\mathrm{FRS}_q(n,k,v,\boldsymbol{\alpha})$.

Algorithm 4.7.1 (Gröbner Basis-Based Multivariate List Decoding)
Input: received word $\boldsymbol{r}=(r_1,\ldots,r_n)$, error threshold $\tau\in\mathbb{N}_0$ satisfying equation (4.2), number of variables $w+1$, and multiplicity value m
Output: list of codewords, $\mathcal{L}_{\tau,\boldsymbol{r}}\subseteq\mathrm{FRS}_q(n,k,v,\boldsymbol{\alpha})$, within Hamming distance τ of $\boldsymbol{r}$

G1. Compute the weighted degree bound $d = \left\lfloor \sqrt[w+1]{(k-1)^w n (m+w)_{w+1}} \right\rfloor + 1$ and the y-degree bound $\ell = \left\lfloor \frac{d}{k-1} \right\rfloor$ for Q, and the number of generators of $I_{\boldsymbol{r},m,\ell}$, $p = \binom{w+\ell}{w}$.

G2. Enumerate $\{\boldsymbol{\rho}_1, \ldots, \boldsymbol{\rho}_p\} = \{\boldsymbol{\rho} \in \mathbb{N}_0^w \mid |\boldsymbol{\rho}| \leq \ell\} =: P$.

G3. For each $\boldsymbol{\rho} \in P$,

(a) if $|\boldsymbol{\rho}| \leq m$, compute $g_{\boldsymbol{\rho}} = \eta^{\rho_0} \prod_{j=1}^{w} (y_j - h_{\sigma^{j-1}(\boldsymbol{r})})^{\rho_j}$ where $\rho_0 = m - |\boldsymbol{\rho}|$,

(b) if $|\boldsymbol{\rho}| > m$, compute $g_{\boldsymbol{\rho}} = g_{\boldsymbol{\rho}'} \boldsymbol{y}^{\boldsymbol{\rho} - \boldsymbol{\rho}'}$ where $\boldsymbol{\rho}'$ is any $\boldsymbol{\rho}' \in P$ such that $|\boldsymbol{\rho}'| = m$ and $\rho_i' \leq \rho_i$ for $1 \leq i \leq w$.

G4. Set $G = \texttt{GröbnerBasis}(w, \ell, \{g_{\boldsymbol{\rho}_1}, \ldots, g_{\boldsymbol{\rho}_p}\})$ *(Algorithm 4.6.3)*.

G5. Let Q be the minimal element of G with respect to $<_{k-1}$.

G6. Let $L = \phi$ be any empty list. Call $\texttt{FoldedRootFind}(Q, k, 0, x^{q-1} - \alpha)$ *(Algorithm 4.4.4)* and for each polynomial $f(x)$ that it returns, if $\text{dist}(\text{ev}_{\boldsymbol{\alpha}}(f), \boldsymbol{r}) \leq \tau$, then add the codeword $\text{ev}_{\boldsymbol{\alpha}}(f)$ to L.

G7. Return the list L.

4.7.1 Final Complexity Analysis

Most of the complexity analysis for this algorithm has been done in Sections 2.3 and 4.6.2. For our analysis, we again only worry about operations in $\mathbb{F}_q$. Step G1 is real arithmetic so we do not need to worry about its cost. Step G2 also requires no field operations. Step G3 requires some new analysis, but since step G4 will be the main factor in complexity, we do not worry about cutting every corner to save operations in G3.

We assess step G3 by first viewing the generators as polynomials in $y_1, \ldots, y_w$ over $\mathbb{F}[x]$; as such, none of the generators has more than $T := \binom{m+w}{w}$ terms. Start by computing $g_{\boldsymbol{\rho}_1} = \eta^m$. This can be done with less than m multiplications in $\mathbb{F}[x]$. Next we iteratively compute the other generators in G3(a) in order by $\boldsymbol{\rho}$ according to $\prec_{k-1}$. For $|\boldsymbol{\rho}| \neq 0$, we find $g_{\boldsymbol{\rho}}$ by multiplying $g_{\boldsymbol{\rho}'}/\eta$ (note: η will divide $g_{\boldsymbol{\rho}'}$ because $|\boldsymbol{\rho}'| < m$) by $(y_i - h_{\sigma^{i-1}(\boldsymbol{r})})$ where $i \in [w]$ is such that $\rho_i > 0$ and letting $\boldsymbol{\rho}' = \boldsymbol{\rho} - e_i$ (here e_i is the i^{th} standard basis vector of $\mathbb{Z}^w$). This can be done with at most T apiece of divisions, multiplications, and subtractions in $\mathbb{F}[x]$. Since there are T generators accounted for in G3(a) we need at most T^2 multiplications in $\mathbb{F}[x]$ to perform the cumulative course of step G3(a). Step G3(b), on the other hand, can be conducted by shifting the degrees of the generators computed in G3(a) appropriately and so it requires no operations in $\mathbb{F}[x]$. Since no monomial (in $x, y_1, \ldots, y_w$) of the basis we produce has x-degree larger than mn, the polynomials we operate on in $\mathbb{F}[x]$ can be multiplied using $\mathcal{O}(mn \log(mn) \log\log(mn))$ multiplications in $\mathbb{F}$ ([38, Theorem

8.23]). Divisions and subtractions in $\mathbb{F}[x]$ can be done with the same efficiency, so we have that step G3 requires at most

$$\mathcal{O}(mn \log(mn) \log\log(mn) T^2) = \mathcal{O}(nm^{w+1})$$

operations in $\mathbb{F}_q$.

By Proposition 4.6.10, step G4 can be done using $\mathcal{O}(n^2 m \ell^{3w+1} e^{3w})$ field operations. Step G5 requires no field operations. In G6, the y-degree of R is at most ℓq^{w-1} and its x degree is no more than mn. Therefore, by Proposition 2.3.12, the call to `FoldedRootFind` will require no more than $\mathcal{O}(k^3 \ell^3 q^{3w-3})$ field operations if $w > 1$ and if $w = 1$ then it will take only $\mathcal{O}(nm\ell^2)$ by Corollary 2.3.8. We know that FoldedRootFind returns no more than $\mathcal{O}(\ell q^{w-1})$ folded roots of Q so using the fast multi-point evaluation of [38, Algorithm 10.7], the second part of G6 can be done using $\mathcal{O}(\ell q^{w-1} n \log n \log\log n)$ operations in $\mathbb{F}_q$.

Combining the complexity bounds for each of the steps in Algorithm 4.7.1 yields the following theorem.

Theorem 4.7.2 *Let $\mathcal{C}$ be the folded Reed-Solomon code $\mathrm{FRS}_q(n, k, v, \boldsymbol{\alpha})$. Then for any $\boldsymbol{r} \in \mathbb{F}_q^n$ and any $m, w \in \mathbb{N}$, Algorithm 4.7.1 outputs the list of codewords in $\mathcal{C}$ within Hamming distance τ of $\boldsymbol{r}$ for $\tau \leq \frac{n}{v} - \left\lfloor \frac{n}{v-w+1} \sqrt[w+1]{\frac{(k-1)^w (m+w)_w}{m^w n^w}} \right\rfloor - 2$ as long as the $\deg_{y_i}(Q) < q$ for $1 \leq i \leq w$ for the polynomial Q produced by step G5, which holds if $w + m < \frac{q - 1/(k-1)}{\sqrt[w+1]{n/(k-1)}}$. The number of operations in $\mathbb{F}_q$ needed by the algorithm is*

$$\mathcal{O}(n^2 m \ell^{3w+1} e^{3w} + (k \ell q^{w-1})^3). \tag{4.31}$$

Proof: The complexity of Algorithm 4.7.1 follows from the preceding analysis. That the algorithm works correctly (when the specified conditions are met) follows from Proposition 4.3.7 and Corollary 4.4.3 and the proofs of correctness for Algorithms 4.3.8, 4.6.3, and 4.4.4. That $\deg_{y_i}(Q) < q$ for $1 \leq i \leq w$ when $w + m < \frac{q - 1/(k-1)}{\sqrt[w+1]{n/(k-1)}}$ follows because

$$\begin{aligned}
deg_{y_i}(Q) &\leq \ell \\
&\leq \frac{\sqrt[w+1]{(k-1)^w n (m+w)_{w+1}} + 1}{k-1} \\
&\leq \sqrt[w+1]{\frac{(k-1)^w n}{(k-1)^{w+1}} \cdot (m+w)^{m+1}} + 1/(k-1) \\
&\leq \sqrt[w+1]{\frac{n}{k-1}} \cdot (m+w) + 1/(k-1) \\
&< \sqrt[w+1]{\frac{n}{k-1}} \cdot \frac{q - 1/(k-1)}{\sqrt[w+1]{n/(k-1)}} + 1/(k-1) \\
&= q.
\end{aligned}$$

□

Remark 4.7.3 *If the complexity of the call to* `FoldedRootFind` *in step G6 could be better bounded, then the complexity of Algorithm 4.7.1 can be simplified to* $\mathcal{O}(n^2 m \ell^{3w+1} e^{3w})$*, which is the asymptotic cost of the interpolation step. It seems likely that this is possible, as in practice the run-time of the root-finding step always seems to be far less than that of the interpolation step.*

Remark 4.7.4 (Berlekamp-Massey Algorithm Comparison) *In [24], Lee and O'Sullivan showed that their bivariate Gröbner basis-based interpolation algorithm is equivalent to the Berlekamp-Massey algorithm in the case where the multiplicities* m *and the list-size bound* ℓ *are both set to 1. Our multivariate interpolation algorithm reduces exactly to the Lee-O'Sullivan algorithm in the bivariate case (where there is no real folding and only one* y*-variable). In that case (*$m = \ell = v = w = 1$*), the complexity of our algorithm is just* $\mathcal{O}(n^2)$*. This on par with standard implementations of the Berlekamp-Massey for minimum distance decoding (which is what our list decoder does in the case* $\ell = 1$*).*

Chapter 5
Conclusion

5.1 Results

Our goals, to reiterate, were to illuminate and deepen the recent results on list decoding methods for Reed-Solomon codes and their derivatives in terms of both the theoretical and practical aspects of these methods.

We established the necessary background on error-correcting codes, Gröbner basis techniques for modules, and polynomial root-finding methods in Chapter 2. Most of these results were already known and well-established, with the exception of the modular bivariate root-finding algorithm, Algorithm 2.3.10. It was very similar to the ordinary bivariate root-finding algorithm, but it required some care to ensure that the recursively defined polynomials were congruent to zero on the appropriate part of the y-root we were seeking but that they were not uniformly zero.

In Chapter 3, we examined some of the bounds on list decoding, notably the matching upper and lower bounds for the asymptotic error rates attainable in error-correcting codes with polynomial length lists, and expounded on the progression of list decoding from Sudan's pioneering two-step algorithm to the Guruswami-Sudan algorithm which still yields the best known error rates for bivariate interpolation-based list decoding. We took care to closely examine the asymptotic error rate of Sudan's original algorithm and observed that for rates $\kappa < \frac{1}{2}$, it actually achieves error rates of approximately $1 - \sqrt{2\kappa} + \kappa/2$, which is better than the commonly stated error rate of $1 - \sqrt{2\kappa}$ for the algorithm (but still not as good as the error rate of $1 - \sqrt{\kappa}$ achieved by the more general Guruswami-Sudan algorithm).

Finally, in Chapter 4, we extended the Gröbner basis-based interpolation algorithm of Lee and O'Sullivan to many variables and from this produced an explicit implementation of the algorithm suggested by Guruswami and Rudra for multivariate list decoding of capacity achieving folded Reed-Solomon codes. The main challenges in solving this lay in determining an initial echelon basis of the submodule $I_{\boldsymbol{r},m,\ell}$ that could be efficiently managed to produce the Gröbner basis and bounding the complexity of the Gröbner basis algorithm. As a result of the complexity analysis of this algorithm, we achieve a bound of $\mathcal{O}((Nv)^2 m(m+w)(N/K)^3(e(m+w))^{3w})$ field operations for the interpolation step in w-variable list decoding the v-folded Reed-Solomon Code of length N and dimension K up to multiplicity m. This bound is the lowest known for decoding these capacity achieving codes.

An actual implementation of this algorithm for SINGULAR is provided in Appendix 5.3. The code is intended to be clear and concise so that anyone who reads this dissertation and has a little programming knowledge can experiment with the program. It requires simply cutting and pasting to run in SINGULAR, but documentation is provided to try to expedite the process of porting the program should someone desire to test it in another computer algebra environment.

5.2 Loose Ends

There are no loose ends. That is what we would like to be able to say someday, but today that is far from the truth. The diligent reader has probably already observed several unanswered questions and it is doubtless that there have been missed opportunities for new theorems. In the course of time perhaps all of these shall be cleared up. For now, here is a list of some of the issues, important or otherwise, that we feel should be addressed.

- The size of the output list from the folded root-finding step is so far bounded only by the y-degree of R, which is at most ℓq^{w-1}. Trials of the algorithm suggest that the number is actually at most ℓ, but establishing a proof of this or any other better bound is not a task that we have thoroughly pursued yet. It would be nice to have a better handle on this component because then the complexity of the Algorithm 4.7.1 could be reduced to $\mathcal{O}(n^2 m \ell^{3w+1} e^{3w})$, which is the complexity of the interpolation step. Indeed, in practice it seems clear that the main loop of the interpolation step is the bottleneck and the complexity of $\mathcal{O}(\ell q^{w-1} n \log n \log\log n))$ from the folded root-finding step should be possible to be done away with.

- The actual complexity of computing a Gröbner basis of $I_{\boldsymbol{r},m}$ is still not understood. However, SINGULAR's built-in Gröbner basis finding routine performs significantly faster at finding a Gröbner basis of $I_{\boldsymbol{r},m}$ as an $\mathbb{F}[x, \boldsymbol{y}]$-ideal than the implementation of Algorithm 4.7.1 does at finding a Gröbner basis of $I_{\boldsymbol{r},m,\ell}$ as an $\mathbb{F}[x]$-submodule. One reason for this could be that the authors of SINGULAR are better attuned to the functionality of their software and can therefore achieve faster performance than we can even though the task their function is performing may have equal or greater complexity. However, in comparison tests (on small codes) of these two methods, the number of generators of the Gröbner basis of $I_{\boldsymbol{r},m}$ was always less than (but not less than half) the number of generators in the basis of $I_{\boldsymbol{r},m,\ell}$. This indicates that a deeper investigation into the problem of just finding a Gröbner basis of $I_{\boldsymbol{r},m}$ may be quite fruitful.

- A basis reduction algorithm proposed by Koetter and Vardy in the context of algebraic soft-decision decoding of Reed-Solomon codes was recently adapted by Kwankyu Lee to his and O'Sullivan's Gröbner basis based interpolation algorithm. The algorithm works by selecting a handful of the interpolation points, interpolating small degree polynomials through these, and then using combinations of those polynomials as a basis $\{e_i\}$ of a supermodule of $I_{\boldsymbol{r},m,\ell}$ to express the initial generating set of $I_{\boldsymbol{r},m,\ell}$ from which to compute a Gröbner basis. The coefficients (polynomials in x) of the initial generating set will be smaller than they are in our method (we essentially use the standard basis $\{e_i = \boldsymbol{y}^{\boldsymbol{\rho}_i}\}$ of $\mathbb{F}[x, \boldsymbol{y}]_\ell$ to represent the generators), but then there is the added cost of computing the basis $\{e_i\}$ in the first place and the cost of converting the minimal element of the computed Gröbner basis back to its corresponding form as a polynomial in $\mathbb{F}[x, \boldsymbol{y}]$ so that its roots can be found. Lee reported

a substantial cutback in the number of field multiplications required by the algorithm using this reduced-basis technique but it has not been shown that the technique actually produces any asymptotic complexity gains.[1] Still, it would be interesting to see if there is anything to be gained in the multivariate extension of it (to our knowledge, it has so far only been implemented for the bivariate interpolation algorithm).

- The probability of error for the multivariate list decoding algorithm, Algorithm 4.3.8, which is the probability that a received word has no codeword within the multivariate list decoding radius (given by Proposition 4.3.7) should be closely analyzed and compared to the probability of error for classical minimum distance decoders, which is (generally) the probability that a received word has no codeword within distance $\left\lfloor \frac{d-1}{2} \right\rfloor$. Each of these probabilities can be upper bounded by taking into account that a sphere of radius $\left\lfloor \frac{d-1}{2} \right\rfloor$ in $\mathbb{F}_q^n$ about a point contains close to $q^{nH_q(\tau/n)}$ vectors (Corollary 3.2.3), multiplying this by the size of the code (q^k), and subtracting the result from $q^n = |\mathbb{F}_q^n|$ to see how many vectors could conceivably be outside the decoding radius of every codeword. However, this bound does not take into account the structure of Reed-Solomon codes, nor does it consider that the distance between words in a v-folded code is determined by the number of blocks of size v that the two words differ on. Also, one must bear in mind that the definition of a decoding error is different in list decoding than in minimum distance decoding. To be consistent, one should redefine a decoding error in list decoding as an instance where there is not a *unique* closest codeword to the received word in the list returned. This makes the counting task even more difficult, but it is an important question that should be analyzed fully at some point.

- It seems likely that the process of 'folding' Reed-Solomon codes can be done for any RS code and not just those whose code locators are defined as $(\alpha^1, \ldots, \alpha^n)$ for some primitive $\alpha \in \mathbb{F}_q$. Adapting the multivariate list decoding algorithm to this purpose will require care, though, as the codewords corresponding to successive polynomials in the folded roots of the interpolation polynomial will no longer be cyclic shifts of each other, but they will still be some permutation of the same fixed set of coordinates. For this to be so, however, it is still necessary that $n = q - 1$, otherwise if $n < q - 1$ the codewords may have different coordinate values (regardless of order). There may still be a way to properly formulate the multivariate list decoding problem for RS codes with $n < q-1$. Also, it would be nice to modify the folding process to GRS codes and create a multivariate list decoding algorithm for these and, consequently, folded alternant (and BCH) codes. All these tasks seem possible if done carefully, but the process of generalizing the multivariate list decoding algorithm to GRS codes will not be quite as simple as generalizing the bivariate list decoding

[1] An implementation of the basis reduction algorithm was tested by the author but it turned out to be much slower than the normal version of the algorithm that is presented herein.

algorithm because of the dependency in the multivariate case on the consecutive code evaluators being fixed multiples of each other.

5.3 Final Thoughts

We have already covered both what was done in this dissertation and what was left undone. The primary result of the project is that we have an efficient implementation of the multivariate list decoding algorithm and we know a specific upper bound on the complexity of this implementation. The algorithm we give is interesting because it takes an algebraic-geometry approach to the interpolation problem by finding the minimal polynomial of a certain ideal, yet it generalizes the classical Berlekamp-Massey algorithm, which (in its conventional form) takes a purely algebraic approach to finding the minimal polynomial of a linearly recurrent sequence. The goal in future work will be to find a faster implementation of the multivariate list decoding algorithm that improves upon our complexity bound. In the end, it is hoped that multivariate interpolation will yield practical list decoding algorithms that significantly improve the decoding radius of present block coding schemes. Our algorithm is a step forward along the path to this goal.

Appendix: Source Code

The following source code is an implementation of Algorithm 4.7.1 for the SINGULAR Computer Algebra System. It was tested in SINGULAR release version 3-0-4 and run on both the UNIX-based T-machines of the University of Kentucky's Mathematical Sciences computing facilities and in Windows Vista on a personal computer through the Cygwin terminal with similar results in both settings. The performance is fast for the bivariate interpolation problem: list decoding a length 256 Reed-Solomon code takes typically on the order of ten to twenty seconds. Raising the number of variables, however, greatly reduces the efficiency of the algorithm. Even in only three variables, list decoding a word in a length 10, twice-folded Reed-Solomon code can take nearly 10 minutes. The implementation is fairly robust. There are several libraries included that the main program relies upon. There is also an example of an execution given for those to follow who would like to test it themselves. The code is documented so hopefully, with the aid of the content of this dissertation, it will be useful and easy to follow for those who will venture to try it out.

SINGULAR Multivariate FRS List Decoding Implementation

This is the main list decoding program. It relies on three homemade libraries, FastInterpolation.lib, RootDigger, and UsefulProcs.lib, the source code for which is provided in the Section 5.3 of this appendix. These libraries should be saved to a directory in SINGULAR's SearchPath in order for the problem to function correctly. The program also uses the standard SINGULAR library inout.lib to display a counter during the execution of the main loop. To run the program, you can simply copy the entire file and paste it into the SINGULAR console. It will output a list of polynomials at the end that correspond to the codewords near the 'received' word it randomly generates (it does not check the distance of the words it finds to the received one so it may return ones further away than sought). The parameters of the code are specified at the beginning of the program. One should be able to easily modify the program to decode any folded Reed-Solomon code (although complexity becomes an issue even for moderately sized codes).

```
//--- LIBRARY FILES ------------------------------------------------------------//
// These library files below should be stored within a directory in the SINGULAR
// SearchPath (or their path names may be given explicitly as an alternative).
//------------------------------------------------------------------------------//
LIB "FastInterpolation.lib";
LIB "RootFinder.lib";
LIB "inout.lib";   // used mainly for debugging
LIB "UsefulProcs.lib";
//------------------------------------------------------------------------------//


//--- PARAMETERS ---------------------------------------------------------------//
// The first set of parameters are defined by the user.  The second set should be
// computed according to the first set (use Maple or something to do that).
//------------------------------------------------------------------------------//
```

```
// USER-DEFINED PARAMETERS
int q=11;    // base field size (of the Alternant Code)
int S=1;     // degree of the field extension
int qs=q^S;  // extension field size (of the GRS Code)
int b=1;     // b=1 unless code is not a narrow sense BCH Code
int n=10;    // length of the GRS code
int k=2;     // dimension of the GRS code
int v=2;     // fold number
int w=2;     // number of y variables (doesn't include x)
int m=2;     // multiplicity number

// COMPUTED PARAMETERS (params D, tau, & l can be found with "maple worksheet"
int L=1-b;            // multiplier exponent factor for wide-sense BCH codes
int N=n/v;            // Length of the Folded GRS Code
int K=k/v;            // Dimension of the Folded GRS Code
int D=10;             // degree bound on the minimal polynomial of the ideal Irm
int tau=1;            // error threshold bound
int l=10;             // bound on y-degree of minimal polynomial
int p=binom(w+l,w);   // # of monomials in y-degree < l (also, # of gens of Irml)

// MISCELLANEOUS VARIABLES
int d; int s; int t; int i; int j; int P; int bc; int bP;

// DEGREE WEIGHT VECTORS
//        k-weight: 1,k-1,...,k-1
//        y-degree vector: 0,1,...,1
//        x-degree vector: 1,0,...,0
intvec kdeg=1;  intvec ydeg=0;  intvec xdeg=1;
for(i=1; i<=w; i++) {  kdeg=k-1,kdeg; ydeg=1,ydeg; xdeg=1,xdeg;  }

// RING DECLARATION
// the order is weighted reverse-lexicographic (with x at end of the alphabet)
ring R1=qs, (y(1..w),x), wp(kdeg);  poly a=2;  // a=primitive element of GF(qs)
poly h=x^(q-1)-a;  // irreducible polynomial used for folded root-finding
ideal H=std(h);    // standard basis of the ideal (h(x)) to mod out by h(x) later

// DEFINING A AND V FOR THE CODE GRS(A,V)
poly B=a^1;
vector A=0;
vector V=0;
for (i=1; i<=n; i++)
{
  A=A+B^(i-1)*gen(i);
  V=V+1/(B^(-L*(i-1)))*gen(i);
}

// OTHER MISCELLANEOUS VARIABLES
intvec lexp_gs; intvec lexp_gt;
list yroots; list yreval;
poly c; poly g_st; poly g_tt; poly Q; poly R;
vector g; poly hc; vector cw; vector e; vector r; vector ylist;

timer=1; // The time of each command is printed (if > 1 sec)
int time=timer; // initialize time by timer
```

```
// COMPUTE POWERS OF eta=(x-A[1])*...*(x-A[n])
poly eta=1; for (i=1; i<=n; i++) {  eta=eta*(x-A[i]);  }
poly eta(0)=1;  for(i=1; i<=m; i++) { poly eta(i)=eta(i-1)*eta; }
time=timer-time;  time;  // yields the time in ticks-per-sec (default 1)

// ENUMERATE THE INDICES OF THE GENERATORS (ordered by y-degree)
intvec rho(0)=0;
for(i=1;i<=w;i++){ intvec rho(i)=1; }
for(i=1;i<w;i++){ for(j=0;j<=w;j++)
{
  if(i>=j) {rho(j)=0,rho(j);} else {rho(j)=rho(j),0;}}
}
P=w;
for(j=1; j<l; j++)
{
  for(i=1; i<=w; i++)
  {
    bc=binom(w-i+j,j);
    bP=binom(w+j,j)-bc-1;
    for(t=1; t<=bc; t++) {  intvec rho(P+t)=rho(bP+t) + rho(i);  }
    P=P+bc;
  }
}
// ADD ETA POWER TO LAST POSITION OF rho(i)
for(j=0; j<p; j++){  rho(j)=rho(j), max(0,m-norm(rho(j)));  }
//-------------------------------------------------------------------------------//


//--- GENERATING A RANDOM CODEWORD TO FUDGE AND DECODE -------------------------//
hc=0;  // initialize message polynomial
cw=0;  // initialize corresponding codeword
e=0;   // initialize error vector
for (i=1; i<=8; i++) {                  // make msg polynomial have ~8 terms
  hc=hc+a^random(0,n)*x^random(0,k-1);  // hc = msg polynomial (deg < k)
}
for (i=1; i<=n; i++) {
  cw=cw+subst(hc, x, A[i])*gen(i);      // pretend codeword cw~msg poly hc is sent
}
for (i=1; i<=tau; i++) {                // tau~=number of errors
  e=e+a^random(0,n)*gen(random(0,n));   // error vector e added to cw during send
}
r=cw+e;                                 // r is our pretend received word
//-------------------------------------------------------------------------------//


//--- BUILDING A BASIS FOR Irml ------------------------------------------------//
// COMPUTE hr(j)=h_{sigma^{j-1}(r)}'s
for(j=1; j<=w; j++)
{
  ylist=0;
  // the list of y-coords of the points is r shifted left j-1 times...
  for(i=1; i<=n; i++)  {  ylist=ylist + r[(i+j-2)%n+1]/V[(i+j-2)%n+1]*gen(i);  }
  poly hr(j)=Interpolate(n, A, ylist);
```

```
}

// COMPUTE THE POWERS OF (y(j)-hr(j))^i
for(j=1; j<=w; j++)
{
  poly y_hr(j)(0)=1;
  for(i=1; i<=l; i++)
  {
    poly y_hr(j)(i)=y_hr(j)(i-1)*(y(j)-hr(j));
  }
}

// COMPUTE THE INITIAL GENERATING SET OF SUBMODULE Irml
  g=0;   // initialize basis
  for (j=0; j<p; j++)
  {
    c=1;  for(i=1; i<=w; i++) { c=c*y_hr(i)(rho(j)[i]); }  c=c*eta(rho(j)[w+1]);
    g=g+c*gen(j+1);
  }
//---------------------------------------------------------------------------//


//--- MAIN LOOP -------------------------------------------------------------//
// Computes a Groebner basis from the initial basis a la Algorithm 4.5.8
//---------------------------------------------------------------------------//

  time=timer;
  for (s=2; s<=p; s++)
  {
    printf(" s=%s.", s);  // display progress (iteration #) during loop
    lexp_gs=leadexp(g[s]);
    // find which rho(t) matches g(s)'s leading exponent...
    t=1;  while(rho(t-1)[1..w] <> lexp_gs[1..w]) { t++; }
    while (t!=s)
    {
      lexp_gt=leadexp(g[t]);
      d=lexp_gs[w+1]-lexp_gt[w+1];
      c=leadcoef(g[s])/leadcoef(g[t]);
      if (d>=0)
      {
        g=g-c*x^d*g[t]*gen(s);
      }
      else
      {
        g=g+(g[s]-g[t])*gen(t) + (x^(-d)*g[s]-c*g[t]-g[s])*gen(s);
      }
      lexp_gs=leadexp(g[s]);
      t=1;  while(rho(t-1)[1..w] <> lexp_gs[1..w]) { t++; }
    }
  }
  time=timer-time;  time;  // compute time elapsed during for loop

//---------------------------------------------------------------------------//
```

```
//--- FINDING THE FOLDED ROOTS ------------------------------------------------//
// Basically like Algorithm 4.4.4
//-----------------------------------------------------------------------------//
// FIND THE MINIMAL POLYNOMIAL Q OF THE GROEBNER BASIS
i=1;
for (j=2; j<=l+1; j++) {  if (g[j] < g[i]) {  i=j;  }  }
Q=g[i];

// COMPUTE R WHOSE y-ROOTS ARE THE FOLDED ROOTS OF Q
R=Q;
for(j=2; j<=w; j++) {  R=subst(R, y(j), y(1)^(q^(j-1)));  }
time=timer;
yroots=RootDig(R, k, h, x, y(1));   // Call to Algorithm 2.3.3
time=timer-time;  time;
print(yroots);

// CONVERT LIST OF VECTORS FROM RootDig TO CORRESPONDING POLYNOMIALS AND CODEWORDS
d=size(yroots);
yreval=list();
for (i=1; i<=d; i++) {
  c=0;
  for (j=1; j<=k; j++) {  c=c+yroots[i][j]*x^(j-1);  }
  yroots=delete(yroots, i);
  yroots=insert(yroots, c);
  yreval=insert(yreval,evalfc);
}
print(yroots);
//print(yreval);  // print the codewords if desired, but output may be long
//--- THE END! ----------------------------------------------------------------//
```

Sample Execution

Here is a sample execution of the list decoding program. The code is the folded Reed-Solomon code $\mathbb{F}_{11}(10, 2, 2(2, 4, 8, 5, 10, 9, 7, 3, 6, 1))$ (we use $\alpha \equiv_{11} 2$ here). Interpolation is done in 3 variables (x, y_1, y_2) with an error vector of weight only 1. The actual bound on the number of errors our algorithm can correct in this setting is 0 by Proposition 4.3.7 but it actual does correct the error showing that the bound we give is not tight (which is good - we can do better than expected), at least on small codes. The message polynomial 'sent' in this example was $-2x$ and two polynomials were recovered: the actual one, $-2x$, corresponding to the codeword $[-2, -4, 3, -5, 1, 2, 4, -3, 5, -1]$, and $-4x - 2$, which corresponds to $[5, 1, 4, -1, 0, 2, -5, 3, -3, -4]$. The later is nowhere close to the received word $r = [-2, -4, 3, -5, 0, 2, 4, -3, 5, -1]$, but it is a folded root of the minimal polynomial $Q = y_1y_2^2 + y_1y_2x + 2y_2^2x - y_1x^2 + 2y_2x^2 - 2x^3 + 2y_1y_2 - 3y_1x + 4y_2x + 5x^2$.

```
                     SINGULAR                                /
 A Computer Algebra System for Polynomial Computations   /   version 3-0-4
                                                       0<
     by: G.-M. Greuel, G. Pfister, H. Schoenemann        \   Nov 2007
FB Mathematik der Universitaet, D-67653 Kaiserslautern    \
> // --- LIBRARY FILES ----------------------------------------------------
```

```
//
. // These library files below should be stored within a directory in the SINGU
LAR SearchPath (or their path names may be given, alternatively)
. // ---------------------------------------------------------------------------
//
. LIB "FastInterpolation.lib";
// ** loaded FastInterpolation.lib (1.10,2001/01/16)
> LIB "RootDigger2.lib";
// ** loaded RootDigger2.lib (3.0.2,2007/08/24)
> LIB "inout.lib";   // used mainly for debugging
// ** loaded /usr/share/Singular/LIB/inout.lib (1.28,2006/07/20)
> LIB "usefulprocs.lib";
// ** loaded usefulprocs.lib (1.28,2006/07/20)
// ** library usefulprocs.lib has old format. This format is still accepted,
// ** but for functionality you may wish to change to the new
// ** format. Please refer to the manual for further information.
> // ---------------------------------------------------------------------------
//
.
.
. // --- PARAMETERS -----------------------------------------------------------
//
. // The first set of parameters are user defined.  The second set should be co
mputed according to the first set (I use Maple for them).
. // ---------------------------------------------------------------------------
//
. // USER-DEFINED PARAMETERS
. int q=11;    // base field size (of the Alternant Code)
> int S=1;     // degree of the field extension
> int qs=q^S;  // extension field size (of the GRS Code)
> int b=1;     // b=1 unless code is not a narrow sense BCH Code
> int n=10;    // length of the GRS code
> int k=2;     // dimension of the GRS code
> int v=2;     // fold number
> int w=2;     // number of y variables (doesn't include x)
> int m=2;     // multiplicity number
>
. // COMPUTED PARAMETERS (ones that aren't computed here can be found using the
 Maple program "Calc LD Parameters.mw"
. int L=1-b;            // defining parameter for multiplier of non-narrow sens
e BCH Codes
> int N=n/v;            // Length of the Folded GRS Code
> int K=k/v;            // Dimension of the Folded GRS Code
> int D=10;             // degree bound on the minimal polynomial of the ideal
Irm
> int tau=1;            // error threshold bound
> int l=10;             // bound on y-degree of minimal polynomial
> int p=binom(w+l,w);   // # of monomials in y vars of degree < l (also, # of g
enerators of submodule Irml)
>
. // MISCELLANEOUS VARIABLES
. int d; int s; int t; int i; int j; int P; int bc; int bP;
>
. // DEGREE WEIGHT VECTORS
```

```
. //        k-weight: 1,k-1,...,k-1
. //        y-degree vector: 0,1,...,1
. //        x-degree vector: 1,0,...,0
. intvec kdeg=1;  intvec ydeg=0;  intvec xdeg=1;
> for(i=1; i<=w; i++) {  kdeg=k-1,kdeg; ydeg=1,ydeg; xdeg=1,xdeg;  }
>
. // RING DECLARATION
. // the monomial order is weighted reverse-lexicographic (with x at the end of
 the alphabet)
. ring R1=qs, (y(1..w),x), wp(kdeg);  poly a=2;  // (a is a primitive element o
f the field GF(qs))
> poly h=x^(q-1)-a;  // irreducible polynomial used for folded root-finding
> ideal H=std(h);    // computes a standard basis of the ideal (h(x)) so we can
 mod out by h(x) later
>
. // DEFINING A AND V FOR THE CODE GRS(A,V)
. poly B=a^1;
> vector A=0;
> vector V=0;
> for (i=1; i<=n; i++)
. {
.   A=A+B^(i-1)*gen(i);
.   V=V+1/(B^(-L*(i-1)))*gen(i);
. }
>
. // OTHER MISCELLANEOUS VARIABLES
. intvec lexp_gs; intvec lexp_gt;
> list yroots; list yreval;
> poly c; poly g_st; poly g_tt; poly Q; poly R;
> vector g; poly hc; vector cw; vector e; vector r; vector ylist;
>
. timer=1; // The time of each command is printed (if > 1 sec)
//used time: 0.58 sec
> int time=timer; // initialize time by timer
>
. // COMPUTE POWERS OF eta=(x-A[1])*...*(x-A[n])
. poly eta=1; for (i=1; i<=n; i++) {  eta=eta*(x-A[i]);  }
> poly eta(0)=1;  for(i=1; i<=m; i++) { poly eta(i)=eta(i-1)*eta; }  // precomp
ute the powers of eta;
> time=timer-time;  time;  // yields the time in ticks-per-sec (default 1)
0
>
. // ENUMERATE THE INDICES OF THE GENERATORS (ordered by y-degree)
. intvec rho(0)=0;
> for(i=1;i<=w;i++){ intvec rho(i)=1; }
> for(i=1;i<w;i++){ for(j=0;j<=w;j++){ if(i>=j) {rho(j)=0,rho(j);} else {rho(j)
=rho(j),0;} } }
> P=w;
> for(j=1; j<l; j++)
. {
.   for(i=1; i<=w; i++)
.   {
.     bc=binom(w-i+j,j);
.     bP=binom(w+j,j)-bc-1;
```

```
.     for(t=1; t<=bc; t++) {  intvec rho(P+t)=rho(bP+t) + rho(i);  }
.     P=P+bc;
.   }
. }
> // ADD ETA POWER TO LAST POSITION OF rho(i)
. for(j=0; j<p; j++){  rho(j)=rho(j), max(0,m-norm(rho(j)));  }
> //--------------------------------------------------------------------------
//
.
.
. //--- GENERATING A RANDOM CODEWORD TO FUDGE AND DECODE ----------------------
//
. hc=0;  // initialize message polynomial
> cw=0;  // initialize corresponding codeword
> e=0;   // initialize error vector
> for (i=1; i<=8; i++) {                  // make msg polynomial have ~8 terms
.   hc=hc+a^random(0,n)*x^random(0,k-1);  // hc = msg polynomial (deg < k)
. }
> for (i=1; i<=n; i++) {
.   cw=cw+subst(hc, x, A[i])*gen(i);      // pretend we're sending the codeword
 cw generated by hc.
. }
> for (i=1; i<=tau; i++) {                // tau~=number of errors
.   e=e+a^random(0,n)*gen(random(0,n));   // an error vector e is added to cw d
uring the transmission.
. }
> r=cw+e;                                 // r is our pretend received word.
> //--------------------------------------------------------------------------
//
.
.
. //--- BUILDING A BASIS FOR Irml ----------------------------------------------
//
. // COMPUTE hr(j)=h_{sigma^{j-1}(r)}'s
. for(j=1; j<=w; j++)
. {
.   ylist=0;
.   for(i=1; i<=n; i++)  {  ylist=ylist + r[(i+j-2)%n+1]/V[(i+j-2)%n+1]*gen(i);
  }  // list of y-coords of the points is r shifted left j-1 tmies.
.   poly hr(j)=Interpolate(n, A, ylist);
. }
// ** redefining FIsubproducttreeM
// ** redefining FIindepvaluesu
// ** redefining FIcombinecoeffsc
>
. // COMPUTE THE POWERS OF (y(j)-hr(j))^i
. for(j=1; j<=w; j++)
. {
.   poly y_hr(j)(0)=1;
.   for(i=1; i<=l; i++)
.   {
.     poly y_hr(j)(i)=y_hr(j)(i-1)*(y(j)-hr(j));
.   }
. }
```

```
>
. // COMPUTE THE INITIAL GENERATING SET OF SUBMODULE Irml
.   g=0;   // initialize basis
>   for (j=0; j<p; j++)
.   {
.     c=1;  for(i=1; i<=w; i++) { c=c*y_hr(i)(rho(j)[i]); }  c=c*eta(rho(j)[w+1
]);
.     g=g+c*gen(j+1);
.   }
> //---------------------------------------------------------------------------
//
.
.
. //--- MAIN LOOP ---------------------------------------------------------------
//
. // Computes a Groebner basis from the initial basis a la Algorithm 4.5.8
. //---------------------------------------------------------------------------
//
.
.   time=timer;
>   for (s=2; s<=p; s++)
.   {
.     printf(" s=%s.", s);  // display progress (iteration #) during loop
.     lexp_gs=leadexp(g[s]);
.     t=1;  while(rho(t-1)[1..w] <> lexp_gs[1..w]) { t++; }  // find which rho(
t) matches g(s)'s leading exponent
.     while (t!=s)
.     {
.       lexp_gt=leadexp(g[t]);
.       d=lexp_gs[w+1]-lexp_gt[w+1];
.       c=leadcoef(g[s])/leadcoef(g[t]);
.       if (d>=0)
.       {
.         g=g-c*x^d*g[t]*gen(s);
.       }
.       else
.       {
.         g=g+(g[s]-g[t])*gen(t) + (x^(-d)*g[s]-c*g[t]-g[s])*gen(s);
.       }
.       lexp_gs=leadexp(g[s]);
.       t=1;  while(rho(t-1)[1..w] <> lexp_gs[1..w]) { t++; }
.     }
.   }
>   time=timer-time;  time;  // compute time elapsed during for loop
431
>
. //---------------------------------------------------------------------------
//
.
.
. //--- FINDING THE FOLDED ROOTS -----------------------------------------------
---//
. // Basically like Algorithm 4.4.4
```

```
. //-------------------------------------------------------------------------
//
. // FIND THE MINIMAL POLYNOMIAL Q OF THE GROEBNER BASIS
. i=1;
> for (j=2; j<=l+1; j++) {  if (g[j] < g[i]) {  i=j;  }  }
> Q=g[i];
>
. // COMPUTE R WHOSE y-ROOTS ARE THE FOLDED ROOTS OF Q
. R=Q;
> for(j=2; j<=w; j++) {  R=subst(R, y(j), y(1)^(q^(j-1)));  }
> time=timer;
> yroots=RootDig(R, k, h, x, y(1));   // Call to Algorithm 2.3.3
> time=timer-time;  time;
0
> print(yroots);
[1]:
   -4*gen(2)-2*gen(1)
[2]:
   -gen(2)-3*gen(1)
>
. // CONVERT LIST OF VECTORS FROM RootDig TO CORRESPONDING POLYNOMIALS AND CODE
WORDS
. d=size(yroots);
> yreval=list();
> for (i=1; i<=d; i++) {
.   c=0;
.   for (j=1; j<=k; j++) {  c=c+yroots[i][j]*x^(j-1);  }
.   yroots=delete(yroots, i);
.   yroots=insert(yroots, c);
.   yreval=insert(yreval,evalfc);
. }
> print(yroots);
[1]:
   -2*x
[2]:
   -4*x-2
> //print(yreval);  // print the codewords if desired, but output may be long
.
.
. //--- THE END! ---------------------------------------------------------------
//
.
. hc;
-2*x
> print(evalf(-2*x));
[-2,-4,3,-5,1,2,4,-3,5,-1]
> print(evalf(-4*x-2));
[5,1,4,-1,0,2,-5,3,-3,-4]
> print(cw);
[-2,-4,3,-5,1,2,4,-3,5,-1]
> print(e);
[0,0,0,0,-1]
> print(r);
[-2,-4,3,-5,0,2,4,-3,5,-1]
```

```
> Q;
y(1)*y(2)^2+y(1)*y(2)*x+2*y(2)^2*x-y(1)*x^2+2*y(2)*x^2-2*x^3+2*y(1)*y(2)-3*y(1)*
x+4*y(2)*x+5*x^2
> subst(subst(Q,y(2),-4*a*x-2),y(1),-4x-2);
0
> subst(subst(Q,y(2),-2*a*x),y(1),-2x);
0
>
```

Maple Worksheet

This short Maple worksheet is a useful tool for computing the parameters `D`, `tau`, and `l` used in the SINGULAR list decoding program. These particular parameters are difficult to calculate in SINGULAR because it does not have a square root function, nor does it do real arithmetic unless you set $\mathbb{R}$ as the ground field of your polynomial ring (which I do not care to do). The first command (or list of such) sets the user-defined parameters of the algorithm. The second command computes the other parameters from those. The third command is helpful for finding a primitive element of the field $GF(qs)$ to define your code.

```
> # User-defined parameters (same as in the SINGULAR program)
> q := 11; qs := 11; b := 1; n := 10; k := 2; v := 2; w := 2; m := 2;
> # Computed parameters for SINGULAR program (the values of d (D in SINGULAR),
  tau, and l, must be manually set).
> N := n/v; K:=k/v;
  d := floor(((k-1)^w*n*(product(m+w-j, j = 0 .. w)))^(1/(w+1)))+1;
  tau:=N-floor(N*v*(((k-1)/n)^w*(product(m+w-j,j=0 .. w)))^(1/(w+1))/m(v-w+1))-2;
  l := floor(d/(k-1)); p1 := binomial(m+w, m); p2 := binomial(w+l, l);
> # Thus is a useful routine for finding a primitive element of Z mod qs.
> # Just pick numbers for a until all qs-1 of the numbers output are distinct.
> a := 2; for i to qs-1 do print(‘mod‘(a^i, qs)) end do;
```

Custom Singular Libraries

FastInterpolation.lib

This library contains a function for performing multi-point based fast univariate interpolation, as described in [38, Ch. 10]. The procedure `Interpolate` is called in the main list decoding program to compute the polynomials $h_{\sigma^j(\boldsymbol{r})}$ interpolating $\sigma^j(\boldsymbol{r})_i = r_{i+j}$ for each of the n code locators α^i. This library also has a useful multi-point evaluation procedure `Multeval` which is used within the `Interpolate` function but not used otherwise in the present version of the list decoding program.

```
////////////////////////////////////////////////////////////////////////////
// version string automatically expanded by CVS
version="$Id: template.lib,v 1.10 2001/01/16 13:48:46 Singular Exp $";
category="Miscellaneous";
// summary description of the library
info="
LIBRARY:   FastInterpolation.lib  Implementation of fast interpolation algorithm
AUTHOR:    Philip Busse, email: pbusse@ms.uky.edu
SEE ALSO:  standard_lib, Guidelines for writing a library,
           Typesetting of help strings
```

```
KEYWORDS:  library, FastInterpolation; FastInterpolation.lib;
                    von zur Gathen, Gerhard

PROCEDURES:
  BuildM(int, vector)              return subproduct matrix of polynomials
  Multeval(poly, int, int)         return evaluation of poly at multiple points
  CombineM(int, int)               return combo of subproducts for interpolation
  Interpolate(int, vector, vector) return poly interpolated through given points
  Log(int)                         return ceil(log base 2 of n)
";
////////////////////////////////////////////////////////////////////////////
proc BuildM (int n, vector u)
"Alg. 10.3 from von zur Gathen & Gerhard: Building up the subproduct tree.
USAGE:    BuildM(n, u);  (int n, vector u[1..n] of distinct algebraic values)
RETURNS:  polynomials M[i,j]=prod(l=1..2^i, x-u[(j-1)*2^i+l])"
{
   int i; int j;
   int k=Log(n);
   int r=2^k;
   matrix M[k+1][r];
   for (j=1; j<=n; j++)  {  M[1,j]=x-u[j];  }
   for (j=n+1; j<=r; j++)  {  M[1,j]=1;  }
   for (i=1; i<=k; i++)
   {
      for (j=1; j<=2^(k-i); j++)
      {
         M[i+1,j]=M[i,2*j-1]*M[i,2*j];
      }
   }
   return(M);
}
example
{
  "EXAMPLE:"; echo=2;
  matrix M=BuildM(5, [1,2,3,4,5]);
  print(M);
}
////////////////////////////////////////////////////////////////////////////
proc Multeval (poly f, int initindex, int termindex)
"Alg 10.5 from von zur Gathen & Gerhard: Going down the subproduct tree
USAGE:    Multeval(f, a, b);  (poly f of deg<n, int a, int b)
GLOBALS:  FIsubproducttreeM
RETURNS:  vector [f(u[a])..f(u[b])]"
{
// printf("poly=%s, initindex=%s, termindex=%s", f, initindex, termindex);
   if (initindex == termindex)
   {
      return([f]);
   }
   else
   {
      int i;
      int n=termindex-initindex+1;
```

```
      int k=Log(n);
// printf("  k=%s", k);
      int n0=2^(k-1);
      poly r0=f-FIsubproducttreeM[k, (initindex-1)/n0+1]*
              (f/FIsubproducttreeM[k, (initindex-1)/n0+1]);
      poly r1=f-FIsubproducttreeM[k, (initindex-1)/n0+2]*
              (f/FIsubproducttreeM[k, (initindex-1)/n0+2]);
      vector r0eval=Multeval(r0, initindex, initindex+n0-1);
      vector r1eval=Multeval(r1, initindex+n0, termindex);
      vector r=r0eval;
      for (i=n0+1; i<=n; i++)  {  r=r+r1eval[i-n0]*gen(i);  }
      return(r);
   }
}
example
{
  "EXAMPLE:"; echo=2;
  vector u=[1,2,3,4,5];
  matrix FIsubproducttreeM=BuildM(5, u);
  Multeval(x3+x+1, 1, 5);  // displays value of x^3+x+1 at u[1],...,u[5]
}
////////////////////////////////////////////////////////////////////////////
proc CombineM (int initindex, int termindex)
"Alg. 10.9 from von zur Gathen & Gerhard: Linear combination for linear moduli.
USAGE:    CombineM(a, b);  (int a, int b)
GLOBALS:  FIindepvaluesu[1..n],
          FIcombinecoeffsc[1..n],
          FIsubproducttreeM
RETURNS:  sum(i=1..n, c[i]*m/(x-u[i]), where
          m=(x-FIindepvaluesu[initindex])*...*(x-FIindepvaluesu[termindex])"
{
   if (termindex == initindex)
   {
      return(FIcombinecoeffsc[initindex]);
   }
   else
   {
      int n=termindex-initindex+1;
      int k=Log(n);
      int n0=2^(k-1);
      poly r0=CombineM(initindex, initindex+n0-1);
      poly r1=CombineM(initindex+n0, termindex);
      return(FIsubproducttreeM[k, (initindex-1)/n0+2]*r0 +
             FIsubproducttreeM[k, (initindex-1)/n0+1]*r1);
   }
}
example
{
  "EXAMPLE:"; echo=2;
  vector FIindepvaluesu=[1,2,3,4,5];
  matrix FIsubproducttreeM=BuildM(5, FIindepvaluesu);
  vector FIcombinecoeffsc=Multeval(x3+x+1, 1, 5);
  CombineM(1, 5);
}
```

```
////////////////////////////////////////////////////////////////////////////
proc Interpolate (int n, vector xlist, vector ylist)
"Alg. 10.11 from von zur Gathen & Gerhard: Fast Interpolation.
USAGE:    Interpolate(n,xlist,ylist);    int n,
                                         vector xlist[1..n] (independent vals),
                                         vector ylist[1..n] (dependent values))
GLOBALS:  vector FIindepvaluesu,
          vector FIcombinecoeffsc,
          matrix FIsubproducttreeM
RETURNS:  unique f(x) of deg < n s.t. f(u[i])=v[i] for i=1..n"
{
   int i;
   vector FIindepvaluesu=xlist;  vector v=ylist;
   matrix FIsubproducttreeM=BuildM(n, FIindepvaluesu);
   int k=Log(n);
   poly m=FIsubproducttreeM[k+1,1];
   poly dm=diff(m, x);
   export(FIsubproducttreeM);
   vector dmeval=Multeval(dm, 1, n);
   vector FIcombinecoeffsc;
   for(i=1;i<=n;i++) {FIcombinecoeffsc=FIcombinecoeffsc+v[i]/dmeval[i]*gen(i);}
   export(FIindepvaluesu, FIcombinecoeffsc);
   return(CombineM(1, n)); } example
{
  "EXAMPLE:"; echo=2;
  Interpolate(5, [1,2,3,4,5], [1,4,9,16,25]);
}
////////////////////////////////////////////////////////////////////////////
proc Log (int n)
"logarithm base 2 of n
USAGE:    Log(n);   int n should be >0.
RETURNS:  the ceiling of the log to the base 2 of n"
{
   int l=0;
   while (n>2^l) {l++;}
   return(l);
}
example
{
  "EXAMPLE:"; echo=2;
  Log(25);
}
```

RootFinder.lib

This is a library for the procedure `RootDig` which is essentially Algorithm 2.3.3. `RootDig` is called by the main list decoding program soon after the main loop (which is near the end).

```
////////////////////////////////////////////////////////////////////////////
// version string automatically expanded by CVS

version="$Id: template.lib,v 3.0.2 2007/08/24 13:48:46 Singular Exp $";
category="Miscellaneous";
```

```
// summary description of the library
info="
LIBRARY:   RootFinder.lib, Implementation of fast bivariate root-finding algorithm
           modulo an irreducible polynomial
AUTHOR:    Philip Busse, email: pbusse@ms.uky.edu

SEE ALSO:  standard_lib, Guidelines for writing a library,
           Typesetting of help strings

KEYWORDS:  library, RootFinder.lib; RootFinder.lib; library, Roth, Ruckenstein

PROCEDURES:
  RootDig(poly, int, poly, var, var)           entry fnc into main recursive proc
  Reconstruct(poly, int, int, ideal, var, var) puts y-roots of poly in global list
";

////////////////////////////////////////////////////////////////////////////////
proc RootDig (poly Q, int k, poly h, x, y)
"Entry function to recursive reconstruction routine
GLOBALS:  vector RFphi
          list RF
yroots
USAGE:    RootDig(Q, k, h, x, y);
RETURNS:  list of coeff. vectors of yroots mod h(x) of Q(x,y)!=0 (poly's in F[x]
          of degree < k) stored in the global list yroots
{
   vector RFphi=0;
   list RFyroots=list();
   ideal H=std(h);
   export(RFphi, RFyroots);
   Reconstruct(Q, k, 1, H, x, y);
   return(RFyroots);
}
example
{
  "EXAMPLE:"; echo=2;
  list L=RootDig(y^2*(y+x+x4), 4, x^36-2, x, y);
  print(L);
}
////////////////////////////////////////////////////////////////////////////////
proc Reconstruct (poly Q, int k, int i, ideal H, x, y)
"Roth & Ruckenstein's recursive procedure to find a superset of consistent poly's
GLOBALS:  vector RFphi
          list RFyroots
USAGE:    reconstruct(Q, k, 1, H, x, y);
RETURNS:  nothing directly; de facto, coeffs of yroots of Q(x,y)!=0 (elts in F[x]
          of degree < k) stored in the global list yroots
{
   int j;  int d;
   int r=0;
   matrix C=coeffs(Q, x);
   if (Q != 0) {  while (C[r+1,1] == 0) {  r++;  }  }
   poly M=reduce(Q/x^r, H);
   list factors=factorize(subst(M, x, 0), 2);
```

```
   int nfactors=size(factors[1]);
   for (j=1; j<=nfactors; j++) {
      d=deg(coef(factors[1][j], y)[1,1]);
      if (d==deg(y)) {
         C=coeffs(factors[1][j], y);
         RFphi=RFphi-(RFphi[i]+C[1,1]/C[2,1])*gen(i);
         if (i==k) {
            RFyroots=insert(RFyroots, RFphi);
         }
         else {
            Reconstruct(subst(M, y, x*y+RFphi[i]), k, i+1, H, x, y);
         }
      }
   }
}
example
{
  "EXAMPLE:"; echo=2;
  vector RFphi=0;
  list RFyroots;
  Reconstruct(y^2*(y+x+x4), 5, 1, std(x^36-2), x, y);
  print(RFyroots);
}
```

UsefulProcs.lib

This is not really a library, but more of a random collection of semi-useful, basic procedures. The procedures in this file that are actually called by the main list decoding program are `binom`, `max`, `norm`, and `evalf`.

```
// COMPUTE THE BINOMIAL COEFFICIENT, n choose k
proc binom (int n, int k)
  {
    if (k<0 or k>n) { return(0); }
    if (k==0 or k==n) { return(1); }
    return(binom(n-1,k-1) + binom(n-1,k));
  }

// COMPUTE n!
proc factorial(int n)
  {
    if (n<1) { return(1); }
    return(n*factorial(n-1));
  }

//COMPUTE THE CEILING OF THE LOG TO THE BASE b OF n: ceil(log[b](n))
proc LOG(int b, int n)
  {
    int l=0;
    while (n>b^l) {l++;}
    return(l);
  }

//COMPUTE THE FLOOR OF THE LOG TO THE BASE b OF n: floor(log[b](n))
```

```
proc log(int b, int n)
  {
    int l=0;
    while (n>=b^l) {l++;}
    return(l-1);
  }

//COMPUTE THE MAXIMUM VALUE OF TWO INTEGERS
proc max(int i, int j)
  {
    if (i>j) {return(i);} else {return(j);}
  }

//COMPUTE THE SUM (1-norm) OF A VECTOR OF INTEGERS
proc norm(intvec nums)
  {
    int n=size(nums);
    int sum=0;
    for(int i=1; i<=n; i++)
    {
      sum=sum+nums[i];
    }
    return(sum);
  }

//EVALUATE POLY f(x) AT VALUES OF n-TUPLE A TO GET NEW n-TUPLE
proc evalf(poly f)
  {
    vector evalf=0;
    for(int i=1; i<=n; i++)
    {
      evalf=evalf+subst(f, x, A[i]*V[i])*gen(i);
    }
    return(evalf);
  }
```

Bibliography

[1] E. Berlekamp. "Algebraic Coding Theory," McGraw-Hill Series in Systems Science, 1968.

[2] D. Bleichenbacher, A. Kiayias, M. Yang, Decoding of Interleaved Reed-Solomon Codes over Noisy Data, *Lect. Notes Computer Sci.* **2719** (2003), 97–108.

[3] B. Buchberger, A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases, *SYM-SAC 79, Lecture Notes in C.S.* **72** (1979), 3–21.

[4] B. Buchberger, A Note on the Complexity of Constructing Gröbner Bases, *Proc. EUROCAL 83, Lecture Notes in C.S.* **162** (1983), 137–145.

[5] B. Buchberger. "Gröbner Bases and Applications," Cambridge University Press, New York, 1998.

[6] D. Coppersmith, M. Sudan, Reconstructing Curves in Three (and Higher) Dimensional Spaces from Noisy Data, *Proc. 35th ACM Symposium on the Theory of Computing (STOC)* (2003), 136–142.

[7] T. Cormen, C. Leiserson, R. Rivest, C. Stein. "Introduction to Algorithms, Second Edition," MIT Press, Cambridge, MA, 2001.

[8] D. Cox, J. Little, D. O'Shea. "Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra ," Springer-Verlag, New York, 2007.

[9] D. Dummit, R. Foote. "Abstract Algebra, Second Edition," Prentice Hall, Upper Saddle River, NJ, 1999.

[10] P. Elias, List Decoding for Noisy Channels, *Res. Lab. Electron. Tech. Rep., MIT, Cambridge, MA* **335** (1957), 94–104.

[11] P. Elias, Error-correcting Codes for List Decoding, *IEEE Transactions on Information Theory* **37** (1991), 5–12.

[12] D. Eisenbud. "Commutative Algebra with a View Toward Algebraic Geometry," Springer-Verlag, New York, 1994.

[13] P. Gaborit, O. Ruatta, Improved Hermite Multivariate Polynomial Interpolation, *Information Theory, IEEE 2006 International Symposium on* (2006), 143–147.

[14] S. Gao, Factoring Multivariate Polynomials Via Partial Differential Equations, *Math. Comp.* **72, no. 242** (2003), 801–822.

[15] S. Gao, A. G. B. Lauder, Hensel Lifting and Bivariate Polynomial Factorization Over Finite Fields, *Math. Comp.* **71** (2002), 1663–1676.

[16] V. Guruswami. "List Decoding Error Correcting Codes," Springer, 2004.

[17] V. Guruswami, Algorithmic Results in List Decoding, *Foundations and Trends in Theoretical Computer Science* **2, no. 2** (2006), 107–195.

[18] V. Guruswami, A. Rudra, Explicit Codes Achieving List Decoding Capacity: Error-correction with Optimal Redundancy, *IEEE Transactions on Information Theory* **54, no. 1** (2008), 135–150.

[19] V. Guruswami, M. Sudan, Improved Decoding of Reed-Solomon and Algebraic-Geometry Codes, *IEEE Transactions on Information Theory* **45, no. 6** (September 1999), 1757–1767.

[20] E. Kaltofen, V. Shoup, Subquadratic-Time Factoring of Polynomials over Finite Fields, *Math. Comput.* **67, no. 223** (1998), 1179–1197.

[21] R. Koetter, A. Vardy, Algebraic Soft-Decision Decoding of Reed-Solomon Codes, *IEEE Transactions on Information Theory* **49, no. 11** (2003), 2809–2825.

[22] R. Koetter, A. Vardy, A Complexity Reducing Transformation in Algebraic List Decoding of Reed-Solomon Codes, *Information Theory Workshop, 2003. Proc. IEEE* (2003), 10–13.

[23] V. Y. Krachkovsky, Reed-Solomon Codes for Correcting Phased Error Bursts, *IEEE Transactions on Information Theory* **49, no. 11** (2003), 2975–2984.

[24] K. Lee, M. O'Sullivan, Sudan's List Decoding of Reed-Solomon Codes from a Gröbner Basis Perspective, preprint arXiv:math.AC/0601022 v2 30 Jan 2006 .

[25] K. Lee, M. O'Sullivan, List Decoding of Reed-Solomon Codes from a Gröbner Basis Perspective, *J. of Symbolic Computation* **43, no. 9** (2008), 645–658.

[26] A. K. Lenstra, Factoring Multivariate Polynomials Over Finite Fields, *J. Comput. System Sci.* **30, no. 2** (1985), 235–248.

[27] R. Lidl, H. Neiderreiter. "Finite Fields, Second Edition," Cambridge University Press, 1997.

[28] F. J. MacWilliams, N. J. A. Sloane. "The Theory of Error-Correcting Codes," North-Holland, 1977.

[29] D. Malik, J. Mordeson, M. Sen. "Fundamentals of Abstract Algebra," McGraw-Hill International Series in Pure and Applied Mathematics, 1997.

[30] F. Parvaresh, A. Vardy, Correcting Errors Beyond the Guruswami-Sudan Radius in Polynomial Time, *Foundations of Computer Science, IEEE 2005 Annual Symposium on* **46** (2005), 285–294.

[31] R. Pellikaan, X. Wu , List Decoding of q-ary Reed-Muller Codes, *IEEE Trans. Inform. Theory* **50, no. 4** (2004), 679–682.

[32] R. Roth. "Introduction to Coding Theory," Cambridge University Press, 2006.

[33] R. Roth, G. Ruckenstein, Efficient Decoding of Reed-Solomon Codes Beyond Half the Minimum Distance, *IEEE Trans. Inform. Theory* **46, no. 1** (2000), 246–257.

[34] A. Rudra, "List Decoding and Property Testing of Error-Correcting Codes," Doctoral dissertation, University of Washington, 2007.

[35] C. E. Shannon, A Mathematical Theory of Communication, *Bell Systems Technical Journal* **27** (1948), 379–423 and 623–656.

[36] A. Shokrollahi, H. Wasserman, List Decoding of Algebraic-Geometry Codes, *IEEE Trans. Inform. Theory* **45, no. 2** (1999), 432–437.

[37] M. Sudan, Decoding of Reed Solomon Codes Beyond the Error-Correction Bound, *J. of Complexity* **13, no. 1** (1997), 180–193.

[38] J. von zur Gathen, J. Gerhard. "Modern Computer Algebra," Cambridge University Press, 1999.

[39] J. von zur Gathen, E. Kaltofen, Factorization of Multivariate Polynomials Over Finite Fields, *Math. Comp.* **45, no. 171** (1985), 251–261.

Vita

Philip R. Busse

Department of Mathematics
University of Kentucky
Lexington, KY 40507-0027

Birthdate: January 10, 1981
Birthplace: Marshfield Clinic in Marshfield, WI

Education

University of Kentucky, Lexington, KY

- Ph.D., Mathematics, expected August 2008
  - Dissertation Topic: "Multivariate List Decoding of Evaluation Codes with a Gröbner Basis Perspective"
  - Advisor: Uwe Nagel
- M.A., Mathematics, August 2005

St. Norbert College, De Pere, WI

- B.A., Mathematics and Computer Science, Magna cum Laude, May 2003

Professional Experience

Researcher, Graduate Mathematics Program, May 31 - August 19, 2005
National Security Agency, Fort Meade, MD.
Computer technician, January 21, 2001 - May 5, 2003
St. Norbert College Survey Center, De Pere, WI.

Teaching Experience

Algebra Cubed Fellow: University of Kentucky, Fall 2007 – Summer 2008
Primary Instructor: University of Kentucky, Fall 2004 – Summer 2007
Recitation Leader: University of Kentucky, Fall 2003 – Spring 2004

Research

Research Interests:

- Algebraic coding theory; specifically, the general interpolation and root-finding algorithms utilized for list decoding algebraic geometry codes.
- Cryptography, especially secret sharing schemes, their algebraic interpretations, and their connection to erasure codes.

Awards and Fellowships

University of Kentucky, Lexington, KY

- Algebra Cubed Fellowship, Fall 2007 - Summer 2008
- Daniel R. Reedy Quality Achievement Award, Fall 2003 - Spring 2006
- Van Meter Award, Summer 2004

St. Norbert College, De Pere, WI

- Pi Mu Epsilon (National Mathematics Honor Society)

Presentations and Talks

Conference Talks

- "List Decoding BCH Codes Using Gröbner Bases," AMS Sectional Meeting, Bloomington, IN, April 6, 2008
- "List Decoding BCH Codes," Joint Meeting of the AMS and MAA, San Diego, CA, January 7, 2008
- "Gröbner Bases and List Decoding BCH Codes," AMS Sectional Meeting, DePaul University, October 6, 2007
- "Error-Correcting Codes and Abstract Algebra," Pi Mu Epsilon National Convention, University of Vermont, August 2002

University of Kentucky, Lexington, Kentucky

- "Multivariate List Decoding of Evaluation Codes with a Gröbner Basis Perspective," Doctoral Defense, June 10, 2008
- "Gröbner Bases and List Decoding BCH Codes," Algebra and Geometry Seminar, Fall 2007
- "List Decoding Reed-Solomon Codes," Algebra and Geometry Seminar, Spring 2006

Poster Presentations

- "Algebra Cubed: Unique Aspects of GK-12 Projects in Rural Areas," NSF Graduate Teaching Fellows in K-12 Education Program (GK-12), Washington, D.C., March 1, 2008
- "List Decoding BCH Codes," IMA Workshop on Complexity, Coding, and Communications, University of Minnesota, April 16, 2007

Other Conferences Attended

IMA Coding & Cryptography Summer Program, University of Notre Dame, June 8-26, 2004

Departmental Services

Graduate Student Council, University of Kentucky, Fall 2006 - Summer 2007

Professional Memberships

American Mathematical Society, Member, 2003 - present
Association of Christians in the Mathematical Sciences, Member, 2008