Algebraic Interleavers in Turbo Codes

California State Polytechnic University, Pomona

and

Loyola Marymount University

Department of Mathematics Technical Report

Jason Dolloff[*], Zackary Kenz[†], Jacquelyn Rische[‡], Danielle Ashley Rogers[§], Laura Smith[¶], Edward Mosteig[‖]

Applied Mathematical Sciences Summer Institute
Department of Mathematics & Statistics
California State Polytechnic University Pomona
3801 W. Temple Ave.
Pomona, CA 91768

August 2006

[*]Southwestern University
[†]Concordia College-Moorhead
[‡]Whittier College
[§]University of Michigan
[¶]Western Washington University
[‖]Loyola Marymount University, Los Angeles

Abstract

Coding theory is the branch of mathematics and electrical engineering that involves transmitting data across noisy channels via clever means. While the transmission can be very error-prone, there are various methods used to send the data so that a large number of the errors can be corrected. Applications in communications, the design of computer memory systems, and the creation of compact discs have demonstrated the value of error-correcting codes. Our research examines a specific class of codes called turbo codes. These high performance error-correcting codes can be used when seeking to achieve maximal information transfer over a limited-bandwidth communication channel in the presence of data-corrupting noise. In order to gain a deeper understanding of turbo codes, we study a particular component, interleavers. An interleaver is a device that scrambles the sequence of data bits before transmission. In order to study this component and how it affects the performance of turbo codes, we examine two of its properties: spread and dispersion. Using computer simulations, we will not only study how these properties affect the error rates, but also work toward creating new properties that will aid in examining the effectiveness of interleavers.

1 Introduction

1.1 What is Coding Theory?

Coding theory is the study of the transmission of data across a noisy channel, with the ultimate goal of successfully recovering from an error-ridden received message back to its original, error-free form. Since the channel is noisy, error-correcting measures must be taken in order to preserve the original message. A part of coding theory, the area we are studying, is dedicated to finding ways to predetermine which coding schemes will result in the fewest uncorrectable errors in the received message.

It is important to note that coding theory is *not* cryptography. Cryptography seeks to hide the contents of a transmitted message; coding theory seeks to successfully transmit a message without irreversible errors occurring. Though these can be used together (i.e., a message can be encoded using cryptography and sent across a channel using coding theory), our focus is on studying the best methods to transmit data so that it is resistant to corruption.

1.2 History/Applications

Coding theory has been used in various applications throughout the past fifty years. Reed-Solomon codes were used to successfully transmit pictures of Jupiter and Saturn back to Earth in the 1970s. Coding theory is also used in modern applications like CDs, DVDs, and cell phones to ensure that the information in each application reaches the end-user nearly free of errors.

1.3 Our Direction

We are examining the inner components of a code to determine which characteristics (such as the code's block length or the interleaver used) directly affect the performance of the code. Some building blocks of the interleaver include a finite field permutation (which is represented by a polynomial) and any monomial ordering used in its creation. These elements and their relation to the project will be discussed later. It is believed that changing these factors may result in improved error-reducing ability of a code; our project looks at various simulations where these factors are changed and draws conclusions from the data. Since it is not well understood why carefully constructed turbo codes tend to have excellent error-correcting abilities, a study of the properties of turbo codes may shed some light into the mechanics of the codes.

2 Turbo Codes

In our study of coding theory, we will focus on turbo codes in particular. These are a class of codes that are currently used in deep-space and satellite communications. Turbo codes use convolutional codes, interleavers and redundancy to encode messages. Figure 1 consists of a diagram depicting the type of turbo code that we will be using.

The process message transmission begins with creating three copies of the original message. One copy of the message (the upper track in the figure) is sent through unchanged,

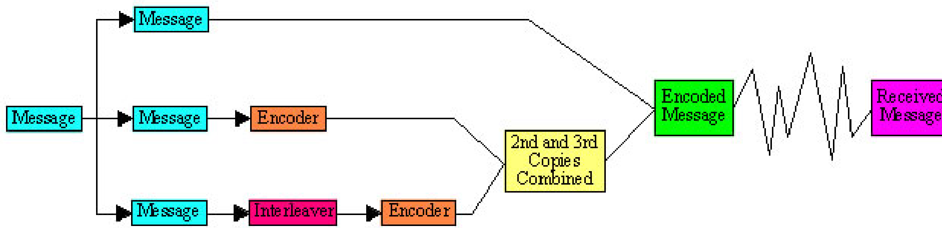


Figure 1: Diagram of a Turbo Code

and is denoted $m = (m_1, m_2, ...)$. A second copy (middle track of figure) is sent through a convolutional code and denoted $u = (u_1, u_2, ...)$. A third copy (bottom track of figure) is sent through an interleaver and then through a convolutional code and denoted $v = (v_1, v_2, ...)$. The second and third messages are then combined together, taking the odd bits from one and the even bits from the other, to form $n = (u_1, v_2, u_3, v_4, ...) = (n_1, n_2, n_3, n_4, ...)$. This vector is then combined with m as follows: $c = (m_1, n_1, m_2, n_2, ...)$. The rate at which the message is sent is $\frac{1}{2}$, which means the encoded message is twice as long as the original. Lower rate codes are more effective at correcting errors, but transmission takes longer. After the final encoded message is computed, it is transmitted across across a (noisy) channel. Once received, the intended recipient attempts to decode the message.

2.1 Decoding the Message

To decode the received message, it is first split into two halves according to the odd and even bits. These two halves are sent to distinct decoders. These two decoders use iterative decoding and belief propagation. During the transmission, the message might have become corrupted with errors due to noise. The two decoders each look at the message and compare it back and forth for a set number of iterations. When one decoder finds an error it tries to fix it, and then gives the "fixed" message to the other decoder. In this process, the decoders use belief propagation to compare the message and find (and hopefully fix) any errors.

2.2 Convolutional Codes

Convolutional codes use shift registers to encode the message that is being sent. Figure 2 is an example of a shift register sequence. Before the message is sent through the convolutional code, the shift registers begin with 0 inside. As the message is encoded, the bits enter one at a time and follow along the paths. Whenever a bit enters a shift register, it pushes out the register's previous occupant.

Example 1. Let the input be 1011011000.... By looking at Figure 3 and the following table, one can see that the output is 1010000011....

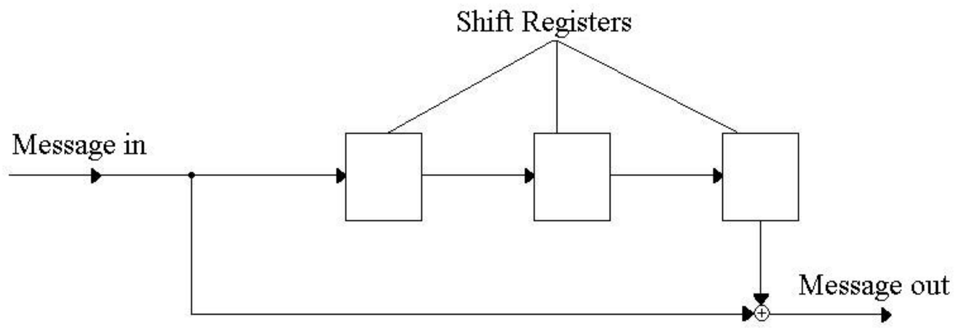


Figure 2: Shift Register

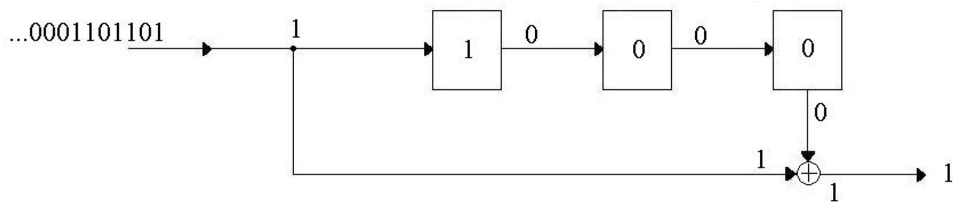


Figure 3: A Message in a Shift Register

Input	Shift Register	Output
1	000	1
0	100	0
1	010	1
1	101	0
0	110	0
1	011	0
1	101	0
0	110	0
0	011	1
0	001	1

Table of Inputs and Outputs from Example 1

Figure 4 shows an example of a shift register with feedback. This sends some of the bits from the message back through the shift registers. For a more in-depth treatment of convolutional codes, one can see [Fo].

2.3 Benefits of the Components of Turbo Codes in Decoding

Each component of a turbo code works to preserve the original message. Adding redundancy gives the decoders multiple places to check when trying to correct errors. Permutations help

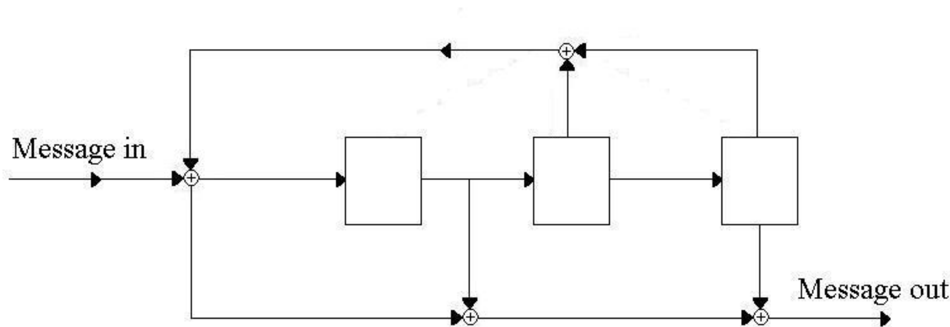


Figure 4: A Shift Register with Feedback

to scatter the data so that if there is a burst of errors, instead of losing a big chunk of the message, little bits are lost from various places, which are more easily corrected.

3 Interleavers

Before we discuss interleavers, we must start with a definition.

Definition 2. For any positive integer q, the set $\mathbb{Z}_q$ is defined to be the set of integers $\{0, 1, ..., q-1\}$.

Definition 3. An **interleaver** is a bijection (i.e. a permutation) that maps $\mathbb{Z}_q$ onto $\mathbb{Z}_q$. We call q the size, or (block) length of the interleaver.

The following is an example of a permutation that could be used as an interleaver.

Example 4. Let π be the following permutation of $\mathbb{Z}_6$:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & 0 & 3 & 1 \end{pmatrix}.$$

Each number in the top row is mapped to a unique number below it.

There are two particular properties of interleavers that we are interested in: dispersion and spread. For more information on these and how they are calculated, see Sections 7 and 8.

4 Finite Fields

In order to construct our interleavers, we use finite fields. Finite fields and finite field arithmetic provide a way to create algebraic interleavers. With algebraic interleavers, we can construct an explicit formula for a permutation. For permutations with large block length, using a formula could be more efficient than storing and looking up elements of a random permutation. For more details on the construction of interleavers, see Section 6.

Definition 5. A **field** $\mathbb{F}$ is a set of elements with two operations defined on the set, "addition" and "multiplication," such that the following properties hold:

(i) Closure: $\forall a, b \in \mathbb{F}, a \cdot b \in \mathbb{F}$, and $a + b \in \mathbb{F}$

(ii) Commutativity: $\forall a, b \in \mathbb{F}, a + b = b + a,$ and $a \cdot b = b \cdot a$

(iii) Associativity: $\forall a, b, c \in \mathbb{F}, (a + b) + c = a + (b + c),$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

(iv) Distributivity: $\forall a, b, c \in \mathbb{F}, a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

(v) Identity: $a + 0 = a,\ a \cdot 1 = a,$ where $a, 0, 1 \in \mathbb{F}$

(vi) Additive Inverses: $\forall a \in \mathbb{F}, \exists\ (-a) \in \mathbb{F}$ such that $a + (-a) = 0$

(vii) Multiplicative Inverses: $\forall a \in \mathbb{F}, a \neq 0, \exists\ b \in \mathbb{F}$ such that $a \cdot b = 1$

Example 6. The set of natural numbers $\mathbb{N}$ is not a field because 2, for example, has neither an additive nor a multiplicative inverse. On the other hand, $\mathbb{Q}$ and $\mathbb{R}$ are fields.

Theorem 7. *The set of integers $\mathbb{Z}_p$ is a field if and only if p is prime.*

It should be noted that if p is prime, we often use the notation $\mathbb{F}_p = \mathbb{Z}_p$. However, one needs to be careful because if p is not prime, $\mathbb{F}_p$ and $\mathbb{Z}_p$ have different meanings.

Definition 8. A field is a **finite field** if it contains only a finite number of elements.

Theorem 9. *There exists a field with q elements if and only if $q = p^r$, where p is prime. We denote this by $\mathbb{F}_q$.*

For proofs of Theorems 7 and 9, one can see [Ro].

Definition 10. Given $\mathbb{F}_p$, define the set $\mathbb{F}_{p^r}$ for any positive integer r as the set of polynomials of degree at most $r - 1$ with coefficients in $\mathbb{F}_p$:

$$\mathbb{F}_{p^r} = \{c_0 + c_1 x + c_2 x^2 + ... + c_{r-1} x^{r-1} | c_i \in \mathbb{F}_p\}$$

Example 11. The set of all polynomials of degree 2 or less with coefficients 0 or 1 is $\mathbb{F}_8 = \mathbb{F}_{2^3} = \{0, 1, x, x^2, 1 + x, 1 + x^2, x + x^2, 1 + x + x^2\}$.

Now that we have the defined the set $\mathbb{F}_{p^r}$, we must examine additional properties of finite fields. These elements lend themselves well to work with interleaver construction. The notion of irreducible polynomials is key to giving fields more structure.

4.1 Irreducible Polynomials

Definition 12. A polynomial is irreducible over a field F if it cannot be written in the form $f(x) = g(x)h(x)$ where $g(x), h(x)$ are non-constant polynomials of degree less than that of $f(x)$.

Theorem 13. *For any positive integer, m, and a prime, p, there exists a polynomial, f, of degree m with coefficients in $\mathbb{F}_p$ that is irreducible.*

For a proof of Theorem 13, see [Ro].

The most important use for irreducible polynomials in giving fields more structure is in the definition of multiplication in a finite field, which will be detailed after a discussion on addition in a finite field.

4.2 Addition and Multiplication in Finite Fields

To add two polynomials in a finite field, one simply needs to add them together and reduce the coefficients modulo p.

Example 14. Adding $1 + x$ and $1 + x + x^2$ in $\mathbb{F}_{2^3}$ gives:

$$(1 + x) + (1 + x + x^2) = 2 + 2x + x^2 \equiv x^2 \mod 2.$$

So $(1 + x) + (1 + x + x^2) = x^2$.

Multiplication in $\mathbb{F}_{p^r}$ is done by the following process:

(i) Fix an irreducible polynomial $h(x)$ of degree r with coefficients in $\mathbb{F}_p$. This is to be used for the multiplication of any pair of polynomials.

(ii) For two polynomials f and g, compute $f \cdot g$ in the normal fashion.

(iii) Divide $f \cdot g$ by h. The remainder is defined to be the product of $f(x)$ and $g(x)$ in $\mathbb{F}_{p^r}$.

Example 15. Working over $\mathbb{F}_8$, take $h(x) = x^3 + x + 1$. We want to determine the product of x and x^2. First, we compute $x \cdot x^2 = x^3$. Dividing this by $h(x) = x^3 + x + 1$, we get a quotient of 1 and a remainder of $1 + x$. Therefore $x \cdot x^2 = 1 + x$ in $\mathbb{F}_8$.

Different choices for $h(x)$ will lead to different definitions of multiplication. If a different $h(x)$ is fixed, then the products of polynomials f, g will be different in general. Therefore, it is important that once an $h(x)$ is set for a problem, it is not changed. To indicate which irreducible polynomial we are using, we use the notation $\mathbb{F}_p[x]/\langle h(x)\rangle$ instead of $\mathbb{F}_{p^r}$.

4.3 Constructing Finite Fields Using Primitive Polynomials

We now have enough knowledge of finite fields and polynomials to examine an alternative method of constructing finite fields. In fact, one can use certain irreducible polynomials to write the elements of a the field as powers of a single element. This is key in describing the algebraic interleavers used in turbo codes, where field arithmetic is done.

For clarification, a brief side note must be made before moving on. Typically, the variable is used instead of x when constructing finite fields:

$$\mathbb{F}_8 = \{0,\ 1,\ \ ,\ {}^2,\ 1+\ ,\ 1+\ {}^2,\ \ +\ {}^2,\ 1+\ \ +\ {}^2\}.$$

The following definition describes the type of polynomial used to write write the nonzero elements of a field as powers of a single variable .

Definition 16. We say that an irreducible polynomial $h(x) \in \mathbb{F}_p[x]$ of degree r is called **primitive** if $\mathbb{F}_p[x]/\langle h(x)\rangle = \{0,\ 1,\ \ ,\ {}^2,\ \ldots,\ {}^{p^r-2}\}$.

Example 17. Using the primitive polynomial $h(\ ) = {}^3 + \ + 1$, we can produce the nonzero elements of $\mathbb{F}_8$ using only powers of :

i	^{i}
1	
2	^{2}
3	${}^3 = \ + 1$
4	${}^4 = {}^2 +$
5	${}^5 = {}^3 + \ = {}^2 + \ + 1$
6	${}^6 = {}^3 + {}^2 + \ = \ + 1 + {}^2 + \ = 1 + {}^2$
7	${}^7 = \ + {}^3 = \ + \ + 1 = 1$
8	${}^8 =$

Powers of using primitive polynomial $h(\ ) = {}^3 + \ + 1$

Example 18. If we had not used a primitive polynomial, we would not produce all of the nonzero elements of the field. We simplify the process again by computing powers of . To produce $\mathbb{F}_{16}$ from $\mathbb{F}_2$, fix $h(\ ) = {}^4 + {}^3 + {}^2 + \ + 1$. This polynomial is irreducible over $\mathbb{F}_2$; however, the calculations below show that it is not primitive.

i	^{i}
1	
2	^{2}
3	^{3}
4	${}^3 + {}^2 + \ + 1$
5	${}^4 + {}^3 + {}^2 + \ = {}^3 + {}^2 + \ + 1 + {}^3 + {}^2 + \ = 1$
6	
⋮	⋮

Powers of using the non-primitive polynomial $h(\ ) = {}^4 + {}^3 + {}^2 + \ + 1$

The following theorem promises the existence of primitive polynomials for use in the construction of a finite field.

Theorem 19. *For any positive integer r, there exists a primitive polynomial of degree r over $\mathbb{F}_p$.*

For a proof of theorem 19, see [Pr].

5 Monomial Orders

Another way in which interleavers can be varied is through monomial orderings. We will use monomial orders as a way to order vectors before the components are permuted. These orders are discussed in further detail in Section 6. To begin, we will first look at the definition of total orders.

Definition 20. We say "$<$" is a **total order** on $\mathbb{Z}^n$ if and only if all the following hold:

(i) for all $\neq \quad \in \mathbb{Z}^n$, $<$ or $<$

(ii) for all $\in \mathbb{Z}^n$, $\not<$

(iii) for all $, \ , \ \in \mathbb{Z}^n$, if $<$ and $<$, then $<$

The next definition places the final two conditions on an order for it to be called a monomial order, which can then be used in the construction of an interleaver.

Definition 21. A total order on $\mathbb{Z}^n$ is called a **monomial order** if and only if

(A) for all $\in \mathbb{N}^n$, if $\neq \mathbf{0}$, then $> \mathbf{0}$

(B) for all $, \ , \ \in \mathbb{Z}^n$, if $>$, then $+ \quad > \quad +$

5.1 Types of Monomial Orderings

There are three types of orderings that we will consider; lexicographic, graded lexicographic, and graded reverse lexicographic. Their definitions all follow.

Definition 22. Lexicographic Order is the order such that $<$ if and only if the first non-zero component of $-$ is positive.

Example 23. Let $= (0, 1, 3, -2, 6)$ and $= (0, 1, 3, 10, -20)$. After computing $- \quad =$ $(0, 0, 0, 12, -26)$, we can see that the first nonzero component, 12, is positive. Therefore, $<$.

Definition 24. Graded Lexicographic Order is the order such that $<$ if and only if $\sum_{i=1}^{n} \ _i < \sum_{i=1}^{n} \ _i$ or $\sum_{i=1}^{n} \ _i = \sum_{i=1}^{n} \ _i$ and the leftmost nonzero component of $-$ is positive, where $_i$ and $_i$ represent the ith components of and respectively.

Example 25. Let $\ = (0, 1, 3, -2, 6)$ and $\ = (0, 1, 3, 10, -6)$. Now, $\sum_{i=1}^{n} \ _i = \sum_{i=1}^{n} \ _i =$ 8. Therefore, we need to look at $\ - \ = (0, 0, 0, 12, -12)$. The leftmost nonzero component, 12, is positive. Hence, $\ < \ $.

Definition 26. Graded Reverse Lexicographic Order is the order such that $\ < \ $ if and only if $\sum_{i=1}^{n} \ _i < \sum_{i=1}^{n} \ _i$ or $\sum_{i=1}^{n} \ _i = \sum_{i=1}^{n} \ _i$ and the rightmost nonzero component of $\ - \ $ is negative, where $_i$ and $_i$ represent the ith component of and respectively.

Example 27. Let $\ = (0, 1, 3, -2, 6)$ and $\ = (0, 1, 3, 10, -6)$. Now, $\sum_{i=1}^{n} \ _i = \sum_{i=1}^{n} \ _i =$ 8. Consequently, we need to look at $\ - \ = (0, 0, 0, 12, -12)$. The rightmost nonzero component, -12, is negative so $\ < \ $.

We now note a theorem regarding monomial orderings that was crucial in the program we used to create turbo code interleavers. The interleaver creation program that we created uses an ordering matrix as one of its arguments.

Theorem 28. *Let M be an n by n invertible matrix with nonnegative, real entries. For $\ , \ \in \mathbb{N}^n$, define $\ < \ $ if and only if the leftmost nonzero component of $M(\ - \ )$ is positive. Then, this order is a monomial order.*

For a proof of Theorem 28, see [JoMo]. From Theorem 28, one can see that there are infinitely many monomial orderings, since the entries of a matrix can be created from infinitely many numbers.

6 Constructing Algebraic Interleavers with Finite Fields

In this section, we will construct interleavers using the tools of the previous sections. Our permutation will consist of a composition of maps:

$$\mathbb{Z}_{p^r} \xrightarrow{\pi_1} (\mathbb{Z}_p)^r \xrightarrow{\pi_2} \mathbb{F}_{p^r} \xrightarrow{\pi_3} \mathbb{F}_{p^r} \xrightarrow{\pi_2^{-1}} (\mathbb{Z}_p)^r \xrightarrow{\pi_1^{-1}} \mathbb{Z}_{p^r}.$$

The following steps detail the procedure for constructing an algebraic interleaver:

(i)A Using base p representation: Rewrite each integer c in $\mathbb{Z}_{p^r}$ in its base p representation: $c = c_0 + c_1 p^1 + c_2 p^2 + ... + c_{r-1} p^{r-1}$. Then place the coefficients of c into a vector: $\pi_1(c) = (c_0, c_1, ..., c_{r-1})$.

(i)B Using a monomial ordering: For a given monomial ordering, order the vectors of $(\mathbb{Z}_p)^r$ in ascending order: $w_0 < ... < w_{p^r-1}$. Then define $\pi_1(c) = w_c$.

(i) Define $\pi_2(w_0, w_1, ..., w_{r-1}) = \sum_{i=0}^{r-1} w_i \ ^i$.

(ii) Choose $\pi_3 : \mathbb{F}_{p^r} \to \mathbb{F}_{p^r}$ to be any permutation of $\mathbb{F}_{p^r}$. For our studies, π_3 is usually of the form $x \mapsto x^i$ for some i.

Example 29. Here is an example using the lexicographic order and the permutation $x \mapsto x^3$:

$$\mathbb{F}_8 \longrightarrow \mathbb{F}_8 \text{ by } x \mapsto x^3$$
$$\mathbb{F}_8 = \mathbb{F}_2/\langle x^3 + x + 1\rangle$$

$\mathbb{Z}_{2^3}$	$(\mathbb{Z}_2)^3$	$\mathbb{F}_{2^3}$	$\mathbb{F}_{2^3}$	$(\mathbb{Z}_2)^3$	$\mathbb{Z}_{2^3}$
0	$(0,0,0)$	0	0	$(0,0,0)$	0
1	$(0,0,1)$	$\;^2$	$1 + \;^2$	$(1,0,1)$	5
2	$(0,1,0)$		$1+$	$(1,1,0)$	6
3	$(0,1,1)$	$+ \;^2$	$1 + \; + \;^2$	$(1,1,1)$	7
4	$(1,0,0)$	1	1	$(1,0,0)$	4
5	$(1,0,1)$	$1 + \;^2$	$+ \;^2$	$(0,1,1)$	3
6	$(1,1,0)$	$1+$	$\;^2$	$(0,0,1)$	1
7	$(1,1,1)$	$1 + \; + \;^2$		$(0,1,0)$	2

6.1 Monomial Permutations

In our research, we tested only monomial permutations rather than polynomial permutations. We originally tried to create permutation polynomials, however these took both a long time to calculate and run in simulations. Because monomial permutations were quicker to generate and test, they were the most practical for us to use.

Definition 30. A **permutation monomial** is a monomial that is also a permutation in $\mathbb{Z}_p \rightarrow \mathbb{Z}_p$ where p is prime.

The following theorems give the criteria for permutation monomials for $\mathbb{Z}_p \rightarrow \mathbb{Z}_p$ when p is prime. These provided us with an easier way of generating permutations to use, along with an assuredness that they were indeed permutations.

Theorem 31. *If p is prime and i is an integer relatively prime to $p-1$, then $f_i : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$ given by $f_i(x) = x^i$ is a permutation.*

Proof. Suppose that p is prime and i is an integer relatively prime to p. Since i is relatively prime to p, the $gcd(i, p-1) = 1$. This implies that there exist a, b $\in \mathbb{Z}$ such that $ai+b(p-1) = 1$. Then $x^{(ai+b(p-1))} = x^1$. Also, $gcd(x,p) = 1$ when $x \in \{1, ..., p-1\}$. By Fermat's Little Theorem, $x^{p-1} \equiv 1 \mod p$. Then,

$$\begin{aligned} x &= x^1 \\ &= x^{(ai+b(p-1))} \\ &= x^{ai}x^{b(p-1)} \\ &= x^{ai}1^b \\ &= x^{ai}. \end{aligned}$$

When $x = 0$, we have $f_i(0) = 0$. As we are working in a finite field, f_i is invertible if we can find j such that $x^{ij} = x$. Furthermore, if f_i is invertible, then this implies that it is a permutation. We know f_i is invertible because $x^{ai} = x$. So, it is a permutation. $\square$

In fact, a more general result holds, which is given in the following theorem.

Theorem 32. *The monomial $x^i \in \mathbb{F}_q[x]$ is a permutation monomial of $\mathbb{F}_q$ if and only if $gcd(i, q-1) = 1$.*

For a proof of Theorem 32 see [JoMo].

7 Dispersion

In order to study the "randomness" of an interleaver, we must calculate the dispersion. This property will help us compare the distance between two values in $\mathbb{Z}_q$ and the distance between their permuted images. A high dispersion indicates a variety of the permuted distances between the elements. The first step of calculating the dispersion is acquiring a complete list of differences.

Definition 33. Given a permutation π of $\mathbb{Z}_q$, the **list of differences** of π is defined to be the set

$$D(\pi) \ = \ \{(j-i,\ \pi(j)-\pi(i)) \mid 0 \ \leq \ i \ < \ j \ < \ q\}.$$

Example 34. We compute the list of differences for the following permutation:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & 0 & 3 & 1 \end{pmatrix}.$$

We begin by calculating all of the pairs for which the input values are of distance 1 away from each other:

$$(1-0, \pi(1)-\pi(0)) = (1, -2),$$
$$(2-1, \pi(2)-\pi(1)) = (1, 3),$$
$$(3-2, \pi(3)-\pi(2)) = (1, -5),$$
$$(4-3, \pi(4)-\pi(3)) = (1, 3),$$

and

$$(5-4, \pi(5)-\pi(4)) = (1, -2).$$

So, the first elements of the set $D(\pi)$ are $(1, -2), (1, 3)$, and $(1, -5)$. Because the list of differences is a set, the repeated elements are not to be counted twice. We can continue calculating the remaining pairs by checking inputs that are $2, 3, 4$, and 5 away from each other. Hence, the permutation has the following list of differences:

$$D(\pi) = \{(1, -2), (1, 3), (1, -5), (2, 1), (2, -2), (3, -4), (3, 1), (4, -1), (5, -3)\}.$$

Therefore,

$$|D(\pi)| = 9.$$

Definition 35. Given a permutation π, the **dispersion** of π is given by

$$\frac{|D(\pi)|}{\binom{q}{2}} = \frac{|D(\pi)|}{\frac{q(q-1)}{2}}$$

From the previous example, we see that $q = 6$ because the permutation acts on 6 objects. Furthermore, we know that the cardinality of $D(\pi)$ is 9. We can now calculate the dispersion using the definition:

$$\frac{|D(\pi)|}{\frac{q(q-1)}{2}} = \frac{9}{\frac{6(5)}{2}} = \frac{9}{\frac{30}{2}} = \frac{3}{5}.$$

7.1 Conjecture

Pursuant to a question asked during a presentation, we decided to investigate whether or not we could determine what proportion of permutations of a given size have a dispersion around 0.8. This proved to be of particular interest, as permutations with this dispersion tend to have the lowest bit error rates.

Upon looking into this, we have discovered that the mean dispersion for all permutations of a given size are around .81. As one increases the size of the set acted on, the average dispersion appears to converge to a number between 0.813 and .814. The standard deviation away from this mean also decreases, and appears to go to zero, as the permutation length gets larger.

We obtained the following data by testing samples of permutations of different lengths and calculating the average dispersion based on these samples. We also calculated the standard deviation away from this mean. The first chart has been calculated exactly, by testing all possible permutations of the given permutation sizes. The data in the second and third charts was obtained using sample data for the given block lengths. For sizes of 10 through 90, we used a sample size of 100000 permutations selected randomly of all possible permutations, and for lengths of 100 to 1000, we used random sample sizes of 10000 from all possible permutations. The reason for this is that as the permutation size increases, the time to calculate the data also increases.

Exact Average Dispersion for Permutation Sizes Between 2 and 10

Length	Average Dispersion	Standard Deviation
2	1	0
3	0.8889	0.1721
4	0.8472	0.1766
5	0.8467	0.1495
6	0.8383	0.1274
7	0.8355	0.1000
8	0.8319	0.0866
9	0.8301	0.0737
10	0.8280	0.0657

Average Dispersion for Permutation Sizes Between 20 and 90 Based on a Random Sample Size of 100000

Length	Average Dispersion	Standard Deviation
20	0.8206	0.0309
30	0.8183	0.0203
40	0.8171	0.0151
50	0.8163	0.0121
60	0.8158	0.0100
70	0.8155	0.0086
80	0.8153	0.0075
90	0.8151	0.0067

Average Dispersion for Permutation Sizes Between 100 and 1000 Based on a Random Sample Size of 10000

Length	Average Dispersion	Standard Deviation
100	0.8150	6.108×10^{-3}
200	0.8143	3.029×10^{-3}
300	0.8141	1.994×10^{-3}
400	0.8140	1.528×10^{-3}
500	0.8139	1.229×10^{-3}
600	0.8139	1.018×10^{-3}
700	0.8138	8.707×10^{-4}
800	0.8138	7.870×10^{-4}
900	0.8137	6.963×10^{-4}
1000	0.8137	6.217×10^{-4}

It should be noted that in almost all permutation block lengths analyzed, we obtained several permutations with a dispersion much lower than this mean. While we are not sure as to why this is, it seems to be an interesting characteristic of algebraic permutations that have a predetermined structure. This could be useful, as it gives some insight as to what type of algebraic permutation one should choose based on dispersion.

8 Spread

We will now move on to examine another useful property of interleavers: spread. Given a permutation π, it is sometimes desirable to determine the relationship between a pair of inputs and their corresponding output values. Specifically, one might ask how spread apart two outputs are if their inputs are a certain distance apart. We can study this property by using a few definitions of spread.

Definition 36. Let $\pi : \mathbb{Z}_q \to \mathbb{Z}_q$ be a permutation. The **spread** of π is the largest integer s such that

$$|i-j| \ < \ s \ \Rightarrow \ |\pi(i)-\pi(j)| \ \geq \ s, \text{ for } 1 \ \leq \ s \ \leq \ q \tag{1}$$

where i, j are input values such that $i \neq j$.

Proposition 37. *The spread of a permutation is always greater than or equal to 1.*

Proof. For $s = 1$, we must test all input values that have a distance of less than 1 between them. Since $i \neq j$, there will never exist two input values that have a distance of 0 between them. Thus, the antecedent of (1) is always false. Consequently, the statement is always true for $s = 1$. Hence, the spread of any permutation must be greater than or equal to 1. □

Proposition 38. *If $s \geq 2$ and satisfies $|i - j| < s \Rightarrow |\pi(i) - \pi(j)| \geq s$ for all $i \neq j$, then for all $s' \leq s$,*

$$|i - j| < s' \Rightarrow |\pi(i) - \pi(j)| \geq s'$$

for all $i \neq j$.

Proof. Let $s \geq 2$ and suppose $|i - j| < s \Rightarrow |\pi(i) - \pi(j)| \geq s$ for all $i \neq j$. Suppose for some $s' \leq s$, $|i - j| < s'$. Since $|i - j| < s' \leq s$, $|\pi(i) - \pi(j)| \geq s \geq s'$. Therefore, for all $s' \leq s$, $|i - j| < s' \Rightarrow |\pi(i) - \pi(j)| \geq s'$ for all $i \neq j$.

□

Example 39. For the following permutation, we calculate spread:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & 0 & 3 & 1 \end{pmatrix}.$$

We look at all the pairs for which $|i - j| < 2$, and check to see if $|\pi(i) - \pi(j)| \geq 2$. Thus,

$$|\pi(1) - \pi(0)| = |2 - 4| = 2,$$

$$|\pi(2) - \pi(1)| = |5 - 1| = 4,$$

$$|\pi(3) - \pi(2)| = |0 - 5| = 5,$$

$$|\pi(4) - \pi(3)| = |3 - 0| = 3,$$

and

$$|\pi(5) - \pi(4)| = |1 - 3| = 2.$$

As we can see, all pairs of inputs that are less than 2 away from each other have a permuted distance of at least 2 away from each other. Hence, $s = 2$ is a possible spread. We must now consider the case for which $s = 3$. We examine all the pairs for which $|i - j| < 3$ and check if $|\pi(i) - \pi(j)| \geq 3$. First we will consider input values that are of distance 2 away from each other; if the definition holds for these inputs, we will check input values that are of distance 1 away from each other. Thus

$$|\pi(2) - \pi(0)| = |4 - 5| = 1,$$

$$|\pi(3) - \pi(1)| = |2 - 0| = 2,$$

$$|\pi(4) - \pi(2)| = |5 - 3| = 2,$$

and

$$|\pi(5) - \pi(3)| = |0 - 1| = 1.$$

We can see that the input values 2 and 0 produce output values that are only of distance 1 away from each other, and so the spread cannot possibly be 3. Therefore, by Proposition 38, the spread of π is 2. $s = 2$.

8.1 The Upper Bound for Spread

We will now present several lemmas and theorems with respect to the upper bound for spread. Theorem 43 gives an upper bound for spread that we originally found in our research; the following example and lemmas prepare for the theorem.

Consider the statement:

$$|i - j| < s \Rightarrow |\pi(i) - \pi(j)| \geq s,$$

and thus define

$$\begin{aligned} A &= \{(i,j) \in \mathbb{Z}_q \times \mathbb{Z}_q |\ i \neq j,\ 1 \leq i,j \leq q\}, \\ B_s &= \{(i,j) \in\ \mathbb{Z}_q \times \mathbb{Z}_q \mid i \neq j \text{ and } |i-j| < s\},\ \text{and} \\ C_s &= \{(i,j) \in\ \mathbb{Z}_q \times \mathbb{Z}_q \mid i \neq j \text{ and } |\pi(i) - \pi(j)| \geq s\}. \end{aligned}$$

Then $|A|\ = q^2 - q$, since there are $q^2 - q$ possible pairs (i,j) where $i \neq j$. We now turn to a brief example that will help us determine a formula for B_s.

Example 40. For $\mathbb{Z}_5 \times \mathbb{Z}_5$, we can list the pairs (i,j).

A list of all pairs (i,j), where $1 \leq i,j \leq q$:

(0,0) **(1,0)** *(2,0)* (3,0) (4,0)

(0,1) (1,1) **(2,1)** *(3,1)* (4,1)

(0,2) **(1,2)** (2,2) **(3,2)** *(4,2)*

(0,3) *(1,3)* **(2,3)** (3,3) **(4,3)**

(0,4) (1,4) *(2,4)* **(3,4)** (4,4)

The bolded sets make up $B_2 = \{(i,j) \in \mathbb{Z}_5 \times \mathbb{Z}_5 \mid i \neq j \text{ and } |i-j| < 2\}$, since they are composed of inputs less than 2 apart. There are $4+4 = 8$ sets that are bolded. Thus $|B_2| = 2(4) = 2(5-1)$. In the above chart, another chart for $s = 3$ is included, which combines the bolded diagonals with the italicized diagonals. Therefore, there are $8+3+3 = 8+6 = 14$ sets that hold true. Thus $|B_3| = 2(4)+2(3) = 2(5-1)+2(5-2)$. Similarly, for $s = 4$, there are $14+2+2 = 16$ sets that hold true, so $|B_4| = 2(4)+2(3)+2(2) = 2(5-1)+2(5-2)+2(5-3)$. We write the preceding formulas vertically to see the pattern emerging:

$$\begin{aligned} s &= 2 : |B_2| = 2(q-1) \\ s &= 3 : |B_3| = 2(q-1) + 2(q-2) \\ s &= 4 : |B_4| = 2(q-1) + 2(q-2) + 2(q-3) \\ &\vdots \end{aligned}$$

The following lemma results from generalizing the specific values for $|B_s|$ in the preceding example.

Lemma 41. *For any q and s, we find $|B_s| = 2\sum_{k=1}^{s-1}(q-k) = -2q + s + 2qs - s^2$.*

Now that we have computed $|B_s|$, we turn find the cardinality of C_s.

Lemma 42. *For any q and s, we find $|C_s| = q^2 + q - s - 2qs + s^2$.*

Proof. Let π be a permutation. We can write C_s as:

$$\begin{aligned} C_s &= \{(i,j) \in \mathbb{Z}_q \times \mathbb{Z}_q \mid i \neq j \text{ and } |\pi(i) - \pi(j)| \geq s\} \\ &= \{(\pi^{-1}(i), \pi^{-1}(j)) \in \mathbb{Z}_q \times \mathbb{Z}_q \mid i \neq j \text{ and } |i-j| \geq s\} \\ &= \{(i,j) \in \mathbb{Z}_q \times \mathbb{Z}_q \mid i \neq j \text{ and } |i-j| \geq s\} \end{aligned}$$

It is now clear that, for all s, $A = B_s \cup C_s$. Then $|A| = |B_s \cup C_s| = |B_s| + |C_s|$. Substituting in the respective values and solving for $|C_s|$, we have

$$q^2 - q = 2\sum_{j=1}^{s-1}(q-j) + |C_s|,$$

and so

$$\begin{aligned} |C_s| &= q^2 - q - 2\sum_{j=1}^{s-1}(q-j) \\ &= q^2 + q - s - 2qs + s^2. \end{aligned}$$

□

Theorem 43. ***[AMSSI 2006]*** *An upper bound for spread is $\lfloor \frac{1}{2}(1 + 2q - \sqrt{2q^2 - 2q + 1}\rfloor$, where q is the length of the permutation.*

Proof. Let $q : \mathbb{Z}_q \to \mathbb{Z}_q$ be a permutation with spread s.

We consider the inequality $|B_s| \leq |C_s|$, using the values for $|B_s|$ and $|C_s|$ as developed in Lemmas 41 and 42. Since $|B_s| \leq |C_s|$, we have

$$-2q + s + 2qs - s^2 \leq q^2 + q - s - 2qs + s^2.$$

Thus,

$$\begin{aligned} 0 &\leq q^2 + 3q - 2s - 4qs + 2s^2 \\ &= 2s^2 + (-2 - 4q)s + (q^2 + 3q). \end{aligned}$$

We solve the corresponding equality for s, using the quadratic equation:

$$s = \frac{(2+4q) \pm \sqrt{(2+4q)^2 - 4\cdot 2\cdot (q^2+3q)}}{2\cdot 2}.$$

Simplifying, we have

$$s = \frac{1}{2}\left((1+2q) \pm \sqrt{2q^2-2q+1}\right).$$

Since $\frac{1}{2}\left((1+2q) + \sqrt{2q^2-2q+1}\right) \geq \frac{1}{2}\left((1+2q) - \sqrt{2q^2-2q+1}\right)$, by Proposition 38, we need only consider $s = \frac{1}{2}\left((1+2q) - \sqrt{2q^2-2q+1}\right)$.

Reintroducing the inequality leads to $s \leq \frac{1}{2}\left((1+2q) - \sqrt{2q^2-2q+1}\right)$. Since the right hand side may not be an integer, we take the floor of this function to find our upper bound for spread is

$$\left\lfloor \frac{1}{2}\left((1+2q) - \sqrt{2q^2-2q+1}\right) \right\rfloor.$$

□

It can also be shown that a more refined upper bound for the spread of a permutation of $\mathbb{Z}_q$ is $\lfloor \sqrt{q} \rfloor$. The lemma below shows that there exists a permutation of $\mathbb{Z}_q$ with spread $\sqrt{q}$ whenever q is a perfect square.

Lemma 44. ***[AMSSI 2006]*** *If $q = s^2$, then there exists a permutation with spread s.*

Proof. Let $i, j \in \mathbb{Z}_q$ such that $|i-j| < s$. Without loss of generality, $i > j$. For all input values $i \neq j$, we can write i and j in base s; i.e., $i = a_0 + sa_1$, and $j = b_0 + sb_1$, where $0 \leq a_n, b_n \leq s-1$. We define $\pi : \mathbb{Z}_q \to \mathbb{Z}_q$ by $\pi(a_0 + sa_1) = (s-1-a_1) + sa_0$ and $\pi(b_0 + sb_1) = (s-1-b_1) + sb_0$.

Since $|i-j| < s$ and $i > j$, either $a_1 = b_1$ or $a_1 = b_1 + 1$. We consider the two cases separately.

Case 1: $a_1 = b_1$. Now,

$$|i-j| = |a_0 + sa_1 - b_0 - sa_1| = |a_0 - b_0|,$$

and by hypothesis, $|a_0 - b_0| < s$. Since $i \neq j$, $|a_0 - b_0| \geq 1$. Thus,

$$\begin{aligned} |\pi(i) - \pi(j)| &= |\pi(a_0 + sa_1) - \pi(b_0 + sb_1)|, \\ &= |(s-1-a_1) + sa_0 - (s-1-a_1) + sb_0|, \\ &= s|(a_0 - b_0)|, \\ &\geq s. \end{aligned}$$

Case 2: $a_1 = b_1 + 1$. So

$$|i-j| = |a_0 + s(b_1+1) - b_0 - sb_1| = |a_0 - b_0 + s|$$

and by hypothesis, $|a_0 - b_0 + s| < s$. Thus $(a_0 - b_0) < 0$, so $a_0 < b_0$, and by definition $0 < |a_0 - b_0| \leq s$. When comparing the respective images of i and j,

$$\begin{aligned} |\pi(i) - \pi(j)| &= |(s-1-b_1-1) + sa_0 - (s-1-b_1) - sb_0)| \\ &= |-1 + s(a_0 - b_0)| \end{aligned}$$

and since both -1 and $s(a_0 - b_0)$ are negative,

$$|-[1 + s(b_0 - a_0)]| = 1 + s(b_0 - a_0) \geq s$$

$\square$

Using the ideas of Lemma 44, we can prove a more general result, namely, that for any positive integer q, there exists a permutation with spread $\lfloor\sqrt{q}\rfloor$. We begin with an example that helps motivate the ideas in the proof.

Example 45. We can design permutations that achieve the maximum spread for a given permutation length q. In this example, we take $q = 9$. Then we define each R_i for $0 \leq i \leq 2$ as follows:

$$\begin{aligned} R_2 &= (2, 5, 8) \\ R_1 &= (1, 4, 7) \\ R_0 &= (0, 3, 6) \end{aligned}$$

and so $R = (2, 5, 8, 1, 4, 7, 0, 3, 6)$, in which case

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 5 & 8 & 1 & 4 & 7 & 0 & 3 & 6 \end{pmatrix}$$

By constructing π in this manner, we create a permutation. In this example, π has a spread of $3 = \sqrt{9}$.

Lemma 46. ***[AMSSI 2006]*** *If $q \geq s^2$, then there exists a permutation, π, with spread at least s.*

Proof. Generalizing the notions developed in Example 45, for each integer i such that $0 \leq i \leq s-1$, define R_i to be the finite sequence of elements of $\mathbb{Z}_q$ given by $R_i = i, i+s, i+2s, ..., i+m_i s$, where m_i is the largest integer such that $i + m_i s \leq q - 1$. Then, define R to be the sequence that is the concatenation of the sequences $R_{s-1}, R_{s-2}, ..., R_2, R_1, R_0$. Define $\pi : \mathbb{Z}_q \Rightarrow \mathbb{Z}_q$ to be the permutation that sends $0, 1, 2, ..., q - 1$ to the terms of R, respectively.

The terms on the ends of each row R_i are the largest in their row, with the maximum of these being $q - 1$ (since all elements are in the range $0, ..., q - 1$) and smallest being $q - s$, because there are s rows. Thus, $i + m_i s \geq q - s$. Since $1 \leq i \leq s - 1$ and, by hypothesis, $q \geq s^2$, we have

$$\begin{aligned} (s-1) + m_i s &\geq i + m_i s \\ &\geq q - s \\ &\geq s^2 - s, \end{aligned}$$

and so

$$m_i s \geq s^2 - 2s + 1.$$

Thus,

$$m_i \geq \ s - 2 + \tfrac{1}{s}. \tag{2}$$

We now proceed to show that the spread of π is s. Let $k, l \in \mathbb{Z}_q$ such that $k \neq l$ and $|k - l| < s$. Without loss of generality, suppose $l < k$. Since $|k - l| < s$ and each row R_i has at least s terms, either $\pi(k)$ and $\pi(l)$ are in the same row or they are in adjacent rows R_i, R_{i-1}. We consider these two cases separately.

If $\pi(k), \pi(l) \in R_i$ for some index i, then for some $\delta,$ such that $\delta >$, $\pi(k) = i + \delta s$ and $\pi(l) = i + \ s$. Then $|\pi(k) - \pi(l)| = (\delta - \ )s > s$.

Otherwise, if $\pi(k)$ is a term of R_i and $\pi(l)$ is a term of R_{i-1} (where $1 \leq i \leq s - 1$), then for some $\delta,$, $\pi(k) = i + \delta s$ and $\pi(l) = (i - 1) + \ s$. The last term of R_i is $i + m_i s$ and so

$$\pi^{-1}(i + m_i s) - k = \pi^{-1}(i + m_i s) - \pi^{-1}(i + \delta s) = m_i - \delta. \tag{3}$$

Moreover, since $i - 1$ is the first term of R_{i-1}, and $i + m_i s$ is the last term of R_i,

$$\pi^{-1}(i - 1) - \pi^{-1}(i + m_i s) = 1. \tag{4}$$

Since $\pi(l) = (i - 1) + \ s$ is also a term of R_{i-1},

$$l - \pi^{-1}(i - 1) = \pi^{-1}((i - 1) + \ s) - \pi^{-1}(i - 1) = \tag{5}$$

Putting (3), (4), and (5) together,

$$\begin{aligned} l - k &= \big(l - \pi^{-1}(i-1)\big) + \big(\pi^{-1}(i-1) - \pi^{-1}(i + m_i s)\big) + \big(\pi^{-1}(i + m_i s) - k\big) \\ &= \quad + 1 + m_i - \delta. \end{aligned}$$

Since we have $l - k < s$, $\quad - \delta + m_i + 1 < s$, and so

$$- \delta + m_i + 2 \leq s.$$

Thus,

$$- \delta \ \leq \ s - 2 - m_i \tag{6}$$

Putting (2) and (6) together, we have

$$\begin{aligned} - \delta &\leq s - 2 - m_i \\ &\leq s - 2 - s + 2 - \frac{1}{s} \\ &\leq -\frac{1}{s} \end{aligned}$$

Rearranging the last inequality leads to

$$\delta - \quad \geq \frac{1}{s}. \tag{7}$$

Then, $\delta >$ by (7), which implies $\delta - \quad \geq 1$, and so we have

$$\begin{aligned} |\pi(k) - \pi(l)| &= |(i + \delta s) - ((i - 1) + \ s)| \\ &= 1 + (\delta - \ )s \\ &\geq s. \end{aligned}$$

$\square$

It has now been shown that, for a given spread s, one can find a sufficiently large field such that the spread of at least one permutation of the field is s. This might lead one to wonder if spread has an upper bound related to the size of the permutation. The following theorem shows that a bound exists.

Theorem 47. ***[AMSSI 2006]*** *For any permutation π of $\mathbb{Z}_q$, the spread of $\pi \leq \lfloor\sqrt{q}\rfloor$. In fact, by Lemma 46, there exists a permutation with spread equal to $\lfloor\sqrt{q}\rfloor$.*

Proof. Let π be a permutation of $\mathbb{Z}_q$ with spread s. Consider and interval $I = \{k, k+1, ..., k+(s-1)\} \subseteq \mathbb{Z}_q$ such that $s-1 \in I$, and define $\pi(I) = \{\pi(j)|j \in I\}$. Because the spread of π is s, and since all the elements of I are within a distance of $s-1$ of each other, all of the images in $\pi(I)$ must be a distance of at least s from one another. Therefore, $0, 1, ..., s-2$ are not in $\pi(I)$. Write the s elements of I in ascending order, i.e., $\pi(I) = \{ {}_1, {}_2, ..., {}_s\}$ where ${}_1 < {}_2 < ... < {}_s$. Since $0, 1, ..., s-2$ are not in $\pi(I)$, $s-1$ is the smallest entry of $\pi(I)$. Therefore, ${}_1 = s-1$. Since the elements of $\pi(I)$ are at least s apart from each other, ${}_j - {}_{j-1} \geq s$ for all $j = 2, 3, ..., s$. Thus

$$\begin{aligned} {}_s - {}_1 &= ({}_s - {}_{s-1}) + ({}_{s-1} + {}_{s-2}) + \cdots + ({}_2 - {}_1) \\ &\geq (s-1)s \\ &= s^2 - s. \end{aligned}$$

Rearranging, we obtain

$$\begin{aligned} {}_s &\geq {}_1 + s^2 - s \\ &= s - 1 + s^2 - s \\ &= s^2 - 1. \end{aligned}$$

Therefore, since ${}_s \in \mathbb{Z}_q$, we have $q - 1 \geq {}_s$, and so $q - 1 \geq s^2 - 1$. Thus, $q \geq s^2$, so $\sqrt{q} \geq s$. Since s is an integer, it follows that

$$s \leq \lfloor\sqrt{q}\rfloor.$$

□

8.2 Other Measures of Spread

We will now consider some other measures of spread in order to study the distance between permuted values. There are four measures that can be examined: spreading factors, extreme spreading factors, s-parameters, and spread factors.

Definition 48. The **spreading factors** of $\pi = \mathbb{Z}_q \to \mathbb{Z}_q$ are all points (s, t) that satisfy

$$|i - j| < s \Rightarrow |\pi(i) - \pi(j)| \geq t, \tag{8}$$

for all $i, j \in \mathbb{Z}_q$ such that $i \neq j$.

Calculating the spreading factors is an extremely long process without a computer program, as the following example shows.

Example 49.

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 0 & 3 & 6 & 9 & 1 & 4 & 7 & 10 & 2 & 5 & 8 \end{pmatrix}$$

We consider $(2,3)$ and determine if it is a spreading factor. That is, we test all pairs of input values (i,j) such that $|i-j|<2$ and determine if the distance between their output values $|\pi(i)-\pi(j)|$ is at least 3. Starting with input values $(2,1)$, we see that,

$$|2-1|<2 \text{ and } |\pi(2)-\pi(1)| = |6-3| = 3 \geq 2$$

The same process has to be repeated for input values $(3,2),(4,3),...,(10,9)$ because they are pairs of indices of distance less than 2 away from each other. If (8) is satisfied for all these pairs in addition to $(2,1)$, then we know that $(2,3)$ is a spreading factor. In order to get all possible spreading factors, this process must be completed for all pairs (s,t) where $s,t \in \mathbb{Z}_q$. After checking all possible spreading factors, we can complete a graph of all the spreading factors that were found. The following graph is a plot of all the spreading factors of π; each one is marked with an x.

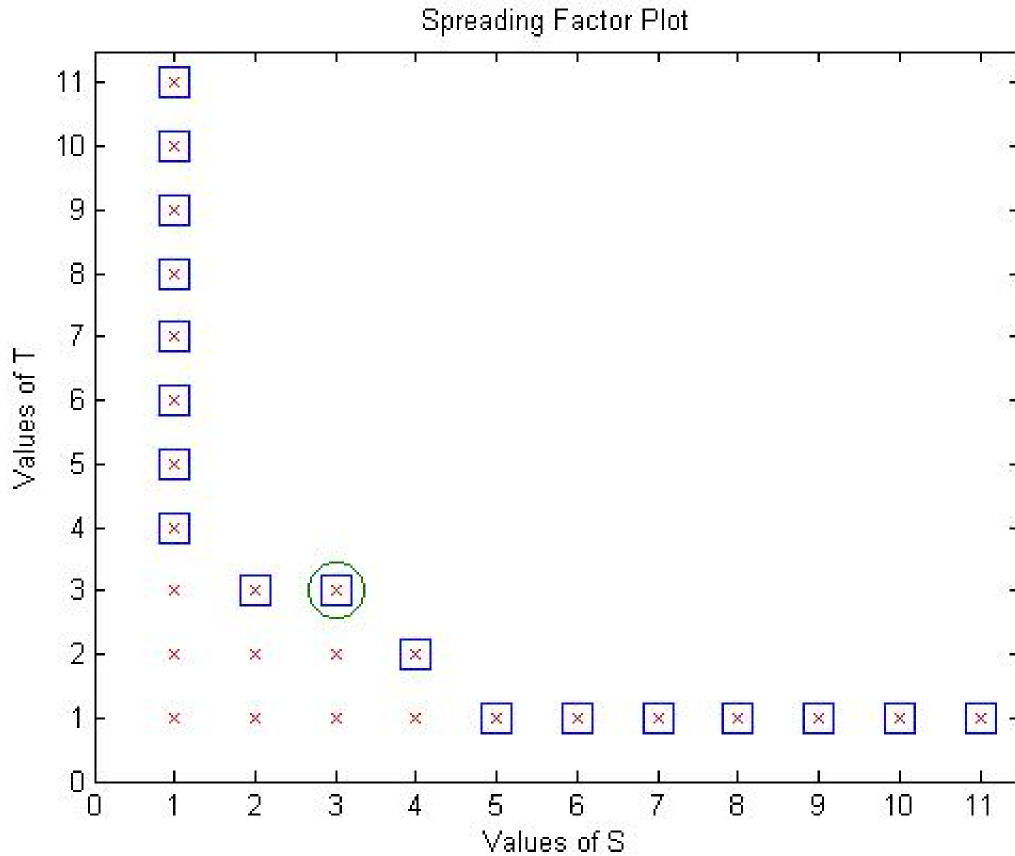


Figure 5: Plot of Spreading Factors, Extreme Spreading Factors, and the s-Parameter

We can study this graph further and gain more insight into spread if we introduce the following properties: extreme spreading factors and s-parameters.

Definition 50. A spreading factor (s, t) is an **extreme spreading factor** if either $(s+1, t)$ or $(s, t+1)$ is not a spreading factor.

Example 51. The spreading factors with boxes around them in Figure 5 are extreme spreading factors because spreading factors either do not exist directly to the right or directly above them.

Definition 52. The **s-parameter** is the maximum value of s such that for some $s \leq t$, (s, t) is an extreme spreading factor.

Example 53. In order to find the s-parameter in Figure 5, we consider all the extreme spreading factors. The largest value of s such that $s \leq t$ is $s = 3$. Therefore, the s-parameter is 3 and is circled in Figure 5.

Definition 54. For all input values $i < j$ and output values $\pi(i), \pi(j)$, and q representing the length of the permutation, let

$$\delta(i, j) = |i - j|_q + |\pi(i) - \pi(j)|_q$$

where

$$|x - y|_q = \min[(x - y) \bmod q, (y - x) \bmod q]$$

Another measure of spread we examined was the spread factor, not to be confused with spreading factors. The spread factor is not equal to spread, however, it seems that when one increases, they both increase, due to the definitions of both.

Definition 55. The **spread factor** of a permutation π is the minimum $\delta(i, j)$ over all $i < j$.

Example 56. Consider the following permutation:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & 0 & 3 & 1 \end{pmatrix}$$

In order to find the spread factor, we must first make a list of all the pairs of input values for which $i < j$.

(0, 1) (0, 2) (0, 3) (0, 4) (0, 5)
(1, 2) (1, 3) (1, 4) (1, 5)
(2, 3) (2, 4) (2, 5)
(3, 4) (3, 5)
(4, 5)

Now that we have all possible pairs, we can now calculate $\delta(i, j)$ for each pair.

$$\begin{array}{lllllllll}
\delta(0,1) & = & |0-1|_6 + |\pi(0) - \pi(1)|_6 & = & 3 & \delta(0,2) & = & |0-2|_6 + |\pi(0) - \pi(2)|_6 & = & 3 \\
\delta(0,3) & = & |0-3|_6 + |\pi(0) - \pi(3)|_6 & = & 5 & \delta(0,4) & = & |0-4|_6 + |\pi(0) - \pi(4)|_6 & = & 3 \\
\delta(0,5) & = & |0-5|_6 + |\pi(0) - \pi(5)|_6 & = & 4 & \delta(1,2) & = & |1-2|_6 + |\pi(1) - \pi(2)|_6 & = & 4 \\
\delta(1,3) & = & |1-3|_6 + |\pi(1) - \pi(3)|_6 & = & 4 & \delta(1,4) & = & |1-4|_6 + |\pi(1) - \pi(4)|_6 & = & 4 \\
\delta(1,5) & = & |1-5|_6 + |\pi(1) - \pi(5)|_6 & = & 3 & \delta(2,3) & = & |2-3|_6 + |\pi(2) - \pi(3)|_6 & = & 2 \\
\delta(2,4) & = & |2-4|_6 + |\pi(2) - \pi(4)|_6 & = & 4 & \delta(2,5) & = & |2-5|_6 + |\pi(2) - \pi(5)|_6 & = & 5 \\
\delta(3,4) & = & |3-4|_6 + |\pi(3) - \pi(4)|_6 & = & 4 & \delta(3,5) & = & |3-5|_6 + |\pi(3) - \pi(5)|_6 & = & 3 \\
\delta(4,5) & = & |4-5|_6 + |\pi(4) - \pi(5)|_6 & = & 3 & & & & &
\end{array}$$

We can see that the pair with the minimum value of $\delta(i, j)$ is $(2, 3)$, with a value of 2. Hence, the spread factor is 2 for this permutation.

8.3 Spread and S-Parameters Coincide: Our Theoretical Results

It can be shown that the s-parameter of a permutation is equal to its spread. Before we state this theorem and its proof, we will first prove three lemmas. The first lemma deals with a bound for the spreading factor.

Lemma 57. ***[AMSSI 2006]*** *The sth column of the spreading factor plot is bounded for $s \geq 2$; that is for all $s \geq 2$, there are only finitely many values of t such that (s, t) is a spreading factor.*

Proof. For any permutation of $\mathbb{Z}_q$, the largest distance any two $i, j \in \mathbb{Z}_q$ can be from one another is $q - 1$. This is also true for any two $\pi(i), \pi(j)$. By the definition of spreading factor, $|i - j| < s \Rightarrow |\pi(i) - \pi(j)| \geq t$. The value of t can be at most $q - 1$. Thus, there are finitely many values of t such that (s, t) is a spreading factor. □

The following lemma guarantees the existence of an extreme spreading factor based on the existence of a spreading factor (s, s).

Lemma 58. ***[AMSSI 2006]*** *If (s, s) is a spreading factor, then there exists $s \leq t$ such that (s, t) is an extreme spreading factor.*

Proof. Let (s, s) be a spreading factor. By Lemma 57, let t be the upper bound for the sth column. Then, $(s, t + 1)$ is not a spreading factor. Therefore (s, t) is an extreme spreading factor. □

The final lemma guarantees that (s, s) is a spreading factor if s is the s-parameter.

Lemma 59. ***[AMSSI 2006]*** *If s is the s-parameter of a permutation, then (s, s) is a spreading factor.*

Proof. For some $t \geq s$, let (s, t) be an extreme spreading factor as defined in Definition 52. Then by the definition of the s-parameter, we have for all $i \neq j$, $|i-j| < s \Rightarrow |\pi(i)-\pi(j)| \geq t$. Thus for all $t' < t$, all pairs (s, t') are spreading factors, including $t' = s$. Thus (s, s) is a spreading factor.

□

In combining the three lemmas, it is possible to prove a very interesting result, namely that the definitions of s-parameter and spread are equivalent. Thus, the following theorem links the two definitions of spread.

Theorem 60. ***[AMSSI 2006]*** *The s-parameter of a permutation is equal to its spread.*

Proof. Let s be the s-parameter of a permutation. By Lemma 59, (s, s) is a spreading factor. For contradiction, suppose $(s + 1, s + 1)$ is also a spreading factor. By Lemma 58, $(s + 1, t)$ is an extreme spreading factor for some $t \geq s + 1$. Since $(s + 1, t)$ is an extreme spreading

factor, then the s-parameter is at least $s+1$. However, the s-parameter is s, which implies that $s+1 \leq s$. This is a contradiction. Therefore, $(s+1, s+1)$ is not a spreading factor.

Since (s, s) is a spreading factor, by the definition of spreading factor, $|i-j| < s \Rightarrow |\pi(i) - \pi(j)| \geq s$. So s satisfies (1). However, since $(s+1, s+1)$ is not a spreading factor, then for some $i \neq j$, $|i-j| < s+1 \nRightarrow |\pi(i) - \pi(j)| \geq s+1$. So $s+1$ does not satisfy (1). Therefore, s is the spread. Thus, the s-parameter of a permutation is equal to its spread.

□

9 Simulations

In order to test the effectiveness of our interleavers, we ran various simulations in a turbo simulator program. The turbo simulator was developed by Yufei Wu [Wu]. To run the simulation, specify a field, a monomial order, and a polynomial. In doing our simulations we worked with the following items:

Definition 61. Bits are the pieces of information that make up our messages. They are in the form 0 or 1.

Definition 62. Frames finite blocks of data of a specified length. In our simulations, this length is the cardinality of the field being used.

Definition 63. The **Signal to Noise Ratio (EbN0)** is the ratio of the strength of the signal to the strength of the surrounding noise in decibels (dB). In particular, we will be looking at signal to noise ratios of 0, 0.5, 1, 1.5, and 2 dB.

Definition 64. The **Bit Error Rate (BER)** is the number of uncorrectable corrupted bits over the total number of bits sent.

The turbo simulator takes as input five items: a permutation polynomial, a convolutional code, a monomial order, the number of decoding iterations, and a frame error limit.

The permutation polynomial and monomial order are those built using the methods from previous sections. In our simulator, these were both specified as arguments. The last three input items were coded into our turbo simulator during our simulations. The number of decoding iterations specified how many times the decoders passed an updated version of the message back and forth to take advantage of belief propagation. This value was set to 8 in our simulations, which means each decoder had 8 chances to try to correct any errors and check this "corrected" message with the other decoder. The frame error limit was the stopping condition in our simulation. When the number of uncorrectable errors reached this number, the simulation stopped. This number was set at 100 errors in our simulations.

The final BER is calculated after reaching the frame error limit at each signal to noise ratio. By looking at the BER for each signal to noise ratio, we can see how our various interleavers performed.

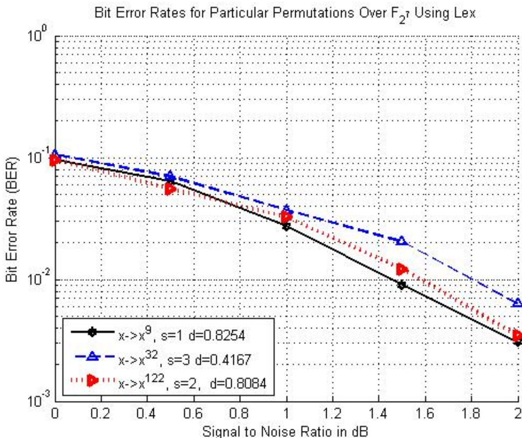
10 Results

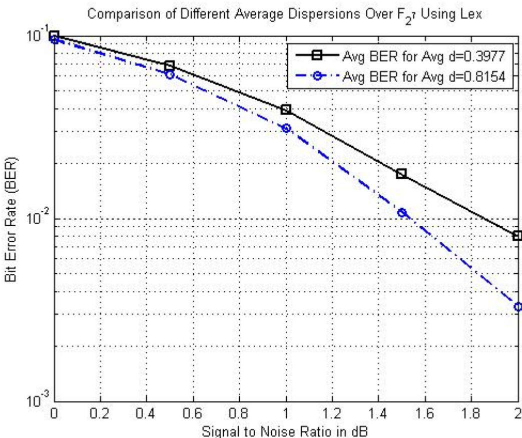
Our simulations aim to examine how the properties of permutations (spread and dispersion) affect the error-correcting ability of turbo codes and to examine how different monomial orderings may affect a turbo code. We look at these factors over fields of differing sizes.

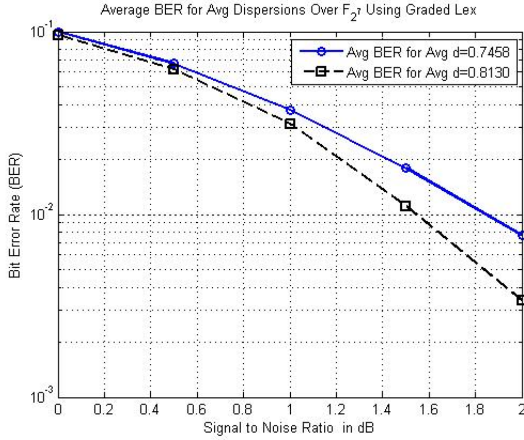
In general, we have found that in smaller fields, changing interleavers in order to obtain different spreads and dispersions has little effect on the ability of the codes to correct errors. It seems that the fields are too small to allow for much effect of the properties on the turbo codes. However, in larger fields like $\mathbb{F}_{2^8}$ or $\mathbb{F}_{5^3}$, the differences that occur with varying spreads, dispersions, and monomial orderings can be quite drastic. In practice, much larger fields than even these are used because of improved error correcting ability when using large block lengths.

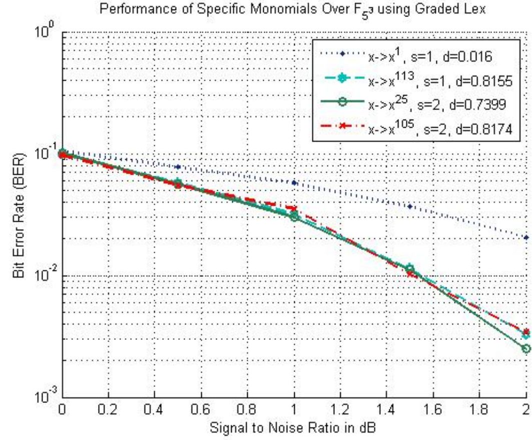
Monomial orderings provide a clear change in permutation action. Without an ordering applied, the spread of a permutation, using a monomial map, is always one, since 0 and 1 are always mapped to themselves under any monomial. Thus, monomial orders are a good place to look to see how dispersion affects a permutation, since by using no monomial ordering one can keep spread constant. Also, these orders provide insight into how changes in ordering can affect the performance of a turbo code, even those whose interleavers have similar spread and dispersion.

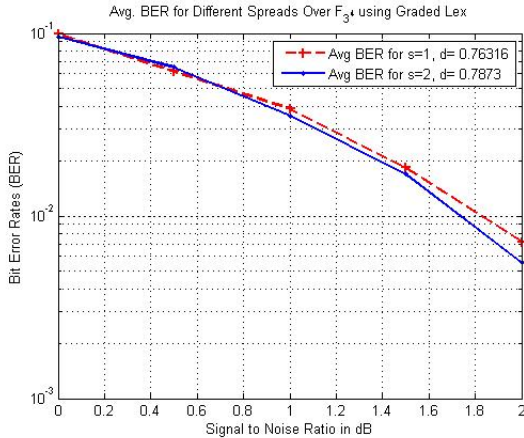
Spread and dispersion seem to be distinct factors in turbo code performance. In general, it seems that spread and dispersion have a weak inverse relationship. Thus, for very high spreads, the permutations tend to have relatively small dispersions. Examination of data has led to the hypothesis that higher dispersion means a better turbo code. Thus, it would seem that interleavers with smaller spreads would consequently create a better turbo code, however, we have also seen that spread can give an indication as to turbo code performance on its own. If dispersion is the same, a higher spread often results in slightly improved performance. Thus, though higher dispersion implies better error correcting in most cases, further examination is necessary to find the reasons for any discrepancies and create a sound theory on turbo code performance. Following are several graphs that illustrate the positive correlation between high dispersion and better performance:

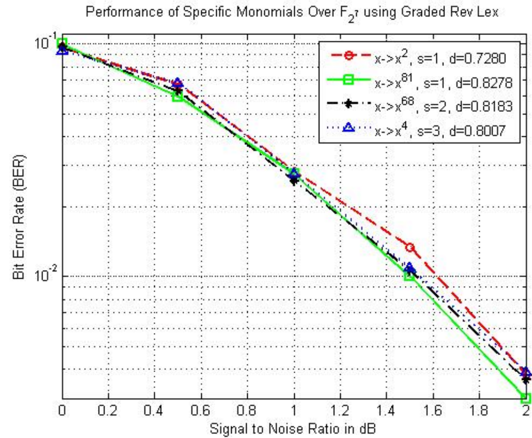












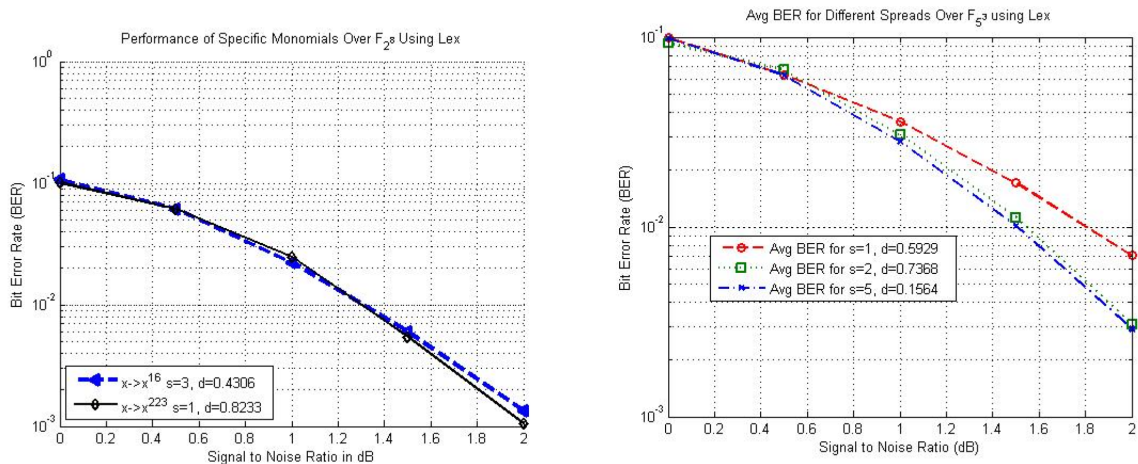
While these graphs exhibit curves with varying spread, dispersion, and monomial ordering, all demonstrate that permutations with higher dispersion perform better than those with lower dispersion. We can also see that the permutations with similar dispersions perform comparably. Dispersion appears to be the principle indicator of turbo code performance rather than spread, as permutations with higher spread but lower dispersion do not perform as well as those with lower spread but higher dispersion.

Differences in performance between permutations composed of different monomial orderings, varying spreads, and different dispersions are apparent. Thus, continuing to study the properties of permutations is a worthwhile pursuit. In order to gain a deeper understanding of our results, please see Appendix A for more graphs.

10.1 Possible Exceptions to Above Conjecture

While high dispersion corresponds with low bit error rates, there are some cases that appear to be exceptions to this general trend. Extreme spreads, like five in $\mathbb{F}_{5^3}$ and three and four in $\mathbb{F}_{2^8}$, seem to create an unpredictable variance of performance in a permutation regardless of monomial ordering or dispersion. While the permutations with these spreads have very low dispersion, their bit error rates are comparable to those with higher dispersion. The same unstable properties manifest themselves when dispersions are very low (which may be a result of the implied high spread). Occasionally, these permutations with both high spread and low

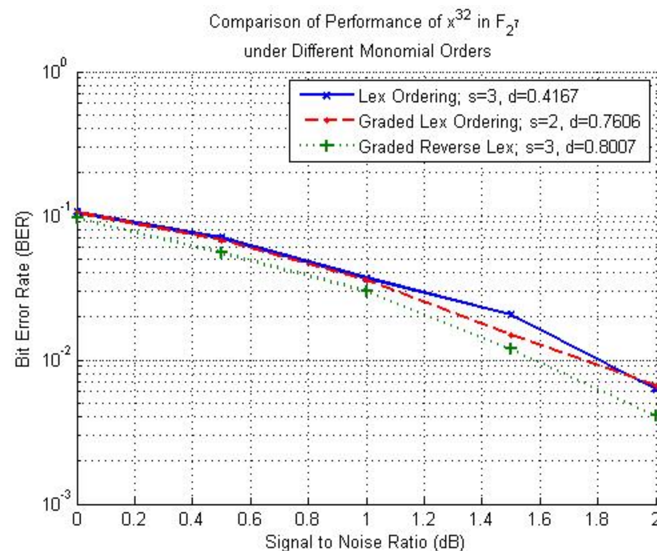
dispersion outperform those with higher dispersion. There are only a few examples of these exceptions, and out of the many block lengths we tested, only two ($\mathbb{F}_{2^8}$ and $\mathbb{F}_{5^3}$ specifically using lexicographic order) do not seem to follow the generally positive correlation between high dispersion and low bit error rates. The two examples are:



Here, the curves of permutations with low dispersions perform similarly or better than those with high dispersions. Since we found only two exceptions out of the large number of fields we tested, further examination of other properties or characteristics of these permutations is necessary to explain their unexpected behavior.

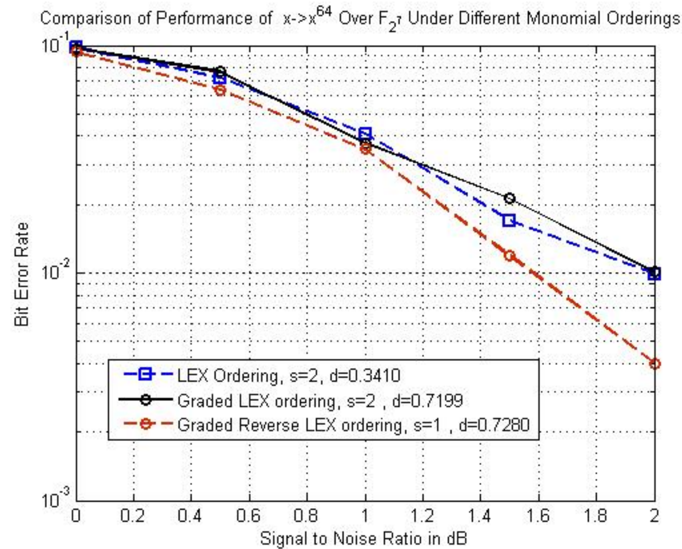
10.2 A Comparison of Di erent Monomial Orderings

Different monomial orderings can change the spread, dispersion, and permuted values of a permutation monomial. Thus, it is possible that a particular monomial will perform better when using one monomial ordering over another. For example:



Here, $x \mapsto x^{32}$ has the highest dispersion when Graded Reverse Lex is used, as well as the lowest bit error rate. In Lex Ordering, it has the lowest dispersion as well as the highest bit error rate. When comparing Lex and Graded Lex, Graded Lex, with the higher dispersion, has a lower bit error rate. Given the differences in their respective dispersions, however, one may expect a more noticeable difference in performance. This could possibly be attributed to the small sample sizes we used in the turbo simulator. Still, while the difference is slight, the higher dispersion yields a lower bit error rate.

Similar observations were made when comparing $x \mapsto x^{64}$ in $\mathbb{F}_{2^7}$:



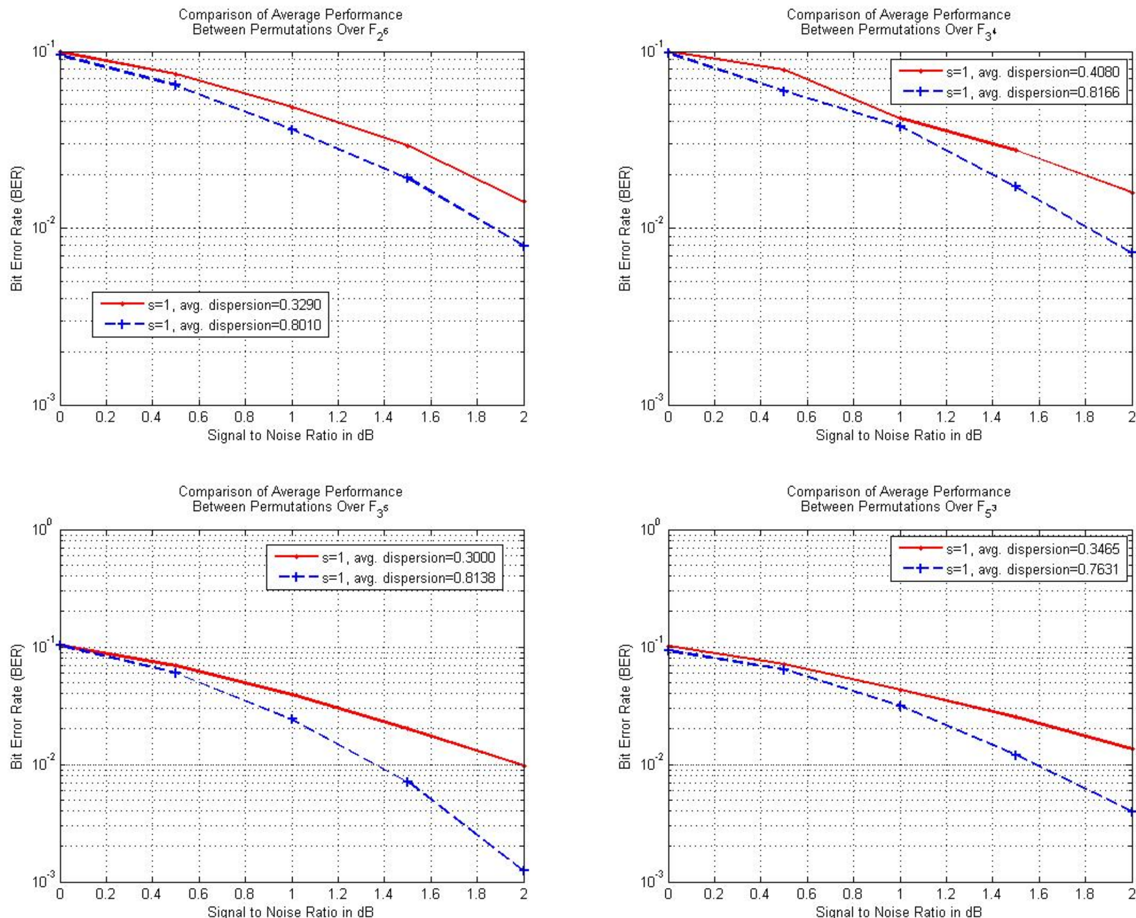
Here, the higher dispersion (also in Graded Reverse Lex) once again yields the lowest bit error rate. Note also that the spread is less than the other two, which once again seems to indicate that dispersion is the greater determinant of performance.

When compared to other permutations of the same block length under the same monomial orderings, dispersion seems to be a good indicator of performance. The fact that when the two monomial orderings (Lex and Graded Lex) do not perform exactly as expected based on dispersion alone could perhaps indicate that the particular monomial ordering used has some effect on performance as well. Along these lines, it should be noted that when using Graded Reverse Lex in all fields we examined, it is very rare that a permutation has a dispersion less than .7000, with the exception of the mapping $x \mapsto x$, which by default has a very low dispersion. This is important, as .7000 seems to be a sort of breaking point between good and poor performance. Permutations with dispersion higher than .7000 perform comparably, while those with dispersion less than .7000 perform with distinctly poorer bit error rates.

10.3 Permutations That Do Not Use Monomial Orders

After examining how permutations performed under different monomial orderings, we looked at some without any ordering at all. Because we examined only monomials for creating our permutations, the spreads for all the permutations created without an imposed ordering is one, simply because 1 and 0 are always mapped next to themselves. With spread being constant regardless of the permutation, better comparisons of the effects of dispersion can

be made. Here are some comparisons of different average dispersions of permutations in several fields. The dispersion values were divided and averaged together in separate intervals as follows : $[0.0000, 0.4999], [0.5000, 0.6999]$, and $[0.7000, 1.0000]$. The ranges were based on how frequently dispersions in certain ranges appear, as well as distinct differences in performance of different dispersion values/ranges. With the fields we tested, dispersions never fell within the second interval. Thus there are only two curves on each graph.



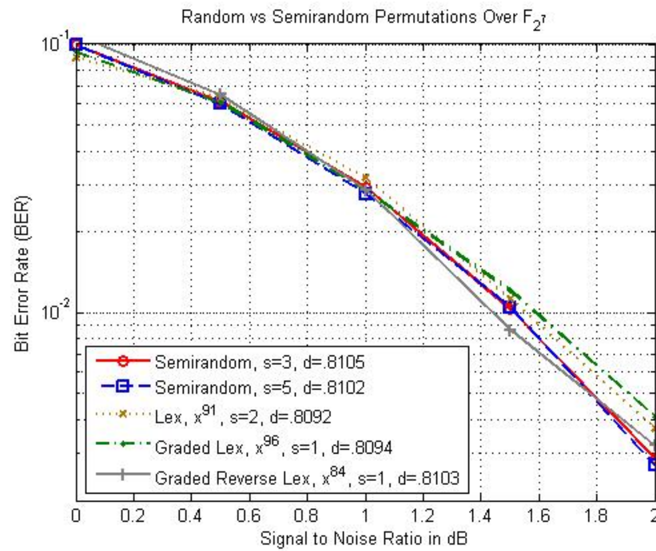
As expected, when different average dispersions are compared, we see that the higher dispersion yields lower bit error rates. Since the value for spread in all permutations is $s = 1$ when no ordering is used, dispersion appears to be the primary indicator for turbo code performance; in all examples we tested, high dispersion corresponded to low bit error rates.

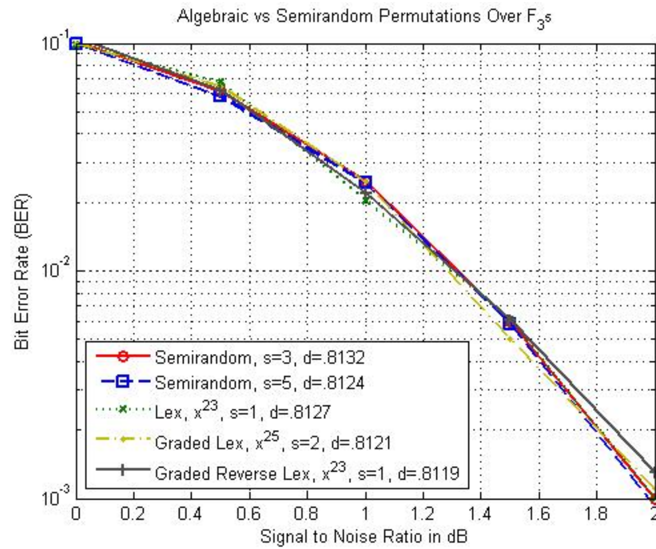
10.4 Semi-Random Permutations versus Algebraic Permutations

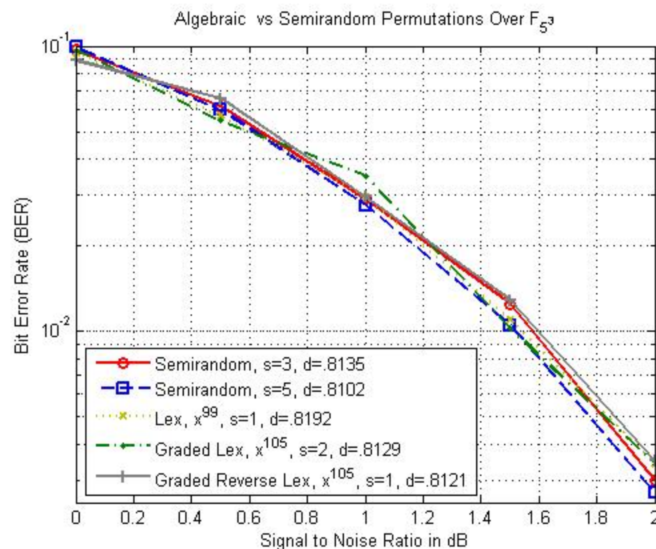
Definition 65. A **semi-random permutation** of $\mathbb{Z}_q$ can be generated by choosing images for $0, 1, ..., q-1$, one after the other, so that each stage of the permutation has a pre-specified spread. For the full algorithm, see Appendix B.10.1

We compared the performance of semi-random permutations to the performance of the algebraic permutations we created. We did this in part to see if the method used in constructing the permutation had an effect on their performance. When compared with permutations

of similar dispersion, both semi-random and algebraic permutations perform just as well. There are some things to note. When semi-random permutations are generated, regardless of spread, they typically have a dispersion around 0.81, unlike most of the algebraic permutations in which dispersion values tend to decrease as the spread increases.



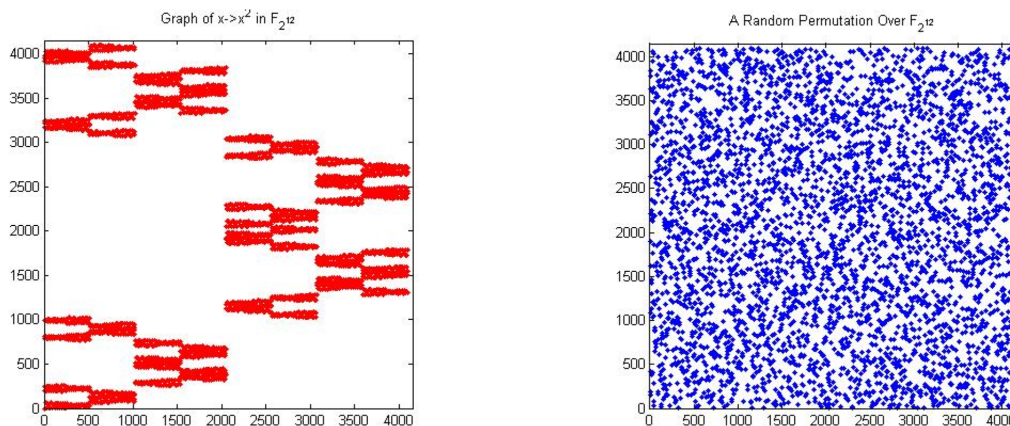




Thus, because each permutation has a similar dispersion, they all perform in much the same way. This also demonstrates that the dispersion is a better indicator of performance than spread.

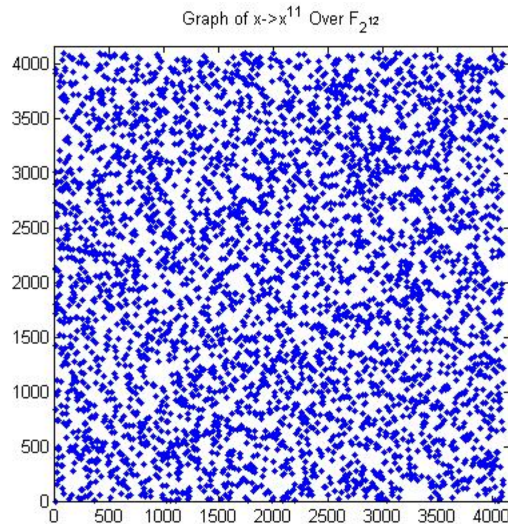
10.5 Pseudo-Random Permutations versus Algebraic Permutations

Plots of both a pseudo-random permutation and an algebraic one of 2^{12} letters.



The first plot, of the algebraic permutation created using $x \mapsto x^2$, has a clear pattern, while the second, of a pseudo-random permutation, lacks any consistent pattern. Because dispersion is a measure of the randomness of a permutation, we conjecture that the first has a low dispersion, while the second has a higher dispersion. In fact, the dispersions of the algebraic permutation is 0.2372 and the dispersion of the pseudo-random permutation is 0.8137. Since most of our findings indicate a correspondence between high dispersion and better turbo code performance, the second permutation should perform better. Therefore, the goal would be to use an algebraic interleaver that yields a permutation with a higher dispersion, which graphically translates to a plot that lacks a clearly defined pattern. Looking

at the plot of the permutation created using $x \mapsto x^{11}$ with dispersion 0.8139 below, one can notice a much more random-looking pattern:



Visually, one can tell that in this particular field, the permutation created by $x \mapsto x^{11}$ has a higher dispersion than that created by $x \mapsto x^2$. The actual dispersion of the permutation created by $x \mapsto x^{11}$ is .8139. One could conclude then, without necessarily running these permutations through the simulator, that using $x \mapsto x^{11}$ should produce overall lower bit error rates than using $x \mapsto x^2$.

11 Suggestions For Future Research

Some suggestions for future research include:

(i) Permutations with longer block lengths) should be studied. This would be useful for real world applications of turbo codes.

(ii) Are monomials with different integer coefficients more or less effective than monomials with coefficients of one? Do they provide better/a greater variety of spread and dispersion values, as opposed to those with coefficients of one?

(iii) How do polynomial permutations within a field perform? Will they give a greater variety of spreads and dispersions to experiment with?

(iv) More examination of the effects of particular monomial orderings on performance should be done. Is there one particular monomial ordering that is ideal for all permutations?

(v) What is the average dispersion and standard deviation for all permutations of a given block length, and what implications may this have for turbo codes? Does this provide insight into why turbo codes perform as well as they do?

12 Conclusion

In our research, we have found that, for the most part, a higher dispersion for a permutation seems to indicate a low bit error rate. In most cases, when two permutations with considerably different dispersions are simulated and their respective bit error rates compared, the one with higher dispersion tends to have a lower bit error rate. Likewise, permutations with similar dispersions perform similarly with similar bit error rates. We have also found that by using monomial orderings, we can sometimes improve upon a given permutation, as properties such as dispersion can be altered (improved) when a different monomial ordering is applied. For algebraic monomial permutations, we found that often when the spread increases, the dispersion decreases. In most fields we tested, this decrease in dispersion resulted in a higher bit error rate for permutations with the higher spreads. For semi-random permutations, however, the dispersion tends to be around .81, regardless of spread.

13 Acknowledgements

This research was conducted at the Applied Mathematical Sciences Summer Institute (AMSSI) and has been partially supported by grants given by the Department of Defense (through its ASSURE program), the National Science Foundation (DMS-0453602), and the National Security Agency (MSPF-06IC-022). Substantial financial and moral support was also provided by Don Straney, Dean of the College of Science at California State Polytechnic University, Pomona. Additional financial and moral support was provided by the Department of Mathematics at Loyola Marymount University and the Department of Mathematics & Statistics at California State Polytechnic University, Pomona. This project would not have been possible without the help of Dr. Edward C. Mosteig and Laura Smith; a special thanks to them and all the AMSSI faculty. The authors are solely responsible for the views and opinions expressed in this research; it does not necessarily reflect the ideas and/or opinions of the funding agencies and/or Loyola Marymount University or California State Polytechnic University, Pomona.

References

[Be] Berrou, Claude, Near Optimum Error Correcting Coding and Decoding: Turbo-Codes, *IEEE Transactions on Communications.* **44** (1996), 1261-1271.

[CoRu] Corrada-Bravo, Carlos J., Rubio, Ivelisse M., Algebraic Construction of Interleavers Using Permutation Monomials, *IEEE Communications Society.* **2** (2004), 911-915.

[CoRu(2)] Corrada-Bravo, Carlos J., Rubio, Ivelisse M., Cyclic Decomposition of Permutations of Finite Fields Obtained Using Monomials, *Springer Berlin/Heidelberg.* **2948** 2004, 254-261.

[CoRu(3)] Corrada-Bravo, Carlos J., Rubio, Ivelisse M., Deterministic Interleavers for Turbo Codes with Random-like Performance and Simple Implementation, *Proc, 3rd Int. Symp. Turbo Codes*, Brest, France, Sep. 2003.

[Cr] Cruz, Louis J., Permutations that Decompose in Cycles of Length 2 and are Fiven by Monomials, *Proceedings of The National Conference On Undergraduate Research (NCUR).* (2006), 1-8.

[DoDi] Dolinar, S., Divsalar, D., Weight Distributions for Turbo Codes Using Random and Nonrandom Permutations, *TDA Progress Report, 42-122, JPL* Aug. 1995, 56-65.

[Fo] Forney, Jr., G. David, Convolutional Codes I: Algebraic Structure, *IEEE Transactions on Information Theory.* **16** (1970), no. 6, 720-738.

[Ga] Garrett, Paul, The Mathematics of Coding Theory, *Pearson Prentice Hall.* (2004).

[HeWi] Heegard, C., Wicker, Stephen, B., Turbo Coding, *Kluwer Academic Publishing.* (1999).

[JoMo] Jones, Alaina, Moreno, Benjamin, Smith, Laura, Viteri, Andrea, Yao, Kouadio David, Mosteig, Edward, Exploring Interleavers in Turbo Coding, *AMSSI 2005 Technical Report.* (2005).

[LiMu(1)] Lidl, Rudolf, Mullen, Gary J., When Does a Polynomial over a Finite Field Permute the Elements of the Field?, *American Mathematical Monthly.* **95** (1988), no. 4, 243-246.

[LiMu(2)] Lidl, Rudolf, Mullen, Gary J., When Does a Polynomial over a Finite Field Permute the Elements of the Field?, II, *American Mathematical Monthly.* **100** (1993), no. 1, 71-74.

[LiMo] Little, John B., Mosteig, Edward, Error Control Codes from Algebra and Geometry, *Notes for SACNAS Minicourse.* Sept., 25 2004.

[LuPe] Luis, Y., B., Perez, L., O., Permutations of Z_q Constructed Using Several Monomial Orderings, *Proceedings of The National Conference On Undergraduate Research (NCUR).* (2005) 1-8.

[Pr] Pretzel, Oliver, Error-Correcting Codes and Finite Fields, *Clarendon Press*. (1992).

[Ro] Roman, Steven, Introduction to Coding and Information Theory, *Springer-Verlag*, New York. (2005).

[Ta] Takeshita, Oscar Y., Permutation Polynomial Interleavers: An Algebraic-Geometric Perspective, avaliable at: http://arxiv.org/abs/cs/0601948 (2006).

[va] van Lint, J.H., Introduction to Coding Theory, *Springer-Verlag*, New York. (1999).

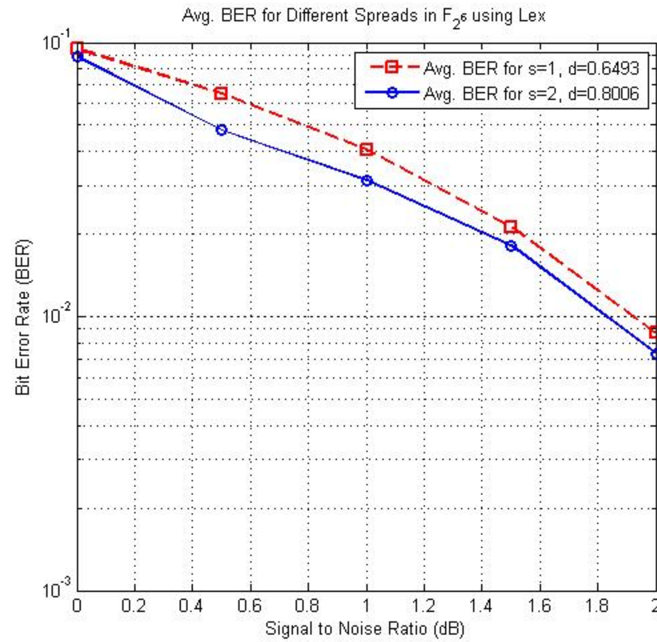
[Wu] Wu, Yufei., Turbo Code Simulator., Nov 1998., MPRG lab, Virginia Tech. `http://www.ee.vt.edu/∼yufei`

A Appendix A

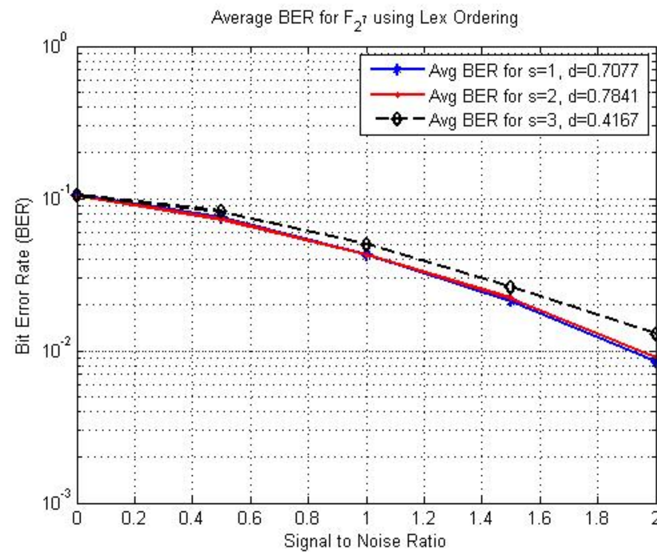
This appendix is composed of much of the data we obtained but did not include in our results section. This is to serve as further evidence for our conclusions and conjectures.

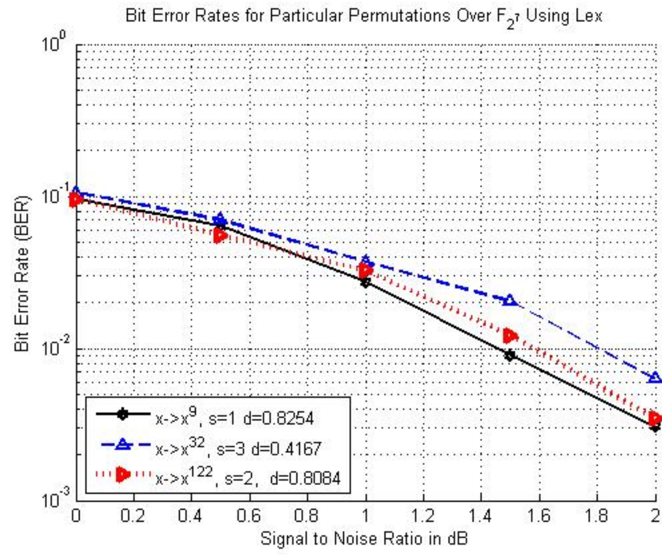
A.1 Lexicographic Ordering

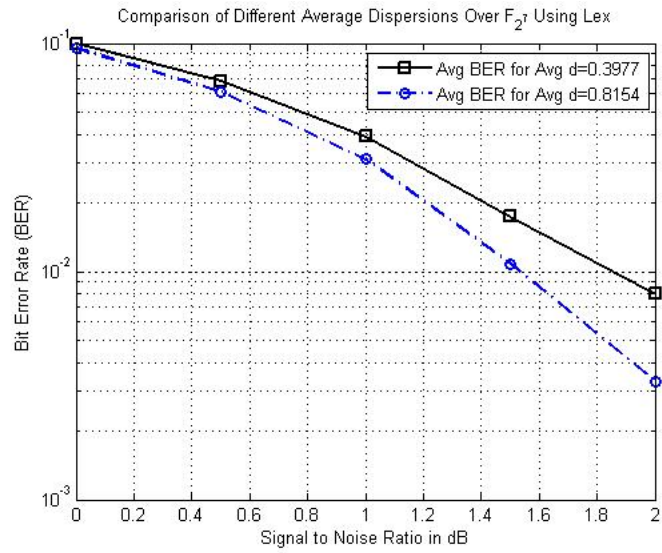
A.1.1 Bit Error Rates in $\mathbb{F}_{2^6}$ Using Lex Ordering



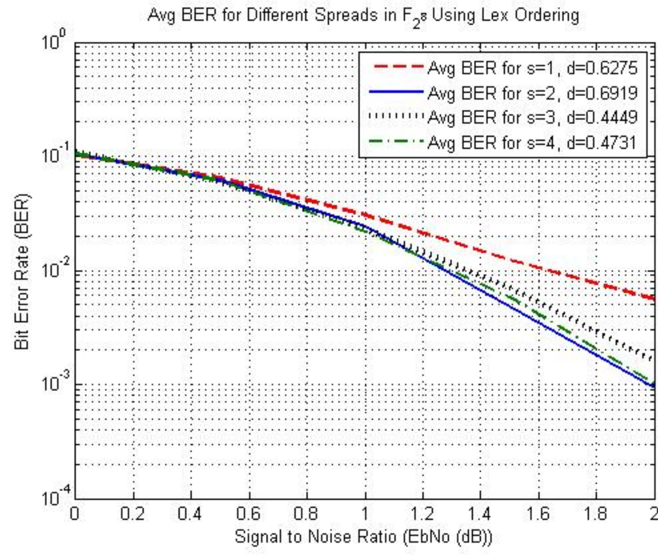
A.1.2 Bit Error Rates in $\mathbb{F}_{2^7}$ Using Lex Ordering

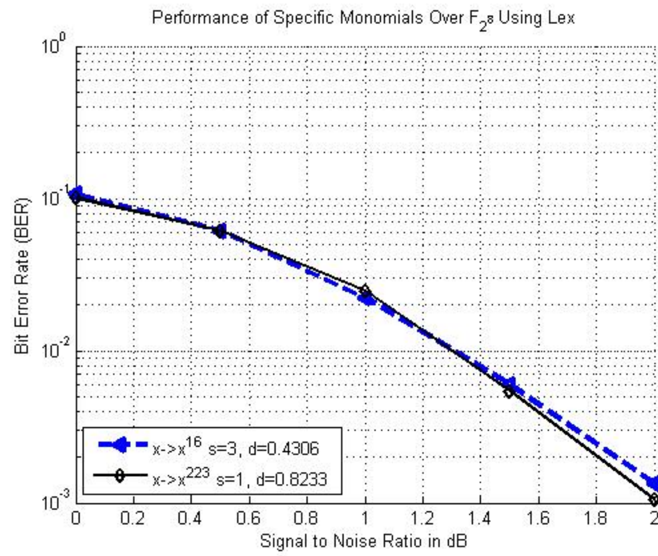



Bit Error Rates for Particular Permutations Over F_{2^7} Using Lex
Bit Error Rate (BER)
Signal to Noise Ratio in dB
x->x^9, s=1 d=0.8254
x->x^32, s=3 d=0.4167
x->x^122, s=2, d=0.8084

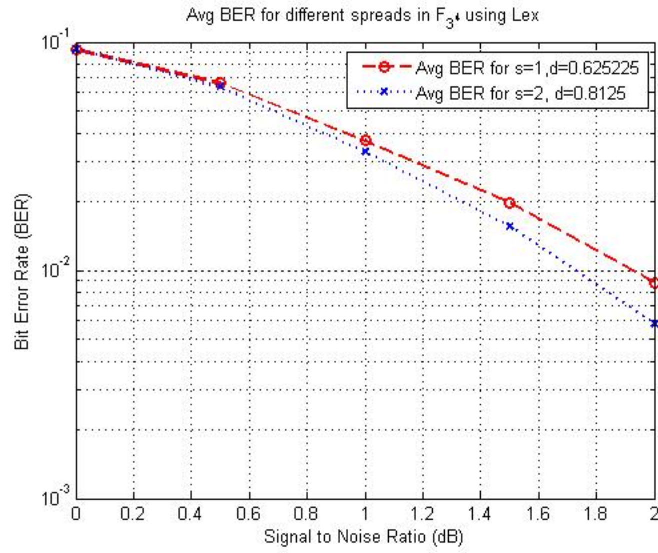

Comparison of Different Average Dispersions Over F_{2^7} Using Lex
Bit Error Rate (BER)
Signal to Noise Ratio in dB
Avg BER for Avg d=0.3977
Avg BER for Avg d=0.8154

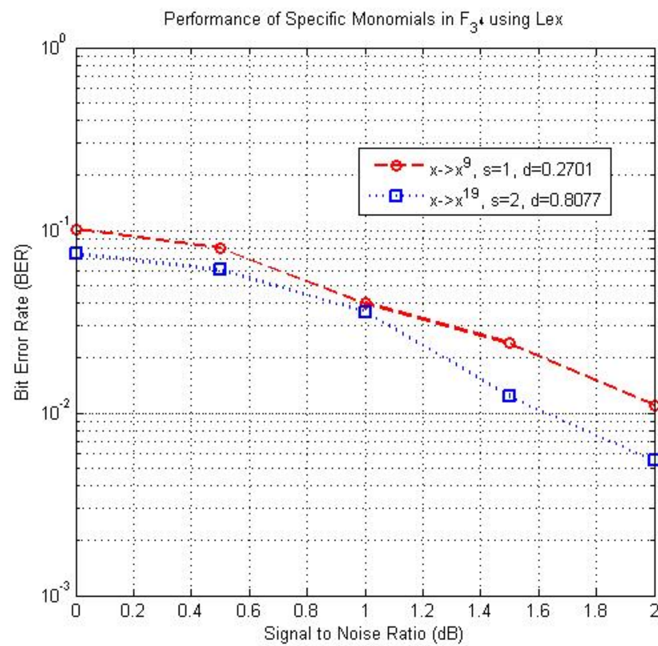
A.1.3 Bit Error Rates in $\mathbb{F}_{2^8}$ Using Lex Ordering



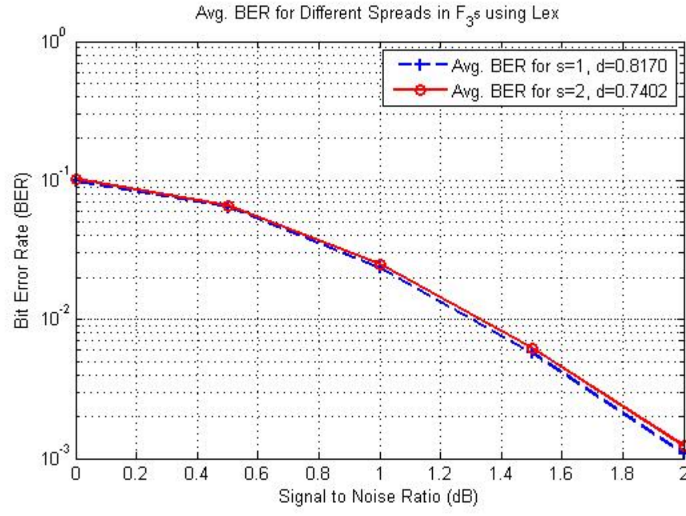


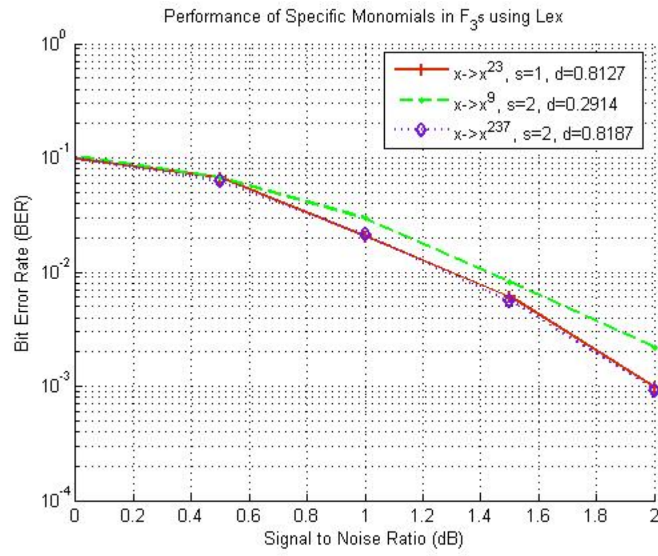
A.1.4 Bit Error Rates in $\mathbb{F}_{3^4}$ Using Lex Ordering



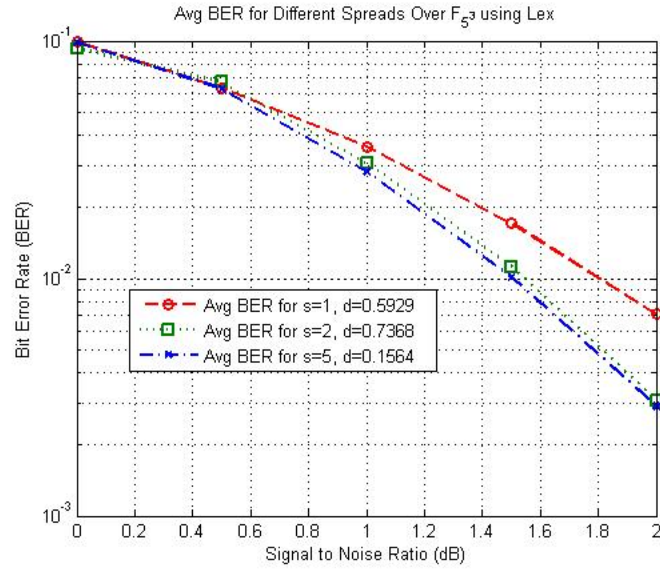


A.1.5 Bit Error Rates in $\mathbb{F}_{3^5}$ Using Lex Ordering



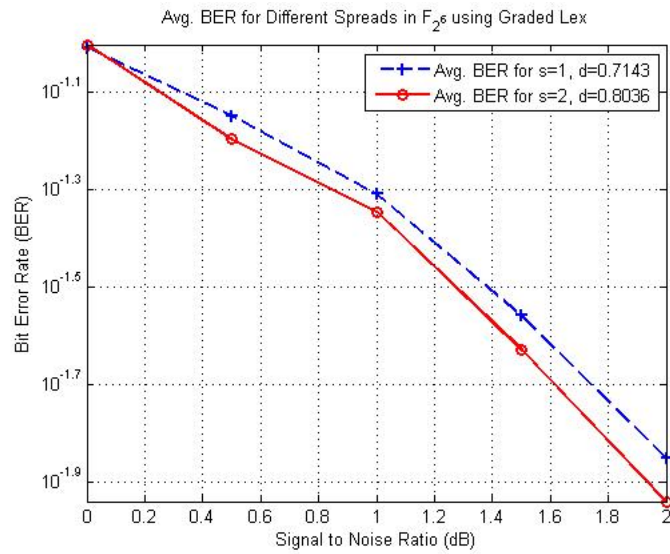


A.1.6 Bit Error Rates in $\mathbb{F}_{5^3}$ Using Lex Ordering

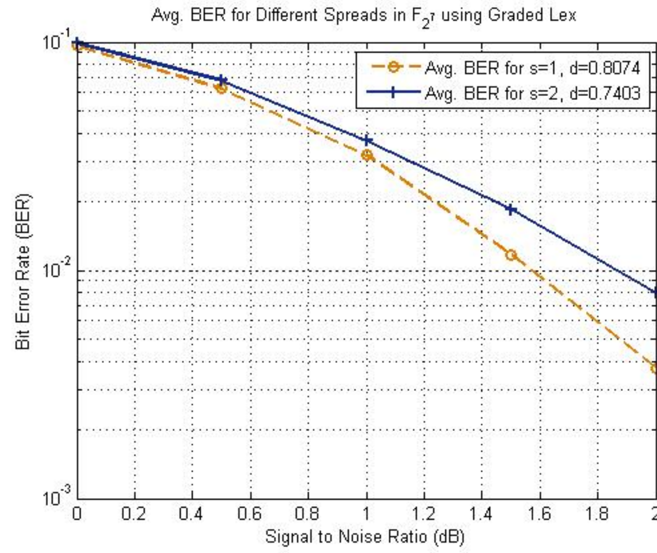


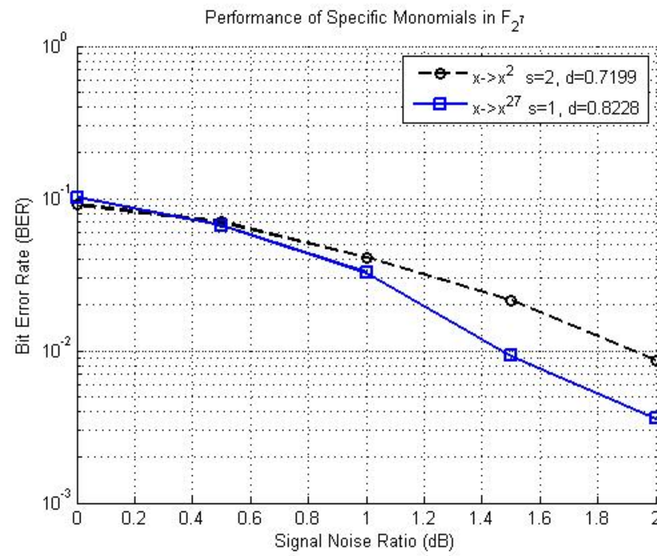
A.2 Graded Lexicographic Ordering

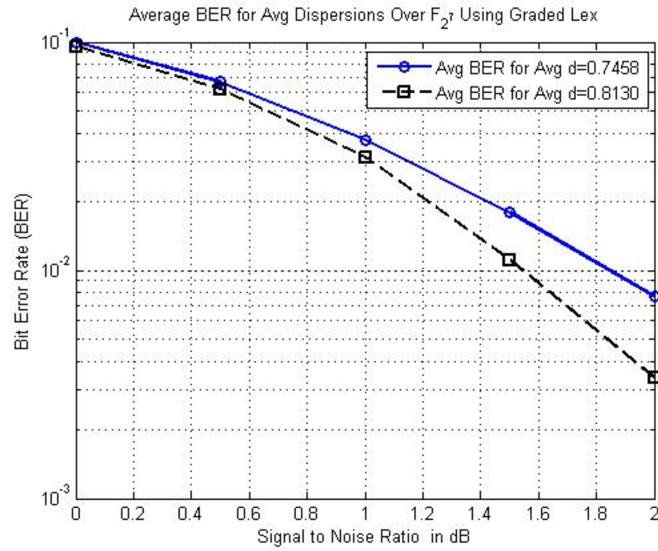
A.2.1 Bit Error Rates in $\mathbb{F}_{2^6}$ Using Graded Lex Ordering



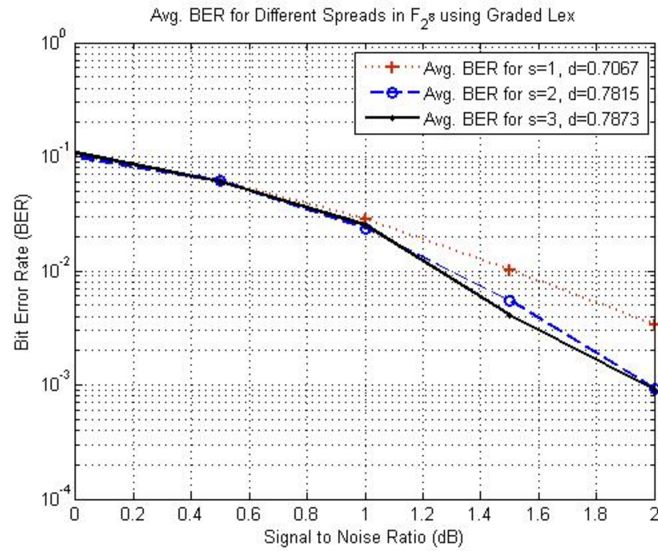
A.2.2 Bit Error Rates in $\mathbb{F}_{2^7}$ Using Graded Lex Ordering

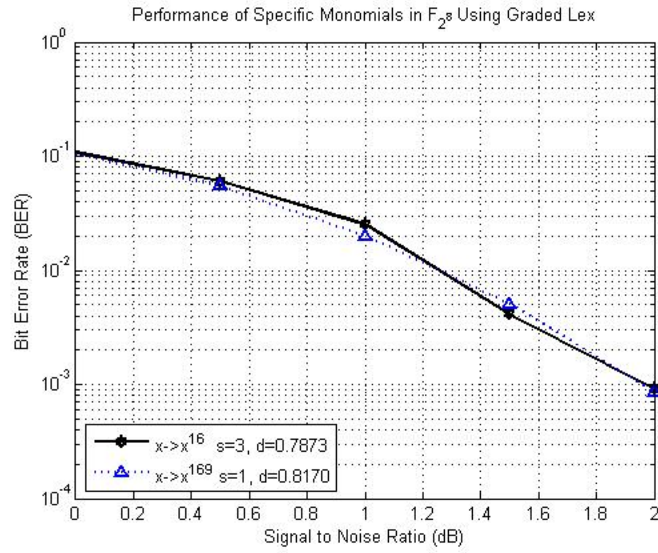




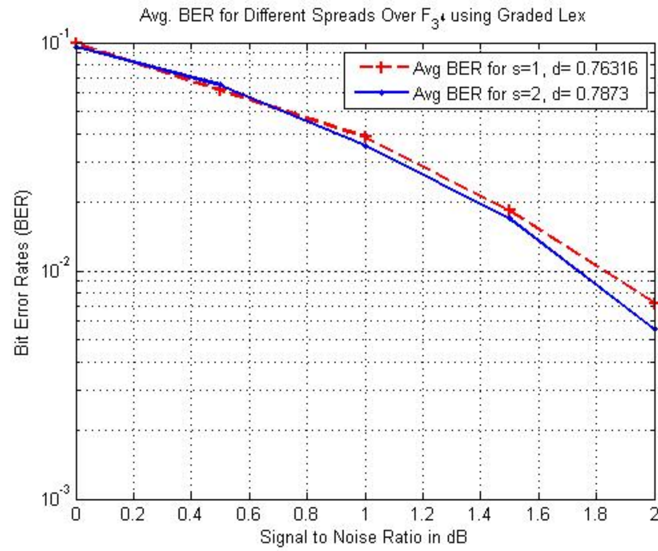


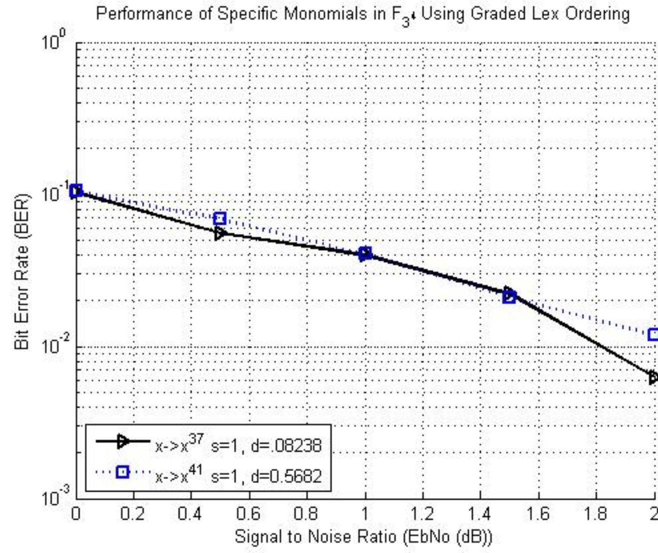
A.2.3 Bit Error Rates in $\mathbb{F}_{2^8}$ Using Graded Lex Ordering



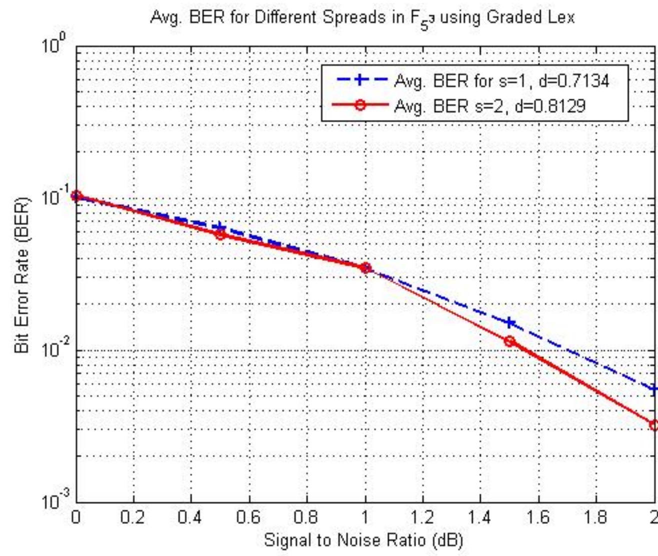


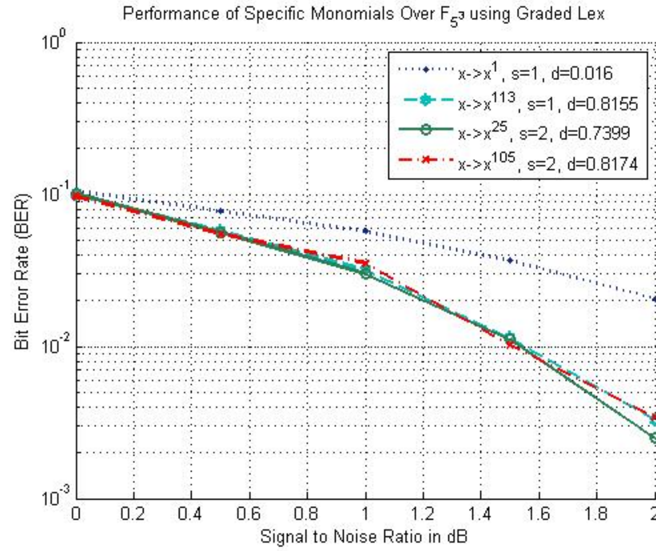
A.2.4 Bit Error Rates in $\mathbb{F}_{3^4}$ Using Graded Lex Ordering





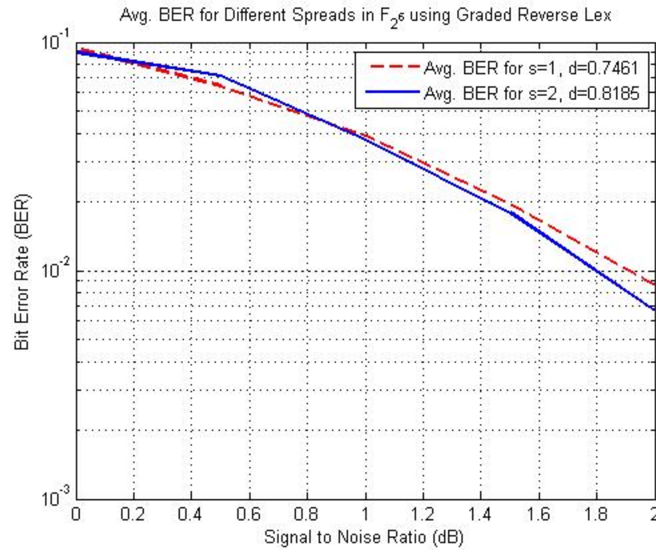
A.2.5 Bit Error Rates in $\mathbb{F}_{5^3}$ Using Graded Lex Ordering

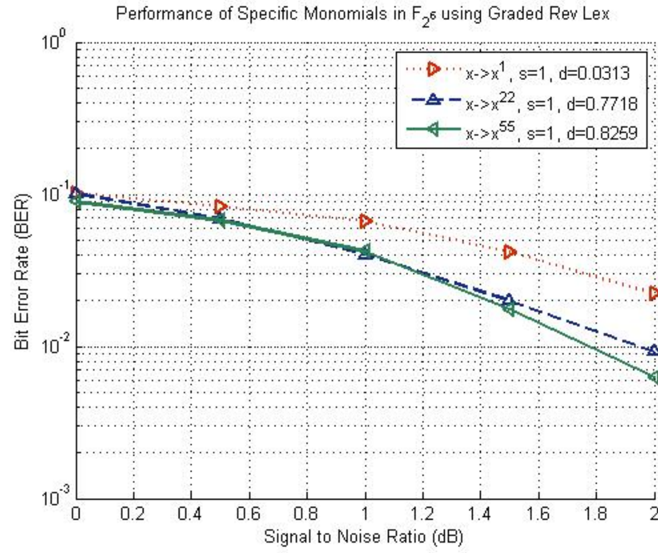




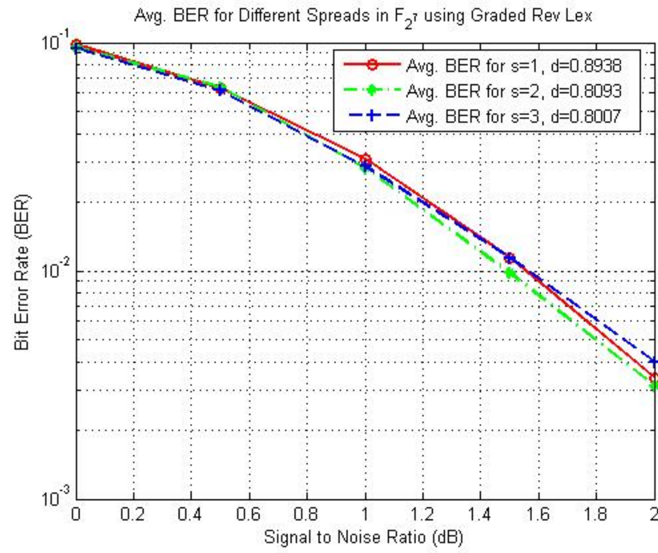
A.3 Graded Reverse Lexicographic Ordering

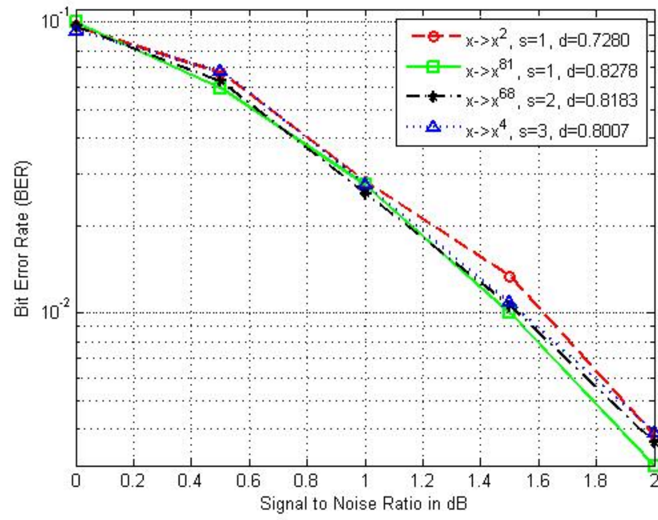
A.3.1 Bit Error Rates in $\mathbb{F}_{2^6}$ Using Graded Reverse Lex Ordering

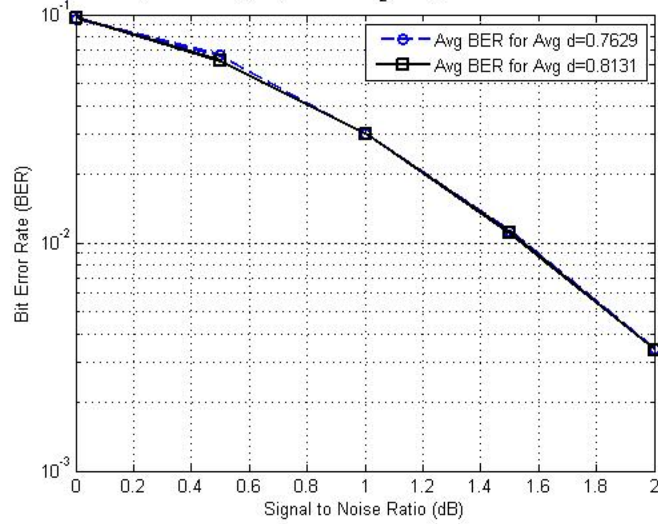




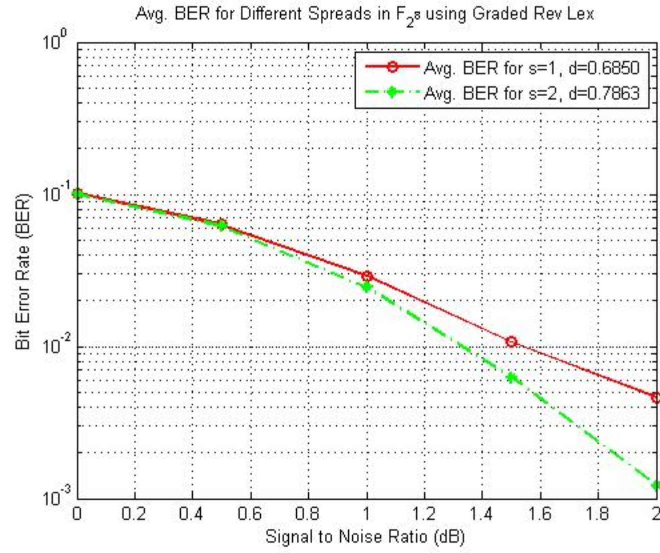
A.3.2 Bit Error Rates in $\mathbb{F}_{2^7}$ Using Graded Reverse Lex Ordering

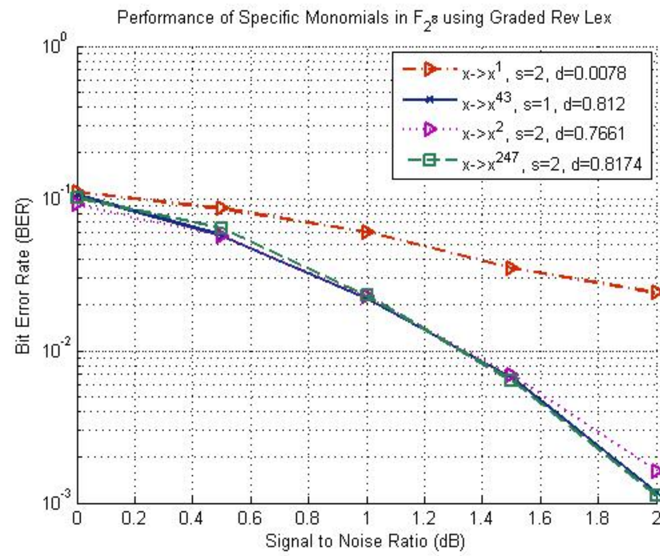



Performance of Specific Monomials Over F_{2^7} using Graded Rev Lex
x->x^2, s=1, d=0.7280
x->x^81, s=1, d=0.8278
x->x^68, s=2, d=0.8183
x->x^4, s=3, d=0.8007
Bit Error Rate (BER)
Signal to Noise Ratio in dB

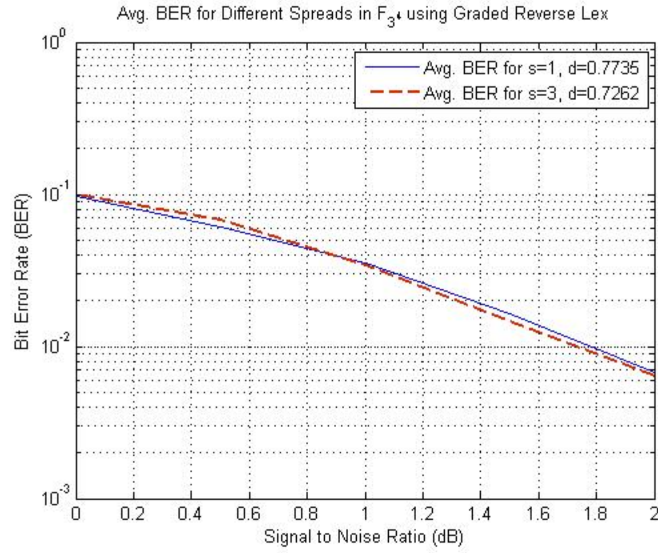

Avg BER for Avg Dispersions in F_{2^7} Using Graded Reverse Lex
Avg BER for Avg d=0.7629
Avg BER for Avg d=0.8131
Bit Error Rate (BER)
Signal to Noise Ratio (dB)

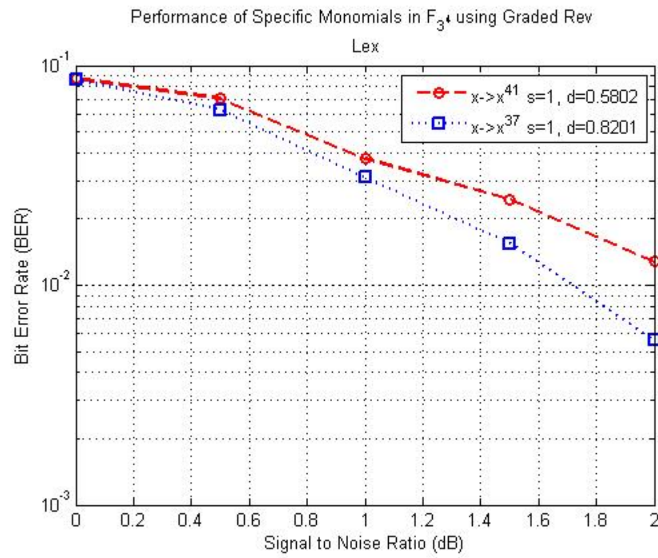
A.3.3 Bit Error Rates in $\mathbb{F}_{2^8}$ Using Graded Reverse Lex Ordering



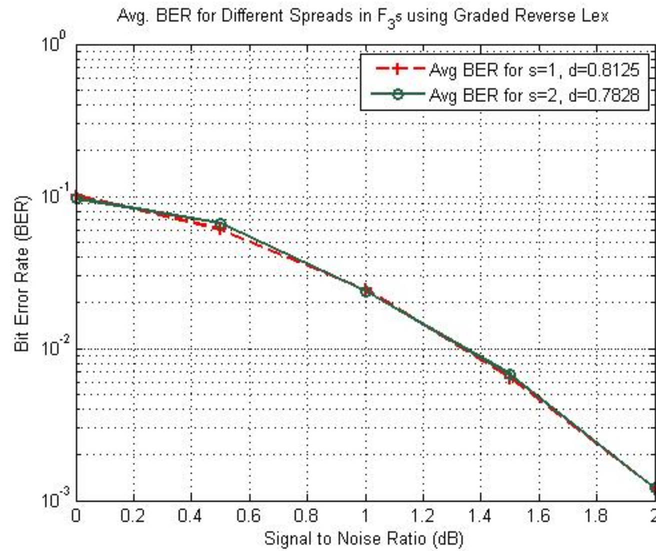


A.3.4 Bit Error Rates in $\mathbb{F}_{3^4}$ Using Graded Reverse Lex Ordering

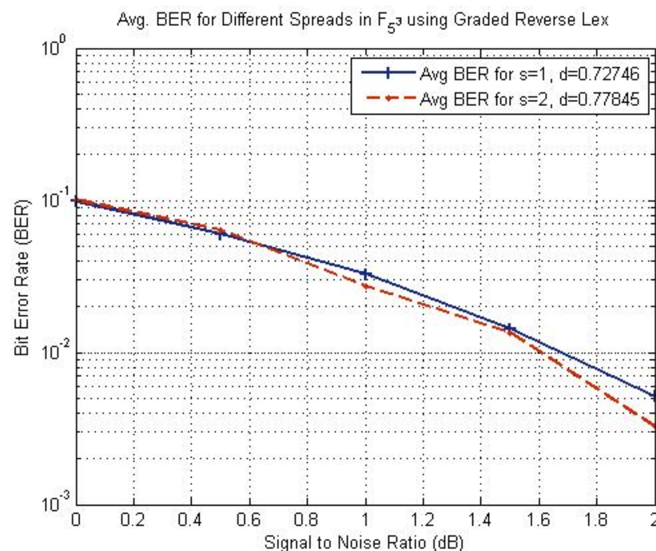




A.3.5 Bit Error Rates in $\mathbb{F}_{3^5}$ Using Graded Reverse Lex Ordering



A.3.6 Bit Error Rates in $\mathbb{F}_{5^3}$ Using Graded Reverse Lex Ordering



B Appendix B

B.1 A Note on Matlab Programs for this Project

These Matlab programs were all created during AMSSI 2006. Some of the programs depend on the Communications Toolbox developed for Matlab. Thus, these will not work on a normal version of Matlab. The programs that will not work without the special toolbox will be noted in their beginning comments.

In all cases, the input of a permutation is given to a program as a vector of the images of the permutation. For example, the permutation

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & 0 & 3 & 1 \end{pmatrix}$$

would be passed to the program as the vector [4 2 5 0 3 1] (or, equivalently, [4,2,5,0,3,1]).

Additionally, polynomials such as $1+\ ^2+\ ^5$ or $4+\ +7\ ^3$ would be passed as [1 0 1 0 0 1] and [4 1 0 7], respectively, with the numbers in the vector representing the coefficients on the powers of the polynomial variable, beginning with the variable to the 0th power as the furthest left entry, and moving up by one power for each entry to the right. Though the programs here are written specifically for monomials like x^i, they can be fairly easily modified to accept polynomials with integer coefficients as the permutation polynomial. Some hints for this are given in the programs likely to be modified.

In addition to the titles of each program, it is important to read the first few lines of the program to examine its inputs and outputs. This will give a greater understanding of the nature of the program.

B.2 *Programs Examining Properties of Permutations*

B.2.1 Program to Determine Spread

This program takes a permutation as input and outputs the value of the spread.

```
function out=spread(vect)
%out=spread(vect)
% Calculates the spread of a permutation 'vect'
% Outputs the numeric value of the spread

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

out=1;                                  % sets up default spread value
q=length(vect);                         % initialize variables
s=2;
flag=0;
while (s<=(sqrt(q/2)+2) && (flag~=1)) % sets conditions for ending loop
    j=2;
    % starts looping j upwards until it reaches the
    %   end of the vector or we find where s fails
    while ((j<=q) && (flag~=1))
        i=(j-(s-1));           % sets up i based on the value of s
        if (i<=0)              % moves i back to 1 if above line sends it 0 or less
            i=1;
```

```
        end
        while ((flag~=1) && (i<j))   % starts looping i upwards until i<j s fails
            if abs(vect(j)-vect(i)) < s   % failure condition for s
                flag=1;
                out=s-1;
            end
        i=i+1;
        end
    j=j+1;
    end
    if flag~=1   % if flag was not tipped off, then all values of i,j for the
        out=s;   %   specific s worked, so s is a new candidate for spread
    end
    s=s+1;
end
```

B.2.2 Program to Determine Spread Factors

```
function mini=permdist(perm)
%mini=permdist(perm)
% Calculates the spread factors of the permutation 'perm'
% Outputs the decimal value of the spread factor of 'perm'

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

n=length(perm);
mini=2*n;          % set 'mini' very large so it's replaced in first loop
for i=1:n-1;
    for j=i+1:n;   % adds the 'distance' between i and j, and
                   %%   the 'distance between perm(i) and perm(j)
        dist=min(mod(i-j,n),mod(j-i,n))+
                min(mod(perm(i)-perm(j),n),mod(perm(j)-perm(i),n));
        mini=min(dist,mini);  % picks smallest between old min and new value
    end
end
```

B.2.3 Program to Plot Spreading Factors

This program uses the above program to determine spreading factors, and then plots those spreading factors. This also plots a yellow dotted $y = x$ line for ease in finding the s-parameter. The program can also plot a line showing the calculated spread value for comparison with the s-parameter, if the correct argument is passed.

```
function [count,s]=plotsprfactor(permutation,plotspread)
%[count,s]=plotsprfactor(permutation,plotspread)
```

```
% Plots the spreading factors of a permutation and, if desired,
%    plots the calculated spread as a vertical line
% If plotspread=1, plots the spread line; does not plot spread line
%    if plotspread=0
% Outputs the number of spreading factors (and only those where s,t>1)
%    and the value of the spread

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

newplot
[input,count]=sprfactor(permutation);
a=input(1,:);
b=input(2,:);
plot(a,b,'xr')
q=length(permutation);
title('Spreading Factor Plot,with Line Representing Spread Value');
xlabel('Values of S');
ylabel('Values of T');
axis([0,q,0,q]);
hold on
d=0:q;
s=spread(permutation);
if plotspread==1
    c=s*ones(1,q+1);
    plot(c,d)
end
plot(d,d,'y:') % plots y=x line
hold off
```

B.2.4 Program to Determine Spreading Factors

```
function [output,count]=sprfactor(perm)
%[output,count]=sprfactor(perm)
% Calculates the spreading factors of the permutation 'perm'
% Outputs a matrix of s values matched up with t values that work:
% [s1 s2 ... ; t1 t2 ...]
%    and the number of spreading factors (where s,t>1)

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

q=length(perm);  % initializing variables
output=[];
count=0;
```

```
for s=1:q;       % sets up values of s and t
    for t=1:q;
        flag=0;  % more initializing variables
        add=1;   % trigger variable to add the (s,t) pair to the output variable
        j=2;
        while (j<=q) && (flag~=1)  % j and i loops just like spread
            i=(j-(s-1));
            if (i<=0)
                i=1;
            end
            while ((flag~=1) && (i<j))
                if abs(perm(j)-perm(i)) < t  % failure condition for t
                    add=0;
                    flag=1;
                end
            i=i+1;
            end
        j=j+1;
        end
        if add==1          % adds a successful (s,t) pair to the output vector
            output=[output [s;t]];
            count=count+1; % increases count since another pair is added
        end
    end
end plot(output(1,:),output(2,:),'+')
count=count-2*q+1;  % removes the pairs where s,t=1; since s,t range from 1 to q
                    % there are q-1 places where s=1, q-1 places where t=1,
                    % and 1 place where s=t=1; thus, s,t=1 in 2(q-1)+1
                    % places, or 2q-1 places
```

B.2.5 Program to Determine Dispersion

```
function out=dispersion(vect)
%out=dispersion(vect)
% Calculates the dispersion of a parameter 'vect'
% Outputs the decimal value of the dispersion of 'vect'

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

q=length(vect);        % setup variables
M=[];                  % setup variables
k=1;                   % flag for the first run
for j=2:q              % j goes from the second slot to the end of the vector
    for i=1:j-1        % i<j hardcoded
```

```
        if (k==1)      % M has no vectors, no reason to check the "other" pairs
            M=[j-i;vect(j)-vect(i)];
            k=2;       % M now has values, must go to next section from now on
        else
            addin=1;             % sets up flag to add pair into matrix
            for l=1:length(M)-1  % checks matrix for the current ordered pair
                if ((j-i)==M(1,l)) && ((vect(j)-vect(i))==M(2,l))
                    addin=0;     % if the pair is found, must not add the pair in
                end
            end
            if (addin==1)        % pair was not found by code above, so it's added
                V=[j-i; vect(j)-vect(i)];
                M=[M V];
            end
        end
    end
end
[m,n]=size(M);        % gets size of the matrix; want # columns
out=(2*n)/(q*(q-1));  % calculates the dispersion value
```

B.2.6 Optimized Program to Determine Dispersion

This is a much-improved version of the dispersion program that will run quite quickly compared with the last program.

```
function out=fastdisp(vect)
%out=fastdisp(vect)
% Calculates the dispersion of a parameter 'vect'
% Outputs the decimal value of the dispersion of 'vect'

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

q=length(vect);
M=zeros(q-1,2*q-1);  % setup variables, (q-1)x(2q-1) matrix of zeros
for j=2:q            % j from the second slot to the end of the vector
    for i=1:j-1      % since i<j, start i at 1 then go to j-1

% changes component of matrix from 0 to 1 if
% calculated dispersion pair corresponding to the coordinates appears
       M(j-i, vect(j)-vect(i)+q)=1 ;
    end
end

%next line takes sum of matrix m, then the sum of the transpose,
```

```
%   which gives cardinality of D, then plugs into dispersion formula
out=(2*sum(sum(M)'))/(q*(q-1));
```

B.2.7 Program to Determine Cyclic Decomposition Cycles

```
function output=decomposition(vect)
%output=decomposition(vect)
% Calculates the cyclic decomposition of the input permutation 'vect'
% Outputs a matrix containing the n-cycles and how many times each appears
%   ie,  1  3  6
%        4  1  2
%     represents 4 1-cycles, 1 3-cycle, and 2 6-cycles

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

tester=sort(vect);  % the next three lines make sure program operates correctly
offset=tester(1);   % only works w/perms that contain 0; thus, sorts input vector,
vect=vect-offset;   % looks first (now smallest) entry, offsets the vector by that

q=length(vect);     % next five lines set up the intial values of variables
counter=[];
used=[];
index=1;
cycle=[];
while(length(used)<q)
    k=index-1;              % accounts for the off-by-one
    while(k~=vect(index))   % cycles through the perm until back to index value
        cycle=[cycle vect(index)];
        used=[used vect(index)];
        index=vect(index)+1;
    end
    cycle=[cycle vect(index)];  % since cycle added pi(index) as first element,
                                %%   we must specially add in the original index
    used=[used vect(index)];
    counter=[counter length(cycle)];
    used=sort(used);            % preps for comparison below
    i=0;
        while (i<length(used)) && (i==used(i+1))  % find the first index not used
            i=i+1;
        end
        index=i+1;
    cycle=[];               % resets the cycle to empty matrix to begin loop again
end
counter=sort(counter);      % sorts to expedite creation of the output vector
```

```
output=[];
for k=1:length(counter)  % following loop goes adds element to counter if unique
    marker=0;            % or increases tally if element is not unique
    l=1;
    [m,n]=size(output);
    while (marker==0) && (l<=n)
        if counter(k)==output(1,l)
            variable=output(2,l);
            output(2,l)=(output(2,l)+1);
            marker=1;
        end
        l=l+1;
    end
    if marker==0
        output=[output [counter(k);1]];
    end
end
```

B.3 *Field Arithmetic Programs*

B.3.1 Program to Evaluate Addition in a Field

This program to add in a field $\mathbb{F}_{p^n}$ uses the function **gfadd**, which is available only in the Matlab Communications Toolbox.

```
function output=gfaddition(alpha,num,field)
% output=gfaddition(alpha,num,field)
% Performs additions of variables in a Galois Field called 'field'
%   adding 'alpha' to itself 'num' times
%   One example is 2(alpha)^3; num=2 in this case

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

output=alpha;     % sets up default alpha value
                  % (won't enter loop if num=1)
for i=1:num-1     % loops up to the correct number of additions
    output=gfadd(output,alpha,field);
end
```

B.3.2 Program to Evaluate Exponents in a Field

This program to evaluate exponents in a field $\mathbb{F}_{p^n}$ uses the function **gfmul**, which is available only in the Matlab Communications Toolbox.

```
 function
```

```
powr=gfexpo(num,i,field)
%powr=gfexpo(num,i,field)
% Evaluates exponents in a Galois Field, taking a field element num^i by
% multiplying it by itself i times
% ie, (alpha)^6   -->   num=alpha, i=6

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

if i==0  % hard codes the basic default case
    powr=0;
else
    powr=num;
    for j=1:i-1  % loop multiplies num times itself i-1 times
        powr=gfmul(powr,num,field);
    end
end
```

B.3.3 Program to Evaluate Polynomials in a Field

This program to calculate values plugged into a polynomial in a field $\mathbb{F}_{p^n}$ uses the functions **gfadd**, **gfexpo**, and **gfaddition**, which are available only in the Matlab Communications Toolbox.

```
function output=gfpoly(num,vect,field)
%output=gfpoly(num,vect,field)
% Program to evaluate plugging variables like (alpha)^6 into polys like 1+x+x^2
% In the example, num=(alpha)^6 and vect=[1 1 1] (which represents 1+x+x^2)
% Outputs the value of the evaluated polynomial

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

output=-Inf;                    % default output; evaluates to 0 in any field
for j=1:length(vect)            % sets up loop to step through every part of vector
    exponen=gfexpo(num,j-1,field);  % calls exponent to evaluate num to power
    if vect(j)==0               % default value if any element of the vector is 0
        added=-Inf;
    else
        % adds up the exponentiated alphas to deal with coeffiecients
        added=gfaddition(exponen,vect(j),field);
    end
    output=gfadd(output,added,field); % adds new (alpha)^x value to previous add'ns
end
```

B.4 *Random Necessary Programs*

B.4.1 Program to Determine Base Representations

This program takes in a decimal number and a prime power p^n, and calculates the number's representation in the new base.

```
function vect=baserep(num,p,n)
%vect=baserep(num,p,n)
% Calculates the base p representation of a decimal number 'num' up to
%    n digits
% Outputs the base representation of the number as a vector

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

digs=n;   % calculates # of digits required in the end number
vect=[];
for j=1:digs             % loops through every digit
    temp=mod(num,p);     % evaluate mod of number, then
    vect=[vect temp];    % add to vector, then
    num=(num-temp)/p;    % subtract the mod off the original and loop back
end
```

B.4.2 Program to Generate Random Integer

There is a built-in program, **randint**, which is available only in the Matlab Communications Toolbox. In order to use this program on other computers without the Toolbox, we had to copy the code to a separately named Matlab program file.

B.5 *Sorting Programs*

B.5.1 Program to Sort a Set of Vectors

This program requires that the vectors be stored as columns of a matrix. Example: to sort [1 2 3], [8 3 2], and [0 9 3], they should be passed to the program as $M = \begin{bmatrix} 1 & 8 & 0 \\ 2 & 3 & 9 \\ 3 & 2 & 3 \end{bmatrix}$.

```
function data=selectionsort(M,data,first)
%data=selectionsort(M,data,first)
% Sorting algorithm based on swapping the largest value into the last slot,
%    then moving the "last" slot up by one and repeating
% M ~ Monomial ordering;
% data ~ vector of items that need sorting;
% first ~ the first index where sorting should begin
```

```
% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

[m,n]=size(data);   % gets length of data to determine length of sorting
for i=n-1:-1:1      % begins at the latest index moving backwards
    big=first;      % starts the selection of big at the first element
                    %%   then loops upward until the largest element is found
    marker=0;
    for j=first+1:first+i
        if (matbigvects(M,data(:,big),data(:,j))==2)  % uses ordering program
            big=j;
            marker=1;
        end
    end
% the next loop swaps biggest element to last slot, then to the next left position
    if marker==1
        temp = data(:,first+i);
        data(:,first+i)=data(:,big);
        data(:,big)=temp;
    end
end
```

B.5.2 Program to Sort Two Vectors According to a Monomial Ordering

```
function greatvect=matbigvects(M,a,b)
%greatvect=matbigvect(M,a,b)
% Calculates the monomial ordering of vectors 'a' and 'b', both of length n,
%    based on the inputted n by n matrix 'M' of nonnegative real values
% Output: 0 if a=b or if M=0; 1 if a>b; 2 if a<b

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

i=1;  % initial values
flag=4;
funvect=M*(b-a);    % sets up evaluation vector
if M*a==M*b         % outputs zero if a and b are equal, or if M is all zeros
    greatvect=0;
    return;
else
    while (flag==4) && (i<=length(M))  % finds first spot where value not 0
                % by definition of this ordering, if the first spot of difference
                %%   is positive, a<b and the program outputs 2
        if funvect(i)>0
            greatvect=2;
```

```
            flag=7;
                % by definition of this ordering, if the first spot of difference
                %%   is negative, a>b and the program outputs 2
        elseif funvect(i)<0
            greatvect=1;
            flag=5;
        end
        i=i+1;  % if the vectors don't differ at i, loops to the next slot
    end
end
```

B.5.3 Program to Output a Graded Lexicographic Matrix Representation

```
 function matrix=grlex(size)
%matrix=grlex(size)
% Creates a 'size' by 'size' matrix that will represent
%   a graded lexicographic ordering when used
%   in the vector ordering program
% Outputs a 'size' by 'size' matrix

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

% first row is all 1s;
% bottom part is a lex matrix with the last row cut off
matrix=[ones(1,size); eye(size-1,size)];
```

B.5.4 Program to Output a Graded Reverse Lexicographic Matrix Representation

```
function matrix=grevlex(size)
%matrix=grevlex(size)
% Creates a 'size' by 'size' matrix that will represent
%   a graded reverse lexicographic ordering when used
%   in the vector ordering program
% Outputs a 'size' by 'size' matrix

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

matrix=[ones(1,size)];  % Top row of grevlex matrix
temp=eye(size-1,size);  % Sets up bottom part of grevlex
temp=abs(temp-1);       % Switches zeros and ones
i=1;
j=size;
```

```
% Flips columns of the temp to reverse direction of the zero diagonal
while (i<j)
    temp2=temp(:,i);
    temp(:,i)=temp(:,j);
    temp(:,j)=temp2;
    i=i+1;
    j=j-1;
end

matrix=[matrix;temp];   % Combines upper and lower parts
```

B.6 *Permutation Calculation Programs*

B.6.1 Program to Calculate a Permutation in a Field, Using a Specified Ordering and Permutation Polynomial

This program uses a number of programs that depend on the Matlab Communications Toolbox.

```
function output=monomialordermap(M,p,n,j)
%output=monomialordermap(M,p,n,j)
% Calculates the resulting permutation in Z_p^n using monomial ordering 'M',
%    in field of size p^n, and the center permuting polynomial x^j
% Outputs the resulting permutation in vector form

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact


%% Goes from Z(p^n) to (Zp)^n, sorts according to monomial order M
%% (ex: from integers to polynomials sorted by lex)

vect=num2vect(M,p,n);
conversion=vect;  % stores conversion matrix for later


%% Defines the field we are working in

field=fielddefine(p,n);


%% Runs the polynomials (in terms of alpha) through a permutation, which
%% can itself be a polynomial
[c,d]=size(vect);
output=[];
for i=1:d
```

```
    [tp,expform]=gftuple(vect(:,i)' ,n,p);  % converts from (Zp)^n to F(p^n)
    output=[output gfexpo(expform,j,field)];
end
vect=output;


%% Converts the output in F(p^n) back into (Zp)^n
output=[];
for i=1:d
    [tp,expform]=gftuple(vect(i),n,p);  % converts from F(p^n) to (Zp)^n
    output=[output;tp];
end
vect=output';


%% Converts the vector into Z(p^n)
output=vect2num(conversion, vect);
```

For working with polynomials that not are simple monomials the following changes can be made:

(i) The first few lines of the program can be replaced with:

```
function output=monomialordermap(M,p,n,permutation)
%output=monomialordermap(M,p,n,permutation)
% Calculates the resulting permutation in Z_p^n using monomial ordering 'M',
%    in field of size p^n, and the center permuting polynomial 'permutation'
%    with integer coefficients
```

(ii) In the middle of the program, the lines where the polynomials are run through the permutation, the lines can be replaced with:

```
for i=1:d
    [tp,expform]=gftuple(vect(:,i)',n,p);  % converts from (Zp)^n to F(p^n)
    output=[output gfpoly(expform,permutation,field)];
end
```

(Note that this modified version requires modifications of the programs on which monomialordermap depends to accept arguments of polynomials, described as vectors, instead of just an i-value.)

B.6.2 Program to Take the Elements of $\mathbb{Z}_{p^n}$ Into $(\mathbb{Z}_p)^n$, and Order Them by a Monomial Ordering

```
function vect=num2vect(M,p,n)
```

```
% vect=num2vect(M,p,n)
% Creates a list of ordered vectors in the specified base p^n
%    using the elements of Z(p^n), converting them to (Zp)^n,
%    then sorting the resulting elements of (Zp)^n
% Outputs a sorted vector of elements of (Zp)^n

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

vect=[];
for i=0:(p^n)-1     % adds all numbers from 0 to base-1 to a vector
    vect=[vect;baserep(i,p,n)];
end
vect=vect';         % transposes the vector so it sorts correctly
vect=selectionsort(M,vect,1);
```

B.6.3 Program to Take the Elements of $(\mathbb{Z}_p)^n$ Back Into $\mathbb{Z}_{p^n}$ Using the a Specified Conversion Ordering

```
function permvect=vect2num(Q,H)
% permvect=vect2num(Q,H)
% Takes in the conversion table from the num2vect and list of outputted
%    vectors from the permutation
% Q ~ conversion factor matrix;
% H ~ result of the previous step
% Outputs the converted integers

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

[m,n]=size(H);
permvect=zeros(1,n); % sets up vector that ends up outputting integers in order
for i=1:n            % loops through the permutation vectors
    for j=1          % loops until conversion table and perm vectors match
        while ~isequal(H(:,i),Q(:,j))
            j=j+1;
        end
    end
    permvect(i)=j-1;  % adds integer value to specified spot in output vector
end
```

B.6.4 Program to Define the Working Field

This program to calculate a field $\mathbb{F}_{p^n}$ uses the function **gftuple**, which is available only in the Matlab Communications Toolbox.

```
function field=fielddefine(p,n)
%field=fielddefine(p,n)
% Method for constructing a field of size p^n and outputting the
%   proper variable, 'field', for future use

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

field = gftuple([-1:p^n-2]',n,p);   % Constructs list of elements.
```

B.7 *Permutation Creation Programs*

B.7.1 Program to Create Permutations in a Field $\mathbb{F}_{p^n}$ using a Monomial Ordering and to Calculate Measurements of Each Permutation

This program depends on a program that utilizes elements in the Matlab Communications Toolbox.

```
function
[permutation,isperm,sprd,dispers,cyclic,sprdfactor]=allofit(M,p,n,i)
%[permutation,isperm,sprd,dispers,cyclic]=allofit(M,p,n,poly)
% Determines if a polynomial is a permutation, and returns the spread,
% dispersion, and cyclic decomposition if the polynomial is a permutation.
% Where M=Matrix describing monomial order, p=base, n=exponent (ie, p^n),
%   and poly=polynomial to be used
% Outputs the permutation; isperm=1 if poly is a permutation, 0 if not;
%   spread; dispersion; and cyclic decomposition

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

permutation=monomialordermap(M,p,n,i);  % permutation creation
isperm=testperm(permutation,p,n);       % tests if a permutation
if (isperm==1)                        % calcs and stores measures of a perm
    sprd=spread(permutation);
    dispers=dispersion(permutation);
    cyclic=decomposition(permutation);
    sprdfactor=permdist(permutation);
else
    sprd='error - your permutation sucks';
    dispers='error - your permutation sucks a lot';
    cyclic='error - why oh why did you pick that permutation';
    sprdfactor='error - even the spread factors hate your permutation!';
end
```

B.7.2 Program to Test if a Candidate is a Permutation

```
function output=testperm(perm,p,n)
%output=testperm(perm,p,n)
% Tests a candidate to see if it is a permutation
% Outputs 1 if the candidate is a permutation, 0 if it is not

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

output=1;
flag=0;
perm=sort(perm);
i=0;
% after sorting, the vector is in ascending order starting at zero,
% so the loop starts i at zero and checks to make sure that the ith
% slot of the sorted permutation equals i
while (flag==0) && (i<p^n)
    if i~=perm(i+1)
        output=0;
        flag=1;
    end
    i=i+1;
end
```

B.8 *Permutation Determination, Storage, and Retrieval Programs*

B.8.1 Program to Calculate Measurements for Permutations in a Field

```
 function allinone(M,p,n,filename)
%allinone(M,p,n,print)
% Determines whether each of the polynomials making up the field are a
% permutation, and prints out the spread, dispersion, and cyclic
% decomposition of the polynomials.
% M ~ Matrix determining monomial ordering  (dimensions n by n)
% p ~ prime number
% n ~ power of prime number  (p^n makes up the order of the field, ie F_p^n)
% print ~ type 1 to print out poly, sprd, disp, and decomp for each
%   polynomial even if not a permutation, type 0 to just print the
%   information for polynomials that are permutations

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact
```

```
perms=[];
ivalues=[];
ordering=M;
spreads=[];
dispersions=[];
sprdfactors=[];
for i=0:(p^n-1)  % loop to set up, calc all the p^n polys w/integer coeff
    if (gcd(i,(p^n-1))==1)
% calls the allofit function to check whether each polynomial generates a
% permutation; if it does, the next lines output the relevant values
        [per,isperm,sprd,disp,decomp,sprdfactor]=allofit(M,p,n,i);
        i
        per;
        sprd;
        disp;
        decomp;
        sprdfactor;

        perms=[perms;per];
        ivalues=[ivalues;i];
        spreads=[spreads; sprd];
        dispersions=[dispersions; disp];
        sprdfactors=[sprdfactors; sprdfactor];
    end
end
        % setup to store variables
        % perms must have 1 added to accomodate turbo_simulator
polys=ivalues;
perms=perms+1;
save(filename,'polys','perms','ordering','spreads','dispersions','sprdfactors')
```

B.8.2 Program to Read Measurements from a Field File

```
function [po,or,sp,di,sf]=readallinonemeasures(filename)
%[po,or,sp,di,sf]=readallinonemeasures(filename)
% Reads the stored polynomials, ordering, spreads, dispersions, and spread
%    factors from a specified field file.
% Outputs five variables to that contain the polynomials, the ordering, the
%    spreads, the dispersions, and the spread factors in that order.

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

load(filename,'polys','ordering','spreads','dispersions','sprdfactors')
po=polys;              % the next five lines set the return variables
```

```
or=ordering;
sp=spreads;
di=dispersions;
sf=sprdfactors;
```

B.8.3 Program to Read Permutations from a Field File

```
function [po,pe]=readallinoneperm(filename)
%[po,pe]=readallinoneperm(filename)
% Reads the stored polynomials and permutations from a specified field file.
% Outputs two variables to that contain the polynomials and the
%   permutations, in that order.

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

load(filename,'perms','polys')
pe=perms;
po=polys;
```

B.9 *Interleaver Simulation Programs*

B.9.1 Program to Run Multiple Simulations on a Field File

```
function simfromfile(filename,nums)
%simfromfile(filename,nums)
% Runs specified number of simulations, 'nums', on the field file.
% Saves simulation data for each permutation to an extended version of the
%    original name; ie, for lex_f2_2, one might be saved as lex_f2_2_2.

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

load(filename,'perms') m=length(nums);
a=readallinonemeasures(filename)'           % gets all i values
for i=1:m
            % j loop finds index where each i value is located
    for j=1:length(a)
        if nums(i)==a(j)
            use=j;
            nums(i);
        end
    end
            % creates extended filename, and runs the specified simulation
    filei=[filename,'_',num2str(nums(i))];
```

```
    turbo_simulator(perms(use,:),filei);
end
```

B.9.2 Program to Read Bit Error Rates from a Field File

This program can only be run after successful simulation of the permutations. It also requires that all the files associated with the field file be in the same directory as the original field file. Example: After simulating lex_f2_2, you may have the associated field files lex f2_2_1 and lex_f2_2_2. The latter files *must* be in the same directory as the original field file for this program to work.

```
function output=readber(filename,ivect)
%output=readber(filename,ivect)
% Reads the specified Bit Error Rates from a field file.
% filename ~ name of field file
% ivect ~ vector listing requested i-values
% Outputs requested Bit Error Rates

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

q=length(ivect);
output=zeros(q,5);
for i=1:q     % loops through the i values and stores the BERs for output
    filei=[filename,'_',num2str(ivect(i))];
    load(filei,'ber');
    output(i,:)=ber(:,length(ber))';
end
```

B.9.3 Program to Graph Specified Bit Error Rates

```
function ber_graph(filename,ivect,len)
%ber_graph(filename,ivect,len)
% Graphs the specified BERs for each signal to noise ratio
% filename ~ filename to get data from
% ivect ~ specifies which values of x^i to plot
% len ~ hack to make legend come out right; this is the number
%       of digits of the highest i-value

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

newplot
q=length(ivect);
hold on   % needed to graph multiple graphs; gets this done early on
legi=[];
```

```
color=[’y’ ’m’ ’c’ ’r’ ’g’ ’k’];   % vectors that set up matlab line styles
line=[’-’,’:’,’-.’,’--’];
style=[’.’ ’o’ ’x’ ’+’ ’s’ ’d’ ’v’ ’^’ ’p’ ’h’ ’<’ ’>’];
for i=1:q
    filei=[filename,’_’,num2str(ivect(i))];  % ith filename created
    load(filei,’EbN0db’,’ber’)    % loads BERs from the ith file
    [m,n]=size(ber);
    EbNoveci=EbN0db;              % vector of signal to noise ratios
    BERveci=ber(:,n);

% the randint program is in the Matlab Communications Toolbox, so one would
% have to create his or her own version to run the next three lines
        l = randint(1,1,[1,length(line)]);  % picks random style
        s = randint(1,1,[1,length(style)]);
        c = randint(1,1,[1,length(color)]);
        plotsym= [style(s) line(l) color(c)]; % random plotstyle for this curve
    semilogy(EbNoveci,BERveci,plotsym);           % actually plots curve

    tempvect=num2str(ivect(i));
    p=length(tempvect);
    if p~=len                 % creates line in legend for each line graphed
        for j=p:len-1
            tempvect=[tempvect,’‘’];   % add in hack to make legend work
        end
    end
    addin=strcat(’x->x^’,tempvect);
    legi=[legi;addin];
end

% adds in graph axis labels, title, and legend
grid on;
title(’Performance of Turbo Codes Using Monomial Permutations’);
xlabel(’Signal to Noise Ratio (EbNo (dB))’);
ylabel(’Bit Error Rate (BER)’);
legend(legi,’Location’,’Southwest’);

hold off
```

B.10 *Programs to Create and Examine Semi-Random Permutations*

B.10.1 Program to Generate a Semi-Random Permutation

```
function perm=randomp(s,q)
%perm=randomp(s,q)
```

```
% input s and q, s=desired spread, q=field size

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

perm=zeros(1,q);          % a 1xq zero matrix
available=ones(1,q);      % available possibilities to choose from
pofflimits=zeros(1,q);    % values off limits permanently
entry=randint(1,1,[1,q]); % picks a random number to insert into perm
perm(1)=entry;    % randomly picks perms first entry without restrictions
for i=2:q
    available(entry)=0;   % changes indices of available matrix to 0
    pofflimits(entry)=1;  % changes pofflimits indices to 1
    offlimits=pofflimits; % sets offlimits to pofflimites
    max1=max(1,i-(s-1));  % next 2 lines pick range to test spread
    w=perm(max1:i-1);

% enter things into offlimits by test spread, dist from other things
    for j=1:length(w)
        max2=max(1,w(j)-(s-1));
        min1=min(q,w(j)+(s-1));
        offlimits(max2:min1)=1;
    end
    available=ones(1,q)-offlimits;   % subtract the ones from offlimits
    v=find(available==1);            % finds spaces in available equal to 1
    if isempty(find(v)), break, end; % tests to see if there are availables
    e=randint(1,1,[1,length(v)]);    % picks randint between 1, length avaiables
    entry=v(e);                      % makes entry the index of random integer
    perm(i)=entry;                   % puts this into perm
end
```

B.10.2 Program to Keep Searching for a Semi-Random Permutation

```
function permy=reallyrandomp(s,q)
%permy=reallyrandomp(s,q)
% Runs really random p until it makes a real permutation
% Outputs an actual semi-random perm

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

flag=0;
while flag==0
    permy=randomp(s,q);
    if ~(permy(q)==0)
```

```
        flag=1;
    end
end
```

B.10.3 Program to Simulate Random Permutation

```
function storereallyrandomp(s,q,filename)
%storereallyrandomp(s,q,filename)
% Runs turbo_simulator with generated semi-random permutation
% Stores information in specified filename
%   - use readberrandom to retrieve information

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

perm=reallyrandomp(s,q)
turbo_simulator(perm,filename);
```

B.10.4 Program to Retrieve Information from a Simulated Permutation

```
function readberrandom(filename)
%output=readberrandom(filename,ivect)
% Reads the specified Bit Error Rates from a field file
%    created by a random permutation.
% filename ~ name of field file
% Outputs requested Bit Error Rates

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

output=zeros(1,5);                % preallocating output vector
load(filename,'ber','alpha');     % load bit error rates
[m,n]=size(ber);                  % get size of BER matrix
output(1,:)=ber(:,n)';            % set up correct outputs

% lines print permutation, spread, dispersion, sprfactor to screen
perm=alpha
spr=spread(alpha)
disp=dispersion(alpha)
sprfact=permdist(alpha)
permber=output
```

B.10.5 Program to Examine Dispersions of Random Permutations

```
function d=randomdispersionloop(s,q,goes)
%d=randomdispersionloop(s,q,goes)
```

```
% Creates several permutations of a given spread and field size, then
%    computes the dispersion.
% s=spread, q=field size, goes=# of permutations

% AMSSI 2006 - Mod 4 Armada
% Public Domain as long as above lines remain intact

d=zeros(1,goes);    %initial dispersion matrix as zeros
for k=1:goes        %runs specified amount of times

%calculates dispersions of perms generated, outputs as vector
    d(k)=fastdisp(reallyrandomp(s,q));
end
```