X-Code: MDS Array Codes with Optimal Encoding *

Lihao Xu Jehoshua Bruck
California Institute of Technology
Mail Code 136-93
Pasadena CA 91125
Email: {lihao,bruck}@paradise.caltech.edu

Abstract

We present a new class of MDS array codes of size $n \times n$ (n a prime number) called X-Code. The X-Codes are of *minimum column distance* 3, namely, they can correct either one *column error* or two *column erasures*. The key novelty in X-code is that it has a simple geometrical construction which achieves encoding/update optimal complexity, namely, a change of any single information bit affects *exactly two* parity bits. The key idea in our constructions is that all parity symbols are placed in *rows* rather than columns.

Keywords: MDS Codes, array codes, update complexity, optimal updates, balanced computation

*Supported in part by the NSF Young Investigator Award CCR-9457811 , by the Sloan Research Fellowship, and by DARPA and BMDO through an agreement with NASA/OSAT.

1 Introduction

MDS (*Maximum Distance Separable*) array codes whose minimum Hamming distance attains the Singleton bound for a given length and dimension [4] have important applications in communication and storage systems [5] [6], and have been studied extensively [1] [2] [3] [7]. A common property of these codes is that the encoding and decoding procedures use only simple XOR and cyclic shift operations, and thus are more efficient than Reed-Solomon codes in terms of computation complexity [5]. In this paper, we present *X-Code*, a new class of MDS array codes of distance 3. Similar to the codes in [1][3], the error model of X-Code is that errors or erasures are columns of the array, i.e., if one symbol of a column is an error or erasure, then the whole column is considered to be an error or erasure.

One important parameter of array codes is the average number of parity bits affected by a change of a single information bit in the codes, called the *update complexity* in this paper. This parameter is particularly crucial when the codes are used in storage applications that need frequent updates of information. The codes in [3] use two *dependent* parity columns to make the distance of the codes to be 3. But the dependency between the two parity columns makes update of one information symbol affecting virtually all the parity symbols. So the update complexity of the codes in [3] increases linearly with the number of the columns of the array codes, just similar to Reed-Solomon codes. To overcome this drawback, the $EVENODD$ codes [1] and their generalizations [2] were designed based on *independent* parity columns resulting in a more efficient information update. The update complexity of $EVENODD$ codes approaches 2 as the number of the columns of the codes increases. But it was proven in [2] that for any linear array codes with only parity columns, the update complexity is always *strictly* larger than 2 (the obvious lower bound). Hence, we asked the following question: Is the update complexity of 2 achievable for general array codes? A positive answer to the foregoing question was given a decade ago [8], and the code in [8] was described by its *parity check matrix* and represented recently in a clearer form, also by a parity check matrix, in [4]. Here we construct a new family of array codes, called X-Codes, which has a simple *geometrical structure* and has an update complexity of *exactly* 2.

Both the X-Codes and the codes in [8] and [4] are combining information and parity symbols within columns in order to achieve optimal update complexity. The redundancy of X-Code is obtained by adding two parity *rows* rather than two parity columns, which results in the nice property that update of one information symbol affects only *two* parity symbols, i.e., the *update complexity* is always 2. In addition, the number of operations for computing parity symbols at every column is the same, namely, the computational load is evenly distributed among all the columns, thus the bottleneck effects of repeated *write* operations are naturally overcome.

The main contribution of this paper is constructing X-Code, a new class of MDS array codes of distance 3, with the properties of optimal update complexity and balanced computations. The simple geometrical structure of X-Code makes its decoding very efficient, for both *two erasures* and *one error*.

This paper is organized as follows. In Section 2, the encoding scheme of X-Code is described, and a proof of its *MDS* property is presented. In Section 3, we provide an efficient decoding algorithm for correcting two erasures, as well as an efficient algorithm for correcting one error. Section 4 concludes the paper and presents some future research directions.

2 X-Code Description

In X-Code, information symbols are placed in an array of size $(n-2) \times n$. Symbols can be defined over any Abelian groups with an addition operation $+$. If the group is the addition group over $\mathrm{GF}(2^m)$, then the addition operation is the usual bit-wise *XOR* operation.

Like other array codes [1] [2] [3] [7], parity symbols are constructed from the information symbols along several *parity check lines* or *diagonals* of some *slopes* with the addition operation $+$. But instead of being put in separate columns, the parity symbols of the X-Code are placed in *two* additional *rows*. So the coded array is of size $n \times n$, with the first $n-2$ rows containing information symbols, and the last two rows containing parity symbols. Notice that each column has information symbols as well as parity symbols, i.e., information symbols and parity symbols are mixed in each column. Errors or erasures can happen in any column. If an error or an erasure occurs to a symbol in a column, then this column is considered to be an error or erasure column. By the structure of the code, if two columns are erasures, the number of remaining symbols is $n(n-2)$, and is equal to the number of original information symbols, which makes it possible for X-Code to recover the two column *erasures*.

2.1 Encoding Procedure

Let $C_{i,j}$ be the symbol at the ith row and jth column, the parity symbols of X-Code are constructed according to the following encoding rules:

$$
\begin{aligned}
C_{n-2,i} &= \sum_{k=0}^{n-3} C_{k,\langle i+k+2\rangle_n} \\
C_{n-1,i} &= \sum_{k=0}^{n-3} C_{k,\langle i-k-2\rangle_n}
\end{aligned} \tag{1}
$$

where $i = 0, 1, \cdots, n-1$, and $\langle x\rangle_n = x \bmod n$.

Geometrically speaking, the two parity rows are just the checksums along diagonals of slopes 1 and -1 respectively. More clearly, to get a parity check row, an imaginary 0-row is placed after the parity check row, then in the resulting array of $n \times n$, checksums of all diagonals of slope 1 (or slope -1) in the square array are all 0s. Thus the two parity rows are constructed *independently* . The following example gives a construction of X-Code of size 5×5.

Example 1 X-Code of size 5×5.

The first parity row is calculated along the diagonals of slope 1, with the last row being only an imaginary 0-row, as follows:

$\triangle$	$\clubsuit$	$\diamondsuit$	$\heartsuit$	$\spadesuit$		1	0	0	1	1
$\spadesuit$	$\triangle$	$\clubsuit$	$\diamondsuit$	$\heartsuit$		0	1	0	1	1
$\heartsuit$	$\spadesuit$	$\triangle$	$\clubsuit$	$\diamondsuit$		0	0	1	0	1
$\diamondsuit$	$\heartsuit$	$\spadesuit$	$\triangle$	$\clubsuit$	←*1st parity check row*→	0	0	1	1	0
$\clubsuit$	$\diamondsuit$	$\heartsuit$	$\spadesuit$	$\triangle$	←*imaginary 0-row* →	0	0	0	0	0

The second parity row is calculated along the diagonals of slope -1, as follows:

$\triangle$	$\clubsuit$	$\diamondsuit$	$\heartsuit$	$\spadesuit$		1	0	0	1	1
$\clubsuit$	$\diamondsuit$	$\heartsuit$	$\spadesuit$	$\triangle$		0	1	0	1	1
$\diamondsuit$	$\heartsuit$	$\spadesuit$	$\triangle$	$\clubsuit$		0	0	1	0	1
$\heartsuit$	$\spadesuit$	$\triangle$	$\clubsuit$	$\diamondsuit$	←*2nd parity check row*→	1	1	0	1	1
$\spadesuit$	$\triangle$	$\clubsuit$	$\diamondsuit$	$\heartsuit$	←*imaginary 0-row*→	0	0	0	0	0

Then the complete codeword is

1	0	0	1	1
0	1	0	1	1
0	0	1	0	1
0	0	1	1	0
1	1	0	1	1

From the construction of X-Code, it is easy to see that the two parity rows are obtained independently, more specifically, each information symbol appears exactly *once* in each parity row, and all parity symbols only depend on information symbols, but *not* on each other. So updating one information symbol results in updating only *two* parity symbols. But from the *MDS property* which will be proven soon, the code is of column distance 3, i.e., the code can correct up to *two* column erasures. So each information symbol must appear at least in *three* columns, otherwise, once all the columns containing this information symbol are erased, the information symbol can never be recovered, which contradicts with the *MDS* property. So the lower bound of the update complexity for any codes of column distance 3 is 2. Thus X-Code has the optimal encoding (or update) property, i.e., it achieves the lower bound 2 of the update complexity.

It is also easy to see that X-Code is a cyclic code in terms of columns, i.e., cyclically shifting columns of a codeword of X-Code results in another codeword of X-Code.

In addition, notice that each column has two parity symbols, each of which is the checksum of $n-2$ information symbols, thus the number of computations (group additions) for parity symbols at each column is $2(n-3)$. This balanced computation property of X-Code is very useful in applications that require evenly distributed computations.

2.2 The MDS Property

In this section, we state and prove the *MDS property* of X-Code.

Theorem 1 (MDS Property)
X-Code has column distance of 3, i.e., it is MDS, if and only if n *is a prime number.*

Proof: Let us start with the *sufficient* condition, namely, we need to prove that for any prime number n, X-Code is *MDS*.

First observe that X-Code is a linear code, thus proving that the code has column distance of 3 *is equivalent to* proving that the code has *minimum column weight* w_{min} of 3, i.e., a valid codeword of X-Code has at least 3 nonzero columns. We will prove it by contradiction.

From the construction of X-Code, checksumming is done along diagonals of slope 1 or slope -1, it is impossible to have only *one* nonzero column, thus $w_{min} > 1$.

Now suppose $w_{min} = 2$, then without loss of generality because of the column cyclic property of X-Code, we can assume the nonzero columns are the 0th and kth columns where $1 \leq k \leq n-1$. Denote the ith symbol of the 0th and kth columns by a_i and b_i respectively.

Observe that one diagonal of slope 1 or -1 only traverses $n-1$ columns, then among the diagonals of slope 1, the diagonal crossing a_{n-1-k} does *not* cross any symbol of the kth column, and the diagonal crossing b_{k-1} does *not* cross any symbol of the 0th column, so $a_{n-1-k} = 0$ and $b_{k-1} = 0$. Because of the same property of the diagonals of slope -1, we can also get $a_{k-1} = 0$ and $b_{n-1-k} = 0$ (or $b_{n-1} = 0$ if $k = 1$).

Starting from $a_{k-1} = 0$, we get $b_{2k-1} = 0$, since they are in same the diagonal of slope 1; then we get $a_{3k-1} = 0$, since it is on the same diagonal of slope 1 with $b_{2k-1}, \cdots$, and so on, we have

$$a_{k-1} = a_{3k-1} = a_{5k-1} = \cdots = a_{(n-2)k-1} = 0$$

and

$$b_{2k-1} = b_{4k-1} = b_{6k-1} = \cdots = b_{(n-1)k-1} = 0$$

all indices above are *mod* n.

Similarly, starting from $a_{n-1-k} = 0$, we have

$$a_{n-1-k} = a_{n-1-3k} = \cdots = a_{n-1-(n-2)k} = 0$$

and

$$b_{n-1-2k} = b_{n-1-4k} = \cdots = b_{n-1-(n-1)k} = 0$$

again, all indices above are *mod* n.

We can describe the above 4 sets of entries in the array as follows. Let $A_0 = \{\langle(2m+1)k-1\rangle_n : m = 0, 1, \cdots, \frac{n-3}{2}\}$, and $A_1 = \{\langle n-(2l+1)k-1\rangle_n : l = 0, 1, \cdots, \frac{n-3}{2}\}$, let $B_0 = \{\langle 2mk-1\rangle_n : m = 1, 2, \cdots, \frac{n-1}{2}\}$, and $B_1 = \{\langle n-2lk-1\rangle_n : l = 1, 2, \cdots, \frac{n-1}{2}\}$, notice that all the sets do not include $n-1$, since n is prime. This can also be seen from the construction of X-Code, since the $(n-1)$th row is just an imaginary all-0 row and it does not need to be considered. An illustration of the above sets for n = 5 and k = 2 is as follows:

A_0		B_1		
A_0		B_1		
A_1		B_0		
A_1		B_0		

Since n is prime, for any $1 \leq k \leq n-1$, $\gcd(n,k)=1$, $\|A_0\| = \|A_1\| = \frac{n-1}{2}$, and if there were such m and l that

$$(2m+1)k - 1 \equiv n - (2l+1)k - 1 \bmod n \tag{2}$$

i.e.,

$$2(m+l+1)k \equiv 0 \bmod n \tag{3}$$

but $1 \leq m+l+1 \leq n-2, \gcd(m+l+1,n)=1, \gcd(2k,n)=1$, so it is impossible to have such a pair of m and l, i.e., $\|A_0 \cap A_1\| = 0$. Notice that $n-1 \equiv (2\frac{n-1}{2}+1)k-1 \bmod n$, we have

$$A_0 \cup A_1 = \{0, 1, \cdots, n-2\}$$

Similarly,

$$B_0 \cup B_1 = \{0, 1, \cdots, n-2\}$$

So all the first $n-1$ symbols in the 0th and the kth columns are 0's, obviously the last symbols in the 0th and the kth columns should be also 0's. Thus, $w_{min} \geq 3$, but it is easy to see there is a codeword of column weight 3, so $w_{min} = 3$. This concludes the proof for the sufficient condition.

Now from the equation Eq. (3), it can be seen that if n was not a prime number, i.e., n could be factored into two factors n_1 and n_2, let $k = n_1$, and $m+l+1 = n_2$, we got a solution (k,l,m) for the equation Eq. (3) or Eq. (2), where $2 \leq k \leq n-1$, i.e., we got a codeword of weight 2 (see (2) in the following remark for an example). Because the code is linear, the code is of distance 2, this contradicts with the fact that the code is *MDS*, i.e., of distance 3. So n being a prime number is a necessary and sufficient condition to the *MDS* property of X-Code. $\square$

Remarks:

1. For the sufficient condition, we can always find a diagonal of one slope which traverses only one of the two columns, and thus the traversed symbol must be 0. Starting from this 0-symbol, use the diagonal of the other slope crossing this symbol, we can determine the crossed symbol by the diagonal in the other column must be also 0. So this saw-like recursive procedure can proceed until it hits a parity symbol at one of the two columns, since a parity symbol can only lie in one diagonal. We call this saw-like recursion as a *decoding chain*. Since there are four parity symbols at the two columns,

there are at most four decoding chains. (A simple calculation can show that there are two decoding chains when $k = 1$ and four decoding chains otherwise.) The procedure of getting the decoding chains will stop with all the symbols at the two columns as 0s if n is prime. Since this procedure is deterministic once the positions of the two columns are given, it also provides an efficient erasure decoding algorithm.

2. For the necessary condition, if n is not prime, then we can always choose k as a factor of n, and as long as all the symbols in the two columns whose indices are in the sets A_0, A_1, B_0 and B_1 are 0s, the obtained codeword is a valid codeword of weight 2. For example, if n = 6, then let $k = 2$, we get $A_0 = A_1 = B_0 = B_1 = \{1, 3\}$, i.e., keep $a_1 = a_3 = b_1 = b_3 = 0$, we can get a valid codeword of weight 2 as follows:

1	0	1	0	0	0
0	0	0	0	0	0
1	0	1	0	0	0
0	0	0	0	0	0
1	0	1	0	0	0
1	0	1	0	0	0

3. In the code construction above, we use diagonals of slopes 1 and -1. This choice of slopes is not unique. In fact, codes constructed by the pair of slopes $(s, -s)$, where $s = 1, \cdots, \frac{n-1}{2}$, are *MDS* if and only if n is prime. The proof is similar to the case where the slope pair is (1,-1). It seems that other slope pairs don't provide advantages over (1,-1), so in this paper we will focus on X-Codes generated by the slope (1,-1).

3 Efficient Decoding Algorithms

In this section, we present decoding algorithms for correcting two erasures or one error of X-Code. As the encoding algorithm of the code, decoding algorithms do not require any finite field operations. Instead, the only operations needed are just cyclic shifts and *XOR*s, which can be implemented very efficiently with software and/or hardware. It is clear how to correct one erasure, since the erasure can be easily recovered along one of the diagonals. So we will proceed with correcting two erasures.

3.1 Correcting Two Erasures

First notice that in an array of size $n \times n$, if two columns are erasures, then the basic unknown symbols of the two columns are the information symbols. So the number of unknown symbols is $2(n-2)$. On the other hand, in the remaining array, there are $2(n-2)$ parity symbols which include all the $2(n-2)$ unknown symbols. Hence correcting the two erasures is only a problem of solving $2(n-2)$ unknowns from the $2(n-2)$ linear equations. From the proven fact that X-Code is of distance 3, it can correct two erasures, thus the $2(n-2)$ linear equations must be linearly independent, i.e., the linear equations are solvable. Now notice that the code has such a feature, as proven in section 2, that no two information symbols in a same column can appear in one parity symbol, each equation has at most two unknown symbols, and some equation has only one unknown symbol. This drastically reduces the complexity of solving the equations. We can always start from the equation (or the parity symbol, the determination of whose position will be discussed soon) with only one unknown symbol in it, then recursively go to the other parity symbol including this just solved symbol to get another unknown symbol until all the unknown symbols are solved.

Suppose the erasure columns are the ith and jth ($0 \le i < j \le n-1$) columns, since each diagonal traverses only $n-1$ columns, if a diagonal crosses a column at the last row, no symbols of that column are included in this diagonal. This determines the position of the parity symbol including only one symbol of the two erasure columns, and the symbol can be immediately recovered from the simple checksum along this diagonal.

First consider the diagonals of slope 1. Suppose the xth symbol of the ith column is the only unknown symbol in a diagonal, then this diagonal hits the jth column at the $(n-1)$th row, and hits the first parity row at the yth column, i.e., the three points $(x, i), (n-1, j)$ and $(n-2, y)$ are on the same diagonal of slope 1, thus the following equations hold:

$$\begin{cases} (n-1) - x \equiv j - i \bmod n \\ (n-1) - (n-2) \equiv j - y \bmod n \end{cases}$$

Since $1 \le j - i \le n-1$, and $0 \le j-1 \le n-2$, the solutions for x and y are

$$\begin{cases} x = \langle (n-1) - (j-i) \rangle_n = (n-1) - (j-i) \\ y = \langle j-1 \rangle_n = j-1 \end{cases}$$

So from the parity symbol $C_{n-2,j-1}$, we can immediately get the symbol $C_{(n-1)-(j-i),i}$ in the ith column. Similarly, the symbol $C_{(j-i)-1,j}$ in the jth column can be solved directly from the parity symbol $C_{n-2,\langle i-1 \rangle_n}$.

Symmetrically with the diagonals of slope -1, the symbol $C_{(j-i)-1,i}$ in the ith column can be solved from the parity symbol $C_{n-1,\langle j+1 \rangle_n}$, and the symbol $C_{(n-1)-(j-i),j}$ in the jth column can be solved from the parity symbol $C_{n-1,i+1}$.

Notice that an information symbol is crossed by the diagonals of slope 1 and -1 exactly once respectively, so if an unknown symbol is solved along a diagonal of slope 1 (or -1), then from the parity symbol along the diagonal of slope -1 (or 1) which crosses the solved symbol, another unknown symbol in the other column can be solved. This procedure can be used recursively until the parity symbol is in an erasure column or the solved symbol itself is a parity symbol.

Suppose the erasures are the ith and jth ($0 \le i < j \le n-1$) columns, and the parity rows are P_0 and P_1 for the diagonals of slopes 1 and -1 respectively, i.e., $P_0[k] = C_{n-2,k}$ and $P_1[k] = C_{n-1,k}$ for $0 \le k \le n-1$, then a formal algorithm for correcting the two erasures in X-Code can be described as follows:

Algorithm 1 (Correcting Two Erasures)

1. *Init_Slope_Set* = { *1, 1, -1, -1* }
   $Init\_Par\_Col\_Set = \{j-1, \langle i-1\rangle_n, \langle j+1\rangle_n, i+1\}$;
   $Init\_Sym\_Col\_Set = \{i, j, i, j\}$;
   $Init\_Sym\_Row\_Set = \{(n-1)-(j-i), (j-i)-1, (j-i)-1, (n-1)-(j-i)\}$;
   $i = -1$;
2. $i++$;
   If $i == 4$ **Then**
      Compute $P_0[i], P_0[j], P_1[i], P_1[j]$ *according to the encoding rule Eq. (1);*
      Stop;
   Else
      $Slope = Init\_Slope\_Set[i]$;
      $Par\_Col = Init\_Par\_Col\_Set[i]$;
      $Sym\_Col = Init\_Sym\_Col\_Set[i]$;
      $Sym\_Row = Init\_Sym\_Row\_Set[i]$;
   End If
3. **If** $Par\_Col == i$ **Or** $Par\_Col == j$ **Then**
      Goto *2;*
   Else
      If $Slope == 1$ **Then**
         $C_{Sym\_Row,Sym\_Col} = P_0[Par\_Col] + \sum_{k=0,k\neq Sym\_Row}^{n-3} C_{k,\langle Par\_Col+k+2\rangle_n}$;
      Else
         $C_{Sym\_Row,Sym\_Col} = P_1[Par\_Col] + \sum_{k=0,k\neq Sym\_Row}^{n-3} C_{k,\langle Par\_Col-k-2\rangle_n}$;
      End If
   End If
4. $Slope = -Slope$;
   $Par\_Col = \langle Sym\_Col - Slope * (Sym\_Row + 2)\rangle_n$;
   If $Sym\_Col == i$ **Then**
      $Sym\_Col = j$;
   Else
      $Sym\_Col = i$;
   End If
   $Sym\_Row = \langle n - 2 - Slope * (Par\_Col - Sym\_Col)\rangle_n$;
   Goto *3;*

□

Step 1 of the algorithm computes the positions of the four parity symbols with only one unknown symbol. Step 2 through 4 include the recursive procedure described above. Step 3 is just the checksum calculation along a diagonal of slope *Slope* crossing the parity symbol $P_0[Par\_Col]$ (or $P_1[Par\_Col]$) to recover the unknown symbol C_{Sym_Row,Sym_Col} if the parity symbol is not in one of the erasure columns, otherwise just restarts with another parity symbol obtained in Step 1. Step 4 decides the position of the diagonal including the next unknown symbol from the just solved symbol. The complexity of the algorithm is easy to analyze. Each iteration solves one unknown symbol, and it needs $(n-3)$ XOR operations. So to correct two erasure columns, the decoding algorithm needs $2n(n-3)$ XOR operations, just the same as that of the encoding algorithm.

Following is a simple example to show how the decoding algorithm works. To be more general, we would use symbols rather than numerical values.

Example 2 Correcting Two Erasures of a 5×5 X-Code

Without loss of generality, we assume the last (i.e., the 4th) column is one of the erasures, and because of the symmetry of the code, we only need to examine the cases where the other erasure is 3rd or 2nd column.

Case 1. i = 3, j = 4

Then the remaining array is follows :

a_0	a_1	a_2	$?(a_3)$	$?(a_4)$
b_0	b_1	b_2	$?(b_3)$	$?(b_4)$
c_0	c_1	c_2	$?(c_3)$	$?(c_4)$
$d_0 = a_2 + b_3 + c_4$	$d_1 = a_3 + b_4 + c_0$	$d_2 = a_4 + b_0 + c_1$	$?(d_3)$	$?(d_4)$
$e_0 = a_3 + b_2 + c_1$	$e_1 = a_4 + b_3 + c_2$	$e_2 = a_0 + b_4 + c_3$	$?(e_3)$	$?(e_4)$

After omitting the obvious checksum calculations, the solution chain for the erasures would be as follows:

$$a_4(d_2) \rightarrow b_3(e_1) \rightarrow c_4(d_0)$$
$$a_3(e_0) \rightarrow b_4(d_1) \rightarrow c_3(e_2)$$

Each chain above represents a recursion starting from a parity symbol, and in each term of the chain, $x(y)$ means that the symbol x can be recovered from the parity symbol y. Obviously, d_3, d_4, e_3 and e_4 can be easily computed after all others are known.

Case 2. i = 2, j = 4

Then the remaining array is follows :

a_0	a_1	$?(a_2)$	a_3	$?(a_4)$
b_0	b_1	$?(b_2)$	b_3	$?(b_4)$
c_0	c_1	$?(c_2)$	c_3	$?(c_4)$
$d_0 = a_2 + b_3 + c_4$	$d_1 = a_3 + b_4 + c_0$	$?(d_2)$	$d_3 = a_0 + b_1 + c_2$	$?(d_4)$
$e_0 = a_3 + b_2 + c_1$	$e_1 = a_4 + b_3 + c_2$	$?(e_2)$	$e_3 = a_1 + b_0 + c_4$	$?(e_4)$

Now the solution chain becomes as follows:

$$c_2(d_3) \rightarrow a_4(e_1)$$
$$b_4(d_1)$$
$$b_2(e_0)$$
$$c_4(e_3) \rightarrow a_2(d_0)$$

Again, d_2, d_4, e_2 and e_4 are easy to get after all other symbols are obtained.

□

3.2 Correcting One Error

To correct one error, the key is to locate the error position. This can be done by computing two *syndrome* vectors from the two parity rows. Since the error is a column error, it is natural to compute the syndromes with respect to *columns* than to *rows* as in the encoding procedure. Once the error location is found, the value of the error can be easily computed along the diagonals of either slope.

Suppose $R = [r_{i,j}]_{0 \le i,j \le n-1}$ is the error-corrupted array, then construct two arrays $U = [u_{i,j}]_{0 \le i,j \le n-1}$ and $V = [v_{i,j}]_{0 \le i,j \le n-1}$ from R, where for $0 \le j \le n-1$,

$$u_{i,j} = v_{i,j} = r_{i,j}, \quad 0 \le i \le n-3 \tag{4}$$

$$u_{n-2,j} = r_{n-2,j}, \; v_{n-2,j} = r_{n-1,j} \tag{5}$$

$$u_{n-1,j} = v_{n-1,j} = 0 \tag{6}$$

$$\tag{7}$$

i.e., U and V are constructed by copying the $n-1$ information rows and parity rows accordingly from R, then adding an imaginary 0-row at the last row, From U and V, compute two *syndrome* vectors S_0 and S_1 as follows:

$$S_0[i] = \sum_{k=0}^{n-1} u_{i+k,k} \tag{8}$$

$$S_1[i] = \sum_{k=0}^{n-1} v_{i-k,k} \tag{9}$$

all sub-indices above are mod n.

It is easy to see that the two syndrome vectors are respectively the column checksums along the diagonals of slope 1 and -1, and they should be all-zero vectors if there is no error in the array R. If there is one error in the array R, then the two syndromes are just the cyclic-shifted version of the error vector with respect to the position of the error column, thus its location can be determined simply by cyclic equivalence test. The following example shows how a single error column is reflected in two syndromes for an X-Code of size 5.

Example 3 *Syndrome Computation for a 5×5 X-Code*

Suppose the 3th column is an error column, then the two syndrome vectors (S_0 and S_1 respectively) and their corresponding error arrays are as follows:

					S_0
0	0	0	e_0	0	e_3
0	0	0	e_1	0	0
0	0	0	e_2	0	e_0
0	0	0	e_3	0	e_1
0	0	0	0	0	e_2

					S_1
0	0	0	e_0	0	e_2
0	0	0	e_1	0	e_4
0	0	0	e_2	0	0
0	0	0	e_4	0	e_0
0	0	0	0	0	e_1

So the two syndromes are actually just the original error column vector (cyclic-)shifted in two different directions for the same number of positions. When they are shifted back, then they only differ in at most one position, the number of the positions shifted gives the location of the error column. □

The above example almost gives the decoding algorithm for one error correction. To describe the algorithm more formally, some notations are introduced. For a vector V, denote V^T as its transpose. Let $V = (V[0], V[1], \cdots, V[n-1])^T$, denote $V^{(1)}$ (or $V^{(-1)}$) as the *down-* (or *up-*) shifted vector from V, i.e., $V^{(1)} = (V[n-1], V[0], \cdots, V[n-2])^T$, and $V^{(-1)} = (V[1], \cdots, V[n-1], V[0])^T$. Also denote $V[l..m]$ as the subvector containing the components $v[l]$ through $v[m]$, i.e., $V[l..m] = (V[l], V[l+1], \cdots, V[m])^T$, where $0 \leq l \leq m \leq n-1$. With these notations, a formal algorithm for correcting one error can be described as follows:

Algorithm 2 Correcting One Error

1. Compute two syndrome vectors S_0 and S_1 from the possibly-error-corrupted array R according to the equations Eq. (4) through Eq. (9);

2. $i = 0$;

3. **If** $S_0[0..n-3] == S_1[0..n-3]$ **And** $S_0[n-1] == S_1[n-1] == 0$ **Then**
  The error position is the ith column, and the error value is
  $E = (S_0[0], S_0[1], \cdots, S_0[n-3], S_0[n-2], S_1[n-1])$;
Else If $i == n$ **Then**
  Declare decoding failure : more than one error occurred;
Else
  $S_0 = S_0^{(1)}, S_1 = S_1^{(-1)}$;
  $i++$;
  Goto *3;*
End If

□

Before proving the correctness of the algorithm, we give a numerical example.

Example 4 Correcting One Error of a 5×5 X-Code

Suppose the possibly-error-corrupted array R is :

0	0	0	1	0
0	0	0	0	0
0	0	0	0	0
0	0	0	1	0
0	0	0	0	0

then U and V, the two constructed arrays from R, and their corresponding syndromes S_0 and S_1 are as follows:

		U			S_0
0	0	0	1	0	1
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	0	0	0	0

		V			S_1
0	0	0	1	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	0	0

Repeat Step 3 of the algorithm until $i = 3$*, then we get* $S_0 = (1, 0, 0, 1, 0)^T$ *and* $S_1 = (1, 0, 0, 0, 0)^T$*, so* $S_0[0..2]$ *equals to* $S_1[0..2]$*, and* $S_0[4] = S_1[4] = 0$*, thus we declare the error occurs at the 3rd column, and the error value is* $E = (1, 0, 0, 1, 0)$*, i.e., the uncorrupted array should be an all-zero array.* □

Now we give a proof of correctness of the algorithm.

Proof: If one error occurs at the ith column, and its value is $e = (e[0], e[1], \cdots, e[n-2], e[n-1])^T$, then the two syndromes (Eq.(4 through Eq.(8)) are:

$$S_0 = ((e[0], \cdots, e[n-3], e[n-2], 0)^T)^{(-i)} \quad (10)$$

$$S_1 = ((e[0], \cdots, e[n-3], e[n-1], 0)^T)^{(i)} \quad (11)$$

thus

$$S_0^{(i)} = (e[0], \cdots, e[n-3], e[n-2], 0)^T \quad (12)$$

$$S_1^{(-i)} = (e[0], \cdots, e[n-3], e[n-1], 0)^T \quad (13)$$

Since X-Code can correct one error, which means the location of a single column error can always be found unambiguously, such a unique i can be found that the two shifted syndrome vectors may only differ in the second last component and their last components are both 0s (Eq. (12) and Eq. (13)). Once the error location i is found, the error value is directly obtained from Eq. (12) and Eq. (13). □

The above algorithm needs $2n(n-2)$ *XOR* operations to compute the two syndrome vectors, and on average n cyclic equivalence test operations to get the error location.

4 Conclusions

We have presented X-Code, a new class of $n \times n$ *MDS* array codes of distance 3. The significant difference of these codes from all other known array codes is that the parity (redundant) symbols are placed in two independent rows rather than columns. Encoding and decoding of the codes may be accomplished using only *XOR* operations. We proved that n being a prime number is necessary and sufficient for X-Code to be *MDS*. X-Code achieves the lower bound of the update complexity for all prime numbers n, and it also has balanced computation at each column, which might be very helpful in storage systems and distributed computing systems. Finally decoding algorithms for correcting erasures and error are given.

One future research problem is to extend X-Code for all positive integers rather than only prime numbers. Another research direction is to extend X-Code to have distance $r+1$, where $r \geq 3$, while all the properties of X-Code still remain unchanged, namely the *MDS* property and the optimal update complexity property. Our primary research shows that in general X-Code can not be easily extended to have larger distance by simply using more parity rows and taking more slopes, except for few lengths n.

References

[1] M. Blaum, J. Brady, J. Bruck and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures", *IEEE Trans. on Computers*, 44(2), 192-202, Feb. 1995.

[2] M. Blaum, J. Bruck, A. Vardy, "MDS Array Codes with Independent Parity Symbols", *IEEE Trans. on Information Theory*, 42(2), 529-542, March 1996.

[3] M. Blaum, R. M. Roth, "New Array Codes for Multiple Phased Burst Correction", *IEEE Trans. on Information Theory*, 39(1), 66-77, Jan. 1993.

[4] M. Blaum, R. M. Roth, "On Lowest-Density MDS Codes", Preprint, 1997.

[5] M. Blaum, P. G. Farrell and H. C. A. van Tilborg, "Chapter on Array Codes", Preprint, 1996.

[6] P. G. Farrell, "A Survey of Array Error Control Codes", *ETT* , Vol.3, No.5, 441-454, 1992.

[7] R. M. Goodman, R. J. McEliece and M. Sayano, "Phased Burst Error Correcting Arrays Codes," *IEEE Trans. on Information Theory*, 39, 684-693,1993.

[8] G. V. Zaitsev, V. A. Zinov'ev, and N. V. Semakov, "Minimum-Check-Density Codes for Correcting Bytes of Errors, Erasures, Or Defects" *Problems of Information Transmission*, 19(3), 197-204, 1983