

10. BCH Polynomial Codes

10.1 Introduction

To expand our use of codes and provide a form of error-correction, it is helpful to extend our use of binary streams into another representation: polynomials. For our purposes, representing binary streams as polynomials requires the definition of a few operations, as well as examining some of their properties. In the next few sections, we will look at how these new tools enable more advanced and robust forms of encoding. Codes of the form outlined in this section are called BCH codes, after the three people who discovered them: Bose, Ray- Chaudhuri, and Hocquengham.

10.2 Binary Codes as Polynomials

The basic similarity between our previous codes and polynomials is that they are an ordered sequence of numbers strung together to represent a single expression. In the case of polynomials, the digits represents the coefficients of each term; in the case of our binary codes, the order instead represents the bit's position in the code. We say that a polynomial defined with only coefficients of 0 and 1 is defined over the field $GF(2)$, or the integers mod 2 (i.e. 0 and 1).

Building a mapping between the two is a simple procedure: we convert a_j to a term $a_j x^{j-1}$, where a is the value of the bit and j is its position in the code.

Example: To convert the sequence 101101 to a polynomial, we would go bit by bit and generate $1 + x^2 + x^3 + x^5$

Addition in a binary system has the following characteristics:

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$1 + 1 = 0$$

Note that no carrying to higher terms or higher order bits is performed.

Example: Adding $1 + x^2 + x^3 + x^5$ and $1 + x + x^3$ results in $x + x^2 + x^5$

Multiplication of Binary Polynomials

Of all the single term expressions, or monomials, x is the only one that is prime, as all others contain x as a factor.

Further, any term without a term of order 0, i.e. 1, can't be prime as it's divisible by x with no remainder. For example, $x^6 + x^2$ will be divisible by x .

Less intuitively, any polynomial $p(x)$ with an even number of terms has $(x + 1)$ as a factor. Why is this so? Well, if $p(x)$ has an even number of terms, then $p(1)$ will always equal zero. Further, when you divide $p(x)$ by $(x + 1)$ you get a remainder $r(x)$ that must have 0 degree, and so must be 0 or 1. (These last two statements are left as exercises to the reader.) We can rewrite the polynomial as $p(x) = q(x)(x + 1) + r(x)$ where $q(x)$ is the quotient that dividing into $p(x)$ gives $(x + 1)$. Evaluating this at $x=1$, we get that $0 = p(1) = q(1)*0 + r(1)$, or $r(1) = 0$. From this we can conclude that the remainder will be zero, and thus any polynomial with an even number of terms cannot be prime.

Any polynomial with all even order terms will be divisible by a polynomial just like the first but with the order of all the terms halved. That is, $1 + x^4$ is divisible by $1 + x^2$, for example. This is because in our binary system, $(x^a + x^b)^2 = x^{2a} + 2x^{a+b} + x^{2b} = x^{2a} + x^{2b}$. Similar expressions can be constructed for polynomials with more terms. Thus no polynomial with all even powered terms can be prime.

Another important property of primitive polynomials is their reversibility. We have chosen to associate the high order bit with the low order term, or translate the leftmost bit in the sequence into the 0th order term, and continue the process to the right. However, we could just as easily encoded the sequence in the other direction, and ended up with a completely different polynomial! For example, when turning the sequence 1011 into a polynomial, we can read from either direction, ending up with $1 + x + x^3$ and $1 + x^2 + x^3$. Each polynomial is the reverse of the other.

We are converting the binary sequence into polynomials for our benefit, to be able to perform certain mathematical operations on them and ultimately to perform error correction on a binary transmission. So which order of reading is the 'correct' order that will allow us to do this? Both are equally valid, with regards to our discussion of prime polynomials, as if a polynomial is prime, its reverse will also be. Later we will see how this enables error correction.

In our example, $1 + x + x^3$ and $1 + x^2 + x^3$ both meet the criteria of having odd order terms, as well as having an odd number of terms.

Let's build up a short list of prime polynomials:

- Of the two polynomials of order 1, both x and $x + 1$ are prime.
- Of the polynomials of order 2, only $1 + x + x^2$ is prime. As it is the only possible polynomial with an odd number of terms, it must be the only prime. Note that it is

symmetric, or that the bit sequence it represents is the same read forwards or backwards.

- Of the polynomials of order 3, only two are prime: $1 + x + x^3$ and $1 + x^2 + x^3$. Note that they are reverses of each other.
- Among polynomials of degree 4, the only polynomials that meet our condition are $1 + x + x^4$, $1 + x^3 + x^4$ (which is the reverse of the last), and the symmetric $1 + x + x^2 + x^3 + x^4$. If it isn't obvious how these three are arrived at, looking at the rules is a good start: we can rule out anything with a 1 term and anything with an even number of terms, which is a good start. We are left with our prime polynomials as well as $1 + x^2 + x^4$, which we know to be a composite because all its terms are odd.
- For the polynomials of order 5, we can use a similar process to arrive at $1 + x + x^5$, $1 + x^2 + x^5$, $1 + x^2 + x^3 + x^4 + x^5$, $1 + x + x^3 + x^4 + x^5$, and the reverses of all of these. However, be careful; even though a polynomial may pass the tests we've been using, it still could be composite. $1 + x^4 + x^5$, the reverse of $1 + x + x^5$, breaks down into $(1 + x + x^2)(1 + x + x^3)$. By this, we know that $1 + x + x^5$ must also be composite. In fact, it factors into the reverse of what $1 + x + x^5$ factors into: $(1 + x + x^2)(1 + x^2 + x^3)$.

10.4 What Makes a Good Code

The polynomials we need are not just prime, but also **primitive**. This means that every possible remainder is a remainder of a power of x . Consider the example $1 + x + x^3$, where the possible remainders of this are polynomials of form $a + bx + cx^2$, where a , b , and c are all either one or zero, and at least one of those variables is non-zero.

Thus, our seven possible remainders are: 1 , x , x^2 , $1+x$, $1+x^2$, $x+x^2$, and $1+x+x^2$; or we can write them in binary: **100**, **010**, **001**, **110**, **101**, **011**, and **111**. Or we can even give them a decimal representation, although this breaks the easily visible connection to the term coefficients: 1, 2, 4, 3, 5, 6, 7. Our polynomial will be primitive if all of these polynomials are remainders of a power of x . We can determine this by writing out the remainders of sequentially higher powers, and seeing if these remainder polynomials show up.

To begin, we know that x^0 (i.e. 1) has remainder 1. Further, if we have a rule that gives us x^{j+1} given x^j , we will be able to generate our list of remainders very easily on a spreadsheet.

We know that $\text{remainder}(x^j) = \text{rem}(x^j) = a + bx + cx^2 \dots sx^{k-2} + tx^{k-1}$, and that we can get to x^{j+1} by multiplying it by x , and we know that $x^j = p(x)q(x) + \text{rem}(x^j)$ for some polynomials $p(x)$ and $q(x)$.

We can combine these facts to write x^{j+1} in terms of the remainder of x^j : $x^{j+1} = p(x)[xq(x)] + x\text{rem}(x^j)$. Since we know that $p(x)*q(x)$ has no remainders, we can further simplify the equation: $\text{rem}(x^{j+1}) = \text{rem}(x * \text{rem}(x^j))$.

If the coefficient of x^{k-1} , t , is 0, multiplying $\text{rem}(x^j)$ by x increases each power by 1, resulting in $\text{rem}(x^{j+1}) = ax + bx^2 + cx^3 \dots + sx^{k-1}$.

If instead $t = 1$, multiplying the last term in $\text{rem}(x^j)$ (i.e. tx^{k-1}) by x results in x^k , whose remainder is $(p(x) - x^k)$. Therefore, the general form of $\text{rem}(x^{j+1})$ is

$$\text{rem}(x^{j+1}) = ax + bx^2 + cx^3 + \dots + sx^{k-1} + t(p(x) - x^k)$$

Now that we have this rule, let's apply it to our polynomial $p(x) = x^3 + x + 1$, and its remainder. If the remainder of x^k is $a + bx + cx^2$, then applying the rule we've developed will transform (a, b, c) to $(c, a + c, b)$ for each successive power.

The remainder of powers table for $p(x)$

Power	1	x	x ²	Binary Number
0	1	0	0	1
1	0	1	0	2
2	0	0	1	4
3	1	1	0	3
4	0	1	1	6
5	1	1	1	7
6	1	0	1	5
7	1	0	0	1

(As discussed earlier, x^0 has remainder 1, which is how we start the table off, and we progressively build from there.)

At the 7th power, the remainders cycle back together again and loop around again. If you look at the list of binary numbers, we find that all seven possible remainders are present, and thus we can conclude that polynomial $x^3 + x + 1$ is primitive. Something to note is that if you look at the 3 by 8 matrix formed by the coefficients of the terms, it is formed by a 3x3 identity matrix sitting atop of a Z-type matrix. This should be recognizable from the previous notes section.

Setting up a spreadsheet

So how do we use this to test whether a polynomial is primitive? We can set up a similar spreadsheet to calculate the remainder of powers, and locate the first non-zero power for which the binary number is 1. If and only if that power is $2^k - 1$ where k is the order of the polynomial (i.e. all possible remainders are covered), then the polynomial is primitive.

Once a spreadsheet is created this way, various polynomials can be tested by changing the line on which you enter its succession rule, and then drag down to fill up the

table with the new rules. Remember that in making up the spreadsheet to bracket every cell in $\mathbf{mod}(x,2)$, where x is the operation performed in that cell.

We can also test a polynomial without using a spreadsheet. The remainder of x^{2^j} is the square of the remainder of x^j . If you only write the remainders of the even powers from k to $2k-2$, you can compute all the remainders of powers of the form 2^j from these, and if you find that the remainder of the 2^k th power is not x , you know that the polynomial is not primitive. It can happen that the remainder of 2^k is x and the code is not primitive, if the remainder of some lower power is also x . You can check this by checking whether the remainders powers that are factors of 2^k-1 are 1. If not the polynomial is primitive.

In our example, we have $\mathbf{rem}(x^4) = x+x^2$. Squaring we get $\mathbf{rem}(x^8) = \mathbf{rem}(x^2 + x^4) = x$, and our code is a candidate for primitivity. Since $2^k - 1$ is a prime, this implies that the polynomial is primitive.

If you only write the remainders of the even powers from k to $2k-2$, you can compute all the remainders of powers of the form 2^j from these, and if you find that the remainder of the 2^k th power is not x , you know that the polynomial is not primitive. It can happen that the remainder of 2^k is x and the code is not primitive, if the remainder of some lower power is also x . You can check this by checking whether the remainders powers that are factors of 2^k-1 are 1. If not, the polynomial is primitive.

The Importance of Primitive Polynomials

Assuming we want to send message polynomial $\mathbf{m}(x)$, we encode it by multiplying it by the encoding polynomial $\mathbf{p}(x)$ and sending the product. We use primitive polynomials as the basis for error correction by dividing the received message, $\mathbf{r}(x)$, by the encoding polynomial, $\mathbf{p}(x)$, and examining its remainder. Since the encoded message, $\mathbf{m}(x)*\mathbf{p}(x)$, has no remainder, we know that any remainder results from an error in communication.

Further, if the error is only one bit, we can find the error location by finding the power of x that has the remainder $\mathbf{rem}(\mathbf{r}(x))$, which we can find using our remainder table, as long as no two rows of the remainder table are the same. Primitive polynomials cycle through every possible value before repetition of the remainder, and the remainder of the next power depends only on the remainder of the current power. That is, if the remainder of x^a is the same as x^b , that will also be true of x^{a+s} and x^{b+s} .

10.5 Decoding a Single Error Correcting Code Generated by a Primitive Polynomial

To decode we first find the remainder of our received message, $r(x)$ upon dividing by the primitive polynomial $\mathbf{p}(x)$.

We have outlined a way above to do this, by simply consulting the tables. But there is another way, namely that the remainder of a sum is the same as the sum of the remainders:

$$\text{rem}(a(x) + b(x)) = \text{rem}(a(x)) + \text{rem}(b(x)).$$

This means that using a table, we can easily sum up the remainders of the terms of a polynomial, instead of trying to take the remainder of the polynomial itself. This is very easily done on a spreadsheet by the following process:

Create a second table, whose rows correspond to the entries in the remainder table each multiplied by the bit of $r(x)$ that corresponds to that row. Summing those rows (mod 2), the binary number corresponding to that sum can be matched to the remainder table. Switching the bit that corresponds to the identified row will then recreate the sent message, $m(x) * p(x)$. Then, all that remains is to divide the message by $p(x)$ to recover the message.

We can create a spreadsheet that does this automatically by writing $m(x) * p(x)$ from right to left as a top-to-bottom column, and writing the code polynomial in the same manner to the column directly left. Beginning on the column directly to the right of the $m(x) * p(x)$ column, we proceed by repeating the same basic steps:

- 1) If the topmost bit of the previous column is 1, then the bit of $m(x)$ corresponding to the current column is one. In my table, I write it above the rest of my calculations. Then, we add the bits of $p(x)$ and the contents of the previous column and write the sum into the current column, while ignoring the topmost bit for both polynomials. In this way, each column is one bit shorter than the last.
- 2) If instead the topmost bit of the previous column is 0, then the bit of $m(x)$ corresponding to the current column is 0, and we take the previous column and copy it into the current column, while ignoring the topmost bit. As in the last case, each column is one bit shorter than the previous.

We continue this process until the previous column is of the same length as $p(x)$. When this happens, the process is completed, and what you write into the current column corresponds to the remainder of the operation, with the topmost bits in the column representing the coefficients of the highest order term.

Here is what you get for the example given (dividing $x^6 + x^3 + x^2 + x$ by $x^3 + x + 1$)

$p(x)$	$m(x)*p(x)$	1	0	1	0	message
1	1	0	1	0	0	last entry is x^2 term in remainder
0	0	1	0	0	0	last entry is x term in remainder
1	0	0	1	0	0	last entry is constant term of remainder
1	1	1	1	0		
	1	1	0			
	1	0				
	0					

The process is started by looking at the second cell in the second column, and seeing that it is a 1, writing 1 in the first cell in the third row. Then, the second cell in the third column is generated by adding the third cells in the first and second columns, the third cell generated by summing the fourth cells of the first and second columns, etc. Once the third column is completely written out, we look at the first bit of it (i.e. the second column), and seeing a 0, write a 0 in the first cell of the fourth column. Then, ignoring the second cell of the third column, we copy the third cell of that column to the second cell of the fourth column, and so on. We complete the rest of the columns in a similar way. Finally, when we get to the sixth column, we find that the fifth column has the same number of cells as $p(x)$ which indicates that our long division is over. We populate the final column in the same way, but once completed we are left with the quotient of $x^3 + x$, and a remainder of 0.

10.6 Comments

The above error correction method (i.e. finding the remainder of $r(x)$ by long division and finding which power has that remainder) may seem completely different from the method used for general matrix codes. That method was to multiply the remainder vector \mathbf{R} by the message killing \mathbf{D} matrix and seeing which row of \mathbf{D} the product matched with.

However, by computing the remainder of a polynomial by adding up the remainders of the terms in $r(x)$, we are in effect matrix multiplying the $r(x)$ row vector by the remainder table matrix, and matching the product with the rows of the remainder table matrix. Since multiplying the remainder table matrix by any code word will give a 0 vector (i.e. any non-zero product results purely from transmission error), our procedures are in a sense completely identical in function.

The correlation between the two methodologies is that the remainder table provides the \mathbf{D} matrix, or a ‘message killer’ as discussed earlier, for our code. The code generated by a polynomial $\mathbf{p}(x)$ can be considered akin to a matrix code; the first row contains the encoding polynomial written lowest power first, followed by a string of 0’s, with successive rows that have the polynomial shifted one position to the right. In the final row, the last bit of the polynomial is in the last column.

The remainders we encounter when dividing by a primitive polynomial can be added in the usual way. But if we have our remainder table, we can also multiply them. Each non-zero remainder is a remainder of a power of x , which can be determined from the remainder table. **We define the product of two remainders to be the remainder corresponding to the power that is the sum of their powers.**

When this sum exceeds 2^k-1 we can use the fact that x raised to that power is 1 to subtract 2^k-1 from that sum.)

For example, with our old remainder table (repeated below) we define the product of 110 and 111, which are the third and fifth powers of x , to be x^8 , which is x or 010. In this way, the remainder table cycles around to arbitrarily high powers.

Power	1	x	x^2	Binary Number
0	1	0	0	1
1	0	1	0	2
2	0	0	1	4
3	1	1	0	3
4	0	1	1	6
5	1	1	1	7
6	1	0	1	5
7	1	0	0	1

Addition and multiplication of the remainders of primitive polynomials follow the standard rules that we've defined earlier in this Note Section, i.e. with the 0 polynomial being equivalent to 0, and the same for polynomial 1. The product of two powers is another power, and never 0, and the sum of two remainders is another remainder. These remainders therefore form what mathematicians call **a field, which implies that they can be considered as number systems, like the real numbers or the rational numbers or the complex numbers or GF(2).**

This is not true for remainders upon dividing by a factorable polynomial, like $p(x)*q(x)$ with each factor of degree at least 1. The problem is that the remainder of the product of $p(x)$ by $q(x)$ is the 0 polynomial. This means that two non-zero remainders can have product 0. This is unacceptable for numbers.

A very important fact about real and complex numbers, in fact a part of the Fundamental Theorem of Algebra that we want any number system to keep intact, is that **a polynomial equation of degree k can have at most k solutions.**

This statement is true if the coefficients of the polynomial are elements of a field and generally false otherwise. This statement is important so we will prove it.

Suppose we have a polynomial equation of degree k with coefficients in a field. If k is 1, the equation takes the form $ax - b = 0$ for non-zero a . If c is a solution we must have $ac = b$ which implies $c = b/a$, which makes c a unique element of the field.

We suppose now that the statement (that an equation of given degree d has at most d solutions) is true for all polynomials of degree $k-1$, and prove that it is true for any $p(x)$ of degree k .

Suppose c is a solution to the equation $p(x)=0$. Then $(x - c)$ must divide p without a remainder. Otherwise we would have $p(x) = (x-c)*q(x) + r$ and $p(c) = r$ and not $p(c) = 0$.

Therefore, we can write $\mathbf{p}(\mathbf{x})$ as $(\mathbf{x}-\mathbf{c})\mathbf{q}(\mathbf{x})$, where $\mathbf{q}(\mathbf{x})$ has degree $k-1$ and q has at most $k-1$ solutions by our induction hypothesis.

Any solution to $\mathbf{p}(\mathbf{x}) = \mathbf{0}$ therefore obeys $(\mathbf{x}-\mathbf{c})\mathbf{q}(\mathbf{x})=\mathbf{0}$.

$(\mathbf{x} - \mathbf{c})$ has only one solution and $\mathbf{q}(\mathbf{x})$, being of degree $k-1$, has at most $k-1$ of them by the induction hypothesis.

Since no two non-zero field elements have product 0, the only solutions occur where one or another of the factors of p is 0.

So $\mathbf{p}(\mathbf{x}) = \mathbf{0}$ can have at most k solutions, as we set out to prove.

Correcting Multiple Error Codes

Now we ask: **what can we possibly do to correct more errors in a polynomial code?** Using a primitive polynomial, $\mathbf{p}(\mathbf{x})$, as the encoding polynomial works splendidly to correct one error, and we certainly want to maintain this power. To correct two errors we will use a polynomial that is the product of a primitive polynomial and a second polynomial which we choose to provide the additional information that we need to locate two errors.

How is the second polynomial chosen? We will choose a polynomial $\mathbf{p}_3(\mathbf{x})$ which obeys the condition: $\mathbf{rem}(\mathbf{p}_3(\mathbf{x}^3)) = \mathbf{0}$, where we take the remainder on dividing by our primitive polynomial p as before.

Additionally, we can correct 3 errors by having a factor, $\mathbf{p}_5(\mathbf{x})$ in the encoding polynomial that obeys: $\mathbf{rem}(\mathbf{p}_5(\mathbf{x}^5)) = \mathbf{0}$.

This process can be extended to handle any number of errors. In the next chapter we address in greater detail the two questions that arise:

- 1) How do we find polynomials $\mathbf{p}_3(\mathbf{x})$ and $\mathbf{p}_5(\mathbf{x})$ and $\mathbf{p}_7(\mathbf{x})$ and so on?
- 2) How can we locate and correct two or more errors if our encoding polynomial has such factors in it?

Exercises

Exercise 1 Find all primitive polynomials of degree 6 (over the two element field $\text{GF}(2)$ defined by $2=0$.)

Exercise 2 Pick a primitive polynomial of degree 5. Construct a spreadsheet encoder for it, that takes any binary message of length 26 and converts it into a coded message using that polynomial as encoding polynomial.

Exercise 3

Construct a spreadsheet single error corrector for it, that starting with any received message of length 31 takes its matrix product with the remainder table, locates the error and corrects it.

Exercise 4

Construct a spreadsheet divider that takes the corrected code word and finds the message that was encoded to it.

- Scott Ostler