

13. Properties and Generalizations of our BCH Codes

So far we have examined

How to find a primitive polynomial p with coefficients 0 and 1.

How to find polynomials p_j which obey: $\text{rem } p_j(x^j)=0$ on dividing by p ;

How to multiply the message m by an encoding polynomial

How to create remainder tables and "up by j " remainder tables, which allow us to find the sum of the j -th powers of the error monomials with a code which has p_j as a factor, by matrix multiplying the received message with the appropriate table.

How to relate such power sums to the coefficients of the error locator polynomial when there are two or fewer errors.

How to test each power to see if obeys the equation $\text{rem } e_l c = 0$, and change the bit for each power that obeys this equation.

How to test whether the correction is successful.

How to divide the corrected message by the encoding polynomial to retrieve the original message.

We will now look the following questions:

1. **How can we prove that all this works?**
2. **Can we extend the relations between power sums of error monomials (we call them t 's) and coefficients of the error locator polynomial (which we call s 's) to correct more errors?**
3. **How hard is it to solve these equations for the s 's?**
4. **What complications if any arise when correcting more than 2 errors?**
5. **How do we attack the problem presented by multiplying or dividing by the 0 remainder?**
6. **How many message bits can be carried by the 2^k-1 total bits in one of these codes, if it is to correct E errors?**
7. **How are BCH codes used?**
8. **How does their efficiency compare to the Shannon Bound?**
9. **What are BCH codes over other fields?**
10. **What are recent developments about such codes?**

1. Why all this works

We can give the following argument.

When there actually are z errors, the only coefficients s_j that are non-zero, are the first z of them, and the error locator equation has degree z .

The actual error monomials will be solutions of this error locator equation, which will account for all z of its solutions.

We deduce from the fundamental theorem of algebra that there can be only z solutions to the error locator equation when it has degree z .

This means every non-error must fail to obey it, and we therefore find the errors and only the errors by testing each power to see if it obeys it.

2. Relations between power sums t_j and coefficients s_j of the error locator equation

We derived the relation we needed between power sums and coefficients of the error locator polynomial by multiplying the equation

$$\text{rem } y^2 + s_1 y + s_2 = 0$$

by y , applying it at each error and summing over all the errors.

When there were two errors this gave us the equation

$$\text{rem } t_3 + s_1 t_2 + s_2 t_1 = 0.$$

Obviously the same approach can be applied multiplying by any power k of y and in particular any non-negative integral power of y .

We get, when there are j errors,

$$\text{rem } t_{j+k} + s_1 t_{j+k-1} + s_2 t_{j+k-2} + \dots + s_j t_k = 0,$$

except when $k=0$ in which case the last term becomes js_j . (we get the constant term in the equation repeated for each of the j errors.)

Actually, the equation obtained when $k=0$ holds as well for all r up to the number of errors j . We have

$$\text{rem } t_r + s_1 t_{r-1} + \dots + s_{r-1} t_1 + r s_r = 0.$$

We can deduce these latter equations by noticing that each t consists of a sum of single monomials and each s is a sum of terms that are each linear in each variable in it.

Thus a typical term in t_r is x_a^r , and a typical term in s_u is $x_1 x_2 \dots x_u$.

This implies there are only two kinds of terms in the product of an s with a t . **Those that are linear in**

all variables in them, and those that are linear in all but one

Those linear in all but one, like for example $x_1^3 x_2 x_4$ will occur in two consecutive products of s and t and therefore appear twice in our sum.

The example will occur in $t_3 s_2$ and also in $t_2 s_3$, with either x_1^3 in t_3 and the rest in s_2 or x_1^2 in t_2 and the rest in s_2 .

Since for us $2=0$ holds, the coefficient of this term, and all like it, will be 0 in the given sum,

On the other hand, terms linear in all variables in them occur in s_r and also r different times in t_{1s_r-1} , since each variable by itself will appear in t_1 .

So these terms will appear a total of $2r$ times or again no times when $2=0$.

(If 2 is not 0 then we get the same result if we alternate signs between adjacent terms.)

3. How hard is it to solve these equations for the s 's?

The solutions are fairly straightforward. We illustrate the procedure when we want to correct up to 3 errors.

s_1 is always t_1 by definition.

When we want to handle 3 errors we need find s_2 and s_3 . We can do so by using the s to t equations of degree 3 and 5. These are, when there are at most 3 errors:

$$\text{rem } s_3 + s_2 t_1 + s_1 t_2 + t_3 = 0$$

and

$$\text{rem } s_3 t_2 + s_2 t_3 + s_1 t_4 + t_5 = 0.$$

If we multiply the first by t_2 and add the two equations we can solve for s_2 :

$$\text{rem } s_2 (t_3 + t_1^3) = t_2 t_3 + t_5,$$

and substituting this expression for s_2 in the first equation yields

$$\text{rem } s_3 (t_3 + t_1^3) = t_1 (t_2 t_3 + t_5) + (t_3 + t_1^3)^2 = t_1^3 t_3 + t_1 t_5 + t_1^6 + t_3^2.$$

These expressions allow us to determine all coefficients in the error locator polynomial **multiplied by $(t_3 + t_1^3)$** by using addition and multiplication only.

Similar looking relations hold for more errors; they are only uglier.

4. What complications arise when trying to correct more errors?

Nothing really gets more complicated when we want to correct more errors, except that **there is a flaw in our method of multiplication which we would have to correct.**

We multiply two remainders, which are characterized by their identifiers, by converting them to powers, adding the powers, then converting the sum power back to an identifier or remainder.

The problem is that **there are 2^k possible remainders and only 2^k-1 powers. The 0 remainder is not a power.**

When we multiply anything by the 0 power, we get 0. But the procedure we used of converting would add 0 to the power of the other factor; in other words it multiplies the other factor by 1 and not by 0.

When we had two errors we never even needed to multiply anything. In general we multiply.

We want to avoid dividing as much as possible, since there is always the possibility of dividing by 0. This can be handled by using an if statement to handle the id(denominator) = 0 some other way but that makes life too complicated.

Fortunately there is an easy way to handle these problems, which are the only special cases we need worry about in correcting more errors than 2.

5. How do we attack the problem presented by multiplying or dividing by the 0 remainder?

When we multiply any remainder by the 0 remainder we want to get the remainder 0.

We can arrange this by assigning a non-integer power to the 0 remainder, like say .3/10.

To do this we need add a new row to our remainder table containing the 0 remainder, the identifier 0 and the power .3. This row does not correspond to any actual power and so is not used in finding the syndrome (namely the t's) of a received message

When we compute the power of the product of 0 with A, we will get a non-integer power, which will either convert to the identifier 0 or fail to match any row, and therefore produce the identifier 0 anyway.

Doing this extends multiplication to 0 remainders.

There can be a problem if you try to divide by 0, since that is never a good idea.

Obviously you would never consciously divide by 0, but here we are trying to develop a general spreadsheet that works no matter what errors are made. In that case we have to worry if we divide by anything, whether the errors could conspire to make the thing we divide by into 0, in which case our computation will go bananas.

In the case of three error correction, it follows from our results above that **we can write down an error locator polynomial multiplied by $(t_1^3+t_3)$ without doing any division at all.** This will lead us to whatever errors there are **except in the horrible case that $\text{rem}(t_1^3+t_3) = 0$ is true.**

In that case our multiplied error locator polynomial will be multiplied by 0 and will consist only of 0's and have no content.

However, we can deduce from our equations that when this happens we must also have $\text{rem } t_1(t_1^5+t_5) =$

0.

By manipulating we can show that it is impossible for both $\text{rem } t_1$ and $\text{rem } t_3$ to be 0 if there are three or fewer errors unless there are no errors at all. On the other hand, if $\text{rem } (t_1^5 + t_5) = 0$ holds, **there can be at most one error and that error will be t_1 .**

Thus, we can handle all possible ways of having three or fewer errors by a procedure that is very much like that used for a two error correcting code. If the error locator polynomial multiplied by $(t_1^3 + t_3)$ has first row all 0, the error, if any is t_1 ; and hence in the power with remainder t_1 (and when that is fractional there is no error). Otherwise the error locator polynomial produces the errors directly just as in the two error case. The only difference is that there is a cubic term as well as the three terms present in the two error case.

6. How many message bits can be carried by the $2^k - 1$ total bits in one of these codes, if it is to correct E errors?

Suppose for example, we use a code with a primitive polynomial of degree 8.

Then our total message length will be 255, and our primitive polynomial has degree 8 so that 8 of those bits will be used for error correction leaving 247 message bits if we want to be able to correct one error.

Each additional error we want to correct can (but sometimes does not as we shall see) require that our encoding factor have another factor (a new p_j) that can have degree as many as 8.

Thus, to correct E errors, we might use up $8E$ bits for error correction, leaving $255 - 8E$ message bits. Thus, to correct 6 errors could require 48 error correction bits leaving 207 message bits. This is an upper bound on what will be needed and therefore a lower bound on the number of possible message bits.

In general, if we want to correct E errors and our primitive polynomial has degree k , the corresponding bounds are

kE error correction bits at most

$2^k - 1 - kE$ message bits at least.

Why are these bounds and not exact statements?

There are two interesting phenomena that happen, usually when you have a number of error correction bits that is on the order of a third of the total or more.

First we can encounter a p_j factor that will correct an additional error that has degree lower than k .

Second, we can find that we get a factor p_j that is a duplicate of a previous one, so we do not need to add it. In other words the code that corrects one fewer error automatically corrects one more error without any new factor.

How can we detect all this?

There is a wonderfully easy way. We can construct a table which will tell us which powers obey the same equation.

Notice that the monomial x , when raised to the power $2^k - 1$ is 1, Thus it obeys the equation

$$\text{Rem } x^Q - 1 = 0 \text{ for } Q = 2^k - 1.$$

All powers of x must also obey this equation since x raised to the power Qj will also be 1.

Each power of x will obey some sort of polynomial equation of degree k

We now ask, which powers obey which equations?

We know quite a bit about this question, Because we have $2=0$, the j -th and $2j$ -th powers of x will obey the same equation.: squaring both sides of the equation for j will produce the same equation for $2j$.

This fact allows us to produce a table whose entries give powers that all obey the same equation.

Suppose j is odd; then the powers $j, 2j, 4j, 8j, \dots$ must all obey the same equation.

On a spreadsheet we can start each row with an odd power and keep doubling powers mod $(2^k - 1)$.

After at most k terms, you will get back to the original odd power. We know this because we know our equation will have degree k at most and can have at most k roots.

Sometimes, however you will get back to it sooner. And sometimes your odd power appears on the power list for a previous odd power.

Let us try this for $k=6$. Then we have $2^k - 1 = 63$

Here is a spreadsheet table as just described.

degree six power table
odd power

1	2	4	8	16	32	1
3	6	12	24	48	33	3
5	10	20	40	17	34	5
7	14	28	56	49	35	7
9	18	36	9	18	36	9
11	22	44	25	50	37	11
13	26	52	41	19	38	13
15	30	60	57	51	39	15
17	34	5	10	20	40	17
19	38	13	26	52	41	19
21	42	21	42	21	42	21

The only entry after the first column is $=\text{mod}(\text{left} * 2, 63)$.

You will notice that the 9 and 21 rows have only 3 and 2 entries. Also 17 and 19 occur in the 5 and 13 rows.

This actually means (as you could prove if you really tried) that the polynomial p_9 has degree 3 here and not degree 6. (for p_{21} has degree 2) And also, you need not multiply by a new polynomial to get p_{17} or p_{19} , since these polynomials are p_5 and p_{13} .

This means that a 5 error correcting code polynomial will have degree 27 rather than 30, and if you construct an 8 error correcting code here, it will actually be a 10 error correcting code. Of course this requires $8*6-3$ or 45 error correction bits leaving only 18 for the message. Two more correction bits will give you an 11 error correcting code,

There are some nice observations we can make.

1. some rows are equal to others read backwards. These powers obey equations that are reverses of one another (reading from right to left instead of left to right). Thus in the table above the 1st and 62nd powers obey the opposite equations, as do the 11th and 13th, the 3rd and 15th and so on,

To see this take the equation satisfied by any power x^s , and divide it by s^k . You will find yourself with the "opposite" equation which will be satisfied by x^{-s} , which is x^{N-s} .

For example, suppose x obeys $rem\ x^5 + x^2 + 1 = 0$. Upon dividing by x^5 we get

$$rem\ x^{-5} + x^{-3} + 1 = 0,$$

which is the opposite equation for x^{-1} .

2. some are symmetric; like the power 7 row, this will obey a symmetric polynomial equation.

3. the primitive polynomial is definitely not symmetric

7. How are BCH codes used?

These codes are often used to correct errors. One important use is to reduce the probability of an error from something fairly small to something small enough to be ignored.

Thus, for example if you are sending information to your bank about a payment, you want to be very sure that the number sent comes out right.

So we might want to take an error rate on the order of 1 error per ten thousand bits, and try to reduce the probability of an erroneous message (not bit) being sent down to something like one in 10^{15} ,

There is another possibility of course, and that is that there are more errors than you correct for, but the received word is too far from any code word to be successfully corrected.

Thus, if you make 3 errors and can only correct 2, you might find that the received word is a distance at least 3 from all code words, and you cannot handle it,

This is actually a much better outcome than an incorrect message. You can simply ask the sender to resend.

You don't want this to happen all the time, but you can tolerate such things say once per 10^5 messages, unless you are a perfectionist, in which case you would probably still be happy having to resend once per 10^{10} messages.

Shannon's theorem refers to being able to get the message right most of the time; which is a much weaker criterion than we want;

If we use a 255 bit code, and have a probability of 10^{-5} of an error per bit, we will get an average number of errors per sent message of one in 400. A probability of error of one in 10^4 would give an expected number of 1/40 for the proportion of messages with an error, if there were no correction.

It often happens in the real world that errors come in bursts. That is, when there is an error, there is a much greater chance that the neighboring bits are unreliable than when there is no error. This is because some sources of error garble more than one bit at a time. Of course the fact that a bit is garbled does not imply that it will be wrong, but it can make it quite unreliable.

BCH codes can be used to handle such errors in two ways. First, you can interlace them. That is, run several of them in parallel next to one another, so that adjacent bits come from different codes and a burst of errors will still produce only one error per code.

An even better way to deal with this possibility is to use a BCH code whose coefficients are polynomials (like our error locator polynomial) Then it can take a number of bits to describe each coefficient of a power (imagine that the coefficients were digits from 0 to 9 instead of just 1 and 0; then expressing each one of them requires several bits). If you interlaced two such codes, a short enough burst of errors would mess up only one coefficient in each code and so a single error correction (correcting a single coefficient rather than a single bit) would handle it.

BCH codes with coefficients that are themselves polynomials are called Reed-Solomon codes, and they are often used to handle burst errors.

8. How does the efficiency of these codes compare to the Shannon Bound?

A single error correcting code of the kind we are considering is "perfect". Every potential received message is within a distance 1 of some code word.

When we correct more errors we lose this property. Typically, correcting E errors requires E or almost that number of correction bits. On the other hand, the number of code words within a distance E of the message is proportional to a binomial coefficient.

Thus, with E errors the number of possible words, 2^N per message word (of which there are 2^M) goes something like 2^{kE} . While the number of words with E or fewer errors behaves more like $C(N, E)$. Since N is roughly 2^k , there is a factor on the order of $E!$ different between these.

Thus, the BCH k error correcting code gets worse in the sense of requiring more bits than the Shannon

bound, by roughly $\log_2(E!)$.

On the other hand, this is not so terrible, in that it means that the probability that a word that has too many errors to be corrected by our code will be within E of any code word, which is something we can see, will be on the order of $1/E!$.

This information allows us to compute how many errors we need to correct to reduce the probability of not-decoding and false decoding to within any desired bounds, given an error probability p .

For p smaller than $1/N$, the probability of making more than E errors per message of length N will be roughly the probability of making $E+1$ errors, which will be

$$C(N, E+1) p^{E+1} (1-p)^{N-E+1}.$$

The probability of a false correction will then be on the order of $1/E!$ times this probability.

Most of the time, in other words, when there are too many errors with a BCH code that can correct many errors, on trying to correct it you will find that it cannot be corrected, and will request resending.

9. BCH Codes with coefficients in Other Fields.

The fields whose elements are remainders on dividing by a primitive polynomial can be used as the coefficients of primitive polynomials, and these can be used to define BCH codes as we have done. Thus you can find primitive polynomials with polynomial coefficients and find polynomials that remainders of odd powers of the variable obey on dividing by them, as we have done here.

These, as noted already, are called Reed Solomon Codes and are particularly useful when there are burst errors.

Such codes are the subject of current research here at MIT by Professor Madhusudan, and others.

They consider the notion of list decoding.

Suppose in a BCH code designed to correct k errors, more errors are actually made. Then we cannot expect to find the original message.

But, we can look for the messages that are nearest to the received message, and make a list of them. With luck, the sent message will be one of the top candidates.

Often it is possible to determine which of a list of possible messages was sent from its context. Thus there is the possibility that you can determine which of the list of nearby messages was the sent one from looking at the list. With luck, in other words, you may be able use outside information to determine which of the close to received messages is the right one.

This means it is interesting to try to find close code words when there are more errors than the code was designed to correct. This only becomes impossible when there are a huge number of code words at the same Hamming distance from the received word.

Methods have been developed recently for doing this.

Exercises:

1. Make degree 5 and degree 8 power tables like the one above for degree 6. Find the first place in each in which you get extra error correction without a new factor, or a factor of smaller degree than the primitive polynomial.
2. How many errors should a code correct to make the probability of a false message at most 10^{-15} and the probability of resending at most 10^{10} given per bit error rates of 10^{-4} and 10^{-5} .