MITOCW | 4. Linear block codes, parity relations

GEORGE VERGHESE: So we're going to continue talking about coding. We're going to focus on linear block codes, which I introduced briefly last time. But just to step back a bit and remind you, we're talking about this piece of the overall channel.

So we've got the source that's done this source coding, compressing all the bits coming out of here. So that one bit, one binary digit, carries a bit of information. And now, we're actually reintroducing redundancy in a controlled way, so that we can protect the message across the physical channel with its noise sources and distortions and so on.

Actually, I should be saying binary digits at this point. Because again, at this point, the binary digit doesn't carry a bit of information. We're introducing redundancy, but I'll leave you now to make the distinctions.

OK, at this point here outside the source coding, one binary digit is one bit of information. But now, when you start to introduce the redundancy, you've got binary digits that are not necessarily one bit of information per binary digit. In fact, it won't be.

And then across the channel at the other end, you do the decoding to try and recover from any errors that the channel might have encountered. And what we said last time is that the key to this is really to introduce some space around the code words that carry your messages. So you might want to expand your set of messages into a longer code word, such that a small number of errors on each code word will not flip you over into another code word.

So you'll be able to recognize the neighborhood of the valid code words. That's the basic idea. So you're trying to put some space around things.

So if you've got k bits in your original message, you've got 2 to the k messages, right? So k message bits, 2 to the k messages-- and what we're planning to do now is, with this input stream that's coming into our channel coder, we're going to take the stream and break it up into blocks.

So each block will have k message bits. And then out come a series of blocks, but each block now has the large number. So we've got n bits.

OK. So we've done some padding here. n is greater than k. And so you have the possibility of 2 to the n possible messages in those n bits, but you're not going to use all of them. You're only going to use 2 to the k, and so you'll leave some space around each valid code word, all right?

So the code words are selected from 2 to the k code words selected from 2 to the n possibilities. OK. You get the idea there? Yeah. And we introduce this notion of Hamming distance then to measure the size of the neighborhood around the code word.

So we have the notion of a Hamming distance, which we'll abbreviate to HD. And this is the Hamming distance between two bit streams or between two, let's say, blocks. And what this is is the number of positions in which they differ.

OK. So it's a very simple notion of distance between bit strings or binary digit strings. All right. And what we then said is you get certain desirable error detection and error correction properties based on the minimum distance. minimum Hamming distance of a code, we use the simple little d for that. That's the minimum distance you find between any two code words in the code.

So based on that, we said that-- we wrote it slightly differently last time. I'm writing it to give you yet another way to think about it. What we basically said is, for instance, if you had a valid code word here, valid code word here, this is just a schematic. One hop, meaning one bit change, brings you to some other word which is not a code word.

Then another bit change brings you to some other word, not a code word. And a further one brings you to a new code word. That's Hamming distance three.

So if the minimum distance you find among all the spacings between code words is a distance of three, measured as the Hamming distance, then you can detect up to two errors. So if you went from this code word in two hops, you'd still not be at a new code word. So you know you've made a mistake.

If you wanted to correct errors, you could correct up to one error in this case assuming that you have no more than one error. If you ended up here, you'd know it had to have come from this code word. If you ended up here, you know it had to have come from the code word on the right.

Now, you have to be a little careful if you're trying to do correction and detection at the same time. So for instance, if you end up over here and if it's possible to get up to two errors, then you might think you've had one error that brought you here. And you might correct to this point.

But if the way you actually got there was two hops from over here, then you've done an incorrect correction. OK. So you've got to be a little bit careful. And that's what the third case tries to deal with. It allows you to deal with combinations of correcting up to a certain number of errors and then detecting a certain number.

So basically, what it's saying is that this entire distance here has to end up with a little gap. You've got to be able to make a number of hops equal to the number of areas you want to detect and still leave enough space to get to a code word unambiguously. So in this particular case, for instance, you couldn't unambiguously detect up to two errors if you were doing error correction for one.

But if I had this picture, OK, this is now Hamming distance 4, 1 2, 3, 4. I could correct single bit errors, and I could detect up to 2 because 2 errors would bring me up to this point. That's clearly an error that I wouldn't try to correct, but I'd recognize it as an error.

OK, so that's what the third case tries to account for. You won't believe how much time I spent trying to distill that statement down into a bullet. And I don't know if I got it right here, but that's the idea.

OK. So our focus today is on linear block codes. We're not talking about codes in general, but linear codes. This would be a general statement for a block code. I haven't said anything about linearity up to this point. All I said was take blocks of k bits, expand them to blocks of n bits, and pick subsets in this fashion.

That's just a general statement about coding. There's nothing linear about this as stated. So if you want to impose linearity, then you've got to introduce this additional piece, which is to say that every bit in your code word is going to be a linear combination of bits from your message.

And the easiest way to understand that is the matrix representation I had last time. Do I have it on this slide? Not yet, OK. But I probably do on the next, so let me pull that up.

OK. So basically, you're going to generate your code words, c. So that's c1 up to cn is going to be d1 up to dk times some matrix, which is k times n matrix. And we'll call it G.

OK. So that's referred to as the generator matrix for the code. We're talking about binary code. So all of these are 0s or 1s. And all of these entries are 0 or 1.

And all computations are done in GF(2). They're done modulo 2. Well, let me just say that all operations are in GF(2). OK. So this is modulo 2 operations or Boolean operations.

So if I'm working with the symbols 0 and 1, what is 0, minus 1? How am I to interpret this? I haven't quite defined it. But how would you interpret that?

You can think of it as the thing I need on the right-hand side that, when I add 1 to both sides, I get 0. Is that one way to think of it? So 0 minus 1 is 1. Or another way to say that is minus 1 is the same as plus 1 in this setting.

OK. So you just have to get used to working with only 0 and 1 in GF(2), but we talked about that last time. All right, so back to the statement. This is for a linear block code.

You're going to see matrix multiplication throughout your careers here. So if you haven't already seen them, this is a good opportunity to learn about matrix multiplications. So let's see, could somebody tell me what procedure I go through to, let's say, get the i-th position here in terms of what I do on the right-hand side if I want to get the i-th position on the left-hand side? What's the operation that I'm thinking of?

Or let me ask you this. Is the entire matrix G relevant when I'm just interested in the i-th position here? Or is this some part of G that's what I should focus on? Yeah.

AUDIENCE: The i-th column?

GEORGE VERGHESE: It's just the i-th column, right? So we think of matrix multiplication as sort of being in the simple case. If you want the i-th position here, it's kind of the dot product of this row with the i-th column.

So if you want the i-th position here, let me give you a particular example. If I've got 1, 0, 1, 1 here and I have 1, 1, 0, 0 here, then this position is going to be this is in the i-th column. What I'm going to find in the i-th column is 1 times 1, which is 1, plus 0 times 1, which is 0, plus 1 times 0 plus 1 times 0. So I just got a 1, right?

So I'm just going to get a 1 or a 0 depending on the specific entries here. So look what we've done. We've found a particular position in the code word as a linear combination of the bits in the message.

We took the combination of these bits with the weights that are displayed out here. So that's really what this statement was on the previous slide. We said that a code is linear.

Well, each of the k message bits is encoded as a linear transformation. Sorry, each of the code bits is encoded as a linear transformation of the message bits. I didn't quite say it that way there. Let me say it here on this slide.

So each code word bit is a specified linear combination of the message bits. This is what I'm referring to. If you wanted to find any particular bit, you're going to take a linear combination of these bits with the weights that are here, OK?

There's another way to think of this also which is the other blue line out there, which is to think of the matrix G as being made up of rows. OK. So can someone describe to me what we're doing with these rows to get this into our code vector? Yeah.

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: OK. So basically, the way matrix multiplication works, if you think about it, is what we're going to be doing is 1 times the first row plus 0 times the second row plus 1 times the third row plus 1 times the fourth row. So another way to think of matrix multiplication, it's going to generate this vector as a linear combination of the rows of this matrix, OK? What linear combination-- well, the linear combination that's described in the message part of this. So this is the message part.

So that's the other statement out here. So each code word is a linear combination of the rows of this generator matrix. So these are concrete ways to think about what a linear code is. But we also saw that there's another way to think of it which is in terms of this property, that the sum of any two code words is also a code word.

So if you have a set of code words with the property that the sum of any two is another word and that set, another code word and that set, then what you have is a linear code. And we argue that the all 0s code word must be in there. Because when you add a code word to itself, you get the all 0s code word, right?

OK. So that's the class of codes we're going to be focusing on. But I'm going to make a further restriction, which is that I'm going to look at code words that are of a very special type. So I'm going to limit myself to code words that have this structure.

So here's my data bits, d1 up to dk. I'm going to pick my code word, so that, let's say, the first k bits are precisely the data bits. And then I'll pick the additional ones to be some set of what we'll call parity bits. So this is p1 up to pn minus k.

All right, so I'm not going to have an arbitrary transformation here. I'm going to restrict myself to transformations that have the property that, when I multiply by the data vector here, what I get is the data vector reproduced in the initial part and then a bunch of new bits representing the redundant relationships that I'm computing. We refer to them as parity bits.

It's not so important that the data bits be in the first k position, so I'm willing to tolerate variations of this where the data bits are somewhere in this code vector. But the key thing about what's called a systematic code is that, when I look at the code word in designated positions, I find the data bits and the other positions are the so-called parity bits that are obtained as linear combinations of the data bits, OK?

So if you are familiar with matrix operations, then that what I'll need is all the way down the diagonal to have a matrix that has 1s along the diagonal, 0s everywhere else, and then something here. Let me just call this matrix of left over 0s and 1s, matrix A. OK. So I've got here a k times k matrix with 1s down the diagonal and 0s everywhere else.

And then I've got a matrix which has 0s and 1s in it. This is going to be, what is it, k times n minus k, right? So do you buy this?

So think about how matrix multiplication works. If I want the first column on the left, I take this row inner product or dot product with the first column here. That just selects out d1. And indeed, I get the d1 there.

And that happens for the first k positions. Beyond that, I'm taking linear combinations with whatever sits here. OK. It turns out that this is not really a special case.

It turns out that any linear code can be transformed to this form by invertible operation. So basically, if you use invertible operations on the rows here and some rearrangement of the columns, you can bring any code to this form. And then the resulting code will have effectively the same error correction properties that the code out here did.

OK. So we're just going to limit ourselves to thinking of linear codes, which are in the so-called systematic form. In other words, some part of the code word is the message bits and the other part is parity bits, OK?

So let's look at a specific code that is of this form, very simple code, referred to as a rectangular code. And you see a particular example on this slide here. So what do we do?

We arrange our data bits into a matrix which could be rectangular or square depending on what you have. So we're going to have r, rows, and c, columns, with the data bits in here, so D1, D2, all the way up to D sub, let's say, r times c, right? In this particular case, r and c are both 2.

And then you're going to generate the parity bits in the simple fashion. What you're going to do is choose a parity bit associated with the first row that basically makes sure that in the first row, including the parity bit, you've got an even number of 1s. OK. So this is a choice for even parity here.

Similarly, P2 will be chosen such that the second row has even parity. In other words, you've got an even number of 1s there. And for the columns, similarly, P3 will be chosen such that D1 and D3 and P3 together have even parity. In other words, the number of 1s in that column is even. Again-- the same thing for this column.

So what you're trying to do is sort of have sentries on the rows and columns that will signal when something has happened to a bit of the intersection. That's the general idea here, all right? So you'll take this out and arrange it then.

So you've got your parity bits. What's the sequence I used-- P1, P2 and then more parity bits here, OK, so row and column parity bits.

So here's a way to think about what these are explicitly. So what you'll do is P1 is D1 plus D2. When I say plus, of course, I mean in GF(2). So that's modulo 2 addition, right?

Does that simple formula ensure that I've got an even number of 1s in that first row, right? If D1 is 0 and D2 is 1, then I'll make this equal to 1, which is what I need. If D1 and D2 are both 1, I'll make this 0, which is what I need and so on. So this simple expression captures it and similarly for the r-th row.

So for each row, you make the parity bit equal to the sum of the data bits in that row, similarly for the columns, OK? Another thing, by the way, can you tell me what P1 plus D1 plus D2 is going to be in this case? If I pick P1 in this fashion, what does it guarantee for P1 plus P2 plus D2?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Sorry?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: 0, right? Because really I'm taking P1 and adding P1 again. And when I take something and add it to itself in GF(2), I get 0. So this is equal to 0.

So these are just two different ways of thinking about the parity bit here. So this is how you compute the parity bit, whereas this might be referred to as parity relation. It's a linear constraint relating the parity bit and the data bit.

In fact, we might try constructing this matrix as we go, right? So we've got D1, D2, D3, D4, P1, P2, P3, P4. Whoops. And here is D1, D2, D3, D4.

I'm going to have my generator matrix here. It's got the identity matrix in this first part. We use the symbol capital I for identity matrix.

So when you see identity matrix, you know it's a square matrix with 1s down the diagonals. OK. So what goes in the next column over here for this particular example? The next column over is going to be P1.

P1 is D1 plus D2. So what I need is 1, 1, 0, 0, right-- and similarly for the other parity bit. So once you're told the rule here, it's easy to generate the matrix that goes with it.

OK. So let's get some practice figuring out what's what here. This is all we're going to be aiming to do in this lecture is construct codes that correct up to a single error. So we're focusing on Single-Error Correction codes or what are referred to as SEC codes, OK, Single-Error Correction.

So assume that only one error has happened or zero errors. You don't get more than one, let's say. If you receive this, I've just rearranged the code word into the pattern that allows you to look at this very easily. So here's D1, D2, D3, D4, and so on. Any errors here in this?

You can see that, if I look along the first row, I've got even parity, even parity, even parity, even parity. So everything looks fine. And I'll declare that there are no errors.

On the other hand, if I receive that, OK, so here I have a parity check failure, right? And so I know that something is wrong in this column. I look along the rows. I see a parity check failure in that row. So I pinpoint the error as being at the intersection of those two. And I know that's the bit that I have to flip, OK?

And another case, here there is a failure on a row, but nothing on the corresponding columns. So what that tells us, it's actually the parity bit that's failed, right? Everything else looks fine, but the parity bit has failed.

If there's a single error to correct, it would be to convert this 1 to a 0. And then all parity relations are satisfied. OK. So you can get errors in the parity bits as easily as you get them in the data bits because the channel doesn't know the difference. The channel is just seeing a sequence of bits.

All right, so this is how you work backwards to figure out what's going on. Another way to say it-- and we'll see this later-- is you get what should be D1 and D2, but you're not sure yet whether they're in error or not. So let me call them D1 and D2 prime for now.

So you compute your estimate of the first parity relationship and compare it with what's sitting in-- well, let me say, are these equal? So what you're doing is you're computing your estimate of the parity relationship based on what's sitting in the code word in these positions and seeing whether it's equal to what you think it should equal. OK. And if it's equal, then you say that parity relation is satisfied.

And otherwise, you try and make a change. Now, we'll see how to do this more systematically next lecture, actually, when we'll go further with the matrix story. But I'm just trying to get you a little oriented here. OK.

So you probably believe by now that this code can correct single errors, right? The rectangular code can correct single errors. Basically, an error in a message bit is pinpointed by parity errors on the row and column.

A message in a parity bit is pinpointed by just an error in the parity row or column. And if you get something other than that, then you say you have an uncorrectable error, right? You're not set up to do things with other errors there.

But now, how do we know the Hamming distance is 3? The minimum Hamming distance is 2. We know that, if the minimum Hamming distance is 3, we can correct a single error.

But it's possible that the minimum Hamming distance is greater than 3 for this case, which might mean we have more possibilities. So how can we establish what the minimum Hamming distance is? Any ideas? Yeah.

AUDIENCE: [INAUDIBLE] the case in which the Hamming distance is [INAUDIBLE] change one of the data bits and and the two parity bits that correspond [INAUDIBLE].

GEORGE VERGHESE: OK. So am I going to search all pairs-- so the suggestion was change something until you find the Hamming distance of 3. And presumably, you won't find anything smaller, right?

OK. Because we know we can correct single errors. But am I going to search through all pairs of code words to do this? Or can I do something better? Yeah.

AUDIENCE: [INAUDIBLE] if you have [INAUDIBLE].

GEORGE VERGHESE: So you're giving me a particular computation here, but I don't know that you've answered my question, which was, am I going to have to search through all pairs of code words to see that I can establish a known distance of 3? Or is there something simpler than that that I can do?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Can you speak up? My hearing is not great, sorry.

AUDIENCE: [INAUDIBLE] high dimensional [INAUDIBLE]. It's whatever minimum Hamming distance is [INAUDIBLE].

GEORGE VERGHESE: Oh, so you've got a general formula, OK. Can you invoke linearity in some way? Because I haven't heard you use the linearity of the code in anything you've said. Were you going to offer a suggestion? Yeah.

AUDIENCE: [INAUDIBLE] what happens when you [? put one ?] [INAUDIBLE] when you [INAUDIBLE] parity [INAUDIBLE] you pick one bit you have to put two other bits. [INAUDIBLE].

GEORGE VERGHESE: OK. So I think I get what your argument is. You're saying start from an arbitrary message bit. And then if you make any flip, you'll get at least 3.

That may have been the earlier argument, which I missed, right? Is there a way to invoke the linearity of the code in making these arguments? You're on the right track, but I just want to see if linearity can be invoked. Yeah.

AUDIENCE: I think you can use the all 0 codes and [INAUDIBLE].

GEORGE VERGHESE: Right. So what we've established is that, for a linear code, the minimum Hamming distance is the minimum weight among all non-0 vectors, right? So all you have to do is start with the 0 code word, everything 0, and then flip a bit in the message and then see if you get Hamming distance 3 or greater.

OK. So we can start with the 0 code word. I guess that's the point I was going to make here. OK. There is another expression that popped up there before completing that argument. Do you agree with what it said about the code rate?

It says the code rate is rc over rc plus r plus c. Do you agree with that? Yeah. Because we have the number of message bits being rc. And then the total number of bits is rc plus r plus c. So the rate is, indeed, what's given by that expression.

OK. And then we'll go on to make this argument about the three cases here. So you can actually go case by case. And the argument here is actually closer to what was being described out there. It doesn't start with the 0 message.

But it says, if you've got two messages that differ in 1 bit in the message area, then they're going to differ by 1 bit in the associated parity areas. And, therefore, the overall code word has moved by 3. And then you go through each of the cases, and you argue that you've moved by at least 3.

So this argument is actually closer to what was being suggested earlier. OK. So you can go through each of the cases and discover that the nearest code word you can get to is Hamming distance 3 away, all right? Why is it that we're flipping a bit in the message section to decide what's a new case?

The way we count our code words is by arranging through all the possible 2 the k, right? So we've got to flip bits in the data section to get to another code word. So we're saying we have a code word corresponding to some set of data bits. We'll flip a bit there, and then look to see what happens, OK?

OK. So here's a little modification to the code, which actually puts in an overall parity bit, P here. So what this is is the sum of all the entries and every other position. OK. And if you go through the argument there, what you'll discover is what you've done is go from-- do I still have it on the board? I might have it on the board here. No.

What you've done is go from the rectangular code that had this structure, Hamming distance 3, to now one that has Hamming distance 4. OK. So adding that overall parity bit has increased the minimum Hamming distance of the code from 3 to 4. Does that improve error correction capabilities?

You still can only correct up to one error. But the difference now is that, if you get two errors, you can actually detect it accurately as a 2-bit error. OK. All right, so I'll leave you to go through that analysis.

This we've pretty much done already. This has just filled out the rest of the matrix. You see that we filled out this column on the board. That was this case.

But you can actually fill them all out once you have the description of the parity check bits. OK. So these other columns you can fill out similarly. And this is for the case of-- let's see, what case is it referring to here?

n equals 3. Let's see-- sorry, n equals 9. k equals 4. d equal 4. So what rectangular picture am I talking about here? This is a rectangular code. What rectangular code has these parameters?

So I must be talking about 2 by 2 for the data bits, 2 rows, 2 columns, and then 1 overall, right? The overall, what gives me the clue is I've got a minimum distance of 4. If it's a rectangular code with minimum distance 4, then I know I must have an overall parity bit.

k is 4 because I just have 4 data bits. And overall, I've got to send 9 bits in each block. OK. So for that particular case, this is what the matrix looks like.

So the only difference is there is an overall parity bit here, P5, which is the sum of all the data bits. Actually, all the data bits on the parity bits, but this is what it works out to be. OK. And we've pretty much talked through the decoding here.

Let me put it all up there. So you calculate all the parity bits. If you see no parity errors, you return all the data bits.

If you detect a row or column parity bit error, then you make the change in that particular position. And otherwise, you flag an uncorrectable error. So the correction is straightforward.

If you look on the slides later, you'll see a little quiz that you can try for yourselves. Or you might try it in recitation. But let me pass on that. OK.

So the question arises, is a rectangular code using this redundant information in an efficient way? Or could we do better? So let's see, we've got a code word that's got k message bits. And then it's got n minus k parity bits.

OK. So here's the data bits. Here's the parity bits. We want to use the parity bits as effectively as possible. How many different conditions can we signal if we have P bits that can only take value 0 or 1? Just 2 to the n minus k conditions, right? So if we're looking at the code word and trying to deduce something from the parity bits, how many different things can we deduce?

Well, n minus k bits can signal 2 to the n minus k different things, right? What do they have to signal? What do the parity bits have to tell us?

They have to tell us either that an error didn't occur or that an error occurred in the first position or second position or third all the way up to the n-th. So the number of things we want to learn from the parity bits is n plus 1. So you would hope that this is true, that the number of things you can signal with the parity bits is at least equal to the number of things you want to get in the case of single-error correction.

All right, we're only trying to correct a single error here. So we want the number of possibilities that the parity bits can indicate to include the case of no errors-- that's the 1 over there-- plus the case of an error in the first position or second position or third position and so on. OK. If you plug in the typical parameters for the rectangular code, you'll see that you're actually exceeding this wildly.

Let's see. For that particular case, 9, 4, 4, what do we have? We have 9 plus 1 on this side. And we have 2 to the what-- 9 minus 4.

So what's that-- 32 on the right-hand side and 10 on the left. So you've got a big gap. If you were going to allow me 1, 2, 3, 4, 5, parity bits, I could do a lot more than tell you what you're asking me to tell you in this particular code.

So I'm not using the parity bits as efficiently as I could. And that motivates the search for better choices. So this is a fundamental inequality here, something we'll keep referring back to. So make sure you understand where that comes from.

And that leads us to what are called Hamming codes. So Hamming codes are codes that actually use the parity bits efficiently in that they match this bound with equality. OK. So can you think of the smallest k and n pair that's going to satisfy this with equality just playing with some small numbers?

Maybe I shouldn't play this game since we're late on the lecture. Here's a suggestion, nkd. The Hamming code is going to be a single-error correcting code with minimum Hamming distance 3. So the 3 will always be there.

This is n, and this is k. And you'll see that this is satisfied with equality. But there are other choices. This is the smallest choice, the smallest non-trivial choice anyway. But you can go to more general possibilities.

So this code is called a perfect code because it matches that inequality with equality, but actually that doesn't necessarily mean it's the best code. It turns out to be a good code provided you're picking these parameters appropriately for your application. But this is perfect code in a very technical sense, meaning it's a code that attains this inequality with equality. OK, that's all that it means there.

OK, so what's the idea on the Hamming code? Let me put it all down there, and then we'll talk through it. So this little Venn diagram conveys for you how the parity bits are picked.

And they end up actually being picked in a very efficient way to provide the coverage you want. So this is the case that was mentioned before. This is the, was it, 7, 4, 3. Is that what we had? Yeah, 7, 4, 3, right?

So let's give ourselves some space here. So with 3 parity bits, we're actually going to indicate whether there was 0 error or whether the error occurred in the first position, second position, and so on, all the way up to the seventh position. So we're going to use 3 bits, 3 parity bits, to indicate eight possibilities, which is what we know you should be able to do.

And here's the arrangement. This picture conveys it. So basically, P1 is D1 plus D2 plus D4. So P1 fires if D1, D2, or D4, if any one of those is 1 and similarly for these other things. OK. So this picture tells you what data bits are included in the coverage of each parity bit.

So that's the way to think of what this picture is. So these are apportioned carefully. So for instance, let's see, if you discover that P1 and P3 indicate an error, that means some data bit in the coverage of P1 and in the coverage of P3 have an error.

But P2 didn't have an error. OK. So what does that tell you? We're only considering up to single errors. If P1 and P3 have an error, but P2 doesn't have an error, well, P1 and P3 share D2, D4.

But P2 didn't have an error, so D4 must be fine. So D2 must be the one that's an error, OK? And so you get full coverage by that kind of reasoning.

One way to think of this, and this is actually how Hamming set it up, was he actually arranged the parity bits and the data bits a little differently down that code word. He had parity bit 1 in the first position, parity bit 2 in the second position, parity bit 3 in the fourth position. If you had a long code word, the next one would be in the eighth position.

So it's 2 to the 0, 2 to the 1, 2 to the 2, 2 to the 3, and so on. So those are the positions in which he puts the parity bits. Everywhere else are the data bits.

And then the data bits that feed into parity P1 or parity relation P1 are the data bits and positions that end with a 1. OK. So if the positions end with the 1, you stick them in the coverage of this parity relationship, so D1, D2, not D3, but D4. For P2, similarly, the parity relation P2 includes the data bits that have a 1 in their second position, so D1, not D2, yes D3, and yes D4.

OK. So the nice thing about that is that, when you get a particular pattern of errors, it actually leads you exactly to the right position in the code word. So I don't want to actually spend a lot of time here. I want you to look at that separately. But just to go over the process, here's what happens.

We know that parity bit P1 was D1 plus D2 plus D4. So we know that this parity relationship was satisfied at the transmitting end. By the time you receive all of this, all of it might have been corrupted. That's why I put a little primes next to these.

Not all of them, but one of them may have been. We're limiting ourselves to a single-error correction, OK? So the D1 prime may not be D1 because of an error.

So what you do is you compute these so-called syndrome bits. If there were no errors, then E1 should be 0, E2 should be 0, E3 should be 0 because that's how it was on the other end. If there's a single error in one of the bits covered by the appropriate relationship, you're going to get the associated syndrome bit going to 1.

So you compute these syndromes and then line them up as a binary number. And it turns out that, depending on the pattern of the syndromes, it'll tell you exactly the position and the code word in which there's an error. So it's kind of cute and powerful.

You can correct up to t errors. And there's a natural relationship that extends from this. I wanted to make one final point, which is that these error correcting codes occur all over the place, not just in the setting of binary.

And one thing you might try and DO if you're carrying a textbook with you, look at the ISBN number. The ISBN number is a 10 digit number x1 up to x10. And it turns out that 1 times x1 plus 2 times x2 plus 10 times x10 is going to be 0 modulo 11.

So try that out on the ISBN number of any book you're carrying. What you'll see is that this is a parity relationship that guards against errors in any single ISBN number or a transposition of two, which turn out to be the two most natural errors. OK. Look for parity relations in other places.