MITOCW | 5. Error correction, syndrome decoding

GEORGE VERGHESE: OK, let's continue. So we're going to continue with linear codes and talk today about error correction. So let me just remind you, we're thinking of linear codes very concretely as being generated through a process like this. We put this up on the board several times. You've got the data bits and then the parity bits being generated by the data bits multiplying into a so-called generator matrix. You've seen this in lecture on recitation as well.

And we've considered different ways to think of this matrix. One way is to think of it as made up of a bunch of rows, and what you're doing is taking linear combinations of these rows to generate a code word. So the dimensions here-- this is going to be n. So when you take a linear combination of these, you're generating a word that's n bits long. But the underlying degrees of freedom only correspond to k bits, because you're just doing a weighted combination of k of these. OK?

Now, we talk of these as though they're vectors, you're combining them, take the linear combinations, and so on. And I just wanted to say a word about in what sense this is a vector. So this is an array of n bits. So we're talking about something-- I'll call it v, let's say-- which is an array of n bits.

And the question is, in what sense is that a vector? In what sense does it live in a vector space? So when we say vector space, we're usually thinking of arrays of n elements with real numbers in them, and the kind that you use in physics, where you take linear combinations of them with real numbers and you get new vectors.

This is the same kind of thing, except it's working over not the real field, but as we've seen, gf2. So this is a vector space over gf2. It's a funny vector space, again, because it has a finite number of elements. The vectors that we're used to thinking of-- Euclidean vector space has an infinity of elements, because you can have an array of n components, but each component could be any real number.

So any point in 3D space would be a vector in r3. So this is a vector space over gf2, and it only has a finite number of components-- only has 2 to the n possible vectors. It's a finite set of vectors, so it's strange that way too. So in what sense is that a vector space?

Well, it turns out that they're pretty abstract things that you can refer to as vectors, provided they satisfy certain axioms. So what you want to be able to do is define a sum of these objects. You need to have a set of scanners and define a scalar times vector multiplication.

And then you need a 0 vector, a vector that, when you add to another vector, gives you the same vector back again. You need certain distributivity properties. So if you take a scalar times the sum of two vectors, you get things like that. So you can list a bunch of these properties. I'm not trying to teach you are the axioms are that define the vector space.

But there's a set of axioms, and you'll recognize very quickly that Euclidean space satisfies those axioms. But the point is there are other objects that satisfy the same axioms, and you can work with them as vectors-- so notions of independence of vectors, a basis in terms of which you write other vectors-- all of these.

Now, I'm not assuming you've done a linear algebra course. I'm assuming you've picked up some of this in the course of doing physics, and so on. I'm just trying to talk intuitively here. One thing we don't have here is a notion of an inner product, or a dot product, or a scalar product. So if you had two n component vectors in Euclidean space-- you're probably used to this from physics-- you'll take inner products defined in this fashion.

Well, we can certainly do this kind of computation with the elements of a vector here, but the resulting object doesn't have the properties of an inner product. For instance, you can take the-- if you take the inner product of two non-zero vectors in real vector space, you'll never get-- well, you can get the inner product to be 0 under very special conditions. There's a notion of orthogonality. It turns out that doesn't actually work quite the same way here over this space.

So what do we do have is we set aside orthogonality. We'll talk about linear combinations of vectors. We'll talk about a set of basis vectors. So a set of basis vectors would be a set of vectors that you can take linear combinations of to get other vectors in the space-- and a minimal such set. So we'll be using a bit of the language of vector spaces. You might have some notions of that might come from what you've done with physics. And that's all really that we want to depend on.

All right, so back to this-- what we have is these arrays of n bits. We think of them as vectors in some space. The dimension of the space is the number of vectors that you need in order to generate other vectors by linear combination. So the question is, can I generate some vector by taking alpha 1 v1 plus alpha 2 v2 plus alpha 3 v3?

So I'd like to be able to generate a vector in the space by taking a linear combination of other vectors. So if you ask for what's the minimum number of such vectors you need here in order to be able to generate any vector by taking a linear combination, that's the dimension of the space. So in that sense, it turns out that these anaerobes live in an n-dimensional space.

But they don't span all of n-dimensional space, because you're just-- you've just got k of them here. It turns out that what you get by taking linear combinations of these is a k-dimensional subspace of an n-dimensional space. So in some sense, when you define a code, what you're doing is you're saying, I have this n dimensional space that my words can live in, but I'm going to restrict myself to words that live in a k-dimensional subspace so that, i a vector pops out of that subspace, I recognize it as being an error. So that's the general idea. All of this can be done more carefully using the notion of vector spaces. I just wanted to give you a rough idea of that.

This is one way of thinking over. Here was another way of thinking of it, which was column-wise. We think of the generator matrix as being made up of a bunch of columns. And that's useful when you want to think about how a parity bit is defined in terms of the data bits. So here's what you see when you think of this column-wise.

So let's take p1 here. Actually, let me specialize this further. We've already said that, because of the form in which we set up our code words, this is in what's called systematic form. We've got the data bits sitting there, and then we add in the parity bits. Because of that, we've said that there's an identity matrix here all the way down as 1's. And then we've got some other matrix here, which is something we'll denote by A.

OK, so when we do this multiplication in the first k positions, we just pick up d1 to dk. In the next position over, I get the expression for p1. So p1 is going to be d1 times the first entry there plus d2 times the second entry, and so on. So here's what I get.

Let me call this A11 in that first row. Here's A21, all the way up to Ak1. So these are just numbering the entries down that first row. So I'm taking a combination like that, a linear combination of the data bits to get that first parity bit.

So the j-th parity bit is found by going over to the j-th column. The entries here are A1j all the way up to Akj. So the way matrix multiplication works-- if I'm looking for the j-th entry-- the j-th parity bit here, I take this and do the dot product kind of expression with the j-th column, so this is what I get.

So this is a typical parity relation. And it goes the other way too. If you had this expression, and not the matrix, you can just take those numbers and translate them back in. And these numbers are just 0, 1 in our-- in the case of a binary field that we're working over. That is just 0, 1. So you either have the data bit there or you don't. All additions or modulo 2 additions, of course.

Actually, let me call this the parity-- it's the parity definition. I may have had another term for it in my slides. Let's see here-- this parity equation. It just defines the parity better. Here's what I think of as a parity relation. What is this sum? What does that sum work out to be?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Sorry?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: 0-- because I'm adding pj to itself, and then gf2. That gives me 0. And this is what I think of as a parity relation. So a parity equation defines my parity bits, but I get immediately from that a parody relation that relates my parity bit to my data bits. Turns out this is important for the way we're going to talk about our correction.

OK, so let me step off the dime here. And this is a particular example. I just wanted to set up the general notation before we got back to this. We've looked at this before-- 9, 4, 4 rectangular code. So what might that be? How many data bits? 9, 4, 4-- 4 data bits, right?

So how would I get in 9, 4, 4? I'd have D1, D2, D3, D4. And then, depending on how I number this, P1, P2, P3, P4, P5 would be one way to number it. I don't know if that corresponds to what's up here. Can we check? So what's P1? P1 is going to be D1 times 1 plus D2 times 1, and that's it. So it's d1 plus D2. So P1 is indeed that element there on the board.

And so you can check each of these entries. Let's see. P5-- that's the last entry up there. That's going to be this row inner producted or dot producted with all the sequence of 1's there, so it's going to be D1 plus D2 plus D3 plus D4, which is indeed how that overall parity bit is defined. So again, you see in the generator matrix, you have the identity matrix there, and then you have this matrix that we're referring to as A.

Now, the notation is a little bit different reading chapters 5 and 6, by the way. I've tried to stick with the notation I had in lecture last time in the chapter 5 notation, which uses capital D for the data bit vector and capital C for the code word. So you'll see slightly different notation in chapter 6, but I think you'll navigate fine.

One other term here-- we say that these code words live in the row space of G. So the space that we generate, the space of vectors, the subspace of the big space that we generate by taking linear combinations of these rows is referred to as the row space of G.

So we define the code by defining a G. If the code's going to be in systematic form, we have the identity here, and then some matrix. This is all for linear codes. And then the code words live in the row space of this matrix. In other words, they're obtained by taking linear combinations of the rows.

Here's what I already have on the board. And it's just to say that the matrix A that's sitting out here-- this piece-- is obtained directly from the parity relations. OK, so let's think about this. Can two columns of the matrix A be the same? What happens if two columns of A are the same?

Let's say these two columns are identical to each other. What is that telling us about the code that we have? Yeah?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Yeah. And so basically, one of those parity bits is not buying you anything, right? Yeah. OK, so if you did discover two columns two columns were identical, then one of those parity bits is not checking a different linear combination-- checking the same linear combination, and so it's not buying you anything.

What about two rows? Can two rows of the matrix be identical? So let's actually think of them. Erase that-- can I have two identical rows here in A? And if I did, what would it mean? OK, let's say I have two identical rows. What does it mean? What does it signify? Someone? Yeah?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Could you speak up? Sorry. My hearing's not good.

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: So there'll be two data bits that are entering the same way in every parity relation. If two rows are the same here, then there are two data bits here that are entering every parity relationship in exactly the same combination. And so you're not going to be able to distinguish between an error that happens in one of them and the-- and an error that happens in another one. So this is a problem.

All right, so there are certain conditions at the A he has to satisfy. All right, here's another important matrix. You may have already seen it in reading. You may not have seen it yet in recitation. And it's a matrix that we call H. And let's just think about what it's doing.

What I'm trying to do is basically summarize this set of equations, the parity relations in matrix form. So let's take a parity relation that we had in this particular case. In fact, let's go back to a specific one. Let's take the first parity relationship that we had over there.

We said that P1 for this code was D1 plus D2, right? That's the equation for the parity. The parity relationship is this. How would I express that in matrix form, as part of a matrix equation? Well, that's what we're starting to assemble here. So let me show you what that is.

Let's look at the top row out here. What's the top row telling us? It says 1 times D1 plus 1 times D2 plus 1 times P1 equals 0. That's all that enters. So that first row is capturing the first parity relationship. And you can go down here. Go to the second row. This is saying D3 plus D4 plus P2 is equal to 0. That's indeed the parity relationship associated with the second row in the rectangular code.

So all that this is doing is listing all the parity relationships. So how many of these do you have? Well, you have as many relationships as you have parity bits. So this is n minus k times n matrix, and it's just listing the parity relations. We call this matrix H, and interestingly-- let's see-- there's an identity matrix sitting here, and the reason is that each of these equations involves only one parity bit. There's only a single PI that's involved in each parity relation, and so there should be only one of these columns picked as a 1 when you get to this segment that multiplies the parity bits.

So there's an identity matrix sitting there, and then there's the rest of it here. So here's the identity matrix and here's the rest of it. And not surprisingly, the rest of it-- well, it comes from the same set of coefficients, and so it relates to the A matrix. And if you look at it carefully, it turns out, well, it is the A matrix, but turned on its side. A superscript T means A transposed-- IE, rows become columns. We're taking the a matrix, and in some sense, turning it on its side.

So that makes sense, because what defined the first parity relationship? Well, we found it in the columns of A here before. That's what defined the parity relationships. And now we're writing it out in this row form, so you've got to transpose things. You've got to take what was a column in A and make it a row. So what you're seeing out here at the top-- that was the first column of A. So now it's the first row of A transpose. This was the second column of A. Now it becomes the second row of A transpose, and so on. So that's what that relationship is.

So I can write the parity relationships in the form H times C equals a whole bunch of 0's. This is k parody relationships. This is k 0's here. But I could read write the same thing turned on its side, and that's what you're seeing over here. So if H is the matrix there, what is H transpose?

Let's see. I'm going to change-- interchange rows and columns. So what's going to happen is that the first row of H is going to become the first column of H transpose. And so if you imagine how this gets turned on its side, here's what H transpose looks like. H transpose looks like that. So when you transpose a matrix whose entries are blocks, you end up flipping the matrix around, but also transposing each of the blocks. So that's A transpose.

Oh, actually, sorry. I wrote this one wrong, didn't I? What is our vector C? C is this vector. It's a code word. When I set this up in matrix form, I got a column vector, so I've got to transpose this as well. That's what I was missing. That's C transpose that I'm looking at.

So when I write down the parity relationships, I've got this, but I could also write it in the form-- when you transpose a product of things, what do you get? Well, it turns out, if I transpose the product of two things, I get the product of the transposes, but in the reverse order.

So these are just different ways of arranging the equations. So I could have written it in this form or I could take the transpose of both sides, and I get something that looks slightly different. So here, this would be a row of 0's. So this is just to get you comfortable with the matrix operations. You'll see lots of this in chapter 6. You want to get a little comfortable with that.

Here's a question for you. Here's my H matrix. Here's my code words. If I have a code word of minimum weight, how many 1's would it have in it? In this particular case, this is a 9, 4, 4 code. If I had a code word of minimum weight, how many 1's would it have in it? I heard something, but I didn't hear where it came from, or didn't hear it very clearly. Yeah?

AUDIENCE: 4--

GEORGE VERGHESE: 4? Yeah, OK. [INAUDIBLE] heard there. OK. So the code word of minimum weight would have weight 4, because this is a code of distance 4. It's a linear code. The minimum Hamming distance is 4. It's a linear code, so we know, for a linear code, the minimum weight you'll find among all code words is 4.

OK, so we've got a vector here that has four 1's in it, and everything else is 0. So when I take this computation, what is it that I'm actually doing? If I take the matrix H and I multiplied by a vector that has four 1's in it, and everything else is 0, what is it that I'm actually doing to that matrix? Yeah?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Of the rows? Am I taking combination of the rows?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: When I do a multiplication in this fashion with the vector on the right-hand side, I end up doing the opposite of what I was doing here. When I have the vector on the left and I multiply, I'm taking the combinations of the rows. When I've matrix times vector, I'm taking combination of the columns.

So if I had a vector year with four 1's in it, and everything's 0-- everything else 0, I'd be taking-- I'd be picking out four columns of this to add, and the result would be a 0 vector. Yeah?

AUDIENCE: [INAUDIBLE] all 0 [INAUDIBLE]

GEORGE VERGHESE: All 0's here?

AUDIENCE: All 0's for your code--

GEORGE VERGHESE: Were you asking, why all 0's for the code?

AUDIENCE: Well, why you can't have all 0's?

GEORGE VERGHESE: You can have all 0.

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Sorry. The minimum weight non-zero vector is the Hamming distance. Sorry. I may have dropped that word. For a linear code, we know that the minimum Hamming distance is the minimum weight non-zero vector. That's what I meant to say. I may have neglected to say that. Thanks for catching that.

So we have a non-zero vector here. It's got four 1's in it. What that tells us is that there are four columns here that we can add together and get the 0 vector. Do you think it's going to be possible to find three columns here that we could add together and get the 0 vector?

I'm not asking you to actually do the computation in your head, but based on the reasoning we just had, if you found three columns that could be added together to give you the 0 vector, that would mean you'd have a vector here, a code word with-- or any vector here with three ones in it, everything else 0, such that this product was 0. So it would be a valid code word.

Is that possible to take a vector here with just three 1's in it, everything else 0, and have a valid code word? No, right? We said the minimum Hamming distance is 4, the minimum weight code word here is-- has got weight 4. You'll not find a valid code word of weight 3.

So you can actually look at this H matrix and figure out what the minimum Hamming distance is. It's basically the minimum number of columns that you can add to get the 0 vector. Now, does that tell you, by the way, why you can't, in this instance, have-- well, if you discovered that you could add two columns together and get the 0 vector, what would that tell you?

If two columns of A transpose were identical or two rows of A were identical, that would tell you that you can add two columns and get the 0 vector. That would mean that the Hamming distance is 2, not 4, right? A Hamming distance 2 code is not worth much. It's no good for error correction.

That comes back to what I said earlier in these questions. If A had two rows the same, well, there's two data bits that are always entering in the same combination, so you're not protecting against individual errors there. And so it's exactly that issue that we're seeing. The Hamming distance ends up being 2, if you have two rows of A that are identical.

OK, so there's a lot that can be gleaned from the generator matrix and the parity check matrix. So this is what I just went through-- that, if you have the H matrix, you can get the minimum distance D by looking to see what's the minimum number of columns you can add to get the 0 vector.

All right, now, how does decoding work? We've gone through this effort to generate a code word, and then, at the receiving end, we get some word. This is some received word, and it's going to be a code word plus possibly an error. And now we want to figure out, is the thing we received already a code word? Or if it's not, what code word can I correct it to? And we're going to assume we have just single-bit errors.

So one way to do it is just an exhaustive search, which is you've got this received code word, you know that it's going to be one of 2 to the k-- so it's going to be no more than having distance 1 from the 2 to the k code words that you have in your code set. So you can compare against those 2 to the k code words, and whichever one it's having distance 1 away from, that's the one that you're going to announce. So that'd be one way to do it.

The thing is that that's not exploiting anything nice about the structure of linear code, so what I want to talk to you about now is a way to actually capture this error in the case of a linear code. So this builds on the relationships that we've been developing some intuition for here. So here's what happens.

You get a received vector and bits, which is valid code word plus a vector E that has a single 1 in it, and everything else 0, or is completely 0. So if you were receiving the code word correctly, you've had no errors, and this is what you get. But if you've had a single bit error, then E is a vector with a single 1 in it. It's n bits long, has a single one in it, and that gets added to the code word to give you what you receive.

So here's what we're going to do. We're going to exploit the relationships that we had up here. We know that H times is a valid code word equals 0. Or I can write it the other way around if I want to do row multiplications-- sorry-- C times H equals 0. So I'm going to take the receipt vector and do that multiplication with it. And in this case, I've chosen to write it as H times the transpose.

So if the received vector was a valid code word, I'm going to get 0. If the received vector was not a valid code word, I'll get something else. And that's what we refer to as a syndrome vector. OK, so let's expand this out. We've got R is C plus E-- so C plus E transpose. That's the multiplication we're going to do.

This matrix multiplication will be the sum of the individual matrix multiplications, so it's going to be H times C plus H-- sorry-- H times C transpose plus H times E transpose. So let's write that out. So we've got H times R transpose is going to be H times C transpose plus H times E transpose. We know this is 0. It's a 0 vector. n minus k here, the number of parity relations-- that's how long that 0 vector is. Yeah. And then we've got some other vector, which we're referring to as a syndrome vector.

OK? So let's see. What does HE transpose look like. E transpose is going to have just a single 1 in it somewhere. And here's H. Let's see. A transpose stacked up next to the identity-- that's what H looked like. So when we compute-- let me write this better-- I didn't write that well-- this is H.

And what I'm writing here is HE transpose. So what is HE transpose doing? When I multiply a matrix like this by a vector that has just a single 1 in it, what am I doing? What do I end up doing? Sorry. I heard a voice from somewhere here.

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Picking up one column, right? I'm just selecting out a column of H. So each error that you can get will give you a syndrome that corresponds to picking on one column of H. And column of H is associated with data bits here or parity bits here. So really, this is all that you have to do to do your error correction. You can pre-compute, or store basically the columns of H in your database-- compute H times the received vector to get the syndrome.

That's really H times the error, which is giving you the syndrome. That's just a single column of H. So the syndromes that you can possibly get are individual columns of H. So you know what H is. You've got it stored. Compute the syndrome, and see which column of H it corresponds to. That's the bit that has the error. And actually, the only cases you're interested in are where you are going to correct the data bit, so this is really all the part that you really have to focus on.

So you compute the syndromes. You compare against the columns of H, which are your syndrome vectors, and then you're done. I think I see the same thing over here. Let's just look at it concretely-- again, for the same code, the rectangular code with all the parity bits there. So this is how you generated a code word.

Sorry. OK, let's take the data bit-- the data vector being all 1's. This is the code word that goes with it. It happens with this particular code that all the parity bits then are 0. What you receive ends up being this, because one of the data bits ends up getting corrupted. When you take that received vector and pre-multiplied by H, here's the resulting syndrome vector that you get.

And what error does on correspond to? Well, actually, if you look in the columns of H, you'll see that what you've pulled out is the second column, so that means a second data bit is an error. And that's the change that you make. So it really is that simple. You take the received word pre-multiplied by the parity check matrix H, look at the syndrome vector, and see which of the columns of H that corresponds to. That's the bit you're going to flip. All right?

So now you're actually only dealing with this many vectors. It's a number of vectors equal to-- how many is that? We've got to do the multiplication, but you just have to compare with the number of vectors in those columns. So it's a much simpler task, computationally.

OK, I think we've said all this. And so let me just wind up on linear block codes with a quick summary, and then we'll go on to talk about some extensions here. We've seen all this. We know what the rate of a linear code is-- k over n-- how many errors we can correct. And we've seen all this-- what a parity bit does, whether repetition code-- it's called replication code in the notes, but the more familiar term-- actually, the more commonly used term is repetition code. We've looked at Hamming codes and the rectangular code as well.

And so these are the ones that you want to have in mind as particular examples to work with when you're trying to come up with examples that will either prove or disprove-- that will illustrate a conjecture or disprove a conjecture, for instance. And you'll see many problems on past quizzes that are of that type. And then what we did today was looking at syndrome decoding.

All right, so this was all focused on single error correction in linear codes. But the point is that that may or may not be the situation that you're dealing with. We actually said that, to get better error protection while maintaining high data rates, you probably want to work with longer and longer strings of data. Well, if you work with longer strings of data, you're going to get more bits and error. So you may not be able to limit yourself to thinking about single-bit error correction.

We have talked a bit-- and you've done this in recitation too, I imagine-- well, you've probably done more in recitation than in lecture-- with independent corruption of multiple bits. So let me say a few words about that. Let's think of a systematic code, for instance, still k bits here, and then parity bits here.

But what if you could have up to t errors, not just a single error-- so if you wanted to protect against t errors? So in some sense, you want your n minus k bits here to signal all those possibilities, so you need the number of possibilities that can be signaled by n minus k parity bits to be greater than or equal to the number of possible conditions that correspond to having up to t errors.

And we've said a little bit about this, but you can have now either no error at all, which is one condition; or an error in one of these bits, which is n separate conditions; or you could have two bits out of here being an error. So how many conditions is that? n choose 2 and so on-- I'll leave you to figure out where you end up on that.

So I just wanted to say-- you've seen this in recitation, but I haven't mentioned it in lecture. I don't want to say later that, oh, I didn't know it was something we had to know for a quiz. We do expect that what n choose m means. So n choose m is the number of ways of picking m things from n things, and we assume that what that-- how that's done.

So you've got n objects. You want to pick m things from there. So you can pick the first one in n ways, the second one in n minus 1 ways, and keep on going until you get to n minus n plus 1, which is also n factorial over n minus m factorial. But when you did that picking, you were paying attention to the order in which you collected the things, but if the ordering doesn't matter to you-- if all these objects are interchangeable-- then you've actually overcounted.

So you've got m things, but the order in which you pick them doesn't matter, because they're all interchangeable. And so you've overcounted. You've got to divide by the number of ways of rearranging m things, and so that's how you get that expression. So you start off with thinking about how you pick m things, and then make a little correction, and so this is what n choose m is.

Another thing that I just threw in for fun, because it's something you might want to carry around in your head-- if you don't have a feel for how n factorial grows with n, well, it actually grows pretty fast. It's actually growing like n to the n. This is a very famous approximation, referred to as Sterling's approximation. So when you get out to large n, the right-hand side here is a very good approximation to n factorial.

And you see that it's sort of like n to the n, which makes sense, because you seem to be multiplying n by itself n times. Except you're multiplying by a little bit less than n each time, so the e over there ends up compensating for it, it turns out. And then there's an extra n to the 1/2 out there.

OK, so we'll assume that how to do the combinatorics. And now, what this is saying is, what's the probability of getting m bits and error in an n-bit word? Well, if you've got m bits in error, that's because those m bits flipped independently, each with probability p. The remaining n minus m did not flip, so that's the probability of getting one such configuration. And then you count all the possible configurations. So what that top expression is is the probability of getting m bits in error out of n, and that's something that we want you to be comfortable with.

OK, now, just to wind up here, I want to go back to this last bullet, which is that, in many situations, the errors don't occur independently in the different bits. If you think of a CD with a scratch or a thumb print or something on it, that's local, and so if you get one bit corrupted, that increases the chances that the next bit is corrupted as well. So errors can occur in bursts.

If you're trying to make a phone call from a car, and you're suddenly shielded from an antenna-- a nearby antenna, then you're going to lose a whole bunch of bits in sequence. So bits can be in error in clusters, and what we've talked about so far doesn't quite manage that.

So here's an idea for how to do that, which is referred to as interleaving. So if we had B different code words that we were going to transmit, and we did it the normal way, we would send out the first one, second one, and so on. This little shading here is supposed to indicate the parity bits going along with the data bits.

If we had a burst of errors, we could lose two entire words over there-- nothing to be done. They derive entirely corrupted, and we wouldn't get them back. The idea of interleaving is stack up B words that you want to transmit, but transmit the bits out one at a time from each of the B words. So you transmit the first bit from the first word, first bit from the second word, and so on. And so this is the sequence in which you're doing the transmission.

Now, if you've got a burst of errors, you're corrupting a more localized set of bits in each of the words, and there's some hope that your error correction then can recover. So this is very often done. I don't think I want to actually walk you through a particular scheme for it, but we'll have it on the slides for you to look through. But basically, it actually turns out to work very well.

And that may be all I want to do for today. We'll see you next time. We're going to talk about linear codes next time, but a much more elaborate kind of code called a convolutional code. Thank you.