MITOCW | 3. Errors, channel codes

GEORGE VERGHESE: I wanted to give you an overview of what the system is that we're talking about, the communication network. We have a source that's trying to communicate to a receiver. We've talked about converting-- well, we'll talk some more, actually-- about converting the source information to binary digits.

And then where we've spent a lot of time is talking about source coding, which is trying to extract the redundancy in the message that you want to send so that basically every binary digit you then put on the channel carries as much information as possible. So now I'm going to really stop making a distinction between bits and binary digits. I'll just say bits when I might mean binary digits. But once you're into the system here and you've done your source coding, a binary digit carries a bit of information, in general, if you've done a good job of extracting the redundancy.

So we're talking about a bit stream here that you're trying to get across to the other end. At the other end, the bitstream is received. There's the decoding step, which is what we've seen with Huffman or LZW, the decoding end. And then you do whatever it is you're going to do in the application. So we've really said all we're going to say about the source coding and decoding, and the rest of what we're going to do is focus on what happens inside here.

Now what happens inside there at some stage involves a physical communication link. So you might be talking bitstreams at either end, but somehow you've got to deal with the fact that most of these channels, most of the channels of interest, they're physical channels, they work with continuous valued, continuous time quantities.

For instance, the voltage on a cable might be used to transmit information, light on the fiber, electromagnetic waves through space, acoustic waves in air or water-- in fact acoustic waves in air is something you see a lot of when you come to the later labs-- indentations on vinyl or plastic. Let's see, that's records or CD's.

And that actually brings up a point. We don't often think of storage as being communication, but storage is really communication with potentially a very long time delay in the channel. You put something on the storage medium, and then weeks or months or years or centuries later, you're trying to extract that information.

So we can still think of all of this as a communication channel, and indeed, decoding ideas are essential to making CDS work, to having a CD resistant to scratches and thumbprints, and everything else that, all the other indignities that they're subject to-- magnetization. So all of these physical modalities that are used to translate information. Here's one you may not have thought about, mud pulse telemetry.

So when you're drilling for oil, you'd like to get information from the drill bit at the bottom. And normal electronics doesn't work too well, because the temperatures are fiercely hot down there. You need that information to help you steer the drill bit, to get information about what sort of rock you're going through, and all of that. So they actually seriously do use pressure pulses and in the slurry that's cooling the drill bit to try and convey information back to the top.

So they'll modulate the pressure down at the end of the drill bit, and hope that you can detect it at the top. One word over there that stands out is they talk about digital information. So even in the context of communicating through mud, they're thinking about how to actually have bits that they communicate on this analog channel. So this is very much in the flavor of what we're trying to do.

We're trying to communicate digital information, this is sequences of numbers or sequences of signs or symbols, but we're trying to do it over a physical channel that takes continuous valued, continuous time, waveforms. So that's really some of what we'll be talking about.

So the kind of link that we have starts off with bits. But in the middle, has to deal with the physical link on which you have signals. We'll refer to these continuous time waveforms. They're not always continuous time. You could sample them and get discrete time waveforms, as well. But the signals that you see in the physical medium will refer to-- the quantities you see in the physical medium will refer to as signals.

You need some way to map the bits to the signals. You've got a bit sequence, you need to convert it to a continuous time waveform in some fashion. And then at the other end, to recover the bitstream. And you might do that by some sampling and processing, and then a translation process back.

The little lightning there is meant to suggest that you're subject to all sorts of noise and disturbances when you're on that physical link. So that's something that's critical to the design of the system. You have to design your overall system so that your robust to perturbations in that middle section.

So the particular application is dealing with your specifics and with producing a bitstream that's got the redundancy mapped out of it. At this point, it doesn't matter to me for communication across the channel what that bitstream is or where it came from. I'm just trying to do a good job of delivering it to the other end where the user can extract that.

Now here's the funny thing. We've just done a lot of work to extract redundancy from the messages here. We're going to put redundancy back in. Because the way you guard against disturbances in the channel is by introducing redundancy. You need to give yourself a little room to recognize that something bad has happened to your signal, or something bad has happened to the data you're sending across, and then to recover from it.

So we will actually be talking about how to reintroduce redundancy, but this is introducing it now in a bitstream where the binary digits are essentially equally likely. You pulled out all the application-specific knowledge, and used it to do the source coding.

Now you've just got a stream of zeros and ones, each one carrying a bit of information, presumably. And now you want to protect it for transport across the channel, and you're going to do that by putting in additional bits. So you're going to introduce bits in a structured way to provide some redundancy. So we'll be talking about that in more detail.

So this is the single link picture. When you've got a network, it's a little different. So I haven't shown all the other links here, but you should imagine a network with many possible paths from a source to a destination. Some of these links might come down, and so you'll want to find an alternative path. Some of these links might be busy, and you want to find an alternative path. So there's a whole network of links in here.

In this setting, it turns out that the good way to do this is to break up your bitstream into what are called packets. These are maybe 1,000 bits or 4,000 bits, or whatever your protocol is. They're broken up into packets of chunks of bits, and then treated as packets for transport along the network.

So the point is that one packet to a given user might travel one particular route, but another packet might travel another route. And then these get reassembled at the destination. So you think in terms of packets when you think in terms of rooting on the network. This idea of packet communication, by the way, there's a name associated with that, which is Kleinrock, again, a PhD student at MIT in the same golden years that I keep referring back to. But it's a very broad area.

So you've got the packets. Those arrive at the links. They're actually switches here that decide which of the links emanating from the switches you want to send the packet on. So, again, imagine all the links. I haven't drawn them in. So the packet gets treated as a unit for shipping on the links.

But once you've committed to a particular link, it's like transmitting on a single link, again. So you've got to go through your packets to bits to signal to bits to packets transformation. So it's not that there's a particular transformation in going from packets to bits, you're just viewing it differently. You're not treating it as a packet, you're looking in on each bit. You're looking to code each bit on an analog signal, get it across the physical medium.

So that's really the key to this. What we end up doing is coding, or mapping, or the word modulating is used, and we'll see more of that. We modulate the desired sequence onto a continuous time waveform. So what you might imagine is, you could have a sequence 01101, and what you're going to do with your mapping is try and generate a continuous time waveform, which in some fashion codes that sequence.

And it could be very simple. It could be a voltage level of 0, held for some interval of time to represent the 1-- sorry, to represent the 0, held for some time again to represent the second symbol, which is, again, a 1. And then you come back down to 0, back to 1 again. So it could be as simple as this.

OK, so this is now-- you can think of it as some voltage. We use the word voltage a lot, but we just mean an analog signal. We'll use the word voltage, we're thinking of voltage on a cable, but it could be any analog signal. So this is really the digital signaling end of it.

So you take the bit sequence represented now on, coded onto a continuous time waveform, or modulated onto a continuous time waveform. We'll see richer uses of that word as we progress. The particular scheme I've shown here is what you'd call a bi-level signaling scheme, the two voltage levels that you use.

We refer to that also as bipolar signaling, although sometimes that's restricted to the case where the two voltage levels are opposite in sign. So you can imagine a signaling scheme that uses this for 0, this for 1. So this could be a bipolar scheme.

And then this continuous time waveform gets put on the physical channel, presumably gets distorted, it gets some noise added to it. So at the other end, you get some approximation to this. And then you've got to reconstruct the bit sequence. You might do that by sampling this and processing the samples, and then taking some measure of the waveform. We'll see more of that later, you'll do that in lab 2, in labs, as well.

So in some fashion, you recover from this your estimate of the bit sequence. But you can imagine, the amount of waveform like this goes through a physical channel with distortion, noise, and then re-sampling and processing and so on, you are likely to get errors back at the receiving end. Ideally, you would get exactly this waveform at the receiving end, you'd have no trouble distinguishing between the samples of the two levels, and you could reconstruct the bit sequence.

Now we are going to, in the middle section of this course, say a lot more about the signals aspect of it. But for us now, it's actually helpful to stick to thinking in terms of bits. So we've got bits in, bits out. Somewhere in here, we've got signals in the physical channel, signals, noise, physical channel.

And then we've got whatever it is that does the transformation from bits to-- so this is some kind of a transformer, let's say, from bits to signals and from signals to bits. But let's look at an abstraction that's end to end here, bits to bits. OK what's coming in as a bitstream, what you're receiving is a bitstream.

There's an idealization of this channel that's used a lot, and that's referred to as the binary symmetric channel. And what that says is, you got a 1 coming in, that's most likely going to be received as 1, but there's some chance it's going to be received as a 0. And let's put probabilities here. I think the notes use epsilon. I seem to have used p, little p, on my slide, so let me stick to little p.

So a 1 coming in is transformed in error to a 0 with some small probability, presumably small, with 1 minus p it's intact, and then the same thing on the side. A 0 comes in with some probability, comes out as a 0, but with some probability it actually gets flipped to 1.

Now, we use the word symmetric here to say that we're assuming identical probabilities of going 1 to 0, 0 to 1. You can easily imagine an unsymmetrical channel where you have different probabilities in the two directions, but we'll stick to the symmetric case. Binary, of course, because we're dealing with binary sequences of the two ends. And what we're imagining is that this is a memory-less channel. In other words, I can look at this transmission by transmission.

So a bit comes in, this is how the output is determined. And then the next bit comes in, and it knows nothing. There's no memory in the system, it knows nothing about what decision was made in the previous case. Now you can imagine more complicated channel models with memory, but this is a good starting point. So that's the binary symmetric channel.

So question now, if we wanted to get a bitstream over reliably, any ideas on how we can counteract the effect of this p, probability of flipping? Yeah?

AUDIENCE: You could have like a range [INAUDIBLE].

GEORGE VERGHESE: Well, at this point I'm back to the ones and zeros. There's no signals. The signals are in here. So what you're thinking of maybe is, how do I reliably map bits to signals? And what you're saying is, you can design your signaling here in a way that reduces the p.

The p that I'm thinking of here is the end to end error probability. If I designed the inner part better, I might lower the p. But for a given p, is there something that I could be doing to improve my chances of getting the bit across correctly? Yeah?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: OK, so the suggestion is that we introduce redundancy by just repeating it. So send the 1, repeat the 1. Repeat it many times if you need to. And so what would you suggest that the receiver should do if you do a repetition like this? How should the receiver decide? If I send five ones in a row-- yeah?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: I'm sorry, say that again?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: You're talked about requesting a-- I'm not hearing well what you--

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: OK, so you're using the word request. You're talking about the receiver sending something back to the sender. But if we're with this channel and the sender has to make its own decisions about how to get things across without a possibility of feedback.

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: OK, so I think I understand a bit now. So if you're repeating this, the chance of more than half of them being wrong is very small. I think that was the idea that you had, as well. So repetition is likely to reduce the chances that you go wrong here if you use a majority rule.

Majority rule would be a simple rule. Send five repetitions, and if only two are flipped, well, you just decide in favor of the majority. Because it's more likely that none or one or two are flipped than that three or four or five are flipped. So that's the idea. So this is what's called a replication code, and actually, it can work very well.

So what you see on the horizontal axis is the replication factor. You can replicate it 5 times, 10 times, 15 times, and here is the probability of error. And it actually goes down. And you can do the computation. This is actually a fairly simple computation, you're basically doing coin tosses and seeing what's the probability that more than half of the coins I flip come up one way. And you're counting that to decide how the majority rule works.

I'm sorry, for the epsilon here, it's supposed to be the p. OK, so is this good? Good enough? Yeah?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: The more you send, the more you're wasting time on that one bit. So this is the point, that you can do the replication and reduce your probability of error, but what's the information you're getting across? You're doing all of this to get that one bit across reliably, but you've got a lot of bits backing up, waiting to get across.

So the code rate, the rate at which information gets across is a 1 over n if you're doing n replications. If you're doing n replications, it's only 1 over n. The rate at which you're getting information across in terms of bits is 1 over n. So you're dropping the probability of error, but you're also dropping the transmission rate. So this is really unacceptable.

It turns out, though, that we can do a lot better. We can do a lot better. What I'm going to do now is say a little bit about what Shannon had to say about it. I hope you'll allow me to teach you about something that we're not going to test you on just so you can learn a little bit about this, and then I'll get back to stuff that we will test you on. OK, is that all right?

I know you didn't pay for this, but we'll do it anyway. So here's Shannon, defining something that the thermodynamics people and so on didn't really think to do. They may have done it indirectly, but it didn't arise in where they were working with entropy, and all of that.

Shannon defined something called a mutual information, given by this symbol. X and Y are random variables, random quantities. What we know about H of x is the entropy in X, so it's our uncertainty about X. It's the expected information when you're told something about X. This symbol denotes the uncertainty in X, given information about Y. So it's the conditional entropy here.

And so what this is asking is, how much is our uncertainty about X reduced by having information, having a measurement of Y? That's very relevant to a channel where the input is X and the output is Y. We're saying, we see the output of the channel, we want to infer what happened to the input, what the input sent. The mutual information between these two random variables surely has to be important.

So what's the reduction and uncertainty that results from having a measurement of Y? That's a question of interest not just in communications as such, but in all sorts of inference questions. OK, I'm going to have a slide of equations. They might look scary, but actually they're very simple, given what you already know how to do.

First, I have to define for you what I mean by conditional entropy. So I'm saying it's the entropy of X conditioned on having a particular measurement of Y. So suppose you know that Y takes the value of little y sub j. You use the same formula that we've used for entropy, except your probabilities are all now probabilities conditioned on that information.

So instead of just p of xi, you have p of xi given yj. So it's the same formula. But if you've been given information, then you have to condition on it. So that's the definition. This is the conditional entropy given a specific value for y. But if all I tell you is I'm going to be giving you a value for Y and I haven't told you the value yet, what's the conditional entropy?

Then what you want to do is average over all possible Y's that you might get. So you're going to take this conditional entropy for the given Y, and then take the weighted average of the probabilities. So that's how you compute this quantity, and it's quite straightforward. It's not very different from what you have.

And then if you put in what you know about how joint probabilities and conditional probabilities worked-- this was the definition of conditional probability that we had in, I think, the first lecture-- you discover that actually the joint entropy of these two random variables can be factored in two particular ways. And that allows you to deduce that the mutual information is symmetric.

In other words, the mutual information between X and Y is the same as the mutual information between Y an X. So there's no difference in that. That might be a little surprising given that we were thinking of X as the input to the channel and Y is the output of the channel, but it turns out that that's the case.

So let's actually compute it for the channel that we know. This is the binary symmetric channel. Let's compute the mutual information between the input and output for the binary symmetric channel. So here is the definition IXY. We've just shown that it doesn't matter which order you take it in, and it turns out the computation is easier if you flip the order.

So I'm going to write this as uncertainty in Y minus the uncertainty in Y given the measurement of H-- of X, sorry. So I'm going to compute it in that fashion. What's the uncertainty in Y? Actually, I should probably have said here that, let's assume X takes 0 and 1-- I might be wrong in saying that this doesn't depend on the distribution of the input. Let's assume 0 and 1 are equally likely at the input.

If the 0 and 1 are equally likely at the input, what's the uncertainty in Y? There's a little bit of uncertainty. It's equally likely to be a 0 or 1. I had actually written that assumption in, and then I took it out, but I think I'm wrong in saying that.

So here we have the 1 for the uncertainty in Y. What about the uncertainty in Y given X? So I give you a particular value for X. Let's say X is equal to 1. So I give you a particular value for X. What's the uncertainty in Y? Well, Y is 0 with probability little p, and it's 1 with probability 1 minus p. And that's really the binary entropy function that we had drawn last time.

So you can actually work out all these pieces and discover-- let's see, here is the binary entropy function. Just to remind you, this is for a coin toss. If something can be 1 with probability p, 0 with probability 1 minus p or the other way around, the entropy associated with that is H of p.

And we have 1 minus H of p for the mutual information between the input and output of the binary channel. So here's what 1 minus H of p looks like. All right, so what's a low-noise channel? A low-noise channel is one with a very small value of p. And what this says is that the mutual information between the input and output is on the order of one bit.

So if you're told what Y is, you've got a very good idea what X is. That makes sense, because it's a low-noise channel. But if you get to a channel that has around the 0.5 probability of flipping the bit, then the mutual information is very small. So it doesn't reflect what you'd like to see.

Here's another notion, which is entirely Shannon, which is the idea of channel capacity. And what he's saying now is in order to characterize the channel, rather than the input or the output, let's ask what the maximum mutual information is over all possible distributions that you might have for X. So I'm not going to specify X being 0 and 1 with equal probability.

If you go through that computation, you find that it's exactly the shape that we had before. It's exactly, for the binary symmetric channel, that happens to be exactly this curve. So the channel capacity for the binary channel is exactly-- for the binary symmetric channel-- is exactly this curve. So that gives us an idea of the maximum information that you could be transmitting across the channel.

Now that's just the definition, but it turns out to have some very practical implications for how fast and how accurately you can transmit data on a channel. And here's Shannon's result. What he says is that you can theoretically transmit information at an average rate below the channel capacity with arbitrarily low error.

So that's the shocking thing, that as long as you stay below channel capacity, you can transmit with arbitrary low error. If you try and get to rates above that, you're going to run into trouble. You can't get that probability of error to vanish.

Now, how do you do this? Well, the prescription is take long strings of that input message stream, take k bits of that input message stream, code it onto it n larger than k code words, send that through the channel. If n is very large and k is very large, the rate at which you're transmitting is k over n, you can transmit at a rate k over n that lives below C with as low an error as you want. The way to make the error smaller is to take longer and longer blocks.

This was kind of an existence proof. He didn't actually show you specific instructions, necessarily, in that proof for how to introduce the redundancy to make this happen. But it was actually a result that said, you can't be satisfied with the replication code. You can do a lot better, and how much better you can do is indicated by that channel capacity.

OK, let's come back to testable stuff. We're going to actually design ways to introduce redundancy, motivated by this [? Shannon ?] result, for very practical settings. And a key notion we're going to use is that of the Hamming distance between two strings.

So the Hamming distance-- you've seen this in some recitations-- basically, the Hamming distance between two strings is just a number of positions in which the two strings differ from each other. so the Hamming distance here between these two strings, let's say string 1, string 2, is what, 1, 2, 3. These strings differ in three positions.

Another way to think of it is, how many bits do I have to flip in one to get the other one? So how many hops does it take, in some sense, to get from one to the other? All right, so here's how the notion of adding redundancy comes in. Suppose we have a 0 or a 1 to send across. This is our bit for that transmission.

What we're going to do is actually code it not as 0 and 1, but as 00 and 11. If we've got just a single bit corrupted, we go to something that's not a code word. We go from 11 01 or to 10, or we go from 00 to 01 or 10. We receive something that's not a code word. That allows us to detect that an error was made.

So what we've essentially done is, in Hamming distance, we've introduced some distance between the code words that we're using to transmit on the channel. It takes two hops to get from one code word to the other. There's a Hamming distance of two.

And, therefore, if you only have a single bit error when you transmit on the channel, you're not going to get all the way to another code word. You won't be at any code word you recognize, and you'll know that you made an error. So this is an example of how you start to introduce redundancy in the stream so that you can detect and perhaps even correct errors.

Here's another example. Now this is-- these, by the way, are still looking replication codes because to send a 0, we're repeating 0 three times, and to send a 1, we're repeating 1 one time. But we're going to do more elaborate versions of this that are not replication codes.

But imagine now I've drawn the corners of a cube. Each circle here across an edge is Hamming distance 1 from the adjacent one. Right So to go from the sequence that I'm using to represent to 0 to the sequence that I use to represent a 1, I've got to do 1, 2, 3 hops, there's a Hamming distance of 3 there.

So if I had only a single bit error, is it possible for me to correct? If I know that my errors are limited to single bit errors, can I correct when I receive an incorrect string? If I start with 000 and I have a single bit error, I can only go to these adjacent vertices. And those are not going to be confused with vertices that are one step adjacent to the 111.

So I'll know that I've made an error, and I'll know to correct it back to 000, provided I know that only one error has been made. Now, I might just assume that only one error has been made. And so once in a while, I'll think I'm correcting, but I'll be getting something wrong. And that has to be contended with and calculated, but this is the basic idea.

More generally what we're thinking about is taking key, message bits. So this corresponds to 2 to the k possible messages. We're going to embed this in ended code words where n is greater than k. So if you imagine a generalization of this picture, what we've got is a hypercube with 2 to the n possible nodes, corresponding to all the possible combinations here.

We're going to assign 2 to the k of those nodes to code words, and the rest will be left free to just leave some space between the code words. the rate of the code then, the rate of the code is, how many message bits are you're getting across on average per transmission? And so the rate is going to be k over n. Because for every n bits that you send, you're getting across k bits of information. This k message bits and every n transmitted bits.

So here is the general statement in terms of Hamming distance and what you can do with the code. So first of all, I've got a set of code words. What's really important is what's the minimum Hamming distance between my code words. Because that's the point of greatest vulnerability. That's where I'm most likely to get confused.

So if you give me a set of code words, I can look at the Hamming distance between any two of them. I've got to search all these pairs and find out which is the minimum Hamming distance. That's the point of maximum vulnerability, and this is what we're calling d, little d.

So the picture I like to think about is I've got some valid code word here. I've got some other valid code word here. And if I told you that over all the code words in my set, the minimum Hamming distance I find is 3, what that means is I've got to do three hops to get to the other code word.

This hop means I've changed 1 bit in the valid code word to get to some other sequence, not a code word. And then 1 bit to get to this one, and 1 more bit to get to this one, and this is now another valid code word. OK, so if I say the minimum Hamming distance is 3, that means that you will find a pair of words with these three hops to get you from one to the other.

How many errors can you detect with a code like this? So you send a valid code word across the channel, bits get flipped, up to how many errors could you detect without being fooled? If I tell you this is less than or equal to e errors, how large can e be to guarantee that you won't get a transmission of one valid code word that ends up as another valid code word? Yeah?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Sorry?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: I'm talking not about correction but about detection of an error. How many errors could you detect in this kind of a picture? So I can afford to have one error, two errors, and the third hour will bring me to about valid code word, and I won't know that I've made an error. So if you want to look at how many errors you can detect, it's what's given by the upper one there, d minus 1.

So if the minimum Hamming distance is d, you can detect up to d minus 1 hours. What about correction? How many errors could you correct here? Can you correct any errors here? Up to one, right? And then you can look at this more generally, and you see the general formula is d minus 1 over 2, the floor of that. So that tells you how many error you can correct. So the minimum Hamming distance is actually a key thing.

Now, how do you build codes which have desired characteristics? For instance, suppose you want to send 4 bit messages. So k equals 4. You want to have single error correction. So that means you want this kind of a picture here. You need Hamming distance 3, at least. How will you produce a code?

All right, so this is not an obvious thing at all. Here's an example of one that satisfies the construction. You need to actually expand to sending 7 bits, so n is equal to 7. How many messages do we have? We have 16 messages, so that corresponds to a k of 4, 2 to the 4 is 16.

So we've got 16 different messages. We could have counted those messages with 4 bits, but we're going to add in redundancy to get 7 bits per message, resulting in these code words. These code words, this set of code words has the property that the minimum Hamming distance is 3. So you can correct up to a single error, here.

But it takes-- in principle, it takes a search, and it's not necessarily easy to do. But we'll see how to do that efficiently. All right. Let me show you how-- and this is something that you're probably quite familiar with-- how by making n equals k plus 1, you can already-- suppose I choose n equals k plus 1, which means I'm taking the message bit and adding 1 bit.

We're going to add what's called a parity bit. And you can do this in different ways, but we're going to do-- let's see-- what I'm going to do with this is guarantee that the minimum distance between valid code words is at least 2. So let's see how to do that.

And there'll be some computation in not just the parity calculations, but other stuff we'll do that builds on computations of zeros and ones. The computations are what you've probably seen elsewhere with Boolean algebra. This is what's called computation in Galois field 2. So GF2 is another symbol you'll see. 0 plus 0 is a 0, 1 plus 0 or 0 plus 1 is 1, 1 plus 1 is 0. So this is like an exclusive or addition, and multiplication works in the usual fashion.

So all our computations are with zeros and ones, and you want to keep that in mind as we go through this. So here's what we do for a simple way to add redundancy. We'll take the message and not a single bit to make the total number of ones in the resulting code word even.

So this is what's called even parity. You can have the opposite choice of odd parity. So if you now receive a code word with an odd number of ones, you know you've made a mistake. How do I know that the minimum Hamming distance is 2 in this case?

I have to be able to produce for you some other code word that I've had with two hops. Any ideas there? So I give you a code word, which is the original message word with a parity bit. Can I make two bit flips in that and get a new code word, I mean, a valid code word? Because then I'd have Hamming distance 2, right? Can you think of what to do? Yeah?

AUDIENCE: If you flip a 1 to 0 or 0 to 1, then n equals [INAUDIBLE].

GEORGE VERGHESE: But that's not yet given me-- so the suggestion was flip one of the bits.

AUDIENCE: So then for the second one, you'll either still have an odd number or [? even. ?]

GEORGE VERGHESE: So if you flip, for instance an easy way to see this is flip one of the message bits and flip the party bit, for instance. No that doesn't do it, does it? Because then you've changed two 0's to 2 or two 1's to a 0, and your parity is wrong, then. So you can have a two bit error that ends up not being detected, but all single bit errors will get detected.

That, again, correlates with the fact that we set the minimum Hamming distance at 2. The number of errors you can detect is d minus 1. That's 1. And the number of errors you can correct, well, it's d minus 1 divided by 2, the floor of that, and you can't correct any errors.

All right, now we're going to be building more elaborate codes than parity or replication. These are going to be called linear block codes. And there are different ways to set this up. Here's one way to think of it. If you're comfortable with the matrix multiplication, here's one way to think of it.

And we're going to be using this actually. So if you aren't already comfortable with matrix multiplication, maybe you should get comfortable soon. What we have is a vector here, which has our message in it. So we stick our message in there. This is just a bunch of zeros and ones.

I've got a matrix here, which I call a generator matrix, generator matrix. And this is a matrix of zeros and ones, as well, and so on. So we've got things in there. How do I generate my code words? I just put in my message, carry out this multiplication, and see what I get for a code word.

So for instance, if I my message is 1 and all 0's, what's my code word? Well, I take this and I multiply it all the way through the matrix. Because of the special structure here, all of these are zeros, so the rows below the first one don't matter at all. What I get for a code word is 0101101. In other words, I get the first row of g.

If I had a 1 and a 1 with 00, I'm going to get the sum of the first two rows of g. And all of these computations are done with the modulo 2 arithmetic, so in GF2. So this is one way to think of what a linear code is. Another way to think of it is, every bit in your code word is a linear combination of the bits in the message.

It's just that you have more bits here, so you're taking multiple linear combinations of the bits in the message to get the bits in the code word. So this is a highly structured kind of code. And the key fact about this is that the sum of any two code words is also a code word. And we'll leave you to look at that in recitation.

So it's true that any code word generated this way plus any other code word generated this way will give you a code word generated this way. can, you deduce from that that the code word of all 0's has to be in any linear code? Why is it true that every linear code has to have the all zero code word?

In this instance-- well, you can see it has to have the all zero code word. Why am I doing that?

AUDIENCE: [INAUDIBLE]

GEORGE VERGHESE: Yeah, so clearly from this picture, if your input is all zeros, you've got to have the zero code word. Can you tell me from this statement that the sum of any two code words has to be a code word? Can you deduce from that the all zero code word has to be in there?

AUDIENCE: So that's [INAUDIBLE] subtraction.

GEORGE VERGHESE: Subtraction or addition. So suppose I take a code word and add it to itself. What do I get? In GFW, if I take a code word and add it to itself, I get the all zero code word. So the all zeros has to always be in there. If you don't have the all zeros code word, you know you don't have a linear code.

Now it turns out that for a linear code, it's easy to determine the minimum distance between-- the minimum Hamming distance between words, which we saw was crucial to establishing what the error correction or detection properties were. If you've got a linear code to determine the minimum distance between words, you only have to look for the distance-- the minimum distance between the zero code word and all the other code words.

So it turns out that in a linear code, the minimum distance that you find between any two code words is the same as the minimum distance you'll find between the zero code word and any other code word. Now what's the distance between the zero code word and some other code word? It's just the number of ones in that other code word.

So all you have to do for a linear code to determine the minimum Hamming distance is look at all the non-zero code words and see which one has the minimum number of ones. So let's see, it's not obvious that these are necessarily linear codes, but they turn out to be.

In this particular case, here's a code with n equals 3. We've got only two messages being sent, so what's the value of k? Two messages means k equals 1, because 2 to the k is the number of messages. So n is 3, k is 1, and the minimum Hamming distance is 3, which is the weight of the-- smallest weight you find among the non-zero words.

Here's another instance. This is again a linear code. Well, is it a linear code? Yeah. The minimum weight you see among the non-zero code words is 2, so the Hamming distance is 2. So the way we denote this code is the value of n is 4, because there are four different code words-- sorry, the four different bits here. 4 bits, sorry, in the code words. It's not four different code words.

2 is the value of k, because 2 to the 2-- 4 is the number of messages that you have. And the minimum Hamming distance is 2. So with each of those, you can actually compute the associated rate. And just to wind up, these are not linear codes. How do we know they're not linear codes?

Well, some two of them, and you'll discover that in some instances, you don't get the remaining one in the set. This is the code set that I put up earlier. It turns out to be a linear code. So if I claim that it's a linear code, can you tell me what the minimum Hamming distance is between code words here? 3.

You find a code word here of weight 3, and you don't find any code words-- here also another one-- you don't find any code words of weight less than 3. All right, so this is enough to get you going. We'll quit with this, you'll continue in recitation, and we'll pick it up again in lecture next time. Thank you.