MITOCW | 6. Convolutional codes

PROFESSOR: I wanted to tell you a little bit about the use of digital communication schemes in the space program. And part of that is it wasn't just the use. A lot of coding theory was developed for use in this program. So in the early days, there was no error control coding. So they had very slow transmission rates and tried to compensate for not having error control coding by taking a long time to send a bit over. But in later years, with the Mariner and Viking probes, they started to use error control codes.

And linear block codes are what we're talking about. This would be the typical parameters for such a code. So we know how to read this. This is 32 bits per block. Six data bits. And minimum timing distance of 16. A particular kind of code called a bar orthogonal code or Hadamard code, which had specific characteristics and specific symmetries that actually helped with the decoding.

So for instance, on Mariner Nine, it's 1971. This went into Mars' orbit. And the code was used to encode the picture transmissions. So each data word was six bits to encode 64 gray levels and a picture. It turned out that, because of transmission issues, the safe number of bits for a block was 30 bits. And after that, you had to do a little bit of realigning or tweaking. So you could send 30 bits at a time safely. And so, that was a choice of n in that vicinity was a natural choice.

One thing you could have thought to do would be take the six bits and repeat them five times in the 30 bit window. And that would be a repetition code. It turned out that, with this particular Hadamard code, you could actually get the same data rates or comparable data rates. Let's see. What would the data rate be? It would be k over n, right? 6 over 32. But with much better error correction properties.

So let's see. How many errors could you correct in this code per block? Somebody? Seven. Yeah. Because you've got a minimum Hamming distance of 16. So you want D minus 1 over 2, the floor of that. So you could correct up to seven errors per block. And this code was actually used on space probes right into the 80s. And as I mentioned, this particular code has various symmetries that allow actually something called the Fast Fourier Transform to be used in the decoding. And so, that's really what drove this.

As you read about these probes, it's actually staggering how much they did with so little. Let's see. This thing went half a billion miles almost. It had an onboard computer with a memory of 512 words. So you can imagine the kind of engineering that went into organizing all of this. The transmitters-- and this is typical of these space probes-- you don't have a lot of energy generated from your solar panels necessarily. So 20 watt transmitters. So these have to transmit over this kind of distance the data that you want to send in the presence of noise and various other errors. OK?

So quite an engineering feat. Now the kinds of pictures that you would get? Well, these are pretty amazing actually, considering what the probes had to do. So over the lifetime, it sent over 7,000 images. Mariner 9 is still orbiting Mars, from what I understand. It's not sending back. It stopped sending back transmissions one or two years after this. But it's still in orbit until it slows down enough to crash in.

OK. So as I said, you're typically talking about low power. 20 watts. WNBR, what's a typical radio station power on a college campus? They advertise something on the order of 700 watts for their transmitter. So we're talking about doing a lot with a little here. A lot of the art is in the antenna. So you have an antenna that directs this power very sharply towards the intended receiver. But the more sharply you try to direct that, the bigger of a control problem you have. Because you've got to point that antenna all that more carefully. So all of these are coupled issues.

And then at the receiver end, you've got very high quality amplifiers and signal processing. But the data coding and error correction schemes are a key part of that. And it turns out that, as you got more ambitious with these transmissions, you had to go to more complicated codes. And these are the codes we're going to talk about today, what are called convolutional codes. We'll talk about the coding today. And then we'll talk about the decoding with what's called the Viterbi algorithm next lecture.

So this has been used extensively from late 1970s onwards. More recently, you have codes that are actually combinations of convolutional codes, what are called Turbo codes. And another family of codes, low density parity chat codes, which were developed in Bob Gallagher's PhD thesis here. Bob Gallagher's on our faculty. But convolutional codes were really a workhorse of the whole system.

OK. So an example is now Cassini, which is in orbit around Saturn. It's actively sending pictures. This, if I read the website correctly, is a picture from August 29. And I saw other pictures posted from June and July. So this is a picture of one of Saturn's moons. And you can see the rings and the shadows of the rings and so on. This is actually recreated in natural color from multiple images. This, I guess, is part picture and part artist's rendition. But that shows you what Cassini looks like. There's only one of them out there. I don't think there's something else to photograph Cassini.

So the kind of code that's used is a convolutional code. We'll learn what these parameters mean, how they enter into the definition of the code. And here is a typical code rate. You're talking about something on the order of 83,000 bits per second as the code rate here. Sorry, not the code rate. This is the data rate. OK? So the messages are coming. Let's see. You're sending six times this amount per second. But this is the rate at which the data is coming in. OK?

So convolutional codes. And again, I keep coming back to MIT names. Peter Elias was on our faculty here. He was a department head for a while. And in a short paper in 1955, he invented the idea of convolutional codes. So the idea here is not to divide up your data into blocks but to actually work on the streaming data. And as the data goes past, you generate parity bets at a regular rate. And what you transmit in most typical schemes are just the parity bits. You don't send the message bits. So this would be a non-systematic code, if you like. So there's no part of that message that's directly observing the message bits.

Now, you will actually generate and send multiple parity bids. So you'll have a message sequence, x zero xn. Sorry, x zero, x one. And from this, you derive parity bits. And you do that using the standard sorts of equations we've seen with block codes. Each parity bid here, for instance, parity bit zero at a time n will be some linear combination of message bits. But it's the message bits as they're streaming by. So you might have, for instance, this as your choice for parity but number zero.

OK. And then parity bit number one could be some other combination here. So for instance, xn plus x n minus 2, for instance. OK. So it's a linear combination of some set of messages, just the way we've been generating parity bits all along. The plus here, of course, is we're talking about binary messages. So this is addition in GF2. So it's exclusive or Modulo 2 addition. And you can imagine a whole bunch of such parity bits.

So in general, you would have R such parity bits computed off some set of message bits and transmitted instead of the message bits. So you might have, for each message been coming in, you might actually be sending out R parity bits. So what you do is just send these out in sequence. You'd send out the P zero value, the P1 value, time n. Then recompute a time n plus 1 and keep going. All right? Well, actually I have them here. I didn't see that.

So all this happens on a sliding window. This happens for a particular choice of n. And then it happens for the next choice of n and the next choice of n. So you're doing this on the fly with a streaming sequence. So let me just put up an equation that explains why this is called a convolutional code. It turns out the expressions of this type where you take a data stream coming in and generate new data streams of this form, it turns out that the operation that's being carried out here is something referred to as convolution.

So in general, what has P zero then? It's some weighted combination of x at the current time, x1 times step back, x2 time steps back. In general, k different values involved. So what I have is a P zero n being a summation from, let's say, J equals zero. So G zero J x of n minus J. All right? So this is just some set of numbers. Zero, one. Just as these bits are, these are zero, one. But this is the general form.

This particular kind of combination is referred to as a convolution operation on the input stream. And we'll see much more of this when we come later to modeling channels, the physical channels. We'll talk about convolution type models. So here, it's not so important that you must have this expression. We'll have plenty of opportunity to work with expressions like this. This is just for you to know that an expression of this type, wherever you see a summation with indices that are in this form, this is referred to as a convolution. OK?

So it's convolution of the message stream with some set of weights.

AUDIENCE: Professor?

PROFESSOR: Yeah?

AUDIENCE: What does G stand for?

PROFESSOR: The G is just a set of weights here. So in this particular case, for parity expression zero, G zero of zero would be 1. G zero of 1 would be 1. G zero of 2 would be 1.

AUDIENCE: OK.

PROFESSOR: It's just a set of weights. So yeah. This expression is a bit of overkill for the kind of use we're making of it. But it's just to explain the origin of the name. It turns out later, when we use it for channel modeling, the x's will not just be zeros or ones. They could take arbitrary real values. And the Gs could take arbitrary real values. So we'll be working with much more elaborate versions of this. OK?

The number k is referred to as the constraint length. And it's the maximum number of message bits involved when you look over all your parity expressions. So in this particular instance, k would be equal to 3. Right? It's the maximum window of data that you're using in a non-trivial way to generate the bits. So here, you are using up to 3 to generate this. Well, in this case also, you're using a window of three message bits. It happens that you're ignoring the one in the center. But the constraint length is the length of message that you're actually looking at.

OK. So in some sense, if you want to think of it this way, the number of parity expressions that you use? Well, that's straightforward. That's just telling you how much redundancy you're willing to put in. Whereas the constraint length is telling you how deeply you're folding that redundancy into the message. So the bigger the constraint length, the more message bits are involved in generating a parity bit. And so, the more you're scrambling up the message and spreading it over a large section of what's transmitted. And so, you might expect that you get a better error correction properties with larger constraint links. OK?

OK. This is not saying anything new. So how do we come to actually transmitting? Well, we generate the parity bits. And then, as I said, you send all the parity bits associated with your computation at time zero. Then all the parity bits associated with the computation at time 1, time 2, and so on. So in the case of the code used on the Cassini probe, that's 1 over 6 rate code. It's actually computing six parity expressions. So it's transmitting six parity bits for each message bit that comes in. What happens then at the next time instant is that you shift everything up by one. And we do the whole thing. OK?

Now, you can actually-- and I'll have this up on the slides-- you can actually crank through the equations. But it's not the most illuminating way to think of things. It's much easier to think of it visually through a block diagram of this type and using the idea of what's called a shift register. So what is a shift register? You may have encountered it in other places.

So we think of a shift register as, it's basically a box that can remember something. OK? That's the register part of it. A register is something that remembers a number. You've got some input stream that comes in and some output stream emerging. At any time, this stores a particular number which is available to the output. So whatever is stored in the register is available to the output. The shift part of this description is that whatever is of the input will get shifted into at the next clock cycle or the next time instance. OK? So the input gets shifted in at the next clock cycle. Whatever is in here is remembered for that one clock cycle and is available at the output. Right?

So if I have a sequence x n being fed in four n zero, one, two, three, and so on, if I'm seeing x n here, what must have gone into the previous time? If I see x n here at time n, if I'm seeing a particular input at time n, what must have gone in the previous time as x n minus 1? So what's sitting here is x minus 1. Right? And x n minus 1 is available to me at the output. The next clock cycle, the next input comes along. The xn goes in here. And the whole thing shifts. All right?

Now what you have up there is a cascade of shift registers. You've got some shift registers. So keep in mind, the operation that I described-- if this is xn, if I'm looking at this time at time n, xn sitting here-- what must be in this shift register is the input of the previous time. So that's x n minus 1. These are shown adjacent. What we really mean is that one shift register is feeding into the next one. They're just shown as adjacent. But what must be sitting here then is x n minus 2. And if I read off something from here, what I'm looking at is x n minus 1. Namely, what's sitting in the register. What I'm looking at here is x n minus 2.

All right. So do you see how this is working now? This is actually the same example that I had written up earlier, I guess, for the computational parity bit. So here's-- except it's-- yeah. It's the same one. P zero n. Maybe I have the equations. Let's see if I can display them for you. No, I can't. OK. So what's P zero n? P zero n is xn, that's connecting from here, plus x n minus 1 plus x n minus 2.

Again, by the way, in this diagram, what I showed as an arrow coming from the output of the shift register is just a shorthand here that shows the arrow coming out from the body of the register. It's the same thing we're talking about. OK? So P zero of n is the sum of these three message bits. So we're talking one constraint length three here. And what about P1 n? It's xn plus x n minus 2 with nothing of xn minus 1. All right? So imagine this being the picture for every n. So you start off at time zero and keep going. Right?

We refer to the state of the shift registers as the pair of numbers that we find in here. So if we're talking about x's that can be zeros or ones, the shift register combination here can be in one of four states. Right? Zero, zero. Zero, 1. 1, zero. Or 1, 1. So four states. So here's a four state shift register into which we're feeding in the stream. And what gets put out on the channel are these parity bits interleaved. That clear enough? OK.

Nothing I haven't said here. Right? So let's actually work through an example step by step. This is clear enough. But let's just see it concretely. Let's assume that I'm starting out with the shift registers in the zero state. And now, I've got this message sequence coming in that I want to send out. OK? So the sequence is 1, zero, 1, 1. So the first bit that appears here is the 1. And I've got to generate P zero and P1. Well, P zero is the exclusive OR of these three things. So it's 1. P1 is the exclusive OR of the first and the last. So it's again 1. So that defines P zero and P1 at time n.

The same way at the next time instant, the next message input bit comes in. So we had 1, zero, 1, 1. We took care of the 1 here. Now comes a zero. We do the same thing. So the exclusive OR of all three of them appears here. That's the 1. The exclusive OR of the first and the last appears there. And that's the zero. So you can see how things are getting folded together because the input that was here before is now sitting in here and plays a role in generation of the parity bit for the next step.

In fact, the word convolve means to fold together. And this is what it's actually trying to capture. You're folding together these two sets of weights, the weights on the top tier and the input sequence weight. And then the next two cases, similarly. OK. And that's what gets sent out at the bottom. So this is the transmitted sequence. So it's the 11100001. Right? That's all there is to it. The implementation of the shift register is very easy. And so, this is actually a very straightforward thing to implement.

Now there's another viewpoint that's also very useful here. Another way to look at what's going on, which is thinking in terms of the state of the register and how you move between the states. I guess, how many here are 004? Are those are the ones with smiles on their faces? OK. You see a lot of this there, I imagine. OK. So how do I read a diagram like this?

I've got a circle for each state that the shift register can be in. So the shift register can be in zero, zero. Zero, 1. 1, zero. 1, 1. Right? Each of these arcs represents a transition from one state to another. So let me ask you this. What does it take-- if I'm in the zero, zero state with my shift register-- so what you've got in the picture is your shift register sitting there with zero, zero. What does it take for me to get to the 1, zero state? What must my input have been to get to the 1, zero state?

Imagine how these shift registers operate. Right? If I'm going to get from zero, zero to 1, zero, I must have fed in a 1 at the previous time instance. So it takes an input of 1 to go from zero, zero to the 1, zero. So to go from zero, zero to 1, zero, use an input of 1. That's the number that we write before the slash. That's our labeling convention for the arcs. We put the input that it takes to make that transition. And then after the slash, we put the parity bits that are omitted. So what we've got for the 1, 1 is the parity bits that are omitted when you've got input 1 sitting here, zero, zero here, and you're using the parity computation that I had before.

Let's see here. So P zero is going to be x n plus x n minus 1 plus x n minus 2. So that gives you 1. And what about P1? P1 is x n plus xn minus 2. So that gives you another one. OK? So if you're in state zero-- the zero, zero state-- and you get an input of 1, you're going to transition to 1, zero. And you're going to omit 1, 1. OK. So the state diagram captures all that.

And similarly, all the way around. So I haven't checked each of these. But I hope there are no mistakes in it. But if you're in 1, zero? Oh well. By the way, if you're in zero, zero, there's no way to get to zero, 1. Right? So you don't see any arc from zero, zero to zero, 1. If you're in 1, zero, you can get to 1, 1. Or you can get to zero, 1 depending on what you feed in. OK? So it's very straightforward, then, to actually build out this diagram. Why don't we do a little bit more on here?

OK. So if I'm actually abstracting from the shift register picture to something that's more like the state picture, I'm going to say, here are my four states. I've just drawn it a little differently than I have in the upper picture. Instead of circles with these states in them, I prefer to think of them this way. So what we said is, if you get an input of 1, you limit 1, 1. And you'll get to that state. What does it take to get to the state? Somebody? Can I have a hand and a loud voice? Yeah?

AUDIENCE: Input zero.

PROFESSOR: OK. And then I guess you've got to go back to this to think about what's happening. So I'll allow you to think of a zero sitting at the input here. So what would the parity bits be? So the first parity bit will be the exclusive OR of the zero, 1, and zero. So it's going to give you a 1. Right? And then the next parity bit is going to be exclusive OR of what's here and there. So that's going to be a zero. I hope that matches with what I have upstairs. We're talking about going from 1, zero to zero, 1. It takes a zero input to do that. And what you omit is 1, zero. Right? So you can fill in all of these. This is the state transition diagram.

OK. Let's see. We say that, if you've got a constraint length of three then, of k equals three, for instance, or let's say if you've got a constraint length of k, you've got 2 to the k minus 1 states. well, that's because, in that constraint length, one of the bits involved is the input bit. That's not sitting in the shift registers. So you've got k minus 1 bits left over. So your shift register is k minus 1 stages long. And so, you've got 2 to the k minus 1 states. All right. So you could imagine generalizing this to more complicated sorts of situations.

Let's see. Just going back to the Cassini example, if you let me jump back a bit, there was a k there. What was it? k of 15. OK. So for Cassini, you're using one input bit and 14 more bits in your register. OK? So you've got 2 to the 14 possible states there. So in these codes, you're actually using very large constraint lengths.

OK. All right. I want to go from the state machine view to another view now, which is what's called-- so this is the state machine view-- to something called the trellis view. This is something-- by the way, there was a way of looking at things that was developed by someone else who was on our faculty, David Forni. In fact, if you visit his home, you'll see his garden. There's a nice trellis around it. And you'll see why when we draw this.

OK. So what's the trellis view? The trellis view says take the state machine but unfold it in time so that all your transitions over time are not happening here. At every time step, you draw the picture again and look to see where you get to. So let's do this. This is the one I want to be most careful with and where I'll introduce a few notational conventions so that our later life is simplified.

OK. So we've got state zero, zero. State zero, 1. State 1, zero. And state 1, 1. OK. Except, though, this is going to be the picture that I have at time-- let's say-- at time n equals zero. At time n equals 1, well, I've got the same shift registers. I'm going to draw this picture again. The easiest way to learn this is to just follow through one example. So please keep your attention here. And you'll have it sorted out. And then you won't have to worry about it again. It's the same thing as with LZW.

All right. So it looks kind of detailed. Maybe tedious. But it's actually very simple. Just hang in there and follow through one example. OK. So what does this say? At time n equals zero, I'm in zero, zero. Suppose I get the input zero. Suppose the input is zero. What state do I transition to? Here. Right? So if I have an input zero, I'm going to transition here. So this is with an input of zero. And what are my parity bits going to be? Both zeros. Right?

What about if I get an input of one? Where do I transition to? Well, we've already seen that here. If I get an input of 1, I'm going to transition to here. And what am I going to omit? Well, we've already calculated that. We're going to omit a 1, 1. Right? Let's do it for one more case. We're in zero, 1. What states can I transition to? I could go to zero, zero. And I would do that if my input was zero. Right? And what would my parity bits be? Well, that's another case for us to look at. If our input is zero and we're in state zero, 1, what would the parity bits be?

For this choice of parity bits, depends on what specific choice you made, of course. 1, 1? Do you agree? And if I get an input of 1 instead, where do I go? If I get an input of 1, I'm going to go to 1, zero, which is here. OK? So if I had an input of 1, I would go to 1, zero. My parity bits would be-- what would they be? Can I have a hand and a voice? Yeah. Zero, zero. Right.

OK. So it's that simple. That's all you have to do. Fill out this picture and you're seeing what this picture translates to at the next time instant. We're not using anything more than is in the state transition-- sorry-- in the state machine diagram. But we're unfolding things in time, which is actually very helpful. Now, this is a simplification we'll make in drawing this. Because I've arranged the states in natural binary counting order-- zero zero, zero 1, 1 zero, and 1, 1-- it's always the case that the upper arrow that emanates from a state corresponds to the input of zero. And the lower corresponds to an input of one. OK?

So I don't really need that first thing before the dash. I'm just going to dispense with it. So if you're going up, of the two choices that you have when you come out of a box, if you're going up, it's zero. The input is zero. And if you're going down, the input is 1. So I'm just going to label that as zero, zero. OK? I'm going to label this as 1, 1. And I guess I've forgotten already what some of these are. But you can see what the whole picture starts to look like. OK?

So let me actually, I'm not going to do these in detail. But let's just see how the next stage would differ, if at all. When I come to n equals 2, well, it's the same story all over again. So whatever pattern of arrows I had coming out of here, I have the same pattern at the next stage with the same labels. Right? Because there's nothing different.

So if you'll allow me, let me actually fill out a few of these. And you'll get practice drawing one of these when you do recitation maybe for another example. So I can keep going with these. Let's see here. This is going to be 1, 1. If I haven't found two arrows coming out of each box, then I'm not done. Oh, this is wrong. Right? Thank you. From zero, 1, I can go to 1, zero.

OK. So the two arrows coming out of each one, the upper arrow corresponds to the input having been zero. The lower arrow corresponds to an input having been 1. And there are two arrows going into each box, as well, corresponding to whether the bit that's going to get dropped off is a zero or a 1. All right? So there's a real symmetry to this. I'll draw one more stage just a little bit to make a point here.

OK. So you can keep going. So how do you generate a code word from a trellis diagram? You're starting in some state. Typically, it's the all zero state. In fact, what you'll usually do is have a header for your message stream which is all zero. So you force the shift register to be in the zero state once the real message bits come in. And then you move from here. So you're typically starting here. And then you navigate, depending on whether you've got a zero or a 1.

So if the first message bit is a 1, you're going to go down here. If the next message bit is a zero, you're going to go up here. If the next message bit is a zero, you're going to go up there. And the code word that you omit is going to be, in that case, 1, 1, 1, zero, 1, 1. And then all zeros. Right? Assuming you're staying at zero from then on. So depending on what the message sequence is, you can actually go through the trellis. It's infinitely long, or as long as your message sequence is. And figure out what the code word is that's omitted. So this is actually just a graphical way of displaying code words.

So the set of code words that I get, does that correspond to a linear code? Let's assume that, somewhere downstream, all these things come back down to zero, zero. OK? So I'm only considering a finite window of things. It's not going to go on forever. So suppose I'm going to end my input messages with zero, zero at the end and come back down to that state. So my messages will always start with zero, zero to force the register to the zero, zero state. And they'll end with zero, zero. OK?

The set of possible code words is a set of parity bits I omit along the way as I navigate through the trellis. Is the set of code words, does the set of code words constitute a linear code? That's the question. Maybe not obvious. Right? The way you answer that is actually thinking back to this setting. So one particular code word would correspond to a particular input sequence that generated it. A particular data message sequence that generated it. Another code would correspond to another message sequence. And the question is, is there a message sequence that would generate the sum of these two code words that you have?

And actually, it turns out that the answer is yes because these parity relationships are based on a nice linear operation. OK? So it turns out that the set of code words that you generate constitutes a linear code. So if you were going to think of a minimum Hamming distance for this code, what would you want to be thinking of? I don't know if I've actually drawn this correctly right now. Has anyone spotted any errors along the way? Or do I have it right? Seems to be OK.

What would, how would you look for a minimum Hamming distance in the set of code words generated over this window?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Sorry? I didn't hear where that came from. Yeah. Can you speak up? The minimum number of ones in a non-zero code word. Right? So it would be the weight, the minimum weight code word you'd find among all the number of code words. So you would have to find a path starting here. All my code words are going to start here and there. You have to find a path through this that picks up the minimum number of ones. A path that's different from the all zeros path.

OK? Find a path through there which has the minimum number of ones in it and the code word. So what would that be? In this particular case, maybe this path we highlighted in another color. I don't have-- this is not a proof. This is just a suggestion that this might be it. And you would have to draw in all the paths, explore all the other paths. But what would be the minimum weight along this one? Let's see. I've got 1, 1 there. 1, zero here. 1, 1 there. So I would get a weight of 5 on this path.

Now the question is whether you could find another path with a smaller number of ones attached to the code word. And I think if you work this out in detail on [INAUDIBLE], that you'll find that you're actually stuck with 5. OK? Now it turns out that the interpretation of this number is not quite as straightforward as the interpretation of the minimum Hamming distance in block codes. And the reason is that actually this is a more complicated kind of picture because it continues on with the structure. So we don't actually call it the minimum Hamming distance. We call it the free distance. So I'm just trying to evoke this.

So the minimum weight code word you find among the non-zero code words will indeed be a code word of eight 5. But the interpretation of that number may not be directly as simple as in the case of the Hamming distance. But it's close. OK? So what it really tells you is that, over a data length that is maybe not much longer than this, the code words that you have are separated by this distance minimum. So you might expect that you could correct two bit errors over data lengths that correspond to code words that are somewhat longer than this perhaps. OK?

So that's all very hand-wavy. But that's all we're going to do with the notion of free distance. So this is more complicated to deal with than a block code. But the free distance actually has that kind of intuition. It has the intuition of minimum Hamming distance locally over this window of data. Even if this went on for thousands of bits, if you got a burst of errors in this stretch that had up to two errors, you could correct them. Now we haven't talked about decoding. We're going to talk about that the next time.

OK. So that answers this piece. Now let me say one thing about decoding, just to set us up for next time. If I didn't have any noise in my channel, it actually turns out that decoding is pretty trivial. How is that? If I gave you the sequence of parity bits, can you think of a way that you could recover the input sequence? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Good. Yeah. You see, if I add these two, I get x n minus 1 equals P zero n plus P1 n. So if you give me the parity bit stream, I can reconstruct for you exactly the input with a one step delay. That's pretty good. I'm happy with that. If it's taken me minutes for the signal to reach me from Saturn, I'm happy with a one step delay here in decoding. All right?

So in the absence of noise, the inversion is simple. The inversion meaning deducing the input message bits from the output, the parity bits. And this is a theme you'll see in many other settings. If there's no noise, inversion is easy. You can look at the output of a system and figure out what the input was. If you know exactly how the system was creating the output from the input. But in the presence of noise, you've got a problem. Because you see, if you have these parity bits corrupted at some rate-- every few bits, you've got errors-- well, you're interpreted message is going to have that same kind of error rate.

OK? So it's really, in the presence of noise, it's an unsatisfactory way to do it. So this doesn't work. We'll be looking at something more careful next time. OK? So well actually, since I have you here, let me put up the spot quiz. We haven't quite hit the mark. So can you answer these for me? What's the constraint length of this code? Anyone? Who hasn't answered? Yeah? 4. Right? Because you've got xn and xn minus 1, xn minus 2, xn minus 3. That's the largest window over which you're picking things.

What about the code rate? 1 over 3? Right? Because for every message bit, you're generating three parity bits. You're going to shift out those three parity bits before you do the shifting on the shift register. So the code rate is 1 over 3. The coefficients of the generators, of course, you can read up there. What about the number of states in the state machine here?

What? 8. Right? Because constraint length is 4. But one of those is the input. So you've got three bits that you're storing in memory. 2 to the 3 is 8. So the number of states in the machine is 8. OK? So more complicated picture than this. But the same principle. All right. We'll continue next time.