PROFESSOR: --we'll start off with a little review, so you might chance it. We're in the middle of the chapter nine on convolutional codes. Today, I intend to hand out problem set six. I believe Ashish will have it when he gets here. And we'll hand it out at the end.

Last time, I introduced you to a lot of things about convolutional codes and their encoders. I stressed that a convolutional encoder had two distinct characters. One is it's a linear time-invariant system. So we can use the linearity and time-invariance and algebraic analysis of convolutional codes. Secondly, I stressed that's it's a finite state system, because it's a finite memory system where each memory element has a finite number of possible values, over F2 or indeed over any finite field.

And therefore, we can use that for different kinds of analysis. We use that, for instance, in conjunction with linear property to find the minimum distance to a particular code. And we'll see that that's the key to getting an efficient optimal maximum likelihood decoder -- is the fact that we only have a finite number of states in this system. All right, so along the way I kind of went all over the map, as far as the introducing various algebraic quantities.

I focused particularly on formal Laurent series. And I want to just give you a little table to make sure you have fixed in your mind what all these different kinds of sequences we're talking about are. On the left side, I have Laurent sequences. These are semi-infinite sequences that must start at some time. Their delay is not equal to minus infinity, which would be if it were all 1's forever. But their delay is some finite time. It could be negative, could be positive. That's their starting time.

And then they could go on, possibly, infinitely into the future. And then, among those, we can look particularly at the ones that start at time 0 or later, whose delay is 0, or whose delay is non-negative. These are called the formal power series, and you've probably encountered them earlier. They're a lot more frequently encountered in mathematics algebra.

The rational sequences are simply those that can be written in a very finite way as a numerator polynomial over a denominator polynomial. Because the formal Laurent series form a field, every element is invertible, and, in particular, every polynomial has an inverse, which, unless the polynomial is a monomial, is going to run on for an infinite length of time.

But nonetheless, there is a subset, a countable subset of these sequences that we can write in this way. And we call these the rational sequences, and they're analogous to the rational numbers in the real field. And their particular characteristic, as we showed, is that they're eventually periodic. And it's easy to see that these, too, form a field.

Obviously, the inverse of N of d over D of d is D of d over N of d. Provided that n of d is not equal to 0, the denominator is always required to be non-zero for a rational sequence. So that's a very nice subset. It may or may not have operational meaning. Certainly, when we're talking about realizations, it has meaning.

And then the finite sequences, by eventually periodic, I will from now on include the finite case. If we have a finite sequence, then its end is all 0's, and that's certainly a periodic sequence. So I would say a finite sequence is also eventually periodic. But among the rational sequences we have, particularly the finite sequence, these do not form a field, because the non-monomial polynomials cannot be inverted. So these are merely a ring. However, D has an inverse.

Over here on this side, D minus 1 has an element of all these things. D has an inverse. Over here, on the causal side, these are the causal analogs to all of these. In the case of causal rational, we don't have particularly good notation. We just say these are the ones that are both causal, formal power series and rational. But they are important, because we've seen that it's this kind of sequence that you can get as the impulse response of a time-invariant linear system that is realizable.

Realizable has two parts. One is the impulse response must start at time 0 or later, by physical causality. Two, the impulse response must be eventually periodic, therefore rational, if it's going to be realizable with a finite number of memory elements. So we include that in our definition of realizability. So these are sometimes called the realizable sequences, or the realizable D transforms.

And these, of course, the causal finite sequences are the polynomials. These include the polynomials, which are trivially easy to realize with shift registers, as we saw in our example. So that's just a review of all these quantities. Does anyone feel the need for further elaboration or explanation? Or can we go forward with this? We tend to use all of them as we go along, so I want you to have them clearly in mind. Yeah.

AUDIENCE: Why was the [INAUDIBLE] uncountable? What was the [INAUDIBLE]?

PROFESSOR: Why is that uncountable? Try to count it. I could put it in one-to-one correspondence with a set of all binary expansions of the real numbers, and that's an uncountable set. Some of these side comments are just to trigger analogies in your mind, maybe make it more reasonable, what we're talking about.

OK, so next. Just again, to continue a quick review, we talked about realizable linear time-invariant systems. First, we talked about single input, single output. I'm imposing an order that didn't necessarily exist. In any case, these are characterized by impulse response, g of D, which, for realizability, that implies that g of D is causal rational. The fact that it's rational means that we can write g of D as n of D over d of D. And the fact that it's causal, well, we would always reduce this to lowest terms in the first place, so there'd be no point carrying along common factors in the numerator and denominator.

By inserting enough powers of D, we can make sure that both of these are polynomials. So if n of D, d of D are -- we can ensure that they're actually polynomials, not just finite sequences. And if it's going to be causal, that means that we can always -- if we remove common factors of D, that means that the numerator might still have factors of D in it, but the denominator can't have factors of D in it. You see that?

Because, once we shift all these over as close to 0 as we can, the denominator has to be closer to 0. If D0 equals 1, that means 1 over d of D is something that starts at times 0. We multiply times the numerator. In order to have the whole thing be causal, the numerator has to be causal, because a point I didn't mention, when you multiply formal Laurent sequences, their delays add.

If a of D and b of D are Laurent, then the delay of a of D times b of D, defined as a convolution, is going to be the sum of the delays of each of the component sequence. You just take the lower order term on both and that's what determines the lower order term on the convolution, just like in polynomials and analogous to degrees. So from this, we find that we can always write a causal rational sequence in this form, if it isn't 0. If it is 0, we just take n of D to be 0, and d of D to be 1. And that satisfies this form, too.

I guess I didn't even have to make that comment. We're already in that form. You can convince yourself this is a unique way to write a causal rational sequence. And every other way is just multiplying the numerator and the denominator by some common polynomial or finite sequence.

So now we had a little theorem that says that, if g of D is causal rational, and therefore equal to n of D over d of D in this form, then realizable with how many memory elements? Does anyone remember? nu equals the maximum of the degrees. And I debated whether to actually do this in class, and I think it's worth taking a few minutes to actually do this in class. It'll also appear as the first homework problem.

So this'll be a little head start on the homework. So anybody have any ideas how we can do this realization? What we want is a circuit with an input, u of D. Eventually, an output, y of D equals u of D times n of D over d of D. For these are both polynomials and D0 equals 1.

A lot of you have probably seen this in discrete-time linear filters. It's going to be just the same technique. But if we don't have volunteers, I don't want to waste time. The key is to use -- we're of course going to need feedback in general. To have a d of D, in general, will not be 1. If it's not 1, then we're going to get an infinite impulse response. To get an infinite impulse response, we need feedback. So we're going to need some feedback term in here.

And here is the motivating idea of the construction. We want to create something here, v of D, which is basically equal to u of D over d of D. If we can do that, we'll be done. I'll show you why. So we created a different sequence, which is u of D divided by d of D. And then we pass that sequence through a shift register.

And what do we get at the various stages of the shift register? Here we would get v of D delayed by one time unit, v of D delayed by two time units, and so forth, up to -- let's make this nu. I want to realize it in nu memory elements, where nu is the maximum degree of either of these defining polynomials. So finally, out here, I get d nu of D, v of D.

Now, what's the trick?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Excuse me?

AUDIENCE: Because [UNINTELLIGIBLE PHRASE].

PROFESSOR: That is the idea. What do we get if we take out these various lines, which are v of D, through d to the nu v of D, and we make a linear combination of them? We're going to do this twice, actually, once to form the feedback, and once to form the output. For the feedback, let me not take this into the linear combination. Otherwise, I'd get a loop without a delay element in that, and that tends not to be well-defined.

So I want to take a linear combination. In general, by taking a linear combination of these things, I can get the multiple of v of D times any polynomial degree, nu or less, just by taking the coefficients of that polynomial as my linear combination.

Do you see that? I'm not seeing people's --

AUDIENCE: [INAUDIBLE].

PROFESSOR: Excuse me?

AUDIENCE: In the linear combination [UNINTELLIGIBLE PHRASE] you also take the one with zero derivative.

PROFESSOR: You're getting ahead of me, which is fine, but I'll ask for that comment in just a second. Let me first create the output. Suppose I had v of D equals u of D over d of D. Then to get the output, I simply want a linear combination, which will give me n of D times u of D over d of D. And that would be the correct output.

And since n of D is a polynomial degree less than nu, I can do that by simply taking n nu times this, n nu minus 1 times that, n2 times that, n1 times that, and 0 times that. And that will give me what I want as an output, if I'm successful in getting v of D of this form here. So let me try a similar trick up here. The trick is, I force d of D to start off with a 1, to have D0 equal 1. So what I'm going to create up here is the linear combination, d of D minus 1 times v of D.

First of all, let's verify that I can do that. d of D minus 1 is what? It's a polynomial. Its degree is the same as the degree of d of D. Now, if d of D is equal to 1, if the denominator is just 1, this whole thing falls out. I don't need anything coming in here. d of D is 1. v of D is equal to u of D, so that's the polynomial case where I don't need feedback.

So let's assume d of D is not equal 1, therefore it's some polynomial of degree nu, degree of d of D, which is less than or equal to nu, and so is this. But by subtracting out the 1, I have no constant term. So it's going to be a multiple of D. That's another way of saying that. It's going to have no constant terms, so I only need to form a linear combination of these terms. So I can do it.

Now, if I do that, let's just solve this equation. I create x of D. This is supposed to be a plus. This is supposed to be a minus, to work over any field. This trick works over real or complex, or what you'd like. x of D is u of d minus d of D minus 1, times v of d, times x of D. Maybe I should just call this v of d. Now I'm going to solve for v of D.

So I actually have a v of D on both sides of the equation. This results in u of D equal v of D times d of D. And dividing both sides by d of D, I see that I succeeded in getting what I wanted. So this is a typical trick of realizing a rational impulse response by including a feedback loop and negative feedback in your system.

This is done in the notes. I'm surprised if you haven't seen this before. Maybe it's that people in communications don't take circuits courses, or they don't take digital signal processing courses. Most people look puzzled. In any case, can you agree that, formally, I've shown that we get what we want here? So I've given a way of realizing, given a causal rational function, where nu is defined is the maximum degree of these two polynomials.

There's a realization with nu memory elements. A little bit more work, we could prove that this is the minimum possible number of memory elements for this system. That gets into linear system theory.

AUDIENCE: [INAUDIBLE].

PROFESSOR: I mean, what I want is down here, I want the sum of n_i times d to the i, v of D. So the linear coefficients are going to be these n_i. If I drew it out, I have n0 here, n1 here. So it's scalar linear combination, over F2. And what is that? v of D is common. That's just n of D times v of D. Same trick up here.

Is that what was puzzling everybody? Think about it. What is a linear combination? Something that looks like that. The second part, let's now talk about a convolutional encoder. And we're just going to talk about rate 1/n in this course. So this is a single input. n output linear time invariant system. That's my definition, I guess, of what I mean by a convolutional encoder. And when I say linear time-invariant, I also want it to be a realizable, in the two senses that its impulse response is causal rational.

So [INAUDIBLE] down. Now it's characterized by n-tuple of impulse responses. So it's going to be sum g of D equal to g1 of D, up to gn of D, where clearly all these have to be causal and rational in order for it to be realizable. If any one of them was not causal or not rational, then that individual impulse response wouldn't be realizable and the whole thing wouldn't be realizable.

So where we have the gj of D are all causal rational. And now, again, I'm going to put this into a standard form. In general, this is going to be n1 of D over d1 of D, so forth, up to n_n of D over d_n of D. So I'm going to have different denominators for each of the numerators. But I can always put it into the form, n1 prime of D over a common numerator, d to D, and up to n_n prime of D over d of D. Let d of D be the least common multiple of all these d_i's.

These are all polynomials. Least common multiple is well-defined from our discussion of factorization. So I can always put in the least common multiple here, and then whatever d1 lacks out of the least common multiple, I multiply top and bottom by both of that. I have the same rational function, now with a common denominator. So I'm always going to put it in that standard form.

And then I will say this is always realizable, with nu equal now the max of any of the degrees that appears in here. So let me just abbreviate that by degree of the numerators, which I'll just write as vector n prime of D, or the degree of the denominator d, of the common denominator d of D. Now, that's very easy to see.

Why is that? Can certainly realize the first one here just by doing that. Yes? If I want to just generate the first output, I build a circuit like this, and I don't need more of the new memory elements. There might even be some redundancy in that.

All right, so how would I then realize the second? Is it just me? Nobody is volunteering anything. I've realized this. Now I'd like to realize a second output. Now I want to realize this. Make this n1 of D. I want to realize n2 of D over d of D, times u of D.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Thank you so much.

AUDIENCE: Just take out [UNINTELLIGIBLE].

PROFESSOR: All right. So all I need is the second linear combination. So let me just make this n linear combinations. Then we're going to have n outputs, an n-tuple of outputs. Then I can form each one as a linear combination of those up there. I really like more of you to be speaking up. Yes, thank you.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Excuse me?

AUDIENCE: There's a second combination -- I mean the inquiry should come [UNINTELLIGIBLE].

PROFESSOR: Yeah, but again, what I want to realize, I have all the delays of v of D up here. What I want to realize is n2 of D times v of D, which is equal to that. n2 of D is the polynomial of degree less than or equal to nu. So I can do it.

AUDIENCE: Yeah, but [UNINTELLIGIBLE PHRASE] parallel to that n1 D over d. The -- [INTERPOSING VOICES]

PROFESSOR: Yeah, OK. I'm confusing you because I put all these here. I actually wrote out the linear combination. To say what I just said, I really need to go back to my original form, forget these multipliers, do the multiplications in here. That's what I mean by the linear combinations. Now I'm OK.

The inputs are just the shifts. Make a linear combination of them, that's the output. So easy proof. What's significantly harder to prove in this case is you can't do any better than. There aren't any realizations with fewer than nu, where nu is computed by first reducing to standard form and then evaluating this. That's the best you can do. But we aren't going to go into minimal system realizations, linear system realizations in this course. I'll just insert it.

So this is how we can always build a convolutional encoder, whether it has feedback or not. And so whatever we come up with as nu, this'll basically determine the state complexity of our decoder. The state space is dimension nu, is finite dimension. And because we're over a finite field, the actual number of states is only 2 to the nu. Still can't get more than 2 to the nu states for that system up there. So we're still in finite state world.

So in particular, it's finite state realization. And again, this is if and only if. If we have if and only if, we have an n-tuple of causal rational, or even a matrix of causal rationals. We can make a realization like this. So that's convolutional encoders. At least write 1/n over F2. And now the next step was what's a convolutional code.

And a convolutional code, we defined as, given a convolutional encoder, which is just the set of impulse response, given g of D, the corresponding convolutional code is just u of D, which is single sequence times this n-tuple of sequences, as u of D ranges through all the formal Laurent sequences. Sequences that start at some time in the all-zero state and then continue perhaps forever.

It's very quick to show that C, its properties is it's linear. Obviously, it's a vector space over F2. Multiplication by scalars is trivial. Addition is trivial, so it's linear. And its time-invariant, meaning the shift of any code sequence is another code sequence. If I have one particular code sequence, and I want to see if the shift of that is in the code, well, that code sequence must have been generated by some use, so if I just shift the u by the amount of time that I want to shift the output, y, then I'll get the shifted output.

So it's easy to show that D to the k of C is simply equal to C for any integer k. Actually, it just suffices to show that the single time unit shift is in the code, and that implies all the rest of this. Yes?

AUDIENCE: [UNINTELLIGIBLE]. We would want at least one of the g_i of D's to start at 0. Otherwise, they don't cancel out, do they?

PROFESSOR: Right. So what you're saying is we don't want d to be a common factor of all the n of D's. And that's true. In fact, we don't want to have any common factors of all the n of D's.

AUDIENCE: [INAUDIBLE].

PROFESSOR: The u we've defined, so it can start at any time.

AUDIENCE: [INAUDIBLE].

PROFESSOR: So notice that these have separate algebraic characters. This is a Laurent sequence. These are called causal rational sequences. So they have different restrictions on them, but they play together that way. So where you're getting to is this idea of code equivalence. I can just keep going.

So code equivalence, let's abbreviate it this way. g of D and some other n-tuple, g prime of D, generate the same code. We say that g of D generates C up here. So two different n-tuples generate the same code, C, which is our notion of equivalence. And we more or less proved last time it is true that g of D is some -- this is just a single sequence multiple of g prime of D. So we have to have a of D not equal to 0. Otherwise, I think there could be any sequence there.

So what you were saying is that, if all of these n's had a common factor of d, then why not just shift them all over? And that would be the same as multiplying -- suppose we had a g prime of D where they all had d's. Suppose we just multiply them with d minus 1. We're still going to get the same code, but we're going to reduce the degrees of all the numerators. And therefore, we're likely to get a simpler realization. And that's the track we're going down right now.

AUDIENCE: [INAUDIBLE].

PROFESSOR: This would only shift the n's. I'm assuming that all the n of D's are multiples of d.

AUDIENCE: [INAUDIBLE] the root of d, right? So if you can reduce the d of D you can't save anything.

PROFESSOR: You might not save anything if you've got a denominator, d of D, that has a larger degree.

AUDIENCE: And that's always the case.

PROFESSOR: No, it's not always the case. In the general case, either one of these things can dominate. In that picture we had, if we have a low degree, d of D, then it's only picking off of these first elements up here.

AUDIENCE: But that definition states [UNINTELLIGIBLE] is the maximum d and also the

[UNINTELLIGIBLE].

PROFESSOR: Right. So we could have a big one here making the outputs, and a small one there going back. In fact, we could have d of D equal to 1. Then we'd have no feedback whatsoever. So it could be either way. But then given this code equivalence concept, suppose we have a d of D that is very big. Suppose it's dominated by d of D. What would be a good thing to do?

We have a certain code we want to keep, but the nu, the numbers for the dimension of the state space is dominated by the degree of d of D. So suppose we have g of D equal to n of D over d of D. And we have degree of d of D is greater than degree of n of D. What would be a good thing to do? Why don't we just let g prime of D be equal to d of D times g of D, and that would be just n of D.

So I'm going to convert from the code generated by these rational generators, which are infinite. I can easily just multiply out the denominator, and now I have a code generated just by the numerator terms. So it's feedback-free. That may or may not be important to me. Actually, it turns out it's very hard to make a case for not having feedback, but it's simpler, in any case.

We don't have this feedback term in the encoders. So we get a feedback-free encoder. And if this is true, we may reduce -- we can certainly never increase it. But if this were true, then we would reduce it. If on the other hand, this were less than or equal, then nu would remain the same. So that seems like that would be a good thing to do.

So in fact, as I would advocate as a first step, that you clear the denominators, and just come up with a polynomial generator sequence which generates the same code. We're still going to have the same minimum distance, the same code sequence, the problem from the decoder's point of view is going to be absolutely unchanged. The decoder only cares what the consequences are. It wants to find the most likely one that was transmitted. So let's not make the encoder any more complicated than we have to.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

PROFESSOR: nu doesn't change in this condition. nu does change in this condition.

AUDIENCE: So we are always better off multiplying by g of D.

PROFESSOR: We can't be worse off. [INTERPOSING VOICES]

AUDIENCE: --or the same.

PROFESSOR: Exactly. Got it. All right, so let's now talk about whether there's anything we can do to n of D. We already suggested that if all the n of D had a factor of d in them, why don't just take it out? Let's just write that down. If all n of D have a factor of d, then let's just transform n of D to d minus 1 n of D.

So that reduces nu also. And let's, of course, do that as many times as we can. So it's a good idea to take out common factors of d. We get a simpler encoder. We're have a simpler state diagram. This leads us to a topic called catastrophicity, which is a misleading title because you have no idea where I'm going right now. So I'm actually going to try to fool you for a little while. Yes?

AUDIENCE: Quick question. In the definition of code equivalence, is there a restriction [UNINTELLIGIBLE] that shouldn't be rational, shouldn't be causal?

PROFESSOR: Should there be a restriction on this? Yes, of course. It has to be rational in order -- But that's a consequence. If these are both causal rational, then what does a of D have to be? It has to be rational. It has to be non-zero. And does it have to be causal? No, in this case it's not causal.

AUDIENCE: [INAUDIBLE].

PROFESSOR: That's right. If there's a delay buried in g of D, if everything starts at time 1 or later, then a of D, in fact, can be non-causal. So the key restrictions are that these two guys have to be causal rational. And that will tell you restrictions on a. Let me introduce this by example. And I'll warn you that I'm going to try to fool you for a while. Let's just take a rate 1/1 encoder. This may seem a little peculiar to you, because it has no redundancy, one input, one output.

And I'm going to let g of D just be 1 plus d. So what do I mean? I'm going to try to get some mileage out of this encoder. Here's u of D. Here's y of D, equals 1 plus d times u of D. Now let's draw the state diagram for this encoder. It's only got two states. One is 0. If we get a 0 in, we of course put a 0 out. We get 0, 0. Or we could get a 1 in. In which case, we would get a 1 out. Sorry, that was just a single output. And go to 1.

If we're in the 1 state, and we get a 0 in, then we get a 1 out. Whereas if we get a 1 in, then they're both 0, and we get the complement. So with the 1, we get a 0 out. So that's the state transition diagram of this encoder.

Now, let's go through a similar kind of argument to what we went through to find the minimum distance was 5. This is a linear system, so we want to find the minimum non-zero weight of any code word. And here's my argument. There's a gold star to the first person who punches a hole in it. Any code word, you have to start in the 0 state, so you're always going to get at least one unit of distance when you go to the 1 state. You can stay out here as long as you like, but eventually you have to come back to the 0 state.

And at that point, you're going to get another unit of distance. So I claim the minimum non-zero weight is 2. Is that correct? Let me try to create some doubt. What is the code here? The code is the set of all multiples of 1 plus d times u of D, where u of D is a formal Laurent sequence. What is that? I can write this briefly as just the set of all formal Laurent sequences times 1 plus d.

AUDIENCE: [INAUDIBLE] 1 minus d minus d squared.

PROFESSOR: Good. That's correct. That's what's wrong with my claim. Let me detour around here. Let me get an answer to this question. What is the code in this case? It's the set of all possible sequences you can get by multiplying 1 plus d times the formal Laurent sequence. What is that?

AUDIENCE: [INAUDIBLE].

PROFESSOR: What?

AUDIENCE: Isn't it [UNINTELLIGIBLE]?

PROFESSOR: It's just the set of all formal Laurent sequences. Certainly for every formal Laurent sequence, I can get such a sequence this way. In particular, the code includes 1. What do I need as an input to get the output sequence 1?

AUDIENCE: String of 1's.

PROFESSOR: String of 1's, right. So an example, if u of D is equal to 1/1 plus d, which is a string of 1's, then y of D is equal to 1 plus d times u of D, which is 1. 1 plus d has an inverse, and this is it. So now we're coming back to Mr. Aggarwal's point.

Suppose we put in u of D equals this sequence, all 0's before times 0, all 1's after time 0. Then where will we go? We'll go 1, and then we'll just stay in this 0 loop forever. Anything wrong with that? Is that a code word? That is a code word, the way we've defined the code. If we had defined the code so that the u of D were only finite input sequences, then this claim would be correct. For any finite input sequence, we eventually have to come back to the 0 state.

So this is true for finite inputs, not true for infinite inputs. So this is wrong. The fact is d3 equals 1. The minimum weight of this code is 1. But you see this is, first of all, it's a confusing situation for analytical purposes. Let me further indicate why it might be bad to have an encoder like this in general.

The key catastrophicity, in general, will be defined as there are finite outputs generated by infinite inputs, as above. Or equivalently, there are 0 loops in the state transition diagram. So by choosing a proper input, you can just keep driving around a 0 loop forever.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Excuse me?

AUDIENCE: [INAUDIBLE] the output is becoming finite, so it's actually -- I'm just commenting on the whole catastrophicity.

PROFESSOR: OK, well, I want to next tell you where catastrophicity comes from. The opposite of this property is non-catastrophicity. What we want is non-catastrophic encoders. We want encoders where, to get a finite output, you get a finite output only from finite inputs. That's better analytically and it's better in practice.

Here, let me draw you the practical argument. We have, for this particular case, u of D comes in, gets multiplied by 1 plus d. That gives us y of D, which -- let me just abbreviate it. We transmit that over a channel, and we get our decoder. Actually, decoder doesn't have much to do here, because all possible sequences are allowed at the output. So if this were a binary symmetric channel, the best the decoder could do would just be to put out whatever it sees as its guess, y-hat of D. So this is the decoder estimate.

So in any case, this can differ by code words from that. All possible code words are possible. And given y-hat of D, to get back to u of D, what do we have to do? We have to divide by 1/1 plus d, in effect, to invert this equation. For y of D is equal to 1 plus d times u of D, then the u of D that corresponds to a particular decoded code sequence, y-hat of D, we get by dividing by this.

But this is infinite sequence. Suppose this makes just one error. It's possible for this to transmit all 0's, and for this to make just one error. That's a code word. That's a legitimate code sequence. So it's something the decoder might come up with. If it does that, then what's this going to look like? This is going to look like an infinite number of 1's, because this is the infinite input that causes this code sequence with a finite number of outputs.

So it's precisely this condition, whenever we make an error that corresponds to a finite output sequence, and we translate it back to the encoder input, that's an infinite encoder input. There's a one-to-one correspondence between inputs and outputs. So it's got to happen whenever we make that type of error. And this is called catastrophic error propagation.

The decoder hasn't done anything bad. It made one error, as it's going to do once in a while. But the consequences are catastrophic. We make an infinite number of errors in the bits that we actually deliver to the user. Now, of course, what's really going to happen is that sometime later the decoder is going to make another little error. And at that point, this will switch state again.

So that will terminate the error burst when the decoder makes another error. But this is completely random. It may take a long time for this to happen, and it's not good. So for practical reasons, we don't want catastrophicity.

I was going to give you another example, but in the interest of time, I'll just write it down. Here's another catastrophic example. Let me just do a rate 1/2 encoder so it looks more reasonable. And we'll make it look very much like the encoders we've had. Our first example looks something like this. 1 plus d squared. I'll say 1 plus d.

So it's a feet-forward encoder. It looks perfectly reasonable. The top channel is impulse response, 1 plus d squared. The second output has impulse response, 1 plus d. If you look at the minimum weight for all finite sequences, it's 4, not quite as good as the other one which had 5. But is this catastrophic? Yes? Who said yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Great. So the same observation, if u of D is equal to 1/1 plus d, which is, again, this infinite sequence, then the output, y of D equals u of D, g of D, is equal to -- 1 plus d divides both of these, so we get 1 plus d1. We get a finite output sequence. You saw that immediately, because we know that polynomials are divisible by 1 plus d if and only if they have an even number of non-zero terms.

So that's a more elaborate example of where we can get catastrophic behavior. So what should we do here? Clearly, we should divide out the common factor from here. We can do that. If all of these polynomials have a common factor, we should divide it out. And we know that the code generated by -- if this is g of D, we're going to say g prime of D is equal to 1/1 plus d times 1 plus d squared, 1 plus d. And that is -- we've already calculated -- 1 plus d1.

So by doing that, first of all, we got rid of the catastrophicity. Second of all, again, we reduced nu. So this is clearly a very good thing to do. So more generally, going back here, if all n of D have any common factor -- in other words, let's call -- I've used d, so what'll I use? b of D equals the greatest common divisor of all of the n of D, 1 plus --

AUDIENCE: [INAUDIBLE].

PROFESSOR: Who --

AUDIENCE: Do you think it could be 1?

PROFESSOR: You're, once again, 30 seconds ahead of me. So yes, that's what I'm going to say. If they all have a common factor, then they have a GCD that is not equal to 1. That's if and only if. So we're going to transform n of D to 1 over b of D. We're going to divide out the common factor, n of D. That will be our n prime. And now what have we achieved?

We've achieved that there is no common factor of these n prime of D. So no common factor, which does turn out. So that will mean it's non-catastrophic, and, furthermore, will reduce nu by whatever the degree of this common factor was, degree of b of D. Let me write degree of GCD.

We want them to have no common factor. We want the greatest common divisor to be 1. Another way of saying this is we want the ni of d to be relatively prime, the nj of d to be relatively prime. And if that's true, then can we prove that it's non-catastrophic, that if we have a finite output, that it must've been generated by a finite input?

AUDIENCE: No, because u of D times n_i of D would have to be a polynomial to be catastrophic, and to get a polynomial. And u of D is rational. It has something in the denominator, because u of D is infinite. That denominator should then get cancelled out to one of these.

PROFESSOR: Yeah, you have the argument in your head. Let me leave that as an exercise for the student, since at least a couple of you see it. The only way that you can get this behavior, if you have an infinite input which has in its denominator a common factor of all the n's. So if the n's have no common factor, then this can't happen. That's the outline of the argument.

So what's our conclusion now? Since we have multiple encoders that generate the same code, we'd like to get the nicest one, in some sense, as our canonical encoder for a given code. So we have a given code. We can specify it by any causal rational transfer function. But then, having done that, we should clean this generator n-tuple up. We should multiply out by the least common multiple of all the divisors to make it polynomial, feedback-free, possibly reduce nu.

And then we should check the numerators and see if they have any common factor, which could be d or it could be any polynomial. And whatever it is, we should divide that out. And that will certainly reduce nu. So at the end of the day, we have a canonical encoder, which is equal to g of D times -- we want to multiply by the least common multiple of the denominators. We want to divide by the greatest common divisor of the numerators.

And this will be some n prime of D, which will be polynomial. It will be delay-free. That means it doesn't have d as a common factor. It will be non-catastrophic. That means it doesn't have anything else as a common factor. And it will have minimal nu, for any encoder that can generate that same code. I haven't quite proved all those properties, but I think we're well along the way. So this is really --

AUDIENCE: [INAUDIBLE]. There should be at least n1 and n2. [UNINTELLIGIBLE]

PROFESSOR: Well, if we have a rate 1/1 code, we just have one of them, then what's the greatest common divisor? Suppose it's n of D over d of D. The greatest common divisor is n of D. The least common multiple is d of D. We get 1. So going through this exercise for any rate 1/1 code, we'll find that the canonical encoder is 1. And that is clearly what we want for a rate 1/1 encoder, because a rate 1/1 is always going to have an out. The code is just going to be the universe code of all the possible formal Laurent sequences.

So we get down to the right encoder for that code, not some stupid encoder. And in a similar way, going through this process, we come up with the right encoder. Is this process unique? Is there only one encoder we could come up with? Think about it a little, and you could convince yourself there's only encoder that you can get from doing this. This is over F2. Over a larger field, it's unique up to units again, up to a non-zero scalar multiple, which of course you can define to be 1 and somehow make this unique.

So it's basically a unique canonical encoder that has all of these properties. You get it in this way. So this is what we're always going to use. For instance, for our original encoder, 1 plus -- what was it? -- 1 plus d squared, 1 plus d plus d squared. Is that already in canonical form? This is irreducible. This is divisible by 1 plus D, so these are relatively prime.

It's feed-forward. It doesn't have any denominator terms, and it's delay-free. So it satisfies all of these, and clearly you can't manipulate it. The only ways you're allowed to manipulate it are by multiplying by a of D over b of D. And it's clear that if you have a non-trivial numerator or denominator, the numerator will make it catastrophic, the denominator will make it to have feedback.

Now, as I was saying, there's really nothing wrong with feedback. And so some people prefer -- the one thing that this doesn't have, in general, is it's not systematic. That means the input stream is not one of the output streams.

So given a canonical encoder like this, can we always make it systematic? Yeah, we just divide by any of these polynomials. So here's an equivalent systematic encoder. For instance, if we divide both of these by 1 plus D squared, we get 1 plus D plus D squared, over 1 plus d squared. This now is not polynomial. It is delay-free. It's non-catastrophic. There's no 1 over something that we can put in to get a finite thing out.

It turns out it's still minimal nu. We know how to realize this with two memory elements now. And in general, if you're not going to introduce a denominator term that has a greater degree than the maximum degree of the numerator terms, so it's still realizable with minimal nu. And so now it's systematic, but now it's not feedback-free.

So I consider this one certainly the nicest. The canonical feedback-free encoder is the nicest one for analytical purposes, for instance, for trying to find minumum-weight finite code words. Because of the finite property, if we're looking for the non-catastrophic property, if we're looking for minimum-weight code words, we only have to look at finite inputs. We can now use the kind of analysis method that we use that says you always have to come back to the 0 state, because only finite inputs can produce finite outputs here.

This also has that property. And people tended not to use these for a long time, but it was founded that in turbo codes, you want to have this, one of the generators be infinite. And it was also nice to have one of them be systematic, so that led to a revival of interest in systematic convolutional encoders. And this, again, it's a unique form. If you say the first one has to be the systematic one, then there's a unique equivalent systematic encoder that generates any code. So it's in alternative canonical form. Take your pick. Yeah?

AUDIENCE: What's systematic?

PROFESSOR: What's systematic? Systematic means that the input that's appear unchanged as a subset of the output bits. For convolutional codes, it means that one of the output sequences, if you have a rate 1/n encoder, one of the output sequences is simply equal to the input sequence. In other words, one of the transfer functions is equal to 1.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yes, very good. Excellent comment. Because now, if you have a finite output sequence, in particular you can just read off the input sequence, the information sequence, from this first term, and if the whole thing was finite, it's finite. So systematic directly ensures non-catastrophicity. Excellent, excellent comment.

Which you might try to prove by the more elaborate techniques that we had.

So that gets us pretty much through the algebra that I wanted to do. Now I want to - - OK, good. Now let's go more to the finite state picture. And our objective here is mostly going to be to get to Viterbi decoding algorithm. But along the way, we'll see that we can use something like the Viterbi algorithm to compute minimum distances, too, once we have a finite state picture to the code.

Again, take our example one, where g of D is 1 plus D squared, 1 plus D plus D squared. We have a state transition diagram, which looks like this. And I won't bother to label it.

And that's a complete description of the encoder operation, if I put labels of inputs and outputs on all these states. So 0 slash 0,0, so forth, 1 slash 1, 1. Just to draw the impulse response, 0, 0, 1, and 0 slash 1, 1. There's the basic impulse response going around, like that. Trellis diagram is a very redundant picture of the same thing. We can start out in the 0 state that we like. Let's say we start off in the 0 state at some time k, as we always are. We just don't know what k is.

Then let's spread the state transition diagram out in time. Let's draw all the states again at time k plus 1. Actually, there are only two it can get to, at time k plus 1. So we can either stay in this state or we can go down here to this state, at the next unit of time. I'm just going to keep drawing all of the possible state trajectories or paths. At time 2, I can reach all possible states. 0, 0, 1, 0, 0, 1, 1, 1.

Here again, I already know what these are. From here, I can get 1, 0, 2, 0. It goes with a 0, 1 out. So 1 goes with a 1, 0. And k plus 3, I now have all possible transitions. It's going to look like this. Where can I go from 0, 1? I can go back to here, or I could go to here. And I could label these. I should label these. But I won't. Here, I can go to these two. There are eight possible state transitions. k plus 4. It's completely repetitive, since it's a time-invariant system. I just keep going like this.

And why have I gone to all this trouble? This clearly contains the same information as this diagram, but it's just spread out in time. The basic reason for doing it is that now there is a unique trellis path corresponding to every code word that can start at time k or later.

So for instance, if I just have an impulse at time k and nothing else, then the unique trellis path is this. It starts at 0, goes through two non-zero states, comes back to 0, 0, and then stays in 0, 0 forever more.

AUDIENCE: [INAUDIBLE].

PROFESSOR: I've just taken an arbitrary starting time, k. After a while, I'll just look at it -- well, it's entrained. This is kind of the steady state version of the trellis. I've included a little starting part here. And if I terminated the trellis, then I would also have an ending time. We'll talk about that.

So at this point, I just want you to see what the trellis diagram is. It's basically just an unrolling of the state transition diagram out in time. And it allows me to associate -- there's a one-to-one correspondence now between code words that start at time k or later, and trellis paths. So every one corresponds to a unique, distinct path through the code. That's what we're going to use in decoding.

Each of these paths represents one possible decision that the decoder could do about what was actually set. The decoder is actually going to try to find the best of these paths, in some sense, the one that's closest to the received sequence, in the Hamming distance or Euclidean distance, or likelihood sense.

Let's pause for a second, and see how this could be used to compute the minimum distance of the code. Let's assume that I have a non-catastrophic encoder. Then I know the finite sequences are going to be those that start in the 0 state, go through some trajectory through the state transition diagram, and then ultimately come back to the 0 state.

If I have canonical encoder, I agreed what I'm going to use now. So one way of computing the minimum distance would just be to perform a search through here, and try to find the minimum detour, the minimum weight detour that starts in the 0 state, eventually gets back to the 0 state. It accumulates Hamming distance along the way. And that is the Hamming distance of some valid code sequence.

So here, it's not very tough. And I've already gone through an argument. If you didn't know, you'd say, OK, here's the first one to look at. This is the only one that gets back in three time units. Let's look for the finite path that gets back to the 0 state in four time units. Again, there's only one of them. It looks like that, and it turns out it has weight 6. So this is, again, it always ends up with a 1:1 when it merges back in here. And this, I think, is 0, 1 or something like that. The only 0, 0 path in here is this one.

But you can easily see the 0, 0 path doesn't form a loop. You can't proceed down 0, 0 forever. So I go through here, and I've already got four units of distance up to here. In fact, I need to follow. Next time I explore out here, 1, 1, 0, I've already got up to four units of distance. Now I know I'm always going to have two at the end. So any time I get to four or more, I can stop.

And by that argument, I'm already done when I get to this one. Or if I didn't want to use that, I'd just explore -- I already know that there's one of weight 5, the impulse response, so I'd explore all the possibilities out until they accumulated at least weight 5, and assure myself that there was no way of getting back here with weight less than 5. And that would be a very systematic way of evaluating the minimum distance to the code. You can see you could do that for any generators.

You with me? Just explore this little graph, picking up weight as you go along. That's basically the way the Viterbi algorithm goes as well, except what we do is we have some received pair at time k, r1 k, r2 k, r1 k plus 1, r2 k plus 1. We get two received symbols at every time, whatever channel we're going over. Assuming that the channel is memory-less, I get a some kind of distance or weight or likelihood weight at each time, and maximum likelihood sequence decoding amounts to finding the sequence that's closest, the code sequence that's closest in Hamming distance or Euclidean distance or likelihood distance to the transmitted sequence.

So I can simply put a metric on each of these branches corresponding to what this was. Now, if it was a binary symmetric channel and I received 0, 0, then I'd have 0 weight here, but weight 2 here, in terms of Hamming distance from the received sequence. And similarly, as I go along, I'd have another set of metrics here. If I received 1, 1, then this one would have weight 2, this one would be good, no distance from the received sequence. Each of these would have distance 1 from the received sequence.

So I get some kind of cost for every path in here. And then the decoding algorithm is simply a matter of finding the minimum cost path. You could all do that. We just had a celebration for Andy Viterbi out in USC, where he's given them $50 million, so they were very happy, and had a great celebration. And the thing he's best known for is the Viterbi algorithm. But what everybody knows is that, once you reduce the problem to this point, anybody could have come up with the Viterbi algorithm. The Viterbi algorithm is just a systematic way for trying to find the lowest-cost path as you go through a trellis.

And if you know dynamic programming and it's obvious how to do it, or even if you don't, the next idea -- do we have a minute? There's only one idea in the Viterbi algorithm, which is, when you get to this point, you can already make a decision about what the best path is to get to each of these states. If the closest path over the first three intervals is this one, not this one, say, then you can forget about this path because it's never going to be the first part of the ultimate closest path, because we're simply adding up independent cost, independent increments.

So you can make a subdecision at each of these states, what the best path is up to that state, the closest path. And you only need to remember that. That's called a survivor. And this was basically, Viterbi was the first one to put down this algorithm with the survivor. You can make interim decisions. You can only keep the survivor. Then all you have to do is extend the survivor's one unit of time out here. Again, you have to add the new metric. You have to compare the metrics to see which one is better. You select the survivor, and once again, you've only got four survivors going forward.

So you get an iterative recursive algorithm that just proceeds forward one unit of time at a time, and keeps finding the four best survivors, or in general, the 2 to the nu best survivors at time k plus 3, k plus 4, k plus 5, ad infinitum. And that's the Viterbi algorithm.

So once you've drawn this trellis picture, it's very clear how you should decode, and you get a very sufficient decoding algorithm. We'll come back next time. I'll do that a little bit more carefully. I'll talk about terminated codes. I'll talk about how it works when the code isn't terminated, and I'll talk about performance analysis. But we're actually pretty close to the end of chapter nine at this point.