PROFESSOR: OK. Welcome back. I hope you had a nice break. The midterms will be handed out at the end of the class. Ashish has just gone to get them in my office. With solutions. I hope you look at the solutions. There's more in the solutions than is strictly necessary. The midterm is partly intended as a learning exercise, and I hope you learned something from it.

I didn't learn too much from it about the performance of the class, because some of the questions were easy enough that practically everybody got them, and some of the questions were too hard for anybody to get. And so the distribution -- you know, what I would really like to get is a nice bell-shaped curve going from 0 to 100 with a median at 50 and the standard deviation about 25. And I got a much more compressed curve than that. So frankly, I didn't learn that much about relative performance.

And I gather from Ashish that you've all been doing the homeworks, and have been doing a fairly steady job on that. So I think everybody is close to being in the same boat here. If anybody has concerns about their performance, of course they're welcome to talk to me. Drop date is a little while off. If you're concerned about what your grade might be, I'll give you the best idea I can. But really, it's going to come down to the final for most of you.

So it's not too much to say here. I give the grade distribution and the midterm solutions, and from that, you could figure out where you are relatively.

And my grading philosophy is at MIT, in graduate classes, roughly they're half As and half Bs. I have that as a guideline in mind. I try to look for where the natural breaks are. I decorate the As and Bs with pluses and minuses, but I don't think anybody pays attention to those. And in other words, I generally try to follow the MIT grading philosophy as I understand it.

So are there any questions about the midterm or anything about the class as we go into the second half? No?

I see some people aren't back yet. It's a somewhat reduced group. Maybe a couple people decided to drop on the basis of the midterm. That sometimes happens. I don't think it necessarily had to happen this year. Or maybe people are just having an extended break.

OK. We're now going into conceptually a break, the second half of the course. The first half of the course we were talking about trying to get to capacity, specifically on the additive white Gaussian noise channel. So far all the schemes that we've talked about have been block coding schemes, which their Euclidean images wind up to be constellations in a finite dimensional space.

This is the kind of picture that Shannon had when he wrote his original paper on capacity and information, to start off information theory. Basically he proved that you could get to capacity on any channel with a randomly chosen code. In the setting where we are, that would be a randomly chosen binary code where you just pick all the bits of all the code words at random by flipping a coin. OK? Just construct a code book at random, and as the code gets long enough, that's going to be good enough to get to capacity. It's a block code. 2 to the nr bits, and length n code words where r is the regular code.

Well, that was a brilliant achievement, to show that you could get to capacity by doing that. If you can do it by choosing codes at random, then there must be some particular codes that are good. Otherwise the average overall codes wouldn't be good. So people started to go out to find good codes.

And there was a fairly quick realization that Shannon hadn't addressed some key parts of the problem. Particularly the complexity of encoding and decoding. The complexity of encoding was pretty quickly recognized to be not much of an issue, because Peter Elias and others proved that, at least for the kinds of channels we're talking about, which are symmetric, linear codes are just as good. A random linear code will get to capacity. In other words, say, a block code with a randomly chosen generator matrix. You pick k generators of length n, again by flipping a coin, that's good enough to get to capacity. So we know we can encode linear codes with polynomial complexity. It's just the matrix multiplication to encode a linear block code.

All right? So encoding isn't the problem. But decoding. What kind of decoding have we talked about so far? I guess particularly with Reed Solomon codes, we've talked about algebraic methods. But our generic method is to do maximum likelihood decoding, which basically involves computing a distance or a metric to each of the code words. Exponential number of code words. 2 to the nr code words.

So we have exponential complexity if we do maximum likelihood or minimum distance decoding. And that, of course, was the motivation for doing some of these algebraic decoding methods, which were the primary focus coding theory for several decades. A great deal of very good effort went into finding good classes of codes that were algebraically decodable, like Reed-Solomon codes, BCH codes, Reed-Muller codes. [UNINTELLIGIBLE] don't have such -- well, they do, we now realize. But that was the focus. And with the algebraic coding methods, there was polynomial complexity.

But it began to be realized that these were never going to get you to capacity. Because of their difficulty in using reliability information, because most of them are bounded distance character, and in fact, that's still true. You can't get to capacity with algebraic decoding methods.

So in parallel with this work on block codes, there was a smaller stream of work that has broadened, I would say, as the ancestor of the capacity-approaching codes of today, on codes with other kinds of structure. And specifically dynamical structure, which is what we're going to talk about when we talk about convolutional codes. And this was broadened into the field of codes on graphs. And codes on graphs are nowadays the way we get to channel capacity. All right?

So that's really where we're going in the second half of the course. We're going to preserve linear structure. Linear is always good for the symmetric channels we're talking about. But in addition, we're going to look for other kinds of structure which are, in particular, suitable for lower-complexity decoding methods, are suitable for maximum likelihood or quasi-maximum likelihood decoding methods. All right? So that's our motivation.

OK. Convolutional codes. Convolutional codes were actually invented by Peter Elias in an effort to find -- just as you can put a linear structure on codes without harming their performance, he was trying to find more and more structure but you could put on codes while still being able to achieve capacity with a random ensemble. And he invented something called sliding parity-check codes, which, if you let the block link go to infinity, become convolutional codes.

And they say there was a trickle of effort basically motivated by the fact that a number of people did a practical comparison of performance versus complexity. Convolutional codes always beat block codes. And so, for instance, I was at a small company was trying to apply coding theory. We quickly gravitated to convolutional codes, because they always gave us a better performance complexity straight off than block codes. And I'm going to try to indicate why that's so.

All right. So I think the easiest way to understand convolutional codes is by example, by putting down an encoder. And the canonical example everybody always uses is this one, so why should I be any different?

Here is a rate 1/2 constraint length 2 convolutional encoder. And it operates on a stream of bits coming in. So what we have here is a time index stream of bits, uk. And you can think of this stream as being infinite in length. We'll talk about how to make this a little bit more precise as we go along.

And we have a shift register structure here. The incoming bit uk is D means a memory element or a delay element. It goes into a flip-flop. It's delayed for 1 time unit. There's some discrete time base going on here, which is indexed by k. So k is simply the set of all integers. Comes from the index set. Here is u k. Here's u k minus 1. Here is u k minus 2. All right? And constraint length 2 means that we save -- we have the present information bit. This is called the input or the information bit. And the two past information bits are saved.

And from that, we generate two output bits. y1 at time k is given by u k plus u k minus 2, where this is a mod 2 sum. Everything is in f2. y2k is the sum of u k plus u k minus 1 plus u k minus 2. All right?

So this is where we get the redundancy. What are we generally doing with coding? We're adding redundant bits in order to get more distance between the sequences that we might send, and thereby hopefully to get a coding gain on the additive white Gaussian noise channel. Here the redundancy comes from the fact that you get 2 bits out for every bit that you put in. So there ought to be the possibility of getting some distance between possible code sequences. The code sequences here are infinite or semi-infinite, but nonetheless, we can establish some minimum distance, or as it's called in this field, the free distance.

OK. There are two kinds of structure here. First of all, this is a linear time invariant system. If we were talking about the real or complex field, we might call this a filter. Or it's actually a single input, two output filter. So each of the y sequence is a filtered version of the u sequence. y2 is a filtered version of the u sequence. And it's linear basically because it's made up of linear elements, lay elements, and mod 2 adders. Over the binary field, it's time-invariant, because if I change the time index, it doesn't really change the equation. It's very hand-waving rough here.

But so it's a linear time-invariant system, and we can analyze it the way we analyze a filter. It's redundant, so it has more outputs than inputs, which we want in a coding system.

But other than that, we're going to find that the same kind of techniques that we use for analyzing discrete time linear filters can be used to analyze the system. It's just over a different field, F2, rather than R or C. All right? So that's one kind of structure.

And secondly, it's a finite state system. Let's forget about all the algebra. There's a discrete time system which has two memory elements in it, each capable of storing 1 bit. So how many states are there in the system? Four. There are four possible states that reflect everything that went on in the past. That's all you need to know about the future, is what the state is at time k, to see what's going to happen from time k onwards.

All right? So this is a simple four-state system, in this case. In general, if we build similar encoders out of shift registers, even if we put feedback in them, it's going to be a finite state system. So we're going to restrict ourselves to finite state realizations, finite state encoders. And that's going to ultimately be the basis for the decoding algorithm, which is going to be Viterbi algorithm, which is a very efficient decoder for any finite state system observed in memoryless noise, which is what we have here. Hidden Markov model, as it's sometimes called.

All right. So we're going to interplay these two types of structure. And I feel it's really the fact that convolutional codes have these two kinds of structure that makes them better than block codes. They have just the amount of algebraic structure that we want, namely, the linear structure. They don't have too much algebraic structure beyond that. We don't have elaborate roots of polynomials and so forth, as we do with Reed-Solomon codes. We have a pretty modest algebraic structure here. And it's sort of different in character than the algebraic structure Reed-Solomon codes. And it's married to this finite state dynamical structure, which is what allows us to have low-complexity decoding.

So that's kind of the magic of convolutional codes. Not that they're very magic.

All right. This is a convolutional encoder. What is a convolutional code? The convolutional code is roughly the set of all possible output sequences. And this is where we're going to measure the minimum distance of the code, minimum free distance. What's the minimum Hamming distance between any two possible output sequences?

And in the notes, I go into considerable detail to explain how we choose the particular definition of all. What do we mean? Do we mean the set of all possible output sequences if we put in a finite input sequence? An input sequences that starts at time 0, and this polynomial, let's say, that ends at some finite time? Are we going to allow semi-infinite, bi-infinite input sequences?

So you see, there's some choices to be made here. And actually, it's of some importance that we make the right choice to get proper insight into these codes.

AUDIENCE: [INAUDIBLE] You have to choose the initial state, then?

PROFESSOR: Yeah, OK. You know from linear systems that the output isn't even well-defined unless you say what the initial state is. And in general, in linear systems, we say, well, we'll assume it starts out in the all-zero state. And that's basically the choice I'm going to make, except I'm going to do it in a couple steps. But that's the right question. How do we initialize the system? That's a part of the specification of this encoder. And of course we initialize it in the all-zero state.

But for that reason, in a general system, we can't allow bi-infinite inputs. Because if we don't know what the starting time is, we don't know how to define the starting state. Again, I'm speaking very roughly. So let me speak a little bit more precisely.

We're going to define the set of all Laurent. Which means semi-infinite sequences over the binary field. And this is called F2 of D. And this means a sequence -- U, say -- which is only a finite number of non-zero terms before time 0.

In other words, it can start at any negative time, but it has to have a definite starting time. I'm not talking about the set of all power series or the set of all sequences that start at time 0 or later. The sequence can start at any time. But it does have to have a definite starting time, and before that, it has to be all 0.

So that for instance, the sequence that -- I haven't introduced D transforms yet. Maybe I should do that immediately. But a sequence which, at minus 5, minus 4, minus 3, minus 2, minus 1, 0 -- these are time indices -- looks like 1 0 1 0 0 1 0, and goes on indefinitely in the future, but has all zeros back here -- that is a legitimate Laurent sequence. Whereas a sequence that's dot dot dot 1 1 1 1 1 1 1 1 and so forth, all ones forever, is not, because it doesn't have a definite starting time.

And I sort of -- all right in this notation -- I need to introduce D transforms. The D transform of a sequence U which has components Uk for k and z it is simply the sum of Uk D to the k. Let me write that as U of D. For all k and z.

This is simply a generating function. It looks very much like the Z transform that you see in discrete time linear filter theory.

We would write for something up here, well -- So a sequence that is 1 at time 0, 0 at time 1, 1 at time 2 -- if U is equal to that, and 0 at all other times, we would just write U of D equals 1 plus D squared. So it's a generating function. The D just gives us a location of where the ones are. It's easier to write this than this, frankly. It's actually hard to write the sequence which is 1 at time k and 0 at all other times in this notation, but it's simply D to the k in this notation.

D here is algebraically simply an indeterminate, or a placeholder. And this is the subtle difference from Z transforms. The Z transforms, we would write U of Z minus one equals 1 plus Z minus 2 for something like this. There's this Z to the minus 1 convention in Z transforms.

That's not the important difference. We could clearly use D to the minus 1 rather than D. The important difference here is that Z is usually imagined to have values in the complex plane. We look for it's poles and zeros in the complex plane and so forth.

And so the Z transform really is a transform. It takes you into the frequency domain. If you make z equals e to the j omega t or something, then you're in the frequency domain.

Here, this is still a time domain expression. Just as a convenient, compact, generating function notation for a particular sequence. So that's when I say the set of all semi-infinite sequences, now I can write -- a Laurent sequence always looks like this. It starts off at some particular time. Let's say the first term is Ud D to the d plus Ud plus 1 D to the d plus 1 plus so forth. It always looks like that. So it has a definite starting time d, which is called the delay. The delay of u of D is equal to d, in this case.

And d can be any integer. It can start at a negative time or at a positive time. We don't require it to start at time 0 or later. But nonetheless, the sequence has to look like this. And we indicate the set of all such sequences here.

One of the immediate observations that you can make is that this is a time invariant set. Which the set of all sequences that start at 0 or later is not. Set of all sequences that start at 0 or later has a definite time in it, and a set is time-invariant if D times the set is equal to the set.

So what is D times F2 of D? I claim it's just F2 of D all over again. And in fact, D to the k for any k is equal to F2 of D.

So the set of all Laurent sequences is time-invariant. Whereas the set of all power series in D, that's sequences that have delay 0 or greater, is not time invariant. It does not satisfy this. Chew on that.

An important consequence of this is that the set of all Laurent sequences forms a field. Let's say what we need to prove for a field. Can we add Laurent sequences? Yes, we simply add them in the usual way, component-wise. Same way as you do in polynomial addition. You just gather together compounds at time k.

Can you multiply Laurent sequences? You can certainly multiply polynomials. 1 plus D times one plus D is 1 plus D squared over F2. Can you multiply Laurent sequences? Well, yes. We can define multiplication by convolution as we do with polynomials. In other words, U of D times V of D is the sequence W of D, where Wk is the sum over k prime in Z of Uk prime Vk minus k prime.

Now what's the potential problem that can arise when you define multiplication by convolution? If any of these terms are infinite, we have no notion of convergence over F2. So we really have no way of defining an infinite sum. An infinite sum of elements in F2 is, in general, not well-defined.

But from the fact that each of these starts at one time -- this is basically multiplying something by the time reversal. We only have a finite number of elements in any such sum. So that's the real reason for using the set of all Laurent sequences. If we had the set of all bi-infinite sequences, then multiplication would not be well-defined.

But by insisting that they have a starting time, we ensure that convolution is always well-defined. Therefore, multiplication of Laurent sequences is well-defined.

OK. So we have addition, we have multiplication. Maybe I'll go over here. What else do we need for a field? So far we just have a ring. Inverses, yeah. Let's directly check inverses.

So suppose I have some Laurent sequence U of D equals U(d), D to the d, plus so forth. Does that always have an inverse? In other words, we want to find something 1 over U of D equals what?

Well, let's just divide U of D into 1 by long division, and we can get an inverse. Let me give you an example. Suppose U of D is equal to 1 plus D. What's 1 over 1 plus D? We take 1 plus D, we divide it into 1, we get 1. 1 plus D, D plus D, D plus D squared. And so forth.

So we get a semi-infinite sequence. Not a polynomial. But nonetheless, it's a legitimate Laurent sequence. It's actually a very simple periodic sequence of period 1. Are you with me on this? Could you find the inverse of any sequence that starts at time 0, let's say? If you can do that, you can certainly find the inverse of any sequence that starts at time D. You just multiply by D to the minus d.

So every non-zero Laurent sequence has an inverse. 1 over U of D, let's say. Which is also a Laurent sequence. Of course, as always, we can't divide by 0. But that's the definition of a field. That doesn't matter. We have inverses. We checked the associative, distributive, commutative laws. They all work. So this is actually a field. Yes?

AUDIENCE: [INAUDIBLE] multiplication [INAUDIBLE].

PROFESSOR: Yeah. The Laurent sequences certainly include polynomials, and this is consistent with polynomial multiplication. So it's just extending polynomial multiplication really as far as we can to these semi-infinite sequences that are semi-infinite in the future, but not in the past. They're infinite in the future, not in the past.

AUDIENCE: I think that's [INAUDIBLE] infinite sequence of [INAUDIBLE]. [UNINTELLIGIBLE]. I mean suppose you -- To find the inverse we just have to keep solving the set of linear equations, right? From the first equation, we get the first one, from the second one, by substitution we get the next one.

PROFESSOR: Yeah. It's long division, which is the same as the Euclidean division algorithm. We just want to find a coefficient here that reduces the remainder to start at time 1 or later, to have delay 1. We want to find the next coefficient to make the remainder have delay 2. And we can always continue this, ad infinitum.

AUDIENCE: [INAUDIBLE]

PROFESSOR: These are semi-infinite sequences. Do you mean sequences that start at time 0 or later, which are called formal power series? Again, there are some --

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh! Sorry, yes. Of course, we need to check that. But you can see, it doesn't matter. We could divide by 1 plus D plus D squared plus so forth, and the algorithm is the same. In that case, of course, if we did that, we'd get 1 plus D plus D squared, plus so forth. D plus D squared, plus so forth. And what do you know? D times that is that, and we get 0, and it terminates after 1.

So yeah. Thank you. Long division also works. If we divide by any Laurent sequence, it's the same. The same algorithm. Excellent.

OK. So every Laurent sequence has an inverse. In general, what do the inverses of polynomial sequences look like? Inverse of a polynomial sequence. Can anyone guess what its special property is?

What is a polynomial sequence, first of all? I say a sequence is finite if it only has a finite number of non-zero terms. I'm going to say it's polynomial if it's finite and also causal. Causal means it starts at time 0 or later. Its delay is non-negative.

All right. So that's the polynomials that we're already familiar with. What does the inverse of a polynomial sequence look like?

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's always going to be infinite, right? Unless it's 1 or something. So unless it's an invertible unit in the polynomials, it's going to be an infinite sequence. But it's going to have a property. Nobody can guess what the property's going to be? I already mentioned it once. Periodic! Right. And this is an if and only if statement. I guess I have to elaborate a little bit, but roughly, that's correct.

This is going to be one of your homework problems. Maybe I'm jumping ahead a little, but let me -- there are a couple of ways of seeing this. One is to use some of the algebra we did back in chapter seven. But since we didn't really complete that, I won't do that.

A second one is to see within this long division operation, again, if I'm dividing by a polynomial, there's only a finite set of possible remainders so I can get up to shifts by multiples of D. So if I see the same remainder again, if I see 1, D, and then D squared, of course the series is going to have to continue in the same way necessarily. Since there are only a finite number of remainders, it's got to repeat at some point.

Well, a way that I can do it which is more in the spirit of this. Suppose I consider the impulse response of a system with feedback, which is set up so the impulse response is equal to, in this case, say, 1 over D plus D squared. You see that the impulse response of this is going to be 1 over 1 plus D plus D squared. If I put in a 1, time 1, then next time, I'm going to get a 1 in here, going to get a 1 feeding back there.

Well anyway, I believe that's right. This gives an impulse response of 1 over -- if it doesn't, then readjust it so it does.

Now, after time 0, the input is always 0. So I just have an autonomous system which is producing the terms at time 1 and later of 1 over 1 plus D plus D squared. And again, it's a finite state system. Because for over F2, there are only 4 states in the system.

So what could its state transition diagram possibly look like? First of all, the 0 state always goes around to the 0 state. So the other states are interconnected in some way. And at best, they're always going to cycle. So this is a quick proof that 1 over 1 plus D plus D squared -- in fact, any polynomial inverse -- is going to give you a periodic response.

What I should really say is -- let me define a rational Laurent sequence. I'll leave out the Laurent. A rational sequence is something of the form n of D over d of D where these are both polynomial, or they're actually finite. Let's say they're both polynomial. You can reduce it to lowest terms, if you like. Actually, I should make this finite up here, so that it can start at time before time 0. And I have to have d of D not equal to 0. And in fact, I generally require d0, the first term down here, to be 1. So this is a polynomial with time 0 term equal to 1. Like 1 plus D plus D squared.

So a rational sequence is one that looks like that. We can generate a rational sequence by feeding n of D into a system that has response 1 over d of D. And the output -- I forget where you take the output off here. Probably here. This will be n of D over d of D.

OK. There's a linear system. Single input, single output. If it has impulse response, 1 over d of D, then if I put in the sequence n of D, I'm going to get out n of D over d of D.

And so by the same finite memory argument, n of D over d of D is going to be eventually periodic. So a sequence is rational if and only if it's eventually periodic. And again, on the homework, I'm going to ask you to prove this more carefully. You can use this finite memory argument if you like.

So this should remind you of something. This should remind you of real numbers, integers, ratios of integers, which are rational real numbers, all that sort of thing. What are the analogies here?

First of all, how big is it? How many real numbers are there? It's uncountably infinite, right? How many Laurent sequences are there? You can I think quickly convince yourself that it's uncountably infinite.

Now, we have a special subset of the real numbers called the integers.

Oh. Let's think of this as a decimal expansion, by the way. So we have a d, a d minus 1, and so forth, down to a0 decimal point a minus 1. These are the coefficients of decimal expansion.

There's something interesting here. Implicitly we always assume there are only a finite number of coefficients in the decimal expansion above the decimal point, right? There can be an infinite number going this way. So that's completely analogous to what we have here.

What are the integers? The integers are the 1's that stop at a0. That are all zero to the right side of the decimal point.

Over here we have something that looks like u d, D to the d plus u d plus 1, D to the d plus 1, and so forth. And in here we have a special subclass called the polynomials in D. And what are these? It's not precisely analogous, because I should really do it with Z minus 1, to make it analogous and expand it in the other direction. But these are the ones that have only a finite number of coefficients to the right of the time 0 point, again.

We noticed before that there is a very close relationship between the factorization properties of Z and the factorization properties of the polynomials. They're both principal ideal domains. They're both unique factorization domains. They're both rings of the very nicest type.

Then once we have in here the rational numbers -- that's certainly an important subset of the reals -- how many rationals are there? The rationals, this is basically a ratio of integers. And there's only a countably infinite number of rationals, right? And what's the distinguishing characteristic of the rationals in terms of their decimal expansion? It's eventually periodic.

So these correspond to the rational functions, which are denoted by this regular parentheses bracket. So this is the polynomials. This is the rational functions.

And because these are eventually periodic, or because all of these can be written as n of D over d of D, and both of these are polynomials, or these are clearly countably infinite sets, this is a countably infinite set. The set of all rational Laurentian polynomials.

So again, this is just mnemonics. Haven't proved anything. But the behavior of these things very closely reminds you of the behavior of these things over here. It's good to keep this in mind.

One other point. Suppose I'd allow bi-infinite sequences. Then 1 over 1 plus D doesn't have a definite expansion. It has two expansions. 1 over 1 plus D, if we require that the expansion be Laurent, then we only have this possibility. But if it could be bi-infinite in the other direction, then you can see that this is also equal to D minus 1 plus D minus 2 plus D minus 3 plus so forth, semi-infinitely.

So another reason we want to rule out non-Laurent sequences is to have a unique inverse. This is just another way of saying that the Laurent sequences form a field. That the bi-infinite sequences don't have the multiplicative property. They're simply a group. And this is all written up in the notes.

All right. So where I want to get to eventually here -- come back over here. Let's now analyze this convolutional encoder. It's a linear time invariant circuit. That means it's characterized by its impulse response. Or responses, if you like, because it has two outputs.

What does y1 of D, if I write the D transform of y1, what is that going to equal to? If I just put in a single impulse, single 1 at time 0 for u k, what am I going to get out up there? I'm going to get out 1 0 1 and then all zeros.

So y1 of D, the impulse response to steps d1 of D, equals 1 plus D squared. That's the impulse response of the first output. And y1 of D is simply u of D times 1 plus D squared, bi-linearity and time invariance. So if I put in 1 plus D squared, I'll get out 1 plus D fourth.

Is this too quick? This is just linear system theory. You've all seen this in Digital Signal Processing or something. This is undergraduate stuff, except for over F2. OK?

You've just got to check that it's linear, it's time-invariant. Therefore it's characterized by its impulse response. And so once you know the impulse response, you want to know the response to any sequence, you convolve that sequence with the impulse response. This is that statement in D transform notation.

OK? People happy with that? I will spend time on it if it's mysterious, but I don't think it should be mysterious.

All right. And let's complete the picture. What's the impulse response for the second output in that picture? 1 plus D plus D squared, thank you. So we have y2 of D is equal to u of D times 1 plus D plus D squared. We can aggregate this by simply saying that a little 2 vector, y of D, is equal to a 1 vector, u of D, times a little 2 vector, g of D, where this means y1 of D, y2 of D, and this means g1 of D, g2 of D. OK? So this is a very simple little matrix equation.

And now the convolutional code. This is the output sequence in response to a particular input sequence u of D. So we're going to define the code as the set of all possible output sequences when the input sequences run through what? Now, with all this elaborate set up -- so it's the set of all output sequences y of D equals u of D g of D as u of D runs through the set of all Laurent sequences. Sequences that start at some definite time.

And now I've said much more precisely what I said back at the beginning. Because all sequences have a definite starting time, we know what the starting state is for any of these sequences. Always when the sequence starts at time D, the system is quiet, it's in the all-zero state, and it's driven from time D onwards. So this convolution is well-defined. If we had an undefined starting state, then this would not be well-defined.

OK. So a long, longer than I attended lecture on Laurent sequences. But it's turned out, in the theory of convolutional codes, it's very important to get this right. People have used various definitions. They've let this run through finite sequences, polynomial sequences, formal power series. And trust me, this is the right way to do it. And I don't think anyone is better qualified to make that statement. So that is the way we define a convolutional code.

Now which direction shall I go from here? I guess I ought to ask, what are we going to allow g of D to be? In the picture up there, we have g1 of D and g2 of D polynomial, which means both causal and finite. Causal means it starts at time 0 or later. Finite means that it has only a finite number of non-zero terms.

OK. I want to allow perhaps more general types of encoders. I do want the encoder to be finite state. So let me impose that I want the encoder to be finite state. But I'm going to allow feedback. Which I don't have there. If I allow feedback, then I can get infinite responses, infinite impulse responses. But by the same argument that I've already made, it's going to have to eventually be periodic, right? If the encoder has finite state.

So we have to have g1 of D and g2 of D are general. We're going to allow up to n of these for rate 1 over n encoders. If we want this to be the impulse response of a finite state encoder -- they also have to be causal, don't they. In order to be realizable, has to start at time 0 or later, the impulse response. Then I'm going to have to require that these all be rational and causal. And that's the only limitations I'm going to put on the g of D.

So in order to get a finite state system, I need to have rational impulse responses. In order to have a realizable system, I need causal responses. So these are the requirements on the impulse responses. Otherwise I'm not going to force them to be anything in particular.

Now let's see. I go into realization theory a little bit in the notes. Since I've spent so much time on Laurent sequences, I don't think I'm going to do that here. Yeah?

AUDIENCE: Does this mean that the denominator is also polynomial?

PROFESSOR: Correct. So in fact, let me always write these. Let me take the least common multiple of all the denominators. I'm going to write these then. g of D is going to be some vector of polynomials up here. I can always write the Least Common Multiple of all the denominators down here and have a single denominator. OK?

So this is going to be my general form for a causal, rational, single input and output impulse response. It's going to consist of something like this, where these are all polynomial. This is a single polynomial with d0 equals 1. Turns out that's necessary for realizability.

And I'll just tell you the fact, which you can read about in the notes. In fact, I will leave this mostly as an exercise for the student, anyway. Realizable with nu equals max of the degree of the n_i of D, or the degree of the denominator. In other words, I take all these polynomials -- the maximum degree of any of them -- and I call that nu. And I claim there's a realization with nu memory elements. And of course, I'm going to assume here that I've reduced this to lowest terms in order to keep that down.

If I didn't have the denominator term, this would be clear. If I just had g of D equal to a set of a vector of n polynomials of maximum degree nu, then it's clear that just by this kind of a shift register realization of here we call the constraint length equal to the number of memory elements, I can realize any n impulse responses which all have degree nu or less, right? So the only trick is showing how to put in a feedback loop which basically implements this denominator polynomial d of D. If I start off by realizing 1 over d of D, then I can basically just realize this in the same way.

And nu is called the constraint length. And I have 2 to the nu states.

So by constraining this to be of this form, I've now gone full circle. The number of states is in fact 2 to the nu, and in particular, it's finite. When I constrain the impulses responses to be like that, then I guarantee that I'm going to have a finite state realization. So from now on, that's what my impulse response is going to look like.

All right. Now let's talk about code equivalence, or encoder equivalence. I've defined the code generated -- now I'm going to characterize an encoder by its impulse responses, g of D. The code generated by g of D is u of D g of D as u of D goes through the set of all Laurent sequences.

Two encoders are equivalent if they generate the same code. Seems reasonable.

What we're really ultimately interested in in communications is the behavior of the code. In particular, the minimum distance of the code, how far the sequence is separated. We're not particularly interested in the encoder. At the decoder, we're just simply going to try to tell which code sequence was sent. So for most purposes, probability of decoding error, performance of the decoder and so forth, we're only interested in the code itself. We're not interested in this little one-to-one map between information bits and the code sequences. So this is a reasonable definition of encoder equivalence.

Now, for the case of rate 1/n codes, which is all I'm talking about here -- 1 input, n output, so the code generator looks like this -- it's very simple. g of D and g prime of D are equivalent if and only if g of D and g prime of D differ by some multiple a of D, where I could let a of D be any Laurent sequence. But then to keep everything in the same ballpark, I think I'd have to keep a of D rational and causal.

But even forgetting this -- in other words, if one is a multiple of the other -- then the two codes are equivalent. That's clear because I can invert a of D. So this is just a single Laurent sequence, has an inverse. This is the same thing as saying g of D times 1 over a of D -- which is another rational, causal sequence, is g prime of D.

If this is true, then the sequence generated by u of D when the encoder is g of D is the same as the sequence generated by u of D a of D when the encoder is g prime of D. And this is another rational sequence. And vice versa. the code sequence generated by u of D, the encoder is g prime of D, is the sequence generated by u of D over a of D if the encoder is g of D. OK? So that's really the proof.

So this is very, very simple theorem. In other words, I can multiply any encoder n-tuple g of D by any rational causal sequence out front, in particular by a polynomial or 1 over a polynomial or something, and I'm going to generate the same code. This is not going to matter.

Now, to see why we might want to take make use of this -- let's see. Well, given this, we might want to pick some particularly nice encoder from this equivalence class of encoders, all of which generate the same code. And basically, I'm going to suggest that the nicest encoder is the one we get if we multiply by d of D. And if there's any common factor of these n of D's, we divide it out. That will give us the least degree polynomial encoder, which is equivalent to the given encoder. If I start off with one like this, and I want to multiply through by the denominator to make it a polynomial, and if there's any common factor to the numerator, I want to divide it out. And that's what I'm going to take as my canonical encoder.

Now let me give you a little motivation for that.

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's one motivation. It will have the simplest realization. It will have a feedback-free realization by getting rid of the denominator term. And it will have the least constraint length of all equivalent encoders, and that's a theorem too. But that's not the primary motivation.

Let me talk a little bit about distance at this point. Let me take my example encoder again. And let me ask, what's the known distance between code sequences? This is clearly going to be an important performance metric of this code, right? How would you go about answering that question, or at least two reasonable ways to go?

Do you think the known distance is greater than 1? You think it's 1? OK. So how would you get two sequences that differ only in one place?

AUDIENCE: I mean, if you take the impulse response. Or you take the --

PROFESSOR: Well, the impulse response, if I put in 1, u of D equals 1, then I get out g of D, which in this case is 1 plus D squared, 1 plus D plus D squared, I actually get a weight 5 sequence out. You see that?

AUDIENCE: [INAUDIBLE] you get 1 and 1 0 1, and [INAUDIBLE] 1 and 1 and 1 --

PROFESSOR: So I get this out, in D transform notation. Or I get 1 0 1 1 1 1 in terms of that times 0 1 1 and so forth. y1 equals this, y2 equals that. Right?

OK. So that's a weight 5 sequence. So if I put in the all-zero sequence, I get out the all-zero sequence. So as always, the all-zero sequence is an element of the code.

This differs from the all-zero sequence in five places. So all you've shown me so far is the minimum distance is not greater than five. I've found two sequences that differ in five places. Are there any that differ in fewer places than that?

The answer is no. So in fact, it's called the free distance. For this code, the so-called free distance is five.

Let's think of the two different kinds of structure for this code. And they lead to different but complementary arguments that both get us to this conclusion.

First of all, the code is linear, right? So C is linear. Is it not? If I have y of D equals u of D g of D, and y prime of D equal to, say, some other sequence. u prime of D times g of D. Then let me check the group property, which is all I have to check for a binary code. And I find that y of D the sequence, the sum of these two sequences, is the sequence that's generated by the sum of the two input sequences that led to these two sequences.

So the binary sum of any two sequences is another code word. All right? Ergo it has the group property. Ergo C is a group. And that's all we need to check. It's actually linear over F2, as a vector space of F2. 0 is in the code, and 1 times any code word in the code. So we've checked closure under vector addition and scalar multiplication. So it's linear.

What is the main conclusion we get from this? Therefore, the minimum distance between code sequences is equal to -- anybody? The minimum non-zero weight of any y of D in the code.

Exactly the same argument. We're talking infinite sequences here, but there's nothing that changes in the argument. And so that really simplifies things. We simply have to ask -- here's a non-zero sequence of Hamming weight 5. You can add this to any code sequence and get another legitimate code sequence. This is the code sequence. Add it to any other, and you get a legitimate code sequence. So from any other code sequence, there's going to be one of distance 5.

Are there any lower weight sequences in this code? Well, what would you --

AUDIENCE: [INAUDIBLE]

PROFESSOR: Good. So here's where we need to get into much more concrete arguments. You can see that first of all, we're only interested in looking at finite sequences in the code. And we might as well have them start at time 0, because if they start later, we could just shift them over by time and variance. So we're only interested in finite polynomial sequences that start at time 0.

OK. So just as you said, in the first place, they're always going to have bing bing! Two 1's. When the first 1 comes in, we're going to get two 1's out. For sure.

Time 2. This is what we get out if the second bit in, in the input sequence, is a 0. What happens if the second bit in is a 1? Then we add this to this, shift it over 1. We have 1 1 in particular at this place. We get a 1 0 out. OK? So we conclude at the second time, we're always going to get at least another unit of distance.

Now jump ahead. However long this is, this is going to have to end at some time. At the time when the last bit, the last non-zero bit, is shifting out of the shift register, we're also going to get 1 1 out. So any finite code word has to end with a 1 1 at the last time out.

People don't seem to be totally comfortable with this, so I'll do it in a more elaborate way in just a second. But this is the basis of the argument.

AUDIENCE: [INAUDIBLE] this?

PROFESSOR: At the end?

AUDIENCE: No, I mean by [UNINTELLIGIBLE] at the same time.

PROFESSOR: At the end of a finite code word?

AUDIENCE: Yes.

PROFESSOR: All right. What does a finite code word consist of? I'm going to claim that we only get a finite code word out when we put a finite input sequence in. OK? There's no feedback here. So there's going to be some last 1 in the input sequence as that shifts through here. At the very last time, the last state, is going to be 0 1. All right? Just before that comes out, there's a 0 coming in, forever after. So the last time we get a 1 at u k minus 2, and that forces these two bits out to be 1.

Or you can do it by polynomials. You can always show that you multiply this by any finite polynomial, you're going to get highest degree terms, both equal to 1.

OK. I'll do this by explicitly drawing out the state diagram. So we can conclude here that we always have 2 here, 1 here, dot dot dot dot, and finally at the end, we have to have at least 2. Therefore, every nonzero finite sequence has to have weight at least 5. Since we've seen one of that has weight 5, that is the minimum distance. OK?

Now let's do this by drawing the state diagram. Which is where we're going.

AUDIENCE: So I know that [INAUDIBLE]

PROFESSOR: No. We had to construct a little argument here, and in effect, make a little search. And this simple argument wouldn't work for more complicated cases. So let me show you how to attack more complicated cases.

We want to draw a finite state machine. How do you analyze finite state machines? Well, a good way to start is to draw the state transition diagram. And I don't know what the curriculum consists of nowadays, but I assume you've all done this.

We draw the four possible states. 0 0. 1 0, we can get to from 0 0. 0 0, if I get an input of 0, then I'm going to put out 0 0, then I'm going to stay in the 0 state. All right? Because this is linear, if the 0 comes in, that's 0 0, stay in the 0 state. Or a 1 could come in. And in that case, as we've noted, we'll put out two 1's as our output, and we'll transition to the state 1 0.

Now from 1 0, where can we go? For 1 0, if a 0 comes in, then we'll put out 0 1 and we'll transition to 0 1. Or we'll get another 1 in. We would put out the complement of that, 1 0. So we get another 1 1 in this linear thing. Work it out. I'll just assert that. And we go to state 1 1.

From 0 1, if we get a 0 in, we return to the 0 0 state and we put out 1 1. So here's our basic impulse response, right down here. We put in a 1, we go through this little cycle, we come back to 0 0. Or if we get a 1 in, we go back up here, and at that point, we actually only put out two 0's. Is that right? Yeah, because it has to be the complement of this.

And if we're in the 1 1 state and another 1 comes in, we stay in the 1 1 state. And we put out what? Let me do the 0 1 first. 0, and we're in 1 1 state, then what do we get out? We get out a 0 down here and we get out a 1 up there. We got 1 0. Or if we get in a 1, we put out 0 1.

All right. So that's the state transition diagram.

Now again, let me make the argument that I made before, now a little bit more concisely, because we're going to have to stop. So every path through this state transition diagram corresponds to a code sequence, right? Every finite or semi-infinite path always start in the 0 state. For a finite code word, I have to come back to the 0 state. Correct? So I'm always going to get two 1's when I start out. Then the next time, I'm either going to go on this transition or this transition, but either way, I'm going to get at least another unit of weight. And then I can do whatever I want through here for a while, but eventually I'm going to have to come back out here.

And when I get back to the 0 state, I'm going to get two more 1's.

So that's maybe a lot easier way to say the minimum weight has to be 5. Again, if it were more complicated, I'd have to make more of an argument.

All right. Next time, when we come back, we'll talk about turning this into a trellis diagram. We'll talk about catastrophicity, which is where I was going with this canonical generator. It's just a minor algebraic point, but one you need to know about. We'll talk about the Viterbi algorithm. I think we can probably get through this chapter on Wednesday. OK?

Ashish has your midterms and the midterm solutions.