[WRITING ON CHALKBOARD]

PROFESSOR: Good morning. I think I'll start even though people are not quite all here, because it's time.

Sorry, I'd hoped to start Chapter 10 last time, but we'll start it this time. When Ashish comes, we'll have handouts for Chapter 10, the new problem set and the old problem set solutions as usual.

This chapter is about trellis diagrams for block codes. Binary linear block codes of the kind that we've seen before, it's actually a much more general notion that applies to -- well, linearity is important, although we don't see it very specifically in the trellis diagram. But we'll see that for constructing a unique minimal trellis, linearity is an important aspect. Or more generally, it's just the group property of codes that allows us to construct unique minimal trellis diagrams.

And I think it's best to motivate this chapter by an example. Here's a nice example of a code that we know very well. It has a nice trellis diagram. It's the 8, 4, 4 Reed-Muller code. Just to remind you of how we construct this code, or one of the methods. We have many methods. We take this kind of universal eight by eight -- well, it's a list of eight tuples. And it's the tensor product of this little matrix three times.

And if we create this, we get a matrix that looks like that. And we just pick the rows that have weight four or greater, and they are the generators of the 8, 4, 4 Reed-Muller code. So we'll take those as our set of generators. It's the 16 code words generated by these four generators here. This goes back to the first part of the term, but I hope you recall that fairly quickly.

Now I'd like to draw a trellis diagram for this code. I don't particularly know why yet. One idea I might have in mind is to do a Viterbi algorithm decoding as a maximum likelihood decoding algorithm for this code.

So let's see if I can come up with an efficient trellis diagram. That means one that first of all has really few states in it. Relatively few branches would be efficient as a decoding map for the Viterbi algorithm. And here's a very nice one. This looks sort of like a rate 1/2 four-state trellis diagram. Of course, it's not time-invariant, doesn't go on forever like for a convolutional code, for a block code. Block code only exists over a finite time. So we're going to get a block trellis that starts at a particular time, that ends at a particular time in a single state.

But we want the basic property that the -- the paths in this trellis are in one-to-one correspondence with the words in the code. So, how many words are there in the code? There are 16 in this code. How many paths are there through this trellis? Well, again it has a very regular structure. There is a four-way branch here. Wherever you get to, you have a two-way branch here. You have a two-way branch here. You have no choice here. So there are 16 possible ways to get through this trellis, 16 trajectories if you like.

And next let's check if they correspond to all the possible code words. Here is the all 0 code word up here. What's next? Let's check for the generators. Here is one of the generators, 1, 1, 1, 1, 0, 0, 0, 0. And it's a 1, 1, 0, 0. This is 1, 1 also. 1, 1 -- no, I'm sorry. 1, 1, 0, 0, 1, 1, 0, 0. That's right. That's this one going through here. And 1, 0, 1, 0, 1, 0, 1, 0. That's here. And what's the last one? All ones. Well, that's up here too. The all one path. Yeah.

AUDIENCE: How did you know whether to make the [INAUDIBLE] procedure?

PROFESSOR: I had side information. A genie told me. I know that this is an efficient trellis. Where we'll get to, and I hope today in this lecture, is a turn-the-crank method of constructing not only a trellis, but the minimal possible trellis. So you'll be able to see how starting from a set of four generators like this, turn the crank, and I'll produce a trellis for you. But of course you don't see that yet.

I just know that this is a trellis, and it's a nice trellis. It has nice, regular structure. You could decode this with a Viterbi algorithm, and you could see the complexity would be comparable to the complexity of decoding a rate 1/2 four-state

convolutional code. And in fact, you remember I compared our example rate 1/2 four-state convolutional code which had a coding gain of 4 dB with this code saying they roughly have the same complexity.

And when I said that, this is what I had in mind, that I could decode each of them with a Viterbi algorithm with approximately the same amount of complexity. And unfortunately this only has a nominal coding gain of 3 dB, and an effective coding gain of slightly less than that. So it's not quite as good as the convolutional code.

And at the end of the day, we're going to find out that that's kind of typical of a comparison between block and convolutional codes of the same trellis complexity. But on the other hand, this is certainly a better way to decode this code, recursive by the Viterbi algorithm, than to do a full maximum likelihood decoding of all 16 code words, which involves just brute force computing the distance to 16 code words. This is if you like a more organized way of performing the computation of how far is the receive sequence from each of the 16 code words.

So the reason we look for trellises of block codes is first of all, it's better than exhaustive maximum likelihood decoding. It is a maximum likelihood decoding algorithm since we're going to find that we can construct a trellis for any linear block code, which at worst, it's never going to be more complicated than just enumerating all the code words and doing exhaustive maximum likelihood decoding. And usually, as in this case, it's going to be a more efficient method of doing maximum likelihood decoding. So we will have a general method of maximum likelihood decoding that is more efficient than the exhaustive method.

Secondly, there's an interesting link to system theory. I won't make a great deal of this in this course. But what are we really doing here? We're representing a block code as a finite state system. In fact, a linear block code is a finite state linear system, although again the linearity is not terribly transparent from this picture. So we have a linear finite state system, just as we did for the convolutional code. However, it's necessarily time varying. It couldn't possibly be shift invariant because we only have a finite time axis, if you like. Time axis only goes over four time units in

this particular picture, where I've grouped two outputs at a time.

So that's interesting, and for our purposes an interesting aspect of this is that we now get a notion of how complex is a block code. So far we've been focused on parameters like n, k, d, which are all algebraic parameters of the code itself. But we really want to know about performance versus complexity.

So how complex is the 8 4 4 code? Well, now we have a little bit of a handle on it. We can say its complexity is that of a four-state machine. So trellis complexity gives us a measure of the complexity of a block code so that we can, for instance, say the Reed-Muller codes tend to be less complex than PCH codes. And I'll try to give some substance to that as we go along. So we get additional parameters that have more to do with what we really care about, which is decoding complexity.

And finally, I introduce this subject because it's a stepping stone to where we're really going, is this notion of codes on graphs, which is the underlying concept for the capacity-approaching codes that is our goal point. Our whole goal in this course is to get to capacity, specifically on the additive white Gaussian noise channel. And the way that people have found they get to capacity turbo codes, low-density parity check codes, the underlying concept that I'm going to be framing that in is codes on graphs. And this is kind of a code on a very elementary graph. So it's a good way from here to there.

So that's why we're taking a little time to look at this subject. Even though as I say, at the end of the day, even though there's a better way of decoding block codes than the ways we've had previously, and a better non-algebraic way, it still is not going to turn out to be as good as convolutional codes, which we already know about.

Any questions on what we're doing, motivation and so forth? Yeah.

AUDIENCE: [INAUDIBLE] convolutional codes is that each node doesn't have two things coming on to it?

PROFESSOR: That's right, it's a little bit more irregular, and this trellis section doesn't look the

same. We had a completely mixing trellis section for the convolutional code. This kind of divides it into two halves which don't meet. This basically expresses the code as a certain subcode, and its coset is down here. So, it's not quite the same. Looks different. Maybe this will turn out to be interesting. Good observation. Anything else at this point?

AUDIENCE: You just said that this is not a convolutional code. Or maybe we can have more than one states, and it is a convolutional code.

PROFESSOR: Let's see, it's not a convolutional code. It's not even a terminated convolutional code, because it doesn't have the simple shift register type of trellis diagram. At least it's not the kind of convolutional code we know about. But why should it be? Let's open up our minds to more possibilities.

We don't really care. The objective that we have in mind here is we're just going to try to come up with the most efficient trellis picture that we can for block code. And we're going to use the linearity, and we're going to basically try to find as few states as possible at each time. That's going to be our measure of efficiency. That's not the only one you could think of, but it'll turn out that any notion of efficient representation comes down to the same thing. So we'll focus on a minimal state representation of a block code, and you can think of this as an exercise in system theory and minimal realizations of a linear system.

This you can view. A set of all 16 code words you can view is the set of possible trajectories of a linear system on a time axis of length 8, and we want to find a minimal realization, minimal state realization of that linear system. So that'll make sense to some of you, and it won't make an awful lot of sense to the rest of you.

So we're after a minimal, in the state sense, minimal state complexity, let's say. And let's continue to focus on this example, and let's focus on a particular time. Let's focus on the halfway point here. Where do the state times occur, by the way? They occur between the symbol tags. You can think of a state as being associated with a cut between a certain set of symbols which we call the past and another set of symbols which we call the future, again using just some theoretic temporal

language.

So if we make a cut -- let's say I do it over here. States are based on cuts between a past and a future, so the state actually occurs between the fourth and the fifth symbol here, not at either of them, if you want to draw where is the stated time. And let's start with this example. And let's ask if we could find any simpler trellis for this. In other words, let's see if we can have fewer than four states at this midpoint.

Could we possibly have fewer than four states at the midpoint? What's the key property of states? The key property of states is that once you get to a state, that then becomes a summary of all the history that you know about the past, and any past sequence or partial sequence that gets you to this state has to have the same set of possible continuations over here in the future in order for this to be a valid trellis representation.

So maybe it will help if I draw this just focusing on the central time. There are two ways to get to this central time. One is to get to what we call the 0 state. We can get there by 0, 0, 0, 0 or 1, 1, 1, 1. There's a second state which we can get to again by two paths, which are 0, 0, 1, 1 or 1, 1, 0, 0. There's a third one, which we can get to by 1, 0, 1, 0 or its complement, 0, 1, 0, 1. We'll see that each of these involves a four-tuple and its complement. And this is 1, 0, 0, 1 or 0, 1, 1, 0.

Those are the set of possible past sequences that can get to any of these four states. There are two of them.

Now what this trellis says is that from this state, regardless of how we got here, either of these two things, there are two possible continuations, which happen to be the same thing. And the property state has to be that either of these continuations is a legitimate continuation of either of the paths that it takes to get there. So we now have four possible code words that go through this particular state, that pass through this state.

And down here, similarly, I think what we have is the same set of code words. There's complete symmetry and so forth. Let's ask, could we combine these two

states, somehow smush them together into a single state? And the answer is obviously no, because 0, 0, 1, 1 is not a continuation of the all 0 sequence, or of the all 1 sequence. And 0, 0, 0, 0 is not a continuation of the 0, 0, 1, 1 sequence or the 1, 1, 0, 0 sequence.

The property of the state is the Markov property. And there are many ways of phrasing this, but this is the defining property of states, is that if two past paths have a future continuation -- I hope you understand this language as I introduce it -- in common, then all their future continuations are in common.

So I have two past paths, let's say. This is the definition of states. If we can find such a situation, then we can say that these two past paths go through the same state at the cut time, because then we can smush them together. You can think of this as our starting from, but suppose we drew all 16 code words, and we drew a -- here would be a 16 state trellis, and the first two elements of it might be 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1.

And we'd ask, can we smush these two states together? And the answer is yes because of this property, because 0, 0, 0 can also be followed by 1, 1, 1, 1. There are actually four states that we're smushing together here. 1, 1, 1, 1 and 0, 0, 0. These are all legitimate code words, and because they're all legitimate, we can basically break this up into a Cartesian product of two paths with two futures, and represent all four of them by a single state. So that's what states do.

If we want a minimal state diagram, we should do this as much as possible. It's not very well-defined right now, but we should merge states wherever this property holds. So this implies they go through a common state, and it's if and only if.

So you could see it very concretely from the trellis diagram. This is the property that we need. You can see that for this particular code, we do have the possibility of combining all 16 possible states here into 4, because we can combine them all pair-wise in this way. This is going to come from the linear property of the code. That's why everything is so symmetric. But we can't go any further than that.

So we can conclude from this is that for the 8, 4, 4 code, the minimal state space at the center has four states, has size four. For simple code like this, you can simply see what all the possibilities are, and this is the best we can do.

So do you get that? This is fundamental system theory state realization theory. I don't know where in the curriculum one gets this nowadays. Probably somewhere over towards the control side, maybe in digital signal processing, but maybe not. But we're doing system theory here. We all get that? Yeah.

AUDIENCE: [INAUDIBLE] properties basically defined, I have two possible paths, and they have one common [INAUDIBLE] data, they have [INAUDIBLE]

PROFESSOR: Yeah, if they go through a common state, that implies this statement. Conversely, if we have a set of past paths and future paths that have this property, then we can define a state. In other words, states are always going to look something like this, with a set of past paths. We have a set of future paths, and the state is simply a node that they go through such that you can combine any of these paths with any of these futures. If it's symmetric, you could say the same thing. If two futures have a past in common, then they have all their paths in common.

But whenever you have a state representation -- let's suppose we have three states there. Then this is what, just looking at those states and the initial and final states, this is what the code is going to have to look like. Because property of state is that however you got there, you have to be able to take any of these over here. Otherwise the trellis is invalid. If not all of these nine sequences are code words, then it's not a valid trellis representation, or equivalently, it's not a valid state representation.

So if, and only if, we can draw the thing in this way, we get it. So for a small code, you can just see how much combining you can do. In particular, you can ask, what can the 0, 0 state -- 0, 0 path sequence -- what can that be combined with? Well, it can only be combined with sequences that have a 0 first part.

So let me introduce now the idea of subcodes. We have a linear code C. Let's

define an interval on the time axis: k, k prime, whatever. The subcode C on that interval is the set of all code words. Elegant way to say it is whose support is in k, k prime.

In other words, which are all 0 outside, which have all 0 symbols outside of this interval that we've identified. So that's just notation. So specifically, let's say the past code with respect to a certain time, like the midpoint there. Well, we can define that as the set of all code words that are 0 outside the past, the first four symbols, and we define the future subcode as the set of all code words that are 0 outside the future.

For this code, what is that? We list the 16 code words. We make this the past. We make this the future. What is the past subcode here? In this particular case, the past subcode is, of course, the all 0 sequence. It was always in this code or this sequence, 1, 1, 1, 0, 0, 0. It's a linear code of dimension one, and it consists of these two code words.

It is precisely the set of all code words that can be followed by the all 0 sequence. So I can read it directly off of this trellis picture here. So everybody with me? The past subcode just consists of those two words. The future subcode consists similarly. Looking at the trellis, it's the 1's that are all 0 outside the future. In other words, can follow a past which is all 0. In other words, it's these two code words.

So here are two little subcodes with a code. So by definition, this is subcode of C. If C is linear, you can quickly convince yourself this is a linear subcode, so it's going to have a certain dimension. It's going to have size equal to a power of two. It's even going to have a minimum distance that is at least as great as the minimum distance of C, because it consists of code words of C, so its minimum non-zero weight is going to be at least as great as that of C. So it has some properties immediately.

These past and future codes seem to have a lot to do with the structure of the trellis up here. In fact, the zero state consists of precisely -- the sequences that go through the zero state here are not coincidentally the set of all past code words in Cp followed by the set of all future code words in Cf.

And why is that? It's because the set of all ways of getting to here, getting to this state, have to be the set of all code sequences that can be followed by the all 0 sequence. Similarly, the set of all continuations from this state have to be the set of all continuations of the all 0 sequence.

So the zero state is always going to be a little sub-trellis that is going to represent in effect -- you can draw it as a sum, or a product, of the past and future subcodes. In other words, there's a two-dimensional code that has these two generators. g1, g2 generates a little two-dimensional code. Here's the fourth code word in it, the all 1 code sequence. And every element in that two-dimensional code goes through the zero state.

AUDIENCE: [INAUDIBLE] code word. So then you know --

PROFESSOR: It's whatever it happens to be.

AUDIENCE: Then you have a separate state just for -- you could combine them.

PROFESSOR: It doesn't matter. Suppose that Cp has dimension one. Cf has dimension one. That means Cp is going to be something like this. Cf is going to be something like this. Whatever, what way.

AUDIENCE: [INAUDIBLE]

PROFESSOR: It just follows from linearity. And clearly, since C itself was linear, we're allowed to add x, x, x, x all 0's to y, y, y, y. That's a code word, because these were both code words in C. So the linearity allows you to fill in this fourth corner of the rectangle, if you like.

So Cp plus Cf is always going to look like this. And we'll always call this the zero state, the state that you get to by the all 0 sequence. And that can be followed by the all 0 sequence. That's always going to be called the zero state. By linearity, it's always going to look like that. It might not be dimension one. It could have any dimension here.

So we're really beginning to get somewhere now. What do all these other states look like, grouped theoretically? Cosets. Somebody -- who said cosets? Good, that's right.

So Cp plus Cf is itself a subcode of C, a two-dimensional subcode. So there are four cosets of Cp plus Cf in C, and we'll see if they correspond to the four states.

So how do we do this algebraically? Let's draw a generator matrix for C in a certain form. Again, I'll draw the past, the future. So that's the only division I'm concerned with right now. And let me draw it in general form first. I've defined this past subcode, Cp, at a certain dimension, so it has a certain number of generators. So we're going to put up here a set of generators for Cp, g of Cp, however many there happen to be.

And what's their common characteristic? They all are all 0 in the future. So they're all going to look like that. Then I'm going to take a set of generators of Cf, and similarly I'm going to use them up over there. These are all code words, and they're clearly linear and independent of those. So I'm on my way to constructing a generator matrix for C.

But obviously I'm going to need some more. We can call this the number of generators in the past, the number of generators in the future. This is the dimension of Cp. This is the dimension of Cf. And now I need some more generators, which by definition are going to have to span both past and future.

Let's see. Since I've already introduced the coset language, let me just mysteriously put that as the generators of a quotient group. C mod Cp and Cf is what that means, and it's a quotient group, if you know or recall what a quotient group is. But anyway, there are k minus kp minus kf of these. And I don't particularly care what I put down in here. Their property is that they have to span both past and future.

So for our particular example, here's the generator for the 8, 4, 4 code. What we're talking about is a generator that looks like 1, 1, 1, 1. That's k past is 1. 0, 0, 0, 1, 1, 1, 1. k future is 1. They're both one-dimensional codes.

Now I need two other generators, which must span past and future. So what'll I take? Let me just take two more. 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0. That would be one possible choice. And so this is k minus kp minus kf, which is 2.

What I'm going to claim is that I'm going to need a state for every possible linear combination of these generators. So claim -- well, let me break it down into more logical sequence. Every code word in C can be written as some linear combination of the generators. What does that amount to? It amounts to a certain code word in the past code plus a certain component in the future code, either all 0 or they're all one, plus -- let me call this the state code, S, as I do in the notes.

So in this case we have a two-dimensional state code. Its elements are eight-tuples. It's just a subcode of the code we started with, so it's not quite clear how it corresponds to states yet. Well, let me just call that S for state sequence.

And I claim I can break this up because the proof is that for every c in C is equal to a sum of -- it's a linear combination of the generators. So I just break that up into the three possible parts. So I get one part that's in the past subcode, one part's in the future subcode, and one is in the state. So this is the past, this is the future, this is the state. So I just break it up according to this. So the 16 code words can all be written in this way in our particular example.

So let's project this code word on the past. What does projection mean? It means you just don't look at the part that's in the future. So we're looking at four-tuples now. So C projected on to the past, this is the part that lives over here. It is equal to -- well, if I project the past code word on the past, I basically get the past code word again. Let me write it like that, but it's a little bit redundant.

Incidental comment: I can regard this past subcode as either an 8 1 4 code, but what is it really? It doesn't live out here. It's support is on these four. It's really a 4 1 4 repetition code that lives on the past. It's support is the past. Similarly, the support here is the future, and it's effectively a 4 1 4 repetition code for our example.

But formally, if we project this on the past, it's a one-to-one projection. What

happens to the future part? This disappears, because the projection of anything in the future code on the past is all 0 by definition. Plus, it's the projection of the state sequence on the past.

Could the projection of the state sequence on the past be all 0? This is actually an important point. Could the the projection of any linear combination of these generators down here be all 0 in the past?

The answer is no by the definition of the future subcode. If I found a linear combination of this that was all 0 in the past, I should add that to the generators of the future subcode. So by defining this in this way, I've forced this to be nonzero, unless, of course, the state sequence itself was 0. The all 0 sequence is in the state code. So this is nonzero if S is nonzero.

And similarly, if I project a code word on the future, I get -- this becomes all 0. I get the element of the future code projected on the future, plus the state on the future, the state projected on the future.

And now I claim further that this means I can draw a trellis as follows. Start from here. For the all 0 sequence, I'll have a bunch of parallel paths going to the zero state that together add up to the past subcode. These are precise. One of these is going to be all 0, and these are precisely the ones over here that can be followed by all 0, as I've already claimed. So these are the ones that can be followed by 0, and similarly over here I'm going to put all the future subcode.

I will say these correspond to the state sequence 0. These are the ones where if I put 0 coefficients down here, I just get linear combinations of the past subcode and the future subcode. I'm going to write those as going through all one trellis state. And I believe I've already made the argument that this state at least is legitimate, that every combination of something in the past subcode with something in the future subcode is a code word. This is simply just Cp plus Cf, what I wrote before. So I've got one state that represents all of Cp plus Cf.

And now let's take another typical state down here. We'll say this state corresponds

to the state sequence S, or S is a general state sequence. Think of it as being nonzero. And what am I going to put on that? I'm going to put Cp plus this state sequence projected on the past. And that way I'll get all past projections that are of this form for a specific S projected on the past, allowing this to vary through the past subcode.

And similarly over here, I will let the set of all these trellis branches be Cf plus the state projected on the future. So all the things that go through this state here will be Cp plus Cf plus this particular state sequence. So for each of the state sequences, I claim I can define a state. So it's the set of all past continuations of anything that projects on the past as anything in the past subcode plus a past projection of the state sequence can be combined with anything in the future, Cf plus the future continuation of the state sequence.

One of these is simply the past projection of the state sequence plus the future projection of the state sequence. And this is S, and by construction this is a code word in the code. Now by linearity I can add any past code word to this past projection. And that's a code word, so any of these elements plus Sf, continued by Sf, the future projection is a code word. See, there we're taking something generated by Cp and the state code. And similarly, anything by here comes out here.

So I think I've left one or two details undone, but are you convinced that I can define a state in this way such that all of these pasts can be followed by all these futures, and they're code words? In fact, they correspond to this subset of elements of the code.

We have examples of it up here. For instance, 0, 0, 1, 1. Or let's see. Let's take a past protection, 1, 1, 0, 0. Here's a typical state sequence, 1, 1, 0, 0, 1, 1, 0, 0. And I can add to this anything in the past, so I get 0, 0, 1, 1, 1, 1, 0, 0. I can add to it any of these two, anything in the future. So I get 1, 1, 0, 0, 0, 0, 1, 1 and 0, 0, 1, 1, 0, 0, 1, 1. And I claim that all four of these have the state projected on the past equals 1, 1, 0, 0. The state projected on the future is 1, 1, 0, 0.

And that any of these pasts can be followed by any of these futures. So this is C past plus C future plus 1, 1, 0, 0, 1, 1, 0, 0. That is the claim.

And furthermore, if I go through this, dot dot dot, I'm going to generate everything in C. So this is the partition of C -- into what? S cosets of Cp plus Cf, subcode of C. So algebraically, that's what's going on.

But having done this partition, at least with respect to this state space, I can create something which has S states the size of S, which is -- what is this? The dimension of S is the dimension of C minus the dimension of Cp minus the dimension of Cf.

So I've argued that I can get a state space with this dimension. It's a linear state space, and it has a certain dimension, which is just the dimension of C minus the dimension of the past subcode minus the dimension of the future subcode. It's the number of generators that I need for S. So in the example, I need two generators for S, so I get a state space of dimension two, or size four. It's just a vector space over F2 in this case.

You all with me? I think you are. I don't see any great puzzles.

So I can get at least these few states. Could I get any fewer? Can I possibly merge any two of these states? Could I draw a trellis in which, say, I've mushed the zero state together with one of these nonzero states?

And the answer to that is clearly no by the definition of the past and the future subcodes. These are all of the future sequences that can follow the all 0 past sequence. There can't be any more down here, so we can't possibly mush them together.

And similarly, just by going through this, if you can't take the past part of one state and combine it with a future part of another state sequence, because that clearly is not in the code. And I don't have a slick proof of that in mind at the moment, but there is one in the notes.

The conclusion from this is what I call the state space theorem. Given code C, any

partition of the time axis, the total index set, into past and future, I can do this not just at the midpoint. I can do this any point along the code that I want to, any partition into past and future. The minimal -- we get a linear state space, in other words a vector space, S. And the dimension of S is equal to the dimension of C minus the dimension of this past subcode minus the dimension of the future subcode. So we can simply calculate what the minimal dimension of a state space is at any point along here.

So for instance, at this point here, what's the minimal size of a state space? What's the past subcode here? What's the set of all code sequences that are all 0 in the future, as we have all 0's out here? This point, Cp, is just 0, 0. It just has dimension 0. Yeah, 0

What's the future subcode? This is the subcode which has support all out here. This is anything that starts with 0, 0. This is 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1. Let me just write down generators. 1, 1, 1, 1, 0, 0. So there is -- this is dimension Cp equals 0. Dimension Cf equals 2. And so the minimum size of the state space is 4 minus 2. That's again 2.

AUDIENCE: Does this mean that [INAUDIBLE] each state's the number of states is the same?

PROFESSOR: Certainly not true in general. Let me ask about time three. Can certainly do that. Time three, what's the past subcode? This is the code words that have support on the first three symbols. Again, that has to have dimension 0, because the minimum weight of this code is 4. So there can't be anything that just looks like 1,1,1 and then five 0's

What is the future subcode for that? Now the only generator of the future subcode is 0, 1, 1, 1, 1. We have to go this far, and then it's what futures can we get from here? There are only two words that are in the future. So in that case we have dimension past is 0. Dimension future is 1.

So if we drew a state space here, we would have to have eight states. In fact, I can do that if I just draw a little state in the middle of each of these lines. I get now a

trellis picture which has an explicit state at this third time here, but it has eight states.

Oo ugly, so I've masked that. I've suppressed that by this nice little four-state trellis. But yeah, if you insisted on drawing a state space at time three, it would have eight states in it. We'll get back to that when I get to this turn-the-crank procedure, if I do, of getting to minimal trellises.

So what have we done? We've established that there is a uniquely defined minimal state space size for any partition between past and future. In other words, if there's any time where you want to make a cut in the time axis between past and future, you can define what the minimal state space size is.

And now you might ask the question, can we simultaneously achieve minimal state spaces at all times? In other words, can we draw a single trellis which gets the minimal state space at each time?

We haven't proved that there is. It might be that if you push in the state space at one time, it forces the state space to balloon out at another time. And in fact, for nonlinear codes, that is typically what happens. But for linear codes, a minor miracle occurs. And the answer is yes.

So how am I going to prove that? I'm going to prove it via this very handy tool, which is also a construction tool which is called a trellis-oriented, or a little bit more formally, a minimum span generator matrix.

The idea here is, given a generator matrix, reduce to trellis-oriented form. We're going to prove that this is unique, or effectively unique, and specifies minimal trellis at all times in a certain way.

Let's start again. So loosely, what is a trellis-oriented generator matrix? Well, if we have a generator, say 0, 0, 1, 1, 1, 1, 0, 0, its span is this interval from its starting time to its ending time. It's this. We say it starts at time 3, it ends at time 6, the span is the interval 3 to 6 in this case.

So we think of this as not being active before time 3. At time 3 we get the first interesting thing happens. A 1 comes up, but then goes through something. These don't have to all be ones. It could be 1, 1, 0, 1. Then there's a last 1 at some point, and then it quiets down, and it's dead again.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes. Except I'm not sure. Would you say the support of this was 3, 4, and 6, or would you say it was the interval from 3 to 6? I don't know. What's the definition of support?

AUDIENCE: I don't know it.

PROFESSOR: I'm not sure I know. It's probably been defined both ways in different literatures. So I'll call it a span, but it's roughly the support of the interesting part.

So that's the definition of span. And what a trellis-oriented general matrix is, a minimal-span matrix, meaning all generators have a short a span as possible. I'll just say, are as short as possible. This is a lecture. I can be loose. Now it's not even clear that's well-defined yet.

But let me again give you an example, and show you how to find a trellis-oriented generating matrix just by example. Let me take our example here, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1.

So there is a generator matrix that's not in trellis-oriented form. In particular, I can see that by adding this to this, I can make this generator have shorter span. I can replace it by one, which is four 0's and then four 1's, which we've already found useful in constructing a trellis. But let me proceed through it more systematically. How can I make -- this generator generates all words in the code. So how can I find a set of linearly independent generators that has shorter span?

There's the sort of greedy way of proceeding. Let's take the first two generators. Can I combine these to find anything that has a shorter span, that I can replace one of the generators with? Anybody?

AUDIENCE: No.

PROFESSOR: No, I can't. That's not the answer I expected to hear.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Let's identify what the spans of these are. That has a span of four. That has a span of six. That has a span of seven. This has a span of eight. What I'm trying to do is to reduce that.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So if I add the first two, that seems like a good thing to do, because then I get 0, 0, 1, 1, and I've reduced the span of that generator.

Any way I can combine these two to reduce the span? Obviously not, because the span of this is completely included in the span of this. If I add these two together, then I'll get something which still starts here and ends here. Why was I able to reduce the span here? Because the spans had the same starting time, so I got a cancellation of the first bit. I got a 0 up here. That's what I wanted.

So that's the key to proceeding. If I see any place, any two that have the same starting time, then I can add them together, and the result will be something that has a shorter span than at least one of them. So let me do that. I see two here that have the same starting time, and so I add that to that, and I get 0 1 0 1. So I've got the span down to here. I can do that down here. And I get this, which is a very nice reduction in span.

Can I go any further? I can clearly look for the same thing in ending times, if I could find two ending times that were the same. I could do the same trick, add those two generators together, and the result would be something that was shorter than at least one of the component generators. But here are my starting times, here, here, here, and here. They're all different. So I have no possibility for combining two and getting a better starting time.

Similarly, the ending times are here, here, and here. They're all different. So I'm done.

So, that's how you find a trellis-oriented generator matrix. And there's a closely associated theorem, which is that a generator matrix g is trellis-oriented if, and only if, all starting times differ, all ending terms differ.

Yeah?

AUDIENCE: What's the definition of all generators as short as possible?

PROFESSOR: I'm now making it. This is really just motivation. So this can be my actual definition of a trellis-oriented generator matrix, which I've proved this theorem. It's a matrix in which all the starting times are different, and all the ending times are different. And what I've achieved is that all generators are as short as possible.

How would I go about proving the theorem? Suppose I have a matrix that's not in this form. And I've already proved that I can shorten the generators. So if it's not in this form, it's not trellis-oriented. The generators are not as short as possible.

So all I need to do is prove the other side. Suppose it is in this form. Then could I possibly find shorter generators? And the proof of that is basically the following lemma, that the subcode C, take any interval kk prime. So given a trellis-oriented generating matrix, namely one that satisfies these conditions, then the subcode consisting of all the code words who have support on this interval is generated by the gj in G that have support in kk prime.

And that's kind of intuitive and obvious. On the one hand, it's clear that any linear combination of these generators is a code word in the subcode. Obviously they all have support in this interval. I can't generate anything that has support outside that interval by linear combinations.

So the only question is, can I possibly get anything that's in the subcode by combining with a generator that is outside this interval? And again, it's clear that say I take kk prime to be here, if all the starting times are different, and all the ending

times are different, and I start to add generators that are outside here, I can't possibly get cancellation of the starting time, or I can't possibly get cancellation of the ending time. So I'm going to wind up with a code word which has support outside the interval. So, it really is a simple lemma.

AUDIENCE: [INAUDIBLE] have the all 0 code word?

PROFESSOR: Excuse me?

AUDIENCE: The all 0 word will be part of the subcode.

PROFESSOR: The all 0 word is always part of the subcode.

AUDIENCE: [INAUDIBLE] who has something outside this interval. Then when you add that with the all 0 one, you would get something that does not belong to the subcode, [INAUDIBLE]

PROFESSOR: Right, but I have to ask if there's any code word. So I have to consider all the code words, really. But I'm saying if I have a code word C that is sum of the generators, and I have a nonzero coefficient here on any generator that has starting time outside this interval or ending time outside this interval, then I can't possibly get cancellation of that nonzero thing. And so if it really requires this to be nonzero, and then I'm going to say that I get a code word that's not in this subcode.

So that's very quick, intuitive sketch of that proof. Again, for writing it out, see the notes.

But this lemma then proves this theorem, because it now says I can't -- if all the starting times are different, and all the ending times are different, then I can't possibly get shorter support for any of the generators. And I don't think so in a totally convincing way, but that basically is the point here. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, now I'm defining this. So, meaning that all the starting times are different, and all the ending times are different. From now on that's going to be my definition of a

trellis-oriented generator matrix. So, saying that this property implies this property.

But now this lemma has an interesting consequence. This implies for any past and future, Cp is generated by the generators in a trellis-oriented generating matrix with support in p. And similarly, Cf is generated by the generators with support in f.

So for any p and f, I get a picture that looks like this. I can draw now this boundary anywhere I want, and if the matrix was trellis-oriented, then I'm going to get a certain subset of generators that live in the past, and they'll be the generators in fact of Cp. The dimension of them will simply be the number of generators that live in the past for that definition of the past.

There will be some other set that live entirely in the future. Those will be the ones that generate the future.

And the ones that are neither wholly in the past or in the future will generate the state code. So that says I can read off the dimensions of the state spaces just by looking at this single trellis-oriented generator matrix.

So now let me very quickly go through the calculation that I went through a little bit laboriously over here. Suppose I make the whole thing the future. In other words, I draw a cut before time 0. Then the state space has dimension 0. All four generators live on the future.

If I make the cut here, the state space has dimension 1, because three of the generators live on the future. One is active at time 1. So that means I'm going to get a two-state trellis at time 1.

If I make the cut here between past and future, I see two generators live on the future and are inactive, haven't started yet. But two of the generators have already started. At this time we have three active generators. Only one is still completely on the future.

At this time in the middle I have two generators which -- past subcode, future subcode, and two that aren't. So I get this form that we looked at over there. And I

get a state space here generated by these two generators of size two.

So just by looking at this I go right through the picture, and I find the size of the state space at each time. So this is an algebraic way of finding the minimal state.

So now I can draw from this a trellis just by writing down these state spaces. Let me leave these trellises here. Let me draw now a full trellis, or an eight-section trellis. I'm only going to put one bit on each trellis.

Here is the starting state, dimension 0. One state. From that I can go out to 0 or 1. I won't label the state spaces. Next time, I'm still just branching. Actually, to make it look like that, I'm going to want to come down here, 0, 1, 1, 1, 1, 0.

I basically just have four states at time two corresponding to whatever the first two bits are. They all go to different states, because all these have possible future continuations. Then I have eight states at this time, still just branching. Sorry, this is going to take some time to do correctly. Add this time, then I come in here. 0, 1. Anyway, and it's symmetrical on the other side.

So that's what a full trellis will look like. And how did I do this trellis in principle? I wrote down all 16 code words. I wrote down what linear combinations they were up here, and I therefore found what states they went through at each time. And then I just drew the graph that goes through that describes those trajectories.

So I wrote down all the code words. I wrote down all these state codes at all these times. I could have written them down in any order. And then I just connected the dots according to which state -- I make this calculation as to which state they go through.

That's a good place to stop. Let me summarize. We now have a method, given a generator matrix for a binary linear block code, of reducing that generator matrix to trellis-oriented form. Just by inspection of the trellis-oriented generator matrix, we can determine what the state space dimensions are at each instant of time, at each possible state cut. And we can then draw a trellis which achieves that minimal state space dimension, or minimal state space size, for every moment of time

simultaneously.

And I assert that this trellis is the minimal trellis in every respect, whether you're trying to minimize state complexity, or if you're using the Viterbi algorithm. Really what you want to minimize is the number of branches, but there's a calculation about branch spaces in the notes that shows that this also achieves the minimal branch space size at every time, regardless of how you draw the trellis.

More elaborate calculations with a more refined notion of Viterbi algorithm complexities still come up with the same result. The result is there is essentially a unique minimal trellis, where it's minimal in every way. And this comes from the linear, or more broadly, the group property of the code.

So for any linear group code, there's a well-defined, essentially unique up to re-labeling, minimal trellis that achieves the minimum of whatever complexity quantity you want to define. So in this sense, the trellis complexity of a group or linear code is very well-defined. It's defined by this minimal trellis, or by its parameters, and we can use that as a measure of the complexity of the corresponding group code.

And from an engineering point of view, once we've got this minimal trellis, we can use the Viterbi algorithm to do maximum likelihood decoding. Just in the same way we did for convolutional codes, sweep from the starting node to the ending node.

Digest that, and we'll come back and talk about it again. Do a few more details on Monday, but that's the basic idea.