PROFESSOR: The topic of this chapter is finding trellis representations for block codes, in particular linear block codes. We're looking only at binary block codes, except one of the homework problems looks at MDS codes over ground field of q. The principles are the same for general Fq or in fact for group codes. We're looking for minimal trellis representations.

And so just to review our line of thought, basically we proved a basic theorem called the state space theorem that tells us that there is a vector space over the ground field at any time which we can identify with the state space. It has a certain dimension.

And we got a little formula in terms of subcodes for the dimension of the state space. We set up a little generator matrix where we isolated the past part of the code with respect to a certain cut time future part of the code, generators for that. We subtract those generators out. What's left? The number of generators is the number of generators it takes to generate the state space, in other words the dimension of the state space. So we got a theorem on the dimension of the state space at any time.

And then we found a nifty little algorithm to find a -- to determine the dimensions of the state spaces at all times, the minimal state spaces. And in fact we can use this algorithm to construct a minimal trellis realization. The algorithm involves a trellis-oriented generator matrix.

And the idea here turns out to be very simple. All you do is you find any generator matrix for your linear block code, and you reduce it to trellis-oriented form. Trellis-oriented form means that the generators have as short a span as possible. It's a minimum span generator matrix. And we do that just in a greedy fashion by any time we have an opportunity to add two generators together such that either their starting position cancels, or their end position cancels, we do it.

And out of that we get a shorter generator, and we can replace one of the two

generators in the combination by that. We can keep shortening everything up. The algorithm must terminate at the point where no further shortening is possible. And that is explicitly the point at which all the starting times of the generators are different, and all the ending times of all the generators are different.

For example, here's a trellis-oriented generator matrix. The starting times, the spans of the generators, are like this. The active times, starting times -- there's the starting time of this one, and its ending time. Here's the starting time of the next one, and its ending time. The starting time of the next one, and its ending time is here. And the starting time of this one, and its ending time is here.

We're not always going to find that the starting and ending times are disjoint. This happens, it turns out, because this is a self-dual code. In a self-dual code, you will always get this each time, either a starting time or an ending time. But that's not a general property.

For instance, another example that we've used is, say, the 8 7 2 single parity check code. What are its generators? Its generators look like this, dot, dot, dot, down to 0, 0, 1, 1. That's a set of trellis-oriented generators with starting time, ending time, the first one here. Starting time, ending time of the second one there. So in general for this one, every time is a starting and ending time up to here, where every intermediate time is both the starting and an ending time.

So it could happen that way, or another example would be the 8 1 8 repetition code. What's a trellis-oriented generator matrix for that? It only has one generator. It looks like that. It has a starting span as the whole thing. The starting time is here. The ending time is here. So for general codes, the starting times and ending times could be anywhere. There will always be k of them.

And otherwise, there's always one at the beginning. If it's a nontrivial code, there's always an ending time at the end, et cetera. But this particular behavior is for self-dual codes.

Anyway, having done that we can now read off the dimensions of the state spaces

by simply looking for how many generators are active at any particular time. For a state space, we're always looking at the times between the symbols, cut times. We could in fact draw a trivial one there, but we're looking between here.

How many are active at this time? One. So dimension of the state space at state time k is 1, 2, 3. This point, we come back to 2 again. 3, 2, 1, 0, 0 at the end. It's always zero at the ends because at that point everything's in the future, or everything is in the past.

So these are trivial state spaces. These are the nontrivial state spaces in the trellis. And in fact, you remember we had a little picture. We can draw this complete trellis out with the state spaces of size 1 2 4 8 4 8 4 2 1. So this is called the state dimension profile, just this set, this ordered set. Or the state space size profile is 2 to the dimension, of binary codes, whatever.

And similarly down here, if we look at state space sizes, we see the dimension is going to be one at every time. So in this case, the dimension of the state space is 0, 1, 1, 1, 1, 1, 1, 1, 1, 0. And similarly down here, we look at the cut time. It cuts one active generator every time. So it's 0, 1, 1, 1, 1, 1, 0. So we get at least the state space dimensions just by inspection of the trellis-oriented generator matrix.

Question?

AUDIENCE: [INAUDIBLE]

PROFESSOR: We proved it in several steps. First we have the state space theorem, which I gave to you in the following form: the dimension of the state space at time k is equal to the dimension of the code minus the past subcode. Let me write that. The dimension of the past subcode minus the dimension of the future subcode at time k relative to this cut time.

And we proved from the trellis-oriented generator matrix that the past subcode is generated by all the trellis-oriented generators that have support on the past, and the future by all the ones that have support on the future. So, it's simply we take those out. And what do we have left? We have the ones that are active at time k,

the ones that are neither wholly supported on the past nor on the future. They're supported both on the past and future. And those, that's what this difference is.

So explicitly we have -- the code is generated by all generators. That's all k generators. Cpk is generated by trellis-oriented generators supported on the past. And Cfk is generated by the trellis-oriented generators supported on the future.

So the difference -- state code sk is generated by the trellis-oriented generators not supported on past or future, or active at time k. Probably good to write all that out at least once. Yeah?

AUDIENCE: [INAUDIBLE] self-dual codes all have such properties that each time is starting and ending?

PROFESSOR: Yes.

AUDIENCE: Why is that?

PROFESSOR: Why? It's a duality theorem, and we're not doing many duality theorems in this course, so I'll simply tell you.

AUDIENCE: [INAUDIBLE] has this property, it must be a self-dual.

PROFESSOR: I don't think that's true. It's possible that's true. Duality theorems are very strong, but I don't think this is a sufficient property.

For instance, suppose I just changed one of the generators to look like that. It still has disjoint starting and ending times, but now I don't think the code is self-dual. So here's another code generated by these four generators, and I don't believe self-dual.

AUDIENCE: [INAUDIBLE]

PROFESSOR: I'm guessing that it isn't. To prove it I would compute the dual code, and I would find that it wasn't same code, didn't have the same code words. But I'm guessing that if I just make random changes in the interior here, I'm not going to get a self-dual code,

because that's a very strong structural property. But I'm not going to take the time in class to do that.

But these are good questions. The whole subject of duality is a wonderful and elegant topic that we just don't have time to do much in this course. If I did it another time, I could spend many lectures on duality. Some of the homework problems have suggested what you might be able to do in orthogonal codes, but -- well, one of the problems you'll do on the homework -- I was going to talk about it a little later. There's a dual state space theorem that basically says the state space sizes, or dimensions, are the same for the dual code as they are for the primal code.

Here's one example. Of course, dual is itself, so it's trivial that the state space size is the same. But here's another less trivial example. These two codes are orthogonal to each other. And what is similar about their trellises? Their trellises are both two-state trellises. They don't look the same. This trellis looks like this, because they're all zeroes up here, and all ones down here. That's the repetition code trellis, whereas this trellis looks like this. It has crosses every time and so forth, abstractly.

So trellises are not the same. They have different branching properties, but the state space sizes are the same at all times. This is a general property of dual codes which you will prove on the homework, if you haven't done so already.

Sorry, it's hard to know what to tell you about and what to suppress, because I'm also trying to cover more material this year. There's a lot of material that's really nice that I have to skip over.

Let's continue from this. Let's make a minimal realization. So, a canonical minimal trellis. I'm doing this in a little bit different order than it's done in the notes, but I think you will see this kind of argument very quickly.

I've got this trellis-oriented generator matrix, but suppose I didn't know what the trellis looked like. I put it up the first time last time. But all I have is this generator matrix, and I want to generate a trellis now for this code which has minimal state spaces. It's also going to turn out to be minimal in every other way.

Well, let's start off by realizing the first generator here. The first generator generates a little one-dimensional code. G1, the code generated by the first generator, is simply the two code words 0, 0, 0, 0, 1, 1, 1, 1, 0, 0. So let me realize that little one-dimensional code with a minimal trellis for that. What would it look like?

If I go through this, I'll see that it has -- my trellis-oriented generator matrix for this code is simply 1, 1, 1, 1, 0, 0, 0. I see from that that the dimensions of the state spaces is 0, 1, 1, 1, 0, 0, 0, 0, 0. And you can easily see that what the trellis, the minimal trellis, a minimal trellis looks like is 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0. So there is the minimal trellis realization for that one-dimensional code.

It has state spaces of dimension 0, 1, 1, 1, 0, 0, 0, 0, 0. Therefore the state spaces are all minimal, as proved from this. It's all sort of elementary.

Similarly, I can realize this one by another little machine that looks like this. 0, 0, 0, 0, 0, 0, 0. This is just the state transition diagram of a linear time-varying finite state machine. And it looks like that, and it too is minimal by the same argument for the one-dimensional code generated by this generator.

What is this saying here? This is saying we really -- what we should imagine here is we have a little memory element that's only active at state times 1, 2, and 3. So to realize this, this is a one-dimensional memory element. Think of it as a little -- well, a memory for an element of f2. It's active during the interval 1 through 3, during the time that this generator is active. It's active at these three times, not otherwise.

So you can think of it as just disappearing. It comes into existence at time 1. It holds whatever the input bit is. Think of -- we're trying to create the code as the set of all -- sum of ui gi. And so what this memory element really does is it holds ui during the time that it's needed, which is at times 1, 2, and 3. Whatever the coefficient is -- what could it be, 0 or 1? Element of the base field, f2. So the memory element holds, it contains u1 in this case.

Yeah?

AUDIENCE: What if we look at this matrix only for the second code.

PROFESSOR: Yes.

AUDIENCE: So that will give 0, 1, 0, 1, 1, 0, 1, 0.

PROFESSOR: Correct.

AUDIENCE: So some places it will suggest that the dimension is --

PROFESSOR: The fact that the values are -- there will always be 1 here at the starting time and the ending time. Otherwise we wouldn't draw a branch there. But they can be arbitrary in between. So don't be thrown off by the fact that there are some 0 values in here. What we're doing here is we're realizing u2 times g2, where u2 can be either 0 or 1. And g2 can be arbitrary, except that it has a starting time and an ending time, so it's not arbitrary for this kind.

AUDIENCE: What if we try to find the state space [INAUDIBLE]

PROFESSOR: For this, what I need is a one-dimensional memory element containing u2 which is active during a different interval, 2 through 6. So think of these as like little stars that wink on or wink off. This thing comes into existence for three consecutive times, and then it disappears. This thing comes into existence for five consecutive times, and then it disappears. Because those are the only times when it's needed to contribute to the output.

So altogether, we can think of there as sort of being four memory elements which contain u1, u2, u3, and u4. And we form the outputs g1k times this, sort of like in the convolutional code. g2k g3k and g4k. The generators step along in time, and we form the outputs. We sum this all together in a summer, and this provides yk, the output, as we go from k equals 1 through 8 as we go through the code word.

But we only need this one for -- it's only on from time 1, 2, and 3. And this is on from time 2 through 6. And this is on from time 3 through 5. And this is on from time 4 through 7. Because we don't need it any other time. So you can think of an input coming in at time 1, at the starting time of the first generator, being saved for as

long as it's needed, then it's discarded. And in fact we don't need that memory element anymore.

How many people are getting this? Raise your hand if you get this.

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's a time -- what I'm trying to do is draw a time-varying machine. So how do I do that? At time 1, it looks like this. Time 1, it just has u1, and we multiply it by g11, which is 1. And that gives me the output at time 1. There's nothing else that contributes.

At time 2, we've got now another input, u1 and u2. And we've got to sum the contributions from both of those, times their respective coefficients. At time 3, we're up to three of them, and we need all of them. We're going to form some linear combination according to the generator matrix.

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's time-varying, yeah. So get used to it. For a block code it obviously has to be time-varying. Obviously if we regard these as impulse responses, this is the impulse response to u1. It only lasts in terms of symbol times from time 0 1 2 3, for four time units of symbol time.

This is the impulse response to u2. u2 you can think of as an input that comes in at time 2, and whose output rings for five more times. So you need to save u2 for five more times, and then it disappears. We don't need to save it anymore. We can let it drop off the end of the shift register, if you like.

u3 comes in at time 3, and this is its impulse response. And u4 comes in at time 5, and this is its impulse response. Does that help? How many people are not getting it now?

AUDIENCE: [INAUDIBLE]

PROFESSOR: We don't need it here, but we're going to need it again at time 4, so we have to save

it. What are you going to do with it at time 3? You going to hide it somewhere? So I still need to save it.

This, again, is linear system theory, except I guess you just don't get taught linear system theory anymore. But this is what you have to do if you -- before, for convolutional codes, we did linear systems theory for linear time-invariant systems. They have to be over a finite field, but who cares? We did general linear system theory. This is general linear system theory for realizing time-varying systems.

In fact, one comment that might be good to make at this point is that -- now go back and remember what we did with convolutional codes. We started out with a rate 1 over n convolutional code c, and we found some generator for it. Basically we can take any code word in the code, and all of its shifts will generate the code. Any non-zero code word, all of its time shifts generate the convolutional code. So take any code word g of d in a convolutional code, and g of d, dg of d, d squared g of d and so forth generate the code.

But then we converted it by doing a little polynomial or Laurent series algebra. We took a rational one and we converted it to a canonical one, g prime of d. Which if you remember, this is the polynomial code word of minimum degree in the code. We call that degree nu.

What were we doing there? The shifts of g prime of d form a trellis-oriented generator matrix for this convolutional code. For instance, for our canonical example, g prime of d looked like this. Polynomial notation, it's 1 plus d squared 1 plus d plus d squared is the generator matrix. Let me write that out as 1, 1 at time 0 or coefficient 0. 0, 1, 1, 1, and then all 0's before that, and all 0's after that.

Is somebody sleeping? Wake him up please, before I turn around. What? Oh, all right. I'm very glad to hear that.

So here is g prime of d written out as bits. dg prime of d looks like what? It looks like the same thing shifted over one. d squared g prime of d looks like this and so forth.

Is this trellis-oriented? In this case, each time interval contains two symbols. Here

are our divisions between symbols. We've got a rate 1/2 code, so we chose to take them two symbols at a time.

Where are the starting and ending times? This one starts here, ends here. This one starts here, ends here. Starts here, ends here. So all the starting times and ending times are different. Ergo, this is a trellis-oriented generator matrix. Modulo some hand-waving about infinite sequences.

Ergo, we ought to be able to use the state space theorem to say what are the state space sizes. Any time I have a cut here, what are the spans? The spans are like there, there, there, so forth. So the state space dimension at each time is going to be 2, 2, 2, 2, 2.

And if we go through this construction procedure, what are we going to get? We're going to get something that at time 0, we need to have in memory, we need to remember u minus 1 and u minus 2. At time 1, just doing what I did up here, we need to have in memory u minus 1 and u0. At time 2, we need to have in memory u0 and u1. And we achieve that just by implementing this with a shift register. In the time-invariant case it's very easy, and you all understand it.

So exactly the same theory goes through for rate 1 over n convolutional codes. I will tell you briefly that to treat rate k over n, it's basically just the same thing, except we need k generators starting at each time here. But in order to get a canonical encoder, we just get a trellis-oriented generator matrix again. And by the way, modulo this hand-waving about infinite sequences, I've now proved what I asserted, that this encoder is minimal among all encoders for this code, that it has the minimum number of memory elements. Because from this argument, the minimal state space dimension at each time is 2 once I've achieved this canonical trellis-oriented generator matrix.

By the way, this theory is very general. It's really all you need to know to do minimal realization theory of linear systems, whether they're time-varying or time-invariant over any field, or over groups.

So that was a parenthetical comment about convolutional codes just to tie it back to what we did in the previous chapter. I haven't completed what I wanted to say about block codes. How am I going to do this without erasing something? Let's go back to block codes. And I guess I'll move over.

The point that I was in the process of making was that I can realize each one of these generators individually by a little trellis diagram that's one-dimensional during the time that the generator is active, and zero-dimensional state space during the time the generator is inactive. And think of this as just one component here.

To generate a code word, a particular code word is just the sum. All I need to do is sum the outputs of all these generators independently. How many different possibilities are there? There are 2 to the k possible code words. Each one of these generates one of two possible code words. If I take the set of all possible sums of these four code words, I'm going to have 16 possible code words. So that very quickly was what I was doing when I said, here's a little machine that generates the first code, c1. Here's a little machine that generates the second code, c2. Only has two possible code words according to the value of what's in storage. This generates the third component. This generates the fourth component. Sum it all together, and you get -- sorry, I'm calling this y now. y equals the sum of the ui gi. You see that? I hope I've beaten it into the ground enough now.

So it's a linear time-varying machine. That is a realization of it, if you've understood my description of it now. And now we can draw the trellis of it. That what can happen at time 1, we can either have u1. Now I can label my states here, at this time u1. u1 can be either 0 or 1. This is u1.

At time 2, I have both u1 and u2 in my state. So this can either be 0, 0 or 0, 1, or 1, 0 or 1, 1. And obviously if I've already determined u1 is 0 here, it's still going to be 0 at time 2.

I label these with their associated outputs. This is 1. This is also 1. This can turn it back to 0. And I just keep going. At the third time, I have u1, u2, u3. I have eight states. At time 4, I come back. I only have u2, u3 at time 4, and so these merge. So

I get something -- this doesn't merge like that, though. It merges down here. 0, 0, 0, 0, 0, 1. This merges to 0, 1 when I drop the first one. It's probably worth doing this more carefully than I'm doing it. 0, 1 can go to 0, 1, 0, or 0, 1, 1. 0, 1, 0 can either go to -- u2 equals 1. I guess these can't cross. u2 --

AUDIENCE: [INAUDIBLE]

PROFESSOR: Excuse me?

AUDIENCE: [INAUDIBLE] u3 and u4.

PROFESSOR: No, this is state time. State time 1, 2, 3. u4 hasn't started yet. So I'm sorry, this is 5 through 7. Is that right? It only lasts for three intervals. So I only have u2, u3. u1, u2 has to be 0, 0, 1, so this would then go to 0, 1. 0, 1. These seem to cross. Yes, they do cross.

So let's see if I can actually complete it. This could go to 1, 0, 0, or 1, 0, 1. And then this one goes up to either 0, 0 or 0, 1. This one can go to 1, 1, 0 or 1, 1, 1. And that can go like this.

So there's the trellis for the first half. It's topologically the same, but looks a little different from what I put up before. I interchanged these two with these two, and this one with this one. But you can recover the trellis that I had before. In any case, I claim when I've labeled this with what the appropriate output is at each time, I've got my minimal trellis.

And now we can make more comments about how the trellis works. You see that whenever a generator starts, we have what's called a divergence in the trellis, a branching process, a branching out, whereas whenever a generator ends, we have a convergence in.

So a starting time -- if we have a generator starting, then we get a branch that looks like this, like here, and here, and here. When we have an ending time, we have branches that look like this, like here. So this is a starting time, starting time, starting time, and ending time. So this tells you when things open and close in the trellis.

Now again, as I've said for these more general trellises, down here we can have starting and ending at the same time. And in that case, we get a cross. We get a starting and ending at the same time, we get something that looks like this. And if we get neither starting nor ending, empty, then nothing happens. We just get -- the trellis segment looks like that, like this one here. In this case we have a starting time, an ending time in this trellis.

Here we have nothing happening. Here we have nothing happening. So you just get an extension, a parallel extension where nothing happens. So again, the starting and ending times tell you what happens in the trellis.

There's one other thing that you can read from the trellis, which is the size of branch spaces. And since as usual I'm running a little later than I intend to -- let me just state without proof that there is a similar theorem for branch spaces.

What is a branch? When we look in a trellis, a branch is one of these things. That's a branch. What do we need to define a branch? We have a initial state, the code output at time k. We're calling that y time k in the next state. So it's that triple. Again, this is linear system theory. State, output, next state. Those are the three things that define a branch.

This branch is state 1 0. Output whatever it is, y k, and next state 1 0 0. And that differentiates it from this down here. If we define the branch as this triple, then we can add triples. States form a vector space over the ground field. These are simply symbols over the ground field. These are vectors over the ground field. So we can add them.

And we find that it's closed, and the set of all branches is a vector space. It's fairly easy to show. Look at the notes, but it's basically because these are all projected from code words in the first place, and code words form a vector space, so this is just a projection of the code words, a certain kind of projection onto the state spaces and symbol spaces and so forth. A projection of a vector space is a vector space. Roughly the proof.

And what is the dimension of the branch space? Again, I'm just going to give you the answer. Suppose we have a trellis-oriented generator matrix. Let's suppose we want to find out the size of the branch space at this time. What doesn't enter into the branch space? What doesn't make a difference to the branches?

All the pasts that arrive at the same state at time k are equivalent. So this generator doesn't enter in. All the futures that depart from time k plus 1 don't enter in, so anything that's supported by the future at time k plus 1 doesn't enter in. And what's left is all of the generators that are active at symbol time k. Notice the symbol times occur between the state times.

So certainly this is an easy -- this is one of those results that's easier to understand than to prove, and I'll refer you to the notes to prove.

The dimension of the branch space is simply the number of generators that are active at symbol time k. Because branches are synchronized with the symbol times, not with the state times.

So how does that come out over here? There's the dimension of the branch space is 1 here. It's 2 here. It's 3 here. It's 4 here. I'm sorry, it's 3 again here. This one hasn't started yet. It's 3 again here. It's 3 again here, 2 here, 1 here. That means that the size of the branch space is 2, 4, 8, 8, 8, 8, 4, 2. Is that right? There are two branches here. There are four branches here. There are eight branches here. There are still eight branches here. And it's symmetric on the other side. So it looks like we got the right answer there, and in fact it's a general rule.

So that's great. That's another property of trellis-oriented generator matrix. We can again, by inspection, get the dimensions of all the branch space sizes. Are the branch sizes important? Yes. In fact, they're more important from a complexity point of view than the state space sizes. 95% of the literature has to do with state spaces. State space is a little bit more elegant mathematically. For instance, you don't have the same kind of dual branch space theorem as you have a dual state space theorem.

But actually, when you're asking what's going on in the Viterbi algorithm, what does the Viterbi algorithm have to do? At a minimum, it has to do one thing for each branch. It has to perform a metric extension, a metric addition computation at least once for each branch. So the Viterbi algorithm complexity is better estimated by the size of the branch space than the size of the state space.

To progress at this point, a Viterbi algorithm has to do at least eight things. To progress again, it has to do at least eight things. So it's more relevant from the point of view of complexity. Furthermore, you can play games with state space size. And you've seen me play games with state space size.

For instance, if I draw this as a two-section trellis -- the first picture I put up was only a four-state trellis, and it looked like this. I went immediately to time 2, and I aggregated these two things. 0, 0 and 1, 1 were up here, and 0, 1 and 1, 0 were down here. And then I had a nice -- I just drew these four states over here. And if I do it like that, then I get a much nicer picture. It looks like this. It looks only like a four-state trellis.

Now, is this still a legitimate trellis? Yeah, it's still a legitimate trellis. If I do a Viterbi algorithm decoding of this trellis, is it any different from this? Well, in detail it is a little bit different. You do the additions and the comparisons and the selection. So you only do selections every this often. And if you think about it, it's really legitimate to think of this -- if you think exactly how the Viterbi algorithm works, really what's it doing? It's not making any selection here. It's just adding an increment there, and then adding another increment, and it's not making any selection until it gets here.

So it's perfectly legitimate to think of this as basically being a four-state algorithm. It really isn't doing any work at this stage. All the work occurs at this stage if you think about the operation of the Viterbi algorithm. So this is a quite legitimate representation.

And so what are we going to say? Is the state complexity of this code, is it four states or is it eight states? Well, to some degree it's just a matter of definition. How do you choose to define it? If you want something more fundamental, it's better to

go to branch complexity.

What's the branch complexity here for the fully displayed trellis? It's eight. By branch complexity, I mean the maximum size of any branch space. What's the branch complexity here? It's eight. We've still got eight. Ultimately we've got to compare these eight things and reduce them to four, whether we do it here or here.

I gave you another trellis which was only a two-section trellis, where I just showed the four central states. And we actually have two branches going there, two branches going there, two branches going there, and two branches going there. What's the branch complexity of that trellis? Still eight. There are eight ways to get from the origin to here.

If you go back to how we do this theorem, if you believe this theorem, what happens now if we say arbitrarily, OK, we're going to define the state spaces as we do in a rate 1/2 convolutional code. We're just going to define the state times to be every two times. Then the dimension of the state spaces, just at those times we've defined, is 0, 2, 2, 2, 0. They don't change. We just have fewer of them by our choice.

What happens to the dimensions of the branch spaces? Again, if the theorem is correct, we should just count the number of active generators at each now of these four bi-symbol times. So it's 2, 3, 3, 2. Is that right? That is the branch complexity of this trellis down here. It's 4 in the initial interval, and 8 over here. So again it's correct.

Suppose we go to that so-called two-section trellis. We just say there's a first half and a second half. Now the states is going to be 0 2 0, and the branch complexity is going to be -- there are three active generators in the first half, and three active generators in the second half.

So by playing this kind of game, which is called sectionalization, we can make the state complexity sometimes appear simpler. There are limits to how much simpler we can make it. Well, there actually aren't. Suppose we took the whole eight

symbols as one time. We'd have 0 dimensional state space at the beginning, and 0 dimensional at the end. What does that trellis look like? It looks like 16 parallel transitions from beginning to end. That's a legitimate trellis, right? We should have one state at the beginning and one state at the end, and 16 transitions.

Is that what we get if we went through this? Yeah. We have 0 0. And what's the branch complexity? It's 4. The dimension of the branch space is 4 if we took the whole thing.

So we can mask state complexity by picking our state spaces at appropriate times, but you can't ever mask branch complexity. And again, it's a very intuitive proof here. Let's take this time here, where there are three active generators. By expanding that time, can we ever get fewer than three active generators? No, any interval that includes this particular time is going to have three active generators in it.

So branch complexity cannot be reduced by this kind of sectionalization game. State complexity can be apparently reduced. It isn't really. But branch complexity cannot.

AUDIENCE: [INAUDIBLE]

PROFESSOR: And if you go too far, it might increase.

So at the very end of the chapter, I talk about sectionalization. It's not a very important issue. There's a very easy heuristic for how you should sectionalize, which is basically you should make the sections as big as you can without increasing branch complexity.

So in this case, the heuristic says we know we're going to have to have branch complexity of 3. But without increasing -- if I just make this one partition at the center, I just take these two sections, first half, second half. Then I have an increased branch complexity. So that's a good sectionalization. This will probably lead to the simplest Viterbi algorithm and the simplest-looking trellis representation of the code.

So that's the character of the sectionalization algorithm. You can read about it. It usually winds up with a sensible time-varying sectionalization, and it's worth five minutes of discussion. Yes.

AUDIENCE: Is it like a monotonic kind of thing, like if you make sections bigger, then you'll only either increase or keep the same branch complexity?

PROFESSOR: Yeah, that was my basic argument, that if I have any section that includes this time, and I start making it bigger, the dimension can only increase. So the algorithm is to keep increasing it until I get to some place where I might have to include another generator.

For instance here, if I'm taking this time, I don't want to push it out over here, because then I'll get the fourth generator in there. So any section that includes time 3 should not be extended in this direction, because we'll get another generator. But no problem with extending in this direction. So that's the basic heuristic.

It's easy. And it's not very productive. It doesn't really change the complexity of the Viterbi algorithm as a representation of code.

So these are just a couple of arguments to say really we might -- the branch complexity is more fundamental. We want to focus on branch complexity a little bit more. But of course, if you have a fully displayed trellis, the maximum state space dimension is either going to be equal to the maximum branch space dimension or one less than it. Because in these binary trellises, we get at most a binary branch at each time.

So any branch complexity is at most twice that of the adjacent state space. So they're not going to differ by very much. So state complexity is a good proxy for branch complexity in a fully displayed or unsectionalized trellis.

While I'm on this subject, there's another interesting direction we can go, which is average dimension bounds. Again, think of our realization here. What does it contribute? What does a single generator contribute overall to state and branch space complexity?

A single generator Gj of span S, which of course has to be greater than or equal to the minimum distance to the code, right? The span can't be less of a generator. It can't be less than the minimum nonzero weight of any code word. Span S contributes overall to all the state space dimensions. It contributes S minus 1 state space dimensions, because it's active for S minus 1 state times. And it contributes S branch dimensions because it's active for S symbols.

If I look at it spread out across time, this generator of span 4, it contributed three times to a state space dimension, and it contributed four times to a branch dimension. So total state dimension has to be greater than or equal to k times d minus 1. And the total branch dimension, just summing up across the trellis, is greater than or equal to k generators times d.

Now how many different nontrivial state spaces are there? There are n minus 1 nontrivial state times. Excuse me. For instance, in this code of length 8, there are 7 nontrivial state spaces. So the average state dimension has to be greater than or equal to k d minus 1 over n minus 1. And the average branch dimension has got to be greater than or equal to k d over the n symbol times.

So actually we might remember that quantity. That's what we called the coding gain of the code, the nominal coding gain on the additive white Gaussian noise channel. Well, if the average dimension is lower bounded by something, then certainly the maximum state space dimension is lower bounded by that. Max has to be at least as great as the average.

So if these are what are called the average dimension bounds on the maximum state space size, and there are some interesting implications. Again, this expression for branch dimension is nicer here, because it's this k d over n nominal coding gain thing. It's in terms of the basic n k d parameters of the code.

And it says what? It says if you want a nominal coding gain of 2, it implies that the max branch dimension is greater than or equal to 2. Or the max size is greater than or equal to 2 to the 2, or you need a branch space of at least size 4, or dimension 2

to get a 3 dB nominal coding gain.

Similarly, if you want a 6 dB gain -- let's get right down to dB. It says you're going to need a branch space size greater than or equal to 4, or at least a 16-state trellis. So we know if we want a 6 dB coding gain, we're going to need at least a 16-state trellis, or a 16-branch trellis. It might be an eight-state trellis.

Well, these are not tight, but they're quite indicative in fact of what we see in both the tables for block codes and for convolutional codes. These same bounds hold for convolutional codes, by the way, just average over infinite time. But they're still perfectly good.

So they tell us that in order to get -- if we want the nominal coding gain to keep going up to 8, to 16, to infinity, it says we're going to need infinite trellis complexity to get infinite coding gain.

Let's do some little asymptotics, be a little bit more refined about this. A sequence of codes Cn of length n going to infinity is called good if both the rate k over n is bounded away from 0 as n goes to infinity, and the distance d over n is bounded away from 0. The rate and the relative distance are both bounded away from 0 as n goes to infinity.

Is that clear? In other words, we want the dimension and the minimum distance both to increase linearly with n in a good sequence of codes. This is a classical algebraic definition, and it's easy to prove there exists good sequences of codes. You just pick a code at random, rate 1/2 of binary linear code. It on average will have a relative minimum distance about 11%. This is just basic information theory.

So there certainly exists sequences of good codes, and this is what most of coding theory was occupied with constructing for a very long time. What happens if you have a good sequence of codes? We have this nominal coding gain, which is equal to kd over n, which is equal to n times k over n times d over n. If both of these are bounded away from zero, then the nominal coding gain has to increase linearly with n.

That means that the average, or the maximum branch dimension, in a minimal trellis representation for your code has to increase at least linearly with n, which means that the actual branch complexity, the size of B, has to increase as -- it goes as 2 to the nominal coding gain, which is in other words exponentially with n.

So basically what this says is that if your objective is to get a good sequence of codes, if this is the way you think you're going to get to channel capacity, then sure enough, your nominal coding gain will go up to infinity linearly with n. But how are you going to decode it? Our first method was simply maximum likelihood decoding. The complexity of that, computing the distance to every code word, that certainly goes up exponentially with n, exhaustive decoding, comparing with every code word.

But now we have a new method which is, we hope, simpler, which is trellis decoding. But we find out, unfortunately, the complexity of trellis decoding is going to go up exponentially with n as well.

So the good news was that with the Viterbi algorithm and a minimal trellis, we can get a simpler optimum maximum likelihood decoding algorithm for block code, simpler than exhaustive decoding. The bad news is that under this assumption, the complexity of that algorithm is still going to go up exponentially with n. Maybe just a lower exponent, that's all. So we haven't really solved our basic problem, which is exponential decoding complexity, by going to trellis decoding.

We have, however, another way to go. Do we really need the coding gain to go to infinity, the nominal coding gain to go to infinity?

If we go back and remember the name of the game, the playing field that we're on, at least in the additive white Gaussian noise channel, we found the distance between uncoded and the Shannon limit was some finite number, something like 9 dB at a 10 to the minus 5 error probability, larger at lower error probability. So that's the maximum effective coding gain we can ever get.

So maybe we only need a finite nominal coding gain. In practice, this means maybe

we don't need a minimum distance that goes up linearly with n. Maybe we can have a very long code with a low minimum distance. And that would be another way to crack the complexity problem. In fact, some of the capacity-approaching codes that we're going to talk about, specifically turbo codes, tend to have bad minimum distances. You have a turbo code that's thousands of bits long, and its minimum distance will be something like 20 or 30. And yet it signals within 1 dB of the Shannon limit. It gives you low error probabilities within 1 dB of the Shannon limit.

So maybe there's another way to go, at least if we're only interested in a certain error probability. Maybe it's 10 to the minus 5. Maybe it's 10 to the minus 10. But it's still only a finite error probability. Maybe we don't need the minimum distance to go to infinity.

Maybe we can get by sampling codes which are lousy in classical terms, which have poor minimum distance. Maybe they'd be lousy in distance terms, or in nominal coding gain terms, but they'll be very good from the point of view of complexity. Maybe there's some trade-off here. We know the code is going to have to be long, but maybe it doesn't have to have large distance. Yeah.

AUDIENCE: [INAUDIBLE] That's what you are saying, isn't it? We try to maintain the --

PROFESSOR: We certainly want to maintain a finite rate. This was the problem with orthogonal and simplex codes and so forth, is that the rate went to zero, and therefore the spectral efficiency went to zero. In practically all cases, that's unacceptable. So we need to maintain some minimal rate or spectral efficiency on the additive white Gaussian noise. So that we need, but this maybe we don't need.

So that's just a interesting little bit of philosophy we can do at this point, is how should you really design capacity-achieving codes? And this seems to be a clue to the way to go that has in fact proved to be good.

However, I should add that not all capacity-approaching codes have this property. A random low-density parity check code is going to have both low complexity and a very good minimum distance as well. So this certainly isn't the full story.

There are just two more topics in chapter 10, and maybe I'll discuss them both briefly right now. One is I skipped over projections. And how shall I introduce this subject?

I talked about the subcode of, say, this code, which is pretty badly marked-up by now, but let me keep using it. I talked about the subcode that's supported, say, on the first half. In this case it's simply the subcode that's generated by the first generator.

Here's another code that we can look at that's kind of a first half code. Suppose we just take the set of all possible first four-tuples in this code. That's called the projection onto the first half. In other words, what are the set of all possible four-tuples? They're the set of code words that are generated by 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, and this doesn't contribute anything.

So it's a linear code. It's length 4. It has three generators, dimension 3. And we can quickly convince ourselves that it's the even weight, or zero sum, or a single parity check code of length 4. So if we project this on to the first half, we get a 4, 3, 2 code.

So that's what I mean by a projection. If I project on to here, I get the 1 0 code. I'm sorry, I get the universe code of length 1, all one-tuples. On here I get all two-tuples. On here I get all three-tuples. But on here I don't get all four-tuples. I get a single parity check code. So those are my projections.

Some things can be done nicely in terms of projections. Projections are kind of the dual to subcodes. In fact, you prove this on the homework in the course of proving the dual state space theorem. In other words, the dual of a subcode of a code C is the projection of the dual code of C, and vice versa. They're always tongue twisters, the duality theorem. But that's what you will prove along the way.

Let me give you the relevance of this to the state space theorem. So I had the state space theorem right here. So let me leave it up.

State space theorem is based on -- in general, we divide the generator matrix g

prime into a part that is a generator for the past subcode 0. 0 generator for the future subcode. And then some stuff down here, which is generators of the state space code.

By construction here, the projections of any nonzero vector in the state space code are nonzero. If they were zero, any nonzero linear combination of these generators down here cannot be zero on here.

So the projection on to the first half here, the dimension of the projection -- I'm making a slight skip here -- is basically equal to the dimension of the past subspace code plus the dimension of the state code. It's basically generated by these generators and these generators projected onto here. We can forget about these generators.

So let me write that down. The dimension of C projected onto the past is equal to the dimension of the past subcode plus the dimension of the state space code. Which leads to another version of the state space theorem. I could simply write state space theorem as dimension of the state space at time k is simply the difference between the dimension of the projection, C projected on the past, minus the dimension of the subcode. It's all of these guys minus these guys.

And again, if we look back here, we see we have the 4, 3 code is C projected on p, 4, 3, 2, and C -- the subcode is 4, 1, 4. In fact, for this, these are both Reed-Muller codes of half the length. This is not an accident, as you will prove in the homework.

The state space size is the difference in the dimensions of these two codes. This is a very handy thing to know, and you're going to need this. This, for instance, will give you that in general for Reed-Muller codes, if you divide them in half -- we had this general picture of say the 8 4 code being made up by the new u plus v construction. These two together make the 8, 4, 4 code.

If you have the u u plus v construction, there you are, first half, second half. You will find that the projection on the first half is always this guy, which is the u code. And the subcode is v code, which is the homework. And so you can quickly read off from

this what the dimension of the central state space is for Reed-Muller codes. In this case it's 2.

But for instance if we have the 32, 16, 8 code, then what is that made up of? That's made up from the 16, 11, 4 code and the 16, 5, 8 code. And so we can see the dimension of the central state space here is going to be 6. We're going to get a 64-state trellis for this code, at least just measuring the size of the central state space.

Now it turns out you can keep having -- if you go down and you make four cuts here, then the relevant codes are what you get one more space back. At this space, we have the 2, 2, 1 as the projected code, and the 2, 0, infinity as the subcode. And again, the difference in dimensions here is going to be the same as the different dimensions here. Minor miracle at first. It turns out to be just from the construction.

Here we go back to the 8, 7, 2 and the 8, 1, 8. And again we see the difference here is 6. So that means that Reed-Muller codes always have trellises, least if you put them in four sections. They always look like this abstractly. They look like what we've already seen for the 8, 4, 4 code. And you'll do this on the homework using projections. That's not very hard.

It doesn't mean that -- so we know, for instance, for the 32 16 8, we're going to do this, and there are going to be 64 states here, 64 states here, 64 states here. We don't know what it's going to be in between, but you can figure that out too. In fact, we get a minimal trellis for the whole thing.

That's what you need to do the homework. There are actually two more topics that I want to do in Chapter 10. One is the Muder bound. And one -- I actually want to see how good are these block code trellises vis-a-vis convolutional code trellises. So I'll do both of those at the beginning of next time.

For homework problem set, let's this week do only the first four problems. So for this Wednesday, do problem set -- what are we up to? Seven, is it? One through four, and for Wednesday, 4/20, we'll do problem set seven, five and six, plus I'll probably have another one.

But we have a holiday next Monday also if you recall, Patriots' Day. So we only get one more class on Wednesday. So I probably won't have much more to add than this.

Is that clear? Maybe Ashish you could put out a email to the class to do that. So good. We'll come back and clean up Chapter 10 quickly on Wednesday, and then move on to 11.