PROFESSOR: --wind up doing a substantial part of Chapter 11, although I do tend to be over-optimistic. So we're handing out Chapter 11, which I expect to revise still further. As we get towards these end chapters, these are things I've only taught from, at most, once before, and so I have more revisions to do than the earlier chapters which I've used for many years. The Problem Set Seven, as we announced last time, you only needed to hand in the first four problems today, so we're only handing out the solutions for the first four today. I haven't made up any additional problems, so Problem Set Eight is simply to do the last two problems in Problem Set Seven, due next Wednesday. And a reminder that we have no class on Monday because of this peculiar local holiday we have called Patriot's Day, when we run the marathon. And I hope you'll all be out there running the marathon. So enjoy the long weekend.

So what have we been doing in Chapter 10? We've basically been getting minimal trellis realizations, or state-state space realizations, in system theory terms, for linear block codes. We've been focusing on binary linear block codes, but clearly everything we've been doing goes to block codes over any finite field or, in fact, over the real or complex field. It's just basic realization theory, minimal realization theory. So as long as we have a linear time-varying system, it works. I had a question after class last time, well, doesn't it work just as well for nonlinear systems? Can't we do the trellis in the same way? In general, not. If it's a nonlinear system with a group property, then you can't. It's really the group property that's key to these constructions.

And we seem to have pretty well cracked the problem. We have this tool called the trellis-oriented or minimal-span generator matrix, from which we can read all the parameters of the trellis, from which we can construct a trellis. Really, we found that the trellis-oriented generator matrix contains all the information we need to construct a minimal trellis. And it's just a matter of turning the crank once we have that. The only degree of freedom that I've allowed so far in trellis constructions is sectionalization. I've shown you that if we can choose where to put the state spaces wherever we like-- typically we put them in the center and other places, sometimes

we put them everywhere we could put them, that's called an unsectionalized trellis-- and by so doing, we can sometimes reduce the apparent state-space complexity. But we can never reduce the branch complexity.

So we're left with branch complexity as a fundamental complexity parameter for linear block codes, or so it seems. Is there any other-- yes?

AUDIENCE: Does the complexity of the number of computations you need to go through the trellis depends only on the number of branches, right?

PROFESSOR: It depends primarily on the number branches. It also depends on the edges. If you go through a detailed operation count, I think I give the formula for if you count additions and comparisons as an addition and selection you don't count as a logical operation, then I think the total count for the Viterbi algorithm is twice the number of edges in the graph, which is the number of branches, minus the number of vertices, which you lose one for every--

AUDIENCE: What do you mean by vertices?

PROFESSOR: By vertices I mean states, now I'm talking about it as a graph. Minus one or plus one or something. So that's an exact count for the Viterbi algorithm. But it's dominated by the total number of edges in the graph which, in turn, is well-estimated by the maximum branch complexity at any one time. So there's a bit of a trade-off between an exact count and just getting a gross handle on the situation. But I say it's certainly got to exceed the branch complexity. So when I find the branch complexity goes up exponentially, even for one branch, then I know that the total Viterbi algorithm complexity has to go up exponentially with that. All right?

So is that the whole story? Do we now have a pretty definitive handle on the, quote, complexity of at least maximum likelihood decoding, trellis decoding of a block code. And if you've read Chapter 11, you know there's one other degree of freedom that we haven't exploited yet in representing a linear block code. Anyone read that far, or are you just going to wait for me to explain it to you? Flip the coordinates, very good. Could that make a difference? Yes, it certainly could. Let's take our standard

example by now, the 8-4-4 code with trellis-oriented matrix that looks like this. OK, and we know everything about this by now. But suppose, say, we flip these two center coordinates. Certainly it would be an equivalent code from all of its n, k, d and that sort of properties. So I want to take these two coordinates and make them 1, 1, 1, 0, 0, 1, 1, 1.

Let's suppose that's a trellis-oriented generator matrix for the code. I think it is. It's got starting times here, here, here, and now here. And it's got ending times now here, here, here, and here. OK. Where I introduce the diverging and merging symbols that I used last time for starting and ending. All right, I think that is a trellis-oriented generator matrix. Well, it is. We found a matrix that has distinct starting times and ending times, so it is.

Now what's its state branch complexity? The state dimension is 0. Now what are our spans now? Two are the same, but two are larger. OK, so now when we count, we get that the state complexity goes 0, 1, 2, 3, there are actually four that are now active in the center. 3, 2, 1, 0. OK, and that's worse than we had before, at least in one place. If we look at the dimensions of the branch spaces, which I'm arguing is more fundamental, then we get 1, 2, 3, again, 4, 4. So the branch complexity is now 16 rather than 8. It's going to be more difficult, definitely, to decode this code with the Viterbi algorithm. This is a more complex representation of an effectively equivalent code.

Well, so is there-- so this is a bad permutation, it's something I didn't want to do. Is it possible there's some good permutation that would reduce the complexity? In general, when we're given a block code, traditionally at least, we think of it as the coordinates as just being not in any particular order. We can write down a generator matrix, but the view is that any permutation of the symbol times is perfectly legitimate. Certainly, if we send it over a memoryless channel, it's not going to affect the performance of the code, what order we send the symbols in. So we should certainly regard all such codes as equivalent.

But we can affect the complexity, for better or for worse, so we should try to --

presumably, if we want a more fundamental complexity metric for the code, we want to find the complexity of the least complex trellis realization under all permutations of the coordinates. All right, so that's a good issue, but very little is known about this issue. It is known that finding the best permutation of an arbitrary linear code is an NP-hard problem. So this is going to be hard.

We know for some specific classes of codes what the best permutation is. For instance, the Reed-Muller codes are nice because it's been proved by Kasami and others that the standard coordinate ordering, the one that I've been using all along based on this length doubling construction, the u-u plus v construction, gives you the minimal trellis complexity. All right, so we know what the best ordering is for the Reed-Muller codes. And in a minute, I'm going to prove what the best possible we can get bounds-- in particular, the Muder bounds, which I'm going to discuss next-- which bound the best possible complexity that you could ever get. And in some cases, we can find coordinate orderings that meet that bound, notably in the case of the 24-12-8 Golay code, we can find a coordinate ordering that's very closely related to the Reed-Muller ordering such that the bound is met. And so we know that we've found the best complexity for that code. It couldn't possibly be any better.

But apart from certain very nice cases like those, we don't know. For instance, take an arbitrary BCH code. What's the best coordinate ordering for that? We don't know. So this is the open question in this subject, and it's now been open long enough and enough smart people have looked at it, and we've got this NP-hardness result, so that it looks like it's going to stay open indefinitely.

All right, let me discuss, however, this Muder bound, because it's simple and sometimes definitive. Yes?

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Minimal branch complexity or any other measure that you like. They all are quite fungible, it turns out. OK, The Muder bound. Let's look at a particular code and suppose we just know its parameters, n, k, d. Let's ask what the minimal trellis complexity could be for a particular coordinate ordering, say, that divides -- we have

a certain number of past symbols, call it n_p, and a certain number of future symbols, n_f. And I'm going to ask what the minimum state complexity, or I could also ask what's the minimum branch complexity for-- let me do now a state complexity, going back to where we started from, for a a certain division in the past and future. To be concrete, let me take the 24-12-8 Golay code. It turns out that it's been proved that all codes with these parameters are equivalent. So we can just call this code the binary Golay code.

And let's look for any division into 16 future coordinates and 8 past coordinates. So we don't know what the ordering is going to be, but we could say a few things. Remember how we get the state complexity. We look for what's the dimension of the past subcode. So here's a generator for the past subcode here, the code whose support is the past. We can ask what the dimension of the future subcode is, and then the dimension of the state space is the dimension of what's left. So it's dimension of c minus the dimension of the past subcode minus the dimension of the future subcode. State-space theorem.

OK, but we know a few things about this code. We know what its length is and we know what its minimum distance is. So for this particular case, we have that c past is an 8-something-8 code, because its minimum distance has got to be at least as great as this. It's a subcode of this code. So any non-zero code words in this past subcode have to have minimum weight of at least 8. Correct? All right. And similarly, the future subcode, in this case, is a 16-k -- let's call this past -- k-future-8 code. At least 8. So that imposes some bounds on the dimension of this code. How large could the dimension possibly be? All right, anybody? For the past subcode its clearly, we have that k_p less than or equal to 1. We could have, at most, one generator here. And if there were one, it would have to be the all-1 code word. Or the all-1 symbols here, because it has to have weight 8. So that's all we can get.

For the future code-- well, I guess you don't know this, but you can guess that the Reed-Muller code with length 16 and minimum distance 8 is the best there is, that distance 16, and that happens to be true. So k-future can't possibly be better than 5, which is the dimension of the Reed-Muller code with those parameters. For more

general cases, you have to look up tables on bounds on the best codes that have ever been found. But certainly for code lengths less than a 100, by this time we pretty much know all the best binary codes. So we can look up in a table what's the best code of length 16 and distance 8? Well, it has five information symbols, dimension five.

So we couldn't have more than one dimension here and we couldn't have more than five dimensions here, so the dimensional code itself is 12. We have to say that the dimension state-space is now going to be greater than equal to-- this is k_p minus 1 minus 5-- so, that's 6. So there have to be at least 6 generators down here that have support neither wholly on the past nor wholly on the future and therefore have to be included in the state-space. All right, that's the idea of the Muder bound.

Now, we could do this for every single state-space. We simply need to know what the dimension is of the best code of each length with minimum distance 8. And we can do a similar thing for every possible partition in the past and future. And that will give us a set of bounds, and we go through that exercise in the notes. And I can write it on the board, maybe for memory. And OK, that tells us what the best possible state profile is. Let me guess state dimension profile is 0, 1, 2, 3, 4, 5, 6, 7. Then it goes down to 6, 7, 8, 9, then it goes down to 8. This is at the central state-space. This is for dividing it in the way that we just did.

Let's do this here. For the central state-space, what's the best k_p for a 12, k_p, 8 code? Let's make the past and future both 12. Anyone want to guess what the best dimension is? Let's think about it. Let's think of a code that's going to have to have minimum weight 8. So let's start out with g1. And we'll just make that look like that. It's going to have length 12. It's going to have to have at least eight 1s, so let's make them like that. We want to add another 1 so that the total code has a minimum distance 8. Subject to up to permutations, there's only one thing it can be. We need another weight-8 generator, and it needs to differ from this in at least four places so that when we take the mod 2 sum of these two, it has distance 8. And we want g1 plus g2. This is really the only way it can be. Are you with me? We have four code words in this dimension 2 code. And if they're going to have minimum

distance 8, they're going to have to look like this, subject to up to permutation.

So from that little argument, we conclude that, in this case, this is less than or equal to 2. This is less than or equal to 2, the Muder bound becomes that the dimension of the state-space has to be at least 8. And so in the same way, and then it becomes symmetrical on the other side. So I'm pretty sure I get it right. And from memory, that's the best possible state dimension profile, dot dot dot. And we actually know a coordinate ordering for the Golay code, a generator matrix for the Golay code, that gives this state dimension profile. So from that, we can conclude that we know an optimum coordinate ordering for the Golay code. I just want you to get the essence of the argument here. You with me?

For branch spaces, the argument is much the same. Remember in branch spaces, we basically -- let me say it in a slightly different way than we did before -- we take the past to be up to time k. And we have the symbol time, which is just one time in here for the actual symbol we're looking at. And the future is one unit shorter than it was for the state-space. So when we compute the branch dimension, the past goes up to k. So this is 0 and before k. And the future goes from k plus one up to n. So this is what the matrix looks like for this.

And the effect over here is that the future subcode now is one unit shorter. So let's do the same calculation here. If we look at a time where we have 12 in the past, this one time, the 13th symbol time that we're actually looking at and then 11 more in the future, the past subcode, again, could have dimension up to 2 because it has length 12 and minimum distance 8. The future subcode now has a shorter length and by this little argument I went through here. If we take out one of these coordinates, it's not going to work. So if we only have length 11, the future subcode is the dimension -- still can only be 1. We can't possibly add another second generator in there. And so the dimension of the branch space has to be vastly greater than or equal to 9, at this point. OK, so the Golay code has branch complexity of at least 512, we conclude.

And, in fact, this best possible branch dimension profile looks like this. The branches

are in the middle here. They're 1, 2, 3, 4, 5, 6, 6. Basically we have generators which look like this. And then there's one that comes down, there are three more that goes up and one that comes down. And from that, we get 1, 2, 3, 4, 6, 7, 7, 8 -- that can't be right. Goes down 7, goes up, it's still 7, then it goes up to 8, 9, and 9, 9, 9. Something like that. That one's probably wrong, and symmetrically. Where this 9 is the one we just calculated.

Similarly, if you come down to the neighbor of this 6, it's got a merge on one side, a diverge on the other side, and that's 7. OK, so we conclude that the Golay code has branch dimension not less than 512 within 9. Yeah?

AUDIENCE: Suppose we take past and future separated by 12, 12 in the past, 12 in the future. Suppose we did it as they did in [UNINTELLIGIBLE] state?

PROFESSOR: OK, but that's not the way we do branches. We've got to isolate the symbol time.

AUDIENCE: Suppose we took 12-branch-12 and we calculate the state complexity, that's 8. And then we took 13 and 11, we get the state complexity 9.

PROFESSOR: All right, you're saying if we do this, we should get the state complexity 9. And that's correct, we do.

AUDIENCE: The maximum of those two numbers, is it the same as branch complexity?

PROFESSOR: It turns out to be in this case.

AUDIENCE: Not always?

PROFESSOR: Not always. But if we go to 14, 10, then it turns out we can add another one here because we can add another generator that looks like something like this. 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1. So if we have two more units, we can add another generator. Sort of Reed-Muller style. And so this goes up to 3 and this goes down to 8.

AUDIENCE: So the branch complexity has to be at least the maximum of those two, right?

PROFESSOR: Obviously, right. And really, together these give you the story of where the thing has

to diverge and merge. In this case, this is a self-dual code, so you only do one at each time. So with this sort of thing, you can pretty well pin it down. Again, I don't know how far to take this lecture. I'm really just trying to get to you the idea of the Muder bound from this kind of argument, therefore we can get a bound on the best possible state dimension profile and the best possible branch complexity profile. If you compute this bound and you're able to find an actual generator matrix that meets this bound, you know you've found the best one.

But this rarely happens, this is a very special code. It's not the way you prove the optimality of the longer Reed-Muller codes, they're not good enough to -- this really assumes that you can get the best possible sub-codes at every point. And you can imagine that's very hard to do. It would be a very special case where you can optimize this for every single possible cut between past and future. OK, so that's the Muder bound. And you have a homework exercise on that as well.

Oh, there's another case where we can find what the best possible trellis complexity is. It's MDS codes over GFq, and you're going to do that on the homework as well. And there, we basically show that an MDS code, its n, k, d are such that it has to have the worst possible state complexity and branch complexity. So because it's such a tightly-packed code, in Hamming distance sense, it forces you to have the worst possible complexity that you could have. In other words, you have diverges immediately and as long as you can, and then they're all packed together on the left side and the merges are all packed together on the right side, and that's the only way it can be. I'm just speaking very impressionistically. But you'll see what I mean when you get to the homework, if you haven't done it already.

OK, so we also know MDS codes, we know their trellis complexity and it's terrible. We can't get any effective reduction. Really, over exhaustive decoding methods with MDS codes, we find that the trellis complexity is like either q to the k or q to the n minus k, depending on which is less. So that's as bad as it could possibly be. So if the trellis doesn't help for MDS codes, in general cyclic codes -- which were subject to a lot of study -- if we put the code into cyclic form, which if you remember Reed-Solomon codes and BCH codes can always be put into cyclic form, at least the

shortened versions of them. In that case, the generator matrix looks like this one up here. It looks like x, x, x, x, x, x, x, 0, 0, 0, 0, 0, 0, 0, x, x, x, x, x, x, x, 0, 0, x, x, x, x, x, x, x. In other words, the generators are just cyclic shifts of one another. Down to x, x, x, x, x, x, x, 0, 0, 0, 0. That is a generator matrix that is a trellis-oriented generator matrix because all the starting times are different and all the stopping times are different.

So this is the trellis-oriented generator matrix for cyclic code, and it's in the coordinate ordering in which it is cyclic. And it's always going to have this worst case -- all the branch diverges are going to occur over here and you may get nothing for a while. And then all the merges over here, which if it's a rate less than 1/2 code, this means that the trellis complexity is basically going to be 2 to the k. Or if it's a high-rate code, it means it's going to be 2 to the n minus k. And again, you get no improvement over some exhaustive minimum distance decoding, or maximum likelihood decoding. So cyclic is very bad from a trellis point of view.

So I think that's about all the general comments I can make. The very last thing that I want to mention is we now want to -- all right, so through this nice little theory, we've got a way of trellis decoding (maximum likelihood decoding), via the Viterbi algorithm, of block codes. And now, did we achieve any improvements in terms of performance versus complexity versus the convolutional codes that we looked at previously? And the basic answer is, no. If you compare the tables, convolutional are always better. I'll just say it. You can imagine that there have been long debates in the fraternity, and that certainly some of the motivation for trellis decoding of block codes was, oh, gee, now we have a good way to decode block codes, just like we did for convolutional codes. And maybe now we'll find that block codes are worthwhile. But the answer is they're not.

I should put this comparison into Chapter 10. You can actually get this comparison by taking the table -- for instance, for Reed-Muller codes. In terms of performance versus complexity, Reed-Muller codes are very good. I would say as a general statement, they're the best block codes that we know of, in terms of performance versus trellis decoding complexity, because they have this nice structure that gives

them very low-complexity trellises for their n, k, d parameters.

So in this sense, it's a fair comparison. Reed-Muller codes are the best general class we know of in terms of trellis complexity. We compare them against binary convolutional codes and they're not nearly as good. I've already talked about the comparison. It seems like a very fair comparison to take our rate 1/2 four-state convolutional code example, which had distance 5, which had not only a nominal coding gain of 4 dB, but it also has an effective coding gain of 4 dB. And for a block code, this 8, 4, 4 code is a very fair comparison. It's also rate 1/2, so it has the same spectral efficiency. It also has a four-state trellis that looks very much like a rate 1/2 convolutional code trellis.

But it turns out it only has minimum distance of 4 rather than 5, so its nominal coding gain is only 3 dB rather than 4 dB. Its effective coding gain is less. Its effective coding gain is 2.6 dN because it has 14 nearest neighbors, which is a 3 1/2 nearest neighbors per bit, and that costs you about 4/10 of a dN. And so really, you compare the performance of the convolutional to the block code, that's very much better. Another head-to-head comparison might be between the 32, 16, 8 Reed-Muller code. That's a rate 1/2 code. It's a very good one, it's known to be the best with those parameters, n, k, d. So what's its nominal coding gain? Anyone? It has rate 1/2, minimum distance 8, its nominal coding gain is? Hello? 6 dB. You take the rate times the minimum distance-- 1/2 times 8, you get 4. So that's the nominal coding gain. A factor of 4, which in dB terms is 6 dB.

But again, it has lots of minimum-weight code words. It has 620 minimum-weight code words. And therefore, the number of minimum-weight code words per bit is 39, which is between five and six factors of 3, so it costs you 5 and 1/2 times 0.2 dB in terms of effective coding gain, which brings you down to an effective coding gain of 4.9 dB. I assume you're all very fluent in doing these, as I am.

And what's its complexity? If you look its state complexity, let's take-- we're looking at the 32, 16, 8. And all the Reed-Muller codes have four section trellises, you proved that this week. If you just measure the state complexity of four places,

equally spaced with four equal sections, then all the trellises wind up looking like this, for a general Reed-Muller trellis -- a four-section trellis.

And for the 32, 16, 8, it turns out you get eight parallel sections like that with eight of these each. In any case, it's 64 states at each of these boundaries. So you could say it's a 64-state trellis. But if you're more honest, just exactly as with the Golay code calculation we went through, you find that the maximum branch complexity is 512. So branch complexity is 512.

OK, and its nominal coding gain we said is 6 dB and its effective coding gain is 4.9 dB. OK, so let's look for a convolutional code of rate 1/2 that has a coding gain of about 5 or 6 dB. And how far do we have to go? We actually only have to go up to a-- well, a 32-state code has an effective coding gain of 5.6 dB. A 64-state code we can get up to an effective coding gain of about 6 dB. In fact, the 64-state rate 1/2 convolutional code was a standard for a very long time, first in space communications and then in much more general applications. This was the code that Linkabit Corporation -- Jacobs and Viterbi -- rode to fame and fortune, made their first pile before they went on to found Qualcomm, where they made a much bigger pile. And so this was their-- they call it k equals 7. But in my terms, nu equals 6, a 64-state rate 1/2 convolutional code.

So let's take that rate 1/2 64-state convolutional code, just with the standard trellis. So its branch complexity is 128 at every time. It has a more regular structure than this guy. Its nominal coding gain is -- there are actually two of them, and I don't remember. There's one of them that has a nominal coding gain of 7, the other one is 6.5. So it's at least 6.5 dB. And its effective coding gain is still 6.1 dB and it's clearly much better than this.

If you kind of try to plot a graph of this, you'll see that the effective coding gains of convolutional codes with the same state complexity is at least one dB better, as a general trend, than that of block codes. And of course, if you do it in terms of branch complexity, it's an even better comparison. So what happened was there was about a decade of effort on basically finding good trellis representations of block codes,

and the net of it was that they couldn't get anywhere close to finding anything as good as convolutional codes. Although, it's a nice little piece of theory. And sometimes you want to block codes for other reasons. Sometimes you want short blocks. Sometimes you're limited to a block of less than 100. And in that case, you're certainly not going to do better in terms of n, k, d than the best block code of a length 100.

Why are convolutional codes better? First of all, they obviously naturally are well-adapted to the trellis idea. They are linear time invariant machines that just have very nice, regular trellises. So they're naturally adapted. But also, they have exactly the right-- a rate 1/2 code is diverge, merge, diverge, merge, diverge, merge forever. If you want to make it into a block code, you can terminate it or you can tail-bite it, which is something we're going to talk about shortly. But whichever way you do it, to make it into a block code, you lose something. The termination, you don't lose minimum distance, but you do lose rate. You have to put in some dummy information symbols, so the rate becomes less than 1/2 if you terminate this code.

Nonetheless, as I believe I commented earlier, that's a good way to come up with good block codes, is to terminate a good convolutional code. If you choose the parameters right, you can come up with pretty good codes. But nonetheless, in terms of block code parameters, you're not going to do better than what 50 years of block code, algebraic coding theory has come up with. So for short block codes with length less than 100, really the best we know of right now-- let's say 128-- is to pick a good Reed-Muller code or BCH code, find its best trellis and do maximum likelihood decoding.

It used to be that you thought of 64 states as a lot of states, but clearly, technology has progressed. And now you wouldn't blink at 1,000 states or maybe even 8,000 or 16,000 states. The biggest Viterbi decoder that was ever built had 2 to the 14 states for a space program out of JPL. So anyway, it's doable now. If you needed the best possible performance and you were constrained in block length, you'd pick the best block code at the rate that you wanted, and the best minimum distance at that rate, and the best effective coding gain, and you'd do maximum likelihood decoding with

trellis decoding.

But if that's not usually the situation -- the situation in data communications, because usually more you have a stream of data or a packet length of at least 1,000, and then it would certainly be better to use convolutional codes. And trellis decoding is -- as long as we're in this class of techniques. Where we're finally going in this course is to the new codes that have been developed in the past decade, capacity-approaching codes, turbo codes, low-density parity-check codes. These are simple enough to do so that as soon as you get past the small codes, length less than 100, you would switch over to that track. You would use a capacity-approaching code. But we don't know about them yet. I guess that's the end of my lecture on this, so it's a very good place for questions about state-of-the-art, where we've been, where we're going. The floor is always open. But that was all I was going to say about Chapter 10.

Chapter 11, Codes on Graphs. And now we begin our final stretch drive towards capacity-approaching codes, which is going to be the subject of the next three chapters. Chapter 11 is sort of the beginnings of codes on graphs. We're going to look at some elementary representations. One of the ones we're going to look at is a trellis representation. That's one of the reasons we've been going through this theory, is we're going to, again, encounter trellises as a natural, simple, cycle-free, graphical representation that, in some sense, is about as good as you can ever do, as long as you don't allow cycles in graphs. But then we're going to go on to graphs with cycles, which is really the way to understand these capacity-approaching codes.

So in this chapter, we'll start the subject, we'll introduce graphical representations of codes in more general terms. In Chapter 12, we'll talk about the generic decoding algorithms for codes on graphs called sum-product and min-sum decoding algorithms. And then in Chapter 13, we'll talk about the actual classes of capacity-approaching codes that people have developed, what their graphs are, how you decode them -- I hope in enough detail so that you'll get a good sense for how all this works. So that's where we're going.

What are we talking about when we're talking about codes on graphs? There are many styles of representations of codes, and at this point, for linear codes, we have seen a number of ways we can characterize a code. One of the ways is just by giving a set of generators for the code, k generators for an n, k code. And that sort of characterizes the code. Another way is to basically give the generator matrix for the dual code, the h matrix, which becomes the parity-check matrix for the code. So we can also characterize the code by a dual-generator matrix. That's a particularly efficient way to do it if you have a high-rate code like a single parity-check code, it's better to say that the dual code is the repetition code and it's the set of all code words that are orthogonal to the all-one code word. That's the simplest characterization of the single parity-check code. In other words, that the sum of all bits is equal to 0. That's what orthogonal to the all-one code word means. So that's a simpler representation than giving a generator matrix formed for the single parity-check code.

Or now we have this trellis representation, which certainly looks like a more graphical representation. It certainly characterizes the code in a very direct way. What does a trellis do? It basically displays all the code words on a graph. There's a 1:1 correspondence between the set of all paths, from the root to the n-node in the graph and the code. So that's another way to characterize the code.

So now we're going to go to higher level graphical representations. As a first step, we talk about behavioral realizations. Again, a term from system theory, closely identified with Jan Willems, who promoted this way of representing linear systems. In linear system theory terms, the basic idea in behavioral systems theory is you describe a system by the set of all its possible trajectories. What are all the things it can do?

And now, how do you characterize the trajectories? Often, you characterize them by a set of local constraints. You can think of it as equations that the system has to satisfy. Ordinarily, they're differential equations or partial differential equations or something. And if it satisfies all of those constraints, then it's a trajectory that satisfies all these partial differential equations, is a valid trajectory. So you can set

up the whole system that way. So it characterizes a system by its trajectories. That's the fundamental thing. And it characterizes trajectories -- legitimate, valid trajectories -- by local constraints.

Let me get a little bit more concrete. In coding terms, by trajectories we just mean code words. So this is a very compatible notion with the way we've been talking in coding. What is a code? It's simply a set of code words. So if we simply say, what are all the possible code words? We have to find all the trajectories of the code, and we have a very explicit example of that in trellises. We've called the paths trajectories through a trellis. What is a code? It's simply the set of all valid code words.

And how do we characterize the valid code words? Well, in each of these styles -- the generator, parity-check, trellis style -- the code words are the code words that satisfy certain equations or constraints. For instance, the generator representation -- we've said a code is the set of all uG, such that u is in a field to the k. That's a generator matrix style of representing a code. In other words, we can say it's the set of all y, such that y equals uG for some u in Fk. Or we can say it's all y, such that y minus uG equals 0 for some u in Fk.

So it's the set of all y that, together with some auxiliary variables which are not part of the code word -- here they're input variables, if you like, or information bits, information symbols. We'll come to think of these as state variables because they're hidden, they're not visible, they're just an auxiliary part of the description. They're code words that, together with some auxiliary variables that, in this case, can be freely chosen, satisfy a certain set of linear homogeneous equations.

Something that's even better in the behavioral style is a parity-check representation. Let me put that up. In a parity-check representation, we say the code is the set of all y, such that y H transpose equals 0, where H is the generator matrix of the dual code. This is sometimes called a kernel representation. y is in the kernel of this linear transformation. We see H transpose is a linear transformation from n-tuples to n minus k-tuples, and the ones whose image is 0 under this transformation, the

ones that map to 0 are legitimate code words. So this is a kernel representation, this is an image representation. Here, we view the code as the image of the linear transformation, which is characterized by g, where the inputs are free.

But the parity-check representation is much more explicitly, we don't need any auxiliary variables here. It's simply the set of n-tuples that satisfy a certain set of linear homogeneous equations. So this maybe is a place where we start and, of course, it's the place where Bob Gallager started when he did low-density parity-check codes, which are a principal class of capacity-approaching codes.

Then in either case, we characterize the code by, in one case, directly satisfying some equations. In the other case, here we say that the total behavior is the set of all y and u, such that y minus uG equals 0. So we're going to take the set of all n-tuples and k-tuples, such that y minus uG equals 0. And we say that's the total behavior of the code. The set of all y's and u's that satisfy this equation is the total behavior. And then we'll say the code word is just the projection of the behavior on the y's. In other words, it's the set of all y's, such that this is satisfied first some u. And that's an equivalent description.

So the general setup, going back and forth, is that we have the observed symbols. And in the coding case, this will always just be the n symbols in the code word. These are the ones that we really care about. But we're also going to have some hidden or auxiliary symbols or state variables, hidden variables, which, in this case, for instance, are the u's. And we can introduce these at our convenience, just to make a good realization.

So we have two types of symbols: the observed, or external variables-- sometimes these are called external variables, internal variables. Again, in system theory, you're consistently seeing this. There are some that we can actually observe, interact with. These are the observed symbols, and then there are internal variables to the system, very often called state variables, which we can't directly see but which are part of the description of the system, as in this case. The whole system, we find the set of all internal and external variables that satisfy a set of constraints.

But we only care about the patterns of observed variables that are consistent with some set of internal variables.

So the final element here in this general setup is constraints, which in this case are simply linear homogeneous equations on subsets of variables. So let me go through similar kinds of examples, as I do in the notes. But let me maybe build up from the simplest one first, parity-check representation of our favorite 8, 4, 4 code. In this case, the parity-check matrix is the same, we can take it to be the same as the generator matrix. Since, by now, we like trellis-oriented generator matrices, we'll take that one.

So to get very explicit, so the code is the set of all y, such that y H transpose equals 0, and y is 8-tuple. So it's the set of all code words that satisfy these parity-check constraints. The first one is y1 plus y2 plus y3 plus y4 equals 0. That's orthogonality to this first code word. The second one is y2 plus y4 plus y5 plus y7 equals 0. The third equation is y3 plus y4 plus y5 plus y6 equals 0. And the fourth is y5 plus y6 plus y7 plus y8 equals 0. If I have any 8-tuple y, such that the elements of the 8-tuples satisfy these four equations, then it's a code word, if and only if. A very explicit description of the code.

Now let's go onto a graph of behavioral realization. So it's very natural to draw the following kind of graph of this realization. We let the y's -- y1, y2, y3, y4 -- sorry, I've changed the index set, haven't I? I get 1 through 8 now. I'm just going to make these into vertices of my graph. So here we have symbols, and I'm going to make a bipartite graph, and over here I'm going to have equations, or constraints, or checks. And I have four checks, which I'm going to draw like this.

The first check is that y1 plus y2 plus y3 plus y4 equals 0, so I'll just draw that like this. That means the same thing as this. The second one is y2 plus y4 plus y5 plus y7 equals 0. And the third one is these four -- two, three, four. And the last one is these four. This, this, this, this. It's done more nicely in the notes. This is called a Tanner graph, after an extremely good paper by Michael Tanner in 1981, which was about the only paper of any consequence in this subject between Gallager's

thesis in 1961 and the rediscovery of low-density parity-check codes around 1995.

That's a graphical picture of these equations. And what do we think of? You can think of testing an 8-tuple. Take an arbitrary, binary 8-tuple, think of these as memory elements, put the bits of that 8-tuple in here, and then ask if all these checks are satisfied. Since we're talking mod 2 arithmetic, we're basically asking if there are an even number of 1s attached to each of these checks. If and only if that's true, we've got a code 8-tuple. From the set of all 256 8-tuples, we find 16 that actually satisfy these checks, and that's the code. That's one way of describing the code.

So that's how the graph is associated with local constraints. Well call them local, each of these constraints, because each of them only involves four of the symbols. In a much larger graph, a low-density parity-check code, the idea is that there are not very many symbols that are checked by each check. The graph is sparse, in that sense. But we're not quite there yet. So that's pretty simple.

Let me do a generator representation. This is going to be a little bit more complicated because we have these auxiliary input variables. But that doesn't really complicate it very much. I keep using this 8, 4, 4 code just to emphasize that a single code can be described in many different ways. We're going to wind up with half a dozen different graphical representations of this code, and they can be grouped according to style. So again, we could use our favorite generator matrix for our favorite code. It looks like that. Now what do we mean? What are our constraints now?

Well, now we have some hidden variables, ui, and we have y1 is equal to u1. We're going to basically multiply this by u1. I've got some u1, u2, u3, u4 multiplying this. So y1 is u1, y2 is u1 plus u2, y3 is equal to u1 plus u3, y4 is equal to u1 plus u2 plus u3, so forth. y5 is u2 plus u3 plus u4, y6 is u2 plus u4. I jumped one here. This is u3 plus u4, y7 is u2 plus u4, and y8 equals u8.

And you will agree that the code is described, if I can find any pair, (y,u), such that this is satisfied, then y is an element of the code. I can choose u freely, that is the

input. If I find a y at any u, such that these equations are satisfied, then I've found a code word. And that's an if and only if. So that's another characterization of the code. And its Tanner graph looks like this. Here in this case, we have hidden variables. So let's put the inputs-- or, we'll see they can also be thought of as states, but they're hidden variables that we don't actually see in the code word. u1, u2, u3, u4. You can think of these as free-driving variables.

And to get y1, here's what we do. We're going to take y1 and we're going to create a mod 2 sum of some of these inputs. In this case, it's just u1, so I can draw it as a straight through. But in general, I'm going to have to make a combination to get all of these. So I have eight symbols over here -- code symbols -- and again, some constraints that have to be satisfied. Equations, mod 2 sums, y8. And I just, again, draw a picture. y2 is u1 plus u2, so u1 plus u2. these sum to give y2. And so, again, if I put down some u's, put down some y's, test whether all these sums are correct, then I will have tested whether I found a valid behavior or trajectory. Next one is u1 plus u3. The next one is u1, u2, u3. The next one is u2, u3, u4. The next one is u3, u4. The next one is u2, u4. And the last one is just that one.

OK, so that's another graph whose constraints describe the code word. Here it's helpful to start to introduce the following convention. We fill in the circle of the variables that we actually want to see, and we leave these open, the free variables over here. Some people would call it a generalized Tanner graph. In the notes, I just call this Tanner graph 2. Tanner didn't actually consider state variables. This was introduced into graphical models by a guy named Wiberg in his PhD thesis in Sweden in about 1995. And that was a huge contribution, to have hidden variables as well as observed variables, because otherwise you couldn't really draw a generator representation as a graph, or a trellis representation. You need hidden state variables.

So those are two graphical styles, and the idea here is that every set of 12 variables, such that these eight equations are satisfied, gives you a legitimate code word. Here you can easily think of this as -- you notice that we've described these as passive constraints. It's not an input-output, block diagram type of

representation. It causes cause-effects. In this case, it implicitly is. How would you actually implement this generator representation? You implement it by picking four input bits and then seeing what code words they generate. In other words, there's a very definite input-output relation here, where we could make this into a directed graph by drawing arrows here.

In general, in our graphical models, we're not going to do this. The behavioral style is very much not in this spirit. But in this case, we clearly can. So if you actually wanted to generate all the code words -- that's why it's called a generator representation -- you simply jump through all 16 possibilities here and you generate all 16 code words. Or if you wanted to do a simulation, generate code words, this is the way you would do it. This has easy cause and effect style. The kernel representation is much more implicit. What is the cause and the effect here? In this case, it's actually possible to find a cause and effect representation.

It turns out that these four bits can be taken as information bits, or these four places form an information set. So suppose we choose these four bits, and it turns out that for this graph, we can trace through as follows. These three bits going into this zero-sum, these all go in. That determines this output, right? d minus 1, if d is the degree of the zero-sum node, any d minus 1 inputs determine the last output, because the parity-check has to check. So that determines this one, which then determines y4. Now, knowing that, we know all but one input here, because I also have this one. And that determines this, and I think it probably also determines this. I now have three of the four inputs here, so I can get this.

And anyway, by tracing through it I can create a directed graph that gives me a cause and effect relationship from here. But it's much more implicit and it can't always be done. Sometimes you can find an information set that works with these particular set of equations and sometimes you can't, sometimes you get stuck. We can come back to this when we talk about binary erasure channel and stopping sets. In this case, given these four, you can determine the remaining four. If you think of it in equation terms, if you're given y1, y2, y3, you can basically go through a Gaussian elimination of these equations and find y4, y6, y7 and y8. So you can

regard y1, y2, y3, and y5 as inputs. This is much more the behavioral style, where it's just simply any n-tuple that satisfies the constraints is a legitimate n-tuple. This is more cause and effect style over here.

Since we're at the end, let me just introduce one more thing. This is a bipartite graph. We basically have a graph with two types of nodes, one representing variables, one representing constraints. And the edges always go between a variable and a constraint. So that's what we mean by a bipartite graph. There are two types of vertices, and all the edges connect one type of vertex to another type of vertex.

So is this over here. I've drawn it in a way where it's not so obvious. In a generalized Tanner graph, you'd put all the variables over on one side and the constraints over on the other side. In this case, these observed variables go naturally with the constraints, so I put them over here. But I could have put them over here and you would see this is also a bipartite graph. So this is the general character.

What do we think of the edges as doing in this graph? We think of the edges as conducting the variables that they're associated with. So all of these edges, in fact, convey u1 to this constraint. We could, if we wanted to, label all of the edges with their associated variables. That's what they mean. They mean here that u1 equals y1, or that u1 plus u2 equals y2. So the edges just serve for communication in this graph.

There's another style of graph, which I introduced, and therefore, people -- not me -- but other people have called it a Forney graph. I call it a normal graph. The difference is that basically the vertices all equal constraints, and the edges represent variables. And it has certain advantages that have led to it being used more and more.

Let me just indicate how we go from a graph of this style to a graph of this style. We basically just make all the variables into equality constraints. So in this case, I draw y1, y2, and so forth. I put a little dongle on them to indicate that they communicate with the outside world. They are observed, external variables. And then I put a little

equality constraint that says all of the variables attached to this constraint have to be the same, which is really doing the same thing. Otherwise, the topology of the graph looks the same. These are, again, zero-sum constraints, or parity-check constraints, and the connections are done just as before. I forget. I'm not going to do them all out.

The only change, in this case, is to replace external variables by an equality constraint in this. So the equality constraint basically says this edge still equals y1, this edge still equals y2. That's what the equality constraint means. Everything tied to that equality constraint has to be equal. So we can think of these as just replicas of this original variable out here. So far, you don't see any reason to prefer this. If I do it in this graph, then, again, I get equality constraints here. I get zero-sum constraints, again, over here. Eight of them. And in this case, I can just represent the external variables directly by these little dongles. And again, I have the same graph here. You still probably don't see that there's very much difference and, in fact, I'd agree. There's hardly any difference between these two styles. So forth.

This is the normal graph that's equivalent to this Tanner graph, it's obtained in the same way. If I wanted to do it more painstakingly, I would put equality constraints for each of these, I'd put little external variables, and then I'd see I don't really need the equality constraints. So I can just compress it into y1 through y8. The advantages, I will say, of the normal graph will appear as we go along. The main advantage is that you can prove a very nice duality theorem for normal graphs -- that basically shows the duality between generator and parity-check constraints. But to say anything about that right now would just hand-waving, so we'll come back to this next week. See you in a week, have a nice weekend.