PROFESSOR: This time we're actually going to be able to do a little work that I hope will give you a good idea of why it is that at least low-density parity-check codes work and even broader classes of capacity-achieving codes. But I think we're going to limit ourselves to a week, so that we can say something next week about everything we've been talking about as binary codes for the power limited additive white Gaussian noise channel.

We've completely left aside codes for the bandwidth limited channel, where we're going to have to talk about non-binary codes of some kind. So next week I'll give you a very rushed overview of that kind of code.

And it's my intention, by the way, that last week, you'll not be held responsible for on the exams, so that we kind have a nice, pleasant week without worrying about whether we're responsible for it or not. And the last homework set, I will prepare a homework set on the bandwidth limited type codes. On Wednesday, I'll hand it out, but I won't expect you to give it back. It's good to do it. I'll hand out solutions on the final day of the exam.

OK, so any questions about where we are, what we're doing? All right, let's get into it. We've been building up to this for a while. First we did trellis representations of block codes. As a warm-up for codes on graphs, we found that some of the things we discovered about trellises were important particularly in the context of the cut-set bound.

The cut-set bound says, where you have a cut-set, the complexity at that point must be at least as great as that of a trellis diagram which divides the two parts of the code into two halves, and the past and future in the same way. And from that we concluded that we're going to need cycles. Because we'd already proved back in chapter 10 that trellis complexity needed to go up exponentially with block length, at least for codes that maintained a nonzero relative minimum distance as well as a nonzero relative rate. So we're going to need cycles in our graph we think -- to get to capacity.

Chapter 12 was a fairly short chapter introducing the sum-product algorithm. I believe that you don't really understand the sum-product algorithm till you do it once, so that's on this week's homework. You will do it once for a very simple case, and then I think you'll understand it.

We developed the sum-product algorithm for cycle-free graphs, where it's a theorem that the sum-product algorithm does APP decoding for every variable in any cycle-free graph in a very systematic and efficient way. It's a finite and exact algorithm in that case. It takes a number of steps approximately equal to diameter of the graph. And when it's finished you have the exact APPs everywhere.

OK, but we really want to have a decoding algorithm for graphs with cycles. Having developed the sum-product decoding algorithm, we see that it has a nice, simple update rule at each computation node. It's a local kind of algorithm. Why don't we just let her rip, start it off and see how it does?

And, in our case, we can make that work by designing graphs that have, in particular, very large girths such that the graph looks-- the term-of-art is locally tree-like. In the neighborhood of any node, if you go out some fixed diameter from that node, you won't run into any repetition. So locally the graph looks like a tree. And we're going to, ultimately, be looking at asymptotically long codes, so we'll be able to make that locally tree-like assumption hold for as far out as we need to. Other disciplines are not so fortunate as to be able to do that.

OK, so in this chapter we're finally going to get capacity approaching codes. I'm first just going to mention a couple of classes of codes, the most famous and important ones, show you what their graphs look like. From that, you will know everything up to schedule what there sum-product decoding algorithm should be. I'll tell you what the conventional schedules for decoding these graphs with cycles are. And that'll be an introduction.

And then we'll get into actual analysis, in particular, for low-density parity-check codes which are, really, the simplest of these codes and therefore the most

amenable to analysis. And we'll see that, under the locally tree-like assumption, we can do an exact analysis first, on the binary erasure channel. Then I'll at least discuss what's involved on more general channels, where we can do a fairly precise analysis. So that's where we're going in chapter 13.

Questions, comments? OK, so let's talk about low-density parity-check codes. These codes have a fascinating history. They were invented by Professor Gallager in his doctoral thesis in 1961. He was a student of Peter Elias.

What people were doing at MIT then was trying to figure out ways to actually build codes that get to capacity. At MIT the emphasis was on so called probabilistic codes. Namely ones that had sort of the random-like elements that Shannon basically prescribed. People here were not so interested in the algebraic codes.

And Gallager basically had the idea that if you take a code -- you can describe it by a generator matrix or parity-check matrix. If you make a sparse generator matrix, that's not going to be any good, a random, sparse generator matrix, because a sparse generator matrix will naturally have some low-weight code words in it. If the rows are sparse, then it will have low, minimum distance.

But maybe if we make the parity-check matrix sparse, very few ones in the parity-check matrix, we can still get a code that's quasi-random. And therefore we can hope that the codes, as they get along, will come close to doing what Shannon said random codes should easily be able to do. The amazing thing about Shannon's proof is he showed that you pick any code out of a hat, and it's highly likely to be good.

So, basically, Gallager was trying to replicate that. And he came up the idea of sparse or low-density parity-check matrices and was able to prove quite a few things about them. That they couldn't quite get to capacity the way he formulated them, but that they could get very close. They had a threshold a little bit below capacity.

He came up with the APP decoding algorithm for them which is probably the first

instance of the sum-product algorithm in any literature, certainly the first that I'm aware of. And he simulated them. At that time, what MIT had was-- over in Building 26 on the first floor, where I think there's a library now. It sticks out into the adjacent parking lot. That room was full of IBM equipment, 70, 90 computer. And there were people in white coats who attended all that equipment.

And by running that computer for hours, he was able to simulate low-density parity-check codes down to an error probability of about ten to the minus fourth. And he showed that he could get down to 10 to the minus fourth, pretty close to capacity. I don't remember exactly how close, but it was enough to be convincing.

Actually, in 1962 there was a little company form called Codex Corporation. It was what we would now call a start-up. Back then, there was no tradition of start-ups. And it was basically founded to exploit Gallager's patents, which he took out on low-density parity-check codes, and Jim Massey's patents on threshold decoding, which is an extremely simple decoding technique that we haven't even talked about, very low complexity.

And I joined that company in 1965 when I finish my thesis. And I was director of research and advanced product planning. And I can assure you that in the 20 years that those patents lived, we never considered once using Gallager's low-density parity-check codes, because they were obviously far too complicated for the technology of the '60s and '70s. We didn't have rooms full of computers to decode them.

And more generally, in the community, they were pretty completely forgotten about. There's a key paper by Michael Tanner in 1981 where he kind of-- that's the founding paper for the field of codes on graphs. he puts it all into a codes-on-graphs context, develops all the basic results. It's a great paper. It too, was forgotten about.

And the whole thing was pretty much forgotten about by us until other people, outside the community, rediscovered it right after turbo codes were invented. Turbo codes were first presented in 1993 at the ICC in Geneva. They had actually been invented some four or five years before that by Claude Berrou in France. And

nobody could believe how good the performance was. This guy is not even a communications guy. He must have made a three dB mistake, didn't divide by two somewhere.

He showed you could get within about a dB of capacity with two simple codes. And we'll see that in a second. But, sure enough, people went and verified that these codes work the way he said they did. And, all of a sudden, there's an explosion of interest. And low-density parity-check codes were rediscovered by a couple of people, notably David MacKay who also was a physicist like Berrou, in England.

But he had an interest in information theory, at least he was trying to solve the right problem. And he had similar ideas to what Gallager had had. And he simulated them-- by now, you can simulate these codes on a desktop computer-- and found that they got within a dB of capacity and so forth. And so then we were off to the races, and the whole paradigm of coding changed.

But it certainly is an embarrassment to those of us who were faithfully in the field for 30, 35 years. Why didn't it occur to us that by the '90s the technology was far enough advanced that we could go back and look at what Gallager did and realize that these were actually pretty good codes and now implementable? So I think that's a nice story. It's kind of humbling to those of us in the field. On the other hand, we can say that, jeez, Gallager did it back in 1960. What's the big deal about turbo codes? Depends what perspective you take.

I never know whether I should tell a lot of stories or not so many stories in this class. If you like more stories, just ask. Anyone have any questions at this point?

All right, low-density parity-check codes, Gallager in 1961, forgotten until 1995, rediscovered by David MacKay, Niclas Wiberg, and others. And this was what they look like. The basic idea is we have a parity-check representation of the code, a long code. So the code is the set of all y such that yht equals 0. And that's standard. And the idea now is that this parity-check matrix should be low density or sparse.

In fact, the number of ones in it-- It's a matrix, so that you would think a random

matrix is 0s and ones. m minus k by n matrix would have a number of ones that goes up as n squared. As n becomes large, let's make sure that the number of ones only goes up linearly with n, as the block length n becomes large. So sparse h means number of ones linear with n. And we'll see that implies that the complexity of the graph is linear with n. The ones are going to correspond to the edges in the graph. And we'll have a linear number of edges, also a linear number of nodes or vertices.

OK, in general, what does a code like this look like? We have y 0, y one through-- what does its graph look like, I should say-- yn minus one. And then, so we're going to have n symbols, which in normal graphs we write as equals nodes. Then over here we're going to have n minus k, a lesser number of constraints or checks, which in our notation we've been writing like this. And we're going to have an edge for each one in here.

You can easily see that every one in the parity-check matrix ensures that it says that one of the symbols participates in one of the checks. And therefore, the number of edges is equal to the number of ones. So this is equal number of edges in graph. Not counting these little half edges out over here, again. So there's some random number of edges. We won't have them got to the same place. But this also is only linear with n, here. And, in fact, let's choose these edges. These edges are like a telephone switchboard. It's just a giant plug board where everything that goes out here, you can consider going into a socket.

So let's draw, for instance, for a regular code, let's draw the same number of sockets going out for each of the symbols. Each symbol is going to participate in the same number of checks. Each one is going to have the same degree, so that's why, in graph terms, it's regular. And each one of these checks, we're also going to say has the same degree.

So this was basically the way Gallager constructed codes, regular. We'll later see if there's an advantage to making these somewhat irregular, these degree distributions. But this is certainly the simplest case.

All right, how do we make a random parity-check code? So that we're going to specify n, n minus k. We're going to specify these degrees. So that's not random. But we'll connect the edges through a big switch board which is denoted by pi, which stands for permutations. This is basically just a permutation. We've got to have the same number of sockets on this side as on this side. And this is a just a reordering of these sockets. That's what a telephone switchboard does. Or it's very often in the literature called an interleaver, but, probably, permutation would've been a better word.

All right, so one way of doing it is just to take this edge here, this socket here and, at random, we have-- let's say, e, for number of edges-- e sockets over here and just pick one of those sockets at random. Take the next one and take the e minus one that are left and connect that to one of the the e minus one at random. So this would give you one of the e factorial permutations of e objects. And it'll give you one of them equiprobably. So we're basically talking about an equiprobable distribution over all possible permutations of e elements.

All right, so we consider all these are equally likely when we come to our analysis. This is why these codes are called probabilistic. We set up some elements of their structure, but then let others be chosen randomly. Clearly it's not going to matter for the complexity of the decoding algorithm how this is done. It might have something to do with the performance of the decoding algorithm, but any decoding algorithm is going to work the same way. It's just a matter of when you send a message in here, where does it come out here or vice versa. So it doesn't really affect complexity.

Let's see, there has to be some relation between the left degree d lambda and the right degree d rho here in order that we have the same number of edges on each side. We have e equals n times d lambda, if we look at the left side. But it's also equal to n minus k times the degree on the right side. Again, I'm only counting the internal degree here and leaving aside the external guy. And this gives us the following relationship that n minus k over n is equal d lambda over d rho. Or, equivalently, with this is one minus the rate of the code, or we can write rate equals one minus d lambda over d rho.

So that by deciding what left degree and what right degree to have, we're basically enforcing this relationship which comes out as the rate of the code. For instance, the choice that we're going to talk about throughout is we're going to take a left degree equal to three, right degree equal to six, as I drew it. If you do that, then you're going to get a code of nominal rate 1/2.

It's only a nominal rate because what could happen here? I'm basically choosing a random parity-check matrix, and there's nothing that ensures that all the rows of this parity-check matrix are going to be independent. So I could have fewer than n minus k independent rows in the matrix, which would actually mean the rate would be higher than my design rate, my nominal rate.

In fact, you can ignore this possibility. It's very low probability even if you choose it at random. If it does happen, people are going to forget about the dependent row and pick another independent row. So I won't make a big deal. We'll just call this the rate.

So if this were three here but only four here, then we would have a rate 1/4 code. Well, we'd have to have 3/4s as many checks as we have symbols in order for everything to add up. So that's really, basically, it. Let's just specify the degrees on the left, the degree on the right, pick our permutation at random, pick n or e.

We've designed a low-density parity-check code. And now, how do we do we decode it? Well, we use the sum-product algorithm. We've got a code on a graph. In fact, it's a very simple graph, notice it's characteristic. It's, again, just got equality nodes and zero-sum nodes or check nodes.

These are really the simplest codes. We could, more generally, have more elaborate codes here. That's one of the things Tanner did in is 1981 paper. But this is very simple. All of our internal variables are binary.

I, for one, thought that it would be necessary to go beyond binary internal variables in order to get to capacity but that proved not to be the case. Really, this structure is all you need to get to capacity it turns out. Except for one thing, you need to make

these degrees irregular. And for the most part, you only need to make the left degrees irregular. So we'll get far enough to see why that is going to do the trick.

But right now, I talked about the decoding algorithm. So the decoding algorithm is pretty much what you would think, I would hope. Now that you've seen some product decoding. You receive something on each of these lines, depending what the channel is.

If it's an additive white Gaussian noise channel, you get a real number, received vector that gives you some intrinsic information about the relative likelihoods of 0 and one on each of these lines. So each of these points, you get an in going message based on what you receive.

At a repetition node that message is simply propagated, if you look at the equations of the sum-product algorithm. Repetition nodes just propagate what they receive at one terminal on all the output terminals.

All the d minus one terminals are determined by the by one terminal. So we get the same thing going as and output message. This, very often, is drawn as a Tanner graph. In that case, all we have here is symbols. That's natural to just take the likelihoods of the symbols and send them out. So we haven't done anything more than that.

These messages all arrive here as incoming messages from various far and distant lands. To find the outgoing message say, on this line here, what's our rule? Our rule is to take the incoming messages on all d minus one of these nodes and combine their likelihoods according to which of these is consistent with a 0 output here, and which of these is consistent with the one going out here.

We get a bunch of weights for 0, and a bunch of weights for one. We add them all up. And that gives us now, a re-computed likelihood. A message consisting of probability of 0 given what we've seen so far, and probability of one given what we've seen so far. But now it goes out. Hopefully that's got some improved information in it now coming from all different parts of the graph, extrinsic.

So this, eventually, will come back somewhere say, here and similarly on all the other lines. And now this will give you additional information. You combine these, and now what you have coming out here is a combination of what you had coming in here, which was originally a state of complete ignorance, but now you combine these you get more information going out and so forth.

So you iterate. You basically do a left side iteration, then a right side iteration, a left side iteration, a right side iteration. Hopefully the quality of your APP vectors is continually increasing during this. You're somehow incorporating more and more information. You're hopefully converging. And so, you just keep going back and forth. And low-density parity-check codes you can do this typically 100 times, 200 times. It's a very simple calculation. You can do it a lot. So you can do that.

When do you stop? A very popular stopping rule is to stop as soon as you've got-- If you made hard decisions on the basis of all these messages, that would give you binary values. And if all those binary values check at all these places, then that's a code word. Or that's a trajectory that's consistent with the code word. Basically, everything that's over here then has to be a code word. So whenever all the checks, check just based on hard decisions, you stop.

So if there's very little noise to start with, this can happen very quickly and terminate the calculation well short. Or you have some maximum number of iterations you're going to allow yourself. If you go 200 iterations, and these things still don't check, probably you would say this, I've detected a decoding error.

One characteristic of low-density parity-check codes, is they tend to have good minimum distance. They tend to have a minimum distance that's sort of what you would get in a random-like code, which goes up linearly with the block length. So for a rate 1/2 of code of length a 1,000, a random code would have a minimum distance of about 110.

And a low-density parity-check code would typically have a large, maybe not 110, but a large minimum distance, unless you were just unlucky. So the probability of your actually converging to an incorrect code word is tiny. You don't actually make

decoding errors. All of your errors are detectable. Which is a nice systems advantage some of the time for low-density parity-check codes. Yes?

AUDIENCE: [INAUDIBLE] possible that if you add one variation [INAUDIBLE]

PROFESSOR: Possible in principal but extraordinarily improbable. And once you've converged to a code word, basically, you've got all of all the messages lined up in the same way. It's like a piece of iron ore where you've got all of magnets going in the same direction. What is there in that that's going to cause them to flip? There's no impetus for any of these to-- They'll tend to get more and more convinced that what they're doing is right. They'll reinforce each other.

So, as a practical matter, I don't think you would ever-- The kind of landscape of these codes is that there's large areas of the parameters where it doesn't tell you to go in any particular direction. And then there are deep wells around code words.

So, typically, the decoding algorithm is kind of wandering around the desert here, but then once it gets into a well, it goes wrong boom, down to the bottom of that well. And once it's down here, it's extremely unlikely that it'll spontaneously work itself out come back over here again. Powerful basins of attraction here.

Which is another reason why when we design the code, we're basically designing this landscape. And we can design it to-- when it starts to converge, it really converges fast. That's why some classes of these codes are called tornado codes. They wander for a while, and then when they get the idea, then [WOOM], everything checks very quickly. Yes?

AUDIENCE: [INAUDIBLE] so you would compute the-- at each [UNINTELLIGIBLE] calculate one of the so do you calculate all of the messages that go back from each [UNINTELLIGIBLE] first and then propagate then back--

PROFESSOR: Yes, correct. I'm sorry. We nominally calculate this one, but at the same time, we calculate every other one. All e of them. Each one of these is based on the d rho minus one-- other inputs. So it ignores the message that's coming in on that

particular edge. It uses all the other messages to give and extrinsic message going out. Which you could combine back here or here, there the same thing, to give you your best overall APP right now from your right going in your left going messages. But you don't.

You now use this new information in your next round over here. And the same thing on this side. You always compute one on the basis of all of the others. And the initialization is that initially you have no information coming this way. You can represent that by an APP vector that's 1/2, 1/2. Probability of one is 1/2, probability of 0 is 1/2. Any other questions?

OK, so that's how it works. And it works great. And, as we'll see, even with regular codes you can get within-- additive white Gaussian noise channel terms, you can get within about a dB, a dB and a half of capacity. And merely by making these degree distributions irregular, this is how, Sae-Young Chung got the results that I showed, I think, in chapter one, where you get within 0.04 dB of white Gaussian noise capacity and that's it. So we knew how to get to capacity since 1961 almost, we just didn't realize it. Yeah?

AUDIENCE: With the LDPC regular [UNINTELLIGIBLE]

PROFESSOR: I'm sorry, I missed--

AUDIENCE: With regular LDPC you can still have a certain gap or?

PROFESSOR: With regular LDPC there's a threshold below which the algorithm works, above which the algorithm doesn't work, as we'll see shortly. And the threshold is not capacity. It's somewhat below capacity. By going to irregular you can make that threshold as close to capacity as you like. You can never make it equal to capacity.

All right, so let's talk about turbo codes. None of these ideas is very difficult, which is, again, humbling to those of us in the community who failed to think of them for 35 years. And I really hope some historian of science someday does a nice job on culture, and why some people were blind to this and some people saw it and so forth.

OK, so this is Berrou, Glavieux, and Thitimajshima-- which I always have problems spelling-- in 1993. And their idea was pretty simple too. Their idea is based on two, rate 1/2, convolutional codes. Let me use notation. We have a certain u of d, which is a set of information bits. And to make a rate 1/2 convolutional code, all I have to do is pass that through a certain g of d. So I'm saying my generator matrix for the convolutional code is one and g of d. That'll make a rate 1/2 code. And this, I'm going to call that a parity bit sequence, occurring at the same rate as the information bit sequence.

Easy enough, now, let me take this same information bit sequence, and let me put it through a large permutation. Which, again, I'll represent by pi. And I want you to think of this as taking a 1,000 bit block and mixing it all up or something like that. Actually, it's better to use a so-called convolutional permutation which is kind of stream based, continuous, the same way that convolutional codes are.

All right, but it's basically the same sequence permuted. And let's also put that through another g prime of d, which, very often, is g of d. So I'll just call that g of d again. And this is second parity bit sequence. And this all together, I will call a rate 1/3 code. For every one information bit in, there are three information bits out. Same sense as convolutional.

All right, so the second code, g prime of d is one g prime of d. And the interesting thing is, these codes don't need to be very complicated. One thing they realized is they do want this to be a recursive code. In other words, g of d should look like n of d over d of d.

And one particular purpose of that is that so a single information bit will ring forever. The impulse response to a single information bit is infinite. It takes at least two. And you can prove to yourself there's always some sequence of length two in order to get a finite number of parity bits, a finite number of nonzero parity bits out of here. And when those two bits are permuted, they're going to come up in totally different places, so they are going to look like two individual information bits down here, and therefore they're going to ring for very long time.

The basic idea is, you don't want an information sequence which gives you a short code word out of here and a short code word out of here. And making it recursive, have feedback is what that means, is what you need to do to prevent that.

It also gives you better codes, but I don't think that's really why-- If you have g of d over n of d, this is going to be equivalent to the code we're multiplying to the non-recursive feedback free code, which is d of d, n of d, right? We worked all that out. It's equivalent to the code where you multiply through by the denominator. And so you get the same code with d of d, n of d, but you don't want to do that, because you are concerned here about the input output relationship between input bits and the encoded bits. And you want the information bit sequence to come through by itself, because it's going to be reused. You want the information bit sequence to be in common between these two. That's going to be the length when we decode these codes. Basically, the messages concerning the information bit sequences is what's going to go back and forth between the two codes.

All right, do you get the set up here? I gave you a very different perspective on it by drawing the graph of this code. I mean this is a graph of the code, but it's not one we want. Let's draw the graph of this code.

So turbo code graph, I'll draw a normal graph, but again, if you like to do it as a Tanner style, be my guest. All right, this first code-- Let me do this in two steps. First of all, a rate 1/2 code graph. So for one of these code, what does it look like? It's a trellis graph. Which in the normal graph picture is a very simple chain graph.

It looks like this. Here's the information sequence. Here are the states. So these might be simple codes. These are typically four to 16 states, so two to four bits of state information, not very complicated. So this is what it looks like. We get one of these for each. This is the branch constraint for each unit of time. This would be y0, y1, y2, y3, you get two bits out for each unit of time.

And this case, one of these is actually the input at that time, u0, u2. Or we could adopt some other indexing scheme. It's systematic so, we think of these as the

inputs sort of driving an output. So maybe a better way to do this is-- an alternative way of doing it is like this. But it's all the same thing, right? It's what you've already seen.

So I just want you to get used to this way of writing the graph of a trellis of a rate 1/2 systematic convolutional code. Here are these systematic bits, which are both inputs and, ultimately, outputs on to the channel. And these are the parity bits which are only output onto the channel. All right, and these are the branch constraints for each time. And these are the state's basis for each time. So that I can always draw that, that way.

All right, now up here, I really have two of these codes, possibly identical. And what's the connection between them? The connection between them is that they have the same information, but in some random order, some pseudo-random interleaved order.

So I draw a graph of that, again, focus on the interleaver. Make it nice and big now. There's always going to be a big interleaver in the center of all our charts. This is always the random-like element. Now, let me draw my first convolutional code over here. I'm just going to tilt this on its side. I'm going to put the parity bits on this side, the first parity sequence. So now this is running in time either up to down or down to up. I don't care. So forth, down to here, et cetera, dot, dot, dot, dot, dot, dot--

And I'll put the information bit over here. And I'm going to do it in this way. The information bits are, first of all, used in this code. So I have a little equality. So these are going to be in my information bits. But then they're going to go through this permutation and also be used in another code over on the right side. Is this clear?

So this is everything up to here. I'm about to go into the interleaver. Stop me with questions if anything is mysterious about this. OK, after some huge permutation, they come out over here. And now these are information bits that go into another identical trellis-- or could be identical. So we have a trellis over on this side that's connected to one parity bit and one information bit. It's another rate 1/2 trellis. This is parity bits, these are, again, info bits. Same information bit sequence but

drastically permuted. Is that clear?

So again this is the same idea of a switchboard with sockets we have the same number sockets on the left side and on the right side. And we make a huge permutation that ties one to the other. It's only edges in here. Notice there's no computation that ever takes place in the interleaver. This is , again, it's purely edges, purely sending messages through here. So the analogy of a switchboard is a very good one. Even though we don't have switchboards anymore. Maybe some of you don't even know what a telephone switchboard is? It's a quaint but apt analogy.

All right, so this is the normal graph of a turbo code. I've drawn it to emphasize its commonality with low-density parity-check code. The biggest element is this big interleaver in here, and that's key to the thing working well.

All right, so how would we apply the sum-product algorithm to this code? What would our decoding schedule be? In this case, when we observe the channel, we get messages in all these places. So we get intrinsic information coming in for all the external symbols. First step might be to

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's first step, we can-- In the same way, this is just a distribution node when originally all else is ignorance. So these same messages just propagate out here, and if we like, we can get them out over here. All these are available too.

OK, somebody else. What's the next step? Somebody was just about to say it. Be bold, somebody.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Excuse me?

AUDIENCE: The trellis decoding--

PROFESSOR: Trellis decoding, right. If we forget about that side and just look at what the sum-product algorithm does over here, we're doing the sum-product algorithm on a

trellis, a BCJR algorithm. So given all these messages, we do our backward forward algorithm, alphas and betas, eventually epsilons, extrinsic information coming out on all nodes there, there, there, and so forth.

And that requires going up and down the whole trellis. If this is a 1,000 bit block, as it often is, or larger it means going from top to bottom through the whole block and doing one sweep of a BCJR algorithm through that block, which will generate a bunch of extrinsic information over here.

Now, we have more over here. We still got the same intrinsic information coming in here, but now we can take this new extrinsic information and combine it with this intrinsic information and, hopefully, get an improved, higher quality information. Because we've incorporated all the information from all these receive symbols on this side, and we send that over here.

And if you think of it in signal to noise ratio terms, as some people do, you hopefully have improved the signal to noise ratio in your information that's coming out. It's as though you received these bits over a higher quality channel. They tell you more about the information bits then you got just from looking at the raw data from the channel.

That's basically where turbo comes from. You then do BCJR over here using this improved intrinsic information, of course, the parity information hasn't improved. You might think of ways to improve this too. But this was the basic scheme. It works just fine.

You come over here. You then do a complete sweep through a 1,000 bits or whatever, and you develop outgoing messages here. Which again, let's hope we've improved the signal-noise ratio a little bit more, that these are even more informative about the input bits. And they come back, and now, with this further improved information, we can sweep through here again. That's where the turbo comes from. You keep recycling the information left to right and right to left. And, hopefully, you build up power. You lower the signal-noise ratio each time. Yes?

AUDIENCE: [INAUDIBLE] and then you [UNINTELLIGIBLE] the other side is kind of [? encouraging. ?] Now, why can't you, because on this side, actually, the equilibrium is just the [UNINTELLIGIBLE] version of the other side. Why would you do it [UNINTELLIGIBLE] carry out, and do it again?

PROFESSOR: So a good question. You certainly could. So after you've done it twice, you basically processed all the same things as this thing does in one iteration, you spent twice as much effort. The question is, is now the quality of your information any better? And I haven't done this myself, but I suspect people in the business did this, and they found they didn't get enough improvement to make up for the additional computation. But that's an alternative that you could reasonably try.

AUDIENCE: The reason I say that is because if you do it that way, that means for the second decoding process of the left side, the information is very fresh. Because it's sort of [? right ?] information. If you do this and this and this, it looks like there is enough propogation of the same provision to itself.

PROFESSOR: There certainly is. There are certainly cycles in this graph.

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, well, these are great questions. What I hope that you will get from this class is that you will be able to go back and say, well why don't we try doing it this way? Whereas everybody else has just been following what they read in the literature, and it says, to do it first here, and then there, and then there. Try it. Maybe it's good. You're suggesting that suppose this guy it actually has poor information. He could actually make things worse over here.

AUDIENCE: Yeah, and also [INAUDIBLE]

PROFESSOR: So then if this guy just operated on the fresh information. It's a probabilistic thing whether this would work better on the average. I don't know. I suspect that, once upon a time, all these things have been tried. But certainly, if you're in any kind of new situation, new channel, new code, you might think to try it again. It's a very reasonable suggestion.

Because computation is not free, there is a marked difference in turbo decoding in that you're spending a lot more time decoding each side. This PCJR algorithm is a lot more complicated than just executing-- well, it's gone now-- but the sum-product update rule for equals or 0 sum node. That's a very simple operation. On the other hand, it doesn't get that much done. You can think of this as doing a lot of processing on this information over here before passing it back. Then this does a lot of processing before passing it back again.

And so for turbo codes, typically, they only do 10 or 20 iterations. Whereas for low-density parity-check codes, one does hundreds of iterations, of much simpler iterations. But, in either case, you've got to deal with the fact that as you get up in the number of iterations, you are getting to the region where cycles begin to have an effect, where you are re-processing old information as though it was new, independent information.

This tends to make you overconfident. Once you get in the vicinity of a decision or of a fixed point, whether it's a decision or not, it tends to lock you into that fixed point. Your processing, after a while, just reinforces itself.

Now, another thing, turbo codes in contrast to low-density parity-check codes, generally have terrible minimum distance. If you can work through the algebra, and there's always some weight 2 code word that produces maybe eight things out here and 10 out here. So the total minimum distance is only 20, no matter how long the turbo code gets. All right, so when I say terrible, I don't mean two, but I mean 20 or 30 could be a number for minimum distance.

Certainly, this is called parallel concatenation, where you have these two codes in parallel, and just pass information bits back and forth between them. It's typical that you get poor minimum distance which eventually shows up as an error floor.

In all these codes, when you draw the performance curve, as we always draw it, you tend to get what's called a waterfall region, where things behave in a very steep Gaussian like measure. When you measure distance from the capacity, you might

be-- Oh, let's do this first, assess an r norm. So capacity is always at 0dB. And this is probability of error. So this is your distance from capacity, and this might be 1dB, or 3/10 dB, or something very good. But then, as you get down below, some cases it could be 10 to the minus three or four, some cases it could be 10 to the minus seven.

So where it starts is important. Then this tends to flatten out like that. And this is simply what you get from the union-bound estimate for poor distance. So, in this region, other things hurt you worse than the distance did. If you are purely limited by distance, then you would have a performance like this.

There a very, tiny number of minimum distance code words. There may be only a few in 1,000 length block, so the error coefficient is very small. That's why this is way down here. The interleaver tends to give you a 1/n effect on the error coefficient. So this is way down here, and it doesn't bother you in the waterfall region, but then at some point it becomes dominant.

You can think of these as being two types of curves. This is the iterative decoding curve. This is simple maximum likelihood of decoding when you have minimum distance of d with a small error coefficient. And so where this error floor is an important issue for turbo codes. In some communications applications really all you're interested in is 10 to the minus four, 10 to the minus five. And then you really want to put all of your emphasis on the waterfall region and just keep the error floor from being not too bad.

So that would be a very good application for turbo code. But if, as some people think they want, error probabilities like 10 to the minus 12, or 10 to the minus 15, probably turbo code is not going to be your best candidate. And it's not that you couldn't create a code with better minimum distance, but it's going to get harder. I don't know that much effort has gone into this, because we have low-density parity-check codes for this other application.

Any questions on turbo code? That's about all I'm going to say. Yeah?

AUDIENCE: Why not expurgate the bad code words?

PROFESSOR: Expurgate the bad code words. Well, how exactly are you going to do that?

AUDIENCE: Any malady report. We disallow certain using [? passes. ?]

PROFESSOR: So you're not going to mess with the g of d, but you're going to go through, you going to find all the u of d's that lead to low weight sequences, and then you're going to expurgate them. Of course, it's a linear code, so if there's a low weight sequence in the neighborhood of 0, then there is going to be a corresponding sequence in the neighborhood of any of the code words. So you're going to have to expurgate, somehow, around all the code words. I don't know how to do it.

AUDIENCE: You would sacrifice rate.

PROFESSOR: You will sacrifice rate? Oh, one other thing I should mention. Very commonly in turbo codes, they don't send all of these bits. They send fewer bits, which. in effect, raises the rate. But then, of course, makes decoding harder, because you don't get any received information from the bits that you don't send. But there are various philosophies of what's called puncturing-- that I'm certainly no expert on-- and it's possible that by being very clever about puncturing you could do something about minimum distance.

Shortening is where you hold some of these bits to 0 and don't send them. Puncturing mean to let them go free and don't send them. So I don't know. But I don't think this is a very profitable way to go. But it's certainly a reasonable question to ask. Any time that you get into a minimum distance limited situation, you should think about, is there some way I can just expurgate that small fraction of code words in the neighborhood of every code word that have that minimum distance.

This, as I said, is called parallel concatenation. Sorry, I didn't mean to cut off questions about turbo codes. I did say, I'm not going to talk about them again. They, of course, gotten a lot of publicity, are very popular initially. This was a revolution in coding. Turbo codes got into all the new standards and so forth. Every communications company had a small group simulating turbo codes. And so they

got a tremendous amount of momentum. Low-density parity-check codes really didn't get looked at till a couple years later.

As you'll see, there's a lot more analytical work you can do a low-density parity-check codes, so they're popular in the academic community for that reason. But, in fact, it turned out that the analysis allowed you to do fine tuning of the design of low-density parity-check codes. There are more knobs that you can do, such that you really can get closer to capacity with low-density parity-check codes than with turbo codes.

They have other differences in characteristics, but, for both of these reasons, there is certainly a move now. The low-density parity-check codes seem to be the favored ones in any new standards. But we still have a lot of existing standards, and people who've already developed turbo code chips and so forth who have a vested interest against moving. Perhaps analog devices is one like that.

So, although you can see the trend is toward low-density parity-check codes, there are many industrial and competitive factors and inertia factors. They'll both be around for a long time.

AUDIENCE: [INAUDIBLE] turbo code they have a patent you have to pay a royalty to the company. And with LDPC you don't have to--

PROFESSOR: Yes, since this is a theoretical class, but, of course, issues like patents are very important. Berrou, Glavieux, and all were working at ENST Bretagne up in Brittany. They were funded by France Telecom. France Telecom went out and owns all the turbo code patents and has decided to enforce them.

Whereas Bob Gallager's patents all-- We owned the ball. We never made a dime. They all duly expired around 1980. And no one needs to worry about at least basic patents on low-density parity-check codes. Of course, refinements can always be patented. So I don't know what the situation is there. And that often makes a big difference as to what people want to see standardized and used in practice. So, excellent point. All right, well, we can come back to this at any point.

Serial concatenation-- Here the idea is you, basically, go through one code, then you interleave, and then you go through another code. This is the kind of setup that I talked about back when we talked about-- This could be a small block code or convolutional code that you do maximum likelihood decoding on, need to do a Viterbi algorithm.

And this could be a Reed-Solomon code, and the interleaver might be in there for some kind of burst protection or to decorrelate any memory in the channel out here. So that's the context in which concatenated codes were originally invented. That's the way they're used on space channel for instance, additive white Gaussian noise channel.

But here we're talking about the same block diagram, but we're talking about much simpler codes. And again, the idea is to build a big code using very simple component codes, which is what we did in low-density parity-check codes. That's a big code built up, basically, out of 0 sum and repetition codes. Turbo code is a code built up out of two simple convolutional codes.

Here, I'm going to talk about repeat-accumulate codes just because they are so simple. These were proposed by Divsalar and McEliece really for the purpose of trying to find a simple enough turbo code so you could prove some theorems about it. There are hardly any theorems about turbo code. It's all empirical.

We've got now some nice analytical tools. In particular, the exit chart, which was a very nice, empirical engineering chart like Bode plots and Nyquist plots, things like that. It has a real engineering flavor, but it works very well.

The repeat accumulate code is basically this. We take input bits here. We repeat n times. So this, if you like, is an n one n repetition code. That's pretty simple. We have a stream and a long block, we would then permute everything within the block. And then this is a rate one or one over one convolutional code called an accumulator.

This is the code with transfer function one over one plus d, that means that y k plus

one is equal to y k plus x k plus one. So we have x coming in. Here is one delay element. Here's y k coming out. And there is what that looks like. Xk plus one. yk plus one. So it's a very simple rate one convolutional encoder with feedback where I guess we take this as the output.

That's called an accumulator because, if you look at it, what is yk doing? You can show that yk plus one is simply the sum up to k plus one of all the xk prime. k prime equals whatever it starts at through k plus one of all the xk prime.

It's nothing more than saying than saying, g of d equals one over one plus d, equals one plus d plus e squared plus d. The impulse response to this is infinite. And every past xk shows up in every y. This is just the sum of all the xks. But, of course, in the binary field it's the mod 2 sum, so it's only going to be 0 and one. Well, it's a little two-state, rate one convolutional code. About as simple as you can get and obviously useless as a standalone coded. Rate one coders have no redundancy. They can't protect against anything.

So that's the structure which prior to turbo codes or capacity approaching codes, you would have said, what? What are we trying to achieve with this? And the reason I talk about them it that the codes actually does pretty well.

All right, what's its graph look like? Over here we have input bits. The input bits aren't actually part of the output in this case, so I won't draw the dongle. We just have an equals sign with n call n equals three, repetition from each.

Oh, what's the overall rate of this code? Here we have a rate one over n code. We don't do anything to the rate here, so the overall rate is one over n. So let's take a rate 1/3. So the graph, dot, dot, dot, looks something like this. And then, as always, this element. We take these bits in. We permute them in some sort of random permutation.

And then over here, what do we have? Well, we have a trellis again. In this case, I can draw the trellis very explicitly. If we consider this to be the information bit, and this to be the previous state bit, so this is yk. And here's-- We're considering the y's

to be the output, so this is xk. I think it works either way. Let's call this x k plus one plus y k equals y k plus one. So this enforces the constraint. This is really the branch. This is just to propagate the y k and present them as outputs.

So this little two state code, here's the state. The state is also the output. The state is what's in here. And this is what it's trellis looks like. That's the way this looks. Of course, again, we need an equal number of sockets on this side and on this side.

So how would you decode this code now? What do we see initially? We initially get received data, measured intrinsic information, for each of the bits that we actually send over the channel. And what's our next step? Trellis? Yeah. We do the sum-product algorithm on this trellis. So we just do BCJR ignoring the fact that it's rate one. That doesn't matter.

And so we will get some information for each of these guys that we compute from the forward backward algorithm. We'll ultimately get messages going in this direction. Which come over here when we get them all. And again, from these two messages, we can compute an output message here, from these two, compute one there, from these two, compute one there. So we combine and re-propagate all these messages. And that gives us new information over here.

And now, based on that, we can do BCJR again. So again, it's this left right sort of alternation, one side, other side which, eventually, spreads out the data through the whole block. We can use all the received data in a block to decode all the data in the block.

Now does this work as well as either of the two previous ones I put up, low-density parity-check or turbo? Not quite, but it works amazingly well. This is the moral here. That even this incredibly simple code, where nothing is going on-- Again, all we have is binary state variables. We have equals and 0 sum nodes, that's it. It's made out of the simplest possible elements.

Even this code can get within a dB or two of channel capacity. And I'm old enough to know that in 1993, we would have thought that was a miracle. So if this had been

the first code proposed at the ICC in Geneva in 1993, and people had said, we can get a dB and a half from capacity, everyone would have said, totally unbelievable until they went home and simulated it.

So I put this up just to illustrate the power of these ideas, and where they come from. Of course, this idea of using the permutation as your pseudo-random element so that you, from very simple components, you get the effect of a large pseudo-random code, could be a block length, a 1,000, 10,000, what have you, this is absolutely key. And this, basically, is what Gallager had when he said, if I just use my parity-check matrix at random, even if it is sparse, it's going to give me a random-like, which means a pretty good, code, because that's the intuition I get from Shannon and Elias.

That's the important part of the code construction, but then the decoding part of the power is the sum-product algorithm used in an iterative way. The idea that you can simply let this algorithm run, and, if the code is properly designed, it will eventually integrate all the information over the block and converge to some fixed point.

Although, even for such a simple code as this, for this code McEliece and his students have a few theorems and can say something about the convergence behavior, and what it's actually going to do. Even for this code, they can't get anything very definitive, I would say.

So to me, the more important thing about-- these are called RA codes-- is that incredibly simple codes can do better than concatenation of a 1,000 byte Reed-Solomon code over g f of two to the 10 with a two to the 14 state Viterbi decoder. That doesn't do as well as this crummy little thing does. So that's good to know.

Any questions? Well, it's a good stopping place. So next time, now, if you kind of have an impressionistic idea of how we do these codes. Next time, I'm going to analyze low-density parity-check codes over the binary erasure channel, where you can do exact analysis, at least in the asymptotic case. And that will show very clearly how it is that these codes actually work. It's a good model for all the codes in general. I guess that means that you can't do a lot of the homework again. So for

Wednesday, what should we do?

AUDIENCE: [INAUDIBLE] on Friday then?

PROFESSOR: Friday or Monday? Monday seems to be popular. So for Wednesday, just do-- There's one problem left over from problem set eight. And that's kind of a grungy problem but worthwhile doing. Try to do the sum-product algorithm at least once. So just do that for Wednesday, and then for next Monday, a week from today, problem set nine other than that. I'll see you Wednesday.