PROFESSOR: So the handouts will be just the problem set 8 solutions, of which you already have the first two. Remind you problem set 9 is due on Friday, but we will accept it on Monday if that's when you want to hand it in to Ashish. And problem set 10 I will hand out next week, but you won't be responsible for it. You could try it if you're so moved.

OK. We're in the middle of chapter 13. We've been talking about capacity approaching codes. We've talked about a number of classes of them, low density parity check, turbo, repeat accumulate, and I've given you a general idea of how the sum product decoding algorithm is applied to decode these codes. These are all defined on graphs with cycles, in the middle of which is a large pseudo random interleaver. The sum product algorithm is therefore done iteratively. In general, the initial observed information comes in on one side, the left side or the right side, and the iterative schedule amounts to doing first the left side, then the right side, then the left side, then the right side, until you converge, you hope. That was the original turbo idea, and that continues to be the right way to do it.

OK. Today, we're actually going to try to do some analysis. To do the analysis, we're going to focus on low density party check codes, which are certainly far easier than turbo codes to analyze, because they have such simple elements. I guess the repeat accumulate codes are equally easy to analyze, but maybe not as good in performance. Maybe they're as good, I don't know. No one has driven that as far as low density parity check codes.

Also, we're going to take a very simple channel. It's actually the channel for which most of the analysis has been done, which is the Binary Erasure Channel, where everything reduces to a one-dimensional problem, and therefore we can do things quite precisely. But this will allow me to introduce density evolution, which is the generalization of this for more general channels like the binary input added white Gaussian noise channel, if I manage to go fast enough.

I apologize today, I do feel in a hurry. Nonetheless, please ask questions whenever you want to slow me down or just get some more understanding.

So, the Binary Erasure Channel is one of the elementary channels, if you've ever taken information theory, that you look at. It has two inputs and three outputs. The two inputs are 0 and 1, the outputs are 0 and 1 or an erasure, an ambiguous output. If you send a 0, you can either get the 0 correctly, or you could get an erasure. Might be a deletion, you just don't get anything. Similarly, if you send a 1, you either get a 1 or an erasure. There's no possibility of getting something incorrectly. That's the key thing about this channel.

The probability of an erasure is p, regardless of whether you send 0 or 1. So there's a single parameter that governs this channel. Now admittedly, this is not a very realistic channel. It's a toy channel in the binary case.

However, actually some of the impetus for this development was the people who were considering packet transmission on the internet. And in the case of packet transmission on the internet, of course, you have a long packet, a very non-binary symbol if you like, but if you consider these to be packets, then on the internet, you either receive the packet correctly or you fail to receive it. You don't receive it at all, and you know it, because there is an internal party check in each packet.

So the q-ary erasure channel is in fact a realistic model, and in fact, there's a company, Digital Fountain, that has been founded and is still going strong as far as I know, which is specifically devoted to solutions for this q-ary erasure channel for particular kinds of scenarios on the internet where you want to do forward error correction rather than repeat transmission.

And a lot of the early work on the analysis of these guys-- Luby, Shokrollahi, other people-- they were some of the people who focused on low density party check codes immediately, following work of Spielman and Sipser here at MIT, and said, OK, suppose we try this on our q-ary erasure channel. And they were able to get very close to the capacity of the q-ary erasure channel, which is also 1 minus p.

This is the information theoretic capacity of the binary channel. It's kind of obvious that it should be 1 minus p, because on the average, you get 1 minus p good bits out for every bit that you send in. So the maximum rate you could expect to send over this channel is 1 minus p.

OK. Let's first think about maximum likelihood decoding on this channel. Suppose we take a word from a code, from a binary code, and send it through this channel, and we get some erasure pattern at the output. So we have a subset of the bits that are good, and a subset that are erased at the output. Now what does maximum likelihood decoding amount to on this channel?

Well, the code word that we sent is going to match up with all the good bits received, right? So we know that there's going to be at least one word in the code that agrees with the received sequence in the good places. If that's the only word in the code that agrees with the received word in those places, then we can declare it the winner, right? And maximum likelihood decoding succeeds. We know what the channel is, so we know that all the good bits have to match up with the code word.

But suppose there are 2 words in the code that match up in all the good places? There's no way to decide between them, right? So basically, that's what maximum likelihood decoding amounts to. You simply check how many code words match the received good bits. If there's only one, you decode. If there's more than one, you could flip a coin, but we'll consider that to be a decoding failure. You just don't know, so you throw up your hands, you have a detected decoding failure.

So in the case of a linear code, what are we doing here? In the case of a linear code, consider the parity check equations. We basically have n minus k parity check equations, and we're trying to find how many code sequences that solve those parity check equations. So we have n minus k equations, n unknowns, and we're basically just trying to solve linear equations. So that would be one decoding method for maximum likelihood decoding.

Solve the equations. If you get a unique solution, you're finished. If you get a space of solutions, so dimension one or more, you lose. OK? So we know lots of ways of solving linear equations like Gaussian elimination, back propagation/back substitution (I'm not sure exactly what it's called). That's actually what we will be doing with low density parity check codes. And so, decoding for the binary ratio channel you can think of as just trying to solve linear equations. If you get a unique solution, you win, otherwise, you fail.

Another way of looking at it in a linear code is what do the good bits have to form? The erased bits have to be a function of the good bits, all right? In linear code, that's just a function of where the good bits are. We've run into this concept before. We called it an information set. An information set is a subset of the coordinates that basically determines the rest the coordinates. If you know the bits in that subset, then you know the code word. You can fill out the rest of the code word through some linear equation.

So basically, we're going to succeed if the good bits cover an information set, and we're going to fail otherwise. So how many bits do we need to cover an information set? We're certainly going to need at least k.

Now today, we're going to be considering very long codes. So suppose I have a long (n,k) code. I have an (n,k) code, and I transmit it over this channel. About how many bits are going to be erased? About pn bits are going to be erased, or (1 minus p) times n. We're going to get approximately-- law of large numbers-- (1 minus p) times n unerased bits, and this has to be clearly greater than k.

OK, so with very high probability, if we get more than that, we'll be able to solve the equations, find a unique code word. If we get fewer than that, there's no possible way we could solve the equations. We don't have enough left. So what does this say? This says that k over n, which is the code rate, has to be less than 1 minus p in order for this maximum likelihood decoding to work with a linear code over the binary erasure channel, and that is consistent with this.

If the rate is less than 1 minus p, then with very high probability you're going to be successful. If it's greater than 1 minus p, no chance, as n becomes 1. OK? You with me?

AUDIENCE: [UNINTELLIGIBLE]?

PROFESSOR: Well, in general, they're not, and the first exercise on the homework says take the 844 code. There's certain places where if you erase 4 bits, you lose, and there are other places where if you erase 4 bits, you win. And that exercise also points out that the low density parity check decoding that we're going to do, the graphical decoding, may fail in a case where maximum likelihood decoding might work.

But maximum likelihood decoding is certainly the best we can do, so it's clear. You can't signal at a rate greater than 1 minus p. You just don't get more than 1 minus p bits of information, or n times (1 minus p) bits of information in a code word of length n, so you can't possibly communicate more than n times (1 minus p) bits in a block.

OK. So what are we going to try to do to signal over this channel? We're going to try using a low density parity check code. Actually, I guess I did want this first. Let me talk about both of these back and forth. Sorry, Mr. TV guy.

So we're going to use a low density parity check code, and initially, we're going to assume a regular code with, say, left degree is 3 over here, right degree is 6. And we're going to try to decode by using the iterative sum product algorithm with a left right schedule. OK. I can work either here or up here.

What are the rules for sum product update on a binary erasure channel? Let's just start out, and walk through it a little bit, and then step back and develop some more general rules. What is the message coming in here that we receive from the channel? We're going to convert it into an APP vector. What could the APP vector be? It's either, say, 0, 1 or 1, 0, if the bit is unerased. So this would be-- if we get this, we know a posteriori probability of a 0 is a 1, and of a 1 is a 0. No question, we have certainty. Similarly down here, it's a 1. And in here, it's 1/2, 1/2. Complete uncertainty. No information.

So, we can get-- those are our three possibilities off the channel. (0,1), (1,0), (1/2, 1/2). Now, if we get a certain bit coming in, what are the messages going out on each of these lines here? We actually only need to know this one. Initially, everything inside here is complete ignorance. Half, 1/2 everywhere. You can consider everything to be erased.

All right. Well, if we got a known bit coming in, a 0 or a 1, the repetition node simply says propagate that through all here. So if you worked out the sum product update rule for here, it would basically say, in this message, if any of these lines is known, then this line is known and we have a certain bit going out. All right? So if 0, 1 comes in, we'll get 0, 1 out. It's certainly a 1.

Only in the case where all of these other lines are erased-- all these other incoming messages are erasures-- then we don't know anything, then the output has to be an erasure. All right? So that's the sum product update rule at a equals node. All right? If any of these d minus 1 incoming messages is known, then the output is known. If they're all unknown, then the output is unknown.

You're going to find, in general, these are the only kinds of messages we're ever going to have to deal with. Either, we're basically going to take known bits and propagate them through the graph-- so initially, everything is erased, and after awhile, we start learning things. More and more things become known, and we succeed if everything becomes known inside the graph. All right? So it's just the propagation of unerased variables through this graph.

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: No. So they're not only known, but they're correct. And like everything else, you can prove that by induction. The bits that we receive from the channel certainly have to be consistent with the correct code word. All these internal constraints are the constraints of the code, so we can never generate an incorrect message. That's basically the hand waving proof of that.

OK. So we're going to propagate either known bits or erasures in the first iteration. And what's the fraction of these lines that's going to be erased in a very long code?

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: Its going to be p. All right? So initially, we have fraction p erased and fraction 1 minus p which are good. OK. And then, this, we'll take this to be a perfectly random interleaver. So perfectly randomly, this comes out there. OK?

All right, so now we have various messages coming in over here. Some are erased, some are known and correct, and that's the only things they can be. All right, what can we do on the right side now? On the right side, we have to execute the sum product algorithm for a zero sum node of this type. What is the rule here? Clearly, if we get good data on all these input bits, we know what the output bit is. So if we get five good ones over here, we can tell what the sixth one has to be.

However, if any of these is erased, then what's the probability this is a 0 or a 1? It's 1/2, 1/2. So any erasure here means we get no information out of this node. We get an erasure coming out. All right, so we come in here, and if p is some large number, the rate of this code is 1/2. So I'm going to do a simulation for like p equals a little less-- small enough so that this code could succeed, 0.4-- so the capacity is 0.6 bits per bit. But if this is 0.4, what's the probability that any 5 of these are all going to be unerased? It's pretty small.

So you won't be surprised to learn that the probability of an erasure of coming back-- call that q-- equals 0.9 or more, greater than 0.9. But it's not 1. So for some small fraction of these over here, we're going to get some information, some additional information, that we didn't have before. And this is going to propagate randomly back, and it may allow us to now know some of these bits that were initially erased on the channel.

So that's the idea. So to understand the performance of this, we simply track-- let me call this, in general, the erasure probability going from left to right, and this, in general, we'll call the erasure probability going from right to left. And we can actually compute what these probabilities are for each iteration under the assumption that the code is very long and random so that every time we make a computation, we're dealing with completely fresh and independent information. And that's what we're going to do. Yes?

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: When they come from the right side, they're either erased or they're consistent. I argued before, waving my hands, that these messages could never be incorrect. So if you get 2 known messages, they can't conflict with each other. Is that your concern?

AUDIENCE: Yeah. Because you're randomly connecting [UNINTELLIGIBLE], so it might be that one of the plus signs gave you an [UNINTELLIGIBLE], whereas another plus sign gave you a proper message. And they both run back to the same equation.

PROFESSOR: Well, OK. So this is pseudo random, but is chosen for once and for all. It determines the code. I don't re-choose it every time, but when I analyze it, I'll assume that it's random enough so that the bits that enter into any one calculation are bits that I've never seen before, and therefore can be taken to be entirely random. But of course, in actual practice, you've got a fixed interleaver here, and you have to, in order to decode the code.

But the other concern here is if we actually had the possibility of errors, the pure binary erasure channel never allows errors. If this actually allowed a 0 to go to a 1 or a 1 to go to 0, then we'd have an altogether different situation over here, and we'd have to simply honestly compute the sum product algorithm and what is the APP if we have some probability of error. And they could conflict, and we'd have to weigh the evidence, and take the dominating evidence, or mix it all up into the single parameter that we call the APP.

All right. So let me now do a little analysis. Actually, I've done this a couple places. Suppose the probability of erasure here-- this is the q right to left parameter. Suppose the probability of q right to left is 0.9, or whatever, and this is the original received message from the channel, which had an erasure probability of p. What's the q left to right? What's the erasure probability for the outgoing message?

Well, the outgoing message is erased only if all of these incoming messages are erased. All right, so this is simply p times q right to left, d minus 1. OK?

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: Assuming it's a long random code, so everything here is independent. I'll say something else about this in just a second. But let's naively make that assumption right now, and then see how best we can justify it.

What's the rule over here? Here, we're over on the right side if we want to compute the right to left. If these are all erased with probability q left to right, what is the probability that this one going out is erased? Well, it's easier to compute here the probability of not being erased. This is not erased only if all of these are not erased. So we get q right to left. One minus q right to left is equal to 1 minus q left to right, to the d minus 1. And let's see, this is d right, and this is d left. I'm doing it for the specific context.

OK, so under the independence assumption, we can compute exactly what these evolving erasure probabilities are as we go through this left right iteration. This is what's so neat about this whole thing.

Now, here's the best argument for why these are all independent. Let's look at the messages that enter into, say, a particular-- this is computing q left to right down here. All right, we've got something coming in, one bit here. We've got more bits coming in up here, and here, which originally came from bits coming in up here. We have a tree of computation.

If we went back through this pseudo random but fixed interleaver, we could actually draw this tree for every instance of every computation, and this would be q left to right at the nth iteration, this is-- I'm sorry. Yeah, this is q left to right at the nth iteration, this is q right to left at the n minus first iteration, this is q left to right at the n minus first iteration, and so forth.

Now, the argument is that if I go back-- let's fix the number of iterations I go back here-- m, let's say, and I want to do an analysis of the first m iterations. I claim that as this code becomes long, n goes to infinity with fixed d lambda, d rho, that the probability you're ever going to run into repeated bit or message up here goes to 0. All right?

So I fix the number of iterations I'm going to look at. I let the length of the code go to infinity. I let everything be chosen pseudo randomly over here. Then the probability of seeing the same message or bit twice in this tree goes to 0. And therefore, in that limit, the independence assumption become valid. That is basically the argument, all right? So I can analyze any fixed number of iterations in this way.

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: OK, yes. Good. So this is saying the girth is probabilistically-- so limit in probability going to infinity, or it's also referred to as the locally tree-like assumption. OK, graph in the neighborhood of any node-- this is kind of a map of the neighborhood back for a distance of m-- we're not ever going to run into any cycles. Good, thank you.

OK, so under that assumption, now we can do an exact analysis. This is what's amazing. And how do we do it? Here's a good way of doing it. We just draw the curves of these 2 equations, and we go back and forth between them. And this was actually a technique invented earlier for turbo codes, but it works very nicely for low density parity check code analysis. It's called the exit chart.

I've drawn it in a somewhat peculiar way, but it's so that it will look like the exit charts you might see an in the literature. So I'm just drawing q right to left on this axis, and q left to right on this axis. I want to sort of start in the lower left and work my way up to the upper right, which is the way exit charts always work. So to do that, I basically invert the axis and take it from 1 down to 0.

Initially, both of these-- the probability is one that everything is erased internally on every edge, and if things work out, we'll get up to the point where nothing is erased with high probability.

OK, these are our 2 equations just copied from over there for the specific case of left degree equals 3 and right degree equals 6. And so I just plot the curves of these 2 equations. This is done in the notes, and the important thing is that the curves don't cross, for a value of p equal 0.4. One of these curves depends on p, the other one doesn't. So this is just a simple little quadratic curve here, and this is a fifth order curve, and they look something like this.

What does this mean? Initially, the q right to left is 1. If I go through one iteration, using the fact that I get this external information-- extrinsic information-- then q left to right becomes 0.4, so we do to the outer. Now, I have q left to right propagating to the right side, and at this point, I get something like 0.922, I think is the first one. So the q right to left has gone from 1 down to 0.9 something. OK, but that's better.

Now, with that value of q, of course I get a much more favorable situation on the left. I go over to the left side, and now I get some p equal to-- this is all done in the notes-- 0.34. So I've reduced my erasure probability going from left to right, which in turn, helps me out as I go over here, 0.875, and so forth.

Are you with me? Does everyone see what I'm doing? Any questions? Again, I'm claiming this is an exact calculation-- or I would call it a simulation-- of what the algorithm does in each iteration. First iteration, first full, left, right, right left, you get to here. Second one, you get to here, and so forth. And I claim as n goes to infinity, and everything is random, this is the way the erasure probabilities will evolve. And it's clear visually that if the curves don't cross, we get to the upper right corner, which means decoding succeeds. There are no erasures anywhere at the end of the day. And furthermore, you go and you take a very long code, like 10 to the seventh bits, and you simulate it on this channel, and it will behave exactly like this. OK, so this is really a good piece of analysis.

So this reduces it to very simple terms. We have 2 equations, and of course they meet here at the (0,0) point. Substitute 0 in here, you get 0 there. Substance 0 here, you get 0 there. But if they don't meet anywhere else, if there's no fixed point to this iterative convergence, then decoding is going to succeed. So this is the whole question: can we design 2 curves that don't cross?

OK. So what do we expect now to happen? Suppose we increase p. Suppose we increase p to 0.45, which is another case that's considered in the notes, what's going to happen? This curve is just a simple quadratic, it's going to be dragged down a little bit. We're going to get some different curve, which is just this curve scaled by 0.45 over 0.4. It's going to start here, and it's going to be this scaled curve. And unfortunately, those 2 curves cross.

So that's the way it's going to look, and now, again, we can simulate iterative decoding for this case. Again, initially, we'll start out. We'll go from 1, 0.45 will be our right going erasure probability. We'll go over here, make some progress, but what's going to happen? We're going to get stuck right there. So we find the fixed point. In fact, this simulation is a very efficient way of calculating what the fixed point of these 2 curves are. Probably some of you are analytical whizzes and can do it analytically, but it's not that easy for a quintic equation.

In any case, as far as decoding is concerned-- all right, this code doesn't work on an erasure channel which has an erasure probability of 0.45. It does work on one that has an erasure probability of 0.4. That should suggest to you-- yeah?

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: Yes, so this code doesn't get to capacity. Too bad. So I'm not claiming that a regular d left equals 3, d right equals 6 LDPC code can achieve capacity. There's some threshold for p, below which it'll work, and above which it won't work. That threshold is somewhere between 0.4 and 0.45. In fact, it's 0.429 something or other. So this design approach will succeed it's near capacity, but I certainly don't claim this is a capacity approaching code.

I might mention now something called the area theorem, because it's easy to do now and it will be harder to do later. What is this area here? I'm saying the area above this curve here. Well, you can do that simply by integrating this. It's integral of p times q-squared dq from 0 to 1, and it turns out to be p over 3. Believe me? Which happens to be p over the left degree. Not fortuitously, because this is the left degree minus 1. So you're always going to get p over the left degree.

And what's the area under here? Well, I can compute-- basically change variables to 1 minus q, q prime, and 1 minus q is q prime over here, and so I'll get the same kind of calculation, 0 to 1, this time q to the fifth over pq, which is 1/6, which not fortuitously is 1 over d right. So the area here is p over 3, and the area here is-- under this side of the curve is-- that must be 5/6. Sorry, so the area under this side is 1/6 so it's 1 minus this.

It's clearly the big part, so this is 5/6. All right. I've claimed my criterion for successful decoding is that these curves not cross. All right, so for successful decoding, clearly the sum of these 2 areas has to be less than 1, right? So successful decoding: a necessary condition is that p over d_lambda -- let me just extend this to any regular code-- plus 1 minus 1 over d_rho has to be less than 1.

OK, what does this sum out to? This says that p has to be less than d_lambda over d_rho, which happens to 1 minus r, right? Or equivalently, r less than 1 minus p, which is capacity. So what did I just prove very quickly? I proved that for a regular low density parity check code, just considering the areas under these 2 curves and the requirement that the 2 curves must not cross, I find that regular codes can't possibly work for a rate any greater than 1 minus p, which is capacity. In fact, the rate has to be less than 1 minus p, strictly less, in order for there to-- unless we were lucky enough just to get 2 curves that were right on top of each other. I don't know whether that would work or not. I guess it doesn't work. But we'd need them to be just a scooch apart.

OK, so I can make an inequality sign here. OK, well that's rather gratifying. What do we do to improve the situation? OK, one-- it's probably the first thing you would think of investigating maybe at this point, why don't we look at an irregular LDPC code?

And I'm going to characterize such a code by-- there's going to be some distribution on the left side, which I might write by lambda_d. This is going to be the fraction of left nodes of degree d. All right, I'll simply let that be some distribution. Some might have degree 2, some might have degree 3. Some might have degree 500. And similarly, rho_d is the fraction of right nodes, et cetera.

And there's some average degree here, and some average degree here. So this is the average degree, or the typical degree. This is average left degree, this is average right degree. If I do that, then the calculations are done in the notes. I won't take the time to do them here, but basically you find the rate of the code is 1 minus the average left degree over the average right degree.

OK, so it reduces to the previous case and the regular case. Regular case, this is 1 for one particular degree and 0 for everything else. It works out. If I do that and go through exactly the same analysis with my computation tree, now I simply have a distribution of degrees at each level of the computation tree, and you will not be surprised to hear what I get out as my left to right equations, is I get out some average of this.

In fact, what I get out now is that q left to right is the sum over d of-- this is going to be lambda_d times p times q right to left to the d minus 1. Which again reduces to the previous thing, if only one of these is 1 and the rest are 0. So I just get the-- this is just an expectation. This is the fraction of erasures. I just count the number of times I go through a node of degree d, and for that fraction of time, I'm going to get this relationship, and so I just average over them. That's very quick. Look at the notes for a detailed derivation, but I hope it's intuitively plausible.

And similarly, 1 minus q right to the left is the sum over d of rho_d, 1 minus q left to right to the d minus 1. OK, this is elegantly done if we define generating functions. We do that over here. I've lost it now so I'll do it over here. So what you'll see in the literature is generating functions to find is lambda_x equals sum over d of lambda_d x to the d minus 1. And rho of x equals sum over d, rho_d, x to the d minus 1. And then these equations are simply written as-- this is p times lambda of q right to left, and this is equal to rho of 1 minus q left to right.

OK, so we get nice, elegant generating function representations. But from the point of view of the curves, we're basically just going to average these curves. So we now replace these equations up here by the average equations. This becomes p times lambda of q right to left, and this becomes rho of 1 minus q left to right.

OK, but again, I'm going to reduce all of this 2 curves, which again I can use for a simulation. And now I have lots of degrees of freedom. I could change all these lambdas and all these rhos, and I can explore the space, and that's what's Sae-Young Chung did in his thesis, not so much for this channel. He did do it for this channel, but also for additive white Gaussian noise channels.

And so the idea is you try to make these 2 curves just as close together as you can. Something like that. Or, of course, you can do other tricks. You can have some of these-- you can have some bits over here that go to the outside world. You can suppress some of these bits here. You can play around with the graph. No limit on invention. But you don't really have to do any of that. So it becomes a curve fitting exercise, and you can imagine doing this in your thesis, except you were not born soon enough.

The interesting point here is that this now becomes-- the area becomes p over d_lambda-bar, again, proof in the notes. This becomes 1 minus 1 over d_rho-bar. And so again, the area theorem-- in order for these curves not to cross, we've got to have p over d_lambda-bar plus 1 minus 1 over d_rho-bar, less than the area of the whole exit chart, which is 1. We again find that-- let me put it this way, 1 minus d_lambda-bar over d_rho-bar is less than 1 minus p, which is equivalent to the rate must be less than the capacity of the channel.

So this is a very nice, elegant result. The area theorem says that no matter how you play with these degree distributions in an irregular low-density parity check code, you of course can never get above capacity. But, it certainly suggests that you might be able to play around with these curves such that they get as close as you might like. And the converse of this is that if you can make these arbitrarily close to each other, then you can achieve rates arbitrarily close to capacity. And that, in fact, is true.

So simply by going to irregular low-density parity check codes, we can get as close as we like, arbitrarily close, to the capacity of the binary erasure channel with this kind of iterative decoding. And you can see the kind of trade you're going to have to make. Obviously, you're going to have more iterations as these get very close. What is the decoding process going to look like? It's going to look like very fine grained steps here, lots of iterations, but-- all right. So it's a 100 iterations. So it's 200 irritations. These are not crazy numbers. These are quite feasible numbers. And so if you're willing to do a lot of computation-- which is what you expect, as you get close to capacity, right-- you can get as close to capacity as you like, at least on this channel.

OK, isn't that great? It's an easy channel, I grant you, but everything here is pretty simple. All these sum product updates-- for here, it's just a matter of basically snow point propagating erasures. You just take the known variables. You keep computing as many as you can of them. Basically, every time an edge becomes known, you only have to visit each edge once, actually. The first time it becomes known is the only time you have to visit it. After that, you can just leave it fixed.

All right, so if this has a linear number of edges, as it does, by construction, for either the regular or irregular case, the complexity is now going to be linear, right? We only have to visit each edge once. There are only a number of edges proportional to n. So the complexity of this whole decoding algorithm-- all you do is, you fix as many edges as you can, then you go over here and you try to fix as many more edges as you can. You come back here, try to fix as many more as you can. It will behave exactly as this simulation shows it will behave, and after going back and forth maybe 100 times-- in more reasonable cases, it's only 10 or 20 times, it's a very finite number of times-- you'll be done.

Another qualitative aspect of this that you already see in the regular code case-- in fact, you see it very nicely there-- is that typically, very typically, you have an initial period here where you make a rapid progress because the curves are pretty far apart, then you have some narrow little tunnel that you have to get through, and then the curves widen up again. I've exaggerated it here.

So OK, you're making great progress, you're filling in, lots of edges become known, and then for a while it seems like you're making no progress at all, making very tiny progress on each iteration. But then, you get through this tunnel, and boom! Things go very fast. And for this code, it has a zero-- the regular code has a zero slope at this point, whereas this has a non-zero slope. So these things will go boom, boom, boom, boom, boom as you go in there.

So these guys at Digital Fountain, they called their second class of codes, [UNINTELLIGIBLE], tornado codes, because they had this effect. You have to struggle and struggle, but then when you finally get it, there's a tornado, a blizzard, of known edges, and all of a sudden, all the edges become known.

Oh by the way, this could be done for packets. There's nothing-- you know, this is a repetition for a packet, and this is a bit-wise parity check for a packet. So the same diagram works perfectly well for packet transmission. That's the way they use it. Yeah?

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: Yeah. Right. So this chart makes it very clear. If you're going to get this tornado effect, it's because you have some gap in here. The bigger the gap, the further away you are from capacity, quite quantitatively.

So I just-- this is the first year I've been able to get this far in the course, and I think this is very much worth presenting because-- look at what's happened here. At least for one channel, after 50 years of work in trying to get to Shannon's channel capacity, around 1995 or so, people finally figured out a way of constructing a code and a decoding algorithm that in fact has linear complexity, and can get as close to channel capacity as you like in a very feasible way, at least for this channel. So that's really where we want to end the story in this class, because the whole class has been about getting to channel capacity.

Well, what about other channels? What about channels with errors here? So let's go to the symmetric input binary channel, which I-- symmetric, sorry-- symmetric binary input channel. This is not standardized. The problem is, what you really want to say is the binary symmetric channel, except that term is already taken, so you've got to say something else. I say symmetric binary input channel. You'll see other things in the literature.

This channel has 2 inputs: 0 and 1, and it has as many outputs as you like. It might have an erasure output. And the key thing about the erasure output is that the probability of getting there from either 0 or 1 is the same, call it p again. And so the a posteriori probability, let's write the APPs by each of these. The erasure output is always going to be a state of complete ignorance, you don't know.

So there might be one output like that, and then there will be other outputs here that occur in pairs. And the pairs are always going to have the character that their APP is going to be 1 minus-- I've used p excessively here. Let me take it off of here and use it here-- for a typical other pair, you're going to have 1 minus pp, or p 1 minus p. In other words, just looking at these 2 outputs, it's a binary symmetric channel. The probability of p of cross over and 1 minus p of being correct.

And we may have pairs that are pretty unreliable where p is close to 1/2, and we may have pairs that are extremely reliable. So this 1 minus p prime, p prime, where p prime might be very close to 0. But the point is, the outputs always occur in these pairs. The output space can be partitioned into pairs such that, for each pair, you have a binary symmetric channel, or you might have this singleton, which is an erasure.

And this is, of course, what we have for the binary input additive white Gaussian noise channel. We have 2 inputs, and now we have an output which is the complete real line, which has a distribution like this. But in this case, 0 is the erasure. If we get a 0, then the probability of the APP message is (1/2,1/2). And the pairs are plus or minus y. If we get to see y, then the probability of y given 0, or given one, that's the same pair as the probability of minus y given-- this is, of course, minus 1, plus 1 for my 2 possible transmissions here. Point is, binary input added white Gaussian noise channel is in this class. It has a continuous output rather than a discrete output.

But there's a key symmetry property here. Basically, if you exchange 0 for 1, nothing changes. All right, so the symmetry between 0 and 1. That's why it's called a symmetric channel. That means you can easily prove that for the capacity achieving input distribution is always (1/2,1/2), for any such channel. If you've taken information theory, you've seen this demonstrated.

And this has the important implication that you can use linear codes on any symmetric binary input channel without loss of channel capacity. Linear codes achieve capacity. OK, whereas, of course, if this weren't (1/2,1/2), then linear codes couldn't possibly achieve capacity.

Suppose you have such a channel. What's the sum product updates? The sum product updates become more complicated. They're really not hard for the equality sign. You remember for a repetition node, the sum product update is just the product of basically the APPs coming in or the APPs going out. So all we've got to do is take the product. It'll turn out the messages in this case are always of the form p, 1 minus p-- of course, because they're binary, and so has to be like this-- so we really just need a single parameter p. We multiply all the p's, and that normalize correctly, and that'll be the output.

For the update here, I'm sorry I don't have time to talk about it in class, but there's a clever little procedure which basically says take the Hadamard Transform of p, 1 minus p. The Hadamard Transform in general says, convert this to the pair of a plus b, a minus b. So in this case, we convert it to a plus b is always 1, and a minus b is, in this case, 2p minus 1. Works out better, turns out this is actually a likelihood ratio.

Take the Hadamard Transform, then you can use the same product update rule as you used up here. So do the repetition node updates, which is easy-- so it says just multiply all the inputs component-wise in this vector, and then take the Hadamard Transform again to get your time domain or primal domain, rather than dual domain. So you work in the dual domain, rather than the primal domain. Again, I'm sorry. You got a homework problem on it, after you've done the homework problem, you'll understand this.

And this turns out to involve hyperbolic tangents to do these. These Hadamard Transforms turn out to be taking hyperbolic tangents, and this is called the hyperbolic tangent rule, the tanh rule. So there's a simple way to do updates in general for any of these channels.

Now, you can do the same kind of analysis, but what's different? For the erasure channel, we only had 2 types of messages, known or erased, and all we really had to do is keep track of what's the probability of the erasure type of message, or 1 minus this probability, it doesn't matter. So that's why I said it was one-dimensional.

For the symmetric binary input channel, in general, you can have any APP vector here. This is a single parameter vector. It's parameterized by p, or by the likelihood ratio, or by the log likelihood ratio. There are various ways to parameterize it. But in any case, a single number tells you what the APP message is. And so at this point-- or I guess, better looking at it in the competition tree-- at each point, instead of having a single number, we have a probability distribution on p.

So we get some probability distribution on p, pp of p, that characterizes where you are at this time. Coming off the channel, initially, the probability distribution on p might be equal to y, I think it is, actually, or p to the minus y, and you get some probability distribution on what p is.

By the way, again because of symmetry, you can always assume that the all-zero vector was sent in your code. It doesn't matter which of your code words is sent, since everything is symmetrical. So you can do all your analysis assuming the all-zero code word was sent. This simplifies things a lot, too. p then becomes the probability which-- well, I guess I've got it backwards. Should be 1 minus pp, because p then becomes the probability. If the assumed probability of the input is a 1, in other words, the probability that your current guess would be wrong-- I'm not saying that well.

Anyway, you get some distribution of p. Let me just draw it like that. So here's pp of p. There's probability distribution. And again, we'll draw it going from 1 to 0. So that doesn't go out here. OK, with more effort, you can again see what the effect of the update rule is going to be. For each iteration, you have a certain input distribution on all these lines. Again, under the independence assumption, you get independently-- you get a distribution for the APP parameter p on each of these lines.

That leads-- you can then calculate what the distribution-- or simulate what it is on the output line, just by seeing what's the effect of applying the sum product rule. It's a much more elaborate calculation, but you can do it, or you can do it up to some degree of precision. This you can't do exactly, but you can do it to fourteen bits of precision if you like. And so again, you can work through something that amounts to plotting the progress of the iteration through here, up to any degree of precision you want.

So again, you can determine whether it succeeds or fails, again, for regular or irregular low-density parity check codes. In general, it's better to make it irregular. You could make it as irregular as you like.

And so you can see that this could involve a lot of computer time to optimize everything, but at the end of the day, it's basically a similar kind of hill climbing, curve fitting exercise, where ultimately on any of these binary input symmetric channels, you can get as close as you want to capacity.

In the very first lecture, I showed you what Sae-Young Chung achieved in his thesis. He got the binary input additive white Gaussian noise channel. He got under the assumption of asymptotically long random codes. He got within 0.0045 dB of channel capacity. And then for a more reasonable number, like a block length of 10 to the seventh, he got within 0.040 dB of channel capacity.

Now, that's still a longer code than anybody's going to use. It's a little bit of a stunt, but I think his work convinced everybody that we finally had gotten to channel capacity. OK, the Eta Kappa Nu person is here. Please help her out, and we'll see you Monday.