PROFESSOR: I'm sorry. I have to give this to you. Basically OK, except you didn't understand it the first time.

OK. Good morning. Happy spring. Happy April. We're running a little late. I'll try to finish chapter nine today and start on chapter ten. Ashish may have chapter ten here later. If not, it'll be on the web.

So we've been talking about convolutional codes. Specifically rate 1/n convolutional codes. One input n outputs. As you may imagine, there are also rate k/n convolutional codes. Generalization to k inputs n outputs. These have not been so much used in practice, because in general, we're using these binary codes in the power limited regime, and we want low rates. So 1/n is a good rate. 1/2, 1/3, 1/4, or down to 1/6 are common kinds of rates. So I haven't bothered to develop the more elaborate theory for rate k/n codes.

The rate 1/n codes, just to review. The code is simply defined as the set of all possible output sequences of a convolutional encoder, whose N impulse responses are written as an n-tuple g of d as the inputs range over all Laurent sequences, bi-infinite sequences. And what we showed last time is that without loss of generality or optimality, we could always take this g of d to be of a particular canonical form. Namely, the g of d could be taken to be a set of n polynomials, g_j of d, that were relatively prime.

And for any code, you can find such a canonical encoder by its unique [UNINTELLIGIBLE] two units, and specifies the code, clearly, by this. So that's a nice canonical encoder to take. It furthermore has the property that there's an obvious realization of this encoder in the [UNINTELLIGIBLE] register form with nu memory units. And therefore 2 to the nu states. And we will prove in chapter ten that this is the minimal possible encoder for this code. In other words, there are lots of different encoders that generate this code, but you can't possibly encode it with fewer than the state space of dimension less than nu. That's the way it's going to sound in chapter ten. So we haven't quite got there yet, but it's a minimal encoder, therefore, in that sense.

All right. Today we're going to go now and exploit the finite state structure of this code, to get a very efficient maximum likelihood sequence decoding algorithm called the Viterbi algorithm, which I'm sure you've all heard of. It's become very famous, not only in this field, but really in any place where you're trying to observe a finite state machine in memory-less noise. A finite state hidden markov model, if you like.

And so now it's used, for instance, in the detection of genomes, where the exons end and the introns start and so forth, and the garbage is. And people use it who have no idea that it was originally a digital communications algorithm. But it's a very obvious algorithm. It's come up in many different forms, and Viterbi, in a way, was just lucky to get his name on it. Because it would come up very easily as soon as you pose the problem correctly.

Let's start out with terminated convolutional codes. Which I forget whether I started on this last time. I think I may have. But we'll start from scratch again.

When I say terminated, I mean that we're going to take only a subset of the code words that start at a certain time -- say, time 0 -- continue for some finite time, and then end. And in this set up, with this canonical encoder, it's polynomial, the easy way to specify that is to let u of d be a finite sequence that starts at time 0. That's called a polynomial. and let's restrict its degree -- degree u of d less than k. In other words, it looks like u0 plus u1 d plus so forth up to uk minus 1 d. Therefore I've chosen k, so I really have k information bits or input bits. All right?

So I specify u0 through uk minus 1. I use that as the input in my encoder. What is then my code, my truncated code, let's call it ck -- that's not a very good notation, but we'll use it just for now -- will be the set of all u of d, g of d such that u of d is polynomial and the degree of u of d is less than k.

OK. So we ask what that code is going to be. And let's take some simple examples. Turns out when we terminate codes, we always choose polynomial encoders, so the code will naturally collapse back to the 0 state. How long? Nu time units after the input stops coming in. The outputs will be all 0 at that point and forevermore. The shift register will clear itself out nu time units later, and then they'll be all 0. So the code really starts at time 0 and ends at time k plus nu.

So we aren't going to worry, in this case, whether the code is non-catastrophic or not. Here one of the principal properties that we had was this property guaranteed non-catastrophic. And in fact, is necessary for non-catastrophicity. So we spent a lot of time talking about why that would be a bad thing, to be catastrophic.

OK. But let's take a catastrophic rate 1/1 encoder, which simply we have g of d equals 1 plus d. You remember this encoder as nu equals 1. The input looks like this. uk, uk minus 1. And we simply add these, yk. So we get y of d equals 1 plus d u of d.

OK. Now, if the input to this is a finite sequence, a polynomial, let's see what this code can possibly be. What the code's going to look like. The code's going to start in state 0, at time 0. So here's the time. At the first time, we can either get a 0 in our a 1 in, and accordingly, it will go to state 0 or 1. At the second time, we can go to state 0 with another 0 or just state 1 with a 1. If we get a 1 in and we're in state 1, then the output is going to be 0, and we'll stay in state 1. If we get a 0 in, the output will be a 1, and we'll go back to state 0.

So this is what a typical trellis section looks like here. We have all possible transitions between the two states. And so now it goes on for a while like this. Time invariantly. There are 1, dot dot dot, and then finally, at some time -- say this is time k, or maybe it's time k minus 1 -- time index is a little fuzzy here -- we don't put any more 1s in. We only put 0s on from that time forward. So we could have one more transition back to state 0, and then after that it just stays in state 0 forevermore.

And this isn't very interesting. It was in time 0 all the time before here. It was in state 0 all the time before there and put out 0s. It was in state 0 all the time after here and put out 0s. And this is clearly not conveying any information. Not an interesting part of the code.

So we're going to consider the code just to be defined over these k plus 1 units of time. And so y of d we're going to consider to be a polynomial of degree k. Now we're going to assume that they're k bits in. Then we're going to wait one more time to let the shift register clear out, which is nu. And at that time, we're going to take whatever y is, and then we're going to terminate it.

So if we consider this now, we have a block code whose length is the number of non-zero coefficients of y of d. So we have n equals k plus 1, we have k by design information bits, and what's the minimum non-zero weight of this? It's linear. What's its minimum non-zero weight, or its minimum distance?

AUDIENCE: [INAUDIBLE]

PROFESSOR: 1. Show me a code word of weight 1.

AUDIENCE: [INAUDIBLE]

PROFESSOR: I claim the minimum non-zero weight is 2 by typical argument. I have the all 0 sequence. If I ever leave the all 0 sequence, I accumulate 1 unit of Hamming weight. And I need to ultimately come back to the all 0 sequence, because I've made everything finite here.

So in this case, I can make the argument that I always have to come back to the all 0 sequence. Whenever I merge back into the all 0 sequence, I'll accumulate another unit of weight.

So this is a code with minimum distance 2. Minimum non-zero weight 2. And in fact, it's just the single parity-check code of length n equals k plus 1. OK?

And if you ask yourself, can you generate any -- this is supposedly the even weight code. It contains all even weight k plus 1-tuples. And I think you can convince yourself that you can generate any of these k plus 1 even weight k-tuples.

In fact, the generator matrix for this code. Look. Here's a set of generators. 1 1 is in the code. 0 1 1 is in the code. 0 0 1 1 in the code. So here's a set of generators. 0 1 1. Generator matrix for the code looks like this. We just go like that. And with these k generators, you can generate any even weight code word.

So we get a kind of convolutional form to the generator matrix. A sliding parity-check form to the generator matrix.

OK. So I assert that this is the single parity-check code. So here's a trellis representation for a block code. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: I'm sure I've got the time indices screwed up. But I put in k bits. I wait 1 more unit of time -- nu equals 1 -- for this to clear out. So I get out k plus 1 bits. Think of it in terms of this here. I put in k, the last input you think of as necessarily being 0, and I take the outputs here for k plus 1 times.

OK. So first of all, a couple points here.

AUDIENCE: In general, the length will be k plus nu?

PROFESSOR: In general, length will be k plus nu. So let's write that down. So in general, by terminating a rate 1/n -- nu is called the constraint length -- constraint length nu, or 2 to the nu state code -- convolutional code after k inputs, we get a binary linear block code whose parameters are -- we have k information bits. We have k plus nu non-trivial output times. At each output time, I get n output bits. So it's n times k plus nu is the total effective length of the code. And the distance, we would have to say, is greater than or equal to the distance of the block code is greater than or equal to the distance of the convolutional code.

Why? Because all of the words in this block code are actually sequences in the convolutional code. I'm assuming here that the code is not catastrophic, so it's sufficient to look at all the finite sequences in the convolutional code.

All right. And in general, the block code distance is almost always going to be equal to the convolutional code distance. And this is just for finite code words.

So as another example, suppose we take our standard rate 1/2, nu equals 2, distance equals 5, convolutional code. Our example 1 that we've been using all along. Suppose I terminate with k equals 4, then I'm going to get what? I'm going to get a 2 times 6, I'm going to get a 12, 4, 5 binary linear block code.

That is not so bad, actually. There's a point that I'm not going to make very strongly here. But terminated convolutional codes can be good block codes. And in fact, asymptotically, by choosing the parameters correctly, you can show that you can get a random ensemble of terminated convolutional codes that is as good as the best possible random ensemble of block codes. So terminated convolutional codes is one way of constructing optimal block codes in the asymptotic limit.

These parameters aren't bad. Here's a rate 1/3 code with distance 5. Not too long. You know, you might compare it to what? The BCH code? There's a 15 7 5 BCH code, which is clearly better, because it's higher rate. But if you shorten this code, you have to shorten this by 3, this by 3 to keep the distance. You would get the same code. So it's not optimum, but it's not too bad.

And furthermore, from this construction, you get a trellis representation, which in this case would look like this. It looks like the trellis that we started to draw last time. Sorry. Doesn't look like that. This guy goes down here, this guy comes here, this guy goes here. This continues for a while.

And then finally, when we start to enforce all zeros, we only have one possible output from each of these. And we get a trellis that looks like that. All right? Which is 1, 2, 3 -- I did it correctly for this code, actually. Because each of these is going to have a 2-tuple output on it. And that looked like 0,0 0,0 0,0 0,0 0,0.

Each of the information bits causes a two-way branch, no matter where you are in the trellis. So here's 1, here's 2, here's 3, here's 4. That's the end of them. We wait for nu equal 2 to let the states converge back to 0. So that's a trellis representation of this 12 4 5 block code.

OK. I want to talk about terminated block codes, terminated trellis codes, convolutional codes as block codes, for two reasons. One is to say we can get block codes that way. But the other is to introduce the Viterbi algorithm for terminated convolutional codes.

I think it's easier first to look at how the algorithm works when you have a finite trellis like this, and then we'll go on and say, well, how would this work if we let the trellis become very long, or in principle, infinite? Would the Viterbi algorithm still work? And it becomes obvious that it does.

So this is going to be a maximum likelihood sequence detection or decoding algorithm. We're going to assume that each time here, we get to see two things. Corresponding to the two bits we transmit. So we transmit yk according to which trellis branch we're on -- y0 at this point -- and we receive a 2-tuple r0. If we were on a binary input additive white Gaussian channel, this would go into, actually, the 2-PAM map, two plus or minus alphas, and be received as two real numbers.

And we do the same thing here. y1, we receive r1, and so forth, up to yk plus nu, or k plus nu minus 1, I think it is, rk plus nu minus 1.

OK. So we have a transmit 2-tuple and a received 2-tuple every time. And we're going to assume that the channel is memoryless so that the probabilities of receiving the various possible received values, the likelihoods, are independent from time unit to time unit. That's the only way this works.

Then if I want to get the total probability of r given y, where now I'm talking about over the whole block, this factors into the probabilities of rk given yk, yes? That's what I'm going to depend on.

This is what's called the memoryless condition, memoryless channel. The transition probabilities do not depend on what's been sent or received in previous blocks. Or it's more convenient to use the minus log likelihood equals now the sum of the minus log p of rk given yk. So maximum likelihood is equivalent to minimize the neg-log likelihood.

OK. I receive only one thing in each time unit. And each trellis branch corresponds to transmitting a specific thing. So I can label each trellis branch with the appropriate minus log p of rk given yk. Do you see that?

In other words, for this trellis branch down here, which is associated with y2 equals 0 1, or s of y2 equals alpha minus alpha, suppose I receive r2 equals r1, r2, I would label this by, say, the Euclidean distance between what I transmitted for that branch and what I received.

For instance, on the additive white Gaussian channel, this would simply be rk minus s of yk. The Euclidean distance squared. Or equivalently, minus the inner product, the correlation between rk and s of yk. You know, these are equivalent metrics. Or on a binary symmetric channel, might be the Hamming distance between what I received and what I transmitted, both of which would be binary 2-tuples, in this case.

It's just some measure of distance -- Log likelihood distance, Euclidean distance, Hamming distance -- between the particular thing I would have transmitted if I'd been on this branch and the particular thing that I actually did receive. So having received a whole block of data, I can now put a cost on each of these branches. What's the minus log likelihood cost if I say that the code word goes through that branch? What does it cost me?

All right. Now what I'm basically depending on is note, there's a one-to-one map between code words y and c and trellis paths.

For this specific trellis up here, how many paths are there through the trellis? Do you see that however I go through the trellis, I'm going to meet four two-way branches? All right? According to [UNINTELLIGIBLE], there are four yes-no questions. It's a binary tree. So there are 16 possible ways to get through this trellis from start to finish, if I view it as a maze problem. And they correspond -- I haven't labeled all the branche -- but they do correspond to all 16 words in this block code.

So now what is maximum likely decoding? I want to find the one of those 16 words that has the greatest likelihood, or the least negative log likelihood, over all of these 12 received symbols or 6 received times. So what will that be? Once I've labeled each of these branches with a log likelihood cost, I simply want to find the least cost path through this trellis. And that will correspond to the most likely code word.

This is the essence of it. Do you all see that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: The independent transmission allows me to break it up into a symbol-by-symbol or time-by-time expression. So I can simply cumulate over this sum right here. And I'm just looking for the minimum sum over all the possible ways I could get through this trellis.

So now we translated this into finding the minimum cost path through a graph. Through a graph with a very special structure, nice regular structure.

OK. Once you've got that, then I think it's very obvious how to come up with a nice recursive algorithm for doing that. Here's my recursive algorithm.

I'm going to start computing weights. I'm going to start here. I'm going to assign weight zero here. Right here I'm going to assign a weight which is equal to the cost of the best path to get to that node in the graph. In this case, it's just the cost of that branch.

Similarly down here. Similarly I proceed to these four. There's only one way to get to each of these four, and I simply add up what the total weight is.

Now here's the first point where it gets interesting, where I have two possible ways to get to this node. One way is via these three branches, and that has a certain cost. Another way is via these three branches, and that has a certain cost. Suppose the cost of these three branches is higher than the cost of these three branches. Just pair-wise. All right? Claim that I can pick the minimum cost path to this node, throw away the other possibility, and I can never have thrown away something which itself is part of the minimum cost path through the whole trellis. Clearly.

Suppose this is the best path from here to here. This is worse. X, X, X, X X. And now I find the minimum cost path through the whole trellis goes through this node.

Say it's like that.

Could it have started with this? No. Because I can always replace this with a path that starts with that and get a better path.

So at this point, I can make a decision between all the paths that get to that node and pick just one. The best one that gets to that node. And that's called the survivor. This is really the key concept that Viterbi introduced. We only need to say one, the best up to that time.

We need to do it for all the 2 to the nu possibilities. So we have 2 to the nu survivors at time k or time i, whatever your time index is. Used k for something else.

OK. So I can throw away half the possibilities. Now each survivor, I remember what its past history is, and I remember what its cost is to that point. Now to proceed, the recursion is to proceed one time unit ahead, to add the incremental cost -- say, to make a decision here, I need to add the incremental cost to the two survivors that were here and here. So I add an incremental cost, according to what these branches are labeled with. And now I find the best of these two possibilities to get to this node.

So the recursion is called an ACS operation for add, we add increments to each of these paths, we compare which is the best, and we select the best one. We keep that. We throw away the other. We now have a one unit longer best path to this node and its cost for all 2 the nu survivors at time i plus 1.

And just proceeding in that way, we go through until we finally get to the terminating node. And at this point, when we've found the best path to this node, we've found the best path through the whole trellis.

OK? So that's all it is. I believe it's just totally obvious once you reduce it to a search for a minimum cost path through the trellis. It's very well suited to the application of convolutional codes, because we really are thinking of sending these bits as a stream in time. And at the receiver, you can think of just proceeding forward one unit of time, 2 to the nu, add, compare, selects, and we've got a new set of survivors. There are nice implications of this that involve, you can build a computer for every node or for every pair of nodes to do the ACS operation, make a very fast recursion through here.

So that's it. That's the Viterbi algorithm for terminated convolutional codes.

Now let's ask about the VIterbi algorithm for unterminated convolutional codes. Suppose this trellis becomes very long. Across the whole page. What are these survivors going to look like? Suppose we start from a definite node here, and we've got a four state convolutional code, and we iterate and we iterate and we iterate with the Viterbi algorithm.

After a long time, somewhere out here, we're going to have four survivors and their histories. And I've greatly simplified, but basically the histories are going to look schematically something -- I don't know -- could be anything. Look like this.

The point I'm illustrating here is that the histories will be distinct right at this time. They have to be, because they're going to four distinct stages. But as you go backwards in time, you will find they merge, any two histories will merge, at a certain time back. And this is a probablistic thing. It depends on what's happening on the channel and so forth. But with high probability, they will have merged not too far back in time.

So even if we never get to a final node, even if we just let this process continue forever, at this point we can say, we can make a definite decision at this time. Right? Because all survivors start with a common initial part, up to here.

So one way to operate the Viterbi algorithm would be just to, at this point, say, OK. I'm going to put out everything before this time. No matter how long I run the decoder, the first part of it is always going to be this part. So these are definitely decided up to this time. And then I still don't know about here. I'm going to have to go a little bit further. You proceed further, and after a while, you find more that's definitely done.

In practice, that would lead to a sporadic output rate. That isn't really what you want. So in practice what you do is you establish a decision delay, delta. And what you put out is you hope that 99.999% of the time, that if you look back delta on all the survivors, they will all be a common path, delta back here. So there's a very high probability you will have converged.

And so simply at this time, you put out what you decided on delta time units earlier. Next time you put out the next one. Next time you put out the next one. So you get a nice, regular, synchronous stream of data coming out of here.

Every so often, it may happen that you get out to here and you still haven't made a decision. Then you have to do something. And it really doesn't matter terribly much what you do. You might pick the guy who has the best metric at this time, or you simply might say, well, I'm always going to pick the one that goes to all zeros. That wouldn't be symmetric. You can make any decision you like. And you could be wrong. But you pick delta large enough so that the probability of this happening is very rare, and this just adds a little bit to your error probability, and as long as the probability of making an error because of this kind of operation is much lower than your probability of making an ordinary decoding error, then you're going to be OK.

For convolutional codes way back at the beginning of time, people decided that a decision delay a 5 times nu, 5 times the constraint length, was the right rule of thumb. And that's been the rule of thumb for rate 1/n codes forever after.

The point is, delta should be a lot more than nu. You know, after one constraint length, you certainly won't have converged. After five constraint lengths, you're highly likely to have converged. And theoretically, the probability of not converging goes down exponentially with delta. So big enough is going to work.

And a final point is that sometimes, you really care that what you put out to be a true code word. In that case, if you get to this situation, you have to make a choice, you make a choice here, then you have to actually eliminate all the survivors that are not consistent with that choice. And you can do that simply by putting an infinite metric on this guy here. Then he'll get wiped out as soon as he's compared with anybody else. And that will ensure that whatever you eventually put out, you keep the sequence being a legitimate code sequence. So that's a very fine point.

OK. So there really isn't any serious problem with letting the Viterbi algorithm run indefinitely in time once you've got it started. How do you get it started? Suppose you came online and you simply had a stream of outputs, transmitted from a convolutional code over a channel, and you didn't know what state to start in. How do you use synchronize to a starting state?

Well, this is not hard either. Basically you start, you're in one of four states, or 2 to the nu states. You don't know which one. Let's just give them all cost 0 and start decoding, using the four state trellis. And we just start receiving 2-tuples from here on. We get rk, r k plus 1, and so forth.

So how should we start? Well, we'll just start like that. And we get sort of the mirror image of this -- that after a time, these will all converge to a single path. Or after a time, when we're way down here, this is what the situation will look like. These things will each have a different route over here, but they will have converged in here. There will be a long path over which they're all converged, and then towards the end, they'll be unrooted again.

OK. Well, that's fine. What does that mean in practice? That means we make errors during the synchronization. But we say we're synchronized when we get this case where all the paths going to all the current survivors have a common central stage.

And how long does it take to synchronize? Again, by analysis, it's exactly this same delta again. The probability of not being synchronized after delta goes down exponentially with delta in exactly the same way. It's just a mirror image from one end to the other.

So from a practical point of view, this is no problem. And just starting the Viterbi decoder up, with arbitrary metrics here, and after five constraint lengths, if you like, it's highly likely to have gotten synchronized. You'll make errors for five constraint lengths and after that, you'll be OK, as though you knew the starting state.

So the moral is, no problem. You can just set up the Viterbi algorithm and let it run. The costs will all, of course, increase without bound, which is [UNINTELLIGIBLE]. You can always renormalize them, subtract a common cost from all of them. That won't change anything. Keep them within running time.

So we don't need to terminate convolutional codes in order to run the Viterbi algorithm. We just let it self-synchronize and we make decisions with some decision delay. And the additional problems that we have are very, very small. There are no additional problems. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: At this point? Well, notice that we don't know that we've synchronized until we've continued further, and we've got -- you know, where we've really synchronized is when we see that every survivor path has the common root. So at that point, there is really only one path here. And we can say that the synchronized part is definitely decoded. And we can't really say too much about this out here, because this depends on what's happened out in the past. So you say this is erasures, if you like. The stuff that we're pretty sure has a high probability of being wrong.

AUDIENCE: [INAUDIBLE]

PROFESSOR: There is only one decoded path during this interval.

AUDIENCE: But before that interval there are branches and so on right.

PROFESSOR: Well here, we don't know anything. Here we know the results of one computation. What are you suggesting? Just pick the best one at that point?

AUDIENCE: And finally after you reach the [INAUDIBLE]?

PROFESSOR: And finally? I'm just not sure exactly what the logic is of your algorithm. It's clear for this, it wouldn't make any difference if we just started off arbitrarily so, we're going to start in the zero state. And we only allow things to start in the zero state. Well, we'll eventually get to this path anyway.

So it really doesn't matter how you start. You're going to have garbage for a while, and then you're going to be OK. There's no point in doing anything more sophisticated than that. I don't want to discuss what you're suggesting, because I think there's a flaw in it. Try to figure out what time you're going to make this decision at.

OK. Do we all understand the Viterbi algorithm? Yes? Good. We can easily program it up. There will be an exercise on the homework. But now you can all do the Viterbi algorithm.

All right. So -- and oh. What's the complexity of the Viterbi algorithm? What is the complexity? We always want to be talking about performance versus complexity.

So it's a recursive algorithm. We do exactly the same operations every unit of time. What do the operations consist of? They consist of add, compare, select. How many additions to we have to make? Additions are basically equal to the number of branches in each unit of time in the trellis, which is 2 to the nu plus one. Is that clear? We have 2 to the nu states. Two branches out of, two branches into each state, always for rate one.

So we have 2 to the nu plus 1 additions. We get 2 the nu compares, one for each state. Which is really, you can consider the select to be part of the compare. Overall, you can say the complexity is of the order of 2 to the nu or 2 to the nu plus 1.

This is the number of states or the state complexity. This is the number of branches. I will argue a little bit later that the branch complexity is really more fundamental. You've got to do at least one thing for each branch. So in different set up, the branch complexity.

But these are practically the same thing, and so we say that the complexity of the Viterbi algorithm is basically like the number of states. We have a four state encoder, the complexity's like four, it goes up exponentially with the constraint length.

This says, this is going to be nice, as long as we have short constraint length. For longer constraint lengths, you know, a constraint length 20, it's going to be a pretty horrible algorithm. So we can only use the Viterbi algorithm relatively short constraint length codes, relatively small numbers of states.

The biggest Viterbi algorithm that I'm aware has ever been built is a 2 the 14th state algorithm. The so-called big Viterbi decoder out at JPL. It was used for the Galileo space missions. It's in a rack that big. I'm sure nowadays you could practically get it on a chip, and you could maybe do 2 the 20th states.

But so this exponential complexity, in computer science terms, not really what we want. But when we're talking about moderate complexity decoders, these really have proved to be the most effective, and become the standard moderate complexity decoder. The advantage is that we can do true maximum likelihood decoding on a sequence basis, using soft decisions, using whatever reliability information the channel has, as long as it's memoryless.

So last topic is to talk about performance. How are we going to evaluate the performance of convolutional codes? You remember what we did on block codes? We basically looked at the pairwise error probability between block code words, we then did the union bound, based on the pairwise error probabilities. And we observed that the union bound was typically dominated by the minimum distance error events, and so we get the union bound estimate, which was purely based on the minimum distance possible errors.

And we can do exactly the same thing in the convolutional case. Convolutional case, again, is a linear code. That means it has the group property, has symmetry such that the distances from every possible code sequence to all other code sequences are going to be the same, since we're talking on a long or possibly infinite sequence basis.

And we need to be just a little bit more careful about what an error consists of. We need to talk about error events. And this is a simple concept. Let us again draw a path corresponding to the transmitted code words. A very long path, potentially infinite, but it is some definite sequence. And let's run it through a memory list channel, use the Viterbi algorithm, and we're going to get some received code word, or decoded code word. This is one place where you might want to insist that the Viterbi algorithm actually put out a code word.

What is that going to look like? Well, if you're running normally, the received code word is going to equal the transmitted code word most the time. Except it's going to make errors. And what will the errors look like? They'll look like a branch off through the trellis, and then eventually a reemerging into the same state. And similarly, you go on longer, and you might have another error.

And any place where there's a difference -- should have done a different color here -- any place where there's difference, this is called an error event. So what I'm illustrating is a case when we had two disjoint error events. Is that concept clear? We're going to draw the trellis paths corresponding the transmitted code word and the decoded code word. Wherever they diverge over a period of time, we're going to call it an error event. But eventually they will re-merge again.

So it could be a short time. Can't be any shorter than the constraint length plus 1. That would be the minimum length error event. Could be a longer time. Unbounded, actually.

OK. What is going to be the probability of an error event starting at some time? And when I say an error event starting at time k, let's suppose I've been going along on the transmitted path, and the decoder has still got that there. I'm asking, what is the probability that this code word is actually more likely on a maximum likelihood sequence detection basis than this one? Well, simply the probability that the received sequence is closer to this one than this one.

We know how to analyze that for a finite differences. What is the difference here? The difference, call this y of d and this y hat of d. What is y of d minus y hat of d? We'll call that e of d. This is a code word. This is a decoded code word. So the error event has to be a code word. Right?

Decoder made a mistake. The mistake has to be a code word. So we're asking, what is the probability that y of d sent y hat of d decoded, where y hat of d equals y of d plus e of d? We'll just ask for that particular event. This is the probability that r of d closer to y hat of d than y of d.

Now, making a big leap. This is equal to -- we're just talking about two sequences in Euclidean space. All that matters is the Euclidean distance between them. What is the Euclidean distance between them? It's 4 alpha squared times the weight of this error event, the Hamming weight of this error event.

And so if you remember how to calculate pairwise error probabilities, this is just alpha squared D over sigma squared, where D is the distance of e of d. The weight, the Hamming weight of e of d. Call it sigma squared. Remember something that looked like that?

So once again, we go from the Hamming weight of a possible error event to a Euclidean weight, which is 4 alpha squared times the Hamming weight. We actually take the square root of that and we only need to make an error, a noise of half of that length. So that's where the 4 disappears. And we just get q of d squared over sigma squared, where d is the distance to the decision boundary. Compressing a lot of steps, but it's all something you felt you knew well a few chapters ago.

So the union bound would simply be the sum over all error events in the code such that the start -- so e of d is -- you want them to start at time 0, say. Let's ask for the probably of an error event starting at time 0.

So we want it to be polynomial, have no non-zero negative coefficients, and have the coefficient at time 0 be 1, or not 0. I'm doing this very roughly. Of q to the square root of alpha squared times the weight Hamming of e of d over sigma squared. Just as before.

So to get the union bound, we sum up over the weights of all sequences that start at time 0. Let's do it for our favorite example. Suppose g of d is 1 plus d squared, 1 plus d plus d squared, then what are the possible e of d's that I'm talking about? I have this itself. 1 plus d squared, 1 plus d plus d squared. This is weight equal to 5. What's my next possible error event? Would be 1 plus d times this. So it's going to be a table. We have g of d, 1 plus d times g of d, which is equal to 1 plus d plus d squared plus d cubed, 1 plus d cubed, that has weight equals 6. What's my next longer one? 1 plus d squared times g of d equals 1 plus d fourth, 1 plus d plus d third plus d fourth, that has weight 6. 1 plus d plus d squared times g of d. 1 plus d plus -- this is 1 plus d plus -- plus d third, plus d forth. I may be making a mistake here. 1 plus d squared plus d fourth. It's hard to do this by hand after a while. OK.

Notice, we start tabulating all the error events, which I can do in order of the degree of u of d, always keeping the non-zero, the time 0 term equal to 1. So this is the only one of length 1. This is [UNINTELLIGIBLE] of degree 0. This is the only one of degree 1. There are two of them of degree 2. There will be four of them of degree 3, and so forth.

So I can lay out what all the error events could be, and I look at what their weights are. Here's my minimum weight error event. Then I have two of weight six and so forth. So the union bound, I'd be adding up this term. I'd get one term where the weight is 5, two where the weight is 6, I don't know how many where the weight is 7. I happen to know there were only two weight 6 error events. And you simply have to find out what the weight profile is and put it in the union bound.

Or you can use the union bound estimate, which is simply -- let's just take k nd, the number of error events of minimum weight d, times q to the square root of alpha squared d, where d equals min weight over sigma squared.

OK. In this case, nd equals 1, d equals 6. Let me take this one step further. nd times q to the square root of -- what is sigma squared equals n0 over 2? And Eb equals n times alpha squared. The energy per transmitted bit is alpha squared. We're going to transmit n bits for every information bit.

So plugging those in, I get 2d over n times Eb over N0, which is again of the form nd q to the square root of 2 times the coding gain of the code times Eb over N0. Bottom line is I get precisely the same performance analysis, or I get a nominal coding gain of -- in general, it's kd over n. If it were a rate k over n code, since we're only considering rate 1 over n codes, it's just d over n. And I get an error coefficient which equals the number of weight d code words, starting time 0.

Of course an error event could start at any time. If there are nd starting at time 0, how many start at time 1? Time invariant code. So it's going to be the same numbers could possibly start at time 1. So what I'm computing here is the probability of an error event starting at a particular time.

For this particular code, what do I have? I have one event of weight 5. So the nominal coding gain is 5 over n, which is 2. Which is -- 5 is 7 dB, 2 is 3 dB, so this is 4 dB. That's pretty good. Nominal coding gain of 4 dB with only a four state code. Obviously a very simple decoder for this code.

And what is kd equals 1? That implies -- again, we have the same argument about whatever the error coefficient is. You could plot this curve. The larger the error coefficient is, the more the curve moves up, and therefore over. You get an effective coding gain which is less. But this means since it's 1, you don't have to do that. The effective coding gain is the same as the nominal coding gain. It's still 4 dB. So it's a real 4 dB of coding gain for the simple little two state code.

This code compares very directly with the 8 4 4 Reed-Muller code, block code. This code also has rate 1/2. It has the same rate. We'll see that it also has a four-state trellis diagram.

But it only has distance four, which means its nominal coding gain is only 2, 3 dB. And furthermore, it has 14 minimum weight words, which even dividing by 4 to get the number per bit, there's still a factor of 3, still going to cost us another couple of tenths of a dB. Its effective coding gain is only about 2.6 or 2.7 dB. All right?

So this code has much better performance for about the same complexity as this code, at the same rate. And this tends to be typical. Convolutional codes just beat block codes when you compare them in this way. Notice that we're assuming maximum likelihood decoding. We don't yet have a maximum likelihood decoding algorithm for this code. We'll find for this code, we can also decode it using the Viterbi algorithm with a four state decoder comparable to this one. So it would be, I would say, for maximum likelihood decoding -- about same complexity. But we simply get much better performance with the convolutional. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Say again?

AUDIENCE: What happened to the k?

PROFESSOR: Why is k equal to 1? Which k? Over here?

AUDIENCE: Yeah. kd [INAUDIBLE].

PROFESSOR: All right. We're only considering rate 1/n codes. If it were a rate k/n code, we would get k over n, because this would be n over k times alpha squared. Just the same as before.

In fact, I just want to wave my hand, say, everything goes through as before. As soon as you get the error event concept, you can reduce it to the calculation of pairwise error probabilities, and then the union bound estimate is as before.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Well, I had my little trellis here. And how long in real time does it take for two paths to merge? An error event has got to take at least nu plus 1 time units for two paths to diverge and then merge again in the trellis. Is that clear?

Or put another way, the lowest degree of any possible error event is nu, which means it actually takes place over nu plus 1 time units. OK? From this.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Why is the lowest --? I'm taking g of d -- the definition of nu is what? The maximum degree of g of d. OK? So if that's so, then the shortest length error event is 1 times g of d, which takes nu plus 1 time units to run out. So error events have to be at least this long, and then they can be any integer length longer than that.

You don't look totally happy.

AUDIENCE: [INAUDIBLE]

PROFESSOR: You understand? OK.

AUDIENCE: [INAUDIBLE]

PROFESSOR: How I see it from this diagram? Where have I got a picture of a trellis? Here I've got a picture of a trellis. I've defined an error event by taking a transmitted code word. It's a code sequence. This is supposed to represent some path through the trellis. There's one-to-one correspondence between code sequences and trellis paths. Then I find another code sequence, which is the one I actually decided on. Call that the decoded code sequence. And I say, what's the minimum length of time they could be diverged from one another? All right?

Let's take this particular trellis. What's the minimum length of time any two paths -- say, here's the transmitted path. Suppose I try to find another path that diverges from it? Here's one. Comes back to it. Here's one. Another one. Comes back to it. I say that the minimum length of time it could take is nu plus 1. Why? Because the difference between these two paths is itself a code sequence, is therefore a non-zero polynomial multiple of g of d. OK? Same argument.

OK. So I guess I'm not going to get into chapter 10, so I'll discourse a little bit more about convolutional codes versus block codes. How do you construct convolutional codes, actually? You see that really, what you want to do is to first of all, maximize the minimum distance for certain constraint length. Subject to that, you want to minimize the number of minimum distance words. You want to, in fact, get the best distance profile. Is there any block codes? We had nice, algebraic ways of doing this. Roots of polynomials, Reed-Solomon codes. We could develop an algebraic formula which told us what the minimum distance was.

Do we have any nice algebraic constructions like that convolutional code? No.

Basically, you've just got to search all the possible polynomials where the maximum degree is nu. Take pairs of polynomials. If you want a rate 1/2 code of degree nu, there's not that many things you have to search. All right? You just take all possible pairs of binary polynomials of degree nu or less, making sure that you don't take any two which have a common divisor, an untrivial common divisor, so you can wipe those out. You want to make sure that the constant term is 1. There's no point in sliding one over so it has a larger constant term than 1.

But subject to those provisos, you simply try all pairs g1, g2, and you just do -- as soon as you've assured yourself they're not catastrophic, they don't have a common divisor, then you can just list, you know, the finite code words are going to be the ones that are generated by finite information sequences. So you can just list all the code words, as I've started to do up here. Or since the trellis is, in fact, a way of listing all the code words, there's a one-to-one correspondence between code words and trellis paths, you can just start searching through the trellis and you will quickly find all the minimum weight code words, and thereby establish the minimum distance, the weight profile as far out as you like. And you choose the best. You try all possibilities.

So Joseph Odenwalder did this as soon as people recognized that short convolutional codes could be practical, and he published the tables back in his PhD thesis in '69, so he got a PhD thesis out of this. It wasn't that hard, it's done once and for all, and the results are in the notes. The tables.

And you can see from the tables that in terms of performance versus moderate complexity, things go pretty well. Here's this four state code. It already gets you 4 dB of effective coding gain. To get to 6 dB of effective coding gain, you need to go up to about 64 states.

First person to do this was Jerry Heller at Jet Propulsion Laboratory. Again, about '68, immediately after Viterbi proposed his algorithm in '67, Heller was the first to go out and say, well, let's see how these perform. So he found a good 64 state rate 1/2 code, and he did probability of error versus Eb over N0, and he said, wow. I get a 6 dB coding gain.

And Viterbi had no idea that his algorithm would actually be useful in practice. He was just using it to make a proof. And he didn't even know it was optimum. But he's always given the credit to Heller for realizing it could be practical. And so that 64 state rate 1/2 code with Viterbi algorithm decoding became very popular in the '70s. Heller and Viterbi and Jacobs went off to form a company called Linkabit.

And for the technology of the time, that seemed to be a very appropriate solution. You see you get approximately the same thing as a rate 1/3, rate 1/4. If you go lower in rate you can do marginally better. If you are after tenths of a dB, it's worthwhile.

Later in the decade, is the best way to get more gain to go to more and more complicated convolutional codes? No. The best way is to use the concatenated idea. Once you use these maximum likelihood decoders, the Viterbi decoders, to get your error rate down to 10 to the minus 3 or something, 1 in 1000, at that point you have very few errors, and you can apply then an outer code to clean up the error events that do occur.

You see, you're going to get bursts of errors in your decoded sequence. It's a very natural idea to have an outer code, which is based on gf of 256, say, 8 bit bytes. And so a Reed-Solomon code comes along and cleans up the errors that do occur, and drives the error probability down to 10 to the minus 12, to whatever you like it, with very little redundancy, very little additional cost.

So that became the standard approach for space communications in the '70s and indeed '80s. I've already mentioned that around '90, they went up to this 2 to the fourteenth state Viterbi decoder. They went to much more powerful outer codes, much cleverer. And they were able to get to within about 2 or 3 dB of the Shannon limit. And that was the state-of-the-art on the eve of the discovery of turbo codes, which is where we're going in all of this.

So from a practical point of view, in the moderate complexity regime, simple convolutional codes with moderate complexity Viterbi decoding are the best still that anybody knows how to do. They have all these system advantages. They work with nice synchronous streams of traffic, which is what you want for data transmission. They use soft decisions. They use any kind of reliability information you have. They're not limited by hard decisions. They're not limited by bounded systems as the algebraic schemes are. So it just proved to be a better way to go on channels like the additive white Gaussian noise channel.

OK. So we didn't get into chapter 10. We'll start that next time.