PROFESSOR: OK, well I'm going to start now. As I say, I know that the wireless course has its final at 11:00, so we might not see the people who are taking it. Is anyone here in the wireless course? OK. Q.E.D. Ipso facto ergo sum. OK, just to remind you, I think we have Problem Set 10 solutions today. In any case, they'll be available on the web. The final is next Tuesday. Do bring calculators if you have them. They won't be absolutely essential, but they might be useful. You may bring five sheets of notes. It's a good exercise to make these up. And otherwise, it's closed-book. Ashish will hold a review session tomorrow, 4:00 to 6:00, 26-328, which I'm sure will be excellent as usual.

All right, just to remind you, we're in Chapter 14. Most of the course we talked about coding for power-limited channels, where basically you could get away with binary codes with very little loss of optimality. For bandwidth-limited, additive white Gaussian noise channels, we've got to consider non-binary, multi-level codes, we've really got to go out into Euclidian space and use Euclidian distance as our metric rather than Hamming distance.

So we started out with a simple idea, build a lattice constellation. A lattice is an infinite set of points in end space that has a group property. That's it, a discrete set of points. We take a finite set of them by chopping out a region R, a compact region, which chops out a finite number of points in this lattice or in a translate of the lattice. And we went through this exercise of computing the union-bound estimate, which I remind you is a reasonable estimate for codes that are not too complex. We're not going to use this for capacity-approaching codes. But for codes, where..., basically dominated by minimum distance properties, how many nearest neighbors are there, the likeliest error events are all to the nearest neighbors, we can use the union-bound estimate, which basically just tallies the probability of error to the nearest neighbor events by the union-bound, which itself is inaccurate if you get too many near neighbor events.

And we've got this nice approximate expression that the probability of error per n dimensions, if we're signaling in n dimensions, is the number of nearest neighbors

in the lattice times the Q to the square root of a function times an expression, which involves a coding gain of the lattice, and separably a shaping gain of the region times 3 times SNR norm. So this embodies all kinds of simplifications. It separates the effects of the lattice and the region. Basically, we can optimize each of these separately to optimize this expression.

And also, by the way, by talking about SNR norm, which is SNR normalized by 2 to the spectral efficiency, it makes this whole expression independent of rate or equivalently independent of spectral efficiency. We saw this back when we talked about m by m QAM, we got a universal expression. The probability of error was simply this expression with a 3SNR norm in here - regardless of what m was (regardless of how big the QAM constellation was), we got the same approximate expression. So this is also independent of the size of the constellations in this large constellation, high SNR approximation. Once you get to enough points, everything sort of washes out. So it's very useful.

And I remind you that the coding gain basically depends on the minimum square distance normalized by the density of the lattice, if you like, normalized to two dimensions. The shaping gain is basically the shaping gain of an n-cube divided by-- this is the normalized second moment of an n-cube, this is the normalized second moment of your region, which better be better than that of an n-cube, meaning smaller. And writing it out, it's equal to this, where this is the second moment of your region. And this is the volume, again, normalized to two dimensions.

So the baseline constellations, we showed these were invariant to Cartesian products. In any number of dimensions, n, we can take the integer lattice or any translate of the integer lattice and we can take a region, which is the n-cube -- which will chop out basically a square or cubic constellation, a regular constellation in n dimensions-- and we'll get that the coding gain is the shaping gain is 1. So we'll get just our baseline expression for any integer lattice shaped by an n-cube centered on the origin.

And this is what it looks like. This is back from Chapter Four or whatever. We get

this universal curve, the baseline curve, which holds for m-PAM, m by m QAM, or you can Cartesian product these up into as high a dimension as you like. The constellation we're talking about is really m-PAM Cartesian product to the nth. For any of those constellations, if we normalize the probability of error to two dimensions-- which is what I'm recommending to you, normalize everything per two dimensions in the high SNR regime-- then we basically get the number of near neighbors in Z-squared, the QAM lattice, which is 4 times Q to the square root of 3SNR norm.

And it looks like this. We plot probability of error per two dimensions over SNR norm. I think at 10 to the minus 5, it's 8.4 dB. Does anybody remember? But it's somewhere around 9 dB. And the Shannon limit, as we recall, is for SNR norm, it's at 0 dB. So this is where we can play. Any code will fall somewhere between these two curves, and the name of the game is to get a code the gets as close to the Shannon limit as possible. And at least in this moderate complexity regime, the way we play the game is we try to optimize the shaping region, we try to optimize the lattice, we're going to get a certain gain from shaping, and a certain gain from coding, de-rated by the number of nearest neighbors, which is going to turn out to be an important effect for some well-known lattices because they have huge numbers of near neighbors. And that will give us some curve somewhere in here, and we plot that and we're done. We've written a paper for that particular lattice constellation, for its performance.

Today I'm going to talk quickly about shaping, because that's a very easy story, and obvious, and it won't take long. Then I'm going to talk about lattices, which is really a subject in mathematics, what's known about lattices. I guess the decoding part of that story is not mathematics. That's engineering. And then I'm going to talk about trellis codes, which are the analog of convolutional codes, and which, similarly to the way convolutional codes beat block codes for the bandwidth for the power-limited regime, trellis codes tend to beat lattice codes in terms of performance versus complexity for the bandwidth-limited regime.

So, shaping. Shaping is really an almost trivial story. This really wasn't recognized

as a separable issue until perhaps sometime in the '80s, when trellis-coded modulation was invented. In '82 was Ungerboeck's paper. That was really the breakthrough in bandwidth-limited coding, that trellis codes were the first bandwidth-limited codes that were widely implemented. And after people analyzed them for a while, they realized that you could separate out the shaping effect from the coding effect, which is what I've shown by this expression.

So anyone care to guess what the optimum region is in n dimensions? I guess I've already mentioned it last time. The optimum region, what do we want to do? We want to minimize the normalized second moment-- well, a second moment with appropriate normalization, what would you do? Well, the optimum region for just about anything is an n-sphere, and it certainly is here. So we take Rn equals an n-sphere. If we have a two-dimensional sphere, what are its parameters? We certainly know that the volume of this region, if this is R, it's pi times R squared. And the second moment of this region, how do we do that? We integrate ... go to polar coordinates, 0 to 2pi, d pi, 0 to R, rdr. And then what are we measuring? The second moment is r squared, but we normalize it per dimension, which is 2.

Oh, and we've got to take the uniform distribution over v of R, and that's one over pi R squared. So have I got it now? This is the second moment of a uniform distribution, a probability distribution over the 2-sphere. And so this will be 1 over 2pi R squared times R 4th over 4, right? Is that all correct? And so p of R over v of R is already not normalized to two dimensions, so we don't have to re-normalize it. I didn't get the 2pi in here. I need some help, class. When it comes down to integration, I'm not necessarily the world's best.

So R squared over 4... R squared over 4pi R squared, so we get 1 over 4pi. Is that right? That looks about right because it's fairly close to 1 over 12. So what is the shaping gain of a 2-sphere? And I can just say 2-sphere, because I know it's invariant to scaling. Rotation isn't going to do much to it. It's 1/12 over this, the normalized second moment, 1 over 4pi, which is pi over 3, which is about 2/10 of a dB. OK, so I've improved the second moment by 2/10 of a dB, reduced it to 2/10 of a dB.

This is not very impressive. So of course, a lot of the early work was on shaping two-dimensional constellations. And this basically says if you get a large constellation, you don't even see this effect to, say, 64-QAM. If you have a 64-QAM constellation, 8 by 8-- Somebody at Paradyne noticed that you could improve it slightly by taking the four corner points out and putting them up here, without reducing the minimum distance. OK, so there is an improved, more spherical 64-point constellation. It's improved by about 0.1 dB, as I remember. And what this calculation says is if you go up to 256, 1024, and you simply take all the points in the QAM grid that are surrounded by a sphere of appropriate size to pick out 2 to the n points, for high, large constellations you get a gain of about 0.2 dB. So this is nothing to write home about.

What happens as n goes to infinity? We look up in a math book what the expressions are for the normalized second moment-- actually, you can do it just by this kind of calculation. It's easy enough. And just by scaling, you'll find that the shaping gain of the n-sphere goes to pi times e over 6, which is 1.53 dB. A famous number not only now in shaping, but also in quantization, where you're also interested in minimizing the second moment of a quantization cell. Or it's something like 0.229 bits per two dimensions, is another way of expressing it.

All right, so what is this? This is the maximum shaping gain you could get by the optimum region in the optimum number of dimensions, which is as large as you can go. So this says this is the ultimate shaping gain. You can never get a shaping gain of more than 1 and 1/2 dB. So it says when we play this game, we've got about 9 dB down here that we want to make up at 10 to the minus 5, 10 to the minus 6, we can get only 1 and a 1/2 dB of it by doing good shaping in large dimensions. That's the best we can ever do.

So that's a little bit discouraging. It says shaping is never going to give you that much. On the other hand, you can't get this 1 and a 1/2 dB in any other way. If you have n-cubed constellations, you will always be limited to be 1 and a 1/2 dB away from the Shannon limit. So if you want to approach the Shannon limit, you do have to address the shaping problem. You can't ignore it. 1 and a 1/2 dB is enough to be

noticed in most applications, so you should do some moderate shaping.

Now, of course we can plot the curve. It's done in the notes for any n whatsoever, and you'll see that it goes up fairly steeply at first. It gets to about 1 dB at 16 dimensions, so if you shape in a 16 dimensional sphere, you can get 1 dB out of this maximum of 1 and a 1/2 dBs you can ever get. If you go to 32, it's 1.2 dB or something like that, it begins to slope out. So that gives you a good idea of what you can actually attain.

In the V.34 modem, for instance, a rather simple shaping scheme is used that gets up to about 0.8 dB of shaping gain. It's called shaping on regions. I just mentioned some techniques for shaping. As with a lot of things, once shaping was realized to be a separate subject from coding, there was a flurry of activity on it. People basically proposed a number of different schemes, particularly in the context of the V.34 modem standards development. Of which trellis shaping, I would say, is the most elegant. This is kind of a dual to trellis-coded modulation where we use trellis codes for shaping rather than coding. So that's elegant.

But a cruder and quite effective, just fine technique, shell mapping, was used in the V.34 modem. The basic idea here is that you take a two-dimensional constellation, you just divide it into shells like this. It turns out in two dimensions, if you take a constant delta here, that the average energy goes up, I think, linearly with delta or some way nicely with delta. You take a constellation bigger than you really need, so you have the possibility of some redundancy. So you identify each point in the constellation by which region it lands in. You then go up to 16 dimensions or something-- I think it is 16 dimensions in the V.34 modem-- and you assign each series of points a score depending on which shells its points land in. And you basically have a big table that chooses the lowest energy combinations from all the possible series of eight shells, which can be done fairly elegantly using generating function techniques.

I realize this is just hand-waving and you can't get it, but the effect is that you use this shell with very high probability. You sort of get a Gaussian-like probability on the

shells. Let me just draw that like this. You don't use the points out in this outermost shell very much. You use this a lot, and they sort of have an overall Gaussian distribution.

And I should mention that this is an equivalent way of getting shaping gain. If we're shaping uniformly over an n-sphere in a high number n of dimensions and we basically have constellations which consist of two-dimensional QAM points, what is the projection of a uniform distribution over an n-sphere down onto two dimensions? If we look at each two-dimensional component of these n-dimensional vectors, what distribution will it have if we impose a uniform spherical distribution on the high dimensional constellation? It will turn out that it becomes Gaussian in two dimensions. Think of the projection of the 2-sphere down on one dimension. The projection of a 2-sphere on one dimension is something like this. It's not Gaussian, but there's more probability that one of the coordinates will be near 0 than it will be out at one of these maxima here. So this is just the same thing raised up to n dimensions.

So an equivalent way of looking at this problem is we want somehow to get a Gaussian-like distribution on a one- or two-dimensional constellation, and this is one way of doing it. And you can get the same pi times e over 6 factor by just looking at it from an entropy point of view, the entropy of a two-dimensional Gaussian distribution relative to the entropy of a uniform distribution over a square. That will give you the same factor if you do everything right.

OK so basically, shaping is easy. These techniques are not difficult to implement. You can get one dB of shaping gain quite easily through a variety of techniques. It's worth getting that one dB, you're not going to get it by any other means. But having done that, that's all you're ever going to get, and so end of story on shaping. So as I said, it was a flurry of activity in the '90s, and no one has looked at shaping since then.

So that's part of the story. Let's continue now with lattices, an old mathematical subject. So what we're interested in is, what's the densest lattice packing in n

dimensions? This appeared in mathematics, long before it appeared in engineering, as Hermite's parameter. And Hermite said this was the right way to measure the density of a lattice, normalized in all the appropriate ways. So any version of a lattice is equivalent. A version is something that you get by scaling or rotating or even by Cartesian products, which I'm not sure how widespread that idea is.

So what's the densest lattice in two dimensions again? We want to find the densest lattice, in terms of coding gain or Hermite's parameter in n dimensions. Anyone care to guess what the densest lattice in two dimensions is? It's the hexagonal lattice. So in two dimensions, hexagonal, which reasons of mathematical history is denoted A_2. And its coding gain, I believe we computed it last time, and it was 0.6 dB.

So what's the most we could ever do by fooling around with two-dimensional constellations? Suppose I want the best possible 256 point QAM constellation? Well, according to these large constellation approximations, the best I could do would be to take the 256 lowest energy points in some translate of the hexagonal lattice. So I might fool around with the translation vector a little bit to try to find the absolute lowest energy point, but it really wouldn't matter very much regardless of what I did. I could expect 0.6 dB of coding gain and 0.2 dB of shaping gain, or 0.8 dB better than the 8 by 8 - sorry the 16 by 16 QAM constellation with the same number of points and the same minimum distance. This basically tells you what your options are. 2D, there's not much to do.

You remember way back in Chapter One, I gave you this problem about the 16 point, 2D constellation. That is precisely 16 points from a certain translate of the hexagonal lattice that fall within a circle that's just big enough to enclose 16 points. And it has, I forget, something like a 0.7 dB gain over the 4 by 4 square constellation. We computed that in the very first homework. So again, staying to two dimensions, things aren't very exciting. Turns out, it's most interesting to go in powers of two. In four dimensions, the densest lattice is something called D4, Schlafly's lattice, 1850-something. And its coding gain is precisely the square root of 2, which is 1.5 dB.

8 dimensions, it's E8, Gosset's lattice, about 1900. The D4 and E8 come, again, from the mathematical literature and have something to do with their relationship to certain groups. Lattices and finite groups are very closely connected. And this has a coding gain of 2 or 3 dB. 16 dimensions, it's something just called lambda 16. It's a Barnes-Wall lattice, as all of these are, actually. And it's 2 to the 3/2, or 4.5 dB. You see a certain pattern here?

In 32 dimensions, we have two lattices that I'll mention. There's the 32-dimensional Barnes-Wall lattice, which has a coding gain of 4.6 dB. Or there's actually Quebbeman in the '60s, I think, came up with a denser lattice. So this is still a subject of active investigation, particularly in dimensions like 19, where it's not so easy to find the densest lattice. And this, I forget, is maybe 6.2 dB. It's definitely denser. But it's something like that.

This basically goes to infinity. You can get denser and denser lattices just by following this chain here which is a chain of Barnes-Wall lattices. The nominal coding gain can be taken to infinity, just as we saw with kd over n. That goes to infinity for rate 1/2 Reed-Muller codes. But of course, Shannon says we can't ever get more than 9 dB of effective coding gain here. Or actually, we can't get more than 7 and a 1/2 dB if we take into account that 1 and a 1/2 is going to be given by shaping gain.

So what must be happening here? We must be having a huge number of near neighbors. The number of nearest neighbors in the hexagonal lattice is six. In D4 it's 24, in E8 it's 240, in lambda 16 I think it's 4320-- the correct numbers are in the notes. By the time we get up here, we get something like a 146,680 or something like that. So these lattices are designed to have a very high degree of symmetry. That means if you stand on any point, you look around you, they've pushed the neighboring points as far away from you as they can. But now there are a great many of them, so you get a uniform sky just filled with points around you, a number of dimensions.

So that's what the problem is and that means that the effective coding gain-- I again

do the calculation. I know here we lose at least two dB. This is something like 3.8 dB. OK, so the effective coding gain actually sags and begins to top out as you get out here. Maybe nowadays you could think of implementing a larger lattice, but I think this is the largest anyone has ever thought about implementing. And what do we get? A coding gain of less than 4 dB -- an effective coding gain. So if you use that, you'd be, say, somewhere around 5 db here. We go over 6 dB, but then we go up by this normalized per two dimensions, so we divide by 16. That's still 10,000, which is still 13 factors of 2, which is 2.6 dB. You can see where the problem is.

So anyway, we get something that's really this moved way up a lot more, and as a result we get a much steeper curve. This method doesn't work up here, but this might be lambda 16 or lambda 32. And then we can get another 1 and a 1/2 dB of shaping gain. So maybe 4 dB, optimistically. OK, not really. So that's a little disappointing. Just by looking up in the math textbooks, we don't do very striking things.

I mention in particular the Barnes-Wall lattices, and if I had more time I would develop them because they're very much analogous to the Reed-Muller codes. They are constructed by a u, u plus v construction in the same way as Reed-Muller codes. They are length-doubling constructions that also double the distance, just as the Reed-Muller codes do now Euclidian distance.

But let me just give you the first couple of sentences in a discussion of the Barnes-Wall lattices, which were only discovered in '59, about the same time as the Reed-Muller codes. Suppose we take the QAM grid like this. An important observation is that if we take every other point in sort-of checkerboard fashion, what do we get? Imagine it's going to infinity in both directions. So this is a subset of half the points, basically. Half the density, twice the volume of points that, itself is a rotated and scaled version of the integer lattice. Take the centers on a checkerboard and they form a 45 degree rotated and square root of 2 scaled version of the integer lattice. If this had minimum squared distance 1 between points, then this has minimum squared distance 2 between points. So we could label the points red, black, red, black, red, black. If we did that, the red points would be a co-set of this sub-lattice

and the black points would be a co-set-- meaning a translate-- of this sub-lattice.

So let me write that as follows. If we take Z-squared, it has a sub-lattice which we call RZ squared, r for rotation. Rotation by the Hadamard matrix, 1, 1, 1 minus 1. If I write d min squared, this has d min squared at 1, this has twice the d min squared. So I partition my lattice, or any translate of this lattice, into two co-sets of the sub-lattice with twice the minimum squared distance. This is the first steps in what Ungerboeck called set partitioning. So it's an obvious way to partition any QAM-type signal set, red and black points.

Having done that, I can do it again. Alright, do it again, what do I get? So now I'm going to take every other one of these points, which will leave me, say, these 4 points. Let me call these A, A, A, A. These points might be B, B, B, B, the other ones that I cut out. I could call these C, C, C, C and D, D, D, D. Now I've partitioned each of these subsets into two subsets. I've partitioned one into A, B and the other into C, D, such that - what does A look like? A is another translate of a lattice. This happens to be Z-squared scaled by 2. It's not rotated, it's just Z-squared scaled by 2. So we get down to 2 times Z-squared. And what's its minimum squared distance?

AUDIENCE: 4.

PROFESSOR: 4. So I've doubled the distance again. This is an elementary but important observation. I can continually go through these, partitioning into two subsets. Every time I do, I double the distance. I get another version of the integer lattice at a higher scale and, perhaps, rotated. So if I did it again, I would get eight subsets with minimum squared distance 8 between points in each subset. So it's just a repeated divide-by-2 scheme.

All right, now to get the Barnes-Wall lattices, you remember how we got the Reed-Muller codes. We took, say, two codes of length 2 and we put them together with the u, u plus v construction and we got codes of length 4 that had the minimum square distance of the higher distance code, if we chose them properly. That's the same thing we do here. If we put together these two lattices, if we choose them-- well, just use the u, u plus v construction where u is from Z-squared and v is from

RZ-squared. So in words, that's what you do. If you're interested in looking it up, look it up in the notes.

What you get is a lattice, D4, that has normalized density halfway between that of Z-squared and RZ-squared, it turns out-- sort of analogous to the code-- and its minimum square distance is 2. Or if you put these two together, you get something called RD4, whose minimum square distance is 4. So you get something whose normalized volume is halfway between that of these two normalized volumes, but it has the same minimum distance as this guy, and that's where you get the coding gain from. This is denser but still has the same minimum distance.

If we do this again, go out to eight dimensions, we get, it turns out, E8. And this has minimum square distance 4. But it has the same normalized volume normalized back to two dimensions as this guy, which only had minimum squared distance 2. So that's precisely why this has a nominal coding gain of 2. We've doubled the minimum square distance without changing its density, normalized back to two dimensions.

All right, so you get all the Barnes-Wall lattices just by doing this. This turns out to be a sub-lattice of this guy. Not surprisingly, just as these lattices are nested, just as the Reed-Muller codes are nested, so you get a similar tableau. You get a similar construction that you can take out to 16 dimensions, 32 dimensions, however many dimensions you want. And every time you do, you increase the coding gain by a factor of square root of 2, so the coding gain eventually gets driven to infinity, and that's the story on Barnes-Wall lattices. So similarly to Reed-Muller codes, these are lattices that are the densest known in certain small dimensions like 4, 8, and 16. We don't know any denser lattices.

You start to get up to 32, 64 dimensions, then we do know of denser lattices. But they are not nearly as nicely structured, and maybe these are still the best from a complexity point of view. You can play similar games on building trellis representations of these lattices, as we did for Reed-Muller codes. They have similar kinds of trellises because they're based on the same construction.

So that's the story of Barnes-Wall lattices. The most famous lattice is the Leech lattice, which is a super-dense lattice in 24 dimensions. It has a coding gain of 6 dB. And unfortunately, it has 195,560 near neighbors or something like that. So you can see, it turns out it's a very close cousin of this 32-dimensional Barnes-Wall lattice. It's related to the 24-12-8 Golay code in the same way as the Barnes-Wall lattices are to the Reed-Muller codes. So the fact that the Golay code is so exceptional can be seen just as a projection of the fact that the Leech lattice is so exceptional. This somehow has the Golay code buried in it, if you can find it in here.

And the Leech lattice is particularly notable not just because it has remarkably high coding gain for its dimension-- it's remarkably dense for its dimension, exceptional-- but also because it has remarkable symmetries, so it's connected to some of the largest exceptional groups, the monster exceptional groups. And it was a key step in the classification of all finite groups to find the symmetry group of this lattice. Just a little liberal arts for you. So that's what we can get with lattices. By going out to the Leech lattice or the Barnes-Wall lattice-- this could equally well be the Leech lattice-- again, effective coding gain is only about 4 dB or maybe even less because of the exceptional number of near neighbors here. To one digit of significance, that's approximately its coding gain.

So how can we go beyond this? As I said already, the next great breakthrough was trellis codes for which Gottfried Ungerboeck became famous. Trellis codes, as I say in the notes, are basically-- first of all, Ungerboeck thought about the problem from a Euclidian distance point of view. So what you had to do to break out of Hamming distance thinking. He had the idea that in the bandwidth-limited regime, what you want to do is expand the constellation rather than expand the bandwidth. So to send four bits per second per Hertz. You could do that using uncoded 16-QAM constellation. Or Ungerboeck saw, well, let's make it into a 32-point constellation. Now we have some redundancy, now we can do some coding on this constellation.

He used the set partitioning idea, which I've just explained to you. Initially, he just looked at a QAM constellation and he said, let's divide it up into A, B, A, B, C, D, C, D, A, B, A, B, C, D, C, D. So let's partition it from A, B, C, D into A and D as I've done

it here, and B and C are these two subsets. And we'll break this up into A and D and B and C. And then some theoretical people came along and said, well, this is just Z-squared, this is RZ-squared, and this is 2Z-squared.

Let's, first of all, partition our enlarged constellation. Now here's the setup we're going to use for coding. We are going to bring in our ...If we're trying to send-- I forget what notation I used in the notes-- b bits per two dimensions. Let's take one bit per two dimensions and put it into a rate 1/2 convolutional code. So we're going to get out two bits. Everything here is going to be per two dimensions because we're going to use a two-dimensional signal set. And we'll bring in our other b minus 1 bits down here. These are called uncoded bits.

For these two bits, we're going to go into a subset selector. Two bits are going to select among-- this going to be something from A, B, C, or D. It's going to tell us which subset to use. All right, then we'll use that in combination with these b minus 1 bits to select a point from a 2 to the b plus 1 bit QAM constellation. Two dimensions. So this is actually our signal point y, our two-dimensional signal point.

Do you see how the numbers work out? We want to send b bits per QAM point, per two dimensions. We're going to use a 2 to the b plus 1 point constellation, so 32 instead of 16, say. Take one bit in here, generate two streams, one redundant. This will tell us A, B, C, D, so that's two of the bits you need to select one of these points. Say, in a 16-point constellation, we'll provide a bit label that looks like this. For A, it will be 0, 0. b is 0, 1, c is 1, 0, D is 1, 1. That's always the low-order bits in any selection for A, B, C, D. And then we'll take two high-order bits, which then tell us which of these four groups it's in. So we'll have another 0, 0, 0, 1, 1, 0, 1, 1.

Now I'm talking about b equals 3. We have two bits that come in here and select one of these blocks, and then two bits that come out of here and select A, B, C, D within the block. There's two parts to the label. One of them determines A, B, C, D, the other one just determines some larger group.

These bits are called uncoded because there's no protection. If you sent point A over here and received point A over here, that would look like a perfectly legitimate

code word. If we make an error of that far, which is d min squared equals 4, say. We make an error of size d min equal 2 all the way over here, then there's no further protection from this code because this code only determines the sequence of As, Bs, Cs, and Ds.

This is called a within-subset error. There's no protection on within subset errors. What the code does is it protects against smaller errors that are between subset errors like A to B, which has squared distance of 1 or A to D has a squared distance of two, and so forth. Are you with me? Nobody's asking questions. Maybe I haven't permitted it. Do you follow what the overall setup is?

I've done everything now, except choose an exact constellation. For the exact constellation, I'll just choose one of the best QAM signal sets. If it's large enough, I'll shape it like a circle. Or I might even try to put Gaussian shaping on it by some completely extraneous thing that works only on these clumps. So shaping takes place on the higher order bits. You can see that, given a large enough constellation just by choosing these clumps, I can make the constellation sort of have the shape of a circle. The subsets of A, B, C, and D points have to all be the same size in order for this scheme to work. But I can do that within a sort of circular constellation if the constellation is large enough. Or I can put some Gaussian distribution on these clumps.

So I do shaping down here. Initially, of course, Ungerboeck didn't worry about shaping. But if I try to do shaping, it's down on this index, which is considered to be the higher order bits that determine the growth structure of the constellation. These are the lower order bits, these two bits, and they determine the fine structure which is determined by this code. So the last thing I have to select is the code, and it turns out that the same code we used in the example is a good code to use here.

Suppose we use the four-state rate 1/2 example code that I've used before that is everybody's favorite example because it has such excellent distance properties. So it looks like this. So that's my rate 1/2 encoder. You remember as a binary code, it had minimum Hamming distance 5. And the only minimum distance code word was

the impulse response. Let me write the impulse response as this way: 11, 01, 11. This is what we get out at time 0, time 1, and time 2.

Let me assign the following labels to these subsets. We'll make that 00, this one 11, this one 01, and that one 10. In other words, this is the complement of that. This is the complement of that. What is the nearest neighbor point in Euclidian space for this sequence? If I complement the first two bits coming out, then I'm going to go from one of these subsets to another, but within one of these RZ-squared subsets, which has minimum squared distance 2. So let's remind ourselves that the minimum squared distance within A, B, C, D is 1, within A, D or B, C is 2, within 2Z-squared is 4.

So I'm claiming-- there's actually a problem on this on Problem Set 10 if you tried to do it-- that any error of this type, 11, is going to cause me a Euclidian error of minimum squared distance 2, at least. It can only cause me to go from A to D, which is always at least squared distance 2, or from B to C, or vice versa.

So in terms of Euclidian distance, d min squared, I'm always going to get a squared distance of 2 at this point. And actually, any place you diverged in the trellis, because of the structure of the code, this is going to be true. I'm always going to get a squared distance of 2 when two trellis paths come together in the trellis. I always have a binary difference of 11, so by the same argument I'm always going to get a Euclidian squared distance of 2 when two trellis paths come together. And furthermore, I'm going to have at least one difference somewhere in between, which is going to, again, in terms of Euclidian distance, lead to a Euclidian distance of at least 1.

So the conclusion is that between any two sequences of constellation points that are labeled by two distinct code words here, I'm going to have a cumulative Euclidian squared distance of at least 5, by the same argument as I had for the binary code. But now it turns out that I'm still in a regime where binary distance, Hamming distance translates directly to Euclidian distance. This is what you saw.

Basically, what we're using here is this is Z-squared, you can think of as the image

of the (2, 2) binary code. RZ-squared is the image of the (2, 1) binary code, and this 2Z-squared is the image of the (2, 0) binary code. You get the same partition structure for these codes and their co-sets. This is the set of all four two-tuples. We divided it into two subsets, each of minimum distance 2 here, and here we divide it into four subsets, each of minimum distance infinity. I'm not going to take the time to make this precise, but think of how that relation goes.

So I've assured myself of a Euclidian squared distance at least 5 between signal point sequences that correspond to two different code words. That's excellent. Is the minimum squared distance of the code then equal to 5, as it was in the binary case of this trellis code? No, it's not. And the problem is I could make this kind of error, which is completely invisible to the code. I can make a within subset error, and this only has minimum square distance of 4.

So the conclusion of this analysis is that this is what's called a four-state two-dimensional trellis code. This four-state two-dimensional trellis code has a minimum squared distance of 4, and it has a number of nearest neighbors per two dimension of 4. That's the number of nearest neighbors within 2Z-squared, which is one of these subsets. So the conclusion is that already, this has a legitimate 3 dB coding gain. Its coefficient is still just the QAM coefficient, and this very simple code has this performance curve moved over. So it's 3 dB all the way up the line. So that's pretty good. This is a trivial code that could be decoding-- by the way, it was, again, done by the Viterbi algorithm of a four-state trellis. You just take the sequences and it's exactly the same. You measure distance in terms of minimum squared distance.

The only difference is when you decode, you first find what the closest point is within the A, B, C and D subsets. That is what you're going to take as your representative for that subset. So you first make basically an uncoded decision, which is the closest point in A? Which is the closest point in B, which is the closest point in C, which is the closest point in D? You're going to get a metric for each of those. How far is each of those from the receive point? So then in trellis decoding, what the trellis is going to look like is something like this. A, D, A, D, B, C, and so forth. The trellis is going to look exactly the same, but here you get A, D, D, A, B, C, C, B.

So what metric are we going to use for each of these branches? First, we find the closest point within each subset, and then we use the metric for that point. Clearly, if we're going to choose A for this time interval, then we should use the closest point in A as our receive decision. We can get that survivor without any memory whatsoever when we make that decision, and then we take the sequences of representatives and find the best of those representatives as our best trellis-coded sequence. Any sequence of points that has labels that correspond to some path through this trellis is a valid point and vice versa. If and only if the sequence of labels is according to this trellis is it a legitimate trellis-coded point. So it's a nice amalgamation.

Great. With a simple four-state code, basically we've doubled the size the constellation. The constellation expansion is 2, the constellation expansion factor with twice as large a constellation, we've been able to achieve a coding gain of 3 dB. Not bad. So how do we extend this idea? First, extension would be to-- all right, let's go up to an 8 state, 16 state, whatever code. But is that going to improve our minimum distance using this block diagram? It's not, because we're still stuck with this within subset distance. So we can never get past d min squared equals 4 with a four-way partition. What we have to do is make one more level of partition and we need to make an eight way partition, at which point we're going to get a within subset distance of 8. I left out one step, I'm sorry. I'll come back to it.

We can get a code minimum squared distance up to 8, if we extend the minimum squared distance up to 8. The thing I didn't use, I didn't tell you before, was in computing the coding gain of this code-- let's just call it c-- what should it be? It should be the minimum squared distance of the code over the volume of the code per two dimensions. In this case, n is 2 so I'm just going to do it over the volume of the code. Now what is the volume of a convolutional code? This is actually an interesting problem, because the Voronoi region of a trellis code is actually an infinite dimensional object. A trellis code is really an infinite dimensional lattice. It's a lattice in infinite dimensional space. And if you look at the Voronoi region, it extends out into all these dimensions. It might not for such a simple code, but in principle it

does.

However, you can find other fundamental regions. The way to think about this is instead of putting the volume here-- well, what have we cost ourselves in volume? We've doubled the size of the constellation. And in two dimensions, that means we've doubled the volume. So we've quadrupled the minimum distance at the cost of doubling the volume. We've covered twice as much space now because we have twice as many points. And so, that's where we get the-- this is equal to, for this code, that's 4. We have a factor of 2 loss in volume, and so that's where we get what I said was a coding gain of 2 or 3 dB.

Just to get you to the bottom line, we should replace the volume by two times-- I call it eta of the code-- where eta of the code is its redundancy per two dimensions. In this case, we introduce one redundant bit per two dimensions, and that's why the constellation has to expand by a factor of 2. You see, we get the right answer. The redundancy has increased by one bit, so this has to be 2 to the b plus 1, so the constellation has to be twice as large, so the volume is twice as large. I'll just hand-wave and tell you that's the factor you want down here representing volume -- 2 to the redundancy. And you can prove this by finding a region which you can show is a fundamental region of the trellis code and has this volume. But I won't go into that.

OK, so that's the calculation which I should've mentioned before, I skipped right over it. Back to what I was saying. To improve trellis codes, the first thing you can do is to, say, go to an eight-way partition. So now we're going to have eight subsets. So we're going to need three bits here. So let's use two bits here and use a rate 2/3 trellis code. We're going to have three bits coming out here to select subsets, and we're going to have b minus 2 bits down here to select a point still a 2b plus 1 point QAM constellation. So our redundancy per two dimensions is still one bit per two dimensions. We're still going to lose a factor of 2, but now we can go up to d min squared equals 8 and we can go up to a coding gain of 6 dB.

So in his original paper, Ungerboeck did this. He showed codes going up to 256 states in two dimensions that gets you up to a nominal 6 dB coding gain. And

because they have low coefficients here, they get very close to 6 dB of effective coding gain. Already in Ungerboeck, we have codes-- I mean, eventually you can get back to just the within-subset coefficient again, which is always 4. You can get up to, say, 6 dB with maybe 512 states of effective coding gain, which is considerably better than the densest lattice we would think of using, and also quite reasonable to implement. Now we're going to need a 256, 512-state Viterbi algorithm decoder.

This was a stunning breakthrough. Immediately in 1993, everybody started building turbo codes. Back in 1982, everybody started building trellis codes. It happened to be the time of the V.32 modem, and everybody said boy, this is just what we need to get an additional 3 or 4 dB coding gain in the V.32 modem. What was actually done there is this was simply an eight-state convolutional code, so it was rate 2/3. And there was tweaking of this. It was made rotationally invariant, made nonlinear. And a lot of activity, which was very nice, stimulated by the standards effort. But that was basically the state of the art in the '80s.

The one other improvement that was made was the following idea. We really don't like doubling the size of the constellation, particularly for a large one, mostly for implementation reasons. There were some feelings against nonlinearities a larger constellation would cost you. You wanted to minimize the constellation expansion ratio. So how were we ever going to do that? You can't do it this way in two dimensions. The way to do that is to go up to four dimensions.

So let's do subset partitioning in four dimensions. So now think of four dimensional constellations. In practice, these are just built from the Cartesian product of two two-dimensional constellations. We start with Z4. The next one down is D4, which has a minimum square distance of 2. It's sometimes called a checkerboard constellation in four dimensions. It's just the red-black idea again. Then below that, we have a four-way partition into RZ4, which still has minimum distance 2. And one more level of partitioning here into something into RD4, which has minimum squared distance of 4.

So this is an eight-way partition of a four-dimensional constellation. This idea was introduced by Lee-Fang Wei. Again, this is an interesting column over here. You can think of this as being the (4, 4, 1) code. This is the Euclidian image of the (4, 3, 2) code. It kind of expanded in the lattice. We know we can't do any better than (4, 2, 2). Down here, this is the (4, 1, 4) code. So again, this is the same partition as we get for binary codes with the distances-- again, for binary codes and this level of partitioning, Hamming distance translates to Euclidian distance. It's really just the image of these codes as you go up to the lattices.

Now in this case, what's the best we can ever do? Coding gain. We still can't go up to a squared distance of greater than 4 because our within-subset distance is going to be within RD4. So this is still limited to 4. But now, what's our redundancy per two dimensions? Here, say, we're going to use still a rate 2/3 convolutional code. We'll now have an eight-subset selector in 4D, so everything is per 4D here. So this is going to be 2b minus two bits and this is going to be, it turns out, a constellation that is twice as large in four dimensions. We still have one redundant bit per four dimensions, so we need twice the size of the constellation in four dimensions. But that's only 1/2 a bit of redundancy per two dimensions. So this denominator becomes 2 to the 1/2. Waving hands furiously here.

And so this could go up to 2 to the 3/2, which is 4.5 dB. So there's actually a particularly nice eight-state code here that Wei came up with which, really, in terms of performance versus complexity is the only code that anyone has ever come up with that is distinctly better than any of Ungerboeck's original codes. All of Ungerboeck's original codes were in one dimension, two dimensions, or PSK codes. You could also do this on phase shift key constellations.

So with the Wei code, it gives you a very simple way to get 4 1/2 dB. Actually, you have to go up to 16 states to get 4 1/2 dB. The Wei code is only 4.2 dB. And this is basically what was adopted for standards in the early '90s. The V.34 modem standard, also the ADSL standard has the same code in it. If you have DSL at home, that's what it's using in it. Again, with a very simple code, the eight-state Viterbi algorithm, you can get 4.2 dB of coding gain. The error coefficient in this

case, this has 24 nearest neighbors in D4, or 12 per two dimensions. So you lose 0.15 dB or something. I'm sorry, 0.3 dB. Anyway, this is a very nice code.

There are 32 and 64-state codes in V.34. They don't gain you very much. They each double the complexity and only gain you about 0.2 dB for each one. All these things tend to saturate. You can't get more than about 5 to 6 dB of coding gain with a reasonable size trellis code. So where are we now? We can get 5 or 6 dB of coding gain with, say, a 256-state trellis code. We can get another 1 to 1 and a 1/2 dB of shaping gain. So all together, we're up to 6 to 7.5 dB. From here, we're up somewhere into the 2 to 3 dB range. In other words, combining trellis code with, say, 256 states with some shaping scheme with maybe 1 dB of shaping gain, this gets you maybe 5 dB or 5.5 dB. We get 6 dB total effective gain, and we're still 3 dB or maybe 2 or 3 dB from Shannon limit.

So, close but not quite there yet. Five minutes to go. So in the notes, there's one page on higher performance schemes. There are higher performance schemes. There are turbo codes for the bandwidth-limited regime that get you to within 1 dB of the Shannon limit. Sequential decoding gets you to within about 1 and a 1/2 dB of the Shannon limit. The idea of multi-level codes -- you can regard each of these two-way partitions as being specified by one bit. So with three bits of labels, you can specify an eight-way partition.

Another idea is to code on each of these levels separately. Use a binary code on each of these levels separately. Each of these corresponds to an effective channel that has different characteristics. You've basically got to take the Gaussian distribution and alias it around in accord with whatever this partition actually is. But you can come up with an effective binary channel at each of these levels. It turns out to be a binary symmetric input channel, always in these cases. So you could think of taking a binary-- now, each one of these channels is going to have its own capacity. It turns out that information theoretically, the capacity of multi-level coding on each of these levels independently is the same as the aggregate capacity of coding it on the whole channel. It's a lovely separability result.

So in principle, if you could code at capacity on each of these binary channels, then you could get to the capacity of the aggregate channel, with a large enough constellation. So that's a good idea. We now know a lot of binary capacity-approaching code, so let's use a turbo code or a low-density parity-check code at each level, an appropriate code. It turns out, as you might expect, that the first level is the really hard level. In this case, you might have a rate 1/3 or 1/2 code and really work hard with your code. At the next level, the effective error probability is much better. Making hard decisions, you might have a probability of 1/10 or lower of making an error.

So these are really much easier problems. It might be a better idea at these levels just to use algebraic coding or something that works very well. The capacity is nearly one per bit at each level, so you want to code with very little redundancy at these next levels, but it's still capable of approaching capacity. Well, it turns out this is a good scenario for Reed-Solomon codes or some kind of algebraic code. So the moral is you only really need your capacity-approaching code at maybe the first or first and second levels. Down here, just use a very high rate-- 1,023, 1,003 Reed-Solomon code over gf of 2 to the 10th-- very little redundancy, and drive the error rate down as low as you like at each of these levels. So multi-level coding is a good way in principle.

I'm not sure if it's been done in practice. In fact, I don't know of any capacity-approaching code that has been done in practice, except there's a very crude technique, I would say, called Bit-Interleaved Coded Modulation, BICM, which is the following simple idea. Let's just take a single low-density parity-check code or some capacity-approaching code. It's going to have to be designed for a BICM channel. Let's totally interleave the bits and then let's use them as the bits that control each of these levels here. At the output of the channel, we know which level the bits came out of. So at the output of the channel, we know what the appropriate a posteriori probabilities are for each bit. They're going to be wildly different. A bit sent through here is going to be sent through a very highly reliable channel, a bit sent through here is going to be sent through a very low reliability channel.

But with all this interleaving and the independence approximation in the analysis, we're going to be able to design a code for this mixture channel, a mixture of high reliability and low reliability binary channels, that's going to get to the capacity of this mixture channel, which as I've already told you is equal to the capacity of the aggregate channel. I guess there's some elegant features of it. But that's, in practice, what people use. They use it more because they've already developed a chip for binary channels, and they want to reuse that chip for some large constellation channel, some bandwidth-limited channel. So this was originally advocated by Qualcomm as a, quote, pragmatic approach.

OK, we've got a chip that works for binary channels, so let's just interleave outputs and use it over this multi-level channel and it'll probably work well enough. And, in fact, that's true. And the information support for it, again, is the fact that you don't lose anything by coding bit by bit from a capacity point of view. I think that's the only one that's been used in practice. And on wireless cellular channels for instance, that tends to be what's used.

And of course, I know that perhaps a couple of you have an exam at 11:00, so I'll make a point of letting you out of here. I've really enjoyed teaching the course this term. It was the first time I was able to get to the analysis of capacity-approaching codes, so I enjoyed doing that. At this point, I think the subject of coding for the additive white Gaussian noise channel, the traditional channel, is pretty well wrapped up. I don't think we're going to see any great improvements.

There's still some very interesting theoretical questions about the convergence of these iterative decoding algorithms. What do they converge to? How fast do they converge? There are interesting connections to statistical physics and inference and so forth that people are still exploring. They're interested in graph theoretical properties. There's attempts to algebraically construct these capacity-approaching codes, so we can say a lot more about their parameters and their performance, particularly for lower probabilities like 10 to the minus 15.

So you go to the information theory symposium, you'll still hear a lot of papers about

coding effectively for additive white Gaussian noise or memoryless binary symmetric input channels. But people ask me, is coding dead now? Are Fourier transforms dead? No, they're not dead. But they've become sort of part of the standard core of what you should know. I think this is a subject that anyone in communication should certainly learn, but is it an area where people are going to make further advances that are important from a practical point of view? Obviously not very much, in view of the Shannon limit. We are very close to the Shannon limit for these channels, and have been now for a few years. So it's important to know about these things. It's important to know when you're in the wireless course, for instance. You basically calculate mutual information and assume that you can go at the mutual information. This is how you do it.

So you should know how it's done, and that's why I think this course continues to be worth teaching. But from a point of view of research, it's become a rather mature area, and the action has gone off to other kinds of channels. Multi this, multi that, so forth. Anything to do with networks or multi-user is where the action in coding is. Data compression, of course, is a subject that we haven't talked about at all, which is also coding. So anyway, I think it's good for you to have all this under your belts. I hope that you find it useful in your future lives, and I wish you all well on the final exam. Thank you.