PROFESSOR: I have our final exam schedule from the registrar. It comes Tuesday morning of exam week. It's on the web page. You won't miss it.

I want to remind you that the midterm exam is on Wednesday, March 16, that it starts at 9 not at 9:30 so that you're less time-limited than you would be otherwise. It's a two-hour exam. And it will cover basically up through chapter eight, which is Reed-Solomon codes. And you're responsible for anything that's been discussed in class. If we haven't discussed it in class, then don't worry about it. All right. Any questions about any of those things?

It's 9 o'clock to 11 o'clock. What's the underlined? This line up here? This says, "This class goes from 9:30 to 11.00." Not 11:15. That's for me, not for you, although I've tried to lay off some of the responsibility on you.

OK, let's continue. I hope to finish up chapter six today, might not completely finish it. There are really three topics left to go. One is the orthogonality and inner product topic, which I skipped. The second is Reed-Muller codes, which is our main objective in this chapter, a family of useful codes. And the last topic is why making hard decisions is not a good idea. And so I'll try to say as much as I can about those three things.

Just to remind you of where we are, we're in the power-limited regime. We're trying to design good, small-signal constellations, or moderate-sized signal constellations now, with nominal spectral efficiency, less than two bits per two dimensions. The technique we're using is we're going to start from a binary linear block code in Hamming space. And we're going to take the Euclidean image of that and hope it's a good constellation in Euclidean space.

And as we were just talking about before class, the fact that a linear code is a subspace of F2 to the n means it's true if and only if, really, it has the group property, which in Euclidean space leads to a geometrical uniformity property. We haven't proved that in its full scope. But we have noticed that from every code word, every code word has the same distance profile to all other code words. And in particular, the minimum distance is the minimum weight of any non-zero code word. That's the main juice we've squeezed out of that at this point.

So we've now been talking a little bit about the algebra of binary linear block codes. We've characterized them basically by three parameters, n, k, d, where n is the length of the code, k is the dimension of the code, d is the minimum Hamming distance of the code. In the literature, this is what you'll mainly find, in the n, k, d code. We have a subsidiary parameter, the number of words of minimum distance d, which we're going to need to get the error coefficient, or a union-bound expression. This has less prominence in the literature.

Given just these numbers, we get a couple of key parameters of our constellation. One is its nominal spectral efficiency, which is 2k over n bits per two dimension, upper bounded by 2. Of course, often in the coding literature, you talk about in a more natural quantity, which is k over n, called the code rate.

Maybe I shouldn't call it cap-R, because R is used for the code rate in bits per second. This means bit per symbol. So just let me call this rate.

But when you're reading the coding literature, the rate of the code means how many information bits per how many transmitted bits. And for n, k, d binary linear block code, it's k over m. And the nominal spectral efficiency is just twice that, since we measure it per two dimensions.

OK, and even more importantly, we get the union bound estimate in terms of a couple of simple parameters. It's just an error coefficient, Kb, the number of nearest neighbors per bit times this Q function expression, which always has 2Eb over N_0 in it, multiplied by this multiplicative factor, which we call the coding gain, which is just kd over n. And this is equal to one for a 1,1,1 code. Now you know coding.

So anything else? Our basic effort is to get a larger coding gain by constructing more complicated codes. This parameter here, which we need in order to actually plot the curve and estimate the effective coding gain -- the effective coding gain is derated from the nominal coding gain by a rule of thumb, which depends on Kb. It's every factor of 2 in Kb costs you 0.2 dB, roughly in the right range, if it's not too big, all those qualifiers. But it's a good engineering rule of thumb. And that's just the number of nearest neighbors per code word divided by k.

And so our object here is to see how well we can do. And the Reed-Muller codes will be an infinite family of codes that give us a pretty good idea of what can be achieved for a variety of n, k, d that kind of cover the waterfront. That's why I talk about them first. They're all so very simple.

OK, but first, we forgot to talk about orthogonality and inner products and duality, both from a geometric point of view and from an algebraic point of view. And so I want to go back and recover that. The definition of an inner product between x and y, where x and y are both binary n-tuples, is, as you'd expect, a sort of dot product expression, a component-wise product, the sum over k of xk yk, where all the arithmetic here is in the binary field, F2. And well, this clearly has the bilinearity properties that you expect of an inner product that's linear in x for a fixed y or vice versa. But it doesn't turn out to have the geometric properties that you expect.

We can define orthogonality, this is another definition, in the usual way. x and y are said to be orthogonal -- do that better. x is said to be orthogonal to y if and only if their inner product is 0, which is the same definition you know from real and complex vector spaces. OK.

But the overall moral I want you to get from this is while this inner product behaves absolutely as you expect in an algebraic sense, it behaves very different from what you expect in a geometric sense. So the algebra's fine. The geometry is screwy. All right. That's the catch word to keep in mind.

Why is that? Where does this inner product live? It's F2 valued, right? I'm just doing this sum in binary space, and the result is either 0 or 1. So when are two n-tuples orthogonal? Simply if they have an even number of places in which they're both equal to 1. Is there a question?

In particular, suppose I try to define a norm in the usual way, like that. Is that going to have the properties that I'd want of a norm? No. Why? Because one of the basic properties we want of a norm is strict positivity, that the norm of x is equal to 0 if and only if x is equal to 0. That's clearly not true here.

What's the requirement for the inner product of x with itself to be equal to 0, in other words x to be orthogonal with itself? It just simply has to have an even number of 1's. If it has an even number of 1's, then this so-called norm is 0. If there's an odd number, it's 1. So it's perfectly possible for a vector to be orthogonal to itself, a non-zero vector to be orthogonal to itself. And that's basically where all the trouble comes from in a geometric sense.

AUDIENCE: So which of them would be [INAUDIBLE] here?

PROFESSOR: It's mod-2. It's in F2. All the arithmetic rules are from F2 which is mod-2 rules, correct. So, good.

This means at the most fundamental level, we don't have a Hilbert space here. We don't have a projection theorem. Projection theorem is the basic tool we use in Euclidean spaces, more generally, Hilbert spaces. Just say that every vector can be partitioned. In a given space and its orthogonal space, we can express a vector as the sum of its projection onto the space and the projection onto the orthogonal space, which are two orthogonal vectors.

So it's an orthogonal decomposition. So we have nothing like the projection theorem here. Therefore we have nothing like, we don't necessarily have orthonormal or even orthogonal basis for subspaces.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: You do have a unique orthogonal complement. I'll get to that in a second. But for instance, we might have that a subspace can be orthogonal to itself. That's what the problem is.

For instance, 0, 0, and 1, 1 is a nice, little, one-dimensional subspace. And what's it's orthogonal subspace? It's itself.

We may not have an orthogonal basis for a subspace. And for an example of that, I'll give you our favorite 3, 2, 2 code, as I'll now call it. It consists of these four code words.

A set of generators for this code consists of any two of the non-zero code words. You can generate all of the code words as binary linear combinations of any two of these. But no two of these are orthogonal.

So there's clearly no orthogonal basis for that code. We shouldn't expect to find orthogonal basis, orthogonal decomposition, so all of these sorts of tools that we relied on heavily in 6.450 in Euclidean spaces.

OK, so this is just a great caution to the student. Don't expect Hamming space to have the same geometric properties as Euclidean space. Yes?

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Nothing special. It's just finite fields have a different geometry. In F2, there's really only one geometry you would impose, which is the Hamming geometry. In other finite fields, there actually could be more than one geometry. But let's say the algebraic properties, however, you can still count on.

For instance, n, k, d code C has, as we've already shown, a basis. It has k dimensions. It has a basis g1 up to gk of k, linearly independent, though not necessarily orthogonal basis vectors. So we can always write the code as the set of all u g such that u is a k-tuple of information bits, let's say.

In other words, this is a compressed form for the set of all binary linear combinations of these generators, where I've written what's called a generator matrix, g as a k by n matrix, whose rows are these generators. And I've multiplied on the left with a row vector u, if I'm doing it in matrix terms. You can do this more abstractly just as a linear transformation.

And side comment, notice that in coding theory, it's conventional to write vectors as row vectors, whereas in every other subject you take, it's conventional to write vectors as column vectors. Why is this? Is there some deep, philosophical reason? No. It's just the way people started to do it in coding theory back at the beginning, and then everybody has followed them.

I may have even had something to do with this myself. And I'll tell you the reason I like to write vectors as row vectors is I don't have to write a little t up next to them. That's the deep philosophical reason. It's like driving on the right side or the left side. Obviously, you could have chosen how to do it one way or another back in the beginning. But once you've chosen it, you better stick with it.

There's actually some insight involved here. This is sort of associated with a block diagram, where we take k bits, and we run it through this linear transformation. And as a result, we get out, what shall I write, x n bits. And in block diagrams, we tend to take the input bits from the left and proceed to the right, left to right kind of thing.

So this formula reflects this left to right picture, whereas I'd say the deeper reason why you usually see G u in system theory is that G is regarded as an operator that operates on u. It's kind of a G of u. And so that's why it's more natural to think of this as being a column vector, because then, we have a different picture. G is the operator that somehow transforms u.

But these are all side comments, obviously. They don't really matter for anything. All right, just remember that vectors are row vectors.

OK. Let's define the dual code in the natural way, as the orthogonal code. Say C dual. The definition is that C dual is the set of all n-tuples y, such that x, the inner product between x and y is 0. In other words, y is orthogonal to x for all the x and C. It's the set of all n-tuples that are orthogonal to all the words in C under our F2 definition of inner product orthogonality.

OK, so that's a natural definition. For example, as I've already shown you, if C is 0, 0, 1, 1, then C dual is 0, 0, 1, 1. If C were 0, 0, 0, 1, then C dual would be 0, 0, what? 1, 0. Thank you. Because 1, 1 is not orthogonal to this, 0, 1 and is not orthogonal to this. So we simply go through and pick it out.

Now as I said, the algebraic properties of the dual code are OK. I emphasize they're algebraic. If C is an n, k code, in other words, has dimension k, what do you expect the parameters of C dual to be? Its length is what? It's n, of course. And what's its dimension going to be? If C has dimension k, what do you guess the dimension of C dual is going to be? Just guess from Euclidean spaces, or any -- it's n minus k. The dual space has dimension n minus k. And that holds.

In the notes, I give two proofs for this. There's the conventional coding theory textbook proof, which involves writing down a generator matrix for C k generators, reducing it to a canonical form, called the systematic form, where there's some k by k identity matrix and a n minus k by k parity check part. Then, by inspection, you can write down a generator matrix for C dual. And you find it has dimension n minus k. It's kind of a klutzy proof, in my opinion.

A second, more elegant proof, but one that you don't have the background for yet, is to use simply the fundamental theorem of homomorphisms. This is in some sense an image. This is the dual of an image, which is a kernel. You work out the dimensions from that. I am still in search of an elegant, elementary proof. And anyone who can come up with a proof suitable for chapter six gets a gold star. Believe me, the gold star will be very valuable to you.

OK, so exercise for any student so inclined, give me a nice proof of this. It's surprisingly hard. And I can't say I've spent great quantities of my life on it, but I've spent some time on it.

So anyway, the dimensions come out as I hope you would expect, based on your past experience. And I won't give you a proof in class. You get the fundamental duality relationship, that the dual of C dual, what would you expects that to be? C. OK.

In words, C is the set of all n-uples that are orthogonal to all the n-tuples in C dual. OK. So this actually means that I can specify C. If I know C dual, I know C. Give me

C dual, the set of all code words orthogonal to it tells me what C is.

So this implies that I can write C in the following form. C is the set of all, just emulating this, y n F2 to the n such that x, y equals zero for all x in C dual. I probably should have interchanged x and y for this. x such that x, y equals 0 for all -- this is symmetrical. The inner product of x and y is equal to the inner product of y and x. So I don't care how I write it.

OK, so I can actually specify a code by a set of parity checks. Now suppose I have a generator matrix, call it H, namely a set of generators, H1 through Hn minus k, for C dual. I'm going to have a set of n minus k generators. Then I hope it's obvious that I can test whether x is orthogonal to all of C dual by just checking whether it's orthogonal to all of these generators.

So I would now have C is the set of all x n-tuples x such that x, H, j equals zero, all j. OK. I've just got to test orthogonality to each of these generators. In other words, in each of these is what we call a parity check. We take the inner product of x with a certain n-tuple, and we ask whether parity checks. In other words, we ask whether the subset of positions in which H, j is equal to 1, in those positions, whether x has an even number of 1's. Get very concrete about it.

So writing this out in matrix form, the test is C is the set of x in F2 to the n such that x h-transpose equals 0. That's just a matrix form of what I've written up there. So let me picture it like this. Here the test is, I take x, which is n bits. I put it into what's called a parity checker or syndrome reformer. And I ask whether this is 0.

We call this, in general, the syndrome. And we ask whether it's 0. If we get a 0, we say x is in the code. If it's not 0, then x is not in the code. That's the test.

So when we summarize, we really have two dual ways of characterizing a code, which you will see in the literature. We have a generator matrix, we might give a k by n generator matrix for the code. And then the code is specified as C is the set of all U g such that U n F2 to the k.

In other words, this is an image representation. We take C as the image of a linear transformation. We call G a linear transformation. It goes from Fk to F2 to the k to F2 to the n. And then the code is simply the image of this transformation algebraically.

OK. Or we can specify it by means of a parity check matrix, H, which is the generator matrix of the dual code. So this would be the parity check matrix for the dual code G. h is the generator matrix of the dual code. And we ask whether -- now we specify it as I have up here, simply the x and F2 to the n such that x H_t equals 0.

And this is what's called a kernel representation, because it's the kernel of a linear transformation defined by H2 from F2 to the end, down to, this is a m by n minus k matrix. So the syndrome is actually an n-minus-k-tuple. The elements of the syndrome are the n minus k individual bits, parity check bits. OK.

And sometimes it's more convenient to characterize the code in one way, and sometimes it's more convenient to characterize it in the other way. For instance, we were talking last time about the n, n minus 1, 2 single parity check code, or the even weight code, the set of all even weight n-tuples, which has dimension n minus 1. And in general, for high-rate codes especially, it may be simpler to give the parity check representation. What is the dual code? Let's call this C. C dual is what?

C dual is what we call the n, 1, n repetition code. In other words, it has dimension one. It has two code words, the all-0 word and the all-1 word.

Clearly, the all-1 word is orthogonal to all of the even weight words. And vice versa, a word is even weight if and only if it's orthogonal to all 1's. So in this case, what is the generator matrix? We had a generator matrix last time consisting of n minus 1 weight 2 code words all arranged in a kind of double diagonal pattern. That's OK. But that's an n minus 1 by n matrix.

Most people would say it's easier to say, OK, what's the parity check matrix? The parity check matrix in this case, H, is simply a one by n matrix consisting of all one's. And what's the characterization of the code? The code consists of all the words that are orthogonal to this.

And that is why we call it single parity check code. There's one parity check. And if you pass the parity check, you're in the code. And if you don't, you're not.

OK, so we see that these, first of all, here's an example of dual codes. Are their dimensions correct? They are. Is each on characterized correctly as the dual of the other? It is.

So we've passed that. This is intended to make point C. So it's a good example.

One final thing is suppose I have a code with generator matrix G and another code with generator matrix H. Are they each other's dual codes are not? And the answer is pretty obvious from all of this. Yes, they are.

Let me a substitute in here, x is supposed to be equal to U g. So another requirement is that UGH_t equals zero for all u. And without belaboring the point, the 5, 2 codes with generator matrix G and H, they are dual codes if and only if they have the right dimension, one is n, k, and the other is n, n minus k, and we satisfy G H_t equals zero.

Basically, this is the matrix of inner products of the generators of C with the generators of C dual. And if we have k generators that are all orthogonal to these n minus k generators, then they must be the generators of dual codes. That's is kind of intuitive and natural. All right, so again, this is a concise form that you would actually, probably most commonly find in the literature. It's not hard to get to.

OK, so in this course, we're probably going to talk quite a bit about orthogonality. Duality is very powerful, but we're not going to be using it very much in this course, I believe. I'll mention duality whenever there's a duality property to mention. But in general, I'm not going to spend an awful lot of time on it.

But it's important you know about it, particularly if you were going to go on and do anything in this subject. And at an elementary level, it's nice to know that we have two possible representations, and one is often going to be simpler than the other.

Use the simple one. In general, use the representation for the low-rate code to determine the high-rate code.

OK. Any questions on this? I've now finished up the preparatory. Yeah?

AUDIENCE: So you're saying that every basis for [INAUDIBLE]?

PROFESSOR: That's necessary and sufficient, right. OK, Reed-Muller codes. Why do I talk about Reed-Muller codes? First of all, they give us an infinite family of codes, so we can see what happens as n gets large, as k goes from 0 to n, whose parameters are sort of representative.

They aren't necessarily the best codes that we know of. They're very simple, as I will show, to construct and to characterize their parameters, so we can do all the proofs in half an hour here. And they're not so bad. In terms of the parameters n, k, d, the Reed-Muller codes, up to length 32, are the best ones we know of, at least for their parameters.

There is no 32, 16 code that's better than a 32, 16, eight Reed-Muller code. There's no 32k, eight code that has k greater than 16, all those sorts of things. Actually, I'm not 100% sure of that.

So for short block lengths, they're as good as BCH codes, or any of the codes that were discovered subsequently. For even longer lengths, up to 64, 128, 256, they are going to be slightly sub-optimal in terms of n, k, d, as we'll see in some cases. But they still are very good codes to look at in terms of performance versus complexity.

The decoding algorithm that I'm going to talk about eventually for them is a trellis-based decoding algorithm, and a maximum likelihood decoding algorithm within. They can be maximum likelihood decoded very simply by these trellis-based algorithms. And that's not true of more elaborate classes of codes. So from a performance versus complexity point of view, they're still good codes to look at for block lengths up to 100, 200.

As we get up to higher block lengths, then we'll be talking about much more random-like, non-algebraic codes in the final section of the course. This is the way you actually get to pass it. You don't worry about n, k, d. Right now, we're talking about moderate complexity, moderate performance.

OK, so they were invented in 1954 independently by Irving Reed and, I think it was, David Muller, D. Muller, in two separate papers, which shows they weren't too hard to find. As you'll see, they're very easy to construct. They're basically based on a length-doubling construction.

So we start off with codes of length or length 2. And then from that, we build up codes of length 4, 8, 16, 32, and so forth. So in general, their lengths are equal to a power of 2, and m is the parameter that denotes the power of 2. So we only get certain block lengths, which are equal to powers of 2.

They have a second parameter, r, whose significance is that d is 2 to the m minus r for 0, less than or equal to r, less than or equal to m. Or some people, including me, are going to put the lower limit at minus 1, just to be able to include one more code in this family at each length. But this is just a matter of taste. You'll see this is a very special case, the minus 1.

So let's guess what some of these codes are going to be. So we have two parameters, m, which can be any integer 0 or higher, so the lengths will be 1,2,4, so forth, and r, which goes basically from 0 to m, which means the distances will go from 2 to the m down to 1. And what are some of the basic codes that we're always going to find?

We always write Rm of little rm. I'm not sure I completely approve of how the notation goes for these codes, but that's the way it is. So it's this notation. That means the Reed-Muller code with the parameters m and r is written Rm of r, m.

All right. So Rm of m, m, this is going to be a code of length 2 to the m and distance 1. What do you suppose that's going to be? This is always going to be the 2 to the m 1. Sorry, the distance is 1. So it's going to be the 2 to the m, 1 universe code, in other words, simply the set of all 2 to the m-tuples, which has Hamming distance 1.

OK. RM of 0, m, what's that going to be? This is a code now that has length 2 to the m and minimum distance 2 to the m. We know what that has to be. It has to be the 2 to the m, 1, 2 to the m repetition code, a very low-rate code, dimension one. OK.

And then if we like, we can go one step further. There is a code below this code. This is the highest-rate code you can get. This, however, is not the lowest-rate code you can get. What's the lowest rate code of length 2 to the m? Well, it's one that has dimension zero and minimum distance infinity.

And I don't think I ever defined this convention. But for the trivial code that consists of simply the all-0 word, what's it's minimum distance? Undefined, or infinity, if you like. So this is the trivial code.

So if we want, we can include the trivial code in this family just by defining it like this. And it works for some things. It doesn't work for all things. It doesn't work for d, for instance. This definition holds only for r between 0 and m. It doesn't hold for r equals minus 1, because there the distance is infinite.

OK, so let's start out and get even more explicit. We want to start with m equals 0. In that case, we have only Rm of 0, 0, and Rm of minus 1, 0. And this is going to be the one by either of these. The universe code is equal to the repetition code. It's the 1, 1, 1 code. And this is the 1, 0, infinity code.

And that's really the only two codes that you can think of that have length 1, right? This is the one that consists of 0 and 1, and this is the one that consists of 0. I don't think there are any other binary linear block codes of length 1. So that's a start, not very interesting. Let's go up to length 2.

So m is going to be equal to 1. Here, Rm of 1, 1 is going to be length 2. And it's going to be the universe code, so it's only going to have distance 1. Rm of 0, 1 is going to be length 2, but it's going to be the repetition code. And Rm of minus 1, 1 is going to be 2, 0, infinity.

OK, so there are really the only three sensible codes of length 2. This is the only one of dimension two. This is the only one of dimension zero. There are other ones of dimension one, but they don't have minimum distance 2, so they're not very good for coding purposes. So this kind of lists all the good coding codes of length 2.

All right. So now let me introduce the length-doubling construction. Let me make the point, first of all, that all these codes are nested. What does that mean? That means that his code is a sub-code of this code, which is a sub-code of this code.

And that's going to be, in general, a property of Reed-Muller codes. We're going to get a family. And each lower one is going to be a sub-code of the next-higher one, which is going to be easy to prove recursively.

All right. This is the key thing to know about Reed-Muller codes, how do you construct them. Once you understand the construction, then you can derive all the properties. It's called the u, u plus v construction for obvious reasons. Apparently, Plotkin was the first person to show this. I think Reed and Muller had two different constructions, and neither one of them was the u, u plus v construction. Reed, in particular, talked about Boolean functions.

All right. But this is the way I recommend you to think about constructing them. And it's simply defined as follows. We assume we've already constructed the Reed-Muller codes of length m minus 1. It's a recursive parameter, m minus 1. Now we're going to construct all the Reed-Muller codes of parameter m of length 2 to the m.

How are we going to do it? We want to construct Rm of r, m. And we'll say it's the set of all code words which consist of two halves, a pair of n-tuples of half the length. so the two halves are going to be u and u plus v, where I choose u and u plus v as follows. u is in Rm of r minus 1, m.

And so what does that mean? Sorry, it's got to be m minus 1. So it's got to be half the length. Is that right? r m minus 1, v is in Rm of r minus 1, m minus 1. Somebody who has the notes, like my valuable teaching assistant, might check whether I got it right or not.

Let's see. What are going to be the parameters of these codes? They both have n equals 2 to the m minus 1. The distance here is 2 to the m minus 1 minus r minus 1, so the distance here is 2 to the m minus r, which is the distance that we want to achieve for this code. And the distance here is 2 to the m minus r minus 1, which is half the distance we want to achieve for this code.

So we reach down, if we wanted to construct now a code of length four and distance two, we would construct it from these two codes, the one of half the length with distance 2, and the one of half the length with half the distance, distance 1. So we would somehow use this to get a code of length 4 and distance 2. And we don't know yet what k is.

OK. So we're going to use these two codes to construct a larger code. Is the construction clear? All right.

So let's now derive some properties from this construction, and from the fact that we've started from a set of codes. Let's say we start from length two codes. Or we could start from length one. You could satisfy yourself that this construction applied to the length one codes gives the length two codes. I won't go through that exercise.

So we have some set of codes, m minus 1. What's the first thing we notice? Obviously, the length is what we want, because we've put together two 2-to-the-m minus 1-tuples, and 2 times 2 to the m minus 1 is 2 to the m. So we've constructed a set of 2-to-the-m-tuples. The length of the resulting code is 2 to the m. This is Rm r, m, constructed in this way.

Second, it's linear. It's a linear code. All we have to check is the group property. If we add two code words of this form, we're going to get another code word of that form, from the fact that these guys are linear, yes.

OK. So it is a linear code, that's important. Then we might ask, what's its dimension? How many code words are there?

Well, do I get a unique code word for every combination of u and v. And however you want to convince yourself, you obviously do. If I'm given this word here, I can deduce from it what was u, that's simply the first half of it. Subtract u from the second half of it, and I find out what v is. So there's a one-to-one map between all possible pairs, u, v, and all possible new code words.

And so what that means is, let me call it the dimension of the code with parameters r, m is simply the sum of the dimensions of the codes with parameters r, m minus 1 and r minus 1, m minus 1. So for instance, in this hypothesized code here, if I want to know the dimension, well, I take all possible combinations of words here and words here. How many are there? There are eight. It has dimension three. So I'm going to get a 4, 3, 2, code, which it's not too hard to see is the single parity check code of length 4. All right.

So that's how I do it. I simply add up the dimensions of the two contributing codes. Not a very nice formula. In the homework, you do a combinatoric exercise that gives you a somewhat more closed form of the formula. But I think this is actually the most useful one. In any case, we eventually get a table that shows what all these things are anyway.

Now, just as a fine point, I want to assert that if I start out from a set of nested codes, then I come up with a set of nested codes, at the next highest level. That, again, is sort of obvious. If these guys were nested, then I get the appropriate -- if I take u, u plus v from sub-codes, I'm going to get a sub-code. Look at the notes if you want more than that little bit of hand-waving.

OK, that's something I need for the next thing. I want to find out what d is. What's the minimum distance here?

Of course, my objective is to make it equal to 2 to the m minus r. Did I succeed? What are the possibilities for u, u plus v? The possibilities are that they're both 0, or that this one is 0 and this is not equal to 0, or this is not equal to 0 and this is 0, or that they're both not equal to 0. And I'm going to consider those four cases.

Since every linear code include the all-0 word, it's certainly possible that this comes out at 0, 0. The only possibility for this to come out 0, 0 is if I choose u equals 0.

Then I have to choose v equals 0. So there's one-code word of weight 0 in my new code. But that's OK. If there were two code words with weight 0, well, then the dimension would be wrong. This is in effect a proof that the kernel is just 0, 0. And so the dimension is OK. It's a one-to-one map.

All right. So I don't really have to worry about that. I'm going to get one all-0 word in my new code. I can afford one all-0 word. I'm always going to have to have it anyway. It's linear.

All right, so these are the more interesting cases. Suppose the first half is 0, but the second half is not 0. That implies that u is 0. That implies that the second half is just v, which is not 0. So v must be a non-zero code word in this code, which has minimum distance 2 to the n minus r. So in this case, I prove that the distance is greater than or equal to 2 to the m minus r.

Similarly, suppose this one is not 0, but this is 0. OK, if that's 0, it can only be because I chose u equal to some v. and that means the first half, then, is that v. So again, v is in this code that has enough minimum distance. So in this case, I proved that the code word has weight 2 to the m minus r.

And finally, let's take the case where they're both non-zero. In that case, u could be an arbitrary word in this code which only has distance 2 to the m r minus 1. So the first half is going to have weight at least 2 to the m minus r minus 1, but that's all I can say about the first half.

But now the second half, what is this? This is a higher-rate code than this. This, by the nesting property, is a sub-code of this.

So if I add a word in a sub-code to a word in the code, I'm going to get another word in this code. So u plus v is still in this Reed-Muller code, still has a minimum weight, if it's non-zero, of 2 to the m minus r minus 1. So the distance is at least this in the first half, this in the second half. And that's, of course, still good enough.

OK. So by this construction. I've assured that I'm going to get a d greater than or equal to 2 to the m minus r. And of course, you can easily find cases of equality, where it's only 2 to the m minus r. If this has a word of weight 2 to the m minus r, then you can clearly set up one like this that has weight 2 to the m minus r. Just pick one of the minimum-weight code words as v, and u as 0. So the minimum distance is 2 to the m minus r.

All righty. So those are all the properties we need. And then, I like to display these properties in a tableau which you have in the notes, which goes as follows.

Let's just start listing these codes. Here are the length 1 ones. We only found 2 of them, 1, 1, 1, and 1, 0, infinity. So there are two codes of length 1.

Now it turns out that if you combine these according to the u, u plus v construction, you get 2, 1, 2, where the weight 2, 1 is just the -- you take the first half is 1, and the second half is 1. So you can build this in the same way. And similarly, we can say just by definition, we're always going to put a universe code at the top and a trivial code at the bottom. So now I've listed all my Reed-Muller codes with length 2.

Now to construct the ones of length 4. Again, I'll put a universe code at the top, a trivial code at the bottom. I'll use my construction now to create a 4, 3, 2 code here. I'm just using all these properties. And down here, my construction, when you combine these two things, you always get a repetition code, again, 4, 1, 4. And I guess I've hand-waved. Exercise for the student, prove that combining a repetition code with a trivial code under the u, u plus v construction always gives a double length repetition code.

It's clear. v is always the all-0 word. u is either all-0 or all-1. So we get two words, one of which is all-0, and one of which is all-1, double length.

All right. So now I can just go on indefinitely, and without a great deal of effort. Here I find that k is 7, just by adding up these two things. The 8, this gives me an 8, 4. 4 code. This gives me an 8. 1, 8 code, and similarly down here. And this, I now just turn the crank.

16, 16, 1. I always put that on top. The next one is 16, 15, 2. Next one is 16, 11, 4. After a while, you don't know what you're going to get. But you get something.

You've proved that all of these codes exist, that they're all linear. They all have the n, k, d that we've specified, and that furthermore, they're nested. And a final property, which you might suspect, looking at these tables, is that the dual of a Reed-Muller code is also a Reed-Muller code. And they're paired up according to k and n minus k.

Let's see. 15 and 11 is 26. 11 and five is 16. 5 and 1 is 6. And 32, 1, 32, and so forth. Did you see how I proved that all these codes exist? And if I continued, I could get arbitrarily long codes, one of your simple homework problems is just to do this, continue this for 64 and 128, see what additional codes you get.

And so I now have this infinite family it of that kind of covers the space n, k, d in some sort of sparse way. But it indicates how k and d go with n. And we come back to our original question, how well can we do with binary linear block codes.

Well, here's some binary linear block codes, pretty close to the best we can find, actually. The ones I've listed up here are all as good as we can find. And how well can we do? Really, we don't need to know much to evaluate the performance of, say, 32, 6, 8 code, which is now getting to be a pretty substantial code, with 2 to the 16 code words, to 65,536 code words.

So we built a fairly sizable constellation here in a 32-dimensional Euclidean space. And how good is it? Let's take 32, 16, 8. Can I graph its probability of error per bit, a good estimate of it? Can I? Is there any information I'm lacking?

OK, I've been talking too long, because when I go to the class, I'd like a little response. So you were going to say something?

AUDIENCE: We don't have [INAUDIBLE].

PROFESSOR: OK, let me tell you that the n, d is approximately 600-something, so let's say 630. It's probably not exactly correct. In the notes I give a formula for n, d -- a formula that I don't derive, that is known for Reed-Muller codes. And from that, you can compute n, d for any of these codes.

This parameter is m, r. All right. So I'll give you n, d as well. Now can I get the good estimate for the probability of error per bit? How do I do that?

Union-bound estimate. All right, so what's it going to look like? What are the two subsidiary parameters I need? One is the coding gain, right? What is the nominal coding gain of this 32, 16, 8 code?

Come on, this is not a difficult computation. OK, what's k over n? 1/2? I think we can all do that one. What's the nominal coding gain? Take 1/2 times d, 8. So now our coding gain is 4, which is what in dB? 6 dB.

Wow, gee whiz. I've already got a nominal coding gain of 6 dB. Remember, my whole gap was, depending on how I measured it, 9 or 10 or 11 dB. This looks like already a sizable fraction of the gap, with just a simple construction.

Well, we better pay attention to this error coefficient as well, or the number of nearest neighbors. Kb is, let's just take a rough estimate here, what's this going to be? About 40. That's good. It's just this divided by 16. So there are about 40 nearest neighbors per bit.

How much is that going to cost us? Not so good. Well, it's a little bit more than 5 factors of 2. Maybe 5 and a 1/2 factors. It's less than 5 and a 1/2. So this very roughly, something will cost me about 1 dB.

And so I get an effective coding gain of 5 dB. That's my first very gross estimate. All right, well, it's not 6 dB. It's only 5 dB. That's still not bad.

Again, if I really wanted to know what this was, I would write this out as 40 or whatever it is times Q to the square root of four times 2Eb over N_0. And can I plot that quantity? Yes, if I have MATLAB.

Or actually, all I need is my baseline. So if this is my baseline, which went through 9.6 dB at 10 to the minus 5, then we remember how to plot that. I just take this whole curve bodily, and I move it 6 dB to the left. So I get the same curve, this is not very good, going through 3.6 dB. Sorry about that. How's that? Get it way down here.

But then I also have to raise it by a factor of 40. So it really looks more like that. That's really going to go more through about 4.6 dB or something like that. That's what these calculations are, Ashish took you through, I believe.

And so while 6 dB was my nominal coding gain, my effective coding gain is 5 dB. But still, hey, not bad. I have very easily been able to construct a code that gives you about 5 dB of coding gain.

Is there any fly in this ointment? Can I just go out and build it now? I need a decoder. Who said that? Thank you. Good point. What's the decoding method assumed for this code? Excuse me?

AUDIENCE: Just a table right now.

PROFESSOR: Just a table, based on what? I get some kind of received n-tuple, which is actually just a random, some kind of vector in 32-dimensional space, 32 numbers, real numbers. And really, I'm assuming minimum distance decoding. So in principle, I want to compute the distance to each of the 2 to the 1/6th, 65,000 code words.

And actually, nowadays, you might just do that. That's not a formidable task. Back when I got into this business, that would have been considered outrageous. But nowadays, you could keep up a pretty good decoding rate, even doing 65,000 distance computations and just finding the best one.

That would do it. That would give you this performance. But of course, we are going to be looking for somewhat more efficient decoding schemes than that.

All right. So at this point, that's the only fly in the ointment. We have a way of getting this kind of error curve, provided that we're willing to do exhaustive maximum likelihood or minimum distance decoding. And furthermore, we can continue.

It's also instructive to see what happens as we let n go to infinity. What are we going to get with this construction? We pretty well know. These are all going to be universe codes, which are not very interesting to us. They all have a nominal coding gain of one, and are just useless. They're basically just send a bit 32 times, send 32 bits, I mean.

OK, what are these codes along here? Let me start there. These are all single parity check codes. What's the nominal coding gain?

Well, as we get out here, what does the nominal coding gain approach? The code rate approaches 1. The code distance stays at 2. So the nominal coding gain goes to 2 or 3 or dB, at a rate of 1 or a nominal spectral efficiency of 2.

OK. Well, these are totally simple codes, single parity check codes. And even with that, it looks like we can get 3 dB of coding gain. But what's the number of nearest neighbors here? Number of nearest neighbors is just n, n minus 1 over 2, n choose 2.

So the number of even dividing by k, which is n minus 1, we still get a Kb that goes up linearly with n. So what's in fact going to happen to the effective coding gain? The nominal coding gain will go up and reach an asymptote of 3 dB. This is nominal coding gain.

But somewhere out here, as this has reached an asymptote, the effective coding gain is always going to be less, and it's going to have to bend over. And according to our rule of thumb, it'll eventually go back through 0, because the cost just keeps going up linearly in terms of Kb. So the effective coding gain is not as great. It has a peak for some n.

And so there's some maximum effective coding gain, which again I've left for you as a homework problem, that's less than 3 dB. And I'll give you a hint that it's of the order of 2 dB. It's pretty easy to work out in the five minutes before the next class, not a difficult problem to find out when this thing reaches its maximum.

But still, these are very simple codes. We'll see in a second they have an extremely simple minimum distance decoding algorithm called Wagner decoding. It's trivial, not hard to do minimum distance decoding for these codes.

And so, OK, not hard to get 2 dB of coding gain. What happens if we go along this line here? These are all codes of minimum distance 4. Again, start here.

They're called extended Hamming codes. A Hamming code has minimum distance three and is suitable for hard decisions, single error correction. These codes all have minimum distance four.

We've seen that it's always worthwhile to add an overall parity check to get an even minimum distance, if we're looking at Euclidean space coding gain. And so these are actually slightly better than Hamming codes. They're called extended Hamming codes, because they're Hamming codes extended by a single parity check. They have one more unit of minimum distance.

So again, asymptotically, these are called extended Hamming. The nominal coding gain goes to what now? This is, the rate is again going to 1. The distance holds at four. So the nominal coding gain goes to 4, or 6 dB, while again, the spectral efficiency goes to two bits per two dimensions, the nominal spectral efficiency rate to 1.

But again, you have this kind of phenomenon. And I ask you to work that out also on the homework, where even though the nominal coding gain plateaus eventually at 6 dB, there is a maximum effective coding gain, which is something more in the range of 4 to 5 dB. You figure out what it is.

And there's a limit to how much effective coding gain you can get with these codes. Does this all makes sense? Are you seeing how I'm arguing?

OK, here's another interesting sequence of codes. These are all half-rate codes, or nominal spectral efficiency one bit per two dimensions. I briefly mentioned that these things pair up in duals. The 16, 5 code is the dual code of the 16, 11 codes.

If you see, there's a symmetry about rate 1 by 2, such that this is k, and this is n minus k. And so you would suspect that this guy is the dual of this guy, which it is. This guy is the dual of this guy. This guy is the dual of this guy. And this guy is its own dual. It's a self-dual code of rate 1 by 2 or spectral efficiency 1. So these are self-dual codes.

And what does their nominal coding gain go to? Well, the rate is always 1 by 2 times four, that's a nominal coding gain of 2. This has a nominal coding gain of 4. The next one in line would be a 128, 64, 16 code, nominal coding gain of 8. So the nominal coding gain actually goes to infinity.

That's pretty good. However, what is its true meaning? What we really want to know is what's the effective coding gain. And given that the nominal coding gain goes to infinity, is it possible the effective coding gain can get us all the way to the Shannon limit? Can we completely close the gap to capacity?

As far as I know, this is an open question. I strongly believe that you go along this sequence through maximum likelihood decoding, you will eventually get to the Shannon limit, that is the Shannon limit for this spectral efficiency, which is not the ultimate Shannon limit, but rather is 0, in terms of Eb over N_0 If you remember, for rate 1/2 for rho equals 1, the Shannon limit on Eb over N_0 was 0 dB. You may or may not remember that.

So there's a question. Does this take you all the way to the Shannon limit? And that would be a nice question for somebody to answer someday. I think it probably does, especially in view of the fact that if you take this set down here, again, this will be a homework problem, this turns out to be a set of Euclidean images of these codes are orthogonal signal sets.

For instance, 32, 6, 16, what is that? That's a set of 64 constellation points in 32 dimensions. And algebraically, it's not hard to figure out that every one of these code words is orthogonal in a Euclidean sense to one another, except for one that's complementary, which is just what you expect in a bi-orthogonal signal set.

So these give you bi-orthogonal, they're called bi-orthogonal codes, or first-order Reed-Muller codes, because the parameter r is 1 for all of these. What's happening to the spectral efficiency here? Spectral efficiency goes to 0. So these become highly inefficient from a bandwidth point of view, as we already know about orthogonal, bi-orthogonal simplex signal sets. They use up a lot of bandwidth.

But what else do we know about them? In this case, we definitely know that the effective coding gain does go to the Shannon limit, and in this case, to the ultimate Shannon limit for rho equals 0 as rho approaches 0.

So here, there's a proof that these codes can get you to the Shannon limit. But as we've already explained earlier, geometrically, it's at the cost of using much more bandwidth than you probably really want to use. But here, at least, is one capacity-approaching set of codes, just among these rather simple Reed-Muller codes along this line.

And of course, we also see our repetition codes, our trivial codes. So this is a nice representative family of codes. And it really does tell you quite well what to expect. Are you looking at your watch? Thank you. I appreciate the hint.

So we didn't quite finish chapter six today. Next time, we'll start out with the penalties of making hard decisions, which at first brush seems like a not unreasonable compromise to make. But it actually costs a serious penalty.

And that will finish chapter six. I'll do that as briefly as I can. And then we'll get into chapters seven and eight, which is finite fields and Reed-Solomon codes, which are the single great triumph of algebraic coding theory. So OK, that's tomorrow.