PROFESSOR: Chapter six, problem set three, problem set two solutions.

OK. Good morning. Nice to see you all again on this mild day.

I want to start off by thanking Ashish greatly for giving the last three lectures. I have very high confidence in Ashish, and I'm sure he give you excellent lectures and was able to answer all your questions, right? But if you feel dissatisfied at any point, you'll have an opportunity to ask the question again today.

We will continue now in chapter six. I understand Ashish got up to section six point one already, so that makes my life a little easier. And we'll just continue to plow ahead through the notes. You have as handouts today chapter six, the outgoing problem set, the solutions to the incoming problem set.

So does anyone have any comments or questions or suggestions as we go forward at this point? Let's say I'll do a very quick review of everything so far, at least what we need to proceed, so you'll have a chance to ask questions along there if you like. Anyone have anything to say? Anyone want to say thanks to Ashish? All right. Great.

OK. So a very quick review. We first said we were going to work with a continuous time Additive White Gaussian noise channel, but we quickly reduced it to a discrete time additive white Gaussian noise model. Vector model, Y equals X plus N, where the vectors may go on as long as you like. And implicit in here is we have some kind of power limitation expressed by various -- I know I've probably introduced too many parameters in the early part of the course. But you'll see them all in the literature, and I thought you might as well see the relation between them all at once.

The noise. The main thing about this is it's an independent, identically distributed noise, also independent of X, and it's characterized simply by its variance per symbol, or average power. Again, there are various notations that we can use for its power.

OK. However you express the power of the signal of the noise, you're going to get some signal-to-noise ratio out of that, as long as you use consistent the same units for signal and noise. And so the key parameters of this channel turn out to be just two. One is the signal-to-noise ratio. And the other is something which in the discrete time domain we might think of as a data rate, rho, the number of bits per two dimensions.

But we were at pains to show -- and the reason we measure it in bits per two dimensions is that it converts directly to the spectral efficiency, or we sometimes say, the nominal spectral efficiency, or the Nyquist spectral efficiency. Basically, this is the spectral efficiency if you assume that you only use the nominal Nyquist bandwidth.

And both in 450 and here it was shown that you can really get as close to the nominal bandwidth as you like, its sharp or roll off as you like, by using filters. Very sharp filters or very sharp digital filters that today we can really realistically get effectively to the Nyquist bandwidth. So this is a good measure of nominal spectral efficiency, which is an extremely important parameter, the spectral efficiency in continuous time.

And then the Shannon limit is expressed in this very simple form that we can never do better in data rate or nominal spectral efficiency than log of 1 plus SNR. And the units are bits per two dimensions.

So there's an awful lot about parameters and units in the first part of the course, and sometimes it's a little overwhelming and confusing. But in my experience in engineering is, it's extremely important to get the units right. It helps you in thinking clearly. It helps you to focus on the right things. And so in the first couple of chapters, there's a lot of emphasis on getting the units right.

For instance, bits per two dimensions. I could simply assert to you that things are always to come out better if you do things per two dimensions. This basically has to do with Gaussian distributions being simpler in two dimensions than they are in one dimension. So you know you can get closed form integrals in two and four and so

forth dimensions that you can't get in one and three and so forth dimensions. So you know, the geometrical formulas for the volume of the sphere and so forth are much simpler in even dimensions than they are in odd dimensions.

This all, I think, has to do with the fact that really a one-dimensional complex Gaussian variable, or in general, complex Gaussian variables, are in some sense mathematically simpler than real Gaussian variables. For instance, the Q function is a very ugly thing. In two dimensions, you can closed-form integrals with probability of error. So there's a lot of clues that we really want to think in terms of pairs of real dimensions.

All right. And then we introduced a couple more parameters. And I understand Ashish got the questions that I get every year. Why do we introduce both of these things? SNR norm, which is SNR normalized by 2 to the rho minus 1, that's motivated directly by the Shannon limit formula. Or Eb over N_0, which is SNR divided by rho. You know, these are both more or less equivalent to each other. From this Eb over N_0 is just 2 to the rho minus 1 over rho times SNR norm, so why do we introduce both of them?

And there's not an intellectually very strong argument to introduce both of them. Because one is easily translated into the other. For fixed nominal spectral efficiency rho, there clearly is just a one-to-one translation. We could use either one. If you graph something versus Eb over N_0 or versus SNR norm, it's just a matter of shifting it by this factor. They're going to be exactly the same graph with the 0 dB point in a different place, according to where 0 dB is.

Philosophically, of course, SNR norm, which is sometimes called the gap to capacity, is exactly a measure of how many dBs away are we from the Shannon limit. Measuring things on a log scale in dB. And so 0 dB is always the Shannon limit point with SNR norm, because this statement translates into SNR norm is greater than 1, which is 0 dB.

So here the focus is always, how far are you from capacity? How far are you from the Shannon limit? And that really is the modern view. In the early days of coding,

the view was, well, how much coding gain can we get over no coding? And as we'll see, Eb over N_0 is often a very convenient parameter to use when we're focusing on coding gain. In fact, for binary linear block codes, which is what we're talking about in chapter six, we get an extremely simple expression that Eb over N_0 it's just kd over N, where N, k, d are the basic parameters of a block code. If you don't know those yet, you will by the end of this lecture.

And Eb over N_0 is what was put forward in the early days, when the principal coding application was coding for deep space communications. It makes sense in the power-limited regime. In the power-limited regime, we showed that this essentially goes -- this is rho log2 over rho, so up to a factor of natural logarithm of two, these are the same, almost independent of the spectral efficiency, as well. And so this is a very natural thing to use, especially in the power-limited regime.

In the bandwidth-limited regime, as rho gets large, then these things become very different. SNR norm always keeps us in the neighborhood of 0 dB. This thing goes up to 10 dB, 20 dB, 30 dB, and so forth. Nonetheless, you'll see in some literature, people continue to measure against Eb over N_0 maybe I would say, just because they don't know any better.

Anyway, since I started writing these notes about eight or nine years ago, I've been advocating SNR norm. It's more and more widely used in our business, in the coding business. Or equivalently, one always shows nowadays how far are you from capacity, and that's what SNR norm is about. And you can always translate this into this and this into this by this simple formula.

So that's why we introduce them both. Eb over N_0 is traditional. SNR norm is more of the modern gap to capacity viewpoint. Any questions about that? Because I understood that Ashish got a number of questions about that. Yes?

AUDIENCE: And the probability of error you mentioned [UNINTELLIGIBLE] is natural with SNR model.

PROFESSOR: Well, this is a slightly different point, actually. So we go on from this to talk about the

power-limited regime, which we defined more or less arbitrarily as the regime where rho is less than or equal to two bits per two dimensions, and the bandwidth-limited regime, which is where rho is larger.

And at this point, I simply assert that it's better to do everything per two dimensions in the bandwidth-limited regime and to do everything per bit in the power-limited regime. And the reason for this is basically long practice and intuition and experience that things do work out better, and this is the proper normalization. But I think at this point in the course, with your background, this is only an assertion, all right?

So I simply say, this is the way we're going to do things in bandwidth-limited and power-limited. For most of the rest of the course, we're going to be in the power-limited regime. Then we'll come back to the bandwidth-limited very late in the course. So you can now forget this assertion for a while. Yes?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, I'm trying to focus on that here. That if two systems have different rhos, then you should take that into account in their comparison. And when you see charts of a rate 7/8 system, or a rho 7/4 system versus one that's rate 1/8, those are very different regimes, and it probably isn't fair to compare two systems, just in terms of their probability of error versus Eb over N_0 at two different spectral efficiencies.

What is more fair is to compare them in terms of their gap to capacity. You can get more powerful codes with more error-correcting capability the lower in rate that you go. And so you should start from the point of view that two systems with different rho are incomparable, and then using this you can say, but, you know, if we can get within 1 dB of the Shannon limit with both of them, then they're both approximately equally powerful. That would be the modern point of view. But it's really apples and oranges if they have different rates, different rhos. Yes?

AUDIENCE: [INAUDIBLE] If [UNINTELLIGIBLE] systems modulation [INAUDIBLE]. Then is it fair -- because if one of them has coding --

PROFESSOR: If none of them have coding, well --

AUDIENCE: No, just modulation.

PROFESSOR: So what shall I say to that. In the power-limited regime, an uncoded system is simply binary modulation or 4-QAM, and there aren't different systems to compare, really. There's only one baseline system.

As you go up into the bandwidth-limited regime, then it is fair to compare systems of the same class, like M by M QAM. That's a simple uncoded system. Now there, rho keeps changing. rho equals 2, rho equals 4, rho equals 6, rho equals 8. Or you can easily find intermediate rates that are uncoded.

And there you find that this normalization makes all these systems comparable. In fact, we saw that we got a universal baseline curve for the bandwidth-limited regime, which was independent of rho, for, say, the class of 4 by 4, by the class of M by M bandwidth-limited QAM systems.

I'm sorry. I got off a plane at nine o'clock last night, so I may not be totally coherent today.

But all of these systems have exactly the same baseline curve up to the approximations we've made that their exactly four nearest neighbors and so forth. But on a curve of probability of error per two dimensions, per QAM symbol, versus SNR norm, we get a universal curve.

So that indicates they really are comparable. They form a class where the performance of the class is independent of rho. And that's sort of typical of the bandwidth-limited regime. Two systems that differ, that basically use the same code with smaller constellation or a larger-based uncoded constellation are going to be directly comparable, and are probably going to have the same gap to capacity. Going to be just as far away from the Shannon limit. But we don't have that phenomenon in the power-limited case.

Other questions, or shall I proceed? OK. Again continuing what I hope will be a

quick review, but I don't want to go any faster than you're comfortable with.

We get this baseline curve for 2-PAM or 2 by 2 QAM, same thing. So this is the baseline of Pb of E on a log scale, where we typically go down to ten to the minus six, ten to the minus five. Here's 0 dB. This goes down through about 9.6 dB here or about 10.5 dB here. And the ultimate Shannon limit is over here. The ultimate Shannon limit for very low rho, as rho goes to 0, is about minus 1.6 dB.

And we get expressions. 0 dB is the Shannon limit at rho equals 1. At rho equals 2, it's up about 1.8 dB. Shannon limit for rho equals 2, and so forth.

And anyway, so we see as a function of rho, we can measure the gap to capacity at different probabilities of error and see how much coding gain is, in principle, possible. And then for a coded system, we can put that on here. The effective coding gain is the distance between here and here. At some target probability of error. It's going to differ according to probably of error. You found a way of getting a good, rough-cut estimate, at least for not very complicated codes called the union bound estimate for any signal constellation in any number of dimensions.

So if we have a signal constellation A, which consists of M points in N dimensions, so forth, basically found that we could, in the power-limited regime, get an approximate expression from considering pairwise error probabilities that's very simple.

And I used this notation. Q of the square root of some coding gain of the constellation times 2 Eb over N_0. If it's just 2-PAM then the coding gain becomes 1. This is the average number of nearest neighbors per transmitted bit.

And so the whole thing reduces to this expression in terms of couple of parameters. the principal parameter is the nominal coding gain, which is the minimum squared distance of A over 4 times the energy per bit of this constellation A. So we really only need to know this kind of normalized measure of goodness of the constellation.

And K_b of A Is the average number of nearest neighbors. I forget what we call it. What is the numerator here? K_min of A, which itself is an average, over the

number of bits that we're actually sending, which is log2 of the size of A.

So basically we only need to know a couple of parameters of this signal constellation. Its minimum square distance is very important. Energy parameter, which we choose to make the energy per bit, or so that we get this expression. And the average number of nearest neighbors per bit that we transmit. OK.

And our best example so far is the tetrahedral constellation, where we basically pick every other point from the 4 simplex.

Maybe I should do that in a different color. I don't think that's a different color. Nope.

Anyway. You know this quite well by now. And if we do that, we find that normalizing everything, this is 4/3 or 1.25 dB. And every point has three nearest neighbors, and we're sending 2 bits, so this is 3/2.

So then we get this handy dandy engineering rule to make a quick plot of the union bound estimate. Given that we decided to put this on a log-log scale to start with, all we have to do to plot this expression -- if we're given, as we always are, the baseline curve, which is simply Q to the square root of 2 Eb over N_0.

How do you convert this curve to this curve on a log-log plot? Well, you simply move it to the left by the coding gain. And you move it up by the log of whatever K_b is.

So if the dominant effect is moving it left, in this case, by 1.25 dB -- I'm going to wind up getting about this curve, so I won't draw it again -- then we move it up by a factor of 3/2, we developed a rule of thumb that said a factor of two is basically going to cost you about 0.2 dB. Around ten to the minus five. This is just based on the slope of this baseline curve, as long as we're somewhat in that region. So this will cost us about 1/10 of a dB, so we'll get an effective coding gain of about 1.15 dB.

So graphically, we take this curve bodily. We move it over 1.25 dB, and we move it up by a factor of 3/2. And what we'll find is the effective coding gain is thereby reduced to, I estimate, about 1.15 dB.

And this is all just engineering rules of thumb. Nice template. I sometimes say you should fill out a copy of this baseline curve, cut it out, put it in your wallet for the duration of this course, because you'll have the opportunity to make this kind of calculation again and again.

All right. So that's union bound SNR. For simple constellations, the union bound estimate is very accurate. So this is a very good way to proceed, from an engineering point of view.

We want to write a paper? You know, it's a little late to write a paper on the performance of the four simplex signal constellations, but if you wanted to, and you wanted to have one graph in that paper, you would have the graph of -- well, you would actually probably compute the exact error of probability, either by analysis or by Monte Carlo simulation, you would put that in there. But in your first draft of the paper, you would put in union bound estimate and you would find that wasn't far off.

OK. Any questions on that? This has basically got us up to chapter six.

So in chapter six now, which Ashish got into last time. This is basically about binary linear block codes. And well, we first just start talking about binary block codes. That's what Ashish did last time.

We basically take 0 and 1 as our binary alphabet. We take a blocks of length n. Sorry. Are we using little n or big N? Little n, where n is called the block plank. So we take the set of all binary symbols of length n, and we're going to convert this to real n space.

How do we convert it? Component-wise, by the standard 2-PAM map. We map 0 into plus alpha, 1 into minus alpha. So this maps into plus or minus alpha to the n. So the entire universe of possible code words that we have is this set of 2 to the n real n-tuples, which are simply the vertices of an n cube of size 2 alpha, obviously, right? Just like this.

We have eight possible vertices in 3-space. They obviously all have the same power. They all lie on the surface of a sphere of squared radius n alpha squared.

So they're not only the vertices of a cube, they're vertices that are spread on an equal energy sphere. And the whole idea is that our code is going to be some subset of this, and the code will map under the same map, which we call S, into some subset of the vertices of the n-cube. OK?

And again, our favorite example is the tetrahedron. For instance, if we take the code as being 0, 0, 0, 0, 1, 1, 1, 0, 1 1, 1, 0, those for binary three-tuples, this maps into T, the tetrahedron. OK?

Well, we get this signal structure, which we've already found the coding gain. It has a little bit of coding gain. We actually accomplished something.

All right. So that's the basic idea that we go through in 6.1. That's our favorite example. And Ashish also shows you that, you know, you might think this is an awfully restricted way of designing constellations.

But when we're talking about low spectral efficiencies, by going through the capacity calculation, we can assure ourselves that in principle, this is not going to be very sub-optimal for it. Capacity is about the regime where n gets very large for very long block codes. We don't lose much in potential coding gain, or in the Shannon limit, equivalently, by restricting the input output input alphabet to be just these two numbers, plus or minus alpha, I should say, rather than using the whole real line as an input, as long as the nominal spectral efficiency is less than or equal to one bit per two dimensions.

And the exact numbers -- at rho equals 1, you lose about 0.2 dB, in principle, using the Shannon limit as your guide. And for lower rho, it just goes to 0. And you did this last time.

OK. So it's a very reasonable, and obviously attractive from an implementation point of view, way of designing signal constellations. And again, this is basically all we're going to be talking about for the majority of the course, is signal constellations like this. But of course as we get to codes that are thousands of bits long, or perhaps even infinitely long, as in the case of convolutional codes and trellis codes,

sometimes you forget that we're really talking about just putting points in Euclidian signal space. Because we're going to be very abstracted back into the binary domain.

But in this course, we're always thinking of codes as means of designing signal constellations. Of course, codes are used far more widely than just for signaling over the additive white Gaussian noise channel. I've sort of packaged this course as a search to get to capacity, we have added the additive white Gaussian noise channel. First of all because this corresponds very closely to history. Within this package, we can talk about all of the principal classes of codes that have been developed to date, and we can compare them in some performance terms. How close do they get to capacity on the additive white Gaussian noise channel.

So you get most of what you would get if you took a communications-free view of coding theory. You know, people are interested in codes for computer memories, for God knows what, lots of other things. And for many of these other applications, you are going to be interested in the same class of codes. Basically, you're going to want to maximize the distance between code words for a given rate, which has to do with the size of the code. And so you're going to be interested in the same codes.

Putting it in the framework of getting to capacity on the additive white Gaussian noise channel gives a motivation, gives a very nice story, because over 50 years, we were able to get to capacity, and gives it a real communications flavor. Some of you, I'm sure, are here without any interest in communications whatsoever. You simply want to know about coding. You will still get that story, but you'll get it in this nice package. That's why I do it this way.

OK. I'm about to get into new stuff. End of review. Any questions? So you must have done a great job. Everyone understands perfectly.

All right. So now let's talk about, as I said, binary linear block codes. When you see the word "linear," that's a signal that there's some algebra ahead. And so this is the very first point at which we get into what's called algebraic coding theory. The

algebra will be extremely simple at this point, not to worry.

And what are we doing here? First thing we do is to identify 0 and 1 with the binary field, which I'm always going to write as F2. The older way of writing this is GF(2), for Galois field with two elements. They mean exactly the same thing. Nowadays most people write just F2. Now we have [UNINTELLIGIBLE], we can write this blackboard F.

And OK. Step one. We've identified our alphabet with a field. So algebraically, this is a field. Some of you know exactly what I mean by that. Others don't. We'll come back to this again. Informally, I would say a field is simply something where you can add, subtract, multiply, or divide. Our best examples of that before now have been the real and complex fields. There is a more formal definition of that.

In this case, we have only two elements. So let me just write down the tables, which you all know, regardless of your background. What's the addition table for this field? Well, 0 is the additive identity. So we know that 0 added to anything gives itself, so that gives us three of the entries of this table.

What do we put down here? There's really only one choice to satisfy one of the axioms of the field, which is that under addition, the field must form a group. In particular, that means that all elements are invertible. That means that each row or column has to be a permutation of the group elements. And so the only possibility here is to put in a 0.

And well, we are forced, if we're going to make 0 and 1 into a field to find an addition table which is the table of mod2 addition. You've had this stated as an axiom before. You can derive it just from the fact that 0 needs to be the additive identity, and then we need to put a 0 in here in order to get invertibility.

Under multiplication. What is the multiplication table? Well, the additive identity is also a nullifier under multiplication in any field. So 0 times anything is equal to 0. 1 times anything is equal to itself. So that completely determines the multiplication table. Again, just the table of mod2 multiplication.

So if I'd done this axiomatically, I would have given you the axioms of field. Then I would show that under this binary addition operation, this multiplication operation, we satisfy the axioms. I'll do that as we get into chapter seven.

OK. So now we have 0 1 to the n, n-tuples, binary n-tuples, we will now regard as n-tuples of field elements, F2. And these will be regarded as vectors in a vector space. Well, I'll just say that F2 to the n is a vector space, which clearly has 2 to the n elements.

Now again, informally. What's a vector space? A vector space is always over a given field. In this case, it's going to be over the binary field, of course. F2. The given field is called a scalar. Just like vector space over the reals, the scalars are real numbers. Here the scalars are elements of F2.

And what do we have to have to make something algebraically a vector space? We have to have that the addition of two vectors is well-defined and gives another vector, and that the multiplication of a vector by a scalar is well-defined and gives another vector in our vector space. All right?

OK. So how are we going to define that to addition? If we want to add two n-tuples -- again, you aren't going to see anything here that you haven't already seen in some other context. What do we do? We add them component-wise, using the component-wise rules of field addition.

OK. So we just add two vector n-tuples, component by component. We obviously get some result, which is itself a binary vector or an F2 vector.

No problem there. Except that -- well, all right. As long as we're talking about all possible n-tuples, the result is certainly in the vector space, right? We add two binary n-tuples, we get a binary n-tuple. So the result is in F2 to the n. This is going to be the key test when we get to subspaces of F2 to the n.

OK. And multiplication is even easier. How do we define multiplication by scalars? Well, we only have two scalars -- 0 and 1. So 0 times any vector is going to equal the all 0 vector. Again, you can view that as just component-wise multiplication of

everything by 0, and since 0 times anything equals 0, we're always going to get the 0 vector. All right?

So is the 0 vector in the vector space? Well, yes. If we're talking about several n-tuples, it of course always is. And one times anything, again, we can just do this component-wise, and it just gives itself. Trivially. So since this was a vector in the vector space, this is certainly a vector in the vector space.

OK. This seems pretty trivial so far. But what's a binary linear block code? Again, focusing on the linear, is a -- now I'll give a formal definition. it's a subspace of F2 to the n for some n called the block length.

All right. What do I mean when I say a subspace? I mean subsets of the elements of a vector space that itself forms a vector space. So in this case when I say subset, I mean a set of binary n-tuples. OK? That itself forms a vector space. OK. What are the components of that?

To check that it forms a vector space, let's see. Multiplication by scalars is, again, easy to check. If it's going to be a subspace, then the all 0 -- so it has to contain the all 0 vector in order that when I multiply by the scalar 0, I get another element of this subspace.

Multiplication by 1 is always trivially satisfied. If I start off with a set, I multiply by 1, I'm going to get the same set. So let's check whether the elements of a subspace are closed under vector addition. What do I mean by that? I mean if you add two elements of the of the subspace together, you get another element of the subspace.

That's the key property to check. Key property -- we can write that as closure under vector addition. Which is also called the group property. It means that just under addition, under vector addition, the set of elements that you have forms a group. You add any two elements, you get another element of the subset.

The example is our favorite example so far. Let's take this little code, and I'm going to ask. Is that a subspace of F2 to the three? So does this equal subspace of the

set of all binary three-tuples?

Anyone care to hazard a guess whether it is or isn't?

AUDIENCE: It is.

PROFESSOR: It is? Why?

AUDIENCE: It has the all 0 vector.

PROFESSOR: It has the all 0 vector, first of all. Good.

AUDIENCE: [INAUDIBLE]

PROFESSOR: And it's closed under addition. Now how might we see that? You could, of course, just take all pair-wise -- you could form the addition table of these four elements, and you would find that you always get another one of these elements. For instance, 0, 1, 1 plus 1, 0, 1, is equal to 1, 1, 0.

In fact, you easily see that you take any two of these, add them together, you get the third one. If you call this a, b, and c, a plus b plus c equals 0, addition is the same as subtraction, because we're in a binary field. So that means that a plus b equals c, a equals b plus c, c equals b plus a, whatever you like. And of course, if you add 0 to anything, it's trivially closed under that.

All right. So it is. It satisfies -- it's all you've got to check.

A more abstract proof of this would be, this is the set of all even-weight three-tuples. If I add even to even, I'm going to get even. So of course my result is going to be another even-weight three-tuple, therefore in the set. That's the more algebraic proof.

All right. Suppose I just add 0, 0, 1 to all of these things. I'll get the set of all odd-weight n-tuples. Add any odd-weight n-tuple to this. Let me take the C prime, which is the set of all odd-weight n-tuples.

Is that a vector space? It doesn't have 0, and in fact, it's not even closed under

vector addition.

All right. If I take in C double prime is equal to 0, 0, 0 0, 1, 1, 1, 0, 1 and I stop there, is that a subspace? No. Because? Not closed. For instance, if I add these two together, I would get that, and that's missing.

OK. So actually, everything is much simpler when we're talking about finite fields. All the finite dimensional vector spaces consist of a finite number of elements. It's easier than real and complex vector spaces. There's no analysis involved. So forth.

All right. So a binary linear block code is in a subspace of F2 to the n. What are some of the key algebraic facts we know about vector spaces from our study of linear algebra, which I assume all of you have had in some form or another?

What's a key algebraic property of a vector space? What's the very first property? All vector spaces have a --

AUDIENCE: [INAUDIBLE]

PROFESSOR: Norm? No. Dimension. Dimension, that's where I'm going. What's the significance of dimension?

AUDIENCE: [INAUDIBLE]

PROFESSOR: The definition of dimension is the number of generators in any basis. So we're talking about generators of the vector space, or a set of generators which form a basis. If you think we have the same properties here in the case of vector spaces over finite fields, well, we probably do. But let's see how it works out.

So we're talking about things like generators, basis, dimension. These are all closely interlinked properties. Again, the fact that everything is finite here gives us very elementary ways of addressing all of these concepts. We don't have any geometry yet. We don't have norms. We don't have distances. We don't have angles. We'll talk about that in a minute. Here we just have a set that basically has these two properties. It contains the all-0 vector, and it's closed as the group property. It's closed under vector addition.

All right. So the first property is that the all-0 vector is always in the subspace, which I'll represent by C, meaning code. So when I talk about a code now, I'm talking about a binary linear block code, which by definition is a vector space, a subspace of F2 to the n, where n is the code length.

All right. Let me try to find a set of generators for the code. All right? If somebody gives me a code, they say, this is a binary linear block code. Let me see if I can find a set of generators for it. If I find three generators, then I'll know the dimension is three. That's basically where I'm going. I'll state this a little bit more formally as we go ahead.

So suppose I'm given a code. I know it's a binary linear block code, so I know it has the all-0 element. So how might I go about finding a set of generators? Let's just take a greedy algorithm. All right? Suppose the code contains only the all-0 vector. Is that a subspace? What's its dimension? 0. All right? So that's a very trivial vector space, but it's a vector space. Satisfies the axioms.

Suppose it is not the trivial code. That means it has more than the all-0 vector. So I take as my first generator any non-zero vector. OK? I can always do that. Don't have the axiom of choice involved here, because everything is finite. So I'm going to take g1 to be any non-zero vector.

Now I've got a generator. And how many code words does it generate? I want to take the set of all binary linear combinations of all the generators that I have so far. At this point, the binary linear combinations are 0 times g1 and 1 times g1 And that just gives me this word and this word. So now I have counted for two words with one generator.

Could it be that that's the whole code? Sure. The all-0 vector and any non-zero vector together form a one-dimensional subspace. That's all you can get from one dimension.

So I could be finished here now. But if I'm not finished, there's still more code words

that I haven't accounted for. Then I greedily pick a second generator. So this is now, let's say, any vector not generated by g1.

So I have a branch here. Either I've finished or I can pick another generator g2. Now with g1 and g-two, how many vectors can I generate? Let me take all binary linear combinations. So a binary linear combination is any vector of the form alpha1 g1 plus alpha2 g2 where these are both scalars. And therefore this can be 0 or 1, this could be 0 or 1.

So what have I got now? I've got 0, g1 g2 and g1 plus g2 I've got four binary linear combinations of two generators.

Could that be the whole code? Certainly. At this point, again, consider our standing example. I start out what I take as my first generator. Let me take g1 equal 1 0, 1, g2 equals whatever, 0, 1, 1. Then if I take all binary linear combinations of these two generators, I'm going to get the whole code, right? These four code words. They can all be expressed in this form.

Or I'm not done, and then I have to pick g3. And how many binary linear combinations are there of g1, g2, and g3? Eight. Are they all necessarily in my subspace? Yes, by the fact that the subspace is closed under scalar multiplication. Alpha g1. Alpha1 g1, alpha2 g2 alpha3 g3 are all in the subspace. Any vector addition of any of these scalar multiples is in the subspace. So I get now eight possible elements of the subspace, and I either may be done or not.

Continuing in this way, I get some number gk of generators, just by picking greedily the next one until I'm done. All right?

When I'm done -- so I have to stop. Why do I have to stop? All right. Let's look at the size. At each point here, this accounts for two code words, this for four, this for eight, this for 2 to the k. How many binary n-tuples are there? 2 to the n. All right?

So I clearly can't find more than n generators. More more than n independent generators, in the sense that the set of all the binary linear combinations are distinct.

All right. So k is, at most, going to be n. So I will stop in a finite number of steps. I'll stop at some number k. When I've stopped, that's because the code consists of all binary linear combinations of these k generators, and therefore has size 2 to the k.

So the only possible size of a subspace is 2 to the k for k less than n. A power of 2 where the power is, at most, n. So any subspace besides the subspace -- I'm repeating myself.

OK. So this means I found a basis. Does this mean that all possible bases of any subspace have the same size? Well yeah, it must. I mean, I've proved now that any subspace has to have this size 2 to the k for some k. So obviously, if I go through this process, no matter how I choose my generators, I could choose any pair of these non-zero n-tuples as my generators. So that would be a legitimate basis. Take any two out of these three.

But it's always going to take exactly two of them, right? Why? Because the size of this subspace is four. Two to the two. So if I go through this process, I'm always going to come up with k generators, where k is the log to the base 2 of the size of this code that I was given, which I was told was a linear code, meaning it's a subspace. So I'm somewhat free to choose the generators, but I'm always going to come up with k of them if the code has size 2 to the k.

So a basis is a set of k linearly independent k-tuples, where linear independence has the same meaning here as you're accustomed to, meaning that all linear combinations of the k generators are distinct. So I get 2 to the k distinct elements of the code.

And I say the dimension of the code is k. In this case, basically it's the size of the code is 2 to the k, then its dimension is k. Has to be. And all basis have k generators in them. And there are, in general, many ways to pick them. All right?

So just by considering this greedy basis construction algorithm, I basically find the size of any subspace is a power of two, and the power is equal to the dimension. And any basis is going to have that cardinality. Are you with me? This is a pretty

simple proof.

And I call this an n, k binary linear block code. n being the length. That just means that every code word is an n-tuple over the binary field. And k is the dimension. So an n, k binary linear block code has size 2 to the k. That's 2 to the k distinct code words. Main example is this guy.

Easy? Any questions? I think this is clear.

AUDIENCE: Can n use the number of code word, I take like 2 to the k?

PROFESSOR: n is something I specify a priori as the length of every vector in the code. In other words, it has size 2 to the k, and it's a subset of the set of all binary n-tuples, which I write like that. In other words, the elements of the code are binary n-tuples. If I write them out, each one has length n. OK? We're good?

What are some other examples? The simplest codes you can think of is, first of all, an n, 0 code. That means a code with dimension zero, has size what? 1. And what does it consist of?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. So this is the so-called trivial code, just containing the all-0 word. Has to mention 0. It is a binary linear block code, but it's not much use for communication. So but nonetheless, we include that in this family.

Another trivial one is n, n. What's that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: F2 to the n, right. What's its size? 2 to the n. That means it has to contain all distinct binary n-tuples. So this is called the trivial code. This is called the universe code. Contains the entire universe of binary n-tuples.

Let's get some slightly less trivial ones. n, 1. What would that be? An n, 1 code. What's its size? 2. What does it consist of? The 0 word, and? Any other non-zero

generator.

And that's correct. This can be 0 and any generator, two words. In communications, where we want to maximize the distance, in some sense, between the two code words, what is g always taken as? 0,1's, right.

So if it's in particular the all-0 and the all-1 word, which I might write as a vector of 0's and a vector of 1's, this is called the repetition code. The binary repetition code of length n. It either gives me a 0 and I repeat it n times, or gives me a 1 and I repeat it n times. So whenever you see n,1, you can pretty well assume it's the repetition code, though it might be any pair 0, g.

And n, minus 1. This is an interesting one. Again, while this could be a lot of things, in communications, whenever you see this, this will always be the set of all even-weight n-tuples. In other words, the set of all n-tuples with even parity such that if you sum up all of the components of any vector, mod2 equals 0. OK?

So this I will call the single parity check, or more briefly, the SPC code, or the even-weight code. That's equally good.

And here we maybe should do a little bit more work. Say, is this in fact a subspace? Does it include the all-zero code word? Yes, all-zero has even weight. The sum of any two even-weight code words, an even-weight code word, an even-weight n-tuple. Yes.

As here. This is an example. This is the three, two SPC code.

OK. Why is this dimension n minus one?

AUDIENCE: [INAUDIBLE] Is every code word orthogonal to the one-vector?

PROFESSOR: OK. That's an excellent answer. It's a little advanced for us right now. I'm looking for an elementary argument.

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. So you're saying we have a set of generators that looks like this. Is that what you're saying? You are correct. And how many such generators are there? There are n minus 1 of them.

I always like to find the most elementary argument possible. I think the most elementary argument here is that the number of even-weight n-tuples is equal to the number of odd-weight n-tuples, and together they form the set of all n-tuples. So exactly half of the n-tuples are even weight. That means there are 2 to the n minus 1 of them, 2 to the n over 2, and therefore, the dimension must be n minus 1. But perhaps this is just as elementary a proof.

Well, however you do it, you'll find that there are 2 to the n minus 1 of them, or that here is clearly a set of generators. It might take a few more lines to show that every even-weight code word is a linear combination of this particular set of generators, but it's certainly true.

All right. So these four classes of codes, these two entirely trivial ones, these two which are actually somewhat more interesting for coding purposes -- we've already seen, we can get a coding game with this length three, dimension two code -- are basically the simplest codes we can think of. The simplest binary linear block codes. Now we'll see them again and again. They turn up.

All right? So the whole course is going to be about finding 1's in between. More complicated ones. There's clearly more room to play. For instance, if k is equal to half of n, which means that rho is equal to one bit per two dimensions, there's a lot of possibilities. And so we're going to explore those possibilities.

AUDIENCE: [INAUDIBLE] [UNINTELLIGIBLE]

PROFESSOR: Sure. Let's take the 6,5 code generated by these five generators. Well, it contains some odd-weight code words, obviously. But it's not as interesting from a coding point of view. It's not the only one, but it's the only one you'll ever see here.

All right. Let's see. One thing I didn't point out in the notes but probably should have here is, what is rho for an n, k binary linear block code? How many bits can we

send? Suppose we take the Euclidean image of this. It is going to have 2 to the k points, 2 to the k vertices of the n-cube, and so what is rho? What is the rate in bits per two dimensions?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Right. The rate is basically k over n over 2, if you like, or 2 k over n bits per two dimensions. You can send k bits in n dimensions, or 2 k over n bits per two dimensions. And since k can't be larger than n, this is going to be less than or equal to two bits per two dimensions. So again, we see we're definitely in the power-limited regime, and that we can really get any nominal spectral efficiency between 0 and 2 by choosing k and n appropriately.

All right. So n and k determine the rate, determine the nominal spectral efficiency.

All right. Let's now talk about things that I think are also very old hat to you. I mean weight and distance. But we begin to get into areas that show us that these vector spaces are very different from the real and complex vector spaces that we're accustomed to.

So what are we doing now? We're starting to get into the geometry of this vector space. The geometry is not Euclidean geometry, but it's Hamming geometry. We define the Hamming weight of a vector as simply the number of 1's in v. So the Hamming weight of the all-0 vector is 0, the Hamming weight of the all-1 vector is n, and in general, the Hamming weight is somewhere between 0 and n. Just the number of 1's. Pretty simple.

And given two vectors x and y, what is their distance? The Hamming distance between x and y is equal to the Hamming weight of x minus y. This is the standard way of converting a weight metric into a distance metric. Or because addition is the same as subtraction in a binary vector space, we might equally well write this is as the Hamming weight of x plus y. And more informally, this is simply the number of places in which they differ.

OK. So if x and y are identical, then x plus y is equal to 0 and the distance is 0. If

they are complementary, y is the complement of x, then they differ in every place. The sum will then be the all-1 vector, and the weight, the Hamming distance, will be n.

And so again, the Hamming distance is somewhere between 0 and n, measures how different they are. Clearly going to be important for coding. It's going to translate directly into Euclidean distance under this standard 2-PAM map. OK.

Again, let's check that it satisfies the distance axioms. I don't know how many of you have seen this, but let's see. What are the distance axioms? Strict non-negativity. In other words, the Hamming distance between x and y -- that's a single comma -- is greater than or equal to 0, and equality if and only if x equals y. That's what strict means. So if we find the Hamming distance is 0, we can assert that x equals y.

We have, of course, symmetry. The Hamming distance between x and y is the same as the Hamming distance between y and x. Order doesn't matter.

And finally we have the triangle inequality, that the Hamming distance between x and z certainly can't be more than the Hamming distance between x and y plus the Hamming distance between y and z. If x differs from y in only n1 places, and y differs from z in only n2 places, then clearly z can't differ from x in more than n1 plus n2 places.

So check, check, check. This is a legitimate metric for defining a geometry on the space, and this is the one that we use on the space of all n-tuples. But notice it's not all like the Euclidean Distance.

Now when we have a linear code -- let's combine these things. When we have a linear code, we have a group property which is, let me write it this way. If we take any code word and add it to any other code word, that's in the code.

And furthermore, c plus c prime is not equal to c prime plus c single prime, c plus c single prime, because we can -- well, I'll finish it. Unless c prime equals c double prime.

Why is that? We can do subtraction, cancellation. Cancel c out from each side. So if we add c to c double prime, we're going to get a different result from adding c to c prime, if c prime and c double prime are different. So this implies that c plus C -- I write that as an abbreviation for the set of all c plus c prime, as a c prime runs through the code C. So this is the 2 to the k sums of the code plus any code word in the code. Sorry if I don't write down all steps.

What is that going to be? C. How did we conclude that?

AUDIENCE: [INAUDIBLE] [UNINTELLIGIBLE]

PROFESSOR: Perfect. Did you all hear that? By the group property, each one of these things is in the code. By this argument, no two of them are the same. That means I get 2 to the k distinct elements all in the code, that's got to be the code, because the code only has 2 to the k elements.

All right. So if I add a code word -- in other words, if I write down the code -- 0, 0, 0, 0, 1, 1, 1 0 1, 1 1 0- and I add any code word to it -- say I add 0 1 1 to the code. So let me just do one column of the addition table. I get 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1. I'll get the code itself again.

This has a very important property. The set of Hamming distances C and c prime, as c prime runs through c, from any given code word C, it is independent of C. So I can start from any code word, measure the Hamming distances, the 2 to the k Hamming distances to all other code words, including C itself -- let c prime run through the entire code, I'm going to get a set of 2 to the k distances called the distance profile of the code.

And I claim that it doesn't matter which code word I start from. I'm going to get the same distance profile regardless of where I start. In other words, the set of all distances from the all-0 word here, which is 0, 2, 2, 2, is the same as the set of all distances from the 0, 1, 1, code word, which is 2, 0, 2, 2. And the proof is basically that this is simply the Hamming weight of C plus c prime as c prime runs through C. What is this equal to? So the proof is that this is equal to the Hamming weight of c

prime as c prime runs through C.

So the distance profile from any code word is simply equal to the weight profile of the code itself. The weight profile of this code is 0, 2, 2, 2, Start from any code word, measure the distances to other code words, I'm always going to get 0, 2, 2, 2, 0, to itself, and the others.

Sounds sort of like the tetrahedron, doesn't it? It's zero distance from a code word to itself and equal distance to all the other code words, in that particular case. OK. So again, everything is very elementary here. The distance profile is independent of C and equal to the weight profile.

So this has an extremely important corollary. What's the minimum Hamming distance of the code? You might expect this is going to be an important code parameter.

AUDIENCE: [INAUDIBLE]

PROFESSOR: The minimum Hamming distance between any two code words is going to be equal to -- I think you said it.

AUDIENCE: The 0 [UNINTELLIGIBLE].

PROFESSOR: Non-zero is important here. If the distance profile is equal to the weight profile, one of the weights is always going to be zero. And that corresponds to the distance between a code word and itself.

All right. If I go through all the other 2 to the k minus 1 distances, they're going to be weights. They're going to be the distances from a code to all other code words. And the minimum distance is simply going to be the minimum non-zero weight of the code.

For example, in this code, the minimum Hamming distance between any two distinct code words is going to be equal to the minimum distance from the 0 code word -- that's another way of doing it. Since it's independent of C, we may as well take the base code word C to be zero. And then what's the minimum distance to 0? To the

0-code word? It's the minimum weight. From the 0 code word, distance equals weight. So the minimum distance is the minimum weight of any non-zero code word, which for this code is two.

Now here the weight profile is 0, 2, 2, 2, 0. The distance profile from any code word to all the others is 0, 2, 2,2 2. This is always the distance to itself. So minimum distance to other code words is always going to be two.

Furthermore, the number of nearest neighbors -- to go back and use chapter five terminology -- the number of nearest neighbors is going to be the number of code words that have that minimum weight -- in this case, three. Still sounding a lot like a tetrahedron, isn't it? This easy map between Hamming distance and Euclidean distance for this case and in general for all of our cases.

So corollary. The minimum Hamming distance, which implicitly means between two distinct code words of C, is equal to the minimum non-zero weight of C, and the number of minimum weight code words is independent -- I'm doing this backwards. From any code word, the number of code words that distance, let's call this d, is equal to the number of weight d code words. Sorry, you probably can't see that.

All right. So we get this symmetry property for codes that follows from the group property of the code that if we stand on any code word and look out, we're always going to see the same thing. We have this constant. It's actually easiest to see this when we make the map from the code to the Euclidean image of the code. So the Euclidean image S of C of the code word is going to be some set of 2 to the k vertices of an n-cube of side alpha.

Let's talk about the Euclidean image of these properties. If the minimum Hamming distance of the code is d, what's the minimum squared Euclidean distance between elements of S of C going to be? Well, let's do it coordinate by coordinates. Let's take two code words, C and c prime, let's say. And let's suppose we have some Hamming distance between C and c prime. That means that c and c prime differ in the Hamming distance, number of places.

So if we map this into the corresponding pair of vertices of the n-cube in Euclidean space, S of C and S of c prime, how many coordinates are these going to differ in? It's going to differ in same number of coordinates, D_h. If they don't differ, what's Euclidean squared distance in those coordinates? 0. If they do differ, the Euclidean squared distance is 4 alpha squared. So the Euclidean distance D_e between S of C and S of c prime is simply going to be 4 alpha squared times the Hamming distance between C and c prime, yes?

So I should say this is the squared Euclidean distance. Why do we always talk about the squared Euclidean distance? Because it's additive, coordinate-wise. And the Hamming distance is additive, coordinate-wise. So there's a nice easy map here.

So what does this mean d_min squared is going to be? d_min squared of, let's say, S of C. This constellation that we've formed by taking the Euclidean image of c. The minimum square distance between points in S of C is just going to be 4 alpha squared times d, where I don't think I ever -- d equals min Hamming distance. And we're always going to talk about n, k, d as the three key parameters of a binary linear block code. n is the code length, F2 to the n, k is the dimension, d is the minimum Hamming distance. So by going into this Hamming geometry, we've got a third key property of the code. And we see it's key, because we can get the minimum squared distance between this Euclidean image constellation, just 4 alpha squared d.

AUDIENCE: nd makes a probability of [UNINTELLIGIBLE] that is dependent on [UNINTELLIGIBLE] then.

PROFESSOR: Correct. This is all we need to know to get the union bound estimate. Well, a few more things. We need to know what K_min average of S of C. And what is that going to be? This is simply going to be the number of words in the code. To get this minimum squared distance, we need a Hamming distance of d.

So the number of words in the code of distance d, which is given by the parameter n sub d, is simply going to be the number of nearest neighbors. Not just the average distance, but I want to emphasize this symmetry property. If we stand on any point,

on any vertex of this cube in n-space, which is the code vertex, and we look at all the other points in the constellation, no matter which point we stand on, we will always see the same profile of distances. We'll see precisely nd code words at Euclidean distance 4 alpha squared d. We'll see nd plus 1 at Euclidean squared distance 4 alpha squared d plus 1, and so forth, right up the profile.

So there's complete symmetry in the constellation. In that universe, you don't know which code point you're standing on just by looking out, because the world looks the same to you. Is that clear?

OK. So from a communications point of view, this is important, because it means it doesn't matter what code word we send. The probability of error from any code word is going to be the same as the probability of error from any other code word, because the geometry is exactly the same. The Voronoi regions are all the same shape. So given the exact probability of error, not just the union-bound estimate, is going to be independent of which code word was sent.

This all follows from the fact that it's a linear code and therefore has the group property, which translates into this very strong geometrical uniformity property in Euclidean space. Or actually in Hamming space too, but it's more striking in Euclidean space. OK?

So we have everything we need to write down the union bound estimate. Union bound estimate was just the probably of error per bit is well approximated by K_b of constellation, in this case, S of C, times Q of the square root of the coding gain of the constellation times 2 Eb over N_0.