PROFESSOR: So OK, what did we do last time? Thanks for coming back by the way. So last time, we did MDS codes. MDS codes, and we looked at this. And we derived some generic properties of these things if they existed.

And then eventually, we moved to Reed-Solomon codes. And so indeed, they exist. We talked a little bit about existence altogether, the main conjection, MDS codes, I just wrote it down. And like I said, you want to be rich and famous, solve this. There's people doing geometry in math that would be deliriously happy if you solved it. So if for nothing else.

So the last thing I wrote down last time was something about a matrix which somebody recognized as some Fourier transform matrix. And that's where I want to pick up here, at least quickly make that connection. So let me rewrite the definition of Reed-Solomon codes again. And so a Reed-Solomon code, the way we defined it last time was all field elements, we would evaluate the polynomial in all field elements. Now we do it slightly different.

Let me make it Solomon-Stein in order to denote that difference. And let's say this is beta 0, beta n. So that's the old definition. Only the, let's say, the beta 0, beta 1, up to beta n, they are the non-zero elements now.

And just for the heck of it, so we talked about punctured Reed-Solomon codes last time that it would still be MDS codes. So nothing has changed there. In particular, all the arguments would be the same. In particular, just to humor me, let's say beta i is omega to the i, where omega is primitive in Fq. So what that means, you remember that primitive means that the powers of omega cycle through all the non-zero elements.

Remember the non-zero elements are a multiplicative group. And it's a cyclic group. And it's the generator of the group. Good. So once we write it like this, we can actually write down the whole Reed-Solomon code in some sort of transfer domain description in the Fourier transform. And the way this goes is the following.

So from now on, I will make implicit identification without making them. So this is an identification which I just jump around between. A vector -- so this one would be a Fqn. And Fx would be maybe 0 to MDS 1. Let's leave it at that. So I make this --

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, let's make this identification free. It's just a vector of length n and a polynomial of length n. And so whenever I want this to have a low degree, the polynomial, then I just write that expression.

So this identification we make freely now. And then how can we write F? Let F now be code word in the Reed-Solomon code. And then I say Fi. What is Fi? The i is element.

Well, we know that this is F of omega to the i. So this is, at the moment, counted from 0. So this is -- you write it out -- Fj omega to the i to the j, which is omega i j.

And now, at this point in time, everybody recognizes this. This would be the discrete Fourier transform of a vector. This is an element of order n. Usually, there's e to the j and then an element of order n. Now it's in the finite field.

But what's the difference? This would be the Fourier transform. So a code word would be the Fourier transform of a vector F. Let's just make sure it also goes the other direction.

So I claim that Fl would be -- so what would be the Fourier transform if it, indeed, is the Fourier transform? It is a minus, and then it goes n minus 1 Fj omega minus jl. So the question is is that true? Because that would be the inverse. Is that true? Well, let's do what we do with proofs of this type. I need that space here.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Sorry?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So let's do what we do with proofs of this type. That means we write it out. That's another beautiful thing about finite fields. You never have to worry about convergence. So for the F, we just plug that in here. Sum, this would be -- what do we call it -- Fi omega to the ij. So this would be Fj. And then I want omega minus jl. So is that true? Is the identity now true? Well, we see two sums. What do we do with two sums? If we're compelled to exchange the order, that's always what we do. So Fi goes out. And then you get omega to the -- what do we do -- i minus l times j.

We have this. And so what is this? How can we work this out? Well, you know how to express the sum.

So this part of the sum is really just omega i minus l to the n minus 1 minus 1. Well, that's a standard formula for summing these things up. But omega is an element of order n. Omega was an element of order n. That's how we chose it.

AUDIENCE: [INAUDIBLE].

PROFESSOR: It's what?

AUDIENCE: [INAUDIBLE].

PROFESSOR: You multiplied omega -- it's primitive in the field, so omega -- we have omega da-da-da-da-da i unequal to 1 for all i less than n and omega n is equal to 1. So OK. That's fine.

So it's an element of order n. That means the n-th power. We can interchange this. So this is 1 to the some power remains 1. So 1 minus 1 is 0.

The denominator -- well, actually, we get two results here. If i is unequal to l, then this is 0. If i is equal to l, then this was just the sum of n once. So we get n here. i equal to 0.

But n was q minus 1, the number of non-zero field elements because it was primitive in the field. And this one is just minus 1 in the field. Because q, we compute q is the power of the prime of p. So that would be 0. If we just come out as minus 1, that explains this minus here.

And we can get rid of the question mark. Look correct? All right. According to popular vote, it is.

So what do we really have here? We have everything we want from a Fourier transform. We have a Fourier transform pair. We have a vector f, which we can think of as being in the frequency domain. Then we have a vector F, which is a Fourier transform of that. This one would be in the code if the degree of this polynomial here would be less than k.

So now how can we understand Reed-Solomon codes now from an engineering point of view? Classical -- what about band-limited functions? What do we know about band-limited functions? Because in a sense, the degree less than k, this means that Fi is 0 for all i greater than k. That's what it means.

So in a sense, it's nothing but a band-limited function if this is a frequency domain, if you consider this as a frequency domain. So what do we know about the band-limited function? Well, if a function is band limited, it cannot be impulsive. We know that. It's a part of the frequency, and the frequency domain is inversely relational to the support of the function.

So the whole idea of Reed-Solomon codes, in a sense, can actually be understood, at least intuitively, from Fourier transform and the duality between time and frequency. Yeah.

AUDIENCE: Was q a prime number?

PROFESSOR: q is not prime. q is a power of a prime. q to the n.

AUDIENCE: [INAUDIBLE].

PROFESSOR: There always is. Oh, sorry.

AUDIENCE: [INAUDIBLE] or because for any q, there would be always a primitive?

PROFESSOR: Why there's always a primitive element in the field -- yeah, sorry. I thought you had gone through that. It's because the non-zero elements in a field always make up a multiplicative group. And it's always a cyclic multiplicative group. So it's a cyclic multiplicative group. And it's just the generator of that group.

AUDIENCE: [INAUDIBLE] it was a --

PROFESSOR: It's the same statement as saying, a cyclic group has a generator.

AUDIENCE: If Fq, for example, was a z, zq, then q has to be prime.

PROFESSOR: Yeah. Yeah, in order for it to be a field, q has to be prime. Right. That's true. That's true. But we are not worried about -- There's also a beautiful theory [UNINTELLIGIBLE PHRASE] rings if this would not be a prime, very nice theory, becomes a little bit more technical. You see, more technical, then more difficult.

I just wanted to give you this correspondence on the Fourier transform because it's, from an engineering point of view, a very nice insight. One of the reasons one can understand that Reed-Solomon codes have a good minimum distance is because it has a transform of a band-limited function. So that's what I wanted to say about the Fourier transforms here. I do not want to spend too much time. I have to get through decoding here.

Since we have now this Fourier transform correspondence, we can do another thing. We can, namely, say, well, F is in the code if and only if the Fourier transform -- so that was the back transform here. Also now, the code word, we can interpret the code word as a polynomial. And the inverse transform is just evaluating it at an omega again. F is in the code if and only the transform of a code word is 0 for all i greater than k up to n minus 1. Yeah.

AUDIENCE: [INAUDIBLE].

PROFESSOR: So it basically just means the support. If this one is band limited, then this one cannot have arbitrarily small support. That's what it means. That's really all I wanted to say.

i greater than k and less than n. So we have that F is in the code if and only the polynomial evaluates to 0 at a bunch of points. Now that leads to something interesting. Because now we can say, well, so what if we've just construct our set of vectors F so that they evaluate to 0 somewhere where we want them to zero. We construct them somewhat differently than here.

And in particular, let's define g as a product of i equal k up to n minus 1 of x minus omega minus i. So it's a polynomial. This is a polynomial which satisfies this. So in particular, we have g of omega minus i is equal to 0 for all i less than n less than -- I'm sorry -- and greater than k.

So this is a code word, right? We are agreed this is a code word? Good. So we found a code word without going through this here, without going through the evaluation map. We found it straight away.

But not only did we find this. This has degree n minus k. This has degree n minus k. So maybe there's another interpretation. Now we can give a whole new definition of the Reed-Solomon code, namely, as a set of polynomials, which are sort of implicitly identified with a set of vectors such that times this g of x and h of x is less then k again.

So what we have here, it's again, you take a vector space of functions the h. It's a k dimensional vector space. And we multiplied with this g. It's again a linear map, this multiplication with a fixed polynomial. A linear map of this vector space gives us, again, a vector space of dimension k. And all elements in this vector space evaluate to 0 for all omega to the minus i.

That's just the same again, just the Reed-Solomon code again. So it's a very nice complementary description to the same Reed-Solomon code we could define up here, we have found again down here. What is so nice about this? Yeah, what do you think is so nice about this? About this description? So now you have to think as engineers.

AUDIENCE: You can still evaluate it. There, we have to evaluate it at end points of function.

PROFESSOR: Yeah, exactly. This is so easy to implement.

So now we think a little bit about encoder. An encoder for a Reed-Solomon code. How would we do that? You look at this here. Hey, let's do this. Let's exactly do this.

Meaning in VLSI, we do something like -- and here, you would write -- and this is just a delay element. This one would be g n minus k. This one -- so these would be the coefficients.

This polynomial, we can evaluate once and for all. We do that once before we start our communication. We evaluate that polynomial once and for all. We know these coefficients. So these are multipliers which multiply with those corresponding coefficients. And we feed into here the coefficients of this polynomial. This is our information symbols, which we simply feed into this circuitry. And into this circuit, out comes F, out comes the coefficient. Polynomial multiplication.

If you show that to a VLSI designer, they are deliriously happy. They say, you know what? I implement you this thing. I implement you in 5,000 gates, which is close to nothing nowadays. Maybe not. Maybe 10,000.

And there you get to the second reason that Reed-Solomon codes are so extremely important in practice. On the one hand, they are MDS codes, meaning they are about as good as it gets. And on the other hand, they are algorithms. They are circuitry for these things whose cost is close to nothing at least today.

When they were invented, that was quite different. In the '60s, that would have not been implementable with transistors on a board or something like that. But today, the cost is close to nothing. So the whole algorithmic treatment of Reed-Solomon codes is very well developed. The encoder you could do like this. Yeah.

AUDIENCE: Question about why do you still require that degree [INAUDIBLE]?

PROFESSOR: You don't need to do that. But then the mapping is not one-to-one anymore. Then the mapping is not one-to-one anymore. Let's put it other ways.

You want code words of length n. If the degree is larger, you run over. In order to still get a code word, then you have to take it modular x to the n minus 1. And that would fold the coefficients back. And still, it gives you a valid code word then, but this is not one-to-one anymore. Anyway, the short answer is if you allow more, then they become longer than n.

So what's the time? OK. So this is a nice encoder. There even is a nicer encoder, which I just want to give the formula for. The one reason that people still have a problem with this is it's not systematic. People like to see the symbols in the code words themselves. They want the systematic part.

How could we achieve that? Well, we go from the same description here. We say, well, let h of x be given. Then compute x to the n minus k times h of x modular g of x.

So we divide this polynomial by this polynomial g -- it has a name. It's called the generator polynomial. We divide it by g. And out comes some polynomial r of x of some low degree, degree less than g. And degree r of x less than n minus k in particular.

So and then we can form a code word. I claim F of x, no, F is -- and here we write r, and here we write h. The coefficient vector of h and the coefficient vector of r. Why is that a code word? Why is that a code word? Maybe minus here. I'm always thinking characteristic 2 anyway. So why is this a code word? Anybody, it's clear?

What this is in terms of F of x is r of x plus h of x. Oh, minus r of x. That's because we wrote it like this. If we now take the F of x modular g of x, well, this part has degree low. It's not affected by this modular operation.

This would be minus r of x plus this one, modular g of x, which is r of x. So the whole thing is 0. If this is 0, that means g of x is a factor of F. Hence, it's a code word.

So we have a code word here. And in the code word, pop up our symbols right away, our encoded information symbols right away. So we get some nice systematic encoding going.

This division circuit by g is pretty much the same size as this. It's not larger at all. So there, we get beautiful algorithms. This is actually what's implemented. If we go to any disc drive [UNINTELLIGIBLE], that is usually what is implemented in there, exactly this.

Algorithms. Algorithmic treatment of Reed-Solomon codes. Do you have any questions about any of this? Yeah.

AUDIENCE: I have a question about do any these Reed-Solomon codes map [INAUDIBLE]?

PROFESSOR: They are very costly map. Usually, it depends a lot on the application. If you think disc drives, [UNINTELLIGIBLE] as just being mapped, you take Reed-Solomon codes over a characteristic 2. Then each field element is represented as a binary vector. And that binary vector is mapped into on-off keying. Straight off the bat. Nothing more fancy. That is for disc drives.

And as you do this, the satellite standard, where it's mapped onto the 256-QAM field elements [UNINTELLIGIBLE]. So it's many different ways. Many different ways.

AUDIENCE: But to prove some performance [UNINTELLIGIBLE] in the [UNINTELLIGIBLE], we need to have mappings, right?

PROFESSOR: In order to prove performance mode, we need to have mappings. And the Hamming distance bounds that we get from here give you bounds on the minimum Euclidean distance. If these are good bounds or not depends a lot on the modulation scheme.

And to be perfectly honest, they usually are not. They usually are not very good, the bounds. But it's a very difficult problem to design a code or to find a representation of the fields that maps nicely onto a modulation scheme. Very difficult problem.

AUDIENCE: How do we know how to think that this is a good code?

PROFESSOR: The code itself is excellent in terms of MDS, the MDS property.

AUDIENCE: But why does MDS mean good codes?

PROFESSOR: In respect to modulation?

AUDIENCE: Yes.

PROFESSOR: It doesn't. It doesn't. It's a bit like this. It's really not easy to define codes in Euclidean space. So all that we do is we find ways to do that and guarantee some performance, some sort of performance.

It's not easy to spread out -- I don't know -- 2 to the 1,000 points in 1,500 dimensions. These would be typical numbers really. It's not easy. And since that problem is practically daunting, it's a daunting task, you have to develop all sort of crutches to do that. And this is really one.

So how to code MDS codes playing a part in this mapping. If you want to be more fancy about that, then you put a convolutional code or some other code also in there and do a combine scheme. I think Professor Forney will talk about that more. So here, it's just the coding theoretic groundwork of these things. Anything else? Yeah.

AUDIENCE: [INAUDIBLE] of the [INAUDIBLE] n minus 1. Close, very close [INAUDIBLE].

PROFESSOR: Yeah, thanks. So the algorithmic treatment of Reed-Solomon codes is extremely elegant. And that's the second main reason they are so much used. It doesn't cost so much to implement them. Well, at least we've seen that for the encoder. That's a fairly small circuit.

So what about decoding? Decoding. How do we decode these things? How could we possibly decode them? And I give you --

AUDIENCE: Fourier?

PROFESSOR: Right, that's true. We could do the Fourier transform. But it doesn't help us so much because we receive something. And we receive a vector, say, y. From now on, let's say x is a code word. rs, right.

So x, it's usually a code word from now on. So this is a code word plus an error. So in particular, if we take the Fourier transform, we take the Fourier transform of the code word, which is fine. But then we get the Fourier transform of the error. So that destroys all the fun.

What else could we do? Here's typical parameters of a code. 255, 239, 256. And you immediately see that, OK, any sort of group force is out.

How many code words do we have here? We have 8 to the 239 code words. Now you don't want to search that. You definitely don't want to search that.

So how could we possibly decode these things? Turns out, to decode them in some sort of optimal fashion, maximum likelihood [INAUDIBLE] actually, it's an NP-hard problem. Maybe last year, actually, it has been shown that it's an NP-hard problem to decode Reed-Solomon codes. It was known that decoding in general was NP-hard. But this is now the constraint to Reed-Solomon codes is still hard.

So what do we do? Yes, OK, what do we do? Let's say x is a code word. We know that. And set rate e, let's just call it t.

So what happens if you don't have an error? Just thought experiment. Thought experiment, if you don't have an error, then yi in all the positions is equal to F of xi for some F of x of degree. In particular, since we know the x's, we know the positions -- sorry, oh, sorry. That's not what I wanted to write. This is definitely not what I wanted to write.

Let's keep that as c as a code word and position i in c is associated with xi in the fields. So meaning ci would be F of xi. That's really what I wanted to write. It makes more sense.

So thought experiment. No errors. Then yi is F of xi for some F. If they ran no errors, then we could just solve this linear system of equations to find the coefficient of F. And the coefficient of F, say, were our information circuits. Is it clear that this is a linear system of equations? Yeah?

You could write it out as y0, y1, y2 equal to -- and here we have f0, f1, up to fk-1. This is the linear system of equations. We know the xi's. This is a linear system of equation we have to solve.

If there are no errors, then life is easy. That seems to be reasonable. So what happens if we do have errors? Somehow, we have to make sure that the errors that we get do not cause any problem for us. And then we do something very ingenious.

We define something called an error locator which is a polynomial x such that x minus xi. So it's a polynomial which is 0 in all error positions. Well, you might say, we do not know the error positions. Well, OK, that's true. Basically, this is, in the end, what we want to find, this polynomial. But nonetheless, this polynomial exists.

We can cast, actually, the coding problem -- this is form 1 s. Yes, what?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: Yeah, it's an additive error model. But you can cast pretty much anything in the [UNINTELLIGIBLE]. If a position is altered, you can always model that as if something was added to it.

Decoding problem one is find lambda of x of minimal degree such that lambda of xi, is 0 for all xi and degree f less than k. So I claim if you solve this problem, namely, this is a problem I give you.

I give you vector y. I give you vector y. Here it is. Here is vector y. And I give you vector x which corresponds to the field elements where you evaluated that in order to get the code word.

And then I said, given y and x, find two polynomials lambda and f such that f has maximum degree k and lambda has minimum degree, the smaller degree possible so that this is true. And I claim this solves the decoding problem. This would solve the decoding problem because once we have found this, then we can take lambda to be the error locator. And we can basically read off the information symbols from the f.

So this decoding formulation now brings it, at least, into the algebraic ground, brings the whole decoding problem, makes it something algebraically. But now the question becomes, is that easy? Or can we do this? This problem here.

Do you see any hope for solving this problem? I guess the only answer that -- OK, anybody says no? Anybody does not see any hope? All right. This is great. You all see hope here. You all see hope here, which seems to make you an opportunistic bunch. Not opportunistic. What's the word? Optimistic. Optimistic bunch.

Let's put it like this. What is the problem in solving this? It's not linear. You get the coefficients of lambda. Multiply the coefficients of f. That whole thing becomes a nonlinear problem, where we say in the end, find the solution to a set of polynomial equations in a field, which is a multivariate polynomial equations, where the coefficients of lambda and f are the variables.

You can do that. You could use techniques like Grobner basis or so, and you could do that. But this is very difficult. This is computationally tedious.

So is that clear why this is nonlinear, and why this is hard to solve a nonlinear problem here? If not, then you have to say something now, or forever hold your peace.

So what do we do with hard problems? Once you take the optimization classes, there's almost like a reflex, there is a relax time. We find the proper relaxation of the problem. Now the proper relaxation of this problem is the following.

Decoding problem two. Find lambda of x of minimal degree such that -- it's almost the same, almost the same -- lambda fi yi minus h of xi is 0, where the degree of h is less than k plus degree lambda.

So all that we did from this formulation, which would give us a clean solution to the whole thing, right now, we multiply in this lambda which gives here, it keeps the lambda times y. And here, we get a new polynomial, lambda times f, which now has degree at most k plus degree lambda. And then we say, let's instead solve this problem. Let's solve this problem.

In particular, the question now becomes well, we do not require this anymore. In this relaxed formulation, we do not require this. And that makes all the difference. It makes a world of difference because this one -- look at it -- it's a linear problem. It's a linear program.

Why is it a linear problem? Do you see it's a linear problem? Could you write down the equation, the matrix equation? It's pretty straight, right? It's pretty straight.

You could, for example, write it like, here, a diagonal matrix yn. Here, you would get something. I think that seems to be all right. And here, just the evaluation of the h. So up to hk plus degree lambda.

That's a linear system of equations. We can certainly solve this. Well, we do not know really what the lambda is, what degree the lambda has. But we're just hypothesizing on all the possible degrees.

Well, we could say, OK, let's assume the degree lambda is 0. It's a constant, which would be the same as saying there are no errors. Then we can look at the system of equations. Does it have a solution? Well, yes, no. If no, then we say, all right, let's assume it's 1. Well, does it have a solution? Yes, no. And so on. Then we can move on.

So we can solve this relaxed problem. Does that help us? It's nice to solve a relaxed problem. And in the end, we get two vectors out of it, two polynomials, lambda and h. Does it really help us?

Well, yes. Why? Because -- let's put it like this. We could easily check if this is true. Once we have our h, we can easily check if this is true. And if it is true, then we have solved this problem. If it is true, we have solved this problem, which is the problem we wanted to solve. Good.

So is that all we need to know about this? We want to give guarantees. We want to give guarantees that we correct up to so many errors, t errors, right? So we have to guarantee that if there are not more than t errors, whatever t will turn out to be, we can guarantee that A, we will find a solution, and B, this will hold. So two things to prove. Is that clear? That we have to prove those two things?

So the first one first. When do we find a solution? And are we guaranteed to find it? Can we guarantee the existence of a solution?

Well, when can we do that? Let's look at this. These are n constraints. This is a system of linear equations with n constraints. If the total number of degrees of freedom exceeds n, then we are left with something non-trivial after we solve the system of equations.

So what's the total number of degrees of freedom? The number of degrees of freedom -- so we get the lambda as a degree of freedom, so which is degree lambda plus 1. And the other one is plus -- what is this one -- plus k plus degree lambda. This is the total number of degrees of freedom here. And the reason is that this one should be minus 1. Sorry.

So this is total number of degrees of freedom. If this is greater than n, then we can guarantee the existence of a solution, a nontrivial solution. Then this thing will have a solution. It's a homogeneous system of equations. So the 0 is always a solution. But that's not much good to us.

So what do we get here? Degree lambda. To a degree lambda greater than n minus k plus minus 1 or n minus k. So the degree lambda greater than n minus k over 2 -- does that remind you of anything here? So this one is d minus 1 over 2.

So very interesting, right? Once upon a time, you learned that if you make less than d/2 errors, you can correct that. Very nice. It pops up here. It pops up here out of the blue. The reason that it pops up here is, of course, the same statement, that as long as we stay within d/2 errors, we are guaranteed to be able to decode this. So this is one.

So we know this one was the number of errors. If t is greater or equal than this, then -- let's forget about the t. If you proceed in this algorithm hypothesizing the degrees of lambda, once we've reached this number in the degree, there will be a solution. And we don't have to go further than that. So that's the first one.

So the second thing we have to prove is that -- I shouldn't have done this -- that for some t -- Yeah?

AUDIENCE: [INAUDIBLE] standard degree of freedom or less constraints?

PROFESSOR: You want to guarantee a solution. If you have more constraints than degrees of freedom, then, since it's homogeneous, you usually would be stuck with a zero solution. Everything's 0, which is no good to us. It solves this problem, but it doesn't give us any information. Is that all right?

AUDIENCE: Why does it have to be the condition that degrees of freedom has to be greater than the constraints?

PROFESSOR: No, no, the degrees of freedom -- well, OK, it's true. We could get lucky. We could have redundancy in this system of equations. And if we get lucky, that's just for the better.

And actually, you can show that you do get lucky if very few errors happen. Then this would have low rank. You can show that. But short of knowing much, we want to guarantee -- I just want, at this point in time, to guarantee that there is a solution. There is a non-zero solution to this system of equations, and the degree of lambda is not more than d minus 1 over 2. There is a solution with a degree of lambda being upper bounded by this.

AUDIENCE: So [INAUDIBLE] less constraints?

PROFESSOR: Yeah, no, no. Once this is satisfied, I guarantee you there is a solution. That means, the smallest solution is guaranteed not to be larger than that. But there are, of course, many, many more larger solutions. There are many solutions of larger degree here, which we are not interested in since we are interested in the minimal degree. So this is an upper bound on the minimal degree lambda.

So now the other one. So now we want to make the statement about this one here. When can we guarantee that our relaxation didn't matter? Despite our relaxation, our solution that we find that in the solution that we find lambda divides h. How can we guarantee this? So how can we guarantee this? How can we guarantee this?

Let's look at lambda xi yi minus h of xi. And we know this is 0 for all xi. We know that.

We know that this guy here can actually be written as ci plus ei minus h of xi. This is just expanding this guy. We also know that we can write lambda xi times ci alone minus f of xi lambda xi. This is 0. We know that this is true too.

AUDIENCE: For some f.

PROFESSOR: For some f. The text is to f, so that this is true too. Then let's just subtract these two guys. Let's just subtract these two things here. And then we know that lambda xi times ei -- so the first two cancel -- minus h of xi minus So we know this is true too.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, sorry. Just in time. So what can we learn from this? So what is the degree of this? What is the degree of this?

Well, this guy here -- we'll write it somewhere else -- well, this guy had degree h was less than n minus k plus degree lambda less than k plus degree lambda. That's because we set up that problem that way.

And we know that this guy degree lambda. We know that. So this whole thing, let's call it S. We know it's 0 for degree S less than degree lambda.

So why does that help us? Well, let's look at the vector. Now let's look at all the positions. Let's look at the vector where we had, in the vector, we have lambda xi ei.

And now the fundamental question. What is the rate of this vector? At most? What's the rate of this guy at most? It's 12 seconds, [UNINTELLIGIBLE]? Well, the rate of this vector -- it's definitely not more than the rate of the error. Because every time the error is 0, the rate drops out. That's at most t.

What is the rate of this vector? So here we have a vector on the other side, S of xi, a vector on the other side. What's the rate of this guy? That's just the polynomial of degree at most k plus degree lambda minus 1. So we have that the rate of this vector is n minus k plus degree lambda minus 1 plus 1 because of the minus.

AUDIENCE: Greater than or equal to?

PROFESSOR: It's greater than or equal, sorry. Otherwise, I would have been in trouble in a second here. Yeah, so what does that mean? So actually, this one is nice to rewrite is equal to d n minus t plus 1 d -- I really should have done it like this in order not to -- it's d minus degree lambda.

So what can we learn from that? So we know this is true for all the positions. So if this is true for all positions, then -- so if d minus degree lambda -- actually, this is t because the rate was defined. t was the number of errors. And the error locator was just defined to have degree t. Right?

The error locator was just x minus xi over all positions where we had an error, so there are t of those positions. The degree of lambda is equal to t. So now we take this. If d minus t is greater than t, so if the rate of this vector is greater than the rate of this vector, then we have a problem because that cannot be. Of course, by this identity, the rate of these two vectors is equal. So what is the way out? What's the way out?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. So if this is true, then S of xi I claim has to be 0 and lambda of xi times ei has to be 0 too for all i. That's the only way out of this. Because, obviously, this was allowed.

If we really did find what we wanted to find, namely, an error locator here, then this vector -- we said it has to be less than t. It can be less. In particular, it can be 0 if this is, indeed, an error locator. And if our relaxation didn't matter -- that means the h that we find factors as that -- if the relaxation didn't matter, then S would be 0 too. So that is a fair solution. That's a possibility.

And if d minus t is greater than t, then this proves this is the only solution that is feasible. That's the only solution you have. So what that means is if t less than t/2, then the relaxation doesn't matter and h of xi is equal to lambda of xi times f of xi.

So is that clear? That's two very simple, yet nice ways to prove things. Here, we guarantee the solution. Here, we guarantee the solution. Here, we guarantee that solution is correct, namely, if not too many errors happen, if less than t/2 errors happens, then relaxing the original problem here to this other form, which we could solve, does not change the solution. That's what's proved there.

But about one-third looks puzzled. About half looks puzzled, I would think. Half looks puzzled. Is there any way I can explain that better? Think.

Let's just go through the steps quickly. We had problem number one. Problem number one, which is a nonlinear problem, but you see that if you could solve this, you could solve the decoding problem. So you see that. Anybody who doesn't see that? All right. You're smart guys.

If we can solve problem number one, we are home free. We have solved the decoding problem. Fine, we cannot solve it. It's a nonlinear problem. Well, we can solve it in exponential time, but that's little fun. So we relax it.

We relax it into problem number two. All that we do, we multiply things out. And we do not require, once we solve the problem, this anymore. Now we have solved this problem. We have solved this linear system of equations. And after doing so, we have found lambda and h. They are lying on the table and looking at us.

Now what? Are they any good? In particular, since a priori we cannot guarantee that this relaxation didn't completely destroy everything. We have solved the system. Now those two things are lying on the table, looking at us, and asking, what am I? And in general, if there's an arbitrary amount of errors happening near the channel, actually, they do not mean much. They do not mean much.

But now I claim that, well, if not too many errors happened, but if the number of errors is bounded to be this, less than half the minimum distance, then, actually, I claim it did not matter if we solved the relaxation or the original problem. And the argument is roughly this. It goes like this.

Here, we start. We have solved this. We have two polynomials lambda and h, so that this is true for all xi. We know that we can write it like this. That's just by definition of yi, just expanding the yi into this.

We know that this is true by the definition of the code. The ci is the evaluation of f, of some polynomial f. So we can write this just by definition of the code. So once we have these two guys here, we can subtract them [UNINTELLIGIBLE] here.

So we know that we have solved the following problem. We have found lambda and h. So that there is guaranteed to exist the polynomial f, so that this whole thing, namely, S of xi, that S is a polynomial of degree at most -- what did we have -- k plus degree lambda minus 1. So that's a semi-hairy step here.

Is that clear that we can guarantee the existence of a polynomial S of degree at most k plus degree lambda minus 1? So that this equation holds. Once we have solved this, we can guarantee the existence of this. That's what we're saying. We can guarantee the existence of this.

So now look at this. If you write this out for all i's and put it in a vector, what's the rate of this vector? Well, it's at most t because all the other e's are 0. If lambda is an error locator, it is 0. But we do not know that yet. All we know, it's at most t.

This is on this side, so that vector has a rate at most t. On this side, the rate of this vector is at least this. Just by evaluating a polynomial of this degree, this is the maximum number of 0's we can get.

So now we have two equalities, this one and this one. But obviously, the two vectors must be the same. So how can these two vectors be the same and still satisfy these two inequalities? Good question. How can they be the same and still satisfy this inequality?

If t is less than d/2, then the rate of this vector would have to be larger than the rate of this. If they are non-zero. If they are non-zero, this would have to be larger than this because then d minus t would be greater than t. If this is true, this implies this. If this is true, this implies this which implies that the rate of this vector would be larger than the rate of this vector if they are non-zero.

But how can that be? Answer is cannot. Cannot be, hence, they must be 0. Both vectors must be 0 completely, which means this is an error locator.

Because otherwise, this wouldn't be 0, and this one -- where did I write it? Somewhere I wrote it. Where did I write it? Oh, here. So this S has to be 0 too. If this S is 0, then h of xi is exactly this, means factors in the way you want it to.

So that's all I can say. I could say it again, but it's exactly the same words. And there's a limited benefit to even the repetition code.

So beautiful, right? We have brought down the entire decoding problem for Reed-Solomon codes to solving a linear system of equations, namely, this one here or this one in short form, which, well, it's no problem at all in the grand scheme of things. And that's the other thing that makes Reed-Solomon codes so beautiful. They're access encoding, they're access decoding, they're everything that we want.

Let me do one more thing about the decoding just to bring it down a little bit, to talk a little bit about complexity. Solving this linear system of equations has been subject to research for 30 years, 40 years. So it started out with an algorithm which essentially solved that linear system directly. It was formulated a bit different, but that's what it is. It's called the Peterson-Gorenstein-Zierler algorithm. They realized it was a polynomial time decoding algorithm, a nice decoding algorithm.

Then Berlekamp -- I should write down the name. Berlekamp came up with a fast algorithm and square algorithm. Massey had his own formulation of that algorithm, which was a bit more streamlined I think. Then there was a later version, Berlekamp-Welch. The complexity of these algorithms is all roughly the same, is all n squared roughly, which solved this here.

And then there's roughly nothing happening for 30 years. And after 30 years, then Madhu Sudan made the next serious dent in the decoding history of Reed-Solomon codes. So he found a way to solve this problem even beyond half the minimum distance. And in hindsight, it's a very nice and very simple trick that he used.

So we started with the following. We started that, well, if we have no errors, then the pairs of points xi, yi lie on this curve. That's another way to formulate what we said that yi minus f of xi has to be 0. We said, well, if there are no errors, all the points that we receive lie on this curve. Because there are errors, we have to multiply this with an error locator and say this is 0 for all xi, yi. So this was problem number one. The relaxed form was lambda xy minus h of x 0 xi, yi.

That's the way we do it. So what did Madhu do? Very clever. He said, all that we have to do is annihilate the effect of the error right in this. The whole reason for that lambda was to annihilate -- so to take out the error influence out of this interpolation formula, to allow for a few errors to happen, which we can put in lambda.

Well, Madhu said, well, what about we take a polynomial in two variables? A polynomial in two variables, then same thing. If no errors happen, then all the points xi, yi will lie on that curve. If a few errors happen, then let's find a lambda xy, two variables of some minimal degree, some very small degree so that this is still satisfied.

And the problem is entirely the same. And this one, you cannot solve either. You cannot solve either. But again, you can solve the relaxation minus -- again, you can solve the relaxation. This is, again, a linear system of equations. And the coefficients of lambda and psi now.

Once you solve this linear system of equations -- actually, this one you can more handily write simply -- now there is no way to distinguish the y anymore since both sides anyway depend on y. Find a polynomial in two variables such that this is 0 for all xi, yi. This is his central problem. This is his relaxation. Find a polynomial in two variables, which he evaluates to 0.

And then he has, essentially, the same proofs we have here, a bit more technical, but not much. I'm sure you can come up with this if you sit down at home. Let's put it like this. There's no heavy machinery in it. There's no heavy math in it. There's a lot of being clever in it, but there's no heavy math in it.

He can now guarantee that q of xy, if t is less than n minus square root of 2k -- is that right -- 2kn, then you can guarantee that y minus fx is a factor q of xy. The same thing we wanted to guarantee. And like I said, the proof is very clever, but no heavy machinery in it. It's no heavy algebraic geometry or any of this stuff in that. It's high school algebra. Well, freshman college algebra.

So that's what happens here. It took 30 years and a computer scientist to solve that problem. That was not all I wanted to say, but that's all I have time for. There's one minute left, so there's no point in starting something now.

Do you have any questions about any of this? I should say that this 2 one can get rid of, but that takes a little bit more machinery. The 2 one can get rid of, but that's a bit more heavy. All right. So thanks so much. That's it. That's it.