ALAN EDELMAN: Hi, everybody. I'm Alan Edelman. And helped a little bit to teach this class last year. But happy to see that it's going great this year.

So Professor Strang came to teach 18.06. Some of you may know the introductory linear algebra course. And Professor Strang came by and gave this great demonstration about the row rank equals the column rank. And I'm wondering if you did that in this class at any time. Or would they have seen that?

AUDIENCE: It's in the notes.

ALAN EDELMAN: It's in the notes. Well, in any event, just as Professor Strang walked out-- so here, I'll just grab this. This is true. I was actually going to start writing the code to do the quadrilateral, but I didn't have enough time. You can see me starting. But here's the 0.5 for the triangle, which was easy. So what's the story with Julia in this class? Have they used it a little, or a lot, or--

AUDIENCE: In the labs.

ALAN EDELMAN: In the labs you've used Julia. But that was just MATLAB. So that was-- but OK. So Professor Strang showed this proof where he would-- he put down a 3 by 3 matrix. It had rank two. And he took the columns that were-- the first two columns were independent. And it was easy to show the row rank equals the column rank.

After Professor Strang went out, I asked, would that work for the zero matrix? So here's the zero matrix. And since I'm not really telling you the proof, I'll just say, if I were to make a matrix of the literally independent columns of this matrix, what would I do?

So zero might seem tricky. It's not really that tricky. But this is what I did the moment you walked out. So yeah, so I've got the 3 by 3 matrix, and I need to make a-- first step in whatever the proof was, I needed to take the columns, the literally independent columns of this matrix, and place them in a matrix of their own.

What would I do? It would be an empty matrix. What would be the size of this empty matrix? Not exactly zero by zero. Because where every column is still in our 3, you see. So the right answer-- I hope this makes sense-- is a 3 by 0 empty matrix. And that's a concept that exists in MATLAB, and in Julia, and in Python, I think, I'm sure, and in any computer language.

So if you had a full rank 3 by 3 matrix, the linear independent columns would be 3 by 3. If you had a rank two, it would be 3 by 2. If I had a rank one matrix, it would be 3 by 1. So if I had no columns, 3 by 0 makes sense.

And to finish the proof, again, I'm not telling these students-- it's in the notes apparently-- the next matrix would be random 0, 3. And of course you multiply it and you get a 3 by 3 matrix of zeros. So it's fun to see that that proof still works, even for the zero matrix, without any real edge. So that was just today.

The other thing is let me-- can I say a word or two about recent stuff about Julia? So I started to put together a talk. It's not really ready yet, but I'll share it with you anyway. So Google did the Julia world a big favor last week. I mean, this is huge.

So you all know machine learning is hot. That's probably why you're here in this class. I probably don't have to tell you. And yet I wouldn't be surprised if a number of you wished this whole class was in Python or something, or maybe MATLAB or something. I don't doubt that some of you might have wanted that to happen.

And we get sort of bombarded with, you know, why not Python. Not so much MATLAB anymore. But you know, why not Python is sort of the issue that comes up a lot.

And I could talk till the cows come home, but nobody would believe me. But Google came out last week and said that when it comes to machine learning, there really are two languages that are powerful enough to do machine learning-- to do machine learning kinds of things that you want to do. And in some sense, the rest of today's lecture that I'm going to give will be maybe illustrations of this.

But if you want, you can go and look at their blog here. What they do is they basically sort of start the race with a whole bunch of programming languages. There's Python. There's R. Java, JavaScript. They sort of look at all of these languages.

And if you read the blog, you'll see. But we're going to filter them out on technical merits. And right away, a lot of them disappear, including Python and Java.

And if you go to the blog, you'll see they spend a great deal of time on the Python story. Because they know that people are going to want to hear that one. I mean, people want to be convinced. And so there's actually multiple screens full on the reason why Python is just not good enough for machine learning.

So they leave four languages left-- Julia, Swift, C++, and Rust. And then if you go to the next part of the blog, they filter on usability. And then two more sort of bite the dust. So C++ and Rust disappeared.

And then they go on to say that these are the only two languages they feel are appropriate for machine learning. And they put this nice quote that it shares many common values. And they actually go on about what machine learning really needs. And I'd recommend you look at it.

And then finally, of course, they're going to push Swift, which they should. So they had somewhere-- blah, blah-- about more people are using Swift. Maybe it's true. I don't know. So they really said is they're more familiar with Swift than Julia, which, you know, if I was speaking, I'd say I'm more familiar with Julia than Swift. So maybe it's fair.

And then I started to put a little cartoon on the psychology of programming languages, just because it's sort of something that I bump into with all the time. People always say all languages are equally good. It doesn't really matter.

But the truth is if you mention a language that you're not using yet, you're going to tune it out, at least until Google comes along. So that that's where we are. OK, enough about-- I just put this together. I was testing it out on you. All right.

So now let me do two more mathematical things. So the first thing I want to do is talk to you about forward mode automatic differentiation. So have you done any automatic differentiation in this course?

AUDIENCE: Very little.

ALAN EDELMAN: OK, so I think this is pretty fun. I hope you'll like it. I have a notebook in Julia on forward mode automatic differentiation. And this notebook came together because I was trying to understand what the big deal was for a long time.

And I had a little trouble. I mean, it's the usual story where on a line-by-line level, it's easy to understand. But what's the big deal part? That's sometimes the harder thing to grasp.

And the first notebook I'm going to show you is sort of the result of my trying to grasp what was the real picture here. And the second thing-- I think I'll just do it on the blackboard. It's not even really ready yet, but I'll sort of unleash it on you folks anyway-- is to show you how to do a particular example of backward mode automatic differentiation, the that you see in the neural net. And I guess you have seen some neural nets here?

AUDIENCE: Yep.

ALAN EDELMAN So I think by now everybody's seen neural nets. I think two years from now, it'll be in high schools. And three years from there, it will be in kindergarten. I don't know. Neural nets seem to be sort of-- they're not that hard to understand.

OK, so let me start things off. And really, the two things that I'd love to convince you of is-- let me just find-- here's my auto diff thing. The two things that I really want to convince you of-- and maybe you already believe some of this-- is one, that-- well, maybe you don't believe this yet-- but that the language matters in a mathematical sense.

The right computer language can do more for you than just take some algorithm on a blackboard and implement it. It could do much more. And this is something that I hope to give a few examples of.

And the other thing that I bet you all believe now, because you've been in this class, is that linear algebra is the basis for everything. Every course should start with linear algebra. I mean, to me, it feels like a unfortunate accident of history that linear algebra came too late for too many reasons.

And so very often, things that would be better done with linear algebra are not. And I mean, to me, it feels like doing physics without calculus. I just don't get it. I know high schools do that. But it just seems wrong.

To me, all of engineering, all of-- it should all be linear algebra. I mean, I just believe that-- almost all. Maybe not all. But quite a lot. More than most people realize, I would say.

OK, so let me start with automatic differentiation. So I'm going to start by this story by telling you that I would go to conferences. I would go to numerical analysis conferences. I would hear people talk about automatic differentiation.

I'm going to be honest. I was reading my email. I was tuned out. Like, who cares about-- I know calculus. You could teach a computer to do it. It seems pretty easy to me. I mean, I'm sure there are technical details. But it didn't seem that interesting to teach a computer to differentiate.

I sort of figured that it was the same calculus that I learned when I took a calculus class. You know, you memorize this table. You teach it to a computer. You learn the chain rule, and the product rule, and the quotient rule. And bump, the computer is doing just what I would do with paper and pencil. So big deal. I didn't pay attention.

And in any event, there was this little neuron in the back of my brain that said, hey, maybe I'm wrong. Maybe it's doing finite differences, you know, the sort of thing where you take the dy by the dx. In some numerical way, you do the finite differences.

And in numerical analysis, they're supposed to tell you if h is too big, you get truncation error. If you have h too small, you get round off error. And the truth is nobody ever tells you what's a good h. But you go to a numerical analysis class hoping somebody would tell you.

But in any event, so I thought maybe it was that kind of a numerical finite difference. And I think the big surprise for me was that automatic differentiation was neither the first nor the second thing, that there's actually a third thing, something different, that's neither the first or the second. And I found that fascinating.

And maybe I'll even tell you how it hit me in the head that this was the story. Because I really wasn't paying attention. But I love the singular value decomposition. I'm glad to see that people are drawing parabolas and quarter circles and figuring out what the minimum SVD value is. The singular value is just-- it's just God's gift to mankind. It's just a good factorization.

One of the things I was playing with with Julia was to calculate the Jacobian matrix for the SVD. So you know, all matrix factorizations are just changes of variables. So if you have a square matrix, n by n, the SVD-- I'm sure you know this-- is the U matrix is really n times n mass 1 over 2 variable. So is the V. And the sigma has got n variables.

Put it all together, you got n squared. So it's just a change of variables. And every time you change variables, you can form that big matrix, n squared by n squared of dy dx, compute its determinant, and get an answer.

And I wanted to know-- I actually knew the theoretical answer for that. And I wanted to see a computer confirm that theoretical answer. And I spoke to some people who wrote auto diff in non-Julia languages. And I was surprised by the answer.

They said, oh, yeah. We could teach the answer to our system. I said, what do you mean you could teach the answer?

Why doesn't it just compute the answer? Why do you have to teach the answer? I thought that was all wrong. Because in Julia, we didn't have to teach it. It would actually calculate it.

And then I started to understand a little bit more about what auto diff was doing and what Julia was doing. And so this is how this notebook came to be. So let me start-- I'm saying too much. Let me start with an example that might kind of hit home.

So I'm going to compute the square root of x, a real simple example. You know, a square root's pretty easy. I'm going to take one of the oldest algorithms known to mankind, the Babylonian square root algorithm.

It says start with a starting guess t. Maybe it's a little bit too low for the square root of x. Get x over t. So that would be too large. Go ahead and take the average, and repeat.

OK. This is equivalent to a Newton's method for taking the square root. And it's been known for millennia to mankind. So it's not the latest research, by any means, for computing square roots. But it works very effectively.

And here's a little Julia code that actually will implement it. It probably looks like code in any language. So I'm going to start off at 1.

So literally, I'm just going to take 1 plus the starting value of x and divide by 2. And then I'm going to repeat. OK? And we can check that the algorithm works.

Here's alpha is pi. And so I'll take the Babylonian algorithm and compare it to Julia's built in. And you see it gives the right answer.

Here it is with the square root of two. It's always good to check your code works. OK? I like to see things graphically, so I ran the algorithm for lots of values of x. And I love doing this.

I kind of wish that in the previous talk-- if I'd only worked fast enough, I wanted to build a little GUI where I can move the points in front of your eyes. Maybe you have one in MATLAB. I bet you do. But I wanted to build it. But I didn't get there fast enough.

But here this is the sort of thing. And I like to see the convergence. And so you could see the digits converging, the parabola on the bottom. The block is the square root, of course. So there it is. There's the Babylonian algorithm.

I would like to get the derivative of square root. But the rules of the game are I'm not going to type method 1 or method 2. I'm not going to do-- you'll never see me type 1/2 x to the minus 1/2. Right?

You all know that's the derivative. I will not type that. I will not. It's not going to come anywhere from Julia.

OK. And the second thing is I'm not going to do a finite difference. All right? I'm going to get that square root, but not by sort of either of the two things that I'm sure you would think of. Right? Here's how I'm going to do it. And I'm going to do a little bit of Julia code. There'll be eight lines of Julia.

But I'm not going to completely say how it works yet. I'll keep you in suspense for maybe about five minutes. And then I'll tell you how it works. All right?

So here's eight lines of Julia code that will get me the square root. So in these three lines, I'm going to create a Julia type. I'm going to call it a D for a dual number, which is a name that goes back at least a century, maybe more. So I'm going to create a D type.

And all this is is a pair of floats. So it's a tuple with a pair of floats. It's going to be some sort of numerical function and derivative pair.

So three of my eight lines is to create a D. In Julia language, this means to use a subtype of a number, so we're going to treat it like a number. Right? We want to be able to add, multiply, and divide these ordered pairs.

But it's just a pair of numbers. Don't let the Julia scare you. It's just a function derivative numerical pair. OK?

And what's these other five lines? Well, I want to teach it the sum rule and the quotient rule. So you all remember the same rule. I guess that's the easy one.

The quotient rule-- I still have my teacher from high school ringing in the back of my ear. The denominator times the derivative of the numerator and minus the numerator-- you all have that jingle in your brain, too? I bet you do. divided by the denominator squared. Can't even get it out of my head.

So there's the quotient rule. And so what are we doing in these five lines? Well, first of all, I want to overlook plus and divide and a few other things.

And Julia wants me to say, are you sure? So the way you say are you sure is that I'm going to import plus and divide. Because it would be dangerous to play with plus.

So here I'm going to plus two dual numbers. We're going to add the function and the derivatives. Divide two dual numbers. We're going to divide the function values and denominator times the numerator, blah, blah, blah, you get it. OK.

That's six of the eight lines. The seventh line is, if I have a dual number, I wanted to convert it. You know how the wheels are embedded in the complexes?

We have to tell Julia to take the dual number and stick a zero in. And then dual numbers and regular numbers can play nicely together. And this actually is the thing that actually says, if I have a dual number and a number in operation, promote them so they'll work as dual numbers-- so eight lines of code.

So the first thing I'm going to tell you is I'm going to remind you I never typed 1/2 x to the minus 1/2. Do you agree?

No one-- I'm not importing any packages. It's not like it's coming in from the-- I'm not sneaking it in from the side. There's no one half x to the minus 1/2.

And there's certainly not any numerical derivatives, either, right? Arguably, a rule that almost feels symbolic, the quotient rule and the addition rule. But no numerical finite differences at all here. OK.

So first of all, let me show you here that I'm applying the Babylonian algorithm without rewriting code to a dual number now. Before we applied it to numbers. But now I'm going to play it to this dual number that I just invented.

I'm going to apply it at 49, 1, because I know the answer. And then I'm going to compare it with-- I'm taking one half of the square root of x just for comparison purposes and not in my own algorithm. And of course, you see that I'm getting magically the right answer without ever- - so you should wonder, how did I do that? How did I get the derivative?

We could take any number you like. Here's 100. If you prefer to see a number like pi, we can do that. I mean, we can do whatever you like. It's going to work.

So there you see this is the square root of pi. And this would be 1/2 over the square root of pi numerically. So when you see it matches these numbers to enough digits, in fact, all the digits, actually.

Yeah. So the thing magically worked. You should all be wondering, how did that happen? I didn't rewrite any code. I actually wrote a code to just compute the square root.

I never wrote a code to compute the root of a square root. And by the way, this is a little bit of the Julia magic that we're pushing numerically. That very often in this world, people will write a code to do something, and then if you want to do something more, like get a derivative, somebody writes another code.

With Julia, very often, you can actually keep to the original code. And if you just use it properly and intelligently, you can do magic things without writing new codes. And you'll see this again in a little bit. But here's the derivative of-- this is the plot of 1/2 over the square root of x in black. And again, you could see the convergence over here. All right.

Well, I'm still not going to show you why it works just yet. I promise I will in just probably a few minutes more. But what I will do first is I'd like to show you something that most people will never look at. I never look at it.

I want to show you-- here's the same Babylonian code. I want to show you the assembler for the computation of the derivative. So I'm going to run Babylonian on a dual number. And we're going to look here.

And I don't know if anybody here reads assembler. I'm betting there is zero or one of you actually reads this stuff. How many of you read assembler? OK.

It wasn't 0, 1. We had a half. Right there's half.

He's kind of going like this. Here's zero. Here's one. He's like this. OK. So I think 0, 1 is like the record.

But I'll bet you'll believe me if I tell you that, when you have short assembler like this and it's not very long, then you have efficient code. It's very tight. It will run very fast. So whatever this thing is doing, it's short.

And this you won't get from any other language. If you did try to do the same thing in Python, I promise you there would be screens and screens and screens full of stuff, even if you could get it. So here's the Babylonian algorithm on the dual number. And here it is in assembler, and it's short.

So the other thing that I'm saying is not only does it work, but Julia also makes it efficient. So before I finally tell you what's really going on and why it works, I'm going to grab a Python symbolic package, which will work nicely with Julia. And I'm going to run the same code through the Python symbolic and show you what-- these are the iterations that you get.

So you actually see the iterations towards the square root. And here are the iterations of the derivative that's actually being calculated. And the key point here is, of course, this is a symbolic computation. We're not doing a symbolic computation.

This is mathematically equivalent to the function we would get if we were to, like, plot it or something. But of course, symbolic computation is very inefficient. I mean, you get these big coefficients.

I mean, look at this number. What is this? 5 million or something? Anyway, you get these big numbers, these even bigger numbers here. Look at these huge numbers, right?

It takes a lot of storage dragging these x's along. There's a big drag on memory. I mean, this is not the way that-- this is why we do numerical computation. But the Babylonian algorithm, in the absence of any round off, is equivalent to computing-- above the line, it's computing the square root here. And then below here, these are the iterates towards the derivative.

So it's not actually calculating 1/2 x to the minus 1/2. It's actually doing something iterative that is approximating 1/2 x to the minus 1/2. All right. Well, let me tell you now. Let me sort of reveal what's going on, just so that I can kind of show you how it's getting the answer.

And like I said, it was the SVD that sort of convinced me how this was happening. Because the SVD is also an iterative algorithm, like this Babylonian square root. But it's easier to show you the point with the Babylonian square root.

So I'm going to do something that I would never want to do, which is explicitly write a derivative Babylonian algorithm. And what I'm doing is I'm going to take the derivative in respect to x of every line on my code. So if every even or odd line-- I never know what's even or odd anymore.

But the original line of code had 1 plus x over 2. Now I'm going to take the derivative. I'll get a half. Here I had this line of code. If I take the derivative I'll, use the quotient rule, and this would be the derivative.

If I run this code, what I'm effectively doing is I'm just using good old plus and times and divide, nothing fancy. There's not a square root to be seen. But what I'm doing is, as I run my algorithm, I'm also running-- I'm actually computing the derivative as I go.

So if I have this infinite algorithm that's going to converge to the square roots, the derivative algorithm will converge to the derivative of the square roots. But I'm not using anything other than plus, minus, times, and divide to make that happen. So if you rewrite any code at all, you could have any code-- iterative, finite, it doesn't matter. If you just take the derivatives back to your variable of every line of your code, then you can get a derivative out.

And as I said, it's not a symbolic derivative, like, you know, all of 18.01, or whatever, wherever we teach calculus these days. And it's not a numerical derivative like in the numerical courses, the 18.3, axyz's, whatever. It's a different beast.

It's using the quotient rule and the addition rule at every step of the way to get the answer. Here's this dBabylonian algorithm. You could see it running. It gives the right answer.

Oop, I have to execute the code first to get the right answer. But if you see, it gives the right answer. Oh, I was just in Istanbul and they challenged me to do sine. I forget about that.

It's still in my notebook. I did it in front of everybody. It worked. I got a cosine. OK. But let me pass all of that.

So let me go back and tell you then how is this all working. Well, what's happening-- let's go back to the eight lines of code, and now, maybe, you can see what's happening. Where's my eight lines of code from the very beginning?

And I've got to watch the time. I want to show you this one other thing, too. So hopefully, I'll have enough time to do that.

But here, let's see. Where are my eight lines of code? Where are they? Here we go. Here are the eight lines of code.

So what I'm doing is, instead of rewriting all your code by taking the derivative of every line the human way, I'm saying that why can't the software just do this in some automatic way? And this is where the automatic differentiation comes in.

And in the old, old days, when people-- and all the numerical code was in Fortran, there would be the source to source translators that would actually input code and output derivatives of code. The Julia way, the more modern way, is to let the git compiler kind of do that for you.

So here, I needed plus and divide. Of course, I would want to add minus and times. But you just add a couple of things and then bump, you don't have to rewrite the dBabylonian. Because the Babylonian, with this type, will just do the work for you. OK?

And that's where the magic of a good piece of software will have it. So you don't have to write a translator. You don't have to hand write it. You just give the rules and you let the computer do it. Right? And that's what computers are supposed to be good at. So that's what's happening.

All right. So that's forward mode automatic differentiation. I've got 10 minutes to go backwards. But let me see if there's any-- anybody have any questions about this? It's really magic, right? But it's pretty wonderful magic.

And I don't know what you've heard about machine learning, but to be honest, machine learning these days, it's forgetting about whether humans will be useless, which I don't believe by the way. But the big thing about machine learning is that it's really just a big optimization. That's all it is, right? One big minimum maximum problem where you've all known from calculus that what you need to do is take derivatives. You know, set them to zero, right?

In the case of multivariate, it's a gradient. You set it to zero. And so really all of this machine learning, all the big stories and everything in the end comes down to automatic differentiation. It's sort of like the workhorse of the whole thing.

And so if we could have a language that gives you that workhorse in a good way, then machine learning really sort of benefits from that. So I hope you all see the big picture of machine learning. It really does come down to taking derivatives. That's the end-- that's how you optimize.

Any quick questions? Otherwise, I'm going to switch topics, and I'm going to move to the blackboard. Yeah?

AUDIENCE: Does the same thing happen for second order derivatives as well?

ALAN EDELMAN: There is a trick that basically lets you go to higher orders, yeah. You can basically make it a combo of two first order derivatives. So yeah, it can be done. Did you have a question?

AUDIENCE: Yeah. Is this notation of [INAUDIBLE], and is this only really used for computing different orders of derivatives or are there other examples?

ALAN EDELMAN: Well, for using types?

AUDIENCE: Or specifically, I guess, the way that you did through this whole presentation, just this generalized other--

ALAN EDELMAN: So it's the biggest trick in the world. It's not this little thing. The idea of making a type to do what you-- I mean, did you see Kronecker products in this class?

AUDIENCE: No.

ALAN EDELMAN: No?

AUDIENCE: [INAUDIBLE].

ALAN EDELMAN: OK. Let me see. What would you have seen in this? Did you see tridiagonal matrices, your favorite? OK. So here. So here's a built in type.

Let's say n is-- oh, n doesn't have to be 4. I'm going to create a strang matrix, if I could spell it right. And it's going to be a SymTridiagonal, which is a Julia type. And we will create two times ones of n and minus ones of n minus 1.

Here's a type. I mean, this is built in. But you could have created it yourself just as easily. And I don't like calling this-- it's certainly not a dense matrix. And I don't like calling it a sparse matrix.

I prefer to call it a structured matrix. Though the word sparse, it's a little tricky here. But the reason why I don't like to call this a sparse matrix is because we're not storing indices in any-- I mean, there a lot of fancy schemes for storing indices for sparse matrices.

Well, all we store is a diagonal vector. There's the 2s on the diagonal. There's this 4 vector with four twos.

And here's a three vector for the off diagonal. And you know, you don't have it twice, by the way. Most sparse matrix structures would have the minus vector twice, the super and the sub.

But really, only the core information that's needed is stored. And in a way, one uses types in Julia to basically-- you only store what you need, not more. And then you define your operations to work. So for example, if I were to take a strang inverse times, oh, anything, times a random 4. I'm going to do a linear solve.

You would want to use a special [INAUDIBLE] that knew that the matrix was a symmetric tridiagonal. So it's a big story of being able to create types and use them for your own purposes without any wastage. And this is the sort of thing that while you can do it in languages like Python, in MATLAB, if you were able the assembler-- and MATLAB would never let you, Python, you just would regret it-- but you would see just how much overhead there is in doing this.

So there would be no performance gain. But in a way, this is what you want to do. You want to use these things to match the mathematics. And so that's really the nice thing to be able to do. All right.

I only have five minutes. I don't know if I'm going to pull this off. But let me see if I could give you the main idea in five minutes of over immersed mode differentiations. But here, as long as you are familiar with neural networks, let me see if I can do this very quickly.

I'm going to start with scalars. OK? I'm going to do a neural network of all scalars. But only for simplicity, for starters, but I think you're going to see that this can generalize to vectors and matrices, which are real neural networks.

So what I'm going to do is I want to imagine that we have our inputs. We'll have a bunch of scalar weights and biases. So here's W1, and I'll go up to wn and bn. All right? So we have a bunch of weights and biases here.

OK? And we'll also have an x1, which will sort of start off our neural network. And we're going to compute-- I'll write it in sort of Julia-like or MATLAB-like notation, for i equals 1 through n. I will update x by taking some function of my current input, maybe something like this.

And what function h to use? I don't really care too much. In the old days, people used to talk about the sigmoid function.

Nowadays, it's the maximum of 0 and t that gets used all the time. It's got this ridiculous name RELU, which I really can't stand. But anyway, the rectified linear unit.

But in any event, I mean, it's just the function that's t of t is greater than or equal to 0. 0, if not. But whatever function you like. And here I'm just going to update. OK. And then ultimately, you might also have some data y.

And you would like to, if everything's a scalar, like I said, this could be generalized pretty quickly. But what we can do is we can minimize, say, 1/2 y minus xm squared. And you're going to want to find the data that would minimize that. All this generalizes to matrices and vectors, which is what most neural nets do. OK?

And since I'm not going to have a lot of time, maybe I can just sort of cut to the chase. If I were to differentiate the key line here, I got a little bit of Julia here. But if I were to differentiate the key line, what would I write? I would write-- well, here, actually, let me use the usual notation.

Let me have delta I be the h prime of wxi plus bi. OK? So that's delta i. And then you can see that the dxi plus 1 is delta i. And I'll have dwi xi plus dxi wi plus dbi would be the differential. This would be how-- so I'm almost done, that's the good news.

So if I make a little change-- I like to think of this as, like, 0.001 changes. I don't like infinitesimals. I like 0.001. That's how I think of them. But you make a little change here, a little change here, a little change here. You get a change here.

You'll get this linear this linear function of the perturbations here gives you perturbations here. OK? Well, I've only got one minute. So I'm going to write all this out with linear algebra, because everything is better when written out with linear algebra.

So I'm going to write down that-- I'm going to write down that I'm actually interested in the last element. But dx dn plus 1 is going to equal and I'm going to have a couple of matrices here. Let me just sort of get the structure right. This will dx2, dxn plus 1 again. Sorry for the squishing.

But here-- in fact, I'd like to use block matrices a little bit. So here I'm going to have dw1 db1. I'm going to put the bias together-- sorry for the mess. But dwn dbn.

And Julia lets you make block matrices. And you can actually use them directly. There'd be a special type right there. OK? And then what goes here you could actually see what it would be. It would be-- I hope I'm doing it right. But there'd be a delta 1x1 and a delta NxN

And this would be a diagonal matrix. OK? And then what do I have over here? Here I'd have the delta w's. And if you check you'll see that this will be-- I'm not going to get the indices right, and I don't have time. So I'm just going to write it like this.

And now I'm just going to give you the end of the story, because I've run out of time. You could write all this as dx is equal to a diagonal matrix times the derivative of the parameters plus a lower triangle or matrix times the x again. And so if you want to solve this, linear algebra just does the propagation. You have I minus L dx is DdP or dx will be I minus L inverse DDP.

And if I only want the last element-- let's say en is the vector that pulls out the last element, then this is all I'm going to need to get all my derivatives. And what's the moral of the story? I apologize for going one minute over.

But the moral of the story is instead of back propagating through your own hard work, you probably know that when you solve a lower triangular matrix, people will read written code that back solves the lower triangular matrix. The back, the big back piece, has already been implemented for you. Why reinvent the wheel in if the back-- if linear algebra already has the back, you see? And so if you just do this, and you do it in a language that lets you get full performance, you don't need to do your own backpropagation. Because a simple backslash will do it for you.

So I apologize for going over. I don't know if Professor Strang had some final words. But anyway, linear algebra is the secret to everything. That's the big message.

AUDIENCE: OK.

[APPLAUSE]

GILBERT STRANG: Well, since it's our last two minutes, or minus two minutes of 18.065. I hope you guys enjoyed it. I certainly enjoyed it, as you could tell. Teaching this class, seeing how it would go, and writing about it. So I'll let you know as about the writing. And meanwhile, I'll get your writing on the projects, which I appreciate very much. And of course, grades are going to come out well.

And I hope you've enjoyed it. So thank you all. You're right. Thanks.

[APPLAUSE]