[SQUEAKING]

[RUSTLING]

[CLICKING]

**STEVEN G. JOHNSON:** OK, so last time I talked about how in order to define a gradient, you need an inner product. So that way, if you have a scalar function of a vector, the gradient is defined-- basically the derivative has to be a linear function that takes a vector in and gives you a scalar out. So it turns out this has to be-- if you have a dot product, this has to be a dot product of something with the x. And we call the gradient.

So the gradient is the thing with the same shape as x that we take the dot product with the x to get the f. So what I didn't mention is that, in fact, not only did we need a dot product to define a gradient, actually we swept something under the rug earlier. We actually need a norm in order to even define a derivative in the first place. All right. If you have a vector space, a norm is some measure of the length of the vector or a measure of distance.

A norm takes in a vector v and gives you out a scalar. And technically, to qualify as a norm, this map has to be non-negative. It can't be negative. It can only be 0 if v is 0. If you multiply the vector by 2, it has to multiply the length by 2. Or if you multiply the length by negative 2, it has to multiply the length by 2, basically the absolute value of any scalar. It has to satisfy something called a triangle inequality.

So usually, most commonly we're going to get a norm just from an inner product. So once you define an inner product-- we talked about those last time. You can even define inner products of matrices-- you get a norm for free. You can take the norm is just a square root of the inner product with itself. And this satisfies all these properties for any inner product.

So the reason I mention this-- oh, and, by the way, just cultural note. So if you have a continuous vector space with an inner product, we call that a Hilbert space. If you have a continuous vector space with a norm, that's called a Banach space. So it's a fancy-sounding name, but it just means you have a norm.

**ALAN EDELMAN:** You can impress your friends with your fancy mathematics, but that's all it is.

**STEVEN G. JOHNSON:** Yes. So the reason I wanted to mention this is that really the definition of the derivative that we used earlier implicitly requires us to have a norm. So it actually is both the input and the output. So it really only applies to Banach spaces. So the reason for that is remember I define the derivative to start with.

If you look at the change in the output, f of x plus delta x minus f of x, for not an infinitesimal but a finite delta x that may be small, remember that we defined the derivative as the linear part, as the linear operator that gives the change to first order, which means we dropped any term that's-- we called it little o of delta x-- any term that goes to 0 faster than delta x. So any term that's small compared to this.

But in order to define what it means to be small, you need a norm. If I have two vectors, a column vector, and I want to say is this column vector smaller than that column vector, how do I check it? I check the length. You need to map it to a real number to get a distance or a smallness.

So formally, the definition of this little o dx is basically any function such that the norm of this over the norm of delta x goes to 0, as delta x goes to 0. And in fact, even to define a limit, you need a norm of delta x because if you've taken [INAUDIBLE], there's this epsilon delta meaning of a norm-- of a limit, sorry.

You can make this arbitrarily small. You can make this less than or equal to epsilon for all epsilon greater than 0. And I'm not going to go through the definition. If you've seen the definition of a limit, there's some absolute values in there that for vector spaces have to turn into norms. But basically it's just--

**ALAN EDELMAN:** My experience is everyone's seen the definition of delta and epsilon limits, and no one really understands it.

**STEVEN G. JOHNSON:** Yeah.

**ALAN EDELMAN:** Is that fair? Maybe some of you guys really do, but most of us don't.

**STEVEN G. JOHNSON:** Yeah, I mean, to be fair, it took people 2,000 years to figure it out. The concept of a limit and an infinitesimal was a big struggle in mathematics going back to the ancient Greeks, Zeno's paradoxes and so forth. So it really took a long time for people to nail down what this meant. But yeah, you need to be able to have a length, a norm of the output, because this has the same shape as the output. These are the same shape as f.

To say that these terms are small compared to delta x, which you also need a norm of delta x. So just implicitly, you always need a norm of all of these things to define it. And usually, we're going to get it because we're going to define-- in most cases, we'll define an inner product, as we'll want that anyway because if we want to take derivatives of scalar functions, we want to be able to write down gradients. But this is what you really need.

So anyway, so I just wanted to-- this is something we swept under the rug in the beginning. But since we defined Hilbert spaces, so I thought I should define a Banach space. I mean, I'm still sweeping some things under the rug. I'm sweeping what does it mean for it to be continuous under the rug? But yeah, I wanted to throw that out there. That's all I wanted to say.

**ALAN EDELMAN:** That's it?

**STEVEN G. JOHNSON:** Yeah. Any questions about that?

**ALAN EDELMAN:** Questions? By all means. OK, good. All right. So this is just a little notebook. And if we really need-- this isn't the live version, so I can't really do anything. But I have a feeling that this will be good enough. But if we need the live version, we can just press a few buttons. So if I understood correctly, last time you got the answer for what is the gradient of the determinant. Is that right? Did you derive this formula?

**STEVEN G. JOHNSON:** I did not derive it. I just gave the answer.

**ALAN EDELMAN:** You gave the answer. And there's a few different formats. Did you--

**STEVEN G. JOHNSON:** I give it the first one. Determinant A--

**ALAN EDELMAN:** Is the cofactor.

**STEVEN G. JOHNSON:** --inverse transpose, yeah.

**ALAN EDELMAN:** Oh, the gradient is the determinant of A times-- that's the second one, right?

**STEVEN G. JOHNSON:** Yeah.

**ALAN EDELMAN:** So the first one is the cofactor of A, which is one of those linear algebra terms that you may or may not remember. This is the one. And just another term is the adjugate of A transpose, which is the-- the adjugate of a matrix is the inverse of the matrix divided by the determinant.

**STEVEN G. JOHNSON:** Multiplied by the determinant.

**ALAN EDELMAN:** Multiplied by the determinant, right. Let's see. Of course, if you have the gradient, then-- did you write down this version as well last time? The d of the determinant.

**STEVEN G. JOHNSON:** Well, I did d of any matrix function, so yes. I defined the dot product and matrix dot product.

**ALAN EDELMAN:** Right, right. I see. So d, the determinant, will be the trace of whatever formula you have over here, this formula times dA. dA.

**STEVEN G. JOHNSON:** Transposed. Transposed times it, yeah.

**ALAN EDELMAN:** Yes, right, sorry.

**STEVEN G. JOHNSON:** [INAUDIBLE]

**ALAN EDELMAN:** Let me clear my head. Yes, the trace of this thing transposed times dA, which is the element-wise-- it's just like the dot product. You all get that. It's just like the vector dot product. You multiply corresponding elements, and you take the sum. Whenever you have the trace of A transpose B, as Steven is writing very nicely over here, that's the A dot B. So if we know the gradient, then the d has to be this formula.

I'm just defining the adjugate right here so that I can have it handy. As Steven was saying, it's just the determinant times the inverse. This is just a definition. And then there's the cofactor matrix, which is the adjugate of A transpose. Once I've defined this one, to define this one, I just get to do the equality. So this defines these functions. And here I've sort of written it every which way.

The inverse in terms of the adjugate and the cofactor, the adjugate in terms of the determinant, the inverse and the cofactor. You get it. All three possibilities are written here. So for 2-by-2 matrices, here's the 2-by-2 matrix, and here's the cofactor matrix. Some of you will recognize that when you form the inverse of a 2-by-2 matrix, the determinant goes in the denominator.

And the thing that goes in the numerator-- right, you're all good at 2-by-2 inverses. Do you know that by heart? Would you be able to do it in your sleep? You switch the a and the d, and you negate the b and the c. Well, let's see. You negate the b and the c, but I'm also doing the transpose. So you negate the b and c and transpose because it's the cofactor. For the adjugate, you just take the minus. And anyway, these are all the formulas. Here's the inverse.

So the inverse is the adjugate divided by the determinant. Doing all this numerically just for fun. So numerically, here's a random matrix, and here's a random perturbation. What we're going to do is look at the determinant of the perturbed A minus the determinant of A. So there's the numerical value. And by the way, I know Steven has recommended always using things on the order of square root of epsilon to make the perturbations 10 to the minus eighth.

And he's right. I never do that, but you should listen to him. I just start typing three or four 0's and a 1. And actually, it's been good enough for my purposes just to check things. I mean, Steven's is more-- it's the best possible one, a square root of epsilon. But with a quick and dirty test, I don't have the time to type all those 0's, and I never remember to type 1e minus 8.

So I just type these four or five 0's or three, four, or five 0's. But in any event, here's what the finite difference. Here's the trace of the adjugate times. We see that they're correct to enough digits to believe it.

**STEVEN G. JOHNSON:** How come the adjugate is not transposed there? There's something missing here. Oh, no, it's the adjugate-- yeah, OK, right. The determinant is the transpose of the adjugate. Never mind. Never mind. Adjugate is the great--

**ALAN EDELMAN:** I have to go back and look at these formulas to answer your--

**STEVEN G. JOHNSON:** It's the transpose of the gradient, so yes.

**ALAN EDELMAN:** Let me say, yes, what you just said, that the adjugate of the transpose is the thing that you want. And so the trace needs to transpose it twice, so it's left non-transposed. Yeah. You got it. The gradient is the one transposed, and this has to be the transpose of the gradient.

**STEVEN G. JOHNSON:** Yeah. This is gradient of determinant.

**ALAN EDELMAN:** Right, like a double negative makes a positive, a double transpose makes for a no op.

**STEVEN G. JOHNSON:** This is our dot product.

**ALAN EDELMAN:** Yep. That's right. OK. So to actually see the gradient, we can rely on Julia's internal forward difference mode, for example, which is-- forward differentiation. It's not forward difference. It's automatic differentiation. It's different from forward differencing. It's the forward mode automatic-- I see this, and I think forward differences, but it's not.

I think in this lecture, if I get a chance in just a little bit, I'll tell you about how this forward mode works. Steven kind of gave you one view of it. I'll give you another view today. But you just say, give me the derivative or the gradient of the determinant function, and Julia will happily do it. And of course, I can compare that with the adjugate of A transpose. And you guys know me by now.

When I see these things matching, it looks like to all the digits, it makes me happy. It makes me think, wow, this formula for the derivative is correct. Right, so this, for sure, is the derivative of the gradient. OK. I don't know. Maybe I tried to say this before, but I'm just going to repeat it, if you've heard me say it.

But just philosophically, I find it remarkable that you could think of a limit of a finite difference, and the great mathematical gods let us have a formula. I mean, you've all done it in calculus, like the difference of a-- you take a sine and a little bit more sine, you get a cosine. Or the log, you get 1 over x. Or x squared, you get 2x. But I don't know. Could you guys imagine a universe where the mathematical gods weren't kind enough? Not every integral could be written as a formula.

I mean, as you know, lots of integrals can't be written in terms of elementary functions. But derivatives, you could always do it. And you could do it for scalar calculus. That's why calculus is so easy to teach and is a beginning subject. You could do it for vector calculus, and you could do it for these complicated matrix functions. I don't know. Do you ever stop and think about that being remarkable, or you just take it as a given and move on with your lives? I think it's amazing that we could have a formula for this difference. I just do. And a simple formula, in effect.

But maybe you guys just take it as a granted given, but I don't know. I think formulas are gifts from the gods, and I don't take them for granted. All right. So this is really just to show you how to do reverse mode in Julia. So it's not much different. I'm just calling Zygote, which is one of the big, popular packages in Julia to do reverse mode autodiff. And you see it's not much different.

This is the ForwardDiff package dot gradient, and this is the Zygote dot gradient. Under the hood, it's getting the answer in a completely different way, by going reverse mode. But it actually gives the same answer. Though I wouldn't be surprised-- maybe Steven knows better-- I wouldn't be surprised if this is just built-in formulas in both cases. I don't know. Let's see. We could do it symbolically, but let's get to the proof.

So there are a couple of ways to prove this mathematically. So one relatively simple proof is to remember the Laplace expansion of determinants. So I suspect you all remember that if you want to calculate the determinant of a big matrix, usually people take maybe the first row. But in fact, you could take any row.

But do you remember? You take the top-left entry times the determinant of what happens if you cross out the first row and first column. Then the second entry, and you cross out that row and column with a minus sign. Plus, minus, plus, minus. You remember that rule? So that's the Laplace expansion. And the key fact is that if-- for example, if I'm working with Ai1. Let's say I'm starting with the i-th row.

Then Ai1 is not inside any of these Cs. Ai1 only appears here. Everything else you see depends on other elements of the matrix, but it doesn't depend on Ai1. Similarly, if you look at Ai2, Ci2, and every other term only depends on Ai2.

To make that point very clear, here what I did was I took this matrix, and I made this matrix-- this 3-by-3 matrix you'll see, it's almost completely numerical, but I put one symbol in the bottom right. And if you take the determinant, you see that this is an-- I don't know whether to call this a linear or an affine function.

**STEVEN G. JOHNSON:** It would be affine.

**ALAN EDELMAN:** Affine for this class. Some people would actually say it's linear in the sense of linear, quadratic, cubic. It's first-degree polynomial. But let's call it affine for the purposes of this class because it's not 13a plus 0. But whatever it is, it's a first-degree polynomial is what it is. And the fact of the matter is the coefficient of a is exactly this determinant. It's 4 times 4 minus 3. It's 16 minus 3. It's 13.

And so the coefficient of every-- if you make any element a symbol, the coefficient that's in front of it is just this minor. And so taking derivatives of first-degree functions is easy. It's just the slope. The derivative of this determinant with respect to this element is clearly the number 13.

And so the way to say this, in general, is if I want to take the derivative of determinant with respect to any Aij element, the slope is the thing that multiplies it. So it's Cij. And so that is one immediate way to conclude that the gradient of the determinant is the cofactor matrix. It's that simple. There's another proof that is sort of-- I mean, this proof is pretty simple. I think it's easy to agree that this is a nice, simple proof.

There's another proof that might seem a slightly harder, in one way, but in a way, it's sort of-- mathematicians like this kind of proof. And so you get to take your pick which one you like best, but let me just show you an alternative proof. So in this alternative proof, what we're going to do is we're going to figure out the right answer near the identity. And then we're going to-- and then we're going to use that to bootstrap ourselves to any other matrix.

You know how to expand-- you're all familiar, if I ask you to compute the characteristic polynomial of a matrix, let's call it M. If I need to do the-- here, I'll do what Steven does, and I'll do it like this. So if anybody asked you to calculate the characteristic polynomial-- and I'm using the mouse, which means it's really sloppy. Not that my handwriting is so great, but it's not this bad. All right. So the characteristic polynomial of any matrix M is usually written like this.

And there's lots of factors. There's lambda to the n and all the way down to the determinant, plus or minus the determinant of M. So you remember that. And if you want, you can-- if you want, you can make this a plus sign, and then you get plus signs in this whole formula.

And so this is not much different. Here if lambda was 1, if you just took lambda equals 1, you'd have determinant of I-- well, let's just see it this way. Determinant of I plus a matrix would be 1 plus. And then there would be the terms that you would get. There would be the next terms. This thing here, as you all know, is the trace of the matrix. So maybe I should have put that in.

You get lambda to the n plus lambda to the n minus 1 times the trace of the matrix. So if you make a tiny, little perturbation, the determinant of I plus dA-- I guess I should have made this 1 plus the trace of dA, to be honest. Let's fix that right now.

So the determinant of I plus dA would be 1. That would be 1 to the n. Plus 1 to the n minus 1 times the trace of the matrix. And then there's the lower-order terms.

So that's one way to think of it. And so there we immediately get the answer around the identity. And now if we want to get this anywhere, all we have to do is recognize that if we want to go to the determinant of A plus dA, then we just go A times A inverse over here, and that's just the identity. But then we can use the properties of determinants to pull out the determinant of A. And you just get I plus A inverse dA.

And this basically here, we just think of this A inverse dA as the trace formula. And therefore, we get our answer, the very answer we're looking for. In a way, this is more complicated, but mathematicians like this one better than-- I don't know why. They're both valid. You get to take your pick. There's something I like about this, though it is a little bit more complicated. But in any event, we get the same answer.

So what do we have here? So application to the derivative of the characteristic polynomial. So once again, there's the simple proof. The characteristic polynomial of a matrix is the product of x minus the eigenvalues. Probably a different sign from what I have here. You take the derivative of this product. You get the sum of these products, n minus 1 at a time, which you could rewrite like this.

But you can also directly do-- with our technology, you can do this and get basically the same answer as the direct proof. And then I have some numerical checks. Let's see. And the derivative of the log determinant. Log determinant comes up a lot. Logs have lots of functions come up a lot. For example, Steven, I don't know, a few lectures ago talked about this f over f prime. It's what shows up whenever you do anybody's Newton's method.

And of course, this could be written as 1 over the log f prime. So basically, the logarithmic derivative and its reciprocal come up all over mathematics. So the derivative of the log of the determinant is simply the trace of the inverse times the dA. This you've seen, A inverse. And that's it. Any questions? That basically covers the gradient of the determinant. Any questions about that?

So maybe a few words about determinant. Interestingly enough, people often tell you that you should never compute a determinant. Or hardly ever might be a fair term. So determinants are real. It's a real favorite of elementary linear algebra classes. Determinants are great for telling you in exact arithmetic whether a matrix is singular or not. So a matrix has determinant 0, it's singular.

If the determinant is not 0, it's not. And that sounds like a really good idea, to have something like that. But it turns out that when you're doing computations in finite precision, if you're doing it on a computer, the determinant turns out to be not so meaningful. It gets to be hard to compute accurately. There are a lot of issues with calculating the determinant.

It turns out that while the pure mathematicians live in a binary world where a matrix is singular or non-singular, the truth of the matter is is that the world of matrices is not so binary. It's a bit more of a spectrum where matrices are singular or nearly singular or a little bit bad or not at all bad. And probably you've all heard the word that I'm referring to.

The word that we use in numerical analysis is conditioning. So ill conditioned means a matrix is nearly singular. And well conditioned means that it's very non-singular. Too many double negatives there, but it's sort of the good side of singular when we say it's well conditioned.

And so the determinant doesn't really give-- the determinant it's not a really good-- give a good job of talking about how nearly singular matrices are. The condition number, which is related to singular values-- I'm not going to talk about that today-- is a much better way of talking about matrices being singular or not. So you learned it all in a course, like 18.06 or elementary linear algebra. You learned about determinants.

And then later on, when you compute, people tell you to forget about determinants mostly. There are times, but mostly. And the other thing we tell people to do is forget about the characteristic polynomial as well. That's not how we calculate eigenvalues either. We don't take roots of polynomials. Anybody happen to know how we compute eigenvalues in the real world? We don't do characteristic polynomials.

Anybody know the magic two letters that happen when you type eigenvalues? How many of you just thought it was the characteristic polynomial? You take the roots. How many of you had any idea how roots got taken-- eigenvalues got taken on the computer? So do you have any idea what the algorithm is being used?

**AUDIENCE:** [INAUDIBLE]

**ALAN EDELMAN:** The power method.

**AUDIENCE:** Yeah.

**ALAN EDELMAN:** OK, so you probably didn't hear the student saying that, well, in 18.06, I learned about something like the power method, which gives you the dominant eigenvalue. Yeah. And then nobody else in the room has any idea how eigenvalues get calculated? Just a little bit of culture here. So people don't know. I see. I kind of feel like I ruined the question then. I should have just asked how are eigenvalues computed? Because I imagine many of you would have said, isn't it the characteristic polynomial? You get the roots.

Because every one of you have formed the characteristic polynomials of 2-by-2 matrices. I know you have. You got that quadratic equation, and you solve for the roots. You remember? Who remembers doing that? Quadratics, you get the roots. If you had a mean teacher, maybe they forced you to do a cubic, but I bet they didn't. Anyone ever do it for a cubic? Maybe it was rigged to be easy though.

So right, so none of that happens on the computer. I'm not going to tell you in detail how it's done, but I will mention just the fact that it's not the characteristic polynomial is half of what I want you to know. And the other half is there's something called the QR algorithm for eigenvalue. So in general, a QR for a matrix factors a matrix to orthogonal times upper triangular. And a funny thing happens.

If you factor a matrix into QR and then reverse it and get RQ, and if you do that again, factor that new matrix into QR and reverse it to RQ, and you keep doing that, essentially the eigenvalues magically appear. And there are some details. If the matrix is symmetric, the matrix will actually become more and more diagonal as you go. If it's not symmetric, but it has real eigenvalues, it will become triangular. And you'll see the eigenvalues on the diagonal eventually.

And if it's complex, you'll get these little 2-by-2 pieces which are easy to get the eigenvalues from. So there are a bunch of tricks to accelerate all this, but the basic idea is QR, RQ, QR, RQ. You might actually try it in Julia or Python or whatever. MATLAB, whatever you like to do one day. You'll see it just works.

**STEVEN G. JOHNSON:** But it is related to the power method under the hood, if you dig deep.

**ALAN EDELMAN:** Oh, dig really deep. That's right. It's sort of like a block power method in multiple dimensions all at once. It's crazy. Yeah. OK. All right. So that's that one. OK. So just to let you know what this notebook is and isn't, this notebook is kind of meant to let you see how automatic differentiation is kind of magical in a way. That's kind of the real purpose.

You'll start to get a bit of a feel for how forward mode works. And what I'd like to emphasize is to what extent this is possibly more computer science than mathematics. We all have this notion that whatever courses that are now being taught in computer science that maybe used to be taught in math courses, like probability statistics, I think everybody agrees that calculus lives in math departments all over the world.

Lots of other math subjects are being hijacked by engineers, computer scientists, and so forth. But calculus, that's sacred. That belongs in mathematics. Well, here's a case where calculus is as much of a computer science topic as it is a math topic. I think that's kind of what fascinated me most about this. Oh, gosh. I first put this together in 2017. Is it really 2023 now? Six years later.

Well, so it's an oldie but goodie, but I promise you you'll like it just the same. And I do like to tell people that I used to go to conferences, and I would hear people talking about automatic differentiation. People talked about it before it was hot. I mean, it became hot because of machine learning. But a couple of decades before machine learning, people would do it, and they would do it on the sidelines. Nobody paid attention back then. It didn't have sort of the big excitement that it has today.

But I would go to conferences, and somebody would get up and talk about it. And I don't know. I would read my email or tune out or work on my own math or something. I didn't really pay attention. And so I missed the boat. I didn't appreciate what I thought was most important about automatic differentiation. I made a jump in my mind of what it is. And I figured it was something symbolic, like Mathematica, Wolfram Alpha.

We've all memorized tables of derivatives. Here's a small table of derivatives. I figured that if I could memorize it when I learned calculus, then a computer could be taught to do this thing also. So maybe that's what it is. That's great. I could do it. A computer could do it better than me. Fine. I didn't care.

Turns out that's not what automatic differentiation is. So then I said to myself, maybe I got it wrong. I'm just guessing anyway, and I don't really care. But maybe it's some sort of numerical difference, like we do to check our answers. I want the derivative at this point x, so I can do a forward derivative and get the slope of the tangent.

Or I could do a backward derivative, go backwards and get the slope of this-- I guess it's a secant, to be technical-- but the slope of this line. Or I could even do a central difference, which connects these two dots. And the whole big deal, of course, is when you do that, as Steven explained, what's a good delta? In math, you want the delta to go to 0. That's the very limiting definition of a derivative.

But on a computer, if delta gets too small, as you've seen, you get that catastrophic cancellation happening. And so numerical analysis is kind of about what's a good delta. And Steven's basically said that something on the order of 2 to the minus 26, which is the square root of double precision machine epsilon is a good rule of thumb.

You might remember that curve where the error went down, like an absolute value sign, as it got smaller, and then it went up again. And the best one was at the bottom. That's the best delta x. But the key thing that's interesting is that automatic differentiation is not this, and it's not that. And so what could it be? And the way I'd like to show people of what it is is I'd like to start with a simple example of it in action.

And this is where, at first, it's going to look like magic. Nothing up my sleeve. And then I will explain to you how it worked. So I'm going to take one of the oldest algorithms, one of the oldest interesting algorithms known to mankind, the Babylonian square root algorithm, which I think maybe you've all seen. But if you haven't, you start with a guess t to square root of x. So I've got a t, which I'm hoping is close to square root of x.

And if I take t and x over t-- if t was too small-- x over t, of course, will be kind of on the large side. If t was a guess-- between t and x over t, one's on the large side, one's on the small side. So why don't we just take the average? And then so this is something we could keep doing. And that's the Babylonian algorithm that converges to the square root of x. Very simple algorithm.

It was known for thousands of years. How complicated could it be? But I still think it was pretty clever, the Babylonians. I mean, they didn't have Julia. I mean, I thought it was pretty clever.

So in any event, just for simplicity sake, I'm going to start at 1 for no reason. I'm just doing an example. So I'm going to start it at 1. So I have 1 and x over 1, and I'll take the average. And then, by default, I'll do 10 iterations of t plus x over 2.

So even if you don't speak Julia, I think this algorithm is easy to understand. Just it's an input x. And by default, we run 10 times, but you can give an argument and make it run more times. And so let's go using ForwardDiff.

Let ForwardDiff do it. And of course, with our modern view of the world, we know how to take square root. Everybody in this building, everybody in this institute could take the derivative square root. It's one half over the square root. And so we get the derivative. But before we do that, let's actually make sure that the code works. Let's actually check the Babylonian algorithm that I wrote.

The second one is Julia's built-in square root of pi. And here's the Babylonian calculating the square root of pi. And I guess this is pretty convincing that the Babylonians knew what they were doing. I mean, we could do the same thing with 2. We could run the Babylonian algorithm, except for some little bit over here. Maybe the last bit. We basically get the same answer with Julia's built-in square root and the Babylonian algorithm.

Let's skip this for a minute. I think that's not important. So just checking the algorithm for the moment, not even the derivative. So I'm actually going to plot a few iterates of the Babylonian algorithm just so you can see. So the first iteration, maybe remember, it was 1 plus x over 2. I still want to call this a linear function. But in this class, I have to call it an affine function.

Yeah, I don't know what the real terminology is. When do I get to say that a first-degree polynomial is a linear function? I think there's a context as to whether I'm calling it a map or a function. But any event, this thing's a line, however you want to call it. The first step of the Babylonian algorithm, iteration 1, is given x, compute 1 plus x divided by 2. And that's a first-degree function.

The second iteration, I'm plotting it, is here. And then it gets closer and closer to the sideways parabola, which, of course, is the square root. In fact, by iteration 5, you can't even see much of a difference with your eye. So iteration 4 is the purple. And iteration 5, you can't even see it because the black parabola's on top of it. And so just to kind of convince you that the Babylonian algorithm works. Anyway, I like Plotly. I love doing this all day. I can go left and right and look at these numbers. I love this. So I like interactive things.

Now what I'm going to do is-- let me see. I change this over the years. So 3, 4, 5, 6. In about nine lines, I'm going to create a function that will calculate the derivative of the Babylonian algorithm. And nowhere will I teach it one half over the square root of x. I will not do that. You'll see there'll be no finite differences.

And there will be no symbolic-- it's going to be by magic, but we're going to get the right answer. So are you ready? So I'm going to do it in nine lines. So here's three lines. I'm going to create a Julia type. I'm going to call it capital D. Maybe some of you have heard this word. D is for dual number, so that's why we're going to use the D. And we're basically going to keep a function derivative pair, an ordered pair.

And so this is nothing but a container to be able to keep two floats. But which floats am I going to keep? I'm going to have the value of a function at a point and the derivative of that function at a point. All right. I've used up three of my nine lines. And everybody agrees there's no finite difference, no symbolic answer, right? So I've got six more lines here.

Yeah, let me just show you what I've done here before I do anything else. So if I wanted to-- if I want to create one of these objects, I would have to put in a tuple, like D of 1, 2. This line lets me remove the parentheses, which are sort of-- I just find them annoying. This line doesn't count. But you see, I've got this dual number. It doesn't do anything yet.

I can't add dual numbers yet. If I try, it gets mad at me. Look, plus not defined. All I can do is define a dual number. That's it. It's just a pair of numbers. Can't do anything at all with it other than store it. But here what I'm going to do is create a-- oh, I don't need this greater than. That's why. I could have one fewer line. I'm going to comment this one out. I think somebody once asked me to define it for greater.

But yeah, I don't need this line. So I'm actually going to have 3 plus 5, 8 lines. So let me tell you about these next five lines. Mainly I want to define, add, and divide on a dual number. I don't need minus and times yet, because if you look at my Babylonian algorithm, if you remember the algorithm-- where is the Babylonian algorithm? I do a plus and a divide and nothing more. Later on, I'll add times and minus, but I don't need it yet.

So I wanted to define a plus and a divide. I don't need this one either. This thing could go away. In Julia, if you want to overload plus and divide and a few other things, you have to import it from base. So this is just like a Julia detail thing that says give me permission to redefine plus and divide and a few other things. And what do I want to do? When I plus a couple of dual numbers-- here let me just do this one only, just so you see.

I'll execute only the plus. Now I can add dual numbers. And it's just going to be like adding vectors. So I'm just going to add the first element of the tuple, 2 and 3, and the second element. So now I can add tuples, but I can't divide them yet.

So this dot notation, this broadcast or pointwise notation, says that basically add the two parts of x and the two parts of y, like a vector. So this adds 2 and 3 to get 5 and 3 and 4 to 7. All right. Now let me bring in the divide. I can't divide yet, by the way. I can try, but it'll get mad at me, you see. Oh, I must have executed it. Oh, did I execute it? I'm sorry.

Then it took away my-- well, whatever. All right. I guess I must have executed it, so I'm not getting away with it. But here it doesn't matter. So this is 2 divided by 3 is 2/3. But notice this isn't 3 divided by 4. So I have a different rule. So the add rule is just adding up a vector, but the divide rule is a little more complicated.

**STEVEN G. JOHNSON:** By the way, Alan, did you want to share your screen on Zoom? I forgot about--

**ALAN EDELMAN:** Oh, my gosh. I didn't share my screen, so you don't see a thing I've done.

**STEVEN G. JOHNSON:** No, I can see it behind you on the--

**ALAN EDELMAN:** Oh, that's funny. OK. There we go. All right. Better? All right. So divide. So everybody, of course, remembers the quotient rule from calculus? When I think of it, I hear my math teacher singing it.

It was denominator times the derivative of numerator minus the numerator times the denominator, or was it denominator squared? I don't know. Did your teacher sing it to you? How did you sort of memorize the quotient rule? Anybody have a good song for it? Anyway, you drill it into your head. vdu minus udv over v squared, or denominator, d numerator.

I mean, I don't know. You may have heard it different ways, but you all know it, right? This thing over here, the quotient rule, everybody knows it. I'm just extracting the parts from-- so y is the denominator. And so 1 is the value. So this is the denominator. x is the numerator. And 2 is the derivative. So it's the denominator times the derivative numerator minus the numerator times the derivative denominator over the denominator squared.

So that is what I'm going to teach. I'm going to teach Julia how to essentially add derivatives, which is just add, and how to divide derivatives, which is just the formula you know, just apply it at a point. And so this division is using all four numbers so that it can get the denominator times the derivative of the numerator minus the numerator times the derivative of the denominator over the denominator squared, you see. And that's what this one ninth is.

All right. That's it. Just these 3 plus-- what did I say? 3 plus-- oh, I haven't told you about convert and promote. These are a little bit more sort of technical details. But do you know how if you add a complex number and a real, like if you go 3 plus 4i, and you add 7? Now what's really going on is that that 7, in some abstract sense, is being converted into 7 plus 0i.

And then you add the real parts and the imaginary parts. Everybody does that all the time. So we want to do that sort of thing where if you have a real number, we want to think of it-- if you have a scalar, we want, in effect-- a constant is really what's going on here. We want to think of this as the constant x, where the value is x and the derivative is 0.

And we want that to be kind of automatic because it would be nuisancy to type it all the time. So that's what that does. And then the promote rule says that if you give it a number, when you see it in the context of a dual number, everything should be promoted to the dual number. Just like it happens with complex numbers, where, like I said, 3 plus 4i plus a real number, you'd put that 0i in your mind or on a computer. But you would promote everything into the complex land and then do the addition.

So those are two necessary things. And now let me go ahead and run the Babylonian algorithm. And without changing the algorithm-- remember the algorithm takes a scalar in. Let's see it again. Let's find it. The Babylonian algorithm, which is up here, it takes a scalar. I'm not going to rewrite the algorithm. I'm just going to feed it something new, something different from a scalar. I'm going to feed it a dual number. And so let's do it.

Where did it happen here? So I'm feeding the Babylonian algorithm 49 comma 1. This is how you seed-- we'll talk more about seeding the start of the story with the number 1. Or if it was matrices, it would be the identity. And we get the square root being 7. Yep, that's good. The square root of 49 is 7.

And the derivative, which you all know-- we could let Julia tech it for us-- is one half over the square root of x is this number right here. So whatever one half over 7 is, 1/14 or something. So this is the number 1/14. You should be astounded by this, that I took an original piece of code without a rewrite, and I fed it this funny kind of argument. And all that argument did was it knew the quotient rule, and it knew the sum rule.

And I got the right answer for the derivative, not symbolically and not with finite differences. Isn't that amazing like that's even possible? Wouldn't that blow your calculus teacher's mind that this could happen? Here's another example where I do it with pi, just in case 7 wasn't convincing enough. So this would be the square root of pi and 1 over 2 square root of pi. And this is the way done with Julia. And you could check the numbers. You see it all works.

And in fact, what you can do is actually look at-- what's happening is the Babylonian algorithm was an iteration. And so somehow, this square root is the result of an iteration. We do 10 steps of an iteration. And so at each time, we must be getting closer and closer to the derivative. So just like we get closer to the square root, somehow, by feeding this in, we must be getting closer and closer to the derivative of the square root.

And in fact, I could plot each step of the algorithm. So remember the first algorithm was first degree. I still want to say linear, but I'll say first degree. And so its derivative, of course, is a constant. It's just the constant one half, in fact. So there it is. And here are a couple of iterations. And I also plotted one half over the square root of x, the true answer, the reciprocal of the parabola, in effect.

And you could see that it's heading closer and closer. And pretty quickly, the eye can't even see it. So this doesn't explain to you how it works, but maybe it kind of adds to the convincing nature of the fact that it does work, and it's still mysterious. I could say a little bit more. I'm going to tell you how it works. But before I do, I'd like to show off a few things. I don't know how well this works these days. But I do like to tell people that, in Julia, you can see assembler.

Nobody reads assembler. Anybody here read assembler? Anybody here actually-- you do or have or a little bit? One person is willing to admit that they do it a little bit. Some computer science classes at MIT teach you this stuff. Most people never look at this, don't want to look at this. The thing that I like to just mention is that, in Julia, the assembler is short. And so this is the assembler for this derivative code, this kind of derived code.

And short assembler is more or less correlated with fast code. And so not only does it get the right answer, but this sort of game is also quite fast in Julia. And that's kind of a nice thing to be able to have. So I'm still not going to tell you how it works, but I'm going to grab SymPy. So this is Python symbolic program. There's is a Julia symbolic program, but I don't completely trust it yet.

Maybe it's ready for prime time, but I did this originally with-- I wrote this before there even was Julia symbolic. And anyway, it just works so well, I would take it. And so one of the things that's interesting is to ask-- how should I say this? I'm going to tell you something that's mathematically equivalent to what we're doing, but I don't want you to get the impression that this is how it's computed. So let's talk about not the derivative yet but just the Babylonian algorithm.

You remember that this is the function at the first step, x plus 1 over 2. I can use Julia's ability to overload to run it on a symbol x. And then I could see what there is at the second step or at the third step. And so in a way, at whatever this is-- if this is the first step, second, third, fourth, fifth. At the fifth step, the Babylonian algorithm exactly computes this rational function. It's a 16th-degree polynomial over a 15th-degree polynomial. But don't get the wrong idea.

Nobody in the real world is calculating the coefficients of this polynomial. I mean, these coefficients, we wouldn't want to store them. They'd be unwieldy to work with. But as a mathematical sense, the fifth step of the Babylonian algorithm is calculating exactly this function. And the plots tell us that this crazy function, the 16th over 15th-degree polynomial, is not a bad approximation to the square root of x, at least visually on the graph.

So this is pretty good for square root of x. That's what we've seen. We could talk about where it is good, where it isn't good. But the point is that what it's computing is this function. And we could do the same game for the derivatives. So the first derivative here is the coefficient of x as a half, that constant. And we can see what's being computed exactly here. This is a ratio of 30th-degree polynomials.

And again, I want to stress we are not-- I'm just building this up just for fun. We are not in the algorithm getting literally these coefficients. They're too big anyway for working with. But in a mathematical sense, the fifth step of this derivative Babylonian algorithm is calculating exactly this thing.

And so this has to be some sort of approximation to one half over the square root of x, the derivative of square root of x. This is what that is. So let me get a little closer as to how-- now you must be wondering. I hope you're all kind of sitting in your seats saying, how is this working? What's happening here? And to get you a little bit kind of closer, let me do what people used to do in the old days. People used to take derivatives of functions by hand.

Before this became automatic, people would take derivatives of functions by hand. And so I'm going to do that for you here. I'm going to create a dBabylonian algorithm, the derivative of the Babylonian algorithm. And you'll recognize that this line and this line are the original algorithm. And below it, I'll create derivative variables, t prime. And so t prime, the derivative of this is, of course, a half.

The derivative of this line of code, well, what is it? It's t prime plus the denominator times the root of the numerator, which is 1 minus x times t prime over t squared. So if you check, this is the ordinary calculus derivative with respect to x. So t prime is the derivative respect to x. So this is the ordinary calculus derivative. And we're doing that at each and every step.

And people used to do that by hand, that you would-- in other words, you don't take the derivative analytically of the big thing. Rather you take the derivative of each line of code. And then you have faith that if you do that, you'll get the derivative of the big thing that you wanted on the outside. And you'll see that, of course, it works. Adding these couple of lines of code with just-- this is now scalars. There's no dual numbers here.

This will give me one half over the square root of pi just by taking the derivative of every line of code. And so you might realize that this is actually an iteration for the derivative of square root of x, an iteration that we stop at-- we stop it at 10, by default. We could take more steps, but this is an iteration for the derivative square root, obtained completely by taking the derivative of every line. And so that's kind of what happened.

And so when I take the Babylonian of D, x, 1, in effect, I am using the magic of Julia's ability to do dispatch and overload and all those fancy words. But to use simple English, I am using the fact that I don't have to rewrite the code to get the derivative.

I just need the code to know the rules of taking derivatives of every operation that-- more atomic operations at the lowest level and rely on the computer to piece it all together. Because humans are really bad at this sort of stuff. They make mistakes all the time. It's worse than long division. I mean, no matter how good you are at long division, humans just make mistakes. We just do.

And so the trick is if you wanted to teach a computer-- if you want to get the answers to a division problem, we humans have taught computers to do the division for us so we don't have to. And this is what's going on with automatic differentiation. We teach the computer to do the atomic steps and then let it just go through the motions.

So the derivative goes in before the JIT compiler, and we get efficient code. So there's a notational trick, which is rather nice, which is instead of taking the dual number, which is a, b, we can write a plus b epsilon. And in effect, what we're doing is the same thing that-- on the first week of class, when Steven said, oh, let's just write everything as a plus bdx. Just write everything to first order.

Physicists do this all the time. They write everything to first order, and they throw away higher-order terms just all the time. So in effect, what's happening on the computer is we're treating every computation as a first-order computation. And then the basic rules are-- let me just see. There was one version of this that's broken, but let me see if this is right. I think this is the right version. So the basic rules-- every computation on a computer that's ever been written always can come down to plus, minus, times, and divide.

Even square root is implemented somewhere as plus, minus, times, and divide. So in effect, if you wanted to do, you can get automatic differentiation just by having these basic rules. This is all you need. Now as a matter of practice, we try to intercept it all.

We're happy to teach sine and square root and cosine because who wants-- whatever method is being used to compute the sine-- and it's not Taylor series, by the way-- but whatever method is being used, we don't want it to go through all this work. So we teach it things. But in principle, all you need are these rules, and you can take the derivative of anything in the world on a computer. This is all you need.

Here's the sum and minus rule, the multiplication rule, which if you look at this right, maybe if you squint correctly, this is the udv plus duv rule, the product rule that you all learned in calculus. And we kind of repeated it in its matrix context in this class. This is udv plus vdu, and this is the quotient rule.

This is denominator times degree of the numerator minus numerator times degree of the denominator over the denominator squared. It's just kind of rewritten in this first-order kind of notation. But these are rules that you all learned in first-year calculus.

And I'll even point out that you could do this symbolically. You don't even have to remember the rules. You could actually derive the quotient rule on the computer by just-- this says basically take a series around epsilon equals 0, and give me two terms, please. No more. So just give me to the first order, an epsilon. And here you see. You get the quotient and the quotient rule from calculus.

So this is one way to get your hands on that. If you wanted the product rule, I guess I could have just done this. And you get the udv plus duv rule. So that's how you can get the rules. So I'm going to do something fun. I am going to tell Julia-- this is Julia magic that says to print a dual number with epsilons. And so now when I type a dual number, you remember it was just with the Ds.

Once I execute this command, I could see it in a way that's nice and human. So I told you this was a function derivative pair, but you could also think of it, if you like, as a first-order expansion of-- it could be a first-order expansion of a function. It could be the first-order expansion of x squared around x equals 1.

So let's go ahead and add these last two rules. Remember I only did plus and divide. I might as well add the-- this seems like a good time to do minus and times. And you see that if I do the dual number 1 and 0, I get this.

Well, actually let me ask you. I'm not going to hit Enter yet. Tell me what I should see when I hit Return, before I do it. Who's quick? What's the first thing I'll see before the epsilon? Let me start with the 0-th-order term. What will I see? Just shout it out.

**AUDIENCE:** 4.

**ALAN EDELMAN:** 4. And then what's the next term?

**AUDIENCE:** 2, 4.

**ALAN EDELMAN:** 2 times 2. 4. Yep. You guys got it. OK. I changed the output. I might as well go the whole direction. Why don't I make the input also? D, 0, 1, I'll call it epsilon. And so now I can actually input epsilons too. Not just see it as an output, but I can do it as an output. So epsilon squared, of course, is second order. So we just get rid of it. By the way, just something fun. I actually never defined how to square.

You'll notice I define times, but I never define square for dual number. But this is sort of a little bit of a lesson, but a good software system would be one where when you square something, it actually replaces it with a thing times itself. So that a matrix square is a matrix times a matrix, and a scalar square is a scalar times a scalar. And in Julia, for whatever reasons, a string times a string is a concatenation of the string.

So a string squared-- I don't even know if this works anymore. I have a feeling it doesn't work. It's not a number. This is going to fail, but it shouldn't fail. Actually I think this is going to fail. Oh, forget it. It does work. So multiplying two strings will concatenate them, and squaring it concatenates it. And so if you have sort of a novice computer system, every time you have another type, you define another square.

But if you have a good computer system, then the square inherits it from multiply, and then you just have to define multiplication. What should I get here when I go 1 over 1 plus epsilon? And again, nothing symbolic. This is actually happening completely numerical, by the way. But what should I get when I hit Return? What should I see? Anybody? You're smiling. You think you know the answer?

**AUDIENCE:** I guess it's just written there.

**ALAN EDELMAN:** So I'll give you a hint. What's written there is not what you'll see.

**AUDIENCE:** [INAUDIBLE] minus 1.

**ALAN EDELMAN:** Yeah, how would it appear? Just read it to me how it would appear.

**AUDIENCE:** I'd guess 1 plus minus epsilon.

**ALAN EDELMAN:** There you go. 1 plus minus 1 epsilon. You could have just said 1 minus epsilon. I would have accepted that. All right. Doesn't that look like symbolic mathematics? But it's not. This whole thing happened through these-- it's all numerical. There's nothing symbolic at all. And this is another thing that people are saying, that there's becoming more and more of a blurring between the symbolic and the numerical, and that numerical stuff is starting to look more and more symbolic.

But it's not symbolic. All right. What's the answer here? I'm not using any weird packages. Everything that I'm using was defined right in front of you. I'm not using ForwardDiff or anything. Everything here is just pure, simple Julia. You saw it. There's nothing up my sleeve. What should this answer be?

**AUDIENCE:** [INAUDIBLE]

**ALAN EDELMAN:** I'm sorry.

**AUDIENCE:** [INAUDIBLE]

**ALAN EDELMAN:** You're right. 1 plus 5 epsilon. OK. And this one, unfortunately, won't work. Oh, it does work. Oh, that's amazing. I don't know why that works. All right. Never mind. I didn't think we could take negative powers, but I guess we could. All right. I'm going to stop. You could do this with n-th roots. You could do lots of other things. But I think this is a good time for a break. And you guys get the right idea.

So now you're starting to see. If I were to summarize-- and I know it's still a little bit magical, but I think you'll see that roughly how this works is that one way or another, we're giving the rules of plus, minus, times, and divide. And we're writing programs. And then these programs are, in effect-- they're not really rewriting themselves.

What's really happening is that every time you execute a plus, a minus, times, and a divide, it's doing not just the basic operation that you'd all expect, but it's also carrying along the derivative as well. And the way Julia works, Julia will actually look at that divide and say I'm not dividing scalars. I'm dividing dual numbers. Or if it sees a star, I'm not multiplying.

How does Julia know what to do? When it sees two matrices, matrix, star, matrix, it knows, because it's a matrix, to do matrix multiply. So here when I did dual numbers, I taught it to do this dual-number thing. And once Julia knows how to do it, it'll just carry all the way through. And in effect, this is really the magic of great software, where you just can define some atomic operations, and the whole thing kind of composes itself almost by magic.

And in a way, it's almost opposite from what we teach students in a lot of classes, where we want to teach-- the old-fashioned thing was to teach a student to carry through every operation and be really competent at it. In a way, the modern world is to teach students how to not have to think, rather how to build a system that is so simply designed that it just works.

And actually, to build a simple system is what takes the real human cleverness, if that sounds not like some sort of contradiction. But that's what it takes. All right. I'm a little late for the break. But after the break, on the Blackboard, I'm going to go into more detail about forward and reverse mode, automatic differentiation.