[SQUEAKING]

[RUSTLING]

[CLICKING]

STEVEN G. JOHNSON: OK. So I think we can get started. So the next part, I'm going to show some slides, so talking about some applications. As usual, these slides will be posted later today to the GitHub page. And also the handwritten notes also from the first part will also be posted. They're probably already posted.

So now I want to talk about some-- remind you of why we're computing derivatives in the first place. And so we talked about some things at a high level in lecture one. I want to just drill down a little bit and then also talk more about computation of derivatives.

So one basic use of derivatives is to solve a nonlinear equations. And you all learned this probably in first semester calculus if you have a-- a scalar function f of x that takes a scalar in and gives a scalar out. And that's some nonlinear-- if it's linear obviously, you can solve it.

If it's a quadratic, you have a quadratic formula. But if it's some arbitrary function like sine of cosine of x or something like that, like some complicated thing, you might not be able to write a closed-form solution. But there's a nice way to get the solution approximately to any accuracy you want, as many digits as you want very quickly if you know at least roughly where the root is, and that's Newton's method.

And what it really is it's really a linear algebra technique. It's really taking the function, approximating it by a line, by linear function. And then once it's linear, it's easy to find the root, and that gives you an approximate root. And you use that as a guess.

So that's the three steps here. So if you're solving f of x equals 0-- so f of x is this blue curve on the right-- then you linearize with your derivatives. So f of x plus dx is approximate-- for delta x, and it is not infinitesimal-- is approximately f of x plus f prime of x delta x. That's what the derivative is.

And then if we treat the equation on the right, the linear problem as an approximation for a real problem, that one's easy to solve. So we can set that linear thing equal to 0. Solve for delta x, and it's just minus f over f prime.

And so that gives us a new x. So that's our guess for the solution is x plus delta x, which is the same thing as x minus f divided by f prime. Of course, it's not an exact route because this linear was only a linear approximation.

But it gets us closer to the root. So on the right is a nice animation from Wikipedia that shows this process, starting from a few x's where you start with an x. What you're doing is approximating a function by its tangent. That's the red line, and that gives you the next x.

And so you start with this x. And the x1 on the right gives you the x2. And then you linearize it. You get another x, x3, and then another x, x4. And then very, very quickly, once you start getting close, this converges extremely fast.

If you're far away, it can be a little unpredictable. But once you're close, it turns out-- it's actually amazing. It doubles the number of decimal places in every step. So if you have it to one digit, on the next step, you'll have it to two digits, then the next step four, then 8, and then 16, and then you run out of digits on your computer very quickly. So it's really, really a great algorithm. It goes back thousands of years, the Newton's algorithm.

But you can see that as soon as we write it this way, then we should be able to see that the same thing works for multi-dimensional functions. So if we have a function that takes a vector in and a vector out-- so suppose x is a vector in rn. I don't know why that comes out in that weird font here.

So you have n equations and n variables. So that's important. You have to have the same number of equations and variables. But now they're nonlinear for some arbitrary nonlinear function, and you're trying to find a root.

And then you do exactly the same thing. You approximate f of by a linear function. f of x plus x is approximately f of x plus f prime of x delta x. And that's what we've been seeing over and over again. This is the definition of f prime. Is that linearization?

Oops, and this always happens when I display it on the computer. Let me see if I can-- and for some reason, this display gets screwed up a little bit. Hold on, how do I get out of this? Yeah. So it looks fine here in non-presentation mode.

So what do we do? We approximate f by a linear thing, and now we know how to do that for any vector space. And then, for example, x is a column vector; then f prime is a Jacobian matrix.

You set this linear approximation equal to 0, and you solve it for delta x, and that's just a linear equation. So that's just a 1806 thing, a linear algebra thing. So now delta x is-- you move f to the other side, and you multiply both sides by f prime inverse.

Delta x is minus f prime inverse times f of x. So it's just the inverse of the Jacobian matrix times f with the minus sign. And so that's our new x. Our new x is this x-- the original x plus this delta x, which should give us a root if it were linear.

So it's just f-- the new x is x minus inverse Jacobian times f. So you solve one linear equation with a Jacobian, and you get your new x. And of course, this is an approximate root, but you just keep repeating the process.

It's much harder to picture this in higher dimensions, compared to the 1D case. But it works amazingly fast. It says the same thing. You solve a linear system in each step. It doubles the number of digits in each step.

And with the caveat that you need a starting guess close enough to the root. If you start very far away from the root, it can be kind of unpredictable what this does. In fact, it can even make these amazing fractal patterns. So if you Google Newton fractal, there's a nice Wikipedia page on this that has lots of beautiful images of what Newton's method does if you start far away from the root in higher dimensions. It has this weird fractal pattern.

So you have to have some rough idea where the solution is, but then you can polish it off super fast. So that's Newton's methond. So this is a great application for Jacobians and for derivatives in general of equations that take vectors in and give vectors out.

And the other even more famous application I would say is optimization. So optimization, you have a scalar function. So you have a vector in and a scalar out, and you're trying to minimize it, for example, or maximize it. Maximizing, minimizing is just flipping the sign.

So for example, machine learning, you have a fx. You have a big neural network. x Is a million parameters of your neural network. And the thing you're trying to minimize is the loss function that basically you evaluate your neural network, compare its output to what you want its output to be.

And you take some norm of the difference, and that's your f. It's the error in the neural network for its function of the parameters. And you want to go downhill and make that error as small as possible.

And so the key point is if you can take the gradient of this-- and now we have a general definition-- for any vector space, any scalar function, if we have a dot product, then we can define a gradient. And that gradient then points in the uphill direction, or minus gradient points in the downhill direction, which is called the steepest descent direction.

And the most basic idea-- and there's many complications on top of this. But the most basic idea is you just go downhill. Imagine you're in the mountains in a fog. You can't see anything. But if you want to find the bottom of a valley, you just go downhill.

You might not find the deepest valley. You're going to find the closest valley. There might be a deeper valley someplace else, but you'll find a local minimum.

And even if you have a million parameters, as you have very commonly in neural networks, this allows you to evolve them all simultaneously in the downhill direction right. You want to update all-- in your million dimensional space, you want to go in this direction. You don't want to optimize each variable one at a time because that'll take forever if there are a million variables.

And as we're going to talk about over and over again, that it turns out that when you have a scalar function of vector inputs, you can compute all million derivatives or all n derivatives, the gradient, with about the same cost as evaluating the function once. So evaluating the function, say, is a big neural network. You have to evaluate that once to get your loss function. It turns out you can get the derivative of that loss with respect to all million inputs with essentially the equivalent of one additional function evaluation.

So for neural networks, this is called backpropagation. In automatic differentiation, this is called reverse mode differentiation. This is also called adjoint methods.

And as I alluded to the other day, it's equivalent to just evaluating the chain rule and multiplying left to right so that you always multiply vectors times matrices and never matrices times matrices. And this is what makes large-scale optimization practical. So when you're optimizing over a million parameters in a neural network, or you're optimizing the shape of an airplane wing characterized by zillions of parameters and so forth, portfolio optimization and in finance and so forth.

So there's a lot of complications in this that I'm not going to go through at all, but I just want to mention some of the complications that if you want to do this for real, that you would cover, to some extent, in a nonlinear optimization class.

So the gradient points uphill minus gradient points downhill. So, great, that tells you what direction to go, but how far do you go in that direction? And usually, you want to take kind of as big a step as you can to go downhill as far as possible.

But if you take too big of a step, the gradient is like a linear approximation. So that's only going to be valid for relatively small steps. If you take too big of a steps, the gradient is not going to be an indication of the downhill direction.

So there's a lot of different strategies. One is you do a line search. You do 1D minimization in this direction. And you take a step. And the simpler thing is you take a big step. And if the function gets better, great. If the function does not get better, then you kind of backtrack along that direction. So you'd divide your step by two or some geometric factor, and you keep doing that until you find you're going downhill.

Another thing is you have something called a trust region. You have some notion of how big a step you're allowed to take. And you take you-- you basically optimize it within this region using the linear approximation. And then the tricky parts are like, how do you decide how big the trust region is as you go along? You might want to grow it or shrink it.

And it turns out this is really tricky to get right. It's really easy to write down an algorithm that looks reasonable, but actually kind of goes in circles forever and ever and never converges. So you have to-- and the history of nonlinear optimization is full of such algorithms that look reasonable, but it turns out, don't converge in general. They might converge sometimes, but then sometimes they might just loop around forever; or worse, they might seem to converge, but they're actually stopping at a point that's not a minimum.

But there's lots of algorithms for this. Usually, most people doing nonlinear optimization do not write their own algorithms. They take one off the shelf. There's a lot of trade out-- and you try different ones. Each one, you provide a function, you provide the gradient, and you let it somehow decide what to do, and you try different ones and see which one works well for your problem.

Another complication is that a lot of real problems, you're not just minimizing function. You also have constraints. So you have often a set of constraints. And the general form is you're minimizing a function f of x, subject to some set of constraints, say, of some functions gk that you want to stay negative; or feasible, is the terminology.

So it's more complicated, but you still need the gradient of f to find the downhill direction. And turns out you still need the gradient of your constraint functions. To approximate the bilinear things that are easy to understand and easy to take steps.

Another complication is that if you just go straight downhill, if you have a function that has kind of a long narrow valley, things tend to zig-zag a lot. So you might converge very, very slowly where you zig-zag back and forth and with steepest descent.

So people have lots of corrections to try and improve this using momentum terms-- you might have heard that-- conjugate gradients, other forms of memory to try and remember the previous steps and not zig-zag, backtrack in a direction you just optimized. Another fancy thing you can do is you can try and estimate second derivatives or even calculate second derivatives-- are called Hessian matrices-- and then use that information to help you. And there's the most important class are called BFGS algorithms.

So I don't expect you to understand all these terms. I just want to throw out some culture. So if you hear, oh, use a momentum term or use BFGS, you're trying to avoid zig-zagging by kind of remembering what you did before.

And it's one of those things where there is no one best algorithm. In certain fields, there's algorithms that are very popular. Like, in machine learning, there's an algorithm called atom that's very popular. But in general, there are a lot of algorithms out there. There's not one that's kind of one that's become the best algorithm.

So in general, people-- you have to do a little bit of trial and error. For a given problem, you experiment a little bit, see what other people on similar problems use, or just try different things and see which one works best for your problem.

And so and some imparting advice on this is that when you're doing large scale optimization, the main trick is not often the detail-- it's often not the details of these optimization algorithms. You're usually using off-the-shelf algorithms. So most of the practitioners doing optimization in practice, they spend most of their time not choosing the algorithm, but on the problem formulation, really thinking about what exact function you're trying to optimize, what constraints, how do you write down the parameters in order to enable you to use the best algorithms and have things behave most nicely?

Because very often, there are multiple mathematically equivalent formulations of the same problem, but that have very different characteristics. And very often, someone if you're optimizing something, someone doesn't hand you a mathematical function that they want to optimize.

They describe in words, "I want to make a stronger airplane wing," or whatever it is. And there are a lot of qualitatively similar ways to write down that problem mathematically. And so it really matters for optimization to choose one that well-match to optimization algorithms, and choose one where you can compute where it's differentiable, where you can compute derivatives efficiently and so forth.

And as I said earlier, if you have lots of parameters, you really have to compute derivatives analytically. If you find differences, you'd have to take evaluate your function once per parameter.

So in particular, I want to talk a little bit more about not machine learning, but something that's near and dear to my heart, which is engineering or physics optimization, where, for example, that you're trying to make an airplane wing that's as strong as possible. So let me try and write down the general process of this.

So you have some engineering problem you're trying to optimize. What does it look like? So first of all, you have some design parameters. Let's call those p. And those may be the materials. It may be the geometry of the shape of the wing.

So it might be the forces. Some way you're going to-- what your input's in the system are going to be, some other kind of unknowns these are the things you can control as an engineer. And then you might have a lot of them. To describe the shape of an airplane wing, you might need a lot of parameters.

And then you're going to take those parameters, and you're going to stick them into some physical model, the physical model. If it's strength, it might be solid mechanics. It might be chemical reactions. It might be heat transport. It might be electromagnetic waves. It might be sound waves. It might be fluid flow problems and so forth.

And you're going to solve that model to find some physics, like the stresses or something like that, or the electromagnetic waves. So the simplest example would be a linear model. So you're solving ax equals b. And your matrix and your right hand side depend on your parameters. Those are the things you can control.

And then you find some solution, x, which is the physics. Say, the heat in your system or the electromagnetic fields in your system or the stresses in your system. And given that solution so that those X's now depend on the parameters through the matrix and through the and through the right hand sides. So these could be forces, displacements, and so forth, magnetic field, pressures, velocities.

Then, you have something you're trying to do, some objective. You're trying to make it as strong as possible or as fast as possible or as efficient as possible in power, or you're trying to match something to experiment. Like, if you have some unknowns, an unknown physics, and you're trying to match your it to your measurement of chemical reactions. So that's something that takes the physical solution and gives you a number that's your objective function.

Your parameters go into the physics. The physics goes into a solution. The solution goes into a design objective function, a number, a scalar. And then you want to optimize it. You want to update the design parameters to make this function better to say maximize or minimize it.

And so you have to compute the gradient. You need to compute the derivative of f, not with respect to f-- I'm sorry, not with respect to x, but with respect to p, your design parameters. So you have to follow the chain rue all the way through this process from the design parameters to the objective to get all these derivatives. And then you can go update the design parameters in the downhill or uphill direction to make your function better. So this is kind of the general cartoon.

And an example of that, I wanted to show-- there's lots and lots of examples, but this is kind of a fun function example. So this was found online. So they're trying to do what's called topology optimization of a chair. So what you're trying to do here is you're trying to make a chair as strong as possible with a minimal amount of material.

And it has a seat and it has a seat back. Those are fixed.

ALAN EDELMAN: I want to buy one of these. Where do I buy it?

STEVEN G. JOHNSON: [CHUCKLES] But the shape of the chair is just completely free. So if you imagine, you're going to start with just a block of metal. And every voxel-- you're going to divide it into 3D pixels, voxels. Every voxel, you can either put metal or air.

This is called topology optimization. Topology in math is how things are connected. Are their hole-- is it one big thing or are there holes? How many holes are there and so forth. And we're going to leave this completely free. So the computer is going to decide where to put the holes or if there are holes and so forth.

And I think what they're doing is they're maximizing the weight with a given amount of material. Either that or they're fixing the weight that it has to hold and minimizing. Probably, they're minimizing the amount of material subject to a constraint of a certain amount of weight.

So there's zillions of parameters here. And this is a movie. Let's see, does this work? Present on this device. OK. So this is a movie of that optimization you can see.

So it starts with just a-- their starting thing is just a solid block of metal. So that certainly will support your weight, but it's very, very heavy. And so now they're going to try and optimize it to minimize that amount of weight, subject to the constraint that it still has to support a person right.

And this thing-- even though there's a zillion parameters, going in the downhill direction allows you to continuously evolve that into this structure. And so it continuously shaves away metal, varying all million parameters at once. And it decides to put these weird this arrange of holes here to make this chair.

And so this is in some art exhibit. I think they said they then use some kind of 3D printing somehow to make life-size chair out of metal with this design, and it works. And they displayed it. More practically, I think, in the first lecture, Alan showed an example of an airplane wing that was designed by this kind of thing. And there's lots of examples of this.

So the key thing in this that makes this practical is your ability to calculate derivatives. You need to be able to calculate the derivative of the strength of this or some function of a solution with respect to all million parameters of every voxel, the density of the material at every voxel.

So as I said, this is sometimes called backpropagation in neural networks. It really is just doing the chain rule left or right. And so I want to talk about that in this kind of specific circumstance. You can really see how it makes a huge difference.

So suppose we're computing a function. We have some scalar function f of x. But our x we get by solving a system of equations, ax equals b, where the matrix depends on some parameters. And I want to compute the derivative of f with respect to the parameters in the matrix.

So x is a inverse b. So I'm really computing f of A inverse b, where the parameters are inside the matrix, and I want to apply the chain rule. So we can just do the-- so this is-- yeah.

So we have f of x and then x of p, and I want to propagate my chain rules to that. So I to ask what's df, the change in f, for a small change in the parameters p? So let's just do chain rule.

So first of all, we have f prime of x. So this is just the derivative of f with respect to x. So that's just a row vector. So x is a column vector. f is a scalar. So f prime is a row vector. It's just the transpose of our gradient of f.

But we're not done yet because that gets multiplied by dx. But that's the change in the solution. And that's not what we want. We want is what happens when you change the parameters or when you change the matrix.

So we do the chain rule. And we have to take the derivative-- x is A inverse b, so we need to take dA. And b depend on the parameters. That's just a constant. So we just need dA inverse b.

Then we saw I think last lecture that the d of A inverse is equal to minus A inverse dA inverse. Do you remember that? We derived that last term just by taking the product rule AA inverse equals I. d of that, we derive this rule-- the D of A inverse is minus A inverse Da A inverse-- really useful formula.

And so now if we group this, let's look at this expression. You get A inverse b, which is just x. So you're really computing f prime, which is a row vec, times A inverse-- so inverse of our system matrix-- times the change in the matrix to the parameters times x times A inverse b.

So the whole trick, the whole stupid trick, which becomes obvious once you see it, is just to take this product and multiply it left to right. So that means you want to put parentheses around the F prime A inverse. Why is that? Because that's independent of dA.

That's one vector. That's solving one system of equations. If you do a row vector times a matrix on the left, that's one matrix vector multiply. So that's the same thing as A inverse transpose times F.

Or equivalently, if you call this-- this is a row vector. If you call this v transpose, it's the same thing as solving A transpose V equals f prime transpose.

So that means this is called the adjoint equation. So you solved-- think what you do. You first solve one equation Ax equals to get x. That might be really expensive because this is a big physics simulation. A is a huge matrix.

But then you solve another equation with the same matrix, just transposed, to get v. And then once you have that, then the derivative with respect to any parameter is just v transpose dAx, which means if you have a parameter, this should be a pk out of that little box there. It's a font problem. It should be a p usb k.

If you have a whole bunch of parameters, if you want the derivative of f with respect to pk, that's just dividing both sides by tpk. That's partial f partial pk is just df dpk is v transpose I guess with the minus sign. I should have put a minus sign there.

How do I get back? And I tried this tablet presentation mode. Let me just-- I'm going to switch to my computer. That's just because this is annoying.

ALAN EDELMAN: Did you develop on a Mac and go to a PC or vice versa?

STEVEN G. JOHNSON: No. I developed it on Google Slides, but then I'm using the Google Slides app on my tablet, which is a little bit flaky, apparently. So let me just share my screen.

ALAN EDELMAN: Yeah, sometimes the trick is to just do the PDF.

STEVEN G. JOHNSON: Well, I wanted to show movies and things. So I can just share it on my-- on my laptop, it should work better. Do you see that the fonts come out? Yes, good. So there should be a minus sign here.

We take this product. We're going to multiply it left to right, which means I multiply row vector-- the first thing I do is multiply row vector times a matrix, which gives you another row vector. But that's multiplying on the left by-- A inverse on the left is equivalent to solving a system of equations with A transpose for this row vector, where f prime transpose is on the right. That's why it's called an adjoint equation because an adjoint is another name for a transpose of a real vector of a real matrix.

So you solved one system of equations to get your x, another system of equations with almost the same matrix just transposed to get v. So it's equally easy or hard. And then your df is just-- there should be a minus sign here.

Or I can just put the minus sign inside. Well, it doesn't matter. df is just minus v transpose dAx. So now you have a whole bunch of parameters, pk. So if you want df/dpk or just divide both sides by dpk. So df/dpk is the same thing as minus v transpose partial a partial plx.

So I just take for each-- if I have a million parameters, I just take a million dot products. I only solve two systems of equations, but this is the expensive part. This is solving my big structural mechanics equation or whatever it is. I solve it twice, and then I just take a bunch of dot products.

And in fact, usually, this matrix is mostly 0. It's very sparse because usually each often each parameter only affects a few entries of the matrix. So this dot product is extremely cheap and because you don't need to multiply by 0's.

So this is what allows you to do this efficiently. You only have to solve the equations twice, your huge systems of equations, your physics equations, twice. And then you can get both the f and its gradient with respect to all the parameters.

So this is the adjoint method. It's exactly the same idea as back propagation in neural networks. It's exactly what people in automatic differentiation call reverse mode, but it's really just doing chain rule left to right.

And I want to make sure you understand why this is so important to do it left or right. So suppose we did it right to left. So if I did right to left, and I wanted partial f partial pk, then I have minus f prime, a inverse, partial a, partial pkx. It's the same equation, but it's going to put parentheses in a different place. Suppose I did that.

So for each parameter, this is a vector. And then multiplying by A inverse is a solve. It's a solving a huge system of equations. So this is-- and that's expensive part of the physics is usually solving this huge system of equations.

So you have to-- but you have one of these vectors for every parameter. So if you just did it right to left, which is called forward mode differentiation, it's terrible. You have to solve-- if you have a million parameters, you have to solve a million mechanics problems or whatever your physics problem is for this.

So this is called forward mode. And this is very bad basically when you have lots of parameters. And so it turns out this reverse mode differentiation is really good. That's what you should use when you have lots of inputs and only one output. Forward mode is going to turn out to be good when you have lots of outputs but only one input or a small number input.

So yeah, actually, I wrote this down. Right-to-left forward mode is better when you have one input and many outputs. Left-to-right, which is also called backward mode, also called adjoint methods. It's also called backpropagation.

It's a sort of thing that's been-- it has a lot of names because it's been I think rediscovered independently multiple times. It's just left to right chain rule. It's better when you have one output or a few outputs and lots of inputs.

And so if you're using automatic differentiation, most of them will use the term reverse mode or forward mode. So in Jax, for example, which is the Python thing. They'll call Jack forward. In Julia, there's a package called forward diff, which stands for forward mode differentiation. We're going to talk pretty soon about dual numbers. That's forward mode, and so forth.

Another thing you might come to mind is to use finite differences. So finite differences, if you have lots of parameters, it's also just as bad. right so if imagine you're using finite differences. So you're going to approximate your partial derivative with respect to each parameter.

What do you do? You take each parameter-- you take f and you perturb each parameter by epsilon. So you think of ek, here is the unit vector in the direction of pk. So you take each parameter. You perturb it by epsilon.

In the case direction is [INAUDIBLE] fp divided by epsilon. That gives you one partial derivative. But now if you have a million parameters, you need to do this a million times, each one is a separate solve.

You need to solve for-- you're perturbing the parameters one by one and, then you have to solve it one by one. So again, it's a disaster. If you had to use finite differences, you would not be able to do neural networks or topology optimization of chairs or airplane wings or things like that.

And you can generalize it in other ways. So that was adjoint methods for linear equations. So this is-- so you're solving f of x, where x solves a linear equation, but you can do the same thing where f solves a nonlinear equation.

Suppose you're solving-- you're computing f of x, where x is your physics, p is your parameters, like your shape of your airplane wing. But now to solve for your stresses or whatever, your physical equations are not linear. So what do nonlinear equations look like?

In general, nonlinear equations, you can write them as a root-finding problem. You have n parameters. You have n physics unknowns, and you have n nonlinear equations. So you have some nonlinear equations. It's called g that depend on the parameters, that depend on the physics, and you're trying to set these equal to 0.

So these are-- g is not a scalar here. Its n component vector. So it's exactly like what we talked about with Newton's method. So how would you solve this in the first place? You compute the Jacobian of this thing, and you do a sequence of linear solves to converge towards the solution.

But now how do we take the derivative of f with respect to p? So now, again, we want to do the chain rule. So df is f prime of x dx. That's the derivative of f of x. So f prime is still a row vector.

What is dx? So now we need to think about the chain rule. So we need to go back and think about the derivative of this equation. So this is our equation that we're going to use to solve for x.

So g equals 0. That means dg is also 0. The solution g is always a constant. So d of 0 is 0. But then if we do-- you can think of this as two terms. There's dg, partial g partial p times dp.

So I'm sneaking back in some 18.02 notation. But think of this as like a Jacobian matrix now. It's not necessarily a scalar derivative. So this is the derivative aspect of dp, the change in the parameters. There's also the derivative with respectto x, which, again-- think of this as an n by n Jacobian matrix of the changes in all n components of g with all n components of x dx.

So we're just linearizing this nonlinear function in two variables, which we haven't-- two parameters, which we haven't done so far. So I'm just kind of introducing this notation. So if we write this down, and we just solve it for dx, I move this to the right hand side, I get a minus this on the right hand side, and I get a Jacobian inverse times dp.

Does everyone see this? So I took this. I linearized it with two Jacobian matrices with respect to the parameters, respects to the x's. And then I solve for dx in terms of dp. And you can see it's this equation. And then we plug that in for dx.

And so what you get is you get f prime of x here with a minus sign from here times this inverse Jacobian matrix times this other Jacobian matrix of the parameters times dp. And notice that this Jacobian matrix is exactly the same Jacobian matrix you would use in the Newton solver for x So if you're solving nonlinear equations, you already have this Jacobian. And you're already solving systems with this-- you're inverting this Jacobian or solving a system of equations.

And so, again, you do the same trick. The whole trick is, to compute df, you just multiply this left to right. So this is a row vector. This is an inverse Jacobian. You call this-- row vector times matrix is a row vector. Call that v transpose.

So multiplying this by the inverse is equivalent to solving a system of equations with the transpose of the Jacobian matrix. So again, we call this an adjoint equation or a transpose equation for the adjoint solution v. And on the right hand side is the transpose of this row vector f prime.

So we solve-- first, we solve a nonlinear system of equations to get x. And we do that by a sequence of linear solves probably, using this inverse Jacobian matrix. We solve with many times.

And then we solve one more linear solve just with the transposed Jacobian matrix. And then once you have that, then you can get the derivatives because this is just-- if you want df/dp, it's just a dot product of this transpose with dg/dp times-- the dp cancels. That's it, so it's just a bunch of dot products.

So this is called-- by the way, this formula where you linearize the nonlinear equations is also called the implicit function theorem. And so these are-- this is all the rage now. You have what's called a bilevel optimization problem or problems with optimizing problems with implicit equations. And it's really, really powerful to be able to do this.

And actually, the automatic differentiation systems usually have to be told that you're doing this. They don't know-- if you solve this nonlinear system by a sequence of Newton's steps, the automatic differentiation doesn't know that what you're solving is g equals 0 by Newton's method.

It thinks, oh, you're just doing a bunch of steps, and it tries to do the chain rule, propagate it through all those Newton's steps. So it's basically trying to exactly differentiate the error in your Newton's solver.

If you tell it, if you know that you're solving a nonlinear systems, you know that this is what you should be doing. You should really only be for the adjoint problem, for the back propagation, or for reverse mode, you should really only do effectively one Newton step, one linear solve. So it's really important to be able to do this basically to know that you have to tell the automatic differentiation to do this either.

You either supply a manual derivative of this part, or there are now ways to basically tell the automatic differentiation solver that one step of it, is solving a nonlinear system. And it should do this for you. Yes, that's my last slide here.

So even if you're using automatic differentiation, and the computer is taking all your derivatives for you, doing all your chain rules and so forth, it's important to have some idea of what's going on under the hood. Understand adjoint methods. Understand reverse mode versus forward mode.

Because, first of all, you need to know what is forward mode? What is reverse mode? When should you use forward? When should you use reverse?

Another problem is that if you're doing you know physics, especially, a lot of physics methods, you're calling large software packages written over decades in various languages that can't be differentiated by automatic differentiation. Jax can understand Python programs, but it doesn't really understand Python programs that call libraries written in Fortran. So, often, you need-- some parts of it, it just can't handle.

And so but it turns out you only need to write down the derivative for that part or a vector Jacobian product, what's called, for that part. And then it can do the rest of the chain rule for you. But you really need to know these adjoint methods in order to do that in a reasonable way.

And then even if you have Ad, even if it works, if you have a model that involves an approximate calculation, like a Newton solve or something like that, automatic differentiation typically doesn't know this and spends a lot of effort trying to differentiate exactly the error in your approximation. It wastes a lot of time.

So it's really, really useful to know that and to know, oh, I need to tell it something to help it. So examples when you're solving nonlinear systems by iterative methods like Newton equations, or even if you're approximating things like by discretizing physics, like finite element methods, often you want to differentiate the physics before you discretize, rather than have it try to do automatic differentiation on the discretization itself.

Anyway, so I'll stop there. And we'll be spending more time on the nuts and bolts of automatic differentiation. And I'll also give some more links to further reading on adjoint methods and backward mode and vector Jacobian products and these kinds of-- these kinds of systems I've-- there's lots of notes written in web pages on these kinds of things.

ALAN EDELMAN: And I'll be back live on Friday.

STEVEN G. JOHNSON: Yes. And so maybe Friday, I don't know if you're going to do your dual numbers.

ALAN EDELMAN: I could do dual numbers and even reverse and forward mode the graph theory if you want.

STEVEN G. JOHNSON: Yeah. So maybe I think maybe it's time to do some automatic differentiation.

ALAN EDELMAN: OK, sounds good. All right, see you then.