[SQUEAKING]

[RUSTLING]

[CLICKING]

STEVEN JOHNSON: So Alan mentioned fine differences as something that automatic differentiation is not doing. But I do want to talk about fine differences because they do come up a lot when you're talking about computing derivatives on a computer.

And as he mentioned, we're going to spend a lot of time talking about automatic differentiation, which is this amazing technology where you can take a program that computes a function. And it will basically compute the derivative analytically in some sense for you. But how it does it takes a little bit of explanation.

And this is great. If you're hand-computing derivatives, even though you might think the rules are simple, as you see from the homework, when you get to more complicated functions, especially involving vectors and matrices, it's quite error-prone. And it's a common source of bugs if you're doing numerical optimization route finding sensitivity analysis-- all of these things on computers where you might want to differentiate a really complicated calculation.

So AD is a great alternative. It's extremely reliable. If it gives an answer, it's probably correct. Unfortunately, there's still a lot of cases where automatic differentiation doesn't work, where if, for example, it's code that calls external libraries and other languages that the automatic differentiation can't handle. Or even in a given language, usually they can only handle a subset of the language.

And there's other cases where, even if automatic differentiation could work, you need to really give it a little help in order to use it effectively-- basically any kind of problem where you're computing an answer approximately. Like, for example, you're solving a nonlinear equation by Newton's method. If you try to use automatic differentiation, even if it works, it wastes a lot of effort basically trying to not only differentiate your function you're computing but trying to exactly differentiate the error in your approximation.

So it tries to propagate the derivatives through every step of the Newton's method. And so it turns out you can usually do much, much better if you know what function you're actually trying to compute. And we'll talk about that a little bit.

And so in the cases where automatic differentiation fails completely or where automatic differentiation works but it's really suboptimal-- like, you'd like to replace portions of it with a more specialized method-- often you only need to give it a little bit of help. So what you do is you take the one piece of your program, the one function, one subroutine that it fails on or where it's doing a poor job on. And you do a manual differentiation of that. But then you let it handle the rest of it, your program.

And AD systems are very good at the chain rule. They're very good at basically-- if they know the derivative of each piece, they can propagate the derivative through the whole thing. So on Julia, there's a package called ChainRules for this. If you're using Python with autograd or Jax, you do it by defining a custom Jacobian vector product, or vector Jacobian product. And we'll talk about these as well.

But the end result is you often find yourself doing at least some manual derivatives of complicated things. And as I said, it's very error-prone. And so you really need to check it. If you have your program to compute the derivative or you're doing homework and you think you've differentiated some complicated expression that we've given you, it's a really good idea to check it. And then the usual way to check it is using finite difference approximations.

And so remember, when we look at the derivatives, right, you should think of it-- this is really a way of computing the small change in the output df for a small change in the input dx. And mathematically, we like these d's to be infinitesimal things. But they're really-- you can think of these as an approximation in the case when you have a finite delta x.

And we actually started in the opposite direction. We started with imagining a finite small change delta x. And that gives a finite small change delta f, right? And in both cases, the linear approximation is f primed. So it's f-- if it's an infinitesimal change in the input, then this has exactly f primed the x, or linearization. If you have a finite small change dx, then the change in the output is approximately the derivative times this. Right?

And so if you just look at the left-hand side, this is easy, right? If you have a function that computes f of x, you can just pick a very small delta x, compute f of x plus delta x minus f of x, take a difference. And it should be approximately equal to your derivative times that or operating on that change, delta x, plus higher-order terms which are negligible if you can make this delta x small enough. And we've actually-- Alan has done a whole bunch of examples of this sort in the notebooks so far. But I want to dig into this in a little more detail, right?

So this is a finite difference approximation, right, for the change in the output. Very conventionally, if delta x is a scalar, people often put the delta x on the bottom there and to think of it as a-- did I write that in? Yeah. I should write that in.

But so if you put the delta x in the bottom there, then-- because if it's a scalar, you can just say f prime of x is approximately f of x plus delta x minus f of x divided by delta x. So this is probably sort of the more conventional form to see this kind of approximation in as an approximation for the derivative rather than the approximation for the differential, right, the df.

But we try to push this format here because this one works even if delta x is not a scalar. We can't divide by it, right? You have a vector, for example, or a matrix, or something like that. And this is a linear operator. f prime is a linear operator acting on delta x, right? So this is still accurate, right?

So if you have a program that computes f, and then you can easily compute the change in f. Oops, I computed the-- the change in f for a small change in the output. And then you can compare that to what you think is your derivative that you worked out laboriously by hand maybe. Or if you're lucky, you have an automatic differentiation. You operate that on delta x. You should get approximately the same thing.

So this formula here, as it says here-- this is called a forward difference. It's called "forward" because you kind of perturb x in the plus direction if delta x is positive. Right? You can also do a backwards difference-- f of x minus f of x minus delta x, which is also approximately f prime of x dx. It doesn't really matter which one you use. So that-- but if I tell you "forward difference," the immediate question is like, what's the backward difference? It's just--

ALAN EDELMAN: Can I just point out that the backwards difference is not the same sense of "backwards" as in backward mode differentiation, or reverse mode?

STEVEN JOHNSON: Yes, yes. We're going to talk about-- forward and backward mode differentiation-- those are totally unrelated to forward and backward differences. So yeah. And as I said, there's a lot of forms. And we'll talk about maybe some more sophisticated forms of it than forward and backward differences.

You can do them-- the ones that give you higher accuracy for a given delta x. But they're generally a last resort as a computational scheme. But they're a first resort as a quick check, right? They're really, really easy to do-- really kind of braindead. And they're hard to get too wrong. So if this is a good match for your f prime of x dx, that's a really good sign. Right?

So let's just start out with an example. And this is an example we've seen a few times in the class, which is f of A. So A is now going to be a square matrix. And f of A is just A-squared OK? And we saw from the product rule several times that df is A dA plus dA A, which is not the same thing as 2AdA, the first-year calculus rule for scalars. Because A and delta A don't commute. We've seen this many times. Right?

So f prime here is now-- the derivative is really a linear operator that gives a small change in the input, gives you-- to first order, to the linear part of the change in the output-- A delta A plus delta A A. Right? And what we can do is just-- suppose I've derived this formula and I want to check that I didn't screw it up. This one's pretty simple. I'm pretty sure I didn't screw it up, but we want to check this.

So what you do is-- so let's define this function, f of A equals A-squared. Right? We'll try it for a random input and a random small perturbation delta A. So I'll define that function, define a random 4-by-4 matrix. randn is Gaussian random numbers. They can be plus, or minus, or kind of bell curve-distributed. So some of them are positive. Some of them are negative. But they're all of order one. All right?

And then delta A, or d-- I'll just call it, right, dA here. But it's really a delta A. It's a finite change. It's also going to be a different random matrix, but I'm going to make it small. And I'm going to multiply it by a 10 to the minus 8th

And we'll talk very soon about how small do you want to make it. Do we want to make it 10 to the minus 100th? Do we want to make it 10 to the minus 1? Where does this-- how do we choose this, right?

So this is going to be a small change. So it's random numbers of order one times 10 to the minus 8. So they're all kind of ordered 10 to minus 8, 10 to minus 9. It's different random numbers than in A. So these will not commute. Should we check that, actually? Let's do A times dA minus dA times A. Right?

And you can see that this is not 0. And you can say, oh, these are kind of small numbers but remember that the dA was a small number of order 10 to the minus 8. And so these are also small numbers. But they're small the same order as dA, right? So these are not actually small compared to dA. So this is not roundoff errors or something else. OK?

So now our finite difference approximation-- we just said it's just going to be f of A plus dA minus f of A. It's just the most simple, stupid thing directly from the definition of the derivative, but plugging in instead of-- we're plugging in a non-infinitesimal change in dA-- pretty small but non-infinitesimal.

So we get this approximate change. It's small, but it's of the same order as delta A, right? So this-- and so the exact derivative, again, is going to be A times dA plus dA times A. And again, this is small, but it's of the same-- it's small because delta A is small.

This is the df, right? This is not the derivative. This is, I should say, maybe the directional differential. Right? Right? So it's not the derivative. It's the derivative operator acting on dA, right?

And you can already see, just to the eye, this kind of matches, right? It looks like it's pretty good, right? There's the number and so forth, right? If we do 2A dA-- just a little bit smaller so I can see this. OK?

If I did 2A dA-- if I pretended they commuted, right, and I wrote A times dA plus dA times A as 2A dA, you get answers that are completely different. And in fact, these look exactly the same as these. But that's because Julia's only printing out six digits of this. And actually, there are more digits in these numbers. It's just not showing them by default.

So it might be useful to look at the exact answer minus-- or let's see, the approximate answer minus the exact answer. Right? So the approximate is the finite difference. The exact is this formula A dA plus dA A that we derive from our derivative operator, right? And the errors are really small. They're 10 to the minus 16th. So they're small compared to these numbers, 10 to the minus 8th.

So this brings us into to a key question when you're talking about errors. And just in general in mathematics, or science, or engineering, whenever someone tells you something is small or something is large, you should always ask, compared to what, right? It's a meaningless question to say, for example, is 1 meter-- is that a small distance?

Well, not compared to the radius of a hydrogen atom. That's a really long distance for an electron to move away from a hydrogen atom. But 1 meter is a tiny distance compared to the radius of the galaxy. It might as well be 0 if you're an astronomer, right? And so it really matters what you compare it to.

We need a yardstick. And a typical yardstick is, we want to have some measure-- so I'm going to talk about a norm. But if these were vectors, you'd know what the norm is. It's the length of the vector.

So the distance from approximate to exact is the size of this error. But I need to compare it to something to decide whether it's large or small. And a natural thing to compare it to is the size of the exact answer. Because the exact answer is that f prime of A dA, right? It's a small number because dA is small.

So you really want to know whether this error is small compared to that. So for example-- and we can see by eye they are, right? These errors are on the order of 10 to the minus 16th. And the exact answer is on the order of 10 to the minus 8th. So this is good, right?

But to make it quantitative, we define this ratio. And we call it the relative error, right? So it's the difference between what we want and what-- the difference between what we want, which is the exact answer, and what we get, which is the approximate answer, and divided by the size of the exact thing. OK?

So this norm is defined in Julia in a package called the LinearAlgebra package. And I can define a function relative_error that takes approx, exact, and computes exactly this-- the norm of the difference divided by the norm of the exact answer.

And if I just compare it here, we see actually it's what we expect. The difference, we said, was on the order of 10 to the minus 16th. The exact was on the order of the 10 to the minus 8th. The ratio was on the order of 10 to the minus 8th. And indeed, what we expected by eye is what we get here-- that it's about 9 times e to the minus 9. It's about 10 to the minus 8th-- so about eight significant digits, right?

So as I said, if we do the-- and let's do also the relative error of our stupid-- of our incorrect way. If I do 2A dA-- if I use that as my exact formula, and if I had done my derivative incorrectly, right-- suppose I thought my exact formula was 2 A dA. Then I'd get a relative error of 0.6 of order one.

So I really would have caught it and said, oh, this-- you notice immediately that this relative error is huge. Even though the 2A dA is small because dA is small, the difference is not small compared to the correct answer.

So you immediately notice something is wrong here-- either have a bug in my finite difference approximation-- but that's really hard to get wrong. It's just f of A plus dA minus f of A, right? It's usually dead simple. So you usually get it right. Or I have a bug in my derivative, which is really, really common, right?

So this is good. So you get a good match. This isn't a proof that something is correct, but it's a pretty big hint that you got something basically fundamentally right. It's pretty unlikely, if you had a wrong formula, if you picked a random input and a random displacement, that's going to give you the correct answer to eight digits, right?

And so it's a really good way to catch, really, screw ups. Oh, I actually wrote this relative error down there. Let me just-- yeah. So here's my-- so if I really screwed up, I'm actually-- I would be checking it against the approximate answer. But yeah, if I really screwed up, you'd notice immediately, right? So it's over 50%, right? Right?

And now I need to go back here and explain something that I kind of glossed over. So you all know what these vertical bars are for a vector, right? And so the norm of a vector is just the square root of the sum of the squares of the entries. It's the length of the vector. And that's one of the first things you learned about a vector, is that it has a direction and then has a length. That's probably the first definition of a vector that you heard.

But now, in linear algebra, we generalized our notion of a vector to a vector space-- things you can add, and subtract, and multiply by scalars. And so for example, in this example, we're using a matrix as a kind of vector. So we're thinking the inputs are matrices, which we can add, subtract, multiply by scalars.

And you really need to not only be able to add, subtract, and multiply by scalars to talk about errors, and derivatives, and magnitudes. You really need to also-- a notion of a length. Or the fancy word in linear algebra for that is a "norm" of a vector. And so we need to be able to-- we need a norm of our matrix.

And so what would that mean? What's the norm-- we know what the norm of a vector is. You probably at least learned the Euclidean norm. So if you have a matrix-- say, a 3-by-3 matrix-- or is this 4-by-4? These are 4-by-4 matrix, right? What does that norm mean?

And it turns out there are multiple choices. And it turns out it doesn't really matter too much for our purposes which one we choose. So I'm going to focus on the simplest one and kind of the most familiar one, which is basically-- this is the same as your Euclidean norm of vectors, right?

So for a Euclidean norm of a, right, column vector, what do you do? You take the square root of the sum of the squares of the entries. And this is also called an L2 norm, right? Or in linear algebra notation, I could write it as x transpose x is my dot product with itself. And then I take the square root.

So for a matrix-- at the simplest level, you could think of it as just a big bag full of numbers, right? The most obvious choice of a norm is you do exactly the same process to the matrix entries. You think of the matrix entries as the components of the matrix.

And you take the square root of the sum of the squares of those entries. And this is, in fact, called a matrix form, a very famous one called the Frobenius norm. And in Julia, you call-- oops-- norm(A). You call the norm function on a matrix. That's what it does by default.

And we can use the Symbolics package to just see what it's doing symbolically. So that's what Alan has been using there. So if I define a matrix full of variables-- and maybe I should use koala bears or something like that. But this works.

I think actually I can do an even nicer one. I can do m 1 to 2, 1 to 3. And M equals collect(m). I think that works. Yes. OK. So now I can have subscripts if you like subscripts better.

And then I can-- to do the norm. And it will do that symbolically. And it's exactly what I just said. So the norm is the square root of the sum of the squares of the entries. And it puts absolute value signs here. Because if they're complex numbers, then you need those. But we're dealing mostly with real numbers in this class. OK? Oh, it used to have an abs2 function, but it doesn't anymore. So let me just delete that comment. Right?

So one of the-- so this Frobenius norm way of measuring the size of a matrix is really intuitive. Because like I said, it's the direct analog of the first and maybe the only norm you learned for column vectors, the Euclidean norm. But it's also nice in linear algebra because-- so this Euclidean norm is really useful in linear algebra because you can express it in terms of these simple linear algebra operations, as the x transpose x. And then that's a number. And you take the square root.

So this Frobenius norm of matrices-- it turns out we can also write it in terms of matrix operations. It turns out it's exactly the trace of A transpose A square root. So A transpose A-- you take the trace. And you work out what it is. And it turns out that's exactly the sum of the squares of the entries of the matrix. And then you take the square root. And we call that the Frobenius norm, or AF.

And later on-- it says "recall," but I think we haven't done this yet-- we're going to generalize this to think about a dot product of matrices. We'll think of B dot A as a trace-- B transpose A. So this is like the dot product of A with itself. So this is going to become important in defining the gradient of matrix functions like determinants, and traces, and so forth.

OK. So this is-- and anyway, so this is kind of an aside. We're going to spend some more time on this pretty soon. But yes, you can absolutely define the norm for a matrix. And you can-- and it turns out the simplest one is kind of what you would expect, the square root of the sum of the squares of the entries. But it has this really nice formula that lets you do linear algebra with this norm, which we're not going to do too much in this class. But that's one of the reasons why this is so useful.

I can compute it in multiple ways in Julia. But I can take the square root of the trace of A transpose A. I can take norm of A. The square root of the sum of abs2 is the absolute value squared of the entries. Or I can take this kind of more complicated expression, square root of the sum of A i, j squared for-- i is 1 to 4. j is 1 to 4. They all give the same answers. OK?

And in fact, if we go back, this is something I kind of swept under the rug earlier. But in the whole definition of a derivative, right, when we define the derivative as the linearization of the difference, right-- so that basically the derivative is what gives you the difference to first order such that every other term is higher order.

In order to define what it means to be higher order, you have to have some measure of the length, of the size of a vector. And so you actually need a norm of any vector space if you want to define a derivative in this way. But fortunately, norms are very, very easy to define. And they're available for just about any vector space you're likely to think of.

OK. So now we have a norm of-- we know what it means to measure the norm of a matrix. We have-- oops, let me scroll down. We have a measure of the error. So this is the relevant error. We want the difference between the approximation and the exact answer, or a derivative formula divided by the magnitude of the exact formula.

And so now I want to go back and talk about this issue of how big dA should be. So we have a small perturbation. Here, I chose them of order 10 to the minus 8th. So why did I choose 10 to the minus 8? And does it matter? Like, what happens if I make it 10 to the minus 4th, or 10 to the minus 1, and so forth?

So before I do any calculations, let's think about what you expect might happen. So this formula, right-- this finite difference formula is an approximation for the derivative times delta x. But it's dropping higher-order terms, right? This only becomes exact in the limit as delta x goes to 0. Right? That's the limit in which we get-- or strictly speaking, I should maybe say, the norm of that goes to 0, right? That's the limit in which we get a derivative out of a difference, right?

So you might expect that, as delta x gets smaller, and smaller, and smaller, this difference formula should get more and more accurate, right? It should match-- these terms that you're neglecting should get smaller and smaller. And this difference should match our derivative more and more closely.

So what I'm going to do in a second is I'm going to plot the relative error as a function of the norm of delta x-- so in this case, delta A, right? And you might expect that the error is going to go down with delta A. And in fact, it-- and if we think about it, these are going to be small magnitudes.

So it's always nice to plot them on a log-log scale. A log-log scale has the nice property that it's going to turn power laws into straight lines. So usually in all of these things, we're going to see power law dependencies. It could be proportional to delta A, delta A-squared, delta A-cubed. On a log-log scale, that turns into a straight line.

And this is something that I try and drill into my students. Like, anytime you're plotting something, if at all possible, you should choose your coordinates-- either logs or some change of variables-- to make the thing you're plotting a straight line. It just makes it-- or at least, you expect it to be a straight line. That makes it much easier to understand what's going on.

So if we expect it to be a power law, we expect a straight line. If it's not a straight line, you should be able to immediately see that. Whereas, if you plot it on a linear scale and it goes on a linear scale, the error might go like this, right? It's really hard to see what kind of dependence that is. And it's also really hard to see what happens when things get really small.

OK. So let's do that. So I wrote down here some Julia code that is going to-- because I'm going to use the plotting package. And I'm going to make some nice labels. There's multiple plotting packages for Julia. I'm going to use one called Plots. And I'm going to compute the relative error of my f of A plus dA minus f of A. But I'm going to scale it dA by different factors. OK?

And I'm going to compare that to my exact answer which I had above, my A dA plus dA A. But if I scale dA by s-- that scale factor, which is going to go from 10 to the minus 8th to 10 to the 8th-- the exact formula is linear. It's a linear operator. So scaling dA by s will just scale that by s.

So this is just going to be-- so I'm going to keep the same random dA, but I'm just going to scale it up or down in magnitude by-- this gives me a range of points from minus 8 to 8. And then I'm going to take 10 to that. So it's going to go from-- this is 50 points from 10 to the minus 8th to 10 to the 8th. OK? And I'm going to plot them. And I'm going to plot also a straight line for reference.

And this is what we get. So this is a plot of the relative error-- so the finite difference minus the derivative formula divided by the derivative formula, magnitude, versus the norm of A-- so for the same direction, but scaling the size. And the direction was chosen randomly. And the dots are these errors. And the line here is just a line proportional to delta A-- just for comparison. OK?

So we see-- stare at this for a second. And just a bunch of things should jump out at you. So if delta A is-- so suppose we start out with a pretty big delta A of order one, right? Then you expect the finite difference is not going to be very accurate if delta A is big. And it's not. The error is, like, 100%. And then as you decrease delta A in magnitude, the error gets smaller just like we expect, right?

The error-- the fine difference formula should get more accurate mathematically as delta A gets closer to a dA, it goes closer to an infinitesimal. So you can drop those higher-order terms. And it actually goes to 0. It gets smaller linearly, proportional to a straight line, proportional to delta A.

But at some point, when the relative error gets about 10 to the minus-- I don't know. This is about 10 to the minus 10th here, it looks like. It stops. The error stops getting smaller. And then as you make delta A even smaller, the error gets bigger. So it seems like math is broken here.

So does anyone-- any guesses on why the error is getting bigger when we make delta A really small, like 10 to the minus 10th, 10 to the minus 12th? Like, why isn't the error just getting better, and better, and better? That's what the equations tell us.

AUDIENCE: Does it have this over there?

AUDIENCE: It's a roundoff error.

STEVEN JOHNSON: It's roundoff error. Yes. So what's happening-- and so there's two main features. So first of all, the relative error decreases linearly with delta. That's what we expect. This is called first-order accuracy, by the way-- this linear decrease.

But then it increases. And it increases because of roundoff errors. Because the computer is only going to keep a certain number of digits when it does arithmetic. And so when delta A gets really, really small, it turns out it's going to cause the answers to be completely wrong. So I'm going to talk about both of these things.

So first of all, I'm going to talk about the right portion of the plot, where it's kind of doing what we expect. The error is decreasing as you make dA smaller. But I want to drill in a little deeper. Like, why is it decreasing linearly with dA? Why are we getting first-order accuracy, as it's called-- so error that goes dA to the first power, right?

And then I'm going to talk about the second portion of the plot, this left portion-- that when dA gets really small, the error actually not only stops decreasing. It actually gets worse. And as someone said, it has to do with roundoff errors because of the finite precision of computer arithmetic. But I want to drill down into that a little bit more as well.

So let's start with the right portion of the plot. This is easier to understand, I think, because it's not about computer arithmetic. You can understand it just from the equations, treating everything as real numbers, as exact arithmetic.

So we're computing-- so basically, the way to understand this is, you can use a Taylor series. So if you can imagine taking f of plus dx and thinking of this not just as the definition of a derivative, but thinking of it as a Taylor series.

So if you take the Taylor series, you get f of x plus dx. What's the first term? It's f of x. What's the second term? It's f prime of x dx. That's our linear term. And this still works even when dx is a vector, or a matrix, or in some vector space.

And think about, again, 18.01. What's the next term in the Taylor series? The next term is the second derivative term, the f double prime term. And that would give you terms that go like dx-squared. And it turns out this still works for arbitrary vector spaces. This second derivative is going to be something called a Hessian or quadratic form.

We'll talk about that later. But the basic idea is this is proportional to a second derivative. And it has terms that go like the square of the change. And then there are even higher-order terms, which I can use this notation in computer science-- this little o of delta x-squared-- terms that go even smaller-- delta x-cubed, delta x to the 4th from the third derivative, and the fourth derivative, and so forth.

So if the function-- and it turns out this is true. It doesn't even require a Taylor series. This is true any time you have a function that has a second derivative. It may not even be 3 times differentiable. But if it has a second derivative, then you can write these first three terms and then the higher-order term.

So imagine what does this mean for our difference approximation, right? So what are we computing here? We're plotting this relative error. We're plotting f of x plus dx minus f of x. That's our finite difference. And we're subtracting our exact derivative f prime of x dx. Right? But then we're dividing by the exact derivative norm.

But what happens when we plug this Taylor series into here? So you-- well, first of all, the first two terms here are just these first two terms here. It's just f of x. If we move the f of x to the left-hand side, it's gone from there. That's this. OK?

So what's left? Well, there's this term. But this cancels the thing we're subtracting, the exact derivative. So what's left over? These terms. OK? It is the terms proportional to delta x-squared plus higher-order terms. Right?

So does everyone follow that? So I know, doing math like this from slides, I need to pause and make sure everyone has a chance to stare at the formula. Because it's easy to kind of scroll faster than you can read compared to when I write it on the blackboard or on a notebook, right?

So all I'm doing is plugging-- and you can think of this as a Taylor series-- into my finite difference formula and thinking about my error, my finite difference minus exact over exact. And this finite difference minus exact-- the first two terms cancel. And what's left over are these terms that go, like, delta x-squared and higher-order terms.

But then we're dividing it by a denominator that's f of x dx is proportional to delta x. So if I take a term that's proportional to delta x-squared plus higher-order terms and divide it by a term that's proportional to delta x, what you get are terms that are proportional to delta x from the first term plus higher-order terms. Right?

So what we expect is exactly this. We expect our relative error to be approximately linear in delta x plus terms that are higher-order that vanish as delta x-- that are negligible as delta X gets small. Does that make sense?

So this is kind of the inherent error in the finite difference approximation. We expect this forward difference formula to have errors that decrease linearly with delta x. This is called the truncation error, the error that you get because you truncated the Taylor series. Because delta x is not infinitesimal.

And that's exactly what we're seeing in this plot in the right-hand side. We're seeing it's actually almost perfectly linear as we decrease delta A. So that's what the math predicts.

So this deviation here is not predicted by that math. And it would not happen if we were doing exact arithmetic with real numbers. So what we're-- this increase over here is due to our roundoff errors. As I said, it's due to the fact that we're not doing exact math, at least exact real number math on a computer. We're doing only approximate real number mathematics.

And that's called floating-point arithmetic. And so the basic thing is-- you can think of it this way. So a computer, right-- it cannot store-- if you have an arbitrary real number, you would need infinitely many digits to store it, right? A computer doesn't have infinite memory. And even working with a huge number of digits would be very expensive, right?

And so a computer-- when it's working with numbers, by default, it keeps only a relatively small number of significant digits-- I mean relatively small compared to infinity. It keeps about 15 decimal digits, which is a lot, right? That's a lot more than you would carry in a hand calculation, right?

And with 15 decimal digits, it can do arithmetic very, very quickly. It's built into the CPU. That's why everyone uses this same number of digits. Whether you're in Matlab, or in Python, or in Julia, or C, or Fortran, this is kind of the default. It's called double-precision floating-point arithmetic. And so everyone uses the same thing most of the time, at least.

And so this is the source of that increase. It's these things called roundoff errors from this finite number of significant digits. And so let's think in a little bit more detail of how that happens.

So what happens if delta x is too small? So then f of x plus delta x and f of x are almost the same number, right? They're almost the same. And when you subtract them, right, then most of the significant digits cancel one another. OK?

And in fact, even when you take x plus dx, if x is really small-- if x is only storing 15 digits and you add a delta x that's 10 to the minus 100th, x plus dx will just give you x. Because it'll just round off x plus dx back to x. It won't be able to store that 1 in the 100th decimal place. Right?

So in general, this is called a catastrophic cancellation-- when you take two nearly equal numbers and you subtract them to finite, but they each have finite accuracy. You subtract them. Most of the significant digits cancel. You're left with only a few significant digits or maybe even no significant digits. Maybe if they're small enough, this difference might just be 0.

So it's easier to see with a simpler function like-- not a matrix function, but just a scalar function. So that's just sine of x. OK? At x equals 1, all right, is it the exact derivative of just cosine of 1, right? And so we can imagine our finite difference approximation is just f of x plus dx. f is just sine minus fx over delta x plus first-order terms, right? That's what happens when I take this formula and just divide it through by delta x, right?

So if dx equals 1e to the minus 5, I get 0.54 something. The cosine of 1-- the exact answer is 0.54 something. But you can see they differ in the third digit already. The relative error is actually 10 to the minus 6. So it's actually pretty good here.

Why is it 10 to the minus-- oh, because it's really not the third digit. It's .540298. And this is "302." So they really differ in almost, like, the sixth digit here versus there. It's-- right? And that's differing by 4 in the sixth digit, right?

And so now what happens if you do delta x equals 10 to the minus 100th, right? Mathematically, that should give you a much better answer. But actually, it just gives you 0. And the computer is perfectly capable of representing 10 to the minus 100th.

So it's basically floating-point. It stores a fixed number of digits and then an exponent. You can think of it as like scientific notation on the computer. So it can store really big, really small numbers just with a finite number of significant digits, right?

But if you do 1 plus 10 to the minus 100th, you get 1. Because what would the correct answer be? The correct answer would be 1 plus 10 to the minus 100th is 1, a decimal point, 99 0's, and then a 1, right? And if the computer is only storing 15 digits, it cannot store 1, 99 0's, and then a 1, right?

And what it does is it takes this result. And effectively, it rounds it to the closest number that it can represent with 15 digits. And the closest number is 1. So that's what it gives you.

So even if you do something less extreme-- so if you do-- it's storing 15 digits. So if you do 10 to the minus 13th for your dx, then 1 plus dx it can represent, right? So then the computer can store this number, right? But if you take sine of 1 plus dx and sine of 1, right, you can see that they're now-- if dx is 10 to the minus 13th, they're almost the same number-- this and this. See? They only differ here, in the last four significant digits.

So when you subtract them in a finite difference approximation and take this minus this, almost all the significant digits cancel. You have 16 digits here-- 15 digits. But 11 of them cancel. And you're just left with only four digits of accuracy. And so if you compute the relative error, you get 10 to the minus 3 from those four digits of accuracy.

So this is the general problem with finite difference is that they're not completely bulletproof. You need to make delta x small enough that the truncation error is small but not so small that the roundoff errors kill you, right? And kind of a rule of thumb is that it works-- it's not perfect, like any rule of thumb. No rule of thumb is perfectly reliable. A good rule of thumb is to use about half the digits for your difference.

So more precisely, the number of significant digits on the computer is represented by this quantity called the machine epsilon. So basically, epsilon x is the size of the last significant digit. Or the next floating-point number out of x-- after x is x times 1 plus epsilon. So in Julia, there's a function called eps that returns this. It's about 10 to the minus 16th. So as I said, it's about 15 or 16 decimal digits.

The reason this is kind of a weird-looking number-- it's not 1e to the minus 16th or 1e to the minus 15th-- is because the computer is actually not storing decimal digits. It's storing binary digits.

So it's actually storing 1.000-- a whole bunch of digits times an exponent, which is a power of 2. And the digits are actually binary digits. So this is not actually 10 to the minus 15th. It's 2 to the minus 52. But it's a crude approximation. You can think of it as 15 decimal digits, right?

And so a crude rule of thumb for finding differences-- it's kind of a good median-- for a small truncation error but not big roundoff errors is to choose your delta x to be about half the significant digits of x. So basically, the magnitude of delta x is on the order of square root of epsilon times x. So square root of epsilon is about 10 to the minus 8th.

So that's why I chose 10 to the minus 8th. My matrix A was random numbers on order one. So my matrix dA, I made them random numbers on order 10 to minus 8th. And that was-- if you go back up here, 10 to the minus 8th is about where we see this crossover point happening. It's about the point where the truncation error is balancing the roundoff error.

And so these aren't perfect rules of thumb, but it's-- and if you have to pick a delta x quickly, that's-- let's see. That's a good way to choose that, right? So--

ALAN EDELMAN: Steven, let me point out that we're just about out of time.

STEVEN JOHNSON: Yep. And I'll show this, too-- just the same thing for the sine plot. So this is the same thing for difference. The dots are the relative error compared to the exact derivative, which is cosine of 1. And the line is just a straight line for reference. You see exactly the same thing.

It's, as delta x gets smaller, initially the error gets smaller because the finite difference is getting better. And it goes down linearly because this is a first-order-accurate formula. But eventually, when delta x gets too small, the cancellation error, the roundoff error kills you. And the error gets worse. And the crossover point-- again, that rule of thumb-- works pretty well. It's about 10 to the minus 8th in here for this, since x is 1.

So I'll stop there. And we'll see you again on Wednesday. And I'll post this notebook. And we'll post-- I'll get Alan's updated notebook from him. And we'll post that on the website as well.

ALAN EDELMAN: Sounds good.