[SQUEAKING]
[RUSTLING]
[CLICKING]
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.