Generalizing Automatic Differentiation to Automatic Sparsity, Uncertainty, Stability, and Parallelism

Christopher Rackauckas[1]

[1]Affiliation not available

April 17, 2023

the WINNOWER

COMPUTER SCIENCES

Generalizing Automatic Differentiation to Automatic Sparsity, Uncertainty, Stability, and Parallelism

CHRISTOPHER RACKAUCKAS

CORRESPONDENCE:
me@chrisrackauckas.com

DATE RECEIVED:
May 18, 2021

DOI:
10.15200/winn.162133.38896

ARCHIVED:
May 18, 2021

CITATION:
Christopher Rackauckas, Generalizing Automatic Differentiation to Automatic Sparsity, Uncertainty, Stability, and Parallelism, *The Winnower* 8:e162133.38896 , 2021 , DOI: 10.15200/winn.162133.38896

Automatic differentiation is a "compiler trick" whereby a code that calculates f(x) is transformed into a code that calculates f'(x). This trick and its two forms, forward and reverse mode automatic differentiation, have become the pervasive backbone behind all of the machine learning libraries. If you ask what PyTorch or Flux.jl is doing that's special, the answer is really that it's doing automatic differentiation over some functions.

What I want to dig into in this blog post is a simple question: what is the trick behind automatic differentiation, why is it always differentiation, and are there other mathematical problems we can be focusing this trick towards? While very technical discussions on this can be found in our recent paper titled "ModelingToolkit: A Composable Graph Transformation System \nFor Equation-Based Modeling" and descriptions of methods like intrusive uncertainty quantification, I want to give a high-level overview that really describes some of the intuition behind the technical thoughts. Let's dive in!

WHAT IS THE TRICK BEHIND AUTOMATIC DIFFERENTIATION? NON-STANDARD INTERPRETATION

To understand automatic differentiation in practice, you need to understand that it's at its core a code transformation process. While mathematically it comes down to being about Jacobian-vector products and Jacobian-transpose-vector products for forward and reverse mode respectively, I think sometimes that mathematical treatment glosses over the practical point that it's really about code.

Take for example $f(x) = \sin(x)$. If we want to take the derivative of this, then we could do $(f(x+h) - f(x))/h$, but this misses the information that we actually know analytically how to define the derivative! Using the principle that algorithm efficiency comes from problem information, we can improve this process by directly embedding that analytical solution into our process. So we come to the first principle of automatic differentiation:

If you know the analytical solution to the derivative, then replace the function with its derivative

So if you see $f(x) = \sin(x)$ and someone calls ``derivative(f,x)``, you can do a quick little lookup to a table of rules, known as primitives, and if it's in your table then boom you're done. Swap it in, call it a day.

This already shows you that, with automatic differentiation, we cannot think of f as just a function, just a thing that takes in values, but we have to know something about what it means semantically. We have to look at it and identify "this is sin" in order to know "replace it with cos". This is the fundamental limitation of automatic differentiation: it has to know something about your code, more information than it takes to call or run your code. This is why many automatic differentiation libraries are tied to specific

implementations of underlying numerical primitives. PyTorch understands ``torch.sin`` as $\sin$, but it does not understand ``tf.sin`` as $\sin$, which is why if you place a TensorFlow function into a PyTorch training loop you will get an error thrown about the derivative calculation. This semantic mapping is the reason for libraries like ChainRules.jl which define semantic mappings for the Julia Base library and allows extensions: by directly knowing this mapping on all of standard Julia Base, you can cover the language and achieve "differentiable programming", i.e. all programs automatically can get derivatives.

But we're not done. Let's say we have $f(x) = \cos(\sin(x))$. The answer is not to add this new function to the table by deriving it by hand: instead we have to come up with a way to make a function $f(x) = h(g(x))$ generate a derivative code whenever h and g are in our lookup table. The answer comes from the chain rule. I'm going to describe the forward application of the chain rule as it's a bit simpler to derive, but a full derivation of how this is done in the reverse form is described in these lecture notes. The chain rule tells us that $f'(x) = h'(g(x))g'(x)$. Thus in order to calculate f', we need to know two things: $g(x)$ and $g'(x)$. If we calculate both of these quantities at every stage of our code, it doesn't matter how deep the composition goes, we will have all of the information that is required to reconstruct the result of the chain rule.

What this means is that automatic differentiation on this function can be thought of as the following translation process:

1. Transform $\sin(x)$ to $y = (\sin(x), \cos(x))$ and evaluate at x
2. Transform $\cos(y)$ to $z = (\cos(y[1]), -\sin(y[1]))$ and evaluate at y
3. Transform $f(x)$ to $(f(x), f'(x)) = (z[1], z[2] * y[2])$. Now the second portion is the solution to the derivative

This translation process is "transform every primitive function into a tuple of (function,derivative), and transform every other function into a chain rule application using the two pieces" is **non-standard interpretation**. This is the process where an interpreter of a code or language runs under different semantics. An interpreter written to do this process acts on the same code but interprets it differently: it changes each operation to a tuple of the solution and its derivative, instead of just the solution $f(x)$.

Thus the non-standard interpretation version of the problem of calculating derivatives is to reimagine the problem as "at this step of the code, how should I be transforming it so that I have the information to calculate derivatives"? There are many ways to do this abstract interpretation process to a non-standard interpretation: operator overloading, prior static analysis to generate a new source code, etc. But there's one question we should bring up.

WHY DO WE SEE IT ALWAYS ON DIFFERENTIATION? WHY IS THERE NO AUTOMATIC INTEGRATION?

One way to start digging into this question is to answer a related question people pose to me often: if we have automatic differentiation, why do we not have automatic integration? While at face value it seems like the two should be analogues, digging deeper exposes what's special about differentiation. If we wanted to do the integral of $f(x) = \sin(x)$, then yes we can replace this with $\cos(x)$. The heart of the question is to ask about the chain rule: what's the integral of $f(x) = h(g(x))$? It turns out that there is no general rule for the "anti-chain rule". A commonly known result is that the standard Gaussian probability distribution, $f(x) = e^{-x^2}$, does not have a solution to its antiderivative that can be written with elementary functions, and that's just the case of $g(x) = x^2$ and $h(x) = e^x$. While that is true, I don't think that captures all that is different about integrals.

When I said "we can replace this with $cos(x)$" I was actually wrong: the antiderivative of $sin(x)$ is not $cos(x)$, it's $\cos(x) + C$. There is no unique solution without imposing some external context or some global information like "and F(x)=0". Differentiation is special because it's purely local: only knowing the value of x I can know the derivative of $f(x)$. Integration is a well-known example of a non-local operation in mathematics: in order to know the anti-derivative at a value of x, you might need to know information about some value $y \neq x$, and sometimes it's not necessarily obvious what that value y

should even be. This nonlocality manifests in other ways as well: while $f(x) = e^{-x^2}$ is not integrable, $f(x) = xe^{-x^2}$ is easy to solve via a u-substitution, making $u = -x^2$ and cancelling out the x in-front of the e. So there is no chain rule not because some things don't have an antiderivative, but because you have nonlocality, so $f(x) = h(g(x))$ can be non-integrable while $f(x) = h(g(x))^* j(x)$ is. There is no chain rule because you can't look at small pieces and transform them, instead you have to look at the problem holistically.

But this gives us a framework in order to judge whether a mathematical problem is amenable to being solved in the framework of non-standard interpretation: it must be local so that we can define a step-by-step transformation algorithm, or we need to include/impose some form of context if we have alternative information.

Let's look at a few related problems that can be solved with this trick.

AUTOMATIC SPARSITY DETECTION IN JACOBIANS

Recall that the Jacobian is the matrix of partial derivatives, i.e. for $y = f(x)$ where x and y are vectors, it's the matrix of terms $\frac{dy_i}{dx_j}$. This matrix shows up in tons of mathematical algorithms, and in many cases it's sparse, so it's common problem to try and compute the sparsity pattern of a Jacobian. But what does this sparsity pattern mean? If you write out the analytical solution to y, a zero in the Jacobian means that y_i is not a function of x_j. In other words, x_j has no influence on y_i. For an arbitrary program f, can we use non-standard interpretation to calculate whether x_j influences y_i?

It turns out that if we make this question a little simpler then it has a simple solution. Let's instead ask, can we use non-standard interpretation to calculate whether x_j can influence y_i? The reason for this change is because the previous question was non-local: $y_i = \sin^2(x_j) + \cos^2(x_j)$ is programmatically dependent on x_j, but mathematically $\sin^2(x_j) + \cos^2(x_j) = 1$ so you could cancel it out if you have good global knowledge of the program. So "does it influence" this output is hard, but "can it influence" the output is easy. "Can influence" is just the question of "does x_j show up in the calculation at all?"

So we can come up with an non-standard interpretation formulation to solve this problem. Instead of computing values, we can compute "influencer sets". The output of $sin(x_1)$ is influenced by x_1. The output of $\sin(x_1)^*\cos(x_2)$ is influenced by x_1,x_2. For $f(x) = h(g(x))$, the influencer set of f is the same as the influencer set of $g(x)$. So our non-standard interpretation is to replace variables by influencer sets, and whenever the collide by a binary function like multiplication, we make the new influencer set be the union of the two. Otherwise we keep propagating it forward. The result of this way of running the program is that output that say "these are all of the variables which can be influencing this output variable". If x_j never shows up at any stage of the computation of y_i, then there is no way it could ever influence it, and therefore $\frac{dy_i}{dx_j} = 0$. So the sparsity pattern is bounded by the influencer set.

This is the process behind the the automated sparsity tooling of the Julia programming language, which are now embedded as part of Symbolics.jl. There is a bit more you have to do if you see branching: i.e. you have to take all branches and take the union of the influencer sets (so it's clear this cannot be generally solved with just operator overloading because you need to take non-local control of control flow). Details on the full process are described in Sparsity Programming: Automated Sparsity-Aware Optimizations in Differentiable Program\nming, along with extensions to things like Hessians which is all about tracking sets of linear dependence.

AUTOMATIC UNCERTAINTY QUANTIFICATION

Let's say we didn't actually know the value x to put into our program, and instead it was some $x \pm y$. What could we do then? In a standard physics class you probably learned a few rules for uncertainty

propagation: $(a \pm b)+(c \pm d)=(a+c)\pm(b+d)$ and so on. It's good to understand where this all comes from. If you said that x was a random variable from a normal distribution with mean a and standard deviation b, and y was an independent random variable from a normal distribution with mean c and standard deviation d, then $x+y$ would have mean $a+c$ and standard deviation $b+d$. This means there are some local rules for propagating normal distributed random variables! What do you do for $y=f(x)$ for a normally distributed input? You could approximate f with its linear approximation: $y=f'(x)x$ (this is another way to state that the derivative is the tangent vector at x). At a given value of x then we just have a linear equation $y=\alpha x$ for some scalar α, in which case we use the rule $\alpha(a \pm b)=\alpha a \pm \alpha b$. This gives an non-standard interpretation implementation to approximately computing with normal distributions! Now all we have to do is replace function calls with automatic differentiation around the mean and then propagate forward our error bounds.

This is a very crude description which you can expand tolinear error propagation theory where you can more accurately treat the nonlinear propagation of variance. However, this is still missing out on whether two variables are dependent: $x / x = 1 \pm 0$, there's no uncertainty there, so you need to treat that in a special way! If you think about it, "dependence propagation" is very similar to "propagating influencer sets", which you can use to even more accurately propagate the variance terms. This gives rise to the package Measurements.jl which transforms code to make it additionally do propagation of normally distributed uncertainties.

I note in passing that Interval Arithmetic is very similarly formulated as an alternative interpretation of a program. David Sanders has a great tutorial on what this is all about and how to make use of it, so check that out for more information.

USING CONTEXT INFORMATION: AUTOMATIC INDEX REDUCTION IN DAES AND PARALLELISM

Now let's look at solving something a little bit deeper: simulating a pendulum. I know you'll say "but I solved pendulum equations by hand in physics" but sorry to break it to you: you didn't. In an early physics class you will say "all pendulums are one dimensional", and "the angle is small, so sin(x) is approximately x" and arrive a beautiful linear ODE that you analytically solve. But the world isn't that beautiful. So let's look at the full pendulum. Instead you have the location of the swinger (x,y) and its velocity (v_x,v_y). But you also have non-constant tension T, and if we have a rigid rod we know that the distance of the swinger to the origin is constant. So the evolution of the system is in full given by:

$$
\begin{aligned}
x' &= v_x \\
v_x' &= Tx \\
y' &= v_y \\
v_y' &= Ty - g \\
0 &= x^2 + y^2 - L^2
\end{aligned}
$$

There are differential equation solvers that can handle constraint equations, these are methods for solving differential-algebraic equations (DAEs), But if you take this code and give it to pretty much any differential equation solver it will fail. It's not because the differential equation solver is bad, but because of the properties of this equation. See, the derivative of the last equation with respect to T is zero, so you end up getting a singularity in the Newton solve that makes the stepping unstable. This singularity of the algebraic equation with respect to the algebraic variables (i.e. the ones not defined by derivative terms) is known as "higher index". DAE solvers generally only work on index-1 DAEs, and this is an index-3 DAE. What does index-3 mean? It means if you take the last equation, differentiate it twice, and then do a substitution, you get:

$$
\begin{aligned}
x' &= v_x \\
v_x' &= xT \\
y' &= v_y \\
v_y' &= yT - g \\
0 &= 2\left(v_x^2 + v_y^2 + y(yT - g) + Tx^2\right)
\end{aligned}
$$

This is mathematically the same system of equations, but this formulation of the equations doesn't have the index issue, and so if you give this to a numerical solver it will work.

It turns out that you can reimagine this algorithm to be something that is also solved by a form of non-standard interpretation. This is one of the nice unique features of ModelingToolkit.jl, spelled out in ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling. While algorithms have been written before for symbolic equation-based modeling systems, it turns out you can use an non-standard interpretation process to extract the formulation of the equations, solve a graph algorithm to determine how many times you should differentiate which equations, do the differentiation using rules and structures from automatic differentiation libraries, and then regenerate the code for "the better version". As shown in a ModelingToolkit.jl tutorial on this feature, if you do this on the pendulum equations, you can change a code that is too unstable to solve into an equation that is easy enough to solve by the simplest solvers.

And then you can go even further. As I described in the JuliaCon 2020 talk on automated code optimization, now that one is regenerating the code, you can step in and construct a graph of dependencies and automatically compute independent portions simultaneously in the generated code. Thus with no cost to the user, an non-standard interpretation into symbolic graph building can be used to reconstruct an\nd automatically parallelize code. The ModelingToolkit.jl paper takes this even further by showing how a code which is not amenable to parallelism can, after context-specific equation changes like the DAE index reduction, be transformed into an alternative variant that is suddenly embarrassingly parallel.

All of these features require a bit of extra context as to "what equations you're solving" and information like "do you care about maintaining the same exact floating point result", but by adding in the right amount of context, we can extend mathematical non-standard interpretation to solve these alternative problems in new domains.

By the way, if this excites you and you want to see more updates like this,please star ModelingToolkit.jl.

FINAL TRICK: CONSTRUCTING A PDE SOLVER OUT OF AN ODE SOLVER WITH NON-STANDARD INTERPRETATION

Let me end by going a little bit more mathematical. You can transform code about scalars into code about functions by using the vector-space interpretation of a function. In mathematical terms, functions f are vectors in Banach spaces. Or, in the space of functions, functions are points. if you have a function f and a function g, then $h = f + g$ is a function too, and so is $h = f * g$. You can do computation on these "points" by working out their expansions. For example, you can write f and g in their Fourier series expansions: $f(x) = a_0 + \sum a_n \cos(nx) + b_n \sin(nx)$. Approximating with finitely many expansion terms, you can represent f via [a[1:n];b[1:n]], and same for g. The representation of $h = f + g$ can be worked out from the finite truncation (just add the coefficients), and so can $h = f * g$. So you can transform your computation about "functions" to "arrays of coefficients representing functions", and derive the results for what $+$, $*$, etc. do on these values. This is an non-standard interpretation of a program that transforms it into an equivalent program about function and measures as inputs.

Now it turns out you can formally use this to do cool things. A partial differential equation (PDE) is an ODE where where instead of your values being scalars at each time $u(t)$, your values are now functions at each time $(u(x))(t)$. So what if you represent those "functions" as "scalars" via their representation in the Sobolev space, and then put those "scalars" into the ODE solver? You automatically transform your ODE solver code into a PDE solver code. Formally, this is using a branch of PDE theory known as semigroup theory and making it a computable object.

In turns out this is something you can do. ApproxFun.jl defines types ``Fun`` which represent the functions as scalars in a Banach space, and defines a bunch of operations that are allowed for such

functions. I showed in a previous talk that you can slap these into DifferentialEquations.jl to have it reinterpret into this function-based differential equation solver, and then start to solve PDEs via this representation.

CONCLUSION: NON-STANDARD INTERPRETATION IS POWERFUL FOR MATHEMATICS

Automatic differentiation gets all of the attention, but its really this idea of non-standard interpretation that we should be looking at. "How do I change the semantics of this program to solve another problem?" is a very powerful approach. In computer science it's often used for debugging: recompile this program into the debugging version. And in machine learning it's often used to recompile programs into derivative calculators. But uncertainty quantification, fast sparsity detection, automatic stabilization and parallelization of differential-algebraic equations, and automatic generation of PDE solvers all arise from the same little trick. That can't be all there is out there: the formalism and theory of non-standard interpretation seems like it could be a much richer field.

BIBLIOGRAPHY

1. ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling
2. Sparsity Programming: Automated Sparsity-Aware Optimizations in Differentiable Program\nming
3. Uncertainty propagation with functionally correlated quantities