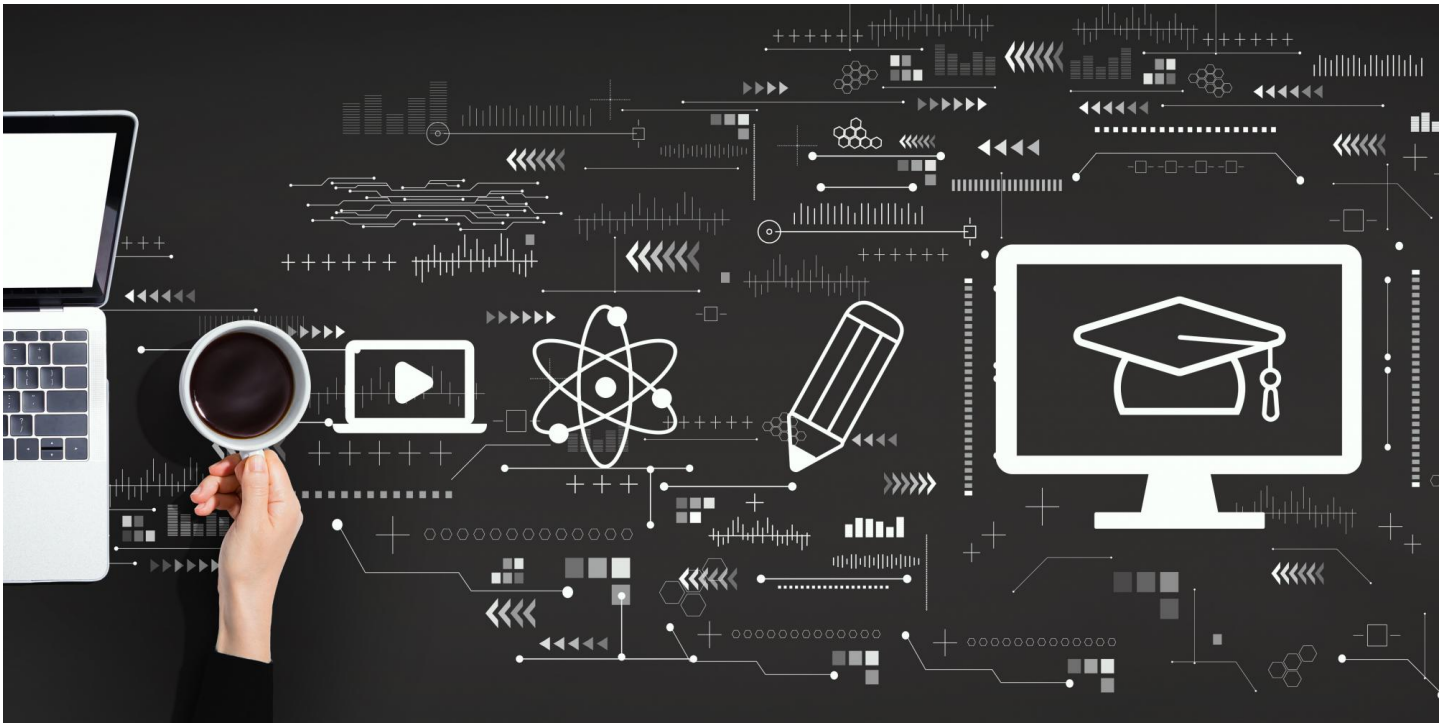


Algorithmic Differentiation Masterclass 1

Published 04/08/2020 By Jacques Du Toit

Introduction to Algorithmic Differentiation (AD) and follow-up questions

This article follows up discussions which arose from the AD Masterclass 1 delivered on 30 July 2020 and is meant for attendees of the session.

What is AD?

AD is a technique to differentiate computer code. Formally, it is a **transformation** of one source code into another. It is important to realise th

- AD takes as input a computer code
- AD produces as output a computer code
- The output computer code has something to do with differentiating the input computer code. We'll try and make this notion precise below.

The fundamental questions of AD, broadly speaking, are

1. What must the output code compute?
2. How must the **transformation** actually work? In other words, how do we actually produce the output code, given an input code?

Differentiability

Let's ignore for the moment the question of what the output AD code should compute, and let's just assume that it somehow computes a mathematical derivative.

We only differentiate differentiable functions, right?

In high school, we all learn that we can only differentiate functions that are differentiable. If we differentiate the non-differentiable, well then we need to ask ourselves some basic questions. What are we doing? Is the function differentiable almost everywhere, except perhaps at some set of points? If so, then we might be in good shape, as long as we avoid the points of non-differentiability.

If the function is non-differentiable in some stronger sense, then we should pause and think why we're trying to differentiate it. What are we intending to do with these derivatives? Will they be suitable for that purpose?

This discussion may seem a little silly at first sight. The problem is that a lot of computer code is non-differentiable. The functions max(), min() and abs() are non-differentiable. Control flow such as if-statements almost always introduce non-differentiabilities if the conditional depends on floating-point data. They can easily introduce non-differentiabilities even if the conditional *doesn't* depend on floating-point data.

Interchanging Limits

Back when we learned Real Analysis, we studied limits of the form

$$\lim_{n\to\infty} f_n(\eta) = f(\eta)$$

and we examined conditions under which such limits might exist. We also looked at what we might be able to conclude about f, for example if f_n is continuous, does this mean that f is continuous? (No, it doesn't). We then moved on to look at limits of the form

$$\lim_{m\to\infty}\lim_{n\to\infty} f_{n,m}(\eta) = f(\eta)$$

and we considered the question of whether we could interchage these limits, i.e. whether

$$\lim_{m\to\infty}\lim_{n\to\infty} f_{n,m}(\eta) = \lim_{n\to\infty}\lim_{m\to\infty} f_{n,m}(\eta)$$

In general, this is a delicate question! There are plenty of examples where this sort of thing isn't true.

So what does this have to do with AD and differentiability? Typically, computer programs are **approximations** of some sort. We create a computer program $f_n(\eta)$ which approximates the actual mathematical object we care about $f(\eta)$. An example might be a PDE solver. We discretise the PDE in some way, we analyse the numerical scheme, we prove convergence of the numerical scheme f_n to the mathematical object f, and we know we're on good ground.

We then apply AD to our simulation code f_n, and immediately we are in fact asserting that

$$\frac{\partial}{\partial\eta} f(\eta) = \frac{\partial}{\partial\eta}\lim_{n\to\infty} f_n(\eta) = \lim_{n\to\infty}\frac{\partial}{\partial\eta} f_n(\eta)$$

which is a very strong statement, and one which would probably cause a pure mathematician some alarm. We have *interchanged the order of the limits* since differentiation is just a limit operation. Given how delicate this operation can be, we should take a moment to admit to ourselves that we doing something which probably merits closer attention.

Understand where the non-differentiability is coming from

When dealing with non-differentiable code it is important to try to understand where the non-differentiability is coming from. Is the mathematical object f differentiable or not? If f is differentiable, then, of course, there is no need in general for each f_n to be differentiable. A common example of where this sort of thing happens is in Monte Carlo when we approximate

$$f(\eta) = \mathbb{E}\big[g(X(\eta))\big] \approx \frac{1}{n}\sum_{i=1}^{n} g(X_i(\eta)) = f_n(\eta)$$

for some stochastic process X. If g is, for example, a Heavyside step function, then applying AD to the computer code will yield a derivative of zero. The mathematical object is differentiable in η (assuming the distribution of X depends on it smoothly), so we know differentiating f makes sense. The problem is just in f_n. If f is differentiable but f_n is not, then we might be able to fix f_n through some kind of smoothing. One might consider dropping in some kind of smoothing function (sigmoid) or smoothing kernel (Gaussian). Usually, it's best to go back to the mathematical object f and see whether one can use its properties to somehow derive a better (smooth) approximation f_n. This way one incorporates problem-specific information, which almost always gives a more powerful approach (at the cost of generality, of course).

Suppose, however, that the mathematical object we are considering is *not* differentiable. While we can create a smooth f_n and differentiate it, in what sense would the limit of such a derivative be useful? Why am I trying to differentiate f when in fact it's not possible to do so? What am I hoping to do with that derivative? Does my use case make sense? It might well do.

One might also adopt the age-old mathematical maxim of changing the problem. If f is not differentiable, perhaps I can replace it with g which *is* differentiable, and use the derivatives of g wherever I wanted to use the derivatives of f.

Conclusion: it is problem-specific

As you can see from our discussion above, there are more questions than answers, and unfortunately, it depends on what you are trying to do. I would urge everyone who is considering using AD to think a bit about differentiability before jumping into the AD machine. The use of sigmoid-type smoothers has been successful in several applications, it may work for you. Vibrato Monte Carlo has looked specifically at the problem of expectations of non-differentiable functionals of diffusion processes and has exploited the smoothness of the diffusion kernel. Ideas from fuzzy logic have also been re-cast as smoothers and have been applied in Monte Carlo simulations.

People sometimes say that finite differences can give them smoother derivatives than AD since they can remove some of the underlying "noise". While certainly a true statement, it depends a bit on what you view as "noise" and what you view as "signal". Usually, the source of this noise is some kind of

non-differentiability of the underlying code f_n. While manageable for first-order derivatives (Jacobians), finding suitable finite difference estimates for higher-order derivatives (Hessians, ...) can become challenging.

As you will learn in subsequent webinars, AD doesn't know whether your code is differentiable. AD effectively ignores all control flow and only looks at calculations on floating-point data. Hence AD will compute an exact local derivative of your code at a point η, *assuming your code is differentiable at* η

Accuracy of AD vs analytic derivatives

This question arises quite often. How accurate is AD compared with analytic adjoints? Here the question is being asked about a given code $f_n(\eta)$, comparing an "AD derivative" of f_n with an "analytic derivative" of f_n. Asking how ∂f compares with AD applied to f_n is not really what I'm talking about here since that depends on how close to convergence f_n is and whether the derivatives of f_n actually have anything to do with the derivatives of f.

As you will learn in subsequent webinars, AD is basically a local application of the chain rule. If you look at any computer code as a sequence of operations on floating-point data, then all operations on that data can be broken down into

- fundamental operators (+,-,/,·)
- primitive functions (sin(), cos(), exp(), ...) which are basically the maths functions in libm

AD knows the analytic derivatives of all these things and combines them using the chain rule, just as students at high school learn how to differentiate more complex expressions. In this sense, AD is mathematically exact.

However, in the real world of finite precision arithmetic, mathematically exact isn't always enough. A treatment of floating-point arithmetic is outside the scope of these lectures, but I would urge interested readers to look at this widely cited paper (https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html).

The summary: in infinite precision arithmetic, AD would compute exact mathematical derivatives. In the real world, the program produced by AD is subject to the same round-off and truncation effects as any other numerical code, and the way these quantities interact can be very subtle. The answer that is generally given is that "AD is exact up to machine precision", which is a statement that may be intuitively appealing, but would take many many pag to unpack in a rigorous manner.

The Tangent and Adjoint Models

Let's return to our first question, namely, what should the output code of AD compute? The answer to this is encapsulated in the two basic models of AD, the tangent model and the adjoint model.

Let us consider a function $y = f(x)$ where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$. Then the tangent model computes the Jacobian-vector product

$$J_f \cdot \vec{x} = \left[\frac{\partial f}{\partial x}\right] \vec{x}$$

for any $\vec{x} \in \mathbb{R}^n$. In other words, the tangent model takes as input $x, \vec{x}$ and produces as output the value y and the Jacobian-vector product with $\vec{x}$.

The adjoint model computes the transpose-Jacobian-vector product

$$J_f^T \cdot \vec{y} = \left[\frac{\partial f}{\partial x}\right]^T \vec{y}$$

for any $\vec{y} \in \mathbb{R}^m$. In other words, the adjoint model takes as input $x, \vec{y}$ and produces as output the value y and the transpose-Jacobian-vector product with $\vec{y}$.

Why do these models return products between Jacobians and vectors, rather than Jacobians directly? The answer is so that both AD models are *closed under function composition*. If one has a tangent model of f and a separate tangent model of g, then the tangent model of $f(g)$ is the tangent model of f composed with the tangent model of g. You can check that the same is true for the adjoint model.

While these definitions are very simple, one needs to look deeper to understand their implications. We will dig into these things in detail in the second lecture, but there is no harm in giving a little preview. Consider a chain of operations $y = h(g(f(x)))$ where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$. The tangent model of this is $J_h(J_g(J_f \cdot \vec{x}))$ or if we write it out slightly differently

$$J_f \cdot \vec{x} \longrightarrow J_g \cdot (J_f \cdot \vec{x}) \longrightarrow J_h \cdot (J_g \cdot (J_f \cdot \vec{x}))$$

Similarly, the adjoint model is $J_f^T \cdot \left(J_g^T \cdot \left(J_h^T \cdot \vec{y}\right)\right)$, or written slightly differently,

$$J_f^T \cdot \left(J_g^T \cdot \left(J_h^T \cdot \vec{y}\right)\right) \longleftarrow J_g^T \cdot \left(J_h^T \cdot \vec{y}\right) \longleftarrow J_h^T \cdot \vec{y}$$

What we notice above is that the *order of computation* of the two models is different. The tangent model follows the same order of computation as the original program: $f \to g \to h$. As we'll see in subsequent sessions, this will mean that tangent mode AD is very simple to implement.

However, the adjoint model follows a different order of computation: $h \to g \to f$ brought about by the transpose in its definition. The adjoint model, in fact, *walks backwards* through the program. In addition, it is clear that if f, g, h above are non-linear, then computing J_g for example will require the value of $f(x)$, and similarly computing J_h will require the value of $g(f(x))$. So the adjoint model is in some sense even stranger - in order to walk *backwards* it must have quantities available which would only be known once the program has been run *forwards*.

This observation forms the basis of our understanding of the adjoint model. Certainly, one can imagine a few ways of achieving this. For example, we can run our program forwards, computing each Jacobian we need and storing it, and then using them in the backward pass. Indeed, we could even multiply up the Jacobians in the forward pass as we go, and then the backward pass consists of a single matrix-vector product. However, this isn't a good idea since the forward pass would then consist of matrix-matrix products, which are expensive. The adjoint model itself only requires matrix-vector products.

Perhaps we can get by without actually storing the Jacobian matrices. Perhaps instead of storing $g(f(x))$ we could try to recompute $g(f(x))$ in the backward pass. And so on.

In general, the adjoint model is a data flow reversal problem and there are multiple different ways of achieving it. Using this flexibility is absolutely essential in creating an efficient implementation of the adjoint of a non-trivial code.

Are there other models of AD?

This question sometimes gets asked. So we have tangent and adjoint models: are there other models? Not really by name, but yes, there are other ways of looking at AD.

I'm not going to discuss this in any depth or rigour, but let's see if we can at least get a heuristic understanding. AD is really computation on a graph. Vertices represent program variables and edges represent local partial derivatives between the program variables. If one asks, for example, how to compute a Jacobian-vector product, then that corresponds to a particular

operation that one needs to perform on that graph. The tangent model outlined above is one way of doing that and proceeds by sweeping forward through the graph, propagating the matrix-vector product.

However, one can also examine the structure of the graph to try and optimize the execution of the operation that needs to be performed. For certain shapes of graph, one can eliminate internal vertices by combining edges to *reduce the number of multiply-adds* that need to be performed by the operation overall.

This kind of graph optimization is combinatorial in nature, and finding the optimal vertex elimination for a given operation on a graph is known to be NP complete (that means it's hard to do). Since the tangent and adjoint models are just two ways of operating on the graph, graph optimization will, in theory, deliver an algorithm that has fewer multiplications and additions than either of these two models. However *implementing* that optimal graph operation may not be straightforward at all, for all the same reasons that implementing an adjoint is not straightforward. On modern computing architectures, doing floating-point arithmetic is *vastly* cheaper than storing and retrieving data from memory.

For this reason, graph optimization is not done very often. However, we do play around with the graph's structure for other reasons. As we'll see in subsequent sessions, adjoint codes are very memory intensive and a key task is controlling memory use. *Jacobian preaccumulation* is a technique that collapses sections of the computational graph into local Jacobians. What this does is replace a given block of vertices and edges by a sparse matrix, with the aim of reducing memory use. We will cover Jacobian preaccumulation in a future session.

Validation

The question often arises how to validate (check for correctness) a given AD implementation. This is the subject of an entire webinar, so I won't say much here. The short answer is: yes, it is possible to check for correctness without having a finite difference estimate of your program. Indeed, given how difficult it can be to choose the correct perturbation size h, it's not a good idea to check an AD program with finite differences unless you are experienced in getting good finite difference estimates of your code.

The way to validate an AD implementation is to verify the *tangent-adjoint identity*. Recall that the tangent model computes $J_f \cdot \vec{x} \in \mathbb{R}^m$ and the adjoint model computes $J_f^T \cdot \vec{y} \in \mathbb{R}^n$. Since the transpose of a scalar is just itself, we see that

$$\vec{y}^T \cdot \left(J_f \cdot \vec{x}\right) = \left(\vec{y}^T \cdot \left(J_f \cdot \vec{x}\right)\right)^T = \vec{x}^T \cdot \left(J_f^T \cdot \vec{y}\right)$$

for every $\vec{x} \in \mathbb{R}^n$ and $\vec{y} \in \mathbb{R}^m$. So to validate an AD implementation you need both the tangent and adjoint models of the code. You seed with arbitrary vectors $\vec{x}$ and $\vec{y}$, run both codes, and then compute the identity above and check that it holds.

In theory, this identity should be checked for a certain number of (linearly independent) seed vectors in the input and output spaces, but in practice, you'll typically find that if you've made a mistake then any random (non-zero) vectors will show it, and conversely, if a given set of random vectors give equality above, then it's highly likely the codes are correct.

GPUs

The question of GPUs came up a few times. Can one use AD on GPUs? Yes, indeed one can. However, we need to be sure we're all talking about the same thing. When the AD community talks about AD on GPUs, they often (but not always) mean *symbolic adjoints*.

Machine learning frameworks like Tensorflow and PyTorch are essentially dealing with known symbolic functions which are evaluated on the GPU. There is a small amount of "user code" there, namely the activation function and perhaps a few other bits and pieces, and the packages know how to differentiate these symbolically, but the majority of the work is simple tensor products. So the packages know what the symbolic adjoints of the neural networks are, and it is these symbolic adjoints that they evaluate on the GPU. Such symbolic adjoints are very efficient since there is no need to create or analyse a computational graph.

However what practitioners in finance and engineering often mean when they talk about "AD on GPU" is AD of *arbitrary simulation code* which happens to be running on a GPU. These are not codes for which symbolic adjoints can be derived (humanely), and there is a need for an AD tool to handle the code which is running on the GPU.

And indeed this is also possible, however, it is not so simple as for a CPU. NAG has an AD tool for CUDA and CPU (C++11) called dco/map. A discussion of dco/map is beyond the scope of the webinar series, but a few words are perhaps in order. The problem with accelerators is, well, they need to accelerate. When they stop accelerating, they stop being accelerators. Hence a very inefficient accelerator code is pretty much pointless. No-one writes CU for fun (*ahem* well some of us do!), we write CUDA because there is a need

make something go significantly faster. While things have become simpler over the years, writing accelerator code is hard, so the benefits have to be worth the pain.

The implication of this is that an AD tool targeting accelerators must also produce efficient *accelerated* tangent and adjoint code. Our dco/map manages this, but the trade-off is that the user needs to know AD very well, and needs to understand the target platform (CPU/CUDA) pretty well too. Consequently, the tool is significantly harder to use than dco/c++. It's a specialist tool.

Doing (discrete) adjoints on GPUs is difficult because of the ratio of memory to the number of threads. A modern compute-grade Tesla GPU such as a V100 has 32GB memory and just over 5000 cores, giving an average of just 6.4MB of main memory per core. Gaming cards will have much less than this. In comparison, a CPU workstation might easily have 128GB of main memory and 64 cores, giving an average of 2GB main memory per core.

This drastic discrepancy in the available memory per core is a huge problem for adjoint AD and prompted us to take an entirely different approach with dco/map. While dco/c++ uses operator overloading to build the computational graph in memory (we call this the *tape*), dco/map works as hard as it can to be tape-free. Not having a tape means you don't have to store tons of data to memory, but it means you need to figure out a different mechanism by which to reverse the computational graph when computing adjoints. This we managed to do (we need to place some restrictions on the user), and the performance is pretty good. The tool is being used in production for Monte Carlo/XVA type codes. During in-house testing on a prototype Monte Carlo code (G2++ model driving LIBOR-type calculations) we managed to achieve an adjoint factor of just over 3x.

Black Boxes

As we've seen, AD is a transformation from one source code into another. What do you do, then, if your source code calls something for which you don't have source code? This happens when code calls 3rd party libraries, which the AD community refer to as "black boxes".

Without access to the source code of the black box, your options are somewhat limited:

- Build a local Jacobian with finite differences and then try to insert this in your AD calculation. Your AD tool needs to support this (dco/c++ doe

- Look up the mathematics that your black box is doing and derive a handwritten tangent or adjoint of that (fun!). Then insert that into your AD calculation. Your AD tool needs to support this (dco/c++ does)
- Persuade the vendor of your black box to also supply you with tangent and adjoint versions of the black box. Of course, this is first prize. Use these in your AD calculation. Your AD tool needs to support this (dco/c++ does)

NAG AD Library

The NAG AD Library is a set of NAG Library (/content/nag-library) routines for which NAG has produced tangents and adjoints. The routines ship as part of the regular NAG Library and although produced with a combination of dco/fortran and handwritten/symbolic techniques, the AD routines can be used with any AD tool. Interfaces are provided which are binary compatible with dco/c++ (/content/algorithmic-differentiation-software), so that it is straightforward to use the routines in a simulation code which is being differentiated with dco/c++.

From the preceding discussion, the need for the AD Library is clear. Customers use NAG Library routines in their codes. If they then want to differentiate those codes, the NAG routines are black boxes. Thankfully, our customers don't have to resort to the workarounds mentioned above: they win first prize! Their library vendor has a deep understanding of AD and has done the necessary work to expose the entire computational graph to the AD tool.

Benchmarks

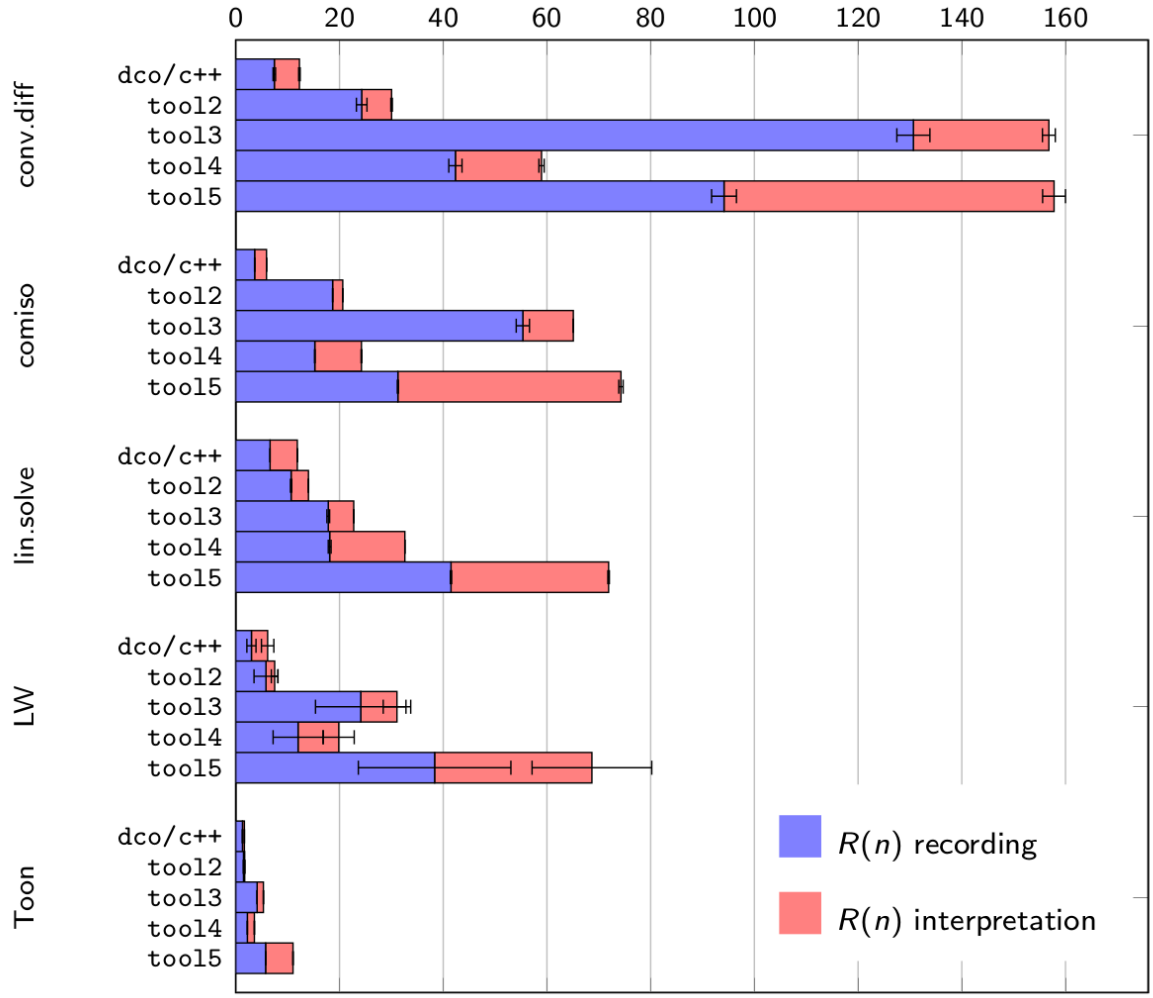
There were several requests for benchmarks of dco/c++ (/content/algorithmic-differentiation-software) against other AD tools. Just as there are lies, damned lies and statistics, so there are lies, damned lies and benchmarks. AD codes can perform quite differently on different codes: after all, they operate on the level of the computational graph, which can vary hugely from one code to another. If you are interested in benchmarks, I would urge you to get dco/c++ (/content/algorithmic-differentiation-software) and benchmark it yourself on your own code. This is what a number of large banks did before buying global licences for NAG's AD software (/content/algorithmic-differentiation-software).

The benchmarks below are anonymised: we're not going to identify which AD tools are represented by which bars. All the tools tested are well known, although one is perhaps a bit specific to finance. NAG has a close relationship with the AD community and has huge respect for their work. We invest hea in our own AD software (dco/c++ and the NAG AD Library represent over 1!

person-years of research and development), but we also collaborate with many groups. For this reason, we feel that some level of discretion is appropriate when reporting benchmark results. The AD community is quite small.

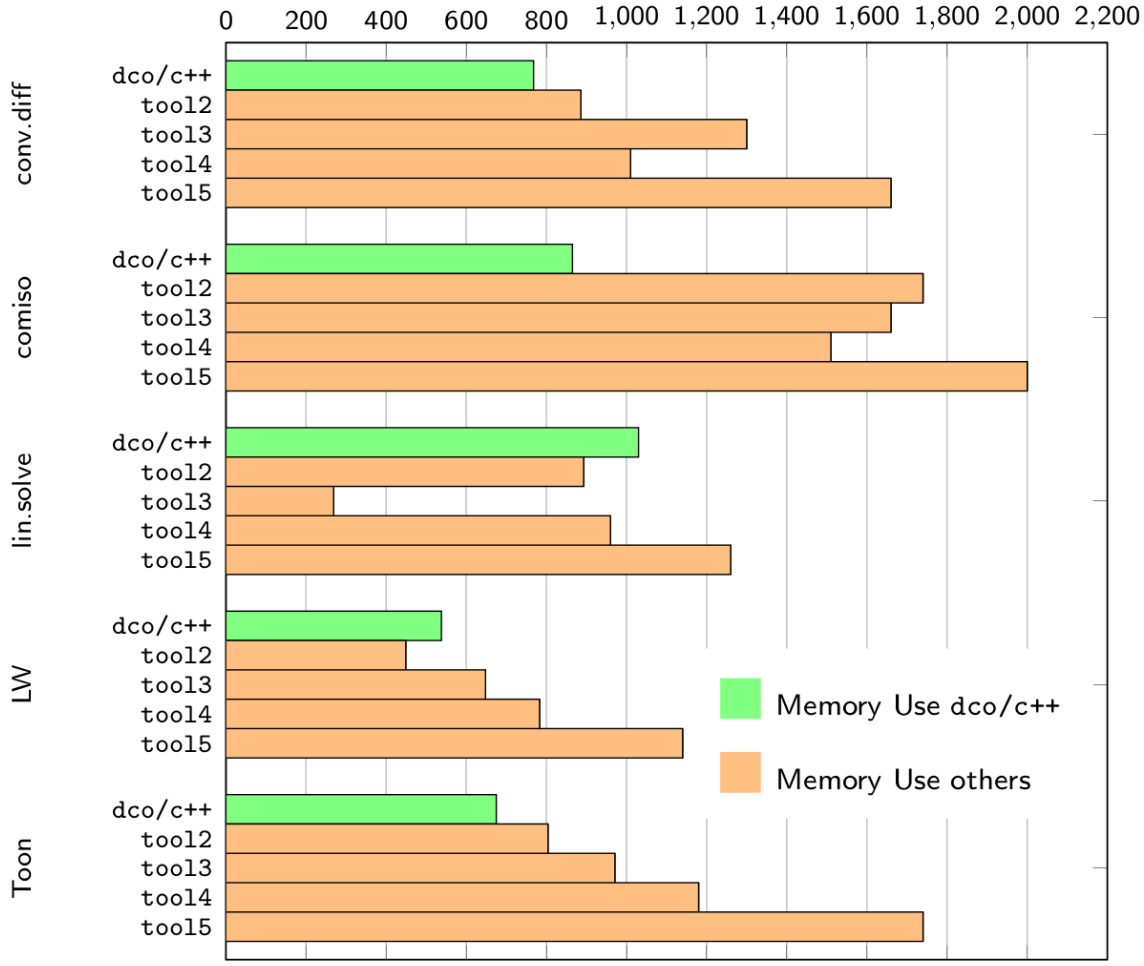
To avoid confusion - none of the tools benchmarked below are machine learning packages such as Tensorflow, PyTorch, MXNet or the likes. The tools we benchmark against are general-purpose AD tools, designed not specifically for tensor arithmetic but for any simulation. These are tools that you can apply to create an adjoint of any C++ code.

The results below are a few years old now but are all I have to hand. I will dig out more recent benchmarks and will update this post.



The image above shows the runtime factors of first-order adjoints for various AD tools on a variety of in-house codes. The tools have been applied in the most naive way possible, no further work was done to optimize the adjoint implementation. "$R(n)$ *recording*" is the runtime of the forward pass of the AD tool divided by the runtime of the original code, and "$R(n)$ *interpretation*" is the runtime of the backward pass of the AD tool divided by the runtime of the original code. Hence the x axis represents a slowdown factor: how many tin

slower the adjoint code was than the original code. This is the standard metric by which adjoint AD codes are judged. The error bars in the plots indicate standard deviations of the two runtimes.



The image above shows the corresponding memory use (in MB) for the various AD tools. The memory use of dco/c++ (/content/algorithmic-differentiation-software) is very competitive and it does have the fastest runtime. Later webinars will discuss methods for reducing memory use and optimizing the adjoint implementation, so these memory results should be seen as a baseline only. The figure people look at more often is the runtime, which shows how efficient the code is that the AD tool is producing.

LEARN MORE ABOUT NAG AD SOLUTIONS →

Author

Jacques Du Toit (/people/jacques-du-toit)

Comments

MH

Michel Herrera

04/08/2020

Is it the dco/c++ using template meta programming under the hood? Its seems to me like the computation graph is generated during compilation time and evaluated during runtime when you need to calculate the adjoints.

- Reply (/comment/reply/node/7209/field_comment/76)

JdT

Jacques du Toit

05/08/2020

Yes dco/c++ is using template meta programming. The computational graph is not really generated during compilation, because during compilation the AD tools doesn't know what the graph looks like. Things like control flow, iteration counts, etc are only known at runtime. So the DAG (directed acyclic computational graph) can only really be built during runtime. At compile time, the code to build the graph is instantiated and compiled.

At runtime during the "forward pass" the DAG is built in memory. During the "reverse pass" the DAG is reversed and the adjoint projection is computed.

- Reply (/comment/reply/node/7209/field_comment/78)

Mark L. Stone

MLS

04/08/2020

Thanks for the insightful blog. I'm looking forward to future installments. There's one item, however, that's easier said than done.

NAG produces tangent and adjoint versions of its black box functions - and that's great.

As for other suppliers of black box functions:
1) The chances the vendor even understands what is being requested (tangent, adjoint, algorithmic differentiation, automatic differentiation) are remote
2) For those few vendors who do understand the request, let's hope they are not drinking milk when they learn of it. They are not likely to think development of tangent or adjoint black boxes is an economically attractive proposition.

Perhaps this would be plauisble for certain academic packages. But those don't tend to be black boxes.

- Reply (/comment/reply/node/7209/field_comment/77)

JdT

Jacques du Toit

05/08/2020

Indeed, I'm not aware of other numerical library vendors who are making AD versions of their functions. From a library vendor's point of view, AD introduces considerable complexity.

- Reply (/comment/reply/node/7209/field_comment/82)

YE

Yassine E.

05/08/2020

Thanks Jacques for this blog and the detailled explanations. I would like to ask if you can elaborate a more why the cost of tangent model is roughly twice t cost of F.

- Reply (/comment/reply/node/7209/field_comment/83)

JdT

Jacques du Toit

05/08/2020

So the output of the tangent model is the original function value F(x) as well as the Jacobian-vector product. Hence the cost is at least as much as computing F(x). The question then becomes how expensive is it to compute the Jacobian-vector product. Here it depends on the function. For example if F(x) = exp(x), then the cost of the Jacobian is more or less free. If F(x) = x1 * x2, then the Jacobian-vector product is a multiplication and a fused multiply-add, which is at most slightly more expensive than the original function. If F(x) = x1 / x2, then the Jacobian is rather more expensive (more multiplications and additions).

So it depends what the function is doing. Simulations typically contain fewer divisions than they do additions or multiplications, and then there are cache effects to take into account as well (the derivative computation typically doesn't need to fetch more data from memory). So in practice we often observe a tangent factor of roughly 2. With dco/map on GPUs I've seen tangent factors of roughly 1, just because it's very hard to saturate a GPUs compute power. In a future session we will see how to push the tangent factor lower by using what we call "vector tangent mode", but that's only a speed improvement if you want the whole gradient or Jacobian, rather than just a single projection. Many applications do actually want the Jacobian.

- Reply (/comment/reply/node/7209/field_comment/84)

Leave a Comment

Your name

Your Comment

SUBMIT →

Sign up for the NAG newsletter

SUBMIT →

ABOUT NAG (/CONTENT/ABOUT-NAG)

Blog (/content/nag-blog)
NAGnews (/content/nagnews-0)
Case Studies (/content/case-studies)
Contact us (/content/worldwide-contact-information)

SUPPORT (/CONTENT/TECHNICAL-SUPPORT-SERVICE-OVERVIEW)

Contact support (/content/technical-support-service-overview#contact)
Documentation (/content/software-documentation)
Installer's & Users' Notes (/content/installers-and-users-notes-nag-products)
Downloads (/content/software-downloads)
Technical Reports (/content/technical-report-repository)

Privacy Notice (/content/privacy-notice) Trademarks (/content/trademarks)

WORLDWIDE LOCATIONS (/CONTENT/WORLDWIDE-CONTACT-INFORMATION-0)


(https://twitter.com/nagtalk)


(https://www.facebook.com/NAGTalk)


(https://www.linkedin.com/company/nag/)


(https://www.youtube.com/user/NumericalAlgorithms)

(https://github.com/numericalalgorithmsgroup)