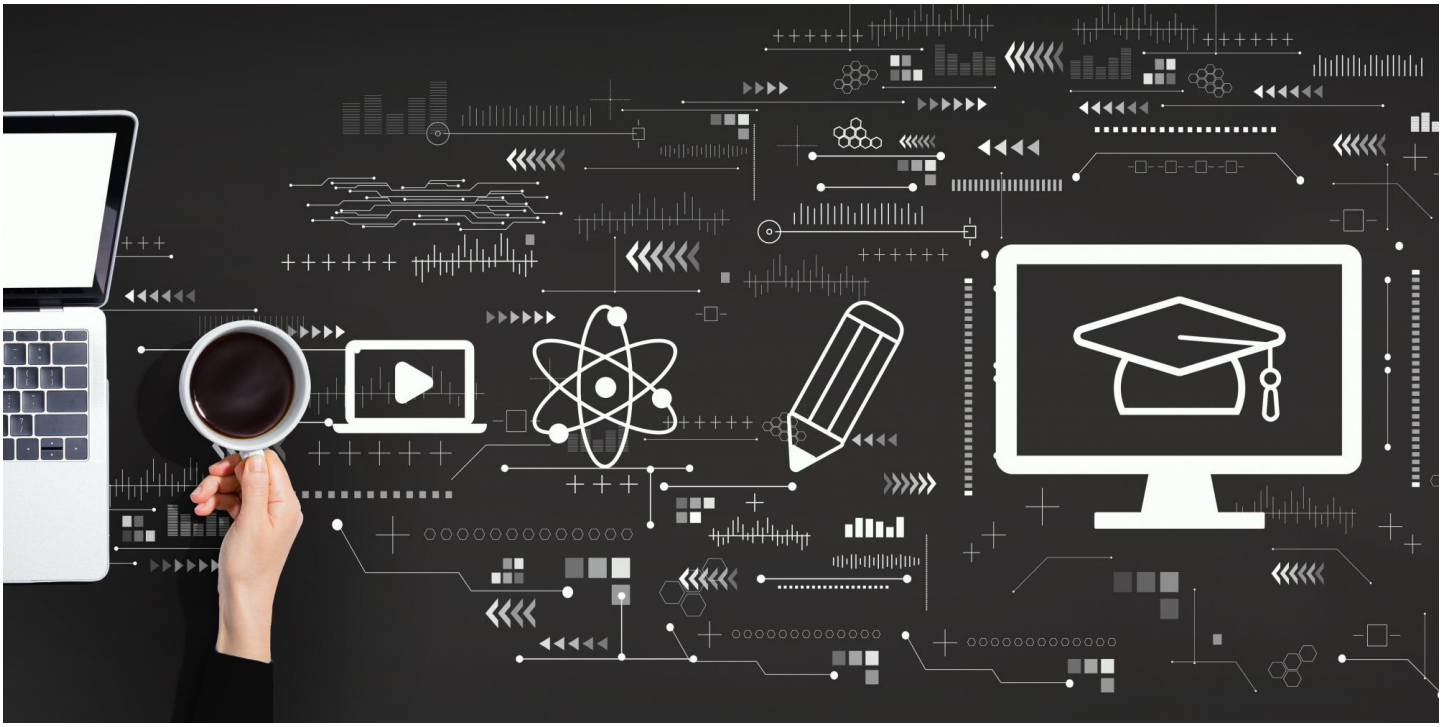

Algorithmic Differentiation Masterclass Series 2

Published 11/08/2020 By Jacques Du Toit

Computing Jacobians and AD Masterclass Follow-up Questions

This blog follows up discussions which arose from the Algorithmic Differentiation Masterclass 2 delivered on 6 August 2020 and is meant for attendees of the class.

So how would you explain that to a friend?

It's hard to define exactly what we mean when we say "What's the intuitior behind that?" My second year of mathematics at university systematically destroyed any intuition I ever thought I had about maths. My lecturer had a

passion for counterexamples and gleefully smashed all our "naive" beliefs about how the world worked. This led me to think that intuition was simply the lies we tell ourselves, to get by in life.

But reducing intuition to nothing more than "useful lies" is a little harsh. A more constructive approach might be to ask "how would you explain that to a friend?" At an AD conference a few years ago I asked a delegate (over a few pints) how he would explain adjoints. He blinked and then said:

> "Well I can tell you how I explained it to my partner. We were having dinner in a Chinese restaurant, and I picked up a baby squid and said: So imagine this squid is my code. If I wiggle a tentacle and the head moves, that's tangent mode AD. If I wiggle the head and the tentacles move, that's adjoint mode AD."

I think there is simple genius in his answer! It beautifully captures the notion that *tangent* has something to do with "perturbing" the inputs and studying the effect on the output, whereas *adjoint* has something to do with perturbing the output and studying the effect on all the inputs, at the same time. A slightly more mathematical way to look at this is as follows. Consider a calculation

$$x \longrightarrow \cdots \longrightarrow v \longrightarrow \cdots \longrightarrow y$$

where we have an input x, some intermediate value v and a final output value y. The dimensionality of these quantities doesn't matter. Suppose we are standing at v in our program. We have started running at x, we've done a whole lot of calculations and have reached v, and we still have a lot of calculations to go before we reach y.

Then the derivative of v with respect to x is tangent mode AD. Adjoint mode AD is the derivative of y with respect to v. In other words, tangent mode AD asks "What is the derivative of my current program state w.r.t my program inputs?" In contrast, adjoint mode AD asks "What is the derivative of my program outputs w.r.t my current program state?"

As we walk backwards, we are moving our program state back through time until our program state is equal to our inputs. We start off with our program state equal to our outputs, which implies that the "adjoint of y is equal to 1". By the time our program state has reached our inputs x, we've computed

dy/dx. Similarly, for tangent mode our program state starts at x, which implies that the "tangent of x is equal to 1", and by the time our program state has reached y, we have computed dy/dx.

Of course, this little story implies that the adjoint of x is equal to the tangent of y since both are equal to dy/dx. And this is a lie. But hopefully, it is a useful lie.

Fixing the lie

For it to stop being a lie we need to introduce the notion of *projection*: both tangents and adjoints compute *projections of the Jacobian*. A definition that I find useful is the following. Fix some auxiliary variable t. Then the tangent of x is dx/dt, while the adjoint of x is dt/dx. For an operation $y = h(g(f(x)))$ we therefore have that the tangent of y is

$$\begin{aligned}\frac{dy}{dt} &= \frac{dh}{dg}\frac{dg}{dt} \\ &= \frac{dh}{dg}\frac{dg}{df}\frac{df}{dx}\frac{dx}{dt} \\ &= J_y \frac{dx}{dt}\end{aligned}$$

Hence the tangent of y is equal to the Jacobian of y times dx/dt, the tangent of x. This is exactly the tangent model definition we had before, and we see that the "vector" in the "Jacobian vector product" is the tangent of x. The chain rule forces this definition of the tangent model so that it is closed under function composition: have a tangent model for f and another for F? Then the tangent model of $F \circ f$ is the composition of the two.

In a similar way, we can look at what happens in the adjoint model. Using our definition we see that the adjoint of x is

$$\begin{aligned}\frac{dt}{dx} &= \frac{df}{dx}\frac{dt}{df} \\ &= \frac{df}{dx}\frac{dg}{df}\frac{dh}{dg}\frac{dt}{dh} \\ &= J_y^T \frac{dt}{dy}\end{aligned}$$

since y is just equal to the output of the function h. Hence the adjoint of x is the transpose-Jacobian of y times the adjoint of y, which again is our "transpose-Jacobian vector product" definition. As before, we see that the

"vector" in this definition is the adjoint of y, which is necessary in the definition of the adjoint model so that it will be closed under function composition.

Hopefully, this discussion gives some intuition of where these arbitrary "vectors" in the definition of tangent and adjoint models are coming from. To make sense of them, it is necessary to consider what happens when functions are composed. Then we see that these vectors are needed in order to complete the chain rule.

Jargon Busting

Every field has its jargon (terminology) and AD is no different. It is a fair criticism that one shouldn't use terminology without first explaining what it means, so let's look at some of this AD jargon. Thankfully, the concepts in AD are very simple (we're not dealing with sigma algebras, for example). Recall from our discussion in the previous blog post that we were looking at a limit of the form

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

Here f_n represents our computer code, η is the input data to the code, and f is the actual mathematical object that our code is trying to approximate. Both Monte Carlo simulation and PDE solvers fit this form, as well as numerical quadrature, numerical root finders, iterative solvers, and a wide range of other numerical codes. AD is not concerned with f, the mathematical thing we are trying to approximate. It only knows about the code f_n for some n given and fixed.

Recall that we defined the tangent model of AD as

$$J_{f_n} \cdot \vec{x}$$

the Jacobian of f_n multiplied by some vector $\vec{x}$ in the input space, and we defined the adjoint model of AD as

$$J_{f_n}^T \cdot \vec{y}$$

the transpose Jacobian of f_n multiplied by some vector $\vec{y}$ in the output space. Keep in mind that **in addition to this**, both tangent and adjoint models **also** compute $f_n(\eta)$, the actual output of the underlying code. We can now start looking at some of this terminology:

- *forward mode AD*: this is a synonym for the tangent model of AD.

- *reverse mode AD*: this is a synonym for the adjoint model of AD.
- *primal code*: the numerical code f_n. When we talk about the *primal* we mean the underlying simulation code.
- *language intrinsic*: by this, we mean the most basic mathematical operations that your computer language permits. These are usually $+, -, *, /$ and the special functions $\sin, \cos, \exp, \ldots$
- *single assignment code (SAC)*: this is what we would obtain if we took the primal code f_n and re-wrote it so that every line of code consisted of a single language intrinsic operation and an assignment, for example, $v = x_1 * x_2$ or $w = \sin(v)$. Think of this as some kind of abstract intermediate representation of the code. SAC is useful when considering AD from a theoretical point of view, and sometimes when making handwritten adjoints, but is not generally useful to people who just want to apply AD to a code.
- *computational graph (DAG)*: imagine that you are the floating-point unit (FPU) in your computer, and also that you have hardware support for special functions that the compiler knows about. Now suppose someone starts executing the code $f_n(\eta)$. Imagine what it is you will be asked to do from the start of the program until the end. FPUs don't see any control flow and don't see any integer operations, so the only thing you'll see is a long string of mathematical operations. These are basically the SAC form of your program: each operation will be a basic operation (arithmetic or special function), and the result is stored to a variable (memory address). When seen this way, the *execution* of the code $f_n(\eta)$ produces a directed acyclic graph (DAG) where each assignment to a variable (for example $y = \sin(x_1 * x_2)$) represents a node: the assignment node (y in this case) will have dependencies on the variables which appear on the RHS (x_1 and x_2 in this case). The adjoint model of AD, in general, requires one to somehow *reverse this DAG*. This is a fundamental requirement of creating (discrete) adjoint codes.
- *tangent seed*: to evaluate the tangent model we need an arbitrary input vector $\vec{x}$. This vector is called the (tangent) seed. From our discussion above, we know that this vector is nothing other than dx/dt where we have the freedom to choose whatever t is. Very often we choose $\vec{x}$ to be one of the Cartesian basis vectors in the input space.
- *adjoint seed*: to evaluate the adjoint model we need an arbitrary input vector $\vec{y}$. This vector is called the (adjoint) seed. From our discussion above, we know that this vector is nothing other than dt/dy, where we have the freedom to chose whatever t is. Very often we choose $\vec{y}$ to be one of the Cartesian basis vectors in the output space.
- *source to source transformation tool*: when considering how an adjoir tangent model of f_n might be implemented, we have three choices: w the code by hand, use "an AD compiler", or use "an operator overload

tool". An AD compiler is called a source to source transformation tool. Its job is to take in the code for f_n and to output the code for either the tangent or adjoint model of f_n. The compiler must analyse the source code and must automatically do all the transformations necessary in order to produce a correct adjoint or tangent. This is a non-trivial task, and the richer the programming language in which f_n is written, the harder this task becomes. For example, source to source transformation tools for C++ are still (even after all these years) in their infancy.

- *operator overloading tool*: the alternative to a compiler is a runtime tool. A compiler has to gather all the information it needs via static analysis. For adjoint AD, this is challenging. A tool which instead defers all its work to runtime has different program information available. Operator overloading tools for AD are not compilers, in the sense that they do not output source code (dco/c++ can in fact output source code, so this boundary is beginning to blur). For adjoint mode they instead build the *computational graph* in memory, along with local partial derivatives on the edges between nodes. This allows the tool to compute the adjoint in a very robust way. We'll talk a little more about *how* this is done below.
- *tape*: the data structure that operator overloading tools use to represent the DAG of the underlying code.
- *seeding*: the process of setting the values of the tangent or adjoint seed on an operator overloading tool is called seeding. Each tool has its own API for seeding, but universally, seeding is nothing more than telling your tool what the values of $\vec{x}$ or $\vec{y}$ are.
- *harvesting*: if there is a seed, then typically there's a harvest. Harvesting is the process by which the Jacobian vector product, or the transpose-Jacobian vector product, is extracted from the operator overloading tool once the code has been run. Each operator overloading tool has its own API for doing this, but universally, *harvesting* is simply the process of retrieving the output of the tangent or adjoint models.
- *registering an input*: inputs to the adjoint model must be "registered" with the tape. This anchors them as root nodes in the DAG. If we don't register an input to the adjoint model, then derivative information won't be propagated into this variable. Note that only *inputs* to the adjoint model need to be registered: outputs and intermediate values are automatically added to the DAG.
- *discrete adjoint* and *continuous adjoint*: a discrete adjoint is what you get when you create an adjoint of your *primal source code* (i.e. build the DAG, reverse the DAG). A continuous adjoint is any way of getting an adjoint which does not involve building a DAG and reversing it. Examples of this are implicit function theorem, or deriving analytic adjoint systems for f_n (for example in PDEs) and solving those. We'll look at some examples of continuous adjoints later on in the series.

Continuous adjoints are also sometimes called *symbolic adjoints*. Prof. Mike Giles has a paper (https://people.maths.ox.ac.uk/gilesm/files/AD2008.pdf) on symbolic tangents and adjoints for certain matrix operations. We have a paper (https://www.researchgate.net/publication/323137160_Adjoint_algorithmic_differentiation_tool_support_for_typical_numerical_patterns_in_computational_finance) which among others looks at root finders and optimisation (these results are well-known in the AD community).

- *adjoint factor*: for a given implementation of the adjoint model, the adjoint factor is the ratio of the runtime of the adjoint code to the runtime of the primal code. In other words, it is how many times the adjoint code is slower than the primal code. This is the key metric by which adjoint implementations are evaluated.
- *tangent factor*: the ratio of the runtime of the tangent code to the runtime of the primal code, in other words, how many times slower the tangent code is.
- *active data*: any data that we are differentiating. For example, these could be model inputs that we want sensitivity information for. The term is also used for intermediate variables which have to be differentiated (chain rule) in order to compute the overall sensitivities we need.
- *passive data*: any data which we are not differentiating. This will include all non-floating point data (integers, pointers, etc) but can also include some floating-point data. For example, the output of a Uniform(0,1) random number generator is almost surely going to be passive. The initial bracketing interval of a bisection search algorithm will be passive. There may be sections of your code that you just don't need to differentiate through, for whatever reason.

The DAG

We described the DAG (computational graph) above at a very low level: we thought about what the FPU might see and said the DAG more or less represents the SAC form of the primal code. Looking at the DAG from the SAC form of the primal is useful for theoretical work, and is sometimes useful for writing tools, but descending this low into the code does limit us. We lose information about the structure of our code: we forget what it is we're actually trying to do, and which steps we need to go through in order to do it.

We've already said that exploiting structure is absolutely key in making efficient adjoint implementations. So a DAG that hides this information is not much use. Hence when we talk about "the DAG" in adjoint AD, it's more helpful to understand the DAG as a "multi-granular" thing. At the finest granularity we see nodes and edges representing the SAC form of the primal. But then are also able to "zoom out" and coarsen this DAG into another DAG which

represents higher-level operations, for example, a matrix multiplication, a path simulation, a linear solve, an optimisation, etc. And if we coarsen even more we start seeing separate steps like calibration, simulation, aggregation and post-processing. These tasks might indeed map onto different machines in our cluster/data centre.

So when we talk about "the DAG" we're actually talking about the structure of the primal code, at any level of detail that helps us do what we need to do.

Overture to the joys of the tangent model

We know the tangent model is suitable for codes with few inputs, while the adjoint model is more suitable for codes with few outputs. In the webinar we talked about when one might choose tangent model over the adjoint model, we spoke about the adjoint factor, and we saw that the adjoint factor, the number of inputs and the number of outputs together determine when one should use tangent model and when one should use adjoint model.

What we did mention in the webinar but couldn't sufficiently emphasise, is *how much easier* the tangent model is to implement than the adjoint model. It is a doddle!! It is basically no work at all. The tangent model is pretty much bulletproof, and your memory requirements are twice that of your primal code. And, as we will see in future webinars, the tangent model can actually be pretty fast on modern hardware (SIMD) since we can compute columns of the Jacobian in parallel, which amortizes certain calculations and results in a more efficient Jacobian calculation overall.

For this reason, it is appropriate to laud the tangent model and not merely view it as the ugly stepsister. Not only is it invaluable in validating an adjoint code, but it

- is fast to implement, hence cheap in terms of manpower
- is intuitive (it's similar in spirit to finite differences)
- is usable with only a modest understanding of AD (so it's something you can delegate to others!)
- has modest memory use
- is actually pretty fast on modern hardware (the tangent factor is pretty low)
- shows no difference in performance between operator overloading tools and source transformation tools
- is trivial to re-use parallelism in the primal code

- can be combined with "vector tangent mode" (more on this in the next webinar) to push the effective tangent factor (to compute the whole Jacobian) even lower
- tangents are necessary to compute Hessians efficiently

This really is a model worth celebrating! Ignore it at your peril.

Writing adjoints by hand

"No pain no gain" is a well-known maxim. No amount of "lecturing" will ever give you a deep understanding of the adjoint model. At some point, you have to take pencil and paper and actually do some of this, and we've reached that point.

Our purpose with the webinar series is not really to teach people how to hand-craft adjoints, or how to write AD tools. If this is your aim, then it is probably better to take one of the many texts on AD and work through it. AD tools need to be robust in the face of bad code, and for languages like C and C++, it's surprising just how bad a piece of code someone can write (pointer aliasing is but the start!).

Our goal with this webinar series is to help people become proficient *users* of AD. You don't actually need to know that much about AD in order to be proficient with a well-written operator overloading tool. A good AD tool really does make it easy to get up and running quickly with adjoints and tangents. To convince yourself of this, just take dco/c++ and try it on some simple examples.

Having said that, there is value in working through a few examples on hand-writing adjoints. We're going to look at a few "tutorial" style questions to try and really dig into the adjoint model. Solutions to the tutorials are given in at the bottom of this blog post, but do try them on your own first. Use the definition of the adjoint of a variable w as being dt/dw and work through the data flow reversal. Remember that we first need to run the code forward, then we need to backpropagate.

Tutorial 1

Let's take another look at the simple example Viktor used in the webinar:

```
double f(double x1, double x2)
{
    // u depends on x1 and x2, hence the adjoints of
    // x1 and x2 are ...
    double u = x1*x2;
    // y depends on u, hence the adjoint of u is ...
    double y = sin(u);
    return y;
}
```

Code this up and run it with an adjoint seed of one. Check your answer by differentiating $y = \sin(x_1 x_2)$ and plugging some numbers in. Remember that in the function signature you need to introduce variables for the adjoints of x_1, x_2 and y. We can start formulating some *rules for writing adjoints*. The first rule would be

1. Duplicate the active data segment: each active variable gets a corresponding adjoint variable

Tutorial 2

The following example is one that I used to try to understand adjoints when I first met them. It's inspired by Euler iteration, such as one might find when solving an ODE or an SDE. Consider the following:

```
// Some smooth function, doesn't matter what it is
double h(double b, double x);
// The derivative of h w.r.t. b
double dh_db(double b, double x);
// The derivative of h w.r.t. x
double dh_dx(double b, double x);

void f(double b, double x, double &y)
{
   double x1 = x + h(b,x);
   y = x1 + h(b,x1);
}
```

Try and work out what the adjoint for this code snippet is. It doesn't matter what h is, you can use the function from Tutorial 1 if you want. Work out the mathematical derivatives df/dx and df/db by crunching through the chain rule (on paper) and compare that to your adjoint model. You should find that something doesn't quite work with the adjoint of b.

To fix this we need to *increment* the adjoint of b each time. So we can add our second rule:

1. Duplicate the active data segment: each active variable gets a corresponding adjoint variable
2. Increment adjoint variables during backpropagation. Initialise all adjoint variables to zero.

Of course when we say "Initialise all adjoint variables to zero" we exclude the adjoint seeds that are the input to the adjoint model. These should be initialised to whatever makes sense for your application, most likely the Cartesian basis vectors of your output space.

Tutorial 3

Now let's look at another example. So much of the complexity in making handwritten adjoints comes from variables that are overwritten. Let's first take an example where nothing is overwritten:

```
void g(double z, double &y)
{
    u = exp(z);
    y = sin(u);
}
```

Once you've made a handwritten adjoint for that, try the following code:

```
void f(double &z)
{
    z = exp(z);
    z = sin(z);
}
```

Can you manage to get the correct derivatives? Can you now *also* get the correct function return value?

Correctly dealing with the function `f` above is somewhat tricky. We need to extend our rules as follows:

1. Duplicate the active data segment: each active variable gets a corresponding adjoint variable
2. Increment adjoint variables during backpropagation. Initialise all adjoint variables to zero.
3. Store values during forward run that are overwritten.
4. Read-and-zero RHS adjoints before propagating them.

Tutorial 4

We can now put everything together to look at the function

```
void f(double &x, double t)
{
    double y;
    y = x*x;
    x = sin(x*y*t);
    y = exp(x*t);
    x = sin(x*y*t);
}
```

Conclusion

The "rules" we wrote down above are incomplete since they don't deal with loops and branching. It's relatively straightforward to extend them to cover these cases. The key points are that loops must be *reversed* (in general) and the same code branches must be taken as were taken in the forward run. This typically means that data must be stored to determine which branch to take. For a full list of rules and a detailed discussion of making handwritten adjoints, see for example Uwe Naumann's book *The Art of Differentiating Computer Programs*.

Hopefully by now we've established a few things:

- Writing adjoints by hand isn't that much fun
- Maintaining handwritten adjoint code is even less fun (especially if someone else wrote it)
- Debugging handwritten adjoint code is absolutely no fun at all! Imagine doing this on a larger code: how will you find where you've made a mistake?
- We always increment adjoints, and we always initialise them to zero
- There is no difference, really, between tangents, adjoints and mathematical differentiation. They are just different ways of performing one and the same thing, namely the chain rule.
- If you have some other way of knowing what the adjoint of a particular part of your code is (maybe you have a library of handwritten adjoint functions, or you have a symbolic differentiation package, or you've formulated an adjoint PDE, or you have an oracle, or whatever), then your adjoint AD tool should give you some way of using that information, since really all we're doing is *applying the chain rule*. dco/c++ allows you to do this.

Source transformation tools for AD

We spoke briefly about what a source transformation tool for AD is. It's a compiler whose job it is to ingest your source code, and output source code implementing the tangent or adjoint model. For the tangent model, there is very little benefit in using source transformation vs an overloading tool, so really we only talk about source transformation for the adjoint model.

The main advantage that source transformation tools have is that they don't need to build the entire DAG in memory. They can analyse the source code and can typically determine the DAG from that, or at least, can determine the various possibilities for what the DAG might be (if we factor in runtime information such as branching and loop counts). Therefore source transformation tools use much less memory than operator overloading tools, and can also perform a certain amount of common subexpression analysis and optimisation.

There are two main problems with source transformation tools. Firstly, they are very limited in the kind of input language they can ingest. Even the most advanced source transformation tool for C, for example, can only handle certain (restrictive) uses of dynamic memory management, certain if-expressions and certain uses of pointers. There are also requirements that some functions be re-entrant. There is no source transformation tool that we are aware of that can handle the whole of C, let alone C++.

Secondly, using a source transformation tool can expose you to operational risk. Any change to the primal code must be followed by someone re-running the source transformation tool and integrating the output in your adjoint code base. This last part isn't always completely automatic. If we forget to re-run the source transformation tool, our adjoint code will be incorrect. Operator overloading tools basically eliminate this kind of error.

What are C++ templates?

To understand how operator overloading tools are designed to be used, it's worth talking a little about C++ templates. Not everyone on the webinar speaks C++, and all our code examples are C++, so it's appropriate that we say a few words here.

A C++ template is code which instructs the compiler how to output more code. Hence it is sometimes called *metaprogramming* since we are writing a program, which will end up producing a program, which will then get compiled. Now this can sound a little scary, and in the hands of a skilled C++ developer it can indeed become something very scary, but the basic idea is quite simple. (It can become scary since the template syntax is Turing complete, but that's a rabbit hole we're not going down today.)

C++ is a strongly typed language. This means in particular that

```
void f(int x);
void f(double x);
void f(float x)
```

are all different objects. All three can happily coexist as *function overloads*, but it's important to keep in mind that they are different things, all the way down to the symbol names baked into the object code.

Suppose we now want to write a sort algorithm. The algorithm basically only needs us to be able to compare two numbers, so the code body for `my_sort(int n, double *x)` and `my_sort(int n, float *x)` and `my_sort(int n, int *x)` will all be basically identical. And we're not even going to contemplate "aberrations" like `my_sort(int n, void *x, char datatype)` or similar.

So what we want is some way to write our sort code once, and then allow the compiler to use that sort code with whatever type we want, as long as it supports a comparison operation. Templates are a way to do just this:

```
template<class T>
void my_sort(int n, T* x)
{
   /* body of sort algoritm */
}
```

The code snippet above defines a *function template*. The *template parameter* `T` is a placeholder for a type. On its own, this template is useless. In fact, compiling it won't even result in anything being put in your object file. This is more or less obvious - the compiler doesn't know what `T` is. It's a place-holder.

It is only once we *call* `my_sort` with a real type (`float, double, MyType,...`) that the compiler can create object code. At this point, the compiler *instantiates* the template function with the *concrete type* and checks whether the type has a comparison operator. If I didn't give `MyType` a comparison operator (oops, I forgot!) then I get a compiler error.

So the key thing about templates is they separate the *algorithm* from the *data type* on which it operates. Operator overloading AD tools assume that your functions are templated. If your functions are not templated, then you will have to have separate function bodies for your primal, your tangent and your adjoint codes, exposing you to operational risk once again.

Operator overloading tools for AD

Operator overloading tools for AD work by replacing your floating-point data type (`float, double`) with a special class which mimics a floating-point type. Instead of computing with `float` or `double` you compute with an AD type which is designed to support all floating-point operations and can be passed to special functions (sin, exp, etc).

In tangent mode the AD type is pretty simple: it contains a value component and a derivative component, and as your code executes, the primal and the tangent projection are both computed at the same time.

In adjoint mode, however, more needs to happen. Now the AD type must compute the primal value (output of the primal code) and it must construct the DAG. Since the AD type has replaced the floating-point data types, it sees all floating-point operations and can form the DAG corresponding to the SAC form of your program. This DAG is built at runtime. As your program executes, the AD type records data to an in-memory data structure called *the tape*. This tape represents the DAG of your program. Once the program has finished, the tape is "played back" or "interpreted" in order to back-propagate the adjoint seed to the adjoints of the inputs.

As you can see, we've not mentioned templates yet. This should be expected since operator overloading AD tools exist for languages such as Fortran and Matlab, which don't have templates. In C++ though, almost all AD tools try to use template metaprogramming techniques to decrease the amount of work they need to do at runtime. Templates are therefore an optimisation only - they are not needed to actually compute the tangent or adjoint.

What is typically done is that the AD adjoint type itself is templated and it tries to use *expression templates* to reduce the amount of data recorded to the tape. Details of exactly *how* this is done are beyond the scope of this webinar series, but heuristically, expression templates are a metaprogramming technique to produce *new types* on-the-fly in order to express some useful information (maths operations in our case) at compile time. For example, in the code snippet

```
dco::ga1s<double>::type x1(5), x2(1);
dco::ga1s<double>::type y = sin(x1*x2) + x1*x1 - exp(x2)/x1;
```

the RHS of the assignment to `y` is in fact constructing a new type on-the-fl invisible to the user, which represents the mathematics being performed. TI AD tool can now use metaprogramming to inspect this type, unpack the

mathematics it represents, get hold of the inputs to the maths, and do calculations with those inputs (for example, computing the local partial derivatives dy/dx_1 and dy/dx_2. The key point is that all this happens *at compile time* and not at runtime. Hence the AD tool has reduced the amount of work it needs to do at runtime and has also reduced the amount of data it must store to the tape. Without expression templates, the AD tool would need to consider separately each of the primitive calculations `x1*x2`, `sin(v)`, `x1*x1`, `v+u`, `exp(x2)`, `w/x1`, `v-z`. Each of these could potentially push data to the tape.

Tapes and dynamic memory management

It's unlikely we will know upfront how much memory our tape will need. Tapes typically grow extremely fast. Conceptually, every assignment your code makes to an active variable will push some data to the tape. Even a few seconds of runtime is enough to create a tape hundreds of GB in size.

dco/c++ has a few different types of tape. There is a "chunk tape" which grows in small(ish) chunks as more memory is needed. This allocation and bookkeeping have a small overhead. Then there is a blob tape which allocates a large amount of memory up-front (this is user-controllable), by default half the available memory on your machine. This tape does not grow. Since Linux only commits physical page frames once they are used, such a large allocation isn't a problem. Big blob tape sizes are very efficient even for small problems that use only a fraction of the available tape size.

Windows, unfortunately, has a different approach to memory management. When you allocate memory in Windows, all the page frames are committed at once, which is a fairly expensive operation. On Windows, we recommend that you either adjust the size of the blob tape to fit your problem (the tape can tell you how much memory it's currently using) or just to use the chunk tape. The latter is a slightly slower option but is safer.

Other than these comments, it's typically not expensive to create or destroy tapes. As you will see in upcoming sessions, there is also a file tape which writes to disk rather than main memory.

Writing a driver

In the webinar, we saw how to write a driver for dco/c++. Now that we have the terminology defined, we can summarise this process in words. For the tangent mode:

- Create dco/c++ tangent types for all your active inputs (suppose there are n of them) and active outputs (suppose there are m of them)
- Allocate your $m \times n$ Jacobian J
- Loop from $i = 1, \ldots, n$
    - Seed all the inputs with the i-th Cartesian basis vector in $\mathbb{R}^n$
    - Run your code
    - Harvest the derivative information from the m output variables and store to the i-th column of J
- Continue the loop

For the adjoint model things are similar:

- Create dco/c++ adjoint types for all your n active inputs and your m active outputs
- Allocate your $m \times n$ Jacobian J
- Run your code, which will record the tape
- Loop from $j = 1, \ldots, m$
    - Seed all the outputs with the j-th Cartesian basis vector in $\mathbb{R}^m$
    - Interpret the tape
    - Harvest the derivative information from the n input variables and store to the j-th row of J
    - Set the derivative information in the input variables to zero
- Continue the loop

Note that we only have to run the code once in order to build the tape. Tape interpretation is much faster than building the tape in the first place, and with some support from your AD tool, the tape interpretations above can be done in parallel (dco/c++ allows this).

Although the APIs for all the operator overloading AD tools are different, these blueprints for the tangent and adjoint drivers are universal.

Dimension reduction and making sense of the output

I'm delighted that a few people have raised this question! Suppose we have a simulation with many inputs and/or outputs. AD can give us the Jacobian. But how useful is that Jacobian actually?

For example, consider an aerodynamic simulation of a wing. We can compute the sensitivity of the lift/drag to every mesh point on the surface of the wing, but that could be hundreds of thousands or millions of points. We can produce an image visualising this data, but that's probably all we can do with it. We can't perturb any "individual mesh point" on the surface of our engineered wing sample: it's a physical sheet of metal, there are limits to how we can work that metal. Indeed, it may be more instructive to think about the wing's geometry: chord length, span, camber, thickness, dihedral angles, etc. Asking how the lift/drag changes with respect to these parameters might give more insight into how the wing shape should be changed, rather than looking at individual points on the surface.

When either the input or output dimensions are high, consider whether there are not perhaps lower-dimensional quantities which would serve better as "explanatory variables" (things which help to explain the sensitivity of the simulation). A finance example might be a yield curve: we could look at each instrument that went into the stripping, or we can look at the curve's "geometry". If a simulation has lots of outputs, then it may make sense to aggregate those somehow and look at the sensitivity of the aggregate, or perhaps even the aggregate of the sensitivities (if the aggregation step is non-linear).

In short: when studying sensitivities, keep in mind that you have the freedom to change the simulation (pre-process, post-process) in order to get derivative information that is more intuitive. Not only can this simplify the task of *understanding* the data, but it can also make it easier to get the sensitivity information in the first place, for example by reducing the input dimensionality we might make the tangent model much more attractive.

Solutions to Tutorials

Let's take a look at how to make the handwritten adjoints for the few examples I gave above.

Tutorial 1

Our function is

```
double f(double x1, double x2)
{
    double u = x1*x2;
    double y = sin(u);
    return y;
}
```

We first need to add derivatives for all the active inputs and outputs. Then we need to run our code forward use the definition of adjoint of a variable w as being dt/dw to write the backpropagation code:

```
double af(double x1, double &ax1, double x2, double &ax2)
{
    // Forward run
    double u = x1*x2;
    double y = sin(u);
    // End of forward run
    // Start of backpropagation

    // dt/du = dy/du * dt/dy
    double au = cos(u) * ay;
    // dt/dx1 = du/dx1 * dt/du
    ax1 = x2 * au;
    // dt/dx2 = du/dx2 * dt/du
    ax2 = x1 * au;

    // Return the primal value
    return y;
}
```

Tutorial 2

Recall that our code is

```
void f(double b, double x, double &y)
{
   double x1 = x + h(b,x);
   y = x1 + h(b,x1);
}
```

To see what's happening here we first need to crunch through the chain rule for df/db:

$$\begin{aligned}\frac{df}{db} &= \frac{dx_1}{db} + h_b(b,x_1) + h_x(b,x_1)\frac{dx_1}{db} \\ &= h_b(b,x_1) + \frac{dx_1}{db}\Big(1 + h_x(b,x_1)\Big) \\ &= h_b(b,x_1) + h_b(b,x)\Big(1 + h_x(b,x_1)\Big)\end{aligned}$$

Now if we use our definition of the adjoint of a variable w as being dt/dw and apply it to this code, we will obtain

```
void f(double b, double &ab, double x, double &ax, double &y, double
 ay)
{
   double x1 = x + h(b,x);
   y = x1 + h(b,x1);

   // dt/dx1 = dy/dx1 * dt/dy
   double ax1 = (1 + dh_dx(b,x1)) * ay;
   // dt/db = dy/db * dt/dy
   ab = dh_db(b,x1) * ay;

   // dt/dx = dx1/dx * dt/dx1
   ax = (1 + dh_dx(b,x)) * ax1;
   // dt/db = dx1/db * dt/dx1
   ab = dh_db(b,x) * ax1;
}
```

Looking at that final assignment, we can see that something isn't right. It's clobbering the previous assignment to `ab`. Setting `ay=1` and comparing our code to the chain rule calculation above, we see that the first assignment to `ab` is correct, but the second assignment should not be an assignment, it should be an *increment* so that we *add* the portion `dh_db(b,x)*ax1` onto the value already in `ab`.

So the rule when making adjoints is: always initialise all adjoint variables to zero, and always *increment* adjoints during backpropagation.

Tutorial 3

The first example in Tutorial 3 should by now pose no difficulty:

```
void g(double z, double &az, double &y, double ay)
{
    u = exp(z);
    y = sin(u);

    // dt/du = dy/du * dt/dy
    double au = cos(u) * ay;
    // dt/dz = du/dz * dt/du
    az += exp(z) * au;
}
```

Now let's consider the second form of the function where everything is overwritten and see whether we can untangle this mess. If you do things naively you end up putting the wrong values into the derivative expressions

```
void f(double &z, double &az)
{
    // We are about to overwrite: store program state
    push(z);
    z = exp(z);
    // We are about to overwrite: store program state
    push(z);
    z = sin(z);
    // Save output state
    double ret = z;

    // Start of backpropagation
    pop(z);
    az = cos(z) * az;
    pop(z);
    az = exp(z) * az;
    // Restore output state
    z = ret;
}
```

When variables which are *inputs to non-linear expressions* are overwritten, they must first be saved somewhere. Since the expression is non-linear, the derivative will need the values of the inputs. We therefore have to save those values so that we have them available during backpropagation. Of course, we could recompute them as well. The point is that during the backpropagation, we somehow need to reconstruct the inputs to non-linear expressions so that we can compute their derivatives.

Similarly we need to save the output state before we start backpropagation. This is because during backpropagation we will be changing the program state (as we pop things off the stack), and in the process we will lose our output value. This is a very easy mistake to make.

As you can see, this kind of thing really isn't much fun. Now imagine debugging something like this that *someone else* has written (and of course they documented their code, didn't they!).

Regarding our rule to "Read-and-zero RHS adjoints before propagating them", if we implement this in our adjoint code then we get:

```
void f(double &z, double &az)
{
    push(z);
    z = exp(z);
    push(z);
    z = sin(z);
    double ret = z;

    // Start of backpropagation.
    pop(z);
    {
        double adj = az; az = 0;
        az += cos(z) * adj;
    }
    pop(z);
    {
        double adj = az; az = 0;
        az += exp(z) * adj;
    }
    z = ret;
}
```

This illustrates the general form of dealing with adjoint variables on the RHS of expressions. As soon as the value is read (and before any other adjoint variables are incremented) the adjoint variable must be zeroed out.

Tutorial 4

The last example puts all these ideas together. Recall that our code is

```
void f(double &x, double t)
{
    double y;
    y = x*x;
    x = sin(x*y*t);
    y = exp(x*t);
    x = sin(x*y*t);
}
```

The adjoint looks like this:

```
void f(double &x, double &ax, double t, double &at)
{
    double y, ay(0);
    y = x*x;
    push(x);
    x = sin(x*y*t);
    push(y);
    y = exp(x*t);
    push(x);
    x = sin(x*y*t);
    double ret = x;

    pop(x);
    {
        double adj = ax; ax = 0;
        ay += cos(x*y*t)*x*t * adj;
        ax += cos(x*y*t)*y*t * adj;
        at += cos(x*y*t)*x*a * adj;
    }
    pop(y);
    {
        double adj = ay; ay = 0;
        ax += exp(x*t)*t * adj;
        at += exp(x*t)*x * adj;
    }
    pop(x);
    {
        double adj = ax; ax = 0;
        ay += cos(x*y*t)*x*t * adj;
        ax += cos(x*y*t)*y*t * adj;
        at += cos(x*y*t)*x*y * adj;
    }
    {
        double adj = ay; ay = 0;
        ax += 2*x * adj;
    }
    x = ret;
}
```

Author

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

Comments

A

Anonymous

14/08/2020

"Note that we only have to run the code once in order to build the tape. Tape interpretation is much faster than building the tape in the first place,"
Can we run adjoint code with AD turned off? How much is it slower than the original code? I hope (and suspect) it should run at similar speed? Sometimes we are not interested in derivatives, just the function values.

- Reply (/comment/reply/node/7215/field_comment/85)

JdT

Jacques du Toit

14/08/2020

This depends very much on your AD tool. For most tools, the answer is No.

For dco/c++ you can create an adjoint implementation of your code and then turn "tape activity" off. This means that no data is pushed to the tape, in other words the DAG of your program is not built and local partial derivatives are not stored. This still has some overhead: it is small, I don't have numbers to hand on how small it is. We recently introduced a Binary Compatible Passive Type. This is a type which is binary compatible

the adjoint type, but has no derivative information. This type can be passed to a code compiled for adjoint types and will result in even less runtime overhead. The code is still a little slower than computing with purely doubles or floats, but is faster than just turning off tape activity.

However, as we discussed this week in the webinar on Validation, it is advisable to have primal, tangent and adjoint versions of the code since this is the best way to validate for a correct implementation. So our recommendation on best practice is that people have all three versions in their code base. The webinar discussed how to do this while minimising operational risk, keeping single-object compile times low, and maximising build system parallelism. When adopting this approach there isn't much of a need for "passivating" an entire adjoint code (one may still wish to passivate parts of it).

- Reply (/comment/reply/node/7215/field_comment/88)

RB

Roland Bole

14/08/2020

This is great. Thank you very much for the effort. Very much appreciated.

- Reply (/comment/reply/node/7215/field_comment/87)

MLS

Mark L. Stone

14/08/2020

In the interest of jargon busting, it should be pointed out that there is alternative AD terminology in widespread, and I believe far more common, use:

forward mode <-----> tangent mode (tangent-linear)
reverse mode <-----> adjoint mode

The Wikipedia "Automatic differentiation" article https://en.wikipedia.org/wiki/Automatic_differentiation primarily uses the terminology "forward mode" and "reverse mode", although it does mention adjoints in connection with reverse mode. The word "tangent" appears exactly once in the Wikipedia article, and that is as the name of a Google package provided as an external link to the article.

I picked up AD on virtual street corners. In those working class neighborhoods, if someone talked about tangent mode or tangent-linear, they'd be met with a dazed look.

- Reply (/comment/reply/node/7215/field_comment/89)

JdT

Jacques du Toit

17/08/2020

Thanks a lot Mark, a very good point indeed. I've updated the Jargon section. You're right, different groups sometimes have different words for the same thing.

- Reply (/comment/reply/node/7215/field_comment/90)

MLS

Mark L. Stone

18/08/2020

Adding the bullets for forward mode and revers mode is a great improvement to the article. Thanks.

- Reply (/comment/reply/node/7215/field_comment/92)

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)