

A Review of Automatic Differentiation and its Efficient Implementation

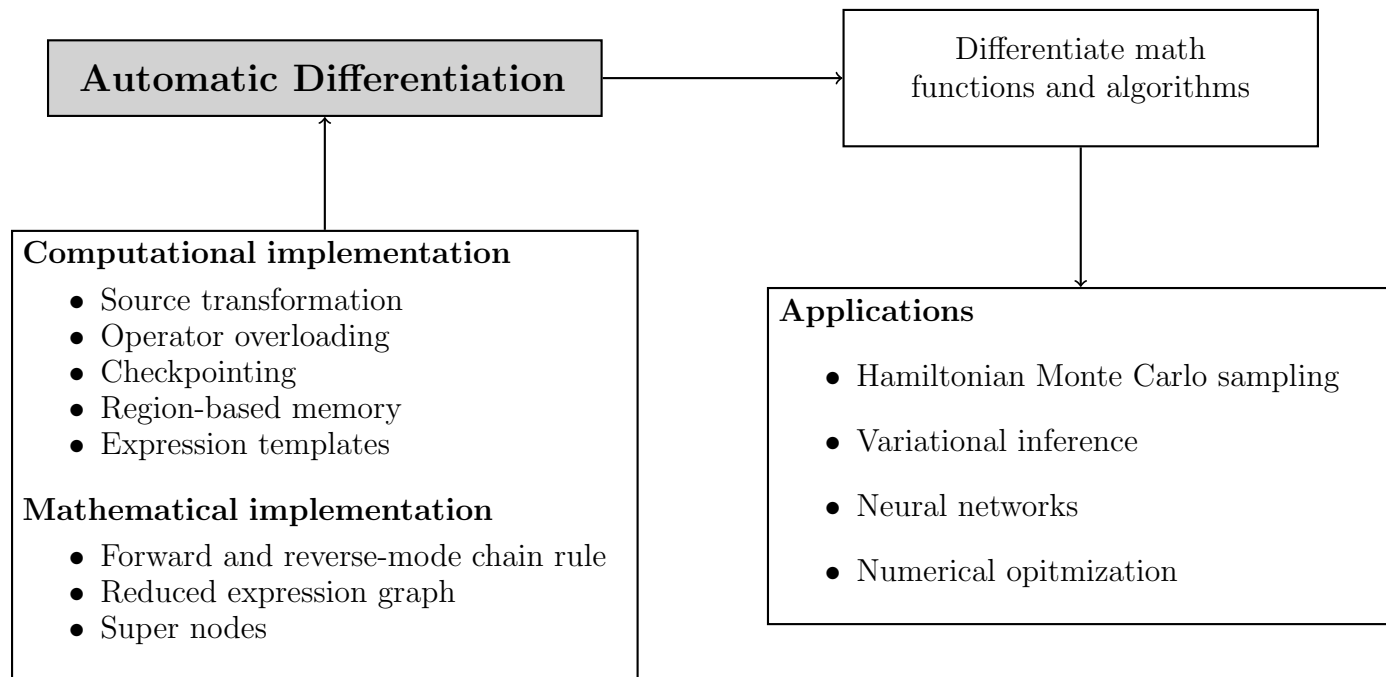
Charles C. Margossian*

Abstract

Derivatives play a critical role in computational statistics, examples being Bayesian inference using Hamiltonian Monte Carlo sampling and the training of neural networks. Automatic differentiation is a powerful tool to automate the calculation of derivatives and is preferable to more traditional methods, especially when differentiating complex algorithms and mathematical functions. The implementation of automatic differentiation however requires some care to insure efficiency. Modern differentiation packages deploy a broad range of computational techniques to improve applicability, run time, and memory management. Among these techniques are operation overloading, region-based memory, and expression templates. There also exist several mathematical techniques which can yield high performance gains when applied to complex algorithms. For example, semi-analytical derivatives can reduce by orders of magnitude the runtime required to numerically solve and differentiate an algebraic equation. Open and practical problems include the extension of current packages to provide more specialized routines, and finding optimal methods to perform higher-order differentiation.

*Department of Statistics, Columbia University – contact: *charles.margossian@columbia.edu*

Graphical table of content



Automatic differentiation is a powerful tool to differentiate mathematical functions and algorithms. It has, over the past years, been applied to many branches of computational statistics. This article reviews some important mathematical and computational considerations required for its efficient implementation.

Introduction

A large number of numerical methods require derivatives to be calculated. Classical examples include the maximization of an objective function using gradient descent or Newton's method. In machine learning, gradients play a crucial role in the training of neural networks (Widrow & Lehr, 1990). Modelers have also applied derivatives to sample the high-dimensional posteriors of Bayesian models, using Hamiltonian Monte Carlo samplers (Neal, 2010; Betancourt, 2017a) or approximate these posteriors with variational inference (Kucukelbir, Tran, Ranganath, Gelman, & Blei, 2016). **As probabilistic models and algorithms gain in complexity, computing derivatives becomes a formidable task.** For many problems, we cannot calculate derivatives analytically, but only evaluate them numerically at certain points. Even when an analytical solution exists, working it out by hand can be mathematically challenging, time consuming, and error prone.

There exist three alternatives that automate the calculation of derivatives: (1) finite differentiation, (2) symbolic differentiation, and (3) automatic differentiation (AD). The first two methods can perform very poorly when applied to complex functions for a variety

Technique	Advantage(s)	Drawback(s)
Hand-coded analytical derivative	Exact and often fastest method.	Time consuming to code, error prone, and not applicable to problems with implicit solutions. Not automated.
Finite differentiation	Easy to code.	Subject to floating point precision errors and slow, especially in high dimensions, as the method requires at least D evaluations, where D is the number of partial derivatives required.
Symbolic differentiation	Exact, generates symbolic expressions.	Memory intensive and slow. Cannot handle statements such as unbounded loops.
Automatic differentiation	Exact, speed is comparable to hand-coding derivatives, highly applicable.	Needs to be carefully implemented, although this is already done in several packages.

Table 1: Summary of techniques to calculate derivatives.

of reasons summarized in Table 1; AD on the other hand, escapes many limitations posed by finite and symbolic differentiation, as discussed by (Baydin, Pearlmutter, Radul, & Siskind, 2018). Several packages across different programming languages implement AD and allow differentiation to occur seamlessly, while users focus on other programming tasks. For an extensive and regularly updated list of AD tools, the reader may consult www.autodiff.org. In certain cases, AD libraries are implemented as black boxes which support statistical and machine learning softwares, such as the python package `PyTorch` (Paszke et al., 2017) or the probabilistic programming language `Stan` (Carpenter et al., 2017).

Automatic differentiation does not, despite its name, fully automate differentiation and can yield inefficient code if naively implemented. For this reason, AD is sometimes referred to as *algorithmic* rather than *automatic* differentiation. We here review some important techniques to efficiently implement AD and optimize its performance. The article begins with a review of the fundamentals of AD. The next section presents various computational schemes, required for an efficient implementation and deployed in several recently developed packages. We then discuss strategies to optimize the differentiation of complex mathematical algorithms. The last section presents practical and open problems in the field.

Throughout the paper, we consider several performance metrics. Accuracy is, we argue, the most important one, but rarely distinguishes different implementation schemes of AD, most of which are exact up to arithmetic precision. On the other hand, differences arise when we compare run time and memory usage; run time, being easier to measure, is much more prominent in the literature. Another key consideration is applicability: does a method solve a broad or only a narrow class of problems? Finally, we discuss ease of implementation and readability, more subjective but nevertheless essential properties of a computer code. We do not consider compile time, mostly because the statistical applications of AD we have in mind compile a program once, before using it thousands, sometimes millions of times.

How automatic differentiation works

Given a target function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the corresponding $m \times n$ Jacobian matrix J has $(i, j)^{\text{th}}$ component:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

This matrix contains the partial derivatives of all the outputs with respect to all the inputs. If f has a one-dimensional output, as is the case when f is an objective function, the Jacobian matrix is simply the gradient. In practice we may care only about the partial derivatives with respect to some of the inputs and calculate a reduced Jacobian matrix. The partial derivatives with respect to these inputs, **respectively the corresponding columns of the Jacobian matrix, are called *sensitivities***.

Now suppose f is a composite function: $f(x) = h \circ g(x) = h(g(x))$, with $x \in \mathbb{R}^n$, $g : \mathbb{R}^n \rightarrow \mathbb{R}^k$ and $h : \mathbb{R}^k \rightarrow \mathbb{R}^m$. Applying the chain rule and elementary matrix multiplication:

$$J = J_{h \circ g} = J_h(g(x)) \cdot J_g(x)$$

with $(i, j)^{\text{th}}$ element:

$$J_{ij} = \frac{\partial f_i}{\partial x_j} = \frac{\partial h_i}{\partial g_1} \frac{\partial g_1}{\partial x_j} + \frac{\partial h_i}{\partial g_2} \frac{\partial g_2}{\partial x_j} + \dots + \frac{\partial h_i}{\partial g_k} \frac{\partial g_k}{\partial x_j}$$

More generally, if our target function f is the composite expression of L functions,

$$f = f^L \circ f^{L-1} \circ \dots \circ f^1 \quad (1)$$

the corresponding Jacobian matrix verifies:

$$J = J_L \cdot J_{L-1} \cdot \dots \cdot J_1 \quad (2)$$

To apply AD, we first need to express our target function f using a computer program. AD can then operate in one of two modes: *forward* or *reverse*. Let $u \in \mathbb{R}^n$. One application or *sweep* of forward-mode AD numerically evaluates the action of the Jacobian matrix on u , $J \cdot u$. As prescribed by Equation 2,

$$\begin{aligned} J \cdot u &= J_L \cdot J_{L-1} \cdot \dots \cdot J_3 \cdot J_2 \cdot J_1 \cdot u \\ &= J_L \cdot J_{L-1} \cdot \dots \cdot J_3 \cdot J_2 \cdot u_1 \\ &= J_L \cdot J_{L-1} \cdot \dots \cdot J_3 \cdot u_2 \\ &\dots \\ &= J_L \cdot u_{L-1} \end{aligned} \quad (3)$$

where the u_l 's verify the recursion relationship

$$\begin{aligned} u_1 &= J_1 \cdot u \\ u_l &= J_l \cdot u_{l-1} \end{aligned} \quad (4)$$

Hence, given a complex function f , we can break down the action of the Jacobian matrix on a vector, into simple components, which we evaluate sequentially. Even better, we can

choose splitting points in equation 1 (correspondingly equation 2) to generate efficient implementations of AD. We will take advantage of this fact when we consider checkpoints and super nodes in later sections of the paper.

Consider now a vector in the output space, $w \in \mathbb{R}^m$. A sweep of reverse-mode AD computes the action of the transpose of the Jacobian matrix on w , $J^T \cdot w$. The reasoning we used for forward mode applies here too and allows us to break down this operation into a sequence of simple operations. Often times, an AD program evaluates these simple operations analytically.

To further illustrate the mechanisms of AD, we consider a typical example from statistics¹. We will compute the gradient of a log likelihood function, for an observed variable y sampled from a normal distribution. The likelihood function is:

$$\text{Normal}(y \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi} \sigma} \exp\left(-\frac{1}{2\sigma^2}(y - \mu)^2\right)$$

with corresponding log likelihood function:

$$f(y, \mu, \sigma) = \log(\text{Normal}(y \mid \mu, \sigma^2)) = -\frac{1}{2} \left(\frac{y - \mu}{\sigma}\right)^2 - \log(\sigma) - \frac{1}{2} \log(2\pi) \quad (5)$$

Our goal is to compute the sensitivities of μ and σ , evaluated at $y = 10$, $\mu = 5$, and $\sigma = 2$. The above function can be broken up into a sequence of maps, yielding the composite structure of Equation 1:

$$\begin{aligned} (y, \mu, \sigma) &\rightarrow (y - \mu, \sigma) \\ &\rightarrow \left(\frac{y - \mu}{\sigma}, \sigma\right) \\ &\rightarrow \left(\frac{y - \mu}{\sigma}, \log(\sigma)\right) \\ &\dots \\ &\rightarrow -\frac{1}{2} \left(\frac{y - \mu}{\sigma}\right)^2 - \log(\sigma) - \frac{1}{2} \log(2\pi) \end{aligned} \quad (6)$$

Equation 4, which gives us the sequential actions of Jacobian matrices on vectors, then applies. Note that we have broken f into functions which only perform an elementary operation, such as $/$ or \log (division or logarithm). For example, the second map is $f^2 : (\alpha, \beta) \rightarrow (\alpha/\beta, \beta)$, and constructs a two dimensional vector by applying the *division* and the *identity* operators. By linearity of Equation 4, the identity operation is preserved. This means that, in our example, the second elements of u_1 and u_2 are identical. Hence, rather than explicitly computing every element in Equation 4, i.e. propagating derivatives through the maps outlined in Equation 6, it suffices to focus on individual operations, such as $/$ and \log , and moreover, any operation other than the identity operation.

To do this, Equation 5 can be topologically sorted into an *expression graph* (Figure 1). At the top of the graph, we have the final output, $f(y, \mu, \sigma)$, and at the roots the input

¹Based on an example from (Carpenter et al., 2015).

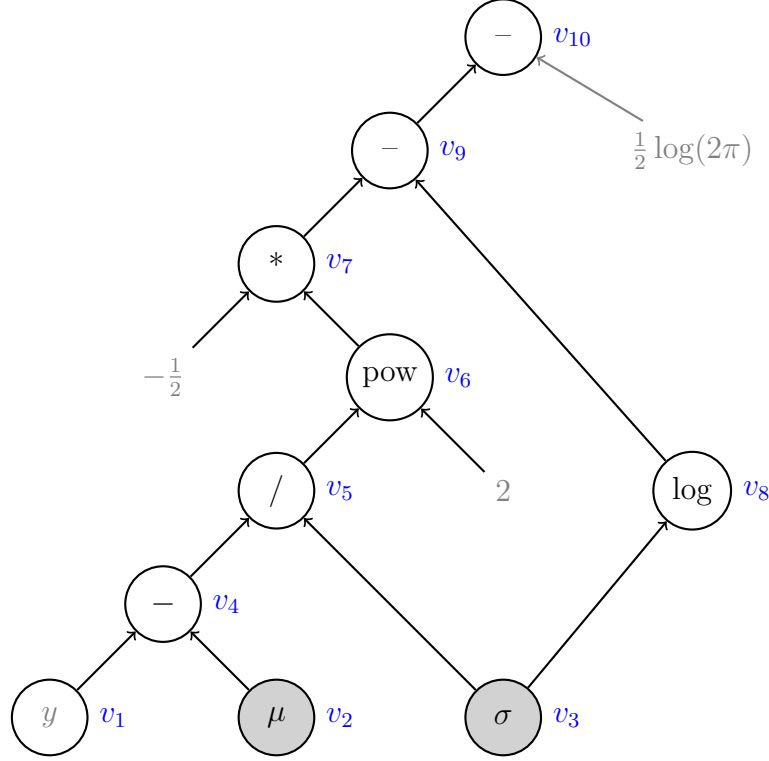


Figure 1: Expression graph for the log-normal density. *The above graph is generated by the computer code for Equation 5. Each node represents a variable, labeled v_1 through v_{10} , which is calculated by applying a mathematical operator to variables lower on the expression graph. The top node (v_{10}) is the output variable and the gray nodes the input variables for which we require sensitivities. The arrows represent the flow of information when we numerically evaluate the log-normal density. Adapted from Figure 1 of (Carpenter et al., 2015).*

variables y , μ , and σ . The nodes in between represent intermediate variables, obtained through elementary operations. A link between variables on the graph indicates an explicit dependence. We then know how to analytically differentiate operators on the expression graph and can get partial derivatives of composite expressions with the chain rule. For example:

$$\frac{\partial v_5}{\partial \mu} = \frac{\partial v_5}{\partial v_4} \times \frac{\partial v_4}{\partial \mu} = \frac{1}{\sigma} \times (-1)$$

We now describe the algorithmic mechanisms of the forward and reverse modes of AD.

Forward mode

Consider the direction, or *initial tangent*, $u \in \mathbb{R}^n$. One forward sweep computes $J \cdot u$. This corresponds to the total derivative of f with respect to u , or to be more precise to the directional or directed partial derivative. The complexity of this operation is linear in the complexity of f . Using fused-multiply adds, denoted OPS, as a metric for computational complexity, $\text{OPS}(f(x), J \cdot u(x)) \leq 2.5 \times \text{OPS}(f(x))$ (see chapter 4 of (Griewank & Walther,

2008)).

The right choice of u allows us to compute one column of the Jacobian matrix, that is the partial derivatives of all the outputs with respect to one input. Specifically, let u_j be 1 for the j^{th} input and 0 for all the other inputs. Then:

$$J_{\cdot j} = J \cdot u_j \quad (7)$$

Here $J_{\cdot j}$ denotes the j^{th} column of the matrix J . We can hence compute a full $m \times n$ Jacobian matrix in n forward sweeps. Naturally, we do not compute Equation 7 by doing a matrix operation, as this would require us to already know J . Instead, we proceed as follows.

Forward mode AD computes a directional derivative at the same time as it performs a forward evaluation trace. In the case where u_j is the above defined initial tangent, the software evaluates a variable and then calculates its partial directed derivative with respect to one root variable. Let \dot{v} denote the directed partial derivative of v with respect to that one root variable. Consider v_5 , which is connected to v_4 and v_3 . As the program sweeps through v_5 it evaluates (i) its value and (ii) its directional derivative. The latter can be calculated by computing the partial derivatives of v_5 with respect to v_4 and v_3 , and plugging in numerical values for v_3 , v_4 , and their directed partial derivatives \dot{v}_3 and \dot{v}_4 , which have already been evaluated. The code applies this procedure from the roots to the final output (Table 2).

Reverse mode

Now suppose instead of computing derivatives with respect to an input, we compute the *adjoints* with respect to an output. The adjoint of a variable x with respect to another variable z is defined as:

$$\bar{x} = \frac{\partial z}{\partial x}$$

Then for an initial *cotangent* vector, $\bar{u} \in \mathbb{R}^m$, one reverse mode sweep computes $J^T \bar{u}$, with complexity $\text{OPS}(f(x), J^T \bar{u}(x)) \leq 4 \times \text{OPS}(f(x))$ (see again chapter 4 of (Griewank & Walther, 2008)).

The right choice of w allows us to compute one row of the Jacobian matrix, i.e. the adjoints of all the inputs with respect to one output. If the function has a one-dimensional output, this row corresponds exactly to the gradient. If we pick w to be 1 for the i^{th} element and 0 for all other elements, one sweep computes the i^{th} row of J :

$$J_{i \cdot} = J^T \bar{w}_i \quad (8)$$

and we get the full $m \times n$ Jacobian matrix in m reverse sweeps.

To calculate the right-hand side of Equation 8, the algorithm proceeds as follows. After executing a forward evaluation trace, as was done for forward mode, the program makes a reverse pass to calculate the adjoints. We start with the final output, setting its adjoint with respect to itself to 1, and compute successive adjoints until we reach the root variables of interest (Table 3).

This procedure means the program must go through the expression graph twice: one forward trace to get the value of the function and the intermediate variables; and one reverse trace to get the gradient. This creates performance overhead because the expression graph

Forward evaluation trace	Forward derivative trace
$v_1 = y = 10$	$\dot{v}_1 = 0$
$v_2 = \mu = 5$	$\dot{v}_2 = 1$
$v_3 = \sigma = 2$	$\dot{v}_3 = 0$
$v_4 = v_1 - v_2 = 5$	$\dot{v}_4 = \frac{\partial v_4}{\partial v_1} \dot{v}_1 + \frac{\partial v_4}{\partial v_2} \dot{v}_2 = 0 + (-1) \times 1 = -1$
$v_5 = v_4 / v_3 = 2.5$	$\dot{v}_5 = \frac{\partial v_5}{\partial v_4} \dot{v}_4 + \frac{\partial v_5}{\partial v_3} \dot{v}_3 = \frac{1}{v_3} \times (-1) + 0 = -0.5$
$v_6 = v_5^2 = 6.25$	$\dot{v}_6 = \frac{\partial v_6}{\partial v_5} \dot{v}_5 = 2v_5 \times (-0.5) = -2.5$
$v_7 = -0.5 \times v_6 = 3.125$	$\dot{v}_7 = \frac{\partial v_7}{\partial v_6} \dot{v}_6 = -0.5 \times (-2.5) = 1.25$
$v_8 = \log(v_3) = \log(2)$	$\dot{v}_8 = \frac{\partial v_8}{\partial v_3} \dot{v}_3 = 0$
$v_9 = v_7 - v_8 = 3.125 - \log(2)$	$\dot{v}_9 = \frac{\partial v_9}{\partial v_7} \dot{v}_7 + \frac{\partial v_9}{\partial v_8} \dot{v}_8 = 1 \times 1.25 + 0 = 1.25$
$v_{10} = v_9 - 0.5 \log(2\pi) = 3.125 - \log(4\pi)$	$\dot{v}_{10} = \frac{\partial v_{10}}{\partial v_9} \dot{v}_9 = 1.25$

Table 2: Forward-mode AD. *The forward derivative trace computes the derivative of a log Normal density with respect to μ . The program begins by initializing the derivatives of the root variables (1 for μ , 0 for the other inputs). The directional derivative is then computed at each node and numerically evaluated using the chain rule, and previously evaluated variables and directional derivatives. To get the sensitivity for σ , we must compute a new forward derivative trace.*

Reverse adjoint trace

$$\bar{v}_{10} = 1$$

$$\bar{v}_9 = \frac{\partial v_{10}}{\partial v_9} \bar{v}_{10} = 1 \times 1 = 1$$

$$\bar{v}_8 = \frac{\partial v_9}{\partial v_8} \bar{v}_9 = (-1) \times 1 = -1$$

$$\bar{v}_7 = \frac{\partial v_9}{\partial v_7} \bar{v}_9 = 1 \times 1 = 1$$

$$\bar{v}_6 = \frac{\partial v_7}{\partial v_6} \bar{v}_7 = (-0.5) \times 1 = -0.5$$

$$\bar{v}_5 = \frac{\partial v_6}{\partial v_5} \bar{v}_6 = 2v_5 \times \bar{v}_6 = 2 \times 2.5 \times (-0.5) = -2.5$$

$$\bar{v}_4 = \frac{\partial v_5}{\partial v_4} \bar{v}_5 = \frac{1}{v_3} \times (-2.5) = 0.5 \times (-2.5) = -1.25$$

$$\bar{v}_3 = \frac{\partial v_5}{\partial v_3} \bar{v}_5 + \frac{\partial v_8}{\partial v_3} \bar{v}_8 = -\frac{v_4}{v_3^2} \times (-2.5) + \frac{1}{v_3} \times (-1) = 2.625$$

$$\bar{v}_2 = \frac{\partial v_4}{\partial v_2} \bar{v}_4 = (-1) \times (-1.25) = 1.25$$

Table 3: Reverse-mode AD. After executing a forward evaluation trace, the reverse derivative trace computes the gradient of the log Normal density with respect to μ and σ . For each node, an adjoint is computed and then combined with previously calculated adjoints using the chain rule. This process should be compared to forward mode AD, depicted in Table 2: because we compute adjoints, rather than starting with the root variables, we start with the output and then work our way back to the roots (hence the term “reverse”). One reverse mode sweep gives us the full gradient. The input and outputs of the trace are highlighted in gray.

and the values of the intermediate variables need to be stored in memory. One way of doing this efficiently is to have the code make a *lazy evaluation* of the derivatives. A lazy evaluation does *not* evaluate a statement immediately when it appears in the code, but stores it in memory and evaluates it when (and only if) the code explicitly requires it. As a result, we only evaluate expressions required to compute the gradient, or more generally the object of interest. There are several other strategies to efficiently handle the memory overhead of reverse-mode AD, which we will discuss in the section on *computational implementation*.

Forward or reverse mode?

For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, suppose we wish to compute *all* the elements of the $m \times n$ Jacobian matrix: which mode of AD should we use? Ignoring the overhead of building the expression graph, reverse mode, which requires m sweeps, performs better when $n > m$. With the relatively small overhead, the performance of reverse-mode AD is superior when $n \gg m$, that is when we have many inputs and few outputs. This is the case when we map

Dimension of input, n	1	8	29	50
Relative runtime	1.13	0.81	0.45	0.26

Table 4: Relative runtime to compute a gradient with reverse-mode, when compared to forward-mode. The table summarizes results from an experiment conducted by (Baydin et al., 2018). The runtimes are measured by differentiating $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Note forward-mode AD is more efficient when $n = 1$, but the result flips as we increase n .

many model parameters to an objective function, and makes reverse mode highly applicable to high-dimensional modeling.

However, if $n \leq m$ forward mode performs better. Even when we have a comparable number of outputs and inputs, as was the case in the log-normal density example, forward mode can be more efficient, since there is less overhead associated with storing the expression graph in memory in forward than in reverse mode. (Baydin et al., 2018) compare the runtime of forward and reverse-mode AD when applied to a statistical mechanics problem. The differentiated function is a many to one map, that is $f : \mathbb{R}^n \rightarrow \mathbb{R}$. For $n = 1$, forward-mode AD proves more efficient, but the result flips as n increases (Table 4).

While a certain mode may overall work best for a function, we can also look at intermediate functions that get called inside a target function. Such intermediate functions may be best differentiated using another mode of AD. (Phipps & Pawlowski, 2012) calculate derivatives of intermediate expressions with reverse mode in an overall forward mode AD framework. They consider two forward-mode implementations; both use *expression templates*, a computational technique we will discuss in a few sections, and one of them uses *caching*. Caching stores the values and partial derivatives of intermediate expressions, and insure they only get evaluated once. As a case study, (Phipps & Pawlowski, 2012) consider an application to fluid mechanics, and report a 30% reduction in runtime with intermediate reverse-mode, compared to standard forward AD without caching. The performance gain when forward AD already uses caching is however minor (less than 10%).

Another example where forward and reverse mode AD are both applied is the computation of higher-order derivatives such as Hessians or Hessian vector products (see for example (Pearlmutter, 1994)). We will dwell a little longer on this in our discussion on higher-order derivatives.

Computational implementation

A computationally naive implementation of AD can result in prohibitively slow code and excess use of memory. Careful considerations can mitigate these effects.

There exist several studies which measure the performance of different implementation schemes but these studies present limitations. First, they are indirect tests, as they often compare packages which have several distinguishing features. Secondly, computer experiments usually focus on a limited class of problems; there exists no standard set of problems spanning the diversity of AD applications and the development of a package is often moti-

vated by a specific class of problems. Nevertheless, these test results work well as proof of concepts, and are aggregated in our study of computational techniques.

Our focus is mainly on C++ libraries, which have been compared to one another by (Hogan, 2014) and (Carpenter et al., 2015). A common feature is the use of *operator overloading* rather than *source transformation*, two methods we will discuss in the upcoming sections. Beyond that, these libraries exploit different techniques, which will be more or less suited depending on the problem at hand. A summary of computational techniques is provided at the end of this section (Table 6).

Source transformation

A classic approach to compute and execute AD is *source transformation*, as implemented for example in the Fortran package ADIFOR (Bischof, Khademi, Mauer, & Carle, 1996) and the C++ packages TAC++ (Voßbeck, Giering, & Kaminski, 2008) and Tapenade (Hascoet & Pascual, 2013). We start with the source code of a computer program that implements our target function. A preprocessor then applies differentiation rules to the code, as prescribed by elementary operators and the chain rule, and generates new source code, which calculates derivatives. The source code for the evaluation and the one for differentiation are then compiled and executed together.

A severe limitation with source transformation is that it can only use information available at *compile time*, which prevents it from handling more sophisticated programming statements, such as while loops, C++ templates, and other object-oriented features (Voßbeck et al., 2008; Hogan, 2014). There exist workarounds to make source transformation more applicable, but these end up being similar to what is done with operator overloading, albeit significantly less elegant (Bischof & Bücker, 2000).

Moreover, it is commonly accepted in the AD community that *source transformation works well for Fortran or C*. (Hascoet & Pascual, 2013) argue it is the choice approach for reverse-mode AD, although they also admit that, from a developer’s perspective, implementing a package that deploys source transformation demands a considerable amount of effort. Most importantly, *source transformation cannot handle typical C++ features*. For the latter, operator overloading is the appropriate technology.

Operator overloading

Operator overloading is implemented in many packages, including the C++ libraries: Sacado (Gay, 2005), Adept (Hogan, 2014), Stan Math (Carpenter et al., 2015), and CoDiPack (Sagebaum, Albring, & Gauger, 2017). The key idea is to introduce a new class of objects, which contain the value of a variable on the expression graph and a *differential component*. Not all variables on the expression graph will belong to this class. But the root variables, which require sensitivities, and all the intermediate variables which depend (directly or indirectly) on these root variables, will. In a forward mode framework, the differential component is the derivative with respect to one input. In a reverse mode framework, it is the adjoint with respect to one output². Operators and math functions are overloaded to handle these

²Provided we use, for forward and reverse mode, respectfully the right initial tangent and cotangent vector.

new types. We denote standard and differentiable types `real` and `var` respectively.

Memory management

AD requires care when managing memory, particularly in our view when doing reverse-mode differentiation and implementing operator overloading.

In forward mode, one sweep through the expression graph computes both the numerical value of the variables and their differential component with respect to an initial tangent. The memory requirement for that sweep is then simply twice the requirement for a function evaluation. What is more, a `var` object, which corresponds to a node on the expression graph, needs only to be stored in memory temporally. Once all the nodes connected to that object have been evaluated, the `var` object can be discarded.

When doing a reverse mode sweep, matters are more complicated because the forward evaluation trace happens first, and only then does the reverse AD trace occur. To perform the reverse trace, we need to access (i) the expression graph and (ii) the numerical values of the intermediate variables on that expression graph, required to calculate derivatives. These need to be stored in a persistent memory arena or *tape*. Note further that neither (i) nor (ii) is known at compile time if the program we differentiate includes loops and conditional statements, and, in general, the memory requirement is dynamic.

Retaping

In certain cases, a tape can be applied to multiple AD sweeps. An intuitive example is the computation of an $n \times m$ Jacobian matrix, where $n > 1$ and $m > 1$. Rather than reconstructing the expression graph and reevaluating the needed intermediate variables at each sweep, we can instead store the requisite information after the first sweep, and reuse it. Naturally, this strategy only works because the expression graph and the intermediate variables do not change from sweep to sweep.

If we evaluate derivatives at multiple points, the intermediate variables almost certainly need to be reevaluated and, potentially, stored in memory. On the other hand, the expression graph may not change. Thence a single tape for the expression graph can be employed to compute AD sweeps at multiple points. This scheme is termed *retaping*; implementations can be found in ADOL-C (Griewank, Juedes, & Utke, 1999; Walther & Griewank, 2012) and CppAD (Bell, 2012).

The conservation of the expression graph from point to point does however not hold in general. Our target function may contain conditional statements, which control the expression graph our program generates. In such cases, new conditions require new tapes. Retaping can then be used, provided we update the expression graph as soon as the control flow changes.

Checkpoints

Checkpointing reduces peak memory usage and more generally trades runtime and memory cost. Recall our target function f is a composite of, say, L simpler functions f^l . Combining equations 2 and 8, the action of the transpose Jacobian matrix, w' , computed by one reverse

sweep of AD verifies

$$\begin{aligned} w' &= J^T \cdot w \\ &= (J_L \cdot J_{L-1} \cdot \dots \cdot J_1)^T \cdot w \end{aligned} \tag{9}$$

where w is the initial cotangent. We can split w' into a sequence of K components

$$\begin{aligned} w' &= (J_{\phi(1)} \cdot \dots \cdot J_1)^T w_1 \\ w_1 &= (J_{\phi(2)} \cdot \dots \cdot J_{\phi(1)+1})^T w_2 \\ &\dots \\ w_{K-1} &= (J_{\phi(K-1)} \cdot \dots \cdot J_{\phi(K-2)+1})^T w \end{aligned} \tag{10}$$

which can be sequentially evaluated by a reverse AD sweep, starting from the last line. The index function ϕ tells us where in the composite function the splits occur.

Correspondingly, the program that implements f can be split into K sections, with a *checkpoint* between each section. We then have several options, the first one being the *recompute-all* approach. Under this scheme, the program begins with a forward evaluation sweep, but here is the catch: we do not record the expression graph and the intermediate values, as we would in previously discussed implementations of reverse-mode AD, until we reach the last checkpoint. That is we only record from the last checkpoint onward. When we begin the reverse sweep, the size of the tape is relatively small, thence the reduction in peak memory cost. Of course, we can only perform the reverse sweep from the output to the last checkpoint. Once we reach the latter, we rerun the forward evaluation sweep from the beginning, and only record between the second-to-last and the last checkpoint, which gives us the requisite information for the next reverse sweep. And so on. The reason we call this approach *recompute-all* is because we rerun a partial forward evaluation trace, with partial recording, for each checkpoint (figure 2). Moreover, the reduction in peak memory comes at the cost of doing multiple forward evaluation traces.

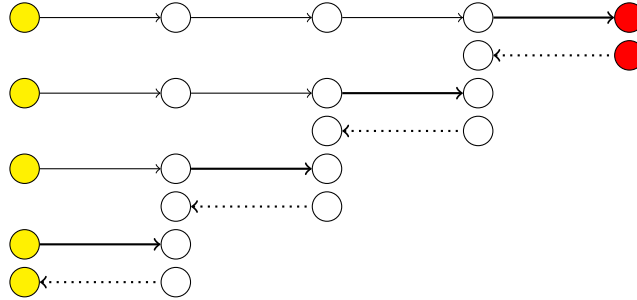


Figure 2: Checkpointing with the recompute-all approach. *In the above sketch, the target function is broken into 4 segments, using three checkpoints (white nodes). The yellow and red nodes respectively represent the input and output of the function. During the forward sweep, we only record when the arrow is thick. The dotted arrow represents a reverse AD sweep. After each reverse sweep, we start a new forward evaluation sweep from the input.*

It may be inefficient to rerun forward sweeps from the start every time. We can instead store the values of intermediate variables at every checkpoint, and then perform forward

sweeps only between two checkpoints (figure 3). Under this scheme, the forward evaluation trace is effectively run twice. This variant of checkpointing is termed *store-all*. Storing intermediate results at each checkpoint may for certain problems be too memory expensive. Fortunately, recompute-all and store-all constitute two ends of a spectrum: in between methods can be used by selecting a subset of the checkpoints where the values are stored. Such checkpoints constitute *snapshots*, and give us more control on peak memory usage.

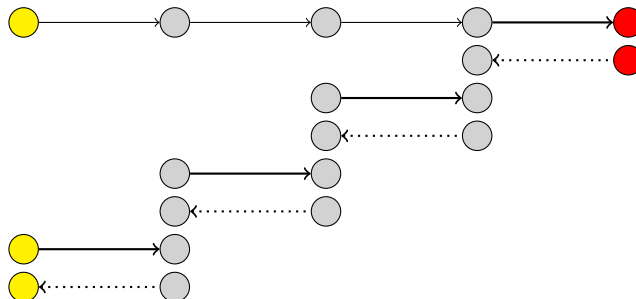


Figure 3: Checkpointing with the store-all approach. *This time, the forward sweep records intermediate values at each checkpoint. The forward sweep can then be rerun between two checkpoints, as opposed to starting from the input.*

A natural question is where should we place the checkpoints? Ideally, we would like, given a memory constraint, to reduce runtime. There is in general, as argued by (Hascoet & Pascual, 2013), no optimal placement and users should be given the freedom to place checkpoints by hand. In the case where we can split the the program for f into sequential *computational steps*, with comparable computational costs, placing the checkpoints according to a *binomial partitioning* scheme (Griewank, 1992) produces an optimal placement (Grimm, Pottier, & Rostaing-Schmidt, 1996).

There are several packages which implement checkpointing: for example **Adol-C** and **CppAD** couple the method with operator overloading, while it is used with source transformation in **Tapenade**.

We conclude this section by drawing a connection between checkpointing and parallelizing AD. Suppose a section of f can be split into segments we can evaluate in parallel. Specifically, each segment is evaluated on a separate computer core. Conceptually, we can place checkpoints around these segments, and create separate tapes for each of these segments. As a result, memory is kept local and does not need to be shared between cores. An example of such a procedure is the *map reduce* functionality in **Stan Math**, describe in the *Stan book* (Stan Development Team, 2018). Note this routine is not explicitly described as a checkpointing method.

Region Based Memory

If we allocate and free memory one intermediate variable at a time, the overhead cost becomes quite severe. A better alternative is to do **region-based memory management** (see for example (Gay & Aiken, 2001)). Memory for the calculation of derivatives is allocated element wise on a custom stack. Once the derivatives have been evaluated, the entire stack is destroyed and

the associated memory freed. This spares the program the overhead of freeing memory one element at a time. (Gay, 2005) implements this technique in the C++ package **RAD** (*Reverse AD*, the reverse mode version of **Sacado**), and benchmarks its performance against the C/C++ package **ADOL-C**. The results show **RAD** runs up to twice as fast, when differentiating quality mesh functions with varying levels of complexity.

Expression templates

Rather than overloading an operator such that it returns a custom object, we can code the operator to return an *expression template* (Veldhuizen, 1995). This technique, which builds on lazy evaluation, is widely used in packages for vector and matrix operations and was first applied to AD in the **Fad** package (Aubert, Di Cesare, & Pironneau, 2001). (Phipps & Pawlowski, 2012) recommend several techniques to improve expression templates and showcase its implementation in **Sacado**. (Hogan, 2014) and more recently (Sagebaum et al., 2017) apply the method to do full reverse-mode AD, respectively in **Adept** and **CoDiPack**.

Under the expression template scheme, operators return a specialized object that characterizes an intermediate variable as a node in the expression graph. As an example, consider the following programming statement:

```
var x;
real a;
var b = cos(x);
var c = b * a;
```

Take the *multiply* or $*$ operator, which is a map $f : \mathbb{R}^2 \rightarrow \mathbb{R}$. Rather than returning a **var** object, we overload $*$ to return a template object of type `multiply<T_a, T_b>`, where `T_a` and `T_b` designate the types of a and b . **T_a or T_b can be real, var, or expression templates**. In the above example, $b = \cos x$, x is a **var**, and a a **real**. Then c is assigned an object of type `multiply<cos<var>, real>`, rather than an evaluated **var** object (Table 5). The nodes in the expression graph hence contain references. An optimizing compiler can then combine the nodes to generate more efficient code, which would be functionally equivalent to:

```
c.val_ = cos(x).val_ * a;
x.adj_ = c.adj_ * a * (- sin(x).val_)
```

where `val_` and `adj_` respectively denote the value and the adjoint stored in the **var** object. This operation is called *partial evaluation*. Notice that **b** does not appear in the above programming statement. **Moreover, the use of expression templates removes temporary var objects, which saves memory and eliminates the overhead associated with constructing intermediate variables**. Furthermore, the code can eliminate redundancies, such as the addition and later subtraction of one variable.

variable	type
x	var
a	real
b	cos<var>
c	multiply<cos<var>, real>
Algorithm	b = cos(x); c = b * a;

Table 5: Variable types when using expression templates

(Hogan, 2014) evaluates the benefits of coupling expression templates and AD by benchmarking **Adept** against other AD packages that use operator overloading, namely **Adol-C**, **CppAD**, and **Sacado**. His computer experiment focuses on an application in fluid dynamics, using a simple linear scheme and a more complex nonlinear method. In these specific instances, **Adept** outperforms other packages, both for forward and reverse mode AD: it is 2.6 - 9 times faster and 1.3 - 7.7 times more memory efficient, depending on the problem and the benchmarking package at hand. (Carpenter et al., 2015) corroborate the advantage of using expression templates by testing two implementations of a function that returns the log sum of exponents. The first implementation looks as follows:

```
for (int = 0; i < x.size(); i++)
    total = log(exp(total) + exp(x(i)));
return total;
```


This code is inefficient because the log function gets called multiple times, when it could be called once, as below:

```
for (int = 0; i < x.size(); i++)
    total += exp(x(i));
return log(total);
```

Stan Math requires 40% less time to evaluate and differentiate the function, when we use the second implementation instead of the first one. On the other hand **Adept** achieves optimal speed with both implementations. When applied to the second algorithm, both packages perform similarly. Hence, while careful writing of a function can match the results seen with expression templates, this requires more coding effort from the user.

Mathematical implementation

We now study several mathematical considerations users can make to optimize the differentiation of a complex function. The here discussed techniques are generally agnostic to which AD package is used; they may however already be implemented as built-in features in



Method	Benefits	Limitations	Packages
Source transformation	Can be optimized by hand or with a compiler.	Can only differentiate functions which are fully defined at compile time.	AdiFor, TAC++ Tapenade
Operator Overloading	Can handle most programing statements.	Memory handling requires some care.	All the packages listed below.
Retaping	Fast when we differentiate through the same expression graph multiple times.	Applicable only if the expression graph is conserved from point to point. When the graph changes a new tape can be created.	Adol-C, CppAd
Checkpointing	Reduces peak memory usage.	Longer runtime. An optimal placement is binary partition, but this scheme does not always apply.	Adol-C, CppAd (Tapenade for source transformation).
Region-based memory	Improves speed and memory.	Makes the implementation less transparent. Some care is required for nested AD.	Adept, Sacado, Stan Math
Expression templates	Removes certain redundant operations, whereby the code becomes faster and more memory efficient, and allows user to focus on code clarity.		Adept, CoDiPack Sacado (with some work).

Table 6: Summary of computation techniques to optimize AD. *The right most column lists packages discussed in this section that implement a given computational method. This is by no means a comprehensive survey of all AD packages. A further word of caution: when choosing a package, users should also look at other criterions, such as ease of use, extensibility, and built-in mathematical functions.*

certain libraries. We exemplify their use and test their performance using **Stan Math** and reverse-mode AD³. The driving idea in this section is to gain efficiency by reducing the size of the expression graph generated by the code.

Reducing the number of operations in a function

A straightforward way of optimizing code is to minimize the number of operations in a function. This can be done by storing the result of calculations that occur multiple times inside intermediate variables. When evaluating a function, introducing intermediate variables reduces runtime but increases memory usage. **With AD, eliminating redundant operations also reduces the generated expression graph, which means intermediate variables both improve runtime and memory usage.**

Consider for example the implementation of a function that returns a 2×2 matrix exponential. The matrix exponential extends the concept of a scalar exponential to matrices and can be used to solve linear systems of ordinary differential equations (ODEs). There exist several methods to approximate matrix exponentials, the most popular one being the Padé approximation, coupled with scaling and squaring; for a review, see (Moler & Van Loan, 2003). For the 2×2 matrix,

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

the matrix exponential can be worked out analytically (see (Rowland & Weisstein, n.d.)). Requiring $\Delta := \sqrt{(a-d)^2 + 4bc}$ to be real, we have:

$$\exp(A) = \frac{1}{\Delta} \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \quad (11)$$

where

$$\begin{aligned} m_{11} &= e^{(a+d)/2} [\Delta \cosh(\Delta/2) + (a-d) \sinh(\Delta/2)] \\ m_{12} &= 2be^{(a+d)/2} \sinh(\Delta/2) \\ m_{21} &= 2ce^{(a+d)/2} \sinh(\Delta/2) \\ m_{22} &= e^{(a+d)/2} [\Delta \cosh(\Delta/2) + (d-a) \sinh(\Delta/2)] \end{aligned}$$

The above expression only contains elementary operators, and can be differentiated by any standard AD library. A “direct” implementation in **C++** might simply translate the above mathematical equations and look as follows:

³Similar results are expected with forward mode.

Standard implementation of 2×2 matrix exponential

```
T a = A(0, 0), b = A(0, 1), c = A(1, 0), d = A(1, 1), delta;
delta = sqrt(square(a - d) + 4 * b * c);
Matrix<T, Dynamic, Dynamic> B(2, 2);
B(0, 0) = exp(0.5 * (a + d)) * (delta * cosh(0.5 * delta) + (a - d) * sinh(0.5 * delta));
B(0, 1) = 2 * b * exp(0.5 * (a + d)) * sinh(0.5 * delta);
B(1, 0) = 2 * c * exp(0.5 * (a + d)) * sinh(0.5 * delta);
B(1, 1) = exp(0.5 * (a + d)) * (delta * cosh(0.5 * delta) + (d - a) * sinh(0.5 * delta));

return B / delta;
```

Note we use the matrix library **Eigen** (Guennebaud, Jacob, et al., 2010). **T** represents a template type which, in need, will either be a **real** or a **var**. An optimized implementation removes redundant operations and introduces variables to store intermediate results:

Optimized implementation of 2×2 matrix exponential

```
T a = A(0, 0), b = A(0, 1), c = A(1, 0), d = A(1, 1), delta;
delta = sqrt(square(a - d) + 4 * b * c);
Matrix<T, Dynamic, Dynamic> B(2, 2);
T half_delta = 0.5 * delta;
T cosh_half_delta = cosh(half_delta);
T sinh_half_delta = sinh(half_delta);
T exp_half_a_plus_d = exp(0.5 * (a + d));
T Two_exp_sinh = 2 * exp_half_a_plus_d * sinh_half_delta;
T delta_cosh = delta * cosh_half_delta;
T ad_sinh_half_delta = (a - d) * sinh_half_delta;

B(0, 0) = exp_half_a_plus_d * (delta_cosh + ad_sinh_half_delta);
B(0, 1) = b * Two_exp_sinh;
B(1, 0) = c * Two_exp_sinh;
B(1, 1) = exp_half_a_plus_d * (delta_cosh - ad_sinh_half_delta);

return B / delta;
```

The two codes output the same mathematical expression, but the second implementation is about a third faster, as a result of producing an expression graph with less nodes (Table 7). This is measured by applying both implementations to 1000 randomly generated 2×2 matrices, labelled “A”. We compute the run time by looking at the wall time required to evaluate $\exp(A)$ and differentiate it with respect to every elements in A (thereby producing 16 partial derivatives). The experiment is repeated 100 times to quantify the variation in the performance results.

Unfortunately the optimized approach requires more coding effort, produces more lines of code, and ultimately hurts clarity, as exemplified by the introduction of a variable called `ad_sinh_half_delta`. Whether the trade-off is warranted or not depends on the user’s preferences. Note expression templates do not perform the here-discussed optimization, as a

compiler may fail to identify expressions which get computed multiple times inside a function.

Implementation	Number of nodes	Run time (ms)	Relative run time
Standard	41	2.107 ± 0.35	1.0
Optimized	25	1.42 ± 0.28	0.68

Table 7: Run time for a standard and an optimized implementation of a 2×2 matrix exponential. *The optimized implementation of the matrix exponential removes redundant operations by introducing new intermediate variables. As a result, the code produces an expression with less nodes, which leads to more efficient differentiation.*

Optimizing with super nodes

There exist more sophisticated approaches to reduce the size of an expression graph. This is particularly useful, and for practical purposes even necessary, when differentiating iterative algorithms. Our target function, f , may embed an implicit function, f^k , such as the solution to an equation we solve numerically. Numerical solvers often use iterative algorithms to find such a solution. The steps of these algorithms involve simple operations, hence it is possible to split f^k into elementary functions, here indexed 1, ..., M :

$$\begin{aligned} f &= f^L \circ \dots \circ f^{k+1} \circ f^k \circ \dots \circ f^1 \\ &= f^L \circ \dots \circ f^{k+1} \circ f_M^k \circ f_{M-1}^k \circ \dots \circ f_1^k \circ \dots \circ f^1 \end{aligned}$$

Note M , the number of elementary functions we split f^k into, depends on the number of steps the numerical solver takes, and hence varies from point to point. Using the above we can then easily deploy our usual arsenal of tools to do forward or reverse-mode AD. But doing so produces large expression graphs, which can lead to floating point precision errors, excessive memory usage, and slow computation. To mitigate these problems, we can elect to not split f^k , and use a custom method to compute f^k 's contribution to the action of the Jacobian matrix. A more algorithmic way to think about this is to say we do not treat f^k as a function generating a graph, but as a stand-alone operator which generates a node or *super node*. A super node uses a specialized method to compute Jacobian matrices and often produces a relatively small expression graph.

As a case study, we present several implementations of a numerical algebraic solver, and run performance tests to measure run time. Ease of implementation is also discussed. In particular, we examine a well-known strategy which consists in using the implicit function theorem. For a more formal discussion of the subject, we recommend (Bell & Burke, 2008) and chapter 15 of (Griewank & Walther, 2008).

Another example where super nodes yield a large gain in performance is the numerical solving of ODEs, where the iterative algorithm is used to simultaneously evaluate a solution and compute sensitivities. Section 13 of (Carpenter et al., 2015) provides a nice discussion on this approach. Finally, (Giles, 2008) summarizes several results to efficiently compute ma-

trix derivatives, including matrix inverses, determinants, matrix exponentials, and Cholesky decompositions.

Differentiating a numerical algebraic solver

We can use a numerical algebraic solver to find the root of a function and solve nonlinear algebraic equations. Formally, we wish to find $y^* \in \mathbb{R}^N$ such that:

$$f(y^*, \theta) = 0$$

where $\theta \in \mathbb{R}^K$ is a fixed vector of auxiliary parameters, or more generally a fixed vector of variables for which we require sensitivities. That is we wish to compute the $N \times K$ Jacobian matrix, J , with $(i, j)^{\text{th}}$ entry:

$$J_{i,j} = \frac{\partial y_i^*}{\partial \theta_j}(y^*, \theta)$$

Consider, for demonstrating purposes, an implementation of Newton's method. For convenience, we note J^y the Jacobian matrix, which contains the partial derivatives f with respect to y .

Standard iterative algorithm: Newton solver with a fixed step size

1. Start with an initial guess y_0 .
2. While a *convergence criterion* has not been met:

$$y^{(i+1)} = y^{(i)} - [J^y(y^{(i)}, \theta)]^{-1} f(y^{(i)}, \theta)$$

3. Return $y^{(I)}$, where I is the number of steps required to reach convergence, or a warning message stating convergence could not be reached.
-

In the case where an analytical expression for J^y is provided, we can readily apply regular AD to differentiate the algorithm. If we are working with overloaded operators, this **simply means templating the solver so that it can handle `var` types**. Unfortunately, if the number of steps required to reach convergence is high, the algorithm produces a large expression graph. The problem is exacerbated when J^y is not computed analytically. Then AD must calculate the higher-order derivative:

$$\frac{\partial^2 f}{\partial y \partial \theta}$$

Coding a method that returns such a derivative requires the implementation of *nested* AD, and a careful combination of the reverse and forward mode chain rule. Tackling this problem with brute-force AD hence demands a great coding effort and is likely to be prohibitively slow.



An alternative approach is to use the implicit function theorem which tells us that, under some regularity conditions in the neighborhood of the solution:

$$J = -[J^y(y^*, \theta)]^{-1} J^\theta(y^*, \theta) \quad (12)$$

where, consistent with our previous notation, J^θ is the Jacobian matrix of f with respect to θ . We have further made the dependence on y and θ explicit. Note the right-hand side is evaluated at the solution y^* .

This does not immediately solve the problem at hand but it significantly simplifies it. Computing $J^y(y^*, \theta)$ and $J^\theta(y^*, \theta)$ is, in practice, straightforward. In simple cases, these partials can be worked out analytically, but more generally, we can use AD. Many algorithms, such as Newton’s method and Powell’s dogleg solver (Powell, 1970) already compute the matrix J^y to find the root, a result that can then be reused. Note furthermore that with Equation 12, calculating J reduces to a first-order differentiation problem.

Ideally, super nodes are already implemented in the package we plan to use. If this is not the case, the user can attempt to create a new function with a custom differentiation method. The amount of effort required to do so depends on the AD library at hand and the user’s coding expertise; the paper further discusses this point in its section on *Extensibility*. For now, our advice is to build new operators when brute force AD is prohibitively slow or when developing general purpose tools.

To better inform the use of super nodes, we compare the performance of two dogleg solvers. The first one uses a templated version of the algorithm and applies regular AD. The second one uses Equation 12. For simplicity, we provide J^y in analytical form; for the second solver, J^θ is computed with AD. The solvers are applied to an archetypical problem in pharmacometrics, namely the modeling of steady states for patients undergoing a medical treatment. The details of the motivating problem are described in the appendix. The number of states in the algebraic equation varies from 2 to 28, and the number of parameters which require sensitivities respectively from 4 to 30. We use **Stan Math**’s built-in algebraic solver as a benchmark, a dogleg algorithm which also uses Equation 12 but computes J^y with AD.

The experiment shows using the implicit function theorem is orders of magnitudes faster than relying on standard AD (Figure 4 and table 8). This also holds for the built-in algebraic solver which automatically differentiates f with respect to y . Access to J^y in its analytical form yields a minor gain in performance, which becomes more obvious when we increase the number of states in the algebraic equation (Figure 5).

Open and practical problems

Many AD packages are still being actively developed; hence we expect some of the issues we present to be addressed in the years to come.

General and specialized code

The first problem we note is that no single package comprehensively implements all the optimization techniques outlined in this paper – perhaps for several good reasons. First of all, two methods may not be compatible with one another. In most cases however, there

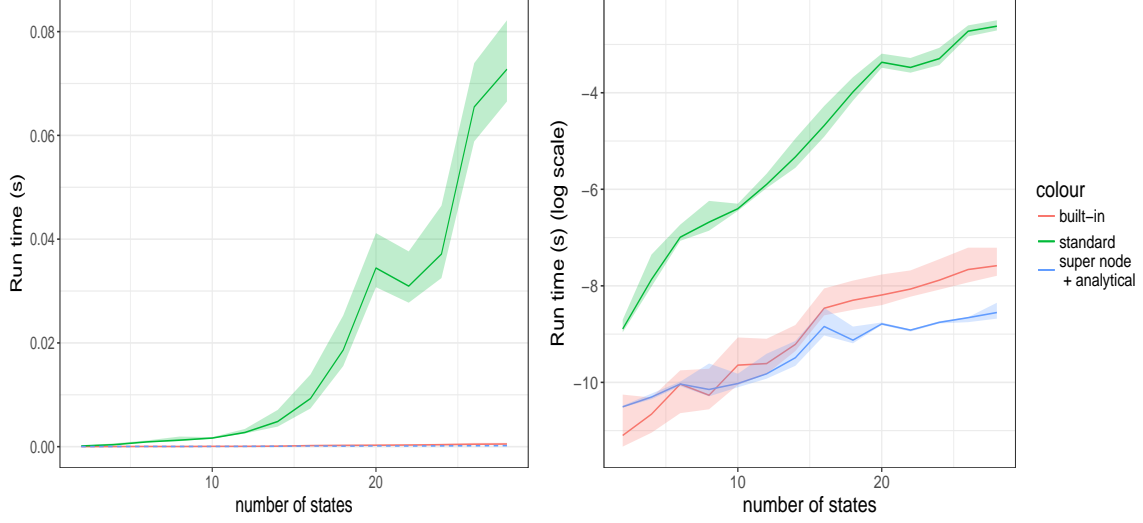


Figure 4: Run time to solve and differentiate a system of algebraic equations. *All three solvers deploy Powell’s dogleg method, an iterative algorithm that uses gradients to find the root, y^* , of a function $f = f(y, \theta)$. The standard solver (green) uses an analytical expression for J^y and AD to differentiate the iterative algorithm. The super node solver (blue) also uses an analytical expression for J^y and the implicit function theorem to compute derivatives. The built-in solver (red) uses AD to compute J^y and the implicit function theorem. The computer experiment is run 100 times and the shaded areas represent the region encompassed by the 5th and 95th quantiles. The plot on the right provides the run time on a log scale for clarity purposes. Plot generated with `ggplot2` (Wickham, 2009).*

number of states	super node		
	standard	+ analytical	built-in
4	0.383 ± 0.0988	0.0335 ± 0.00214	0.0235 ± 0.00842
12	2.74 ± 0.396	0.0542 ± 0.0123	0.0670 ± 0.0259
20	34.4 ± 3.28	0.152 ± 0.0145	0.278 ± 0.173
28	72.8 ± 5.56	0.193 ± 0.0301	0.51 ± 0.214

Table 8: Run time (ms) to solve and differentiate a system of algebraic equations. *This table complements Figure 4, and highlights the results for a selected number of states. The uncertainty is computed by running the experiment 100 times and computing the sample standard deviation.*

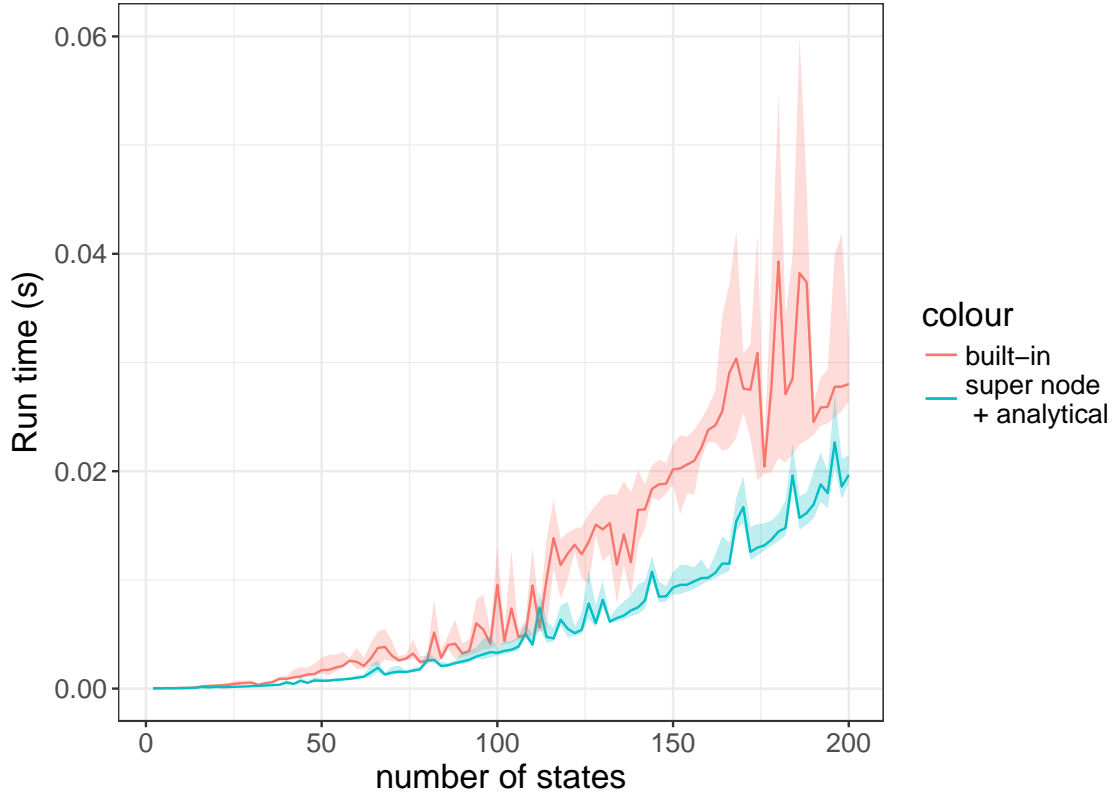


Figure 5: Run time to solve and differentiate a system of algebraic equations. *This time we only compare the solvers which use the implicit function theorem. The “super node + analytical” solver uses an analytical expression for J^y both to solve the equation and compute sensitivities. The built-in solver uses AD to compute J^y . The former is approximately two to three times faster. The computer experiment is run 100 times and the shaded areas represent the region encompassed by the 5th and 95th quantiles. Plot generated with **ggplot2** (Wickham, 2009).*

is simply a trade-off between development effort and impact. This is a tricky question for strategies that are not generalizable, but very effective at solving a specific class of problems. For example, retaping is not helpful when the expression graph changes from point to point and we need to evaluate the gradient at several points; but it is well suited for computing a single Jacobian matrix.

A similar argument applies to specialized math functions. For some problems, such as solving ODEs and algebraic equations, we can take advantage of custom differentiation methods, which are agnostic to which solver gets used. This does not hold in general. Counter-examples arise when we solve partial differential equations or work out analytical solutions to field-specific problems. The coding burden then becomes more severe, and developers must work out which routines are worth implementing.

Extensibility

Making a package *extensible* aims to reduce the burden on the developers and give users the coding flexibility to create new features themselves. Open-source projects greatly facilitate this process. As an example, **Torsten**⁴ extends **Stan Math** for applications in pharmacometrics (Margossian & Gillespie, 2016). In our experience however, extending a library requires a great deal of time, coding expertise, and detailed knowledge of a specific package. The problem is worse when the AD package works as a black box.

One solution is to allow users to specify custom differentiation methods when declaring a function. Naturally users will have to be cautioned against spending excessive time coding derivatives for minor performance gains. PyTorch provides such a feature in the high-level language Python, namely for reverse-mode AD⁵. CasADi (Andersson, Gillis, Horn, Rawlings, & Diehl, 2018) also offers the option to write custom derivative functions in all its interfaces (including Python and Matlab), though this feature is described as “experimental” and still requires a technical understanding of the package and its underlying structure in C++⁶. Another solution, which is well underway, is to educate users as to the inner workings of AD packages. This is in large part achieved by papers on specific packages, review articles such as this one, and well documented code.

Higher-order differentiation

Algorithms that require higher-order differentiation are less common, but there are a few noteworthy examples. Riemannian Hamiltonian Monte Carlo (RHMC) is a Markov chain Monte Carlo sampler that requires the second and third order derivatives of the log posterior distribution, and can overcome severe geometrical pathologies first-order sampling methods cannot (Girolami, Calderhead, & Chin, 2013; Betancourt, 2013). A more classical example is Newton’s method, which requires a Hessian vector product when used for optimization. Much work has been done to make the computation of higher-order derivatives efficient. In general, this task remains significantly more difficult than computing first-order derivatives

⁴The package is still under development. See <https://github.com/metrumresearchgroup/example-models>.

⁵See <https://pytorch.org/docs/stable/notes/extending.html>.

⁶See <http://casadi.sourceforge.net/api/html/>.

and can be prohibitively expensive, which is why we label it as a “practical problem”. Practitioners are indeed often reluctant to deploy computational tools that require higher-order derivatives. One manifestation of this issue is that RHMC is virtually never used, whereas its first-order counterpart has become a popular tool for statistical modeling.

As previously touched upon, most packages compute second-order derivatives by applying AD twice: first to the target function, thereby producing first-order derivatives; then by sorting the operations which produced these first-order derivatives into a new expression graph, and applying AD again. Often times, as when computing a Newton step, we are not interested in the entire Hessian matrix, but rather in a Hessian vector product.

For simplicity, consider the scalar function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. To compute a Hessian vector product, one can first apply a forward sweep followed by a reverse sweep (Pearlmutter, 1994; Griewank & Walther, 2008). For second-order derivatives, AD produces an object of the general form

$$u \nabla f + w \nabla^2 f \cdot v$$

where ∇f designates the gradient vector and $\nabla^2 f$ the Hessian matrix, and where $v \in \mathbb{R}^n$, $u \in \mathbb{R}$, and $w \in \mathbb{R}$ are initialization vectors (possibly of length 1). The Hessian vector product $\nabla^2 f \cdot v$ is then obtained by setting $u = 0$, $w = 1$, and $v = v$, an efficient operation whose complexity is linear in the complexity of f . The full $n \times n$ Hessian matrix is computed by repeating this operation n times and using basis vectors for v . As when computing Jacobian matrices, the same expression graph can be stored in a tape and reused multiple times. It is also possible to exploit the structural properties of the Hessian as is for example done by (Gebremedhin, Tarafdar, Pothen, & Walther, 2009) with the package **Adol-C**.

Consider now the more general case, where we wish to compute higher-order derivatives for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. We can extend the above procedure: apply AD, sort the AD operations into a new expression graph, and repeat until we get the desired differentiation order.

Whether or not the above approach is optimal is an open question, most recently posed by (Betancourt, 2017b, 2018), who argues recursively applying AD leads to inefficient and sometimes numerically unstable code. There exists two alternatives to compute higher-order derivatives. We only briefly mention them here and invite the interested reader to consult the original references. The first alternative uses univariate Taylor series and begins with the observation that the i^{th} coefficient of the series corresponds to the i^{th} derivative of our target function; more is discussed in (Bischof, Corliss, & Griewank, 1993; Griewank, Utke, & Walther, 2000). Particular attention is given to the computation of Hessian matrices and ways to exploit their structure.

For the second alternative, (Betancourt, 2018) proposes a theoretical framework for higher-order differential operators and derives these operators explicitly for the second and third order cases. In some sense, this approach imitates the scheme used for first-order derivatives: create a library of mathematical operations for which higher-order partial derivatives are analytically worked out and compute sensitivities using “higher-order chain rules”. Doing so is a tricky but doable task that requires a good handle on differential geometry. The here described strategy is prototyped in the C++ package **Nomad** (Betancourt, 2017b), which serves as a proof of concept. At this point however **Nomad** is not optimized to compute first-

order derivatives and more generally needs to be further developed in order to be extensively used⁷.

Conclusion

In recent years, AD has been successfully applied to several areas of computational statistics and machine learning. One notable example is Hamiltonian Monte Carlo sampling, in particular its adaptive variant the No-U-Turn sampler (NUTS) (Hoffman & Gelman, 2014). NUTS, supported by AD, indeed provides the algorithmic foundation for the probabilistic languages **Stan** (Carpenter et al., 2017) and **PyMC3** (Salvatier, Wiecki, & Fonnesbeck, 2016). AD also plays a critical part in *automatic differentiation variational inference* (Kucukelbir et al., 2016), a gradient based method to approximate Bayesian posteriors. Another major application of AD is neural networks, as done in **TensorFlow** (Abadi et al., 2016).

We expect that in most cases AD acts as a black box supporting a statistical or a modeling software; nevertheless, a basic understanding of AD can help users optimize their code, notably by exploiting some of the mathematical techniques we discussed. Advanced users can edit the source code of a package, for example to incorporate an expression templates routine or write a custom derivative method for a function.

Sometimes, a user must pick one of the many tools available to perform AD. A first selection criterion may be the programming language in which a tool is available; this constraint is however somewhat relaxed by packages that link higher-level to lower-level languages, such as **Rcpp** which allows coders to use **C++** inside **R** (Eddelbuettel & Balamuta, 2017). Passed this, our recommendation is to first look at the applications which motivated the development of a package and see if these align with the user’s goals. **Stan Math**, for instance, is developed to compute the gradient of a log-posterior distribution, specified by a probabilistic model. To do so, it offers an optimized and flexible implementation of reverse AD, and an extensive library of mathematical functions to do linear algebra, numerical calculations, and compute probability densities. On the other hand, its forward-mode is not as optimized and well tested as its reverse-mode AD⁸. More generally, we can identify computational techniques compatible with an application of interest (see Table 6), and consider packages that implement these techniques. Another key criterion is the library of functions a package provides: using built-in functions saves time and effort, and often guarantees a certain degree of optimality. For more esoteric problems, extensibility may be of the essence. Other considerations, which we have not discussed here, include a package’s capacity to handle more advanced computational features, such as parallelization and GPUs.

Acknowledgments

I thank Michael Betancourt, Bob Carpenter, and Andrew Gelman for helpful comments and discussions.

⁷Personal communication with Michael Betancourt.

⁸Personal communication with Bob Carpenter.

Appendix: computer experiments

This appendix complements the section on *Mathematical implementations* and describes in more details the presented performance tests. The two computer experiments are conducted using the following systems:

- Hardware: Macbook Pro computer (Retina, early 2015) with a 2.7 GHz Intel Core i5 with 8 GB of 1867 MHz DDR3 memory
- Compiler: clang++ version 4.2.1
- Libraries: Eigen 3.3.3, Boost 1.66.0, Stan Math 2.17 - develop⁹

The compiler and libraries specifications are those used when running unit tests in **Stan Math**. We measure runtime by looking at the wall clock before and after evaluating a target function f and calculating all the sensitivities of interest. The wall time is computed using the C++ function `std::chrono::system_clock::now()`. The code is available on GitHub.

Differentiating the 2×2 Matrix Exponential

We obtain 2×2 matrices by generating four elements from a uniform distribution $U(1, 10)$, that is a uniform with lower bound 1 and upper bound 10. This conservative approach insures the term Δ in Equation 11 is real (i.e. has no imaginary part). We generate the random numbers using `std::uniform_real_distribution<T> unif(1, 10)`.

Differentiating a numerical algebraic solver

To test the performance of an algebraic solver, we consider a standard problem in pharmacometrics, namely the computation of steady states for a patient undergoing a medical treatment. We only provide an overview of the scientific problem, and invite the interested reader to consult (Margossian, 2018) or any standard textbook on pharmacokinetics.

In our example, a patient orally receives a drug which diffuses in their body and gets cleared over time. In particular, we consider the *one compartment model with a first-order absorption from the gut*. The following ODE system then describes the drug diffusion process:

$$\begin{aligned}\frac{dy_1}{dt} &= -k_1 y_1 \\ \frac{dy_2}{dt} &= k_1 y_1 - k_2 y_2\end{aligned}$$

where

- y_1 is the drug mass in the gut
- y_2 the drug mass in a central compartment (often times organs and circulatory systems, such as the blood, into which the drug diffuses rapidly)

⁹Versioned branch `test/autodiff_review`.

- k_1 and k_2 represent diffusion rates
- and t is time.

This system can be solved analytically. Given an initial condition y_0 and a time interval δt , we can then define an *evolution operator* $g(y_0, \delta t)$ which evolves the state of a patient, as prescribed by the above ODEs.

Often times, a medical treatment undergoes a cycle: for example, a patient receives a drug dose at a regular time interval, δt . We may then expect that, after several cycles, the patient reaches a state of equilibrium. Let $y_{0-} = \{y_1(0), y_2(0)\}$ be the drug mass at the beginning of a cycle, m the drug mass (instantaneously) introduced in the gut by a drug intake, and $y_{0+} = \{y_1(0) + m, y_2(0)\}$. Then equilibrium is reached when:

$$f(y_{0-}) = g(y_{0+}, \delta t) - y_{0-} = 0$$

This algebraic equation can be solved analytically, but for demonstrating purposes, we solve it numerically. We then compute sensitivities for k_1 and k_2 .

To increase the number of states, we solve the algebraic equations for n patients, yielding $2n$ states. The coefficients in the ODEs for the i^{th} patient are $k_i = \{k_{1i}, k_{2i}\}$. These coefficients vary from patient to patient, according to $k_i = \phi_i k$, where k is a population parameter and ϕ_i a random scaling factor, uniformly sampled between 0.7 and 1.3. Hence, for $2n$ states, we have $2n + 2$ parameters that require sensitivities.

References

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, *abs/1603.04467*. Retrieved from <http://arxiv.org/abs/1603.04467>
- Andresson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., & Diehl, M. (2018). CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*.
- Aubert, P., Di Cesare, N., & Pironneau, O. (2001). Automatic differentiation in c++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*. doi: <https://doi.org/10.1007/s007910000048>
- Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *CoRR*, *abs/1502.05767*. Retrieved from <http://arxiv.org/abs/1502.05767>
- Bell, B. M. (2012). Cppad: a package for c++ algorithmic differentiation. *Computational Infrastructure for Operations Research*.
- Bell, B. M., & Burke, J. V. (2008). Algorithmic differentiation of implicit functions and optimal values. , *64*. doi: https://doi.org/10.1007/978-3-540-68942-3_17
- Betancourt, M. (2013). A general metric for riemannian manifold hamiltonian monte carlo. Retrieved from <https://arxiv.org/pdf/1212.4693.pdf>;
- Betancourt, M. (2017a, January). A conceptual introduction to hamiltonian monte carlo. *arXiv:1701.02434v1*.

- Betancourt, M. (2017b). Nomad: A high-performance automatic differentiation package [Computer software manual]. <https://github.com/stan-dev/nomad>.
- Betancourt, M. (2018). A geometric theory of higher-order automatic differentiation. Retrieved from <https://arxiv.org/pdf/1812.11592.pdf>
- Bischof, C., & Bücker, H. (2000). Computing derivatives of computer programs. In J. Grotendorst (Ed.), *Modern methods and algorithms of quantum chemistry: Proceedings, second edition* (Vol. 3, pp. 315–327). Jülich: NIC-Directors. Retrieved from https://juser.fz-juelich.de/record/44658/files/Band_3.Winterschule.pdf
- Bischof, C., Corliss, G., & Griewank, A. (1993). Structured second-and higher-order derivatives through univariate taylor series. *Optimization Methods and Softwares*. doi: <https://doi.org/10.1080/10556789308805543>
- Bischof, C., Khademi, P., Mauer, A., & Carle, A. (1996). Adifor 2.0: automatic differentiation of fortran 77 programs. *Computational Science Engineering, IEEE*, 3(3), 18-32. doi: 10.1109/99.537089
- Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., ... Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software*. doi: 10.18637/jss.v076.i01
- Carpenter, B., Hoffman, M. D., Brubaker, M. A., Lee, D., Li, P., & Betancourt, M. J. (2015). The stan math library: Reverse-mode automatic differentiation in c++. Retrieved from <https://arxiv.org/abs/1509.07164>
- Eddelbuettel, D., & Balamuta, J. J. (2017, aug). Extending extitR with extitC++: A Brief Introduction to extitRcpp. *PeerJ Preprints*, 5, e3188v1. Retrieved from <https://doi.org/10.7287/peerj.preprints.3188v1> doi: 10.7287/peerj.preprints.3188v1
- Gay, D. (2005). Semiautomatic differentiation for efficient gradient computations. In H. M. Buecker, G. F. Corliss, P. Hovland, U. Naumann, & B. Norris (Eds.), *Automatic differentiation: Applications, theory, and implementations* (Vol. 50, pp. 147–158). Springer, New York.
- Gay, D., & Aiken, A. (2001, May). Language support for regions. *SIGPLAN Not.*, 36(5), 70–80. Retrieved from <http://doi.acm.org/10.1145/381694.378815> doi: 10.1145/381694.378815
- Gebremedhin, A. H., Tarafdar, A., Pothén, A., & Walther, A. (2009). Efficient computation of sparse hessians using coloring and automatic differentiation. *INFORMS Journal on Computing*, 21, 209 - 223. doi: <https://doi.org/10.1287/ijoc.1080.0286>
- Giles, M. (2008). An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation. *The Mathematic Institute, University of Oxford, Eprints Archive*.
- Girolami, M., Calderhead, B., & Chin, S. A. (2013). Riemannian manifold hamiltonian monte carlo. *arXiv:0907.1100*. Retrieved from <https://arxiv.org/abs/0907.1100>
- Griewank, A. (1992). Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1), 35-54. doi: 10.1080/10556789208805505
- Griewank, A., Juedes, D., & Utke, J. (1999). Adol-c: A package for the automatic differentiation of algorithms written in c/c++. *ACM Transactions on Mathematical Software*, 22. Retrieved from <http://www3.math.tu-berlin.de/Vorlesungen/SS06/AlgoDiff/adolc-110.pdf>

- Griewank, A., Utke, J., & Walther, A. (2000). Evaluating higher derivative tensors by forward propagation of univariate taylor series. *Mathematics of computation*, 69(231), 1117 - 1130. doi: 10.1090/S0025-5718-00-01120-0
- Griewank, A., & Walther, A. (2008). Evaluating derivatives: Principles and techniques of algorithmic differentiation. *Society for Industrial and Applied Mathematics (SIAM)*, 2. doi: <https://doi.org/10.1137/1.9780898717761>
- Grimm, J., Pottier, L., & Rostaing-Schmidt, N. (1996). Optimal time and minimum space-time product for reversing a certain class of programs. *INRIA*. doi: inria-00073896
- Guennebaud, G., Jacob, B., et al. (2010). *Eigen v3*. <http://eigen.tuxfamily.org>.
- Hascoet, L., & Pascual, V. (2013). The tapenade automatic differentiation tool: principles, model, and specification. *ACM Transaction on Mathematical Software*. doi: 10.1145/2450153.2450158
- Hoffman, M. D., & Gelman, A. (2014, April). The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*.
- Hogan, R. J. (2014). Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software*, 40(4). doi: 10.1145/2560359
- Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., & Blei, D. M. (2016). Automatic differentiation variational inference. Retrieved from <https://arxiv.org/abs/1603.00788>
- Margossian, C. C. (2018, January). Computing steady states with stan’s nonlinear algebraic solver. In *Stan conference 2018 california*.
- Margossian, C. C., & Gillespie, W. R. (2016, October). Stan functions for pharmacometrics modeling. In *Journal of pharmacokinetics and pharmacodynamics* (Vol. 43).
- Moler, C., & Van Loan, C. (2003, March). Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*.
- Neal, R. M. (2010). Mcmc using hamiltonian dynamics. In *Handbook of markov chain monte carlo*. Chapman & Hall / CRC Press.
- Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., ... Lerer, A. (2017). Automatic differentiation in pytorch.
- Pearlmutter, B. A. (1994). Fast exact multiplication by the hessian. *Neural Computation*. doi: <https://doi.org/10.1162/neco.1994.6.1.147>
- Phipps, E., & Pawlowski, R. (2012). Efficient expression templates for operator overloading-based automatic differentiation. *arXiv:1205.3506*.
- Powell, M. J. D. (1970). A hybrid method for nonlinear equations. In P. Rabinowitz (Ed.), *Numerical methods for nonlinear algebraic equations*. Gordon and Breach.
- Rowland, T., & Weisstein, E. W. (n.d.). *Matrix exponential*. MathWorld. Retrieved from <http://mathworld.wolfram.com/MatrixExponential.html>
- Sagebaum, M., Albring, T., & Gauger, N. R. (2017). High-performance derivative computations using codipack. Retrieved from <https://arxiv.org/pdf/1709.07229.pdf>
- Salvatier, J., Wiecki, T. V., & Fonnesbeck, C. (2016). Probabilistic programming in python using pymc3. *PeerJ Computer Science*. doi: <https://doi.org/10.7717/peerj-cs.55>
- Stan Development Team. (2018). *Bayesian statistics using stan, version 2.18.1*. Retrieved from https://mc-stan.org/docs/2_18/stan-users-guide/index.html
- Veldhuizen, T. (1995). *Expression templates* (Tech. Rep.). C++ Report 7.

- Voßbeck, M., Giering, R., & Kaminski, T. (2008). Development and first applications of tac++. , 64. doi: https://doi.org/10.1007/978-3-540-68942-3_17
- Walther, A., & Griewank, A. (2012). Getting started with adol-c. In U. Naumann & O. Schenk (Eds.), *Combinatorial scientific computing* (pp. 181–202). Chapman-Hall CRC Computational Science.
- Wickham, H. (2009). *ggplot2: Elegant graphics for data analysis*. <http://ggplot2.org>: Springer-Verlag New York.
- Widrow, B., & Lehr, M. A. (1990, Sep). 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9), 1415-1442. doi: 10.1109/5.58323