BOB CARPENTER, EDITOR

# AUTOMATIC DIFFERENTIATION HANDBOOK

# *Preface*

The goal of this book is to introduce the basic algorithms for automatic differentiation along with an encyclopedic collection of automatic differentiation rules for popular mathematical and statistical functions.

## *Overview of this book*

Automatic differentiation is a general technique for converting a function computing values to one that also computes derivatives. Derivative computations add only a constant overhead to each operation used to compute the function value, so that the differentiable function has the same order of complexity as the original function.

After describing the standard forms of automatic differentiation, this book supplies an encyclopedic collection of tangent and adjoint rules for forward-mode and backward-mode automatic differentiation, covering most widely used scalar, vector, matrix, and probability functions.

The appendix contains working example code for forward-mode, reverse-mode, and mixed-mode automatic differentiation.

## *Why derivatives?*

Applications of derivatives, which measure the change in one quantity relative to a change in another quantity, are ubiquitous in applied mathematics. Although derivatives can be computed mechanistically by hand in many cases, the process is painstaking and error prone.

The present text is motivated by the fact that most state-of-the-art statistical inference algorithms are based on first- or higher-order derivatives. Examples include

- maximum likelihood and maximum a posteriori estimation with quasi-Newton or gradient descent methods (first-order derivatives),

- Bayesian posterior sampling with Hamiltonian Monte Carlo sampling (first-order derivatives of log density functions for Euclidean geometry; third-order for Riemannian),
- standard error or posterior covariance estimation with Laplace approximations (second-order derivatives of log density functions),
- maximum marginal likelihood or maximum marginal a posteriori estimation with Monte Carlo expectation maximization (first-order derivatives of expectations computed via Monte Carlo samples)
- variational inference with stochastic gradient descent based on nested expectations (first-order derivatives of expectations computed via Monte Carlo samples).

Automatic differentiation is also useful for computing the sensitivity of one quantity to another, expressed as a derivative. Examples in statistical inference include first- and second-order derivatives to characterize

- data sensitivity—the sensitivity of a maximum likelihood or Bayesian parameter estimate to one or more data points,
- hyperparameter sensitivity—the sensitivity of a maximum likelihood or Bayesian parameter estimate to one or more model hyperparameters, and
- parameter sensitivity—the sensitivity of a posterior log density or a log likelihood to the model parameters for a given data set.

Sensitivities are also of interest in many numerical analysis applications including

- sensitivity of the solution to a system of algebraic or differential equations to initial conditions or parameters.

## *Why automatic differentiation?*

Deriving derivatives analytically by hand is not only painstaking and error prone under the best of circumstances, it is difficult to do efficiently for iterative functions involved in matrix factorization or differential equation solving. As its name implies, automatic differentiation automatically lifts a function computing a value to one computing the value and its derivatives. Perhaps more surprisingly, this can be done in full generality with both efficiency and high arithmetic precision.

## *Overview of automatic differentiation techniques*

Forward-mode automatic differentiation employs the tangent method of propagating the chain rule forward from independent (input)

variables. Forward mode computes derivatives of all outputs of a function $f : \mathbb{R} \to \mathbb{R}^M$ with respect to its input. Forward mode can also be used to efficiently compute directional derivatives, or more generally, gradient-vector products.

Reverse-mode automatic differentiation employs the adjoint method of propagating the chain rule backward from a dependent (output) variable. Reverse-mode computes the gradient of a function, that is the derivatives of the output a function $f : \mathbb{R}^N \to \mathbb{R}$ with respect to each input.

Either forward-mode or reverse-mode automatic differentiation may be used to compute Jacobians, that is the $M \times N$ matrix of all first-order derivatives of a function $f : \mathbb{R}^N \to \mathbb{R}^M$. Jacobians require $N$ passes of forward-mode or $M$ passes of reverse-mode automatic differentiation.

Nesting reverse-mode automatic differentiation within forward mode provides efficient calculation of Hessians, that is the matrix of all second-order derivatives of a function $f : \mathbb{R}^N \to \mathbb{R}$. Hessians require $N$ passes of reverse mode nested in forward mode. Just as forward-mode automatic differentiation may be used to efficiently calculate gradient-vector products, reverse mode nested in forward mode can be used to compute Hessian-vector products efficiently.

# *Derivatives*

## *Smooth functions*

A smooth function is one that is continuously differentiable over its domain. Unless explicitly stated, all functions under consideration are differentiable as many times as necessary.

## *Derivatives*

Consider a smooth, univariate function $f : \mathbb{R} \to \mathbb{R}$. Its derivative function, $f' : \mathbb{R} \to \mathbb{R}$, maps a scalar $x \in \mathbb{R}$ to the derivative of $f(x)$ with respect to $x$, and is defined as a limit

$$f'(x) = \frac{\mathrm{d}}{\mathrm{d}x} f(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon},$$

where the numerator is the change in $y = f(x)$ and the denominator is the change in $x$.

Rewriting the expression for the derivative,

$$\lim_{\epsilon \to 0} \frac{f(x+\epsilon) - (f(x) + \epsilon \cdot f'(x))}{\epsilon} = 0.$$

Thus, as $\epsilon \to 0$,

$$f(x+\epsilon) \approx f(x) + \epsilon \cdot f'(x).$$

## *Partial derivatives*

For a function $f : \mathbb{R}^N \to \mathbb{R}$, the partial derivative with respect to $x_n$ at $x \in \mathbb{R}^N$ is

$$\frac{\partial}{\partial x_n} f(x) = g'(x_n),$$

where $g : \mathbb{R} \to \mathbb{R}$ is the univariate function defined by

$$g(u) = f(x_1, \ldots, x_{n-1}, u, x_{n+1}, \ldots, x_N).$$

That is, we differentiate $f(x)$ with respect to $x_n$, holding all other values $x_1, \ldots, x_{n-1}, x_{n+1}, \ldots, x_N$ constant.

### *Chain rule*

Automatic differentiation is driven by the chain rule, which states that

$$\frac{\partial}{\partial x} f(u) = \frac{\partial}{\partial u} f(u) \cdot \frac{\partial}{\partial x} u = f'(u) \cdot \frac{\partial}{\partial x} u.$$

Including the differentiated function in the numerator makes the relationship clearer,

$$\frac{\partial f(u)}{\partial x} = \frac{\partial f(u)}{\partial u} \cdot \frac{\partial u}{\partial x}.$$

### *Gradients*

Consider a smooth vector function $f : \mathbb{R}^N \to \mathbb{R}$. Its gradient function, $\nabla f : \mathbb{R}^N \to \mathbb{R}^N$, maps a vector $x \in \mathbb{R}^N$ to the vector of partial derivatives of $f(x)$,

$$\nabla f(x) = \left[ \frac{\partial}{\partial x_1} f(x) \quad \cdots \quad \frac{\partial}{\partial x_N} f(x) \right].$$

The notation indicates that the result is a $1 \times N$ row vector.

The notation $\nabla^\top f$ indicates the function which transposes the results of $\nabla f$ to produce column vectors, i.e.,

$$\nabla^\top f(x) = (\nabla f(x))^\top = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_N} f(x) \end{bmatrix}.$$

### *Gradient-vector products*

Given a smooth function $f : \mathbb{R}^N \to \mathbb{R}$, the derivative of $f$ along a vector $v$ at a point $x \in \mathbb{R}^N$ is defined by

$$\nabla_v f(x) = \lim_{\epsilon \to 0} \frac{f(x + \epsilon \cdot v) - f(x)}{\epsilon}.$$

This definition is equivalent to defining the derivative along a vector as a standard gradient times the vector,

$$\nabla_v f(x) = \nabla f(x) \cdot v = \sum_{n=1}^{N} f_n(x) \cdot v_n = \sum_{n=1}^{N} \frac{\partial f(x)}{\partial x_n} \cdot v_n.$$

The vector multiplication is conformal because $\nabla f(x)$ is a row vector by definition, whereas $v$ is a standard column vector.

### *Directional derivatives*

A unit vector $u \in \mathbb{R}^N$, i.e., one where where $u^\top \cdot u = \sum_{n=1}^{N} u_n^2 = 1$, picks out a point on a sphere and hence a direction. The derivative of $f$ along a unit vector $u$ at the point $x$ is called a directional derivative, as it provides the change in $f$ in the direction picked out by $u$.

### *Jacobians*

Consider a smooth multivariate function $f : \mathbb{R}^N \to \mathbb{R}^M$. Its Jacobian function, $\mathrm{J}_f : \mathbb{R}^N \to \left(\mathbb{R}^N \times \mathbb{R}^M\right)$, maps a vector $x \in \mathbb{R}^N$ to the $M \times N$ matrix of partial derivatives of each element of $f(x)$ with respect to each $x_n$,

$$\mathrm{J}_f(x) = \frac{\partial}{\partial x} f(x).$$

Row $m$ of the Jacobian matrix is a gradient for a single output, and the entries make up all of the partial derivatives of $f$,

$$\mathrm{J}_f(x) = \begin{bmatrix} \nabla f_1(x) \\ \vdots \\ \nabla f_M(x) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(x) & \cdots & \frac{\partial}{\partial x_N} f_1(x) \\ \vdots & \vdots & \vdots \\ \frac{\partial}{\partial x_1} f_M(x) & \cdots & \frac{\partial}{\partial x_N} f_M(x) \end{bmatrix},$$

where $f_m(x)$ is defined by selecting the $m$-th element of $f(x)$,[1]

$$f_m(x) = f(x)[m].$$

[1] Notations $v[i]$ and $v_i$ will be used interchangeably for the $i$-th element of vector $v$, and similarly for $m[i, j]$ and $m_{i,j}$ for the elements of matrices.

Elementwise, the entries of the Jacobian are

$$\mathrm{J}_f(x)[m, n] = \frac{\partial}{\partial x_n} f_m(x).$$

### *Hessians*

Consider a smooth multivariate function $f : \mathbb{R}^N \to \mathbb{R}$ of a single output. The Hessian function $\mathrm{H}_f$ maps an element $x \in \mathbb{R}^N$ to its matrix of second derivatives, and is defined by applying the gradient operator twice (with a transposition in between),

$$\mathrm{H}_f(x) = \nabla \nabla^\top f(x) = \nabla \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_N} f(x) \end{bmatrix} = \begin{bmatrix} \nabla \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \nabla \frac{\partial}{\partial x_N} f(x) \end{bmatrix} = \begin{bmatrix} \frac{\partial^2}{\partial x_1 \partial x_1} f(x) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_N} f(x) \\ \vdots & \vdots & \vdots \\ \frac{\partial^2}{\partial x_N \partial x_1} f(x) & \cdots & \frac{\partial^2}{\partial x_N \partial x_N} f(x) \end{bmatrix}.$$

Elementwise, the entries in the Hessian evaluated at $x$ are

$$\mathrm{H}_f(x)[m, n] = \frac{\partial^2}{\partial x_m \partial x_n} f(x) = \frac{\partial}{\partial x_m} \frac{\partial}{\partial x_n} f(x).$$

The Hessian matrix is symmetric, with

$$\mathrm{H}_f(x)[m,n] = \mathrm{H}_f(x)[n,m].$$

The diagonals are the second partial derivatives,

$$\mathrm{H}_f(x)[n,n] = \frac{\partial}{\partial x_n}\frac{\partial}{\partial x_n} f(x) = \frac{\partial^2}{\partial x_n^2} f(x).$$

## *Hessian-vector products*

Suppose $f : \mathbb{R}^N \to \mathbb{R}$ is a smooth function. The product of the Hessian of $f$ at a point $x \in \mathbb{R}^N$ and an arbitrary vector $v \in \mathbb{R}$ can be calculated as the gradient of a vector-gradient product,

$$\mathrm{H}_f(x) \cdot v = \Big(\nabla\nabla^\top f(x)\Big) \cdot v = \nabla \left(\nabla f(x) \cdot v\right)^\top.$$

This can be verified elementwise,

$$\begin{aligned}
\Big(\mathrm{H}_f(x) \cdot v\Big)[m] &= \mathrm{H}_f(x)[m] \cdot v \\
&= \textstyle\sum_{n=1}^N \mathrm{H}_f(x)[m,n] \cdot v[n] \\
&= \textstyle\sum_{n=1}^N \left(\nabla\nabla^\top f(x)\right)[m,n] \cdot v_n \\
&= \textstyle\sum_{n=1}^N \frac{\partial^2}{\partial x_m \partial x_n} f(x) \cdot v_n \\
&= \textstyle\sum_{n=1}^N \frac{\partial}{\partial x_m}\frac{\partial}{\partial x_n} \left(f(x) \cdot v_n\right) \\
&= \textstyle\frac{\partial}{\partial x_m} \sum_{n=1}^N \frac{\partial}{\partial x_n} \left(f(x) \cdot v_n\right) \\
&= = \nabla \left(\nabla f(x) \cdot v\right)^\top [m].
\end{aligned}$$

Because Hessians are symmetric, $v$ can be multiplied on the left or right

$$v^\top \cdot \mathrm{H}_f(x) = \Big(\mathrm{H}_f(x) \cdot v\Big)^\top.$$

## *References*

An excellent reference for both matrix algebra and multivariate differential calculus is (Magnus and Neudecker 2019).

# *Finite Differences*

Derivatives may be computed numerically using finite differences. There are many ways that derivatives can be computed using finite differences, and this chapter does not attempt a complete survey of the field.

## *Derivatives with one difference*

The simplest for of finite differences directly follows the definition of derivatives. Suppose $f : \mathbb{R} \to \mathbb{R}$ is a smooth unary function. Then by definition,

$$f'(x) = \frac{\mathrm{d}}{\mathrm{d}x} f(x) = \lim_{\epsilon \to 0} \frac{f(x) - f(x+\epsilon)}{\epsilon}.$$

By fixing an $\epsilon > 0$, an approximation to the derivative may be calculated using finite differences as

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}.$$

Multiplying both sides by $\epsilon$,

$$\epsilon \cdot f'(x) \approx f(x+\epsilon) - f(x),$$

and then adding $f(x)$ to both sides yields

$$f(x+\epsilon) \approx f(x) + \epsilon \cdot f'(x).$$

This shows that finite differences are only going to be accurate to the extent that $f$ can be well approximated by a linear function in the neighborhood of $x$.

## *Partial derivatives*

For multivariate functions $f : \mathbb{R}^N \to \mathbb{R}$, partial derivatives $\frac{\partial}{\partial x_n} f(x)$ are also calculated following their definition by binding $x_1, \ldots x_{n-1}, x_{n+1}, \ldots x_N$ and applying finite differences to the resulting unary function $f(u) = f(x_1, \ldots, x_{n-1}, u, x_{n+1}, \ldots, x_N)$. This method is applicable no matter what method is used for finite differences of unary functions.

### *Arithmetic precision*

It would seem that better and better approximations would be available as $\epsilon \rightarrow 0$, but that is unfortunately not the case with fixed-precision, floating-point arithmetic. Instead, making $\epsilon$ smaller than machine precision (about $10^{-14}$ in IEEE double-precision arithmetic) leads to rounding of $1 + \epsilon$ to 1 and all precision is lost. More generally, if $\epsilon$ is of order 1, then no precision is lost in $1 + \epsilon$. But if $\epsilon << 1$, arbitrary amounts of precision can be lost in calculating $1 + \epsilon$.. A fixed $\epsilon = 10^{-n}$ loses $n$ digits of precision in calculating $1 + \epsilon$.

A typical choice of $\epsilon$ for performing finite differences using double-precision floating-point arithmetic (type `double` in C++) is $10^{-8}$, in an attempt to avoid catastrophic rounding in the arithmetic while also keeping $\epsilon$ small enough that the finite-difference approximation is accurate. With double-precision arithmetic, this reduces precision from $10^{-15}$ to roughly $10^{-7}$, or roughly the equivalent of single-preicsion arithmetic (type `float` in C++).

### *Efficiency of finite differences*

Evaluating a derivative by simple finite differences requires two function evaluations. Evaluating an $N$-dimensional gradient by finite differences requires $N + 1$ function evaluations, because $f(x)$ may be reused for each $x_n$ with respect to which derivatives are being taken.

# *Forward Mode*

The forward-mode automatic differentiation algorithm, also known as the tangent method, is a means for efficiently computing derivatives of smooth functions $f : \mathbb{R} \to \mathbb{R}^M$ with a single input and multiple outputs.

Suppose $x \in \mathbb{R}$ and that $v = f(u)$. We write a dot over an expression to indicate a derivative with respect to a distinguished variable $x$,

$$\dot{u} = \frac{\partial}{\partial x} u.$$

The term $\dot{u}$ is called the tangent of $u$ with respect to $x$; the $x$ is implicit in the notation, but assumed to be the same $x$ in expressions with multiple dotted expressions.

For example, if $v = -u$, then by the chain rule,

$$\dot{v} = \frac{\partial}{\partial x} v = \frac{\partial}{\partial x} - u = -\frac{\partial}{\partial x} u = -\dot{u}.$$

Similarly, if we have $v = \exp(u)$, then

$$\dot{v} = \frac{\partial}{\partial x} v = \frac{\partial}{\partial x} \exp(u) = \exp(u) \cdot \frac{\partial}{\partial x} u = \exp(u) \cdot \dot{u}.$$

Derivative propagatin in forward mode works the same way for multivariate functions. For example, if $y = u \cdot v$ is a product, the usual derivative rule for products applies,

$$\dot{y} = \frac{\partial}{\partial x} u \cdot v = \left( \frac{\partial}{\partial x} u \right) \cdot v + u \cdot \left( \frac{\partial}{\partial x} v \right) = \dot{u} \cdot v + u \cdot \dot{v}.$$

## *Dual numbers*

Forward-mode automatic differentiation can be formalized using dual numbers consisting of the value of an expression and its derivative, $\langle u, \dot{u} \rangle$. Smooth functions may then be extended to operate on dual numbers. For example, negation is defined for dual numbers by

$$-\langle u, \dot{u} \rangle = \langle -u, -\dot{u} \rangle.$$

For sums,

$$\langle u, \dot{u} \rangle + \langle v, \dot{v} \rangle = \langle u + v, \dot{u} + \dot{v} \rangle,$$

and for differences

$$\langle u, \dot{u} \rangle - \langle v, \dot{v} \rangle = \langle u - v, \dot{u} - \dot{v} \rangle,$$

For products,

$$\langle u, \dot{u} \rangle \cdot \langle v, \dot{v} \rangle = \langle u \cdot v, \dot{u} \cdot v + u \cdot \dot{v} \rangle$$

and for quotients,

$$\langle u, \dot{u} \rangle / \langle v, \dot{v} \rangle = \langle u / v, \dot{u} / v - u / v^2 \cdot \dot{v} \rangle.$$

For exponentiation,

$$\exp\left(\langle u, \dot{u} \rangle\right) = \langle \exp(u), \exp(u) \cdot \dot{u} \rangle,$$

and for the logarithm,

$$\log \langle u, \dot{u} \rangle = \langle \log u, \frac{1}{u} \cdot \dot{u} \rangle.$$

These all translate directly into rules of tangent propagation.

## *Gradient-vector products*

The product of the gradient of a function at a given point and an arbitrary vector can be computed efficiently using forward-mode automatic differentiation. Suppose $f : \mathbb{R}^N \to \mathbb{R}$ is a smooth function and $x \in \mathbb{R}^N$ is a point in its domain. To compute the derivative of $f$ at $x$ along an arbitrary vector $v \in \mathbb{R}^N$, it suffices to compute the gradient-vector product $\nabla_v f(x) = \nabla f(x) \cdot v$. This can be done with forward-mode automatic differentiation by initializing the tangents of the input variable $x$ with the vector being multiplied,

$$\dot{x} = v.$$

Then dual arithmetic is used as usual to compute the result, and the final dual number's tangent is the gradient-vector product.

For example, suppose

$$f(x) = x_1 \cdot x_2 + x_2,$$

$$x = \begin{bmatrix} 12.9 & 127.1 \end{bmatrix}^\top,$$

and

$$v = \begin{bmatrix} 0.3 & -1.2 \end{bmatrix}^\top.$$

The gradient-vector product $\nabla_v f(x) = \nabla f(x) \cdot v$ is derived as

$$\begin{aligned}\langle 12.9, 0.3\rangle \cdot \langle 127.1, -1.2\rangle + \langle 127.1, -1.2\rangle &= \langle 12.9 \cdot 127.1,\ 0.3 \cdot 127.1 + 12.9 \cdot -1.2\rangle + \langle 127.1, -1.2\rangle \\ &= \langle 12.9 \cdot 127.1 + 127.1,\ 0.3 \cdot 127.1 + 12.9 \cdot -1.2 + -1.2\rangle \\ &= \langle 1767, 21\rangle.\end{aligned}$$

Checking the gradient-vector product analytically,

$$\nabla f(x) = \begin{bmatrix} x_2 & x_1 + 1 \end{bmatrix} = \begin{bmatrix} 127.1 & 12.9 + 1 \end{bmatrix},$$

so that

$$\nabla f(x) \cdot v = \begin{bmatrix} 127.1 & 13.9 \end{bmatrix} \cdot \begin{bmatrix} 0.3 & -1.2 \end{bmatrix}^\top = 21.$$

## *Directional derivatives*

A directional derivative measures the change in a multivariate function in a given direction. A direction can be specified as a point on a sphere, which corresponds to a unit vector $v$, i.e., a vector of unit length, where $\sum_{n=1}^{N} v_n^2 = 1$. Given a smooth function $f : \mathbb{R}^N \to \mathbb{R}$, its derivative in the direction of unit vector $v \in \mathbb{R}^N$ at a point $x \in \mathbb{R}^N$ is $\nabla f(x) \cdot v$. That is, a directional derivative is a gradient-vector product where the vector is a unit vector.

Partial derivatives can be defined as vector-gradient products,

$$\frac{\partial}{\partial x_n} f(x) = \nabla f(x) \cdot u_n,$$

by taking $u_n$ to be the unit vector pointing along the $n$-th axis,

$$u_n = \begin{bmatrix} \underbrace{0 \cdots 0}_{n-1 \text{ zeros}} & 1 & \underbrace{0 \cdots 0}_{N-n \text{ zeros}} \end{bmatrix}.$$

# *Reverse Mode*

The reverse-mode automatic differentiation algorithm, also known as the adjoint method, is a means for efficiently computing derivatives of smooth functions $f : \mathbb{R}^N \to \mathbb{R}$ from multiple inputs to a single output (i.e., gradients).

## *Adjoints*

Suppose $y \in \mathbb{R}$ is the final dependent output variable and $v$ is a subexpression used in the calculation of $y$. The adjoint of $v$ is the derivative of $y$ with respect to $v$, and is written with a bar,

$$\overline{v} = \frac{\partial}{\partial v} y.$$

Suppose that $v = f(u)$. The chain rule tells us that

$$\overline{u} = \frac{\partial y}{\partial u} = \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial u} = \frac{\partial y}{\partial v} \cdot \frac{\partial f(u)}{\partial u} = \frac{\partial y}{\partial v} \cdot f'(u) \cdot \frac{\partial u}{\partial u} = \overline{v} \cdot f'(u).$$

Because a sub-expression may be involved in more than one expression, the adjoint rules are expressed as increments,

$$\overline{u} \ += \ \overline{v} \cdot f'(u).$$

For example, if $v = \exp(u)$, with $\exp'(u) = \exp(u)$, the rule is

$$\overline{u} \ += \ \overline{v} \cdot \exp(u).$$

For logarithms, with $v = \log u$, with $\log'(u) = \frac{1}{u}$, the adjoint rule is

$$\overline{u} \ += \ \overline{v} \cdot \frac{1}{u}.$$

When there is more than one argument, adjoints propagate independently from the result by multiplying the result's adjoint times the partial with respect to the argument. For example, if $w = u \cdot v$, then because $\frac{\partial}{\partial u} u \cdot v = v$ and $\frac{\partial}{\partial v} u \cdot v = u$, the adjoint rules are

$$\overline{u} \ += \ \overline{w} \cdot v$$

and

$$\overline{v} \ += \ \overline{w} \cdot u.$$

## *Adjoint propagation and continuations*

Adjoint rules increment the adjoints of operands based on the adjoint of the result, and as such, must be executed after the adjoint of the result has been computed. This leads to adjoints being propagated in the reverse order of function computation, hence the name of the method.[2]

[2] Any topological sort of the expressions where results are greater than operands will suffice.

Computationally, as each operation executes, the code to handle adjoint propagation is pushed onto a stack. After the final value is computed for which derivatives are required, the adjoints are executed by popping adjoint code off the stack and executing it. The adjoint code amounts to a continuation, that is code that is stored with some of its context to be executed later. As usual, continuations can be implemented via closures and the closures stored on a stack to be executed in last-in/first-out order.

For example, consider a simple compound expression $y = \log(u \cdot v)$. To compute the value of $y$, First, $u \cdot v$ is executed, then the logarithm of the product is computed. This can be expressed with intermediate variables as a function of independent variables $u$ and $v$ as follows, where the numbers in parentheses indicate the order of steps performed.

| execution order ↓ | forward values | reverse adjoints | execution order ↑ |
|---|---|---|---|
| (1) | $a = u \cdot v$ | $\overline{u} \mathrel{+}= \overline{a} \cdot v$ | (6) |
| | | $\overline{v} \mathrel{+}= \overline{a} \cdot u$ | (5) |
| (2) | $b = \log a$ | $\overline{a} \mathrel{+}= \overline{b} \cdot \frac{1}{a}$ | (4) |
| | | $\overline{b} = 1$ | (3) |

First, the values are computed in a forward pass (steps 1 and 2). Then the adjoint of the final result is set to one (step 3) and all other adjoints are initialized to zero. Then the adjoints are propagated in a reverse pass (steps 4, 5, and 6). These steps are executed with concrete values. For example, taking $u = 1.2$ and $v = 3.9$, the execution order is as follows, with values rounded to two decimal places,

| step | variable | op | value | symbolic | numeric |
|---|---|---|---|---|---|
| | $u$ | $=$ | 1.2 | $u$ | 1.20 |
| | $v$ | $=$ | 3.9 | $v$ | 3.90 |
| (1) | $a$ | $=$ | $u \cdot v$ | $u \cdot v$ | 4.68 |
| (2) | $b$ | $=$ | $\log a$ | $\log v \cdot u$ | $\approx 1.60$ |
| (3) | $\overline{b}$ | $=$ | 1 | 1 | 1.00 |
| (4) | $\overline{a}$ | $+=$ | $\overline{b} \cdot \frac{1}{a}$ | $\frac{1}{u \cdot v}$ | $\approx 0.21$ |
| (5) | $\overline{v}$ | $+=$ | $\overline{a} \cdot u$ | $\frac{1}{v}$ | $\approx 0.26$ |
| (6) | $\overline{u}$ | $+=$ | $\overline{a} \cdot v$ | $\frac{1}{u}$ | $\approx 0.83$ |

Before the algorithm begins, the independent variables $u$ and $v$ are assumed to be set to their values. Then steps (1) and (2) calculate the values of subexpressions, first of $a = u \cdot v$ and then of $b = \log u \cdot v$, the final result. To start the reverse pass in step (3), the final result $b$ has its adjoint $\overline{b}$ set to 1. The reverse pass then increments the adjoint of each operand involved in the following expression. The final adjoints $\overline{u}$ and $\overline{v}$ are the elements of the gradient of $f(u, v) = \log u \cdot v$, evaluated at $\begin{bmatrix} 1.2 & 3.9 \end{bmatrix}$,

$$\nabla f(1.2, 3.9) = \begin{bmatrix} \frac{\partial}{\partial u} \log u \cdot v & \frac{\partial}{\partial v} \log u \cdot v \end{bmatrix} \approx \begin{bmatrix} 0.83 & 0.26 \end{bmatrix}.$$

The results of automatic differentiation can be verified with analytical derivatives,

$$\overline{u} = \frac{\partial}{\partial u} \log u \cdot v = \frac{1}{u} \approx 0.83$$

and

$$\overline{v} = \frac{\partial}{\partial v} \log u \cdot v = \frac{1}{v} \approx 0.26.$$

# *Nested Forward Mode*

Forward-mode automatic differentiation is defined with respect to an arbitrary scalar type. With a real scalar type, forward mode computes derivatives. By taking the scalar inside forward mode to itself be an automatic differentiation variable, higher-order derivatives may be calculated, even though only first-order derivatives need be defined.

## *Forward nested in forward*

Consider a dual number $\langle u, \dot{u} \rangle$, where $u$ is a scalar variable and $\dot{u} = \frac{\partial u}{\partial x}$ for some distinguished independent variable $x$. Suppose that instead of a simple scalar, $u = \langle v, \acute{v} \rangle$ is itself a forward-mode autodiff variable defining derivatives with respect to $w$, so that $\acute{v} = \frac{\partial v}{\partial w}$. A different diacritic is used above the nested variable to distinguish it from the outer diacritic dual number. A nested forward-mode autodiff variable is thus of the form $\big\langle \langle v, \acute{v} \rangle, \, \langle \dot{v}, \dot{\acute{v}} \rangle \big\rangle$, where

$$\acute{v} = \frac{\partial v}{\partial w} \qquad \dot{v} = \frac{\partial v}{\partial x} \qquad \dot{\acute{v}} = \frac{\partial^2 v}{\partial x \partial w}.$$

The tangent values of nested forward-mode autodiff variables are initialized following the derivatives. For the input variable $w$, the initial nested autodiff variable is

$$\begin{aligned} \big\langle \langle w, \acute{w} \rangle, \, \langle \dot{w}, \dot{\acute{w}} \rangle \big\rangle &= \big\langle \langle w, \, \tfrac{\partial w}{\partial w} \rangle, \, \langle \tfrac{\partial w}{\partial x}, \, \tfrac{\partial^2 w}{\partial x \partial w} \rangle \big\rangle \\ &= \big\langle \langle w, 1 \rangle, \, \langle 0, 0 \rangle \big\rangle. \end{aligned}$$

Input variable $x$ corresponds to an autodiff variable

$$\begin{aligned} \big\langle \langle x, \acute{x} \rangle, \, \langle \dot{x}, \dot{\acute{x}} \rangle \big\rangle &= \big\langle \langle x, \, \tfrac{\partial x}{\partial w} \rangle, \, \langle \tfrac{\partial x}{\partial x}, \, \tfrac{\partial^2 x}{\partial x \partial w} \rangle \big\rangle \\ &= \big\langle \langle x, 0 \rangle, \, \langle 1, 0 \rangle \big\rangle. \end{aligned}$$

Any other input variable $u$ such that $u \neq x$ and $u \neq y$ will be initialized with nested autodiff variable

$$\begin{aligned} \big\langle \langle u, \acute{u} \rangle, \, \langle \dot{u}, \dot{\acute{u}} \rangle \big\rangle &= \big\langle \langle u, \, \tfrac{\partial u}{\partial w} \rangle, \, \langle \tfrac{\partial u}{\partial x}, \, \tfrac{\partial^2 u}{\partial x \partial w} \rangle \big\rangle \\ &= \big\langle \langle x, 0 \rangle, \, \langle 0, 0 \rangle \big\rangle. \end{aligned}$$

Recall the definition of exponentiation and multiplication for forward-mode autodiff variables,

$$\exp(\langle u, \dot{u} \rangle) = \langle \exp(u), \exp(u) \cdot \dot{u} \rangle,$$

and

$$\langle u, \dot{u} \rangle \cdot \langle z, \dot{z} \rangle = \langle u \cdot z, \dot{u} \cdot z + u \cdot \dot{z} \rangle.$$

Plugging $\langle v, \acute{v} \rangle$ in for $u$ in the forward-mode rule for exponentiation,

$$\begin{aligned}
\exp\big(\langle \langle v, \acute{v} \rangle,\ \langle \dot{v}, \dot{\acute{v}} \rangle \rangle\big) &= \big\langle \exp(\langle v, \acute{v} \rangle),\ \exp(\langle v, \acute{v} \rangle) \cdot \langle \dot{v},\ \dot{\acute{v}} \rangle \big\rangle \\
&= \big\langle \langle \exp(v),\ \exp(v) \cdot \acute{v} \rangle,\ \ \langle \exp(v),\ \exp(v) \cdot \acute{v} \rangle \cdot \langle \dot{v},\ \dot{\acute{v}} \rangle \big\rangle \\
&= \big\langle \langle \exp(v),\ \exp(v) \cdot \acute{v} \rangle,\ \ \langle \exp(v) \cdot \dot{v},\ \exp(v) \cdot \acute{v} \cdot \dot{v} + \exp(v) \cdot \dot{\acute{v}} \rangle \big\rangle
\end{aligned}$$

The value is correct,

$$\exp(v) = \exp(v).$$

The first nested derivative is also correct,

$$\frac{\partial \exp(v)}{\partial w} = \exp(v) \cdot \frac{\partial v}{\partial w} = \exp(v) \cdot \acute{v},$$

as is the other nested derivative,

$$\frac{\partial \exp(v)}{\partial x} = \exp(v) \cdot \frac{\partial v}{\partial x} = \exp(v) \cdot \dot{v}.$$

The second derivative also checks out,

$$\begin{aligned}
\tfrac{\partial^2 \exp(v)}{\partial x \partial w} &= \tfrac{\partial}{\partial x} \tfrac{\partial \exp(v)}{\partial w} \\
&= \tfrac{\partial}{\partial x} \left( \exp(v) \cdot \tfrac{\partial v}{\partial w} \right) \\
&= \tfrac{\partial}{\partial x} (\exp(v)) \cdot \tfrac{\partial v}{\partial w} + \exp(v) \cdot \tfrac{\partial}{\partial x} \tfrac{\partial v}{\partial w} \\
&= \exp(v) \cdot \tfrac{\partial v}{\partial x} \cdot \tfrac{\partial v}{\partial w} + \exp(v) \cdot \tfrac{\partial}{\partial x} \tfrac{\partial v}{\partial w} \\
&= \exp(v) \cdot \dot{v} \cdot \acute{v} + \exp(v) \cdot \dot{\acute{v}}.
\end{aligned}$$

Working through a complete example, suppose

$$f(a, b, c) = a \cdot \exp(b \cdot c).$$

To evaluate $\frac{\partial^2}{\partial a \partial b} f(2.1, 1.5, -0.3)$, the dual numbers will be of the form

$$\langle \langle u, \acute{u} \rangle,\ \langle \dot{u}, \dot{\acute{u}} \rangle \rangle = \langle \langle u, \frac{\partial u}{\partial a} \rangle,\ \langle \frac{\partial u}{\partial b}, \frac{\partial^2 u}{\partial a \partial b} \rangle \rangle.$$

Thus the variables are initialized with appropriate derivatives,

$$\begin{aligned}
a &\Rightarrow \langle \langle 2.1, 1 \rangle,\ \langle 0, 0 \rangle \rangle \\
b &\Rightarrow \langle \langle 1.5, 0 \rangle,\ \langle 1, 0 \rangle \rangle \\
c &\Rightarrow \langle \langle -0.3, 0 \rangle,\ \langle 0, 0 \rangle \rangle.
\end{aligned}$$

Then evaluation is carried out from most nested outward, with

$$\begin{aligned} b \cdot c &\Rightarrow \langle\langle -0.45, 0\rangle, \langle -0.3, 0\rangle\rangle \\ \exp(b \cdot c) &\Rightarrow \langle\langle 0.64, 0\rangle, \langle -0.19, 0\rangle\rangle \\ a \cdot \exp(b \cdot c) &\Rightarrow \langle\langle 1.3, 0.64\rangle, \langle -0.40, -0.19\rangle\rangle \end{aligned}$$

Reading the results out of the nested dual number, the value is 1.3, the derivative with respect to $a$ is 0.64, and the derivative with respect to $b$ is -0.4, and the second derivative with respect to $a$ and $b$ is -0.19. The analytic second derivative is

$$\frac{\partial}{\partial a \, \partial b} a \cdot \exp(b \cdot c) = c \cdot \exp(b \cdot c).$$

Plugging in $a = 2.1, b = 1.5, c = -0.3$ yields $-0.19$, matching the results obtained through automatic differentiation.

The simplicity of the nested method derives from never having to define anything other than first derivatives. Computing second-order derivatives of the exponential only required the rules for first-order derivatives of exponentiation and first-order derivatives of products.

## *Hessians with forward mode*

The Hessian matrix of all second derivatives for a function $f : \mathbb{R}^N \to \mathbb{R}$ may be computed by running forward-mode autodiff $\binom{N}{2} + N$ times, once for each unique pair $m, n$ and one for second derivatives with respect to a single variable. Assuming evaluation of $f(x)$ for $x \in \mathbb{R}^N$ is $\mathcal{O}(g(N))$, forward-mode autodiff can be used to compute Hessians in $\mathcal{O}(N^2 \cdot g(N))$ time and $\mathcal{O}(N^2)$ space.

## *Third-order derivatives and beyond*

The same logic applies for further nesting of forward mode within forward mode within forward mode. The result will have eight nested values (four for the value and four for the tangent). From most to least nested, these values represent third-order derivatives $\frac{\partial^3}{\partial v \partial w \partial x}$, the three pairs of second-order derivatives, the three first-order derivatives, and the value. Third derivatives are simple to compute this way, but provide quite the bookkeepoing obstacle to manual computation, as should be clear from the explicit evaluation of second-order derivatives for the exponential function using nested forward mode.

The nesting may be iterated to compute fourth-order derivatives, etc.

### *Reverse nested in forward*

A more efficient way to compute Hessians is to nest reverse-mode autodiff in forward-mode autodiff. Thus rather than scalars in $\langle u, \dot{u} \rangle$, the $u$ are taken to be reverse-mode autodiff variables. The forward pass records the subexpressions used in the evaluation of $f(\langle u, \dot{u} \rangle)$ as before. The reverse pass is then run from the final result $\dot{u}$, to compute the adjoint $\overline{\dot{u}}$. This is possible because all the calculations used to produce $\dot{u}$ are scalar operations.

Suppose $f : \mathbb{R}^N \to \mathbb{R}$. Let $\dot{u} = \frac{\partial u}{\partial x_n}$ for some $n \in 1{:}N$. If $y = f(x)$, then $\dot{y} = \frac{\partial y}{\partial x_n}$, and reverse-mode autodiff started from $\dot{y}$ will compute

$$\overline{\dot{y}} = \nabla \frac{\partial f(x)}{\partial x_n} = \left[ \frac{\partial}{\partial x_1} \frac{\partial f(x)}{\partial x_n} \ \cdots \ \frac{\partial}{\partial x_N} \frac{\partial f(x)}{\partial x_n} \right] = \left[ \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \ \cdots \ \frac{\partial^2 f(x)}{\partial x_N \partial x_n} \right].$$

### *Computing Hessians with reverse nested in forward*

Applying the adjoint method to a forward-mode derivative results in a row of the Hessian matrix. Thus to compute a Hessian using reverse mode nested in forward-mode autodiff requires $N$ function evaluations, one for each row $\nabla \frac{\partial}{\partial x_m} f(x)$ of the Hessian. Assuming each function evaluation is $\mathcal{O}(g(n))$ leads to Hessian complexity of $\mathcal{O}(N \cdot g(N))$ time and $\mathcal{O}(N^2)$ space. For example, if the time to compute $f(x)$ for $x \in \mathbb{R}^N$ is linear, i.e., $\mathcal{O}(N)$, then the Hessian $\nabla \nabla^\top f(x)$ can be computed in $\mathcal{O}(N^2)$.

### *Higher-order nesting of reverse in forward mode*

Applying the adjoint method to forward mode nested in forward mode leads to third derivatives. Suppose $f : \mathbb{R}^N \to \mathbb{R}$ and $x \in \mathbb{R}^N$. Nested forward mode computes $\acute{y} = \frac{\partial^2}{\partial x_m \partial x_n} f(x)$. Applying the adjoint method to this result produces third derivatives,

$$\overline{\acute{u}} = \nabla \frac{\partial^2}{\partial x_m \partial x_n} f(x) = \left[ \frac{\partial^3}{\partial x_1 \partial x_m \partial x_n} f(x) \quad \cdots \quad \frac{\partial^3}{\partial x_N \partial x_m \partial x_n} f(x) \right].$$

Computing all third-order derivatives of $f$ at $x$ requires $\mathcal{O}(N^2)$ evaluations of $f(x)$, each using reverse mode nested in forward mode further nested in forward mode. Thus if evaluating $f$ is $\mathcal{O}(g(N))$ then evaluating all third order derivatives requires $\mathcal{O}(g(N) \cdot N^2)$ time.

# *Matrix Derivatives*

If $C = f(A, B)$ where $A$, $B$, and $C$ are matrices (or vectors or scalars), the chain rule remains the same,

$$\mathrm{d}C = \frac{\partial f(A,B)}{\partial A} \cdot \mathrm{d}A + \frac{\partial f(A,B)}{\partial B} \cdot \mathrm{d}B.$$

The total differential notation $\mathrm{d}C$ may be understood by plugging in a scalar $x$ with respect which to differentiate,

$$\frac{\mathrm{d}C}{\mathrm{d}x} = \frac{\partial f(A,B)}{\partial A} \cdot \frac{\mathrm{d}A}{\mathrm{d}x} + \frac{\partial f(A,B)}{\partial B} \cdot \frac{\mathrm{d}B}{\mathrm{d}x}.$$

In the general case, if $C = f(A_1, \ldots, A_N)$, then

$$\mathrm{d}C = \sum_{n=1}^{N} \frac{\partial f(A_1, \ldots, A_N)}{\partial A_n} \mathrm{d}A_n.$$

If $C$ is an $K \times L$ matrix and $A$ is a $M \times N$ matrix, then the Jacobian $\frac{\partial C}{\partial A}$ has dimensions $(K \times L) \times (M \times N)$. These results may be construed as involving standard vectors and Jacobians by collapsing all matrices to vectors. Or they may be read directly by raising the type of operations and multiplying the $(K \times L) \times (M \times N)$ Jacobian by the $M \times N$ matrix differential.[3]

[3] In the automatic differentiation literature in computer science, this is sometimes called a "tensor" product, where "tensor" just means array of arbitrary dimensionality and should not be confused with the tensor calculus used in physics.

## *Forward mode*

The definitions of values and tangents remain the same when moving to vector or matrix functions. As with scalars, the tangent of a matrix (or vector) $U$ is defined relative to a scalar variable $x$ as

$$\dot{U} = \frac{\partial U}{\partial x}.$$

This derivative is defined elementwise, with

$$\dot{U}_{i,j} = \frac{\partial U_{i,j}}{\partial x}.$$

Forward mode automatic differentiation for matrices follows the chain rule,

$$\frac{\partial C}{\partial x} = \frac{\partial C}{\partial A} \cdot \frac{\partial A}{\partial x} + \frac{\partial C}{\partial B} \cdot \frac{\partial B}{\partial x},$$

which, using tangent notaton, yields

$$\dot{C} = \frac{\partial C}{\partial A} \cdot \dot{A} + \frac{\partial C}{\partial B} \cdot \dot{B}.$$

In general, if $C = f(A_1, \ldots, A_N)$, then

$$\dot{C} = \sum_{n=1}^{N} \frac{\partial C}{\partial A_n} \cdot \dot{A}_n.$$

As with forward-mode autodiff for scalars, this matrix derivative rule is straightforward to work out and to implement.

The tangent rules for matrix operations carry over neatly from the scalar case. For example, $C = A + B$ is the sum of two matrices, the corresponding tangent rule is

$$\dot{C} = \dot{A} + \dot{B}.$$

Here and throughout, matrices used in arithmetic operations will be assumed to conform to the required shape and size constraints in the expressions in which they are used. For $A + B$ to be well formed, $A$ and $B$ must both be $M \times N$ matrices (i.e., they must have the same number of rows and columns).

Similarly, if $C = A \cdot B$ is the product of two matrices, the tangent rule is the same as that for scalars,

$$\dot{C} = \dot{A} \cdot B + A \cdot \dot{B}.$$

Simple tangent rules exist for many linear algebra operations, such as inverse. If $C = A^{-1}$, then the tangent rule is

$$\dot{C} = -C \cdot \dot{A} \cdot C.$$

Results such as these are derived through algebraic manipulation and differentiation (see [4] for general rules). For inverse, because [4]

$$C \cdot A = A^{-1} \cdot A = \mathrm{I}.$$

Differentiating both sides yields

$$\frac{\partial}{\partial x} C \cdot A = \frac{\partial}{\partial x} \mathrm{I}.$$

Replacing with dot notation yields

$$\dot{C} \cdot A + C \cdot \dot{A} = 0.$$

Rearranging the terms produces

$$\dot{C} \cdot A = -C \cdot \dot{A}.$$

Multiplying both sides of the equation on the right by $A^{-1}$ gives

$$\dot{C} \cdot A \cdot A^{-1} = -C \cdot \dot{A} \cdot A^{-1}.$$

This reduces to the final simplified form

$$\dot{C} = -C \cdot \dot{A} \cdot C,$$

after dropping the factor $A \cdot A^{-1} = \mathrm{I}$ and replacing $A^{-1}$ with its value $C$.

### *Reverse mode*

Using the same adjoint notation as for scalars, if $U$ is an $M \times N$ matrix involved in the computation of a final result $y$, then

$$\overline{U} = \frac{\partial y}{\partial U},$$

with entries defined elementwise by

$$\overline{U}_{i,j} = \frac{\partial y}{\partial U}[i,j] = \frac{\partial y}{\partial U_{i,j}}.$$

The definition applies to vectors if $N = 1$ and row vectors if $M = 1$.

The adjoint method can be applied to matrix or vector functions in the same way as to scalar functions. Suppose there is a final scalar result variable $y$ and along the way to computing $y$, the matrix (or vector) $A$ is used exactly once, appearing only in the subexpression $B = f(\ldots, A, \ldots)$. By the chain rule,[5]

$$\frac{\partial y}{\partial A} = \left(\frac{\partial B}{\partial A}\right)^{\top} \cdot \frac{\partial y}{\partial B} = \mathrm{J}_f^{\top}(A) \cdot \frac{\partial y}{\partial B},$$

where the Jacobian function $\mathrm{J}_f$ is generalized to matrices by[6]

$$\mathrm{J}_f(U) = \frac{\partial f(U)}{\partial U}.$$

Rewriting using adjoint notation,

$$\overline{A} = \mathrm{J}_f^{\top}(A) \cdot \overline{B},$$

or in transposed form,

$$\overline{A}^{\top} = \overline{B}^{\top} \cdot \mathrm{J}_f(A).$$

The adjoint of an operand is the product of the Jacobian of the function and adjoint of the result.

An expression $A$ may be used as an operand in multiple expressions involved in the computation of $y$. As with scalars, the adjoints need to be propagated from each result, leading to the fundamental matrix autodiff rule for a subexpression $B = f(\ldots, A, \ldots)$ involved in the computation of $y$,

$$\overline{A} \mathrel{+}= \mathrm{J}_f^{\top}(A) \cdot \overline{B}.$$

[5] The terms in this equality can be read as vector derivatives by flattening the matrices. If $A$ is an $M \times N$ matrix and $B$ is a $K \times L$ matrix, then $\frac{\partial y}{\partial A}$ is a vector of size $M \cdot N$, $\frac{\partial B}{\partial A}$ is matrix of size $(K \cdot L) \times (M \cdot N)$, and $\frac{\partial y}{\partial B}$ is a vector if size $K \cdot L$. After transposition, the right-hand side is a product of an $(M \cdot N) \times (K \cdot L)$ matrix and a vector of size $K \cdot L$, yielding a vector of size $M \cdot N$, as found on the left-hand side.

[6] Matrix Jacobians may be understood by generalizing definitions to matrices or by flattening.

## *Trace algebra*

The Jacobian of a function with an $N \times M$ matrix operand and $K \times L$ matrix result has $N \cdot M \cdot K \cdot L$ elements, one for each derivative of an output with respect to an input. This makes it prohibitively expensive in terms of both memory and computation to store and multiply Jacobians explicitly. Instead, algebra is used to reduce adjoint computations to manageable sizes.

Suppose the $M \times N$ matrix $C$ is used in the computation of $y$. By the chain rule,

$$\begin{aligned}
\mathrm{d}y &= \sum_{n=1}^{N} \sum_{m=1}^{N} \frac{\partial y}{\partial C_{m,n}} \cdot \mathrm{d}C_{m,n} \\
&= \sum_{n=1}^{N} \sum_{m=1}^{N} \overline{C}_{m,n} \cdot \mathrm{d}C_{m,n} \\
&= \sum_{n=1}^{N} \sum_{m=1}^{N} \overline{C}_{n,m}^{\top} \cdot \mathrm{d}C_{m,n} \\
&= \sum_{n=1}^{N} \overline{C}_{n,\cdot}^{\top} \cdot \mathrm{d}C_{\cdot,n} \\
&= \mathrm{Tr}(\overline{C}^{\top} \cdot \mathrm{d}C),
\end{aligned}$$

where the notation $U_{m,\cdot}$ indicates the $m$-th row of $U$, $U_{\cdot,m}$ the $m$-th column of $U$, and $\mathrm{Tr}(U)$ the trace of an $N \times N$ square matrix $U$ defined by

$$\mathrm{Tr}(U) = \sum_{n=1}^{N} U_{n,n}.$$

Clarifying that differentiation is with respect to a distinguished scalar variable $x$ on both sides of the above equation yields

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x} = \mathrm{Tr}\left(\overline{C}^{\top} \cdot \frac{\mathrm{d}C}{\mathrm{d}x}\right).$$

Suppose that $C = f(A, B)$ is a matrix function. As noted at the beginning of this chapter, the chain rule yields

$$\mathrm{d}C = \frac{\partial f(A,B)}{\partial A}\mathrm{d}A + \frac{\partial f(A,B)}{\partial B}\mathrm{d}B.$$

Using the result of the previous section and substituting the right-hand side above for $\mathrm{d}C$,

$$\begin{aligned}
\mathrm{d}y &= \mathrm{Tr}(\overline{C}^{\top} \cdot \mathrm{d}C) \\
&= \mathrm{Tr}\left(\overline{C}^{\top} \cdot \left(\tfrac{\partial f(A,B)}{\partial A}\mathrm{d}A + \tfrac{\partial f(A,B)}{\partial B}\mathrm{d}B\right)\right) \\
&= \mathrm{Tr}\left(\overline{C}^{\top} \cdot \tfrac{\partial f(A,B)}{\partial A}\mathrm{d}A + \overline{C}^{\top} \cdot \tfrac{\partial f(A,B)}{\partial B}\mathrm{d}B\right) \\
&= \mathrm{Tr}\left(\overline{C}^{\top} \cdot \tfrac{\partial f(A,B)}{\partial A}\mathrm{d}A\right) + \mathrm{Tr}\left(\overline{C}^{\top} \cdot \tfrac{\partial f(A,B)}{\partial B}\mathrm{d}B\right).
\end{aligned}$$

Recall the transposed form of the adjoint rule,

$$\overline{A}^{\top} = \overline{C}^{\top} \cdot \frac{\partial C}{\partial A},$$

and similarly for $\overline{B}^\top$. Plugging that into the final line of the previous derivation yields the final form of the trace rule for matrix functions $C = f(A, B)$,

$$\mathrm{Tr}(\overline{C}^\top \cdot \mathrm{d}C) = \mathrm{Tr}(\overline{A}^\top \cdot \mathrm{d}A) + \mathrm{Tr}(\overline{B}^\top \cdot \mathrm{d}B).$$

This can be generalized to functions of one or more arguments in the obvious way.

## *Examples*

For example, if $C = A + B$, then

$$\frac{\partial C}{\partial A} = 1 \qquad \frac{\partial C}{\partial B} = 1,$$

and hence the adjoint rules are

$$\overline{A} \mathrel{+}= \overline{C}$$

and

$$\overline{B} \mathrel{+}= \overline{C}.$$

In the more interesting case of multiplication, with $C = A \cdot B$,

$$\frac{\partial C}{\partial A} = B,$$

and

$$\frac{\partial C}{\partial B} = A,$$

leading to adjoint rules

$$\overline{A} \mathrel{+}= \overline{C} \cdot B^\top$$

and

$$\overline{B} \mathrel{+}= A^\top \cdot \overline{C}.$$

## *References*

The section on trace algebra fills in the steps of the derivation presented in (Giles 2008b, 2008a).

# *Arithmetic Functions*

*Addition*

$$c = a + b$$

*Derivatives*

$$\frac{\partial}{\partial a} c = 1 \qquad \frac{\partial}{\partial b} c = 1$$

*Tangent*

$$\dot{c} = \dot{a} + \dot{b}$$

*Adjoints*

$$\bar{a} \mathrel{+}= \bar{c} \qquad \bar{b} \mathrel{+}= \bar{c}$$

*Subtraction*

$$c = a - b$$

*Derivatives*

$$\frac{\partial}{\partial a} c = 1 \qquad \frac{\partial}{\partial b} c = -1$$

*Tangent*

$$\dot{c} = \dot{a} - \dot{b}$$

*Adjoints*

$$\bar{a} \mathrel{+}= \bar{c} \qquad \bar{b} \mathrel{+}= -\bar{c}$$

# *Matrix Arithmetic Functions*

*Addition*

$$C = A + B$$

*Derivatives*

$$\frac{\partial}{\partial A} C = 1 \qquad \frac{\partial}{\partial B} C = 1$$

*Tangent*

$$\dot{C} = \dot{A} + \dot{B}$$

*Adjoints*

$$\overline{A} \ += \ \overline{C} \qquad \overline{B} \ += \ \overline{C}$$

*Subtraction*

$$C = A - B$$

*Derivatives*

$$\frac{\partial}{\partial A} C = 1 \qquad \frac{\partial}{\partial B} C = -1$$

*Tangent*

$$\dot{C} = \dot{A} - \dot{B}$$

*Adjoints*

$$\overline{A} \ += \ \overline{C} \qquad \overline{B} \ += \ -\overline{C}$$

*Multiplication*

$$C = A \cdot B$$

*Derivatives*

$$\frac{\partial}{\partial A} C = B \qquad \frac{\partial}{\partial B} C = A$$

*Tangent*

$$\dot{C} = \dot{A} \cdot B + A \cdot \dot{B}$$

*Adjoints*

$$\overline{A} \mathrel{+}= \overline{C} \cdot B^{\top} \qquad \overline{B} \mathrel{+}= A^{\top} \cdot \overline{C}$$

## *Negation*

$$C = -A$$

*Derivatives*

$$\frac{\partial}{\partial A} C = -1$$

*Tangent*

$$\dot{C} = -\dot{A}$$

*Adjoint*

$$\overline{A} \mathrel{+}= -\overline{C}$$

# *Hidden Markov Models*

A hidden Markov model (HMM) defines a density function for a sequence of observations $y_1, \ldots, y_N$, where

- the observations are conditionally independent draws from a mixture distribution with $K$ components, and
- the unobserved mixture components $z_1, \ldots, z_N \in 1 : K$ form a Markov process.

The Markov process for the mixture coponents is governed by

- an initial probability simplex $\phi \in \mathbb{R}^K$,
- stochastic matrix $\Theta \in \mathbb{R}^{K \times K}$, and

with

$$p(z \mid \phi, \Theta) = \phi_{z[1]} \cdot \prod_{n=2}^{N} \theta_{z[n-1], z[n]}.$$

The sequence of observations $y$ is conditionally independent given $z$,

$$p(y \mid z) = \prod_{n=1}^{N} p(y \mid z_n = k).$$

The $N \times K$ emission matrix is defined by taking

$$\Lambda_{n,k} = p(y \mid z_n = k).$$

The complete data density for HMMs is

$$p(y, z \mid \phi, \Theta, \Lambda) = \phi_{z[1]} \cdot \prod_{n=2}^{N} \Theta_{z[n-1], z[n]} \cdot \prod_{n=1}^{N} \Lambda_{n, z[n]}.$$

The density is defined by marginalizing out the unobserved latent states $z$,

$$p(y \mid \phi, \Theta, \Lambda) = \sum_{z \in (1:K)^N} p(y, z \mid \phi, \Theta, \Lambda).$$

The goal is to compute the derivatives of this function for a fixed observation sequence $y$ with respect to the parameters $\phi$, $\Theta$, and $\Lambda$.

The direct summation is intractable because there are $K^N$ possible values for the sequence $z$. The forward algorithm uses dynamic

programming to compute the marginal likelihood in $\mathcal{O}(K^2 \cdot N)$. The forward algorithm is neatly derived from the matrix expression for the density,

$$p(y \mid \phi, \theta, \Lambda) = \phi^\top \cdot \text{diag}(\Lambda_1) \cdot \Theta \cdot \text{diag}(\Lambda_2) \cdots \Theta \cdot \text{diag}(\Lambda_N) \cdot 1$$

where $1 = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}^\top$ is a vector of ones of size $K$, and

$$\text{diag}(\Lambda_n) = \begin{bmatrix} \Lambda_{n,1} & 0 & \cdots & 0 \\ 0 & \Lambda_{n,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \Lambda_{n,K} \end{bmatrix}.$$

The forward algorithm is traditionally defined in terms of the forward vectors,

$$\alpha_n = \begin{bmatrix} \phi^\top \cdot \text{diag}(\Lambda_1) & \cdots & \Theta \cdot \text{diag}(\Lambda_n) \end{bmatrix}^\top,$$

which are column vectors formed from the prefixes of the likelihood function. A final multiplication by $1$ yields a means to compute the likelihood function. This forward algorithm may be automtically differentiated and the resulting derivative calculation also takes $\mathcal{O}(K^2 \cdot N)$. But the constant factor and memory usage is high, so it is more efficient to work out derivatives analytically.

The backward algorithm defines the backward row vectors,

$$\beta_n = \begin{bmatrix} \Theta \cdot \text{diag}(\Lambda_n) & \cdots & \Theta \cdot \text{diag}(\Lambda_N) \cdot 1 \end{bmatrix}.$$

The recursive form of the backward algorithm begins with $\beta_N$, then defines $\beta_{n-1}$ in terms of $\beta_N$.

The derivative of the HMM density can be rendered as a sum of terms involving forward and backward variables by repeatedly applying the chain rule to peel pairs of terms off of the product, resulting in a sum

$$\begin{aligned}
\frac{\partial}{\partial x} p(y \mid \phi, \Theta, \Lambda) &= \frac{\partial}{\partial x} \phi^\top \cdot \text{diag}(\Lambda_1) \cdot \Theta \cdot \text{diag}(\Lambda_2) \cdots \Theta \cdot \text{diag}(\Lambda_N) \cdot 1 \\
&= \left( \frac{\partial}{\partial x} \phi^\top \cdot \text{diag}(\Lambda_1) \right) \cdot \Theta \cdot \text{diag}(\Lambda_2) \cdots \Theta \cdot \text{diag}(\Lambda_N) \cdot 1 \\
&\quad + \phi^\top \cdot \text{diag}(\Lambda_1) \cdot \left( \frac{\partial}{\partial x} \Theta \cdot \text{diag}(\Lambda_2) \cdots \Theta \cdot \text{diag}(\Lambda_N) \right) \\
&= \qquad\qquad \vdots \\
&= \left( \frac{\partial}{\partial x} \phi^\top \cdot \text{diag}(\Lambda_1) \right) \cdot \beta_1 + \sum_{n=2}^{N} \alpha_{n-1}^\top \cdot \left( \frac{\partial}{\partial x} \Theta \cdot \text{diag}(\Lambda_n) \right) \cdot \beta_n
\end{aligned}$$

involving the forward terms $\alpha$, backward terms $\beta$, and the derivatives of the parameters $\phi, \Theta, \Lambda$.

To simplify the notation, let

$$\mathcal{L} = p(y \mid \phi, \Theta, \Lambda).$$

The derivative with respect to the initial distribution $\phi$ is

$$\frac{\partial}{\partial \phi}\mathcal{L} = \text{diag}(\Lambda_1) \cdot \beta_1.$$

The derivative with respect to the initial emission density $\Lambda_1$ is

$$\frac{\partial}{\partial \Lambda_1}\mathcal{L} = \text{diag}(\phi) \cdot \beta_1.$$

The derivative with respect to the emission density $\Lambda_n$ for $n > 1$ is

$$\frac{\partial}{\partial \Lambda_n}\mathcal{L} = \alpha_{n-1}^\top \cdot \Theta \cdot \beta_n.$$

The deriative with respect to the stochastic transition matrix $\Theta$ is

$$\frac{\partial}{\partial \Theta}\mathcal{L} = \sum_{n=2}^{N} \alpha_{n-1}^\top \cdot \text{diag}(\Lambda_n) \cdot \beta_n.$$

## *References*

The matrix form of the likelihood and forward-backward algorithm, as well as the matrix derivatives are based on the presentaiton in (Qin, Auerbach, and Sachs 2000).

# *Ordinary Differential Equations*

A differential equation solver takes a system function defining the differential equation, a starting state, a starting time and a set of requested solution times. The solutions to the differential equation given the starting state are then calculated and returned. To be useful for automatic differentiation, the sensitivies of the solution to their input must be calculated.

# *Algebraic Equations*

Algebraic equations take the form $f(x) = 0$. It's possible to calculate sensitivies of these solutions and hence automatically differentiate through solvers.

# *Complex Numbers*

A complex number is described in terms of a real component $x$ and imaginary component $w$ as $z = x + w \cdot i$, where $i = \sqrt{-1}$. There is nothing special about autodiffing complex functions in that they can be considered as just functions on pairs $(x, w)$. Both adjoints $\bar{x}, \bar{w}$ and tangents $\dot{x}$ and $\dot{w}$ may be computed. What is not computed is complex derivatives, that is differentiating with respect to the entire complex number $z$.

# *Definite Integrals*

It is possible to autodiff through definite integrals of the form

$$y = \int_a^b f(x, \theta)\, \mathrm{d}x.$$

to calcuate derivatives with respect to $\theta$.

# *Changes of Variables*

When working with an inference or optimization algorithm, it is convenient to have functions defined on all of $\mathbb{R}^N$. Yet most models are more naturally formulated in terms of constrained parameters like probabilities (values in $[0, 1]$), correlations (values in $[-1, 1]$), variances (in $[0, \infty)$). Even more complex constraints are imposed on covariance matrices (positive definite matrices) or simplexes (non-negative sequence of values summing to one).

In all of these cases, it is possible to transform the constrained variable to the unconstrained scale. When transforming back from the constrained scale to the unconstrained scale, a Jacobian adjustment is necessary to account for the change of variables. And to make the resulting function differentiable, this Jacobian must be differentiable.

This chapter collects autodiff results for the inverses of several popular constraining transforms.

# *Monte Carlo Methods*

Monte Carlo methods are used to compute high-dimensional integrals corresponding to expectations. For example, if $Y \in \mathbb{R}$ is a real-valued random variable and $f : \mathbb{R} \to \mathbb{R}$ is a real-valued function, then the expectation of the function applied to the random variable is

$$\mathbb{E}[f(y)] = \int_{\mathbb{R}} f(y) \cdot p_Y(y) \, \mathrm{d}y,$$

where $p_Y(y)$ is the density function for $Y$.

Suppose it possible to generate a sequence of random draws

$$y^{(1)}, \ldots, y^{(m)}, \ldots \sim p_Y(y)$$

distributed according to $p_Y(y)$. With these random draws, the expectation may be reformulated as as the limit of the average value of the function applied to the draws,

$$\mathbb{E}[f(y)] = \lim_{M \to \infty} \frac{1}{M} \sum_{m=1}^{M} f(y^{(m)}).$$

Given finite computation time, an approximate result can be computed for some fixed $M$ as

$$\mathbb{E}[f(y)] \approx \frac{1}{M} \sum_{m=1}^{M} f(y^{(m)}).$$

This is known as a Monte Carlo method, after the casino of that name in Monaco.

If the function $f(y)$ depends on other parameters $z$, e.g., $f(y) = g(y, z)$ and derivatives are required of the expectation, they can be moved inside the integral. That is, taking the expetactation over random variable $Y$ and ddifferentiating with respect to $z$ yields

$$\frac{\partial}{\partial z} \mathbb{E}[g(y, z)] = \mathbb{E}\left[\frac{\partial}{\partial z} g(y, z)\right].$$

The Monte Carlo approach is to approximate the derivative with

$$\frac{\partial}{\partial z} \mathbb{E}[g(y, z)] \approx \frac{1}{M} \sum_{m=1}^{M} \frac{\partial}{\partial z} g(y^{(m)}, z).$$

The internal derivative $\frac{\partial}{\partial z} g(y^{(m)}, z)$ may be computed via automatic differentiation and the results summed. No new rules of automatic differentiation are required, as the resulting term is a simple sum.

## *Max marginal optimization and sampling*

A common application for Monte Carlo integration is computing marginal densities,

$$p_\Phi(\phi) = \int_{\mathbb{R}} p_{\Phi,\Theta}(\phi, \theta) \, \mathrm{d}\theta,$$

for a given input value $\phi$. A Monte Carlo approximation can be calculated as

$$p_\Phi(\phi) \approx \frac{1}{M} \sum_{m=1}^{M} p_{\Phi,\Theta}(\phi, \theta^{(m)}),$$

where for $m \in 1 : M$,

$$\theta^{(m)} \sim p_{\Theta|\Phi}(\theta \mid \phi).$$

Gradient-based samplers or optimizers when $\Phi$ is multivariate require the gradient $\nabla p_\Phi(\phi)$. This can be calculated via Monte Carlo as

$$\nabla p_\Phi(\phi) \approx \frac{1}{M} \sum_{m=1}^{M} \nabla p_{\Phi,\Theta}(\phi, \theta^{(m)}).$$

For example, the maximum marginal likelihood estimate $\phi^*$ is defined as an optimization,

$$\phi^* = \arg\max_\phi \, p_\Phi(\phi).$$

Optimizing $p_\Phi$ with the marginalization computed by Markov chain Monte Carlo methods leads to an algorithm known as Markov chain Monte Carlo expectation maximization (MCMC-EM).

## *Stochastic gradients*

These are all examples of stochastic gradient calculations.

# *Bibliography*

# *(APPENDIX) Appendices*

# *Reference C++ Implementations*

## *Forward-mode automatic differentiation in C++*

Forward-mode automatic differentiation can be implemented directly in C++ following the pattern established in the standard library for complex numbers. A forward-mode autodiff variable is represented by a dual number holding two scalar values, a constructor where the second value defaults to zero, and getters for the value and tangent.

```
namespace autodiff {
template <typename T>
class dual {
  T val_;
  T tan_;
 public:
  dual(const T& val = 0, const T& tan = 0)
      : val_(val), tan_(tan) { }
  const T& val() const { return val_; }
  const T& tan() const { return tan_; }
};
}
```

The type of value is templated to supported nested forward mode. The default copy constructor `dual(const dual<T>&)`, destructor `~dual()`, and assignment operator `operator=(const dual<T>&)` are sufficient.

Dual numbers are constructed from either a value and a tangent, or just a value with default zero tangent, or with neither and default zero values and tangent.

```
using autodiff::dual;
dual<double> a(2.2, -3.1);
dual<double> b(5);
dual<double> c;
```

The first has a value of 2.2 and a tangent of -3.1, whereas the second has a value of 5 and a tangent of 0 (the default) and the last a

value and tangent of 0.

The constructor is not declared implicit and thus will support the assignment of primitives, giving them default 0 tangent values. The assignments

```
dual<double> c = 5;
```

and

```
dual<double> d = dual<double>(5);
```

both invoke the default copy assignment operator `operator=(const dual<double>&)`; the former promotes `5` to `dual<double>(5)` using the non-explicit unary constructor.

Functions and operators are coded following the dual arithmetic. All autodiff functionality will be declared in the namespace `autodiff`, though the namespace qualification will not be shown. For example, exponentiation is defined following the dual arithmetic rule.

```
#include <cmath>
template <typename T>
inline dual<T> exp(const dual<T>& x) {
  using std::exp;
  T y = exp(x.val());
  return dual(y, x.tan() * y);
}
```

So is the logarithm function.

```
template <typename T>
inline dual<T> log(const dual<T>& x) {
  using std::log;
  T y = log(x.val());
  return dual(y, x.tan() / y );
}
```

The standard library `cmath` is required for definitions of `exp` and `log` for primitive values. The function definitions begin with using statements, e.g., `using std::exp`. This allows the `exp` defined in the standard library `cmath` to be used for primitives and the `exp` defined for type `T` to be found by argument-dependent lookup for autodiff value types `T`.

Binary operations can be implemented following the dual number definitions.

```
template <typename T>
```

```
inline dual<T> operator*(const dual<T>& x1, const dual<T>& x2) {
  return dual(x1.val() * x2.val(),
              x1.tan() * x2.val() + x1.val() * x2.tan());
}
```

This suffices for the case where both arguments are autodiff variables,

```
dual<double> x1 = -1.3;
dual<double> x2 = 2.1;
dual<double> u = x1 * x2;
```

There is no need to explicitly bring in `autodiff::operator+` because it is included implicitly by argument-dependent lookup.

The following statements which mix autodiff variables and primitives will not match the templated `operator*` because the primitive argment types do not match the template.

```
dual<double> u = 1.2;
dual<double> v = u * 3.2;
dual<double> w = 2 * u;
```

The multiplication operator (`operator*`) can be further overloaded in order to support these mixed types.

```
#include <type_traits>
template <typename T, typename U,
          typename = std::enable_if_t<std::is_arithmetic_v<U>>>
inline dual<T> operator*(const dual<T>& x1, const U& c2) {
  return dual(x1.val() * c2, x1.tan() * c2);
}

template <typename T, typename U,
          typename = std::enable_if_t<std::is_arithmetic_v<U>>>
inline dual<T> operator*(const U& c1, const dual<T>& x2) {
  return dual(c1 * x2.val(), c1 * x2.tan());
}
```

The third template argument invlves a C++ idiom that requires the template parameter `U` to be a primitive.[7]

`U` being preventing a match of the template function unless `U` is a primitive; the functions `enable_if_t` and `is_arithmetic_v` are declared in the standard library header `<type_traits>`.

[7] Arithmetic types include only the built-in primitive types `float`, `double`, `long double`, `bool`, `char`, `short`, `int`, or `long int` as of C++17.

## *Reverse-mode automatic differentiation in C++*

Like forward mode, reverse-mode automatic differentiation can be implemented through operator overloading in C++. As with forward

mode, argument-dependent lookup means that templated code will just work with autodiff variables as long as all primitive functions invoked are defined for autodiff types.

A template using statement will reduce the boilerplate in requiring arithmetic arguments.

```
#include <type_traits>

template <typename T>
using enable_if_arithmetic_t
= std::enable_if_t<std::is_arithmetic_v<T>>;
```

The core code for reverse-mode autodiff defines a class `adj` used to store values and an index that will be unique for each subexpression.

```
#include <cstddef>

std::size_t next_ = 0;

class adj {
  double val_;
  std::size_t idx_;
 public:
  template <typename T, typename = enable_if_arithmetic_t<T>>
  adj(T val = 0, int idx = next_++)
    : val_(val), idx_(idx) { }
  double value() const { return val_; }
  double index() const { return idx_; }
};
```

The global counter `next_` is used to assign unique identifiers in sequence to each autodiff variable as it is constructed, so it must be initialized to zero before any autodiff calculations. The autodiff vaiable class `adj` holds a double precision value and a unique index. The constructor is responsible for generating indexes and storing values. The default copy constructor, assignment operator, and destructor suffice here.

Usage is similar to that of forward-mode autodiff variables.

```
using autodiff:adj;
autodiff::next_ = 0;    // initialize stack before starting
adj x(3.7);   // construct from value
```

The constructor call for `x` allocates a unique index and increments the global index counter. Assignment of arithmetic values works by promotion using the implicit constructor, so that

```
adj y = 2.9;  // assignment works by promoting
```

is equivalent to

```
adj y = adj(2.9);
```

In order to carry out reverse-mode automatic differentiation, each expression must create and store a continuation used to propagate adjoints from the result to the operands in the reverse sweep. In the reference implementation, these continuations are pushed ont a global stack as they are created.

```
#include <vector>
#include <functional>

std::vector<std::function<void(std::vector<double>&)>> stack_;
```

The reverse sweep is implemented by the `chain()` function, which takes the variable `y` from which derivatives should be propagated.

```
std::vector<double> chain(const adj& y) {
  std::vector<double> adjoints(y.idx() + 1, 0);
  adjoints[y.idx()] = 1;
  for (auto chain_f = stack_.crbegin();
       chain_f != stack_.crend();
       ++chain_f)
    (*chain_f)(adjoints);
  return adjoints;
}
```

First, the vector `adjoints` of adjoint values is allocated at size `y.idx_ + 1` so that it's large enough to the adjoints of every expression involved in the calculation of `y`; this is guaranteed to be enough because every expression involved in the calculation of `y` has an index lower than `y`'s. The initial values are set to zero in the constructor for `adjoints`. To begin the reverse sweep, the adjoint for `y`, namely `adjoints[y.idx_]` is set to one. Then the stack of continuations is traversed from `y` down to the independent variables, executing each continuation on the stack applied to the adjoint vector. Finally, it returns the adjoints that are calculated so that derivatives may be retrieved.

A simple operation like addition is overloaded as follows.

```
inline adj operator+(const adj& x1, const adj& x2) {
  adj y(x1.val() + x2.val());
  auto f = [=](std::vector<double>& adj) {
    adj[x1.idx()] += adj[y.idx()];
```

```
    adj[x2.idx()] += adj[y.idx()];
  };
  stack_.emplace_back(f);
  return y;
}
```

First, the result `y` is constructed with value equal to adding the values of the arguments, `x1.val_` and `x2.val_`. Then a continuation `f` for the chain rule is defined as an anoymous function using a lambda. The notation `[=]` indicates that the lambda captures the values of variables for later execution by copying. Here, the variables captured are `x1`, `x2`, and `y`. The continuation is declared to take a mutable reference to a vector of double-precision floating point values as an argument—these hold the adjoints of all the subexpressions as declared in the `chain()` function. The body of the continuation follows the reverse-mode adjoint rule for addition, namely adding the adjoint of the result `y` to the adjoint of each of the operands, `x1` and `x2`. After the continuation is defined, it is pushed back onto the global stack. Finally, the value `y` is returned.

While the above code will work by promoting arithmetic values to the adjoint class, it is more efficient to define further overloads that are more specific and avoid the redundant work on the stack.

```
template <typename T, typename = enable_if_arithmetic_t<T>>
inline adj operator+(const adj& x1, T x2) {
  adj y(x1.val() + x2);
  stack_.emplace_back([=](std::vector<double>& adj) {
      adj[x1.idx()] += adj[y.idx()];
    });
  return y;
}

template <typename T, typename = enable_if_arithmetic_t<T>>
inline adj operator+(T x1, const adj& x2) {
  adj y(x1 + x2.val());
  stack_.emplace_back([=](std::vector<double>& adj) {
      adj[x2.idx()] += adj[y.idx()];
    });
  return y;
}
```

Rather than defining a temporary for the continuation, it is pushed directly onto the stack. The value is computed using the value from the adjoint variables and the primitives directly, and the only propagation is to the adjoint operand.

Multiplication is defined similarly, with the captured operand's values and indexes both being used.

```
inline adj operator*(const adj& x1, const adj& x2) {
  adj y(x1.val() * x2.val());
  stack_.emplace_back([=](std::vector<double>& adj) {
      adj[x1.idx()] += x2.val() * adj[y.idx()];
      adj[x2.idx()] += x1.val() * adj[y.idx()];
    });
  return y;
}

template <typename T, typename = enable_if_arithmetic_t<T>>
inline adj operator*(const adj& x1, T x2) {
  adj y(x1.val() * x2);
  stack_.emplace_back([=](std::vector<double>& adj) {
      adj[x1.idx()] += x2 * adj[y.idx()];
    });
  return y;
}

template <typename T, typename = enable_if_arithmetic_t<T>>
inline adj operator*(T x1, const adj& x2) {
  adj y(x1 * x2.val());
  stack_.emplace_back([=](std::vector<double>& adj) {
      adj[x2.idx()] += x1 * adj[y.idx()];
    });
  return y;
}
```

Non-linear functions like exponentiation also follow their definitions. We need the <cmath> library for a definition of the exponential function.

```
#include <cmath>
namespace autodiff {
inline adj exp(const adj& x) {
  adj y(std::exp(x.val()));
  auto f = [=](std::vector<double>& adj) {
    adj[x.idx()] += y.val() * adj[y.idx()];
  };
  stack_.emplace_back(f);
  return y;
}
}
```

The constructor defines the value of `y` to be the value of `x` exponentiated. The adjoint is incremented using the captured value of `y`, namely `exp(x.val_)`, which is the derivative of `y` with respect to `x`.

The following code computes $\nabla f(10.3, -1.1)$, where $f(x_1, x_2) = x_1 \cdot \exp(x_2 \cdot 2) + 7$.

```
#include <iostream>
int main() {
  using autodiff::adj;
  next_idx = 0;
  stack_.clear();
  adj x1 = 10.3;
  adj x2 = -1.1;
  adj y = x1 * exp(x2 * 2) + 7;
  std::vector<double> adjoints = chain(y);
  double dy_dx1 = adjoints(x1.idx_);
  double dy_dx2 = adjoints(x2.idx_);
  std::cout << "grad f = [" << dy_dx1 << ", " << dy_dx2 << "]" << std::endl;
  return 0;
}
```

First, the index counter and stack are reset. Then the inddependent variables `x1` and `x2` are initialized. The resulting dependent variable `y` is computed as a single expression and is also of autodiff variable type. The definitions of `operator*`, `operator+`, and `exp()` are found through argument-dependent lookup. Next, the reverse sweep is carried out starting from the result `y` using the `chain()` function. The resulting adjoints for `x1` and `x2` are found by indexing the vector of adjoints returned by `chain()`. These are then printed and the default success code (zero) is returned.

## *References*

The reverse-mode autodiff implementation is based on (Carpenter 2018). Matrices are implemented with the Eigen C++ library (Guennebaud, Jacob, and others 2020). A thorough and precise introduction to modern C++ template programming is (Vandevoorde, Josuttis, and Gregor 2017).

Carpenter, Bob. 2018. "A New Continuation-Based Autodiff by Refactoring." Stan developer forums. `https://discourse.mc-stan.org/t/5037`.

Giles, Mike B. 2008a. "An Extended Collection of Matrix Derivative Results for Forward and Reverse Mode Automatic Differentiation." Report 08/01. Oxford University.

———. 2008b. "Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation." In *Advances in Automatic Differentiation*, 64:35–44. Lecture Notes in Computational Science and Engineering. Springer.

Guennebaud, Gaël, Benoît Jacob, and others. 2020. "Eigen Version 3." Web site. `http://eigen.tuxfamily.org`.

Magnus, Jan R, and Heinz Neudecker. 2019. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Third Edition. John Wiley & Sons.

Qin, Feng, Anthony Auerbach, and Frederick Sachs. 2000. "A Direct Optimization Approach to Hidden Markov Modeling for Single Channel Kinetics." *Biophysical Journal* 79 (4): 1915–27.

Vandevoorde, David, Nicolai M Josuttis, and Douglas Gregor. 2017. "C++ Templates: The Complete Guide." Addison-Wesley.