WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

Automatic differentiation

In mathematics and computer algebra, **automatic differentiation** (**auto-differentiation**, **autodiff**, or **AD**), also called **algorithmic differentiation**, **computational differentiation**,[1][2] is a set of techniques to evaluate the partial derivative of a function specified by a computer program.

Automatic differentiation exploits the fact that every computer calculation, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, partial derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor of more arithmetic operations than the original program.

Difference from other differentiation methods

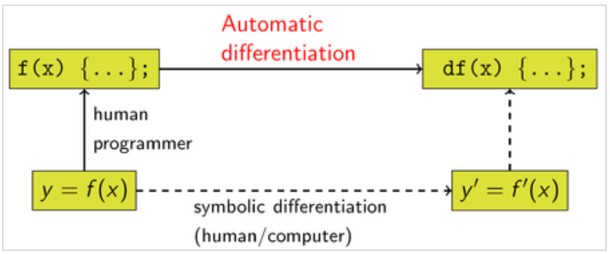

Automatic differentiation
f(x) {...};
df(x) {...};
human programmer
y = f(x)
y' = f'(x)
symbolic differentiation (human/computer)

Figure 1: How automatic differentiation relates to symbolic differentiation

Automatic differentiation is distinct from symbolic differentiation and numerical differentiation. Symbolic differentiation faces the difficulty of converting a computer program into a single mathematical expression and can lead to inefficient code. Numerical differentiation (the method of finite differences) can introduce round-off errors in the discretization process and cancellation. Both of these classical methods have problems with calculating higher derivatives, where complexity and errors increase. Finally, both of these classical methods are slow at computing partial derivatives of a function with respect to *many* inputs, as is needed for gradient-based optimization algorithms. Automatic differentiation solves all of these problems.

Forward and reverse accumulation

Chain rule of partial derivatives of composite functions

Fundamental to automatic differentiation is the decomposition of differentials provided by the chain rule of partial derivatives of composite functions. For the simple composition

$$\begin{aligned} y &= f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3 \\ w_0 &= x \\ w_1 &= h(w_0) \\ w_2 &= g(w_1) \\ w_3 &= f(w_2) = y \end{aligned}$$

the chain rule gives

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x} = \frac{\partial f(w_2)}{\partial w_2}\frac{\partial g(w_1)}{\partial w_1}\frac{\partial h(w_0)}{\partial x}$$

Two types of automatic differentiation

Usually, two distinct modes of automatic differentiation are presented.

- **forward accumulation** (also called **bottom-up**, **forward mode**, or **tangent mode**)
- **reverse accumulation** (also called **top-down**, **reverse mode**, or **adjoint mode**)

Forward accumulation specifies that one traverses the chain rule from inside to outside (that is, first compute $\partial w_1/\partial x$ and then $\partial w_2/\partial w_1$ and at last $\partial y/\partial w_2$), while reverse accumulation has the traversal from outside to inside (first compute $\partial y/\partial w_2$ and then $\partial w_2/\partial w_1$ and at last $\partial w_1/\partial x$). More succinctly,

- Forward accumulation computes the recursive relation: $\frac{\partial w_i}{\partial x} = \frac{\partial w_i}{\partial w_{i-1}}\frac{\partial w_{i-1}}{\partial x}$ with $w_3 = y$, and,
- Reverse accumulation computes the recursive relation: $\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}}\frac{\partial w_{i+1}}{\partial w_i}$ with $w_0 = x$.

The value of the partial derivative, called *seed*, is propagated forward or backward and is initially $\frac{\partial x}{\partial x} = 1$ or $\frac{\partial y}{\partial y} = 1$. Forward accumulation evaluates the function and calculates the derivative with respect to one independent variable in one pass. For each independent variable $x_1, x_2, \dots, x_n$ a separate pass is therefore necessary in which the derivative with respect to that independent variable is set to one ($\frac{\partial x_1}{\partial x_1} = 1$) and of all others to zero ($\frac{\partial x_2}{\partial x_1} = \dots = \frac{\partial x_n}{\partial x_1} = 0$). In contrast, reverse accumulation requires the evaluated partial functions for the partial derivatives. Reverse accumulation therefore evaluates the function first and calculates the derivatives with respect to all independent variables in an additional pass.

Which of these two types should be used depends on the sweep count. The computational complexity of one sweep is proportional to the complexity of the original code.

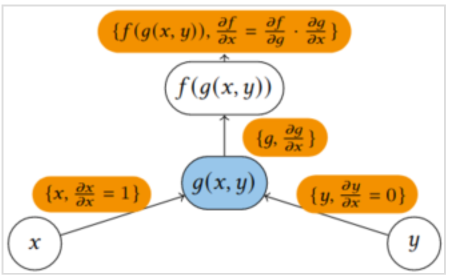
- Forward accumulation is more efficient than reverse accumulation for functions $f: \mathbf{R}^n \to \mathbf{R}^m$ with $n \ll m$ as only n sweeps are necessary, compared to m sweeps for reverse accumulation.
- Reverse accumulation is more efficient than forward accumulation for functions $f: \mathbf{R}^n \to \mathbf{R}^m$ with $n \gg m$ as only m sweeps are necessary, compared to n sweeps for forward accumulation.

Backpropagation of errors in multilayer perceptrons, a technique used in machine learning, is a special case of reverse accumulation.[2]

Forward accumulation was introduced by R.E. Wengert in 1964.[3] According to Andreas Griewank, reverse accumulation has been suggested since the late 1960s, but the inventor is unknown.[4] Seppo Linnainmaa published reverse accumulation in 1976.[5]

Forward accumulation

In forward accumulation AD, one first fixes the *independent variable* with respect to which differentiation is performed and computes the derivative of each sub-expression recursively. In a pen-and-paper calculation, this involves repeatedly substituting the derivative of the *inner* functions in the chain rule:


$\{f(g(x,y)), \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\cdot\frac{\partial g}{\partial x}\}$
$f(g(x,y))$
$\{g, \frac{\partial g}{\partial x}\}$
$\{x, \frac{\partial x}{\partial x} = 1\}$
$g(x,y)$
$\{y, \frac{\partial y}{\partial x} = 0\}$
x
y

Forward accumulation

$$\begin{aligned}\frac{\partial y}{\partial x} &= \frac{\partial y}{\partial w_{n-1}}\frac{\partial w_{n-1}}{\partial x} \\ &= \frac{\partial y}{\partial w_{n-1}}\left(\frac{\partial w_{n-1}}{\partial w_{n-2}}\frac{\partial w_{n-2}}{\partial x}\right) \\ &= \frac{\partial y}{\partial w_{n-1}}\left(\frac{\partial w_{n-1}}{\partial w_{n-2}}\left(\frac{\partial w_{n-2}}{\partial w_{n-3}}\frac{\partial w_{n-3}}{\partial x}\right)\right) \\ &= \cdots\end{aligned}$$

前向过程其实与泛函一样，表述一个小的扰动对输出的影响。因此相当于泛函微分元的前向传导

This can be generalized to multiple variables as a matrix product of Jacobians.

Compared to reverse accumulation, forward accumulation is natural and easy to implement as the flow of derivative information coincides with the order of evaluation. Each variable w_i is augmented with its derivative $\dot{w}_i$ (stored as a numerical value, not a symbolic expression),

$$\dot{w}_i = \frac{\partial w_i}{\partial x}$$

as denoted by the dot. The derivatives are then computed in sync with the evaluation steps and combined with other derivatives via the chain rule.

Using the chain rule, if w_i has predecessors in the computational graph:

$$\dot{w}_i = \sum_{j\in\{\text{predecessors of i}\}} \frac{\partial w_i}{\partial w_j}\dot{w}_j$$

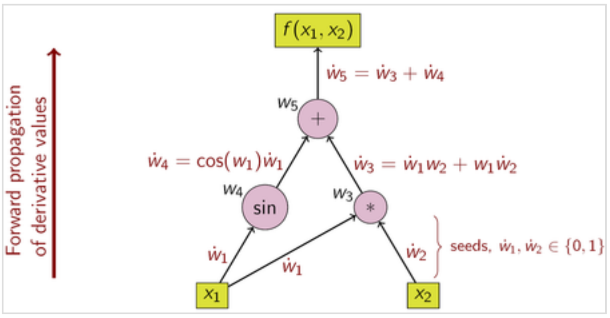
As an example, consider the function:

$$\begin{aligned}y &= f(x_1, x_2) \\ &= x_1 x_2 + \sin x_1 \\ &= w_1 w_2 + \sin w_1 \\ &= w_3 + w_4 \\ &= w_5\end{aligned}$$

For clarity, the individual sub-expressions have been labeled with the variables w_i.

The choice of the independent variable to which differentiation is performed affects the *seed* values $\dot{w}_1$ and $\dot{w}_2$. Given interest in the derivative of this function with respect to x_1, the seed values should be set to:

$$\dot{w}_1 = \frac{\partial w_1}{\partial x_1} = \frac{\partial x_1}{\partial x_1} = 1$$
$$\dot{w}_2 = \frac{\partial w_2}{\partial x_1} = \frac{\partial x_2}{\partial x_1} = 0$$



$f(x_1, x_2)$
$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$
w_5
$+$
$\dot{w}_4 = \cos(w_1)\dot{w}_1$
$\dot{w}_3 = \dot{w}_1 w_2 + w_1 \dot{w}_2$
w_4
sin
w_3
$*$
$\dot{w}_1$
$\dot{w}_1$
$\dot{w}_2$
seeds, $\dot{w}_1, \dot{w}_2 \in \{0, 1\}$
x_1
x_2
Forward propagation of derivative values

Figure 2: Example of forward accumulation with computational graph

With the seed values set, the values propagate using the chain rule as shown. Figure 2 shows a pictorial depiction of this process as a computational graph.

Operations to compute value	**Operations to compute derivative**
$w_1 = x_1$	$\dot{w}_1 = 1$ (seed)
$w_2 = x_2$	$\dot{w}_2 = 0$ (seed)
$w_3 = w_1 \cdot w_2$	$\dot{w}_3 = w_2 \cdot \dot{w}_1 + w_1 \cdot \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \cdot \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

To compute the gradient of this example function, which requires not only $\frac{\partial y}{\partial x_1}$ but also $\frac{\partial y}{\partial x_2}$, an *additional* sweep is performed over the computational graph using the seed values $\dot{w}_1 = 0; \dot{w}_2 = 1$.

Implementation

Pseudo Code

Forward accumulation calculates the function and the derivative (but only for one independent variable each) in one pass. The associated method call expects the expression *Z* to be derived with regard to a variable *V*. The method returns a pair of the evaluated function and its derivation. The method traverses the expression tree recursively until a variable is reached. If the derivative with respect to this variable is requested, its derivative is 1, 0 otherwise. Then the partial function as well as the partial derivative are evaluated.[6]

```
tuple<float,float> evaluateAndDerive(Expression Z, Variable V) {
   if isVariable(Z)
      if (Z = V) return {valueOf(Z), 1};
      else return {valueOf(Z), 0};
   else if (Z = A + B)
      {a, a’} = evaluateAndDerive(A, V);
      {b, b’} = evaluateAndDerive(B, V);
      return {a + b, a’ + b’};
   else if (Z = A - B)
      {a, a’} = evaluateAndDerive(A, V);
      {b, b’} = evaluateAndDerive(B, V);
      return {a - b, a’ - b’};
   else if (Z = A * B)
      {a, a'} = evaluateAndDerive(A, V);
      {b, b'} = evaluateAndDerive(B, V);
      return {a * b, b * a' + a * b'};
}
```

Python

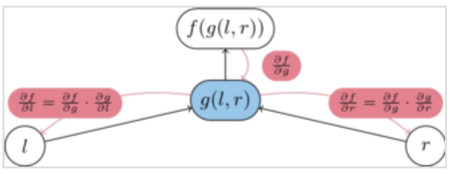
```python
class ValueAndPartial:
    def __init__(self, value, partial):
        self.value = value
        self.partial = partial

    def toList(self):
        return [self.value, self.partial]

class Expression:
    def __add__(self, other):
        return Plus(self, other)

    def __mul__(self, other):
        return Multiply(self, other)

class Variable(Expression):
    def __init__(self, value):
        self.value = value

    def evaluateAndDerive(self, variable):
        partial = 1 if self == variable else 0
        return ValueAndPartial(self.value, partial)

class Plus(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB

    def evaluateAndDerive(self, variable):
        valueA, partialA = self.expressionA.evaluateAndDerive(variable).toList()
        valueB, partialB = self.expressionB.evaluateAndDerive(variable).toList()
        return ValueAndPartial(valueA + valueB, partialA + partialB)

class Multiply(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB

    def evaluateAndDerive(self, variable):
        valueA, partialA = self.expressionA.evaluateAndDerive(variable).toList()
        valueB, partialB = self.expressionB.evaluateAndDerive(variable).toList()
        return ValueAndPartial(valueA * valueB, valueB * partialA + valueA * partialB)

# Example: Finding the partials of z = x * (x + y) + y * y at (x, y) = (2, 3)
x = Variable(2)
y = Variable(3)
z = x * (x + y) + y * y
xPartial = z.evaluateAndDerive(x).partial
yPartial = z.evaluateAndDerive(y).partial
print("∂z/∂x =", xPartial) # Output: ∂z/∂x = 7
print("∂z/∂y =", yPartial) # Output: ∂z/∂y = 8
```

C++

```cpp
#include <iostream>
struct ValueAndPartial { float value, partial; };
struct Variable;
struct Expression {
   virtual ValueAndPartial evaluateAndDerive(Variable *variable) = 0;
};
struct Variable: public Expression {
   float value;
   Variable(float value): value(value) {}
   ValueAndPartial evaluateAndDerive(Variable *variable) {
      float partial = (this == variable) ? 1.0f : 0.0f;
      return {value, partial};
   }
};
struct Plus: public Expression {
   Expression *a, *b;
   Plus(Expression *a, Expression *b): a(a), b(b) {}
   ValueAndPartial evaluateAndDerive(Variable *variable) {
      auto [valueA, partialA] = a->evaluateAndDerive(variable);
      auto [valueB, partialB] = b->evaluateAndDerive(variable);
      return {valueA + valueB, partialA + partialB};
   }
};
struct Multiply: public Expression {
   Expression *a, *b;
   Multiply(Expression *a, Expression *b): a(a), b(b) {}
   ValueAndPartial evaluateAndDerive(Variable *variable) {
      auto [valueA, partialA] = a->evaluateAndDerive(variable);
      auto [valueB, partialB] = b->evaluateAndDerive(variable);
      return {valueA * valueB, valueB * partialA + valueA * partialB};
   }
};
int main () {
   // Example: Finding the partials of z = x * (x + y) + y * y at (x, y) = (2, 3)
   Variable x(2), y(3);
   Plus p1(&x, &y); Multiply m1(&x, &p1); Multiply m2(&y, &y); Plus z(&m1, &m2);
   float xPartial = z.evaluateAndDerive(&x).partial;
   float yPartial = z.evaluateAndDerive(&y).partial;
   std::cout << "∂z/∂x = " << xPartial << ", "
             << "∂z/∂y = " << yPartial << std::endl;
   // Output: ∂z/∂x = 7, ∂z/∂y = 8
   return 0;
}
```

Reverse accumulation

In reverse accumulation AD, the *dependent variable* to be differentiated is fixed and the derivative is computed *with respect to* each sub-expression recursively. In a pen-and-paper calculation, the derivative of the *outer* functions is repeatedly substituted in the chain rule:


$f(g(l,r))$
$\frac{\partial f}{\partial g}$
$\frac{\partial f}{\partial l} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial l}$
$g(l,r)$
$\frac{\partial f}{\partial r} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial r}$
l
r
Reverse accumulation

$$\begin{aligned}\frac{\partial y}{\partial x} &= \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x}\\ &= \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x}\\ &= \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x}\\ &= \cdots\end{aligned}$$

In reverse accumulation, the quantity of interest is the *adjoint*, denoted with a bar $\bar{w}_i$; it is a derivative of a chosen dependent variable with respect to a subexpression w_i:

$$\bar{w}_i = \frac{\partial y}{\partial w_i}$$

Using the chain rule, if w_i has successors in the computational graph:

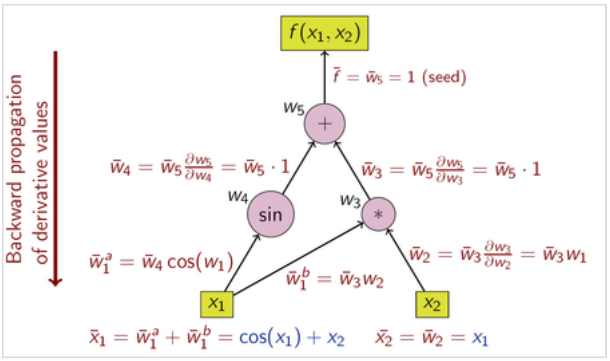
$$\bar{w}_i = \sum_{j \in \{\text{successors of i}\}} \bar{w}_j \frac{\partial w_j}{\partial w_i}$$

Reverse accumulation traverses the chain rule from outside to inside, or in the case of the computational graph in Figure 3, from top to bottom. The example function is scalar-valued, and thus there is only one seed for the derivative computation, and only one sweep of the computational graph is needed to calculate the (two-component) gradient. This is only half the work when compared to forward accumulation, but reverse accumulation requires the storage of the intermediate variables w_i as well as the instructions that produced them in a data structure known as a "tape" or a Wengert list[7] (however, Wengert published forward accumulation, not reverse accumulation[3]), which may consume significant memory if the computational graph is large. This can be mitigated to some extent by storing only a subset of the intermediate variables and then reconstructing the necessary work variables by repeating the evaluations, a technique known as rematerialization. Checkpointing is also used to save intermediary states.

The operations to compute the derivative using reverse accumulation are shown in the table below (note the reversed order):

Operations to compute derivative

$$\begin{aligned}
\bar{w}_5 &= 1 \text{ (seed)} \\
\bar{w}_4 &= \bar{w}_5 \cdot 1 \\
\bar{w}_3 &= \bar{w}_5 \cdot 1 \\
\bar{w}_2 &= \bar{w}_3 \cdot w_1 \\
\bar{w}_1 &= \bar{w}_3 \cdot w_2 + \bar{w}_4 \cdot \cos w_1
\end{aligned}$$


$f(x_1, x_2)$
$\bar{f} = \bar{w}_5 = 1$ (seed)
Backward propagation of derivative values
w_5
$+$
$\bar{w}_4 = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = \bar{w}_5 \cdot 1$
$\bar{w}_3 = \bar{w}_5 \frac{\partial w_5}{\partial w_3} = \bar{w}_5 \cdot 1$
w_4
sin
w_3
$*$
$\bar{w}_1^a = \bar{w}_4 \cos(w_1)$
$\bar{w}_1^b = \bar{w}_3 w_2$
$\bar{w}_2 = \bar{w}_3 \frac{\partial w_3}{\partial w_2} = \bar{w}_3 w_1$
x_1
x_2
$\bar{x}_1 = \bar{w}_1^a + \bar{w}_1^b = \cos(x_1) + x_2$
$\bar{x}_2 = \bar{w}_2 = x_1$
Figure 3: Example of reverse accumulation with computational graph

The data flow graph of a computation can be manipulated to calculate the gradient of its original calculation. This is done by adding an adjoint node for each primal node, connected by adjoint edges which parallel the primal edges but flow in the opposite direction. The nodes in the adjoint graph represent multiplication by the derivatives of the functions calculated by the nodes in the primal. For instance, addition in the primal causes fanout in the adjoint; fanout in the primal causes addition in the adjoint;[a] a unary function $y = f(x)$ in the primal causes $\bar{x} = \bar{y} f'(x)$ in the adjoint; etc.

Implementation

Pseudo Code

Reverse accumulation requires two passes: In the forward pass, the function is evaluated first and the partial results are cached. In the reverse pass, the partial derivatives are calculated and the previously derived value is backpropagated. The corresponding method call expects the expression *Z* to be derived and *seed* with the derived value of the parent expression. For the top expression, Z derived with regard to Z, this is 1. The method traverses the expression tree recursively until a variable is reached and adds the current *seed* value to the derivative expression.[8][9]

```
void derive(Expression Z, float seed) {
   if isVariable(Z)
      partialDerivativeOf(Z) += seed;
   else if (Z = A + B)
      derive(A, seed);
      derive(B, seed);
   else if (Z = A - B)
      derive(A, seed);
      derive(B, -seed);
   else if (Z = A * B)
      derive(A, valueOf(B) * seed);
      derive(B, valueOf(A) * seed);
}
```

Python

```
class Expression:
    def __add__(self, other):
        return Plus(self, other)
    def __mul__(self, other):
        return Multiply(self, other)

class Variable(Expression):
    def __init__(self, value):
        self.value = value
        self.partial = 0

    def evaluate(self):
        pass

    def derive(self, seed):
        self.partial += seed

class Plus(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB
        self.value = None

    def evaluate(self):
        self.expressionA.evaluate()
        self.expressionB.evaluate()
        self.value = self.expressionA.value + self.expressionB.value

    def derive(self, seed):
        self.expressionA.derive(seed)
        self.expressionB.derive(seed)

class Multiply(Expression):
    def __init__(self, expressionA, expressionB):
        self.expressionA = expressionA
        self.expressionB = expressionB
        self.value = None

    def evaluate(self):
        self.expressionA.evaluate()
        self.expressionB.evaluate()
        self.value = self.expressionA.value * self.expressionB.value

    def derive(self, seed):
        self.expressionA.derive(self.expressionB.value * seed)
        self.expressionB.derive(self.expressionA.value * seed)

# Example: Finding the partials of z = x * (x + y) + y * y at (x, y) = (2, 3)
x = Variable(2)
y = Variable(3)
z = x * (x + y) + y * y
z.evaluate()
print("z =", z.value)       # Output: z = 19
z.derive(1)
print("∂z/∂x =", x.partial) # Output: ∂z/∂x = 7
print("∂z/∂y =", y.partial) # Output: ∂z/∂y = 8
```

C++

```
#include <iostream>
struct Expression {
   float value;
   virtual void evaluate() = 0;
   virtual void derive(float seed) = 0;
};
struct Variable: public Expression {
   float partial;
   Variable(float _value) {
      value = _value;
      partial = 0;
   }
   void evaluate() {}
   void derive(float seed) {
      partial += seed;
   }
};
struct Plus: public Expression {
   Expression *a, *b;
   Plus(Expression *a, Expression *b): a(a), b(b) {}
   void evaluate() {
      a->evaluate();
      b->evaluate();
      value = a->value + b->value;
   }
   void derive(float seed) {
      a->derive(seed);
      b->derive(seed);
   }
};
struct Multiply: public Expression {
   Expression *a, *b;
   Multiply(Expression *a, Expression *b): a(a), b(b) {}
   void evaluate() {
      a->evaluate();
      b->evaluate();
      value = a->value * b->value;
   }
   void derive(float seed) {
      a->derive(b->value * seed);
      b->derive(a->value * seed);
   }
};
int main () {
   // Example: Finding the partials of z = x * (x + y) + y * y at (x, y) = (2, 3)
   Variable x(2), y(3);
   Plus p1(&x, &y); Multiply m1(&x, &p1); Multiply m2(&y, &y); Plus z(&m1, &m2);
   z.evaluate();
   std::cout << "z = " << z.value << std::endl;
```

```
    // Output: z = 19
    z.derive(1);
    std::cout << "∂z/∂x = " << x.partial << ", "
              << "∂z/∂y = " << y.partial << std::endl;
    // Output: ∂z/∂x = 7, ∂z/∂y = 8
    return 0;
}
```

Beyond forward and reverse accumulation

Forward and reverse accumulation are just two (extreme) ways of traversing the chain rule. The problem of computing a full Jacobian of $f : \mathbf{R}^n \to \mathbf{R}^m$ with a minimum number of arithmetic operations is known as the *optimal Jacobian accumulation* (OJA) problem, which is NP-complete.[10] Central to this proof is the idea that algebraic dependencies may exist between the local partials that label the edges of the graph. In particular, two or more edge labels may be recognized as equal. The complexity of the problem is still open if it is assumed that all edge labels are unique and algebraically independent.

Automatic differentiation using dual numbers

Forward mode automatic differentiation is accomplished by augmenting the algebra of real numbers and obtaining a new arithmetic. An additional component is added to every number to represent the derivative of a function at the number, and all arithmetic operators are extended for the augmented algebra. The augmented algebra is the algebra of dual numbers.

Replace every number x with the number $x + x'\varepsilon$, where x' is a real number, but ε is an abstract number with the property $\varepsilon^2 = 0$ (an infinitesimal; see *Smooth infinitesimal analysis*). Using only this, regular arithmetic gives

$$\begin{aligned}
(x + x'\varepsilon) + (y + y'\varepsilon) &= x + y + (x' + y')\varepsilon \\
(x + x'\varepsilon) - (y + y'\varepsilon) &= x - y + (x' - y')\varepsilon \\
(x + x'\varepsilon) \cdot (y + y'\varepsilon) &= xy + xy'\varepsilon + yx'\varepsilon + x'y'\varepsilon^2 = xy + (xy' + yx')\varepsilon \\
(x + x'\varepsilon)/(y + y'\varepsilon) &= (x/y + x'\varepsilon/y)/(1 + y'\varepsilon/y) = (x/y + x'\varepsilon/y) \cdot (1 - y'\varepsilon/y) = x/y
\end{aligned}$$

using $(1 + y'\varepsilon/y) \cdot (1 - y'\varepsilon/y) = 1$.

Now, polynomials can be calculated in this augmented arithmetic. If $P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$, then

$$\begin{aligned}
P(x + x'\varepsilon) &= p_0 + p_1(x + x'\varepsilon) + \cdots + p_n(x + x'\varepsilon)^n \\
&= p_0 + p_1 x + \cdots + p_n x^n + p_1 x'\varepsilon + 2p_2 x x'\varepsilon + \cdots + np_n x^{n-1} x'\varepsilon \\
&= P(x) + P^{(1)}(x) x'\varepsilon
\end{aligned}$$

where $P^{(1)}$ denotes the derivative of P with respect to its first argument, and x', called a *seed*, can be chosen arbitrarily.

The new arithmetic consists of ordered pairs, elements written $\langle x, x' \rangle$, with ordinary arithmetics on the first component, and first order differentiation arithmetic on the second component, as described above. Extending the above results on polynomials to analytic functions gives a list of the basic arithmetic and some standard functions for the new arithmetic:

$$\begin{aligned}
\left\langle u,u'\right\rangle + \left\langle v,v'\right\rangle &= \left\langle u+v, u'+v'\right\rangle \\
\left\langle u,u'\right\rangle - \left\langle v,v'\right\rangle &= \left\langle u-v, u'-v'\right\rangle \\
\left\langle u,u'\right\rangle * \left\langle v,v'\right\rangle &= \left\langle uv, u'v+uv'\right\rangle \\
\left\langle u,u'\right\rangle / \left\langle v,v'\right\rangle &= \left\langle \frac{u}{v}, \frac{u'v-uv'}{v^2}\right\rangle \quad (v\neq 0) \\
\sin\left\langle u,u'\right\rangle &= \left\langle \sin(u), u'\cos(u)\right\rangle \\
\cos\left\langle u,u'\right\rangle &= \left\langle \cos(u), -u'\sin(u)\right\rangle \\
\exp\left\langle u,u'\right\rangle &= \left\langle \exp u, u'\exp u\right\rangle \\
\log\left\langle u,u'\right\rangle &= \left\langle \log(u), u'/u\right\rangle \quad (u>0) \\
\left\langle u,u'\right\rangle^k &= \left\langle u^k, u'ku^{k-1}\right\rangle \quad (u\neq 0) \\
\left|\left\langle u,u'\right\rangle\right| &= \left\langle \left|u\right|, u'\operatorname{sign} u\right\rangle \quad (u\neq 0)
\end{aligned}$$

and in general for the primitive function g,

$$g(\langle u,u'\rangle,\langle v,v'\rangle) = \langle g(u,v), g_u(u,v)u' + g_v(u,v)v'\rangle$$

where g_u and g_v are the derivatives of g with respect to its first and second arguments, respectively.

When a binary basic arithmetic operation is applied to mixed arguments—the pair $\langle u, u'\rangle$ and the real number c—the real number is first lifted to $\langle c, 0\rangle$. The derivative of a function $f : \mathbb{R} \to \mathbb{R}$ at the point x_0 is now found by calculating $f(\langle x_0, 1\rangle)$ using the above arithmetic, which gives $\langle f(x_0), f'(x_0)\rangle$ as the result.

Implementation

An example implementation based on the dual number approach follows.

Pseudo Code

```
Dual plus(Dual A, Dual B) {
  return {
    realPartOf(A) + realPartOf(B),
    infinitesimalPartOf(A) + infinitesimalPartOf(B)
  };
}
Dual minus(Dual A, Dual B) {
  return {
    realPartOf(A) - realPartOf(B),
    infinitesimalPartOf(A) - infinitesimalPartOf(B)
  };
}
Dual multiply(Dual A, Dual B) {
  return {
    realPartOf(A) * realPartOf(B),
    realPartOf(B) * infinitesimalPartOf(A) + realPartOf(A) * infinitesimalPartOf(B)
  };
}
X = {x, 0};
Y = {y, 0};
Epsilon = {0, 1};
xPartial = infinitesimalPartOf(f(X + Epsilon, Y));
yPartial = infinitesimalPartOf(f(X, Y + Epsilon));
```

Python

```python
class Dual:
    def __init__(self, realPart, infinitesimalPart=0):
        self.realPart = realPart
        self.infinitesimalPart = infinitesimalPart

    def __add__(self, other):
        return Dual(
            self.realPart + other.realPart,
            self.infinitesimalPart + other.infinitesimalPart
        )

    def __mul__(self, other):
        return Dual(
            self.realPart * other.realPart,
            other.realPart * self.infinitesimalPart + self.realPart * other.infinitesimalPart
        )

# Example: Finding the partials of z = x * (x + y) + y * y at (x, y) = (2, 3)
def f(x, y):
    return x * (x + y) + y * y
x = Dual(2)
y = Dual(3)
epsilon = Dual(0, 1)
a = f(x + epsilon, y)
b = f(x, y + epsilon)
print("∂z/∂x =", a.infinitesimalPart) # Output: ∂z/∂x = 7
print("∂z/∂y =", b.infinitesimalPart) # Output: ∂z/∂y = 8
```

C++

```cpp
#include <iostream>
struct Dual {
   float realPart, infinitesimalPart;
   Dual(float realPart, float infinitesimalPart=0): realPart(realPart), infinitesimalPart(infinitesimalPart) {}
   Dual operator+(Dual other) {
      return Dual(
         realPart + other.realPart,
         infinitesimalPart + other.infinitesimalPart
      );
   }
   Dual operator*(Dual other) {
      return Dual(
         realPart * other.realPart,
         other.realPart * infinitesimalPart + realPart * other.infinitesimalPart
      );
   }
};
// Example: Finding the partials of z = x * (x + y) + y * y at (x, y) = (2, 3)
Dual f(Dual x, Dual y) { return x * (x + y) + y * y; }
int main () {
   Dual x = Dual(2);
   Dual y = Dual(3);
   Dual epsilon = Dual(0, 1);
   Dual a = f(x + epsilon, y);
   Dual b = f(x, y + epsilon);
```

```
    std::cout << "∂z/∂x = " << a.infinitesimalPart << ", "
              << "∂z/∂y = " << b.infinitesimalPart << std::endl;
    // Output: ∂z/∂x = 7, ∂z/∂y = 8
    return 0;
 }
```

Vector arguments and functions

Multivariate functions can be handled with the same efficiency and mechanisms as univariate functions by adopting a directional derivative operator. That is, if it is sufficient to compute $y' = \nabla f(x) \cdot x'$, the directional derivative $y' \in \mathbb{R}^m$ of $f : \mathbb{R}^n \to \mathbb{R}^m$ at $x \in \mathbb{R}^n$ in the direction $x' \in \mathbb{R}^n$ may be calculated as $(\langle y_1, y'_1 \rangle, \ldots, \langle y_m, y'_m \rangle) = f(\langle x_1, x'_1 \rangle, \ldots, \langle x_n, x'_n \rangle)$ using the same arithmetic as above. If all the elements of ∇f are desired, then n function evaluations are required. Note that in many optimization applications, the directional derivative is indeed sufficient.

High order and many variables

The above arithmetic can be generalized to calculate second order and higher derivatives of multivariate functions. However, the arithmetic rules quickly grow complicated: complexity is quadratic in the highest derivative degree. Instead, truncated Taylor polynomial algebra can be used. The resulting arithmetic, defined on generalized dual numbers, allows efficient computation using functions as if they were a data type. Once the Taylor polynomial of a function is known, the derivatives are easily extracted.

Implementation

Forward-mode AD is implemented by a nonstandard interpretation of the program in which real numbers are replaced by dual numbers, constants are lifted to dual numbers with a zero epsilon coefficient, and the numeric primitives are lifted to operate on dual numbers. This nonstandard interpretation is generally implemented using one of two strategies: *source code transformation* or *operator overloading*.

Source code transformation (SCT)

The source code for a function is replaced by an automatically generated source code that includes statements for calculating the derivatives interleaved with the original instructions.

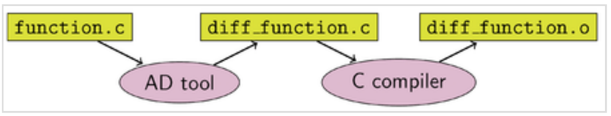


Figure 4: Example of how source code transformation could work

Source code transformation can be implemented for all programming languages, and it is also easier for the compiler to do compile time optimizations. However, the implementation of the AD tool itself is more difficult and the build system is more complex.

Operator overloading (OO)

Operator overloading is a possibility for source code written in a language supporting it. Objects for real numbers and elementary mathematical operations must be overloaded to cater for the augmented arithmetic depicted above. This requires no change in the form or sequence of operations in the original source code for the function to be differentiated, but often requires changes in basic data types for numbers and vectors to support overloading and often also involves the insertion of special flagging operations. Due to the inherent operator overloading overhead on each loop, this approach usually demonstrates weaker speed performance.

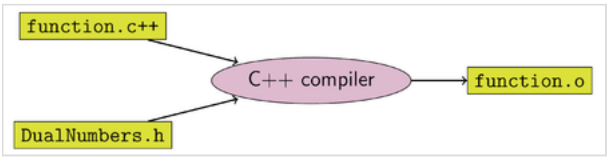


Figure 5: Example of how operator overloading could work

Operator overloading and Source Code Transformation

Overloaded Operators can be used to extract the valuation graph, followed by automatic generation of the AD-version of the primal function at run-time. Unlike the classic OO AAD, such AD-function does not change from one iteration to the next one. Hence there is any OO or tape interpretation run-time overhead per Xi sample.

With the AD-function being generated at runtime, it can be optimised to take into account the current state of the program and precompute certain values. In addition, it can be generated in a way to consistently utilize native CPU vectorization to process 4(8)-double chunks of user data (AVX2\AVX512 speed up x4-x8). With multithreading added into account, such approach can lead to a final acceleration of order 8 × #Cores compared to the traditional AAD tools. A reference implementation is available on GitHub.[11]

See also

- Differentiable programming

Notes

a. In terms of weight matrices, the adjoint is the transpose. Addition is the covector $[1\cdots 1]$, since $[1\cdots 1]\begin{bmatrix}x_1\\ \vdots \\ x_n\end{bmatrix} = x_1+\cdots+x_n$, and fanout is the vector $\begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}$, since $\begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}[x] = \begin{bmatrix}x\\ \vdots \\ x\end{bmatrix}$.

References

1. Neidinger, Richard D. (2010). "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming" (http://academics.davidson.edu/math/neidinger/SIAMRev74362.pdf) (PDF). *SIAM Review*. **52** (3): 545–563. CiteSeerX 10.1.1.362.6580 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.362.6580). doi:10.1137/080743627 (https://doi.org/10.1137%2F080743627). S2CID 17134969 (https://api.semanticscholar.org/CorpusID:17134969).
2. Baydin, Atilim Gunes; Pearlmutter, Barak; Radul, Alexey Andreyevich; Siskind, Jeffrey (2018). "Automatic differentiation in machine learning: a survey" (http://jmlr.org/papers/v18/17-468.html). *Journal of Machine Learning Research*. **18**: 1–43.
3. R.E. Wengert (1964). "A simple automatic derivative evaluation program" (https://doi.org/10.1145%2F355586.364791). *Comm. ACM*. **7** (8): 463–464. doi:10.1145/355586.364791 (https://doi.org/10.1145%2F355586.364791). S2CID 24039274 (https://api.semanticscholar.org/CorpusID:24039274).

4. Griewank, Andreas (2012). "Who Invented the Reverse Mode of Differentiation?" (https://ftp.gwdg.de/pub/misc/EMIS/journals/DMJDMV/vol-ismp/52_griewank-andreas-b.pdf) (PDF). *Optimization Stories, Documenta Matematica*. Extra Volume ISMP: 389–400. doi:10.4171/dms/6/38 (https://doi.org/10.4171%2Fdms%2F6%2F38). ISBN 978-3-936609-58-5.
5. Linnainmaa, Seppo (1976). "Taylor Expansion of the Accumulated Rounding Error". *BIT Numerical Mathematics*. **16** (2): 146–160. doi:10.1007/BF01931367 (https://doi.org/10.1007%2FBF01931367). S2CID 122357351 (https://api.semanticscholar.org/CorpusID:122357351).
6. Maximilian E. Schüle, Maximilian Springer, Alfons Kemper, Thomas Neumann (2022). "LLVM code optimisation for automatic differentiation". *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning*. pp. 1–4. doi:10.1145/3533028.3533302 (https://doi.org/10.1145%2F3533028.3533302). ISBN 9781450393751. S2CID 248853034 (https://api.semanticscholar.org/CorpusID:248853034).
7. Bartholomew-Biggs, Michael; Brown, Steven; Christianson, Bruce; Dixon, Laurence (2000). "Automatic differentiation of algorithms". *Journal of Computational and Applied Mathematics*. **124** (1–2): 171–190. Bibcode:2000JCoAM.124..171B (https://ui.adsabs.harvard.edu/abs/2000JCoAM.124..171B). doi:10.1016/S0377-0427(00)00422-2 (https://doi.org/10.1016%2FS0377-0427%2800%2900422-2). hdl:2299/3010 (https://hdl.handle.net/2299%2F3010).
8. Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, Stephan Günnemann (2021). "In-Database Machine Learning with SQL on GPUs". *33rd International Conference on Scientific and Statistical Database Management*. pp. 25–36. doi:10.1145/3468791.3468840 (https://doi.org/10.1145%2F3468791.3468840). ISBN 9781450384131. S2CID 235386969 (https://api.semanticscholar.org/CorpusID:235386969).
9. Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, Stephan Günnemann (2022). "Recursive SQL and GPU-support for in-database machine learning" (https://doi.org/10.1007%2Fs10619-022-07417-7). *Distributed and Parallel Databases*. **40** (2–3): 205–259. doi:10.1007/s10619-022-07417-7 (https://doi.org/10.1007%2Fs10619-022-07417-7). S2CID 250412395 (https://api.semanticscholar.org/CorpusID:250412395).
10. Naumann, Uwe (April 2008). "Optimal Jacobian accumulation is NP-complete". *Mathematical Programming*. **112** (2): 427–441. CiteSeerX 10.1.1.320.5665 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.320.5665). doi:10.1007/s10107-006-0042-z (https://doi.org/10.1007%2Fs10107-006-0042-z). S2CID 30219572 (https://api.semanticscholar.org/CorpusID:30219572).
11. "AADC Prototype Library" (https://github.com/matlogica/aadc-prototype). June 22, 2022 – via GitHub.

Further reading

- Rall, Louis B. (1981). *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science. Vol. 120. Springer. ISBN 978-3-540-10861-0.
- Griewank, Andreas; Walther, Andrea (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (https://epubs.siam.org/doi/book/10.1137/1.9780898717761). Other Titles in Applied Mathematics. Vol. 105 (2nd ed.). SIAM. doi:10.1137/1.9780898717761 (https://doi.org/10.1137%2F1.9780898717761). ISBN 978-0-89871-659-7.
- Neidinger, Richard (2010). "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming" (http://academics.davidson.edu/math/neidinger/SIAMR

ev74362.pdf) (PDF). *SIAM Review*. **52** (3): 545–563. CiteSeerX 10.1.1.362.6580 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.362.6580). doi:10.1137/080743627 (https://doi.org/10.1137%2F080743627). S2CID 17134969 (https://api.semanticscholar.org/CorpusID:17134969). Retrieved 2013-03-15.

- Naumann, Uwe (2012). *The Art of Differentiating Computer Programs*. Software-Environments-tools. SIAM. ISBN 978-1-611972-06-1.
- Henrard, Marc (2017). *Algorithmic Differentiation in Finance Explained*. Financial Engineering Explained. Palgrave Macmillan. ISBN 978-3-319-53978-2.

External links

- www.autodiff.org (http://www.autodiff.org/), An "entry site to everything you want to know about automatic differentiation"
- Automatic Differentiation of Parallel OpenMP Programs (http://www.autodiff.org/?module=Applications&application=HC1)
- Automatic Differentiation, C++ Templates and Photogrammetry (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.7749&rep=rep1&type=pdf)
- Automatic Differentiation, Operator Overloading Approach (https://web.archive.org/web/20070927120356/http://www.vivlabs.com/subpage_ad.php)
- Compute analytic derivatives of any Fortran77, Fortran95, or C program through a web-based interface (http://tapenade.inria.fr:8080/tapenade/index.jsp) Automatic Differentiation of Fortran programs
- Description and example code for forward Automatic Differentiation in Scala (http://www.win-vector.com/dfiles/AutomaticDifferentiationWithScala.pdf) Archived (https://web.archive.org/web/20160803214549/http://www.win-vector.com/dfiles/AutomaticDifferentiationWithScala.pdf) 2016-08-03 at the Wayback Machine
- finmath-lib stochastic automatic differentiation (https://www.finmath.net/finmath-lib/concepts/stochasticautomaticdifferentiation/), Automatic differentiation for random variables (Java implementation of the stochastic automatic differentiation).
- Adjoint Algorithmic Differentiation: Calibration and Implicit Function Theorem (https://web.archive.org/web/20140423121504/http://developers.opengamma.com/quantitative-research/Adjoint-Algorithmic-Differentiation-OpenGamma.pdf)
- C++ Template-based automatic differentiation article (http://www.quantandfinancial.com/2017/02/automatic-differentiation-templated.html) and implementation (https://github.com/omartinsky/QuantAndFinancial/tree/master/autodiff_templated)
- Tangent (https://github.com/google/tangent) Source-to-Source Debuggable Derivatives (https://research.googleblog.com/2017/11/tangent-source-to-source-debuggable.html)
- Exact First- and Second-Order Greeks by Algorithmic Differentiation (http://www.nag.co.uk/doc/techrep/pdf/tr5_10.pdf)
- Adjoint Algorithmic Differentiation of a GPU Accelerated Application (http://www.nag.co.uk/Market/articles/adjoint-algorithmic-differentiation-of-gpu-accelerated-app.pdf)
- Adjoint Methods in Computational Finance Software Tool Support for Algorithmic Differentiationop (http://www.nag.co.uk/Market/seminars/Uwe_AD_Slides_July13.pdf)
- More than a Thousand Fold Speed Up for xVA Pricing Calculations with Intel Xeon Scalable Processors (https://www.intel.com/content/dam/www/public/us/en/document

s/white-papers/xva-pricing-application-financial-services-white-papers.pdf)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Automatic_differentiation&oldid=1183706049"

-