# Report for Project of SI211: Numerical Analysis (2020, Fall) about Algorithmic Differentiation

**xxx xxxxxx, xxxxxx**

*School of Information Science and Technology, ShanghaiTech University (e-mail: xxxx@shanghaitech.edu.cn).*

**Abstract:** In mathematics and computer algebra, algorithmic differentiation (AD), also called automatic differentiation, is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. In this report, the principles of algorithmic differentiation with forward mode will be briefly explained, as well as the backward mode. The details of implementing the framework of algorithmic differentiation will be included in this report as well. Also, the comparisons of the accuracy and running-time for algorithmic differentiation with different mode and numerical differentiation using finite differences over a special function will be illustrated in this report.

*Keywords:* automatic differentiation, numerical analysis, computational differentiation, algorithmic differentiation, evaluating derivatives

## 1. INTRODUCTION

Applications of derivatives, which measure the change in one quantity relative to a change in another quantity, are ubiquitous in applied mathematics. Most state-of-the-art statistical inference algorithms are based on first- or higher-order derivatives. Although 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. Algorithmic differentiation is a general technique for converting a function computing values to one that also computes derivatives. The process of algorithmic differentiation has the same order of complexity as the original function. This can be done in full generality with both efficiency and high arithmetic precision.

Although algorithmic differentiation is an efficient method to obtain the derivative, there are different ways, namely modes to implement algorithmic differentiation such as forward-mode, backward-mode, also called reverse mode and mixed-modeCarpenter (2020). The mixed-mode is often nesting reverse-mode automatic differentiation within forward mode to provides efficient calculation of Hessians, which will not be discussed in detail in this report. Also, the derivatives can be computed numerically using finite differences, which is the limitation of the very small differences for dependent variables divided by the small differences for independent variables.

### *1.1 Finite Differences*

There are many ways that derivatives can be computed using finite differences. The simplest way directly follows the definition of derivatives. Suppose $f : \mathbb{R} \to \mathbb{R}$ is a smooth function, then:

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

when $h$ is small sufficiently, we can approximate $f^{'}(x)$ by

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

There are also many variations of finite difference, the five-point differentiation is one of them with relatively high precision. The five-point differentiation formula is as follows:

$$\begin{aligned} \Delta f = &-25f(x_0) + 48f(x_0 + h) \\ &- 36f(x_0 + 2h) + 16f(x_0 + 3h) - 3f(x_0 + 4h) \\ f'(x_0) &\approx \frac{\Delta f}{12h} \end{aligned}$$

The mathematical approximation error of this formula is $\mathbf{O}(h^4)$. Since the numerical error is approximately:

$$\mathbf{O}(\frac{\mathrm{eps}}{h})$$

Therefore, we can choose $h$ such that

$$h^{\star} \approx \arg\min_{h}(h^4 + \frac{\mathrm{eps}}{h})$$

Hence, $h^{\star} = \sqrt[5]{\frac{\mathrm{eps}}{4}}$, which will be used in the implemented finite differences methodBurden and Faires (2010).

### *1.2 Algorithmic Differences*

Fundamental to AD is the decomposition of differentials provided by the chain rule. Forward accumulation specifies that one traverses the chain rule from inside to outside, while reverse accumulation has the traversal from outside to inside. Take the function $y = f(g(h(x)))$, i.e. $f = g \circ h$ as example. Let

$$w_0 = x$$
$$w_1 = h(w_0)$$
$$w_2 = g(w_1)$$
$$y = f(w_2)$$

In forward mode, we compute the derivative of $y$ over $x$ in the follows sequence:

$$\frac{\mathrm{d}y}{\mathrm{d}}x = \frac{\mathrm{d}y}{\mathrm{d}w_2}(\frac{\mathrm{d}w_2}{\mathrm{d}w_1}(\frac{\mathrm{d}w_1}{\mathrm{w}_0}))$$

While in the backward mode, we compute the derivative of $y$ over $x$ in reverse sequence as:

$$\frac{\mathrm{d}y}{\mathrm{d}}x = ((\frac{\mathrm{d}y}{\mathrm{d}w_2})\frac{\mathrm{d}w_2}{\mathrm{d}w_1})\frac{\mathrm{d}w_1}{\mathrm{w}_0}.$$

*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. Given a smooth function $f : \mathbb{R}^N \to \mathbb{R}^M$, its derivative in the direction $v \in \mathbb{R}^N$ at a point $x \in \mathbb{R}^N$ is $\nabla f(x) \cdot v$, which can be computed with forward mode algorithmic difference. There are also direction derivative for objective vector. For above function $f$, its derivative in the direction $v \in \mathbb{R}^M$ at $y \in \mathbb{R}^M$ is $v^T \cdot \nabla f(x)$, which can be computed with backward mode algorithmic differenceMagnus and Neudecker (2019)Neidinger (2010).

## 2. CODE DESIGN

To implement the algorithmic difference, we need to store some extra information while computing the objective value. One way to implement it is operator overloading, which can help us store some necessary information when obtain the target value. These operations could be done in a procedural programming style, but Object-Oriented Programming(OOP) is much more natural. Hence, we will use OOP with operator overloading to implement the algorithmic differenceNeidinger (2010)Baydin et al. (2017).

### *2.1 Forward Mode*

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. Therefore, the object can be defined as:

```
class Ops:
  def __init__(self, value, grad, op):
      self.value = value
      self.grad = grad
      self.op = op
```

For example, negation is defined for dual numbers by

$$-  < u, \mathrm{d}u >=< -u, -\mathrm{d}u >$$

where the first value represent the target value, the last value represent the present derivative.
For sums,

$$< u, \mathrm{d}u > + < v, \mathrm{d}v >=< u + v, \mathrm{d}u + \mathrm{d}v >$$

For differences,

$$< u, \mathrm{d}u > - < v, \mathrm{d}v >=< u - v, \mathrm{d}u - \mathrm{d}v >$$

Also, we can use sum plus negation to implement difference.
For products,

$$< u, \mathrm{d}u > \cdot < v, \mathrm{d}v >=< u \cdot v, \mathrm{d}u \cdot v + u \cdot \mathrm{d}v > .$$

For quotients,

$$< u, \mathrm{d}u > / < v, \mathrm{d}v >=< u/v, (\mathrm{d}u \cdot v - \mathrm{d}v \cdot u)/v^2.$$

For cosine,

$$\cos(< u, \mathrm{d}u >) =< \cos u, -\sin u \cdot \mathrm{d}u >$$

For sin,

$$\sin(< u, \mathrm{d}u >) =< \sin u, \cos u \cdot \mathrm{d}u >$$

According to the above formula, we can finish the corresponds computing process. Take the negation for example, we can define the overloading operations as:

```
def _neg(var):
    return Ops(-var.value, -var.grad,
    op="neg")
```

At the same time, we have to consider the constant as operand, in which case, the derivative will not change and the value will update accordingly. Take the summation for example:

```
def _add(a, b):
  if isinstance(b, Ops):
      return Ops(a.value + b.value,
      a.grad + b.grad, op="add")
  else:
      # add constant
      return Ops(a.value + b,
      a.grad, op="add")
```

Also, for each variable, we have to initialize them into the designed object. We construct a subclass of `Ops` to represent the initialization.

```
class Var(Ops):
  def __init__(self, value, grad=0.0):
      super(Var, self).__init__(value=value,
      grad=grad, op="var")
```

According to the above procedure, we can obtain the derivative while computing the corresponds objective value over given directions $d$. Take $N = 2, M = 1$ for example. Firstly, we initialize the variables using:

```
x = [Var(x[0], d[0]), Var(x[1], d[1])]
```

Then, compute the objective value using function $f$:

```
y = f(x)
```

Finally, the derivative of $f$ over $x$ in the direction $d$ can be got as `y.grad` and `y.value` represent the objective value.

*Jacobians* Algorithmic difference with forward mode can be used to compute Jacobians, which is the $M \times N$ matrix of all first-order derivative of the function $f : \mathbb{R}^N \to \mathbb{R}^M$. One direct method is to take $N$ passes of forward mode with each direction $d = e_i \in \mathbb{R}^N$, i.e. $d_i = 1$ and $d_j = 0$ for all $j \neq i$ for $i = 1, 2, \cdots, N$. Each iteration $i$ will return the derivatives of $\mathbf{y}$ over $x_i$, where $\mathbf{y} = f(\mathbf{x})$.

*2.2 Backward Mode*

Compared to the forward mode, we compute the derivative of $\mathbf{y}$ over $\mathbf{x}$ in reverse sequence when taking algorithmic difference in backward mode for the function $f : \mathbb{R}^N \to \mathbb{R}^M$ with $\mathbf{y} = f(\mathbf{x})$. Therefore, the computation graph needs to be stored to transport the gradient from the root nodes, i.e. $\mathbf{y}$ to the leaf nodes, i.e. $\mathbf{x}$. Hence, we need to store the children nodes for each node. The object can be defined as:

```
class Ops:
  def __init__(self, value, grad_back,
   children, op):
      self.value = value
      self.op = op
      self.grad_back = grad_back
      self.children = children
```

The `grad_back` represent the back gradient from the root node. Given the parent node grad $w_i$ from the gradient, we can compute the child node grad with $\frac{\partial w_i}{\partial w_{i-1}}$, also called as partial grad or local grad, multiplying by the parent node grad, $\frac{\partial y}{\partial w_i}$, which is stored in `grad_back`. Hence, besides each child node, we also need the local grad of each child node. Take the negation for example,

```
def _neg(var):
  return Ops(value=-var.value,
  children=[(var, -1)], op="neg"
```

where `var` represents the child node, and `-1` represents the local grad.
The local grad for different operators can refer to Table 1 Similarly, we have to consider the constant as operand.

Table 1. Local grad table

| Operator | $\partial a$ | $\partial b$ |
|---|---|---|
| $-a$ | $-1$ | |
| $a+b$ | $1$ | $1$ |
| $a-b$ | $1$ | $-1$ |
| $a \cdot b$ | $b$ | $a$ |
| $a/b$ | $1/b$ | $-a/b^2$ |
| $\cos(a)$ | $-\sin(a)$ | |
| $\sin(a)$ | $\cos(a)$ | |

Take the summation for example:

```
def _mul(a, b):
if isinstance(b, Ops):
    return Ops(value=a.value * b.value,
    children=[(a, b.value), (b, a.value)],
    op="mul")
else:
    return Ops(value=a.value * b,
    children=[(a, b)], op="mul"
```

With each node and its children nodes, we can use BFS traversal to compute the derivative for each leaf node $x_i$. The traversal can follow Algorithm 1.
However, the Algorithm 1 has a problem. The grad from the root node will be discarded when the child node has been visited. One way to solve it is to give up the pruning strategy, i.e. for each node, all of its children will be added into the queue. However, it will traverse some node in exponential order, the time complexity would be $\mathcal{O}(2^h)$ where $h$ is height of the computation graph. Another method is to get the topological sequence such that each node would not be needed by the later nodes. Thus, the backward procedure is shown as Algorithm 2.
Since the computation graph is a directed acyclic graph(DAG), we can use Kahn's algorithm to obtain the topological sequence as Algorithm 3.

**Algorithm 1:** Breadth First Search(BFS) in backward-mode

**Input :** root node $y$
**Output:** leaf nodes with grad

```
queue = []
visited = []
res= []
queue.push(y)
while queue is not empty do
    v = queue.pop(0)
    visited ← visited ∪ {v}
    if v is leaf node then
        res ← res ∪ v
    end
    if v has child then
        foreach u, d in v.children do
            u.grad_back += v.grad_back * d
        end
        if u is not in visited then
            queue ← queue ∪ {u}
        end
    end
end
return res
```

**Algorithm 2:** backward-mode algorithm

**Input :** root node $y$
**Output:** leaf nodes with grad

```
res = []
sequence = topological_sort(y)
foreach node v in sequence do
    if v is leaf node then
        res ← res ∪ v
    end
    if v has child then
        foreach u, d in v.children do
            u.grad_back += v.grad_back * d
        end
    end
end
return res
```

As for the in degree dictionary, we can obtain it with simple variant of BFS as Algorithm 1.

*Jacobians* Algorithmic difference with backward mode can also be used to compute Jacobians. One direct method is to take $M$ passes of backward mode with each direction $d = e_i \in \mathbb{R}^N$, i.e. $d_i = 1$ and $d_j = 0$ for all $j \neq i$ for $i = 1, 2, \cdots, M$ over function $f = \mathbb{R}^N \leftarrow \mathbb{R}^M$. Each iteration $i$ will return the derivatives of $y_i$ over $\mathbf{x}$, where $\mathbf{y} = f(\mathbf{x})$.

**Algorithm 3:** Topological Sort Algorithm

**Input :** root node $y$, in_degree $d$
**Output:** topological sequence

```
queue = []
visited = []
res= []
visited ← visited ∪ {y}
queue.push(y)
while queue is not empty do
    v = queue.pop(0)
    res ← res ∪ {v}
    if v has child then
        foreach u, _ in v.children do
            if u is not in visited then
                d[u] -= 1
                if d[u] is 0 then
                    queue ← queue ∪ {u}
                    visited ← visited ∪ {u}
                end
            end
        end
    end
end
return res
```

## 3. USER MANUAL

There are four modules included in this project.

- `ForwardDiff`,it implements the algorithmic difference of forward mode. Two API are provided, one is `ADForward`,which will return a function $g$ to compute the derivative over the function $y = f(x)$ along the direction $d$ about $x$. Another is `Jacobian`,which will return a function to compute the Jacobian matrix over the function $f$.
  Example:

```
from ForwardDiff.base import *
def f(x):
    return [sin(cos(x[0] - x[1]))]
print(ADforward(f)([2, 1], [1, 0]))
print(ADforward(f)([2, 1], [0, 1]))
print(-cos(cos(1)) * sin(1))
print(cos(cos(1)) * sin(1))
```

  The result are:

```
[-0.7216061490634433]
[0.7216061490634433]
-0.7216061490634433
0.7216061490634433
```

- `BackwardDiff`,it implements the algorithmic difference of backward mode. Two API are provided, one is `ADBackward`,which will return a function $g$ to compute the derivative over the function $y = f(x)$ along the direction $d$ about $y$. Another is `Jacobian`,which will return a function to compute the Jacobian matrix over the function $f$.
  Example:

```
from BackwardDiff.base import *
def f(x):
    return [sin(cos(x[0] - x[1]))]
print(ADbackward(f)([2, 1], [1]))
print(-cos(cos(1))*sin(1))
print(cos(cos(1))*sin(1))
```

  The result are:

```
[-0.7216061490634433, 0.7216061490634433]
-0.7216061490634433
0.7216061490634433
```

- `ADDiff`,it implements the algorithmic difference of backward mode and forward mode to compute the results, therefore, the object has been designed to fit both two modes. This module is intended to compare the results of the two modes, while the time-consuming will be higher than that use one of the above since some operations are unnecessary when only one kind of mode is run. Four API are provided, one is `ADforward`,the second is `JacobianForward`,the last two are `ADbackward`and `JacobianBackward`,`which`will return the accordingly function of forward mode and backward mode respectively.
- `Numerical_diff`,it implements the numerical differentiation using finite differences with five-point method. The `five_point_diff`will return a function which computes the derivative of $y$ over $x$ for function $y = f(x)$ with given $x$ and $h$.
  Example:

```
from Numerical_diff.base import *
def f(x):
    return [sin(cos(x[0] - x[1]))]
print(five_point_diff(f)([2, 1]))
print(-cos(cos(1)) * sin(1))
print(cos(cos(1)) * sin(1))
```

  The result are:

```
[[-0.72160615  0.72160615]]
-0.7216061490634433
0.7216061490634433
```

All the code has been open source. To learn about more details, please refer to `https://github.com/xrrain/Auto_Diff`.

## 4. NUMERICAL RESULTS

Here we use the $f$ as the given function to illustrate the results of algorithmic difference with different modes and numerical differentiation using finite differences.

```
function f(x)
  a = 1;
  b = 1;
  for i=1:length(x)
    y = 0.3*sin(a)+0.4*b;
    z = 0.1*a+0.3*cos(b)+x[i];
    a = y;
    b = z;
  end
  return [a;b];
end
```

### 4.1 Analysis

To compare the accuracy of the three different method, the derivative can be derived by the following formula.
Let $a^{'}_{i,j}$ represent the gradient of the i-th iteration of $a$ over $x_j$, the same as $b^{'}_{i,j}$, $y^{'}_{i,j}$ and $z^{'}_{i,j}$. From the definition of $f$, we know that:

$$\begin{aligned} y^{'}_{i,j} &= 0.3 * \cos(a_{i-1}) * a^{'}_{i-1,j} + 0.4 * b^{'}_{i-1,j} \\ y_i &= 0.3 * \sin(a_{i-1}) + 0.4 * b_{i-1} \\ z^{'}_{i,j} &= 0.1 * a^{'}_{i-1,j} - 0.3 * \sin(b_{i-1}) * b^{'}_{i-1,j} + x^{'}_{i,j} \\ z_i &= 0.1 * a_{i-1} + 0.3 * \cos(b_{i-1}) + x_i \\ a^{'}_{i,j} &= y^{'}_{i,j} \\ a_i &= y_i \\ b^{'}_{i,j} &= z^{'}_{i,j} \\ b_i &= z_i \end{aligned}$$

when $x = [1, 1, \cdots, 1] \in \mathbb{R}^2020$, we have,

$$a_0 = b_0 = 1, a^{'}_{0,j} = b^{'}_{0,j} = 0$$

when $i \neq j$,

$$x^{'}_{i,j} = 0$$

when $i = j$,

$$x^{'}_{i,j} = 1$$

Therefore, when $i < j$,

$$a^{'}_{i,j} = b^{'}_{i,j} = 0,$$

when $i = j$,

$$a^{'}_{i,j} = 0, b^{'}_{i,j} = 1.$$

when $i = j + 1$,

$$a^{'}_{i,j} = 0.4, b^{'}_{i,j} = -0.3 * \sin(b_{i-1}).$$

Assume $a^{'}_{i,1:i}$ and $b^{'}_{i,1:i}$ have been obtained(the procedure make it satisfied),

$$\begin{aligned} a^{'}_{i+1,1:i-1} &= 0.3 * \cos(a_i) * a^{'}_{i,1:i-1} + 0.4 * b^{'}_{i,1:i-1} \\ b^{'}_{i+1,1:i} &= 0.1 * a^{'}_{i,1:i} - 0.3 * \sin(b_i) * b^{'}_{i,1:i} \end{aligned}$$

Hence, we can use the following script to compute the derivative:

```
function df(x)
  n = length(x)
  a = 1;
  b = 1;
  da = zeros(n)
  db = zeros(n)
  db[1] = 1
  for i=2:length(x)
    db[i] = 1
    da[1:i-1]=0.3*cos(a)*da[1:i-1]+0.4*db[1:i-1]
    db[1:i-1]=0.1*da[1:i-1]-0.3*sin(b)*db[1:i-1]
    y = 0.3*sin(a)+0.4*b;
    z = 0.1*a+0.3*cos(b)+x[i];
    a = y;
    b = z;
  end
  return [da;db];
end
```

In the following part, the function $gf$ will be acted as baseline to compare the accuracy and run-time of the above three methods.

### 4.2 Metrics

To compare the performance of such three methods, we will compare the run-times and the relative accuracy to compute the Jacobian of given function with $n = 2020$. The numerical difference uses five-point method, the forward mode algorithmic difference uses 2020 seeds and the back-ward mode algorithmic difference use 2 seeds.

### 4.3 Results

For time-cost, we run each method over the given function 10 times and report the average consuming time. The results are shown in Table 2. Note that since for the mix-mode `ADDiff` the time-consuming will be higher than that use one of the above since some operations are unnecessary when only one kind of mode is run, we use it compare the accuracy, while using the corresponding module to obtain the consuming time.

From Table 2, we can find *df* is the fastest method to obtain the results, since we only go pass the function $f : \mathbb{R}^N \to \mathbb{R}^M$ once and there are no extra operators needed. Then, the time-consuming of ADbackward is the ranked second to *df*, since it only goes pass the function $f : \mathbb{R}^N \to \mathbb{R}^M$ with $M$ times, where $M = 2$. The results of time-consuming about ADforward and Numerical diff are similar, which is higher than the other two methods greatly since it will go pass the function $f : \mathbb{R}^N \to \mathbb{R}^M$ with $N$ times, where $N = 2020$.

Table 2. Running Time Comparisons

| method | time(s) |
|---|---|
| *df* | **0.067** |
| ADforward | 63.538 |
| ADbackward | 0.286 |
| Numerical diff | 64.777 |

For accuracy, we report the $l_2$ norm of the difference between results from the above three methods and the result from the *df*. The results are shown in Table 3.

From Table 3, we can find the results of *df*, ADforward and ADbackward are almost the same with each other. And the difference of Numerical diff is relatively larger than the others even though the difference is not very high in absolute respective. From the Section 1, we know that the five-point formula has error $\mathcal{O}(h^4 + \frac{\text{eps}}{h})$, which is $\mathcal{O}(5h^4) = \mathcal{O}(4.95 \times 10^{-13})$ when $h = \sqrt[5]{\frac{\text{eps}}{4}}$, thus the relatively larger difference of numerical difference behaves as expected, while all the error of forward mode algorithmic difference and backward mode algorithmic difference almost comes from numerical error eps.

Table 3. Accuracy Comparisons

| method | $\|\|res - res_{df}\|\|_2^2$ |
|---|---|
| ADforward | $1.561 \times 10^{-17}$ |
| ADbackward | $\mathbf{1.557 \times 10^{-17}}$ |
| Numerical diff | $1.775 \times 10^{-11}$ |

After considering the time-consuming and accuracy, we can find algorithmic differences have higher performance compared with numerical difference. To compute the Jacobian of function $f : \mathbb{R}^N \to \mathbb{R}^M$, the forward mode is more efficient if the output dimension $M$ is larger than the input dimension $N$, otherwise, backward mode is more efficient

in some cases the input dimension $N$ is larger than the output dimension $M$, typically, like machine learning and deep learning.

## 5. CONCLUSION

In this report, we have introduced the basic ideas to compute the derivatives such as forward mode algorithmic difference, backward mode algorithmic and numerical difference. Also, we implement a simple framework to compute the algorithmic difference in forward and backward mode with OOP and operator overloading. The details are discussed in Section 2 including the code design and relative algorithms, which demonstrates the principles and implementing clearly. In Section 4, we discuss the features of algorithmic difference with forward mode and backward mode as well as the numeric difference in the view of time-consuming and accuracy. In addition, we analyze the reason for some phenomenons and give some suggestions about when to use algorithmic difference with different mode.

As for the future work, we consider designing a more general algorithmic difference framework such that it can support more common used operators and their combination rather than the combination of the atom operations `+, -, *, /, sin, cos` as now.

## REFERENCES


Baydin, A.G., Pearlmutter, B.A., Radul, A.A., and Siskind, J.M. (2017). Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1), 5595–5637.

Burden, R.L. and Faires, J.D. (2010). Numerical analysis, brooks.

Carpenter, B. (2020). Automatic differen tiation handbook.

Magnus, J.R. and Neudecker, H. (2019). *Matrix differential calculus with applications in statistics and econometrics*. John Wiley & Sons.

Neidinger, R.D. (2010). Introduction to automatic differentiation and matlab object-oriented programming. *SIAM review*, 52(3), 545–563.