Numerical Analysis Project–Numerical Differentiation, Algorithmic Forward Differentiation and Algorithmic Backward Differentiation

Name: Chengrui Zhang **Student Number: 2019233183**

Shanghaitech University, 393 Middle Huaxia Road, Pudong, Shanghai (e-mail: zhangchr@shanghaitech.edu.cn).

Abstract: Associated with the grown of deep learning, the traditional differentiation method–Numerical Differentiation (ND) is not suitable because of its low accuracy. Recently, a high accuracy differentiation method is proposed in order to solve this problem, called Algorithmic Differentiation (AD). The AD method utilizes computation graph and operation overloading to realize differentiation, and it contains two methods, AD forward and AD backward. In this report, I regenerate the ND, AD forward and AD backward methods respectively and compare their differences.

Keywords: Numerical Differentiation, Algorithmic Differentiation

1. INTRODUCTION

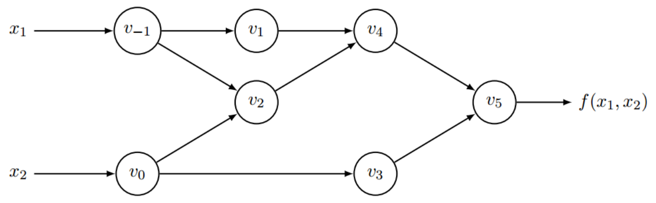
Nowadays, there exists several methods for us to get the derivative of a function f, for example, the manual differentiation, numerical differentiation, symbolic differentiation and algorithmic differentiation. In this four cases, except numerical differentiation whose accuracy relies on the step size, the other three methods are accurate, which means that we can get an accurate derivative by these three methods.
This report focuses on the algorithmic differentiation (AD) method, which utilizes computation graph and operation overloading to realize differentiation and is friendly for programming. This method contains two methods, AD forward and AD backward, which will be introduced in *Sec 1.1* and *Sec 1.2* respectively.

1.1 AD Forward

AD in forward accumulation mode is the conceptually most simple type. Consider the evaluation trace of the function $f(x_1, x_2) = ln(x_1) + x_1x_2 - sin(x_2)$ given on the left-hand side in Table 1 and in graph form in Fig. 2. For computing the derivative of f with respect to x_1, we start by associating with each intermediate variable v_i a derivative: $\dot{v_i} = \frac{\delta v_i}{\delta x_1}$. Applying the chain rule to each elementary operation in the forward primal trace, we generate the corresponding tangent (derivative) trace, given on the right-hand side in Table 1. Evaluating the primals v_i in lockstep with their corresponding tangents $\dot{v_i}$ gives us the required derivative in the final variable $\dot{v_5} = \frac{\delta y}{\delta x_1}$.

1.2 AD Backward

AD in the reverse accumulation mode corresponds to a generalized back-propagation algorithm, in that it propagates derivatives backward from a given output. This is done by complementing each intermediate variable vi with an adjoint $\bar{v_i} = \frac{\delta y_i}{\delta v_i}$ which represents the sensitivity of a considered output y_j with respect to changes in v_i.
In reverse mode AD, derivatives are computed in the second phase of a two-phase process. In the first phase, the original function code is run forward, populating intermediate variables v_i and recording the dependencies in the computational graph through a bookkeeping procedure. In the second phase, derivatives are calculated by propagating adjoints $\bar{v_i}$ in reverse, from the outputs to the inputs.
Returning to the example $y = f(x_1, x_2) = ln(x_1) + x_1x_2 - sin(x_2)$, in Table 2 we see the adjoint statements on the right-hand side, corresponding to each original elementary operation on the left-hand side.



$$\begin{tikzcd}
x_1 \arrow[r] & v_{-1} \arrow[r] \arrow[dr] & v_1 \arrow[r] & v_4 \arrow[dr] & & \\
& & v_2 \arrow[ur] & & v_5 \arrow[r] & f(x_1,x_2) \\
x_2 \arrow[r] & v_0 \arrow[ur] \arrow[rr] & & v_3 \arrow[ur] & &
\end{tikzcd}$$

Fig. 1. Computational graph of the example $f(x_1, x_2) = ln(x_1) + x_1x_2 - sin(x_2)$

Algorithm 1 Numerical Differentiation

```
Require: $x, f, h$
1: for $i$ in $length(x)$ do
2:     Copy the value of $x$ into $x_{tmp}$
3:     Add $h$ into $i^{th}$ number of $x_{tmp}$
4:     Compute $y_{tmp}$ by $x_{tmp}$ with function $f$
5:     for $j$ in $range(2)$ do
6:         Let $[i^{th}, j^{th}]$ of derivative $df$ be the differences between $y_{tmp}$ and $f(x)$ divided by $h$
7:     end for
8: end for
9: return $df$
```

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\boldsymbol{\dot{y}}$	$\boldsymbol{= \dot{v}_5}$	$\boldsymbol{= 5.5}$

Fig. 2. AD forward computational flow of the example $f(x_1, x_2) = ln(x_1) + x_1x_2 - sin(x_2)$

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Reverse Adjoint (Derivative) Trace			
$\boldsymbol{\bar{x}_1}$	$\boldsymbol{= \bar{v}_{-1}}$		$\boldsymbol{= 5.5}$
$\boldsymbol{\bar{x}_2}$	$\boldsymbol{= \bar{v}_0}$		$\boldsymbol{= 1.716}$
$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5$	$= \bar{y}$	$= 1$	

Fig. 3. AD backward computational flow of the example $f(x_1, x_2) = ln(x_1) + x_1x_2 - sin(x_2)$

2. CODE DESIGN

2.1 Numerical differentiation

For numerical differentiation, the idea is quiet simple. Since I need to use the finite differences, first I set the step size be $h = 1e - 8$, which is the best step size. Then I use the algorithm 1 to realize it.

2.2 AD Forward

For AD forward, I use *JULIA* to implement it. The idea is similar to which we learned in the course. First, I create a data structure that contains the real value and the derivative value, then I do the operator overloading to make the operators be satisfied with the derivative operations, finally, I use the created data structure to substitute for the original input, and run the function f to get the result.
In my implementation, I use four methods to realize the AD forward, which are special case (only for this function), general case (with 2020 seeds, for every function), fast case (with 1 seeds) and list case (not use ADV structure but using list to implement).

- Special case: As shown in Fig. 4, I derive the function manually to find a *df* function only suitable for a special function f, then start the process of AD forward shown above to get the result.
- General case: In this case, I simply build a ADV structure which stores the real value and the derivative value, and then, for each dimension of input x, I do the following algorithm 2:

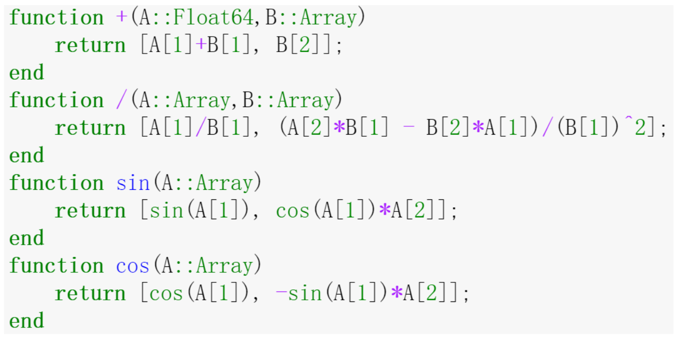
```
function df(x, lambda)
    a = 1; b = 1; da = 0; db = 0;
    for i = 1 : length(x)
        tmp_a = da; tmp_b = db;
        da = 0.3*cos(a)*tmp_a + 0.4 * tmp_b;
        db = 0.1 * tmp_a + 0.3*(-sin(b))*tmp_b + lambda[i];
        y = 0.3*sin(a)+0.4*b;
        z = 0.1*a+0.3*cos(b)+x[i];
        a = y;
        b = z;
    end
    return [a;b;da;db]
end
```

Fig. 4. Special Case

Algorithm 2 AD Forward Differentiation-General Case

```
Require: $x, f, d$
1: Copy the value of $x$ into $X.value$ by the ADV structure, and make all $X.dx = 0$
2: Initialize $y = zeros(length(f(x)), length(x))$
3: for $i$ in $length(x)$ do
4:     Make $X[i].df = d[i]$ and $X.[others].df = 0$
5:     Compute $y_{tmp}$ by $X$ with function $f$
6:     Store $y_{tmp}.df$ into $y[:, i]$
7: end for
8: return $y.sum(axis = 0)$
```

- List case: Since I find that the ADV structure may not be the properest data structure to do AD forward, I use list structure to realize this algorithm, which means that a list has the shape of $(2, length(x))$, the dimension $(1,:)$ stores the real value and the dimension $(2,:)$ store the derivative value. In addition, I re-overloading the operator which is shown in Fig. 5



```julia
function +(A::Float64,B::Array)
    return [A[1]+B[1], B[2]];
end
function /(A::Array,B::Array)
    return [A[1]/B[1], (A[2]*B[1] - B[2]*A[1])/(B[1])^2];
end
function sin(A::Array)
    return [sin(A[1]), cos(A[1])*A[2]];
end
function cos(A::Array)
    return [cos(A[1]), -sin(A[1])*A[2]];
end
```

Fig. 5. List Case

- Fast case: In this case, I modify the general case in order to accelerate the program. As shown in the lecture, I use $d[i]$ instead of 0 in the process of initializing the ADV structure $X[i].df$, and then use this data structure to do only one seed to get the result.

2.3 AD Backward

For AD Backward, I also use *JULIA* to implement it. In this algorithm, I use two methods to realize it. One method is what Boris said in the class, creating a *df* function by a program automatically. In this method, I do operator overloading to let each operator write some codes in the created function *df* serially. The other method is using recursion method, for an ADV structure, the operator overloading aims to build a tree to describe the relationship of function f, and compute the derivative by doing the reverse flow of this tree.

3. USER MANUAL

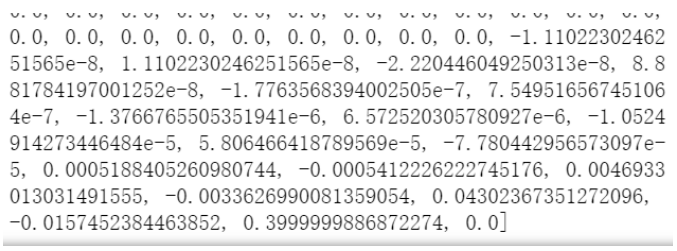
The code of Numerical differentiation, AD forward and AD backward are implemented by *JULIA*, and these codes are shown in **Numerical-Differentiation-ZCR-Final.ipynb, AD-forward-ZCR-Final.ipynb and (AD-Backward-ZCR-Final.ipynb, AD-Backward-Boris-Final.ipynb)** respectively.
For each of these codes, you need to install **IJulia** and **jupyter notebook** in order to successfully run them. After installing, you can simply choose **Kernel-Restart & Run All** in notebook to see the result for each code. The results are started with "The result is:" in my codes, and it is worth noting that, 1) for the AD forward, I use 4 different methods to implement it: special case, general case (with 2020 seeds), fast case and list case, you can check it in AD forward code; 2) for the AD Backward, I use 2 different methods to implement it: one is for ADV structure, and the other is from Boris.
If you want to change the parameters of these programs, you can change the function f in each file (without some complex operations that are not concluded in operator overloading).

4. NUMERICAL RESULTS

4.1 Numerical Differentiation

0.471438 seconds (539.51 k allocations: 58.195 MiB, 2.43% gc time)

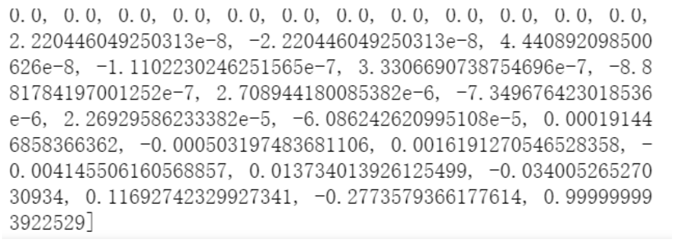
The result is:
$[0.42856221904585823; 0.8164166676039031]$ The last 20 elements of $\frac{\delta y_1}{x}$ is shown in Fig. 6



```
0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  -1. 1102230246251565e-8,  1. 1102230246251565e-8,  -2. 220446049250313e-8,  8. 881784197001252e-8,  -1. 7763568394002505e-7,  7. 54951656745106 4e-7,  -1. 3766765505351941e-6,  6. 572520305780927e-6,  -1. 0524914273446484e-5,  5. 806466418789569e-5,  -7. 780442956573097e-5,  0. 0005188405260980744,  -0. 0005412226222745176,  0. 0046933013031491555,  -0. 0033626990081359054,  0. 04302367351272096,
-0. 0157452384463852,  0. 3999999886872274,  0. 0]
```

Fig. 6. The last 20 elements of $\frac{\delta y_1}{x}$ by ND

The last 20 elements of $\frac{\delta y_2}{x}$ is shown in Fig. 7



```
0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,  0. 0,
2. 220446049250313e-8,  -2. 220446049250313e-8,  4. 440892098500626e-8,  -1. 1102230246251565e-7,  3. 3306690738754696e-7,  -8. 881784197001252e-7,  2. 708944180085382e-6,  -7. 349676423018536e-6,  2. 26929586233382e-5,  -6. 086242620995108e-5,  0. 000191446858366362,  -0. 000503197483681106,  0. 0016191270546528358,  -0. 004145506160568857,  0. 013734013926125499,  -0. 0340052652703 0934,  0. 11692742329927341,  -0. 277357936617761 4,  0. 999999993922529]
```

Fig. 7. The last 20 elements of $\frac{\delta y_2}{x}$ by ND

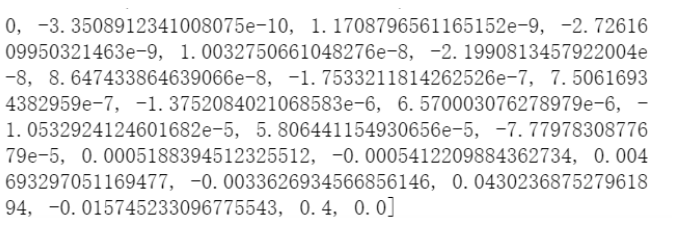
4.2 AD Forward

In those four methods of AD forward, the final results are the same, which are $[0.42856225295947, 0.816416595203433]$. However, the run time, memory allocations (MA) and garbage collection time (gc time) of these methods are different, which is shown in the table 1.

Table 1. AD Forward Performance

Method	run time (s)	MA	gc time
Fast case	0.07549	7.182 MiB	0%
Special case	0.3212	24.976 MiB	4.69%
List case	7.789354	2.977 GiB	11.89%
General case	10.4486	2.879 GiB	6.64%

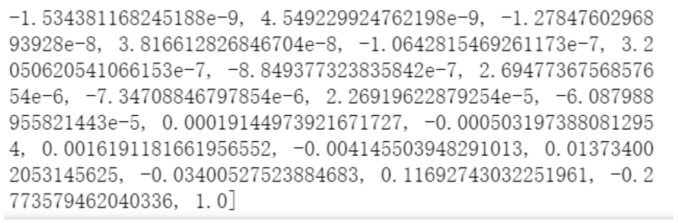
The last 20 elements of $\frac{\delta y_1}{x}$ is shown in Fig. 8



```
0,  -3. 3508912341008075e-10,  1. 1708796561165152e-9,  -2. 7261609950321463e-9,  1. 0032750661048276e-8,  -2. 1990813457922004e-8,  8. 647433864639066e-8,  -1. 7533211814262526e-7,  7. 50616934382959e-7,  -1. 3752084021068583e-6,  6. 570003076278979e-6,  -1. 0532924124601682e-5,  5. 806441154930656e-5,  -7. 7797830877679e-5,  0. 0005188394512325512,  -0. 0005412209884362734,  0. 004693297051169477,  -0. 0033626934566856146,  0. 043023687527961894,  -0. 015745233096775543,  0. 4,  0. 0]
```

Fig. 8. The last 20 elements of $\frac{\delta y_1}{x}$ by AD forward

From these cases I can find that, if I use 1 seeds to do AD forward in Fast case, embedded the direction derivative in computing, the running time is the fastest, and if I use 2020 seeds to do AD forward, in the General case, I can find that the speed is very slow compared to the Fast case. Besides, I find that compared to ADV structure, the list structure can be faster, the result is shown in the List case and the General case, using list can achieve nearly 30% acceleration than ADV structure. The last 20 elements of $\frac{\delta y_2}{x}$ is shown in Fig. 9



```
-1. 534381168245188e-9,  4. 549229924762198e-9,  -1. 27847602968
93928e-8,  3. 816612826846704e-8,  -1. 0642815469261173e-7,  3. 2
050620541066153e-7,  -8. 849377323835842e-7,  2. 69477367568576
54e-6,  -7. 34708846797854e-6,  2. 26919622879254e-5,  -6. 087988
955821443e-5,  0. 00019144973921671727,  -0. 000503197388081295
4,  0. 0016191181661956552,  -0. 004145503948291013,  0. 01373400
2053145625,  -0. 03400527523884683,  0. 11692743032251961,  -0. 2
773579462040336,  1. 0]
```

Fig. 9. The last 20 elements of $\frac{\delta y_2}{x}$ by AD forward

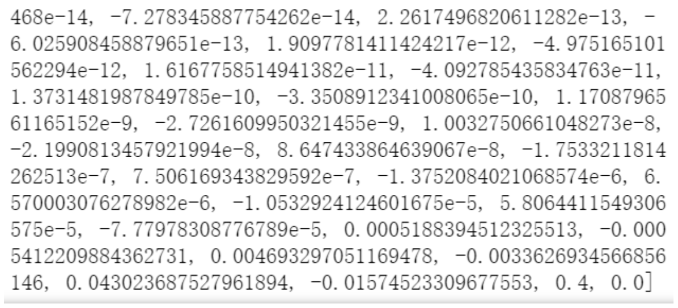
4.3 AD Backward

In those two methods of AD backward, similar to the AD forward, the final results are the same, which are $[0.42856225295947, 0.816416595203433]$. The performance of is shown in the table 2:

Table 2. AD Backward Performance

Method	run time (s)	MA	gc time
ZCR	0.110535	13.680 MiB	21.38%
Boris	2965.8631	2.315 GiB	0.10%

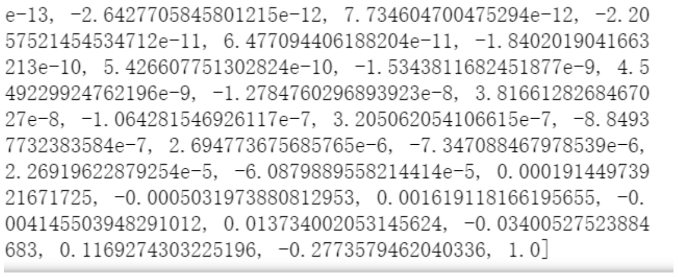
The last 20 elements of $\frac{\delta y_1}{x}$ is shown in Fig. 10



```
468e-14,  -7. 278345887754262e-14,  2. 2617496820611282e-13,  -
6. 025908458879651e-13,  1. 9097781411424217e-12,  -4. 975165101
562294e-12,  1. 6167758514941382e-11,  -4. 092785435834763e-11,
1. 3731481987849785e-10,  -3. 3508912341008065e-10,  1. 17087965
61165152e-9,  -2. 7261609950321455e-9,  1. 0032750661048273e-8,
-2. 1990813457921994e-8,  8. 647433864639067e-8,  -1. 7533211814
262513e-7,  7. 506169343829592e-7,  -1. 3752084021068574e-6,  6.
570003076278982e-6,  -1. 0532924124601675e-5,  5. 8064411549306
575e-5,  -7. 77978308776789e-5,  0. 0005188394512325513,  -0. 000
5412209884362731,  0. 004693297051169478,  -0. 0033626934566856
146,  0. 043023687527961894,  -0. 01574523309677553,  0. 4,  0. 0]
```

Fig. 10. The last 20 elements of $\frac{\delta y_1}{x}$ by AD backward

The last 20 elements of $\frac{\delta y_2}{x}$ is shown in Fig. 11



```
e-13,  -2. 6427705845801215e-12,  7. 734604700475294e-12,  -2. 20
57521454534712e-11,  6. 477094406188204e-11,  -1. 8402019041663
213e-10,  5. 426607751302824e-10,  -1. 5343811682451877e-9,  4. 5
49229924762196e-9,  -1. 2784760296893923e-8,  3. 81661282684670
27e-8,  -1. 064281546926117e-7,  3. 205062054106615e-7,  -8. 8493
7732383584e-7,  2. 694773675685765e-6,  -7. 347088467978539e-6,
2. 26919622879254e-5,  -6. 0879889558214414e-5,  0. 000191449739
21671725,  -0. 0005031973880812953,  0. 001619118166195655,  -0.
004145503948291012,  0. 013734002053145624,  -0. 03400527523884
683,  0. 1169274303225196,  -0. 2773579462040336,  1. 0]
```

Fig. 11. The last 20 elements of $\frac{\delta y_2}{x}$ by AD backward

From this result I can find that, for the large scale derivative computing tasks, the recursion method is better than the function-build method, since it will take a long time to compile the function.

4.4 Numerical Analysis

From the results I can find that, the results from these three algorithm are nearly the same, which means that in this case, the Numerical Differentiation method has the same accuracy compared to the accurate algorithm AD Forward/ AD Backward.

The run time of these methods shows that the numerical differentiation, the AD forward-fast and the AD backward-ZCR have nearly the same level run time, and the list structure can achieve nearly 30% speedup compared to the ADV structure. In addition, the more seeds we use, the more run time we have, if we use AD forward-fast and the AD backward-ZCR, the run time is significantly less than that of AD forward-general with 2020 seeds.

5. CONCLUSION

In this project, I use 7 different methods to find the derivative of function f, and compare their differences. I not only use the ADV structure shown in the course, but use a list structure to realize it and get a better performance than ADV structure (30% speedup). I will try to use list structure to re-implement the AD forward-fast and AD backward-ZCR methods and get the speedup in the future work.