

浙江大学

本科实验报告

Autograd for Algebraic Expressions

2023.11.7

Chapter 1: Introduction

Problem description:

The application of automatic differentiation technology in frameworks such as torch and tensor flow has greatly facilitated people's implementation and training of deep learning algorithms based on back propagation. Now, we hope to implement an automatic differentiation program for algebraic expressions.

Input Format

First, input an infix expression composed of the following symbols.

Operators

Type	Examples	Notes
Bracket	()	
Power	^	
Multiplication & Division	* /	
Addition & Subtraction	+ -	
Argument separator	,	optional , only used as argument separators in multivariate functions

Operands

Type	Examples	Notes
Literal constant	2 3 0 -5	Just consider integers consisting of pure numbers and minus signs.
Variable	ex cosval xy xx	Considering the above "mathematical functions" as reserved words, identifiers (strings of lowercase English letters) that are not reserved words are called variables.

Background:

Automatic differentiation (AD), also known as automatic derivative, is a generalization of the backpropagation algorithm, which can calculate the derivative value of the derivative function at a certain point. The core problem to be solved by automatic differentiation is to compute the derivatives of complex functions, usually multilayer composite functions, at a certain point, gradients, and Hessian matrix values. It shields the user from tedious derivation details and processes. At present, well-known deep learning open-source libraries provide automatic differentiation functions, including TensorFlow, pytorch, etc.

Chapter 2: Algorithm Specification

The basic idea of automatic derivation is to use the chain rule to split complex functions into simple computational units, and then derive these elements separately, and propagate forward or backward according to the topological order of the computational graph, and finally obtain the derivative of the objective function.

The steps are:

Reads a line of string from standard input as a function expression

Remove whitespace from expressions

Converts the characters in the expression into structs of type element, stored in an array

Convert an array of elements into the form of a suffix expression

Create a tree with a suffix expression

Iterate through the element array to find all the variables, for each variable, call a custom function Derivative, find the corresponding derivative, and store the tree of the derivative in the gen array

For each derivative tree, simplify to remove the extra 0s and 1s, and output the form of its fixed expression

The time complexity of the code is $O(n^2 * m)$, where n is the length of the expression and m is the number of variables, because the outer loop needs to traverse n elements, the inner loop needs to traverse n elements, and the derivative and simplification process needs to traverse m variables

The spatial complexity of the code is $O(n)$, where n is the length of the expression, because your code only needs to store arrays or pointers such as input, head, root, gen, etc., which are no larger than the length of the expression

The pseudocode is as follows:

```
1. // Allocate a character array of size MAX+1 for input, to store the input ex
   pression
2. input <- new char[MAX+1]
3.
4. // Define two integer variables i and j, for looping
5. i <- 0
6. j <- 0
7.
8. // Read a string from standard input, and store it in input
9. input <- read()
10.
11. // Call CleanSpace function, to remove spaces from input
12. CleanSpace(input)
13.
14. // Define a pointer of type element, to store the converted elements
15. head <- null
16.
```

```

17.    // Call TurnInputIntoElement function, to convert the characters in input to
        elements of type element, and return a pointer to an array of elements, and assign it to head
18.    head <- TurnInputIntoElement(input)
19.
20.    // Call TurnElementIntoSuffix function, to convert the infix expression in head to a suffix expression, and return a pointer to an array of elements, and assign it to head
21.    head <- TurnElementIntoSuffix(head)
22.
23.    // Define a pointer of type Tree, to store the root node of the expression tree
24.    root <- null
25.
26.    // Call BuildTree function, to build an expression tree from the suffix expression in head, and return a pointer to the root node, and assign it to root
27.    root <- BuildTree(head)
28.
29.    // Define an integer variable count, to record the number of variables to be differentiated, initialized to 0
30.    count <- 0
31.
32.    // Define an integer variable flag, to indicate whether a variable has been differentiated, initialized to 1
33.    flag <- 1
34.
35.    // Define a second-level pointer of type Tree, to store the root nodes of the derivative expression trees for multiple variables, and allocate an array of size MAX for Tree pointers
36.    gen <- new Tree*[MAX]
37.
38.    // Use i to loop through all the elements in head, until an element of type 0 is encountered, and find the variables in it, and differentiate them
39.    for i from 0 to length of head - 1
40.        // Use j to loop through the first i elements in head, if there is a variable that is the same as the i-th element, set flag to 0, to prevent repeated differentiation for the same variable
41.        for j from 0 to i - 1
42.            if head[i].store == head[j].store
43.                flag <- 0
44.            end if
45.        end for
46.        // If the i-th element is of type 1, and flag is 1, it means that the element is a variable that has not been differentiated

```

```

47.         if head[i].element_type == 1 and flag == 1
48.             // Print the name of the variable, followed by a colon
49.             print head[i].store + ":"
50.             // Get the length of the variable name, and assign it to length
51.             length <- length of head[i].store
52.             // Call Derivative function, to differentiate the expression tree po
               inted by root with respect to the variable, and return a pointer to the root nod
               e of the derivative expression tree, and assign it to gen[count]
53.             gen[count] <- Derivative(root, head[i].store, length)
54.             // Call inorderTraversalSimplify function, to traverse the expressio
               n tree pointed by gen[count] in order, and simplify the expression, and store th
               e result in output[count]
55.             inorderTraversalSimplify(gen[count], count)
56.             // Use DetectpreString and DeletepreString functions, to delete the
               invalid strings such as "+0" and "*1" from output[count]
57.             while DetectpreString(output[count], "+0") == 1 or DetectpreString(o
               utput[count], "*1") == 1
58.                 DeletepreString(output[count], "+0")
59.                 DeletepreString(output[count], "*1")
60.             end while
61.             // Use DetectpostString and DeletepostString functions, to delete th
               e invalid strings such as "0+" and "1*" from output[count]
62.             while DetectpostString(output[count], "0+") == 1 or DetectpostString
               (output[count], "1*") == 1
63.                 DeletepostString(output[count], "0+")
64.                 DeletepostString(output[count], "1*")
65.             end while
66.             // Use DeleteMulString function, to partially simplify the multiplic
               ation in output[count]
67.             DeleteMulString(output[count])
68.             // Print the string in output[count], as the result of the different
               iation
69.             print output[count] + "\n"
70.             // Increment count, to record the number of variables differentiated
71.             count <- count + 1
72.             // Set IndexOutput to 0, to clear the output index
73.             IndexOutput <- 0
74.         end if
75.         // Set flag to 1, to reset the indicator
76.         flag <- 1
77.     end for

```

Chapter 3: Testing Results

Disclaimer: I'm very sorry I'm very dish, so I can only take the data given by the teacher to test and look for possible problems on this, and the simplification formula I use can't be perfected to simplify, but the overall correctness is correct, but it's not so simple, the requirements in this question just say that there are two simplification rules I hope the reviewer understands

Test data: $a+b^c*d$ (teacher's test data1)

The result of all three algorithms is

a:1

b: $c*b^{(c-1)*d}$

c: $b^c*\ln(b)*d$

d: b^c

Test data: $a*10*b+2^a/a$ (teacher's test data2)

The test result is

a: $10*b+(2^a*\ln(2)*a/a^2)-(2^a/a^2)$

b: $a*10$

Test data: $xx^2/xy*xy+a^a$ (teacher's test data3)

The test result is

xx: $(2*xx^{(2-1)*xy/xy^2})*xy$

xy: $-(xx^2/xy^2)*xy+xx^2/xy$

a: $a^a*(1+\ln(a))$

Chapter 4: Analysis and Comments

To analyze the time complexity and space complexity of the code, we need to consider the cost of each function and the number of times they are called. The main function consists of the following steps:

Allocate memory for the input string and scan it from the standard input. This takes $O(MAX)$ time and space, where MAX is the maximum length of the input string.

Call the CleanSpace function to remove any spaces from the input string. This takes $O(MAX)$ time and $O(1)$ space, as it modifies the string in place.

Call the TurnInputIntoElement function to convert the input string into a sequence of elements. This takes $O(MAX)$ time and $O(MAX)$ space, as it creates an array of elements with the same length as the input string.

Call the TurnElementIntoSuffix function to convert the sequence of elements into a postfix expression. This takes $O(\text{MAX})$ time and $O(\text{MAX})$ space, as it uses a stack to store the intermediate results and creates a new array of elements for the output.

Call the BuildTree function to create a binary tree from the postfix expression. This takes $O(\text{MAX})$ time and $O(\text{MAX})$ space, as it creates a node for each element and links them together.

Allocate memory for an array of tree pointers to store the derivative trees for each variable. This takes $O(\text{MAX})$ space.

Loop through the sequence of elements and find the variables. For each variable, call the Derivative function to compute the derivative tree and store it in the array. This takes $O(\text{MAX} * N)$ time and $O(\text{MAX} * N)$ space, where N is the number of variables, as the Derivative function may create a new tree with the same size as the original tree.

Loop through the array of derivative trees and print them in the standard output. This takes $O(\text{MAX} * N)$ time and $O(1)$ space, as it uses the inorderTraversalSimplify function to traverse and simplify the tree in place.

Therefore, the total time complexity of the code is $O(\text{MAX} * N)$, and the total space complexity is $O(\text{MAX} * N)$. The code can be improved by using a more efficient data structure to store the elements and the trees, such as a linked list or a hash table, which can reduce the space complexity and avoid unnecessary memory allocation. The code can also be optimized by using more advanced techniques to simplify the derivative expressions, such as common subexpression elimination, constant folding, and algebraic simplification.

Appendix: Source Code (in C)

If your machine cannot see my code, please see the following code. If you can see it, please ignore the following content.

```
1.  #include <stdio.h>
2.  #include <math.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.
6.  #define MAX 1000 //can modify the maximum read value
7.
8.  char output[10][10000]; //The final output is an array
9.  int IndexOutput;
10.
11.  typedef struct Node{
12.      int element_type; //0 means gone, 1 means letter variable, 2 means operator, 3 means integer, and 4 means decimal
13.      char store[20]; //Both numbers and letters are placed inside this
```

```

14.     }element;
15.
16.     typedef struct Stack{//This stack is used to build the tree
17.         element data;
18.         struct Stack* next;
19.     }Stack;
20.
21.     typedef struct Tree{//Tree nodes
22.         element data;
23.         struct Tree* left;
24.         struct Tree* right;
25.     }Tree;
26.
27.
28.     void CleanSpace(char* input);//Be clear about irrelevant spaces
29.     element* TurnInputIntoElement(char* input);
30.     //int MathFuction(char* alph);//bonus Not considered
31.     element* TurnElementIntoSuffix(element* in);//Turns the infix expression of the element into a suffix
32.     Stack* PushStack(element in,Stack* input);//A function that presses into the stack
33.     element PopStack(Stack* head);//Out-stack functions
34.     Stack* initStack();//Functions that initialize the stack
35.     int IsEmpty(Stack*);//Determine if a stack is empty
36.     element PeekStack(Stack*);//Look at the top element of the stack but don't pop it out
37.     int ReturnTheValueofOp(char*);//Returns the priority of this operator
38.     int IsLetter(char object);//Determine whether it is a variable or not
39.     int IsOperator(char object);//Determines whether it is an operator
40.     int IsNumber(char object);//Determine if it's a number
41.     Tree* BuildTree(element* input);//A function that creates a tree
42.     Tree* initTree();//Initialize the function of the tree
43.     Tree* PushTreeStack(Tree*,Tree*);//The press-in stack function during the tree creation process
44.     Tree* PopTreeStack(Tree* head);//The ejection function in the process of creating a tree
45.     Tree* GradForTree(element* letter,Tree* root);//

```



```

46.  Tree* Derivative(Tree* node, char* letter, int length); //The
    e main derivative function
47.  Tree* copy(Tree* input); //Duplicate the entire tree
48.  Tree* create_multiply_node(Tree* left, Tree* right); //An o
    perational function for the derivative rule of multiplicatio
    n
49.  Tree* create_divided_node(Tree* Deriva, Tree* needed, Tree*
    right, int index); //An operational function for dividing der
    ivative rules
50.  Tree* create_cifang_node(Tree* input); //For operations on
    the square of the denominator in the derivative of division
51.  Tree* create_multimins_node(Tree* left, Tree* right); //For
    the power of the derivation rule
52.  Tree* create_ln_node(Tree* input); //Create logarithmic no
    des
53.  int contains_variable(Tree* node, char* letter, int length)
    ; //Determine if there are any variables in the subtree that
    you want to derive
54.  void inorderTraversalSimplify(Tree* root, int count); //Whe
    n exporting, use the mid-order traversal to simplify at the
    same time
55.  Tree* create_zero_node(); //Zero in on some things that do
    n't need to be used
56.  void DeleteString(char* front, char* behind); //Delete some
    of the excess parts
57.  void DeletepreString(char* front, char* behind);
58.  void DeletepostString(char* front, char* behind);
59.  int ContainsZero(char* input); //Check whether there is a
    0 that needs to be deleted
60.  char* SimplyfiyZero(char* input); //Delete the part with 0
61.  void DeleteMulString(char* front); //Delete the part conta
    ining *0
62.  Tree* CreateDoubleLeft(Tree* input);
63.  Tree* CreateDoubleRight(char* letter); //Supplements for s
    pecial cases
64.  int DetectpreString(char* input, char* behind);
65.  int DetectpostString(char* input, char* behind); //Delete by
    the position
66.
67.  int DetectpreString(char* input, char* behind)
68.  {
69.      int frontlength = strlen(input);
70.      int behindlength = strlen(behind);

```

```

71.     int i,j,k;
72.
73.     for(i = 0 ; i < frontlength;i++){
74.         if (strncmp(input + i, behind, behindlength) == 0
            &&(IsNumber(input[i-1])==1||i == 0||IsLetter(input[i-1])==1)
            &&(i==frontlength-2||IsNumber(input[i+2])!=1)) //如果找到要删除的字符串
75.         {
76.             return 1;
77.         }
78.     }
79.
80.     return 0;
81. }
82.
83. int DetectpostString(char* input,char*behind)
84. {
85.     int frontlength = strlen(input);
86.     int behindlength = strlen(behind);
87.     int i,j,k;
88.
89.     for(i = 0 ; i < frontlength;i++){
90.         if (strncmp(input + i, behind, behindlength) == 0
            &&(IsOprator(input[i-1])==1||i == 0))
91.         {
92.             return 1;
93.         }
94.     }
95.
96.     return 0;
97. }
98.
99. int main()
100. {
101.     char* input = (char*)malloc(sizeof(char)*MAX+1);
102.     int i,j;
103.
104.     scanf("%s",input);
105.
106.     CleanSpace(input);
107.
108.     element* head ;

```

```

109.
110.     head = TurnInputIntoElement(input);//Convert the input into an element element
111.
112.     head = TurnElementIntoSuffix(head);//Convert an infix expression to a suffix expression
113.
114.     Tree* root = BuildTree(head);//A tree is created from a suffix expression
115.
116.     int count = 0;
117.     int flag = 1;//The setting indicates whether the derivative has been found for the amount of change
118.
119.     Tree** gen = (Tree**)malloc(sizeof(Tree*)*MAX);//A second-order pointer to the root node is used to store the root node of the tree after derivatives of multiple variables
120.
121.     for(i = 0 ; head[i].element_type!=0; i++){//Traverse the entire input element to find the variables in it to find the derivative
122.         for(j = 0 ; j < i;j++){
123.             if(strcmp(head[i].store,head[j].store)==0){
124.                 flag = 0;//Prevent multiple derivatives for a variable
125.             }
126.         }
127.         if(head[i].element_type == 1&&flag == 1){
128.             printf("%s:",head[i].store);
129.             int length = (int)strlen(head[i].store);//This length is the length of the character that the variable occupies
130.             gen[count] = Derivative(root,head[i].store,length);
131.             inorderTraversalSimplify(gen[count],count);
132.
133.             //Directly remove everything from the string level to simplify it
134.             while(DetectpreString(output[count],"+0")==1||DetectpreString(output[count],"*1")==1){//Use a function to delete the content
135.                 DeletepreString(output[count],"+0");
136.                 DeletepreString(output[count],"*1");

```

```

137.         }
138.
139.         while(DetectpostString(output[count], "0+")
    ==1 || DetectpostString(output[count], "1*")==1){//Use a functi
    on to delete the content
140.             DeletepostString(output[count], "0+");
141.             DeletepostString(output[count], "1*");
142.         }
143.
144.         DeleteMulString(output[count]);//Partial
    simplification of multiplication
145.         printf("%s\n", (output[count]));//output
146.         count++;//Records are the first few varia
    bles
147.         IndexOutput = 0;
148.     }
149.     flag = 1;//Re-set the flag
150. }
151.
152.
153.
154.     return 0;
155. }
156.
157. void CleanSpace(char* str)//Find a way to remove spaces
158. {
159.
160.     char* p = str;
161.     int i = 0;
162.
163.     while(*p){
164.         if(*p!=' '){
165.             str[i++] = *p++;
166.         }
167.     }
168.
169.     str[i] = '\0';
170.
171. }
172.
173. element* TurnInputIntoElement(char* input)
174. {

```

```

175.
176.     element* head = (element*)malloc(sizeof(element)*MAX)
    ;
177.     int index = 0;
178.     while(*input!='\0'){//Read in characters one by one
179.         if(IsLetter(*input)==1){//If it's a character
180.             head[index].element_type = 1;
181.             int temp_for_count;
182.             for(temp_for_count = 0;IsLetter(*(input))==1;
temp_for_count++){ //Fill in the string with a loop
183.                 head[index].store[temp_for_count] = *input;
184.                 input++;
185.             }
186.             head[index].store[temp_for_count] = '\0';//heal
al
187.         }else if(IsOprator(*input)== 1||(index != 0&&(*(input-1))== ')' || IsNumber(*(input-1)) || IsLetter(*(input-1))))
{//Deposit operator
188.             head[index].element_type = 2;
189.             head[index].store[0] = *input;
190.             head[index].store[1] = '\0';//heal
191.             input++;
192.         }else if(IsNumber(*input)== 1 || *input == '.' ||
(*input == '-' && (index == 0 || IsOprator(head[index-1].store[0]) == 1 || head[index-1].store[0] == '('))){ //Added judgment for decimal and minus signs
193.             head[index].element_type = 3;
194.             int temp_for_count;
195.             for(temp_for_count = 0;IsNumber(*(input))==1
|| *(input) == '.' || (*input == '-' && (index == 0 || IsOprator(head[index-1].store[0]) == 1 || head[index-1].store[0] == '('));temp_for_count++){ //Added judgment for decimal and minus signs
196.                 if(*input=='.'){
197.                     head[index].element_type = 4;//Added decimal judgment
198.                 }
199.                 head[index].store[temp_for_count] = *input;
200.                 input++;
201.             }
202.             head[index].store[temp_for_count] = '\0';//heal
al

```

```

203.         }
204.         index++;
205.     }
206.     head[index].element_type = 0; //Changing the type to 0
    at the end of the day indicates the end of the whole
207.
208.     return head;
209. }
210.
211. //Define a function, the parameter is an element pointer,
    and the return value is also an element pointer
212. element* TurnElementIntoSuffix(element* in)
213. {
214.     //Create an array of suffix expressions, allocate mem
    ory space with malloc, the size is the maximum number of ele
    ments
215.     element* postfix = (element*)malloc(sizeof(element)*M
        AX);
216.     //Create a stack to store operators
217.     Stack* temp = initStack();
218.
219.     int length;
220.
221.     length = 0 ;
222.
223.     //Loop through the input elements until you encounter
    an element type of 0 as the end flag
224.     while(in->element_type!=0){
225.         if(in->element_type == 1 || in->element_type
            == 3 || in->element_type == 4){//Not an operator
226.             postfix[length++] = *in; //Add the element to
            the postfix array
227.             in++; //Move to the next element
228.             }else if(in->element_type == 2 && in->store[0]
                == '('){ //If the element is a left parenthesis
229.                 temp = PushStack (*in,temp); //Push it into t
                he stack
230.                 in++; //Move to the next element
231.                 }else if(in->element_type == 2 && in->store[0]
                    == ')'){ //If the element is a right parenthesis
232.                     while(PeekStack(temp).store[0]!='('){ //Pop t
                        he stack until you encounter a left parenthesis

```

```

233.         postfix[length++] = PopStack(temp); //Add
           the popped element to the postfix array
234.     }
235.     PopStack(temp); //Pop the Left parenthesis
236.     in++; //Move to the next element
237.     }else if(in->element_type==2){ //If the element
           is an operator
238.         if (IsEmpty(temp)==1||PeekStack(temp).store[0]=='('||ReturnTheValueofOp(in->store)>ReturnTheValueofOp(PeekStack(temp).store)){ //If the stack is empty, or the top of the stack is a left parenthesis, or the priority of the current operator is higher than the top of the stack
239.             temp = PushStack(*in,temp); //Push the current operator into the stack
240.         }else{ //Otherwise
241.             while(IsEmpty(temp)==0&&ReturnTheValueofOp(in->store)<=ReturnTheValueofOp(PeekStack(temp).store))
                { //Pop the stack until the stack is empty or the priority of the current operator is higher than the top of the stack
242.                 postfix[length++] = PopStack(temp)
                ; //Add the popped element to the postfix array
243.             }
244.             temp = PushStack(*in,temp); //Push the current operator into the stack
245.         }
246.         in++; //Move to the next element
247.     }
248. }
249.
250. while(IsEmpty(temp)!=1){ //Pop the remaining elements in the stack
251.     postfix[length++] = PopStack(temp); //Add them to the postfix array
252. }
253.
254. return postfix; //Return the postfix array
255. }
256.
257. int ReturnTheValueofOp(char*input)
258. {
259.     int result = 0;
260.     if(input[0]=='^'){
261.         result = 3;

```

```

262.     }else if(input[0]=='*' || input[0]=='/'){
263.         result = 2;
264.     }else if(input[0] == '+' || input[0] == '-'){
265.         result = 1;
266.     }else if(input[0] == ','){
267.         result = 0;
268.     }
269.
270.     return result;
271. }
272.
273. Stack* PushStack(element in,Stack*input)//Standard operation for press-into stacks
274. {
275.     Stack* cur = (Stack*)malloc(sizeof(Stack));
276.     Stack* temp;
277.     cur ->data = in;
278.     temp = input->next;
279.     input->next = cur;
280.     cur->next = temp;
281.
282.     return input;
283. }
284.
285. element PopStack(Stack* head)//Regular ejection operations
286. {
287.     element result = (head->next)->data;
288.     head ->next = head ->next->next;
289.
290.     return result;
291. }
292.
293. element PeekStack(Stack*input)
294. {
295.     element result = input->next->data;
296.
297.     return result;
298. }
299. //Define a function, the parameter is an element pointer,
    and the return value is a tree pointer

```



```

300. Tree* BuildTree(element* input)
301. {
302.     //Create an array of tree pointers, allocate memory space with malloc, the size is the maximum number of elements
303.     Tree** head = (Tree**)malloc(sizeof(Tree*)*MAX);
304.     //Initialize a variable to store the index of the top of the array
305.     int top = -1;
306.
307.     //Loop through the input elements until you encounter an element type of 0 as the end flag
308.     while (input->element_type != 0){
309.         //If the element is an operand (number, variable, or constant)
310.         if((input->element_type==1||(input->element_type==3||(input->element_type==4)){
311.             //Create a new tree node and assign the element to its data field
312.             Tree* Node = initTree();
313.             Node->data = *input;
314.             //Push the node into the array
315.             head[++top] = Node;
316.         }else{ //If the element is an operator
317.             //Create a new tree node and assign the element to its data field
318.             Tree* Node = initTree();
319.             Node->data = *input;
320.             //Pop two nodes from the array and assign them to the right and left children of the current node
321.             Node->right = head[top--];
322.             Node->left = head[top--];
323.             //Push the current node into the array
324.             head[++top] = Node;
325.         }
326.         //Move to the next element
327.         input++;
328.     }
329.
330.     //Return the root node of the tree, which is the last element in the array
331.     return head[top];
332. }
333.

```

```

334. Stack* initStack()//Initialize the stack
335. {
336.     Stack* new = (Stack*)malloc(sizeof(Stack)*MAX);
337.     new->next = NULL;
338.
339.     return new;
340. }
341.
342. int IsEmpty(Stack*input)
343. {
344.     int result = 0;
345.     if(input->next ==NULL){
346.         result = 1;
347.     }
348.
349.     return result;
350. }
351.
352. int IsLetter(char object)
353. {
354.     int result = 0;
355.
356.     if(object >= 'a'&& object <='z'){
357.         result = 1;
358.     }
359.
360.     if(object >= 'A'&& object <='Z'){
361.         result = 1;
362.     }
363.
364.     return result;
365. }
366.
367. int IsOprator(char object)
368. {
369.     int result = 0;
370.     if(object=='+' ||object =='*' ||object == '/' ||object
    == '^' ||object == ',' ||object=='(' ||object==')'){
371.         result = 1;
372.     }//Additional judgment is required

```

```

373.     return result;
374. }
375. int IsNumber(char object)
376. {
377.     int result = 0;
378.     if(object>='0'&&object<='9'){
379.         result = 1;
380.     }
381.     return result;
382. }
383. Tree* initTree()
384. {
385.     Tree* node = (Tree*)malloc(sizeof(Tree));
386.     node->left = NULL;
387.     node->right = NULL;
388.
389.     return node;
390. }
391.
392. Tree* copy(Tree* input)//Create a duplicate version so th
    at the original tree is not changed
393. {
394.     if(input == NULL){
395.         return NULL;
396.     }
397.
398.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
399.
400.     new_node->data = input->data;
401.     new_node->left = copy(input->left);
402.     new_node->right = copy(input->right);
403.
404.     return new_node;
405. }
406.
407. Tree* create_multiply_node(Tree* left,Tree* right)
408. {
409.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
410.     new_node->data.element_type = 2;
411.     new_node->data.store[0] = '*';

```

```

412.     new_node->left = left;
413.     new_node->right = right;
414.
415.     return new_node;
416. }
417. //Define a function, the parameters are four tree pointer
s, and the return value is a tree pointer
418. Tree* create_divided_node(Tree* Deriva,Tree* needed,Tree*
    right,int index)
419. {
420.     //Create a new node for the root of the quotient rule
tree, allocate memory space with malloc, and assign the '/'
operator to its data field
421.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
422.     new_node->data.element_type = 2;
423.     new_node->data.store[0] = '/';
424.
425.     //Create a new node for the left child of the root no
de, allocate memory space with malloc, and assign the '*' op
erator to its data field
426.     Tree* new_node_left= (Tree*)malloc(sizeof(Tree));
427.     new_node_left->data.element_type = 2;
428.     new_node_left->data.store[0] = '*';
429.
430.     //Assign the first parameter (Deriva) to the Left chi
ld of the new node left, which is the derivative of the nume
rator of the original function
431.     new_node_left->left = Deriva;
432.
433.     //Create a temporary pointer to traverse the tree
434.     Tree* temp = (Tree*)malloc(sizeof(Tree));
435.     temp = Deriva;
436.
437.     //Find the leftmost node of the Deriva tree
438.     while(temp->left!=NULL){
439.         temp = temp->left;
440.     }
441.
442.     //Create a new node for the left parenthesis, allocat
e memory space with malloc, and assign the '(' character to
its data field
443.     Tree* Left = (Tree*)malloc(sizeof(Tree));
444.     Left->data.element_type = 3;

```

```

445.     Left->data.store[0] = '(';
446.     Left->left = NULL;
447.     Left->right = NULL;
448.
449.     //Assign the left parenthesis node to the left child
of the leftmost node of the Deriva tree
450.     temp->left = Left;
451.
452.     //Assign the second parameter (needed) to the right c
hild of the new node left, which is the denominator of the o
riginal function
453.     new_node_left->right = needed;
454.
455.     //Assign the new node left to the left child of the r
oot node
456.     new_node->left = new_node_left;
457.
458.     //Create a new node for the right child of the root n
ode, allocate memory space with malloc, and assign the '^' o
perator to its data field
459.     Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
460.     new_node_right->data.element_type = 2;
461.     new_node_right->data.store[0] = '^';
462.     //Assign the third parameter (right) to the left chil
d of the new node right, which is the denominator of the ori
ginal function
463.     new_node_right->left = right;
464.
465.     //Create a new node for the right child of the new no
de right, allocate memory space with malloc, and assign the
'2)' characters to its data field
466.     Tree* new_node_right_right = (Tree*)malloc(sizeof(Tre
e));
467.     new_node_right_right->data.element_type = 3;
468.     new_node_right_right->data.store[0] = '2';
469.     new_node_right_right->data.store[1] = ')';
470.
471.     //Set the left and right children of the new node rig
ht right to NULL
472.     new_node_right_right->left = NULL;
473.     new_node_right_right->right = NULL;
474.     //Assign the new node right right to the right child
of the new node right
475.     new_node_right->right = new_node_right_right;

```

```

476.
477.     //Assign the new node right to the right child of the
         root node
478.     new_node->right = new_node_right;
479.
480.     //Return the root node of the quotient rule tree
481.     return new_node;
482. }
483.
484. Tree* create_cifang_node(Tree* input)
485. {
486.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
487.     Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
488.
489.     new_node->data.element_type = 2;
490.     new_node->data.store[0] = '^';
491.     new_node->left = input;
492.
493.     new_node_right->data.element_type = 3;
494.     new_node_right->data.store[0] = '2';
495.     new_node_right->left = NULL;
496.     new_node_right->right = NULL;
497.
498.     new_node->right = new_node_right;
499.
500.     return new_node;
501. }
502.
503. //Define a function, the parameters are two tree pointers,
         and the return value is also a tree pointer
504. Tree* create_multimins_node(Tree* left, Tree* right)
505. {
506.     //Create a new node for the root of the power rule tr
         ee, allocate memory space with malloc, and assign the '^' op
         erator to its data field
507.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
508.     new_node->data.element_type = 2;
509.     new_node->data.store[0] = '^';
510.     //Assign the first parameter (left) to the left child
         of the root node, which is the base of the original functio
         n
511.     new_node->left = left;

```

```

512.
513.     //Create a new node for the right child of the root n
ode, allocate memory space with malloc, and assign the '-' o
perator to its data field
514.     Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
515.     new_node_right->data.element_type = 2;
516.     new_node_right->data.store[0] = '-';
517.
518.     //Create a new node for the left child of the new nod
e right, allocate memory space with malloc, and assign the '
(' character and the second parameter (right) to its data fi
eld
519.     Tree* new_node_right_left = (Tree*)malloc(sizeof(Tree)
);
520.     new_node_right_left->data.element_type = 1;
521.     new_node_right_left->data.store[0] = '(';
522.     int i;
523.
524.     //Copy the data of the right parameter to the new nod
e right left's data field, starting from the second position
525.     for(i = 0; right->data.store[i]!='\0';i++){
526.         new_node_right_left->data.store[i+1] = right->dat
a.store[i];
527.     }
528.
529.     //Assign the left and right children of the right par
ameter to the left and right children of the new node right
left
530.     new_node_right_left->left = right->left;
531.     new_node_right_left->right = right->right;
532.
533.     //Assign the new node right left to the left child of
the new node right, which is the exponent of the original f
unction
534.     new_node_right->left = new_node_right_left;
535.
536.     //Create a new node for the right child of the new no
de right, allocate memory space with malloc, and assign the
'1)' characters to its data field
537.     Tree* new_node_right_right = (Tree*)malloc(sizeof(Tre
e));
538.     new_node_right_right->data.element_type = 3;
539.     new_node_right_right->data.store[0] = '1';
540.     new_node_right_right->data.store[1] = ')';

```

```

541.      //Set the left and right children of the new node right
      ht right to NULL
542.      new_node_right_right->left = NULL;
543.      new_node_right_right->right = NULL;
544.
545.      //Assign the new node right right to the right child
      of the new node right, which is the constant to be subtracte
      d from the exponent
546.      new_node_right->right = new_node_right_right;
547.      //Assign the new node right to the right child of the
      root node, which is the new exponent of the power rule
548.      new_node->right = new_node_right;
549.
550.      //Return the root node of the power rule tree
551.      return new_node;
552.  }
553.
554.  //Define a function, the parameter is a tree pointer, and
      the return value is also a tree pointer
555.  Tree* create_ln_node(Tree* input)
556.  {
557.      //Create a new node for the root of the logarithm tre
      e, allocate memory space with malloc, and assign the input's
      data to its data field
558.      Tree* new_node = (Tree*)malloc(sizeof(Tree));
559.      new_node->data.element_type = 2;
560.      new_node->data.store[0] = input->data.store[0];
561.
562.      //Create a new node for the left child of the root no
      de, allocate memory space with malloc, and assign the 'ln('
      characters to its data field
563.      Tree* new_node_left = (Tree*)malloc(sizeof(Tree));
564.      new_node_left->data.element_type = 2;
565.      new_node_left->data.store[0] = 'l';
566.      new_node_left->data.store[1] = 'n';
567.      new_node_left->data.store[2] = '(';
568.      new_node_left->left = NULL;
569.      new_node_left->right = NULL;
570.
571.      //Create a new node for the right child of the root n
      ode, allocate memory space with malloc, and assign the ')' c
      haracter to its data field
572.      Tree* new_node_right = (Tree*)malloc(sizeof(Tree));

```



```

573.     new_node_right->data.element_type = 2;
574.     new_node_right->data.store[0] = ')';
575.     new_node_right->left = NULL;
576.     new_node_right->right = NULL;
577.
578.     //Assign the new node left to the left child of the r
oot node
579.     new_node->left = new_node_left;
580.     //Assign the new node right to the right child of the
root node
581.     new_node->right = new_node_right;
582.
583.     //Return the root node of the logarithm tree
584.     return new_node;
585.
586. }
587.
588. Tree* Derivative(Tree* node, char* letter, int length)
589. {
590.     if(node == NULL){
591.         return NULL;
592.     }
593.
594.     int i;
595.     char temp[20];
596.
597.     for(i = 0 ; i < length;i++){
598.         temp[i] = letter[i];
599.     }
600.
601.     temp[i] = '\0';
602.
603.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
604.
605.     if(node->data.element_type == 2 && node->data.store[0]
        == '+'){
606.         new_node->data.element_type = 2;
607.         new_node->data.store[0] = '+';
608.         if(contains_variable(node->left,temp,length) == 1)
        {
609.             new_node->left = Derivative(node->left,letter,
                length);
610.         }else{

```

```

611.         new_node->left = create_zero_node();
612.     }
613.     if(contains_variable(node->right,temp,length)==1){
614.         new_node->right = Derivative(node->right,lette
r,length);
615.     }else{
616.         new_node->right = create_zero_node();
617.     }
618.
619.     }else if(node->data.element_type == 2&& node->data.st
ore[0] == '-') {
620.         new_node->data.element_type = 2;
621.         new_node->data.store[0] = '-';
622.         if(contains_variable(new_node->left,letter,length)
==1){
623.             new_node->left = Derivative(node->left,letter,
length);
624.         }else{
625.             new_node->left = create_zero_node();
626.         }
627.         if(contains_variable(new_node->right,letter,leengt
h)==1){
628.             new_node->right = Derivative(node->right,lette
r,length);
629.         }else{
630.             new_node->right = create_zero_node();
631.         }
632.     }else if(node->data.element_type == 2&& node->data.st
ore[0] == '*') {
633.         new_node->data.element_type = 2;
634.         new_node->data.store[0] = '+';
635.         if(contains_variable(node->left,letter,length)==0)
        {
636.             Tree* new_node_left = (Tree*)malloc(sizeof(Tr
ee));
637.             new_node_left->data.element_type = 3;
638.             new_node_left->data.store[0] = '0';
639.             new_node_left->left = NULL;
640.             new_node_left->right = NULL;
641.             new_node->left = new_node_left;
642.         }else{
643.             new_node->left = create_multiply_node(Derivat
ive(node->left,letter,length), copy(node->right));
644.         }

```

```

645.
646.         if(contains_variable(node->right,letter,length)==
        0){
647.             Tree* new_node_right = (Tree*)malloc(sizeof(T
ree));
648.             new_node_right->data.element_type = 3;
649.             new_node_right->data.store[0] = '0';
650.             new_node_right->left = NULL;
651.             new_node_right->right = NULL;
652.             new_node->right = new_node_right;
653.         }else{
654.             new_node->right = create_multiply_node(Deriva
tive(node->right,letter,length), copy(node->left));
655.         }
656.     }else if(node->data.element_type == 2 && node->data.s
tore[0] == '/'){
657.         new_node->data.element_type = 2;
658.         new_node->data.store[0] = '-';
659.         new_node->left = create_divided_node(copy(Derivat
ive(node->left,letter,length)),copy(node->right),copy(node->
right),0);
660.         new_node->right = create_divided_node(copy(Deriva
tive(node->right,letter,length)),copy(node->left),copy(node->
right),1);
661.     }else if(node->data.element_type == 2 && node->data.s
tore[0] == '^'){
662.         new_node->data.element_type = 2;
663.         if(strcmp(temp,node->left->data.store)==0 &&strcm
p(temp,node->right->data.store)==0){
664.             new_node->data.element_type = 2;
665.             new_node->data.store[0] = '*';
666.             new_node->left = CreateDoubleLeft(copy(node))
;
667.             new_node->right = CreateDoubleRight(letter);
668.         }
669.         else if(strcmp(temp,node->left->data.store)==0){
670.             new_node->data.element_type = 2;
671.             new_node->data.store[0] = '*';
672.             new_node->left = node->right;
673.             new_node->right = create_multimins_node(copy(
node->left),copy(node->right));
674.         }else if(strcmp(temp,node->right->data.store)==0)
{
675.             new_node->data.element_type = 2;

```

```

676.         new_node->data.store[0] = '*';
677.         new_node->left = copy(node);
678.         new_node->right = create_ln_node(node->left);
679.         }else if(strcmp(temp,node->left->data.store)!=0&&
        strcmp(temp,node->right->data.store)!=0){
680.             new_node->data.element_type = 3;
681.             new_node->data.store[0] = '0';
682.             new_node->left = NULL;
683.             new_node->right = NULL;
684.         }
685.         }else if(node->data.element_type == 3||node->data.ele
        ment_type == 4){
686.             new_node = node;
687.         }else if(node->data.element_type == 1&&strcmp(node->d
        ata.store,temp)!=0){
688.             new_node->data.element_type = 3;
689.             new_node->data.store[0] = '0';
690.             new_node->left = node->left;
691.             new_node->right = node->right;
692.         }else if(node->data.element_type == 1&&strcmp(node->d
        ata.store,temp)==0){
693.             new_node->data.element_type == 3;
694.             new_node->data.store[0] = '1';
695.             new_node->left = node->left;
696.             new_node->right = node->right;
697.         }
698.
699.         return new_node;
700.     }
701.
702.     int contains_variable(Tree* node, char* letter,int length)
        {
703.
704.         int i;
705.         char temp[20];
706.
707.         for(i = 0 ; i < length;i++){
708.             temp[i] = letter[i];
709.         }
710.         temp[i] = '\0';
711.
712.         if (node == NULL) {
713.             return 0;
714.         }

```

```

715.     if ((node->data.element_type == 1 || node->data.element
       _type == 3 || node->data.element_type == 4) && strcmp(node->da
       ta.store,temp)==0) {
716.         return 1;
717.     }
718.     return contains_variable(node->left, letter,length) |
       | contains_variable(node->right, letter,length);
719. }
720.
721. void inorderTraversalSimplify(Tree* root,int count)
722. {
723.     if(root != NULL){
724.         int i;
725.
726.         inorderTraversalSimplify(root->left,count);
727.         for(i = 0;i < strlen(root->data.store);i++){
728.             output[count][IndexOutput++] = root->data.
       store[i];
729.         }
730.         inorderTraversalSimplify(root->right,count);
731.     }
732. }
733.
734.
735. Tree* create_zero_node()//Save the node with a derivative
       of 0
736. {
737.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
738.     new_node->data.element_type = 3;
739.     new_node->data.store[0] = '0';
740.     new_node->left = NULL;
741.     new_node->right = NULL;
742.
743.     return new_node;
744. }
745.
746. void DeleteString(char* front,char* behind)
747. {
748.     int frontlength = strlen(front);
749.     int behindlength = strlen(behind);
750.     int i,j,k;

```

```

751.
752.     for(i = 0 ; i < frontlength;i++){
753.         if (strncmp(front + i, behind, behindlength) == 0
            &&(IsOprator(front[i-1])==1||i == 0)) //If you find the string you want to delete
754.             {
755.                 for (j = i, k = i + behindlength; k < frontlength; j++, k++) //Move the elements behind you forward
756.                     {
757.                         front[j] = front[k];
758.                     }
759.                 front[j] = '\0'; //Set the end character
760.                 break;
761.             }
762.     }
763. }
764. void DeletepreString(char* front,char* behind)
765. {
766.     int frontlength = strlen(front);
767.     int behindlength = strlen(behind);
768.     int i,j,k;
769.
770.     for(i = 0 ; i < frontlength;i++){
771.         if (strncmp(front + i, behind, behindlength) == 0
            &&(IsNumber(front[i-1])==1||i == 0||IsLetter(front[i-1])==1)
            &&(i==frontlength-2||IsNumber(front[i+2])!=1)) //If you find the string you want to delete
772.             {
773.                 for (j = i, k = i + behindlength; k < frontlength; j++, k++) //Move the elements behind you forward
774.                     {
775.                         front[j] = front[k];
776.                     }
777.                 front[j] = '\0'; //Set the end character
778.                 break;
779.             }
780.     }
781. }
782.
783. void DeletepostString(char* front,char* behind)
784. {
785.     int frontlength = strlen(front);

```

```

786.     int behindlength = strlen(behind);
787.     int i,j,k;
788.
789.     for(i = 0 ; i < frontlength;i++){
790.         if (strncmp(front + i, behind, behindlength) == 0
            &&(IsOprator(front[i-1])==1||i == 0)) //If you find the string you want to delete
791.         {
792.             for (j = i, k = i + behindlength; k < frontlength; j++, k++) //Move the elements behind you forward
793.             {
794.                 front[j] = front[k];
795.             }
796.             front[j] = '\0'; //Set the end character
797.             break;
798.         }
799.     }
800. }
801.
802. int ContainsZero(char* input)//Check whether the following node contains 0
803. {
804.     int result = 0;
805.     int i;
806.     for(i = 0 ; i < strlen(input)-1;i++){
807.         if(input[i]=='0'&&(input[i+1]=='*'||input[i+1] ==
            '/')){
808.             result = 1;
809.         }
810.         if((input[i] == '*'||input[i] == '/')&&input[i+1]
            == '0'){
811.             result = 1;
812.         }
813.     }
814.
815.     return result;
816. }
817.
818. char* SimplyfyZero(char* input)//Simplify the number of nodes with 0
819. {
820.     int index = 0;

```

```

821.     int i;
822.     int begin = 0;
823.
824.     char* temp = (char*)malloc((strlen(input)+1));
825.     for(i = 0 ; i < strlen(input);i++){
826.         temp[i] = input[i];
827.     }
828.
829.     for(i = 0;i < strlen(input);i++){
830.         if(input[i]=='+'||input[i]=='-'||input[i] == '\0')
831.         {
832.             char* sub = (char*)malloc(sizeof(char)*(i-begin+1));
833.             strncpy(sub,input+begin,i-begin+1);
834.             if(ContainsZero(sub)==1){
835.                 DeleteString(temp,sub);
836.                 begin = i+1;
837.             }
838.         }
839.     }
840.     return temp;
841. }
842. void DeleteMulString(char* front)
843. {
844.     int frontlength = strlen(front);
845.     int i,j,k;
846.
847.     for(i = 0 ; i < frontlength;i++){
848.         if(((front[i] == '0')&&front[i+1]== '*'&&i == 0)||
849.         (front[i]=='*'&&front[i+1]=='0')||((front[i]=='0')&&front[i+1]=='*')&&!IsNumber(front[i-1])))
850.         {
851.             //Find the start and end positions of the multiplication expression
852.             int start = i-1; //Start with *0 or the previous character of 0*
853.             int end = i+2; //Start with *0 or the last character of 0* and look backwards
854.             while(start >= 0 && front[start] != '+' && front[start] != '-') //If you don't encounter a plus or minus sign, keep looking
855.             {

```



```

855.         start--;
856.     }
857.     while(end < frontlength && front[end] != '+'
    && front[end] != '-'&&front[end]!='') //If you don't encounter a plus or minus sign, keep looking backwards
858.     {
859.         end++;
860.     }
861.     //Move the following elements forward, covering the entire multiplication expression
862.     for (j = start, k = end; k < frontlength; j++, k++)
863.     {
864.         front[j] = front[k];
865.     }
866.     front[j] = '\0';
867.     frontlength = strlen(front); //Update the string length
868.     break;
869. }
870. }
871. }
872.
873. Tree* CreateDoubleLeft(Tree* input)
874. {
875.     Tree* new_node = input;
876.
877.     return input;
878. }
879.
880. Tree* CreateDoubleRight(char* letter)//Handling of special cases to the second side
881. {
882.     Tree* new_node = (Tree*)malloc(sizeof(Tree));
883.     Tree* new_node_left = (Tree*)malloc(sizeof(Tree));
884.     Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
885.
886.     new_node->data.element_type = 2;
887.     new_node->data.store[0] = '+';
888.
889.     new_node_left->data.element_type = 3;
890.     new_node_left->data.store[0] = '(';

```

```

891.     new_node_left->data.store[1] = '1';
892.     new_node_left->left = NULL;
893.     new_node_left->right = NULL;
894.
895.     new_node_right->data.element_type = 1;
896.     new_node_right->data.store[0] = '1';
897.     new_node_right->data.store[1] = 'n';
898.     new_node_right->data.store[2] = '(';
899.     int i;
900.     for(i = 0; letter[i]!='\0'; i++){
901.         new_node_right->data.store[i+3] = letter[i];
902.     }
903.     new_node_right->data.store[i+3] = ')';
904.     new_node_right->data.store[i+4] = ')';
905.     new_node_right->left = NULL;
906.     new_node_right->right = NULL;
907.
908.     new_node->left = new_node_left;
909.     new_node->right = new_node_right;
910.
911.     return new_node;
912. }

```

Declaration

I hereby declare that all the work done in this project titled "Autograd for Algebraic Expressions" is of my independent effort.