浙江大學

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:

```
    // Allocate a character array of size MAX+1 for input, to store the input expression
    input <- new char[MAX+1]

    // Define two integer variables i and j, for looping
    i <- 0
    j <- 0

    // Read a string from standard input, and store it in input
    input <- read()

    // Call CleanSpace function, to remove spaces from input
    CleanSpace(input)

    // Define a pointer of type element, to store the converted elements
    head <- null

```

```
    // 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
    head <- TurnInputIntoElement(input)

    // 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
    head <- TurnElementIntoSuffix(head)

    // Define a pointer of type Tree, to store the root node of the expression tree
    root <- null

    // 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
    root <- BuildTree(head)

    // Define an integer variable count, to record the number of variables to be differentiated, initialized to 0
    count <- 0

    // Define an integer variable flag, to indicate whether a variable has been differentiated, initialized to 1
    flag <- 1

    // 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
    gen <- new Tree*[MAX]

    // 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
    for i from 0 to length of head - 1
        // 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
        for j from 0 to i - 1
            if head[i].store == head[j].store
                flag <- 0
            end if
        end for
        // 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
```

```
        if head[i].element_type == 1 and flag == 1
            // Print the name of the variable, followed by a colon
            print head[i].store + ":"
            // Get the length of the variable name, and assign it to length
            length <- length of head[i].store
            // Call Derivative function, to differentiate the expression tree pointed by root with respect to the variable, and return a pointer to the root node of the derivative expression tree, and assign it to gen[count]
            gen[count] <- Derivative(root, head[i].store, length)
            // Call inorderTraversalSimplify function, to traverse the expression tree pointed by gen[count] in order, and simplify the expression, and store the result in output[count]
            inorderTraversalSimplify(gen[count], count)
            // Use DetectpreString and DeletepreString functions, to delete the invalid strings such as "+0" and "*1" from output[count]
            while DetectpreString(output[count], "+0") == 1 or DetectpreString(output[count], "*1") == 1
                DeletepreString(output[count], "+0")
                DeletepreString(output[count], "*1")
            end while
            // Use DetectpostString and DeletepostString functions, to delete the invalid strings such as "0+" and "1*" from output[count]
            while DetectpostString(output[count], "0+") == 1 or DetectpostString(output[count], "1*") == 1
                DeletepostString(output[count], "0+")
                DeletepostString(output[count], "1*")
            end while
            // Use DeleteMulString function, to partially simplify the multiplication in output[count]
            DeleteMulString(output[count])
            // Print the string in output[count], as the result of the differentiation
            print output[count] + "\n"
            // Increment count, to record the number of variables differentiated
            count <- count + 1
            // Set IndexOutput to 0, to clear the output index
            IndexOutput <- 0
        end if
        // Set flag to 1, to reset the indicator
        flag <- 1
    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 (MAX) time and O (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 (MAX) time and O (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 (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 (MAX * N) time and O (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 (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 (MAX * N), and the total space complexity is O (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.

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#define MAX 1000  //can modify the maximum read value

char output[10][10000];//The final output is an array
int IndexOutput;

typedef struct Node{
    int element_type;//0 means gone, 1 means letter variable, 2 means operator, 3 means integer, and 4 means decimal
    char store[20];//Both numbers and letters are placed inside this
```

```
    }element;

    typedef struct Stack{//This stack is used to build the tree
        element data;
        struct Stack* next;
    }Stack;

    typedef struct Tree{//Tree nodes
        element data;
        struct Tree* left;
        struct Tree* right;
    }Tree;


    void CleanSpace(char* input);//Be clear about irrelevant spaces
    element* TurnInputIntoElement(char* input);
    //int MathFuction(char* alph);//bonus Not considered
    element* TurnElementIntoSuffix(element* in);//Turns the infix expression of the element into a suffix
    Stack* PushStack(element in,Stack* input);//A function that presses into the stack
    element PopStack(Stack* head);//Out-stack functions
    Stack* initStack();//Functions that initialize the stack
    int IsEmpty(Stack*);//Determine if a stack is empty
    element PeekStack(Stack*);//Look at the top element of the stack but don't pop it out
    int ReturnTheValueofOp(char*);//Returns the priority of this operator
    int IsLetter(char object);//Determine whether it is a variable or not
    int IsOprator(char object);//Determines whether it is an operator
    int IsNumber(char object);//Determine if it's a number
    Tree* BuildTree(element* input);//A function that creates a tree
    Tree* initTree();//Initialize the function of the tree
    Tree* PushTreeStack(Tree*,Tree*);//The press-in stack function during the tree creation process
    Tree* PopTreeStack(Tree* head);//The ejection function in the process of creating a tree
    Tree* GradForTree(element* letter,Tree* root);//
```

```
    Tree* Derivative(Tree* node,char* letter,int length);//The main derivative function
    Tree* copy(Tree* input);//Duplicate the entire tree
    Tree* create_multiply_node(Tree* left,Tree* right);//An operational function for the derivative rule of multiplication
    Tree* create_divided_node(Tree* Deriva,Tree* needed,Tree* right,int index);//An operational function for dividing derivative rules
    Tree* create_cifang_node(Tree* input);//For operations on the square of the denominator in the derivative of division
    Tree* create_multimins_node(Tree* left,Tree* right);//For the power of the derivation rule
    Tree* create_ln_node(Tree* input);//Create logarithmic nodes
    int contains_variable(Tree* node, char* letter,int length);//Determine if there are any variables in the subtree that you want to derive
    void inorderTraversalSimplify(Tree* root,int count);//When exporting, use the mid-order traversal to simplify at the same time
    Tree* create_zero_node();//Zero in on some things that don't need to be used
    void DeleteString(char* front,char* behind);//Delete some of the excess parts
    void DeletepreString(char* front,char* behind);
    void DeletepostString(char* front,char* behind);
    int ContainsZero(char* input);//Check whether there is a 0 that needs to be deleted
    char* SimplyfiyZero(char* input);//Delete the part with 0
    void DeleteMulString(char* front);//Delete the part containing *0
    Tree* CreateDoubleLeft(Tree* input);
    Tree* CreateDoubleRight(char* letter);//Supplements for special cases
    int DetectpreString(char* input,char*behind);
    int DetectpostString(char* input,char*behind);//Delete by the position

    int DetectpreString(char* input,char*behind)
    {
        int frontlength = strlen(input);
        int behindlength = strlen(behind);
```

```
        int i,j,k;

        for(i = 0 ; i < frontlength;i++){
            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
            {
                return 1;
            }
        }

        return 0;
    }

    int DetectpostString(char* input,char*behind)
    {
         int frontlength = strlen(input);
        int behindlength = strlen(behind);
        int i,j,k;

        for(i = 0 ; i < frontlength;i++){
            if (strncmp(input + i, behind, behindlength) == 0 &&(IsOprator(input[i-1])==1||i == 0))
            {
                return 1;
            }
        }

        return 0;
    }

    int main()
    {
        char* input = (char*)malloc(sizeof(char)*MAX+1);
        int i,j;

        scanf("%s",input);

        CleanSpace(input);

        element* head ;
```

```
    head = TurnInputIntoElement(input);//Convert the input into an element element

    head = TurnElementIntoSuffix(head);//Convert an infix expression to a suffix expression

    Tree* root = BuildTree(head);//A tree is created from a suffix expression

    int count = 0;
    int flag = 1;//The setting indicates whether the derivative has been found for the amount of change

    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

    for(i = 0 ; head[i].element_type!=0; i++){//Traverse the entire input element to find the variables in it to find the derivative
        for(j = 0 ; j < i;j++){
            if(strcmp(head[i].store,head[j].store)==0){
                flag = 0;//Prevent multiple derivatives for a variable
            }
        }
            if(head[i].element_type == 1&&flag == 1){
                printf("%s:",head[i].store);
                int length = (int)strlen(head[i].store);//This length is the length of the character that the variable occupies
                gen[count] = Derivative(root,head[i].store,length);
                inorderTraversalSimplify(gen[count],count);

                //Directly remove everything from the string level to simplify it
                while(DetectpreString(output[count],"+0")==1||DetectpreString(output[count],"*1")==1){//Use a function to delete the content
                    DeletepreString(output[count],"+0");
                    DeletepreString(output[count],"*1");
```

```
                }

                while(DetectpostString(output[count],"0+")==1||DetectpostString(output[count],"1*")==1){//Use a function to delete the content
                    DeletepostString(output[count],"0+");
                    DeletepostString(output[count],"1*");
                }

                DeleteMulString(output[count]);//Partial simplification of multiplication
                printf("%s\n", (output[count]));//output
                count++;//Records are the first few variables
                IndexOutput = 0;
            }
        flag = 1;//Re-set the flag
    }



    return 0;
}

void CleanSpace(char* str)//Find a way to remove spaces
{

    char* p = str;
    int i = 0;

    while(*p){
        if(*p!=' '){
            str[i++] = *p++;
        }
    }

    str[i] = '\0';

}

element* TurnInputIntoElement(char* input)
{
```

```
    element* head = (element*)malloc(sizeof(element)*MAX);
    int index = 0;
    while(*input!='\0'){//Read in characters one by one
        if(IsLetter(*input)==1){//If it's a character
            head[index].element_type = 1;
            int temp_for_count;
            for(temp_for_count = 0;IsLetter(*(input))==1;temp_for_count++){ //Fill in the string with a loop
                head[index].store[temp_for_count] = *input;
                input++;
            }
            head[index].store[temp_for_count] = '\0';//heal
        }else if(IsOprator(*input)== 1||(index != 0&&(*(input-1)== ')'||IsNumber(*(input-1))||IsLetter(*(input-1))))){//Deposit operator
            head[index].element_type = 2;
            head[index].store[0] = *input;
            head[index].store[1] = '\0';//heal
            input++;
        }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
            head[index].element_type = 3;
            int temp_for_count;
            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
                if(*input=='.'){
                    head[index].element_type = 4;//Added decimal judgment
                }
                head[index].store[temp_for_count] = *input;
                input++;
            }
            head[index].store[temp_for_count] = '\0';//heal
```

```
        }
        index++;
    }
    head[index].element_type = 0;//Changing the type to 0 at the end of the day indicates the end of the whole

    return head;
}

//Define a function, the parameter is an element pointer, and the return value is also an element pointer
element* TurnElementIntoSuffix(element* in)
{
    //Create an array of suffix expressions, allocate memory space with malloc, the size is the maximum number of elements
    element* postfix = (element*)malloc(sizeof(element)*MAX);
    //Create a stack to store operators
    Stack* temp = initStack();

    int length;

    length = 0 ;

    //Loop through the input elements until you encounter an element type of 0 as the end flag
    while(in->element_type!=0){
            if(in->element_type == 1 || in->element_type == 3||in->element_type == 4){//Not an operator
            postfix[length++] = *in; //Add the element to the postfix array
            in++; //Move to the next element
            }else if(in->element_type == 2 && in->store[0]=='('){ //If the element is a left parenthesis
            temp = PushStack (*in,temp); //Push it into the stack
            in++; //Move to the next element
            }else if(in->element_type == 2 && in->store[0]==')'){ //If the element is a right parenthesis
            while(PeekStack(temp).store[0]!='('){ //Pop the stack until you encounter a left parenthesis
```

```
                postfix[length++] = PopStack(temp); //Add the popped element to the postfix array
            }
            PopStack(temp); //Pop the left parenthesis
            in++; //Move to the next element
            }else if(in->element_type==2){ //If the element is an operator
                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
                    temp = PushStack(*in,temp); //Push the current operator into the stack
                }else{ //Otherwise
                    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
                        postfix[length++] = PopStack(temp); //Add the popped element to the postfix array
                    }
                    temp = PushStack(*in,temp); //Push the current operator into the stack
                }
                in++; //Move to the next element
            }
    }

    while(IsEmpty(temp)!=1){ //Pop the remaining elements in the stack
        postfix[length++] = PopStack(temp); //Add them to the postfix array
    }

    return postfix; //Return the postfix array
}

int ReturnTheValueofOp(char*input)
{
    int result = 0;
    if(input[0]=='^'){
        result = 3;
```

```
    }else if(input[0]=='*'||input[0]=='/'){
        result = 2;
    }else if(input[0] == '+'||input[0] =='-'){
        result = 1;
    }else if(input[0] == ','){
        result = 0;
    }

    return result;
}

Stack* PushStack(element in,Stack*input)//Standard operation for press-into stacks
{
    Stack* cur = (Stack*)malloc(sizeof(Stack));
    Stack* temp;
    cur ->data = in;
    temp = input->next;
    input->next = cur;
    cur->next = temp;

    return input;
}

element PopStack(Stack* head)//Regular ejection operations
{
    element result = (head->next)->data;
    head ->next = head ->next->next;

    return result;
}

element PeekStack(Stack*input)
{
     element result = input->next->data;

     return result;
}
//Define a function, the parameter is an element pointer, and the return value is a tree pointer
```

```
Tree* BuildTree(element* input)
{
    //Create an array of tree pointers, allocate memory space with malloc, the size is the maximum number of elements
    Tree** head = (Tree**)malloc(sizeof(Tree*)*MAX);
    //Initialize a variable to store the index of the top of the array
    int top = -1;

    //Loop through the input elements until you encounter an element type of 0 as the end flag
    while (input->element_type != 0){
        //If the element is an operand (number, variable, or constant)
        if((input)->element_type==1||(input)->element_type==3||(input)->element_type==4){
            //Create a new tree node and assign the element to its data field
            Tree* Node = initTree();
            Node->data = *input;
            //Push the node into the array
            head[++top] = Node;
        }else{ //If the element is an operator
            //Create a new tree node and assign the element to its data field
            Tree* Node = initTree();
            Node->data = *input;
            //Pop two nodes from the array and assign them to the right and left children of the current node
            Node->right = head[top--];
            Node->left = head[top--];
            //Push the current node into the array
            head[++top] = Node;
        }
        //Move to the next element
        input++;
    }

    //Return the root node of the tree, which is the last element in the array
    return head[top];
}

```

```
Stack* initStack()//Initialize the stack
{
    Stack* new = (Stack*)malloc(sizeof(Stack)*MAX);
    new->next = NULL;

    return new;
}

int IsEmpty(Stack*input)
{
    int result = 0;
    if(input->next ==NULL){
        result = 1;
    }

    return result;
}

int IsLetter(char object)
{
    int result = 0;

    if(object >= 'a'&& object <='z'){
        result = 1;
    }

    if(object >= 'A'&& object <='Z'){
        result = 1;
    }

    return result;
}

int IsOprator(char object)
{
    int result = 0;
    if(object=='+' ||object =='*'||object == '/'||object == '^'||object == ','||object=='('||object==')'){
        result = 1;
    }//Additional judgment is required
```

```
    return result;
}
int IsNumber(char object)
{
    int result = 0;
    if(object>='0'&&object<='9'){
        result = 1;
    }
    return result;
}
Tree* initTree()
{
    Tree* node = (Tree*)malloc(sizeof(Tree));
    node->left = NULL;
    node->right = NULL;

    return node;
}

Tree* copy(Tree* input)//Create a duplicate version so that the original tree is not changed
{
    if(input == NULL){
        return NULL;
    }

    Tree* new_node = (Tree*)malloc(sizeof(Tree));

    new_node->data = input->data;
    new_node->left = copy(input->left);
    new_node->right = copy(input->right);

    return new_node;
}

Tree* create_multiply_node(Tree* left,Tree* right)
{
    Tree* new_node = (Tree*)malloc(sizeof(Tree));
    new_node->data.element_type = 2;
    new_node->data.store[0] = '*';
```

```
    new_node->left = left;
    new_node->right = right;

    return new_node;
}
//Define a function, the parameters are four tree pointers, and the return value is a tree pointer
Tree* create_divided_node(Tree* Deriva,Tree* needed,Tree* right,int index)
{
    //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
    Tree* new_node = (Tree*)malloc(sizeof(Tree));
    new_node->data.element_type = 2;
    new_node->data.store[0] = '/';

    //Create a new node for the left child of the root node, allocate memory space with malloc, and assign the '*' operator to its data field
    Tree* new_node_left= (Tree*)malloc(sizeof(Tree));
    new_node_left->data.element_type = 2;
    new_node_left->data.store[0] = '*';

    //Assign the first parameter (Deriva) to the left child of the new node left, which is the derivative of the numerator of the original function
    new_node_left->left = Deriva;

    //Create a temporary pointer to traverse the tree
    Tree* temp = (Tree*)malloc(sizeof(Tree));
    temp = Deriva;

    //Find the leftmost node of the Deriva tree
    while(temp->left!=NULL){
        temp = temp->left;
    }

    //Create a new node for the left parenthesis, allocate memory space with malloc, and assign the '(' character to its data field
    Tree* Left = (Tree*)malloc(sizeof(Tree));
    Left->data.element_type = 3;
```

```
        Left->data.store[0] = '(';
        Left->left = NULL;
        Left->right = NULL;

        //Assign the left parenthesis node to the left child of the leftmost node of the Deriva tree
        temp->left = Left;

        //Assign the second parameter (needed) to the right child of the new node left, which is the denominator of the original function
        new_node_left->right = needed;

        //Assign the new node left to the left child of the root node
        new_node->left = new_node_left;

        //Create a new node for the right child of the root node, allocate memory space with malloc, and assign the '^' operator to its data field
        Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
        new_node_right->data.element_type = 2;
        new_node_right->data.store[0] = '^';
        //Assign the third parameter (right) to the left child of the new node right, which is the denominator of the original function
        new_node_right->left = right;

        //Create a new node for the right child of the new node right, allocate memory space with malloc, and assign the '2)' characters to its data field
        Tree* new_node_right_right = (Tree*)malloc(sizeof(Tree));
        new_node_right_right->data.element_type = 3;
        new_node_right_right->data.store[0] = '2';
        new_node_right_right->data.store[1] = ')';

        //Set the left and right children of the new node right right to NULL
        new_node_right_right->left = NULL;
        new_node_right_right->right = NULL;
        //Assign the new node right right to the right child of the new node right
        new_node_right->right = new_node_right_right;
```

```

    //Assign the new node right to the right child of the root node
    new_node->right = new_node_right;

    //Return the root node of the quotient rule tree
    return new_node;
}

Tree* create_cifang_node(Tree* input)
{
    Tree* new_node = (Tree*)malloc(sizeof(Tree));
    Tree* new_node_right = (Tree*)malloc(sizeof(Tree));

    new_node->data.element_type = 2;
    new_node->data.store[0] = '^';
    new_node->left = input;

    new_node_right->data.element_type = 3;
    new_node_right->data.store[0] = '2';
    new_node_right->left = NULL;
    new_node_right->right = NULL;

    new_node->right = new_node_right;

    return new_node;
}

//Define a function, the parameters are two tree pointers, and the return value is also a tree pointer
Tree* create_multimins_node(Tree* left,Tree* right)
{
    //Create a new node for the root of the power rule tree, allocate memory space with malloc, and assign the '^' operator to its data field
    Tree* new_node = (Tree*)malloc(sizeof(Tree));
    new_node->data.element_type = 2;
    new_node->data.store[0] = '^';
    //Assign the first parameter (left) to the left child of the root node, which is the base of the original function
    new_node->left = left;
```

```
    //Create a new node for the right child of the root node, allocate memory space with malloc, and assign the '-' operator to its data field
    Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
    new_node_right->data.element_type = 2;
    new_node_right->data.store[0] = '-';

    //Create a new node for the left child of the new node right, allocate memory space with malloc, and assign the '(' character and the second parameter (right) to its data field
    Tree* new_node_right_left = (Tree*)malloc(sizeof(Tree));
    new_node_right_left->data.element_type = 1;
    new_node_right_left->data.store[0] = '(';
    int i;

    //Copy the data of the right parameter to the new node right left's data field, starting from the second position
    for(i = 0; right->data.store[i]!='\0';i++){
        new_node_right_left->data.store[i+1] = right->data.store[i];
    }

    //Assign the left and right children of the right parameter to the left and right children of the new node right left
    new_node_right_left->left = right->left;
    new_node_right_left->right = right->right;

    //Assign the new node right left to the left child of the new node right, which is the exponent of the original function
    new_node_right->left = new_node_right_left;

    //Create a new node for the right child of the new node right, allocate memory space with malloc, and assign the '1)' characters to its data field
    Tree* new_node_right_right = (Tree*)malloc(sizeof(Tree));
    new_node_right_right->data.element_type = 3;
    new_node_right_right->data.store[0] = '1';
    new_node_right_right->data.store[1] = ')';
```

```
    //Set the left and right children of the new node right right to NULL
    new_node_right_right->left = NULL;
    new_node_right_right->right = NULL;

    //Assign the new node right right to the right child of the new node right, which is the constant to be subtracted from the exponent
    new_node_right->right = new_node_right_right;
    //Assign the new node right to the right child of the root node, which is the new exponent of the power rule
    new_node->right = new_node_right;

    //Return the root node of the power rule tree
    return new_node;
}

//Define a function, the parameter is a tree pointer, and the return value is also a tree pointer
Tree* create_ln_node(Tree* input)
{
    //Create a new node for the root of the logarithm tree, allocate memory space with malloc, and assign the input's data to its data field
    Tree* new_node = (Tree*)malloc(sizeof(Tree));
    new_node->data.element_type = 2;
    new_node->data.store[0] = input->data.store[0];

    //Create a new node for the left child of the root node, allocate memory space with malloc, and assign the 'ln(' characters to its data field
    Tree* new_node_left = (Tree*)malloc(sizeof(Tree));
    new_node_left->data.element_type = 2;
    new_node_left->data.store[0] = 'l';
    new_node_left->data.store[1] = 'n';
    new_node_left->data.store[2] = '(';
    new_node_left->left = NULL;
    new_node_left->right = NULL;

    //Create a new node for the right child of the root node, allocate memory space with malloc, and assign the ')' character to its data field
    Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
```

```
    new_node_right->data.element_type = 2;
    new_node_right->data.store[0] = ')';
    new_node_right->left = NULL;
    new_node_right->right = NULL;

    //Assign the new node left to the left child of the root node
    new_node->left = new_node_left;
    //Assign the new node right to the right child of the root node
    new_node->right = new_node_right;

    //Return the root node of the logarithm tree
    return new_node;

}

Tree* Derivative(Tree* node,char* letter,int length)
{
    if(node == NULL){
        return NULL;
    }

    int i;
    char temp[20];

    for(i = 0 ; i < length;i++){
        temp[i] = letter[i];
    }

    temp[i] = '\0';

    Tree* new_node = (Tree*)malloc(sizeof(Tree));

    if(node->data.element_type == 2 && node->data.store[0] =='+'){
        new_node->data.element_type = 2;
        new_node->data.store[0] = '+';
        if(contains_variable(node->left,temp,length) == 1){
            new_node->left = Derivative(node->left,letter,length);
        }else{
```

```c
                new_node->left = create_zero_node();
            }
            if(contains_variable(node->right,temp,length)==1){
                new_node->right = Derivative(node->right,letter,length);
            }else{
                new_node->right = create_zero_node();
            }

        }else if(node->data.element_type == 2&& node->data.store[0] == '-'){
            new_node->data.element_type = 2;
            new_node->data.store[0] = '-';
            if(contains_variable(new_node->left,letter,length)==1){
                new_node->left = Derivative(node->left,letter,length);
            }else{
                new_node->left = create_zero_node();
            }
            if(contains_variable(new_node->right,letter,length)==1){
                new_node->right = Derivative(node->right,letter,length);
            }else{
                new_node->right = create_zero_node();
            }
        }else if(node->data.element_type == 2&& node->data.store[0] == '*'){
            new_node->data.element_type = 2;
            new_node->data.store[0] = '+';
            if(contains_variable(node->left,letter,length)==0){
                Tree* new_node_left = (Tree*)malloc(sizeof(Tree));
                new_node_left->data.element_type = 3;
                new_node_left->data.store[0] ='0';
                new_node_left->left = NULL;
                new_node_left->right = NULL;
                new_node->left = new_node_left;
            }else{
                new_node->left = create_multiply_node(Derivative(node->left,letter,length), copy(node->right));
            }
```

```
        if(contains_variable(node->right,letter,length)==0){
            Tree* new_node_right = (Tree*)malloc(sizeof(Tree));
            new_node_right->data.element_type = 3;
            new_node_right->data.store[0] ='0';
            new_node_right->left = NULL;
            new_node_right->right = NULL;
            new_node->right = new_node_right;
        }else{
            new_node->right = create_multiply_node(Derivative(node->right,letter,length), copy(node->left));
        }
    }else if(node->data.element_type == 2 && node->data.store[0] == '/'){
        new_node->data.element_type = 2;
        new_node->data.store[0] = '-';
        new_node->left = create_divided_node(copy(Derivative(node->left,letter,length)),copy(node->right),copy(node->right),0);
        new_node->right = create_divided_node(copy(Derivative(node->right,letter,length)),copy(node->left),copy(node->right),1);
    }else if(node->data.element_type == 2 && node->data.store[0] == '^'){
        new_node->data.element_type = 2;
        if(strcmp(temp,node->left->data.store)==0 &&strcmp(temp,node->right->data.store)==0){
            new_node->data.element_type = 2;
            new_node->data.store[0] = '*';
            new_node->left = CreateDoubleLeft(copy(node));
            new_node->right = CreateDoubleRight(letter);
        }
        else if(strcmp(temp,node->left->data.store)==0){
            new_node->data.element_type = 2;
            new_node->data.store[0] = '*';
            new_node->left = node->right;
            new_node->right = create_multimins_node(copy(node->left),copy(node->right));
        }else if(strcmp(temp,node->right->data.store)==0){
            new_node->data.element_type = 2;
```

```
                new_node->data.store[0] = '*';
                new_node->left = copy(node);
                new_node->right = create_ln_node(node->left);
            }else if(strcmp(temp,node->left->data.store)!=0&&strcmp(temp,node->right->data.store)!=0){
                new_node->data.element_type = 3;
                new_node->data.store[0] = '0';
                new_node->left = NULL;
                new_node->right = NULL;
            }
        }else if(node->data.element_type == 3||node->data.element_type == 4){
            new_node = node;
        }else if(node->data.element_type == 1&&strcmp(node->data.store,temp)!=0){
            new_node->data.element_type = 3;
            new_node->data.store[0] = '0';
            new_node->left = node->left;
            new_node->right = node->right;
        }else if(node->data.element_type == 1&&strcmp(node->data.store,temp)==0){
            new_node->data.element_type == 3;
            new_node->data.store[0] = '1';
            new_node->left = node->left;
            new_node->right = node->right;
        }

        return new_node;
    }

    int contains_variable(Tree* node, char* letter,int length){

        int i;
        char temp[20];

        for(i = 0 ; i < length;i++){
            temp[i] = letter[i];
        }
        temp[i] = '\0';

        if (node == NULL) {
            return 0;
        }
```

```c
    if ((node->data.element_type == 1||node->data.element_type == 3|| node->data.element_type ==4) && strcmp(node->data.store,temp)==0) {
        return 1;
    }
    return contains_variable(node->left, letter,length) || contains_variable(node->right, letter,length);
}

void inorderTraversalSimplify(Tree* root,int count)
{
        if(root != NULL){
            int i;

            inorderTraversalSimplify(root->left,count);
            for(i = 0;i < strlen(root->data.store);i++){
                output[count][IndexOutput++] = root->data.store[i];
            }
            inorderTraversalSimplify(root->right,count);
        }
}


Tree* create_zero_node()//Save the node with a derivative of 0
{
    Tree* new_node = (Tree*)malloc(sizeof(Tree));
    new_node->data.element_type = 3;
    new_node->data.store[0] = '0';
    new_node->left = NULL;
    new_node->right = NULL;

    return new_node;
}

void DeleteString(char* front,char* behind)
{
    int frontlength = strlen(front);
    int behindlength = strlen(behind);
    int i,j,k;
```

```

    for(i = 0 ; i < frontlength;i++){
        if (strncmp(front + i, behind, behindlength) == 0 &&(IsOprator(front[i-1])==1||i == 0)) //If you find the string you want to delete
        {
            for (j = i, k = i + behindlength; k < frontlength; j++, k++) //Move the elements behind you forward
            {
                front[j] = front[k];
            }
            front[j] = '\0'; //Set the end character
            break;
        }
    }
}
void DeletepreString(char* front,char* behind)
{
    int frontlength = strlen(front);
    int behindlength = strlen(behind);
    int i,j,k;

    for(i = 0 ; i < frontlength;i++){
        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
        {
            for (j = i, k = i + behindlength; k < frontlength; j++, k++) //Move the elements behind you forward
            {
                front[j] = front[k];
            }
            front[j] = '\0'; //Set the end character
            break;
        }
    }
}

void DeletepostString(char* front,char* behind)
{
    int frontlength = strlen(front);
```

```
        int behindlength = strlen(behind);
        int i,j,k;

        for(i = 0 ; i < frontlength;i++){
            if (strncmp(front + i, behind, behindlength) == 0 &&(IsOprator(front[i-1])==1||i == 0)) //If you find the string you want to delete
            {
                for (j = i, k = i + behindlength; k < frontlength; j++, k++) //Move the elements behind you forward
                {
                    front[j] = front[k];
                }
                front[j] = '\0'; //Set the end character
                break;
            }
        }
    }

    int ContainsZero(char* input)//Check whether the following node contains 0
    {
        int result = 0;
        int i;
        for(i = 0 ; i < strlen(input)-1;i++){
            if(input[i]=='0'&&(input[i+1]=='*'||input[i+1] == '/')){
                result = 1;
            }
            if((input[i] == '*'||input[i] == '/')&&input[i+1] == '0'){
                result = 1;
            }
        }

        return result;
    }

    char* SimplyfiyZero(char* input)//Simplify the number of nodes with 0
    {
        int index = 0;
```

```
    int i;
    int begin = 0;

    char* temp = (char*)malloc((strlen(input)+1));
    for(i = 0 ; i < strlen(input);i++){
        temp[i] = input[i];
    }

    for(i = 0;i < strlen(input);i++){
        if(input[i]=='+'||input[i]=='-'||input[i] == '\0'){
            char* sub = (char*)malloc(sizeof(char)*(i-begin+1));
            strncpy(sub,input+begin,i-begin+1);
            if(ContainsZero(sub)==1){
                DeleteString(temp,sub);
            }
            begin = i+1;
        }
    }
    return temp;

}
void DeleteMulString(char* front)
{
    int frontlength = strlen(front);
    int i,j,k;

    for(i = 0 ; i < frontlength;i++){
        if(((front[i] =='0')&&front[i+1]== '*'&&i == 0)||(front[i]=='*'&&(front[i+1]=='0')||((front[i]=='0')&&front[i+1]=='*'&&!IsNumber(front[i-1]))))
        {
            //Find the start and end positions of the multiplication expression
            int start = i-1; //Start with *0 or the previous character of 0*
            int end = i+2; //Start with *0 or the last character of 0* and Look backwards
            while(start >= 0 && front[start] != '+' && front[start] != '-') //If you don't encounter a plus or minus sign, keep looking
            {
```

```
                start--;
            }
            while(end < frontlength && front[end] != '+' && front[end] != '-'&&front[end]!=')') //If you don't encounter a plus or minus sign, keep looking backwards
            {
                end++;
            }
            //Move the following elements forward, covering the entire multiplication expression
            for (j = start, k = end; k < frontlength; j++, k++)
            {
                front[j] = front[k];
            }
            front[j] = '\0';
            frontlength = strlen(front); //Update the string length
            break;
        }
    }
}

Tree* CreateDoubleLeft(Tree* input)
{
    Tree* new_node = input;

    return input;
}

Tree* CreateDoubleRight(char* letter)//Handling of special cases to the second side
{
    Tree* new_node = (Tree*)malloc(sizeof(Tree));
    Tree* new_node_left = (Tree*)malloc(sizeof(Tree));
    Tree* new_node_right = (Tree*)malloc(sizeof(Tree));

    new_node->data.element_type = 2;
    new_node->data.store[0] = '+';

    new_node_left->data.element_type = 3;
    new_node_left->data.store[0] = '(';
```

```
        new_node_left->data.store[1] = '1';
        new_node_left->left = NULL;
        new_node_left->right = NULL;

        new_node_right->data.element_type = 1;
        new_node_right->data.store[0] = 'l';
        new_node_right->data.store[1] = 'n';
        new_node_right->data.store[2] = '(';
        int i;
        for(i = 0;letter[i]!='\0';i++){
            new_node_right->data.store[i+3] = letter[i];
        }
        new_node_right->data.store[i+3] = ')';
        new_node_right->data.store[i+4] = ')';
        new_node_right->left = NULL;
        new_node_right->right = NULL;

        new_node->left = new_node_left;
        new_node->right = new_node_right;

        return new_node;
    }
```

Declaration

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