Autograd for Algebraic Expressions

LU Jialiang

Thesis submitted to
Zhejiang University

in partial fulfilment for the course

Fundamental Data Structure
Computer Science

College of Information Science & Electronic Engineering
https://github.com/Riverlu233

March 26, 2025

Contents

Chapter 1 Introduction

1.1 Problem Description

The algorithm includes an input, which is an in-order expression that includes operators, variables and constants.It demands an output which contains the in-order expressions of the derivative of the original expression with respect to each variable.

1.2 Background

The usual methods of Numerical Analysis always return the value of the derivative of a function at some point, but this usually involves a lot of floating-point arithmetic and is less flexible. For example, it would be difficult to get the value of the second order derivative directly from the value of the first order derivative. But if we can get the symbolic expression of the derivative function of the original one at a relatively low cost, then we can avoid a lot of floating-point arithmetic and have a substantial increase in the flexibility with which we can approach the problem.

Chapter 2 Algorithm Specification

Bonus: This algorithm does not use the complex containers provided in the STL and provides at least 2 simplifications. However, this algorithm only supports basic operator derivation.

2.1 The Read-in Algorithm

Main Data Structure: Binary-Tree; Stack
The goal of the input section is to construct an expression tree that conforms to the original order of operations and whose nodes are classified. The construction process makes use of two stacks, and follows specific rules.
When building the tree structure, in addition to the addresses of the left and right subtrees, we require it to contain a variable indicating the meaning of the data stored in the tree, and a space for storing its values. Here we use an integer value to classify the meaning of the data stored and an array of strings to store the values.During input, either operators, variables or constants are all stored as a tree node.
Both stacks accept data types that are pointers to nodes of the tree, which means that each pointer in the stack points to a subtree. But one of the two stacks is used to store operator nodes(We name it stack-op) and the other store variable/constant nodes(We name it stack-var).
The constructing process follows the following rules:

- Variables or constants will be pushed directly into the stack-variant
- When the current operator has a higher priority than the top-of-stack-op operator, push it directly into the stack-op.
- When the current operator has a lower priority than the top-of-stack-op operator or both have equal priority, the top-of-stack-op operator is popped until the top-of-stack-op operator has a lower priority than the current operator or the stack-op is empty.

- When an operator goes off the stack-op, the variable stack-var pop out one element as its right subtree and one more element as its left subtree, and then push the new tree into the stack-var.
- When left bracket is encountered, do expression tree construction for the contents up to the right bracket and push it to the stack-var.

Pseudo-Code:

```
function parse_expression(end_char):
    operand_stack = init_stack()
    operator_stack = init_stack()
    current_token = ""

    while (ch = get_next_char())  end_char:
        if is_operator(ch):
            # Flush accumulated token
            if current_token  "":
                operand_stack.push(create_operand_node(current_token))
                current_token = ""

            # Process operator precedence
            while not operator_stack.empty() and get_priority(ch) <=
                get_priority(operator_stack.top().value[0]):
                build_subtree(operator_stack, operand_stack)

            operator_stack.push(create_operator_node(ch))

        elif ch == '(':
            # Recursive call for sub-expression
            operand_stack.push(parse_expression(')'))

        else:
            # Accumulate token characters
            current_token += ch

    # Process remaining token
    if current_token  "":
        operand_stack.push(create_operand_node(current_token))

    # Build final tree
    while not operator_stack.empty():
        build_subtree(operator_stack, operand_stack)

    return operand_stack.pop()

# Helper function: build operator subtree
function build_subtree(operator_stack, operand_stack):
    operator_node = operator_stack.pop()
    operator_node.right = operand_stack.pop()
    operator_node.left = operand_stack.pop()
```

```
operand_stack.push(operator_node)
```

2.2 The Derivative Algorithm

Main Data Structure: Binary-Tree
Since the data stored in the tree nodes is in the form of strings, trying to compare it with the various function forms is very time consuming and greatly increases the difficulty of writing the programme. Therefore, we would like to make the derivation rules as simple and feasible as possible. Therefore, I decided to use the following symbols based on the operation of the derivation rule:

- Derivatives of constant and non-target variables are 0
- Derivatives of target variable is 1
- $(f(x) \pm g(x))' = f'(x) \pm g'(x)$
- $(f(x) \cdot g(x))' = f'(x)g(x) + f(x)g'(x)$
- $\left(\dfrac{f(x)}{g(x)}\right)' = \dfrac{f'(x)g(x) - f(x)g'(x)}{g^2(x)}$
- $\left(f(x)^{g(x)}\right)' = f(x)^{g(x)}\left(g'(x)\ln f(x) + g(x)\dfrac{f'(x)}{f(x)}\right)$

By means of the last four rules we can always reduce the problem to the first two rules, so that the derivation for operator nodes will be performed recursively by means of the last four rules, while the first two rules are set to recursive exit.The implementation of these derivation laws in the code is done by creating subtrees in the manner specified by the laws, i.e., it is straightforward to translate this code into expression trees. Moreover, the last three laws are a bit complicated, which makes the implementation code a bit long, so they are not described in the pseudo-code, and the reader can refer to Appendix A to get the specific implementation.

Pseudo-Code:

```
Differentiation:
  - derivative():
    if operator node:
      Case based on operator:
        +,-: Apply rule 3
        *:   Apply rule 4
        /:   Apply rule 5
        ^:   Apply rule 6
    else:
      return 1 if variable matches target, else 0 #(rule 1 & 2)
```

2.3 The Simplify Algorithm

Main Data Structure: Binary-Tree

One of the costs of using the above derivation rules is that there are a very large number of redundant terms, so we have to perform a larger number of simplifications in order to obtain a more concise expression. The author's main considerations in simplification are the direct merging of constants and the use of unit and inverse elements in the algebraic system for simplification.So I choose the following simplification rules:

- Operators that all the children of whom are constants can be merged.
- If one of the children of an operator is the zero element of that operator, it can be turned to zero element directly.
- One of the children of the operator is the unit element of that operator, then it can be merged.

Pseudo-Code:

```
def simplify(node):
    if not node: return
    simplify(node.left)
    simplify(node.right)
    if operator(node):
        if both_children_constants(node):
            fold_constants(node)
        if has_zero_identity(node):
            apply_zero_rules(node)
        if has_identity_operations(node):
            apply_identity_rules(node)

# Core rules condensed
def fold_constants(node):
    a, b = num(node.left), num(node.right)
    node.value = str(calc(a, b, node.op))
    node.set_as_constant()
    delete_children(node)

def apply_zero_rules(node):
    if (node.op == '+' and 0 in children) or \
       (node.op == '*' and 0 in children) or \
       (node.op == '^' and left=0):
        node.set_zero()
    if (node.op == '-' or node.op=='/') and left=right:
        node.set_zero()

def apply_identity_rules(node):
    if (node.op in ['*','/'] and 1 in children) or \
       (node.op == '^' and right=1):
        keep = left if right=1 else right
        node.replace_with(keep)
```

2.4 The Output Algorithm

Main Data Structure: Binary-Tree
The simplified expression tree is the result we need, but it still needs to be converted to the form of a in-order expression. To achieve this, we also need to introduce parentheses to protect arithmetic precedence in appropriate places when we output the expression. I introduces a function to compare the operator priority of an operator node with its children node(If it is an operator node), and the output needs to be protected when the child node has a low priority. In addition, since the output may contain the operator 'ln()' , which is not in the read-in operators, the output should also be handled specially.

Pseudo-Code:

```
function print_infix(node):
    if node exists:
        print_infix(node.left)
        print(node.value)
        print_infix(node.right)

# Postfix printing
function print_postfix(node):
    if node exists:
        print_postfix(node.left)
        print_postfix(node.right)
        print(node.value + ",") # Comma-separated format

# Parenthesized infix (core logic)
function generate_paren_infix(node, parent_priority=0, is_right_child=False) ->
    str:
    if not node:
        return ""

    if node is leaf: # Constant/variable
        return node.value

    current_priority = get_priority(node.value[0])
    left_expr = generate_paren_infix(node.left, current_priority, False)
    right_expr = generate_paren_infix(node.right, current_priority, True)

    # Wrap sub-expressions based on priority rules
    if needs_parenthesis(parent_priority, current_priority,
                         is_right_child, node.value[0]):
        return f"({left_expr} {node.value[0]} {right_expr})"
    else:
        return f"{left_expr} {node.value[0]} {right_expr}"

# Helper predicate
function needs_parenthesis(parent_p, current_p, is_right, op) -> bool:
    return (current_p < parent_p) or \
```

```
(current_p == parent_p and (
    (op is right-associative and not is_right) or
    (not right-associative and is_right)
))
```

Chapter 3 Testing Results

Instructions: Since the read-in algorithm is online, the full-memory scenario only occurs when the tree node does not have enough space for the string. The 'Largest Sizes' case tested refers to this situation.

3.1 The choice of the test cases

Based on the difference between the following two directions, we set up different test cases and the scenarios they test are shown in the header of the table.Since increasing the character storage space in the tree nodes simply allows for longer variable names, the space sizes here are all set to 4.

Case	Same priority	Different Priorities	Include Brackets
One-byte	$a+b-c$	$4*a-8^b$	$a(b-3)/(4+2*d)$
Mixed-size	$aa*bb/fg$	$45+ab*hf$	$2^{(bb+fg)}/bc$
Largest Size	aaa^{bbb}	$639-aaa^{bbb}$	$aaa^{(bbb/ccc)}-ddd$

3.2 Test Cases

See: Figure 1: Test Cases

Chapter 4 Analysis & Comments

Conclusions: The time complexity of the whole algorithm is $\Theta(N)$, while the space complexity is also $\Theta(N)$.

4.1 The Read-in Algorithm

4.1.1 Time Complexity

Although this read-in algorithm contains recursive parts, since the **getchar()** function does not backtrack, this algorithm is actually online. It performs the contents of the function **expression()** once for each character read in. The number of operations in one time is limited to at least 1, or at most a finite value independent of the amount of data N (after all, the function does not include loops). Thus we can easily conclude that the complexity of the algorithm is $\Theta(N)$.

Case	Input	Variant	Output	State
1	a+b-c	a	1	Pass
		b	1	
		c	-1	
2	4*a-8^b	a	4	Pass
		b	0-8^b*ln(8)	
3	a*(b-3)/(4+2*d)	a	(b-3)*(4+2*d)/(4+2*d)^2	Pass
		b	a*(4+2*d)/(4+2*d)^2	
		d	0-a*(b-3)*2/(4+2*d)^2	
4	aa*bb/fg	aa	bb*fg/fg^2	Pass
		bb	aa*fg/fg^2	
		fg	0-aa*bb/fg^2	
5	45+ab*hf	ab	hf	Pass
		hf	ab	
6	2^(bb+fg)/bc	bb	2^(bb+fg)*ln(2)*bc/bc^2	Pass
		fg	2^(bb+fg)*ln(2)*bc/bc^2	
		bc	0-2^(bb+fg)/bc^2	
7	aaa^bbb	aaa	aaa^bbb*bbb*1/aaa	Pass
		bbb	aaa^bbb*ln(aaa)	
8	639-aaa^bbb	aaa	0-aaa^bbb*bbb*1/aaa	Pass
		bbb	0-aaa^bbb*ln(aaa)	
9	aaa^(bbb/ccc)-ddd	aaa	aaa^(bbb/ccc)*(bbb/ccc)*1/aaa	Pass
		bbb	aaa^(bbb/ccc)*ccc/ccc^2*ln(aaa)	
		ccc	aaa^(bbb/ccc)*(0-bbb/ccc^2)*ln(aaa)	
		ddd	-1	

Figure 1: Test cases

4.1.2 Space Complexity

The upper bound on the space complexity of the built expression tree is clearly $O(N)$. It's because in addition to the space of constants needed in the algorithm, if every byte of input exists in a tree node (meaning that all variables, constants are one byte long), there are as many byte inputs as there are nodes of the tree, while the size of one node is constant. But the lower bound is relatively more complicated; it is actually related to the maximum word length of the variables that can be entered. If we assume that the maximum word length is m, then since all of them are binary operators and the length of the operators are all 1 byte, the number of operators is at least $N/(m+1)$. And the number of tree nodes at this time is $N/(m+1)+1$, which is also a linear function of N. Therefore, the lower bound is also $\Omega(N)$, and so the space complexity is actually $\Theta(N)$.

4.1.3 Optimization Direction

Firstly, the code in the input section is less readable compared to the derivation and simplification sections. I think this is because it involves string handling. Consideration should be given to encapsulating some of the repetitive operations, and trying to develop implementation criteria that are easy to categorise, which should improve the readability of this code.
In addition, since the input algorithm is online, this poses a huge problem in that it is very inconvenient to work with operators with multiple word lengths like log, cos, exp. If we move to a non-online input approach, we can more easily determine these multi-word operators at the cost of some complexity, though this makes the total length of the input limited by the size of the structure used to store it.

4.2 The Derivative Algorithm

4.2.1 Time Complexity

Obviously, the difference in complexity between the different operations varies wildly. We will find that the exact number of operations is mainly determined by the number and type of operators. According to the codes in Appendix A, the '+' and '-' signs correspond to 4 operations, the '*' sign corresponds to 8 operations, the '/' sign corresponds to 16 operations and the '^' sign corresponds to 15 operations, and all variable/constant nodes actually correspond to only 3 operations.Thus, the upper bound on time complexity occurs when all arguments have the length 1 byte, when the number of variables and operators are at their maximum, $(N+1)/2$ and $(N-1)/2$. At this point, if we then make all operators division signs, the operands will be $3(N+1)/2+8(N-1)$, which means the upper bound on time complexity is $O(N)$. The lower bound occurs when the length of all parameters is the maximum m and all operators are '+' or '-'. According to the discussion in the read-in algorithm, the number of operators at this point is $N/(m+1)$ and the number of variables is $N/(m+1)+1$, which means the opperands are $2N/(m+1)+3(N/(m+1)+1)$ and the lower bound is also $\Omega(N)$. In summary, the time complexity of this algorithm is $O(N)$.

4.2.2 Space Complexity

Similarly to the time complexity analysis, the space increment is also mainly caused by the operators, and the corresponding space increment of each operator is also constant, so we can also get the space complexity of the algorithm is $\Theta(N)$.

4.2.3 Optimization Direction

Firstly, the code in the input section is less readable compared to the derivation and simplification sections. I think this is because it involves string handling. Consideration should be given to encapsulating some of the repetitive operations, and trying to develop implementation criteria that are easy to categorise, which should improve the readability of this code.
In addition, since the input algorithm is online, this poses a huge problem in that it is very inconvenient to work with operators with multiple word lengths like log, cos, exp. If we move to a non-online input approach, we can more easily determine these multi-word operators at the cost of some complexity, though this makes the total length of the input limited by the size of the structure used to store it.

4.3 The Simplify Algorithm

4.3.1 Time Complexity

The simplification is done recursively and starts from the bottom non-constant/variable node. For the same operator node, at most one simplification rule will be applied, while one simplification takes constant steps. Therefore, the maximum number of operations is performed when the operators are most and all need to be simplified. According to the analysis in the derivative algorithm, the operation at this point is $C*(N-1)/2$, and the upper bound of complexity is $O(N)$. Correspondingly, when no simplification is required at all, it is still nec-

essary to traverse each node, and the number of operations at this point is exactly N, so the lower bound of complexity is $\Omega(N)$. Therefore, the time complexity of this algorithm is $\Theta(N)$

4.3.2 Space Complexity

Since this function is actually an algorithm for simplifying expressions and only takes up constant space, the space complexity is $\Theta(1)$.

4.3.3 Optimization Direction

The simplification section actually has a lot of unimplemented features, for example, the inverse element in algebraic systems, the merging of like terms in polynomial theory, and the merging between exponential terms have not been introduced, and in addition there is no checking of the results, which are all directions for optimisation.

4.4 The Output Algorithm

4.4.1 Time Complexity

For the output algorithm, it actually consists of only two traversals, one for finding nodes that require bracket-protected priority, and the other for outputting the results. The number of operations performed during each traversal is also constant, so it is easy to get that its time complexity is $\Theta(N)$.

4.4.2 Space Complexity

In the same way as the previous algorithm, the space complexity of this algorithm is $\Theta(1)$.

4.4.3 Optimization Direction

The output section still contains some flaws, such as not following a certain order, as well as different parts of the same product term not following the dictionary order, in addition to some effects of incomplete simplification. If some adjustments can be made in the input section so that functions such as log(), cos(), exp() etc. are introduced, the ln function does not need to be considered specifically in the input section and can be handled modularly.

Chapter A Source Code (in C)

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

//Declaration of the used structs
typedef struct Tree{ //Binary-Tree
    int state; //indicate the type of the node:0-opperand,1-variant,2-constant
    int bracket_state; //to show whether it needs brackets when output
    struct Tree* left;
```

```
    struct Tree* right;
    char value[4]; //The space used to store values, whether they are operators,
        constants, or variables.
}tree;

typedef struct Stack{
    tree** content; //the stack stores the pointer of tree node
    int top;
    int capacity; //the capacity of the stack
}stack;

typedef struct Set{ //an array to store all the variables
    char **variants; //place to save the pointers of the variables
    int capacity;
    int count;    //number of the variables in the set
}set;

//All the operations of stacks that needed
stack* init_stack(void);
void  push_stack(stack* target_stack,tree* new_node);
tree* pop_stack(stack* target_stack);
int   empty_stack(stack* target_stack);
tree* check_stack(stack* target_stack);

//functions that build the tree
tree* expression(char end); //"end" is the letter that indicates the end of input
tree* init_variant(char*);
tree* init_opperand(char);
void state_tree(tree* node);
int get_priority(char op);

//functions used to collect the variables
void collect_variant(tree* node, set* set_var);
void add_variable(set* set_var,char* var);
set* init_set();

//function uesd to get the derivative
tree* derivative(tree* root,char *target_variant);
tree* derivative_plus(tree* root,char *target_variant);
tree* derivative_minus(tree* root,char *target_variant);
tree* derivative_multi(tree* root,char *target_variant);
tree* derivative_div(tree* root,char *target_variant);
tree* derivative_pow(tree* root,char *target_variant);

//function used to simplify the tree
void simplify_tree(tree*node);
void simplify_const(tree* node);
void simplify_zero(tree* node);
void simplify_identity(tree* node);
```

```
//some auxiliary functions
int int_pow(int a,int b); //to avoid using pow in math.h
void convert_zero(tree* root);
void convert_one(tree* node);
void replace(tree* node1,tree* node2);
void test_print_post(tree* leaf);

//Functions used to output
void test_print_in(tree* leaf);
void update_bracket(tree* node);

int main(){
    int i;
    tree* result;
    tree* express=expression('\0'); //Init the expression tree
    state_tree(express); //update the type of every node
    set* set_variant=init_set(); //Init a new set to save the variables
    collect_variant(express,set_variant); //Collect the new variables
    for (i=0;i<set_variant->count;i++){
        result=derivative(express,set_variant->variants[i]); //find the derivative
        state_tree(result); //renew the type of every node
        simplify_tree(result); //simplify the expression
        update_bracket(result); //find the nodes that needs brackets
        printf("%s : ",set_variant->variants[i]);
        test_print_in(result); //print the in-order expression
        printf("\n");
    }
}

stack* init_stack(){ //to establish a new stack
    stack* new_stack=(stack*)malloc(sizeof(stack));
    new_stack->top=0;
    new_stack->capacity=4; //The initial capacity is set to 4
    new_stack->content=(tree**)calloc(new_stack->capacity,sizeof(tree*));
    return new_stack;
}

void push_stack(stack* target_stack,tree* new_node){ //push new_node into stack
    if (target_stack==NULL){
        printf("The to-push stack doesn't exist.\n");
        exit(-1);
    }
    else{
        if (target_stack->top==target_stack->capacity){ //to expand the capacity if
            it is full
            target_stack->capacity*=2;
            tree**
                tmp=(tree**)realloc(target_stack->content,sizeof(tree*)*target_stack->capacity);
```

```
            if (tmp){
                target_stack->content=tmp;
            }
            else{
                printf("Failed to realloc the space.\n");
                exit(-1);
            }
        }
        target_stack->content[target_stack->top]=new_node;
        target_stack->top++;
        return;
    }
}

tree* pop_stack(stack* target_stack){ //pop out the top node
    if (target_stack==NULL){
        printf("The to-pop stack doesn't exist.\n");
        exit(-1);
    }
    else{
        if (target_stack->top==0){
            printf("The stack is empty.\n");
            exit(-1);
        }
        else{
            target_stack->top--;
            return target_stack->content[target_stack->top];
        }
    }
}

int empty_stack(stack* target_stack){ //to check whether the stack is empty
    if (target_stack==NULL){
        printf("The to-judge-empty stack doesn't exist.\n");
        exit(-1);
    }
    else{
        return target_stack->top==0 ? 1 : 0 ;
    }
}

tree* check_stack(stack* target_stack){ //to check the element that at the top of
    stack
    if (target_stack==NULL){
        printf("The to-check stack doesn't exist.\n");
        exit(-1);
    }
    else{
        if (target_stack->top){
```

```
            return target_stack->content[target_stack->top-1];
        }
        else{
            printf("The to-check stack is empty.\n");
            exit(-1);
        }
    }
}

void state_tree(tree* node){
    int i=0;
    if (node){
        node->bracket_state=0; //First initialize all to 0,we will update it by
            "update_bracket"
        if (node->state){
            node->state=2; //First assume they are all constant
            while (node->value[i]!='\0'&&i<4){
                if
                    (node->value[i]>='a'&&node->value[i]<='z'||node->value[i]>='A'&&node->value[i
                    //if one of the constant is a letter,it should be a
                        variable,eg'6a4'is variable
                    node->state=1;
                    break;
                }
                i++;
            }
        }
        state_tree(node->left); //update the type of the tree nodes recursively
        state_tree(node->right);
    }
    return;
}

void simplify_tree(tree* node){
    if (node){
        simplify_tree(node->left); //simplify the children node first
        simplify_tree(node->right); //so that we can avoid repetitive work

        simplify_const(node); //simplify rule 1
        simplify_zero(node); //simplify rule 2
        simplify_identity(node); //simplify rule 3
    }
    else{
        return;
    }
}

void simplify_const(tree* node){ //rule 1:if all children are constant, merge by
    operators.
```

```
    if (node->state==0){
        int a,b,result=0;
        char op;
        if (node->left->state+node->right->state==4) { //to judge all children are
            constant
            a=atoi(node->left->value); //change left->children->value to int number
            b=atoi(node->right->value); //likewise
            char op=node->value[0]; //get the operators
            switch(op) {
                case '+': result=a+b; break; //process according to op
                case '-': result=a-b; break;
                case '*': result=a*b; break;
                case '/': result=a/b; break;
                case '^': result=int_pow(a, b); break;
            }
            node->state = 2; //now the new node is a constant
            sprintf(node->value, "%d", result); //change the int result to char*
                and copy it to value
            node->left=NULL; //now a constant don't have children node
            node->right=NULL;
        }
    }
    return;
}

void simplify_zero(tree* node){ //rule 2:operate with zero element returns zero
    element
    if (node->state==0){
        char* left_value=node->left->value; //get the left&right children's value.
        char* right_value=node->right->value;
        char op=node->value[0]; //get the operator
        switch(op) {
            case '-':
                if
                    (strcmp(left_value,right_value)==0&&node->left->state!=0&&node->right->state
                    convert_zero(node); //special case: a-a=0
                    break;
            case '*':
                if (strcmp(left_value,"0")==0||strcmp(right_value,"0")==0)
                    convert_zero(node); //zero element of * is 0
                break;
            case '/':
                if (strcmp(left_value,"0")==0)
                    convert_zero(node); //left-zero-element of / is 0
                else
                    if(strcmp(left_value,right_value)==0&&node->left->state!=0&&node->right->stat
                    convert_one(node); //special case:a/a=1
                break;
            case '^':
```

```
            if (strcmp(left_value,"0")==0)
                convert_zero(node); // special case:0^a=0
            else if (strcmp(right_value,"0")==0||strcmp(left_value,"1")==0)
                convert_one(node); //zero-element of ^ is 1 and special case:1^a=1
            break;
        }
    }
    return;
}

void simplify_identity(tree* node){ //rule 3:operate with unit-element is identtity
    if (node->state==0){
        char* left_value=node->left->value; //get the left&right children's value.
        char* right_value=node->right->value;
        char op=node->value[0]; //get the operator
        switch(op) {
            case '+':
                if (strcmp(left_value,"0")==0)
                    replace(node,node->right); //unit-element of + is 0
                else if (strcmp(right_value,"0")==0)
                    replace(node,node->left);
                break;
            case '-':
                if (strcmp(right_value,"0")==0) //right-unit-element of - is 0
                    replace(node,node->left);
                break;
            case '*':
                if (strcmp(left_value,"1")==0)
                    replace(node, node->right); //unit element of * is 1
                else if (strcmp(right_value,"1")==0)
                    replace(node, node->left);
                break;
            case '/':
                if (strcmp(right_value,"1")==0) //right-unit-element of / is 1
                    replace(node, node->left);
                break;
            case '^':
                if (strcmp(right_value,"1")==0) //right-unit-element of ^ is 1
                    replace(node, node->left);
                break;
        }
    }
    return;
}

void convert_zero(tree* root){ //change a node to constant 0
    root->left=NULL;
    root->right=NULL;
    strcpy(root->value,"0");
```

```
    root->state=2;
}

void convert_one(tree* node){ //change a node to constant 1
    node->left=NULL;
    node->right=NULL;
    node->state=2;
    strcpy(node->value,"1");
}

void replace(tree* node1,tree* node2){ //replace node 1 by node 2
    node1->left=node2->left;
    node1->right=node2->right;
    node1->state=node2->state;
    strcpy(node1->value,node2->value);
}

int int_pow(int a,int b){ //a function to return int a^b
    int i,mul=1;     //avoid the floating calculate of pow() in <math.h>
    for (i=0;i<b;i++){
        mul*=a;
    }
    return mul;
}

void test_print_in(tree* leaf){ //output the expression
    if (leaf){ //when meet function "ln()",we process specially
        if (leaf->state==0&&leaf->value[0]=='l'){
            printf("ln(%s)",leaf->right->value);
        }
        else{
            if (leaf->bracket_state){ //if it need a bracket,before printing its
                left side,print a (
                printf("(");
            }
            test_print_in(leaf->left); //as the operator are binoncular,we print
                left element first
            printf("%s",leaf->value); //i.e. print in-order traversal
            test_print_in(leaf->right);
            if (leaf->bracket_state){ //if it need a bracket,before printing its
                left side,print a )
                printf(")");
            }
        }
    }
    return;
}
void test_print_post(tree* leaf){ //a function just for test, using to print
    post-order traversal
```

```
    if (leaf){
        test_print_post(leaf->left);
        test_print_post(leaf->right);
        printf("%s %d,",leaf->value,leaf->bracket_state);
    }
    return;
}

void update_bracket(tree* node){ //to judge whether a operator needs bracket
    if (!node) return;
    if (node->state==0){
        if (node->left->state==0||node->right->state==0){ //if one of the children
            is operator
            if
                (node->left->state==0&&(get_priority(node->left->value[0])<get_priority(node->va
                //if priority of thr left operator is lower ,then it needs brackets.
                node->left->bracket_state=1;
            }
            if
                (node->right->state==0&&(get_priority(node->right->value[0])<get_priority(node->
                //if priority of thr right operator is lower ,then it needs brackets.
                node->right->bracket_state=1;
            }
        }
        update_bracket(node->left); //update the state recursively
        update_bracket(node->right);
    }
    return;
}

int get_priority(char op) { //return the priority of the operator
    switch(op) {
        case '+':
        case '-': return 1; //prior 1:+ & -
        case '*':
        case '/': return 2; //prior 2:* & /
        case '^': return 3; //prior 3:^
        default: return 0; //not an operator
    }
}

tree* expression(char end){ //end is the letter that indicate the end of input
    int i=0; //index for temp
    char ch='0'; //to store the letter that read in
    char temp[5]; //a temporary space for those long variables.
    tree* temp_tree; //store a tree* variable temporarily
    stack* stack_variant=init_stack(); //init a stack to store the variables
    stack* stack_opperand=init_stack(); //init a stack to store the operators
    while ((ch=getchar())!='\n'&&ch!=end){ //if the input didn't end
```

```
if (get_priority(ch)){ //if the readin letter is an operator
    if (i!=0){
        //Sometimes when we readin an operator,the former variable maybe
            didn't exist.
        //eg a*(b-1)/c,when / is read in,there's no former varaible
        //to avoid a empty temp is turned to node ,we judge whether i==0
        temp[i]='\0';i=0; //when temp is not empty,we can turn it to a new
            node
        tree* new_variant=init_variant(temp);
        //right now we don't now the type,just know it is an operator
        push_stack(stack_variant,new_variant);
        //push the new node into the stack
    }
    tree* new_opperand=init_opperand(ch); //turn the new operator into a
        operator type node
    //as you can see,all the operators will be update their type here,so
        "state_tree" will skip them
    if (empty_stack(stack_opperand)){ //if the operator stack is empty,just
        push it in
        push_stack(stack_opperand,new_opperand);
    }
    else{
        //if the stack is not empty
        if
            (get_priority(ch)<=get_priority(check_stack(stack_opperand)->value[0])){
            /*When the current operator has a lower priority than the
                top-of-stack-op operator or both have
            equal priority, the top-of-stack-op operator is popped until the
                top-of-stack-op operator has a
            lower priority than the current operator or the stack-op is
                empty.*/
            while
                (!empty_stack(stack_opperand)&&get_priority(ch)<=get_priority(check_stack
                temp_tree=pop_stack(stack_opperand);
                temp_tree->right=pop_stack(stack_variant);
                temp_tree->left=pop_stack(stack_variant);
                push_stack(stack_variant,temp_tree);
            }
        }
        //righr now push the new operator into stack
        push_stack(stack_opperand,new_opperand);
    }
}
else if (ch=='('){
    /*When left bracket is encountered, do expression tree construction for
        the contents up to
    the right bracket and push it to the stack-var.*/
    push_stack(stack_variant,expression(')')); //to control it stop at the )
}
```

```
        else{
            //nothing happened,it is a longer variable
            temp[i++]=ch;
        }
    }
    //when the read-in finished,all things should be popped out
    if (i!=0){
        //samely,if temp isn't empty,then it stores the last variable,we should
            push it into the stack
        temp[i]='\0';
        tree* new_variant=init_variant(temp);
        push_stack(stack_variant,new_variant);
    }
    while (!empty_stack(stack_opperand)){ //now we should empty all the stacks
        /*When an operator goes off the stack-op, the variable stack-var pop out
            one element as its
        right subtree and one more element as its left subtree, and then push the
            new tree into the
        stack-var.*/
        temp_tree=pop_stack(stack_opperand);
        temp_tree->right=pop_stack(stack_variant);
        temp_tree->left=pop_stack(stack_variant);
        push_stack(stack_variant,temp_tree);
    }
    return pop_stack(stack_variant);
    //finally only the pointer of the root will remain in the stack_variable
}

tree* init_variant(char *variant){ //init a new node with variant/constant,here we
    don't distinguish them
    tree* leaf=(tree*)malloc(sizeof(tree));
    leaf->state=1; //just assume it is a variable
    leaf->left=NULL;
    leaf->right=NULL;
    strcpy(leaf->value,variant);
    return leaf;
}

tree* init_opperand(char op){ //init a new node with operator
    tree* leaf=(tree*)malloc(sizeof(tree));
    leaf->state=0; //it must be an operator
    leaf->left=NULL;
    leaf->right=NULL;
    leaf->value[0]=op;
    leaf->value[1]='\0';
    return leaf;
}

set* init_set(){ //init the set to save the variables
```

```
    set* set_var=malloc(sizeof(set));
    set_var->capacity=8; //initial capacity is 8
    set_var->count=0; //init it to be empty
    set_var->variants=malloc(sizeof(char*)*set_var->capacity); //init the char*
        pointers array
    return set_var;
}

void add_variable(set* set_var,char* var){ //to add new variables into the set
    int i;
    for (i=0;i<set_var->count;i++) { //if the variable has appeared,no need to
        continue
        if (strcmp(set_var->variants[i],var)==0) {
            return;
        }
    }
    //if the variable hasn't appeared before
    if (set_var->count>=set_var->capacity) { //if it is full,expand
        set_var->capacity*=2;
        set_var->variants=realloc(set_var->variants,sizeof(char*)*set_var->capacity);
    }
    set_var->variants[set_var->count]=(char*)malloc(sizeof(char)*4); //allocate a
        space to store the new
    strcpy(set_var->variants[set_var->count],var);//copy the new variable into it
    set_var->count++;
}

void collect_variant(tree* node, set* set_var) { //collect the variables in the
    expression
    if (!node){
        return;
    }
    else if (node->state == 1) { //if it is a variable,check whether it need to be
        add
        add_variable(set_var, node->value);
    }
    collect_variant(node->left, set_var); //collect the variabls recursively
    collect_variant(node->right, set_var);
}


tree* derivative(tree* root,char *target_variant){ //derivative the expression
    tree* new=NULL;
    if (!root->state){ //if it is a operator
        char op = root->value[0];
        switch(op){
            //case the op to derivative them
            case '+': new=derivative_plus(root,target_variant);break;
            case '-': new=derivative_minus(root,target_variant);break;
```

```
            case '*': new=derivative_multi(root,target_variant);break;
            case '/': new=derivative_div(root,target_variant);break;
            case '^': new=derivative_pow(root,target_variant);break;
            default: break;
        }
    }
    else{
        if (strcmp(root->value,target_variant)==0){
            //the derivative of target_variant=1
            new=init_variant("1");
            new->state=2;
        }
        else{
            //the derivative of non-target_variant or constant is 0
            new=init_variant("0");
            new->state=2;
        }
    }
    return new;
}

tree* derivative_plus(tree* root,char *target_variant){
    //+,-: Derive both children
    tree* new=init_opperand('+');
    new->left=derivative(root->left,target_variant);
    new->right=derivative(root->right,target_variant);
    return new;
}

tree* derivative_minus(tree* root,char *target_variant){
    //+,-: Derive both children
    tree* new=init_opperand('-');
    new->left=derivative(root->left,target_variant);
    new->right=derivative(root->right,target_variant);
    return new;
}

tree* derivative_multi(tree*root,char *target_variant){
    //*: Apply product rule (f'g + fg')
    tree* new=init_opperand('+');
    //Initialize left side
    new->left=init_opperand('*');
        new->left->left=derivative(root->left,target_variant);
        new->left->right=root->right;
    //Initialize right side
    new->right=init_opperand('*');
        new->right->left=root->left;
        new->right->right=derivative(root->right,target_variant);
    return new;
```

```
}

tree* derivative_div(tree *root,char *target_variant){
    //Apply quotient rule [(f'g - fg')/g²]
    tree* new=init_opperand('-');
    //Initialize left side
    new->left=init_opperand('/');
        //Init left->left,namely deri(divident)*divisor
        new->left->left=init_opperand('*');
            new->left->left->left=derivative(root->left,target_variant);
            new->left->left->right=root->right;
        //Init left->right,namely divisor^2
        new->left->right=init_opperand('^');
            new->left->right->left=root->right;
            new->left->right->right=init_variant("2");
    //Initialize right side
    new->right=init_opperand('/');
        //Init right->left,namely divident*deri(divisor)
        new->right->left=init_opperand('*');
            new->right->left->left=root->left;
            new->right->left->right=derivative(root->right,target_variant);
        //Init right->right,same as left->left
        new->right->right=init_opperand('^');
            new->right->right->left=root->right;
            new->right->right->right=init_variant("2");
    return new;
}

tree* derivative_pow(tree* root,char *target_variant){
    //Apply power rule [f^g*(g'lnf + gf'/f)]
    tree* new=init_opperand('*');
    new->left=root;
    new->right=init_opperand('+');
        new->right->left=init_opperand('*');
        new->right->right=init_opperand('*');
            new->right->left->left=derivative(root->right,target_variant);
            new->right->left->right=init_opperand('l');//to "l"
            new->right->left->right->state=0;
                new->right->left->right->left=init_variant("e");
                new->right->left->right->right=root->left;
            new->right->right->left=root->right;
            new->right->right->right=init_opperand('/');
                new->right->right->right->left=derivative(root->left,target_variant);
                new->right->right->right->right=root->left;
    return new;
}
```

Chapter B Declaration

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