FDS 2025 春夏

project2

Autograd for Algebraic Expressions

徐天杰

2025-04-12

Contents

1 Introduction

In this project, we need to write a program to do autograd for the given infix expression.And this report is mainly about the result of my program's performance and what is need to be done to improve it.

(1). In Chapter 2, I will introduce the the main algorithm of process used in the program.

(2). In Chapter 3, I present my test data , including input ,output ,and some analysis for the result.

(3). In Chapter 4, I conduct an analysis of my program, including what function has been achieved and what is needed to be improved.

(4). In Chapter 5, I include the complete code in the appendix.

2 Algorithm Specification

The following are the pseudocodes and explanations for main algorithm of process used in the program.

2.1 Infix Expression → Postfix Expression

```
function infix_to_postfix(infix_expression):
create empty stack operator_stack
create empty list output

for each token in infix_expression:
    if token is an operand (number or variable):
        append token to output
    else if token is a left parenthesis '(':
        push token onto operator_stack
    else if token is a right parenthesis ')':
        while operator_stack is not empty and top is not '(':
            pop from operator_stack and append to output
        pop the '(' from operator_stack (discard)
    else if token is an operator:
        while operator_stack is not empty and
              precedence(top of operator_stack) >= precedence(token):
            pop from operator_stack and append to output
```

```
            push token onto operator_stack

    while operator_stack is not empty:
        pop from operator_stack and append to output

    return output (postfix expression)
```

To construct the infix expression as a tree ,we first need to transform it into a postfix expression .Here I deploy the traditional way of using a stack to save operators and do push & pop operations with the operators and brackets.

2.2 Postfix Expression → Abstract Syntax Tree

```
    div_con(arr, sum, left_p, right_p, count):
    res = 0 //initialize result

    //only begin search if the size is smaller than 20
    if (right_p - left_p) > 20:
        mid = (right_p + left_p) / 2 //calculate the mid num
        res_left = div_con(arr, sum, left_p, mid, count) //recursionly
            check the left part
        if res_left == 1: //if find in the left half
            res = 1
            return res //target found

        res_right = div_con(arr, sum, mid, right_p, count) //recursionly
            check the right part
        if res_right == 1: //if find in the left half
            res = 1
            return res //target found

    //if the size if smaller than 20 or the smaller branch didn't find
        the result
    for i from left_p to right_p - 1:
        for j from i + 1 to right_p - 1:
            if arr[i] + arr[j] == sum: //check the sum
                res = 1
                // printf("%d %d + %d = %d\n", count, arr[i], arr[j], sum)
```

```
            return res //target found

    return res //target not found
```

Then we need to transform the postfix expression into an ast .The basic algorithm is to use a stack to save the variable and number nodes .When it meet a operator ,it will pop the last two node of the stack as the left and right child of the operator node ,and push it into the stack.

2.3 Recursive Differentiation on AST

```
function derivative(node):
if node is a constant:
    return a node representing constant 0
if node is a variable:
    if node.name matches the differentiation variable:
        return a node representing constant 1
    else:
        return a node representing constant 0

if node is an operator:
    let u = node.left
    let v = node.right (may be NULL for unary operators)

    match node.operator:
        case '+':
            return derivative(u) + derivative(v)
        case '-':
            return derivative(u) - derivative(v)
        case '*':
            return u * derivative(v) + v * derivative(u)
        case '/':
            return (v * derivative(u) - u * derivative(v)) / (v^2)
        case '^':
            if v is a constant:
                return v * u^(v - 1) * derivative(u)
            else:
                return u^v * [derivative(v) * ln(u) + v * derivative(u)/
```

```
                    u]
        case 'ln':
            return derivative(u) / u

    return the newly constructed node (expression subtree)
```

We use a recursive function, typically named derivative(node), which takes the root of a subtree and returns a new subtree representing the derivative of that expression. The recursion works by decomposing a complex expression into smaller parts (its children), computing the derivative of each, and then combining the results according to differentiation rules.

(1). **Constant Rule**

$$\frac{d}{dx}(c) = 0$$

where c is a constant.

(2). **Variable Rule**

$$\frac{d}{dx}(x) = 1, \quad \frac{d}{dx}(y) = 0 \quad (\text{if } x \neq y)$$

(3). **Addition / Subtraction Rule**

$$\frac{d}{dx}(u \pm v) = \frac{du}{dx} \pm \frac{dv}{dx}$$

(4). **Multiplication Rule (Product Rule)**

$$\frac{d}{dx}(uv) = u \cdot \frac{dv}{dx} + v \cdot \frac{du}{dx}$$

(5). **Division Rule (Quotient Rule)**

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v \cdot \frac{du}{dx} - u \cdot \frac{dv}{dx}}{v^2}$$

(6). **Power Rule**

- If exponent is constant n:

$$\frac{d}{dx}(u^n) = n \cdot u^{n-1} \cdot \frac{du}{dx}$$

- General case:

$$\frac{d}{dx}(u^v) = u^v \cdot \left(\frac{dv}{dx} \cdot \ln u + v \cdot \frac{1}{u} \cdot \frac{du}{dx}\right)$$

3 Testing Data

To analyze the performance of the autograd program , we try to input different infix expression and compare the output answer with expected answer.

3.1 Simply add and minus

Input:

```
a+b-c
```

Output:

```
a: 1
b: 1
c: -1
```

Expected output:

```
a: 1
b: 1
c: -1
```

3.2 Multiply and division

Input:

```
x*a-b/x
```

Output:

```
x: a-(-b)/(x*x)
a: x
b: -(x)/(x*x)
```

Expected output:

```
x:a+1/(x*x)
a:x
b:-1/x
```

Here we find that the true output is not completely the same as expected output. But if we do some simple calculation ,we can find that after some simplification the output is exactly the same as the expected output and we will analyze this in the next chapter.

3.3 Bracket and power

Input:

```
(x-1)^3+a^c
```

Output:

```
x: 3*(x-1)^2
a: a^c*(c*1/a)
c: a^c*(ln(a))
```

Expected output:

```
x: 3*(x-1)^2
a: a^(c-1)*c
c: a^c*ln(a)
```

3.4 Long variable

Input:

```
aa*bb-ab/aa
```

Output:

```
aa: bb-(-ab)/(aa*aa)
bb: aa
ab: -(aa)/(aa*aa)
```

Expected output:

```
aa: bb+ab/(aa*aa)
bb: aa
ab: -1/aa
```

3.5 Combination of situation

Input:

```
aa^(aa-b)-x/(d+b)*5
```

Output:

```
      aa: aa^(aa-b)*(ln(aa)+(aa-b)*1/aa)
      b: aa^(aa-b)*((-1)*ln(aa))-(-x)/((d+b)*(d+b))*5
      x: -(d+b)/((d+b)*(d+b))*5
      d: -(-x)/((d+b)*(d+b))*5
```

Expected output:

```
      aa: aa^(aa-b)*(ln(aa)+(aa-b)/aa)
      b: aa^(aa-b)*(-ln(aa))+x/((d+b)*(d+b))*5
      x: -1/(d+b)*5
      d: +x/((d+b)*(d+b))*5
```

4 Analysis and Comments

Firstly ,I'll briefly analyze the time and space complexity of the main algorithm used in the program. Then I will comment on the performance of the program.

4.1 Infix expression -> tree

(1). **Time complexity:**

- During this process, we have two step :1.infix->postfix,2.construct the tree.And for both the process ,we need to traverse all the characters in the expression. Therefore ,the time complexity of Infix expression -> tree process is $O(n)$.

(2). **Space complexity:**

- The main space cost is the stacks we use in the process and the node space for the tree.We only need $O(n)$ for the stack and tree ,therefore the total space complexity is $O(n)$.

4.2 Differentiation

(1). **Time complexity:**

- The differentiation process use the recursion to process the tree.When the worst case happens ,we need to process multiply and divide process ,which will cost 2-3 times of node .Therefore ,theoretically the worst case of time complexity is $O(2^n)$.

(2). **Space complexity:**

- The same as time complexity ,in worst case the space complexity will be $O(2^n)$.

4.3 Performance analysis

- The program can perform the operation of auto-differentiation for given infix expression.It can deal with add,minus,multiply,divide,bracket and power quite well.Even in some cases we find the output is not exactly the same as the expected output ,but if we artificially simplify the output expression ,we can find the output answer is right.

- The program also involves some simple simplification ,which can deal with operations that involves 0 and 1 ,which helps to simplify most cases like a+0,a*0,a0.etc

4.4 Things need to be improved

- Just as mentioned before ,the program can't deal with simplifying operations with same variable and combining same sub-expression.This will make some simple answer printed in a more complicated way.But yet I don't find a proper way to deal with it.

```
            xy: (-xx^2)/(xy*xy)*xy+xx^2/xy = xy: 0
```

- This program also can't deal with any complicated math functions like ln(x),sin(x),pow(x,y) .etc This will involve a more complicated constructing rule for the tree.

5 Appendix

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

typedef struct node{
    char* variable;                         //variable
    char operator;                          //operator
    int num;                                //num
    int type;                               //1: variable 2:operator 3:
        num
```

```c
    struct node* right;                     //right child
    struct node* left;                      //left child
}node;

int precedence(char op);                    //return the priority of operation
int is_right_associative(char op);          //return if the operator is ^
int is_operator(char c);                    //return if the char is an operator
node* create_node(char* token);             //create the node for each item
int get_root(char* exp ,char** var ,node** root); //construct the tree of the expression
int get_token(char* exp ,char** var);       //divide the expression into operators and operands
node* autograd_for_var(char* var ,node* root); //calculate the grad of the tree
char* join(node* root ,char* expression);   //simplify the tree for 0 and 1 and()

node* new_operator_node(char op ,node* left ,node* right){ //create node of operator in grad
    node* n = (node*)malloc(sizeof(node));          //prepare the space for new node
    n->type = 2;                                    //mark type as 2
    n->operator = op;
    n->left = left;
    n->right = right;
    return n;
}

node* new_variable_node(char* var){                 //create node of variable in grad
    node* n = (node*)malloc(sizeof(node));          //prepare the space for new node
    n->type = 1;                                    //mark type as 1
    n->variable = var;
```

```
    n->left = n->right = NULL;                          //leaf node
    return n;
}

node* new_number_node(int val){                         //create node of number in grad
    node* n = (node*)malloc(sizeof(node));              //prepare the space for new node
    n->type = 3;                                        //mark type as 3
    n->num = val;
    n->left = n->right = NULL;                          //leaf node
    return n;
}

int main(void){
    char expression[100];                               //expression is the infix input
    char** variables = (char**)malloc(sizeof(char*)*100); //to save the division first ,then save the variable list
    node* ast_root = NULL;                              //the root for the tree
    scanf("%[^\n]" ,expression);
    // printf("%s",expression);
    // printf("\n");
    //build the tree of the expression and find the variables
    int len_of_var = get_root(expression ,variables ,&ast_root); //build the tree of expression and get the number of variable
    printf("There's %d variables in the expression.\n",len_of_var); // print out the lenth of variable
    for(int i = 0;i < len_of_var;i++){                          //do grad for each variable
        printf("%s",variables[i]);
        printf(": ");
        node* a_of_var;                                         //to save the tree after grad
        a_of_var = autograd_for_var(variables[i] ,ast_root);    // calculate the grad
        char* join_exp = (char*)calloc(1000 ,sizeof(char));     //simplify
```

```c
            and turn it into a string
        join_exp = join(a_of_var ,join_exp);                    //print out the result
        printf("%s" ,join_exp);

        printf("\n");
    }
    return 0;
}

int precedence(char op){
    switch(op){
        case 'l':
        case '^': return 3;             //^ is the highest
        case '*':
        case '/': return 2;             //* / is the 2nd
        case '+':
        case '-': return 1;             //+ - is the lowest
        default: return 0;
    }
}

int is_right_associative(char op){     //return if the operator is ^
    return op == '^';
}

int is_operator(char c){               //return if the char is an operator
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
}

node* create_node(char* token){
    node* n = (node*)malloc(sizeof(node)); //create the new node
    n->left = NULL;
    n->right = NULL;
    if(is_operator(token[0])){          //if the token is an operator
        n->type = 2;                    //mark type as 2
        n->operator = token[0];
```

```c
    }else if((token[0] >= 'a' && token[0] <= 'z') || (token[0] >= 'A' && token[0] <= 'Z')){ //if the token is variable
        n->type = 1;        //mark type as 1
        n->variable = token;
    }else{      //if the token is number
        n->type = 3;        //mark type as 3
        int i = 0;
        int result = 0;         //turn the string number into a int type
        while (token[i] != '\0'){
            if(token[i] >= '0' && token[i] <= '9'){
                result = result * 10 + (token[i] - '0');          //calculate the number
            }else{
                printf("Invalid number: %s\n" ,token);          //exit if it is not a number
                exit(1);
            }
            i++;
        }
        n->num = result;
    }

    return n;
}

int get_root(char* exp ,char**var ,node** root){
    int var_num = 0;        //count the number of variables
```

```
    //transition of infix into postfix
    char** stack = (char**)malloc(sizeof(char*) * 20);                    //operator stack for in-post transition
    char** post = (char**)malloc(sizeof(char*) * 20);                     //save postfix expression
    int len = get_token(exp ,var);                                        //divide into tokens
    int post_p = 0;
    int stack_p = 0;
    int flag = 0;                                                         //count backects
    for(int i = 0;i < len;i++){
        if(is_operator(var[i][0])){                                       //if operator push into stack or pop operator out
            while (stack_p > 0 && stack[stack_p - 1][0] != '(' &&
                (precedence(stack[stack_p - 1][0]) > precedence(var[i][0]) ||
                (precedence(stack[stack_p - 1][0]) == precedence(var[i][0]) && !is_right_associative(var[i][0])))) {
                post[post_p++] = stack[--stack_p];                        //pop operator with higher priority
            }
            stack[stack_p++] = var[i];
        }else if(var[i][0] == '('){                                       //if ( push into stack
            stack[stack_p] = var[i];
            stack_p++;
            flag++;
        }else if(var[i][0] == ')'){                                       //if ) pop operator out until (
            while(stack_p > 0 && stack[stack_p - 1][0] != '('){
```

```
                post[post_p++] = stack[--stack_p];                  //pop operator out
            }
            if(stack_p > 0 && stack[stack_p - 1][0] == '('){          //meet ( and throw it
                --stack_p;
            }else{
                printf("Mismatched parentheses!\n");                //if () didn't match
                exit(1);
            }
        }else if((var[i][0] >= 'A' && var[i][0] <= 'Z') || (var[i][0] >= 'a' && var[i][0] <= 'z')|| (var[i][0] >= '0' && var[i][0] <= '9')){
            post[post_p] = var[i];                                  //if variable or num push into post
            post_p++;
        }else{
            printf("Wrong input!");                                 //if not allowed input
            exit(1);
        }
    }
    while(stack_p > 0){
        // printf("%s",stack[stack_p]);
        // printf("\n");                                            // pop out the remaining operators in the stack
        if(stack[stack_p - 1][0] == '('){
            printf("Mismatched parentheses!\n");                    //if () didn't match
            exit(1);
        }
        post[post_p++] = stack[--stack_p];                          //push into post
    }
```

```
    len = len - flag * 2;
    printf("Postfix expression:");
    for(int i = 0;i < post_p;i++)printf("%s",post[i]);
    printf("\n");
    //now we start construct the ast tree
    node** node_stack = (node**)malloc(sizeof(node*) * len);
                                //stack for elements
    int n_stack_p = 0;
    for(int i = 0;i < len;i++){
        node* temp;

            //create new tree node
        temp = create_node(post[i]);
        if(temp->type == 2){
                                                        //use
            operator as root
            temp->right = node_stack[--n_stack_p];
                                        //pop the operand of the
                operator from the stack
            temp->left = node_stack[--n_stack_p];
            node_stack[n_stack_p++] = temp;
                                            //push the current
                 root into the stack
        }else if(temp->type == 1){
            node_stack[n_stack_p++] = temp;
            int flag = 1;

                //check if the variable has appeared
            for(int i = 0;i < var_num;i++){
                if(strcmp(temp->variable ,var[i]) == 0){
                    flag = 0;
                }
            }
            if(flag){

                //if the first time count and push into varlist
                var[var_num++] = temp->variable;
            }
```

```
        }else{
            node_stack[n_stack_p++] = temp;                                    //if number ,push into stack
        }
    }
    *root = node_stack[0];                                                     //save the root for the tree
    return var_num;                                                            //return the number of variables
}

int get_token(char* exp ,char** var){
    int count = 0;                                                             //count the elements
    int i = 0;
    var[count] = (char*)malloc(sizeof(char) * 20);                             //use var to save the divided element for the tree
    int temp_p = 0;
    while(exp[i] != '\0'){                                                     //go through every char
        if((exp[i] >= 'A' && exp[i] <= 'Z') || (exp[i] >= 'a' && exp[i] <=
            'z') || (exp[i] >= '0' && exp[i] <= '9')){
            var[count][temp_p] = exp[i];                                       //if number or letter
            temp_p++;
        }else if(is_operator(exp[i]) || exp[i] == '(' || exp[i] == ')'){
            if(temp_p != 0){                                                   //save number or variable until meet an operator
                var[count][temp_p] = '\0';                                     //end it as a
```

```
                    string
                count++;
                var[count] = (char*)malloc(sizeof(char) * 20);
                temp_p = 0;
            }
            var[count][temp_p++] = exp[i];                                    //add the
                operator
            var[count][temp_p++] = '\0';
            count++;
            var[count] = (char*)malloc(sizeof(char) * 20);
                                                    //start next element
            temp_p = 0;
        }
        i++;
    }
    if(temp_p != 0){                                                                   //
        save tje last element
        var[count][temp_p] = '\0';
        count++;
    }
    return count;
}

node* autograd_for_var(char* var ,node* root){
    if(!root){
        return NULL;                                                               //meet
            the leaf node
    }
    if(root->type == 3){
        return new_number_node(0);                                                 //for
            const ,grad is 0
    }
    if(root->type == 1){                                                           //for
        var ,check if it is the target var for grad
        if(strcmp(root->variable ,var) == 0){
            return new_number_node(1);                                             //is
```

```
                return 1
        }else{
            return new_number_node(0);                                  //or
                return 0
        }
    }
    if(root->type == 2){                                                //for
        operator ,recursionly process the child
        node *u = root->left ,*v = root->right;                         //u,v
            as the operand of the operator
        node *du = autograd_for_var(var ,u) ,*dv = autograd_for_var(var ,v
            );//du,dv as the grad of u,v
        switch (root->operator){
            case '+':                                                   //d(u+v
                )=du+dv
                return new_operator_node('+' ,du ,dv);
            case '-':                                                   //d(u-v
                )=du-dv
                return new_operator_node('-' ,du ,dv);
            case '*':                                                   //d(u*v
                )=du*v+u*dv
                return new_operator_node('+',new_operator_node('*' ,du ,v)
                    ,new_operator_node('*' ,u ,dv));
            case '/':                                                   //d(u/v
                )=(du*v-u*dv)/(v*v)
                return new_operator_node('/',new_operator_node('-',
                    new_operator_node('*' ,du ,v),new_operator_node('*' ,u ,
                    dv)) ,new_operator_node('*' ,v ,v));

            case '^':{
                int is_u_const = (u->type == 3);
                int is_v_const = (v->type == 3);
                if(is_v_const && !is_u_const){                          //d(u^c
                    )=c*u^(c-1)*du
                    int c = v->num;
                    node* new_exp = new_number_node(c - 1);
                    node* u_pow_c_minus_1 = new_operator_node('^' ,u ,
                        new_exp);
```

```
                    return new_operator_node('*',new_operator_node('*',new_number_node(c),u_pow_c_minus_1),du);
                }else if(is_u_const && !is_v_const) {              //d(c^v)=c^v*ln(c)*dv
                    int c = u->num;
                    node* ln_c = new_operator_node('l' ,u ,NULL);    //ln(u)
                    node* c_pow_v = new_operator_node('^' ,u ,v);
                    return new_operator_node('*',new_operator_node('*',c_pow_v,ln_c),dv);
                }else{                                             //d(u^v)=u^v*(dv*ln(u)+v*du/u)
                    node* ln_u = new_operator_node('l' ,u ,NULL);    //ln(u)
                    node* term1 = new_operator_node('*' ,dv ,ln_u);
                    node* u_div = new_operator_node('/' ,du ,u);     //du/u
                    node* term2 = new_operator_node('*' ,v ,u_div);
                    node* sum = new_operator_node('+' ,term1 ,term2); //dv*ln(u)+v*du/u
                    node* u_pow_v = new_operator_node('^' ,u ,v);
                    return new_operator_node('*' ,u_pow_v ,sum);
                }
            }
            default:
                printf("Unsupported operator: %c\n" ,root->operator); //operator unsupported
                exit(1);
        }
    }
    return NULL;
}

char* join(node* root ,char* expression){
    if(!root){                                   //meet the leaf node
        return NULL;
    }
    if(root->type == 1){                         //variable cat into expression
        strcat(expression ,root->variable);
        return expression;
```

```
    }
    else if(root->type == 3){                         //number cat into expression
        char num_str[20];
        sprintf(num_str ,"%d" ,root->num);            //change int into str
        strcat(expression ,num_str);
        return expression;
    }
    else if(root->type == 2){
        if(root->operator == 'l'){                    //opertor 'l' as ln
            strcat(expression ,"ln(");
            join(root->left ,expression);             //put ln(x) into expression
            strcat(expression ,")");
            return expression;
        }

        char* left_str = (char*)calloc(100 ,sizeof(char)); //save the middle result for the two child
        char* right_str = (char*)calloc(100 ,sizeof(char));
        left_str = join(root->left ,left_str);        //recursively calculate the result
        right_str = join(root->right ,right_str);

        if(root->operator == '+'){                    //simplify when 0+a, a+0,0+0
            if(strcmp(left_str ,"0") == 0){           //0+a->a
                strcat(expression ,right_str);
                goto done;
            }
            if(strcmp(right_str ,"0") == 0){          //a+0->a
                strcat(expression ,left_str);
                goto done;
            }
        }
        else if(root->operator == '-'){               //simplify when 0-a, a-0
```

```
            if(strcmp(left_str ,"0") == 0 && strcmp(right_str ,"0") == 0){
                strcat(expression ,"0");                //0-0->0
                goto done;
            }
            if(strcmp(left_str ,"0") == 0){
                strcat(expression ,"-");                //0-a->-a
                strcat(expression ,right_str);
                goto done;
            }
            if(strcmp(right_str ,"0") == 0){
                strcat(expression ,left_str);           //a-0->a
                goto done;
            }
        }
        else if(root->operator == '*'){                 //simplify when 0*a, a*0,1*a,a*1
            if(strcmp(left_str ,"0") == 0 || strcmp(right_str ,"0") == 0){
                strcat(expression ,"0");                //0*0->0
                goto done;
            }
            if(strcmp(left_str ,"1") == 0){
                strcat(expression ,right_str);          //1*a->a
                goto done;
            }
            if(strcmp(right_str ,"1") == 0){
                strcat(expression ,left_str);           //a*1->a
                goto done;
            }

        }
        else if(root->operator == '/'){                 //simplify when 0/a, a/1
            if(strcmp(left_str ,"0") == 0){
                strcat(expression ,"0");                //0/a->0
                goto done;
            }
            if(strcmp(right_str ,"1") == 0){
                strcat(expression ,left_str);           //a/1->a
```

```
                goto done;
            }
        }else if(root->operator == '^'){             //simplify when a^1, a^0
            if(strcmp(right_str ,"1") == 0){
                strcat(expression ,left_str);        //a^1->a
                goto done;
            }
            if(strcmp(right_str ,"0") == 0){
                strcat(expression ,"1");             //a^0->1
                goto done;
            }
        }
        if(root->left->type == 2 && precedence(root->operator) > precedence(root->left->operator)){
            strcat(expression ,"(");
            strcat(expression ,left_str);            //add () if the left child operator is smaller than root
            strcat(expression ,")");
        }else{
            strcat(expression ,left_str);            //else no ()
        }
        int len = strlen(expression);
        expression[len] = root->operator;            //push the operator
        expression[len + 1] = '\0';

        if((root->right->type == 2 && precedence(root->operator) > precedence(root->right->operator))
        || (root->right->type == 2 && root->operator == '/')){
            strcat(expression ,"(");
            strcat(expression ,right_str);           //add () if the right child operator is smaller than root
            strcat(expression ,")");
        }else{
            strcat(expression ,right_str);           //else no ()
        }

    done:
```

```
        free(left_str);                            //free the temporal
            string
        free(right_str);
        return expression;                         //return expression
    }

    return NULL;                                   //return NULL when
        something wrong
}
```

6 Declaration

I hereby declare that all the work done in this project titled "Project 2 : Autograd for Algebraic Expressions " is of my independent effort.