浙江大学

ZHEJIANG UNIVERSITY

FDS 2025 春夏

project2

Autograd for Algebraic Expressions

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

```
1    function infix_to_postfix(infix_expression):
2    create empty stack operator_stack
3    create empty list output
4
5    for each token in infix_expression:
6        if token is an operand (number or variable):
7            append token to output
8        else if token is a left parenthesis '(':
9            push token onto operator_stack
10       else if token is a right parenthesis ')':
11           while operator_stack is not empty and top is not '(':
12               pop from operator_stack and append to output
13           pop the '(' from operator_stack (discard)
14       else if token is an operator:
15           while operator_stack is not empty and
16                 precedence(top of operator_stack) >= precedence(token):
17               pop from operator_stack and append to output
```

```
18              push token onto operator_stack
19
20      while operator_stack is not empty:
21          pop from operator_stack and append to output
22
23      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

```
1       div_con(arr, sum, left_p, right_p, count):
2       res = 0 //initialize result
3
4       //only begin search if the size is smaller than 20
5       if (right_p - left_p) > 20:
6           mid = (right_p + left_p) / 2 //calculate the mid num
7           res_left = div_con(arr, sum, left_p, mid, count) //recursionly
                check the left part
8           if res_left == 1: //if find in the left half
9               res = 1
10              return res //target found
11
12          res_right = div_con(arr, sum, mid, right_p, count) //recursionly
                check the right part
13          if res_right == 1: //if find in the left half
14              res = 1
15              return res //target found
16
17      //if the size if smaller than 20 or the smaller branch didn't find
            the result
18      for i from left_p to right_p - 1:
19          for j from i + 1 to right_p - 1:
20              if arr[i] + arr[j] == sum: //check the sum
21                  res = 1
22                  // printf("%d %d + %d = %d\n", count, arr[i], arr[j], sum)
```

```
23              return res //target found
24
25      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

```
1      function derivative(node):
2      if node is a constant:
3          return a node representing constant 0
4      if node is a variable:
5          if node.name matches the differentiation variable:
6              return a node representing constant 1
7          else:
8              return a node representing constant 0
9
10     if node is an operator:
11         let u = node.left
12         let v = node.right (may be NULL for unary operators)
13
14         match node.operator:
15             case '+':
16                 return derivative(u) + derivative(v)
17             case '-':
18                 return derivative(u) - derivative(v)
19             case '*':
20                 return u * derivative(v) + v * derivative(u)
21             case '/':
22                 return (v * derivative(u) - u * derivative(v)) / (v^2)
23             case '^':
24                 if v is a constant:
25                     return v * u^(v - 1) * derivative(u)
26                 else:
27                     return u^v * [derivative(v) * ln(u) + v * derivative(u)/
```

```
                    u]
28              case 'ln':
29                  return derivative(u) / u
30
31      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:

```
1        a+b-c
```

Output:

```
1        a: 1
2        b: 1
3        c: -1
```

Expected output:

```
1        a: 1
2        b: 1
3        c: -1
```

3.2 Multiply and division

Input:

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

Output:

```
1        x: a-(-b)/(x*x)
2        a: x
3        b: -(x)/(x*x)
```

Expected output:

```
1        x:a+1/(x*x)
2        a:x
3        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:

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

Output:

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

Expected output:

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

3.4 Long variable

Input:

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

Output:

```
1        aa: bb-(-ab)/(aa*aa)
2        bb: aa
3        ab: -(aa)/(aa*aa)
```

Expected output:

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

3.5 Combination of situation

Input:

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

Output:

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

Expected output:

```
1          aa: aa^(aa-b)*(ln(aa)+(aa-b)/aa)
2          b: aa^(aa-b)*(-ln(aa))+x/((d+b)*(d+b))*5
3          x: -1/(d+b)*5
4          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 $O(2^n)$.

April 6 2025

(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.

```
1               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
1  # include<stdio.h>
2  # include<stdlib.h>
3  # include<string.h>
4
5  typedef struct node{
6      char* variable;                      //variable
7      char operator;                       //operator
8      int num;                             //num
9      int type;                            //1: variable 2:operator 3:
           num
```

```
10      struct node* right;                    //right child
11      struct node* left;                     //left child
12   }node;
13
14   int precedence(char op);                   //return the priority of
        operation
15   int is_right_associative(char op);         //return if the operator is
        ^
16   int is_operator(char c);                   //return if the char is an
        operator
17   node* create_node(char* token);            //create the node for each
        item
18   int get_root(char* exp ,char** var ,node** root); //construct the tree of
         the expression
19   int get_token(char* exp ,char** var);      //divide the expression
        into operators and operands
20   node* autograd_for_var(char* var ,node* root); //calculate the grad of
        the tree
21   char* join(node* root ,char* expression);  //simplify the tree for 0
        and 1 and()
22
23   node* new_operator_node(char op ,node* left ,node* right){ //create node
        of operator in grad
24      node* n = (node*)malloc(sizeof(node));       //prepare the space
           for new node
25      n->type = 2;                                 //mark type as 2
26      n->operator = op;
27      n->left = left;
28      n->right = right;
29      return n;
30   }
31
32   node* new_variable_node(char* var){          //create node of
        variable in grad
33      node* n = (node*)malloc(sizeof(node));       //prepare the space
           for new node
34      n->type = 1;                                 //mark type as 1
35      n->variable = var;
```

```
36        n->left = n->right = NULL;                      //leaf node
37        return n;
38    }
39
40    node* new_number_node(int val){                     //create node of
          number in grad
41        node* n = (node*)malloc(sizeof(node));          //prepare the space
              for new node
42        n->type = 3;                                    //mark type as 3
43        n->num = val;
44        n->left = n->right = NULL;                      //leaf node
45        return n;
46    }
47
48    int main(void){
49        char expression[100];                           //expression is the
              infix input
50        char** variables = (char**)malloc(sizeof(char*)*100); //to save the
              division first ,then save the variable list
51        node* ast_root = NULL;                          //the root for the
              tree
52        scanf("%[^\n]" ,expression);
53        // printf("%s",expression);
54        // printf("\n");
55        //build the tree of the expression and find the variables
56        int len_of_var = get_root(expression ,variables ,&ast_root); //build
              the tree of expression and get the number of variable
57        printf("There's %d variables in the expression.\n",len_of_var); //
              print out the lenth of variable
58        for(int i = 0;i < len_of_var;i++){              //do grad
              for each variable
59            printf("%s",variables[i]);
60            printf(": ");
61            node* a_of_var;                             //to save
                  the tree after grad
62            a_of_var = autograd_for_var(variables[i] ,ast_root);    //
                  calculate the grad
63            char* join_exp = (char*)calloc(1000 ,sizeof(char));    //simplify
```

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

66
67          printf("\n");
68      }
69      return 0;
70  }
71
72  int precedence(char op){
73      switch(op){
74          case 'l':
75          case '^': return 3;          //^ is the highest
76          case '*':
77          case '/': return 2;          //* / is the 2nd
78          case '+':
79          case '-': return 1;          //+ - is the lowest
80          default: return 0;
81      }
82  }
83
84  int is_right_associative(char op){   //return if the operator is ^
85      return op == '^';
86  }
87
88  int is_operator(char c){             //return if the char is an
        operator
89      return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
90  }
91
92  node* create_node(char* token){
93      node* n = (node*)malloc(sizeof(node)); //create the new node
94      n->left = NULL;
95      n->right = NULL;
96      if(is_operator(token[0])){       //if the token is an operator
97          n->type = 2;                 //mark type as 2
98          n->operator = token[0];
```

```
 99        }else if((token[0] >= 'a' && token[0] <= 'z') || (token[0] >= 'A' &&
              token[0] <= 'Z')){ //if the token is variable
100            n->type = 1;

                   //mark type as 1
101            n->variable = token;
102        }else{

               //if the token is number
103            n->type = 3;

                   //mark type as 3
104            int i = 0;
105            int result = 0;
                                                                           //
               turn the string number into a int type
106            while (token[i] != '\0'){
107                if(token[i] >= '0' && token[i] <= '9'){
108                    result = result * 10 + (token[i] - '0');
                                             //calculate the number
109                }else{
110                    printf("Invalid number: %s\n" ,token);
                                                 //exit if it is not a
                       number
111                    exit(1);
112                }
113                i++;
114            }
115            n->num = result;
116        }

117
118        return n;
119    }

120
121    int get_root(char* exp ,char**var ,node** root){
122        int var_num = 0;
                                                                           //
               count the number of variables
```

```
123        //transition of infix into postfix
124        char** stack = (char**)malloc(sizeof(char*) * 20);
                                    //operator stack for in-post
               transition
125        char** post = (char**)malloc(sizeof(char*) * 20);
                                    //save postfix expression
126        int len = get_token(exp ,var);
                                                //divide into
               tokens
127        int post_p = 0;
128        int stack_p = 0;
129        int flag = 0;

           //count backects
130        for(int i = 0;i < len;i++){
131            if(is_operator(var[i][0])){
                                                //if operator
                 push into stack or pop operator out
132                while (stack_p > 0 && stack[stack_p - 1][0] != '(' &&
133                    (precedence(stack[stack_p - 1][0]) > precedence(var[i][0])
                          ||
134                    (precedence(stack[stack_p - 1][0]) == precedence(var[i][0])
                          && !is_right_associative(var[i][0])))) {
135                    post[post_p++] = stack[--stack_p];
                                                //pop operator with
                       higher priority
136                }
137                stack[stack_p++] = var[i];
138            }else if(var[i][0] == '('){
                                                //if ( push
                 into stack
139                stack[stack_p] = var[i];
140                stack_p++;
141                flag++;
142            }else if(var[i][0] == ')'){
                                                //if ) pop
                 operator out until (
143                while(stack_p > 0 && stack[stack_p - 1][0] != '('){
```

```
144                 post[post_p++] = stack[--stack_p];
                                                    //pop operator out
145             }
146             if(stack_p > 0 && stack[stack_p - 1][0] == '('){
                                        //meet ( and throw it
147                 --stack_p;
148             }else{
149                 printf("Mismatched parentheses!\n");
                                                    //if () didn't match
150                 exit(1);
151             }
152         }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'
            )){
153             post[post_p] = var[i];
                                                                //if
                variable or num push into post
154             post_p++;
155         }else{
156             printf("Wrong input!");
                                                                //if not
                allowed input
157             exit(1);
158         }
159     }
160     while(stack_p > 0){
161         // printf("%s",stack[stack_p]);
162         // printf("\n");
                                                                    //
            pop out the remaining operators in the stack
163         if(stack[stack_p - 1][0] == '('){
164             printf("Mismatched parentheses!\n");
                                                    //if () didn't match
165             exit(1);
166         }
167         post[post_p++] = stack[--stack_p];
                                                    //push into post
168     }
```

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

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

                   //check if the variable has appeared
186                for(int i = 0;i < var_num;i++){
187                    if(strcmp(temp->variable ,var[i]) == 0){
188                        flag = 0;
189                    }
190                }
191                if(flag){

                       //if the first time count and push into varlist
192                    var[var_num++] = temp->variable;
193                }
```

```
194          }else{
195              node_stack[n_stack_p++] = temp;
                                                        //if number ,push
                    into stack
196          }
197      }
198      *root = node_stack[0];
                                                        //save
             the root for the tree
199      return var_num;
                                                        //
             return the number of variables
200  }
201
202  int get_token(char* exp ,char** var){
203      int count = 0;

             //count the elements
204      int i = 0;
205      var[count] = (char*)malloc(sizeof(char) * 20);
                                                 //use var to save the divided
             element for the tree
206      int temp_p = 0;
207      while(exp[i] != '\0'){

                                                        //go
             through every char
208          if((exp[i] >= 'A' && exp[i] <= 'Z') || (exp[i] >= 'a' && exp[i] <=
                 'z') || (exp[i] >= '0' && exp[i] <= '9')){
209              var[count][temp_p] = exp[i];
                                                        //if number or
                    letter
210              temp_p++;
211          }else if(is_operator(exp[i]) || exp[i] == '(' || exp[i] == ')'){
212              if(temp_p != 0){
                                                        //
                    save number or variable until meet an operator
213                  var[count][temp_p] = '\0';
                                                        //end it as a
```

```
                            string
214             count++;
215             var[count] = (char*)malloc(sizeof(char) * 20);
216             temp_p = 0;
217         }
218         var[count][temp_p++] = exp[i];
                                                            //add the
            operator
219         var[count][temp_p++] = '\0';
220         count++;
221         var[count] = (char*)malloc(sizeof(char) * 20);
                                        //start next element
222         temp_p = 0;
223       }
224       i++;
225     }
226     if(temp_p != 0){
                                                                    //
        save tje last element
227         var[count][temp_p] = '\0';
228         count++;
229     }
230     return count;
231 }
232
233 node* autograd_for_var(char* var ,node* root){
234     if(!root){
235         return NULL;                                        //meet
            the leaf node
236     }
237     if(root->type == 3){
238         return new_number_node(0);                          //for
            const ,grad is 0
239     }
240     if(root->type == 1){                                    //for
        var ,check if it is the target var for grad
241         if(strcmp(root->variable ,var) == 0){
242             return new_number_node(1);                      //is
```

```
                        return 1
243        }else{
244            return new_number_node(0);                        //or
                        return 0
245        }
246    }
247    if(root->type == 2){                                        //for
           operator ,recursionly process the child
248        node *u = root->left ,*v = root->right;                //u,v
               as the operand of the operator
249        node *du = autograd_for_var(var ,u) ,*dv = autograd_for_var(var ,v
           );//du,dv as the grad of u,v
250        switch (root->operator){
251            case '+':                                        //d(u+v
               )=du+dv
252                return new_operator_node('+' ,du ,dv);
253            case '-':                                        //d(u-v
               )=du-dv
254                return new_operator_node('-' ,du ,dv);
255            case '*':                                        //d(u*v
               )=du*v+u*dv
256                return new_operator_node('+',new_operator_node('*' ,du ,v)
                   ,new_operator_node('*' ,u ,dv));
257            case '/':                                        //d(u/v
               )=(du*v-u*dv)/(v*v)
258                return new_operator_node('/',new_operator_node('-',
                   new_operator_node('*' ,du ,v),new_operator_node('*' ,u ,
                   dv)) ,new_operator_node('*' ,v ,v));
259
260            case '^':{
261                int is_u_const = (u->type == 3);
262                int is_v_const = (v->type == 3);
263                if(is_v_const && !is_u_const){                //d(u^c
                   )=c*u^(c-1)*du
264                    int c = v->num;
265                    node* new_exp = new_number_node(c - 1);
266                    node* u_pow_c_minus_1 = new_operator_node('^' ,u ,
                       new_exp);
```

```
267                        return new_operator_node('*',new_operator_node('*',
                                new_number_node(c),u_pow_c_minus_1),du);
268                    }else if(is_u_const && !is_v_const) {            //d(c^v
                            )=c^v*ln(c)*dv
269                        int c = u->num;
270                        node* ln_c = new_operator_node('l' ,u ,NULL);    //ln(u)
271                        node* c_pow_v = new_operator_node('^' ,u ,v);
272                        return new_operator_node('*',new_operator_node('*',
                                c_pow_v,ln_c),dv);
273                    }else{                                          //d(u^v
                            )=u^v*(dv*ln(u)+v*du/u)
274                        node* ln_u = new_operator_node('l' ,u ,NULL);    //ln(u)
275                        node* term1 = new_operator_node('*' ,dv ,ln_u);
276                        node* u_div = new_operator_node('/' ,du ,u);    //du/u
277                        node* term2 = new_operator_node('*' ,v ,u_div);
278                        node* sum = new_operator_node('+' ,term1 ,term2); //dv*
                            ln(u)+v*du/u
279                        node* u_pow_v = new_operator_node('^' ,u ,v);
280                        return new_operator_node('*' ,u_pow_v ,sum);
281                    }
282                }
283            default:
284                printf("Unsupported operator: %c\n" ,root->operator); //
                    operator unsupported
285                exit(1);
286        }
287    }
288    return NULL;
289 }
290
291 char* join(node* root ,char* expression){
292    if(!root){                                      //meet the leaf node
293        return NULL;
294    }
295    if(root->type == 1){                             //variable cat into
            expression
296        strcat(expression ,root->variable);
297        return expression;
```

```
298          }
299          else if(root->type == 3){                      //number cat into
                                                              expression
300              char num_str[20];
301              sprintf(num_str ,"%d" ,root->num);          //change int into
                                                                  str
302              strcat(expression ,num_str);
303              return expression;
304          }
305          else if(root->type == 2){
306              if(root->operator == 'l'){                  //opertor 'l' as ln
307                  strcat(expression ,"ln(");
308                  join(root->left ,expression);           //put ln(x) into
                                                                  expression
309                  strcat(expression ,")");
310                  return expression;
311              }
312
313              char* left_str = (char*)calloc(100 ,sizeof(char)); //save the
                      middle result for the two child
314              char* right_str = (char*)calloc(100 ,sizeof(char));
315              left_str = join(root->left ,left_str);      //recursively
                      calculate the result
316              right_str = join(root->right ,right_str);
317
318              if(root->operator == '+'){                  //simplify when 0+a,
                      a+0,0+0
319                  if(strcmp(left_str ,"0") == 0){         //0+a->a
320                      strcat(expression ,right_str);
321                      goto done;
322                  }
323                  if(strcmp(right_str ,"0") == 0){        //a+0->a
324                      strcat(expression ,left_str);
325                      goto done;
326                  }
327              }
328              else if(root->operator == '-'){             //simplify when 0-a,
                      a-0
```

```
329            if(strcmp(left_str ,"0") == 0 && strcmp(right_str ,"0") == 0){
330                strcat(expression ,"0");              //0-0->0
331                goto done;
332            }
333            if(strcmp(left_str ,"0") == 0){
334                strcat(expression ,"-");              //0-a->-a
335                strcat(expression ,right_str);
336                goto done;
337            }
338            if(strcmp(right_str ,"0") == 0){
339                strcat(expression ,left_str);         //a-0->a
340                goto done;
341            }
342        }
343        else if(root->operator == '*'){               //simplify when 0*a,
           a*0,1*a,a*1
344            if(strcmp(left_str ,"0") == 0 || strcmp(right_str ,"0") == 0){
345                strcat(expression ,"0");              //0*0->0
346                goto done;
347            }
348            if(strcmp(left_str ,"1") == 0){
349                strcat(expression ,right_str);        //1*a->a
350                goto done;
351            }
352            if(strcmp(right_str ,"1") == 0){
353                strcat(expression ,left_str);         //a*1->a
354                goto done;
355            }
356
357        }
358        else if(root->operator == '/'){               //simplify when 0/a,
           a/1
359            if(strcmp(left_str ,"0") == 0){
360                strcat(expression ,"0");              //0/a->0
361                goto done;
362            }
363            if(strcmp(right_str ,"1") == 0){
364                strcat(expression ,left_str);         //a/1->a
```

```
365              goto done;
366          }
367      }else if(root->operator == '^'){          //simplify when a^1,
          a^0
368          if(strcmp(right_str ,"1") == 0){
369              strcat(expression ,left_str);      //a^1->a
370              goto done;
371          }
372          if(strcmp(right_str ,"0") == 0){
373              strcat(expression ,"1");           //a^0->1
374              goto done;
375          }
376      }
377      if(root->left->type == 2 && precedence(root->operator) >
          precedence(root->left->operator)){
378          strcat(expression ,"(");
379          strcat(expression ,left_str);          //add () if the left
              child operator is smaller than root
380          strcat(expression ,")");
381      }else{
382          strcat(expression ,left_str);          //else no ()
383      }
384      int len = strlen(expression);
385      expression[len] = root->operator;          //push the operator
386      expression[len + 1] = '\0';
387
388      if((root->right->type == 2 && precedence(root->operator) >
          precedence(root->right->operator))
389      || (root->right->type == 2 && root->operator == '/')){
390          strcat(expression ,"(");
391          strcat(expression ,right_str);          //add () if the
              right child operator is smaller than root
392          strcat(expression ,")");
393      }else{
394          strcat(expression ,right_str);          //else no ()
395      }
396
397  done:
```

```
398          free(left_str);                              //free the temporal
                 string
399          free(right_str);
400          return expression;                           //return expression
401      }
402
403      return NULL;                                      //return NULL when
         something wrong
404  }
```

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.