Autograd for Algebraic Expressions

Author: Anonymous Author

April 7, 2026

Contents

Chapter 1

Introduction

1.1 Question illustration

In this issue,we need to accomplish a algorithm that can do automatic differentiation.
In the user's window , one would input an infix expression that composed of the symbols of operators and variations,like bracket power and etc.
Particularly, I accomplish the part -**Bonus** -which would reach the need of doing differentiation for *ln*,*cos* and etc

1.2 My Way of Thinking

I think we can divide this huge question into parts

1. Depending on the given infix expression,we need to build a tree to store the operators and variation.
2. Process the tree though recursion.
3. Print the tree in the infix oder.

That's how I write this programme.

Chapter 2

Algorithm Specifcation

2.1 specialization of my program

2.1.1 To Build A Tree for the Given Indix Expressions

initiation: I build two **double link list** to simulate two stack– variation_stack and operator_stack

Listing 2.1 c

```c
struct node* tree_establish_function(char* input){
    int i=0;
    variation_head=NULL;
    variation_tail=NULL;
    operator_head=NULL;
    operator_tail=NULL;
    ......
}
```

Listing 2.2 c

```c
struct node{
    char value[LENGTH];
    int precedence;
    int type;//0 for operator,1 for variation
    struct node *left,*right;//left means the front element in
       the  stack,right means the following element in the
       stack
    struct node *son1,*son2;//son means the son node in the
       tree
};
```

structure: I use the following strategies to build a tree.

1. to check if it is a variation or an operator
2. if it is a vatiation

- than push it into the variation stack

3. if it is an operator
   - to check its precedence
   - if its precedence is lower than that in the stack
   - or push it into the stack
   - than to operate until the current operator(put it in the tree) owns the lowest precedence.
   - push it into the stack

Listing 2.3 structure

```
while(input[i]!='\0'){
    char s[LENGTH];
    int si=0;
    //to store the element in the char-list s
    //and I identify a variation-is_variation-to check if
       it is a variation.
    if((input[i]>='a'&&input[i]<='z')||(input[i]>='0'&&
       input[i]<='9')){
        while((input[i]>='a'&&input[i]<='z')||(input[i]>='0
           '&&input[i]<='9')){
            s[si]=input[i];
            si++;
            i++;
        }
        s[si]='\0';

        int is_variation;
        if(if_it_is_reserved(s)){
            is_variation = 0;
        }
        else{
            is_variation = 1;
        }
        struct node* n=(struct node*)malloc(sizeof(struct
           node));
        strcpy(n->value,s);
        n->type = is_variation;
        //than push it into the relevent stack
        if(is_variation){
            push_variation(n);
        }
        else{
            push_operator(n);
        }
    }
    //handling the cases where the variation is nagtive
```

```
33          //we recognize it as a variation
34          else if(input[i]=='-'&&(i==0||input[i-1]=='('||input[i
                -1]==',')))){
35              s[si]=input[i];
36              i++;
37              si++;
38              while((input[i]>='a'&&input[i]<='z')||(input[i]>='0
                    '&&input[i]<='9')){
39                  s[si]=input[i];
40                  si++;
41                  i++;
42              }
43              s[si]='\0';
44              struct node* n=(struct node*)malloc(sizeof(struct
                    node));
45              strcpy(n->value,s);
46              n->type=1;
47              push_variation(n);
48          }
49          //handling the operation
50          else{
51              s[si]=input[i];
52              si++;
53              i++;
54              s[si]='\0';
55              struct node* n=(struct node*)malloc(sizeof(struct
                    node));
56              strcpy(n->value,s);
57              n->type=0;
58              //"("has the lowerst precedence . But if there is a
                    ")",we need to push all the operator out under
                    "("
59              if(strcmp(s,"(")==0){
60                  push_operator(n);
61              }
62              else if(strcmp(s,")")==0){
63                  free(n);
64                  while(operator_tail&&strcmp(operator_tail->
                        value,"(")!=0){
65                      combine_one();
66                  }
67                  struct node* left_bracket=pop_operator();
68                  if(left_bracket){
69                      free(left_bracket);
70                  }
71                  if(operator_tail&&if_it_is_reserved(
                        operator_tail->value)){
72                      combine_one();
73                  }
```

```
74              }
75              //to handle the cases where ","which can infer that
                    there is a reserved word in the front.
76              else if(strcmp(s,",")==0){
77                  free(n);
78                  while(operator_tail&&strcmp(operator_tail->
                        value,"(")!=0){
79                      combine_one();
80                  }
81              }
82              //handling the normalsituation
83              else{
84                  while(operator_tail&&strcmp(operator_tail->
                        value,"(")!=0&&return_it_precedence(
                        operator_tail->value)>=return_it_precedence(
                        s)){
85                      combine_one();
86                  }
87                  push_operator(n);
88              }
89          }
90      }
91      //to make sure that all the operators is in the tree
92      while(operator_tail){
93          combine_one();
94      }
95      return variation_tail;
```

2.1.2 Do the Diferentiation

Structure We need to use recursion to simulate the process of differentiation. The recursive differentiation approach in my program follows these principles:

- Base Case: Leaf Nodes If the current node is a leaf (variable or constant), we directly return its derivative:
- +; ;=; : For each operator node, we recursively differentiate its child nodes and combine them according to calculus rules:
- reserved word: Special handling for built-in functions:
  - $\ln(u)$: derivative is $\frac{u^{\theta}}{u}$
  - $\sin(u)$: derivative is u^{θ} $\cos(u)$
  - $\cos(u)$: derivative is u^{θ} ($\sin(u)$)
  - $\exp(u)$: derivative is u^{θ} $\exp(u)$
  - $\log_b(a)$: derivative is $\frac{\ln(a)^{\theta}}{\ln(b)}$ $\frac{\ln(a)\ \ln(b)^{\theta}}{\ln(b)^2}$

This recursive approach simulates the differentiation.

Specialization: The following is the code

Listing 2.4 Specialization

```
void do_the_differentiation(struct node* root,char*
    all_the_variations,int variations_count,struct node*
    res_root){
     if(root==NULL)
         return ;
     if(root->son1==NULL&&root->son2==NULL){
         res_root->type=1;
         char*v=root->value;
         char*d=all_the_variations;
         // v means the value of root
         // d means the target variable
         //if v==d than the result=1,or the result=0;
         //in my algorithm ,i have already considered the
            situation --nagative variable.
         if(strcmp(v,d)==0){
         strcpy(res_root->value,"1");
         }
         else if(v[0]=='-'&&strcmp(v+1,d)==0){
         strcpy(res_root->value,"-1");
         }
         else if(d[0]=='-'&&strcmp(v,d+1)==0){
         strcpy(res_root->value,"-1");
         }
         else if(v[0]=='-'&&d[0]=='-'&&strcmp(v+1,d+1)==0){
         strcpy(res_root->value,"1");
         }
         else{
         strcpy(res_root->value,"0");
         }
         res_root->son1=res_root->son2=NULL;
         return;
         }
     // the situation that v is an operator
     if(root->type==0){
         //+,-
         if(strcmp(root->value,"+")==0||strcmp(root->value,"-")
            ==0){
             res_root->son1=(struct node*)malloc(sizeof(struct
                node));
             res_root->son2=(struct node*)malloc(sizeof(struct
                node));
             strcpy(res_root->value,root->value);
             res_root->type=0;
             do_the_differentiation(root->son1,
                all_the_variations,variations_count,res_root->
                son1);
```

```
39              do_the_differentiation(root->son2,
                    all_the_variations,variations_count,res_root->
                    son2);
40          }
41          //*
42          else if(strcmp(root->value,"*")==0){
43              res_root->son1=(struct node*)malloc(sizeof(struct
                    node));
44              res_root->son2=(struct node*)malloc(sizeof(struct
                    node));
45              strcpy(res_root->value,"+");
46              res_root->type=0;
47              strcpy(res_root->son1->value,"*");
48              res_root->son1->son1=(struct node*)malloc(sizeof(
                    struct node));
49              do_the_differentiation(root->son1,
                    all_the_variations,variations_count,res_root->
                    son1->son1);
50              res_root->son1->son2=copy_tree(root->son2);
51              strcpy(res_root->son2->value,"*");
52              res_root->son2->son1=copy_tree(root->son1);
53              res_root->son2->son2=(struct node*)malloc(sizeof(
                    struct node));
54              do_the_differentiation(root->son2,
                    all_the_variations,variations_count,res_root->
                    son2->son2);
55          }
56          //  /
57          else if(strcmp(root->value,"/")==0){
58
59              res_root->son1=(struct node*)malloc(sizeof(struct
                    node));
60              res_root->son2=(struct node*)malloc(sizeof(struct
                    node));
61              strcpy(res_root->value,"/");
62              res_root->type=0;
63              strcpy(res_root->son1->value,"-");
64              res_root->son1->son1=(struct node*)malloc(sizeof(
                    struct node));
65              res_root->son1->son2=(struct node*)malloc(sizeof(
                    struct node));
66              strcpy(res_root->son1->son1->value,"*");
67              res_root->son1->son1->son1=(struct node*)malloc(
                    sizeof(struct node));
68              do_the_differentiation(root->son1,
                    all_the_variations,variations_count,res_root->
                    son1->son1->son1);
69              res_root->son1->son1->son2=copy_tree(root->son2);
70              strcpy(res_root->son1->son2->value,"*");
```

```
71              res_root->son1->son2->son1=copy_tree(root->son1);
72              res_root->son1->son2->son2=(struct node*)malloc(
                    sizeof(struct node));
73              do_the_differentiation(root->son2,
                    all_the_variations,variations_count,res_root->
                    son1->son2->son2);
74              strcpy(res_root->son2->value,"^");
75              res_root->son2->son1=copy_tree(root->son2);
76              res_root->son2->son2=(struct node*)malloc(sizeof(
                    struct node));
77              strcpy(res_root->son2->son2->value,"2");
78              res_root->son2->son2->son1=res_root->son2->son2->
                    son2=NULL;
79          }
80          else if(strcmp(root->value,"pow")==0||strcmp(root->
                value,"^")==0){
81          strcpy(res_root->value,"*");
82          res_root->type=0;
83          res_root->son1 = copy_tree(root);
84          res_root->son2 = (struct node*)malloc(sizeof(struct
                node));
85          strcpy(res_root->son2->value,"+");
86          res_root->son2->type=0;
87          res_root->son2->son1 = (struct node*)malloc(sizeof(
                struct node));
88          strcpy(res_root->son2->son1->value,"*");
89          res_root->son2->son1->type=0;
90          res_root->son2->son1->son1 = (struct node*)malloc(
                sizeof(struct node));
91          do_the_differentiation(root->son2, all_the_variations,
                variations_count, res_root->son2->son1->son1);
92          res_root->son2->son1->son2 = (struct node*)malloc(
                sizeof(struct node));
93          strcpy(res_root->son2->son1->son2->value,"ln");
94          res_root->son2->son1->son2->type=0;
95          res_root->son2->son1->son2->son1 = copy_tree(root->son1
                );
96          res_root->son2->son1->son2->son2 = NULL;
97          res_root->son2->son2 = (struct node*)malloc(sizeof(
                struct node));
98          strcpy(res_root->son2->son2->value,"*");
99          res_root->son2->son2->type=0;
100         res_root->son2->son2->son1 = copy_tree(root->son2);
101         res_root->son2->son2->son2 = (struct node*)malloc(
                sizeof(struct node));
102         strcpy(res_root->son2->son2->son2->value,"/");
103         res_root->son2->son2->son2->type=0;
104         res_root->son2->son2->son2->son1 = (struct node*)malloc
                (sizeof(struct node));
```

```
105         do_the_differentiation(root->son1, all_the_variations,
               variations_count, res_root->son2->son2->son2->son1);
106         res_root->son2->son2->son2->son2 = copy_tree(root->son1
               );
107     }
108     //log
109     else if(strcmp(root->value,"log")==0){
110     struct node* new_root=(struct node*)malloc(sizeof(struct
           node));
111     strcpy(new_root->value,"/");
112     new_root->type=0;
113     struct node* ln_u=(struct node*)malloc(sizeof(struct node))
           ;
114     strcpy(ln_u->value,"ln");
115     ln_u->type=0;
116     ln_u->son1=copy_tree(root->son2);
117     ln_u->son2=NULL;
118     struct node* ln_b=(struct node*)malloc(sizeof(struct node))
           ;
119     strcpy(ln_b->value,"ln");
120     ln_b->type=0;
121     ln_b->son1=copy_tree(root->son1);
122     ln_b->son2=NULL;
123     new_root->son1=ln_u;
124     new_root->son2=ln_b;
125     do_the_differentiation(new_root,all_the_variations,
           variations_count,res_root);
126     return;
127 }
128    //sin
129     else if(strcmp(root->value,"sin")==0){
130     strcpy(res_root->value,"*");
131     res_root->type=0;
132     res_root->son1=(struct node*)malloc(sizeof(struct node));
133     strcpy(res_root->son1->value,"cos");
134     res_root->son1->type=0;
135     res_root->son1->son1=copy_tree(root->son1);
136     res_root->son1->son2=NULL;
137     res_root->son2=(struct node*)malloc(sizeof(struct node));
138     do_the_differentiation(root->son1,all_the_variations,
           variations_count,res_root->son2);
139 }
140     //ln
141     else if(strcmp(root->value,"ln")==0){
142     strcpy(res_root->value,"/");
143     res_root->type=0;
144     res_root->son1 = (struct node*)malloc(sizeof(struct node));
145     do_the_differentiation(root->son1, all_the_variations,
           variations_count, res_root->son1);
```

```
146     res_root->son2 = copy_tree(root->son1);
147 }
148     //cos
149     else if(strcmp(root->value,"cos")==0){
150     strcpy(res_root->value,"*");
151     res_root->type=0;
152     res_root->son1=(struct node*)malloc(sizeof(struct node));
153     res_root->son2=(struct node*)malloc(sizeof(struct node));
154     strcpy(res_root->son1->value,"*");
155     res_root->son1->type=0;
156     res_root->son1->son1=(struct node*)malloc(sizeof(struct
          node));
157     strcpy(res_root->son1->son1->value,"-1");
158     res_root->son1->son1->type=1;
159     res_root->son1->son1->son1=res_root->son1->son1->son2=NULL;
160     res_root->son1->son2=(struct node*)malloc(sizeof(struct
          node));
161     strcpy(res_root->son1->son2->value,"sin");
162     res_root->son1->son2->type=0;
163     res_root->son1->son2->son1=copy_tree(root->son1);
164     res_root->son1->son2->son2=NULL;
165     res_root->son2=(struct node*)malloc(sizeof(struct node));
166     res_root->son2->son1=res_root->son2->son2=NULL;
167
168     do_the_differentiation(root->son1,all_the_variations,
          variations_count,res_root->son2);
169 }
170     //exp
171     else if(strcmp(root->value,"exp")==0){
172     strcpy(res_root->value,"*");
173     res_root->type=0;
174     res_root->son1 = (struct node*)malloc(sizeof(struct node)); *  on1);
175     strcpy(res_root->son1->value,"exp");
176     res_root->son1->type=0;
177     res_root->son1->son1 = copy_tree(root->son1);
178     res_root->son1->son2 = NULL;
179     res_root->son2 = (struct node*)malloc(sizeof)     e*)mallocll -UL  e*)mall
```

Chapter 3

testing result

Figure 3.1 4 variable

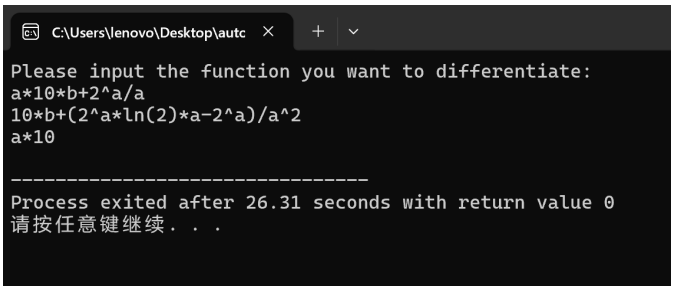

Figure 3.2 with constant

Figure 3.3 long name

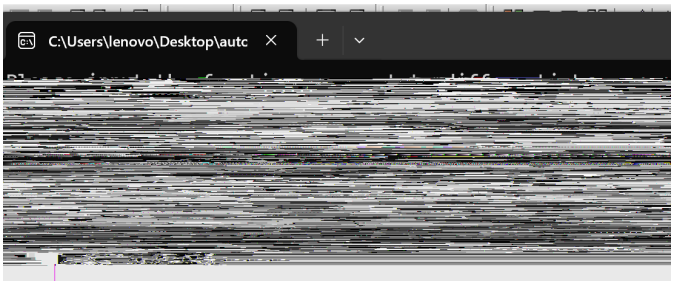

Figure 3.4 ln

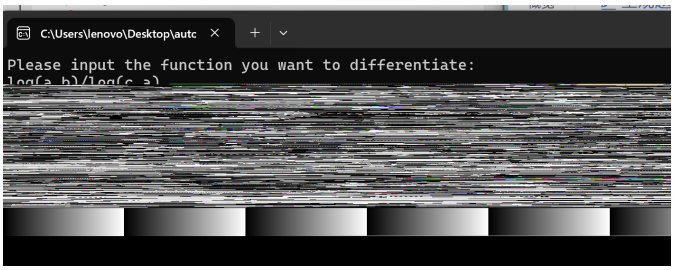

Figure 3.5 log

Particularly, i will show you something special.

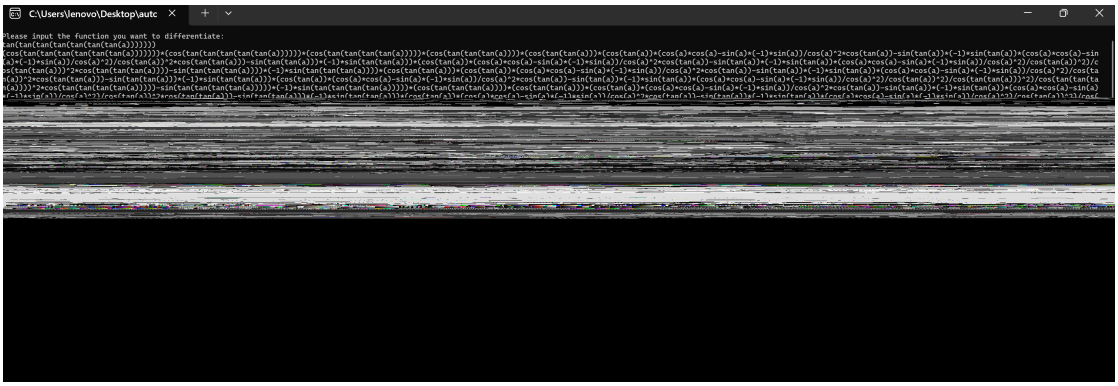

Figure 3.6 **tan**

Chapter 4

Analysis and Comment

we consider the most complex part of my programme. It is definately the differentiation part.

4.1 time complexity

We consider the worst cases. In the '+','*'...section,their recursions need to be conduct twice. So that the mathmatics equation should be:

$$T(k) = 2 \quad T(k \quad 1) + C; \quad T(1) = C$$
$$T(k) = 2[2T(k \quad 2) + C] + C = 2^2T(k \quad 2) + (2 + 1)C$$
$$= 2^3T(k \quad 3) + (2^2 + 2 + 1)C$$

$$= 2^{k \ 1}T(1) + (2^{k \ 2} + 2^{k \ 3} + \quad + 2^0)C$$
$$= 2^{k \ 1}C + (2^{k \ 1} \quad 1)C$$
$$= C \quad (2^k \quad 1)$$
$$T(k) = O(2^k)$$

4.2 space complexity

we just need to consider the worst cases,differentiation part.
consider the worst cases where all the node has two child. Then,every node would push to node into the stack :

Space = nodes + stack depth

m = original tree nodes; d = tree height; $C = 84$ bytes per node

Original tree space = $C \quad m = O(m)$

Stack space = $O(d) = O(m)$ (worst case: chain tree)

Derivative tree space = $C \quad T(k) = C \quad (2^k \quad 1) = O(2^m)$

Total space = $O(m) + O(m) + O(2^m) = O(2^m)$

Worst-case space complexity = $O(2^m)$

Chapter 5

Declaration

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

Appendix A

code

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define LENGTH 20
#define MAXN 1000
struct node{
    char value[LENGTH];
    int precedence;
    int type;//0 for operator,1 for variation
    struct node *left,*right;//left means the front element in the  stack,right means the following element in the stack
    struct node *son1,*son2;//son means the son node in the tree
};
struct node *variation_head=NULL,*variation_tail=NULL;
struct node *operator_head=NULL,*operator_tail=NULL;
int if_it_is_reserved(char* s);
int to_store_all_the_variations(char* input,char all_the_variations[MAXN][LENGTH]){
    //in this section we would find all the variables and then store it in a array. finally , return it with the total number of variation.
    int i=0,variations_count=0;
    while(input[i]!='\0'){
        char s[LENGTH];
        int si=0;//si is the number of stack number.
        if(input[i]>='a'&&input[i]<='z'){
            while(input[i]>='a'&&input[i]<='z'){
                s[si]=input[i];
                si++;
                i++;
            }
            s[si]='\0';
            int flag=1;
```

```
251             }
252             s[si]='\0';
253             struct node* n=(struct node*)malloc(sizeof(struct
                    node));
254             strcpy(n->value,s);
255             n->type=1;
256             push_variation(n);
257         }
258         //handling the operation
259         else{
260             s[si]=input[i];
261             si++;
262             i++;
263             s[si]='\0';
264             struct node* n=(struct node*)malloc(sizeof(struct
                    node));
265             strcpy(n->value,s);
266             n->type=0;
267             //"("has the lowerst precedence . But if there is a
                    ")",we need to push all the operator out under
                    "("
268             if(strcmp(s,"(")==0){
269                 push_operator(n);
270             }
271             else if(strcmp(s,")")==0){
272                 free(n);
273                 while(operator_tail&&strcmp(operator_tail->
                        value,"(")!=0){
274                     combine_the_node();
275                 }
276                 struct node* left_bracket=pop_operator();
277                 if(left_bracket){
278                     free(left_bracket);
279                 }
280                 if(operator_tail&&if_it_is_reserved(
                        operator_tail->value)){
281                     combine_the_node();
282                 }
283             }
284             //to handle the cases where ","which can infer that
                    there is a reserved word in the front.
285             else if(strcmp(s,",")==0){
286                 free(n);
287                 while(operator_tail&&strcmp(operator_tail->
                        value,"(")!=0){
288                     combine_the_node();
289                 }
290             }
291             //handling the normalsituation
```

```
292             else{
293                 while(operator_tail&&strcmp(operator_tail->
                        value,"(")!=0&&return_it_precedence(
                        operator_tail->value)>=return_it_precedence(
                        s)){
294                     combine_the_node();
295                 }
296                 push_operator(n);
297             }
298         }
299     }
300     //to make sure that all the operators is in the tree
301     while(operator_tail){
302         //to operate all the node in the remaining array.
303         combine_the_node();
304     }
305     return variation_tail;
306 }
307 void print_tree_structure(struct node* root,int level){
308     //in this section,we need to print the tree structure ,so
            that we can easily check whether  our tree establishment
            is done.
309     if(root==NULL)
310         return;
311     //recursing to the next level.
312     print_tree_structure(root->son2,level+1);
313     for(int i=0;i<level;i++)
314         printf("    ");
315     printf("%s\n",root->value);
316     //print the right node
317     print_tree_structure(root->son1,level+1);
318 }
319 void print_the_variations(char all_the_variations[MAXN][LENGTH
        ],int variations_count){
320     //in this section ,we need to print an array .
321     printf("The variations in the function are:\n");
322     for(int i=0;i<variations_count;i++){
323         printf("%s\n",all_the_variations[i]);
324     }
325 }
326 void print_infix(struct node* root,int parent_precedence) {
327     // in this section , we need to print the final anwser that
            in the infix order.
328     if (root == NULL) return;
329     if (root->son1 == NULL && root->son2 == NULL) {
330         // if there is a '-' that we need to add it with a pair
                of ()
331         if(root->value[0]=='-'){
332             printf("(");
```

```
333             printf("%s", root->value);
334             printf(")");
335             return;
336         }
337         else{
338         printf("%s", root->value);
339         return;
340         }
341     }
342     // to cover the situation if there is a reserved word
343     if (if_it_is_reserved(root->value)) {
344         printf("%s(", root->value);
345         print_infix(root->son1,parent_precedence);
346         if (root->son2 != NULL) {
347             printf(",");
348             print_infix(root->son2,return_it_precedence(root->
                value));
349         }
350         printf(")");
351     }
352     else {
353         //the other situation is that the precedence of the
            current node is lower than the front node.
354         // then it needs a pair of ()
355         if(return_it_precedence(root->value)<parent_precedence
            ||strcmp(root->value,"-")==0)
356        { printf("(");
357         print_infix(root->son1,return_it_precedence(root->value
            ));
358         printf("%s", root->value);
359         print_infix(root->son2,return_it_precedence(root->value
            ));
360         printf(")");
361     }
362     else{
363         //the most common situation that just print
364         print_infix(root->son1,return_it_precedence(root->value
            ));
365         printf("%s", root->value);
366         print_infix(root->son2,return_it_precedence(root->value
            ));
367     }
368     }
369 }
370 struct node* copy_tree(struct node* root){
371     //in this section , we need to replace the root node with
        the n node
372     if (!root)
373         return NULL;
```

```
374     //copy the node elements one by one .
375     struct node* n=(struct node*)malloc(sizeof(struct node));
376     *n=*root;
377     n->son1=copy_tree(root->son1);
378     n->son2=copy_tree(root->son2);
379     n->left=n->right = NULL;
380     return n;
381 }
382 void do_the_differentiation(struct node* root,char*
        all_the_variations,int variations_count,struct node*
        res_root){
383     if(root==NULL)
384         return ;
385     if(root->son1==NULL&&root->son2==NULL){
386         res_root->type=1;
387         char*v=root->value;
388         char*d=all_the_variations;
389         // v means the value of root
390         // d means the target variable
391         //if v==d than the result=1,or the result=0;
392         //in my algorithm ,i have already considered the
                situation --nagative variable.
393         //the following situation handles the constant
                situation.
394         if(strcmp(v,d)==0){
395         strcpy(res_root->value,"1");
396         }
397         else if(v[0]=='-'&&strcmp(v+1,d)==0){
398         strcpy(res_root->value,"-1");
399         }
400         else if(d[0]=='-'&&strcmp(v,d+1)==0){
401         strcpy(res_root->value,"-1");
402         }
403         else if(v[0]=='-'&&d[0]=='-'&&strcmp(v+1,d+1)==0){
404         strcpy(res_root->value,"1");
405         }
406         else{
407             // if it is not the target variable.
408             //then it is zero
409         strcpy(res_root->value,"0");
410         }
411         res_root->son1=res_root->son2=NULL;
412         return;
413         }
414     // the situation that v is an operator
415     if(root->type==0){
416         //+,-
417         // Handles differentiation for nodes with type 0 where
                the operation
```

```
418         // is either addition ('+') or subtraction ('-'). For a
                node representing
419         // f(x) ± g(x), the derivative is computed as f'(x) ± g
               '(x). This function
420         // recursively differentiates both operands and
               constructs a new node
421         // with the same operator, linking the differentiated
               results as children.
422         if(strcmp(root->value,"+")==0||strcmp(root->value,"-")
               ==0){
423             res_root->son1=(struct node*)malloc(sizeof(struct
                   node));
424             res_root->son2=(struct node*)malloc(sizeof(struct
                   node));
425             strcpy(res_root->value,root->value);
426             res_root->type=0;
427             do_the_differentiation(root->son1,
                   all_the_variations,variations_count,res_root->
                   son1);
428             do_the_differentiation(root->son2,
                   all_the_variations,variations_count,res_root->
                   son2);
429         }
430         //*
431         // Processes nodes with operator '*'. Implements the
               product rule:
432         // d(u*v)/dx = u' * v + u * v'. The result is
               structured as an addition
433         // node ('+') with two children, each being a
               multiplication node ('*').
434         // The left child contains (u' * v) and the right child
                contains (u * v').
435         // Recursive calls are made to differentiate u and v
               accordingly.
436         else if(strcmp(root->value,"*")==0){
437             res_root->son1=(struct node*)malloc(sizeof(struct
                   node));
438             res_root->son2=(struct node*)malloc(sizeof(struct
                   node));
439             strcpy(res_root->value,"+");
440             res_root->type=0;
441             strcpy(res_root->son1->value,"*");
442             res_root->son1->son1=(struct node*)malloc(sizeof(
                   struct node));
443             do_the_differentiation(root->son1,
                   all_the_variations,variations_count,res_root->
                   son1->son1);
444             res_root->son1->son2=copy_tree(root->son2);
445             strcpy(res_root->son2->value,"*");
```

```
446             res_root->son2->son1=copy_tree(root->son1);
447             res_root->son2->son2=(struct node*)malloc(sizeof(
                    struct node));
448             do_the_differentiation(root->son2,
                    all_the_variations,variations_count,res_root->
                    son2->son2);
449         }
450         //  /
451         // Processes nodes with operator '/'. Implements the
                quotient rule:
452         // d(u/v)/dx = (u'*v - u*v') / v^2. The result is a
                division node ('/').
453         // The numerator is a subtraction node ('-') containing
                 two product nodes:
454         // (u'*v) and (u*v'). The denominator is a power node
                ('^') representing v^2.
455         else if(strcmp(root->value,"/")==0){
456             res_root->son1=(struct node*)malloc(sizeof(struct
                    node));
457             res_root->son2=(struct node*)malloc(sizeof(struct
                    node));
458             strcpy(res_root->value,"/");
459             res_root->type=0;
460             strcpy(res_root->son1->value,"-");
461             res_root->son1->son1=(struct node*)malloc(sizeof(
                    struct node));
462             res_root->son1->son2=(struct node*)malloc(sizeof(
                    struct node));
463             strcpy(res_root->son1->son1->value,"*");
464             res_root->son1->son1->son1=(struct node*)malloc(
                    sizeof(struct node));
465             do_the_differentiation(root->son1,
                    all_the_variations,variations_count,res_root->
                    son1->son1->son1);
466             res_root->son1->son1->son2=copy_tree(root->son2);
467             strcpy(res_root->son1->son2->value,"*");
468             res_root->son1->son2->son1=copy_tree(root->son1);
469             res_root->son1->son2->son2=(struct node*)malloc(
                    sizeof(struct node));
470             do_the_differentiation(root->son2,
                    all_the_variations,variations_count,res_root->
                    son1->son2->son2);
471             strcpy(res_root->son2->value,"^");
472             res_root->son2->son1=copy_tree(root->son2);
473             res_root->son2->son2=(struct node*)malloc(sizeof(
                    struct node));
474             strcpy(res_root->son2->son2->value,"2");
475             res_root->son2->son2->son1=res_root->son2->son2->
                    son2=NULL;
```

```
476         }
477         //pow, ^
478         // Processes nodes with operator 'pow' or '^'. For u(x)
               ^v(x), derivative:
479         // u^v * (v' * ln(u) + v * u' / u). Implemented as a
               product node that
480         // multiplies the original function by a sum: (v' * ln(
               u)) + (v * u' / u).
481         else if(strcmp(root->value,"pow")==0||strcmp(root->
               value,"^")==0){
482         strcpy(res_root->value,"*");
483         res_root->type=0;
484         res_root->son1 = copy_tree(root);
485         res_root->son2 = (struct node*)malloc(sizeof(struct
               node));
486         strcpy(res_root->son2->value,"+");
487         res_root->son2->type=0;
488         res_root->son2->son1 = (struct node*)malloc(sizeof(
               struct node));
489         strcpy(res_root->son2->son1->value,"*");
490         res_root->son2->son1->type=0;
491         res_root->son2->son1->son1 = (struct node*)malloc(
               sizeof(struct node));
492         do_the_differentiation(root->son2, all_the_variations,
               variations_count, res_root->son2->son1->son1);
493         res_root->son2->son1->son2 = (struct node*)malloc(
               sizeof(struct node));
494         strcpy(res_root->son2->son1->son2->value,"ln");
495         res_root->son2->son1->son2->type=0;
496         res_root->son2->son1->son2->son1 = copy_tree(root->son1
               );
497         res_root->son2->son1->son2->son2 = NULL;
498         res_root->son2->son2 = (struct node*)malloc(sizeof(
               struct node));
499         strcpy(res_root->son2->son2->value,"*");
500         res_root->son2->son2->type=0;
501         res_root->son2->son2->son1 = copy_tree(root->son2);
502         res_root->son2->son2->son2 = (struct node*)malloc(
               sizeof(struct node));
503         strcpy(res_root->son2->son2->son2->value,"/");
504         res_root->son2->son2->son2->type=0;
505         res_root->son2->son2->son2->son1 = (struct node*)malloc
               (sizeof(struct node));
```

```
669
670             if(root->son2->type==1&&strcmp(root->son2->value,"1
                    ")==0)
671                 return simplify_the_tree(root->son1);
672         }
673         // 0/
674         if(strcmp(root->value,"/")==0){
675             if((root->son1->type==1&&strcmp(root->son1->value,"
                    0")==0)||(root->son2->type==1&&strcmp(root->son2
                    ->value,"0")==0)){
676                 struct node* zero=(struct node*)malloc(sizeof(
                        struct node));
677                 zero->type=1;
678                 strcpy(zero->value,"0");
679                 zero->son1=zero->son2=NULL;
680                 return zero;
681             }
682         }
683     }
684
685     return root;
686 }
687 int main(){
688     char input[MAXN],all_the_variations[MAXN][LENGTH];
689     int the_variations_number;
690     printf("Please input the function you want to differentiate
            :\n");
691     scanf("%s",input);
692     the_variations_number=to_store_all_the_variations(input,
            all_the_variations);
693     //print_the_variations(all_the_variations,
            the_variations_number);
694     struct node* root=tree_establish_function(input);
695     //print_tree_structure(root,0);
696     //traverse all the variable that inputing.
697     for(int i=0;i<the_variations_number;i++){
698         struct node* res_root=(struct node*)malloc(sizeof(
                struct node));
699         do_the_differentiation(root,all_the_variations[i],
                the_variations_number,res_root);
700         res_root=simplify_the_tree(res_root);
701         print_infix(res_root,0);
702         printf("\n");
703     }
704
705     return 0;
706 }
```