FDS Project 2 (Hard): Autograd for Algebraic Expressions (with Bonus)

Anonymous Author

Date: 2026-04-06

1. Introduction

> 题目：Autograd for Algebraic Expressions
> 目标：输入一个中缀表达式，输出其对每个变量的导函数表达式。
> 输出格式：对于每个变量名，分别输出 `变量名：导函数表达式`。

本项目实现了一个“符号树驱动”的自动求导程序，支持：

- 运算符：`+ - * / ^`
- 括号：`( )`
- 变量与整数常量
- Bonus 函数：`ln, log, sin, cos, tan, pow, exp`

1.1 该程序的特性

与题目要求相比，本程序在功能上更完整：

1. **支持复杂函数与多参数函数**。

```
***@ubuntu-linux-2404:***/project2$ ./main
Please input an algebraic expression:
cos(sin(ln(KFCv50)))
KFCv50: -sin(sin(ln(KFCv50))) * cos(ln(KFCv50)) * 1 / KFCv50
```

2. **提高容错性**（例如输入中非变量名内的任意随机空格，divided by zero，非法变量名提示，非法函数名提示等）。

```
x/0
==========Error: division by zero.==========

hello(world)
==========Error: invalid function name "hello(world)".==========

4KFCv50
==========Error: invalid token name "4KFCv50".==========
```

3. **支持多重负号正号嵌套**（例如连续 `---x`、`A-(-B)` 等）。

```
Please input an algebraic expression:
-- - -x
x: 1
```

4. **表达式化简**（常量折叠 + 多条代数规则），输出更短更可读。
   (a) 常数折叠
   (b) 零元规则
   (c) 单位元规则
   (d) 符号化简

5. 函数体系做了“封闭化”处理，防止 `tan(x)` 求导后出现 `sec()` 等其他函数。

1.2 Bonus

1.2.1 Not using complex containers provided in STL

在 `Stack.h` 中定义链式通用栈的结构体和相关函数；在 `Stack.c` 中实现这些函数。

1.2.2 Support for expressions containing mathematical functions

详见第 1.4 章。

1.2.3 Simplify an algebraic expression to reduce the length of the result by applying at least two rules.

详见第 1.5 章。

1.3 区分双操作数减号与单操作数负号

在 `infixToPostfix()` 中，进行了初步**清洗**操作，即过滤多于空格，并规范返回字符数组的格式：

- 减号的前一字符（如有）为空格，后一字符（如有）为空格；
- 负号后侧紧跟变量名/数字/函数。

随后在 `buildTree()` 中利用这一结果进行区分：

- 减号对应一个符号节点；
- 负号对应任意节点的 `minusFlag = 1`。

1.4 缩小初等函数表示体系

建树时将多参量函数重写：

1. `tan(A) -> sin(A)/cos(A)`
2. `log(A,B) -> ln(B)/ln(A)`
3. `pow(A,B) -> exp(B*ln(A))`

因此后续求导只需要维护四类函数规则：`sin/cos/ln/exp`。

并且求导后表达式仍 **只含这四类单操作数函数**。

形成一个**封闭函数体系**，函数数量减少，且均为单操作数函数。既减少了求导函数的压力，又方便输出与维护。

1.5 较强的化简

在 `main.c` 中的化简函数 `simplify()` 实现了：

1. 常数折叠：对 +、-、*、/、^ 的纯数值运算直接计算结果；
2. 应用零元规则：
   (a) $0 + A = A$，$A + 0 = A$，$A - 0 = A$，
   (b) $A * 0 = 0$，$0 * A = 0$，$0 / A = 0$，$A \hat{} 0 = 1$，$0 - A = -A$
3. 应用单位元规则：
   $A * 1 = A$，$1 * A = A$，$A / 1 = A$，$A \hat{} 1 = A$
4. 应用特殊等价化简：
   $A / A = 1$
5. 符号规范化化简：
   (a) $A - (-B) \to A + B$
   (b) $A + (-B) \to A - B$
   (c) $(-1) * A \to -A$
   (d) $A * (-1) \to -A$
   (e) $(-A) * (-B) \to A * B$

1.6 输出尽可能少的括号

- 子树运算符优先级低于或等于当前节点才需要加括号
- 且 `*` 和 `+` 在同等优先级时无需加括号，因为满足交换律

2. Algorithm Specification

2.1 主要数据结构

2.1.1 树

1. 表达式树结点 `TreeNode`，在 `BuildTree.h` 头文件中定义：

```
typedef struct TreeNode *Tree;
struct TreeNode{
    char op;                /* operators :'+', '-', '*', '/', '(',
                               ')','^'
                               or  0  for numbers
                               or 'A' for variables
                               or '$' for functions */
    int minusFlag;          /* whether the operator is a unary minus */
    char val[LENGTH + 5];   /* valid when op == 0,
                               to store numbers or variables as strings */
    Tree left;
    Tree right;
};
```

- `op`：结点类型（常量/变量/函数/运算符）
- `val`：数值字符串、变量名或函数名
- `minusFlag`：是否整体带负号
- `left/right`：左右子树

2.1.2 栈

2. 链式通用栈 `Stack`，在 `Stack.h` 头文件中定义：

```
typedef struct StackNode *Stack;
struct StackNode {
    void *data;          /* Store value: TreeNode or ASCII of operator */
                         /* void*: generic pointer, which means it can
                         point to any type of data */
    Stack next;          /* Pointer to next node */
};
```

> 在使用时，`Stack s` 是栈顶指针，初始化时 `s` 指向 `NULL`。

- 中缀转后缀时保存运算符
- 后缀建树时保存子树指针

2.2 主要算法的伪代码与说明

2.2.1 算法 A：中缀转后缀

> **Note**
>
> 函数：在这一步先当作一整个变量名处理。
> 输入：合法的中缀代数表达式字符串（含变量、函数、运算符及括号）。
> 输出：对应的后缀表达式字符串。

```
逐字符扫描：
  操作数/函数token -> 输出
  //函数：在这一步当作一整个变量名处理，去除括号内的空格。
  //清洗数据以防后续建树时难以界定。
  '(' -> 入栈
  ')' -> 弹栈到'('
  运算符op -> 弹出优先级>=op的栈顶，再把op入栈
最后把栈内剩余运算符全部输出
```

2.2.2 算法 B：后缀建树

> **Note**
>
> 函数：先当作一整个变量名，接着化为单操作数函数，然后在提取括号中的表达式递归展开。
> 输入：后缀表达式字符串。
> 输出：表达式树根节点。

```
扫描postfix：
  若是变量名/函数/数字 -> 新建节点入栈，标记类型
  若是二元运算符 -> 出栈右子树、左子树，组装后入栈
  若是函数 ->
    /* log(A,B) = ln(B)_temp / ln(A)_temp2 */
    /* pow(A,B) = exp(B * ln(A)) */
    /* tan(A) = sin(A) / cos(A) */
    将内层表达式递归展开
结束时栈顶即根节点
```

2.2.3 算法 C：递归求导

> **Note**
>
> 输入：表达式树根节点、求导变量。
> 输出：导数表达式树根节点。

```
常量 -> 0
变量 -> (自身为目标变量 ? 1 : 0)
+ - * / ^ -> 根据求导法则求导
ln/sin/cos/exp -> 链式求导
```

2.2.4 算法 D：化简

> **Note**
>
> 输入：表达式树根节点。
> 输出：化简后的表达式树根节点

```
后序遍历：
  先化简子树，再应用规则：
      常数折叠：对 +、-、*、/、^ 的纯数值运算直接计算结果；
    应用零元规则：
      0 + A = A, A + 0 = A, A - 0 = A,
      A * 0 = 0, 0 * A = 0, 0 / A = 0, A ^ 0 = 1, 0 - A = -A
    应用单位元规则：
      A * 1 = A, 1 * A = A, A / 1 = A, A ^ 1 = A
    应用特殊等价化简：
      A / A = 1
    符号规范化化简：
      A - (-B) → A + B
      A + (-B) → A - B
      (-1) * A → -A
      A * (-1) → -A
      (-A) * (-B) → A * B
  调整负数标记位并释放冗余节点，返回最简表达式树。
```

2.3 实现亮点

1. `cloneTree` 拷贝整棵树，避免原树与导数树共享节点。
2. `equalTree` 结构比较，用于 `A/A -> 1`。
3. `minusFlag` 统一负号的表达，与减法区分，并贯穿建树-求导-化简-输出。
4. 函数收敛策略让规则系统更稳定。

3. Testing Results

3.1 测试环境

- 系统：macOS 26, Ubuntu 22.04, Windows 11 25H2 中均正常运行。
- 编译：`make` 工具（mac, Linux）。

> **Note**
>
> 如果编译工具失效或在 Windows 中编译，请使用通用的指令：
>
> ```
> gcc main.c BuildTree.c Stack.c -o main -lm
> ```

- 运行：输入一行合法表达式

3.2 测试用例（普通文本版）

3.2.1 T1：基础四则与幂求导

- 输入表达式：

```
a+b^c*d
```

- 题目样例期望：

```
a: 1
b: c*b^(c-1)*d
c: ln(b)*b^c*d
d: b^c
```

- 程序实际输出：

```
Please input an algebraic expression:
a: 1
b: b ^ c * c * 1 / b * d
c: b ^ c * ln(b) * d
d: b ^ c
```

- 等价性说明：`c*b^(c-1)*d` 与 `b^c*c*(1/b)*d` 等价，程序未把 `c*1/b` 进一步压成 `c/b`

3.2.2 T2：基础四则与幂求导

- 输入表达式：

```
a*10*b+2^a/a
```

- 题目样例期望：

```
a: 10*b-2^a/a^2+2^a*ln(2)/a
b: a*10
```

- 程序实际输出：

```
Please input an algebraic expression:
a: 10 * b + (2 ^ a * ln(2) * a - 2 ^ a) / (a * a)
b: a * 10
```

- 等价性说明：商法则展开后与题目答案完全等价，只是没有继续合并到 `a^2` 形式

3.2.3 T3：多字符变量名

- 输入表达式：

```
xx^2/xy*xy+a^a
```

- 题目样例期望：

```
a: a^a*(1+ln(a))
xx: 2*xx
xy: 0
```

- 程序实际输出：

```
Please input an algebraic expression:
xx: (xx ^ 2 * 2 * 1 / xx * xy) / (xy * xy) * xy
xy: -(xx ^ 2) / (xy * xy) * xy + xx ^ 2 / xy
a: a ^ a * (ln(a) + a * 1 / a)
```

- 等价性说明：`a^a*(ln(a)+1)` 与 `a^a*(1+ln(a))` 等价；`xx`、`xy` 的结果是链式/商法则直接展开后的未完全化简式

3.2.4 T4：函数

- 输入表达式：

```
x*ln(y)
```

- 程序实际输出：

```
Please input an algebraic expression:
x: ln(y)
y: x * 1 / y
```

- 等价性说明：`x * 1 / y` 与 `x/y` 等价

3.2.5 T5：函数

- 输入表达式：

```
x*ln(x*y)+y*cos(x)+y*sin(2*x)
```

- 程序实际输出：

```
Please input an algebraic expression:
x: ln(x * y) + x * y / (x * y) + y * -sin(x) + y * cos(2 * x) * 2
y: x * x / (x * y) + cos(x) + sin(2 * x)
```

- 等价性说明：结构上与标准导数一致，程序保留了较多局部展开项

3.2.6 T6：函数

- 输入表达式：

```
log(a,b)/log(c,a)
```

- 程序实际输出：

```
Please input an algebraic expression:
b: ((1 / b * ln(a)) / (ln(a) * ln(a)) * ln(a) / ln(c)) / (ln(a) / ln(c) * ln(a) / ln(c))
a: ((-(ln(b) * 1 / a)) / (ln(a) * ln(a)) * ln(a) / ln(c) - ln(b) / ln(a) * (1 / a * ln(c)) / (ln(c) * ln(c))) / (ln(a) / ln(c) * ln(a) / ln(c))
c: (-(ln(b) / ln(a) * (-(ln(a) * 1 / c)) / (ln(c) * ln(c)))) / (ln(a) / ln(c) * ln(a) / ln(c))
```

- 等价性说明：与“先重写后求导”的标准公式一致；结果较长是因为当前化简规则只做局部折叠

3.2.7 压力测试

经实测，`tan()` 的多层嵌套可最多达 4 层，`sin()`、`cos()`、`exp()` 和 `ln()` 等的嵌套可达10层以上。

3.3 测试小结

1. `a+b^c*d`：输出与样例等价，差异只在未进一步化简。
2. `a*10*b+2^a/a`：输出与题目答案等价，程序保留了商法则展开式。
3. `xx^2/xy*xy+a^a`：变量名 `xx`、`xy` 被正确识别为单个变量；`a` 的结果与样例等价，`xx`、`xy` 的式子较长但正确。
4. `x*ln(y)`：符合题意，`x` 的导数为 `ln(y)`，`y` 的导数为 `x/y` 的等价形式。
5. `x*ln(x*y)+y*cos(x)+y*sin(2*x)`：乘积法则、链式法则、三角函数求导均正确。
6. `log(a,b)/log(c,a)`：程序先把双参数 `log` 收敛成 `ln` 的比值，再统一求导。

程序可以完成题目要求的主流程。

4. Analysis and Comments

4.1 时间复杂度

设输入表达式长度为 n，表达式树结点数为 m，不同变量个数为 k。

1. 中缀转后缀是一次顺序扫描，所以时间复杂度是 $O(n)$。
   这里的“顺序扫描”指的是，每个字符最多被读入、处理几次，不会反复回头扫描整段字符串。
2. 由后缀表达式建树时，每个 token 只会被压栈、出栈一次，所以时间复杂度也是 $O(n)$。
   这一步的开销主要来自结点分配和指针连接。
3. 对单个变量求导时，本质上是对整棵树做一次递归遍历，因此基础复杂度约为 $O(m)$。
   但由于乘法、除法、幂运算会构造新的子树，所以实际常数比普通遍历更大。
4. 如果表达式里有 k 个变量，程序会分别对每个变量求导一次，因此总复杂度大约是 $O(km)$。
   也就是说，变量越多，总时间越长；树越大，每次求导也越慢。
5. 化简函数 `simplify` 也是遍历整棵树，单轮复杂度约为 $O(m)$。
   当前实现是“局部规则化简”，例如常量折叠、`A+0`、`A*1` 这类规则；因此它不会做很重的全局重写。
6. 中缀输出 `treeToInfix` 同样需要遍历整棵树，单轮复杂度约为 $O(m)$。

4.2 空间复杂度

1. 表达式树本身需要 $O(m)$ 空间。每个运算符、常量、变量和函数都会对应一个结点。
2. 中缀转后缀和输出过程需要额外缓冲区，空间约为 $O(n)$。
3. 递归调用栈的深度取决于表达式树高度 h，平均接近 $O(h)$，最坏情况下可以退化到 $O(m)$。
4. 求导时会临时创建新树，所以峰值空间通常高于原表达式树，但仍然与树规模同阶。

4.3 不足与改进

不足之处：

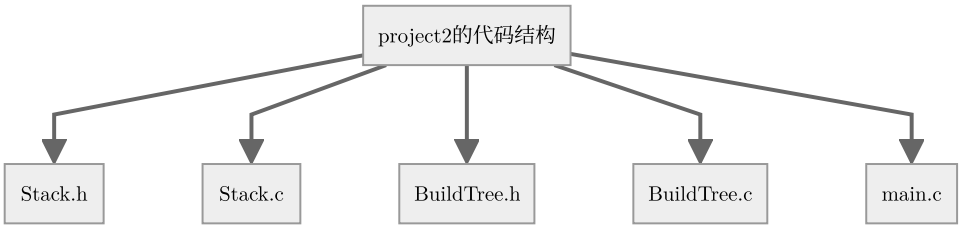
- 合并同类项与复杂结构的树的变换（如 x*(1/x)，当 1/x 是子树时候，没法消掉）。

改进方向：

1. 引入更强的化简模式匹配与结果表达式规范化。
2. 扩展更多函数支持，如原生的 `tan()`。
3. 优化部分算法，如：
   (a) 当指数函数的幂/底数为常数时，采用更简单的求导公式 $(x^n)' = nx^{n-1}$，$(a^x)' = a^x lna$；
   (b) 对数学常数如自然常数 e 的支持 (运算，化简)；

5. Appendix: Source Code

代码长度：约900行。



5.1 Stack.h

```c
#ifndef STACK_H
#define STACK_H

typedef struct StackNode *Stack;

/* === BONUS== =*/
struct StackNode {
    void *data;           /* Store value: TreeNode or ASCII of operator */
    /* void*: generic pointer, which means it can point to any type of data */
    Stack next;           /* Pointer to next node */
};
/*
 top(s: dummy head node)
  ↓
node1 → node2 → node3 → NULL
*/
Stack newStack(void);
void freeStack(Stack s);
void push(Stack s, void *data);
void *pop(Stack s);
void *top(Stack s);
int isEmpty(Stack s);

#endif
```

5.2 Stack.c

```c
#include "Stack.h"
#include <stdlib.h>

Stack newStack(void){
    /* Create a new stack with a dummy head node. */
    /* Dynamically allocate memory. */
```

```
    Stack s = (Stack)malloc(sizeof(struct StackNode));
    s->data = NULL;
    s->next = NULL;
    /*s: dummy head node*/
    return s;
}

int isEmpty(Stack s){
    return s->next == NULL;
}

void freeStack(Stack s){
    while (!isEmpty(s)) pop(s);
    free(s);
}

void push(Stack s, void *data){
    Stack newNode = (Stack)malloc(sizeof(struct StackNode));
    newNode->data = data;
    /* Make the new node point to the current top node of the stack. */
    /* Attach the original top node after the new node. */
    newNode->next = s->next;
    /* Update the dummy head node to point to the new node. */
    s->next = newNode;
}

void *pop(Stack s){
    if (isEmpty(s)) exit(1); /* Exit if the stack is empty. */
    /* Store the current top node of the stack in a temporary variable.
    */
    Stack temp = s->next;
    void *data = temp->data;
    s->next = temp->next;
    free(temp);
    return data;
}

void *top(Stack s){
    if (isEmpty(s)) exit(1); /* Exit if the stack is empty. */
    return s->next->data;
}
```

5.3 BuildTree.h

```
#ifndef BUILD_TREE_H
#define BUILD_TREE_H

#define LENGTH 5000

#define NUMBER 500    /* maximum number of variables */

```

```
8  typedef struct TreeNode *Tree;
9
10 struct TreeNode{
11     char op;                /* operators :'+', '-', '*', '/', '(',
       ')','^'
12                             or  0  for numbers
13                             or 'A' for variables
14                             or '$' for functions */
15     int minusFlag;          /* whether the operator is a unary minus.
       */
16     char val[LENGTH + 5];   /* valid when op == 0, to store numbers or
       variables as strings */
17     Tree left;
18     Tree right;
19 };
20 int isLetterOrDigit(char c);
21
22 int isLetter(char c);
23
24 int priority(char c);
25
26 char *infixToPostfix(const char *s);
27
28 /* Build a syntax tree from a null-terminated expression string. */
29 Tree buildTree(const char *s);/* Read only. */
30
31 /* Free tree */
32 void freeTree(Tree t);
33
34 /* Print tree. */
35 void printTree(Tree t, int level);
36
37 char *treeToInfix(Tree t, char *infix);
38
39 void findVariables(Tree t, char vars[][NUMBER], int *count);
40
41 #endif
```

5.4 BuildTree.c

```
1  #include "BuildTree.h"
2  #include "Stack.h"
3  #include <stdlib.h>
4  #include <string.h>
5  #include <stdio.h>
6
7  int isLetterOrDigit(char c){
8      return (c >= '0' && c <= '9') || (c >= 'a' && c <= 'z')
9          || (c >= 'A' && c <= 'Z') || (c == '_');
10 }
11
```

```c
int isLetter(char c){
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c == '_');
}

int priority(char op){
    if(op == '+' || op == '-') return 1;
    if(op == '*' || op == '/') return 2;
    if(op == '^') return 3;
    /* if op is not a valid operator, return 0 */
    return 0;
}

char *infixToPostfix(const char *s){
    Stack n = newStack();
    int length = strlen(s);
    char *postfix = (char *)calloc(length + 100, sizeof(char));
    //memset(postfix, 0, sizeof(char) * (length + 100));
    /* Dynamically allocate memory. */
    int index = 0, operationFlag = 1;
    postfix[0] = ' ';
    while(s[index] != '\0'){
        /* Handle unary operators '-','+' */
        while((s[index] == '-' || s[index] == ' ' || s[index] == '+') && operationFlag){
            /*Handle negative numbers or variables or functions. */
            if(s[index] == '-') postfix[strlen(postfix)] = '-';
            index++;
            if(isLetterOrDigit(s[index])) operationFlag = 0;
        }

        /* If the character is '(', push it onto the stack. */
        if(s[index] == '('){
            push(n, (void *)&s[index]);
            operationFlag = 1;
            //while(s [index] == ' ') ++index;
        }


        /* If the character is ')', pop elements from the stack
        and add them to the output string until '(' is encountered. */
        else if(s[index] == ')'){
            while(!isEmpty(n) && *(char *)top(n) != '('){
                postfix[strlen(postfix)] = *(char *)pop(n);
                postfix[strlen(postfix)] = ' ';
            }
            /* Pop the '(' from the stack. */
            if(!isEmpty(n) && *(char *)top(n) == '(') pop(n);
        }

```

```
60          /* If the character is a number/letter, add it to the output
            string. */
61          else if(isLetterOrDigit(s[index])){
62              /* flag to indicate if there is a '(' in the token */
63              while(isLetterOrDigit(s[index])||s[index] == '('){
64                  /* === BONUS== =
65                  If there is a '(', it means the token is a function
                    name.
66                  We need to delete the ' ' in the paren. */
67
68                  /*inside a function parenthesis*/
69                  if(s[index] == '('){
70                      int functionMode = 1, parenFlag = 0;
71                      while(functionMode){
72                          if(s[index] == '(') parenFlag ++;
73                          if(s[index] == ')') parenFlag --;
74                          /* If parenFlag != 0, we are inside a function
                            parenthesis */
75                          if(parenFlag == 0) functionMode = 0;
76                          /* Remove spaces from the postfix expression,
                            this will benefit
77                          buildTree() function, which will handle it as
                            a whole part */
78                          if(s[index] != ' ') postfix[strlen(postfix)] =
                            s[index];
79                          ++index;
80                          if(!functionMode) goto finish;
81                      }
82                  }
83
84                  /*outside a function parenthesis*/
85                  else{
86                      postfix[strlen(postfix)] = s[index];
87                      ++index;
88                  }
89                  operationFlag = 0;
90              }
91              finish:
92              index--;
93              postfix[strlen(postfix)] = ' ';
94          }
95
96          /* If the character is an operator(priority != 0), pop
            elements from the stack
97          and add them to the output string until a lower priority
            operator is encountered. */
98          else if(priority(s[index])){
99              while(!isEmpty(n) && (priority(*(char *)top(n)) >=
                priority(s[index]))){
100                 postfix[strlen(postfix)] = *(char *)pop(n);
101                 postfix[strlen(postfix)] = ' ';
```

```
102             }
103             push(n, (void *)&s[index]);
104             operationFlag = 1;
105             /* if '-' next to an operator, it's a unary minus */
106         }
107         ++index;
108     }
109     /* Pop any remaining operators from the stack
110     and add them to the output string. */
111     while(!isEmpty(n)){
112         postfix[strlen(postfix)] = *(char *)pop(n);
113         postfix[strlen(postfix)] = ' ';
114     }
115     postfix[strlen(postfix)] = '\0';
116 
117     return postfix;
118 }
119 
120 Tree buildTree(const char *s){
121     char *postfix = infixToPostfix((char *)s);
122     Stack n = newStack();
123     int index = 0;
124     while(postfix[index] != '\0'){
125 
126         if(isLetterOrDigit(postfix[index]) || (postfix[index] == '-'
            &&
127                                                (index == 0 ||
                                                   postfix[index - 1] == '
                                                   ') && postfix[index +
                                                   1] != ' ')){
128             Tree node = (Tree)calloc(1, sizeof(struct TreeNode));
129             /* Initialize the val array to zero. */
130             //memset(node-> val, 0, LENGTH * sizeof(char));
131             while(postfix[index] == '-') node->minusFlag ^= 1,
                index++;
132             /* ^= 1: XOR with 1, that is: if A = 0, A ^ 1 = 1; if A =
                1, A ^ 1 = 0 */
133             /* For more than one minus sign, if the number of minus
                signs is odd, the node is negative. */
134             int hasLetter = 0, isFunction = 0;
135             while(postfix[index] != ' '){
136                 if(isLetter(postfix[index])) hasLetter = 1;
137                 /* is Function. */
138                 if(postfix[index] == '(') isFunction = 1;
139                 //printf("==== ==== ==Info: function detected.== ====
                    ====\n");}
140                 node->val[strlen(node->val)] = postfix[index];
141                 ++index;
142                 if(strlen(node->val) >= LENGTH - 1){
143                     printf("==========Error: token length exceeds
                        limit %d.==========\n", LENGTH);
```

```
144                     exit(1);
145                 }
146             }
147             /* Check if val is an invalid name. */
148             /* That is: if it is not a number, it must starts with a
                letter or '_' */
149             if(strlen(node->val) >= 2 && !isLetter(node->val[0]) &&
                hasLetter){
150                 printf("==========Error: invalid token name
                    \"%s\".==========\n", node->val);
151                 exit(1);
152             }
153             index--;
154
155             /* To distinguish between variables and numbers and
                Functions. */
156             if(isFunction) node->op = '$';
157             else if(hasLetter) node->op = 'A';
158             else node->op = 0;
159             node->left = NULL;
160             node->right = NULL;
161
162
163             /*==== = BOUNS ==== =*/
164             if(node->op == '$'){
165                 //printf("==== ==== ==Info: function \"%s \"
                    detected.== ==== ====\n", node-> val);
166                 /* If it's a function, we need to build a subtree for
                    the function arguments. */
167                 /* Dynamically allocate memory. */
168                 char *temp = (char *)calloc(LENGTH + 100,
                    sizeof(char));
169                 char *temp2 = (char *)calloc(LENGTH + 100,
                    sizeof(char));
170                 /* Remove the function name and the parentheses.
171                 Build the subtree for the function arguments. */
172
173                 /* For double variable functions, we need to change
                    them to single
174                 variable functions to make the autograd easier to
                    implement. */
175                 /* log(A, B) = ln(B)_temp / ln(A)_temp2 */
176                 if(strncmp(node->val, "log(", 4) == 0){
177                     /* change "log(A, B)" to "ln(A)" */
178                     /* strcpy(offset = 1): "og(A, B)-> ln(A, B)"*/
179                     if(node->minusFlag) strcat(temp2, "-");
180                     strcat(temp, "ln(");
181                     strcat(temp, node->val + 4);
182
183                     int nodeIndex = 0, parenFlag = 0;
184                     for(int i = 0; temp[i] != '\0'; ++i){
```

```
185                     if(temp[i] == '(') parenFlag++;
186                     if(temp[i] == ')') parenFlag--;
187                     /* If parenFlag == 1 and temp [i] == ',', we
                        find the separator of the two variables. */
188                     if(parenFlag == 1 && temp[i] == ','){
189                         /*ln(A, B)-> ln(A)*/
190                         temp[i] = ')', temp[i + 1] = '\0';
191                         nodeIndex = i + 1;
192                         break;
193                     }
194                 }
195                 /* strcat: separate the two variables, temp:
                    ln(A), temp2: ln(B) */
196                 strcat(temp2, "ln(");
197                 strcat(temp2, node->val + nodeIndex + 1);
198                 //printf("temp: %s\n", temp);
199                 //printf("temp2: %s\n", temp2);
200 
201                 node->op = '/';
202                 node->minusFlag = 0;
203                 node->left = buildTree(temp2);
204                 node->right = buildTree(temp);
205             }
206 
207             /* For double variable functions, we need to change
                them to single
208             variable functions to make the autograd easier to
                implement. */
209             /* pow(A, B) = exp(B * ln(A)) */
210             else if(strncmp(node->val, "pow(", 4) == 0){
211                 /* change "pow(A, B)" to " exp(B*ln(A))" */
212                 /* temp: ln(A)*(B) */
213                 int parenFlag = 0;
214                 strcpy(temp, node->val + 1);
215                 for(int i = 0; temp[i] != '\0'; ++i){
216                     if(temp[i] == '(') parenFlag++;
217                     if(temp[i] == ')') parenFlag--;
218                     /* If parenFlag == 1 and temp [i] == ',', we
                        find the separator of the two variables. */
219                     if(parenFlag == 1 && temp[i] == ','){
220                         temp[0] = 'l', temp[1] = 'n';
221                         temp[i] = ')', temp[i + 1] = '*', temp[i +
                            2] = '(';
222                         strcpy(temp + i + 3, node->val + i + 2);
223                         break;
224                     }
225                 }
226                 strcpy(node->val, "exp");
227                 node->left = buildTree(temp);
228             }
```

```
229
230             /* single variable functions */
231             else if(strncmp(node->val, "cos(", 4) == 0){
232                 strcpy(temp, node->val + 3);
233                 strcpy(node->val, "cos");
234                 node->left = buildTree(temp);
235             }
236             else if(strncmp(node->val, "sin(", 4) == 0){
237                 strcpy(temp, node->val + 3);
238                 strcpy(node->val, "sin");
239                 node->left = buildTree(temp);
240             }
241             /* tan(A) = sin(A) / cos(A) */
242             /* We need to change "tan" to "sin" and "cos" in case
243             other functions like "cot", "sec" or "csc" are used.
                */
244             else if(strncmp(node->val, "tan(", 4) == 0){
245                 strcpy(temp, node->val + 3);
246                 node->op = '/';
247                 /* create left child: sin(...) */
248                 node->left = (Tree)calloc(1, sizeof(struct
                    TreeNode));
249                 node->left->op = '$';
250                 strcpy(node->left->val, "sin");
251                 node->left->left = buildTree(temp);
252
253                 /* create right child: cos(...) */
254                 node->right = (Tree)calloc(1, sizeof(struct
                    TreeNode));
255                 node->right->op = '$';
256                 strcpy(node->right->val, "cos");
257                 node->right->left = buildTree(temp);
258             }
259             else if(strncmp(node->val, "exp(", 4) == 0){
260                 strcpy(temp, node->val + 3);
261                 strcpy(node->val, "exp");
262                 node->left = buildTree(temp);
263             }
264             else if(strncmp(node->val, "ln(", 3) == 0){
265                 strcpy(temp, node->val + 2);
266                 strcpy(node->val, "ln");
267                 node->left = buildTree(temp);
268             }
269             else{
270                 printf("==========Error: invalid function name
                    \"%s\".==========\n", node->val);
271                 exit(1);
272             }
273         }
274         /* Above functions fall into 4 categories: sin, cos, ln,
            exp */
```

```
275
276             push(n, (void *)node);
277         }
278         /* If the character is an operator(priority != 0) */
279         else if(priority(postfix[index])){
280             Tree node = (Tree)calloc(1, sizeof(struct TreeNode));
281             node->op = postfix[index];
282             /* Pop right subtree first.
283             '-' and '/' are non-commutative */
284             node->right = (Tree)pop(n);
285             node->left = (Tree)pop(n);
286             if(node->op == '/' && node->right->op == 0 && strcmp(node-
                >right->val, "0") == 0){
287                 printf("==========Error: division by
                    zero.==========\n");
288                 exit(1);
289             }
290             push(n, (void *)node);
291             /* Take "5 - 3" as an example:
292                '-'
293             5(L) 3(R)
294             */
295         }
296         ++index;
297     }
298     return (Tree)pop(n);
299 }
300
301 void freeTree(Tree t){
302     /* Post-order traversal */
303     if(!t) return;
304     if(t->left) freeTree(t->left);
305     if(t->right) freeTree(t->right);
306     free(t);
307 }
308
309 void printTree(Tree t, int level){
310     /*=== test only, unused in main() function== =*/
311     /* graphical representation of the tree */
312     if(!t) return;
313     printTree(t->left, level + 1);
314     for (int i = 0; i < level; ++i) printf("    ");
315     /* Print the operator, but not functions.*/
316     if(t->op && t->op != '$' && t->op != 'A') {
317         if(t->minusFlag) printf("-");
318         printf("%c\n", t->op);
319     }
320     /* Print the value or function name. */
321     else{
322         if(t->minusFlag) printf("-");
323         printf("%s\n", t->val);
```

```
    }
    printTree(t->right, level + 1);
}

char *treeToInfix(Tree t, char *infix){
    /* Convert the syntax tree back to infix expression. Used for
    output. */
    if(!t||!infix) return NULL;
    /* If it's a leaf node, add the value to the infix string. */
    if(t->op == 0 || t->op == 'A'){
        if(t->minusFlag) strcat(infix, "-");
        strcat(infix, t->val);
        return infix;
    }
    else if(t->op == '$'){
        /* If it's a function node, add the function name and the
        parentheses to the infix string. */
        if(t->minusFlag) strcat(infix, "-");
        strcat(infix, t->val);
        strcat(infix, "(");
        treeToInfix(t->left, infix);
        strcat(infix, ")");
        return infix;
    }
    /* If it's an operator node, add parentheses and the operator to
    the infix string. */
    else{
        if(t->minusFlag) strcat(infix, "-(");
        char opstr[2] = { t->op, '\0' };
        //如果左子树运算符优先级低于或等于当前节点，则需要加括号
        //* and + 在同等优先级时无需加括号，因为满足交换律，- 和 / 在同等优先级
        时需要加括号，因为不满足交换律
        /*
        Parentheses needed if left subtree precedence <= current node.
        * & + need no parentheses at equal precedence (commutative);
        - & / require parentheses (non-commutative).
        */
        if((priority(t->op) >= priority(t->left->op)) && priority(t-
        >left->op) != 0
           && !(((t->op == '+' || t->op == '*') && priority(t->op) ==
           priority(t->left->op)) || t->left->op == '$')){
            strcat(infix, "(");
            treeToInfix(t->left, infix);
            strcat(infix, ")");
        }
        else treeToInfix(t->left, infix);
        /* inorder traversal */
        strcat(infix, " ");
        strcat(infix, opstr);
        strcat(infix, " ");

```

```
369         if((priority(t->op) >= priority(t->right->op)) && priority(t-
            >right->op) != 0
370            && !(((t->op == '+' || t->op == '*') && priority(t->op) ==
               priority(t->right->op)) || t->right->op == '$')){
371             strcat(infix, "(");
372             treeToInfix(t->right, infix);
373             strcat(infix, ")");
374         }
375         else treeToInfix(t->right, infix);
376         if(t->minusFlag) strcat(infix, ")");
377         return infix;
378     }
379 }
380 
381 void findVariables(Tree t, char vars[][NUMBER], int *count){
382     if(!t) return;
383     if(t->op == 'A'){
384         /* Check if the variable is already in the array. */
385         for(int i = 0; i < *count; ++i){
386             if(strcmp(vars[i], t->val) == 0) return;
387         }
388         /* If not, add it to the array. */
389         strcpy(vars[*count], t->val);
390         (*count)++;
391     }
392     findVariables(t->left, vars, count);
393     findVariables(t->right, vars, count);
394     return;
395 }
```

5.5 main.c

```
1  #include "BuildTree.h"
2  #include "Stack.h"
3  #include <stdlib.h>
4  #include <string.h>
5  #include <stdio.h>
6  #include <math.h>
7  
8  #define MAXLEN 5000
9  
10 Tree cloneTree(Tree src){
11     if(!src) return NULL;
12     Tree n = (Tree)calloc(1, sizeof(struct TreeNode));
13     /* preorder traversal */
14     n->op = src->op;
15     n->minusFlag = src->minusFlag;
16     strcpy(n->val, src->val);
17     n->left = cloneTree(src->left);
18     n->right = cloneTree(src->right);
19     return n;
```

```
}

int equalTree(Tree a, Tree b){
    if(!a && !b) return 1;
    if(!a || !b) return 0;
    if(a->op != b->op) return 0;
    if(strcmp(a->val, b->val) != 0) return 0;
    /* preorder traversal */
    return equalTree(a->left, b->left) && equalTree(a->right, b->right);
}

/* Build a new tree. */
Tree autoGrad(Tree t, char *var){
    if(!t) return NULL;
    Tree a = (Tree)calloc(1, sizeof(struct TreeNode));
    a->left = NULL;
    a->right = NULL;
    a->minusFlag = t->minusFlag;
    a->op = 0;
    /*
    (A + B)' = A' + B'
    (A - B)' = A' - B'
    (A * B)' = A' * B + A * B'
    (A / B)' = (A' * B - A * B') / (B * B)
    (A ^ B)' = A ^ B * (B' * ln(A) + B * A' / A)
    (f(A))' = f'(A) * A'
    */
   /* ==== = leaf node ==== = */
   /* if t is a number */
    if(t->op == 0){
        /* constant C' = 0 */
        strcpy(a->val, "0");
        a->op = 0;
        a->minusFlag = 0;
    }
    /* if t is a variable */
    else if(t->op == 'A'){
        /* variable A' = 1 if we are differentiating A,
           otherwise A' = 0 */
        if(strcmp(t->val, var) == 0){
            a->op = 0;
            strcpy(a->val, "1");
        }
        else{
            a->op = 0;
            strcpy(a->val, "0");
            a->minusFlag = 0;
        }
    }
    /* ==== = non-leaf node ==== = */
```

```
    else if(t->op == '+'){
        /* (A + B)' = A' + B' */
        a->op = '+';
        a->left = autoGrad(t->left, var);
        a->right = autoGrad(t->right, var);
    }
    else if(t->op == '-'){
        /* (A - B)' = A' - B' */
        a->op = '-';
        a->left = autoGrad(t->left, var);
        a->right = autoGrad(t->right, var);
    }
    else if(t->op == '*'){
        /* (A * B)' = A' * B + A * B' */
        a->op = '+';
        Tree leftMul = (Tree)calloc(1, sizeof(struct TreeNode));
        leftMul->op = '*';
        leftMul->left = autoGrad(t->left, var);
        leftMul->right = cloneTree(t->right);
        Tree rightMul = (Tree)calloc(1, sizeof(struct TreeNode));
        rightMul->op = '*';
        rightMul->left = cloneTree(t->left);
        rightMul->right = autoGrad(t->right, var);
        a->left = leftMul;
        a->right = rightMul;
    }
    else if(t->op == '/'){
        /* (A / B)' = (A' * B - A * B') / (B * B) */
        a->op = '/';
        /* numerator: "分子"; denominator: "分母" */
        Tree numerator = (Tree)calloc(1, sizeof(struct TreeNode));
        numerator->op = '-';
        Tree leftMul = (Tree)calloc(1, sizeof(struct TreeNode));
        leftMul->op = '*';
        leftMul->left = autoGrad(t->left, var);
        leftMul->right = cloneTree(t->right);
        Tree rightMul = (Tree)calloc(1, sizeof(struct TreeNode));
        rightMul->op = '*';
        rightMul->left = cloneTree(t->left);
        rightMul->right = autoGrad(t->right, var);
        numerator->left = leftMul;
        numerator->right = rightMul;
        Tree denominator = (Tree)calloc(1, sizeof(struct TreeNode));
        denominator->op = '*';
        denominator->left = cloneTree(t->right);
        denominator->right = cloneTree(t->right);
        a->left = numerator;
        a->right = denominator;
    }
    /* (A ^ B)' = A ^ B      *      (B' * ln(A) + B * A' / A) */
    else if(t->op == '^'){
```

```
121         a->op = '*';
122         /* A ^ B */
123         Tree exp = (Tree)calloc(1, sizeof(struct TreeNode));
124         exp->op = '^';
125         exp->left = cloneTree(t->left);
126         exp->right = cloneTree(t->right);
127 
128         /* ln(A) */
129         Tree logA = (Tree)calloc(1, sizeof(struct TreeNode));
130         logA->op = '$';
131         strcpy(logA->val, "ln");
132         logA->left = cloneTree(t->left);
133         /* A' */
134         Tree Agrad = autoGrad(t->left, var);
135         /* B' */
136         Tree Bgrad = autoGrad(t->right, var);
137 
138         /* B' * ln(A) */
139         Tree BmulLogA = (Tree)calloc(1, sizeof(struct TreeNode));
140         BmulLogA->op = '*';
141         BmulLogA->left = Bgrad;
142         BmulLogA->right = logA;
143         /* A' / A */
144         Tree AdivA = (Tree)calloc(1, sizeof(struct TreeNode));
145         AdivA->op = '/';
146         AdivA->left = Agrad;
147         AdivA->right = cloneTree(t->left);
148 
149         /* B * (A' / A) */
150         Tree BmulAdivA = (Tree)calloc(1, sizeof(struct TreeNode));
151         BmulAdivA->op = '*';
152         BmulAdivA->left = cloneTree(t->right);
153         BmulAdivA->right = AdivA;
154         Tree sum = (Tree)calloc(1, sizeof(struct TreeNode));
155         sum->op = '+';
156         sum->left = BmulLogA;
157         sum->right = BmulAdivA;
158 
159 
160         a->left = exp;
161         a->right = sum;
162     }
163     else if(t->op == '$'){
164         /* In buildTree() function, functions fall into 4 categories:
            sin, cos, ln, exp.
165            So we only need to handle these 4 cases. */
166 
167         /* ln(A)' = A' / A */
168             if(strcmp(t->val, "ln") == 0){
169             a->op = '/';
170             Tree Agrad = autoGrad(t->left, var);
```

```
171             a->left = Agrad;
172             a->right = cloneTree(t->left);
173         }
174 
175         /* sin(A)' = cos(A) * A' */
176         else if(strcmp(t->val, "sin") == 0){
177             a->op = '*';
178             Tree cos = (Tree)calloc(1, sizeof(struct TreeNode));
179             cos->op = '$';
180             strcpy(cos->val, "cos");
181             cos->left = cloneTree(t->left);
182             cos->right = NULL;
183             a->left = cos;
184             a->right = autoGrad(t->left, var);
185         }
186 
187         /* cos(A)' = -sin(A) * A' */
188         else if(strcmp(t->val, "cos") == 0){
189             a->op = '*';
190             Tree sin = (Tree)calloc(1, sizeof(struct TreeNode));
191             sin->op = '$';
192             strcpy(sin->val, "sin");
193             sin->left = cloneTree(t->left);
194             sin->right = NULL;
195             /* Use minusFlag to represent -sin(A), instead of 0-
                sin(A). */
196             sin->minusFlag = 1;
197             a->left = sin;
198             a->right = autoGrad(t->left, var);
199         }
200 
201         /* exp(A)' = exp(A) * A' */
202         else if(strcmp(t->val, "exp") == 0){
203             a->op = '*';
204             a->left = cloneTree(t);
205             Tree Agrad = autoGrad(t->left, var);
206             a->right = Agrad;
207         }
208     }
209     return a;
210 }
211 
212 /* Change directly on the original tree. */
213 Tree simplify(Tree t){
214     if(!t) return NULL;
215     t->left = simplify(t->left);
216     t->right = simplify(t->right);
217 
218     /* "+ - * /" on numbers */
```

```
219     if((t->op == '+' || t->op == '-' || t->op == '*' || t->op == '/'
        || t->op == '^') && t->left && t->left->op == 0 && t->right && t-
        >right->op == 0){
220         /* Both number, then directly compute it. */
221         int leftVal = atoi(t->left->val);
222         int rightVal = atoi(t->right->val);
223         if(t->left->minusFlag) leftVal = -leftVal;
224         if(t->right->minusFlag) rightVal = -rightVal;
225         int tmp = 0;
226         if(t->op == '+') tmp = leftVal + rightVal;
227         else if(t->op == '-') tmp = leftVal - rightVal;
228         else if(t->op == '*') tmp = leftVal * rightVal;
229         else if(t->op == '/') tmp = leftVal / rightVal;
230         /* division by zero is already detected & handled in the
            buildTree function */
231         else if(t->op == '^') tmp = (int)pow(leftVal, rightVal);
232         t->op = 0;
233         t->minusFlag = (tmp < 0);
234         if(tmp < 0) tmp = -tmp;
235         sprintf(t->val, "%d", tmp);
236         free(t->left);
237         free(t->right);
238         t->left = NULL;
239         t->right = NULL;
240     }
241     if(t->op == '+' && t->left && t->right && t->left->op == 0 &&
        strcmp(t->left->val, "0") == 0){
242         /* 0 + A = A */
243         Tree temp = t->right;
244         free(t);
245         t = temp;
246     }
247     if((t->op == '+' || t->op == '-') && t->right && t->left && t-
        >right->op == 0 && strcmp(t->right->val, "0") == 0){
248         /* A + 0 = A, A - 0 = A */
249         Tree temp = t->left;
250         free(t);
251         t = temp;
252     }
253     if((t->op == '*' && t->right && t->right->op == 0 && strcmp(t-
        >right->val, "0") == 0) ||
254             (t->op == '*' && t->left && t->left->op == 0 && strcmp(t-
                >left->val, "0") == 0) ||
255             (t->op == '/' && t->left && t->left->op == 0 && strcmp(t-
                >left->val, "0") == 0)){
256         /* A * 0 = 0, 0 * A = 0, 0 / A = 0 */
257         t->op = 0;
258         t->minusFlag = 0;
259         strcpy(t->val, "0");
260         freeTree(t->left);
261         freeTree(t->right);
```

```
262         t->left = NULL;
263         t->right = NULL;
264     }
265     if(t->op == '*' && t->right && t->right->op == 0 && !t->right->minusFlag && strcmp(t->right->val, "1") == 0){
266         /* A * 1 = A */
267         Tree temp = t->left;
268         free(t);
269         t = temp;
270     }
271     if(t->op == '*' && t->left && t->left->op == 0 && !t->left->minusFlag && strcmp(t->left->val, "1") == 0){
272         /* 1 * A = A */
273         Tree temp = t->right;
274         free(t);
275         t = temp;
276     }
277     if(t->op == '/' && t->right && t->right->op == 0 && !t->right->minusFlag && strcmp(t->right->val, "1") == 0){
278         /* A / 1 = A */
279         Tree temp = t->left;
280         free(t);
281         t = temp;
282     }
283     if(t->op == '^' && t->right && t->right->op == 0 && !t->right->minusFlag && strcmp(t->right->val, "1") == 0){
284         /* A ^ 1 = A */
285         Tree temp = t->left;
286         free(t);
287         t = temp;
288     }
289     if(t->op == '^' && t->right && t->right->op == 0 && strcmp(t->right->val, "0") == 0){
290         /* A ^ 0 = 1 */
291         t->op = 0;
292         strcpy(t->val, "1");
293         freeTree(t->left);
294         freeTree(t->right);
295         t->left = NULL;
296         t->right = NULL;
297     }
298     if(t->op == '/' && t->left && t->right && equalTree(t->left, t->right)){
299         /* A / A = 1*/
300         t->op = 0;
301         strcpy(t->val, "1");
302         freeTree(t->left);
303         freeTree(t->right);
304         t->left = NULL;
305         t->right = NULL;
306     }
```

```
307     if(t->op == '-' && t->left && t->left->op == 0 && strcmp(t->left-
        >val, "0") == 0){
308         /* 0 - A = -A */
309         freeTree(t->left);
310         t = t->right;
311         t->minusFlag = !t->minusFlag;
312     }
313
314
315     if(t->op == '-' && t->right->minusFlag){
316         /* Change "A - (-B)" to "A + B" */
317         t->op = '+';
318         t->right->minusFlag = 0;
319     }
320
321     if(t->op == '+' && t->right->minusFlag){
322         /* Change "A + (-B)" to "A - B" */
323         t->op = '-';
324         t->right->minusFlag = 0;
325     }
326
327     if(t->op == '*' && t->left && t->left->minusFlag && t->left->op ==
        0 && strcmp(t->left->val, "1") == 0){
328         /* Change "(-1) * A " to "-A " */
329         freeTree(t->left);
330         int originalMinusFlag = t->minusFlag;
331         t = t->right;
332         t->minusFlag = t->minusFlag ^ originalMinusFlag ^ 1;/* toggle
            the minusFlag */
333     }
334
335     if(t->op == '*' && t->right && t->right->minusFlag && t->right->op
        == 0 && strcmp(t->right->val, "1") == 0){
336         /* Change " A * (-1)" to "-A " */
337         freeTree(t->right);
338         int originalMinusFlag = t->minusFlag;
339         t = t->left;
340         t->minusFlag = t->minusFlag ^ originalMinusFlag ^ 1;/* toggle
            the minusFlag */
341     }
342
343     if(t->op == '*' && t->left->minusFlag && t->right->minusFlag){
344         /* Change "(-A) * (-B)" to " A * B" */
345         t->left->minusFlag = 0;
346         t->right->minusFlag = 0;
347     }
348     return t;
349 }
350
351 int test(){
352     /*
```

```
    //=== test only, unused in main() function == =
    char s [500] =" a+b^c*d ";
    char *autograd = (char *)malloc(500 * sizeof(char));
    memset(autograd, 0, strlen(autograd)*sizeof(char));
    char *simp_infix = (char *)malloc(500 * sizeof(char));
    memset(simp_infix, 0, strlen(simp_infix)*sizeof(char));


    printf("==== ==== ==Original Tree== ==== ====\n");
    Tree t = buildTree(s);
    printTree(t, 0);
    printf("==== ==== ==Autograd Tree== ==== ====\n");
    char var [] = "x";
    Tree grad = autoGrad(t, var);
    printTree(grad, 0);
    treeToInfix(grad, autograd, 0);
    printf("==== ==== ==Simplified Autograd Tree== ==== ====\n");
    Tree simp_tree = simplify(grad);
    printTree(simp_tree, 0);
    treeToInfix(simp_tree, simp_infix, 0);


    printf("Original expression:\n");
    printf("%s\n", s);
    printf("Autograd expression:\n");
    printf("%s\n", autograd);
    printf("Simplified autograd expression:\n");
    printf("%s\n", simp_infix);
    freeTree(t);
    //freeTree(grad);
    free(autograd);
    free(simp_infix);
    */
    return 0;

}

/*==== = Autograd for Algebraic Expressions ==== =*/
int main(){

    char s[MAXLEN]={};
    /* MAXLEN: maximum length of the input,
       NUMBER: maximum length of variables. */
    char variable[50][NUMBER] = {};
    int varCount = 0;

    printf("Please input an algebraic expression:\n");
    fgets(s, MAXLEN, stdin);

    Tree t = buildTree(s);/* build the expression tree */
```

```
403     findVariables(t, variable, &varCount);/* find all variables in the
        expression */
404     for(int i = 0; i < varCount; ++i){
405         Tree grad = autoGrad(t, variable[i]);/* autograd each variable
            respectively */
406 
407         //printTree(grad, 0);/* print the autograd tree for testing
            only, can be removed in the final version. */
408 
409         Tree simp = simplify(grad);/* simplify the autograd tree */
410         char *out = (char *)calloc(MAXLEN, sizeof(char));
411         //memset(out, 0, MAXLEN * sizeof(char));
412         treeToInfix(simp, out);/* convert the simplified autograd tree
            to infix notation */
413         printf("%s: %s\n", variable[i], out);
414         //printf("the location of out in memory: %p\n", (void *)out);
415         free(out);
416         freeTree(simp);
417     }
418 
419     freeTree(t);
420 
421     return 0;
422 }
```

Declaration

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