Autograd for Algebraic Expressions

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

Chapter 1: Introduction

Problem description

The application of automatic differentiation technology in frameworks such as torch and tensorflow has greatly facilitated people's implementation and training of deep learning algorithms based on backpropagation. The project is aimed to implement an automatic differentiation program for algebraic expressions.

Background of the algorithms

In this task, the input expression is stored in the data structure of a binary tree. Operations on the tree are performed to carry out differentiation, simplification, and ultimately output the result through a special in-order traversal. The main focus of the task includes three key steps:

1. Conversion of the input infix expression into an expression tree;
2. Performing differentiation operations on the expression tree;
3. Correcting, simplifying, and outputting the results.
4. For convenience, numerous repetitive operations in the `buildInfixExpression` function have been macro-defined to reduce code length.

It's worth mentioning that this task extensively utilizes recursion for differentiation, correcting and simplifying on the tree.

Chapter 2: Algorithm Specification

First, let's take a look at the definitions of several structures.

- struct Infix : Used to store different units (numbers, variables, functions, operators) in the input expression, while also recording their attributes. It's worth mentioning that 1 represents operators, 2 represents variables or numbers, 3 and 4 represent the sin and cos functions respectively, 5 represents the ln function, 6 represents the exp function, 7 represents the tan function, and 8 and 9 represent the pow and log functions, respectively.

```c
typedef struct Infix *InfixPoint;
struct Infix {
    char *data;    // Data element
    int category; // Determine the category of each Infix unit.
};
```

- TreeNode: The basic units for the tree structure.

```c
typedef struct TreeNode *TreePoint;
struct TreeNode {
    char *element;    // Data element
    TreePoint left; // Left child
    TreePoint right; // Right child
    int category; //Determine the category of tree nodes.
};
```

- struct Variables: Used to store variables in lexicographical order.

```c
typedef struct Variables {
    char name[MAXN];
} Variables;
```

Next, we will focus on analyzing the three main parts of the task.

- ### Build an expression tree.

  1. **First, we need to convert the infix expression to postfix notation.Here is the specific algorithm:**

     Initialize a stack and a postfix expression string.

     Process each character in the infix expression from left to right until the expression ends:

     - If the character is '(', push it onto the stack.
     - If the character is a digit, add it to the postfix expression string.
     - If the character is an operator, pop operators from the stack with a priority not lower than the current operator, add them to the postfix expression, and then push the current operator onto the stack. Note that when '(' is on the top of the stack, it has the lowest priority.
     - If the character is ')', pop elements from the stack and add them to the postfix expression until '(' is popped.

     If the expression ends but the stack is not empty, pop all elements from the stack and add them to the postfix expression.

2. Next, we build a tree based on the postfix expression. Here is the pseudo-code:

```
function buildExpressionTree(postfixExpression):
    stack = empty stack

    for each token in postfixExpression:
        if token is operand or math function:
            create a new node with the operand and push it onto the
stack
        else if token is operator:
            create a new node with the operator
            set the right child of the node to pop from the stack
            set the left child of the node to pop from the stack
            push the new node onto the stack

    return the root of the stack
```

- **Perform autograd on the expression tree.**

At this point, we have obtained an expression tree where leaf nodes represent numbers, variables, or function expressions (in this case, the function name is omitted, and for example, the storage result of $\cos(a)$ is a, but its category is 3). Next, we will obtain a new tree based on differentiation formulas and recursive operations:

Here are all the differentiation formulas used in this task:

$f+g$	$f-g$	$f*g$	f/g	f^g	$sin(f)$
$f'+g'$	$f'-g'$	$f'*g+g'*f$	$f'/g-g'*f/g^2$	$g*f^{g-1}+f^g*ln(f)*g'$	$f'*cos(f)$
$cos(f)$	$tan(f)$	$ln(f)$	$exp(f)$	$pow(f,g)$	$log(f,g)$
$-f'*sin(f)$	$f'/cos(f)^2$	f'/f	$f'*exp(f)$	$(f^g)'$	$(ln(g)/ln(f))$

Here is a brief overview of the derivativeTree-building process:

```
Function: automaticDifferentiation(root, variable)
Input: root (TreePoint), variable (char *)
Output: derivativeRoot (TreePoint)

1. If root is not NULL:
   a. Create a new node derivativeRoot.
   b. If root's category is 1:
      i. If root's element is "+" or "-":
         - Set derivativeRoot's element to root's element.
         - Set derivativeRoot's left child to
automaticDifferentiation(root's left, variable).
         - Set derivativeRoot's right child to
automaticDifferentiation(root's right, variable).
```

```
      ii. If root's element is "*":
         -Based on the differentiation formula,perform a recursive
operation similar to the one before.
      iii. If root's element is "/":
         -Based on the differentiation formula,perform a recursive
operation similar to the one before.
      iv. If root's element is "^":
         -Based on the differentiation formula,perform a recursive
operation similar to the one before.
   c. If root's category is 2:
      i. If root's element is equal to the variable:
         - Set derivativeRoot to a new node with element "1" and category
2.
      ii. Else:
         - Set derivativeRoot to a new node with element "0" and category
2.
   d. If root's category is 3 (cos()):
      i. Set derivativeRoot's element to "*".
      ii. Set derivativeRoot's left child to a new node with element
root's element.
      iii. Set derivativeRoot's left child's left child to a new node
with element "-sin(".
      iv. Set derivativeRoot's left child's right child to a new node
with element ")".
      v. Build Infix expression from root's element and convert it to
Postfix.
      vi. Build Expression Tree from the Postfix expression.
      vii. Set derivativeRoot's right child to
automaticDifferentiation(resulting tree, variable).
   e. If root's category is 4 (sin()):
      -Based on the differentiation formula,perform a recursive operation
similar to the one before.
   f. If root's category is 5 (ln()):
     -Based on the differentiation formula,perform a recursive operation
similar to the one before.
   g. If root's category is 6 (exp()):
     -Based on the differentiation formula,perform a recursive operation
similar to the one before.
   h. If root's category is 7 (tan()):
     -Based on the differentiation formula,perform a recursive operation
similar to the one before.
   i. If root's category is 8 (pow):
      -Based on the differentiation formula,perform a recursive operation
similar to the one before.
   j. If root's category is 9 (log):
     -Based on the differentiation formula,perform a recursive operation
similar to the one before.
   k. Return derivativeRoot.
2. Return NULL.
```

- **Simplify the expression and output the result**

  After successfully performing the differentiation operation, we need to perform a correction operation on the resulting derivative tree. Why? Because our function storage omits the function names. For example, when calculating $a * cos(a)$, since $cos(a)$ is stored in the tree as a, the output is $1 * a$ + $-sin(a) * a$, which is obviously incorrect. Therefore, we need to correct tree nodes with the category of functions.

  Here is the code:

```
TreePoint CorrectedTree(TreePoint root) {
    // Check if the current node is a leaf node
    if(root->left == NULL && root->right == NULL) {
        // If it's a leaf node, check its category for correction
        if(root->category == 3) {
            // Category 3: Trigonometric function (cos)
            root->left = createDerivativeNode("cos(", -1);
            root->right = createDerivativeNode(")", -1);
        } else if(root->category == 4) {
            // Category 4: Trigonometric function (sin)
            root->left = createDerivativeNode("sin(", -1);
            root->right = createDerivativeNode(")", -1);
        } else if(root->category == 5) {
            // Category 5: Natural logarithm (ln)
            root->left = createDerivativeNode("ln(", -1);
            root->right = createDerivativeNode(")", -1);
        } else if(root->category == 6) {
            // Category 6: Exponential function (exp)
            root->left = createDerivativeNode("exp(", -1);
            root->right = createDerivativeNode(")", -1);
        } else if(root->category == 7) {
            // Category 7: Tangent function (tan)
            root->left = createDerivativeNode("tan(", -1);
            root->right = createDerivativeNode(")", -1);
        } else if(root->category == 8) {
            // Category 8: Power function (pow)
            root->left = createDerivativeNode("pow(", -1);
            root->right = createDerivativeNode(")", -1);
        } else if(root->category == 9) {
            // Category 9: Logarithmic function (log)
            root->left = createDerivativeNode("log(", -1);
            root->right = createDerivativeNode(")", -1);
        }
    } else {
        // If the current node is not a leaf, recursively correct its left
and right children
        root->left = CorrectedTree(root->left);
        root->right = CorrectedTree(root->right);
    }
    // Return the corrected tree
    return root;
}
```

Now, we can begin our simplify implimentations. The following lists are all the simplification operations:

- 0 + x = x; 0 - x = -x;
- x - 0 = 0; x - (-y) = x + y;
- x * 0 = 0; x * 1 = x; x ^ 1 = x;
- x * (1/y) = x / y; 0 / x = 0; x / 1 = x;
- pure number1 (operator) pure number2 = pure number3;

  Here is the pseudo-code:

```
Function Simplify(root):
    If root is not null:
        Simplify the left and right subtrees

        // Simplification for addition
        If root represents addition:
            If both operands are 0, set root to 0
            If one operand is 0, replace root with the non-zero operand

        // Simplification for subtraction
        Else If root represents subtraction:
            If both operands are 0, set root to 0
            If one operand is 0, replace root with the non-zero operand
            If the right operand is "-", simplify the expression

        // Simplification for multiplication
        Else If root represents multiplication:
            If either operand is 0, set root to 0
            If one operand is 1, replace root with the other operand
            If one operand is "/", simplify the expression

        // Simplification for division
        Else If root represents division:
            If the numerator is 0, set root to 0
            If the denominator is 1, replace root with the numerator
        // Simplification for other cases
            ......
    Return root
```

Finally, we print the obtained differential tree using a special output function.

The function `printTreeInOrder` is designed to print a binary tree in infix order with parentheses to ensure correct precedence of operators. Here's an overview of its functionality:

1. **Parameters:** The function takes a pointer to the root of the binary tree as its parameter (`TreePoint root`).
2. **Printing Infix Expression:** The function recursively traverses the tree in infix order (left subtree, current node, right subtree).
3. **Handling Operators and Parentheses:** It takes into account the priority of operators to determine whether parentheses are needed for correct precedence. The conditions check if the current node has both left and right children, and based on the priority of operators and the current context, it decides when to add opening and closing parentheses.
4. **Flags (`flag` , `sign1` , `sign2`):**
   - `flag` : Used to track whether the left subtree has been processed.
   - `sign1` and `sign2` : Used to count the number of opening and closing parentheses to ensure proper balancing.
5. **Output:** The function prints the elements of the tree nodes (operators or operands) along with parentheses as needed to maintain correct operator precedence.

   Here is the code:

```
// Function to print the tree in infix order with parentheses for correct
precedence
void printTreeInOrder(TreePoint root) {
    if (root) {
        // If the current node is not a leaf node, add an opening
parenthesis
        if (root->left && root->right && flag == 1 && getPriority(*(root-
>element)) > getPriority(*(root->left->element))) {
            printf("(");
            sign1++;
        } else if (root->left && root->right && flag == 0) {
            flag = 1;
        }

        // Recursively print the left subtree
        printTreeInOrder(root->left);

        // Print the current node's element
        printf("%s", root->element);

        // Recursively print the right subtree
        if (root->left && root->right && flag == 1 && getPriority(*(root-
>element)) >= getPriority(*(root->right->element)) && *(root->right-
>element) != ')') {
            printf("(");
            sign3++;
            printTreeInOrder(root->right);
            if(sign4 < sign3){
                printf(")");  // Added closing parenthesis here
                sign4++;
            }

        } else {
            printTreeInOrder(root->right);
        }

        // If the current node is not a leaf node, add a closing parenthesis
        if (root->left && root->right && sign2 < sign1) {
            printf(")");
            sign2++;
        }
    }
}
```

Some other key points.

For the input variables, I use sort to print the differentiation results in lexicographical order. To save time, I directly utilized the qsort function from the library.

Chapter 3: Testing Results

The following are test cases and outputs. I promise that all results are obtained from running my program:

- input $a + b^c * d$

  output

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

- input $a * 10 * b + 2^a / a$

  output

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

- input $xx^2 / xy * xy + a^a$

  output

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

- input $x * ln(y)$

  output

```
x : ln(y)
y : x/y
```

- input $x * ln(x * y) + y * cos(x) + y * sin(2 * x)$

  output

```
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)
```

- input $log(a, b)/log(c, a)$

  output

```
a : (-ln(b))/a/ln(a)^2/log(c,a)-1/a/ln(c)*log(a,b)/log(c,a)^2
b : 1/b/ln(a)/log(c,a)
c : -(-ln(a))/c/ln(c)^2*log(a,b)/log(c,a)^2
```

- input $tan(2 * x)$

  output

```
2/cos(2*x)^2
```

- input $pow(x, x) + pow(y, y)$

  output

```
x : x^x*ln(x)+x*x^(x-1)
y : y^y*ln(y)+y*y^(y-1)
```

This task has completed the differentiation of variables and functions within the specified scope, and it accommodates symbols for both numeric and alphanumeric input. Additionally, efforts have been made to simplify the output as much as possible.

Chapter 4: Analysis and Comments

Time Complexities

Analyzing the time complexity of the provided C code involves examining the main operations and loops within the program. Here are some key points to consider:

1. Input Reading:

- The while loop reading the input runs in O(n) time, where 'n' is the length of the input string.

```
while ((input[i++] = getchar()) != '\n');
```

2. Sorting Variables:

- The `qsort` function for sorting variables has a time complexity of O(n log n), where 'n' is the number of variables.

```
qsort(varArray, count, sizeof(Variables), compare);
```

3. Build Infix Expression:

- The `buildInfixExpression` function scans the input and constructs the infix expression, which has a time complexity of O(n).

```
buildInfixExpression(input, Infix);
```

4. Infix to Postfix Conversion:

- The `InfixToPostfix` function converts the infix expression to postfix. The time complexity is O(n).

```
InfixToPostfix(Postfix, Infix);
```

5. Build Expression Tree:

- The `buildExpressionTree` function constructs the expression tree from the postfix expression. The time complexity is O(n).

```
buildExpressionTree(Postfix, length);
```

6. Automatic Differentiation:

- The `automaticDifferentiation` function performs automatic differentiation on the expression tree. The process of building and differentiating the expression tree involves visiting each node only once, so the time complexity for that part of the code would be O(n).

```
automaticDifferentiation(root, varArray[i].name);
```

7. Corrected Tree and Simplify:

- The `CorrectedTree` and `Simplify` functions both traverse the expression tree and perform operations. Their time complexity would be O(n).

```
CorrectedTree(derivativeRoot);
Simplify(CorrectedRoot);
```

Overall, the time complexity of the entire program is O(n log n).

Space Complexities

In this process, the storage of the tree plays a dominant role in space complexity calculation, with its space complexity being O(n), as each tree node precisely corresponds to an arithmetic unit. In conclusion, the overall program's space complexity is **O(n)**.

Comment

This code implements the differential functionality of various elementary functions, and the expression results are accurate. However, due to time and capability constraints, the final expression cannot be completely simplified, and only partial simplification is possible. In this regard, there is indeed room for further improvement.