1. Introduction

We are faced with processing a mathematical expression S in **infix expression** and calculating its gradient with respect to each variable. The results should be output in **lexicographical order**, with the gradient represented by literal constants and other variables.

Automatic gradient calculation plays an important role in **Machine Learning**. Famous packages like `PyTorch` has functions such as `.backward()` to compute gradients during **backpropagation**, optimizing the *weights and biases* of the model. This project provides a chance to implement `autograd` from scratch, which is undoubtedly interesting.

2. Algorithm Specification

Overall Insight

We can handle this problem in a **Step-by-step operation**.
- First, we should **tokenize** the expression, separating **variables**, **literal values** and **operators**.
- After the tokenization, I constructed a **tokenList** to store the different elements with their types explicitly labeled.
- And then, in `constructExpressionTree()`, I construct a **Expression Tree** with stacks to handle the problem of precedence of different calculations.
- What's next, we need to use `collectVariables()`**collect all the variables** and use `derive()` to **calculate the derivatives** of every variable from the root of the expression tree.
- Last, We use `calculateGrad()` to traverse all the variable and **output the derivatives** in the lexicographical order.

Specifications for structs

struct Node

Description

- This struct is used to store one node in the expression tree, with its type and corresponding value.

- Note that **only one** in operator, number and variable need to be stored, according to the type.

Pseudo-code

```
typedef struct Node {
  int type;
  char operator;
  int number;
  char variable[N];
  struct Node *Left, *Right;
  struct Node *parent;
} Node;
```

struct TokenList

Description

- This struct is used to **store tokens**, after they are extracted from the input infix expression. And then, they will be sent to **construct the expression tree.**

- **types** are the corresponding types of the token with the same index.

Pseudo-code

```
typedef struct TokenList {
  char tokens[N][N];
  char types[N];
  int cnt;
}
```

Specifications for functions

tokenize()

Description:

- This function is used to **separate the expression into fractions** with types labeled, so that we can do further process to variables, literal constants and operators.

Pseudo-code

```
void tokenize(char * expression, TokenList * tokenListPtr) {
  int i = 0;
  while (i < expressionLength) {
    if (isspace(expression[i])) {
      continue;
    } else if (isdigit(expression[i])) {
      while loop until !isdigit(expression[i]) {
        store each bit of the number in the tokenlist;
      }
```

```
      flag the type;
    } else if (isOperator(expression[i])) {
      store the operator;
      flag the type;
    } else if (isalpha(expression[i]) || expression[i] == '_') {
      while loop until !(isalnum(expression[i]) || expression[i] == '_') {
        store each bit of the variable name in the tokenlist;
      }
      flag the type;
    } else {
      continue;
      /*invalid input*/
    }
  }
}
```

createExpressionTree()

Description

- `createExpressionTree()` is used to construct a binary tree storing all the operators and operands with data from the tokenList. We maintain **two stacks**, **nodeStack** is used to store operands and one is used to store operators. And then we maintain **opStacksss** according to the precedence of operators.

Pseudo-code

```
Node * createExpressionTree(TokenList * tokenListPtr) {
  intialize tokenListLen, nodeStack, opStack;
  for (i < tokenListLen) {
    if (tokenListPtr->type == TOKEN_IS_NUM) {
      createNode(num_type);
      push in nodeStack;
    } else if (tokenListPtr->type == TOKEN_IS_VAR) {
      createNode(var_type);
      push in nodeStack;
    } else if (tokenListPtr->type == TOKEN_IS_OPERATOR) {
      get operator;
      if (operator == '(') {
        createNode(op_type);
        push in opstack;
      } else if (operator == ')') {
        while (opTop >= 0 && opStack[opTop]) {
          pop opNode, rightOperand, leftOperand;
          setChildren(opNode, leftOperand, rightOperand);
          push into nodeStack;
```

```
      }
      if (top >= 0) {
        pop all the remaining operators;
      }
    } else {
      while (opTop >= 0 && getPrecedence(topOperator) >= getPrecedence(currentOperator)) {
        pop opNode, left, right;
        setChildren(opNode, left, right);
        push into nodeStack;
      }
      Node * newOpNode = createNode(operator_type);
      push into opStack;
    }
  }
}
while (opTop >= 0) {
  pop opNode, left, right;
  setChildren(opNode, left, right);
  push into nodeStack
}
return nodeTop;
}
```

getNodeExpr

Description

- This function is used to generate the corresponding string expression of a certain node. If the node is a **operand node**, then we output the string form of the operand. If the node is an **operator node**, then we output the expression string, combining the left operand, operator and the right operand. Note that the operand here came from **recursive call of getNodeExpr()**

Pseudo-code

```
char * getNodeExpr(Node * node) {
  if (node->type == TOKEN_IS_VAR) {
    return strdup(node->variable);
  } else if (node->type == TOKEN_IS_NUM) {
    return transferToString(node->number);
  } else if (node->type == TOKEN_IS_OPERATOR) {
    get left, right, op;
    return (string)("(%s %c %s)", left, right, op);
  } else return '0';
}
```

collectVariables

Description

- This function is used to **collect all the existing variables** in the expression. If the variable we are looking for exists, then the flag is true. If it is not found, then we add the variable from the current node into our variable list. After this, we **recursively** call collectVariables to check the variables from the left subtree and right subtree.

Pseudo-code

```
void collectVariables(Node * node, char ** vars, int * count) {
  if (node->type == TOKEN_IS_VAR) {
    bool exists = false;
    for (int i = 0; i < *count; i ++) {
      if (strcmp(vars[i], node->variable) == 0) {
        exists = true;
      }
    }
    if (!exists) {
      vars[*count] = strdup(node->variable);
      (*count) ++;s
    }
  }
  collectVariables(node->Left, vars, count);
  collectVariables(node->Right, vars, count);
}
```

derive

Description

- This function is used to **calculate the derivative** from the current node for variable var. For the simplest case for variables and numbers, we can directly get the gradient of 0 or 1. If the type is an operator, then we can calculate the derivative with the **expression and derivative** of left and right. Note that we shall apply a series of **simplification rules** for 0s and 1s. We can use `getNodeExpr()` to get the expression for a certain node and use `formatExpr()` to get the expression.

Pseudo-code

```
char * derive(Node * node, char * var) {
  if (node->type == TOKEN_IS_VAR) {
    if (strcmp(node->variable, var) == 0) {
      return "1";
    } else {
```

```
      return "0";
    }
  } else if (node->type == TOKEN_IS_NUM) {
    return "0";
  } else if (node->type == TOKEN_IS_OPERATOR) {
    get operator, leftDeriv, rightDeriv, leftExpr, rightExpr;
    initialize result;
    switch(operator) {
      case '+':
        result = leftDeriv + rightDeriv;
        /*note that we need to process special cases here*/
        break;
      case '-':
        result = leftDeriv - rightDeriv;
        break;
      case '*':
        result = leftDeriv * rightExpr + rightExpr * rightDeriv;
        /*note that there are a lot of special cases here*/
        break;
      case '/':
        result = (leftDeriv * rightExpr - leftExpr * rightDeriv)/(rightExpr ^ 2);
        break;
      case '^':
        result = getNodeExpr(node) * (rightDeriv * ln(leftExpr)+(rightExpr*leftDeriv/leftExp
        break;
    }
  }
}
```

calculateGrad

Description

- Collect all the variables first and then sort the variables in lexicographical order. Then we traverse through all the variables from the root of the entire expression tree, and output the derivative of the variable.

Pseudo-code

```
void calculateGrad(Node * root) {
  initialize varCount, variables;
  collectVariables(root, variables, &varCount);
  qsort(variables, varCount, sizeof(char *), compareStrings);
  /*compareStrings() is used to implement the lexicographical sort*/
  for (int i = 0; i < varCount; i ++) {
    char * derivExpr = derive(root, variables[i]);
    printf(variables[i], derivExpr);
```

The running time of *Selection Sort*

Figure 1: The running time of *Selection Sort*

```
    free(derivExpr);
  }
}
```

3. Testing Results

Table of test cases. Each test case usually consists of its purpose, the expected result, the program's actual behavior, the possible cause of a bug if your program does not function as expected, and the current status (*pass*, or *corrected*, or *pending*).

Table 1 shows some typical test cases for verifying the *Selection Sort* implementation and capturing potential bugs.

Table 1: Test cases for the *Selection Sort* implementation.

Test Cases	Design Purpose	Result	Status
[3]	Minimum array with a single element	[3]	*pass*
[1,2,4,5,9]	Array in ascending order	[1,2,4,5,9]	*pass*
[9,5,4,2,1]	Array in descending order	[1,2,4,5,9]	*pass*
[5,-9,2,-1,4]	Array in random order with negative values	[-9,-1,2,4,5]	*pass*
[1,1,1,1,1]	Array with repeated values	[1,1,1,1,1]	*pass*

Figures 1 shows the running time of the *Selection Sort* implementation. We observe a quadratic-like curve from the figure, which implies an $O(n^2)$ algorithm (see analysis in the next section).

4. Analysis and Comments

Analysis of the time and space complexities of the algorithms. Comments on further possible improvements.

For the runtime of Selection Sort, there are two subtasks inside the *for*-loop. Specifically,

- finding the smallest integer between `list[i]` to `list[n-1]` takes $n - i$ time units;

- interchaning `list[i]` and `list[min]` takes a constant time c.

This is independent of any particular input. Therefore, both the average and worst time complexity can be computed as:

$$T(n) = \sum_{i=0}^{n-1}(n - i + c) = \Theta(n^2).$$

There is still considerable room for improvement. For example, we could adopt the divide-and-conquer strategy to achieve an $O(n \log n)$ algorithm.

For the space requirement, since we merely need an array to store the n integers, the space complexity is $\Theta(n)$ and should be optimal.

Appendix: Source Code (in C)

Please make sure that your code is sufficiently commented. Otherwise, it will *not* be evaluated.

Sample Code:

```
/*
 * Find the smallest element in a range of an array
 * ------------------------------------------------
 *
 *   arr: an array of integers
 *   begin: the index to start the search
 *   end: the index to stop the search (non-inclusive)
 *
 *   returns: the index of the smallest element in the range
 */
int find_min(int arr[], int begin, int end) {
  int min_idx = begin;
  int min_val = arr[min_idx];

  for (int idx = min_idx+1; idx < end; ++idx) {
    if (arr[idx] < min_val) {
      min_idx = idx;
      min_val = arr[min_idx];
    }
  }

  return min_idx;
}

/*
 * Swap the values of two integers
```

```c
 * -------------------------------
 *
 *   a: pointer to the 1st integer
 *   b: pointer to the 2nd integer
 *
 *   returns: none
 */
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

/*
 * Rearrange an array of integers into increasing order in-place
 * -------------------------------------------------------------
 *
 *   arr: an unsorted array of integers
 *   n: number of elements in the array
 *
 *   returns: none
 */
void sort(int arr[], int n) {
  for (int i = 0; i < n; ++i) {
    int min_idx = find_min(arr, i, n);
    /*after the swap, arr[i] is the i-th smallest integer*/
    swap(&arr[i], &arr[min_idx]);
  }
}
```

Note: Feel free to use your own style of code or comments. Just make sure to be consistent with yourself.

Declaration

I hereby declare that all the work done in this project is of my independent effort.