EXPRESSION FRAMEWORK

PRODUCED BY: ADAM OSUSKY

adam.osusky02@gmail.com

Table of Contents

1

SECTION

Introduction

The purpose of this text is to document a C++ implementation of an expression evaluation and manipulation library. It allows users to create mathematical expressions using variables and perform various operations on these expressions. The library supports derivations, simplification and factorization of expressions.

This text will have two parts: a user and a code documentation. The user part will describe how to use this library. The code documentation will describe how the library is implemented.

SECTION

2 User manual

2.1 Setting up the library

To use the provided header library in your project, follow these steps to install it:

1. Download the `ExpressionNode.hpp` and `Expression.hpp` files that contain the header library code. Make sure to place them both in the same folder in a convenient location within your project directory.

2. In your project's source code files where you want to use the library, include the header files:

   ```cpp
   #include "PATH_IN_YOUR_PROJECT/Expression.hpp"
   ```

3. Compile your project. Once the header files are included and the project is compiled and linked successfully, you can start using the library in your code. Refer to the library's documentation on how to use its classes and functions effectively.

2.2 Usage

Create an instance of the `Expression` class specifying the type of the expression (e.g., `double` or `long double`):

```cpp
Expression<double> expression;
```

Create variables using the `create_variable` method of the `Expression` class. This method takes a value and a name for the variable. It returns a reference to the created variable.

```
auto & x = expr.create_variable(0.69, "x");
auto & y = expr.create_variable(-4.20, "y");
```

Define the desired mathematical expression using the provided operators: +, -, *, and /.

```
auto r = x/y + x * 0.5;
expression.set_expr(std::move(r));
```

The variable `r` is a unique pointer to a root of the computation tree. We want to move the root to the `Expression` class so we can use its helper methods for further manipulations. Equivalently, we can do it in one line like this:

```
expression.set_expr(x/y + x * 0.5);
```

Evaluate the expression using the `evaluate` method.

```
double value = expression.evaluate();
std::cout << value << std::endl; // -> 0.180714
```

You can get derivations of your variables with `differentiate` method.

```
expression.differentiate();
std::cout << x.derivative << std::endl; // -> 0.261905
std::cout << y.derivative << std::endl; // -> -0.0391156
```

Use the `to_string` method to obtain a string representation of the expression.

```
std::string expressionStr = expression.to_string();
std::cout << expressionStr << std::endl; // -> (x * (1/y) + x * 0.5)
```

Use the `normalize` method to simplify and factorize the expression.

```
expression.normalize();
std::cout << expression.to_string() << std::endl;
// -> ((1/y) + 0.5) * x
```

If you only need to factorize or simplify your expression, you can utilize the `factorize` or `simplify` method, respectively.
It is possible to access variables directly from the expression class.

```
    auto & x = expr["x"];
```

2.3 Demo showcase

Here you can see an example code for finding a minimum of a function with a gradient descent.

```
using T = long double;

auto expr = Expression<T>();
auto & x = expr.create_variable(0.69, "x");
auto & y = expr.create_variable(-4.20, "y");

expr.set_expr(x*x + y*y);

for (std::size_t i = 0; i < 10000; ++i) {
    expr.differentiate();
    x.value -= 0.01 * x.derivative;
    y.value -= 0.01 * y.derivative;
}
std::cout << "Optimization of " << expr.to_string() << std::endl;
std::cout << "x : " << x.value << std::endl;
std::cout << "y : " << y.value << std::endl;
std::cout << std::endl;
```

SECTION

3 System documentation

3.1 Code structure

Source files:

- `src/`
  - `src/ExpressionNode.hpp` - data tree nodes for computation graph, and operator overloading for creation of a computation tree
  - `src/Expression.hpp` - class for manipulation of expressions
- `tests/` - testing
  - `tests/creation_eval_test.cpp`
  - `tests/differentiation_test.cpp`
  - `tests/simplify_test.cpp`
  - `tests/complex_test.cpp`
- `demo.cpp` - showcasing of an example usage
- `CMakeLists.txt` - cmake for compiling tests or demo

3.2 Expression tree

`ExpressionNode`: This is the base class for computation tree nodes. It represents a node in the expression tree and contains properties and methods common to all node types. It is an abstract class with pure virtual methods for evaluation, differentiation, and string representation.

Derived Node Classes: The code defines several derived classes from ExpressionNode that represent specific types of nodes in the expression tree, such as `Number` (for numeric values of variables), `ConstantNode` (for constant values), `AddNode` (for addition operations), `MultNode` (for multiplication operations), and `DenominatorNode` (for denominator operations). Also in the future versions new type of nodes will be created.

Overloaded Operators: The code overloads several operators (+, *, -, and /) to enable convenient creation of expression nodes using operator syntax.

Differentiation of composite functions are computed by few simple rules:

- $(f \pm g)\prime = f\prime \pm g\prime$
- $(f \cdot g)\prime = f\prime \cdot g + f \cdot g\prime$
- $(\frac{1}{f})\prime = \frac{-f\prime}{f^2}$
- $(f(g))\prime = f\prime(g) \cdot g\prime$

3.3 `class Expression`

This class represents a mathematical expression and provides methods for creating and manipulating expressions. It uses the expression tree structure defined by the `ExpressionNode` classes. The class maintains a map of `Var` structs that are representing variables and a root pointer to the expression tree. It includes methods for differentiation, simplification, normalization, and evaluation of expressions.

For possibility of using one variable in multiple places of an expression, for every occurrence we create a `Number` node class, that has reference to a `Var` struct in the map.

Currently, only basic factorization is implemented. It checks if a node for addition has children with a variable of the same name, and simplifies the expression accordingly:

$$a \cdot x + b \cdot x = (a + b) \cdot x$$

Simplification involves checking for multiplication by 1 or division by 1, as well as multiplication by 0 and addition of 0.