Mathematical Programming Language

Andrea Miranda, Gerardo Rosetti

Set up

In here will be found the information related to our specific language about mathematics made with C++.

About

This is a specific language oriented to solve basic mathematics problems using numerical methods, this by simplifying and handling access to mathematical algorithms such as calculating the determinant of a matrix, solving integrals using the simpson method, and much more. This language was made using the C++ language as the base. Everything is based on numerical analysis algorithms.

For further information about numerical analysis, consult the book *“Métodos Numéricos Aplicados Con Software. Autor: Sholchlro Nakamura”.*

Languages Semantics

Data Types:

- Pair
- Vector
- Matrix
- Number
- Variable
- Impossible

These Data Types, that are Expression as well, are used to make complex expressions, and keep in mind that they are the possible solutions for the methods applied. Note: the lenguaje currently does not support complex numbers.

Expressions

We have the following expressions based on the following interface

```
class Expression
{
public:
   virtual std::shared_ptr<Expression> eval(Environment&) const =
0;
   virtual std::string toString() const noexcept = 0;
   virtual ~Expression() {}
};
```

The eval method will be to evaluate the given expression, which will behave according to it, and will return the result of the operation. This receives an ***Environment***, that is a compound of letters and associated values (examples will be shown).

- Expression Value: From the value expression we create the expressions associated with the data types mentioned previously.

- Expression UnaryExpression: To compose other expressions.
- Expression BinaryExpression: To compose other expressions.

- Expression Addition: With the purpose of adding two expressions, such as two numbers or two matrices.

- Expression Subtraction: With the purpose of subtracting two expressions, such as two numbers or two matrices.

- Expression Multiplication: With the purpose of multiplying two expressions, such as two numbers or two matrices.

- Expression Division: With the purpose of dividing two expressions, such as two numbers or two matrices.

- Expression Power: With the purpose of elevating one expression to another.

- Expression NaturalLogarithm, Logarithm, Sine, Cosine, Tangent, Cotangent: with the purpose of having the mentioned function of an expression.

- Expression SquareRoot, Root:with the purpose of calculate the square of a expression, if its possible.

- Expression InverseMatrix: With the purpose of obtaining the matrix inverse of a given expression matrix.

- Expression MatrixLu: With the purpose of obtaining the matrix Lower Upper of a given expression matrix.

- Expression TridiagonalMatrix: With the purpose of obtaining the matrix tridiagonal of a given expression matrix.

- Expression RealEigenValues: With the purpose of obtaining the real eigenvalues of a given expression matrix.

- Expression Determinant: With the purpose of obtaining the determinant of a given expression matrix.

- Expression Function: With the purpose of making a function out of a given expression.

- Expression Integral: With the purpose of obtaining the value of a definite integral, given a function, an interval and the letter where it should operand.

- Expression Interpolate: With the purpose of interpolating a value in a set of values, given an expression vector, and expression with the number to interpolate.

- Expression ODEFirstOrderInitialValues: With the purpose of calculate a value in a Ordinary Differential Equation of First Order, and initial values, given the expression function, the expression with the initial values, the expression with the value to calculate, this should be larger than the one in initial values, and the variable of the equation.
- Expression FindRootBisection: With the purpose of obtaining the root of a given expression function between a given expression interval (Pair) .

Statements

We have the following statements based on the following interface

```
class Statement
{
public:
    virtual void execute(std::shared_ptr<Expression>, std::string)
const = 0;
    virtual ~Statement() {}
};
```

- Statement Display: With the purpose of showing in the console the given expression, this class is implemented with the design pattern Singleton.

- Statement Print: With the purpose of showing in the console a given string, this class is implemented with the design pattern Singleton.

If the user makes a mistake with the declaration of an expression the eval should return a nullptr, if it is an error, such as divided by 0, it should return IMPOSIBBLE.

Language Paradigm

The expression-oriented programming paradigm focuses on evaluating expressions and manipulating data in a declarative manner. This approach is highly suitable for mathematical and scientific operations, as transformations and calculations are often performed on data in a systematic manner.

Set Up

The Lenguaje will be found in a repository in github under the name "Mathematical Programing Lenguaje", The README.md file will provide the instructions for its execution.

The execution of the program will show the execution of a variety of Tests for the implemented expressions. Examples and explanations of what is shown will be provided below.

Examples

Important: For the use of the statements Display and Print, since they are implemented as a singleton, remember to get their instance, with the getInstance method.

```
Display& display = Display::getInstance();
Print& print = Print::getInstance();
```

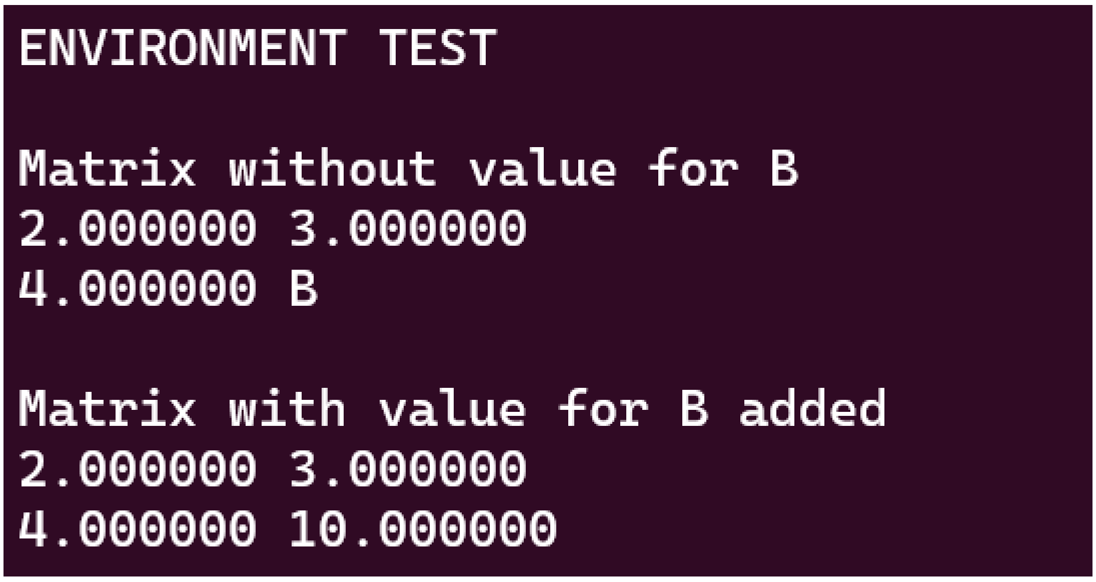
-Will explain the method *EnvTest* of the class **Test.**

```
void Test::EnvTest()
{
1     print.execute(nullptr, "\n\nENVIRONMENT TEST\n");
2
3     Environment env = std::forward_list<std::pair<char, std::shared_ptr<Expression>>>{};
4
5     std::vector<std::shared_ptr<Expression>> vec1 = {std::make_shared<Number>(2),
std::make_shared<Number>(3)};
6     std::vector<std::shared_ptr<Expression>> vec2 = {std::make_shared<Number>(4),
std::make_shared<Variable>('B')};
7     std::vector<std::vector<std::shared_ptr<Expression>>> mat1 = {vec1, vec2};
8     std::shared_ptr<Expression> expMat1 = std::make_shared<Matrix>(mat1);
9
10    print.execute(nullptr,"\nMatrix without value for B\n");
11    display.execute(expMat1->eval(env));
12
13    env.push_front(std::make_pair('B', std::make_shared<Number>(10)));
14    print.execute(nullptr, "Matrix with value for B added\n");
15    display.execute(expMat1->eval(env));
}
```

line 1: Shows the use of the statement print and execute method, this receives an expression (but does nothing with it, that's why it sends nullptr) and the string to show in the console.
line 3: An Environment it's created, at this moment is empty.
line 5,6: It is created two vectors of expressions that contain expressions numbers and expression variables.

line 7: It is created a vector that contains vectors of expressions, that contains the two previous vectors.
line 8: It’s created the expression Matrix.
line 10: It is used de print.execute to show a message.
line 11: It is used the statement display and execute method, this receives an expression and shows it. also we evaluate the matrix, return the matrix evaluated.
line 13: Adding in the environment a value for the letter ‘B’.
line 14: It is used the print.execute to show a message.
line 15: It is used the statement display and execute method, which receives an expression and shows it. also we evaluate the matrix, return the matrix evaluated.

Output of the Code



```
ENVIRONMENT TEST

Matrix without value for B
2.000000 3.000000
4.000000 B

Matrix with value for B added
2.000000 3.000000
4.000000 10.000000
```

-Will Explain the method *ODEFirstOrderInitialValuesTest* of the class **Test**

void Test::ODEFirstOrderInitialValuesTest()
{
1:   print.execute(nullptr, "\n\nORDINARY DIFFERENTIAL EQUATION FIRST ORDER INITIAL VALUES TEST\n");

2:   Environment emptyEnv = std::forward_list<std::pair<char, std::shared_ptr<Expression>>>{};

3:   std::shared_ptr<Expression> to = std::make_shared<Number>(0);
4:   std::shared_ptr<Expression> xo = std::make_shared<Number>(0);

5:   std::shared_ptr<Expression> num1 = std::make_shared<Number>(3);
6:   std::shared_ptr<Expression> num2 = std::make_shared<Number>(125);
7:   std::shared_ptr<Expression> var = std::make_shared<Variable>('x');

```
std::shared_ptr<Expression> div = std::make_shared<Division>(num1,num2);
std::shared_ptr<Expression> mult = std::make_shared<Multiplication>(div,var);

std::shared_ptr<Expression> num3 = std::make_shared<Number>(0.6);
std::shared_ptr<Expression> sum = std::make_shared<Subtraction>(num3,
mult);

std::shared_ptr<Expression> tFinal = std::make_shared<Number>(30);

std::shared_ptr<Expression> ODE1 =
std::make_shared<ODEFirstOrderInitialValues>(std::make_shared<Function>(sum),
std::make_shared<Pair>(to,xo), std::dynamic_pointer_cast<Number>(tFinal),
std::dynamic_pointer_cast<Variable>(var));

print.execute(nullptr, "Expression1: \n");
display.execute(ODE1);

print.execute(nullptr, "Eval1: \n");
display.execute(ODE1->eval(emptyEnv));

std::shared_ptr<Expression> ODE2 =
std::make_shared<ODEFirstOrderInitialValues>(std::make_shared<Function>(sum),
std::make_shared<Pair>(to,xo), std::make_shared<Number>(-10),
std::dynamic_pointer_cast<Variable>(var));

print.execute(nullptr, "\nExpression2: \n");
display.execute(ODE2);

print.execute(nullptr, "Eval2: \n");
display.execute(ODE2->eval(emptyEnv));
}
```

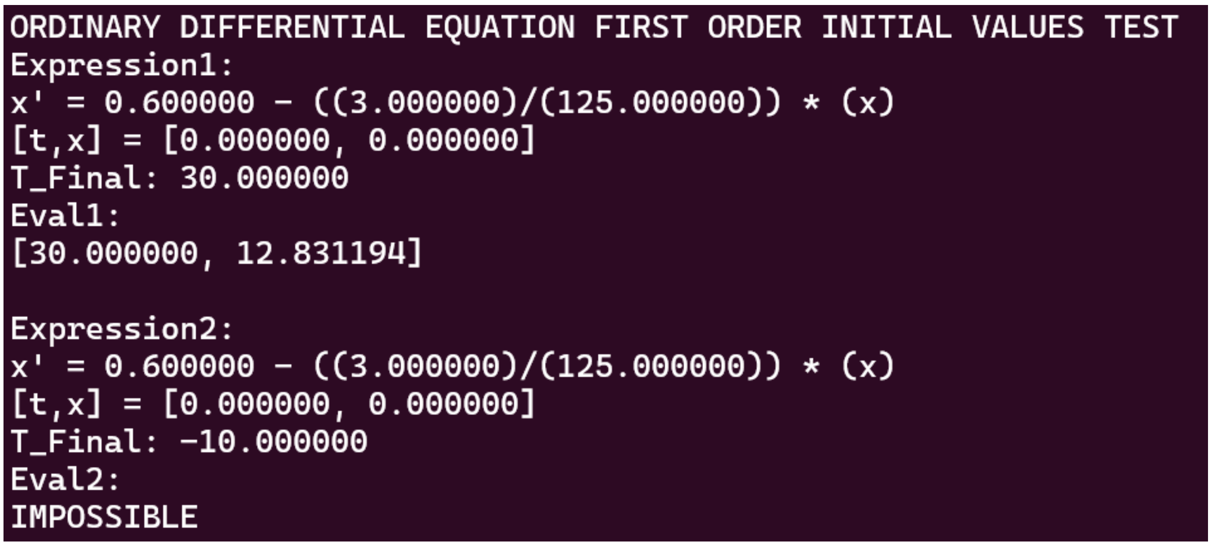
In this code we are solving the following ODE with initial values problem

$$\frac{dx}{dt} = 0.6 - \frac{3}{125}x$$

$$x(0) = 0,\ x(30) = ?$$

line 1: Shows the use of the statement print and execute method, this receives an expression (but does nothing with it, that's why it sends nullptr) and the string to show in the console.
line 2: An Environment it's created, at this moment is empty.
line 3, 4: It creates the expression numbers that represent the initial values of the problem.
line 5, 6, 7: It is created the expressions number that correspond to the second term of the ODE, and it is created the variable of the ODE.
line: 8, 9: It is created the expressions Division and multiplication that correspond to the second term of the ODE.
line 10: it is created the expression number that corresponds to the first term of the ODE.
line 11: It is created the expression subtraction of the two terms of the ODE previously declared.
line 12: It is created the expression number that represents the value of which x we want to find.
line 13: It is created the expression ODE, with all of his requirements, note that the expression sum it is converted to an expression function.
line 14: It uses the statement print and execute.
line 15: It uses the statement display and execute to show the ODE.
line 16: It uses the statement print and execute.
line 17: It uses the statement display and execute to show the eval of the ODE.
line 18: It is created the expression ODE, using the same function an initial values, but with the difference of a value of -10 to calculate (a value minor than the one in the initial values.
line 19: It uses the statement print and execute.
line 20: It uses the statement display and execute to show the ODE.
line 21: It uses the statement print and execute.
line 17: It uses the statement display and execute to show the eval of the new ODE.

Output of the Code



```
ORDINARY DIFFERENTIAL EQUATION FIRST ORDER INITIAL VALUES TEST
Expression1:
x' = 0.600000 - ((3.000000)/(125.000000)) * (x)
[t,x] = [0.000000, 0.000000]
T_Final: 30.000000
Eval1:
[30.000000, 12.831194]

Expression2:
x' = 0.600000 - ((3.000000)/(125.000000)) * (x)
[t,x] = [0.000000, 0.000000]
T_Final: -10.000000
Eval2:
IMPOSSIBLE
```