FormInterpreter

FormInterpreter

Class to

Evaluate

Math Formulas

Version 1.1.0

—

07. April 2016

last change: January 5, 2018

Contents

1 FormInterpreter

formInterpreter is a C++ class to evaluate a mathematical formula containing multiple variables and functions.

1.1 Features

The following features are supported:

- Basic operations addition, subtraction, multiplication, division and power.
- Order of operations: Power first, then multiplication and division and then addition and subtraction
- Nested parentheses and function calls.
- Default variables Pi and Euler are predefined
- Mathematical functions SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, SQRT, ABS, INT, EXP, LOG, LOG10
- Arguments for trigonometric functions can be defined as radians, degree or gon.
- Any number of variables can be defined from calling code without changing the source code of the formInterpreter Class.
- Any number of custom functions with any number of parameters can be defined from calling code without changing the source code of the formInterpreter Class.

Within this documentation,

The word *formInterpreter* is used for the class itself.

The word *formula* is used for the formula you want to evaluate.

The word *function* is used for mathematical functions within the formula.

Commands or code that must be entered is typed in this color.

Commands or code representing an example or reference is typed in this color.

1.2 Purpose

formInterpreter can be used for any application that must not only accept numbers for input, but also formulas to avoid the need for calculations done by the user of that application.

1.3 Compiling

formInterpreter is compiled using GNU compiler g++ version 4.4.3 using the -std=c++0x option. If your compiler does not support C++11 it will not compile the code.

To create the library, go to the directory containing the code and type

```
make libFormInterpreter.a
```

After creating the library, create the test program by typing

```
make test
```

and start it by typing

```
./test
```

The output should end with the lines

```
---------------------
| All tests passed! |
---------------------
```

The examples from this document can be created by typing

```
make exaX
```

where X must be replaced by the number of the example.

2 Usage

Formulas contain a mathematical expression containing operators, functions and - if needed - parentheses. Formulas can contain any number of blanks ot tabs. They are removed prior to the evaluation.

2.1 Operators

formInterpreter allows the basic mathematic operations (`+ - * / ^`) representing addition, subtraction, multiplication, division and power.

The `+` , `-` , `*` and `/` , operators are used like `4 + 5 - 2 * 10 / 5` .

The `^` operator is used like `2^3` .

The power operation `^` precedes all other operations.

The operations (`*`) and (`/`) precede the operations (`+`) and (`-`) .

So if the formula `4 + 5 - 2 * 10 / 5` is evaluated the result is 5 (the multiplication and the division are done first).

2.2 Mathematical Functions

The following mathematical functions are supported:

SIN(a): The sine of an angle. The angle has to be passed in radians except you define another type using the constructor (see page 9) or `initFunctions()` .

COS(a): The cosine of an angle. For angle type see SIN(a) above.

TAN(a) The tangens of an angle. For angle type see SIN(a) above.

ASIN(x): The arcus sine of a value. The resulting angle type depends definition, see SIN(a) above.

ACOS(x): The arcus cosine of a value. The resulting angle type depends definition, see SIN(a) above.

ATAN(x): The arcus tangens of a value. The resulting angle type depends definition, see SIN(a) above.

ATAN2(x,y): The arcus tangens of the value x/y. The resulting angle type depends definition, see SIN(a) above.

SQRT(x): The square root of a value.

ABS(x): The value multiplied by `-1` if it is negative.

FLOOR(x): The next smaller integral value.

EXP(x): The exponent base e.

LOG(x): The logarithm base e.

LOG10(x): The logarithm base 10.

All mathematical functions are upper case. This helps to avoid confusion with defined variables used within a formula. The argument to the function must be enclosed into parentheses.

2.3 Parentheses

A formula can contain any number of nested parentheses as long they are balanced. If the evaluation method finds an opening parenthesis the corresponding closing parenthesis is searched and the part between these parentheses is evaluated like a single formula.

2.4 Examples

Some examples may help to get an idea how a formula may look like.

```
3*5
```

```
1+2*3
```

```
SIN((0.2 * x)^2)/SQRT(x/45)
```

```
(x+4)*(x+2)*(x+1)*(x-1)*(x-3)/20+2
```

```
COS(x) + 0.1 * COS(6*x)
```

```
radius * COS(angle * Pi/180) + outerradius * COS(5 * angle * PI/180)
```

3 Reference

The following sections describe the usage of *formInterpreter* within your own code.

3.1 Constructor

The constuctor takes the formula to be evaluated as its only required argument. It is passed as std::string. All other arguments are optional.

```
formInterpreter(std::string formula,
                angleType type = isRad,
                double round = 0.,
                bool doExceptionText = false)
```

`formula` This is a `std::string` containing the formula that must be evaluated. The formula is checked for balance of parentheses.

`type` can be set to one of three specifications:

1. `formInterpreter::isRad`: if you want to calculate with angles defined in radians (from 0 to 2*Pi),
2. `formInterpreter::isDegree`: if you want to calculate with angles defined in degree (from 0° to 360°),
3. `formInterpreter::isGon`: if you want to calculate with angles defined in gon (from 0^g to 400^g).

When using trigonometric functions, the argument is interpreted as radians, degree or gon depending on the type definition. When using the arcus functions, the result is converted to that type. The default is `formInterpreter::isRad`.

`round` is added to a value from witch you want to calculate the next smaller integral value `FLOOR()` . If you want to round to the next nearest integral value, set `round = 0.5` . Else set it to `round = 0.0` which is the default. Be aware that a negative argument returns a result with an absolute value larger than the absolute value of the argument (for example, FLOOR(-0.5) $=$ -1).

`doExceptionText` must be set to true to write the reason of an exception to the stdout if one is thrown. Alternatively, catch the exception and call `std::string getLastError()`.

3.2 Calculating the Value

To calculate the result of the formula, the method

`calc()`

is used. Before the evaluation starts, *formInterpreter* removes all white spaces. Then the interpretation runs recursively through the formula. First, all variables are replaced by their values. After doing this the formula is ready for interpretation. *formInterpreter* reads the first value and the first operator. Then a loop processes the remaining parts (value, operator, value, operator, ...) of the formula, always checkig if the next operator has a higher relevance than the present one. A value can be a number, a variable, an expression within parentheses or a mathematical function. Parentheses or functions can be nested. An opening Parenthesis forces *formInterpreter* to interpret the contents within these parentheses recursively.

In the following example the formula `2*3` is passed to the constructor (line 6). For `type`, `round` and `doExceptionText` the defaults are used.

```
1  #include <iostream>
2  #include "FormInterpreter.h"

4  int main()
5  {
6    formInterpreter MyFormula("2*3");
7    double result = MyFormula.calc();
8    std::cout << "Result = " << result << std::endl;

10   return 0;
11 }
```

Listing 3.1: Example of Constructor

If you run this example (exa1.cpp) its output is

```
Result = 6
```

3.3 Using one Variable

If a function contains one variable, a method

`calc(double x)`

is available for convenience to pass the variable as `x`. To use `x` within your formula, just place it as you would place a number in it, for example `x^2`. `x` is replaced with the passed argument of `calc(double x)` and the formula is evaluated.

Using `calc(double x)` in a loop allows to calculate a whole range of values for one formula, for example (exa2.cpp):

```
1  #include <iostream>
2  #include "FormInterpreter.h"

4  int main()
5  {
6    formInterpreter MyFormula("x^2");
7    for (int i=1; i<=3; i++) {
8      double result = MyFormula.calc((double)i);
9      std::cout << "x = " << i << " Result = " << result << std::endl;
10   }
```

```
12    return 0;
13 }
```

Listing 3.2: formInterpreter and Loops

Within this loop the squares of the numbers 1, 2 and 3 are calculated with the result:

```
x = 1 Result = 1
x = 2 Result = 4
x = 3 Result = 9
```

3.4 More Variables

If more than one variable is present in the formula they have to be defined prior to the calculation. This can be done in two ways:

3.4.1 New Variables without changing formInterpreter's code

The method to introduce a variable is

`defineVariable(std::string name, double value)`

where `name` is the name of the variable and `value` is the value the variable represents. The variable name is case sensitive.

It is strongly recommended to use NO uppercase-only variable names to avoid conflicts with the predefined variables or supported mathematical functions (that should be in upper case only like SIN for sinus etc.). If a variable name is already defined as a function name or is part of a function name, an exception will be thrown. This is done to avoid messing up your formula by replacing a function with the contents of a variable. Simply use at least one lower case character within you variable name.

The following example evaluates `radius * COS(angle)` using the variables `radius = 2` and `angle = 45` defined by using the `defineVariable()` method. When `calc()` is called the variables within the formula are replaced by their values, then the formula is evaluated and the result is calculated.

```
1  #include <iostream>
2  #include "FormInterpreter.h"

4  int main()
5  {
6    formInterpreter MyFormula("radius * COS(angle)",
7                              formInterpreter::isDegree);
8    MyFormula.defineVariable("radius", 2.);
9    MyFormula.defineVariable("angle", 45.);
10   double result = MyFormula.calc();
11   std::cout << "Result = " << result << std::endl;

13   return 0;
14 }
```

Listing 3.3: Multiple Variables

The output of this example (exa3.cpp) is:

```
Result = 1.41421
```

When replacing the variables with their values the variables with the longest names will be replaced first to avoid conflicts when a smaller variable name is contained in a longer one. So it is no problem to define for example variable names `varname` and `myvarname` at the same time.

3.4.2 New Variables by changing formInterpreter's code

If some variables are used frequently you may not want to define them for every instance of *formInterpreter*. Two predefined variables `Pi` and `Euler` are already inserted. Add your own frequently needed constants into the class by extending the method `initVariables()` (see 4.3).

3.4.3 Redefining Variables

When evaluating the formula for more sets of variables, simply redefine a variable by using `defineVariable()` again for that variable. `defineVariable()` will look up the variable and replace its value.

3.4.4 Deleting a Variable

You can also delete the variable list using the method `clearVariables()` . This results in a variable list where only the predifined variables `Pi` and `Euler` (and of course the variables you added to *formInterpreter's* code) are defined.

3.4.5 Check if Variable exists

A call to the method

```
bool checkVariableExists(std::string name)
```

returns true or false depending of the existence of the passed variable name.

3.4.6 Example redefining Variables

The following example calculates the points on a circle using two instances of *formInterpreter*. For every calculation the variable for the angle is redefined in both instances of *formInterpreter* (exa4.cpp).

```
1  #include <iostream>
2  #include "FormInterpreter.h"

4  int main()
5  {
6    formInterpreter xFunction("radius * COS(angle)",
7                              formInterpreter::isDegree);
8    formInterpreter yFunction("radius * SIN(angle)",
9                              formInterpreter::isDegree);
10   xFunction.defineVariable("radius", 2.);
11   yFunction.defineVariable("radius", 2.);
12   for (int i=0; i<=360; i++) {
13     xFunction.defineVariable("angle", (double)i);
14     yFunction.defineVariable("angle", (double)i);
15     double x = xFunction.calc();
16     double y = yFunction.calc();
17     std::cout << "x = " << x << " y = " << y << std::endl;
18   }
19   return 0;
20 }
```

Listing 3.4: Multiple Variables in Loop

In the last example we only used one changing variable per instance of *formInterpreter* so we can also use the method `calc(double x)` to calculate the circle (exa4a.cpp):

3.5 Change Function Factor or Formula

It is possible to redefine the values passed to the constructor:

3.5.1 New Factor

To change the factor for trigonometric functions or a value to be added when using the FLOOR function use

```
void initFunctions(angleType type, double round)
```

See the constructor description in 3.1 for information about the parameters. Be aware that this call removes all custom defined functions from the internal function vector.

3.5.2 New Formula

When you want to change the formula that es evaluated by an instance of *formInterpreter*, call

```
initFormula(std::string formula)
```

with your new formula and proceed. This method is also called by the constructor and initialized everything as needed.

3.6 Adding custom Functions

Custom functions can be added from the code that uses this class. This version of *formInterpreter* supports functions with any number of double arguments (See page 17).

The function you want to add must be defined in your code. For example a function using two arguments is defined like this:

```
//------------------------------------------------------------
double _Formula_AreaRectangle(double width, double height) {
  return (width * height);
}
```

Then, call to

```
void addFunction(std::string fname,
                 double (*ffunction)(double, double),
                 double preFactor = 1.,
                 double postFactor = 1.,
                 double preAdd = 0.);
```

adds your function to the function list:

```
addFunction("AREA", _Formula_AreaRectangle, 1, 1, 0);
```

Here, `AREA` is the name of the function used within the formula.

The arguments for `preFactor`, `postFactor` and `preAdd` will not change the arguments or the result (the factors are 1 and 0 is added). Because these values are default, you can also call

```
addFunction("AREA", _Formula_AreaRectangle);
```

In general you can omit the last three parameters in most of the cases, when not dealing with arguments that have usually other representation for a human typing a formula than the mathematical function taking an argument (like sin() that takes radians which are not comfortable to handle for a user).

The following example shows how to define a function `AREA` with two arguments and use it within a formula:

```
1  #include <iostream>
2  #include "FormInterpreter.h"
3  //------------------------------------------------------------
4  double _Formula_AreaRectangle(double width, double height) {
5    return (width * height);
6  }
7  //------------------------------------------------------------
8  int main()
9  {
10   formInterpreter Function("2 * AREA(5., 6.)");
11   Function.addFunction("AREA", _Formula_AreaRectangle, 1, 1, 0);
12   std::cout << "2 * AREA = " << Function.calc() << std::endl;
13   return 0;
14 }
```

Listing 3.5: Defining Functions at Runtime

The output of this example (exa5.cpp) is:

```
2 * AREA = 60
```

4 Changing the Code

There might be reasons to change or enhance the code. Besides removing bugs (I hope there are none but you never know) there are two main reasons for code changes: Add more mathematical functions or add more frequently used constants without the need to do that in every program you write.

4.1 More Math Functions

If you want to add more mathematical functions simply enhance the `functions` vector defined in `initFunctions()` . Be aware that the function must accept between one and two arguments. The number of arguments must not be passed because the interpreting method counts the number of arguments and passes them to the proper function.

Functions using more than 2 arguments must be defined in your code at runtime as described in 4.2.

```
functions.push_back(new c_function("NEWFUNCTION", newfunction, 1./trigFactor, 1., 0.0));
```

To add a function to the function vector, pass an instance of `s_function` to it. `s_function` takes 5 parameters:

The function name (here: `NEWFUNCTION`) as it will show up in a formula,

The function itself (here the `newfunction()`. function from the math library),

The factor the function argument will be multiplied by prior the function call (here `1./trigFactor`),

The factor, the result of the function will be mulitilped with (here it is just 1) and

The value that is added to the argument before it is passed to the function (here 0.).

These factors/values allow to calculate with different angle types (SIN, COS, TAN, ASIN, ACOS, ATAN) and allow more flexible rounding (INT). `1./trigFactor` depends on the definition of the constructors `1./type` argument and specifies the factor to convert the function's argument to radiens. It is defined in the `initFunctions()` method.

```
if (type == isRad) trigFactor         = 1.;
else if (type == isDegree) trigFactor = 180. / M_PI;
else if (type == isGon) trigFactor    = 200. / M_PI;
```

Simply add your function to the code and pass its name for the interpreter and the function itself to the function vector like this:

```
functions.push_back(new c_function("MYFUNCTIONNAME"   , myfunction   ,
                                   1., 1., 0.0));
```

While interpreting, *formInterpreter* will call `myfunction` with its arguments when it finds `MYFUNCTIONNAME` in the formula.

4.2 Add Functions with multiple Arguments

Before we begin: There is a more elegant way to add arguments in newer c++ versions but this one also works with older compilers:

In your code, prepare the function(s) to be added:

We use a function that takes a vector of doubles. Forminterpreter collects all parameters that appear within the parenteses after a function name into that vector. For example, the function `SUM(3, 5, 7, 9)` will lead to a vector of doubles containing the numbers 3, 5, 7, and 9.

The definition of this function looks like this:

```
double _FormInterpreter_Sum(std::vector<double> parameters);
```

The function itself is defined as follows:

```
double _FormInterpreter_Sum(std::vector<double> parameters) {
  double result = 0.;
  for (unsigned int i=0; i<parameters.size(); i++) {
    result += parameters.at(i);
  }
  return result;
}
```

Now the function can be passed to the Forminterpreter class giving it a name for interpretation:

```
myFormInterpreter.addFunction("SUM", _FormInterpreter_Sum, 1., 1., 0.0);
```

That's it. If you want to add a function that requires an exact number of arguments it is recommended to throw an exception if the number of arguments passed by the user is wrong:

```
double _FormInterpreter_Half(std::vector<double> parameters) {
  if (parameters.size() != 1) throw FormulaException(100, "Half: One parameter needed");
  return parameters.at(0) / 2.;
}
```

4.3 More Predefined Variables

To add more variables to the class simply append them by adding lines like

`addVariable("MyConstant", 4.38677);`

to the method `initConstants()` . Please remember to use NO uppercase only variables. If a variable name matches a function name or a part of it, an exception will be thrown when you create an instance of *formInterpreter*.

Listings

Index