MathCompiler Version 1.0.0.0

Frequently asked Questions (FAQ)

What is MathCompiler?

MathCompiler is a mathematical function compiler that allows efficient and fast calculation of function values of arbitrary complex single-line mathematical formulas that are provided as a string at design or at runtime.

What means "efficient and fast"?

Mathematical compilers differ in the number and the nature of their commands due to their different applied optimization techniques (if they have any at all). Commands should be optimized for calculation speed in combination with maximum versatility (which of course contradicts each other). Optimization techniques for example recognize identical sub expressions or normalize the calculation order. Consider the formula

```
1/(X0 + 2.34) + pi/2 + exp(X1/(2.34 + X0))
```

A math compiler should recognize that the terms `(X0 + 2.34)` and `(2.34 + X0)` are identical. Then this term should only be calculated once and then reused throughout the calculation. In addition the constant expression `pi/2` can be calculated a priori to a resulting constant value. Advanced techniques provide optimizations like these on successive levels of abstraction. *MathCompiler* implements optimization techniques for an excellent calculation performance in balance with a fast compilation speed. The precise optimization behavior of *MathCompiler* instances may be controlled by 4 independent optimization options: Option 1 affects the a priori calculation of constant value expressions, option 2 tries

to reduce the necessary calculations due to identical or similar sub expressions, option 3 tries to minimize the number of internal stack calls necessary for the function value calculation and option 4 affects the elimination of identical vectors.

What about MathCompiler compilation optimization options?

You can set 4 independent *MathCompiler* optimization options: Option 1 affects the a priori calculation of constant value expressions, option 2 tries to reduce the necessary calculations due to identical or similar sub expressions, option 3 tries to minimize the number of internal stack calls necessary for the function value calculation and option 4 affects the elimination of identical vectors. By default all optimization options are active. Dependent on the complexity of your function of interest these compiler options may influence the function value calculation speed and the compilation speed significantly. As a rule of thumb: The faster the function value calculation the slower the function compilation – but you can simply balance the overall performance: If you need a lot of *MathCompiler* instances but only a few (i.e. thousands or tens of thousands) function value calculations you may deselect all compilation options. On the other hand if you need millions of functions value calculations the activation of all optimization options may be a good choice.

What are the limits of optimization?

MathCompiler assumes that a formula already has an optimum structure. Optimization techniques are only applied to minimize the number of internal commands necessary to calculate the function value. For example optimization of the formula

```
mean({X0,X1,X2}) - sum({X0,X1,X2})/count({X0,X1,X2})
```

leads to a single (and not threefold) evaluation of the identical argument vectors but *MathCompiler* does not recognize that this formula always evaluates to 0.

What is the difference to a math interpreter?

A math compiler transforms the mathematical formula string into an optimized sequence of elementary commands that are executed whenever a function value is calculated. In contrast a math interpreter "interprets" the formula string "step by step" every time a function value is calculated. Thus a math interpreter function value calculation is usually slower than the optimized calculation of a math compiler – especially for complex formulas.

What platforms is MathCompiler available for?

MathCompiler is coded in C# and is available for the .NET platform.

Is MathCompiler open?

Yes, *MathCompiler* is licensed under GNU General Public License V3 with the source code published at GitHub (https://github.com/zielesny/MathCompiler).

Is MathCompiler thread-safe?

Yes.

How fast are MathCompiler function value calculations?

This surely depends on the mathematical formula in question as well as the used hardware. As a rule of thumb you can instantiate O(10.000) *MathCompiler* objects per second and perform about O(10 Million) function value calculations in another second.

What about languages and internationalization?

MathCompiler uses the "standard" mathematical notation of the "Anglo-Saxon world" which cannot be changed. Information about possible internal problems is provided in form of comment strings in English or as coded information in form of enumerated items and corresponding values. The latter allow the creation of information messages in arbitrary languages. All arguments of the comment strings

are provided separately via the *CodedInfoItem* instances together with enumerated *CodedInfo* items.

What may MathCompiler formulas consist of?

Formulas may consist of scalar and vector arguments, scalar and vector constants, vectors, standard mathematical and logical operators, a conditional IF operator, scalar and vector functions:

Scalar arguments are coded with an initial x or X and a following index starting from zero: X0, X1, X2, …, X32,… The indices refer to the positions in the argument array passed to the `Calculate()` method.

Vector arguments consist of an initial x or X, a following index starting from zero and a "{}" combination to indicate their vector character. Vector arguments are passed via a jagged array to the `Calculate()` method.

Scalar constants can be predefined (like pi) or may have a standard numerical definition (with a point as a decimal separator) like 3.57 or a scientific notation like 1.2E-3. User-defined constants may be provided by customized classes implementing the *IConstant* interface.

Vector constants as arguments for vector functions are defined in curly brackets and consist of scalar constants, e.g. {2.1, 4.7, 3.3}.

Vectors as arguments for vector functions are defined in curly brackets and may consist of arbitrary complex expressions as components, e.g. {2.1, sin(X0), 3.3}. Vectors may have vector constants or vector arguments as components, e.g. {2.1, X0, {3.3, 0.4}, X1, X0{}, 7.4} but are not allowed to be generally nested: {{2.1, sin(X0)}, 3.3} is forbidden (and unnecessary: Simply replace by {2.1, sin(X0), 3.3}).

The **logical operators** AND, OR and NOT are provided. The double representation value for a logical true is 1.0 and for a logical false the value 0.0.

All double values which are different from 0.0 are treated as a logical true in logical expressions.

The **comparison operators** < (less), <= (less or equal), = (equal), <> (unequal), >= (greater or equal) and > (greater) evaluate to the double representation of a logical true or false.

The **standard mathematical operators** + (addition), - (subtraction), * (multiplication), / (division) and ^ (power) are supplied.

The **operator precedence** is as follows: OR below AND below =, <> below <, <=, >=, > below +, - below *, / below ^ below unary +, -, NOT. Normal brackets – "(" or ")" – may be used to assign precedence in an arbitrary nested fashion.

A **conditional IF** with syntax: IF(condition, true, false), e.g. IF(X0 > 5, exp(X0 / (X0 - 3.4)), sin(X0)): If X0 is greater 5 the term exp(X0 / (X0 - 3.4)) is calculated, otherwise sin(X0).

Basic **scalar functions** like sin, cos or exp are provided in source code. Arbitrary scalar functions may be added to a *MathCompiler* instance by user defined classes that implement the *IScalarFunction* interface. Scalar functions may have an arbitrary number of scalar arguments.

Basic **vector functions** with vector and scalar arguments like sum, mean or rmserror are provided in source code. Additional vector functions may be added to a *MathCompiler* instance by user defined classes that implement the *IVectorFunction* interface. Vector functions may have an arbitrary number of vector and scalar arguments in arbitrary (but defined) order. Vector arguments may have an arbitrary number of components.

Expressions may be arbitrarily complex (only restricted by the available memory of your hardware) and nested with brackets. The formula string may contain arbitrary white space characters, characters a-z and A-Z, operators, the decimal point, the comma, digits as well as normal and curly brackets.

What is the difference between the "conditional IF" and a standard if-function with 3 arguments?

A standard if-function is evaluated like every function: First all (!) arguments are calculated, then the function value is determined. The "conditional IF" is a true compiler operator, i.e. the true or false case – which each may be complex mathematical expressions – is only calculated if necessary. This may lead to considerable performance enhancements.

How do formulas look like?

Examples of valid *MathCompiler* formulas are:

```
1 + 1
```

```
X0/(2.16 – X2) * exp(X1/(2.16 – X2)) + X3
```

```
mean(X0{}) + 3.77
```

```
IF(X0 < count(X0{}), subtotal{X0{}, 0, X0}, sum{X0{}})
```

```
mean({2.1, 4.8, 6.3}) + count({2.1, 4.8, 6.3})/sum({2.1,
4.9, 6.3}) - count({2.1, 4.8, 6.3})
```

```
mean({2.0, X0, X1 - 1}) + 1/mean({2.0, X0, X1 - 1})
```

```
IF(X0 < X1 + 2.44 AND X0 > 0, exp(X0), sin(X0))*
1/exp(X1/(4.76 – X0))
```

Examples of invalid formulas are:

`sin(X0{})` (a scalar function is not allowed to take a vector argument)

`X0{} + 3*exp(X0)` (a vector argument is only allowed as an argument of a vector function)

The formula string may contain arbitrary white space characters, characters a-z and A-Z, operators, the decimal point, digits as well as normal and curly brackets.

How do I work with MathCompiler?

A *MathCompiler* instance consists of two main functions: `SetFormula()` for the definition of a mathematical formula and `Calculate()` for the calculation of function values. Here is a code snippet:

```
using MathCompiler;

Double[] myArguments;
Double myCalculatedFunctionValue;
String myFormula;
MathCompiler myMathCompiler;

// Create a new MathCompiler instance
myMathCompiler = new MathCompiler();
// Set formula
myFormula = "X0 / (X1 - 3.5) * exp(-X2^2)";
if (!myMathCompiler.SetFormula(myFormula))
{
      // Something went wrong: See properties Comment ...
      String detailedComment = myMathCompiler.Comment;
      // ... or CodedComment ...
      CodedInfoItem detailedInfoItem = myMathCompiler.CodedComment;
      // ... for detailed information ...
}
// Set Arguments:
// myArguments[0] corresponds to X0,
// myArguments[1] corresponds to X1 etc.
myArguments = new Double[] { 2.1, 6.3, 4.7 };
// Calculate function value
try
{
       myCalculatedFunctionValue = myMathCompiler.Calculate(myArguments);
}
catch (Exception)
{
       // Exception handling ...
}
```

The components of the argument vector (elements of the array) `myArguments` refer to the arguments of the formula: `myArguments[0]` to `X0`, `myArguments[1]` to `X1` etc.

If the formula contains vector functions the overload for vector arguments of the `Calculate()` method is to be used:

```
using MathCompiler;


Double[] myArguments;
Double[][] myVectorArguments;
Double myCalculatedValue;
String myFormula;
MathCompiler myMathCompiler;
```

```
// Create a new MathCompiler instance
myMathCompiler = new MathCompiler();
// Set formula
myFormula = "mean(X0{}) * (X0 + 2.5)";
if (!myMathCompiler.SetFormula(myFormula))
{
      // Something went wrong: See properties Comment ...
      String detailedComment = myMathCompiler.Comment;
      // ... or CodedComment ...
      CodedInfoItem detailedInfoItem = myMathCompiler.CodedComment;
      // ... for detailed information ...
}
// Set Arguments:
// myArguments[0] corresponds to X0,
// myArguments[1] corresponds to X1 etc.
myArguments = new Double[] { 3.5 };
// myVectorArguments[0] corresponds to X0{},
// myVectorArguments[1] corresponds to X1{} etc.
myVectorArguments = new Double[1][];
myVectorArguments[0] = new Double[] { 1,2,3,4 };
// Calculate function value
try
{
       myCalculatedValue =
              myMathCompiler.Calculate(myArguments, myVectorArguments);
}
catch (Exception)
{
       // Exception handling ...
}
```

For both overloaded `Calculate()` methods there is also a “safe” implementation available which avoids the use of pointers.

Does the C# implementation of MathCompiler use “unsafe” code?

Yes, it does for the `Calculate()` methods (which leads to a performance gain of about 25%). But there are also “safe” implementations available.

What standard constants, functions and vector functions does MathCompiler provide?

MathCompiler consists of the following standard constants, functions and vector functions:

Constants:

e - natural logarithmic base.

false - double value representation of logical false.

pi - mathematical constant pi (greek).

true - double value representation of logical true.

undefined - double value representation of an undefined value.

Functions:

abs(arg0) - absolute value of the specified number.

acos(arg0) - angle measured in radians whose cosine is the specified number.

asin(arg0) - angle measured in radians whose sine is the specified number.

atan(arg0) - angle measured in radians whose tangent is the specified number.

cos(arg0) - cosine of an angle in radians.

cosh(arg0) - hyperbolic cosine of an angle in radians.

exp(arg0) - e raised to the specified power.

lg(arg0) - base 10 logarithm of a specified number.

ln(arg0) - natural logarithm of a specified number.

log(arg0, arg1) - logarithm of a specified number (first argument) in a specified base (second argument).

min(arg0, arg1) - smaller of two numbers.

max(arg0, arg1) - larger of two numbers.

round(arg0, arg1) - rounded value to the specified precision: Second argument is the number of digits.

sin(arg0) - sine of an angle in radians.

sinh(arg0) - hyperbolic sine of an angle in radians.

sqrt(arg0) - square root of a specified number.

tan(arg0) - tangent of an angle in radians.

tanh(arg0) - hyperbolic tangent of an angle in radians.

Vector functions:

component(arg0{}, arg1) - specified component of vector argument: Scalar argument = index of component.

count(arg0{}) - number of components of argument vector.

maxComponent(arg0{}) - maximum component of vector argument.

mean(arg0{}) - arithmetic mean of components of vector argument.

minComponent(arg0{}) - minimum component of vector argument.

sampleError(arg0{}) - sample standard deviation of components of vector argument.

subTotal(arg0{}, arg1, arg2) - sum of range of components of vector argument: Scalar argument 1 = first component, scalar argument 2 = last component.

sum(arg0{}) - sum of components of vector argument.

This basic set of constants, functions and vector functions may be easily extended: Just write additional customized classes implementing the *IConstant*, *IScalarFunction* or *IVectorFunction* interface.

How can I write additional customized constants, functions and vector functions?

Additional customized constant, function of vector function classes have to implement interfaces *IConstant IScalarFunction, IVectorFunction*. The *MathCompiler* source code provides all standard constants, functions and vector functions which may be used as templates for customized developments.

Why should I put constant value sub expressions into brackets?

… because it helps the optimization procedure to recognize them which leads to a faster calculation. Consider the formula

```
(X0 + 3) - (4 + exp(5/3)*(X1 - 2)) - pi/2.34 + X0 - 4/2 + 1
```

The final term should be put into brackets

```
(X0 + 3) - (4 + exp(5/3)*(X1 - 2)) - pi/2.34 + X0 - (4/2 - 1)
```

to support the optimization procedure. For the other constant value sub expressions `exp(5/3)` and `pi/2.34` this is not necessary since they already have a higher priority than their surroundings. To avoid any performance drawbacks always put constant value sub expressions in (maybe unnecessary) brackets because this never leads to slower calculations:

```
(X0 + 3) - (4 + (exp(5/3))*(X1 - 2)) - (pi/2.34) + X0 - (4/2 - 1)
```