ANALYTICS C# Development Library Version 6.5

Manual

Copyright © Sergey L. Gladkiy
Email: lrndlrnd@mail.ru
URL: https://vk.com/analytics_and_physics

Contents

Introduction

ANALYTICS is a C# library for developers. ANALYTICS library contains special classes to work with 'analytical' expressions in .NET programs – parse expressions, calculate expression values and so on.

ADVANTAGES of ANALYTICS library:

1. 100% C# code.
2. Strongly structured class hierarchy.
3. Universal algorithms (working with formulae of any complexity).
4. Many predefined functions.
5. Easy to introduce new functions for any argument types.
6. Easy to overload operators for any argument types.
7. Working with Complex numbers.
8. Working with Common Fractions.
9. Working with 3D vectors and tensors.
10. Working with physical values and units of measurement.
11. Working with indexed data (arrays, matrixes and higher dimensioned data).
12. Working with special functions.
13. Analytical derivative calculation.
14. Ready-to-use numerical tools integrated with symbolic features.

Dependences

ANALYTICS C# library depends on:

1. NRTE (NET Run-Time Environment) library.
2. MATHEMATICS library.

Extensions

There are the following extensions of ANALYTICS C# library:

1. ANALYTICS Fractions (depends on MATHEMATICS C# library).
2. ANALYTICS Linear Algebra (depends on MATHEMATICS C# library).
3. ANALYTICS Complex (depends on MATHEMATICS C# library).
4. ANALYTICS Special (depends on MATHEMATICS C# library).
5. ANALYTICS Mathphysics (depends on MATHEMATICS and PHYSICS C# libraries).
6. ANALYTICS Numerics (depends on MATHEMATICS library).
7. ANALYTICS Statistics (depends on MATHEMATICS library).

Basic concepts

The main goal of ANALYTICS library is to provide easiest way to evaluate mathematical expressions. This part explains main concepts, used in ANALYTICS library to work with mathematics expressions and formulae.

Expressions

Mathematical expression is a sequence of elements for which its result value can be calculated. A simple example of expression is 'x+y' – knowing the values of x and y we can calculate their sum. Generally, expression contains such elements as **constants** (**literals**), **variables**, **operators**, **functions**. The result of an expression calculation depends on elements the expression consists of. For an example if 'x' and 'y' are real numbers (variables) – the result of the given sum expression is a real number, if one of the values is a complex value – result is a complex value too.

Literals

Literal is an unnamed constant value or a symbol staying for 'standard' constant value. For an example, in the expression **'2*(x+y)'** – **'2'** is a numerical literal. **Real** and C**omplex** literals are supported.

Real literals can be written in simple and exponential form. Simple form examples: **'100'**, **'1.23'**, **'-45.67'**[1]. Exponent form examples: **'1.23E-3'**, **'-2.45e+5'**.

For complex literals symbol **'I'** is used to denote **imaginary unit**. Complex literal consists of Real part and Imaginary part separated by **'+'** or **'-'**symbols. Real part is just a real literal, imaginary part is real literal plus imaginary unit symbol (**NOTE:** there is no multiplication symbol between real literal and imaginary unit symbol). Complex literal examples: **'I'**, **'-4I'**, **'2+3.2I'**, **'-2e2-4.45I'**, **'2.45+I'**, **'-1.2e-3+4.5e+2I'**.

The following 'standard' constants are recognized by parser: 'e' – Euler number, 'π', 'Pi' – Pi number, '∞' – positive infinity.

Variables

Variable is a named value (that can be changed during calculation). A variable **name** can include alpha-numeric symbols, subscript and superscript digits and underscore symbol (the first symbol may be letter only)[2]. The value of a variable can be of any type. The type of a variable cannot be changed during calculation. The variable value is accessed via variable name. That is in an expression the variable name stands for the current variable value. For an example, let the variable **'A'** is a real variable whose current value is **'1.0'**, then the result of 'A+1' expression is **'2.0'**. Changing variable values allows calculation of the same expression for various values.

Operators

Syntactically **operator** is a symbol representing some mathematical operation. For an example, operator **'+'** stands for the sum operation. From the functional point of view, an operator implements some action with data values, called **operands**, and returns the result value.

All operators can be divided into many categories. The most common used are **unary** and **binary** operators. **Unary operators** have one operand and can be **prefix** (stands before its operand) and **postfix** (stands after its operand). An example of unary prefix operator is the **negation operator** (**'-x'**, **'-'** is the operator, **'x'** is the operand). An example of unary postfix operator is the **factorial operator** (**'n!'**, **'!'** is the operator, **'n'** is the operand). **Binary operators** have two

---

[1] Current Culture Decimal separator is used in real literals.

[2] WARNING: do not use along imaginary symbol ***I*** as variable name, it will always be treated as literal value because of literal parsing precedence.

operands and commonly stand between them. An example of binary operator is the **addition operator** (**'x+y'**, **'+'** is the operator, **'x'** and **'y'** are the operands).

Each operator has such attributes as **precedence** and **associativity**. The **precedence** determines the order of expression calculation. The operators with higher precedence applied to their operands before the operators with lower precedence. For an example, in the expression **'x+y*z'** the first operation performed is the multiplication of **'y'** and **'z'** and then the sum of **'x'** and the product result is calculated, because the **'*'** operator is of higher precedence than the **'+'** is. The order of calculation can be changed by using parantheses **'()'**. The **associativity** determines how operators with the same precedence are grouped in expressions. Let us consider the expression **'2^3^4'** (where the **'^'** operator stands for the power). The result of the expression depends on how the expression is interpreted – **'(2^3)^4=8^4'** or **'2^(3^4)=2^81'**. **Left** associativity means that operators are grouped from left to right (the first case), **right** associativity means grouping from right to left (the second case).

The following operators are defined in ANALYTICS library:

'+' addition operator (binary);
'-' subtraction operator (binary);
'*' multiplication operator (binary);
'/' division operator (binary);
'^' power operator (binary);
'•' dot operator (binary);
'x' cross operator (binary);
'-' negation operator (unary, prefix);
'~' tilde operator (unary, prefix);
'√' square root operator (unary, prefix);
'!' factorial operator (unary, postfix);
''' apostrophe operator (unary, postfix);
'`' accent operator (unary, postfix);
'≡' identically equal operator (binary);
'≈' approximately equal operator (binary);
'≠' not equal operator (binary);
'>' greater than operator (binary);
'<' less than operator (binary);
'≥' greater than or equal operator (binary);
'≤' less than or equal operator (binary);
'&' logical and operator (binary);
'\' logical or operator (binary);
'¬' logical not operator (unary, prefix);
'?' question operator (unary, prefix);
'#' number operator (unary, prefix);
'||' absolute operator (unary, outfix);
'‖‖' norm operator (unary, outfix);

Also the following 'Special'[3] operators defined:

'←' left arrow operator (binary);
'→' right arrow operator (binary);
'↑' up arrow operator (binary);
'↓' down arrow operator (binary);
'↔' left-right arrow operator (binary);
'↕' up-down arrow operator (binary);
'∂' derivative operator (unary, prefix);
'∫' integral operator (unary, prefix);
'∆' delta operator (unary, prefix);
'∑' sum operator (unary, prefix);
'∏' product operator (unary, prefix);

General rules for all operators:

- Binary algebraic operators (+, -, *, etc.) have precedence as determined with standard mathematical rules.
- Relational operators (binary) have lower precedence than algebraic ones.
- Binary Logical operators have lower precedence than relational ones.
- Arrow operators (binary) have higher precedence than power operator.
- Binary operators are all left-associative.
- Unary operators have higher precedence than binary operators.
- Postfix operators have higher precedence than prefix operators.

The standard mathematical operators have default implementation for real operand types. Operators for other operands (complex, 3D vectors and tensors and so on) implemented in ANALYTICS extension libraries (see above).

All predefined operators can be overloaded (defined) for any other operand types. The following restrictions are applied for operator overloading:

- new operators cannot be defined;
- number of operands and attributes of operators (precedence and associativity) cannot be changed.
- attributes of operators (precedence and associativity) cannot be changed.

[3] 'Special' operators have no predefined meaning for common data types (real, complex and so on) and can be used for special needs with program specific types. Note that the syntax rules cannot be changed for the operators in any case.

Functions

Syntactically **function** can be considered as a named operation with some data values – arguments. An example is the sine function **'sin(x)'**, where 'sin' is the name of the function and 'x' is the argument. The function name determines what operation is performed with the arguments. Function's name can include alpha-numeric symbols, subscript and superscript digits and underscore symbol (the first symbol may be letter only)[4].

In addition to arguments, a function can have parameters. For an example, the logarithm of **'a'** to base **'b'** function $\log_b(a)$ has one argument **'a'** and one parameter **'b'**. Semantically parameters have the same meaning as the arguments have – data values on which the operation is performed. Syntactically (in ANALYTICS library) parameters are enclosed in braces **'{}'**.

General rules for functions:

- function arguments are always enclosed in parantheses **'()'**;
- function parameters are always enclosed in braces **'{}'**;
- if the function has no parameter, parameter braces are not obligatory, argument parantheses are always obligatory;
- function can have many parameters and/or many arguments;
- function arguments and parameters are separated by spaces ' ';
- there can be many functions with the same name and different number and/or type of arguments and/or parameters.

Some examples of syntactically correct function expressions:

max(a b) – maximum function of two arguments;
random() – random function with no arguments;
log{b}(a) – logarithm of **'a'** to base **'b'** (one parameter **'b'**, one argument **'a'**);
P{n m}(x) – the Associated Legendre function (two parameters **'n'** and **'m'** and one argument **'x'**).

The ANALYTICS library contains many predefined functions of real arguments. Functions for other argument types implemented in ANALYTICS extension libraries (see above).

Indexing

ANALYTICS library allows using **indexing** in expressions. Indexing is a method of access to the structured data elements. An example of structured data is array. Each array's element has the unique index by which the element value is accessed. By syntax rules data indexes are written in square brackets '[]'. For an example, the i-th array element can be written as **'A[i]'**. For multiple indexes, each index must be written in different brackets. For an example, a matrix element can be written as **'M[i][j]'**. Such syntax allows **slicing** implementation. Slicing allows to get a part of structured data. Slicing is implemented via index omitting. Let we have a matrix (two-dimensional array) **'M'**. Then **'M[i][]'** is the i-th matrix' row and **'M[][j]'** is the j-th matrix' column. The 'trail' indexes can be omitted with their brackets. Thus, the matrix' row can be written as **'M[i]'** (which is equivalent to **'M[i][]'** of the previous example).

General indexing rules:

- indexing can only be applied to variables, the variables must implement indexing interface (see below);
- many indexes allowed, each index must be enclosed in square braces **'[]'**;
- all index expressions must return values of real type only, the values must be (almost) integer;
- some indexes can be omitted that means slicing, slicing can only be applied to variables which implement slicing interface (see below);

ANALYTICS library contains predefined array variable types. Array variables can contain data of any type. Array variables have default indexing and slicing implementation for arrays up to third dimension.

Arrays

ANALYTICS library supports using **array** expressions. **Array** expression is a structured expression that contains other expressions as its components. An example of array expression is vector – one-dimensional set of ordered items. Array expressions has the dimension – the number of indexes by which the components of the array are ordered.

By syntax rules of ANALYTICS library, the array expression must be enclosed by square brackets '[]' and its components are separated by spaces. For example, the vector expression with three components can be written as **'[i+1 j*2 k]'**. For multiple dimensions, the components for each dimension must be written in separate brackets. For example, the 2x3 matrix expression can be written as **'[[a b c] [d e f]]'**. Only rectangular array expressions allowed.

General rules for array expressions:

- Array expressions enclosed with square brackets.
- Each dimension of array expression must be enclosed with its own brackets.
- Array expression's components are separated by spaces.
- Only rectangular array expressions allowed.

ANALYTICS library supports vector and matrix expressions for boolean, real, and complex components.

Syntax summary

- Any expression can contain literals, variables, operators, functions and indexing.
- Literals can be real numbers in simple and exponent form and complex numbers.

---

[4] Along imaginary symbol ***I*** may be used as function name, because function syntax can always be recognized by parantheses '()'.

- Variable names can include alpha-numeric symbols, subscript and superscript digits and underscore symbol.
- Only predefined (see above) operators can be used, new operators cannot be defined.
- Operations are performed in order of operator precedence. Operations order can be changed using parantheses '()'.
- Functions have parameters and arguments. Parameters are enclosed in braces '**{}**', arguments are enclosed in parantheses '()'. Parameters and arguments are separated by spaces ' '.
- Indexing can only be applied to variables (those implement indexing interface. Indexes are enclosed in brackets '[]' (each index in separate brackets). Slicing of indexed data can be implemented by omitting some indexes.
- Array expressions enclosed with square brackets for each dimension. Array components are separated by spaces. Only rectangular arrays allowed.

Some examples of syntactically correct formulae:

```
'sin(b)^2+cos(b)^2'                          - Pythagorean trigonometric identity formula.
'sin(a)*cos(b)+cos(a)*sin(b)'                - Sine of angle sum formula.
'log{c}(a)+log{c}(b)'                        - Logarithm of product formula.
'A[n]*r^n*sin(n*a)+B[n]*r^-n*cos(n*a)'       - Laplace's equation solution in polar coordinates.
'A[n]*sinh(n*Pi/L*(x-a))*sin(n*Pi/L*(y-b))'  - Laplace's equation solution in Cartesian system.
'[[a b -1] [0 c -c]]×[sin(a-1) e^b a*b]'     - Matrix-Vector multiplication.
```

Class Hierarchy

This part explains the base class hierarchy of ANALYTICS library. It is not needed to understand the hierarchy to use the library for common analytical calculations. The knowledge is only useful for extending library (introducing new function, overloading operators and so on).

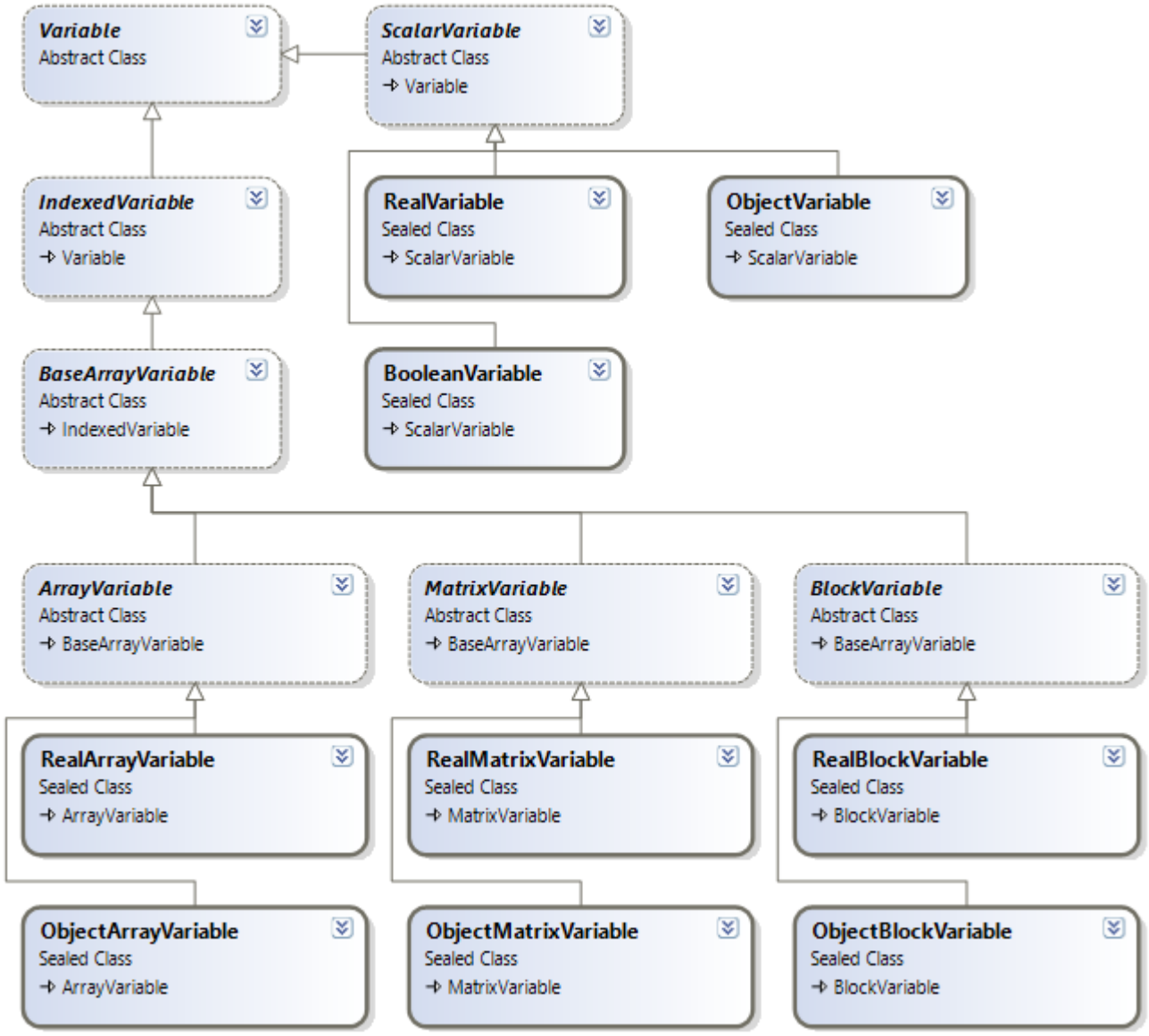
The main concept of ANALYTICS library is easy external extensibility. That is the library has the complete core which is not changed and new functionality can be implemented by (only) attaching external libraries. So there is strongly structured class hierarchy to implement this concept.

Variable classes

Variable is a name with associated value (of some type). The base abstract class ***Variable*** is intended to use this concept inside the program. The main properties of the class are: ***Name***, ***Value*** and ***ValueType***. There are two abstract classes, inherited from the ***Variable*** class. ***ScalarVariable*** class can contain one value. ***IndexedVariable*** class introduces interface for indexed value, that can contain another values accessed by indexes. The following classes inherited from ***ScalarVariable***: ***RealVariable*** (contains real value), ***BooleanVariable*** (contains boolean value), ***ObjectVariable*** (contains value of any type). ***BaseArrayVariable*** class is inherited from ***IndexedVariable*** and is abstract class for variables those contain arrays (of any dimension). Three abstract classes ***ArrayVariable***, ***MatrixVariable*** and ***BlockVariable*** realize interfaces for one-, two- and three-dimensional arrays accordingly. And finally, concrete classes of real and object arrays up to the third dimension are realized.

All concrete classes, introduced in this hierarchy, are fully functional. That is they realize all functionality to use them in calculations. All array variables realize indexing and slicing interfaces.

There are other variable types realized in extension libraries (complex variables, 3D vector and tensor variables and so on). They can be used in the same way as variables built in the ANALYTICS core library.



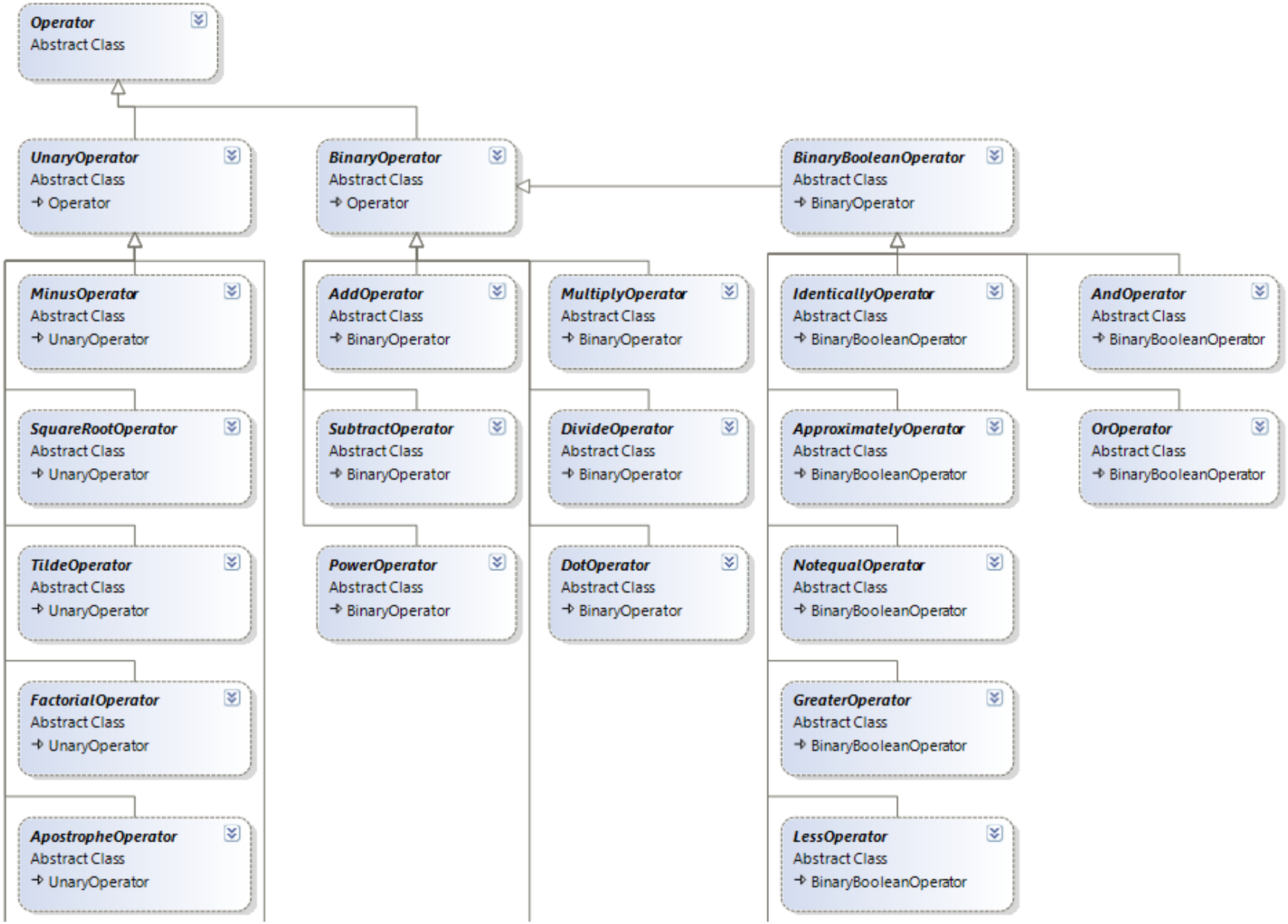
Picture 2.1. Variable class hierarchy diagram.

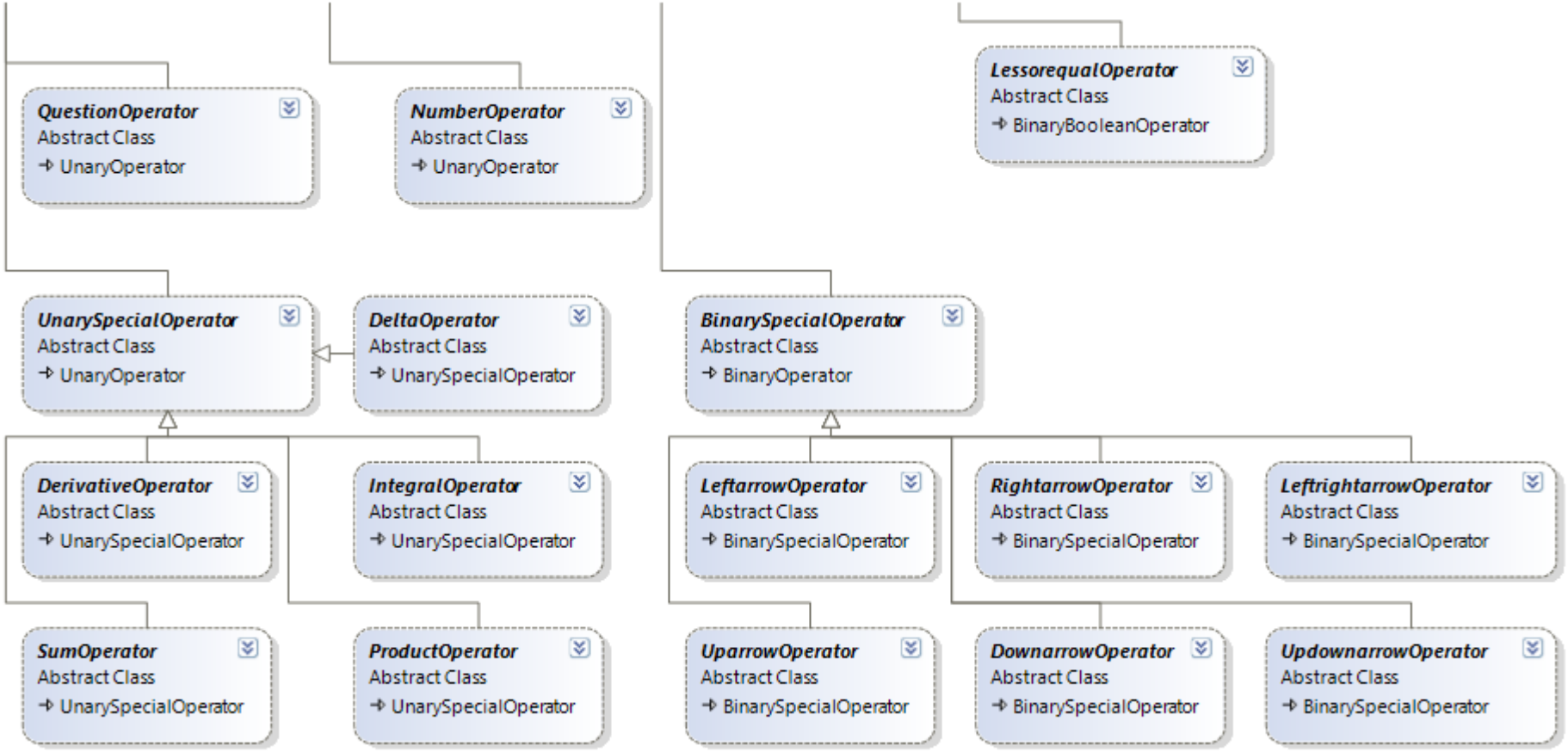
Operator classes

Base abstract class ***Operator*** is intended to use the concept of operator (symbol for some operation with operand values) inside the program. This class has properties and functions to define the symbol of the operator, the operation result type and the method to calculate value. In accordance with operator types two abstract classes are inherited from the base one: ***UnaryOperator*** and ***BinaryOperator***. These classes introduce interface to define operand type(s). More specific classes, inherited from the last ones, define the concrete operator type and symbol ('+', '^' and so on).

The concrete classes, those realize the complete functionality (define the operand types and perform operations on the operand's values) are not shown on the diagram. There are all realized operators for Real operands. Operators for other operand types (complex, 3D vectors and tensors and so on) realized in ANALYTICS extension libraries.

There are also **generic** analogues for all operator classes. For an example, generic analogue for ***AddOperator*** is ***GenericAddOperator*** class. These generic classes can be also used to derive new operator classes. The generic form of these classes simplifies inheritance because they already contain implementation of some methods (see operator overloading below).





Picture 2.2. Operator class hierarchy diagram.

Function classes

The base abstract class ***Function*** implements the concept of function. The class has interface to define the name, the count and type of parameters and arguments, the type of returned value and the method to make operation on the data values. The class ***MonotypeFunction*** is directly inherited from the ***Function*** and implements the concept of function such that all parameters and arguments are of the same type as the returned value. At the next level of hierarchy all functions are divided into ***Elementary*** and ***Special***. This division is to correspond with mathematics, not for programming convenience.

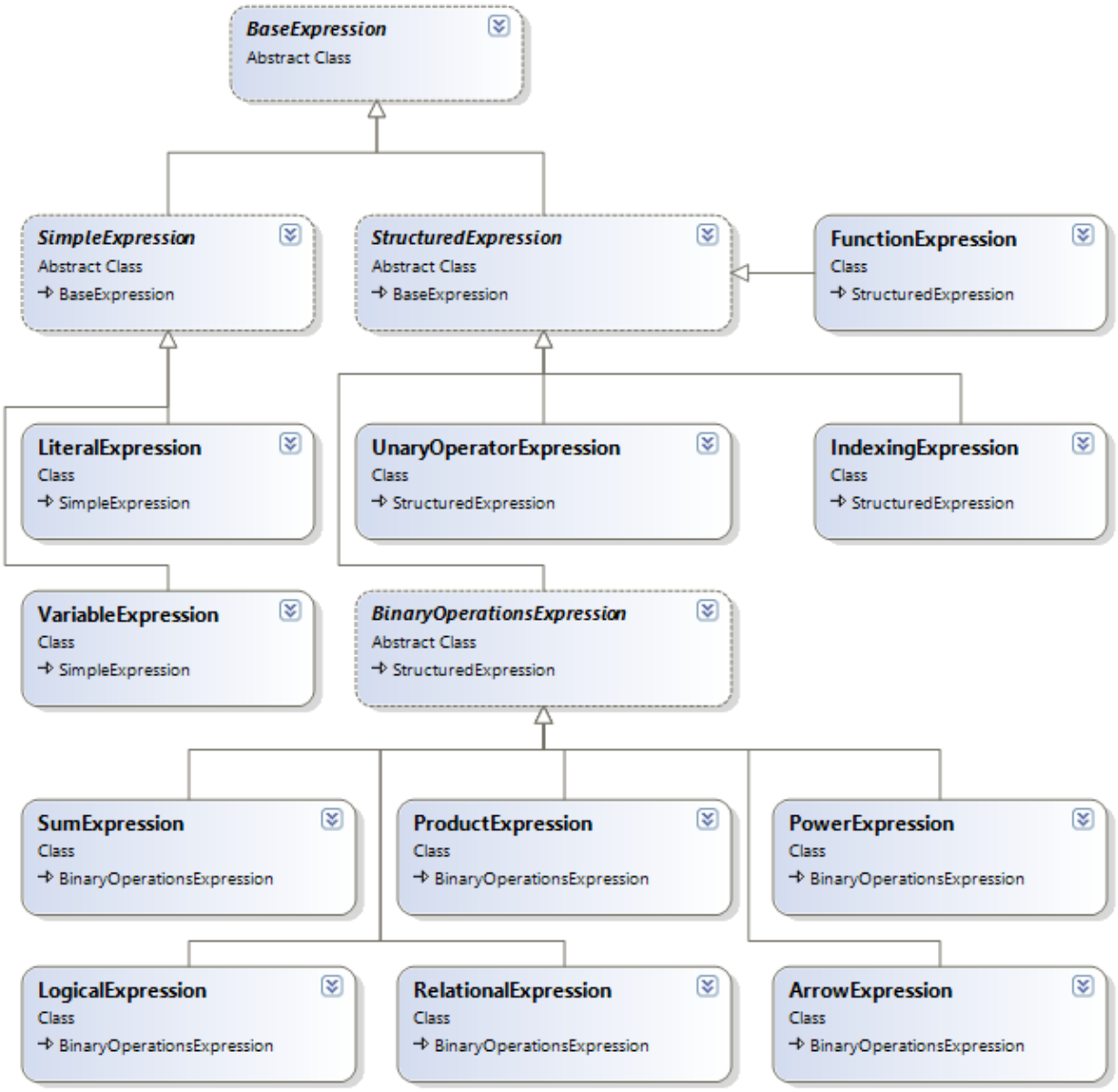
Picture 2.3. Function class hierarchy diagram.

Further, each of classes is divided into Real and Complex subclasses (functions with Real and Complex arguments accordingly) and finally all classes are divided into simple and Parametric functions. Simple functions have one argument, parametric – one parameter and one argument.

The concrete classes those realize the calculation are not shown on the diagram. There are many realized elementary and special functions of Real arguments – algebraic, trigonometric, inverse trigonometric, exponential, logarithmic, hyperbolic, inverse hyperbolic. Functions for other argument types (complex, 3D vectors and tensors and so on) implemented in ANALYTICS extension libraries (see above).

Expression classes

Expression classes are intended to represent data inside core algorithms of the ANALYTICS library. The base abstract class for all expressions is ***BaseExpression***.



Picture 2.4. Expression class hierarchy diagram.

This class defines abstract interface for all expressions and implements some static methods to manipulate with expressions (build expressions form strings, simplify expression list and so on). All other expressions are divided into **simple** (do not contain other expressions) and **structured** (do contain other expressions) ones. Simple expressions are ***LiteralExpression*** and ***VariableExpression***. Structured expressions include ***FunctionExpression***, ***IndexingExpression***, ***UnaryOperatorExpression*** and ***BinaryOperationsExpression*** (***LogicalExpression***, ***RelationalExpression***, ***SumExpression***, ***ProductExpression***, ***PowerExpression***, ***ArrowExpression***). All expression classes realize methods to manipulate with them: build expressions from strings, simplify expressions, reconstruct expressions (convert them back to the string), building new expressions from existing and others.

It is not obligatory to understand expression class functionality to use the ANALYTICS library. It is only needed to extend the functionality of the library in the sense of analytical derivative calculation (see below).

Working with ANALYTICS

The main functionality of ANALYTICS library (evaluation of mathematical expressions) is realized in ***Translator*** class. ***Translator*** class encapsulates a set of variables and a set of operations (registered operators and functions). Thus ***Translator*** can parse string expressions and calculate their values for current variable values.

Translator is not a static class because various translators can have different operation and variable sets. So to use ***Translator*** functionality an instance of the class must be created. Hereinafter it is supposed that the instance 'translator' of ***Translator*** class has been created

```
Translator translator = new Translator();
```

Working with variables

Translator class has complete interface to add and remove variables and to change their values.

To add variables use ***Add*** method. This method has many overloads to add variables of different types. The following code examples demonstrate the process of variable addition:

- adding Real variable

```
string name = "a";
double v = 1.0;
translator.Add(name, v);
```

- adding Complex variable

```
string name = "x";
Complex z = new Complex(1.0, -2.0);
Variable v = new ComplexVariable(name, z);
translator.Add(v);
```

- adding Real array variable

```
string name = "A";
double[] a = new double[] { 0.0, 1.0, 2.0, 3.0 };
Variable v = new RealArrayVariable(name, a);
translator.Add(v);
```

- adding variable of any type

```
string name = "A";
Complex[] a = new Complex[] { new Complex(0,0), new Complex(1,-1), new Complex(-2,2) };
Variable v = new ComplexArrayVariable(name, a);
translator.Add(v);
```

There are two ways to change variable values.

1. If the direct reference to a variable is available, the value can be changed using ***Value*** property of the variable class.

```
string name = "a";
double a = 1.0;
Variable v = new RealVariable(name, a);
translator.Add(v);
// some code here...
v.Value = (double)2.0;
```

2. If there is no direct reference to a variable, it can be got by using ***Get*** method of the ***Translator*** class (the name of variable or its index can be used).

```
Variable x = translator.Get("x");
x.Value = (double)3.0;
// some code here...
Variable y = translator.Get(1);
y.Value = (double)4.0;
```

NOTE about changing variable values: the type of the variable value cannot be changed. In spite of the property ***Value*** of the ***Variable*** class is of type ***Object***, when setting variable value its type must be the same as current ***ValueType***. The value type is determined at the variable creation and cannot be changed during variable lifetime.

The ***Delete*** method of ***Translator*** class can be used to remove variables. The variable can be removed from the translator instance by name or its index in the translator's table. The ***DeleteAll*** method removes all variables. The following code demonstrates the process of variable deleting.

```
translator.Delete("x");
// some code here...
translator.Delete(2);
// some code here...
translator.DeleteAll();
```

Calculating formula values

In the simplest case the ***Calculate*** method of ***Translator*** class can be used to calculate formula value. The following code demonstrates the method using

```
string f1 = "sin(a)^2+cos(a)^2";
double r = (double)translator.Calculate(f1);
// some code here...
string f2 = "2*exp(z)-I*sin(z/3)";
Complex c = (Complex)translator.Calculate(f2);
```

In the code above it is supposed that there are variables "a" and "z" in translator's variable set and "a" is of Real type and "z" is a complex one. Note, that the return type of the ***Calculate*** method is ***Object*** and it can return the value of any type depending on the formula contents. So, the returned value must be directly casted to the type and the type must be known.

The ***Calculate*** method is rather slow. This is because it parses string expression and creates internal structure to calculate result value. Parsing methods are not optimized for speed, they are optimized for strong object oriented structuring and easy extensibility. Thus the usage of ***Calculate*** method is only recommended for single formula evaluation.

Another case of formula evaluation comes from the need to calculate one expression for various variable values. This can be done in such a way:

- creating ***Formula*** object from string expression;
- changing variable values;
- calculation formula values for current variable values.

Formula is a class intended for internal program representation of parsed mathematical expressions. The following code demonstrates calculation of the table of function values for various argument values by the above algorithm.

```
string s = "2*(sin(x)+cos(x))";
Formula f = translator.BuildFormula(s); // parsing string expression
Variable x = translator.Get("x");
double v = 0.0;
double[] ax = new double[101];
double[] ay = new double[101];
for (int i = 0; i <= 100; i++)
{
        ax[i] = v;
        x.Value = v; // setting new variable value
        ay[i] = (double)f.Calculate(); // calculating formula value for current x value
        v += 0.01;
}
```

In the code, the string expression is parsed only once by the ***BuildFormula*** function which returns the ***Formula*** object. When the ***Formula*** instance created, its value can be calculated many times for various 'x' variable values.

Checking expression syntax

In all above code examples it was supposed that the string expressions were syntactically correct. End user applications, of course, must check the syntax correctness of user defined expressions. The ***Translator*** class provides methods to check expressions before calculation.

The syntax correctness must be checked in three steps. The first, syntax rules, those do not need the expression to be parsed, must be checked. For an example, the parantheses in a mathematical expression must be pairwise and it can be checked without expression decomposition. Such syntax rules must be checked before any calculation by ***Translator***'s ***CheckSyntax*** method. This function returns true, if all rules are fulfilled and throws an exception if not.

The second step is to check, that a string can be decomposed into known expression types. For an example, the string 'sin(x+1)' can be decomposed as a function with name 'sin' and one argument 'x+1' which is itself the known expression – the sum of constant '1' and variable with name 'x'. This step does not need to know that the function 'sin' is defined or the variable 'x' exists.

The third step checks that all elements in decomposed expression are defined. It means that all variables must exist. Moreover, this step needs to know the types of expression results to check that all operations in the expression can be performed. For an example, in given expression, if variable 'x' is Real, the operator '+' must be defined for Real operands and the function 'sin' with one Real argument must be registered.

The second and the third steps of syntax checking are implemented in ***Translator***'s ***BuildFormula*** method. This method returns built ***Formula*** object, if the string expression is correct, and throws an exception if not.

The following example code demonstrates common syntax checking algorithm:

```
string s = "2*(sin(x)+cos(x))";
try
{
        // the first step of syntax checking
        if (translator.CheckSyntax(s))
        {
                // the second and third steps of syntax checking
                Formula f = translator.BuildFormula(s);
                if (f != null)
                {
                        // here the formula calculation code
                        // using f instance.
                }
        }
```

```
}
catch (Exception ex)
{
        // here the exception handling code.
}
```

Analytical derivative calculation

The ANALYTICS library allows to calculate derivatives of expressions. The ***Translator*** class (see above) contains the following method:

```
public string Derivative(string formula, string vName)
```

This method calculates analytical derivative of the *formula* by variable *vName*. The result is string representing the derivative expression.
Here are some example codes of analytical derivative calculation:

```
string formula;
string derivative;
// #1
formula = "A*ln(x)*sin(2*x)";
derivative = translator.Derivative(formula, "x");
// derivative = (1/x)*(A*sin(2*x))+(cos(2*x)*2)*(A*ln(x))
// #2
formula = "e^(1-2*x)";
derivative = translator.Derivative(formula, "x");
// derivative = e^(1-2*x)*(-2)
// #3
formula = "(x+1)^(x-1)";
derivative = translator.Derivative(formula, "x");
// derivative = ln(x+1)*(x+1)^(x-1)+(x-1)*(x+1)^((x-1)-1)
```

The examples show that the calculation is very simple.
Notes about derivative calculations:

1. The *formula* parameter in ***Derivative*** method must be syntactically correct. The correctness can be checked by ***Translator***'s ***CheckSyntax*** method.
2. There is no need variable with the name *vName* (or any other) to be existing. The ***Derivative*** method manipulates variable names only, not their values.
3. The derivative method does some simplification after calculation (removes zero values from sums, unit multipliers from products and so on). But some results can be not 'beautiful to see'. They can contain some superfluous braces and other 'artifacts'. It is because the ANALYTICS library is not computer algebra system. The derivative calculation result is intended to use in the subsequent calculations (inside the program), not to show it to user.
4. Not all operators supported for derivative calculations. For an example, '**!**' operator not supported, because the derivation rule is not defined for it. It is not possible to introduce new derivation rules for operators (without rewriting core algorithms).
5. Most of standard transcendental functions supported for derivative calculation. Derivation rule can be defined for any function (see below).

Symbolic expression simpification

Another feature of ANALYTICS library is symbolic expression simplification. The Translator class contains the following method:

```
public string Simplify(string value)
```

This method simplifies the input symbolic expression and returns the simplified value as the result. The following main simplifications supported:

- Remove all zero operands from sum expressions and all unit operands from product and power expressions.
- Reduce minus operation pairs.
- Reduce same expression operands in numerator and denominator.
- Collect same expression operands in sum.
- Reduce real (nonstandard) constants in sum expressions.
- Reduce real (nonstandard) constants in product expressions.
- Reduce real (nonstandard) constants in power expressions.

Other auxiliary simplifications implemented for rare, nonstandard cases.
Here are some example codes of analytical expression simplifications:

```
…

// #1
formula = "(A-13)*(27/4)*(-5e-1)/(9/5)/x";
simplified = translator.Simplify(formula);
// simplified = -15/8*(A-13)/x

…
```

```
// #2
formula = "2*(x-1)*(3/(2*(x-1)))";
simplified = translator.Simplify(formula);
// simplified = 3

…

// #3
formula = "A*sin(Pi+1-(x-1))+B*ln(6*e/(3*x))";
simplified = translator.Simplify(formula);
// simplified = A*sin(π+2-x)+B*ln(2*e/x)
…
```

NOTE: the standard constants such as the Euler number and Pi remains named literals and not applied for simplification process.

Implicit operations in symbolic expressions

Symbolic expressions in ANALYTICS library can contain implicit operations. Such operations suppose 'implicit' manipulation with the expression, before it is evaluated or other operation implemented. Examples of implicit operation are derivative and indefinite integral.

Implicit derivative operator has the following syntax:

∂<supescript>expression/***∂***V1<supescript>***∂***V2<supescript>…***∂***Vn<supescript>

where '***∂***' is the derivative operator symbol; superscript is a superscript digit, specifying the order of differentiation; <> means that the element is not obligatory; expression – symbolic expression, operand of the differentiation operation; '**/**' – division operator symbol; V1, V2, .. Vn – names of the variables for differentiation.

Implicit derivative operator implements partial differentiation of specified orders on the operand by specified variables. The order of the differentiation defined by one digit only, so the orders supported are 1..9. The total order of the derivative (superscript before the operand expression) must be equal to the sum of the orders of variables.

NOTE: Implicit derivative operator has precedence, equal to the **precedence of the division operation**, so, it must be enclosed with parentheses where required by syntax rules.

There are some examples of the syntactically correct expressions with the implicit derivative operator:

1. ∂sin(x)/∂x
2. ∂²(x^2*sin(a*x))/∂x²
3. ∂²(y^2+1-e^-x)/∂x∂y
4. ∂sinh(x*y)/∂x-∂ln(y/x)/∂y
5. (∂³F/∂x³)/3!*x^3

Symbolic expressions with implicit operations can be used as other expressions: evaluated, simplified, calculating symbolic derivative and so on. Evaluation of the expressions done for the result expression, after the implicit operation. Here is an example code for calculating value of the expression, containing implicit derivative operator:

```
string f = "e^(b*x)*(∂y^2/∂y)-∂sin(a*x)/∂x";
object v = translator.Calculate(f);
// code for using 'v' value...
```

The code is the same as for calculating values of any other expression – we do not need to calculate symbolic expressions for derivatives explicitly and substitute them into final expression.

The advantage of using implicit operation is that there is no need to do the operation before the evaluation – it is done **implicitly** by the evaluation system. The implicit operations allows write some mathematical formulae in shorter and compact way, without including in the formula 'explicit' results of the implicit operation.

In the case where explicit expression required, there is the '***Explicit***' method of the ***Translator*** class for evaluating the symbolic expression:

```
public string Explicit(string value)
```

The method evaluates all implicit operations in the 'value' expression and returns explicit symbolic expression. The code for the case is the following:

```
string f = "e^(b*x)*(∂y^2/∂y)-∂sin(a*x)/∂x";
string ef = translator.Explicit(f);
// code for using explicit symbolic expression 'ef'...
```

The result explicit expression for the example is **'e^(b*x)*(2*y)-cos(a*x)*a'**.

NOTE: for getting explicit symbolic expression there is no need the variables in the expression being added to the ***Translator***, the '***Explicit***' method works with symbols, not with values. The '***Calculate***' method requires the variables, because it evaluates values of the expressions.

Applying the '***Derivative***' method of the ***Translator*** class for an expression with implicit derivative operators implemented with some features. Let us consider an example of evaluating symbolic derivative for two expressions with implicit derivative operators:

```
string f1 = "a*x^2+∂²(sin(a*x)*e^(b*y))/∂x∂y";
string df1 = translator.Derivative(f1, "x");

string f2:= "a*x^2+∂²(sin(a*x)*e^(b*y))/∂a∂b";
string df2:= translator.Derivative(f2, "x");

// code for using symbolic derivatives 'df1' and 'df2'...
```

Result symbolic expressions for derivatives are the following df1='**a*2*x+∂³(sin(a*x)*e^(b*y))/∂x²∂y**' and df2='**a*2*x+∂²(cos(a*x)*a*e^(b*y))/∂a∂b**'. As can be seen from the result expressions, derivative operation applied for implicit derivative with the following rules:

- If the implicit operation contains derivative by the same variable – just derivative order in the implicit operation increases.
- If the implicit operation does not contain derivative by the same variable – implicit operation remains the same and the derivative evaluated for the operand (by the rules of analysis, derivative operations are distributive and their order of application can be exchanged).

Functional operators

Functional operators are like 'user defined' functions. They allow defining an expression of the function, depending on some variables, and then using the name of the function in different symbolic expressions with different arguments. For example, one can define the functional operator for general square polynomial: **F(x a b c)=a*x^2+b*x+c** and then use the operator in expressions with different arguments **F(sin(x) a b -1)**, **F(y 2 -4 0)**. The former function is equivalent to '**a*sin(x)^2+b*sin(x)-1**' and the latter gives '**2*y^2-4*y**'. Functional operators can be used to manipulate easily with some predefined, frequently used expressions.

For using functional operators, they must be added to the translator first. This can be done with the method '***AddFunction***' of the ***Translator*** class:

```
public bool AddFunction(string aName, string f, string[] vars)
```

where '**aName**' is the name of the function, used in expressions, must be syntactically valid name; '**f**' is the formula of the function; '**vars**' is the array of variable's names the function depends on (arguments).

Here is the example code of defining some functional operators:

```
translator.AddFunction("F", "x^2+1", new string[] {"x"});
translator.AddFunction("G", "2-sin(y)", new string[] {"y"});
translator.AddFunction("SH", "(e^x-e^-x)/2", new string[] {"x"});
translator.AddFunction("CH", "(e^x+e^-x)/2", new string[] {"x"});
translator.AddFunction("SQRE", "a*x^2+b*x+c", new string[] {"a", "b", "c", "x"});
```

Rules for adding functional operators:

- Any functional operator must have unique name (functional operators with the same name and with different number of variables not allowed).
- Variables of a functional operator must have unique (for the operator) and valid names.
- Functional operator formula must be syntactically correct expression and can contain literals, variables, operators, functions (including other functional operators). The formula can contain as functional variables as 'free' variable, not in the variable names. The variables in the formula have not to exist in the ***Translator*** – the expression manipulates with symbolic data, not with the values.

Functional operators can be deleted with the '***RemoveFunction***' method of the ***Translator*** class using the name of the function.

Functional operators used in symbolic expressions as other functions – using the name of the function with arguments in parentheses. Functional operators do not allow parameter interface, only arguments. When value of an expression with functional operators evaluated - the actual arguments substituted to the formula and the final value calculated.

Here is an example code of evaluating simple formula with functional operators:

```
translator.AddFunction("S", "Pi*r^2", new string[]{"r"});
string f = "S(a)-S(a/2)";
object v = translator.Calculate(f);
// using the 'v' value…
```

First we introduced the function '**S**' with one argument '**r**' which calculates area of a circle with specified radius value '**Pi*r^2**'. Then we used the function in the formula to calculate area of a ring with outer radius '**a**' and inner radius '**a/2**'. The example shows that functional operators are useful for defining functions that used frequently in expressions with different arguments.

Functional operators can be used with any other expressions, including implicit operations (derivatives). One allowed defining functional operators with functional arguments and so on. Here is the code example of a more complicated use-case of functional operators:

```
translator.AddFunction("Series",
"1+(∂F/∂x)*x+(∂²F/∂x²)/2!*x^2+(∂³F/∂x³)/3!*x^3+(∂⁴F/∂x⁴)/4!*x^4", new string[] { "F", "x" });
string f = "Series(e^y y)";
object v = translator.Calculate(f);
// using the 'v' value…
```

The introduced function **'Series'** is the Tailor's series with degrees up to fourth. The functional operator has two arguments – function **'F'** and variable **'x'**. When we use the functional operator in expressions, we can call it with different arguments. The first argument must be a function or an expression (**'e^y'** in the example code). It is used to evaluate implicit derivatives in the series formula. The second argument must be a variable name (**'y'** in the example code) because it is used in the denominator of the implicit derivative expressions.

The substitution of actual arguments to the functional operators can be done with the **'Explicit'** method (see **'Implicit operations in symbolic expressions'**). Let us consider the following code:

```
translator.AddFunction("Laplace", "∂²F/∂x²+∂²F/∂y²", new string[] { "F", "x", "y" }); // 1
string f = "Laplace(A*sin(x)*y^2+B*cos(x)*e^-y x y)";
string ef = translator.Explicit(f); // 2
ef = translator.Simplify(ef); // 3
// using the 'v' value…
```

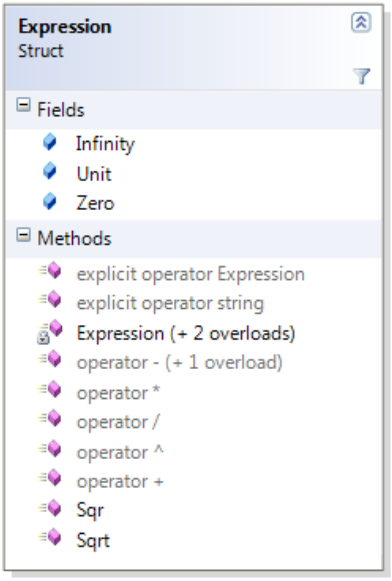
Here we introduced the Laplace's operator with two variables **'x'** and **'y'** (line 1) and then used the operator for evaluating symbolic formula for some expression (line 2). For the expression in the example **'A*sin(x)*y^2+B*cos(x)*e^-y'** we get expression for **'ef'**: **'(-A*sin(x)*y^2-B*cos(x)*e^-y)+(A*sin(x)*2+B*cos(x)*e^-y)'**. That is the ***'Explicit'*** method substituted the expression into Laplace's formula and evaluated symbolic derivatives. Then the symbolic expression was simplified (line 3) and the final expression is: **'-A*sin(x)*y^2+A*sin(x)*2'**. Note that there is no **'B'** variable in the final expression because the Laplacian for the **'B*cos(x)*e^-y'** is 0.

Explicit string expressions manipulation

The ANALYTICS library allows 'explicit' manipulations with 'string expressions'. Let there are two strings inside the program, containing mathematical expressions **'1-2*x'** and **'sin(x)+2'**. And let there is need to get the expression which is the multiplication of the given ones. This task can be done just manipulating with C# strings. For the above example strings the following steps must be done: enclose the first expression into parentheses, enclose the second expression into parentheses and finally concatenate the result strings with the '*' operator. The algorithm seems to be simple, but it becomes more and more complicated with the number of operations increase.

The ***Expression*** structure (do not confuse with the ***BaseExpression*** from class hierarchy) simplifies such string manipulations. This structure type is just 'simple string wrapper' (pic. 3.1). It contains explicit conversion operators – from string and to string and overloaded operators implementing algebraic operations **'+'**, **'-'**, **'*'**, **'/'**, **'^'** (power). This implementation allows manipulating string expressions in 'natural' manner.

Consider an example of using the ***Expression*** structure for the following task: realize 3D 'analytical' vector. This vector must contain three string components, all components must be mathematical expressions and all vector manipulations (sum, vector product, length and so on) must be analytical.



Picture 3.1. Expression class.

The following code is the (partial) implementation of 'analytical' 3D vector:

```
public struct StringVector
{
```

```
        #region Data members
        private Expression e1;
        private Expression e2;
        private Expression e3;
        #endregion Data members

        /// <summary>
        /// Constructor.
        /// </summary>
        /// <param name="x1"></param>
        /// <param name="x2"></param>
        /// <param name="x3"></param>
        public StringVector(string x1, string x2, string x3)
        {
            e1 = (Expression)x1;
            e2 = (Expression)x2;
            e3 = (Expression)x3;
        }

        /// <summary>
        /// Dot Product
        /// </summary>
        /// <param name="v1"></param>
        /// <param name="v2"></param>
        /// <returns></returns>
        public static string Dot(StringVector v1, StringVector v2)
        {
            return (string)(v1.e1 * v2.e1 + v1.e2 * v2.e2 + v1.e3 * v2.e3);
        }

        /// <summary>
        /// Cross Product
        /// </summary>
        /// <param name="v1"></param>
        /// <param name="v2"></param>
        /// <returns></returns>
        public static StringVector operator *(StringVector v1, StringVector v2)
        {
            return new StringVector(
                (v1.e2 * v2.e3 - v1.e3 * v2.e2),
                (v1.e3 * v2.e1 - v1.e1 * v2.e3),
                (v1.e1 * v2.e2 - v1.e2 * v2.e1)
            );
        }
    }
```

The ***StringVector*** structure contains three data members e1, e2, e3 of type ***Expression*** to encapsulate data for three vector components. The constructor takes three string arguments, which are converted to ***Expression*** data using explicit operator.

Main advantages of using ***Expression*** structures are demonstrated in ***Dot*** and ***Cross*** methods. The calculation codes of vector dot and cross products are **similar** to if the vectors have ***double*** components.

Consider the result of using StringVector structure:

```
StringVector x1 = new StringVector("sin(u)", "cos(v)", "0");
StringVector x2 = new StringVector("cos(u)", "-sin(v)", "0");
StringVector r =  x1 * x2;
```

The result vector r will contain three expression components '0', '0' and 'sin(u)*(-sin(v))-cos(v)*cos(u))'.

NOTE about using ***Expression*** structure: all string values to manipulate with ***Expression*** structure must be syntactically correct; else, the exception will be thrown. The syntax can be checked by ***Translator***'s ***CheckSyntax*** method (see above).

Extending ANALYTICS

The ANALYTICS library provides complete core for parsing and calculating mathematical expressions. It also contains many predefined 'standard' functions – trigonometric, hyperbolic and so on and realized operators for real arguments. So, the library can be used without modification to provide in the end user program the interface for user to input data in the form of mathematical expressions.

Another goal of the library is to provide the possibility of mathematical calculations with the program specific data. For an example, let there is a program for signal processing. And the program must allow the end user to make various operations with signals – add two signals, multiply a signal by real values, calculate exponents and logarithms of signals and so on. This can be done using the ANALYTICS library. The core algorithms of ANALYTICS library can be used to calculate expressions containing signals as variables. The only thing required is to provide operations, defined for signals (because they are program specific data).

The ANALYTICS library is built by such technology that allows easily introduce operations with program specific data without changing core algorithms. All data operations (operators, functions) are presented inside the library as classes. To add an operation with program specific

data a descendant of some class must be implemented. This class will be automatically found and used to calculate expression results.

The next part explains implementation of classes to define operations with program specific data.

Overloading operators

To overload an operator for program specific data operand types the descendant of one of the defined operator classes must be implemented. There are base abstract classes for all defined operators (see class hierarchy): AddOperator, SubtractOperator and so on. The following code demonstrates the operator overloading for arrays of real numbers:

```
/// <summary>
/// (Double Array) * (Double) = (Double Array)
/// </summary>
public sealed class ArrayRealMultiply : MultiplyOperator
{
    protected override Type GetOperand1Type()
    {
        return typeof(double[]);
    }

    protected override Type GetOperand2Type()
    {
        return typeof(double);
    }

    protected override Type GetReturnType()
    {
        return typeof(double[]);
    }

    protected override object Operation(object operand1, object operand2)
    {
        if (operand1 == null) return null;

        double[] a = (double[])operand1;
        double r = (double)operand2;

        int l = a.Length;
        double[] result = new double[l];

        for (int i = 0; i < l; i++)
        {
            result[i] = a[i] * r;
        }

        return result;
    }
}
```

The ***ArrayRealMultiply*** class is inherited from ***MultiplyOperator***, this means it overloads "*" operator. The class overrides ***GetOperand1Type***, ***GetOperand2Type*** and ***GetReturnType*** methods and defines that the first operand is of type array of doubles and the second is of type double, the operation's result is double array. Another overridden method is ***Operation*** – this method implements the action that must be performed on operands.

The following example demonstrates overloading of "+" operator for arrays of real values:

```
/// <summary>
/// (Double Array) + (Double Array) = (Double Array)
/// </summary>
public sealed class ArrayAdd : AddOperator
{
    protected override Type GetOperand1Type()
    {
        return typeof(double[]);
    }

    protected override Type GetOperand2Type()
    {
        return typeof(double[]);
    }

    protected override Type GetReturnType()
    {
        return typeof(double[]);
    }

    protected override object Operation(object operand1, object operand2)
    {
        if (operand1 == null || operand2 == null) return null;
```

```
            double[] a1 = (double[])operand1;
            double[] a2 = (double[])operand2;

            int l = a1.Length;
            double[] result = new double[l];

            for (int i = 0; i < l; i++)
            {
                result[i] = a1[i] + a2[i];
            }

            return result;
        }
    }
```

The classes of operators will be automatically found by ANALYTICS library and used to implement operations with operands of defined types.

The same result, as in two previous code examples, can be achieved by using generic analogues of base operators. The following code demonstrates this case:

```
    /// <summary>
    /// (Double Array) * (Double) = (Double Array)
    /// </summary>
    public sealed class ArrayRealMultiply : GenericMultiplyOperator<double[], double, double[]>
    {
        protected override double[] TypedOperation(double[] operand1, double operand2)
        {
            if (operand1 == null) return null;

            int l = operand1.Length;
            double[] result = new double[l];

            for (int i = 0; i < l; i++)
            {
                result[i] = operand1[i] * operand2;
            }

            return result;
        }
    }

    /// <summary>
    /// (Double Array) + (Double Array) = (Double Array)
    /// </summary>
    public sealed class ArrayAdd : GenericAddOperator<double[], double[], double[]>
    {
        protected override double[] TypedOperation(double[] operand1, double[] operand2)
        {
            if (operand1 == null || operand2 == null) return null;

            int l = operand1.Length;
            double[] result = new double[l];

            for (int i = 0; i < l; i++)
            {
                result[i] = operand1[i] + operand2[i];
            }

            return result;
        }
    }
```

Using generic base operators is "shorter" because only one method (***TypedOperation***) must be overridden. Operand and return types are defined as parameters of the generic class the operator is inherited from. Both inheritance cases, generic and not generic, absolutely equivalent for ANALYTICS core algorithms.

Explicitly overloaded operators

The operator overloading in ANALYTICS library is intended to implement operations on program specific type data. Often, for such data, some operators are already overridden "inside" the program (that means C# operator overloading). And these "explicitly overloaded operators" can be used to implement operations on such data inside ANALYTICS core algorithms. In other words, if there is an operator "explicitly overloaded" it will be automatically found and used to calculate result for the suitable operation. For an example, as ***Complex*** type overloads math operators, such actions as addition, subtraction and so on can be performed for complex numbers without deriving new operator classes.

There are some constraints for using "explicitly overloaded" operators:

1. Not all operators can be "explicitly overloaded", because not all ANALYTICS operators have analogous C# operators or their "meaning" can be ambiguous ("^" for an example). So, only following operators can be "explicitly overloaded": "**+**", "**-**" (unary and binary), "*****", "**/**", '≡' (==)[5], '≠' (!=), '>', '<', '≥' (>=), '≤' (<=), '&', '\' (|), '¬' (!).
2. "Implicit" overloading (deriving new operator classes) has higher priority. That is if there are both "explicit" and "implicit" operators – the last will be used to calculate operation result.

Both overloading methods can be used in combination. "Implicit" overloading (using new operator classes) can be used as for complimenting "explicit" method, as for overriding its behavior.

Introducing new functions

Implementing functions for program specific data is more general way of extending library functionality. A Function can have any valid name, any number of arguments of different types and any return type. To introduce new function a descendant of one of abstract function classes must be implemented. The choice of a base class is wider, than for operators (see class hierarchy). For common argument types (real or complex), one of the predefined abstract classes can be used as ancestor class: ***RealElementaryFunction***, ***RealParametricElementaryFunction***, ***ComplexElementaryFunction***, ***ComplexParametricElementaryFunction***. As an axample, the code of class, implementing sine function of complex argument, is listed below:

```
/// <summary>
/// Sine function of Complex Argument
/// </summary>
public sealed class SineComplex : ComplexElementaryFunction
{
    protected override string GetName()
    {
        return "sin";
    }

    protected override Complex Func(Complex x)
    {
        return Complex.Sin(x);
    }
}
```

The class is inherited from ComplexElementaryFunction because it implements a function with one argument of Complex type. The only methods overridden are GetName and Func. The former defines the function name (used in expressions) and the later implements the function operation itself. Another example demonstrates the logarithm function implementation of complex argument and base:

```
/// <summary>
/// Logarithm function of Complex argument
/// (NOTE: the Base of logarithm is the Parameter of the function)
/// </summary>
public sealed class LogarithmComplex : ComplexParametricElementaryFunction
{
    protected override string GetName()
    {
        return "log";
    }

    protected override Complex Func(Complex parameter, Complex argument)
```

[5] C# analogue operators shown in parentheses.

```
        {
            return Complex.Log(argument)/Complex.Log(parameter);
        }
    }
```

The class is inherited from ComplexParametricElementaryFunction because it implements a function with one parameter and one argument of Complex type.

In general case, to introduce new function the class must be inherited from the base abstract class ***Function***. All abstract methods of the class must be overridden. The methods must define the number and types of parameters and arguments and the function return type. The following code example demonstrates the implementation of Min function that calculates the minimum value in an array of real numbers:

```
    /// <summary>
    /// Min array element
    /// </summary>
    public sealed class MinArray : Function
    {
        protected override string GetName()
        {
            return "Min";
        }

        protected override int GetArgumentCount()
        {
            return 1;
        }

        protected override Type[] GetArgumentTypes()
        {
            return new Type[] { typeof(double[]) };
        }

        protected override int GetParameterCount()
        {
            return 0;
        }

        protected override Type[] GetParameterTypes()
        {
            return null;
        }

        protected override Type GetResultType()
        {
            return typeof(double);
        }

        protected override object DoCalculate(object[] parameters, object[] arguments)
        {
            double[] a = (double[])arguments[0];
            if (a == null || a.Length == 0) return double.NaN;

            double result = a[0];
            int l = a.Length;
            for (int i = 1; i < l; i++)
                if (a[i] < result) result = a[i];

            return result;
        }
    }
```

The function has one argument of type real array and returns real value.

A little shorter way to introduce new function for the specific types is using base generic function classes. These classes use generic parameters to define the types of arguments. Thus, only Name and calculation algorithm must be provided to define new function class. The following code demonstrates the previous example function implementation using base generic class:

```
    /// <summary>
    /// Min array element
    /// </summary>
    public sealed class MinArray : GenericSimpleFunction<double[], double>
    {
        protected override string GetName()
        {
            return "Min";
        }

        protected override double Func(double[] a)
```

```
        {
            if (a == null || a.Length == 0) return double.NaN;

            double result = a[0];
            int l = a.Length;
            for (int i = 1; i < l; i++)
                if (a[i] < result) result = a[i];

            return result;
        }
    }
```

The class is inherited from ***GenericSimpleFunction***, which means it has one argument and zero parameters. The argument type is real array and the function's result is real which is provided by the first and the second generic parameters accordingly.

The ANALYTICS library contains base generic function classes to implement functions with up to two parameters and two arguments of any type.

NOTE about functions: there can be many functions with the same name but different count or/and type of arguments (parameters).

Implementing indexing

Indexing can be applied for variables only. To implement indexing for program specific data new variable class must be inherited from the abstract class ***IndexedVariable*** (or from one of its descendants). The inherited class must override methods defining the number of data indexes, data access methods and slicing implementation.

As an example of indexing implementation, the (partial) code of standard ***MatrixVariable*** class is listed below:

```
    /// <summary>
    /// Base abstract class for all matrix variables.
    /// NOTE: Slicing is implemented.
    /// </summary>
    public abstract class MatrixVariable : BaseArrayVariable
    {
        /// <summary>
        /// 2 indexes
        /// </summary>
        /// <returns></returns>
        protected override sealed int GetIndexCount()
        {
            return 2;
        }

        /// <summary>
        /// Slicing is implemented for Matrix variables
        /// </summary>
        /// <returns></returns>
        protected override bool GetSlicingImplemented()
        {
            return true;
        }

        /// <summary>
        /// Sliced item is array of BaseType
        /// </summary>
        /// <param name="indexes"></param>
        /// <returns></returns>
        public override Type GetItemType(int[] indexes)
        {
            if ( (indexes[0] >= 0 && indexes[1] < 0)
                || (indexes[0] < 0 && indexes[1] >=0) )
            {
                return BaseType.MakeArrayType();
            }

            return BaseType;
        }

        /// <summary>
        /// Implements Array Slicing
        /// </summary>
        /// <param name="indexes"></param>
        /// <returns></returns>
        public override object GetItemValue(int[] indexes)
        {
            // Row
            if (indexes[0] >= 0 && indexes[1] < 0)
            {
                int len = ColumnCount;
```

```
                Array result = Array.CreateInstance(BaseType, len);
                int[] row = new int[] { indexes[0], 0 };

                for (int j = 0; j < len; j++)
                {
                    row[1] = j;
                    object x = data.GetValue(row);
                    result.SetValue(x, j);
                }

                return result;
            }
            else
            {
                // Column
                if (indexes[0] < 0 && indexes[1] >= 0)
                {
                    int len = RowCount;
                    Array result = Array.CreateInstance(BaseType, len);
                    int[] column = new int[] { 0, indexes[1] };

                    for (int i = 0; i < len; i++)
                    {
                        column[0] = i;
                        object x = data.GetValue(column);
                        result.SetValue(x, i);
                    }

                    return result;
                }
            }

            // Item
            return base.GetItemValue(indexes);
        }
    }
```

The class overrides GetIndexCount method which returns 2, because matrix element has two indexes. The method ***GetSlicingImplemented*** returns true, that means the variable supports slicing. ***GetItemType*** method returns the type of indexed data, taking into account slicing implementation. If one of indexes is less than zero it means that the data must be 'sliced' by this index. For matrix data it means that the returned data is array (matrix row or column). The ***GetItemValue*** method implements access to data implementing slicing analogously to ***GetItemType*** method.

NOTE about indexing implementation: index type is always supposed to be integer and cannot be redefined.

Introducing function derivatives

Extending ANALYTICS library in the sense of derivative calculation differs from the other extensions. For example, the derivative rules cannot be redefined for operators, because these rules are the base of the mathematics.

The only derivative definition allowed is the function derivative. To define a function derivative the descendant of ***FunctionalDerivative*** class must be derived and its abstract methods must be implemented. The methods are:

```
protected abstract string GetFunctionName(); - name of the function.
protected abstract int GetParameterCount();  - the number of function's parameters.
protected abstract int GetArgumentCount(); - the number of function's arguments.
public abstract BaseExpression Derivative(DerivativeContext context, FunctionExpression
function, string vName);
```

The first three methods define what function the derivative rule is applied for. Note that the types of parameters and arguments are not defined. It differs from defining function for calculation because derivative calculation does not depend on the types, but calculation of functions does.

The last method defines the rule itself. This method takes *function* parameter of ***FunctionExpression*** class (see class hierarchy above) and *vName* parameter – the name of variable for derivative. The *context* parameter is only needed to pass into other derivative methods. The result of the method is ***BaseExpression*** object. So, the method must get result expression from the input *function* expression.

In general case it is rather complicated to implement the algorithm to calculate function derivative. The methods of expression classes (see class hierarchy) must be used to build the result expression. It is because the derivative rules are **really** complicated for general case of function with many arguments and parameters. Consider the derivative of logarithm function as an example:

$$\frac{d}{dx}\left(\log_{a(x)} g(x)\right) = \frac{1}{g(x)\ln(a(x))}\frac{d}{dx}\left(g(x)\right) - \frac{\ln(g(x))}{a(x)\ln^2(a(x))}\frac{d}{dx}\left(a(x)\right)$$

These rules cannot be defined for all functions generally, they must be defined for any function individually.

But for standard elementary functions of one argument the derivative rule can be generalized, using the following mathematical formula:

$$\frac{d}{dx}(f(g(x))) = \frac{df}{dg}\frac{dg}{dx}$$

The ***SimpleFunctionalDerivative*** class implements this rule. This is an abstract class to implement derivative rule for functions with one argument. It overrides the base ***Derivative*** method (realizing the above rule). The only abstract method of the class which must be overridden in descendants is

```
protected abstract BaseExpression BaseDerivative(BaseExpression argument);
```

The *argument* parameter is g function in the formula above. The implementation of the method is rather simple for most of standard transcendental functions. Consider the code of class, implementing the derivative for sine function, as an example:

```
/// <summary>
/// Sine derivative
/// </summary>
public sealed class SineDerivative : SimpleFunctionalDerivative
{
    protected override string GetFunctionName()
    {
        return "sin";
    }

    protected override BaseExpression BaseDerivative(BaseExpression argument)
    {
        return FunctionExpression.CreateSimple("cos", argument);
    }
}
```

The implementation is rather simple. The method ***GetFunctionName*** tells that the derivative for function with name 'sin' is defined. The method ***BaseDerivative*** creates new expression – function 'cos' with one given argument, because $\frac{d}{dx}(\sin(x)) = \cos(x)$.

As can be seen from the example, the methods of expression classes (see class hierarchy) can be used to easily build result expressions. There are many useful methods to create sum, difference, product, division, power or other expressions from the existing ones. Furthermore, operator overloadings for all algebraic operations ('+'. '-', '*', '/', '^' (power)[6]) with expressions are available for convenience of using natural notations in program code.

ANALYTICS derivative engine supports easy definition of differentiation rules for other, more complicated functions. For an example, in the case of two variable function, the chain rule for partial differentiation can be generalized by the following formulae:

$$\frac{\partial}{\partial x}(f(g(x,y),h(x,y))) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

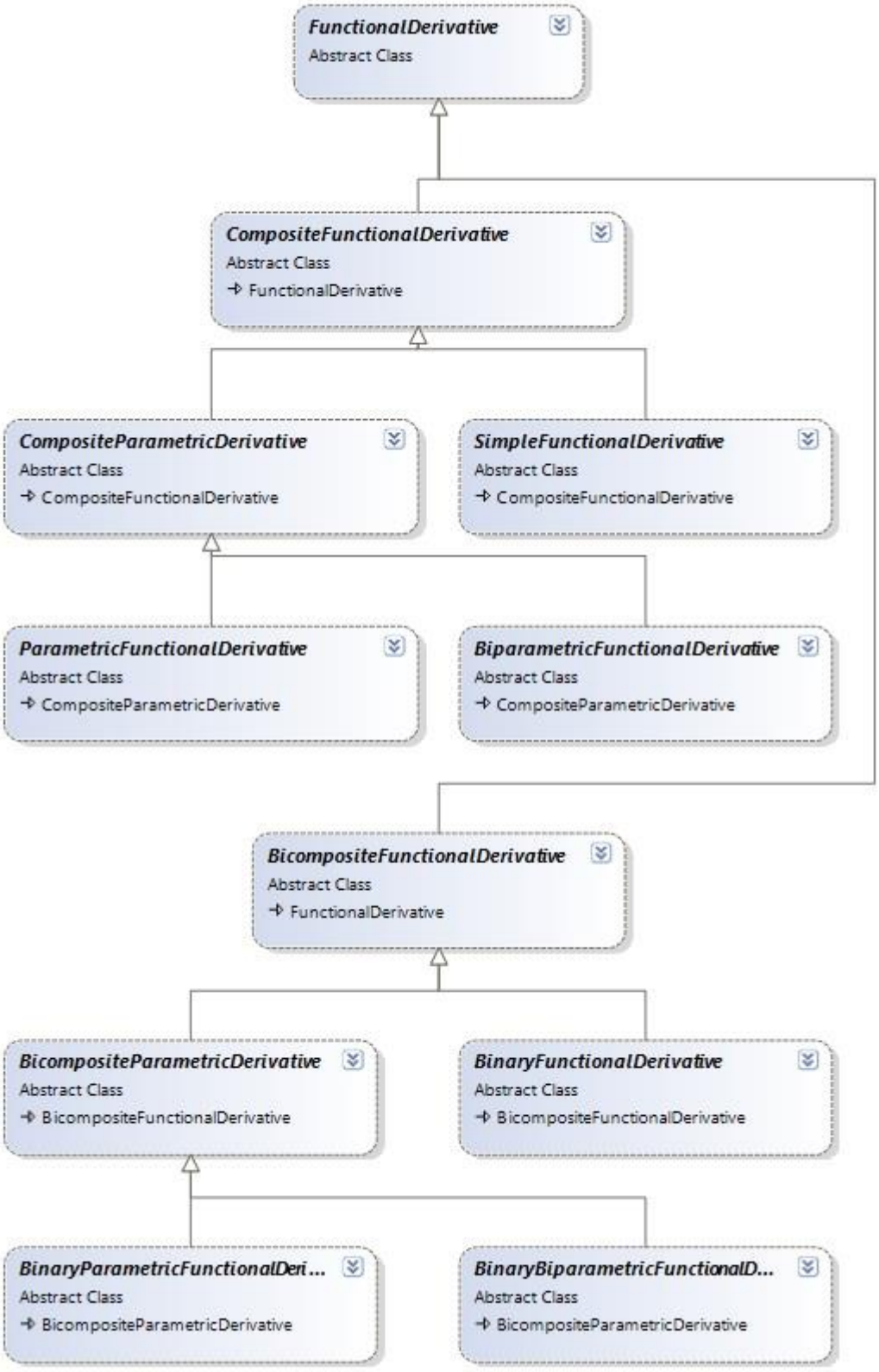
$$\frac{\partial}{\partial y}(f(g(x,y),h(x,y))) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial y} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial y}$$

Abstract class ***BinaryFunctionalDerivative*** implements this rule and all binary function derivatives can be realized easily using this abstraction. Only two methods must be defined in this case:

```
protected abstract BaseExpression BaseDerivative1(BaseExpression argument1, BaseExpression
argument2);
protected abstract BaseExpression BaseDerivative2(BaseExpression argument1, BaseExpression
argument2);
```

As for parametric functions, the chain rule can be applied for them in the case when parameter does not depend on the variable(s). There are abstract implementation of supporting chain rule for functions with up to two variables and two parameters (not depending on variables). The abstract class hierarchy diagram presented on picture 4.1.

[6] Overloaded operator '^' MUST be enclosed in parentheses when using in expressions, because C# '^' operator has lower precedence than algebraic operators have.



Picture 4.1. Functional derivatives class hierarchy diagram.

ANALYTICS Extensions

The ANALYTICS is a complete core library for parsing and calculating mathematical expressions. And there are some extensions of the library those implement special functionality. Extension is a library that is not needed for core algorithm but which introduces new functionality if available. Each extension can introduce the following features:

- recognition of new literal types;
- new variable types;
- operators for special types;
- functions for special argument types.

The next part contains description of existing extensions of the ANALYTICS library.

ANALYTICS Statistics

Statistics extension for ANALYTICS library introduces base statistical analysis functions for real sample data. It allows evaluating base statistical properties of the sample, such as mean value, median, mode, standard deviation, variance and covariance. Other functionality includes: generation of number sequences (Fibonacci, prime numbers and many others); creating arithmetic, geometric and harmonic progressions; working with probability distributions.

The next part contains detailed information about the functions, realized in the Statistics extension library and base processing cases.

Base statistical functions

Most of statistical functions work with real array data. The most of the functions also realized for real matrices and they processed by rows, that is statistical analysis is made for each separate row and the result is array of data for each row.

Evaluating base statistical characteristics:

Mean{P}(X)– mean of the ***X*** array (matrix rows); here ***P*** parameter is the 'power' of the mean algorithm (https://en.wikipedia.org/wiki/Mean#Power_mean), -1 – harmonic, 0 – geometric, 1 – arithmetic, 2 – quadratic.
Median(X) – median of the ***X*** array (matrix rows).
Mode(X) – mode of the ***X*** array (matrix rows), commonly used for arrays of integer values.
Variance(X) – variance of the ***X*** array (matrix rows).
Deviation(X) – standard deviation of the ***X*** array (matrix rows).
Covariance(X Y) – covariance of the ***X*** and **Y** arrays (matrix rows).

Processing data:

Sort{P}(X)– sorts the ***X*** array values; here ***P*** parameter is the order of sorting -1 – descending order, 1 – ascending order.
Sort{C P}(X)– sorts the ***X*** matrix rows; here ***P*** parameter is the order of sorting -1 – descending order, 1 – ascending order; **C** parameter is the column number for sorting the matrix' rows.
Array(M) – transforms the ***M*** matrix to the array (by rows).
Array{min max}(N) – generates array of ***N*** uniform real values on the interval [**min**..**max**].
Reverse(X) – reverses the order of the ***X*** array elements.
Odd(X) – elements of the ***X*** array (rows of matrix) with odd indexes (the first index is 0).
Even(X) – elements of the ***X*** array (rows of matrix) with even indexes (the first index is 0).
Sample(X) – different samples in the ***X*** array (matrix rows) in ascending order, commonly used for integer values.
Frequency(X) – number of different samples (in ascending order) in the ***X*** array (matrix rows), commonly used for integer values.
Histogram{N}(X)– creates the histogram for ***X*** array values (matrix rows) with ***N*** subintervals.
Values{min max}(X) – extracts values from the ***X*** array (matrix rows) laying on the [**min**..**max**] interval. The order of values kept as in the ***X***.
Items{N}(X) – extracts items of the ***X*** array (matrix rows) with indexes, defined in the ***N*** array (must contain integer values). The number of indexes can be greater than the length of ***X***, the indexes with the same values allowed many times, the order of items defined by the indexes.

Here is the example code for base statistical characteristics evaluation of a real array:

```
double[] av = new double[16] { 0.2, 0.2, 0.1, 0.25, 0.35, 0.15, 0.04, 0.01, 0.12, 0.1, 0.02, 0.1,
0.15, 0.25, 0.05, 0.4 };
Translator translator = new Translator();
translator.Add("A", av);

string fm = "Mean{1}(A)";
double m = (double)translator.Calculate(fm);

string fmn = "Median(A)";
```

```
double mn = (double)translator.Calculate(fmn);

string fmd = "Mode(A)";
double md = (double)translator.Calculate(fmd);

string fv = "Variance(A)";
double v = (double)translator.Calculate(fv);

string fd = "Deviation(A)";
double d = (double)translator.Calculate(fd);
```

The output result for the data is:

```
Mean      of A = 0.155625
Median    of A = 0.135
Mode      of A = 0.1
Variance  of A = 0.012124609375
Deviation of A = 0.440447215906742
```

Number sequences and progressions

The Statistics extension contains functions for generating sequences of special numbers. There are the following functions for three special cases:

xxx(N) – generates sequence of ***N*** numbers.
xxx{X0}(N) – generates sequence of ***N*** numbers beginning from the ***X0*** value.
xxx(X1 X2) – generates sequence of numbers on the interval **[X1..X2]**.

Here ***xxx*** is the name of the sequence to generate. The following sequences supported: ***Fibonacci***, ***Primes***, ***Composites***, ***Naturals***, ***Integers***, ***Odds***, ***Evens***, ***Squares***, ***Cubes***, ***Factorials***. The type of numbers generated follows directly from the name. The ***X0*** parameter in the second function can be not a member of the sequence, and then the first generated value is greater or equal to ***X0***.

Another type of number sequence is progression. The library supports arithmetic, geometric and harmonic progressions. There are the following functions for working with progressions:

xxx{A1 P}(N) – generates ***N*** values of progression with the first member ***A1*** and parameter ***P***.
xxx{N}(A1 P) – generates ***N***-th element of progression (the first number is 1) with the first member ***A1*** and parameter ***P***.
xxxSum{A1 P}(N) – calculates sum of ***N*** members of progression with the first member ***A1*** and parameter ***P***.

Where ***xxx*** is the name of progression: ***Arithmetic***, ***Geometric*** or ***Harmonic***.

Let us consider simple example code for generating number sequences:

```
Translator translator = new Translator();

string fp = "Primes(0 100)";
double[] p = (double[])translator.Calculate(fp);

string fg = "Geometric{2 1/2}(10)";
double[] g = (double[])translator.Calculate(fg);
```

The output for the code is the following:

```
Prime numbers = [25]
(2      3       5       7       11      13      17      19      23      29      31      37      41
        43      47      53      59      61      67      71      73      79      83      89      97)
Geometric progression = [10]
(2      1       0.5     0.25    0.125   0.0625  0.03125 0.015625        0.0078125       0.00390625)
```

The first function generated the sequence of all prime numbers on the interval [0..100]. The second function produced first ten members of geometric progression with initial value 2.0 and common ratio 0.5.

Probability distributions

Probability distributions allow generate discrete values for probability distribution functions (PDF), cumulative distribution functions (CDF), inverse distribution functions (quantiles) and random numbers. The functions for the purposes are:

xxxPDF{X0 P}(X1 X2 N) – generates ***N*** values of the PDF on the interval **[X1..X2]** (***X0*** and ***P*** are the parameters of the distribution according to https://en.wikipedia.org/wiki/List_of_probability_distributions).
xxxCDF{X0 P}(X1 X2 N) – generates ***N*** values of the CDF on the interval **[X1..X2]**.
xxxQuantile{X0 P}(X1 X2 N) – generates ***N*** values of the quantile function on the interval **[X1..X2]**.
xxxRnd{X0 P}(X1 X2 N) – generates ***N*** random values for the distribution on the interval **[X1..X2]**.

Here ***xxx*** is the name of distribution. The following distributions supported: ***Gauss*** (normal distribution), ***Laplace***, ***Cauchy***, ***Gumbel***, ***Logistic***, ***Exponential*** (exponential distribution has one parameter only and all functions for it must be called without the ***X0*** value).

Let us consider a simple example code of using random values generators:

```
Translator translator = new Translator();

string fg = "Histogram{11}(GaussRnd{4 1}(0 5 1000000))";
double[] g = (double[])translator.Calculate(fg);
```

The output for the code is:

```
Histogram = [11]
(190    946     3955    12497   32818   70985   124786  180530  211171  202693  159429)
```

The function ***'GaussRnd{4 1}(0 5 1000000)'*** generates one million of random numbers on the interval [0..5] distributed according to the Gauss probability function with parameters μ=4 and σ=1. Then the histogram of the values created for 11 subintervals. As it is expected, the maximum number of values is in the 9-th subinterval that includes the point 4 (the mean or expectation of the distribution).

Numerics extension for ANALYTICS

Numerics extension for ANALYTICS library is a set of ready-to-use numerical tools totally integrated with the symbolic capabilities of the library, like analytical differentiation. The numerical tools include: least squares approximation (curve fitting and higher dimensional data approximation); numerical calculation of one- and two- dimensional definite integrals; ordinary differential equation system solution (initial value problems); function analysis (one dimensional function roots and extremum search); nonlinear equation systems solution.

The most of numerical algorithms realized in MATHEMATICS library and the Numerics extension just provides special classes for the integration between the numerical analysis and the analytical capabilities. The integration classes allows providing all input data for the algorithms in the symbolic form.

The advantages of using analytical data with the numerical methods:
- Convenient data representation (as math expression) for developer and user;
- User defined data for algorithms (for example, approximation function, integrand function);
- Automatic derivative evaluation if required (for example, nonlinear equation solution);
- No need to write special classes for function method references;
- Simple result representation as a string data (for subsequent serialization or network transfer).

The next part contains information about main classes for numerical algorithms and explains the common usage of realized numerical tools with the analytical capabilities.

Approximation

The approximation tool allows making fitting multidimensional data (depending on many variables) with arbitrary, user defined basis functions. The main purpose of the fitting is to find such coefficients of basis functions, which gives minimal error (in some math sense) between the data and the approximation function.

The base abstract class for approximation algorithm ***Approximator<T>*** defined in ***Analytics.Numerics.Approximation***. The main method of the class is:

```
public abstract double[] Approximate(T basis, double[][] vData, double[] fData);
```

where 'basis' – the ***Basis*** instance for approximation; 'vData' - variable values (approximation nodes); 'fData' - function values. The method returns calculated coefficients for specified basis function and approximation data.

The most popular approach of approximation is the least squares method (https://en.wikipedia.org/wiki/Least_squares). The ***LinearLeastSquares*** class realizes the least squares approximation with linear basis (linearly depending on the coefficients).

The approximation algorithm requires the instance of ***Basis*** class, defined in ***Analytics.Numerics.Basis***. This is an abstract class for base basis functionality. The main method of the class is the function for calculation of the basis value in specified point:

```
public abstract double F(double[] vValues, double[] cValues);
```

The method accepts as arguments the variable values (point coordinate) and the coefficient values and returns the basis value.

The fully functional basis class for linear least squares approximation is ***LinearScalarBasis***. It has the following constructor:

```
public LinearScalarBasis(string[] variables, string coefficient, Variable[] parameters, string[] functions)
```

where 'variables' – names of variables (commonly 'x', 'y', …); 'coefficient' – name of coefficient variable (commonly 'C'); 'parameters' – additional parameter variables (can be **nil**); 'functions' – the array of basis functions expressions (depending on specified variables and parameters).

Let us consider an example of curve fitting (approximation of one-dimensional data) with arbitrary set of basis functions. The code for the algorithm is the following:

```
// List of variable names (1 variable 'x' for 1D approximation).
string[] variables = new[] {"x"};
// List of basis functions for approximation (depending on the 'x' variable).
string[] functions = new[] { "e^x", "sin(x)", "e^-x", "x^(-2)", "3^(-x)", "sinh(x/2)" };

// Create the basis with the specified functions.
LinearBasis basis = new LinearScalarBasis(variables, "C", null, functions);

LinearApproximator approximator = new LinearLeastSquares(); // Linear Least Squares approximator.
double[] cValues = approximator.Approximate(basis, vData, fData); // Calculate the optimal basis coefficients.

// Use the basis with optimal coefficients.
```

As can be seen from the example above, the approximation made in five lines of code:

1. Create the array of variable names (containing one value 'x' for one dimensional case).
2. Create the array of basis functions (analytical expressions), containing six elements.
3. Create the basis instance with specified basis functions.
4. Create the approximator instance.
5. Find the approximation coefficients.

Once the coefficients have calculated, they can be used to evaluate the basis value for any point (variable values).

The example code above uses the instance of ***LinearScalarBasis*** class. This class realizes the basis concept for scalar basis functions – it is defined by a set of functions, each function returns one scalar (real) value. Another functional class for linear approximation is ***LinearVectorBasis***. The class realizes concept of vector linear basis – it is define by one basis function which returns vector value. The class has the following constructor:

```
public LinearVectorBasis(string variable, int order, string coefficient, int dimension, Variable[] parameters, string function)
```

where 'variable' – names of vector variable (commonly 'X'); 'coefficient' – name of coefficient variable (commonly 'C'); 'parameters' – additional parameter variables (can be **nil**); 'f' – the basis function expressions (depending on specified variables and parameters) – it must return vector value (array of real values); 'order' – order of basis; 'dimension' – dimension of basis.

Here is the code for curve fitting using linear vector basis:

```
// Setting parameters for approximation.
int dim = 1; // One dimension.
double[] exponents = new double[] {0.0, 0.5, -0.5, -3.0}; // exponent values
int order = exponents.Length; // Basis order is the size of the array.
Variable A = new RealArrayVariable("A", exponents); // Vector variable for exponent array.
double[] nums = new double[] {1.0, 0.2, 0.1, 0.4}; // Sine period values
Variable B = new RealArrayVariable("B", nums); // Vector variable for period array.
Variable[] parameters = new Variable[]{A, B}; // Parameters for Vector basis.
string func = "e^(A*X[0])*sin(B*X[0])"; // Vector function for basis - returns real vector.


// Create the basis with the specified function and parameters.
LinearBasis basis = new LinearVectorBasis("X", order, "C", dim, parameters, func);
LinearApproximator approximator = new LinearLeastSquares(); // Linear Least Squares approximator.
double[] cValues = approximator.Approximate(basis, vData, fData); // Calculate the optimal basis
coefficients.
// Use the basis with optimal coefficients.
```

The latter example's general algorithm is the same as the former: create basis instance, create appropriate approximator and calculate the basis' coefficients. The difference is in the basis instance creation. First, only one basis function required (**func** variable), not an array of functions. The basis function must be vector function – must return array of real values. For this purpose, commonly, it requires additional parameters of vector type (**A** nad **B** variables). Finally, the dimension and order of the basis must be directly specified in constructor.

Advantages of using vector basis are the following:
- Simple basis function expression (in vector form).
- Easily create high-order basis (using big array parameter).
- Change the basis without changing the function (use another parameter array).

The advantages only remain when using basis function of the same type (exponents and periods in the case). When functions of different type required it is recommended using the scalar basis.

There are predefined basis classes for commonly used cases of basis functions. The classes do not require the function expressions and the parameter variables for construction, they implement the functionality internally. The classes are:
- ***GeneralizedExponential*** (a^x basis functions).
- ***ExponentBasis*** (e^x basis functions).
- ***Fourier*** (sine and cosine Fourier series).
- ***Polynomial*** (x^k basis functions).
- ***Fourier2D*** (sine and cosine Fourier series for two- dimensional case).

For nonlinear least squares approximation (https://en.wikipedia.org/wiki/Non-linear_least_squares) there is the fully functional descendant of ***NonlinearBasis*** – ***NonlinearScalarBasis*** which has the following constructor:

```
public NonlinearScalarBasis(string[] variables, string[] coefficients, Variable[] parameters, string
function)
```

where 'variables' – names of variables (commonly 'x', 'y', …); 'coefficients' – names of coefficient variables (commonly 'A', 'B', …); 'parameters' – additional parameter variables (can be **nil**); 'f' – basis function expression (depending on specified variables, coefficients and parameters).

Besides the nonlinear basis, a special nonlinear approximator must be used for nonlinear least squares. The ***GaussNewtonLeastSquares*** is recommended for this purpose.

Here is the code of a nonlinear least squares approximation example:

```
// Example of Nonlinear basis approximation.
// List of variable names (1 variable 'x' for 1D approximation).
string[] variables = new[] { "x" };
// List of basis coefficient names (2 coefficients).
string[] coefficients = new[] { "A", "B" };
// Basis function for approximation - nonlinearly depending on the coefficients.
string func = "sin(A*x)*e^(B*x)";
// Create the basis with the specified parameters.
NonlinearBasis basis = new NonlinearScalarBasis(variables, coefficients, null, func);
NonlinearApproximator approximator = new GaussNewtonLeastSquares(); // Gauss Newton Least Squares
approximator.
approximator.C0 = new[] { 0.0, -1.0 }; // Set up initial guess for coefficients.
SolverOptions opt = new SolverOptions();
```

```
opt.MaxIterationCount = 100;
opt.Precision = 0.002;
approximator.Options = opt; // Set up the appropriate nonlinear solution options.
double[] cValues = approximator.Approximate(basis, vData, fData); // Calculate the optimal basis
coefficients.
SolutionResult res = approximator.SolutionResult; // Nonlinear solution result can be used to analyse the
convergence.
double[] eValues;
double aerr = approximator.Error(basis, vData, fData, cValues, out eValues); // Calculate the final
approximation error.
// Use the basis with optimal coefficients.
```

The approximation algorithm consists of the following steps:

1. Create the array of variable names (containing one value 'x' for one dimensional case).
2. Create the array of nonlinear basis coefficient names.
3. Set up one basis function, depending on the variables, coefficients and optionally on parameters.
4. Create the basis instance with specified parameters.
5. Create the approximator instance.
6. Set up the initial guess for the coefficient values.
7. Set up the appropriate nonlinear solution options.
8. Find the approximation coefficients.

The nonlinear approximation algorithm is slightly complicated than the linear one. It requires setting the initial guess for the approximation coefficients and the appropriate nonlinear solution options (such as the solution precision). Nonlinear least squares approximation uses nonlinear iterative solvers for finding the optimal coefficient values. The convergence of the process depends on the initial guess for the unknown variables. It is recommended to set up the initial guess for the solution accurately (close to the optimal values), based on some special knowledge of the problem. The base class ***NonlinearApproximator*** has special property '*SolutionResult*' that is assigned after call of the '*Approximate*' method. The structure contains data about the solution convergence and can be used for analyzing the nonlinear approximation convergence.

Numerical integration

The numerical integration tool allows calculating definite integrals for one- and two-dimensional functions. The numerical algorithms of integration realized in ***Mathematics.Integration***. The base classes for 1D and 2D integration are ***Integrator1D*** and ***Integrator2D*** accordingly. They have the following functions for definite integral calculation:

```
public abstract double Integral(Function1D f, double x1, double x2, int n);

public abstract double Integral(Function2D f, double x1, double x2, double y1, double y2, int nx, int ny);
```

The input parameters for the functions are the method references to the integrand functions, limits of integrations and the number of integration nodes. The return value in the definite integral of the specified function over the specified integration region.

There are the following realized integrator classes for 1D integration: ***RectIntegrator***, ***SimpsonIntegrator***, ***Gauss2NodeIntegrator***, ***Gauss3NodeIntegrator***; and the following for 2D case: ***BrickIntegrator***, ***Gauss4NodeIntegrator2D***, ***Gauss9NodeIntegrator2D***.

For using analytical capabilities with the integration tool, there are the classes of symbolic functions, realized in the ***Analytics.Numerics.Functions*** unit. The classes are ***SymbolicFunction1D*** and ***SymbolicFunction2D*** for one- and two- dimensional cases accordingly. They have the following constructors:

```
public SymbolicFunction1D(string v, string f, Variable[] parameters)
public SymbolicFunction2D(string v1, string v2, string f, Variable[] parameters)
```

The variable names, analytical function expression and additional parameter variables must be specified for construction. The classes have the methods for evaluating them in any point which can be used for numerical integration:

```
public double F(double v)
public double F(double x, double y)
```

Below here is the code for example of numerical integration of 1D function:

```
Integrator1D integrator = new Gauss3NodeIntegrator();
```

```
SymbolicFunction1D sf1D = new SymbolicFunction1D("x", "sin(x)*e^(x^2/8)"); // Create the symbolic function instance for integration.
double ivalue = integrator.Integral(sf1D.F, x1, x2, n);   // Integrate the function with the integrator.
// Code for using integral value 'ivalue'
```

The integration algorithms is simple: create the appropriate integrator; create the integrand function instance; calculate the definite integral value.

Ordinary differential equation solution

This tool allows solving initial value problems (https://en.wikipedia.org/wiki/Initial_value_problem) for the systems of ordinary differential equations. The numerical solution algorithms realized in the ***Mathematics.ODE.Solver*** unit. The base abstract class for solving the problems is ***ODESolver***. And the only its method is:

```
public abstract double[][] Solve(ODESystem system, double[] y0, double t1, int N, ref double[] t);
```

The method solves the initial value problem for specified ODE system[7], initial condition and time interval. The result of the solution is the array of function values on each time strep (the time steps returned as **var** method parameter).

There are the following classes implementing the ODE solver functionality: ***EulerSolver***, ***RungeKutta4Solver***, ***FehlbergSolver***.

The ***Solve*** method requires the instance of the ***ODESystem*** (defined in the ***Mathematics.ODE.System*** unit) class as the first parameter. The main method of the class is the following:

```
public abstract double[] Evaluate(double t, double[] y);
```

The method evaluates the equations of the system for the specified variable value and functions values.

For using analytical capabilities with the ODE solution tool, there is the base abstract class of the symbolic ODE function - ***AnalyticalODE***. Its implementation ***ScalarODE*** has the following constructor:

```
public ScalarODE(string v, string[] fv, string[] equations, Variable[] parameters)
```

where 'v' is the variable name; 'fv' is the unknown function names; 'equations' is the array of analytical expressions representing the equations.

As an example, let us consider the solution of the initial value problem for one ordinary differential equation. The code for the example is the following:

```
// Create the ODE solver.
ODESolver solver =  new RungeKutta4Solver();
string[] fv = new string[1]{ "y" };            // Create the arrays, containing one
string[] equations = new[1]{ "2*sin(t)+t/y" }; // element only, because we solve here
double[] y0 = new double[1] { y10 };           // the initial problem for one ODE.
ODESystem ode = new ScalarODE("t", fv, equations, null); // Create the ODE system containing one equation.
double[] t = null;
double[][] y = solver.Solve(ode, y0, t1, N, ref t); // Solve the problem with the specified initial condition,
// Use the 'y' solution.
```

The example demonstrates common steps for the initial value problem solution: create the analytical ODE system with specified equation expressions; select the appropriate solver; specify the initial values; solve the system with specified time interval and number of discrete steps.

NOTE: the number of returned time and function values can differ from the specified time steps. There are automatic time step solvers, like Fehlberg's method, which select the step by some precision formula. The only thing guaranteed is that the last time value is more or equal to the specified. The precision of the step selected can be specified using the '***StepTolerance***' property of the solver's class.
Another implementation of the analytical ODE system ***VectorODE*** is intended for modeling systems set up in 'matrix' form (https://en.wikipedia.org/wiki/Matrix_differential_equation). The class has the following constructor:

```
public VectorODE(string v, string fv, string equation, int dimension, Variable[] parameters)
```

[7] All equation must specify the ODE of the first order. The systems of higher orders must be transformed first to the equivalent systems of the first order.

The constructor requires one unknown function name and one equation. The unknown function is supposed to be an array of float values as well as the equation return. The extension package '***Analytics.LinearAlgebra***' must be initialized to support all array/matrix operations.

The following code demonstrates an example of solving an initial value problem for a vector ODE system:

```
// Example of solving a Vector ODE system.
double t1 = 3.0; // The end value of the interval for solution.
int N = 1000;    // The number of steps for interval discretization.
// Parameter variables:
double[,] ma = new double[2,2]
{
    {3, -4  },
    {4, -0.7}
};
Variable A = new RealMatrixVariable("A", ma); // A-matrix.
double[] vb = new double[2] {-1, 1};
Variable B = new RealArrayVariable("B", vb); // B-vector.
Variable[] prms = new Variable[]{A, B};
double[] y0 = new double[2] {1.0, -1.0}; // Initial conditions.
// Use the Fehlberg (Fehlberg-Runge-Kutta 4-5) solver.
ODESolver solver = new FehlbergSolver();
ODESystem ode = new VectorODE("t", "y", "A×y+B/(t^2+1)", 2, prms); // Create the Vector ODE system with
the specified parameters.
double[] t = null;
double[][] y = solver.Solve(ode, y0, t1, N, ref t); // Solve the problem with the specified initial
condition,
                                                    // integration interval and number of discretization
steps.
// Use the y solution.
```

As can be seen from the code above, the matrix/array parameters must be created and added to the ODE system constructor. This is because the result of the equation must be array of float values. The constructor parameter must also directly provide the dimension of the system. But only one equation required to set up the system.

Function analysis

The analysis tool provides functionality for finding roots and extremum points for univariate functions. The base abstract class for function root search is ***RootFinder*** defined in the ***Mathematics.Analysis*** unit. It has the following function:

```
public abstract SolutionResult Solve(Function1D f, Function1D df, double x1, double x2, SolverOptions opt,
ref double xr);
```

The function solves the nonlinear equation *f(x)=0* for the specified univariate function, interval and solution options (defining the precision, maximum iteration count and so on). The returned result contains the information about solution convergence, made iterations and so on. The found root returned via **ref** 'xr' parameter.
All solvers requires a delegate of type ***Function1D*** (defined in the ***Mathematics.Numerics***). Some solvers require also the reference to the derivative function. The method ***DerivativeRequired*** of the class defines if the derivative required for the solver.

There are the following classes, implementing the root search: ***Bisection***, ***Secant***, ***Newton***.

NOTE: All algorithms for root search can find only one root on the specified interval. The convergence depends on the algorithm and the specified parameters.

The ***SymbolicFunction1D*** class (defined in the ***Analytics.Numerics.Functions*** unit) provides the functionality for finding roots of the analytical functions, including automatic derivative calculation.

The simple code example for finding a root of a function is the following:

```
// Create the root finder.
RootFinder solver = new Newton();
SymbolicFunction1D sf1D = new SymbolicFunction1D("x", "sin(x^2)*e^-x"); // Create the symbolic function
instance for analysis.
Function1D fm = sf1D.F;
SymbolicFunction1D dsf1D = null;
Function1D dfm = null;
if (solver.DerivativeRequired()) // If the solver requires derivative evaluation -
{
    dsf1D = sf1D.Derivative(); // get the derivative instance.
    dfm = dsf1D.F;
```

```
}

SolverOptions opt = new SolverOptions();     // Default options for the solver.
double xr = double.NaN;
SolutionResult sr= solver.Solve(fm, dfm, x1, x2, opt, ref xr); // Finding root with the selected solver
and specified parameters.
// Use the xr root value.
```

The example consists of the following steps:

1. Select the appropriate solver.
2. Create the instance of the symbolic function and get its derivative.
3. Set up the options for the solver.
4. Solve the nonlinear equation.

The class ***FunctionAnalyser*** from the ***Analytics.Numerics.Analysis*** unit provides advanced functionality – it allows finding many roots and extremums on some interval. Let us consider an example of the analysis.

```
RootFinder solver = new Newton();
SymbolicFunction1D sf1D = new SymbolicFunction1D("x", "sin(x^2)*e^-x"); // Create the symbolic function
instance for analysis.
SolverOptions opt = new SolverOptions();     // Default options for the solver.
FunctionPoint[] points = FunctionAnalyser.Analyse(sf1D, solver, opt, 1.0, 3.0, 10); // Finding all
function special points.
// Use the 'points' result.
```

In the code above, the Analyse method divides the specified interval [1.0..3.0] by 10 uniform subintervals and try to find a root and an extremum on each of them. The result is the array of found special points. Each point contains variable value, function value and the type – root, minimum or maximum.

Nonlinear equation systems solution

The base abstract class for solving nonlinear equation systems is the ***NonlinearSolver*** defined in the ***Mathematics.NL.Solver*** unit. It has the following method:

```
public abstract SolutionResult Solve(NonlinearSystem system, double[] x0, SolverOptions opt, ref double[]
x);
```

where 'system' is the definition of a nonlinear equation system; 'x0' is the initial guess for solution; 'options' specifies such parameters as precision, number of iterations and so on. The method returns as result the information about solution convergence. The found root of the system returned as the **ref** 'x' parameter.

There implemented nonlinear solver class is ***NewtonRaphsonSolver***.

The class ***AnalyticalSystem*** from the ***Analytics.Numerics.Nonlinear*** implements full functionality for definition of nonlinear systems in symbolic form, including analytical Jacobian calculation.

Here is the code of example for solving three-dimensional nonlinear system, which comes from the problem of intersection of three nonlinear surfaces.

```
RealVariable R = new RealVariable("R", 1); // Variable for the radius value
string[] variables = new[] { "x", "y", "z" };  // equations' variables
string[] equations = new[] { "x^2+y^2+z^2-R^2",// sphere
                             "x^2+y^2-z",      // paraboloid equation - along x-axis
                             "-x+2*y^2+z^2"    // paraboloid equation - along z-axis
                           };
Variable[] parameters = new[] { R };           // equations' parameters
AnalyticalSystem system = new AnalyticalSystem(variables, equations, parameters); // nonlinear equation
system
double[] x0 = new[] { 1.0, 1.0, 1.0 };              // initial guess for solution
SolverOptions options = new SolverOptions();        // default solution options
NonlinearSolver solver = new NewtonRaphsonSolver(); // Newton-Raphson solver
double[] x = null;
SolutionResult r = solver.Solve(system, x0, options, ref x); // solving the system
// Use the 'x' result.
```

The solution algorithm is simple: construct the analytical system instance with provided arrays of variable names and equations; set up the initial guess and options for the solution; create the solver instance; solve the problem for the specified data.

ANALYTICS Linear Algebra

The ANALYTICS Linear Algebra extension library introduces features to implement analytical calculations with 3D vectors and tensors and n-dimensional vectors and rectangular matrices. The following features realized[8]:

1. Variable types: ***Vector3DVariable***, ***Tensor3DVariable***.
2. Overloaded operators for ***Vector3D*** operands: '**•**' (dot product).
3. Overloaded operators for ***Tensor3D*** operands: '**'**' (transpose), '**^**' (power).
4. Overloaded operators for Array/Matrix operands:
   - Addition operations '+', '-' for Array and Matrix operands implemented by rows – the array elements added to (subtracted from) each matrix row.
   - Number operator '#': number of elements in an array or a matrix.
   - Cross product '×' of Arrays and Matrices implemented with common vector/matrix rules. For two matrices it is the matrix multiplication. For two arrays it is the outer product – result is a matrix. For mixed array/matrix operands: when array is the first operand it is treated as a 'row-vector', when it is the second operand – as a 'column-vector'.
   - Multiplication '*' of Arrays and Matrices implemented as 'by-element' operation. When applied for mixed Array/Matrix operands the by-component operation is implemented by rows (as addition or subtraction).
   - The dot product operation '•' defined for Array operands only and follows common rules of linear algebra.
   - Division '/' of Arrays and Matrices implemented as 'by-element' operation (see multiplication operator).
   - The power '^' operation for arrays and matrices is implemented by components.
   - Minus operator '-': sign inverse of elements.
   - Square root operator '√': square root of elements.
   - Sum operator '∑': sum of an array elements; sum of a matrix rows (array).
   - Delta operator '∆': finite difference D[i]=A[i+1]-A[i] for array elements; for a matrix the operator evaluates by each row.
   - Product operator '∏': product of an array elements; product of a matrix rows (array).
   - Apostrophe operator ''': transposition of a matrix.
   - Accent operator '`': inverse of a matrix (applicable for square matrices only).
   - Absolute operator '||': absolute values of array or matrix elements.
   - Norm operator '‖‖': vector L2 norm of an array or a matrix.
5. Elementary and base special functions (see **Appendix A**) are applicable for array/matrix arguments. These functions apply the evaluation to every component of an array or a matrix and return an array or a matrix of the same size.
6. Functions for ***Vector3D*** arguments/parameters: ***Vector*** (creates vector by real components), ***Length***, ***Dot***, ***Cross***, ***Angle***, ***Distance***, ***Direction***, ***Normal***, ***Area***.
7. Functions for ***Tensor3D*** arguments/parameters: ***Tensor*** (creates tensor by real components), ***Invariant1***, ***Invariant2***, ***Invariant3***, ***Transpose***, ***Inverse***, ***Symmetric***, ***Antisymmetric***, ***Invariant1***, ***Solve*** (solves linear equation system).
8. Functions for ***Vector*** (n-dimensional) arguments/parameters: ***Min***, ***Max***, ***Length***, ***Range***.
9. Functions for ***Matrix*** (rectangular) arguments/parameters: ***Min***, ***Max***, ***RowCount***, ***ColumnCount***, ***Diagonal***, ***Antidiagonal***, ***Row***, ***Column***, ***Range***, ***Transpose***, ***Inverse***, ***Minor***, ***Determinant***, ***Adjoint***, ***Solve*** (solves linear equation system).
10. Additional types: ***StringVector***, ***StringTensor***. These special classes implement 'analytical' vector and tensor (their components are string expressions, all operations implemented in analytical form).
11. Other functions for Array/Matrix arguments:
    CumSum(X) – cumulative sum of the array (for matrix it is evaluated for each separate row).
    CumProduct(X) – cumulative product of the array (for matrix it is evaluated for each row).
    Outer(X Y) – outer product of two arrays, the result is the matrix.
    det(M) – determinant of a square matrix.
    tr(M) – trace of a square matrix.
    adj(M) – adjoint of a square matrix.
    cond(M) – condition number of a square matrix (using L2 norm).
    pinv(M) – pseudo-inverse of a rectangular matrix.

Logical array and matrix operations

The Linear Algebra extension also introduces operations with logical array and matrices (containing Boolean values) and comparison operations with Real/Array/Matrix values.

The following logical operators supported for logical array and matrix operands: not '**¬**', or '****', and '**&**'. The comparison operations are: '**≡**', '**≈**', '**≠**', '**>**', '**<**', '**≥**', '**≤**'. The logical and comparison operations are element-wise ones, the operation's result is logical array or matrix of the same size as operands are. Binary operations also supported when one of the operands is a scalar value. The multiplication operator '***' also applicable for real/array/matrix operand and logical array/matrix. The operation is equivalent to the element-wise multiplication for logical values '**true**'=1 and '**false**'=0.

The following functions, described above, are also applicable for the logical arrays and matrices: **Range**, **RowCount**, **ColumnCount**, **Row**, **Column**; and constructors **Array**, **Matrix**.

---

[8] ANALYTICS Linear Algebra library depends on MATHEMATICS library because it uses special types, such as Vector3D, Tensor3D and so on, from it. As these special types implement operators overloading for base mathematical operations, extension library introduces only additional operators ('explicitly' overloaded operators used automatically).

The special function **Count{x}(A)** returns the number of 'x' values (true or false) in logical array 'A'. There are special variants of **'if'** function for the real arrays and matrices with logical parameter: **if{B}(X Y)**. Here the 'B' parameter is logical array or matrix of the same size as the 'X' and 'Y' arguments – real arrays or matrices. The result is also real array or matrix, which elements are the results of element-wise **'if'** function.

ANALYTICS Fractions

The ANALYTICS Fractions extension library implements analytical operations with Common (simple) fractions. The following features realized:

1. Common Fraction Literal: the library introduces support of parser to recognize fraction literals. After the literal is registered it recognizes the expression **'a/b'** as fraction if **'a'** and **'b'** are integer numbers. NOTE: due to precedence of operators, the fraction literal must be enclosed by parentheses, if the operators outside have the higher or the same precedence as '/' operator does. For an example, **'2/3+4/2'** – parentheses NOT obligatory, **'(2/3)*(4/2)'** – parentheses ARE obligatory.
2. Variable type: ***FractionVariable***.
3. Overloaded operators for Fraction operands: **'^'** (power), **'~'** (conversion to real value), **''** (inversed fraction) and all standard binary algebraic and relational operators.
4. Overloaded standard binary algebraic operators for Real-Fraction and Complex-Fraction operands and overloaded relational operators for Real-Fraction operands.
5. Functions for Fraction arguments: **Fraction** (constructs fraction from numerator and denominator or converts double value to common fraction), ***Numerator***, ***Denominator***, ***Real*** (converts fraction to real value), ***frac*** (fractional part of a common fraction), ***int*** (integer part of a common fraction), ***sgn*** (sign of a fraction).

ANALYTICS Complex

The ANALYTICS Complex extension library implements operations with Complex numbers. As was said above, ANALYTICS library supports Complex literals by default. Also, ANALYTICS library supports 'explicit' operator overloading. So, some operations with complex numbers are available without any extension library. As an example, the expression **'(2-I)*(3+2I)'** can be evaluated correctly. But the default functionality is limited by literal expressions with operators those can be 'explicitly' overloaded.

The ANALYTICS Complex library introduces the following features:

1. Variable types: ***ComplexVariable***, ***ComplexArrayVariable***, ***ComplexMatrixVariable***, ***ComplexBlockVariable***.
2. Overloaded operators for Complex operands: **'^'** (power), **'~'** (conjugate), **''** (reciprocal).
3. Overloaded binary operators for Complex-Real operands: **'+'**, **'-'**, **'*'**, **'/'**, **'^'**.
4. Functions for Complex and Complex-Real arguments/parameters: ***Complex*** and ***Polar*** (create complex number by Real arguments), ***Conjugate***, ***Reciprocal***, ***Re*** (real part), ***Im*** (imaginary part), ***Arg*** (argument), ***abs*** (absolute), ***sqrt*** (square root), ***root*** (n-th root), ***pow*** (power), ***log*** (logarithm), ***ln*** (natural logarithm), ***lg*** (decimal logarithm), ***lb*** (binary logarithm), ***exp*** (exponent), trigonometric functions (***sin***, ***cos***, ***tan***), inverse trigonometric functions (***arcsin***, ***arccos***, ***arctan***), hyperbolic functions (***sinh***, ***cosh***, ***tanh***), inverse hyperbolic functions (***arcsinh***, ***arccosh***, ***arctanh***). For entire list of elementary and base special functions see **Appendix A**.
5. Overloaded operators for Complex arrays and matrices (see **ANALYTICS Linear Algebra** extension).
6. By-element elementary and base special functions for Complex arrays and matrices (see **ANALYTICS Linear Algebra** extension).
7. Overloaded operators for Logical/Complex Arrays/Matrices (see **ANALYTICS Linear Algebra** extension).

ANALYTICS Mathphysics

The ANALYTICS Mathphysics extension library introduces features to work with physical entities (units of measurement and physical values)[9] in analytical expressions. The following features realized:

1. Literal types: the library supports following literals – ***unit literal***, ***scalar value literal***, ***vector value literal*** and ***tensor value literal***. Unit literal is sequence of symbols, representing valid notation of a unit of measurement. As there is a lot of unit names and prefixes, and the unit notation is 'rather close' to mathematical expression notation (used in ANALYTICS library) unit literals MUST BE ENCLOSED in triangle braces '‹›'[10]. The examples of valid unit literals: **'‹mm^2›'**, **'‹m kg/s^2›'**, **'‹m kg s^-2›'**. The scalar value literal is a real number (literal) with associated unit of measurement. In other words – Real literal + Unit literal. The real literal and the unit can be separated by space ' ' or can be NOT separated. The examples of valid scalar value literals: **'4‹m/s›'**, **'2.2‹m kg s^-2›'**, **'-3 ‹m/s^2›'**. NOTE: as the space symbol ' ' used in unit literals instead of product symbol (for notation being

[9] ANALYTICS Mathphysics library depends on PHYSICS library. For total information about units of measurement and physical values see PHYSICS manual.

[10] Do not confuse symbols of triangle braces '‹ ›' with keyboard symbols '<' – less and '>' – more.

close to human) but in ANALYTICS library the symbol used for argument separator, the unit (or scalar value) literal should be enclosed in additional parenthesis when used as function parameter/argument. Vector value literal is vector literal and unit literal separated (optionally) with space. Vector literal is three real literals (vector components) separated with spaces and enclosed in parentheses. Tensor value literal is again tensor literal and unit literal. Tensor literal is like vector literal but contain nine real literals (components, positioned by rows). Examples of correct literals: vector value **‘(1 0 -1) ‹m/s›’**, tensor value **‘(1 0.3 -1 -0.3 2 0.4 1 -0.4 3) ‹N/mm^2›’**.

2. Variable types: ***UnitVariable***, ***PhysicalScalarVariable***, ***PhysicalVectorVariable***, ***PhysicalTensorVariable***.
3. Overloaded operators for Scalar-Unit-Real operands: **‘+’**, **‘-’**, **‘*’**, **‘/’**, **‘^’** (power – integer, including negative values).
4. Functions for Scalar-Unit-Real arguments/parameters: ***Convert*** (converts values from one unit to another), ***Value*** (the Real value of physical scalar), ***Unit*** (the unit of physical scalar).
5. Overloaded operators for Vector-Unit-Real operands: **‘+’**, **‘-’**, **‘*’**, **‘/’**, **‘•’** (dot product of vector values).
6. Overloaded operators for Tensor-Vector-Unit-Real operands: **‘+’**, **‘-’**, **‘*’**, **‘/’**, **‘^’** (power – integer, including negative values), **‘'’** (transpose tensor).
7. Functions for Vector and Tensor arguments/parameters: ***Vector*** (creates physical vector value). ***Tensor*** (creates physical tensor value).

ANALYTICS Special

The ANALYTICS Special extension library introduces possibility of calculating special functions in analytical expressions. The following functions realized: ***P*** – Legendre’s polynomial and associated Legendre’s polynomial of the first kind, J_0, J_1, Y_0, Y_1, I_0, I_1, K_0, K_1 – Bessel functions and modified Bessel functions of the first and second kind (of order 0 and 1).

The Special extension library introduces derivative rules for all special functions written above. In addition, derivative rules defined for: J_n, Y_n, I_n, K_n (Bessel functions of order n), ***Q*** – Legendre’s polynomial and associated Legendre’s polynomial of the second kind, B – beta function and incomplete beta function, Γ – gamma, incomplete and logarithmic gamma function, ψ – digamma and polygamma function, erf – error function, erfc – complementary error function.

Converting expressions to external formats and drawing formulae

ANALYTICS library works with formulae, written as plain C# strings (using Unicode characters). The main reason for this is providing simple interface for dealing with formulae in the code. As plain text is not a natural form for math expression representation, the library uses special syntax to write some types of operations: power operator, vector and matrix expressions and so on. Math expressions, written using the syntax can be converted to external formats to show the result of symbolic evaluations as math formula.
Conversion classes realized in ***Exversion*** assembly. The base class for formula conversion is ***AnalyticsConverter*** that has the following main method:

```
public abstract string Convert(string value);
```

This method converts any symbolic expression, written according the sysntax of ANALYTICS library, to some external format. The realized descendant of the class is ***AnalyticsTeXConverter*** that converts formulae to **TeX** format (https://en.wikibooks.org/wiki/LaTeX/Mathematics). TeX format is widely used for creating scientific articles and there are many software tools to draw TeX formula and convert them to DOC or PDF formats.
Common conversion algorithm is the following:

```
string f = "A*sin(n*x)+B/2*e^(m*y)";
AnalyticsConverter converter = new AnalyticsTeXConverter();
string texf = "";
try
{
    texf = converter.Convert(f);
}
catch (Exception ex)
{
    // Show exception message...
}
// Using string in TeX format...
```

First, an instance of converter created. Then the method **'*Convert*'** used for converting string to the TeX format. After the code execution, the **'*texf*'** variable will have the following value:

```
{{{A}\cdot{{sin}\left({{n}\cdot{x}}\right)}}+{{\frac{B}{2}}\cdot{{e}^{{m}\cdot{y}}}}}
```

This TeX string can be then used to draw the expression as natural math formula in desktop or Web application. For example, to draw the formula in Web browser, the MathJax CDN service (https://en.wikipedia.org/wiki/MathJax) can be used. For doing this it is enough to create a file with the following HTML code:

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript" src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML"></script>
<title> Analytics & Physics </title>
</head>
<body>
 $$ XXX $$
</body>
</html>
```

Here **'XXX'** must be replaced with the TeX formula to draw. Opening the file with Web browser, the formula will be drawn like this:

$$A \cdot sin\left(n \cdot x\right) + \frac{B}{2} \cdot e^{m \cdot y}$$

The TeX converter supports all types of expressions, defined in ANALYTICS library. Here are the examples of conversion for different types of expressions with features description:
Power expressions shown as superscripts: **'(x+1)^(y-2)'**

$$\left(x + 1\right)^{y-2}$$

Multiple powers enclosed with parentheses to keep the order of operations: **'(x^y)^-z'**

$$\left(x^{y}\right)^{-z}$$

Multiple division/multiplication operations presented as a sequence of fractions: **'(x-1)/2*(x+1)/A/B*(x-A)*(x+B)'**

$$\frac{x-1}{2} \cdot \frac{x+1}{A \cdot B} \cdot (x-A) \cdot (x+B)$$

Indexed data (array and matrix elements) shown with subscripts: **'A[i+1][j-1][n*k]'**

$$A_{i+1\,j-1\,n \cdot k}$$

Parametric (special) functions use subscripts for parameters: **'log{a}(x+1)'**

$$log_a\,(x+1)$$

Special functions with two parameters use super- and sub- scripts: **'P{n m}(x+1)'**

$$P_m^n\,(x+1)$$

Special operators use special symbols and format: **'√(x^2-1)*∑(A*x/N!)'**

$$\sqrt{x^2-1} \cdot \sum \left(\frac{A \cdot x}{N!} \right)$$

Implicit derivative operator drawn as partial derivative: **'∂²ψ(δ*ε)/∂ε²+∂³(ζ(α)^η(β))/∂α∂β²'**

$$\frac{\partial^2 \psi\,(\delta \cdot \epsilon)}{\partial \epsilon^2} + \frac{\partial^3 \zeta\,(\alpha)^{\eta(\beta)}}{\partial \alpha \partial \beta^2}$$

Matrix and vector expressions shown with brackets and braces: **'[[x y z] [0 -1 a]]×[i j k]'**

$$\begin{bmatrix} x & y & z \\ 0 & -1 & a \end{bmatrix} \times \begin{Bmatrix} i \\ j \\ k \end{Bmatrix}$$

Thus, all symbolic expressions can be shown in natural form of math formulae.

Appendix A. Analytics operators and functions

Table A.1. List of operators, defined in ANALYTICS library.

Operator	Symbol	Type	Derivative[11]
Logical And	&	Binary	False
Logical Or	\	Binary	False
Identically equal	≡	Binary	False
Approximately equal	≈	Binary	False
Not equal	≠	Binary	False
Greater	>	Binary	False
Less	<	Binary	False
Greater or equal	≥	Binary	False
Less or equal	≤	Binary	False
Add	+	Binary	True
Subtract	-	Binary	True
Multiply	*	Binary	True
Divide	/	Binary	True
Dot	•	Binary	False
Cross	×	Binary	False
Power	^	Binary	True
Left arrow	←	Binary	False
Right arrow	→	Binary	False
Up arrow	↑	Binary	False
Down arrow	↓	Binary	False
Left-right arrow	↔	Binary	False
Up-down arrow	↕	Binary	False
Logical Not	¬	Unary, Prefix	False
Question	?	Unary, Prefix	False
Number	#	Unary, Prefix	False
Minus	-	Unary, Prefix	True
Tilde	~	Unary, Prefix	False
Square Root	√	Unary, Prefix	True
Derivative	∂	Unary, Prefix	True
Integral	∫	Unary, Prefix	True
Delta	Δ	Unary, Prefix	True
Sum	∑	Unary, Prefix	True
Product	∏	Unary, Prefix	False
Factorial	!	Unary, Postfix	False
Apostrophe	'	Unary, Postfix	False
Accent	`	Unary, Postfix	False

[11] If derivative is not defined for some operator, you can still use it in expressions and get symbolic derivative for this expression in the case when operand(s) for the operator do not depend on variable. Dot operator may not be used in symbolic derivation process in any case.

Operator	Symbol	Type	Derivative[11]
Absolute	||	Unary, Outfix	True
Norm	‖‖	Unary, Outfix	False

Table A.2. List of basic functions, defined in ANALYTICS library.

Function[12]	Name	Example	Derivative[13]
Absolute value	abs	abs(x)	sgn(x)
Signum [R]	sgn	sgn(x)	2*delta(x)
Dirac delta function [R]	delta	delta(x)	not defined
Heaviside step function [R]	H	H(x)	delta(x)
If (conditional) function	if	if{x>0}(x x^2)	if{x>0}(1 2*x)
Ceiling function [R]	ceil	ceil(x)	not defined
Floor function [R]	floor	floor(x)	not defined
Fractional part [R]	frac	frac(x)	not defined
Sine	sin	sin(x)	cos(x)
Cosine	cos	cos(x)	-sin(x)
Tangent	tan	tan(x)	1/cos(x)^2
Cotangent	cotan	cotan(x)	-1/sin(x)^2
Secant	sec	sec(x)	sin(x)/cos(x)^2
Cosecant	cosec	cosec(x)	-cos(x)/sin(x)^2
Inverse sine	arcsin	arcsin(x)	1/(1-x^2)^(1/2)
Inverse cosine	arccos	arccos(x)	-1/(1-x^2)^(1/2)
Inverse tangent	arctan	arctan(x)	1/(1+x^2)
Inverse cotangent	arccot	arccot(x)	-1/(1+x^2)
Inverse secant	arcsec	arcsec(x)	1/(x^2*(1-1/x^2)^(1/2))
Inverse cosecant	arccsc	arccsc(x)	-1/(x^2*(1-1/x^2)^(1/2))
Hyperbolic sine	sinh	sinh(x)	cosh(x)
Hyperbolic cosine	cosh	cosh(x)	sinh(x)
Hyperbolic tangent	tanh	tanh(x)	1/cosh(x)^2
Hyperbolic cotangent	coth	coth(x)	-1/sinh(x)^2
Hyperbolic secant	sech	sech(x)	-tanh(x)*sech(x)
Hyperbolic cosecant	cosech	cosech(x)	-coth(x)*cosech(x)
Inverse hyperbolic sine	arsinh	arsinh(x)	1/(x^2+1)^(1/2)
Inverse hyperbolic cosine	arcosh	arcosh(x)	1/(x^2-1)^(1/2)
Inverse hyperbolic tangent	artanh	artanh(x)	1/(1-x^2)
Inverse hyperbolic cotangent	arcoth	arcoth(x)	1/(1-x^2)
Inverse hyperbolic secant	arsech	arsech(x)	-1/((x^2*(1/x-1)^(1/2))*(1/x+1)^(1/2))
Inverse hyperbolic cosecant	arcsch	arcsch(x)	-1/(x^2*(1+1/x^2)^(1/2))
Logarithm to base	log	log{a}(x)	1/(ln(a)*x)
Natural logarithm	ln	ln(x)	1/x
Decimal logarithm	lg	lg(x)	1/(ln(10)*x)
Binary logarithm	lb	lb(x)	1/(ln(2)*x)

[12] Most of the basic functions support real and complex arguments/parameters. If some function does not support complex numbers – it is marked with 'R'.

[13] If derivative is not defined for some function, you can still use it in expressions and get symbolic derivative for this expression in the case when arguments/parameters for the function do not depend on variable.

Function[12]	Name	Example	Derivative[13]
Exponent	exp	exp(x)	exp(x)
Square root	sqrt	sqrt(x)	1/(2*x^(1/2))
Root (with index)	root	root{a}(x)	1/a*x^(1/a-1)
Power	pow	pow{a}(x)	a*x^(a-1)
Beta[D] function[14]	B	B(x y)	B(x y)*(ψ(x)-ψ(x+y))
Incomplete Beta[D]	B	B{n m}(x)	x^(n-1)*(1-x)^(m-1)
Gamma[D] function	Γ	Γ(x)	Γ(x)*ψ(x)
Logarithm of Gamma[D]	Γlog	Γlog(x)	ψ(x)
Incomplete gamma[D]	Γ	Γ{n}(x)	-(x^(n-1)*e^-x)
Digamma function[D]	ψ	ψ(x)	ψ{1}(x)
Polygamma[D] function	ψ	ψ{n}(x)	ψ{n+1}(x)
Error[R] function	erf	erf(x)	(2/π^(1/2))*e^-(x^2)
Complementary[R] error	erfc	erfc(x)	(-2/π^(1/2))*e^-(x^2)
Inversed error function[R]	erfi	erfi(x)	√π/2*e^(erfi(x)^2)
Bessel[R] function of order 0	J_0	$J_0(x)$	$-J_1(x)$
Bessel[R] function of order 1	J_1	$J_1(x)$	$J_0(x)-J_1(x)/x$
Bessel[R] function of the second kind, order 0	Y_0	$Y_0(x)$	$-Y_1(x)$
Bessel[R] function of the second kind, order 1	Y_1	$Y_1(x)$	$Y_0(x)-Y_1(x)/x$
Modified Bessel[R] function of order 0	I_0	$I_0(x)$	$I_1(x)$
Modified Bessel[R] function of order 1	I_1	$I_1(x)$	$I_0(x)-I_1(x)/x$
Modified Bessel[R] function, second kind, order 0	K_0	$K_0(x)$	$-K_1(x)$
Modified Bessel[R] function, second kind, order 1	K_1	$K_1(x)$	$-K_0(x)-K_1(x)/x$
Bessel[D] function of order n	J	J{n}(x)	-J{n+1}(x)+n*(J{n}(x)/x)
Bessel[D] function of the second kind, order n	Y	Y{n}(x)	-Y{n+1}(x)+n*(Y{n}(x)/x)
Modified[D] Bessel function of order n	I	I{n}(x)	I{n+1}(x)+n*(I{n}(x)/x)
Modified[D] Bessel function, second kind, order n	K	K{n}(x)	-K{n+1}(x)+n*(K{n}(x)/x)
Legendre polynomial[R]	P	P{n}(x)	((n+1)/(x^2-1))*(P{n+1}(x)-x*P{n}(x))
Legendre polynomial[R] of the second kind	Q	Q{n}(x)	((n+1)/(x^2-1))*(Q{n+1}(x)-x*Q{n}(x))
Associated Legendre polynomial[R]	P	P{n m}(x)	(((n+1)-m)*P{n+1 m}(x)-((n+1)*x)*P{n m}(x))/(x^2-1)
Associated Legendre polynomial[R] of the second kind	Q	Q{n m}(x)	(((n+1)-m)*Q{n+1 m}(x)-((n+1)*x)*Q{n m}(x))/(x^2-1)

[14] For all functions, marked with '[D]' symbol, only symbolic derivatives defined, they cannot be evaluated.