TMS Analytics Delphi Development Library

DEVELOPERS GUIDE

Jul 2018

Web: http://www.tmssoftware.com
Email: info@tmssoftware.com

Index

Introduction

ANALYTICS is a Delphi library for developers. ANALYTICS library contains special classes to work with 'analytical' or 'symbolic' expressions in Delphi programs. It allows checking expressions syntax, parsing expressions, calculating expression value, evaluating symbolic derivatives and so on.

ADVANTAGES of ANALYTICS library:
1. 100% Delphi code.
2. Strongly structured class hierarchy.
3. Universal algorithms (working with formulae of any complexity).
4. Analytical (symbolic) derivative calculation.
5. Many predefined functions.
6. Easy to introduce new functions for any argument types.
7. Easy to overload operators for any argument types.
8. Working with Complex numbers, 3D vectors and tensors.
9. Working with indexed data (arrays, matrixes and higher dimensioned data).
10. Ready-to-use numerical tools, integrated with analytical capabilities.
11. Platform independent (many platforms supported – x32, x64, VCL, FMX).

Dependences

ANALYTICS Delphi library depends on:
1. RTL.
2. DRTE (Delphi Run-Time Environment) library.
3. MATHEMATICS library.

Extensions

There are the following extensions of ANALYTICS Delphi library:
1. Analytics Float (depends on MATHEMATICS library).
2. Analytics Derivatives.
3. Analytics Complex (depends on MATHEMATICS library).
4. Analytics Linear Algebra (depends on MATHEMATICS library).
5. Analytics Special (depends on MATHEMATICS library).
6. Analytics Fractions (depends on MATHEMATICS library).
7. Analytics Numerics (depends on MATHEMATICS library).
8. Analytics Statistics (depends on MATHEMATICS library).

Basic concepts

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

Expressions

A **mathematical expression** is a syntactically correct sequence of elements. Syntax defined with basic math rules. Using these rules, we can evaluate the expression, simplify it or make other manipulations. A simple example of expression is 'x+y'. If we know values of x and y we can calculate their sum. Generally, math expressions contain such elements as **constants** (**literals**), **variables**, **operators**, **functions**. The result of an expression calculation depends on elements the expression consists of. For example, if 'x' and 'y' are real numbers (variables) – the result of the given sum is a real number. If one of the variables is complex, the result will be complex too.

Literals

A **literal** is an unnamed constant value or a symbol, standing for 'standard' constant value. For example, in the expression **'2*(x+y)'** – **'2'** is a numerical literal. ANALYTICS library recognizes **Real** and **Complex** literals.

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

A complex literal consists of a real and an imaginary parts separated by '**+**' or '**-**' symbol. A real part is just a real literal. An imaginary part is a real literal plus imaginary unit symbol '**I**'. This symbol stands for the **imaginary unit** value. **NOTE**: there is no multiplication symbol between a real literal and the imaginary unit symbol. Complex literal examples: '**I**', '**-4I**', '**2+3.2I**', '**-2e2-4.45I**', '**2.45+I**', '**-1.2e-3+4.5e+2I**'.

ANALYTICS library supports the following 'standard' constants: 'e' – Euler number; 'π', 'Pi' – Pi; '∞' – positive infinity.

Variables

A **variable** is a named value. The value of a variable can be changed during its lifetime. A variable name can include alphanumeric symbols only and underscore symbol (the first symbol can be letter only). The value of a variable can be of any type. The type of a variable cannot be changed during its lifetime. In mathematical expressions, a variable value is accessed via the

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

variable name. That is, in an expression the variable name stands for the current variable value. For example, let the variable '**A**' is a real variable whose current value is '**1.0**', then the result of 'A+1' expression evaluation is '**2.0**'. Changing variable values allows calculation of the same expression for various variable values.

Operators

Syntactically an **operator** is a symbol standing for some mathematical operation. For example, in the expression 'x+y' the operator **'+'** stands for the addition operation. From the functional point of view, an operator implements some action with data value(s), called **operand(s)**, and returns the result of the operation.

All operators can be divided into many categories. The most common used are **unary** and **binary** operators. An **unary operator** has one operand and can be **prefix** (stands in front of its operand) and **postfix** (stands behind its operand). An example of an unary prefix operator is the **negation operator** ('**-x**', '**-**' is the operator, '**x**' is the operand). An example of an unary postfix operator is the **factorial operator** ('**n!**', '**!**' is the operator, '**n**' is the operand). **Binary operators** have two operands and commonly stand between them. An example of a 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 example, in the expression '**x+y*z**' the first operation performed is the multiplication of '**y**' and '**z**', then the sum of '**x**' and the product's result is calculated. This is because the '*****' operator is of higher precedence than the '**+**' is. The order of calculation can be changed by using parentheses '**()**'. The **associativity** determines how operators with the same precedence are grouped in an expression. Let us consider the expression '**2^3^4**' (where the '**^**' operator stands for the power). The result of the expression evaluation depends on how it 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).

ANALYTICS library supports all basic algebraic operators ('**+**', '**-**', '*****', '**/**', '**^**'), logical operators ('**&**', '**\|**', '**¬**'), relational operators ('**>**', '**<**', '**≡**', '**≠**' etc.). There are also some special operators those can be used in advanced cases. Total list of defined operators can be found in **Appendix A**.

ANALYTICS introduces the following syntax rules for the operators:

- Algebraic binary operators have precedence as determined with common math 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.
- Unary operators have higher precedence than binary operators.

- Postfix operators have higher precedence than prefix operators.
- Binary operators are all **left-associative**.

All supported operators can be overloaded (defined) for any operand types. The following restrictions are applied for operator overloading:
- New operators cannot be defined.
- Number of operands cannot be changed.
- Attributes of operators (precedence and associativity) cannot be changed.

Functions

Syntactically a **function** is a named operation with some data value(s) called argument(s). 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 argument.

In addition to arguments, a function can have parameters. For example, the logarithm of '**a**' to base '**b**' function $\mathbf{log_b(a)}$ has one argument '**a**' and one parameter '**b**'. Semantically parameters have the same meaning as 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 enclosed in parentheses '**()**'.
- Function parameters are enclosed in braces '**{}**'.
- Function arguments and parameters are separated by spaces ' '.
- If some function has no parameter, parameter braces are not mandatory.
- Argument parentheses are always mandatory.
- A function can have many parameters and/or many arguments.
- There can be many functions with the same name but 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) – associated Legendre function (two parameters '**n**' and '**m**', one argument '**x**').

ANALYTICS library contains many predefined functions of real and complex arguments. Total list of basic functions can be found in **Appendix A**.

Indexing

ANALYTICS library allows using **indexing** in expressions. **Indexing** is a method of accessing separate elements of a structured data. An example of a structured data is the array. Each array element has an unique index (or several indexes) by which the element value is accessed.

By syntax rules of ANALYTICS library, the indexes are written in square brackets '[]'. For example, the i-th array element can be written as '**A[i]**'. For multiple indexes, each index must be written in separate brackets. For example, a matrix element can be written as '**M[i][j]**'.
ANALYTICS syntax rules allow **slicing** implementation. **Slicing** is a method to get some part of a structured data, which is also a structure. Slicing syntax is implemented via index omitting. Let we have a matrix (two-dimensional array) '**M**'. Then, according to the slicing syntax, '**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 be applied to variables only, the variables must implement indexing interface.
- Many indexes allowed, each index must be enclosed in separate brackets '**[]**'.
- All index expressions must return values of real type only, the values must be close to integer numbers.
- Some indexes can be omitted that means slicing, slicing can only be applied to variables those implement slicing interface.[2]

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

[2] About indexing and slicing implementation see the 'Extending ANALYTICS' part.

Syntax summary

- Any math expression can contain literals, variables, operators, functions and indexing.
- Real and complex literals supported. Some common named math constants supported, like Euler number, Pi.
- Variable names can include only alphanumeric symbols and underscore symbol.
- Only predefined operators can be used in expressions, new operators cannot be defined.
- Operations are performed in order of the precedence. Operations order can be changed using parentheses '**()**'.
- Functions have parameters and arguments. Parameters are enclosed in braces '**{}**', arguments are enclosed in parentheses '**()**'. Parameters and arguments are separated by spaces ' '.
- Indexing can only be applied to variables those implement special interface. Indexes are enclosed in brackets '**[]**' (each index in separate brackets). Slicing is implemented by omitting some of the 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.

Using ANALYTICS

The main functionality of ANALYTICS library is evaluation of mathematical expressions. The functionality realized in the ***Translator***[3] class. This class encapsulates a set of variables and a set of operations (registered operators and functions). Thus, it can parse string expressions and calculate their values for current variable values.

The ***Translator*** class is not static. Various instances can have different operation and variable sets. Therefore, to use the functionality an instance of the class must be created. Hereinafter it is supposed that the instance 'translator' of the ***Translator*** class has been created

```
var
  translator: TTranslator;
  …
begin
  translator:= TTranslator.Create;
  …
```

Working with variables

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

To add a variable use the ***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

```
var
  name: string;
  v: TFloat;
begin
  name:= 'a';
  v:= 1.0;
  translator.Add(name, v);

…
```

- adding Real array variable

```
var
```

[3] Hereinafter in the text (not code) prefix 'T' is omitted for all class names.

```
    name: string;
    v: TArray<TFloat>;
  begin
    name:= 'A';
    v:= TArray<TFloat>.Create(-1.0, 0.0, 1.0, 2.0);
    translator.Add(name, 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 its ***Value*** property.

```
var
  name: string;
  a: TFloat;
  v: TVariable;
begin
  …
  name:= 'a';
  a:=1.0;
  v:= TRealVariable.Create(name, a);
  translator.Add(v);
  // some code here...
  v.Value:=TValue.From<TFloat>(2.0); // changing variable value
                                     // by direct reference
…
```

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

```
var
  x, y: TVariable;
begin
  …
  x:=translator.Get('x');             // get the reference by name
  x.Value:=TValue.From<TFloat>(2.0); // changing variable value
                                     // by got reference
  // some code here...
  y:=translator.Get(1);               // get the reference by index
  y.Value:=TValue.From<TFloat>(3.0); // changing variable value
                                     // by got reference
  …
```

NOTE about changing variable values: **do not** change the type of a variable value. The value type is determined at the moment of the variable creation and **must not** be changed during its lifetime. In spite of the property ***Value*** of the ***Variable*** class is of type ***TValue*** (from standard Delphi unit RTTI), when setting new variable value its type must be **the same** as the initial type is.

Use the ***Delete*** method of the ***Translator*** class to remove variables. Name or index of the variable can be used for this purpose. Use the ***DeleteAll*** method to remove all variables. The following code demonstrates the process of variable deleting.

```
…
translator.Delete('x'); // deleting variable with 'x' name
// some code here...
translator.Delete(1); // deleting variable with index 1
f:=translator.Calculate('u+z').AsType<TFloat>;
// some code here...
translator.DeleteAll; // removing all variables
…
```

Evaluating expression values

The simplest way to evaluate an expression value is using the ***Calculate*** method of the ***Translator*** class. The following code demonstrates this case.

```
var
  r: TFloat;
  c: TComplex;
  f1, f2: string;
begin
  …
  f1:= 'sin(a)^2+cos(a)^2';
  r:=translator.Calculate(f1).AsType<TFloat>;
  f2:= 'exp(z)-I*sin(z)';
  c:=translator.Calculate(f2).AsType<TComplex>;
  …
```

In the code above, it is supposed, that there are the variables “a” and “z” in the translator’s variable set. The variable “a” is of real type and the variable “z” is a complex one.
Note, that the return type of the ***Calculate*** method is ***TValue*** and therefore it can return the value of any type, depending on the expression contents. So, the returned value must be directly casted to the required type for using in the subsequent code.

The ***Calculate*** method can be rather slow. 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 the ***Calculate*** method is recommended for single expression evaluation only.

Another case of formula evaluation comes from the need to calculate one single expression for several values of the variables. In this case, it is recommended to use the following algorithm:

1. Create a ***Formula*** object for the string expression.
2. Change the variable values.
3. Calculate the formula value for the current variable values.
4. Return to the point 2 until all values calculated.

The ***Formula*** class intended for internal program representation of parsed mathematical expressions. This class contains information about the operations to evaluate the expression and implements the final evaluation algorithm.
The following code demonstrates how to use the **Formula** class for calculation of a table of function values for various argument values.

```
var
  s: string;
  f: TFormula;
  v: TFloat;
  x: TVariable;
  ax, ay: TArray<TFloat>;
  i: Integer;
begin
  …
  x:=translator.Get('x');  // getting reference to the variable
  s:= '2*(sin(x)+cos(x))';
  f:=translator.BuildFormula(s); // parsing string expression

  SetLength(ax, 100);
  SetLength(ay, 100);
  v:=0.0;
  for i:=0 to 99 do
  begin
    ax[i]:=v;
    // setting new variable value
    x.Value:=TValue.From<TFloat>(v);
    // calculating formula value for current x value
    ay[i]:=f.Calculate.AsType<TFloat>;
    // incrementing the value by step
    v:=v+0.01;
```

```
  end;
  …
```

In the code above, the string expression is parsed only once using the ***BuildFormula*** method. It returns an instance of the ***Formula*** class. Then the instance is used for evaluating the built formula many times for various 'x' 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 syntax of expressions before calculation.

The syntax correctness checking includes three steps. The first step is checking syntax rules, those do not need the expression to be parsed. For example, the parentheses 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 the ***CheckSyntax*** method of the ***Translator*** class. This function returns true if all rules are fulfilled and throws an exception if not.

The second step is to check, that an expression can be decomposed into a sequence of known expression types. For example, the string 'sin(x+1)' can be decomposed as a function with name 'sin' and one argument 'x+1'. The argument is itself a known expression – addition operation for 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 checks the types of all intermediate results to be sure, that all operations can be performed. For 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 implemented in the ***BuildFormula*** method of the ***Translator*** class. 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:

```
var
  s: string;
  f: TFormula;
```

```
  r: TFloat;
begin
  …
  s:= '2*(sin(x)+cos(x))';
  try
    // the first step of syntax checking
    if translator.CheckSyntax(s) then
    begin
      // the second and the third steps of syntax checking
      f:=translator.BuildFormula(s);
      if Assigned(f) then
      begin
        // here the formula calculation code
        // using f instance.
        r:=f.Calculate.AsType<TFloat>;
      end;
    end;
  except
    on ex: Exception do
    begin
      // here the exception handling code.
    end;
  end;
…
```

Analytical derivative calculation

One of the advanced features of ANALYTICS library is analytical (symbolic) calculation of derivatives for mathematical expressions. The ***Translator*** class contains the following method:

```
function Derivative(const formula, vName: string): string;
```

This method calculates analytical derivative of the *formula* by variable *vName*. The result is the symbolic representation of the derivative.

Here are some example codes of analytical derivative calculation:

```
var
  formula, derivative: string;
begin
  …
```

```
// #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-2)
…
```

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

1. The *formula* parameter in the ***Derivative*** method must be syntactically correct. The correctness can be checked by the ***CheckSyntax*** method.
2. The ***Derivative*** method does not require that the '*vName*' variable (or any other) is existing. The method manipulates variable names only, not their values.
3. The derivative method does some simplification after calculation (see the next part). However, some results can be not 'beautiful to see'. Nevertheless, the result symbolic expression for the derivative is **syntactically and mathematically correct**.
4. Not all operators supported for derivative calculations. For example, '**!**' operator not supported, because there is no derivation rule for it. However, these operators can be used in expressions and the derivative successfully evaluated if the operands of such operators do not depend on the variable.
5. It is not possible to introduce new derivation rules for operators (without modifying the core library).
6. Most of the basic special and transcendental functions supported for derivative calculation. Derivation rule can be introduced for any function (see below).

Symbolic expression simpification

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

```
function Simplify(const value: string): string;
```

This method simplifies the input symbolic expression and returns the simplified value as the result. The following 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.
- Reduce real (nonstandard) constants in sum expressions.
- Reduce real (nonstandard) constants in product expressions.
- Reduce real (nonstandard) constants in power expressions.

Here are some example codes of analytical expression simplifications:

```
var
  formula, simplified: string;
begin
  …

  // #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-x+2)+B*ln(2*e/x)
  …
```

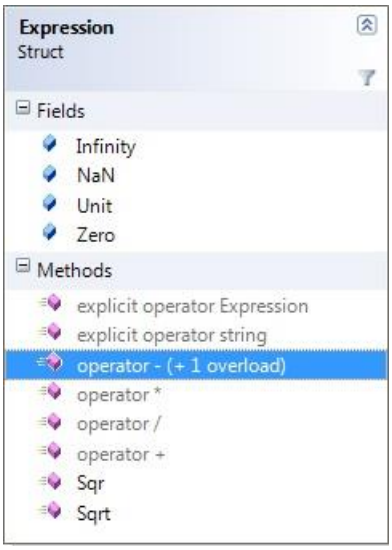
NOTE: the standard constants such as the Euler number and Pi remains named literals and not applied for simplification process.

Explicit string expressions manipulation

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 Delphi strings. For the above example strings, the following steps must be done: enclose the first expression into parentheses, enclose the second expression into parentheses; 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 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. In addition, it overloads operators for algebraic operations '**+**', '**-**', '***', '**/**'. This implementation allows manipulating string expressions in 'natural' manner.



Picture 3.1. Expression record.

Main advantages of using the ***Expression*** structures are demonstrated in the following example code:

```
var
  f1, f2, r1, r2: string;
  e1, e2, e: TExpression;
begin
  …
  f1:='2*x-1';
  f2:='x+1';
  e1:=TExpression(f1);
  e2:=TExpression(f2);
  …
  e:=e1*e2;
  r1:=string(e); // r1='(2*x-1)*(x+1)'
  …
  e:=TExpression.Power(e2, e1);
  r2:=string(e); // r2='(x+1)^(2*x-1)'
  …
```

```
end;
```

In the example above two expression created e1 and e2 (by explicit conversion from string). Then some manipulations done to get new expressions. The manipulations implemented in 'natural' manner (for example e1*e2) using overloaded operators. The result expressions are syntactically correct – the sum expressions are enclosed in parentheses to follow the operator precedence.
NOTE about using the ***Expression*** structure: all string values to manipulate with the ***Expression*** record must be syntactically correct; else, an exception will be thrown. The syntax can be checked by the ***CheckSyntax*** method.

Managing registered operators, functions and derivatives

ANALYTICS library uses the Delphi reflection mechanisms (RTTI information) to find and register the classes of analytical operations: operators, functions, derivatives. The search is made over the current RTTI context, so the set of the registered operations depends on the classed loaded into the context at the time. The operations registration is made for each instance of the ***Translator*** class in the constructor call.
There are two different cases of building applications: with run-time packages and without them. The first case implies that there are different built packages with analytical operations, so called extensions (see the **Introduction** part). In this case, the set of registered operations depends on the loaded run-time packages. For example, if an application is intended to use operations with complex number, the **Analytics.Complex** package must be included in the list of the application run-time packages, and this package must be loaded at the start. If the dynamic package loading is used in an application, the registered operation set can be updated calling the ***UpdateDefault*** method of the translator's property ***Operations***.
The second case with no run-time packages is slightly different. If the application is a standalone executable, the only classes, those have a direct reference in the source code, are included into the compiled code. The library provides a simple way of including all required operations set into an executable. Each of the library extensions includes the assembly 'pas' file. For example, if an application is supposed to use analytical derivative, it must include the assembly file into a uses list and call the ***Analytix.Derivatives.Assembly.InitializeAssembly*** function. Any library extension can be included into the executable by the same way.

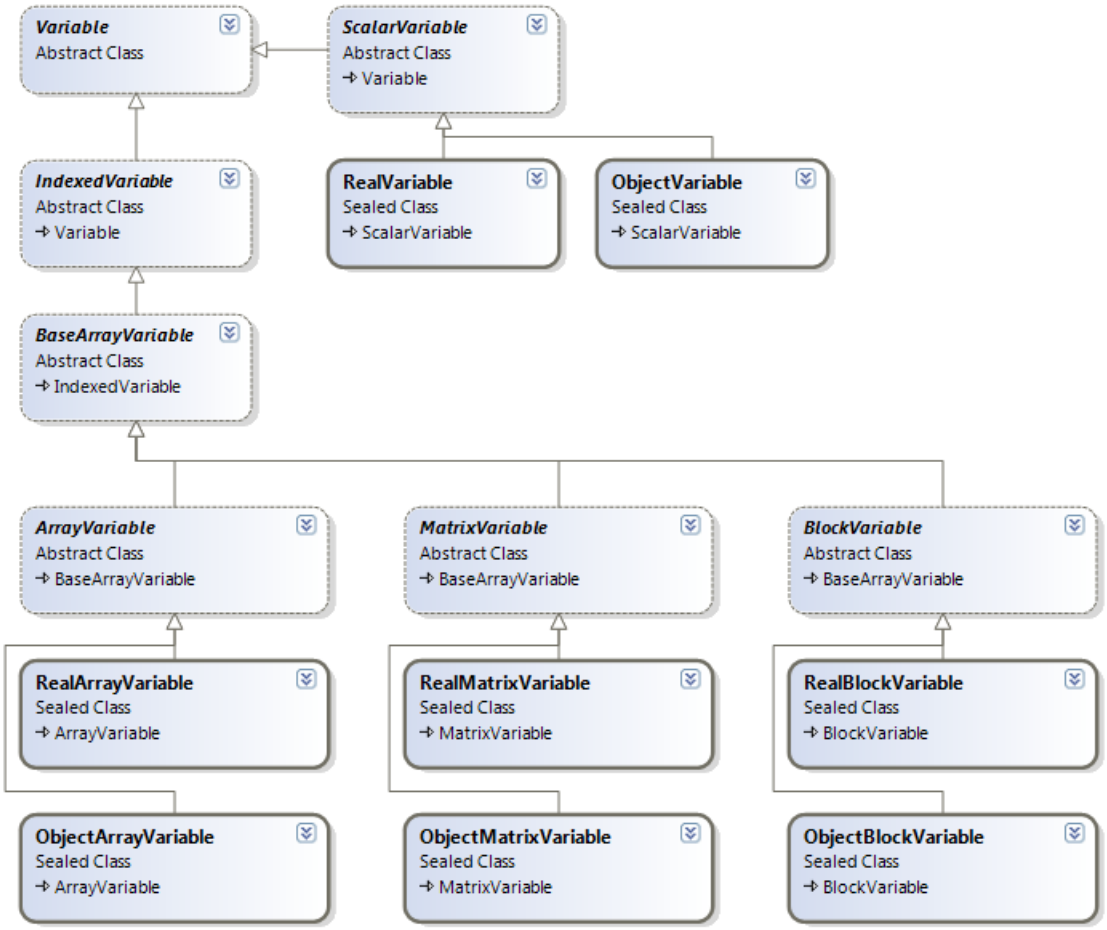
Class Hierarchy

This part explains the base class hierarchy of ANALYTICS library. There is no need to know the hierarchy for using main features of the library: math expressions evaluation and symbolic derivatives calculation. The knowledge is only useful for extending the library - introducing new function, overloading operators and creating new function derivation rules.

The main concept of ANALYTICS library is easy external extensibility. This means that the library has the complete core. The core algorithms have not to be changed to add new functionality. The functionality can be added by attaching new external libraries (packages). So there is strongly structured class hierarchy to implement this concept.

Variable classes

A variable is a name with associated value of some type. The base abstract class for all variables is ***Variable***. Its main properties are: ***Name***, ***Value*** and ***ValueType***.
Two abstract classes inherited from the ***Variable*** class (pic. 2.1): ***ScalarVariable*** and ***IndexedVariable***. The ***ScalarVariable*** class contains one value (scalar). The ***IndexedVariable*** class introduces base interface for an indexed value. An indexed value contains other values accessed by indexes.
The following classes inherited from the ***ScalarVariable***: ***RealVariable*** – contains a real value, ***ComplexVariable*** – contains a complex value, ***BooleanVariable*** – contains a boolean value, ***ObjectVariable*** – contains a value of any type.
The ***BaseArrayVariable*** class is inherited from the ***IndexedVariable***. This is an abstract class for variables those contain arrays of any dimension. Then, three abstract classes ***ArrayVariable***, ***MatrixVariable*** and ***BlockVariable*** realize interfaces for one-, two- and three-dimensional arrays accordingly. And finally, classes for real, complex, boolean and object arrays are realized up to the third dimension.



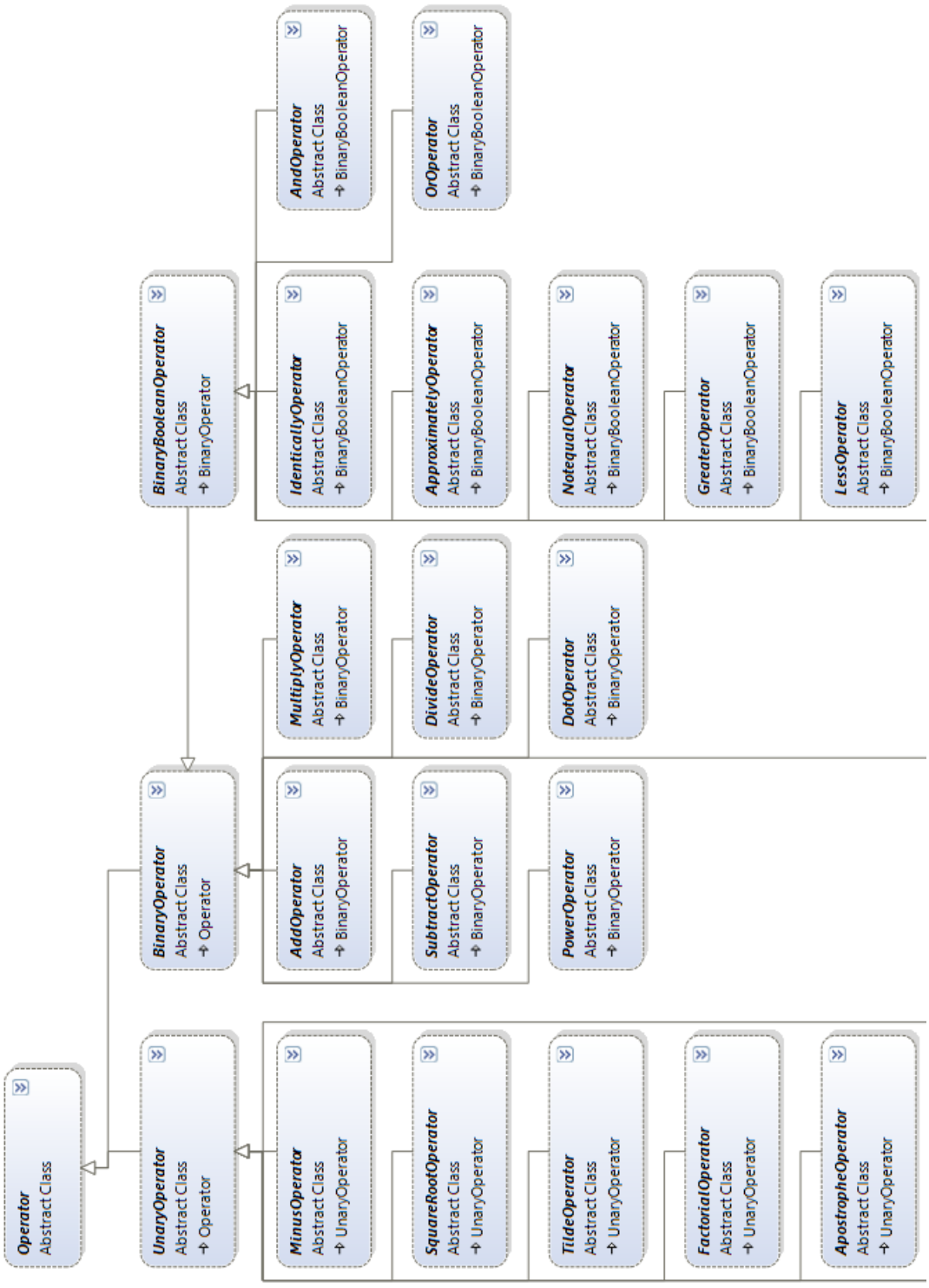
Picture 2.1. Variable class hierarchy diagram.

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

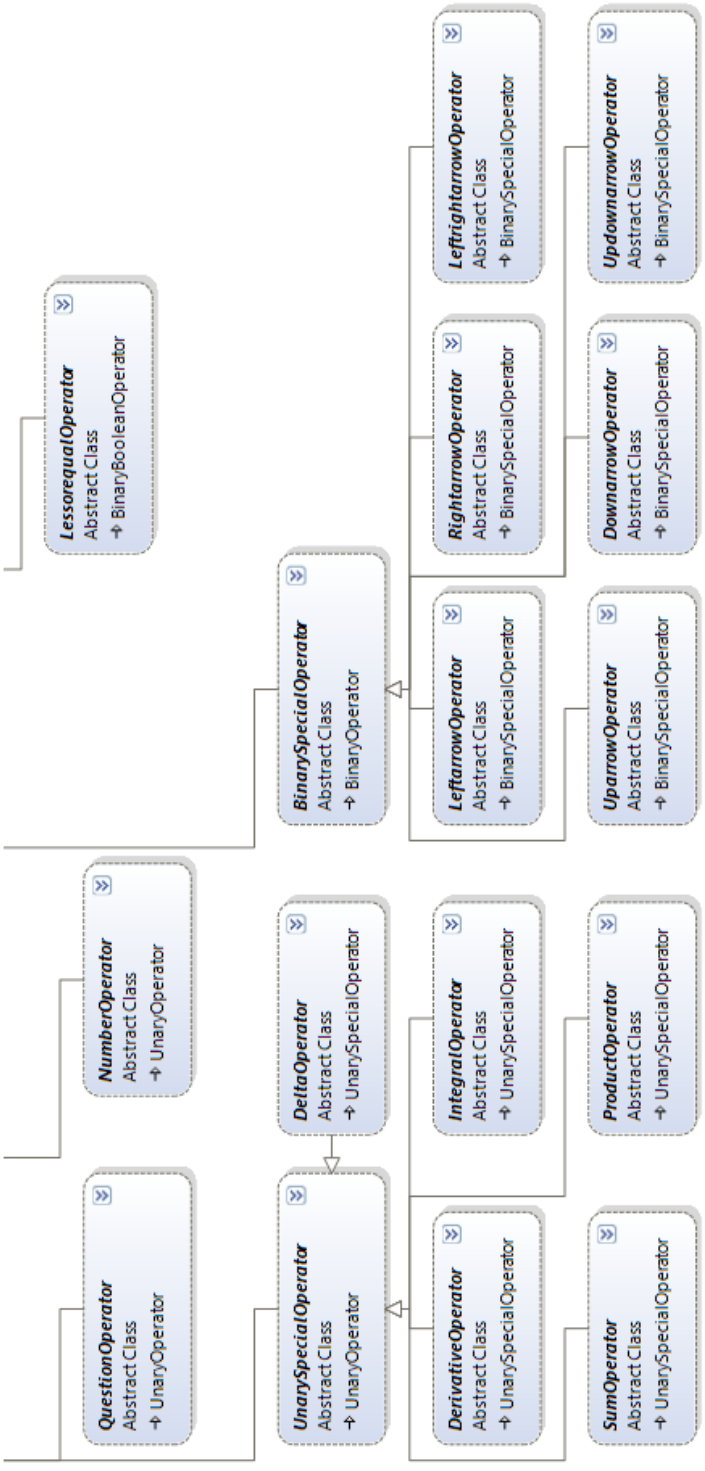
There are other variable types realized in extension libraries (as 3D vector and tensor variables and). They can be used in the same way as variables built in the core of ANALYTICS library.

Operator classes

An operator is a symbol with associated operation on some value(s). Base abstract class for all operators is ***Operator***. This class has properties and methods to define the symbol of the operator and the operation result type. Also it contains a reference to the function to calculate result value.



Picture 2.2. Operator class hierarchy diagram.



Picture 2.2. Operator class hierarchy diagram.

Two abstract classes inherited from this base class (pic. 2.2): ***UnaryOperator*** and ***BinaryOperator***. These classes introduce interface to define operand(s) type(s). More specific classes, inherited from the last ones, define the operator type and its symbol ('+', '^' and so on).

The final 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 and complex operands. Operators for other special operand types (3D vectors and tensors and other) realized in ANALYTICS extension libraries.

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

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 abstract class ***MonotypeFunction*** is directly inherited from the ***Function*** class (pic. 2.3). It implements the concept of a function with one same type for all parameters, arguments and the returned value.

Picture 2.3. Function class hierarchy diagram.

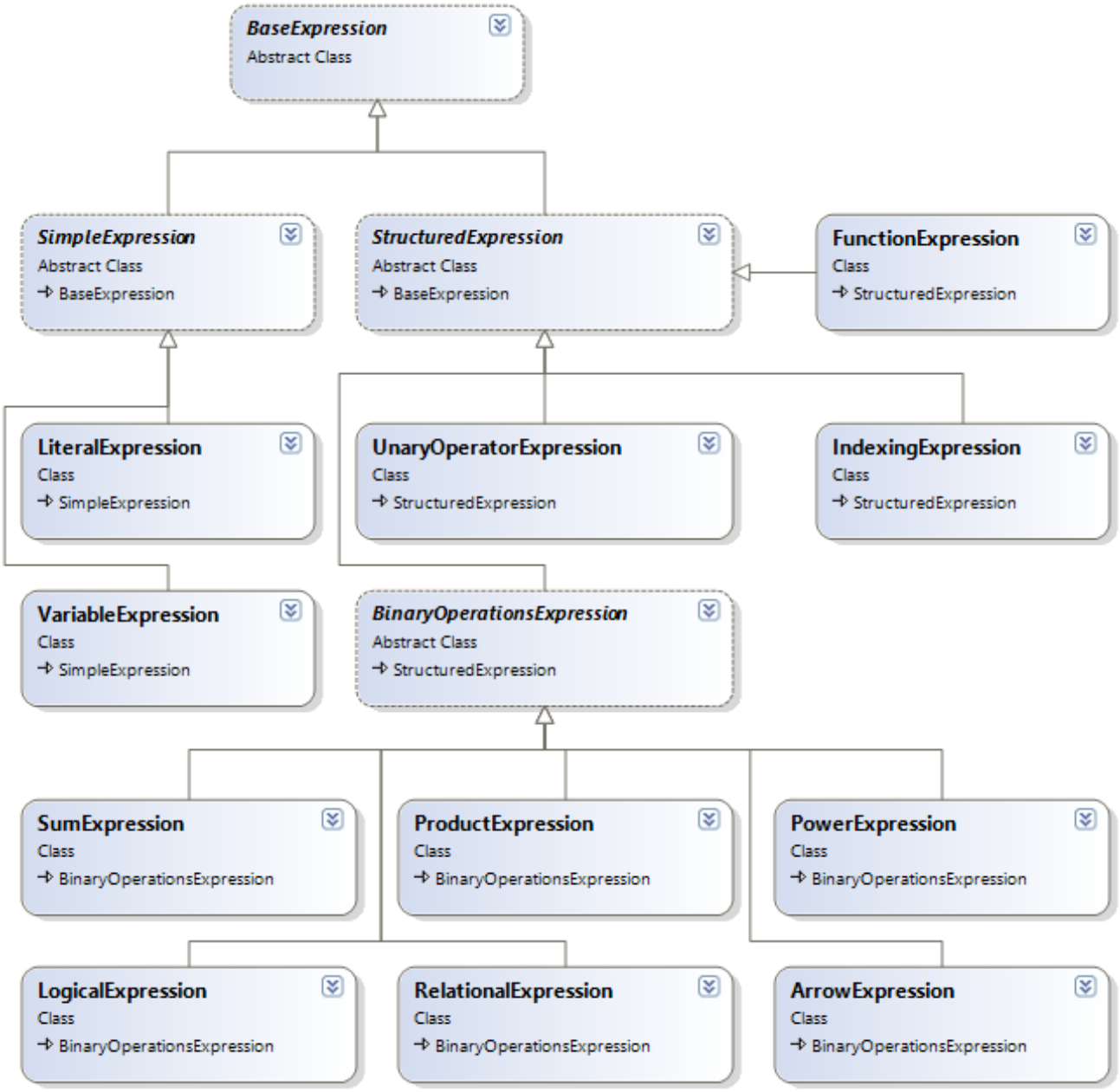
At the next level of the hierarchy all functions are divided into ***Elementary*** and ***Special***. This division is to correspond with mathematics, not for programming convenience. Further, each of the classes is divided into ***Real*** and ***Complex*** subclasses (functions with real and complex arguments accordingly). And finally, all classes are divided into ***Simple*** and ***Parametric*** functions. Simple functions have one argument, parametric functions have one parameter and one argument.

The final classes those realize the calculation are not shown on the diagram. There are many realized elementary and base special functions of real and complex arguments – algebraic, trigonometric, inverse trigonometric, exponential, logarithmic, hyperbolic, inverse hyperbolic. Total list of basic functions can be found in **Appendix A**.

Expression classes

Expression classes are intended to represent parsed string data inside the core algorithms of ANALYTICS library.
The base abstract class for all expressions is ***BaseExpression.*** This class defines an 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 classes are divided into **simple** expressions (do not contain other expressions) and **structured** expressions (do contain other expressions). The simple expressions are ***LiteralExpression*** and ***VariableExpression***. The structured expressions include ***FunctionExpression***, ***IndexingExpression***, ***UnaryOperatorExpression*** and ***BinaryOperationsExpression*** (***LogicalExpression***, ***RelationalExpression***, ***SumExpression***, ***ProductExpression***, ***PowerExpression***, ***ArrowExpression***), ***ArrayExpression*** (***VectorExpression***, ***MatrixExpression***). 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 ones.

There is no need to know the hierarchy of the expression classes to use the functionality of ANALYTICS. The hierarchy is a part of core algorithms and have to be used only for extending functionality of the library in the sense of analytical derivative calculation (see below).



Picture 2.4. Expression class hierarchy diagram.

Extending ANALYTICS

ANALYTICS library provides the complete core for parsing and calculating mathematical expressions. It also contains many predefined functions: trigonometric, hyperbolic and others and many realized operators for real and complex arguments. Thus, the library can be used without any modification for its main purpose: provide an end user application with the interface to input data in the form of mathematical expressions.

Another goal of the library is to provide an interface for using in math expression not only 'base' number types (real and complex), but any program specific data types. For example, let there is a program for signal processing. The program must allow the end user to make various math operations with signals – add two signals, multiply a signal with a real value, calculate exponents and logarithms of the signals and so on. ANALYTICS library suggests the easiest way to create such program. The core algorithms of ANALYTICS library work with expressions containing data of any type (for example, the signals). The only thing required is to define the operations with the data.

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) presented inside the library as classes. To add an operation with program specific data a descendant of some class must be implemented. The library will use this class to work with expressions, containing the program specific data.

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

Overloading operators

To overload an operator for program specific data operand types, a 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 addition operator overloading for complex numbers:

```
type
  /// <summary>
  /// Compex + Compex = Compex
  /// </summary>
  TComplexAdd=class sealed(TAddOperator)
  protected
    function GetOperand1Type(): PTypeInfo; override;
    function GetOperand2Type(): PTypeInfo; override;
```

```
    function Operation(const operand1, operand2: TValue): TValue;
override;
    function GetReturnType(): PTypeInfo; override;
  public
    class function IsRealized: boolean; override;
  end;
…
function TComplexAdd.GetOperand1Type: PTypeInfo;
begin
  Result:=TTypes.ComplexType;
end;

function TComplexAdd.GetOperand2Type: PTypeInfo;
begin
  Result:=TTypes.ComplexType;
end;

function TComplexAdd.GetReturnType: PTypeInfo;
begin
  Result:=TTypes.ComplexType;
end;

class function TComplexAdd.IsRealized: boolean;
begin
  Result:=true;
end;

function TComplexAdd.Operation(const operand1, operand2: TValue):
TValue;
var
  x1, x2, x: TComplex;
begin
  x1:= operand1.AsType<TComplex>;
  x2:= operand2.AsType<TComplex>;
  x:= x1+x2;
  result:= TValue.From<TComplex>(x);
end;
```

The ***ComplexAdd*** class is inherited from the ***AddOperator***, this means it overloads the addition "+" operator. The class overrides ***GetOperand1Type***, ***GetOperand2Type*** and ***GetReturnType*** methods. They define that the first operand, the second operand and the operation result is of type ***Complex***. Another overridden method is ***Operation*** – this method implements the

operation itself (adds two complex numbers). The overridden ***IsRealized*** method returns true, it means that the class is fully realized and can be used in calculations.

The following example demonstrates overloading of the multiplication "*" operator for a complex number and a real value:

```
type
  /// <summary>
  /// Compex * Float = Compex
  /// </summary>
  TComplexRealMultiply=class sealed(TMultiplyOperator)
  protected
    function GetOperand1Type(): PTypeInfo; override;
    function GetOperand2Type(): PTypeInfo; override;
    function Operation(const operand1, operand2: TValue): TValue;
override;
    function GetReturnType(): PTypeInfo; override;
  public
    class function IsRealized: boolean; override;
  end;
…
function TComplexRealMultiply.GetOperand1Type: PTypeInfo;
begin
  result:=TTypes.ComplexType;
end;

function TComplexRealMultiply.GetOperand2Type: PTypeInfo;
begin
  result:=TTypes.RealType;
end;

function TComplexRealMultiply.GetReturnType: PTypeInfo;
begin
  result:=TTypes.ComplexType;
end;

class function TComplexRealMultiply.IsRealized: boolean;
begin
  result:=true;
end;
```

```
function TComplexRealMultiply.Operation(const operand1, operand2:
TValue): TValue;
var
  x1, x: TComplex;
  x2: TFloat;
begin
  x1:= operand1.AsType<TComplex>;
  x2:= operand2.AsType<TFloat>;
  x:= x1*TComplex.Create(x2);
  result:= TValue.From<TComplex>(x);
end;
```

The realized operators will be used to implement appropriate operations with the operands of defined types.

The same result, as in two previous code examples, can be achieved by using generic analogues of the operators. The following code demonstrates this case for overloading the power operator and the tilde operator (the conjugate of a complex value):

```
type
  /// <summary>
  /// Compex ^ Compex = Compex
  /// </summary>
  TComplexPower=class sealed(TGenericPowerOperator<TComplex, TComplex,
TComplex>)
  protected
    function TypedOperation(const operand1: TComplex; const operand2:
TComplex): TComplex; override;
  public
    class function IsRealized: boolean; override;
  end;

  /// <summary>
  /// ~ Compex = Compex
  /// </summary>
  TComplexConjugate=class sealed(TGenericTildeOperator<TComplex,
TComplex>)
  protected
    function TypedOperation(const operand: TComplex): TComplex;
override;
  public
    class function IsRealized: boolean; override;
```

```
  end;
…

class function TComplexPower.IsRealized: boolean;
begin
  result:=true;
end;

function TComplexPower.TypedOperation(const operand1, operand2:
TComplex): TComplex;
begin
  result:=TComplex.Power(operand1, operand2);
end;

class function TComplexConjugate.IsRealized: boolean;
begin
  result:=true;
end;

function TComplexConjugate.TypedOperation(const operand: TComplex):
TComplex;
begin
  result:=TComplex.Conjugate(operand);
end;
```

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

Explicitly overloaded operators

The operator overloading in ANALYTICS library is intended to implement operations on program specific data. Often, for such data, some operators are already overridden “inside” the program (that means Delphi operator overloading). ANALYTICS core algorithms support these “explicitly overloaded operators” for implementing appropriate operations on the data. In other words, if there is an “explicitly overloaded” operator, it will be used to calculate the result for the suitable operation. For example, let there is the ***Complex*** record type implementing complex number algebra and it overloads math operators, such as addition, subtraction and so on. Then, these operations will be performed with complex numbers in math expressions without deriving new operator classes.

There are some constraints for using “explicitly overloaded” operators:

1. Not all operators can be “explicitly overloaded”. The first reason is that not all ANALYTICS operators have analogous Delphi operators (for example the factorial operator “**!**”). The second reason is that operators “meaning” can be ambiguous (for example the power “**^**”). Considering this, only following operators can be “explicitly overloaded”: “**+**”, “**-**” (unary and binary), “*****”, “***/***”.
2. “Implicit” overloading (deriving new operator classes) has higher priority. That is, if there are both an “explicit” and an “implicit” operators, the second will be used to calculate the operation result.

Both overloading methods can be used in combination. The “implicit” method can be used as for complimenting the “explicit” method, as for overriding its behavior.

Introducing new functions

Introducing new functions is another way of extending the library functionality. A function can be defined to work with any data types. The function must have a valid name. The function can have any number and type of arguments (parameters) and any return type.

To introduce new function a descendant of one of the abstract function classes must be implemented. The choice of the base class depends on the number of arguments and parameters and on their types. For the common number types (real or complex), one of the predefined abstract classes can be used as the ancestor class: ***RealElementaryFunction***, ***RealParametricElementaryFunction***, ***ComplexElementaryFunction***, ***ComplexParametricElementaryFunction***.

As example, the code of the class, implementing sine function of complex argument, is listed below:

```
type
  /// <summary>
  /// Complex Sine function
  /// </summary>
  TComplexSine=class sealed(TComplexElementaryFunction)
  protected
    function GetName(): string; override;
    function Func(const x: TComplex): TComplex; override;
  public
    class function IsRealized: boolean; override;
  end;
…
{ TComplexSine }

function TComplexSine.Func(const x: TComplex): TComplex;
begin
```

```
  result:=TComplex.Sin(x);
end;

function TComplexSine.GetName: string;
begin
  result:='sin';
end;

class function TComplexSine.IsRealized: boolean;
begin
  result:=true;
end;
```

The class is inherited from the ***ComplexElementaryFunction*** because it implements a function with one complex argument. The overridden methods are ***GetName***, ***Func*** and ***IsRealized***. The first defines the function name (used in expressions). The second implements the function operation itself (calculates sine of a complex number). The last tells that the class is fully realized and can be used in calculation process.
Another example demonstrates the power function implementation of complex base and exponent:

```
type
  /// <summary>
  /// Power of Complex value
  /// NOTE: argument is the base and parameter is
  ///       the exponent of the power function.
  /// </summary>
  TComplexPower=class sealed(TComplexParametricElementaryFunction)
  protected
    function GetName(): string; override;
    function Func(const parameter, argument: TComplex): TComplex;
override;
  public
    class function IsRealized: boolean; override;
  end;
…

{ TComplexPower }

function TComplexPower.GetName: string;
begin
  result:='pow';
```

```
end;

class function TComplexPower.IsRealized: boolean;
begin
  result:=true;
end;

function TComplexPower.Func(const parameter, argument: TComplex):
TComplex;
begin
  result:=TComplex.Power(argument, parameter);
end;
```

The class is inherited from the ***ComplexParametricElementaryFunction*** because it implements a function with one complex parameter and one complex argument.

In general case, to introduce new function the class must be inherited from the base abstract class ***Function***. All its abstract methods must be overridden. The methods define the number and types of parameters and arguments and the function return type. The following code example demonstrates the implementation of Power function, where the base is a complex number and the exponent is a real number:

```
type
  /// <summary>
  /// Complex^Float Power function
  /// </summary>
  TComplexRealPower=class sealed(TFunction)
  protected
    function GetName(): string; override;
    function GetArgumentCount(): integer; override;
    function GetArgumentTypes(): TArray<PTypeInfo>; override;
    function GetParameterCount(): integer; override;
    function GetParameterTypes(): TArray<PTypeInfo>; override;
    function GetResultType(): PTypeInfo; override;
    function DoCalculate(parameters, arguments: TArray<TValue>):
TValue; override;
  public
    class function IsRealized: boolean; override;
  end;
…

{ TComplexRealPower }
```

```
function TComplexRealPower.GetName: string;
begin
  result:='pow';
end;

function TComplexRealPower.GetArgumentCount: integer;
begin
  result:=1;
end;

function TComplexRealPower.GetArgumentTypes: TArray<PTypeInfo>;
begin
  result:=TArray<PTypeInfo>.Create(TTypes.ComplexType);
end;

function TComplexRealPower.GetParameterCount: integer;
begin
  result:=1;
end;

function TComplexRealPower.GetParameterTypes: TArray<PTypeInfo>;
begin
  result:=TArray<PTypeInfo>.Create(TTypes.RealType);
end;

function TComplexRealPower.GetResultType: PTypeInfo;
begin
  result:=TTypes.ComplexType;
end;

class function TComplexRealPower.IsRealized: boolean;
begin
  result:=true;
end;

function TComplexRealPower.DoCalculate(parameters, arguments:
TArray<TValue>): TValue;
var
  z: TComplex;
  e: TFloat;
  r: TComplex;
```

```
begin
  z:=arguments[0].AsType<TComplex>;
  e:=parameters[0].AsType<TFloat>;
  r:=TComplex.Power(z, e);
  result:=TValue.From<TComplex>(r);
end;
```

The function has one complex argument, one real parameter and returns a complex value as the function result.

A little shorter way to introduce new function for some specific data types is using one of the base generic function classes. These classes use generic parameters to define the types of arguments and parameters. Thus, only the ***GetName*** method and the calculation algorithm must be provided by the descendant class. The following code demonstrates an example of a logarithm function implementation using the base generic class:

```
type
  /// <summary>
  /// Complex Logarithm function by Real base
  /// (NOTE: the Base of logarithm is the Parameter of the function)
  /// </summary>
  TRealComplexLogarithm=class
sealed(TGenericParametricFunction<TFloat, TComplex, TComplex>)
  protected
    function GetName(): string; override;
    function Func(const parameter: TFloat; const argument: TComplex):
TComplex; override;
  public
    class function IsRealized: boolean; override;
  end;
…

{ TRealComplexLogarithm }

function TRealComplexLogarithm.GetName: string;
begin
  result:='log';
end;

class function TRealComplexLogarithm.IsRealized: boolean;
begin
  result:=true;
```

```
end;

function TRealComplexLogarithm.Func(const parameter: TFloat; const
argument: TComplex): TComplex;
begin
  result:=TComplex.LogN(parameter, argument);
end;
```

The class is inherited from the ***GenericParametricFunction***, which means it has one argument and one parameter. The argument is a complex number, the parameter is a real number and the function result is a complex number. The first, the second and the third generic parameters provide this information accordingly.

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

NOTE about introducing new functions: many functions with the same name allowed if they have different count or/and type of arguments (parameters).

Implementing indexing

Indexing can be applied to variables only. A variable class, implementing indexing, must be inherited from the abstract class ***IndexedVariable*** (or from one of its descendants). The inherited class must override abstract methods defining the number of indexes, data access methods and slicing interface methods.

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

```
type
  /// <summary>
  /// Base abstract class for all matrix variables.
  /// NOTE: Slicing is implemented.
  /// </summary>
  TMatrixVariable<TheBaseType>
=class(TBaseArrayVariable<TArray<TheBaseType>>)
  protected
    function GetBaseType: PTypeInfo; override; final;
    /// <summary>
    /// 2 indexes
    /// </summary>
    function GetIndexCount: integer; override; final;
```

```
    /// <summary>
    /// Slicing is implemented for Matrix variables
    /// </summary>
    function GetSlicingImplemented(): boolean; override;
  public
    /// <summary>
    /// Sliced item is array of BaseType
    /// </summary>
    function GetItemType(indexes: TArray<integer>): PTypeInfo;
override; final;
    /// <summary>
    /// Implements Array Slicing
    /// </summary>
    function GetItemValue(indexes: TArray<integer>): TValue; override;
final;
  end;
…

{ TMatrixVariable<TheBaseType> }
…

function TMatrixVariable<TheBaseType>.GetBaseType: PTypeInfo;
begin
  result:= TypeInfo(TheBaseType);
end;

function TMatrixVariable<TheBaseType>.GetSlicingImplemented: boolean;
begin
  result:=true;
end;

function TMatrixVariable<TheBaseType>.GetIndexCount: integer;
begin
  result:= 2;
end;

function TMatrixVariable<TheBaseType>.GetItemType(indexes:
TArray<integer>): PTypeInfo;
begin
  if
((indexes[0]>=0)and(indexes[1]<0))or((indexes[1]>=0)and(indexes[0]<0))
then
```

```
    result:= TypeInfo(TArray<TheBaseType>)
  else
    result:= GetBaseType;
end;

function TMatrixVariable<TheBaseType>.GetItemValue(indexes:
TArray<integer>): TValue;
var
  i: integer;
  j: integer;
  a: TArray<TheBaseType>;
  n: integer;
begin
  if (indexes[0]>=0)and(indexes[1]<0) then
  begin
    i:= indexes[0];
    a:= System.Copy(FData[i]);
    result:= TValue.From<TArray<TheBaseType>>(a);
    exit;
  end
  else
    if (indexes[1]>=0)and(indexes[0]<0) then
    begin
      j:= indexes[1];
      n:= RowCount;
      SetLength(a, n);
      for i:= 0 to n-1 do
        a[i]:= FData[i][j];
      result:= TValue.From<TArray<TheBaseType>>(a);
      exit;
    end
    else
    begin
      result:=
TValue.From<TheBaseType>(FData[indexes[0]][indexes[1]]);
    end;
end;
```

The class overrides the ***GetIndexCount*** method which returns 2, because a matrix element has two indexes. The method ***GetSlicingImplemented*** returns true, it means that the variable supports slicing. The ***GetItemType*** method returns the type of indexed data, taking into account slicing implementation. If one of the 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 an array (matrix row or column). The ***GetItemValue*** method implements access to indexed data analogously to the ***GetItemType*** method (takes the slicing into account). The ***GetBaseType*** method returns the type of matrix elements (based on the generic parameter of the class).

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

Introducing function derivatives

ANALYTICS library allows introducing derivation rules for any function. It is another way of extending the library functionality. This case slightly differs from the extensions, described above. For example, the derivative rules cannot be redefined for operators, because these rules are the base of math and therefore they realized as the core algorithms of the library.

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

```
function GetFunctionName(): string; virtual; abstract;
function GetParameterCount(): integer; virtual; abstract;
function GetArgumentCount(): integer; virtual; abstract;
class function IsRealized: boolean; virtual; abstract;
function Derivative(context: TDerivativeContext; afunction: TFunctionExpression; const
vName: string): TBaseExpression; virtual; abstract;
```

The first three methods define the function signature, the derivative rule is applied for. Note that the types of parameters and arguments are not defined. This is due to the derivation process - it manipulates symbols only, not values or types.
The last method defines the derivation rule itself. This method takes *afunction* parameter of the ***FunctionExpression*** class and *vName* parameter – the name of variable for derivative. The *context* parameter is only to pass it into other derivative methods. The result of the method is ***BaseExpression*** object. Thus, the method must build the result expression from the input *function* expression.

The ***IsRealized*** method must be overridden in descendant classes and must return true for fully realized function derivative.

In the most common case of an elementary function with one argument, the derivative rule can be generalized, using the following mathematical formula (chain rule):

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

The ***SimpleFunctionalDerivative*** class implements this rule. This is an abstract class to define derivative rule for a function with one argument. It overrides the base ***Derivative*** method realizing the above chain rule formula. The class has another abstract method

function BaseDerivative(argument: TBaseExpression): TBaseExpression; **virtual**; **abstract**;

The *argument* parameter of the method is the g function in the formula above. It is enough to override the ***BaseDerivative*** method in a descendant class. The chain rule formula will be applied then automatically to get the total derivative expression for the function.
The implementation of the method is rather simple for most of the standard transcendental functions. For example, consider the code of the class, implementing the derivative for sine function:

```
type
  /// <summary>
  /// Sine derivative
  /// </summary>
  TSineDerivative=class sealed(TSimpleFunctionalDerivative)
  protected
    function GetFunctionName(): string; override;
    function BaseDerivative(argument: TBaseExpression):
TBaseExpression; override;
  public
    class function IsRealized: boolean; override;
  end;

…

function TSineDerivative.BaseDerivative(argument: TBaseExpression):
TBaseExpression;
begin
  result:= TFunctionExpression.CreateSimple('cos', argument);
end;

function TSineDerivative.GetFunctionName: string;
begin
  result:= 'sin';
end;
```

```
class function TSineDerivative.IsRealized: boolean;
begin
  result:= true;
end;
```

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 the expression classes (see class hierarchy) should be used to build the result expressions. There are many predefined useful methods to create sum, difference, product, division, power or other expressions from the existing ones.

In general case of a function with many arguments and parameters it is more complicated to define the algorithm for the function derivative implementation. It is because the derivative rules for that case can be **really** complicated. For example, consider the derivative of the logarithm function:

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

Nevertheless, the library allows defining derivation rules for any function. The methods of the expression classes (see class hierarchy) can be used to build the result expressions for such complicated derivatives. One can consider the source code of many predefined function derivatives in the library. The ***LogarithmDerivative*** class is the example of the code, implementing the complicated formula for logarithmic derivative.

The algorithm of parametric functions derivative can be simplified in the case, when the parameter(s) does not depend on the variable. Then, the 'chain rule' is applicable for the function. The specific class ***ParametricFunctionalDerivative*** introduced in the library for this case.

For example, consider derivative for the special Polygamma function (https://en.wikipedia.org/wiki/Polygamma_function). Its derivative defined by the following equation:

$$\frac{d}{dx}\left(Y^m(x)\right) = Y^{m+1}(x)$$

The code of the class, implementing the derivative is the following:

```
type
```

```
  /// <summary>
  /// Polygamma derivative
  /// </summary>
  TPolygammaDerivative = class sealed(TParametricFunctionalDerivative)
  protected
    function GetFunctionName(): string; override;
    function BaseDerivative(parameter, argument: TBaseExpression):
TBaseExpression; override; final;
  public
    class function IsRealized: boolean; override;
  end;
…
function TPolygammaDerivative.GetFunctionName: string;
begin
  result:= 'Y';
end;

class function TPolygammaDerivative.IsRealized: boolean;
begin
  result:= true;
end;

function TPolygammaDerivative.BaseDerivative(parameter, argument:
TBaseExpression): TBaseExpression;
var
  p1: TBaseExpression;
begin
  p1:= TSumExpression.MakeSum(parameter, TLiteralExpression.Unity);

  result:= TFunctionExpression.CreateParametric('Y', p1, argument);
end;
```

As can be seen from the class above, the code for the derivative is not complicated: only three methods must be overridden. The base class ***ParametricFunctionalDerivative*** realizes the rest: checks that the function's parameter does not depends on the variable; implements the 'chain rule'.

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 ***Analytix.Numerics.Approximation*** unit. The main method of the class is:

```
function Approximate(basis: T; vData: TArray<TArray<TFloat>>; fData:
TArray<TFloat>): TArray<TFloat>; virtual; abstract;
```

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 ***Analytix.Numerics.Basis*** unit. 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:

```
function F(vValues, cValues: TArray<TFloat>): TFloat; virtual;
```

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:

```
constructor Create(variables: TArray<string>; coefficient: string;
parameters: TArray<TVariable>; functions: TArray<string>);
```

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:

```
var
  approximator: TLinearApproximator;
  basis: TLinearBasis;
  variables: TArray<string>;
  functions: TArray<string>;
  cValues: TArray<TFloat>;
begin
variables:= TArray<string>.Create('x'); // 1
functions:= TArray<string>.Create('e^x', 'sin(x)', 'e^-x', 'x^2',
'3^x', 'sinh(2*x)'); // 2
basis:= TLinearScalarBasis.Create(variables, 'C', nil, functions); //3
approximator:= TLinearLeastSquares.Create; // 4
cValues:= approximator.Approximate(basis, vData, fData); // 5

// Use the basis and the calculated coefficients…
end;
```

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:

```
constructor Create(variable, coefficient: string; order, dimension: integer; parameters: TArray<TVariable>; f: string);
```

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:

```
var
  approximator: TLinearApproximator;
  basis: TLinearBasis;
  variable, func: string;
  parameters: TArray<TVariable>;
  exponents: TArray<TFloat>;
  A: TVariable;
  order: integer;
  cValues: TArray<TFloat>;
begin
  exponents:= TArray<TFloat>.Create(0.0, 0.5, -0.5, 2.5, -3.0, 4.5, -4.2); // 1
  order:= Length(exponents); // 2
  A:= TRealArrayVariable.Create('A', exponents); // 3
  parameters:= TArray<TVariable>.Create(A); // 4
  func:= 'e^(A*X[0])'; // 5
  basis:= TLinearVectorBasis.Create('X', 'C', order, 1, parameters, func); // 6
```

```
  approximator:= TLinearLeastSquares.Create; // 7
  cValues:= approximator.Approximate(basis, vData, fData); // 8
end;
```

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 (line 5), 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 (lines 3 and 4). 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 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:

```
constructor Create(variables, coefficients: TArray<string>;
parameters: TArray<TVariable>; f: string); overload;
```

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:

```
var
  approximator: TNonlinearApproximator;
  basis: TNonlinearBasis;
  variables: TArray<string>;
  coefficients: TArray<string>;
  func: string;
  opt: TSolverOptions;
  cValues: TArray<TFloat>;
begin
  variables:= TArray<string>.Create('x'); // 1
  coefficients:= TArray<string>.Create('A', 'B'); // 2
  func:= 'sin(A*x)*e^(B*x)'; // 3
  basis:= TNonlinearScalarBasis.Create(variables, coefficients, nil,
func); // 4
  approximator:= TGaussNewtonLeastSquares.Create; // 5
  approximator.C0:= TArray<TFloat>.Create(0.0, -1.0); // 6
  opt:= TSolverOptions.Create(true);
  opt.MaxIterationCount:= 100;
  opt.SolutionPrecision:= 0.02;
  approximator.Options:= opt; // 7
  cValues:= approximator.Approximate(basis, vData, fData); // 8

 // Use the basis and the calculated coefficients…
end;
```

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 ***Mathematix.Integration*** unit. The base classes for 1D and 2D integration are ***Integrator1D*** and ***Integrator2D*** accordingly. They have the following functions for definite integral calculation:

```
class function Integral(f: TFunction1D; const x1, x2: TFloat; n:
integer): TFloat; virtual;

class function Integral(f: TFunction2D; const x1, x2, y1, y2: TFloat;
nx, ny: integer): TFloat; virtual;
```

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 ***Analytix.Numerics.Functions*** unit. The classes are ***SymbolicFunction1D*** and ***SymbolicFunction2D*** for one- and two- dimensional cases accordingly. They have the following constructors:

```
constructor Create(const v, f: string; parameters: TArray<TVariable> =
nil);
constructor Create(const v1, v2, f: string; parameters:
TArray<TVariable> = nil);
```

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:

```
function F(const v: TFloat): TFloat;
function F(const x, y: TFloat): TFloat;
```

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

```
var
  sf1D: TSymbolicFunction1D;
  integrator: TIntegrator1DClass;
  ivalue: TFloat;
begin
  integrator:= TGauss3NodeIntegrator;
  sf1D:= TSymbolicFunction1D.Create('x', 'sin(x)*e^(x^2/8)');
  ivalue:= integrator.Integral(sf1D.F, 1.0, 3.0, 10);
  // Code for using integral value 'ivalue'…
end;
```

The integration algorithms is simple: choose the appropriate integrator class; create the integrand function instance; calculate the definite integral value.
NOTE: there is no need to create an instance of the integrator, because of using virtual class function for polymorphic behavior realization.

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 ***Mathematix.ODE.Solver*** unit. The base abstract class for solving the problems is ***ODESolver***. And the only its method is:

```
class function Solve(system: TODESystem; y0: TArray<TFloat>; const t1:
TFloat; N: integer; var t: TArray<TFloat>): TArray<TArray<TFloat>>;
virtual;
```

The method solves the initial value problem for specified ODE system[4], 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 ***Mathematix.ODE.System*** unit) class as the first parameter. The main method of the class is the following:

function Evaluate(**const** t: TFloat; y: TArray<TFloat>): TArray<TFloat>; **virtual**;

[4] 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 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:

```
constructor Create(const v: string; fv, equations: TArray<string>;
parameters: TArray<TVariable> = nil);
```

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:

```
var
  solver: TODESolverClass;
  ode: TAnalyticalODE;
  fv, equations: TArray<string>;
  y0: TArray<TFloat>;
  t: TArray<TFloat>;
  y: TArray<TArray<TFloat>>;
begin
  fv:= TArray<string>.Create('y');
  equations:= TArray<string>.Create('2*sin(t)+t/y');
  ode:= TScalarODE.Create('t', fv, equations);
  solver:= TRungeKutta4Solver;
  y0:= TArray<TFloat>.Create(1.0);
  y:= solver.Solve(ode, y0, 10.0, 1000, t);
  // Code for using ‘y’ values …
end;
```

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:

```
constructor Create(const v, fv, equation: string; dimension: integer;
parameters: TArray<TVariable> = nil);
```

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:

```
var
  t1: TFloat;
  N: integer;
  solver: TODESolverClass;
  ode: TAnalyticalODE;
  A, B: TVariable;
  ma: TArray<TArray<TFloat>>;
  vb: TArray<TFloat>;
  prms: TArray<TVariable>;
  y0, t: TArray<TFloat>;
  y: TArray<TArray<TFloat>>;
begin
  // Create parameter variables.
  SetLength(ma, 2, 2);
  ma[0,0]:= 3; ma[0,1]:=-4;
  ma[1,0]:= 4; ma[1,1]:=-0.7;
  A:= TRealMatrixVariable.Create('A', ma); // A-matrix.
  SetLength(vb, 2);
  vb[0]:=-1; vb[1]:= 1;
  B:= TRealArrayVariable.Create('B', vb); // B-vector.
  prms:= TArray<TVariable>.Create(A, B);

  y0:= TArray<TFloat>.Create(1.0,-1.0); // Initial conditions.

  solver:= TFehlbergSolver; // Use the Fehlberg solver.

  ode:= TVectorODE.Create('t', 'y', 'A×y+B/(t^2+1)', 2, prms);

  y:= solver.Solve(ode, y0, 3.0, 1000, t);
 // Code for using 'y' values …
end;
```

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 ***Mathematix.Analysis*** unit. It has the following function:

```
class function Solve(f, df: TFunction1D; const x1, x2: TFloat; const
opt: TSolverOptions; var xr: TFloat): TSolutionResult; virtual;
```

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 **var** 'xr' parameter.
All solvers requires the method reference of type ***Function1D*** (defined in the ***Mathematix.Numerics*** unit). 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 ***Analytix.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:

```
var
  sf1D, dsf1D: TSymbolicFunction1D;
  solver: TRootFinderClass;
  opt: TSolverOptions;
  xr: TFloat;
  sr: TSolutionResult;
begin
  solver:= TNewton; // 1
```

```
  sf1D:= TSymbolicFunction1D.Create('x', 'sin(x^2)*e^-x'); // 2
  dsf1D:= sf1D.Derivative;
  opt:= TSolverOptions.Create(true); // 3
  sr:= solver.Solve(sf1D.F, dsf1D.F, 1.0, 3.0, opt, xr); // 4
  if sr.Converged then
  // Code for using the root value 'xr'...
end;
```

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

```
var
  sf1D: TSymbolicFunction1D;
  solver: TRootFinderClass;
  opt: TSolverOptions;
  points: TArray<TFunctionPoint>;
begin
  solver:= TNewton;
  sf1D:= TSymbolicFunction1D.Create('x', 'sin(x^2)*e^-x');
  opt:= TSolverOptions.Create(true);
  points:= TFunctionAnalyser.Analyse(sf1D, solver, opt, 1.0, 3.0, 10);
  // Code for using found special function points 'points'…
end;
```

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 ***Mathematix.NL.Solver*** unit. It has the following method:

```
function Solve(system: TNonlinearSystem; x0: TArray<TFloat>; const
options: TSolverOptions; var x: TArray<TFloat>): TSolutionResult;
virtual;
```

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 **var** ‘x’ parameter.

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

The class ***AnalyticalSystem*** from the ***Analytix.Numerics.Nonlinear*** unit 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.

```
var
  variables, equations: TArray<string>;
  system: TAnalyticalSystem;
  x0, x: TArray<TFloat>;
  opt: TSolverOptions;
  solver: TNonlinearSolver;
  r: TSolutionResult;
begin
  variables:= TArray<string>.Create('x', 'y', 'z');
  equations:= TArray<string>.Create(
    'x^2+y^2+z^2-1' {sphere equation},
    'x^2+y^2-z'     {paraboloid equation - along x-axis},
    '-x+2*y^2+z^2'  {paraboloid equation - along z-axis});

  system:= TAnalyticalSystem.Create(variables, equations);
  x0:= TArray<TFloat>.Create(1.0, 1.0, 1.0);
  opt:= TSolverOptions.Create(true);
  solver:= TNewtonRaphsonSolver.Create();
  r:= solver.Solve(system, x0, opt, x); // solving the system
  if r.Converged then
  // Code for using the solution 'x'...
end;
```

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.

Linear algebra extension for ANALYTICS

Linear algebra extension for ANALYTICS library introduces the operations on N-dimensional arrays and matrices of real numbers for using in the analytical expressions. All common

array/matrix operators realized – addition, multiplication, summation and so on. All elementary functions and base special functions (see Appendix A) are applicable for the array/matrix data and evaluated for each data element.

The next part contains detailed information about the operators and functions, realized in the Linear Algebra extension library and the.

Introducing arrays and matrices

There are two ways to use N-dimensional arrays and matrices in the analytical expressions. The first one is to add array/matrix variables to the translator instance. The following code is an example of adding an array and a matrix variable:

```
var
  av: TArray<TFloat>;
  mv: TArray<TArray<TFloat>>;
  A: TRealArrayVariable;
  M: TRealMatrixVariable;
begin
  av:=TArray<TFloat>.Create(0.2, -0.1, 0.4, -0.25);
  A:=TRealArrayVariable.Create('A', av);
  translator.Add(A);

  SetLength(mv, 3, 4);
  mv[0,0]:= 0.5; mv[0,1]:=-0.4; mv[0,2]:= 0.33; mv[0,3]:= 0.44;
  mv[1,0]:=-0.3; mv[1,1]:= 0.1; mv[1,2]:= 0.28; mv[1,3]:= 0.25;
  mv[2,0]:=-0.4; mv[2,1]:=-0.2; mv[2,2]:=-0.75; mv[2,3]:= 1.0;
  M:=TRealMatrixVariable.Create('M', mv);
  translator.Add(M);
  ...
```

After the code, the variables ‘A’ and ‘M’ can be used in analytical expressions. The ‘A’ variable represent the array of 4 components, the ‘M’ variable is the 3x4 matrix.

Another way is using special constructing functions directly in the expressions. The constructing functions are:

Array{N}(x) – returns the array with N elements, each element is x;
Matrix{M N}(x) – returns the MxN matrix, each element is x;
Diagonal{N}(x) – returns the square diagonal NxN matrix, each diagonal element is x;
Antidiagonal{N}(x) – returns the square antidiagonal NxN matrix, each element on the antidiagonal is x.

Array and matrix operators

The following binary operators are defined for Real/Array/Matrix operands: '**+**', '**-**', '*****', '**•**', '**×**', '***/***', '**^**'. These operators are implemented to satisfy common linear algebra rules for vector and matrix operations where applicable. The following implementation features have to be taken into account:

- Addition operations '**+**', '**-**' for Array and Matrix operands implemented by rows – the array elements added to (subtracted from) each matrix row.
- 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.

The unary operators for Array and Matrix arguments are:

- Number operator '**#**': number of elements in an array or a matrix.
- 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.

Here is the code for examples of using array/matrix operators. It is supposed that the array 'A' and the matrix 'M' variables added as in the previous section. The first example is to calculate the standard statistics deviation (https://en.wikipedia.org/wiki/Deviation_(statistics)).

```
var
  f: string;
  v: TValue;
  d: TFloat;
begin
  f:= '∑((A-1/2)^2)/#A';
  v:= ftranslator.Calculate(f);
```

```
  d:= TUtilities.FromValue<TFloat>(v);
  // d - the deviation value
end;
```

The deviation value calculated for the mean ½. Note that there are obligatory the parentheses for the power operator, because the unary prefix sum operator is of higher precedence than the binary power one.

The next example is the solution of an overdefined system of linear equations by the least squares method (https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)).

```
var
  f: string;
  v: TValue;
  x: TArray<TFloat>;
begin
  f:= 'M'×A×(M'×M)`';
  v:= ftranslator.Calculate(f);
  x:= TUtilities.FromValue<TArray<TFloat>>(v);
 // x - the system solution
end;
```

NOTE: use the '***TUtilities.FromValue<Generic Type>***' method to convert ***TValue*** to a generic type (like *TArray<TFloat>*). This method solves the problem of using generics types with run-time Delphi packages.

Array and matrix functions

All 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. Parametric functions, like '***log{b}{a}***', allow as array/matrix parameters as real parameters and arguments.

There are also specific functions for array/matrix arguments:

Min(X), ***Max(X)*** – minimal/maximal component of an array or a matrix.
Range{i1 i2}(A) – range of an array components from 'i1' to 'i2' inclusively.
Range{i1 i2 j1 j2}(M) – submatrix of 'M' containing 'i1'-'i2' rows an 'j1'-'j2' columns inclusively.
RowCount(M), ***ColumnCount(M)*** – number of rows/columns of a matrix.
Diagonal(M), ***Antidiagonal(M)*** – main diagonal/antidiagonal of a matrix.
Row{N}(M), ***Column{N} (M)*** – 'N'-th row/column (array) of a matrix.
Minor{i j}(M) – minor (matrix) of the [i,j]-th matrix' element.
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***, ***Minor***; and constructors **Array**, **Matrix**.
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.

Let us consider as an example of the logical array application the following problem solution: there are two currencies course data (arrays) for the same ten periods; we need to calculate, how many periods the course difference was more than 0.1.
The code for the problem solution is the following:

```
var
  a1, a2: TArray<TFloat>;
  f: string;
  x: TFloat;
begin
  a1:=TArray<TFloat>.Create(1.10, 1.15, 1.20, 1.17, 1.17, 1.18, 1.20,
1.17, 1.15, 1.16);
  a2:=TArray<TFloat>.Create(1.31, 1.32, 1.32, 1.33, 1.30, 1.28, 1.27,
1.25, 1.23, 1.22);
  translator.Add('A1', a1);
  translator.Add('A2', a2);

  f:= 'Count{true}(A2-A1>0.1)';
```

```
  x:= translator.Calculate(f).AsType<TFloat>;
  // x – is the number of periods
end;
```

For the example course values the answer is 6 periods. In the code above, logical array using is implicit: logical array returned by the comparison operation '>' and then its components used to count the number of 'true' values.
The array **if{B}(X Y)** function allows mixed **X** and **Y** arguments, that is one of them can be scalar (real) value. Then the result of the function is also the array of the same as B size and the scalar value used to fill the elements of the array according to the condition.
As an example, let us consider the following problem: calculate the sum of array's elements those are greater than the specified value. The code for solution is:

```
var
  av: TArray<TFloat>;
  f: string;
  s: TFloat;
begin
  av:=TArray<TFloat>.Create(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.Add('A', av);

  f:= '∑if{A>1/10}(A 0)';
  s:= translator.Calculate(f).AsType<TFloat>;
  // use s value...
end;
```

The output for the example is:

```
∑if{A>1/10}(A 0) = 2.07
```

The example formula contains 0 value as the second argument of '**if**' function with array condition, so, when the condition is true, the result array element picked from '**A**' array, else the element is 0. Then the sum of the elements evaluated with the '∑' operator and it is the solution of the stated problem.

Statistics extension for ANALYTICS

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:

```
var
  av: TArray<TFloat>;
  f: string;
  m, mn, md, v, d: TFloat;
begin
  av:=TArray<TFloat>.Create(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.Add('A', av);

  f:= 'Mean{1}(A)';
  m:= translator.Calculate(f).AsType<TFloat>;
  f:= 'Median(A)';
  mn:= translator.Calculate(f).AsType<TFloat>;
  f:= 'Mode(A)';
  md:= translator.Calculate(f).AsType<TFloat>;
  f:= 'Variance(A)';
  v:= translator.Calculate(f).AsType<TFloat>;
  f:= 'Deviation(A)';
  d:= translator.Calculate(f).AsType<TFloat>;

  // using the values
end;
```

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:

```
var
  f: string;
  v1, v2: TValue;
  pn, gp: TArray<TFloat>;
begin
  f:= 'Primes(0 100)';
  v1:= translator.Calculate(f);
  pn:= TUtilities.FromValue<TArray<TFloat>>(v1);

  f:= 'Geometric{2 1/2}(10)';
  v2:= translator.Calculate(f);
  gp:= TUtilities.FromValue<TArray<TFloat>>(v2);

  // using pn and gp values…
end;
```

The output for the code is the following:

```
Prime numbers = TArray<TFloat>[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 = TArray<TFloat>[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:

```
var
  f: string;
  v: TValue;
  h: TArray<TFloat>;
begin
  f:= 'Histogram{11}(GaussRnd{4 1}(0 5 1000000))';
  v:= translator.Calculate(f);
  h:= TUtilities.FromValue<TArray<TFloat>>(v);

  // using the h value…
end;
```

The output for the code is:

```
Histogram = TArray<TFloat>[11]=
(211 957 3794 12404 33147 71146 125464 179899 211445 202334 159199)
```

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).

Appendix A. Analytics operators and functions

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

Operator	Symbol	Type	Derivative[5]	Commutative
Logical And	&	Binary	False	Yes
Logical Or	\	Binary	False	Yes
Identically equal	≡	Binary	False	Yes
Approximately equal	≈	Binary	False	Yes
Not equal	≠	Binary	False	Yes
Greater	>	Binary	False	No
Less	<	Binary	False	No
Greater or equal	≥	Binary	False	No
Less or equal	≤	Binary	False	No
Add	+	Binary	True	Yes
Subtract	-	Binary	True	No
Multiply	*	Binary	True	Yes
Divide	/	Binary	True	No
Dot	•	Binary	False	No
Cross	×	Binary	False	No
Power	^	Binary	True	No
Left arrow	←	Binary	False	No
Right arrow	→	Binary	False	No
Up arrow	↑	Binary	False	No
Down arrow	↓	Binary	False	No
Left-right arrow	↔	Binary	False	No
Up-down arrow	↕	Binary	False	No
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	–

[5] 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[5]	Commutative
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	–
Absolute	||	Unary, Outfix	True	–
Norm	||||	Unary, Outfix	False	–

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

Function[6]	Name	Example	Derivative[7]
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))

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

7 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[6]	Name	Example	Derivative[7]
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)
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[8]	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[R] function	Γ	Γ(x)	Γ(x)*ψ(x)
Logarithm of Gamma[R]	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	$J_0(x)$	$-J_1(x)$

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

Function[6]	Name	Example	Derivative[7]
Bessel[R] function of order 1	J1	J1(x)	J0(x)-J1(x)/x
Bessel[R] function of the second kind, order 0	Y0	Y0(x)	-Y1(x)
Bessel[R] function of the second kind, order 1	Y1	Y1(x)	Y0(x)-Y1(x)/x
Modified Bessel[R] function of order 0	I0	I0(x)	I1(x)
Modified Bessel[R] function of order 1	I1	I1(x)	I0(x)-I1(x)/x
Modified Bessel[R] function, second kind, order 0	K0	K0(x)	-K1(x)
Modified Bessel[R] function, second kind, order 1	K1	K1(x)	-K0(x)-K1(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)