VBA Expressions Reference Manual

Wilfredo García[*]
April 2025
Version 1.0.2

Summary

This document serves as a reference and user guide for *VBA Expressions*. The library specifications, common and advanced usage are duly addressed[1]. All the required information are included to helping users of the *VBA Expressions* library to quickly familiarize themselves with the tool. This manual can be read sequentially, recommended for users with little or no VBA® experience, or by topic of interest. For users who are new to VBA® programming, know the basic concepts, and want to start using *VBA Expressions*, they can read sections from 2 to 3. In these they will find the information they need to use the library, evaluate basic and intermediate complexity expressions. This sections will cover operators, syntax rules, variables, internal library behavior, some basic configurations, methodology used by the library to evaluate given expressions and some use cases. All kind of users can read this manual starting on section 3, in page 12, over this code examples are shown to quickly library usage introduction. The section 7 covers use of variables in a broad fashion, dealing with advanced techniques to create, assign and reuse variables to change the way expressions are handled. Limitations are covered in section 12.

More experienced users can read the full documentation to understand how *VBA Expressions* works and discover advanced methodologies for doing more complex stuffs. The library structure, available methods and properties for class instantiated objects are addressed is section 4. In deep use cases are detailed in section 6. This section will include data management, linear algebra, statistics and data analysis, engineering and physics applications. Matrix syntax, operations with matrices and overloading such variables are covered in section 8. A complete reference on creating, declaring and managing user defined functions (UDFs) is presented in section 9. A whole list of currently available functions; segregated into basics, mathematical, statistical, financial, date-time and string functions groups; is given at section 10. Unit testing through Rubberduck[2] is detailed in section 11. A fully functional VBA procedure is shown in the appendix.

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 4.0 International" license.

[*] https://github.com/ws-garcia
[1] The library version used for this manual was v3.2.8, future releases may include changes in library structure leading to a slight or extensive deprecation of this technical guide
[2] https://rubberduckvba.com/

Contents

1 Introduction 5
- 1.1 Advantages . . . 5
- 1.2 Get in ready! . . . 5

2 Key Essentials 6
- 2.1 Operators and Symbols . . . 6
  - 2.1.1 Operators . . . 6
  - 2.1.2 Symbols . . . 7
- 2.2 Syntax And Rules . . . 7
  - 2.2.1 Grammar . . . 8
  - 2.2.2 Variables . . . 9
  - 2.2.3 Functions . . . 9
  - 2.2.4 Matrices and Arrays . . . 9
  - 2.2.5 Lists . . . 10
- 2.3 Core Internals . . . 10
  - 2.3.1 Parsing Methodology . . . 10
  - 2.3.2 Evaluation Tree . . . 10

3 Quick Start 12
- 3.1 The Basis . . . 12
- 3.2 Expressions with Variables . . . 14
- 3.3 Evaluating into Loops . . . 14
- 3.4 Using Binary Relations . . . 15
- 3.5 Working with Strings . . . 15

4 Library Brief Structure 16
- 4.1 VBAexpressions.cls . . . 16
  - 4.1.1 Properties . . . 16
  - 4.1.2 Methods . . . 21
  - 4.1.3 Enumerations . . . 23
- 4.2 VBAexpressionsScope.cls . . . 24
  - 4.2.1 Properties . . . 24
  - 4.2.2 Methods . . . 25

5 Notes for LibreOffice Users 26
- 5.1 Variables Treatment . . . 26
- 5.2 Recursion . . . 27
- 5.3 Library Loading . . . 27

6 Main Use Cases 27
- 6.1 Provide Evaluation Capability to other Applications . . . 28
- 6.2 Data Management . . . 28
- 6.3 Linear Algebra . . . 28
- 6.4 Statistics and Data Analysis . . . 30
- 6.5 Engineering and Physics . . . 31
- 6.6 Analytical Geometry . . . 32

7 Variables Definition and Assignment 33
- 7.1 Scope of Variables . . . 33
- 7.2 Accessing Variables . . . 34
- 7.3 Special Uses . . . 35

8 Working with Matrices/Arrays 35
- 8.1 Some Key Notes . . . 35
- 8.2 Matrices Overloading . . . 36

9 Managing User Defined Functions 37

10 Library Built-in Functions 38
- 10.1 Transcendental Functions . . . 39
  - 10.1.1 ABS . . . 39
  - 10.1.2 ACOS . . . 39
  - 10.1.3 ASIN . . . 39
  - 10.1.4 ATN . . . 40
  - 10.1.5 AVG . . . 40
  - 10.1.6 CEIL . . . 40
  - 10.1.7 COS . . . 40
  - 10.1.8 EXP . . . 40
  - 10.1.9 FLOOR . . . 41
  - 10.1.10 LGN . . . 41
  - 10.1.11 LN . . . 41
  - 10.1.12 LOG . . . 41
  - 10.1.13 MAX . . . 41
  - 10.1.14 MIN . . . 42
  - 10.1.15 PERCENT . . . 42
  - 10.1.16 POW . . . 42

10.1.17 ROUND 42
10.1.18 SGN 42
10.1.19 SIN 43
10.1.20 SQR, SQRT 43
10.1.21 SUM 43
10.1.22 TAN 43
10.2 Mathematical Functions 43
10.2.1 CHOLESKY 43
10.2.2 CHOLINVERSE 44
10.2.3 CHOLSOLVE 44
10.2.4 DET 44
10.2.5 DISTANCE 45
10.2.6 FZERO 45
10.2.7 GAMMA 45
10.2.8 GAMMALN 46
10.2.9 INVERSE 46
10.2.10 LINESINTERSECT 46
10.2.11 LSQRSOLVE 46
10.2.12 LUDECOMP 47
10.2.13 LUSOLVE 47
10.2.14 MMULT 47
10.2.15 MNEG 47
10.2.16 MROUND 48
10.2.17 MSUM 48
10.2.18 MTRANSPOSE 48
10.2.19 PARALLEL 48
10.2.20 PERPENDICULAR 49
10.2.21 QR 49
10.2.22 REM 49
10.2.23 SOLVE 49
10.3 Statistical Functions 50
10.3.1 ACHISQ 50
10.3.2 AERF 50
10.3.3 AFISHF 50
10.3.4 AGAUSS 51
10.3.5 ANORM 51
10.3.6 ASTUDT 51
10.3.7 BETA.DIST 51
10.3.8 BETAINV 52
10.3.9 CHISQ 52
10.3.10 ERF 52
10.3.11 FISHF 52
10.3.12 FIT 53
10.3.13 GAUSS 53
10.3.14 IBETA 53
10.3.15 MLR 54
10.3.16 NORM 54
10.3.17 STUDT 55
10.3.18 TINV 55
10.3.19 TINV_1T 55
10.3.20 TINV_2T 55
10.4 Financial Functions 56
10.4.1 DDB 56
10.4.2 FV 56
10.4.3 IPMT 56
10.4.4 IRR 56
10.4.5 MIRR 57
10.4.6 NPER 57
10.4.7 NPV 57
10.4.8 PMT 57
10.4.9 PPMT 57
10.4.10 PV 58
10.4.11 RATE 58
10.4.12 SLN 58
10.4.13 SYD 58
10.5 Date, Time and String Functions . 58
10.5.1 ASC 59
10.5.2 CHR 59
10.5.3 DATE 59
10.5.4 DATEADD 59
10.5.5 DATEDIFF 59
10.5.6 DATEPART 60
10.5.7 DATESERIAL 60
10.5.8 DATEVALUE 60

10.5.9 DAY 60
10.5.10 FORMAT 61
10.5.11 HOUR 61
10.5.12 INSTR 61
10.5.13 LCASE 61
10.5.14 LEFT 62
10.5.15 LEN 62
10.5.16 MID 62
10.5.17 MINUTE 62
10.5.18 MONTH 62
10.5.19 MONTHNAME 63
10.5.20 NOW 63
10.5.21 REPLACE 63
10.5.22 RIGHT 63
10.5.23 TIMESERIAL 64
10.5.24 TIMEVALUE 64
10.5.25 TRIM 64
10.5.26 UCASE 64
10.5.27 WEEKDAY 65
10.5.28 WEEKDAYNAME 65
10.5.29 YEAR 65
10.6 Programming Functions 65
10.6.1 ARRAY 65
10.6.2 CHOOSE 66
10.6.3 GET 66
10.6.4 IFF 66
10.6.5 SWITCH 66

11 Testing **67**
11.1 Rubberduck 67
11.2 Running Tests 67

12 Limitations **67**

13 Conclusions **68**

14 Credits **68**

15 License **68**

16 Review History **69**

17 Appendix **70**

1 Introduction

VBA Expressions is a powerful string expression evaluator library for VBA®, which puts more than 100 mathematical, statistical, financial, date-time, logic and text manipulation functions at the user's fingertips. Library mediates almost all VBA®and custom user defined functions (UDF) exposed through the tool, making it a support for students and teachers of science, accounting, statistics and engineering. The later is due to the capability on matrix operations (including factorization/decomposition), linear and over-determined equations systems solving, non-linear equations in one variable support, curve fitting, Multivariate Linear Regressions (MLR), and much more.

1.1 Advantages

- *Easy to use and integrate.*
- *Basic math, logical and binary relations operators.*
- *More than 100 built-in functions.*
- *Very flexible and powerful.*
- *Implied multiplication support for variables, constants and functions*
- *Evaluation of arrays/matrix expressions.*
- *Floating point notation input support.*
- *Free of external VBA or COM dependencies:* does not use dll.

1.2 Get in ready!

In order to use *VBA Expressions* in your Microsoft©Office®applications, or other hosts that support automation using the VBA®language, you need to import four class modules from the VBA®IDE into the target project where you intend to code. First, download the contents of the tool's repository from GitHub[3] and extract the contents from the `VBA-Expressions.zip` file into a folder of your choice.

Install: To import the modules required by the library, one must access the code editor, VBA®IDE, from the "Developer" tab. If the editor has never been accessed, it is necessary to enable the function using the Office configuration[4]. Once opened, the modules can be imported one by one from the `./src` folder in the directory where previously compressed files were extracted using the `File->Import File...` option from the IDE. An alternative is to import using drag and drop functionality from the folder into the IDE.

Library members: The presence of all these files must be verified in the user's IDE:

1. `VBAexpressions.cls`: the library's core class module. From this class, objects with all the functionality provided by the library are built.
2. `VBAexpressionsScope.cls`: template for sharing variables across `VBAexpressions.cls` instances. Its main utility is to allow multiple instances to share an environment of variables, so it is only

[3] https://github.com/ws-garcia/VBA-Expressions/releases/latest
[4] https://support.microsoft.com/en-us/office/show-the-developer-tab-e1192344-5e56-4d45-931b-e5fd9bea2d45

necessary to define it once and reuse it through different objects as many times as required.

3. `UDFunctions.cls`: an example class module used as a container for UDF functions. The user can define and use another class module to host its predefined functions, provided that it is properly declared and its name exposed using the procedures and mechanisms provided by the library.

! →

4. `VBAcallBack.cls`: acts as the link between UDF functions and `VBAexpressions.cls` instances. This module exposes the names of all class modules containing user-defined functions.

LibreOffice install: Starting in version 3.2 the library has an extension for LibreOffice applications. To install it, users must download the file `VBAExpression.oxt` from the GitHub repository and load the library from the dialog box opened by the options tools→extensions. From there it is possible to locate the downloaded file and proceed with the extension installation.

> **!** **Attention**
>
> The extension only works on LibreOffice versions 7.5 or later.

> **@** LibreOffice users can keep the extension up to date using the Extensions manager. If an error is received when updating, the update can be downloaded by clicking on the author's name from its description in the extension manager.

Once the installation process is finished, the user can proceed to write lines of code that implement the tool, however, it is necessary to know its basic operation. For this user are invited to explore sections 2 and 2.2.2.

2 Key Essentials

An expression is a combination of symbols/characters, evaluating it includes parsing and subdividing them into the smaller parts of which they are composed. In an expression we can find one or more combinations of operands, operators, functions and arguments.

2.1 Operators and Symbols

2.1.1 Operators

Operator : character or characters relating one or two operands, this nature subdivides them into unary and binary respectively. Typically, they return the result of operating on their operands with a specific function.

The classical operators are those used in elementary arithmetic and algebra, encompassed here as arithmetical and logical operators. There is another set frequently used in mathematics and computing called relational operators. The *VBA Expressions* library supported operators are listed below

Arithmetical : `+ - * / \ ^ !`

Logical : `& (AND)`, `| (OR)`, `|| (XOR)`

Relational : `<`, `<=`, `<>`, `>=`, `=`, `>`, `$ (LIKE)`

Operators Precedence When evaluating expressions, the functions invoked by the various operators must be called in a certain order to ensure the correctness of calculations. In this environment, operators are said to have a precedence that determines their evaluation order.

1. `()` *Grouping*: evaluates functions arguments as well.
2. `! - +` *Unary operators*: exponentiation is the only operation that violates this. Ex.: `-2 ^ 2 = -4 | (-2) ^ 2 = 4`.
3. `^` *Exponentiation*: Although Excel and Matlab evaluate nested exponentiations from left to right, Google, Mathematica©and other several modern programming languages, such as Perl, Python and Ruby, evaluate this operation from right to left. *VBA Expressions* also evaluates in Python way: `a^b^c = a^(b^c)`.
4. `* / \ %` *Multiplication, division, modulo*: from left to right.
5. `+ -` *Addition and subtraction*: from left to right.
6. `<` $^{(least-or-equal)}$ `<=` $^{(distint)}$ `<>` $^{(greatter-or-equal)}$ `>=` `=` `>` ^{like} `$` *Binary relations*
7. `~` *Logical negation.*
8. `&` *Logical AND.*
9. `||` *Logical XOR.*
10. `|` *Logical OR.*

2.1.2 Symbols

Symbol : special characters that serve a certain purpose in defining the syntax of expressions. They are used as control marks by expression evaluators.

By definition, operators are also symbols with the particularity that their nature is dynamic. Symbols groups, delimits and pass information between operators in an expression so that functions are performed on their arguments.

The *VBA Expressions* library defines the following symbols

→ Section 2.2

- *Separator*: this character acts as identifier and separates expressions elements. The default separator symbol is the semicolon character (`;`).

→ Section 8

- `{}` *Curly brackets*: this symbols identifies arrays/matrix in the expressions.
- `[]` *Square brackets*: identifier to tell the parser if a variable overload is required.

2.2 Syntax And Rules

Before evaluating expressions, all parsers impose a series of rules that users must follow in order to avoid potential errors. The review of this syntax is carried out with the implementation of grammars that clearly define the components accepted by the evaluator.

2.2.1 Grammar

The grammar defining the expressions supported by the utility is shown below

Expression : `([{"("}] SubExpr [{Operator [{"("}] SubExpr [{")"}]}] [{")"}]`
`| {["("] [("{" | "[")] List [{";" List}] [("}" | "]")] [")"]})`

SubExpr : `Token [{Operator Token}]`

Token : `[{Unary}] Argument [(Operator | Function)`
`["("] [{Unary}] [Argument] [")"]]`

Argument : `(List | Variable | Operand | Literal)`

List : `["{"] ["{"] SubExpr [{";"`
`SubExpr}] ["}"] ["}"]`

Unary : `"-" | "+" | ~`

Literal : `(Operand | "'"Alphabet"'")`

Operand : `({Digit} [Decimal] [{Digit}] ["E"("-" | "+"){Digit}]`
`| (True | False))`

Variable : `Alphabet [{Decimal}] [{(Digit | Alphabet)}]`

Alphabet : `"A-Z" | "a-z"`

Decimal : `"." | ","`

Digit : `"0-9"`

Operator : `"+" |"-"|"*"|"/"|"\"|"^"|"%"|"!"|"<"|"<="|"<>"|">"`
`|">="|"="|"$"|"&"| "|" | "||"`

Function : `"abs" | "sin" | "cos" | "min" |...|[UDF]`

Bird Eyes to Grammar The above grammar lays the foundation for expressions supported by *VBA Expressions*. From it, one can deduce that an *expression* is a set of sub-expressions related by an operator that may contain parentheses to group sub-expressions together or, alternatively, a *list* defined as a set of sub-expressions separated by a common character. From the list grammar derives the syntax for arrays/matrices.

Arrays/matrices syntax:

```
{List} | {List; List;...} | [List]
```

In this context, a *sub-expression* is defined as a combination of one or more operators and tokens. A set of one or two arguments preceded or not by unary operators affected or not by a function defines a *token*, intermediate level unit of classification. *Argument*, on their own, can be a single list, a variable, an operand and quoted string.

By their grammar, the *operand* may be a combination of digits, which may include the decimal point and scientific notation, whether or not affected by unary operators. Alternatively, it may literally represent boolean values `True` or `False`. *Variables* are defined as a sequence of alphanumeric symbols which may include the decimal symbol.

Valid expressions by grammar:

```
5*avg(2;abs(-3-7*tan(5));9)-12*pi-e+(7/sin(30)-4!)*
min(cos(30);cos(150))
```

```
GCD(1280;240;100;30*cos(0);10*
DET({{sin(atn(1)*2);0;0};{0;2;0};{0;0;3}}))
```

2.2.2 Variables

→ See section 7

Variables are commonly used as value carriers between function calls. Similarly, instead being containers of values, they may contain other sub-expressions.

According to the established grammar, variable names must meet the following requirements:

1. Start with a letter
2. End in a letter or digit symbol. `x.1`, `number1`, `value.a` are valid variable names.
3. The symbol `E` cannot be used as variable name due it is reserved for floating point representation.

! → Is important to note that a variable named `A` is distinct from another named `a`, since names are case-sensitive.

Before evaluating any expressions, the values for variables must be assigned. So, if it has been decided to define an expression with multiple variables in order to simplify its complexity, special care must be taken in this respect. For example, one of the long functions from 2.2.1 can be expressed as `5a+b+c*d` , where

```
a=avg(2;abs(-3-7*tan(5));9)
b=-12*pi-e
c=(7/sin(30)-4!)
d=min(cos(30);cos(150))
```

Special Variables The variables PI and e are special with a self-defined and constant value, so they can be used in expressions without the need to define their value.

Using *PI* and *e*: `2.5pi+3.5e`

2.2.3 Functions

→ Sections 9, 10

Predefined, built-in together with the tool, and user-defined are the two types of functions supported by *VBA Expressions*. As opposed to variables, the names given to functions are case-insensitive, so the expressions `Sin(x)+log(4)` and `SIN(X)+LOG(4)` are equivalents.

2.2.4 Matrices and Arrays

As stated earlier in 2.2.1, an expression pointing to an array/matrix must adhere to the syntax `{List} | {List; List;...} | [List]`. Roughly speaking, we can say that an array is defined by a specific type of list, these being variables, in the case of arrays/matrices that follow grammar `{List}` and `{List; List;...}`, or sub-expressions, for those lists that follow grammar part `[List]`. This last case is the basis for the variable overloading exposed in section 8.

Example of matrices/arrays: `{{2;1;4};{1;5;-3};{5;-2;3}}`

2.2.5 Lists

Under the grammar proposed in subsection 2.2.1, lists serve multiple purposes. The first of these is a variable that contains a structured set of values that defines the rows of a matrix/array. Lists also acts as sub-expressions that work together with functions, acting as containers for arguments.

Lists examples: `{sin(30);1;pi} | 0.5;1;3 |`

2.3 Core Internals

The traditional way to evaluate mathematical expressions involves the conversion from infix (operands separated by an operator) to post-fix. In this way, the evaluation is performed using stacks, or parsed tree, which ensure follows operators precedence defined in the mathematical expressions.

VBA Expressions does not translate to post-fix, evaluating directly over infix inputs. It is a consensus that the implementation difficulty for infix evaluation techniques is overwhelming, although that, this toll implements an adequate and relatively efficient human-like evaluation methodology. The utility meets following requirements:

- Receives a text string as input and returns a string result obtained by executing the operations contained in the expression.
- Variables found in each parsed expression, as well the values relative to them, are exposed to the user.
- Hard-coded expressions are not required, variable values are assigned as text strings.

2.3.1 Parsing Methodology

The technique leverage an parsing process, prior to evaluation, consisting in creation of an evaluation tree through the following steps:

1. The expression is split into related sub-expressions using operators.
2. Each sub-expression is split into tokens, each of which is defined by 1 or 2 arguments related or not by an operator.
3. A token is created and stored for each sub-expression.
4. Each token is segregated into arguments, the lowest level of logic used, represented by a list, a variable or an operand.
5. Once all tokens are parsed, the evaluation tree is complete.

2.3.2 Evaluation Tree

As an example, figure 1 shows the sub-expressions into which the expression `(1+(2-5)*3+8/(5+3)^2)/sqrt(4^2+3^2)` has been segregated. In the exemplified case, the expression is divided into an evaluation tree of five sub-expressions, entry point for subsequent parsing routines.

Evaluation tree : hierarchical tree structure whose elements are defined by the token trees derived from the parsed expression.

Token tree : hierarchical tree structure whose elements represent the decomposition into arguments of each sub-expression. This structure can be seen as branch of the evaluation trees.

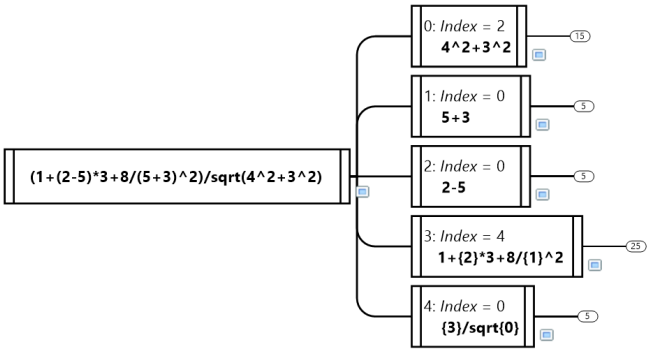


Figure 1: Segregation of expression into sub-expressions

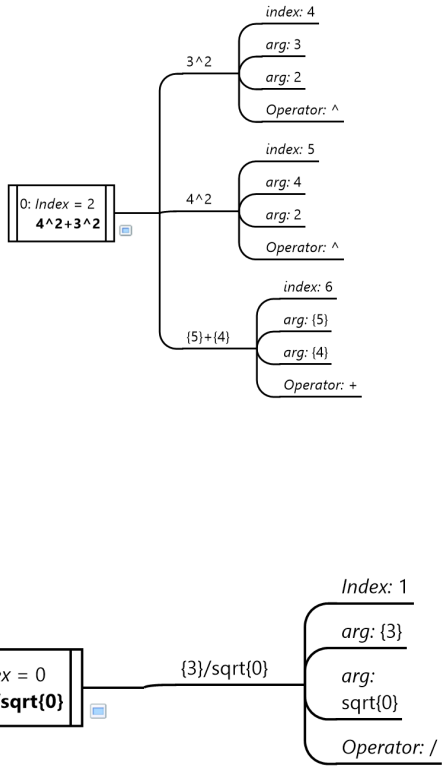


Figure 2: Tokens evaluation trees

```
Target expression:  (1+(2-5)*3+8/(5+3)^2)/sqrt(4^2+3^2)
Eval parentheses:   A<-- 4^2+3^2 =25
Eval parentheses:   B<-- 5+3 =8
Eval parentheses:   C<-- 2-5 =-3
Replacing into:   (1+C*3+8/B^2)/sqrt(A)
Precedence eval:   D<-- B^2 =64
Precedence eval:   E<-- C*3 =-9 |
Replacing into:   (1+E+8/D)/sqrt(A)
Precedence eval:   F<-- 8/D =0.125
Replacing into:   (1+E+F)/sqrt(A)
Precedence eval:   G<-- 1+E =-8
Replacing into:   (G+F)/sqrt(A)
Precedence eval:   H<-- G+F =-7.875
Replacing into:   H/sqrt(A)
Eval result:   I<-- -7.875/sqrt(25)= -1.575
```

Figure 3: Human like evaluation example

Figure 2 shows an example of created tokens for some sub-expressions. Each tree stores information about the number of tokens contained in the tree. Arguments that reuse previous computations are represented with the syntax `{#}`, where the `#` symbol represents relative position of the token within a tokens tree or the main evaluation tree.

Evaluating expressions involves tree traversal and arguments operation. An example of a calculation describing the evaluation methodology used by *VBA Expressions* is presented in figure 3. Because of the systematic way in which expressions are evaluated with the like-human method, one would think that evaluation and token trees are not necessary. Indeed, the method was conceived as a linear process applied sequentially on a text string. However, the need to evaluate an expression multiple times makes this idea an inefficient concept. Trees come, then, to fill the gap and provide the ability to define once and execute multiple times.

Another advantage of using trees is that they allow tokens to be classified into constants and variables. In this way, tokens are traversed only if they are not constants, saving computation time in subsequent evaluations.

3 Quick Start

In this section, examples are illustrated with lines of code oriented to configure the objects created by instantiate the `VBAexpressions.cls` class module. From this point on, the reader must support the reading with coding and execution of the instructions described in the rest of this manual.

3.1 The Basis

A code structure as show in the listing 1 will be gradually filled as the reader progresses through sections of the manual, any changes to this code structure will be explicit indicated. In short, the code is explained as:

1. The line `Private Sub VBAexpressionsDemo()` tells compiler there is a procedure (`Sub`) named `VBAexpressionsDemo` that only runs in the desired module or object from host application.

Early binding:

2. Line `Dim expr As VBAexpressions` instructs compiler to reserve memory for allocation of a variable named `expr` of type `VBAexpressions`.

3. `Set expr = New VBAexpressions` will allocates memory and create an object of type `VBAexpressions` by instantiate the `VBAexpressions.cls` class module. This step is always required when working with *VBA Expressions*.

! →

4. The `With...End With` block will expose all methods and properties from the `expr` variable by entering a dot `.` inside it. All code from this manual will be placed here unless strictly stated otherwise. The text in line 6, for the listing 1 need to be replaced with given code, so this line will mark the start of code inserting and the entering will be placed all below the line 5 and above the line 7.

5. The method `Eval` will be evaluate defined expression, will be discussed in more details in section 4.

6. The command `Debug.Print` from line 8 will print to console the specified elements from this line. In the case shown, the `expression, result, CurrentVarValues` properties from the `expr` variable, along with specified strings, will be output to console. These printed results will be persistently cited in subsequent sections of this manual.

! →

7. The line `Set expr = Nothing` will delete the `expr` object by erasing allocated memory. This lines is of utmost importance for keeping the system fine tuning.

8. `End Sub` indicates procedure ends.

```
1  Private Sub VBAexpressionsDemo()
2      Dim expr As VBAexpressions
3
4      Set expr = New VBAexpressions
5      With expr
6          '%YOUR CODE GOES HERE, PLEASE DELETE THIS LINE%
7          .Eval
8          Debug.Print .expression & " = "; _
9                      .result; " for "; .CurrentVarValues
10     End With
11     Set expr = Nothing
12 End Sub
```

Listing 1: Demo testing procedure structure

The best way to become familiar with programming languages and their syntax is through the classic "Hello World" program, which in our case will greet us by computing the trigonometric function $cos(\pi)+sen(\pi/2)$. To achieve this, replace the line 6 from the demo program in listing 1 whit `.Create "cos(pi)+sin(pi/2)"`, then run the procedure by pressing the `F5` key. This will output

```
cos(pi)+sin(pi/2) = 0 for pi = 3.14159265358979
```

The library is evaluating the expression in radians, for evaluation in degrees replace line 6 with these two lines shown below. The result will remain the same after evaluation, except for the printed message which now indicates that there are no variables or constants used at evaluation stage.

```
.Degrees = True
.Create "cos(180)+sin(90)"
```

In worked examples, the `Create` method is parsing the given string expression and constructing the evaluation tree, more in deep in section 4.

3.2 Expressions with Variables

On most occasions where an expression evaluator is required, the use of variables is necessary to avoid repetitive creation and parsing processes. Replace line 6 from listing 1 with

```
.Create "Sqrt(x-Ln(2))"
.Eval "x=0.8"
```

Now an expression with a variable named `x` is defined and its value is passed with the `.Eval "x=0.8"` instruction. Affter hit `F5` the output is

```
Sqrt(x-Ln(2)) = 0.326883495209004 for x = 0.8
```

You can create multi-variable expressions. An example is the formulation to compute the displacement of a projectile, at time t, fired at initial speed v_0. The formulation is as follows

$$s = v_0 \cdot \ t + \frac{1}{2} a \cdot \ t^2$$

Using *VBA Expressions* you can solve this problem with the follow lines of code

```
.Create "v0*t + a*t^2/2"
.Eval "v0=5; a=-9.81; t=3"
```

! → Note that all variables assignment is separated with semicolons.

The given code will compute projectile displacement when elapsed 3 seconds.

```
v0*t + a*t^2/2 = -29.145 for v0 = 5; t = 3; a = -9.81
```

3.3 Evaluating into Loops

Let us now study the case where a snapshot of the projectile's displacement needs to be calculated from time 0 (when it was fired) until 3 seconds have elapsed, with a difference between readings of 0.5 seconds. This can be modeled by restructuring the `With...End With` block from listing 1. The complete code for this example is shown in listing 2.

```
Private Sub ProjectileDisplacement()
    Dim expr As VBAexpressions
    Dim i As Double

    Set expr = New VBAexpressions
    With expr
        .Create "v0*t + a*t^2/2"
        For i = 0 To 3 Step 0.5
            .Eval "v0=5; a=-9.81; t=" & CStr(i)
```

```
10              Debug.Print .expression & " = "; _
11                          Round(CDbl(.result), 3); _
12                          " for "; .CurrentVarValues
13          Next i
14      End With
15      Set expr = Nothing
16  End Sub
```

Listing 2: Projectile procedure listing

As can be seen, new code elements have been added. The For...Next block increments the variable i by 0.5 units at each iteration, while the `Round(CDbl(.result), 3)` command rounds our calculations to 3 decimal places. Here is the output by executing the code from listing 2

```
v0*t + a*t^2/2 =  0  for v0 = 5; t = 0; a = -9.81
v0*t + a*t^2/2 =  1.274  for v0 = 5; t = 0.5; a = -9.81
v0*t + a*t^2/2 =  0.095  for v0 = 5; t = 1; a = -9.81
v0*t + a*t^2/2 = -3.536  for v0 = 5; t = 1.5; a = -9.81
v0*t + a*t^2/2 = -9.62  for v0 = 5; t = 2; a = -9.81
v0*t + a*t^2/2 = -18.156  for v0 = 5; t = 2.5; a = -9.81
v0*t + a*t^2/2 = -29.145  for v0 = 5; t = 3; a = -9.81
```

3.4 Using Binary Relations

A clever way to use expressions is through binary relationships between elements. For example, we can define piecewise functions to be evaluated in a single instruction, by inserting lines of code, starting from line 6 of listing 1, like the ones shown below

```
1 .Create "(x<=0)* x^2 + (x>0 & x<=1)* Ln(x+1) + (x>1)* Sqrt(x-
      Ln(2))"
2 .Eval "x=6"
```

```
(x<=0)* x^2 + (x>0 & x<=1)* Ln(x+1) + (x>1)*
Sqrt(x-Ln(2)) = 2.30366074313039 for x = 6
```

3.5 Working with Strings

So far we have exemplified the computation of numerical values using *VBA Expressions*. We can also evaluate expressions having literal string variables, as we will see in this section.

The grammar stated in section 2.2.1 defines a `Literal` component. From this, a literal string is a sequence of alphabet symbols enclosed by single quotes. Recapitulating the structure of listing 1, by replacing line 6 with `.Create "MID('Demo String';6)"` will output

```
MID('Demo String';6) = 'String' for
```

→ More useful functions on section 10

User must explore combine this with other capabilities exposed into several creative ways. For example, checking equality of strings with binary relations, section 3.4.

```
1 .Create "IFF(x='VBA';'VBA Langage';'Not Listed')"
2 .Eval "x='C'"
```

```
IFF(x='VBA';'VBA Langage';'Not Listed') =
'Not Listed' for x = 'C'
```

4 Library Brief Structure

In this section the `VBAexpressions.cls` and `VBAexpressionsScope.cls` modules, being the most relevant components of the library, will be discussed. In all members exposed in this section, the `expression` variable is an object created by instantiate the corresponding class.

4.1 VBAexpressions.cls

4.1.1 Properties

Constants : Gets the constants collection. By default this is pre-populated with PI and e.
Property Type: `Object:Collection`.

Accesor	*Syntax*
`Get`	`expression.Constants`

CurrentVariables : Gets a string with all parsed and defined variables.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.CurrentVariables`

CurrentVarValues : Gets a string with the variables values to be used in evaluations.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.CurrentVarValues`

DecimalSymbol : Gets or sets the decimal symbol.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.DecimalSymbol`
`Let`	`expression.DecimalSymbol = value`

! → *Remarks:* The allowed options are the dot or comma, if another character is specified dot will be used which is the default setting. Specified decimal symbol must match the "Region Number Format"[5]. This is because, internally, the evaluator use VBA®functions to convert strings into they numeric representation. Thus, if the configurations doesn't match, users will get unexpected results after evaluate expressions.

[5] https://support.microsoft.com/en-au/office/change-the-windows-regional-settings-to-modify-the-appearance-of-some-data-types-in-access-databases-edf41006-f6e2-4360-bc1b-30e9e8a54989

Degrees : Gets or sets the behavior for trigonometric functions computations. If True, the program will calculate angles in degrees.
Property Type: `Boolean`.

Accesor	*Syntax*
`Get`	`expression.Degrees`
`Let`	`expression.Degrees = value`

ErrorDesc : Returns the last error message generated.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.ErrorDesc`

ErrorType : Returns the type of last generated error message.
Property Type: `Enum:ExpressionErrors`. See page 23.

Accesor	*Syntax*
`Get`	`expression.ErrorType`

EvalScope : Sets/gets the scope for current expression.
Property Type: `Object:VBAexpressionsScope`.

Accesor	*Syntax*
`Get`	`expression.EvalScope`
`Let`	`Set expression.EvalScope = value`

Remarks: users can define an scope and share it across multiple objects. This is illustrated in the example, for more information about the `VBAexpressionsScope` class, refer to section 4.2.

```
1  Private Sub ScopeExample()
2      Dim expr As VBAexpressions
3      Dim vScope As VBAexpressionsScope
4
5      Set expr = New VBAexpressions
6      Set vScope = New VBAexpressionsScope
7      With expr
8          vScope.VarValue("x") = "-7"
9          Debug.Print "Current variables: "; vScope.
       CurrentVarValues
10
11         .Create "POW(x;3)"
12         Set .EvalScope = .EvalScope.CopyScope(vScope)
13         Debug.Print "x^3 = "; .Eval
14     End With
15 End Sub
```

```
Current variables: x = -7
x^3 = -343
```

expression : Returns the expression to be evaluated.
Property Type: `String`.

Accesor	*Syntax*
Get	`expression.expression`

EnforceBoolean : Defines the behavior of evaluator when facing an error.
Property Type: `Boolean`.

Accesor	*Syntax*
Get	`expression.EnforceBoolean`
Let	`expression.EnforceBoolean = value`

Remarks: When this property is set to `True` a `False` value is returned instead of strings flags on evaluation errors. This can be useful when evaluating piece-wise functions.

```
Private Sub EvalBehavior()
    Dim expr As VBAexpressions

    Set expr = New VBAexpressions
    With expr
        .Create "SQRT(x)"
        Debug.Print "Eval result: "; .Eval("x=-7")
        .EnforceBoolean = True
        Debug.Print "Eval result, enforced: "; .Eval
    End With
End Sub
```

```
Eval result: #VALUE!
Eval result, enforced: False
```

FormatResult : Indicates if the result output will be converted to standard VBA®string.
Property Type: `Boolean`.

Accesor	*Syntax*
Get	`expression.FormatResult`
Let	`expression.FormatResult = value`

Remarks: When this property is set to `False`, default value, the result of string operations will be enclosed in single quotes. Change this configuration if comparisons with a regular VBA®formatted string is required after computations.

```
Private Sub FormatingOutput()
    Dim expr As VBAexpressions

    Set expr = New VBAexpressions
    With expr
        .Create "MID(x;8)"
        Debug.Print "Eval result: "; .Eval("x='Asia &
    Oceania'")
        .FormatResult = True
        Debug.Print "Formatted result: "; .Eval
    End With
End Sub
```

```
Eval result: 'Oceania'
Formatted result: Oceania
```

GallopingMode : Turns on or off the evaluation in galloping mode.
Property Type: `Boolean`.

Accesor	*Syntax*
`Get`	`expression.GallopingMode`
`Let`	`expression.GallopingMode = value`

Remarks: When set to `True`, the evaluator will discriminate,gallop over constant tokens, from variable tokens. This can reduce computation time when evaluating expression multiple times.

! → This is an experimental utility, a pre-populated can be a better approach to improve this on.

ImplicitVarValue : Sets the value for a variable in terms of previously defined variables.
Property Type: `None`.

Accesor	*Syntax*
`Let`	`expression.ImplicitVarValue(aVarName) = value`

Argument	*Type*
`aVarName`	`String`

Remarks: tif `aVarName` is numeric, the argument is treated as an index. The `value` can be an expression to be evaluated prior value assignment. Is required that all variables not present in the main expression but contained in the `value` string to be assigned.

```
1  Private Sub ImplicitVariables()
2      Dim expr As VBAexpressions
3
4      Set expr = New VBAexpressions
5      With expr
6          .Create "ABS(c)"
7          .VarValue("a") = -3
8          .VarValue("b") = 9
9          .ImplicitVarValue("c") = "a*b"
10         Debug.Print .expression; " = "; .Eval; " for "; .
       CurrentVarValues
11     End With
12 End Sub
```

```
ABS(c) = 27 for c = -27; a = -3; b = 9
```

ReadyToEval : Gets the parsed status of the actual expression.
Property Type: `Boolean`.

Accesor	*Syntax*
`Get`	`expression.ReadyToEval`

Remarks: a `False` value indicates syntax error at parsing stage.

result : Gets the result for latest evaluated expression.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.result`

Remarks: a returned `vbNullString` value indicates error at evaluation stage, the sane applies if the returned value keeps the same after consecutive evaluations where values of the variables involved in the calculations have been changed beforehand.

SeparatorChar : Gets or sets the character used as separator for functions arguments and array elements.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.SeparatorChar`
`Let`	`expression.SeparatorChar = value`

Remarks: by default the char used is the semicolon `";"`.
! → This parameter cannot be set to a reserved operator, nor can it be an alphanumeric character or any other special symbol used internally by the evaluator to define functions or variables. It is not recommended to change this parameter.

VarValue : Gets or sets the current value from/to the given variable.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.VarValue(aVarName)`
`Let`	`expression.VarValue(aVarName) = value`

Argument	*Type*
`aVarName`	`String`

Remarks: if `aVarName` is numeric, the argument is treated as an index. The value for a variable not present in the main expression can also be defined. For code details, refer to the `ImplicitVarValue` code example in page 19.

VarValue2 : Gets or sets the current value for the given variable.
Property Type: `String`.

Accesor	*Syntax*
`Get`	`expression.VarValue2(aVarKey, treatAsArray)`
`Let`	`expression.VarValue2(aVarKey, treatAsArray) = value`

Argument	*Type*
`aVarKey`	`Variant`
`treatAsArray`	`Boolean`

Remarks: if `aVarKey` is numeric, the argument is treated as an index. This property use `Variant` data type to handle arrays. The string array representation is retrieved as usual if the parameter

! → `treatAsArray` is set to `False`, an array is returned otherwise. This property is intended to achieve variable overloading, its use is recommended only for most experienced users.

```
Private Sub VarValue2()
    Dim expr As VBAexpressions
    Dim arr As Variant

    Set expr = New VBAexpressions
    With expr
        .VarValue2("a", True) = "{{2;5};{-4;3};{8;1}}"
        arr = .VarValue2("a", True)
        Debug.Print "Rows in returned array: "; _
        UBound(arr) - LBound(arr) + 1
        Debug.Print "Columns in returned array: "; _
        UBound(arr, 2) - LBound(arr, 2) + 1
        Debug.Print .VarValue2("a", False)
    End With
End Sub
```

```
Rows in returned array:  3
Columns in returned array:  2
{{2;5};{-4;3};{8;1}}
```

4.1.2 Methods

AddConstant : Appends a constant to the current constants list.
Method Type: `Sub`
Returns Type: `None`.

Argument	*Type*
`aValue`	`String`
`aKey`	`String`

Remarks: this method is used to assign the value to variables that remain unchanged between successive evaluations.

ArrayFromString : Turns a like Java array string (`{{*};{*}}`) into a 1D or 2D VBA®array with n rows and m columns.
Method Type: `Function`
Returns Type: `Array:String`.

Argument	*Type*
`strArray`	`String`

Remarks: the method will only check for balanced curly-brackets, users must be in charge of providing syntax error free inputs.

ArrayFromString2 : Turns a like Java array string (`{{*};{*}}`) into VBA®jagged array with n rows.
Method Type: `Function`
Returns Type: `Array:Variant`.

Argument	*Type*
`strArray`	`String`

Remarks: this method behave like the `ArrayFromString` one, being only different in they outputs.

ArrayToString : Turns a `m x n` array into a like Java array string `({{*};{*}})`.
Method Type: `Function`
Returns Type: `String`.

Argument	*Type*
`InputArray`	`Variant`

Remarks: this method acts as the reverse of `ArrayFromString` and `ArrayFromString2` methods. The `InputArray` argument can be a 1D, 2D or jagged array.

Create : Parses an expression and stores its evaluation tree.
Method Type: `Function`
Returns Type: `Object:VBAexpressions`.

Argument	*Type*
`aExpression`	`String`
`[resetScope]`	`Boolean`

Remarks: the `resetScope` option determines if the expression will be evaluated with a new evaluation/variable scope or the existing one. See section 4.2 for more details about evaluation scope. The `aExpression` parameter must follows the guidelines exposed in section 2.

DeclareUDF : Declares new user defined function.
Method Type: `Sub`
Returns Type: `None`.

Argument	*Type*
`UDFname`	`Variant`
`[UDFlib]`	`String`

Remarks: if required, an array of strings with the list of names can be passed in the `UDFname` argument to declare most than one UDF. The default value for the `UDFlib` optional parameter is `"UserDefFunctions"`, indicating the name of the class module containing the functions to be declared.
Users can register custom modules to expose and use their functions through the `VBAcallBack.cls` module. All UDFs must have a single Variant argument that will receive an one-dimensional array of strings (one element for each function argument). Here is the code that shows how the library is declaring its two default UDFs: `GDC` and `Concat`.

```
Sub AddingNewFunctions()
    Dim Evaluator As VBAexpressions
    Dim UDFnames() As Variant

    Set Evaluator = New VBAexpressions
    UDFnames() = Array("GCD", "Concat")
    With Evaluator
        .DeclareUDF UDFnames, "UserDefFunctions"
```

```
9       End With
10  End Sub
```

Additionally, after declaring UDFs, user must append a line of code to `VBAcallBack.cls` class module in order to expose new declared UDFs.

```
1  Public UserDefFunctions As New UDFunctions
```

Eval : Evaluates the current expression by evaluation tree processing.
Method Type: `Function`
Returns Type: `String`.

Argument	*Type*
`ValuesToEvalWith`	`Variant`

Remarks: the `ValuesToEvalWith` argument is used to assign variables values not defined through the `VarValue` property, see 20.

IsConstant : Determines if a variable is a constant.
Method Type: `Function`
Returns Type: `Boolean`.

Argument	*Type*
`aVarName`	`String`

Remarks: a variable is constant if defined using the `AddConstant` method, see page 21.

ToDblArray : Changes data type of elements from the given array to `Double` data type.
Method Type: `Function`
Returns Type: `Array:Double`.

Argument	*Type*
`aArray`	`Variant`

Remarks: jagged arrays aren't admitted by the method.

4.1.3 Enumerations

DecimalSymbol : Options for symbol used as a decimal character in parsing and evaluation stages. Also see `DecimalSymbol` property at section 4.1.1, page 16, for details on how to use it.

Name	*Value*
`dsDot`	`0`
`dsComma`	`1`

ExpressionErrors : List of errors types returned by the library. Also see `ErrorType` property at section 4.1.1, page 17, for more details.

OperatorToken : The table 1 holds a list of library supported operators. Also see section 2.1, page 6, for more details.

Name	*Value*
errNone	0
errUnbalancedBrackets	1
errSyntaxError	2
errEvalError	3
errVariableNotAssigned	4
errMissingArgsOrTooManyArgs	5

Name	*Value*
otNull	0
otSum	1
otDiff	2
otMultiplication	3
otDivision	4
otIntDiv	5
otPower	6
otMod	7
otEqual	101
otNotEqual	102
otGreaterThan	103
otLessThan	104
otGreaterThanOrEqual	105
otLessThanOrEqual	106
otLike	107
otLogicalAND	201
otLogicalOR	202
otLogicalXOR	203

Table 1 Supported operators

4.2 VBAexpressionsScope.cls

4.2.1 Properties

In this section only the properties not covered in later sections will be discussed, on the understanding that the properties related to the treatment and assignment of values to variables, covered in section 4.1.1, are merely calls to properties of the `VBAexpressionsScope.cls` class module. Refer to pages 16 and 20 for further information.

AssignedArray : Returns True if the given variable has an array assigned.
Property Type: `Boolean`.

Accesor	*Syntax*
Get	`expression.AssignedArray`

Remarks: use this property to determine whether the value of a given variable has been assigned with the `VarValue2` property, see section 4.1.1 on page 20.

DefinedScope : Returns `True` if all stored variables have an assigned value or when there are no stored variables, `False` when the value of any variable is missing.
Property Type: `Boolean`.

Accesor	*Syntax*
`Get`	`expression.DefinedScope`

VariablesCount : Returns the count of stored variables.
Property Type: `Long`.

Accesor	*Syntax*
`Get`	`expression.VariablesCount`

4.2.2 Methods

As stated in section 4.2.1, only not covered methods will be discussed.

AddConstant : Adds a variable with constant value to the current evaluation scope.
Method Type: `Sub`
Returns Type: `None`.

Argument	*Type*
`aValue`	`String`
`aKey`	`String`

Remarks: defining constant is useful when a expression has variables that remains unchanged between executions. A call to the `FillPredefinedVars` method can be required.

ConstantsInit : Erases constants defined in the current scope.
Method Type: `Sub`
Returns Type: `None`.

Argument	*Type*
`None`	`None`

Remarks: after initialization, only the values of `pi` and `e` are retained as constants.

CopyScope : Returns a copy of the given scope.
Method Type: `Function`
Returns Type: `Object:VBAexpressionsScope`.

Argument	*Type*
`sourceScope`	`Object:VBAexpressionsScope`

FillPredefinedVars : Assigns values to the variables contained in the parsed expression that have been previously defined as constants.
Method Type: `Sub`
Returns Type: `None`.

Argument	*Type*
`None`	`None`

IsConstant : Determines whether a given variable was defined as a constant.
Method Type: `Function`
Returns Type: `Boolean`.

Argument	*Type*
`aVarName`	`String`

VariablesInit : Erases all variables defined in the current scope.
Method Type: `Sub`
Returns Type: `None`.

Argument	*Type*
`None`	`None`

5 Notes for LibreOffice Users

LibreOffice (LO BASIC) has some important limitations in the use and implementation of class modules. This particularity required the adaptation of several elements to make them properly working.

Note

The library is brought to users with 57 test methods (434 LoC) to ensure that most of the functions are not broken, but there is no guarantee that it is free of bugs or crashes when run in LO BASIC. Visit the extension release available at official website[a].

[a] https://extensions.libreoffice.org/en/extensions/show/70059

5.1 Variables Treatment

One of the changes made was the replacement by methods and functions of those properties with more than one parameters, because the second parameter was simply ignored by the language. This change particularly affects the way in which variable values are accessed and assigned. For illustration, the `ImplicitVariables` procedure, refer to page 19, is written in listing 3 for LibreOffice use.

```
1  Private Sub ImplicitVariables_LOBasic()
2      Dim expr As Object
3
4      Set expr = New VBAexpressions
5      With expr
6          .Create "GET('c'; a*b)"
7          .LetVarValue("a",-3)
8          .LetVarValue("b", 9): .Eval
9          .Create "ABS(c)", False 'Keep scope
10         MsgBox .expression &" = " & .Eval &" for " &.
       CurrentVarValues
11     End With
12 End Sub
```

Listing 3: Working with variables in LO Basic

In listing 3, the first difference is that you start by defining the implicit variable and assigning values to the variables on which it depends using the `LetVarValue` method. Once all the variables are defined, you create an expression that uses them without reinitialize their scope by calling the `Create` method with the `resetScope` option set to `False`, refer to section 4.1.2 on page 22 for more details.

Note

The `ImplicitVarValue` property is not available in LO Basic due to unwanted objects reset issue. The work around here is using the `GET` function instead for indirect variable assignments.

5.2 Recursion

Other peculiarity, no less important, is the fact that recursive procedure calls tend to cause problems in LO Basic when invoked from class modules. This led to the implementation of non recursive algorithm for the `GCD` UDF function. Here the root of problems is the exception triggered when trying to increment a static variable.

Warning

The extension's behavior has not been tested in others recursive UDFs. The use of this kind of procedures is at the user's discretion.

5.3 Library Loading

Another draw back is the requirement for library loading before the first use. Users must invoke a load procedure on an event[6], for files saved on trusted locations[7], or in a main procedure before triggering evaluation methods. The function described in listing 4 has been provided for these purposes.

```
1 Sub LoadVBAexpressions
2     GlobalScope.BasicLibraries.loadLibrary("VBAExpressionsLib
      ")
3 End Sub
```

Listing 4: Loading VBA Expressions library in LO Basic

6 Main Use Cases

The basic behavior of string expression evaluators allows them to be used in a variety of ways. The main function of these is to act as back-end programs that receive an input, either from an interface or through direct interaction with the host application, and return a result communicated to the procedure from which they are invoked. This is illustrated in figure 4.

The *VBA Expressions* evaluator receives and returns text strings (`String I/O`), so it is sometimes necessary to implement specialized interfaces.

[6] https://help.libreoffice.org/latest/ro/text/sbasic/shared/01040000.html
[7] https://help.libreoffice.org/latest/en-GB/text/shared/optionen/macrosecurity_ts.html?&DbPAR=IMPRESS&System=UNIX

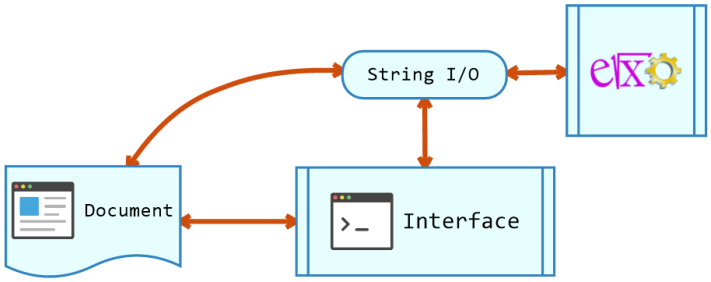


Figure 4: Expressions evaluator flow chart

6.1 Provide Evaluation Capability to other Applications

Excel has an `Evaluate()` function, which allows arbitrary strings of text to be evaluated. *Calc*, on the other hand, lacks this functionality. In the same vein, the writing applications *Word* or *Writer* are not well known for their computational capabilities. These are areas where an expression evaluator can enrich the user experience.

A typical example of this case is when an user input is requested through an input-box, a label or some other element and the numerical value of the input is calculated and the result is returned to a predetermined object.

6.2 Data Management

Segregating records, determining those that do or don't meet certain conditions specified for the fields that compose them, is another possible use of an expression evaluator. A good example of this particular case is the filtering function implemented by *CSV Interface*[8]. This implementation uses an interface to interpret user input, which is responsible for generating a string input compatible with *VBA Expressions* syntax. The interface then calls the evaluator and receives a string with the evaluation result.

This technique results in a very powerful filtering engine, capable of supporting virtually any filter parameters that can include functions and a wide variety of operators.

6.3 Linear Algebra

Since *VBA Expressions* supports a variety of input types, there are multiple fields within algebra in which it can be used.

Functions Plotting: Graphing a function is nothing more than determining the values of the ordinate when a given value is assigned to the abscissa. This can be modeled within a loop as illustrated in listing 5.

```
'.....Code here
For j = m To n
    dataPair(0) = j 'Abscissa
    dataPair(1) = CDbl(.Eval("x=" & j)) 'Ordinate
Next j
'.....Code here
```

Listing 5: Function plotting pseudocode

[8] https://ws-garcia.github.io/VBA-CSV-interface/api/methods/filter.html

Systems of Linear Equations:

Solving a system of linear equations is a basic task in algebra studies. Listing 6 shows the code to solve the following system

$$\begin{cases} x & +4z = 2 \\ x + y + 6z = 3 \\ -3x & -10z = 4 \end{cases}$$

```
Private Sub LinearSystemSolve()
    Dim Evaluator As VBAexpressions

    Set Evaluator = New VBAexpressions
    With Evaluator
        .Create "LUSOLVE(ARRAY(a;b;c);{{'x';'y';'z
    '}};{{2;3;4}};True)"
        Debug.Print .Eval("a={1;0;4};b={1;1;6};c={-3;0;-10}")
    End With
End Sub
```

Listing 6: Solving linear system

Executing gives

```
x = -18; y = -9; z = 5
```

Note

For array output, replace the `True` parameter with `False` in `.Create "LUSOLVE(...;True)"`. The output then is `{{-18; -9; 5}}`. There are other functions that allow users to solve systems of linear equations in *VBA Expressions*, see section 10, on page 38, for more information.

Warning

LibreOffice users should be aware that `Debug.Print` functionality is not available in LO Basic.

Overdeterminated Systems:

Solving systems of over-determined equations involves finding the combination of variables that minimize the residuals. This is achieved by means of QR decomposition. Listing 7 shows how to solve the system

$$\begin{bmatrix} 2 & 4 \\ -5 & 1 \\ 3 & -8 \end{bmatrix} \begin{bmatrix} 10 \\ -9.5 \\ 12 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{1}$$

```
Private Sub LSQRsolve()
    Dim Evaluator As VBAexpressions

    Set Evaluator = New VBAexpressions
    With Evaluator
        .Create "MROUND(LSQRSOLVE(A;b);4)"
        Debug.Print .Eval("A={{2;4};{-5;1};{3;-8}};b
    ={{10;-9.5;12}}")
    End With
```

```
End Sub
```

Listing 7: Solving over-determined system

Executing gives

```
{{2.6576};{-0.1196}}
```

Matrices: The library can be used to perform operations involving matrices: addition, difference, multiplication (matrix by matrix, matrix by vector, matrix by scalar). This makes it ideal for coding solutions for high school students. Details of the use of these types of functions are described in section 8, page 35.

6.4 Statistics and Data Analysis

Although the library has a moderate amount of relevant statistical functions, the most outstanding functionality is the analysis of regressions over a set of observations.

Regressions: Regressions are used in data analysis for explain variables relationship. This is a basic tool is statistic. In *VBA Expressions* users can perform linear, polynomial, exponential, logarithmic, power and multi-variate linear regressions. The program can compute the best fitting curve to explain the observations through given predictors. The listing 8 shows how to compute multi-variate linear regression over a set of observations including predictors interactions.

```
Private Sub MultiVariateReg()
    Dim Evaluator As VBAexpressions

    Set Evaluator = New VBAexpressions
    With Evaluator
        .Create "MLR(X;Y;True;'Height:Width';'Height;Width')"
        Debug.Print .Eval("X
    ={{1;1};{2;2};{3;3};{4;4};{5;1};{6;2};{7;3};{8;4}};Y
    ={{2;4.1;5.8;7.8;5.5;5.2;8.2;11.1}}")
    End With
End Sub
```

Listing 8: Multiple linear regression

```
{{0.8542 + 0.4458*Height + 0.945*Width +
0.0792*Height*Width};{0.947;0.9072}}
```

This is a function with fairly flexible arguments that allow computations to be performed in a variety of ways. For example, to obtain only one array with the coefficients of independent variables/predictors only requires changing the value specified in `True` to `False`. The parameter `'Height;Width'` defines predictors and `'Height:Width'` its interactions. For non named predictors, nominal I/O syntax, like `'X1:X2'` is required to define predictors interactions.

Note

VBA Expressions v3.2.11 doesn't performs ANOVA analysis, this is a feature that is currently under review for implementation. More details on this function usage are available on section 10.

Trend Lines: The trend line is a functionality provided by spreadsheet programs such as Excel or Calc. This feature is not integrated in other applications within their respective office packages. The listing 9 shows how to compute a 4th degree polynomial trend line

```
Private Sub TrendLine()
    Dim Evaluator As VBAexpressions

    Set Evaluator = New VBAexpressions
    With Evaluator
        .Create "FIT(A;1;4)"
        Debug.Print .Eval("A
    ={{-2;40};{-1;50};{0;62};{1;58};{2;60}}")
    End With
End Sub
```

Listing 9: Computing polynomial trend line

The output after executing is an array with the form `\{{POLYNOMIAL};{R-squared VALUE}}` as shown

```
{{62 + 3.6667*x -9.6667*x^2 + 0.3333*x^3 +
1.6667*x^4};{1}}
```

6.5 Engineering and Physics

Zeroing Functions: In the field of engineering and physics, it is often necessary to zeroing functions in order to calculate their critical points. For example, recall the projectile's displacement problem in listing 2, page 14. The output tell us that in some point, in range $0 \leq t \leq 1.5$, the projectile is at the same height at which it was fired. For determine this precise instant, a program like listing 10 can be used.

```
Private Sub ProjectileDisplacement2()
    Dim expr As VBAexpressions

    Set expr = New VBAexpressions
    With expr
        .Create "FZERO('5*t -9.81*t^2/2';0;1.5)"
        Debug.Print .Eval
    End With
    Set expr = Nothing
End Sub
```

Listing 10: Zeroing projectile displacement function

The result printed is `'t = 1.01936799184199'`.

Note

Use `FZERO('5*t -9.81*t^2/2';0;1.5;False)` to get this non text formatted numerical result: `1.01936799184199`.

Warning

Zeroing a function with more than one variable will result in an evaluation error.

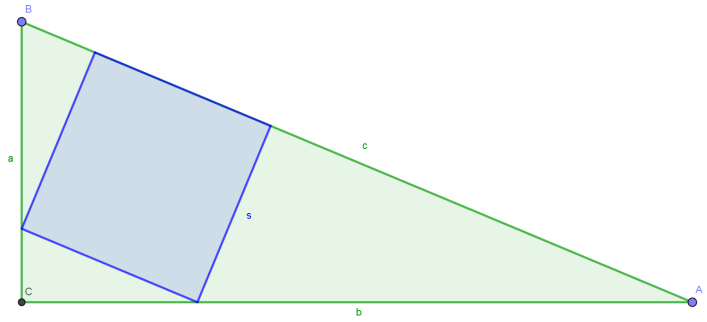


Figure 5: Geometrical problem

6.6 Analytical Geometry

Solving geometrical problems:

The versatility of *VBA Expressions*, starting on *v3.2.11*, allows its users to apply their creativity to solve a wide variety of geometrical problems in an automated way. To demonstrate this we will solve the problem shown in figure 5. We are asked to calculate the area of the largest square inscribed in a right triangle of legs a and b. The square corners configuration is: one corner intersecting the leg a, another with intersection on leg b and the others corners intersecting the hypotenuse c.

The problem can be solved analytically using proportional triangles and some trigonometrical identities, but we'll solve it using analytical geometry. First, we define the point O as the square's intersection in leg b. The right angle is assumed to be located at the origin of Cartesian plane. In the same way we define P as one intersection of the square with the hypotenuse and Q as the intersection with the leg a. Since the figure we are looking for is a square, all sides have a dimension s, hence the problem is reduced to ensuring that the magnitudes $\overline{OP}$ and $\overline{OQ}$ have the same value. The code to solve the problem is shown in listing 11.

```
1  Public Function LargestSquareInRATriangle(A As Double, B As
       Double) As String
2      Dim contFunct As String
3      Dim evalHelper As VBAexpressions
4      Dim xVal As String
5      Dim Area As String
6
7      contFunct = "FZERO('DISTANCE(LINESINTERSECT(PERPENDICULAR
       (" _
8      & "{{" & B & ";0};{0;" & A & "}};{{x;0}})" _
9      & ";{{" & B & ";0};{0;" & A & "}})" _
10     & ";{{x;0}})" & "-" _
11     & "DISTANCE(LINESINTERSECT(PERPENDICULAR(PERPENDICULAR("
       _
12     & "{{" & B & ";0};{0;" & A & "}};{{x;0}})" _
13     & ";{{x;0}})" _
14     & ";{{0;0};{0;" & A & "}})" _
15     & ";{{x;0}})'" _
16     & ";0;" & B & ")"
17     Set evalHelper = New VBAexpressions
18     With evalHelper
19         .Create contFunct
20         .Eval
21         If .ErrorType = errNone Then
22             xVal = Mid(.Result, 2, Len(.Result) - 2)
23             .Create "(DISTANCE(LINESINTERSECT(PERPENDICULAR("
        _
24                     & "{{" & B & ";0};{0;" & A & "}};{{x;0}})
       " _
```

```
                    & ";{{" & B & ";0};{0;" & A & "}})" _
                    & ";{{x;0}}))^2", False
            .Eval xVal
            If .ErrorType = errNone Then
                Area = "Area = " & .Result
                LargestSquareInRATriangle = xVal & "; " &
    Area
            End If
        End If
    End With
End Function
```

Listing 11: Computing the area for the largest square in a right triangle

Since the magnitude of the segments $\overline{OP}$ and $\overline{OQ}$ must be the same, the `FZERO` function can be used to determine the coordinate $O = (x, 0)$ by iteratively computing the value of x that satisfies the continuous equality $\overline{OP} - \overline{OQ} = 0$. After execute the call `LargestSquareInRATriangle(5,12)`, the result returned is

```
x = 3.14410480351349; Area = 11.6016094276853
```

The listing 11 is a masterpiece demonstrating the power and versatility offered by the latest versions of *VBA Expressions*.

7 Variables Definition and Assignment

As discussed in section 3.2, an expression may contain variables. These, themselves, may or may not be related to each other, belong or not to the same object. The scope of variables is derived from this conjecture.

7.1 Scope of Variables

→ Section 4.1.1, page 17

In *VBA Expressions*, the scope of a variable is defined by the evaluation scope of a given object. This scope is a property defined within all `VBAexpressions` objects. In this context, variables may or not share a certain scope defined through the `VBAexpressionsScope.cls` module.

This feature makes possible to assign the same set of variables to different objects for evaluating different expressions. This is illustrated in listing 12.

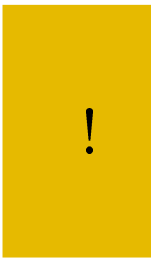
```
Private Sub ManagingScopes()
    Dim scope As VBAexpressionsScope
    Dim evaluator1 As VBAexpressions
    Dim evaluator2 As VBAexpressions

    Set scope = New VBAexpressionsScope
    Set evaluator1 = New VBAexpressions
    Set evaluator2 = New VBAexpressions

    ' Scope definition
    With scope
        .VarValue("x") = 3
        .VarValue("y") = -2
        .VarValue("z") = 7
    End With
    ' Setting evaluator 1
    With evaluator1
        .Create "x^2+y-2z"
        Set .EvalScope = scope
        Debug.Print .expression & " = " & .Eval
    End With
```

```
22      ' Setting evaluator 2
23      With evaluator2
24          .Create "x+y+z"
25          Set .EvalScope = scope
26          Debug.Print .expression & " = " & .Eval
27      End With
28      Set evaluator1 = Nothing
29      Set evaluator2 = Nothing
30  End Sub
```

Listing 12: Managing scopes

Executing the listing 12 results in this output

```
x^2+y-2z = -7
x+y+z = 8
```



Warning

This procedure does NOT work in LibreOffice. Users should handle the scope using the `GET` function as shown in listing 13. This is because, for some reason, LO Basic fails to handle objects passed as references to *VBA Expressions* methods.

```
1   Private Sub ManagingScopes()
2       Dim evaluator As Object
3
4       Set evaluator = New VBAexpressions
5       With evaluator
6           ' Setting initials values
7           .create "GET('x';3); GET('y';-2); GET('z';7)": .Eval
8
9           ' Create expression without restoring scope
10          .Create "x^2+y-2z",False
11          .Eval 'Evaluate
12          MsgBox .expression & " = " & .Result
13
14          ' Create another expression, same scope
15          .Create "x+y+z",False: .Eval 'Evaluate
16          MsgBox .expression & " = " & .Result
17      End With
18      Set evaluator = Nothing
19  End Sub
```

Listing 13: Managing scopes in LibreOffice

7.2 Accessing Variables

As seen in previous examples, it is possible to assign variables values before and after creating an expression to be evaluated. Early assignment requires variables to be identified by name, while late assignment allows variables to be identified by their position number within the expression.

Note

Variables are stored as they appear in the expression, so the first variable, from left to right, has the 0 index. This is demonstrated in listing 14.

```
Private Sub AccessingVariables()
    Dim evaluator As VBAexpressions
    Dim arrVarNames() As String
    Dim i As Integer

    Set evaluator = New VBAexpressions
    With evaluator
        .Create "x^2+y-2z"
        'Accesing by index
        For i = 0 To .EvalScope.VariablesCount
            .VarValue(CStr(i)) = i
        Next i
        Debug.Print .CurrentVarValues
        'Accesing by names
        arrVarNames = Split(.CurrentVariables, "; ")
        For i = LBound(arrVarNames) To UBound(arrVarNames)
            .VarValue(arrVarNames(i)) = "'" & i & "'"
        Next i
        Debug.Print .CurrentVarValues
    End With
    Set evaluator = Nothing
End Sub
```

Listing 14: Accessing to expressions variables

The output by executing listing 14 is as follows

```
x = 0; y = 1; z = 2
x = '0'; y = '1'; z = '2'
```

7.3 Special Uses

Using the `GET` function is an example of a special evaluation scope use. This mechanism implements a bypass of the evaluator to support a prior evaluation of expressions whose implicit variables are defined as the expression is evaluated. In other words, the evaluator does NOT know the number of variables within the expression and invokes routines that allow it to declare and access them at run time.

This `GET` function is used in listing 13 to recreate a multi-use scoping behavior and can be useful when evaluating unusual expressions.

8 Working with Matrices/Arrays

Matrices are another fundamental part of algebra, manipulating them is, therefore, a basic task of any system that tries to provide assistance in the field of mathematics.

8.1 Some Key Notes

→ See page 9

As discussed in section 2.2.4, an array is defined as a list of lists. *VBA Expressions* does not support the Array data type, but offers specialized functions to convert them to a compatible input type (refer to pages 21 and 22). This makes assigning values to array variables or dump calculations to them extremely simple.

The syntax `{List[;List]}` defines an array. In this case the `List` parameter is a vector and must have the syntax `{SubExpr[;SubExpr]}`. Refer to section 2.2.1 in page 8.

The listing 15 shows how to calculate the internal rate of return presented in Microsoft© website[9] with out providing any guess. The result is nearly the same (`'-44.35%'`), with only rounding difference.

```
Private Sub IRRFunctTest()
    Dim expr As VBAexpressions

    Set expr = New VBAexpressions
    With expr
        .Create "FORMAT(IRR({{-70000;12000;15000}};true);'Percent')"
        Debug.Print .Eval
    End With
    Set expr = Nothing
End Sub
```

Listing 15: Calculating internal rate of return

Array support does not end with just supporting this type of input or output. The library includes powerful functions to perform calculations on arrays. Listing 16 contains an example demonstrating that multiplying a matrix by its inverse results in the identity matrix.

```
Public Sub MatrixMult()
    Dim expr As Object

    Set expr = New VBAexpressions
    With expr
        'Create a matrix
        .Create "GET('f';{{1;0;4};{1;1;6};{-3;0;-10}})": .Eval
        'Compute inverse
        .Create "GET('g';INVERSE(f))", False: .Eval
        'Compute multiplication
        .Create "MMULT(f;g)", False
        Debug.Print .Eval
    End With
    Set expr = Nothing
End Sub
```

Listing 16: Operations with matrices

8.2 Matrices Overloading

VBA Expressions allows referencing elements of those arrays stored through the `VarValue2` property or by using the `GET` function. The library use the variables overloading broad concept to explain this behavior for dealing with variable, which in this particular case are arrays.

To access the elements of an array, row and column indices must be provided as required. Listing 17 shows code that accesses the diagonal elements of the returned matrix/array after executing a series of evaluations.

```
Private Sub MatricesAccess()
    Dim expr As Object
```

[9] https://support.microsoft.com/en-gb/office/irr-function-64925eaa-9988-495b-b290-3ad0c163c1bc

```
3
4       Set expr = New VBAexpressions
5       With expr
6           .Create "GET('A';{{2;1;3};{3;-2;-1}}); " _
7           & "GET('B';{{2;3};{1;-5};{-2;4}})": .Eval
8           .Create "GET('C';MMULT(A;B))", False: .Eval
9           .Create "ROUND(SUM(SIN(C[0;0]);SIN(C[1;1]));4)",
        False: .Eval
10          MsgBox .expression & " | " & _
11          "Sum of sines for diagonal elements on matrix C: " _
12          & .Result & " | " & " for: " & .CurrentVarValues
13      End With
14      Set expr = Nothing
15  End Sub
```

Listing 17: Accessing to elements in a matrix/array

→ Refer to section 2.2.4 in page 9

@ The syntax for accessing elements in a matrix/array requires the use of square-brackets: `Arr[row;col]`. Trying to access elements with wrong numbers of indices will result in an error.

9 Managing User Defined Functions

One of the most exciting features offered by the library is its ability to trigger procedures from user-supplied class modules. This is very useful when having modules with functions that you want to make available for use in expressions. This set of functions is known as UDF.

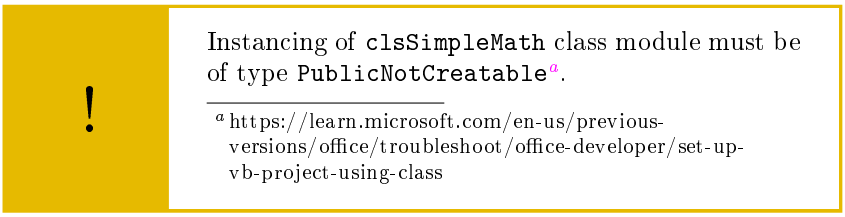
! *VBA Expression* passes an array of type `String` as a single argument when calling a UDF. All UDFs are subject to this condition, so functions must be modified to handle their arguments in a single argument of type Variant.

→ See also section 4.1.2 in page 22

As mentioned above, UDFs need to be declared and registered. The `DeclareUDF` method is used to handle user-defined functions. The registration of UDFs must be done manually in the `VBAcallBack.cls` module. For illustrative purposes the declaration of the functions contained in a simple module called `clsSimpleMath` will be exemplified:

1. Create a new class module named `clsSimpleMath.cls`. For LibreOffice users, add a standard module[10] named `clsSimpleMath` add put in it the lines shown in listing 18.
2. In `clsSimpleMath` module add the code shown in listing 19.
3. Go to the `VBAcallBack` module and register the newly created `clsSimpleMath` module by adding to it the line of code shown in listing 20.
4. Once you have completed these steps, in the module of your choice, add the code shown in listing 21. It declares and uses the registered UDF.

[10] https://help.libreoffice.org/latest/en-US/text/sbasic/shared/01030400.html



Instancing of `clsSimpleMath` class module must be of type `PublicNotCreatable`[a].

[a] https://learn.microsoft.com/en-us/previous-versions/office/troubleshoot/office-developer/set-up-vb-project-using-class

```
Option Explicit
Option Compatible
Option VBASupport 1
Option ClassModule
```

Listing 18: Heading for clsSimpleMath class module

```
Option Explicit
Public Function SimpleProduct(ByRef aValues As Variant) As
    Double
    Dim firstNumber As Double
    Dim secondNumber As Double

    firstNumber = CDbl(aValues(LBound(aValues)))
    secondNumber = CDbl(aValues(UBound(aValues)))
    SimpleProduct = firstNumber * secondNumber
End Function
```

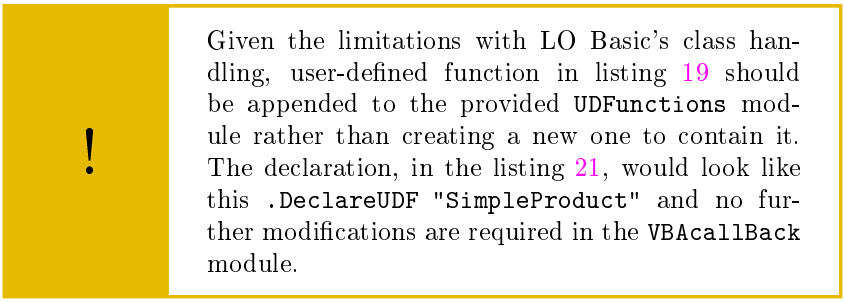
Listing 19: Simple product listing

```
Public SimpleMath As New clsSimpleMath
```

Listing 20: Registering clsSimpleMath module

```
Private Sub DeclareAndUseUDFs()
    Dim expr As Object

    Set expr = New VBAexpressions
    With expr
        .DeclareUDF "SimpleProduct", "SimpleMath"
        .Create "SimpleProduct(3;5)": .Eval
        MsgBox .Eval
    End With
    Set expr = Nothing
End Sub
```

Listing 21: Declaring and use UDFs



Given the limitations with LO Basic's class handling, user-defined function in listing 19 should be appended to the provided `UDFunctions` module rather than creating a new one to contain it. The declaration, in the listing 21, would look like this `.DeclareUDF "SimpleProduct"` and no further modifications are required in the `VBAcallBack` module.

10 Library Built-in Functions

This section will list all the functions supported by the library. For functions incorporated natively in VBA®or Excel, their syntax will be noted

along with a reference to their official documentation archive hosted at https://web.archive.org. All other functions will have their corresponding keynotes and usage examples when the concept is not covered in earlier sections.

The functions will be grouped into five categories, namely: *Transcendental*; *Mathematical*; *Statistical*; *Financial*; *Date, Time and String*; *Programming*.

Although functions will be described using their definition in capital letters, it should be noted that function calls are not case sensitive. For example, `COS(x) = Cos(x) = cos(x)`.

10.1 Transcendental Functions

10.1.1 ABS

Description : Returns the absolute value of a number.

Syntax : `ABS(number)`

Keynotes : Refer to ABS function Microsoft©documentation for more details.

10.1.2 ACOS

Description : Returns, in radians, the arc-cosine of a number.

Syntax : `ACOS(number)`

Keynotes : Refer to ACOS function Microsoft©documentation for more details.

10.1.3 ASIN

Description : Returns, in radians, the arc-sine of a number.

Syntax : `ASIN(number)`

Keynotes : Refer to ASIN function Microsoft©documentation for more details.

10.1.4 ATN

Description : Returns, in radians, the arc-tangent of a number.

Syntax : `ATN(number)`

Keynotes : Refer to ATAN function Microsoft© documentation for more details.

10.1.5 AVG

Description : Returns the average of given numbers.

Syntax : `AVG(list)`.

Keynotes : The `list` argument must be composed entirely by numeric data, otherwise an error is returned. Refer to section 2.2.5 in page 10 for more details on `list` syntax.

10.1.6 CEIL

Description : Returns the smallest integer greater than or equal to the given number.

Syntax : `CEIL(number)`.

Keynotes : This function differs from the Excel implementation. The concept used is purely mathematical, so literal integers are returned with no multiplies or significance. See this publication to known more about the behavior.

10.1.7 COS

Description : Returns the cosine for a given radian angle.

Syntax : `COS(angle)`

Keynotes : Refer to COS function Microsoft© documentation for more details.

10.1.8 EXP

Description : Returns e raised to the power of a given number.

Syntax : `EXP(number)`

Keynotes : Refer to EXP function Microsoft© documentation for more details.

10.1.9 FLOOR

Description : Returns the largest integer less than or equal to the given number.

Syntax : `FLOOR(number)`

Keynotes : This function differs from the Excel implementation. The concept used is purely mathematical, so literal integers are returned with no multiplies or significance. See this publication to known more about the behavior.

10.1.10 LGN

Description : Returns the logarithm of a number to a specified base.

Syntax : `LGN(number;base)`

Keynotes : Refer to LOG function Microsoft©documentation for more details.

10.1.11 LN

Description : Returns the natural logarithm of a number.

Syntax : `LN(number)`

Keynotes : Refer to LN function Microsoft©documentation for more details.

10.1.12 LOG

Description : Returns the base-10 logarithm of a number.

Syntax : `LOG(number)`

Keynotes : Refer to LOG10 function Microsoft©documentation for more details.

10.1.13 MAX

Description : Returns the maximum value in a list of numbers.

Syntax : `MAX(list)`.

Keynotes : The `list` argument must be composed entirely by numeric data, otherwise an error is returned. Refer to section 2.2.5 in page 10 for more details on `list` syntax.

10.1.14 MIN

Description : Returns the minimum value in a list of numbers.

Syntax : `MIN(list)`.

Keynotes : The `list` argument must be composed entirely by numeric data, otherwise an error is returned. Refer to section 2.2.5 in page 10 for more details on `list` syntax.

10.1.15 PERCENT

Description : Returns the result of dividing the number by 100.

Syntax : `PERCENT(number)`.

Keynotes : No format is applied to the output.

10.1.16 POW

Description : Returns the result of a number raised to a power.

Syntax : `POW(number;power)`.

Keynotes : Refer to POWER function Microsoft©documentation for more details.

10.1.17 ROUND

Description : Rounds a number to a specified number of digits.

Syntax : `ROUND(number;digits)`.

Keynotes : Refer to ROUND function Microsoft©documentation for more details.

10.1.18 SGN

Description : Returns the sign of a number.

Syntax : `SGN(number)`.

Keynotes : Refer to SIGN function Microsoft©documentation for more details.

10.1.19 SIN

Description : Returns the sine for a given radian angle.

Syntax : `SIN(angle)`.

Keynotes : Refer to SIN function Microsoft© documentation for more details.

10.1.20 SQR, SQRT

Description : Returns the positive square root of a number.

Syntax : `SQR(number)` or `SQRT(number)`.

Keynotes : Refer to SQRT function Microsoft© documentation for more details.

10.1.21 SUM

Description : Adds a lists of numbers.

Syntax : `SUM(list)`.

Keynotes : The `list` argument must be composed entirely by numeric data, otherwise an error is returned. Refer to section 2.2.5 in page 10 for more details on `list` syntax.

10.1.22 TAN

Description : Returns the tangent for a given radian angle.

Syntax : `TAN(angle)`.

Keynotes : Refer to TAN function Microsoft© documentation for more details.

10.2 Mathematical Functions

10.2.1 CHOLESKY

Description : Returns the Cholesky decomposition for a matrix.

Syntax : `CHOLESKY(matrix)`.

Keynotes : Requires positive-definite symmetric square matrix $A[0..n-1][0..n-1]$ as input. The Cholesky decomposition verifies $A = L \cdot L^T$. The result is the lower triangular matrix L. Failure of the decomposition indicates that the matrix A is not positive-definite. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.2 CHOLINVERSE

Description : Returns the inverse of a matrix computed by Cholesky decomposition.

Syntax : `CHOLINVERSE(matrix)`.

Keynotes : Limitations related to the typology of the input `matrix`, from the `CHOLESKY` function, are also valid. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.3 CHOLSOLVE

Description : Solves the linear system $A \cdot X = B$ using Cholesky decomposition.

Syntax : `CHOLSOLVE(A;X;B;[IncludeNames=False])`.

Keynotes : The argument `A` is a matrix that satisfies conditions required by the `CHOLESKY` function; `X` is an one dimensional array containing the name for each of the `n` variables in the system; `B` is a one row by n columns matrix containing the right-hand side of the system. Use the`IncludeNames` parameter to indicate if the resulting calculations will be returned as a matrix or as formatted string by including the names of variables. Refer to `Literal` argument in section 2.2.1, also to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

e. g.,

```
REM Variables: a={6;15;55};b={15;55;225};c={55;225;979}
MROUND(CHOLSOLVE(ARRAY(a;b;c);{{'x';'y';'z
    '}};{{76;295;1259}};False);4)
```

outputs the array `{{1;1;1}}`.

```
CHOLSOLVE(ARRAY(a;b;c);{{'x';'y';'z'}};{{76;295;1259}};
    False)
```

outputs

```
x = 0.999999999999988; y = 1.00000000000002; z =
    0.999999999999997
```

using the same variables values.

10.2.4 DET

Description : Returns the determinant of a matrix computed using its LU decomposition.

Syntax : `DET(matrix)`.

Keynotes : The `matrix` must be square. Although the subroutine allows the determinant of a submatrix to be computed, it is not advisable to pass a non-square matrix as an argument. Refer to the `LUDECOMP` function in section 10.2.12 for more details. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.5 DISTANCE

Description : Returns the distance between two Cartesian points.

Syntax : `DISTANCE(Point1;Point2)`.

Keynotes : Each point must be a proper matrix with one row and two columns. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.6 FZERO

Description : Finds a zero of an uni-variate function using the modified bisection method.

Syntax : `FZERO(aFunction;a;b;[IncludeVarName=False];[epsilon=1E-8])`.

Keynotes : An error will be returned if the `aFunction` contains more than one variable. The parameter `a` indicates the lower limit, towards left, while the parameter `b` represents the upper limit, towards right, of the numerical interval in which the function's zero will be searched. The parameter `IncludeVarName` is used to control the output format. Tolerance for the iterative searching is defined with the `epsilon` parameter. Refer to the listing 10, in page 31 for an usage example.

Setting `epsilon` to an inadequate value can produces unexpected iterative function fails.

10.2.7 GAMMA

Description : Returns the Gamma function, $\Gamma(x)$, value.

Syntax : `GAMMA(number)`.

Keynotes : Refer to GAMMA function Microsoft© documentation for more details.

10.2.8 GAMMALN

Description : Returns the natural logarithm of the gamma function, $\Gamma(x)$.

Syntax : `GAMMALN(number)`.

Keynotes : An error is returned when the given number is a negative integer or zero.

10.2.9 INVERSE

Description : Computes the inverse of a matrix using the LU decomposition.

Syntax : `INVERSE(matrix)`.

Keynotes : Refer to the `LUDECOMP` function in section 10.2.12 for more details. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.10 LINESINTERSECT

Description : Returns the intersection point between two lines defined by two points each.

Syntax : `LINESINTERSECT(Line1;Line2)`.

Keynotes : Each line must be a proper matrix with two rows and two columns. The function returns a `#Null` error if the given lines are parallels. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.11 LSQRSOLVE

Description : Returns the least squares solution for a given over-determined equations system.

Syntax : `LSQRSOLVE(A;b)`.

Keynotes : Given a system $A \cdot X = b$, the function returns X that minimizes the two norm of $Q \cdot R \cdot X - b$. `A` is a matrix of coefficients of the system variables, `b` is the right side vector. The procedure will return an error when matrix `A` is rank deficient. Row and columns dimensions must agree. Also see `QR` function in section 10.2.21 for more details. For usage example see listing 7 in page 29. Refer to `Literal` argument in section 2.2.1, also to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.12 LUDECOMP

Description : Returns the LU decomposition for the given matrix.

Syntax : `LUDECOMP(matrix)`.

Keynotes : The `matrix` must be square. The decomposition is achieved performing row-wise permutations over the give matrix. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.13 LUSOLVE

Description : Solves the linear system $A \cdot X = B$ using LU decomposition.

Syntax : `LUSOLVE(A;X;b;[includeNames=False])`.

Keynotes : `A` must be a square matrix. The argument `X` is an one dimentional array containing the name of each variable; the first name will be applied to the first column of coefficients, the second name to the second column and so on, `b` is the right side vector. Use the`IncludeNames` parameter to indicate if the resulting calculations will be returned as a matrix or as formatted string by including the names of variables. Refer to `Literal` argument in section 2.2.1, also to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.14 MMULT

Description : Returns the matrix product.

Syntax : `MMULT(A;B)`.

Keynotes : The function requires`A` to be a matrix. Argument `B` can be a matrix, a column vector or an scalar number. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.15 MNEG

Description : Returns negation of a matrix.

Syntax : `MNEG(matrix)`.

Keynotes : This operation is equivalent to multiply the `matrix` by `-1` scalar. Requires all values to be of numeric type. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.16 MROUND

Description : Rounds all elements of a matrix to the specified significant digits.

Syntax : `MROUND(matrix;Digits)`.

Keynotes : For usage examples see pages 29 and 44. Refer to `Literal` argument in section 2.2.1, also to `list` and `matrix` arguments in pages 10 and 35, for syntax details. See also `ROUND` function, section 10.1.17 in page 42.

10.2.17 MSUM

Description : Returns the matrix sum.

Syntax : `MSUM(A;B;[Difference=False])`.

Keynotes : Requires `A` and `B` to be matrices with equal number of rows and columns, otherwise error is returned. Setting the `Difference` parameter to `True` will adds the negation of matrix `B` to `A` by computing $A + (-B)$. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.18 MTRANSPOSE

Description : Returns the transpose of a matrix.

Syntax : `MTRANSPOSE(matrix)`.

Keynotes : The function rotates rows and columns by swapping them. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.19 PARALLEL

Description : Returns two points representing a line parallel to the given one and containing the given point.

Syntax : `PARALLEL(Line;Point)`.

Keynotes : The `Line` argument must be a proper matrix with two rows and two columns, the `Point` argument must has one row and the same amount of columns. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.20 PERPENDICULAR

Description : Returns two points representing a line perpendicular to the given one and containing the given point.

Syntax : `PERPENDICULAR(Line;Point)`.

Keynotes : The `Line` argument must be a proper matrix with two rows and two columns, the `Point` argument must has one row and the same amount of columns. Refer to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.21 QR

Description : Returns QR decomposition of a matrix using Householder reflections.

Syntax : `QR(matrix;[PositiveDiag=False])`.

Keynotes : For an m-by-n matrix A with $m \geq n$, the QR decomposition is an m-by-n orthogonal matrix Q and an n-by-n upper triangular matrix R so that $A = Q \cdot R$. When the `PositiveDiag` is `True` the function enforce signs of the diagonal elements to be positive. This function is adapted from JAMA library. Refer to `Literal` argument in section 2.2.1, also to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.2.22 REM

Description : Returns the remainder after integer division n by d.

Syntax : `REM(n;d)`.

Keynotes : Refer to MOD function Microsoft© documentation for more details.

10.2.23 SOLVE

Description : Solves the linear system $A \cdot X = B$ using Over Relaxation (SOR) iteration.

Syntax : `SOLVE(A;X;B;[IncludeNames=False])`.

Keynotes : `A` must be a square matrix containing the coefficients of all equations; `X` is an one dimensional array containing the name for each of the `n` variables in the system; `B` is a one row by n columns matrix containing the right-hand side of the system. Use the`IncludeNames` parameter to indicate if the resulting calculations will be returned

as a matrix or as formatted string by including the names of variables. Refer to `Literal` argument in section 2.2.1, also to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

> **!** This is an iterative function. It will iterate until it obtains 9 digits of precision or completes 500 cycles, whichever occurs first. The use of `CHOLSOLVE` or `LUSOLVE`, see pages 44 and 47, is recommended for more reliable solutions.

10.3 Statistical Functions

10.3.1 ACHISQ

Description : Returns the inverse of the one-tailed probability of the chi-squared distribution.

Syntax : `ACHISQ(probability;deg_freedom)`.

Keynotes : This implementation does not return errors when the probability is $p \geq 1$ like Excel does. Refer to CHIINV function Microsoft© documentation for more details. See the `CHISQ` function in section 10.3.9, page 52 for more details.

10.3.2 AERF

Description : Returns the inverse of Gauss error function (`ERF`).

Syntax : `AERF(probability)`.

Keynotes : Excel doesn't have an implementation for this function. e. g.,`AERF(0.95)` will output `1.38590382434968`, the value of the `ERF` function for the given probability. See the `ERF` function in section 10.3.10, page 52 for more details.

10.3.3 AFISHF

Description : Returns the inverse of `FISHF` function.

Syntax : `AFISHF(probability;deg_freedom1;deg_freedom2)`.

Keynotes : Refer to FINV function Microsoft© documentation for more details. See the `FISHF` function in section 10.3.11, page 52 for more details.

10.3.4 AGAUSS

Description : Returns the inverse of `GAUSS` function.

Syntax : `AGAUSS(probability)`.

Keynotes : Excel doesn't have an implementation for this function. e. g.,`AGAUSS(0.95)` will output `1.64485362695148`, the value of the `GAUSS` function for the given probability. See the `GAUSS` function in section 10.3.13, page 53 for more details.

10.3.5 ANORM

Description : Returns the inverse of `NORM` function.

Syntax : `ANORM(probability)`.

Keynotes : The function works with 2-tail p-values. For computations like the NORM.S.INV Excel function, users can do `ANORM(2*(1-0.908789))`. See the `NORM` function in section 10.3.16, page 54 for more details.

10.3.6 ASTUDT

Description : Returns the inverse of `STUDT` function.

Syntax : `ASTUDT(probability;deg_freedom)`.

Keynotes : The function works 2-tail p-values. This is equivalent to the Excel T.INV.2T. For computations like T.INV Excel function, users can do `ASTUDT(2*(1-0.75);2)`. See the `STUDT` function in section 10.3.17, page 55 for more details.

10.3.7 BETA.DIST

Description : Calculates beta distribution values from either the probability density function (PDF) or the cumulative distribution function (CDF).

Syntax : `BETA.DIST(x;alpha;beta;Cumulative;[A=0];[B=1])`.

Keynotes : setting the `Cumulative` argument to `True` will compute the CDF function, the PDF function will computed otherwise. Parameters `A` and `B` can be used to set the lower and upper bounds of the distribution. For more details see the Microsoft BETA.DIST documentation.

10.3.8 BETAINV

Description : Returns the inverse cumulative beta inverse probability distribution function.

Syntax : `BETAINV(probability;alpha;beta)`.

Keynotes : The returned value is as if `probability` = `BETADIST`$(x, ...)$, `BETAINV(probability,...)` = x. The cumulative beta distribution can be used in project planning to determine likely completion times given a variability and an expected completion time.

10.3.9 CHISQ

Description : Returns the one-tailed (right-tail) probability value for a chi-square (χ^2) test.

Syntax : `CHISQ(x;deg_freedom)`.

Keynotes : The returned value represent the area under the χ^2 distribution from the χ^2 value to positive infinity, given the chi-square value and the degrees of freedom. This is equivalent to the Excel CHISQ.DIST.RT function.

10.3.10 ERF

Description : Returns the value of the Gauss error function.

Syntax : `ERF(x)`.

Keynotes : The returned value represent the area under the error function from 0 to x, given the limit of integration x. For compute values between lower and upper limits, like the Excel ERF function, users can do `ERF(Upper_limit)-ERF(Lower_limit)`.

10.3.11 FISHF

Description : Returns the probability value of an F-test given the F-value, numerator degrees of freedom, and denominator degrees of freedom.

Syntax : `FISHF(Fvalue;deg_freedom1;deg_freedom2)`.

Keynotes : You can use this function to determine whether two data sets have different degrees of diversity. The function doesn't returns error for negative `Fvalue`. Also see the Excel F.DIST.RT function.

10.3.12 FIT

Description : Returns a least squares regression curve fitting.

Syntax : `FIT(data;fitType;[polDegree])`.

Keynotes : The `data` parameter is an array holding data pairs (x, y). Users can choose from different fitting curves by selecting between `fitType` options:

- `fitType=1`: `Polynomial` $\rightarrow y = a + b \cdot x + c \cdot x^2 + ... + m \cdot x^n$
- `fitType=2`: `Exponential` $\rightarrow y = a \cdot e^{b \cdot x}$
- `fitType=3`: `Exponential` $\rightarrow y = a \cdot b^x$
- `fitType=4`: `Power` $\rightarrow y = a \cdot x^b$
- `fitType=5`: `Logarithmic` $\rightarrow y = a \cdot ln(x) + b$

The `polDegree` parameter is only required when the `fitType` is set to `1`. A straight line can be fitted calling the function with two or three arguments: `FIT(A;1)`; `FIT(A;1;1)`. For usage example refer to listing 9 in page 31.

10.3.13 GAUSS

Description : Returns the probability that a member of a standard normal population will fall between the mean and `z` standard deviations from the mean.

Syntax : `GAUSS(z)`.

Keynotes : The result is 0.5 off from the cumulative distribution function (CDF), given the upper limit of integration `z`, so $CDF(mean := 0; sd := 1; z) = GAUSS(z) + 0.5$. Refer to FINV function Microsoft© documentation for more details.

10.3.14 IBETA

Description : Returns the cumulative distribution function (CDF) for the beta distribution.

Syntax : `IBETA(x;alpha;beta)`.

Keynotes : The result represents the area under beta distribution from 0 to x, given values for the shape parameters β and α and the point at which to evaluate the function.

This implementation uses the incomplete beta function, so x must satisfy $0 \leq x \leq 1$ or an error will be returned. See the `BETA.DIST` function in page 51 if full compatibility with Excel's BETA.DIST function is needed.

10.3.15 MLR

Description : Performs a multiple linear regression over a model with more than two regressors/predictors.

Syntax : `MLR(X;Y;formatOutput;[PredInteractions];[PredNames])`.

Keynotes : Parameter `X` represents the model, being a N by K array with K regressors variables for all the N observations. The `Y` is a N rows vector with a total N observations for the model. The output can be formatted as polynomial by setting the `formatOutput` parameter to `True`, an array is returned otherwise.

The function allows regress models with interactions between predictors. The parameter `PredInteractions` can be used altogether with `PredNames`, not mandatory parameter, to define predictors interactions properly, both being literal strings. The following conditions must be fulfilled by the `PredInteractions` and `PredNames` parameters:

1. The syntax `'X#:X#;...;X#:X#'` must be used to define predictors interactions with nominal names. The `#` symbol indicates the position for predictor `X#` in the model array.
2. The alternative syntax `'$#:$#;...;$#:$#'` must be used to define interactions with named predictors. Here the symbol `$` refers to a predictor name from the `PredNames` list. If named predictors are used, the list of names must be supplied.
3. `PredNames` must follows the syntax `'$;$;$'`. It is mandatory to provide a list of names for all the K named predictors in the model.

For a predictors relation `X{i}:X{j}` the method will try to ensure $i \geq j$.

See listing 8, page 30 for usage example. Refer to `Literal` argument in section 2.2.1, also to `list` and `matrix` arguments in pages 10 and 35, for syntax details.

10.3.16 NORM

Description : Returns the two-tailed probability value for standard normal curve, given a z-score.

Syntax : `NORM(z)`.

Keynotes : The result is the two-tailed probability from $\pm z$ to ∞ on both tails of the distribution. For computations like the NORM.S.DIST Excel function, users can do `1-NORM(1.333333)/2`. See also the `ANORM` function in section 10.3.5, page 51.

10.3.17 STUDT

Description : Returns the two-tailed probability values of a t-test, given the t-value and the degrees of freedom.

Syntax : `STUDT(t;deg_freedom)`.

Keynotes : For one-tailed probabilities computations, like the T.DIST Excel function, users can do `1-STUDT(60;1)/2`. See also the `ASTUDT` function in section 10.3.6, page 51.

10.3.18 TINV

Description : Returns the one-tailed or two-tailed t-value (Student t-value).

Syntax : `TINV(confidence;deg_freedom;tOption)`.

Keynotes : Setting the `tOption` parameter to `1` will compute one-tailed t-value, setting it to `2` computes two-tailed t-value. This function behave exactly like the Excel T.INV function for one-tailed computations, but the results provided are rounded to 8 precision digits.

10.3.19 TINV_1T

Description : Returns the one-tailed t-value (Student t-value).

Syntax : `TINV(confidence;deg_freedom)`.

Keynotes : See `TINV` function.

10.3.20 TINV_2T

Description : Returns the two-tailed t-value (Student t-value).

Syntax : `TINV(confidence;deg_freedom)`.

Keynotes : This function behave like the online iCalculator, to get results like the Excel T.INV.2T function users can do `TINV_2T(1-0.546449;60)`. See also the `TINV` function.

10.4 Financial Functions

10.4.1 DDB

Description : Returns the depreciation of an asset for a specific time period by using the double-declining balance method or some other method.

Syntax : `DDB(cost;salvage;life;period;[factor])`.

Keynotes : Refer to DDB function Microsoft©documentation for more details.

10.4.2 FV

Description : Returns the depreciation of an asset for a specific time period by using the double-declining balance method or some other method.

Syntax : `FV(rate;nper;pmt;[pv;[type]])`.

Keynotes : Refer to FV function Microsoft©documentation for more details.

10.4.3 IPMT

Description : Returns the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax : `IPMT(rate;per;nper;pv;[fv;[type]])`.

Keynotes : Refer to IPMT function Microsoft©documentation for more details.

10.4.4 IRR

Description : Returns the internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax : `IRR(values;[negativeSearch])`.

Keynotes : The `values` argument must be an array. The optional `negativeSearch` boolean argument defines when the `IRR` function will search in negative range if not positive solution is found. i. e.,`FORMAT(IRR({{-70000;12000;15000}};true);'Percent')` returns `'-44.35%'`.

@ When `negativeSearch` is set to `False` and there is no positive `IRR` solution, an error is returned.

Refer to `matrix` arguments in page 35, for syntax details, and also to IPMT function Microsoft©documentation for more details.

10.4.5 MIRR

Description : Returns the modified internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax : `MIRR(values;finance_rate;reinvest_rate)`.

Keynotes : Usage example: `FORMAT(MIRR({{-70000;22000;25000;28000;31000}};0.10;0.12);'Percent')`, will return `'15.51%'`. See also `IRR` function. Refer to MIRR function Microsoft© documentation for more details.

10.4.6 NPER

Description : Returns the number of periods for an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax : `NPER(rate;pmt;pv;[fv;[type]])`.

Keynotes : Refer to NPER function Microsoft© documentation for more details.

10.4.7 NPV

Description : Returns the number of periods for an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax : `NPV(rate;pmt;pv;[fv;[type]])`.

Keynotes : Refer to NPV function Microsoft© documentation for more details.

10.4.8 PMT

Description : Returns the payment for an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax : `PMT(rate;nper;pv;[fv;[type]])`.

Keynotes : Refer to PMT function Microsoft© documentation for more details.

10.4.9 PPMT

Description : Returns the principal payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax : `PPMT(rate;per;nper;pv;[fv;[type]])`.

Keynotes : Refer to PPMT function Microsoft© documentation for more details.

10.4.10 PV

Description : Returns the present value of an annuity based on periodic, fixed payments to be paid in the future and a fixed interest rate.

Syntax : `PV(rate;nper;pmt;[fv;[type]])`.

Keynotes : Refer to PV function Microsoft© documentation for more details.

10.4.11 RATE

Description : Returns the interest rate per period for an annuity.

Syntax : `RATE(nper;pmt;pv;[fv;[type;[guess]]])`.

Keynotes : Refer to RATE function Microsoft© documentation for more details.

10.4.12 SLN

Description : Returns the straight-line depreciation of an asset for a single period.

Syntax : `SLN(cost;salvage;life)`.

Keynotes : Refer to SLN function Microsoft© documentation for more details.

10.4.13 SYD

Description : Returns the straight-line depreciation of an asset for a single period.

Syntax : `SYD(cost;salvage;life;period)`.

Keynotes : Refer to SYD function Microsoft© documentation for more details.

10.5 Date, Time and String Functions

Some of the date-time functions return literal string results, see `Literal` tokens in section 2.2.1. The user should pay particular attention when reusing the latter in subsequent evaluations, in order to avoid functions receiving unexpected data types.

10.5.1 ASC

Description : Returns the character ASCII code corresponding to the first letter in a string.

Syntax : `ASC(string)`.

Keynotes : The `string` argument represents a `Literal` string, see section 2.2.1. Refer to ASC function Microsoft© documentation for more details.

10.5.2 CHR

Description : Returns a `Literal` string containing the character associated with the specified character code.

Syntax : `CHR(charcode)`.

Keynotes : Refer to CHR function Microsoft© documentation for more details.

10.5.3 DATE

Description : Returns a `Literal` string containing the current system date.

Syntax : `DATE()`.

Keynotes : Refer to DATE function Microsoft© documentation for more details.

10.5.4 DATEADD

Description : Returns a `Literal` string containing a date to which a specified time interval has been added.

Syntax : `DATEADD(interval;number;date)`.

Keynotes : The `date` argument is a `Literal` string, see section 2.2.1. Refer to DATEADD function Microsoft© documentation for more details.

10.5.5 DATEDIFF

Description : Returns the number of time intervals between two specified dates.

Syntax : `DATEDIFF(interval;date1;date2;[firstdayofweek;[firstweekofyear]])`.

Keynotes : The `date1` and `date2` arguments are `Literal` strings, see section 2.2.1. Refer to DATEDIFF function Microsoft© documentation for more details.

10.5.6 DATEPART

Description : Returns the specified part of a given date.

Syntax : `DATEPART(interval;date;[firstdayofweek;[firstweekofyear]])`.

Keynotes : The `date` argument is a `Literal` string, see section 2.2.1.

> **!** There is an issue with the use of this function. The last Monday in some calendar years can be returned as week 53 when it should be week 1. For more information and a workaround, see this Microsoft© article.

Refer to DATEPART function Microsoft© documentation for more details.

10.5.7 DATESERIAL

Description : Returns a `Literal` string containing the date for a specified year, month, and day.

Syntax : `DATESERIAL(year;month;day)`.

Keynotes : Refer to DATESERIAL function Microsoft© documentation for more details.

10.5.8 DATEVALUE

Description : Returns a `Literal` string containing a date.

Syntax : `DATEVALUE(date)`.

Keynotes : The `date` argument is a `Literal` string, see section 2.2.1. Refer to DATEVALUE function Microsoft© documentation for more details.

10.5.9 DAY

Description : Returns a whole number between 1 and 31, inclusive, representing the day of the month.

Syntax : `DAY(date)`.

Keynotes : The `date` argument can be a `Literal` string, see section 2.2.1. Refer to DAY function Microsoft©documentation for more details.

10.5.10 FORMAT

Description : Returns a `Literal` string containing an expression formatted according to instructions contained in a format expression.

Syntax : `FORMAT(Expression;[Format];[FirstDayOfWeek];[FirstWeekOfYear])`.

Keynotes : The `Expression` argument can be a `Literal` string, see section 2.2.1. Refer to FORMAT function Microsoft©documentation for more details.

10.5.11 HOUR

Description : Returns a whole number between 0 and 23, inclusive, representing the hour of the day.

Syntax : `HOUR(time)`.

Keynotes : The `time` argument can be a `Literal` string, see section 2.2.1. Refer to HOUR function Microsoft©documentation for more details.

10.5.12 INSTR

Description : Returns a number specifying the position of the first occurrence of one string within another.

Syntax : `INSTR([start];string1;string2;[compare])`.

Keynotes : If the `start` argument is omitted, search begins at the first character position. The start argument is required if compare is specified. `string1` and `string1` are `Literal` strings, see section 2.2.1. Refer to INSTR function Microsoft©documentation for more details.

10.5.13 LCASE

Description : Returns a `Literal` string that has been converted to lowercase.

Syntax : `LCASE(string)`.

Keynotes : The `string` argument is a `Literal` string, see section 2.2.1. Refer to LCASE function Microsoft©documentation for more details.

10.5.14 LEFT

Description : Returns a `Literal` string containing a specified number of characters from the left side of a string.

Syntax : `LEFT(string;length)`.

Keynotes : The `string` argument is a `Literal` string, see section 2.2.1. Refer to LEFT function Microsoft© documentation for more details.

10.5.15 LEN

Description : Returns the number of characters in a string.

Syntax : `LEN(string)`.

Keynotes : The `string` argument is a `Literal` string, see section 2.2.1. Refer to LEN function Microsoft© documentation for more details.

10.5.16 MID

Description : Returns a `Literal` string containing a specified number of characters from a string.

Syntax : `MID(string;start;[length])`.

Keynotes : The `string` argument is a `Literal` string, see section 2.2.1. Refer to MID function Microsoft© documentation for more details.

10.5.17 MINUTE

Description : Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.

Syntax : `MINUTE(time)`.

Keynotes : The `time` argument can be a `Literal` string, see section 2.2.1. Refer to MINUTE function Microsoft© documentation for more details.

10.5.18 MONTH

Description : Returns a whole number between 1 and 12, inclusive, representing the month of the year.

Syntax : `MONTH(date)`.

Keynotes : The `date` argument can be a `Literal` string, see section 2.2.1. Refer to MONTH function Microsoft©documentation for more details.

10.5.19 MONTHNAME

Description : Returns a `Literal` string indicating the specified month.

Syntax : `MONTHNAME(month;[abbreviate])`.

Keynotes : Refer to MONTHNAME function Microsoft©documentation for more details.

10.5.20 NOW

Description : Returns a `Literal` string indicating the current date and time according to your computer's system date and time.

Syntax : `NOW()`.

Keynotes : Refer to NOW function Microsoft©documentation for more details.

10.5.21 REPLACE

Description : Returns a `Literal` string which is a substring of a string expression beginning at the start position (defaults to 1), in which a specified substring has been replaced with another substring a specified number of times.

Syntax : `REPLACE(expression;find;replace;[start;[count;[compare]]])`.

Keynotes : The `expression`, `find` and `replace` arguments are `Literal` strings, see section 2.2.1. Refer to REPLACE function Microsoft©documentation for more details.

10.5.22 RIGHT

Description : Returns a `Literal` string containing a specified number of characters from the right side of a string.

Syntax : `RIGHT(string;length)`.

Keynotes : The `string` argument is a `Literal` string, see section 2.2.1. Refer to RIGHT function Microsoft©documentation for more details.

10.5.23 TIMESERIAL

Description : Returns a `Literal` string containing the time for a specific hour, minute, and second.

Syntax : `TIMESERIAL(hour;minute;second)`.

Keynotes : Refer to TIMESERIAL function Microsoft©documentation for more details.

10.5.24 TIMEVALUE

Description : Returns a `Literal` string containing the time.

Syntax : `TIMEVALUE(time)`.

Keynotes : The `time` argument can be a `Literal` string, see section 2.2.1. Refer to TIMEVALUE function Microsoft©documentation for more details.

10.5.25 TRIM

Description : Returns a `Literal` string containing a copy of a specified string without leading and trailing spaces.

Syntax : `TRIM(string)`.

Keynotes : The `string` argument is a `Literal` string, see section 2.2.1. Refer to TRIM function Microsoft©documentation for more details.

10.5.26 UCASE

Description : Returns a `Literal` string containing the specified string, converted to uppercase.

Syntax : `UCASE(string)`.

Keynotes : The `string` argument is a `Literal` string, see section 2.2.1. Refer to UCASE function Microsoft©documentation for more details.

10.5.27 WEEKDAY

Description : Returns a whole number representing the day of the week.

Syntax : `WEEKDAY(date;[firstdayofweek])`.

Keynotes : The `date` argument can be a `Literal` string, see section 2.2.1. Refer to WEEKDAY function Microsoft© documentation for more details.

10.5.28 WEEKDAYNAME

Description : Returns a `Literal` string indicating the specified day of the week.

Syntax : `WEEKDAYNAME(weekday;abbreviate;firstdayofweek)`.

Keynotes : Refer to WEEKDAYNAME function Microsoft© documentation for more details.

10.5.29 YEAR

Description : Returns a whole number representing the year.

Syntax : `YEAR(date)`.

Keynotes : The `date` argument can be a `Literal` string, see section 2.2.1. Refer to YEAR function Microsoft© documentation for more details.

10.6 Programming Functions

10.6.1 ARRAY

Description : Returns the array created from the given list of lists.

Syntax : `ARRAY(list)`.

Keynotes : The `list` argument must satisfy the syntax `{...};{...};...{...}`. i. e.,`ARRAY(a;b;c)` will return `{{6;15;55};{15;55;225};{55;225;979}}` for the variables `a={6;15;55};b={15;55;225};c={55;225;979}`.

10.6.2 CHOOSE

Description : Selects and returns a value from a list of arguments.

Syntax : `CHOOSE(index;choice-1;[choice-2;...;[choice-n]])`.

Keynotes : Refer to CHOOSE function Microsoft© documentation for more details.

10.6.3 GET

Description : Assigns/returns the value to/from a variable.

Syntax : `GET(VarName;[VarValue])`.

Keynotes : This function is used for managing expressions variables. The `VarName` argument is a `Literal` string, refer to section 2.2.1 in page 8. If the `VarValue` parameter is omitted, the previous stored variable value is returned. For usage examples see listings 3, 13, 16 and 17 in pages 26, 34 and 36.

10.6.4 IFF

Description : Returns one of two parts, depending on the evaluation of an expression.

Syntax : `IFF(expr;truepart;falsepart)`.

Keynotes : The `expr` can be any valid expression. This function behave like the VBA® IIF function.

10.6.5 SWITCH

Description : Evaluates a list of expressions and returns a Variant value or an expression associated with the first expression in the list that is True.

Syntax : `SWITCH(expr-1;value-1;[expr-2;value-2...;[expr-n;value-n]])`.

Keynotes : The function will return an error if no expression can be evaluated to `True`. Also see the VBA® SWITCH function.

11 Testing

In order to offer quality solutions to all users, Test Driven Development (TDD) has been adopted. Under this scheme, more than 50 unit tests have been coded to ensure that, between releases, proper functioning of *VBA Expressions* is maintained. For these purposes, Rubberduck[11] has been chosen to run the tests.

11.1 Rubberduck

> *"The Visual Basic Editor (VBE) has stood still for over 20 years, and there is no chance a first-party update to the legacy IDE ever brings it up to speed with modern-day tooling. Rubberduck aims to bring the VBE into this century by doing exactly that." Mathieu Guindon*

Rubberduck makes it easy to code, run and manage unit tests in VBA® projects. For this reason it has been chosen as the auxiliary development platform. All releases comes with a `TestRunner.bas` module used for unit testing.

Rubberduck must be installed to run unit tests. If you try to run the unit tests without this requirement, you will get a runtime error.

However, it is a reality that the *VBA Expressions* LibreOffice release does not have this tool. LO Basic needed a simple implementation that allows you to run unit tests in your development environment. This will be discussed later in this section.

In previous versions of *VBA Expressions*, up to v1.0.6 for LibreOffice, the unit testing module was named `TestVBAExpr`.

11.2 Running Tests

To run the unit tests in VBA®, go to the Rubberduck tab and select Unit Test → Test Explorer . From there click the Refresh button and then select Run → All Tests . Rubberduck will display those tests that passed and those that failed. You can also create your own tests following the guidelines described in the official blog[12].

In LibreOffice, unit tests use the `ScriptForge` library to show users the results of unit tests. To run them, the `RunAllTests` procedure of the `TestRunner` module must be executed.

12 Limitations

Currently, performance is one of the main limitations of *VBA Expressions*. The library is not designed to be the fastest among its alternatives, it is focused on offering functionality not provided by any other utility and with an elegant syntax that resembles that used by the BASIC language.

[11] https://rubberduckvba.com/
[12] https://rubberduckvba.blog/2017/10/19/how-to-unit-test-vba-code/

Although a great effort has been made to keep the code free of bugs, it is likely that users will encounter unresolved problems while using *VBA Expressions*. This is why we encourage users to report any detected issues.

We hope that the compatibility of LO BASIC with VBA®will increase in the future, which could solve the problems that prevent *VBA Expressions* from behaving exactly the same in LibreOffice and Microsoft©Office®. So far this is a constraint with which we must wait patiently.

13 Conclusions

VBA Expressions is a support tool for students and teachers of science, accounting, statistics and engineering; this due to the its full set of computations capabilities. The library can solve systems of equations and non-linear equations in one variable, resolve over-determinate equations systems; perform matrices operations and much more. The library also have a version for LibreOffice users, thus reaching a diversity of operating systems and breaking down payment barriers in office suites. Similarly, the support for numbers in various formats makes their use a fact in international contexts. *VBA Expressions* is well tested, providing its users with a considerable set of executable unit tests, making releases somewhat homogeneous; although there are still many untested procedures that could lead to unidentified bugs. In this sense, use, use, use and more use is the solution!

14 Credits

The development of *VBA Expressions* would not have been possible without the knowledge provided by the people and organizations listed below:

- ©William H. Press. Book: *Numerical recipes in C: the art of scientific computing.*
- ©Douglas C. Montgomery and George C. Runger. Book: *Applied Statistics and Probability for Engineers.*
- ©David M. Lane
- ©Dr. Hossein Arsham
- ©John C. Pezzullo
- ©iCalculator^{TM}
- ©Microsoft
- ©mozilla.org

15 License

This work is free: you can redistribute it and/or modify it under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike International License as published by the Creative Commons Foundation, either version 4.0 of the License.

This work is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

16 Review History

- April-2024: initial release.
- August-2024: added documentation for the `BETA.DIST` function. Fixed some typos.
- April-2025: added documentation for the `DISTANCE`, `LINESINTERSECT`, `PARALLEL`, `PERPENDICULAR` and `INSTR` functions.

17 Appendix

This section shows the code to evaluate a piece-wise function composed of a series of sub-functions defined within their domain ranges.

$$f(x) = \begin{cases} x^2 & ,x \leq 0 \\ log(x+1) & ,0 < x \leq 1 \\ \sqrt{x - log(2)} & ,x \geq 2 \end{cases}$$

In these functions it should be noted that the logarithm of zero does not exist, nor do square roots of negative numbers. At the time of evaluation, inevitably, an internal error will occur in *VBA Expressions*. However, the `EnforceBoolean` property allows `False` to be returned when errors occur on evaluate certain components of expressions.

```
Public Function evalPieceWiseFunctions(ByRef tRange As Range,
     xmin As Double, xmax As Double, samples As Long,
    ParamArray condValuePairs() As Variant) As Boolean
    Dim expr As VBAexpressions
    Dim j As Long, UB As Long
    Dim tmpFunct As String
    Dim step As Double
    Dim tmpResult() As String
    Dim outRange As Range

    On Error GoTo err_Handler
    ReDim tmpResult(0 To samples - 1, 0 To 1)
    step = (xmax - xmin) / (samples - 1)
    UB = UBound(condValuePairs)
    For j = LBound(condValuePairs) To UB Step 2
        tmpFunct = tmpFunct & condValuePairs(j) & ";" & _
        condValuePairs(j + 1)
        If j + 2 < UB Then tmpFunct = tmpFunct & ";"
    Next j
    Set expr = New VBAexpressions
    With expr
        .EnforceBoolean = True: .Create "SWITCH(" & tmpFunct
    & ")"
        For j = 0 To samples - 1
            .Eval "x = " & CStr(xmin + j * step)
            tmpResult(j, 0) = .CurrentVarValues
            tmpResult(j, 1) = "f(x) = " & .Result
        Next j
    End With
    With tRange
        .Range(Cells(tRange.Row, tRange.Column), _
        Cells(tRange.Row + samples - 1, tRange.Column + 1)).
    value2 = tmpResult
    End With
    evalPieceWiseFunctions = True
    err_Handler:
    Set expr = Nothing
End Function
```

Listing 22: Evaluating piece-wise fucntions

The above function will evaluate piece-wise functions given by the users, in our example the line `evalPieceWiseFunctions Sheets(1).range("A1"),-2,2,20,"x<=0", "x^2", "0<x & x<=1", "Log(x+1)", "x>1", "Sqr(x-Log(2))"` carry out the assessment under conditions set out in the previous statement. It can be noted that complex solutions to a variety of problems can be offered by abstracting the end-users from complexity.

Index

accessing variables, 33
arguments, 8
arguments, separator, 20

basic functions, abs, 37
basic functions, acos, 38
basic functions, asin, 38
basic functions, atn, 38
basic functions, avg, 38
basic functions, ceil, 39
basic functions, cos, 39
basic functions, exp, 39
basic functions, floor, 39
basic functions, lgn, 39
basic functions, ln, 40
basic functions, log, 40
basic functions, max, 40
basic functions, min, 40
basic functions, percent, 40
basic functions, pow, 41
basic functions, round, 41
basic functions, sgn, 41
basic functions, sin, 41
basic functions, sqr, sqrt, 41
basic functions, sum, 42
basic functions, tan, 42
built-in functions, 37

data management, 28
date-time-string functions, array, 63
date-time-string functions, asc, 56
date-time-string functions, choose, 63
date-time-string functions, chr, 57
date-time-string functions, date, 57
date-time-string functions, dateadd, 57
date-time-string functions, datediff, 57
date-time-string functions, datepart, 57
date-time-string functions, dateserial, 58
date-time-string functions, datevalue, 58
date-time-string functions, day, 58
date-time-string functions, format, 58
date-time-string functions, get, 63
date-time-string functions, hour, 59
date-time-string functions, iff, 63
date-time-string functions, lcase, 59
date-time-string functions, left, 59
date-time-string functions, len, 59
date-time-string functions, mid, 60
date-time-string functions, minute, 60
date-time-string functions, month, 60
date-time-string functions, monthname, 60
date-time-string functions, now, 60
date-time-string functions, replace, 61
date-time-string functions, right, 61
date-time-string functions, switch, 64
date-time-string functions, timeserial, 61
date-time-string functions, timevalue, 61
date-time-string functions, trim, 62
date-time-string functions, ucase, 62
date-time-string functions, weekday, 62
date-time-string functions, weekdayname, 62
date-time-string functions, year, 62
demo code structure, 13

engineering, physics, 31
enumeration, decimalsymbol, 23
enumeration, expressionerrors, 23
enumeration, operatortoken, 23
essentials, 6
evaluating expressions, 13
evaluating loops, 14
evaluation tree, 10
expression result, 19
expressions, 8
expressions grammar, 8
expressions syntax, 7

functions syntax, 9

gallop, 19

human evaluation, 12

implicit variables, 19
install, 5

library members, 16
library structure, 16
libray members, 5
libreoffice install, 6
libreoffice, load library, 27
libreoffice, managing scopes, 32
libreoffice, notes, 26
libreoffice, recursion, 27
libreoffice, variables, 26
limitations, 65
linear algebra, 28
linear equations systems, 29
lists, 10

main use cases, 27
math functions, cholesky, 42
math functions, det, 43
math functions, fzero, 43
math functions, gamma, 44
math functions, gammaln, 44
math functions, inverse, 44
math functions, inverse by cholesky, 42
math functions, lsqrsolve, 44

math functions, ludecomp, 45
math functions, lusolve, 45
math functions, mmult, 45
math functions, mneg, 46
math functions, mround, 46
math functions, msum, 46
math functions, mtranspose, 46
math functions, qr, 47
math functions, rem, 47
math functions, solve by cholesky, 43
math functions, solve by solve, 47
matrices syntax, 9
matrices, arrays, 34
matrices, overloading, 35
method, addconstant, 21, 25
method, arrayfromstring, 21
method, arrayfromstring2, 21
method, arraytostring, 22
method, constantsinit, 25
method, copyscope, 25
method, create, 22
method, declareudf, 22
method, eval, 23
method, fillpredefinedvars, 25
method, isconstant, 23, 26
method, todblarray, 23
method, variablesinit, 26
multi-variables expressions, 14

operands, 8
operators, 6
overdeterminated systems, 29

parsing status, 19
precedence, 7
property, assignedarray, 24
property, constants, 16
property, currentvariables, 16
property, currentvarvalues, 16
property, decimalsymbol, 16
property, definedScope, 24
property, degrees, 17
property, enforceboolean, 18
property, errordesc, 17
property, errortype, 17
property, evalscope, 17
property, expression, 17
property, formatresult, 18
property, gallopingmode, 19
property, implicitvarvalue, 19
property, readytoeval, 19
property, result, 19
property, separatorchar, 20
property, variablescount, 25
property, varvalue, 20
property, varvalue2, 20

quick start, 12

regional format, 16
regressions, 30
rubberduck, 64

special variables, 9
stat functions, achisq, 48
stat functions, aerf, 48
stat functions, afishf, 48
stat functions, agauss, 48
stat functions, anorm, 49
stat functions, astudt, 49
stat functions, betainv, 49
stat functions, chisq, 49
stat functions, ddb, 53
stat functions, erf, 50
stat functions, fishf, 50
stat functions, fit, 50
stat functions, fv, 54
stat functions, gauss, 51
stat functions, ibeta, 51
stat functions, ipmt, 54
stat functions, irr, 54
stat functions, mirr, 54
stat functions, mlr, 51
stat functions, norm, 52
stat functions, nper, 55
stat functions, npv, 55
stat functions, pmt, 55
stat functions, ppmt, 55
stat functions, pv, 55
stat functions, rate, 56
stat functions, sln, 56
stat functions, studt, 52
stat functions, syd, 56
stat functions, tinv, 53
stat functions, tinv_1t, 53
stat functions, tinv_2t, 53
statistics, 30
strings, 15
sub-expressions, 8
symbols, 7

testing, 64
token tree, 10
tokens, 8
trend line, 31

udf management, 36
unit testing, 64

variable scope, 32
variables definition, 9
variables, special uses, 34
vbaexpressions enumerations, 23
vbaexpressions methods, 21

vbaexpressions properties, 16
vbaexpressions.cls, 16
vbaexpressionsScope methods, 25
vbaexpressionsScope properties, 24
vbaexpressionsScope.cls, 24

zeroing functions, 31