



OCX Controls & software technologies
<http://www.precisionocx.com>

Precision Expression Evaluator

Overview

Purpose

Precision Expression Evaluator is a powerful OCX control which enables your applications to treat numeric or logical mathematical expressions. Ask to the user an expression which states the relationship between your variables, or declares logical properties and constraints: with this control you will be able to calculate the expression's result according to the user's formula or decide whether a user's constraint is verified. Give to the user full freedom: ask him a symbolic expression instead of letting him choose between predefined calculation algorithms. Of course you can check the expression's correctness and warn the user with appropriate messages.

Optimization

Precision Expression Evaluator was written entirely in C++ and is composed of strongly optimized code. The only required libraries are the MFC and MSVCRT DLL that are shipped with every version of Microsoft Windows. You do not need any additional libraries.

Small size

Expression Evaluator is only 60 k in size ! This means that it is very fastly loaded into your application and it does not significantly increase your final application size.

Any special requirement ?

If you are developing an application and found Precision Expression Evaluator nearly perfect for you needs except for small details, please feel free to contact us: let's talk about it !

Documentation

All the documentation you may need on Precision Expression Evaluator is on line !

Quick start

To start right now using the control:

- 1 If you have purchased a license number and you are using the full version, you must call the *SetLicense* method passing your license number as argument. If you are simply trying this control in evaluation version, ignore this step.
- 1 Set the *Expression* property to the desired expression string; if you have not set the license number and you are trying the evaluation version of this control, this step is still required but the expression will remain this predefined string:

`Exp(Sqrt(1-Sin(3.1415))) > Exp(Sqrt(1-Cos(3.1415)))`

- ┆ Set the variables you intend to use invoking the *SetVariable* method.
- ┆ Call the *Correctness* method and make sure it returns zero:
 - ┆ If the result is non-zero, report the error.
 - ┆ If the result is zero (everything is ok), call *Evaluate* and obtain the result!
 - ┆ Now you can invoke *ShowExprTree* to show an interactive expression tree viewer.

Properties and methods

Expression (property)

This property contains the current expression. It is a string which can contain a well-formed expression, made of literal constants (like *12.345*), variable names (like *Pressure*), operators (like *+*) and functions (like *sin*). You can freely combine these elements in order to obtain arbitrarily complex expressions. Expressions are case-insensitive: both variables and function names are bound disregarding the case in which they are written.

Correctness (method)

This method perform a correctness check on the expression and let you identify possible problems with your expression. You can call this method after setting the *Expression* property; the result is zero if the expression is correct, otherwise it is a bit mask in which every bit indicates the presence of an error, with the following meaning:

bit 0	Uninitialised expression	This error arises when you invoke the <i>Evaluate</i> or <i>Correctness</i> method without having set an expression to evaluate before, or when the expression string is empty, or composed by whitespaces only.
bit 1	Wrong number of arguments	This error arises when you call a function with a bad number of arguments, for example when you try to evaluate the expression: <i>sin(45,2)</i> , when the <i>sin()</i> is known to accept one only argument.
bit 2	Wrong use of reserved word	This error occur when you try to use a name of a function as if it were a variable or a constant or something else, in practice this is true whenever the parentheses which indicate function call are not used. Probably you forgot to put parentheses around arguments. for example the following expressions generate this sort of error: <i>1 + sin</i> <i>(COS + TAN)</i> <i>sin 45</i>
bit 3	An argument was expected	This error usually occurs when an expression is truncated or when you forget an operand or an argument; this error is also generate when you forget a parenthesis at the end of a correct expression, rendering it an unusable subexpression. For example the following expressions generate this sort of error: <i>Sin(</i> <i>(1+5)/(4+</i> <i>(1+5)/(4+5</i>
bit 4	Undefined function	This error is reported when you use an identifier name (a name that can be used to name a variable) immediately before an open parenthesis: it seems to be a function name but it is not (probably because you misspelled the name, or perhaps because you intended to multiply a variable by something that is inside the parenthesis). <i>x(5+4)+y</i> you intended: <i>x*(5+4)+y</i> <i>sine(3.14)</i> you intended: <i>sin(3.14)</i>
bit 5	Undefined variable	This error is reported when you use an identifier name (a name that can be used to name a variable) but there is not a corresponding

|||variable definition, probably because you misspelled the name. |||

DeleteVariable(in string variable_name) (method)

This method tries to remove the variable with the given name and the associated value from the internal list of variables. If a variable with the given name cannot be found, nothing is done. **Attention:** since variable matching is case-insensitive, if you define two variables called *foo* and *FOO*, only one variable will be created, and a call like *DeleteVariable("Foo")* or like *DeleteVariable("fOo")* will destroy it.

double Evaluate (method)

This method tries to evaluate the given expression obtaining a double precision real number: if the expression is correct and all the mentioned variables could be found (use the *Correctness* method first to make sure), then the result of the evaluation of the expression is returned, otherwise the returned value is meaningless and should be ignored.

double GetVariable(in string variable_name) (method)

This method returns the double-precision value which has been assigned to a variable. If the variable is not defined, the result is meaningless and should be ignored. Please use the *IsDefinedVariable* first in order to know whether the variable is defined.

bool IsDefinedVariable(in string variable_name) (method)

This method returns True (1) if a value has been assigned to the variable which name has been passed as argument, it returns False (0) if a variable with the given name has never been defined. Variables can be defined by using the *SetVariable* method or by the user, when the Variables dialog box is displayed. To display it, simply invoke the *ShowVariables* method.

ResetAll (method)

This method clears all the internal buffers, reset the object to its initial state (when no expression has been set yet) and clear all the variables.

bool SetLicense(long LicenseNumber) (method)

This method is used to activate the full-version control. You should call this method before calling any other method, passing the license number you obtained at purchase time as *LicenseNumber* parameter, otherwise the control will not draw itself on the screen. Make sure it returns *True*, this means your license has been correctly verified. **Attention:** you do not need to use this method using the trial version of the control.

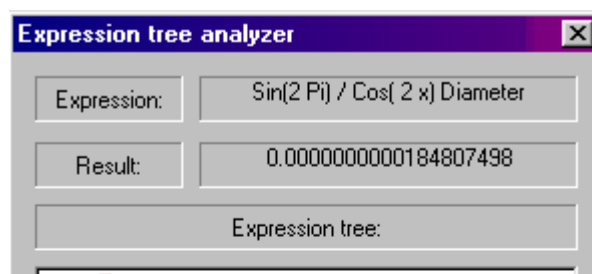
SetVariable(in string variable_name, in double variable_value) (method)

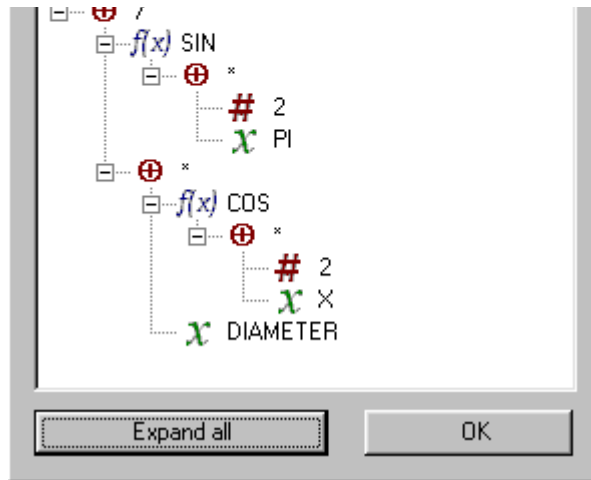
This method sets the variable with the given name to the specified value. If the variable was not present, it will be created; if it was already defined, it will be overwritten and the old value will be lost. Since the variable matching is not case-sensitive, variables are stored in uppercase format.

ShowExprTree() (method)

This method cause the control to open a dialog box in which the structure of the expression is depicted in a tree graph view.

The result should be as in the image here reported. The purpose of this view is to give a graphical feedback to the user, in order to let him verify that the given expression is parsed and interpreted exactly as he expected.

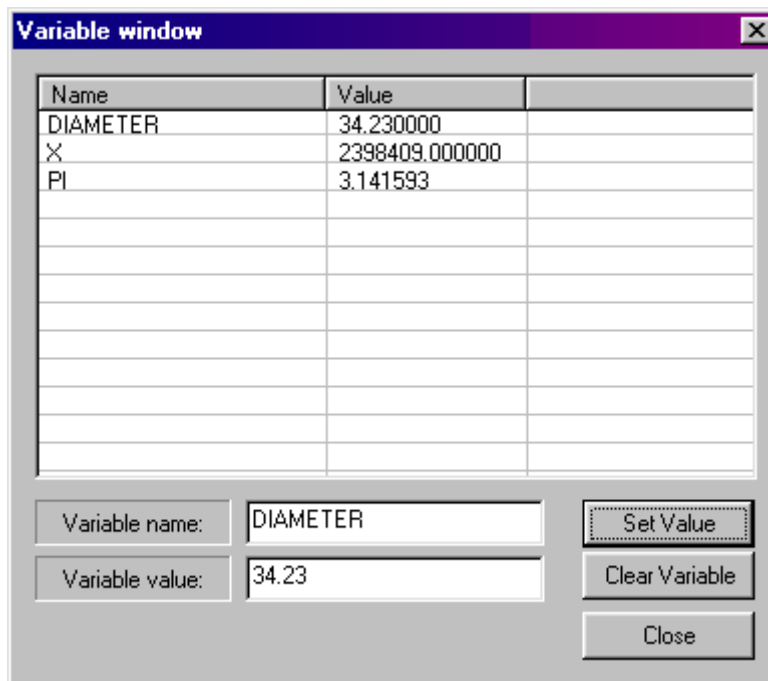




ShowVariables() (method)

This method cause the control to open a dialog box in which the list of all the defined variables is shown.

The result should be as in the image here reported. The purpose of this view is to let the user know at a glance which are the defined variables, what is their value, and let him modify, delete and create new variables.



Variables

Variables are values to which a name has been assigned, in order to use this value in an expression without knowing it (before) or in order to write a reusable calculation formula which can be applied to a number of different values assigned to the variables that appear in it. Variables must have a name, that can be used in an expression tu access its value. Variable names are a succession of one or more of the following characters:

- | '_' (underscore)
- | 'A' ... 'Z' (the uppercase international alphabet letters)
- | 'a' ... 'z' (the lower international alphabet letters)

- | '0' ... '9' (the numbers)
- | any character whose ASCII code is greater than 127 (any international symbol)

with a **constraint**: a variable name cannot have as first character a number. Examples of allowed variable names are: *x*, *qty0*, *_Café_2000*, *_*, *_1* ; examples of invalid variable names are: *4x*, ***y***, *dollars?* . The variable matching mechanism adopted by Precision Expression Evaluator is case-insensitive: this means that *Quantity*, *quantity*, *QUANTITY*, *QuantitY* and so on, denote all the same variable. Variable names are internally stored as uppercase strings and when they appear in an expression are converted to the uppercase form before being looked up in the internal variable table.

Constants

You can specify constant numbers exactly like in your programming language, using the "e" or the "E" to separate mantissa from exponent and the point "." as decimal separator. Do not use the comma "," as decimal separator as it's used as parameter separator; as well do not use the space " " as decimal separator as it could represent implied multiplication. Examples:

Correct:	Incorrect:
1	1,000,000
1.0	1,5
1e5	1e+0.5
1.02984E+4	1 000 000
1e+5	.05

Operators

This is the table containing all the operators, in order of increasing evaluation precedence:

&, AND, , OR
NOT
>=, <=, >, =, <, <>, >>, !=
+, -
-
*, /
^
!

Operator meaning:

&, AND

The logical AND operators ('&' and 'AND', they are fully equivalents) are used to combine multiple conditions formed using relational or equality expressions. The logical-AND operator returns the integral value 1 if both operands are nonzero; otherwise, it returns 0.

|, OR

The logical OR operators ('|' and 'OR', they are fully equivalents) are used to combine multiple conditions formed using relational or equality expressions. The logical-OR operator returns the integral value 1 if both operands are nonzero; otherwise, it returns 0.

NOT

The logical-negation (logical-NOT) operator produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0).

>, =, <=, >=, <

The binary relational and equality operators compare their first operand to their second operand to test the validity of the specified relationship. The result of a relational expression is 1 if the tested relationship is true and 0 if it is false.

<>, ><, !=

The binary inequality operators compare their first operand to their second operand to test whether they are different. The result is 1 if they are not equal, 0 if they are equal.

+, -

The binary addition operator (+) causes its two operands to be added, the binary subtraction operator (-) subtracts the second operand from the first.

-

The unary minus operator causes the following operand to undergo a change in sign.

*, /

The binary multiplication operator (*) causes its two operands to be multiplied, the binary division operator (/) causes the first operand to be divided by the second.

^

The binary power operator calculates the first operand raised to the power of the second operand.

!

The unary factorial operator calculates factorial of the operand.

Spaces can be freely introduced to visually separate expression elements. Parentheses ('(' and ')') can be used to override operator precedence as wanted.

Implied multiplications

A smart and extended piece of code is dedicated to the detection of implied multiplications. An implied multiplication sign '*' is introduced whenever two subexpressions are put one after another without any sign between them.

Examples:

Expression	internally converted to	result
2 3	2 * 3	6
2 cos(Pi)	2 * cos(Pi)	-2
2 2 2 2 2 2	2 * 2 * 2 * 2 * 2 * 2	64

Functions

ABS(x)

Returns the absolute value of x.
Example: ABS(-5) returns 5

ACOS(x)

Returns the arc-cosine of x.
The return value is intended to be an angle specified in radians.

ASIN(x)

Returns the arc-sine of x.
The return value is intended to be an angle specified in radians.

ATAN(x)

Returns the arc-tangent of x.
The return value is intended to be an angle specified in radians.

CEIL(x)

Returns a value representing the smallest integer that is greater than or equal to x.
Example: CEIL(4.3) returns 5.

COS(x)

Returns the cosine of x.
The x value is intended to be an angle expressed in radians.

COSH(x)	Returns the hyperbolic cosine of x. The x is intended to be an angle expressed in radians.
DIV(x,y)	Returns the quotient of the integer division between x and y. The x and y values must be integer values, otherwise unpredictable results will be generated.
MOD(x,y)	Returns the remainder of the integer division between x and y The x and y values must be integer values, otherwise unpredictable results will be generated.
EXP(x)	Returns the exponential of x, that is to say the neperian constant <i>e</i> raised to the x-th power.
FLOOR(x)	Returns a floating-point value representing the largest integer that is less than or equal to x. Example: FLOOR(6.7) returns 6.
HYPOT(x,y)	Returns the hypotenuse of a right triangle with the given cateti x and y. Example: HYPOT(3,4) returns 5.
LN(x)	Returns the natural logarithm of x. If x is negative, returns an indefinite. If x is 0, returns INF (infinite).
LOG(x)	Returns the base-10 logarithm of x. If x is negative, returns an indefinite. If x is 0, returns INF (infinite).
MAX(x,y)	Returns the maximum value between x and y.
MIN(x,y)	Returns the minimum value between x and y.
RAND(x)	Returns a pseudorandom integer in the range 0 to x.
SIN(x)	Returns the sine of x; x is intended to be an angle expressed in radians.
SINH(x)	Returns the hyperbolic sine of x; x is intended to be an angle expressed in radians.
SQRT(x)	Returns the square root of x; x is intended to be an angle expressed in radians.
TAN(x)	Returns the tangent of x; x is intended to be an angle expressed in radians.
TANH(x)	Returns the hyperbolic tangent of x; x is intended to be an angle expressed in radians.

Internal number representation

Precision Expression Evaluator makes always use of the *type double* to internally represent the numbers. Double precision values with double type have 8 bytes. The format is similar to the float format except that it has an 11-bit excess-1023 exponent and a 52-bit mantissa, plus the implied high-order 1 bit. This format gives a range of approximately 1.7E-308 to 1.7E+308 for type double. The double type contains 64 bits: 1 for sign, 11 for the exponent, and 52 for the mantissa. Its range is +/-1.7E308 with at least 15 digits of precision.

Mixing logical and numerical results

Attention: the evaluation engine makes no difference between the results coming from numerical operators or functions (like SIN, +, and so on) and results coming from logical expressions that contain logical and comparison operators ("A AND NOT (B OR (SIN(Alpha)>1)", which can be only 0 or 1. Therefore it is your responsibility to execute "type checking" and not to sum 1's representing "True" and 1's representing "1.00000".
