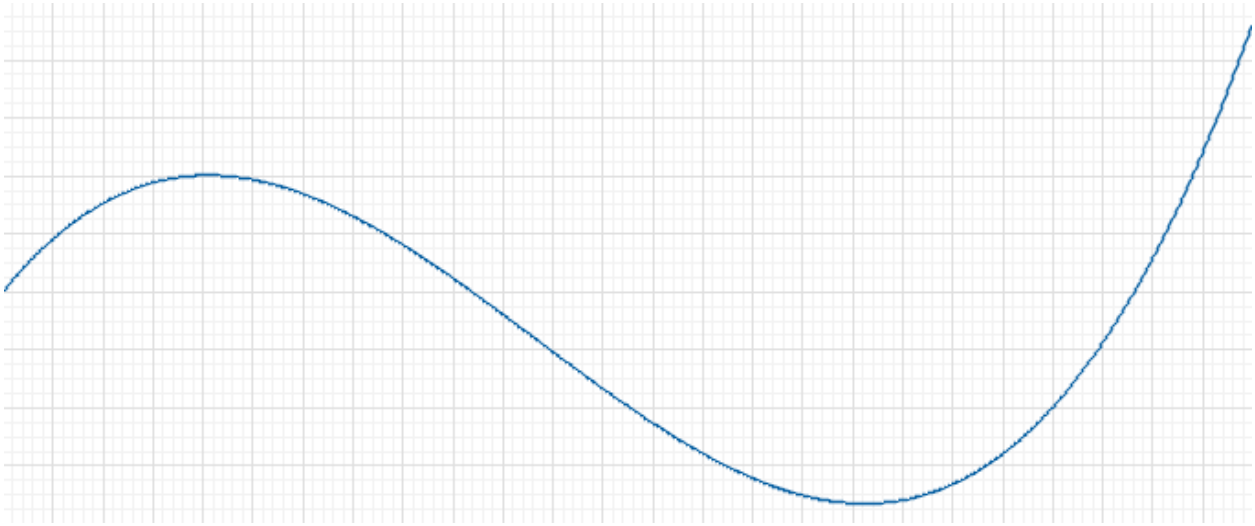
GPower

User guide for the
Expression Parser
LabVIEW toolset

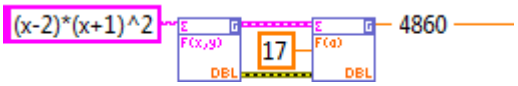
Rev. 2018

March 19th 2018



1 Introduction

The GPower Expression Parser toolset enables you to evaluate mathematical expressions, supplied as strings, into numeric values:



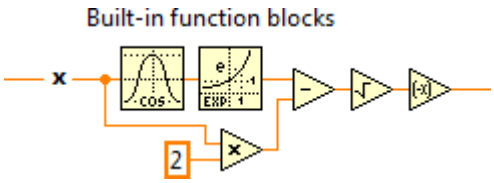
Being able to evaluate string expressions enables you to define and change mathematical expressions at runtime instead of being forced to define them statically at edit time.

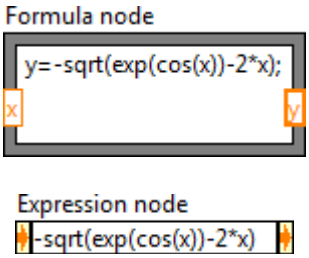
Evaluating mathematical expressions is an important part of any programming language, and of LabVIEW especially, as LabVIEW is used often in domains where math is fundamental, such as simulations, modelling, machine control, signal conditioning and much more. The GPower Expression Parser toolset sports a number of unique features:

- Build and change your math expressions at runtime.
- More than 260 math functions and constants supported.
- Very high performance.
- Supports any number of variables of any name.
- Supports VI Registers (variables which let your math depend on anything in LabVIEW).
- Reports overflow if that occurs during evaluation.
- Supports all 14 numeric data types that LabVIEW offers, including complex evaluation.
- Offers special expression control like conditionals, piecewise defined functions, pulse trains, and defining your own custom periodic functions.

2 Math evaluation methods

Many ways to do math are already built into LabVIEW, however common for them all (with a single exception) is that they are all statically defined, i.e. you define them at edit time and you can't modify them at runtime. Here are some examples of static ways to implement an expression:

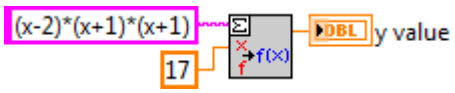




Each has its own advantages and drawbacks:

- **Built-in function blocks**
  Performs very well and are highly flexible. However, statically defined and hard to decipher the actual mathematical expression that is being evaluated without extra documentation on the block diagram.
- **Formula node**
  Performs very well and the math is easy to read. However, statically defined, limited function support, restrictions on variables, and in some cases awkward notation.
- **Expression node**
  Similar to the Formula node, but limited to a single variable.

There's only one way built into LabVIEW to do dynamic expression parsing at runtime, namely with the NI Formula Parsing VIs:



- **Formula Parsing VIs**
  Dynamic expression parsing. However, quite poor performance, the same limited function support, restrictions on variables, and awkward notation as Formula nodes.

2.1 Method comparison

	GPower	NI LabVIEW built-in			
	Expression Parser	Formula Parsing VIs	Formula node	Expression node	Function blocks[1]
Runtime definition of expression	✓	✓	-	-	-
Functions and operators supported	168	51	64	61	190[2]
Mathematical constants supported	97	1	1	1	19
Single precision support	✓	-	✓	✓	✓
Double precision support	✓	✓	✓	✓	✓
Extended precision support	✓	-	-	✓	✓
Complex numbers support	✓	-	-	-	✓
Integer evaluation mode	✓	-	✓	✓	✓
64-bit integer support	✓	-	-	-	✓
, and . decimal separator support	✓	-	-	-	✓
Bin, Oct, Dec, and Hex constants support	✓	-	✓	-	✓
Single variable supported	✓	✓[3]	✓	✓	✓
Multiple variables supported	✓	✓[3]	✓	-	✓
External variables supported[4]	✓	-	-	-	✓
Automatic overflow reporting	✓	-	-	-	-
Logical operators supported (&&, || etc.)	✓	-	✓	✓	✓
Conditionals supported (If-Then-Else)	✓	-	✓	✓	✓

[1] Discrete function blocks on the block diagram can of course be used to program almost anything, so support for a feature in this column means if that capability is available with one or at most a few built-in functions combined. If elaborate programming is needed, that functionality is listed as unsupported.
[2] This is an approximate number as LabVIEW has many additional built-in math functions that operate on compound or complex data types such as waveforms and matrices. This number is a count of singular, discrete math functions with a few simple inputs and a few simple outputs.
[3] Supports a limited range of variable names, namely a, a0, …, a9, … z, z0, …, z9.
[4] External variables serve values that depend on external asynchronous processes rather than constants being supplied at the start of expression evaluation. In an ordinary LabVIEW block diagram that could be local or global variables, values from queues, or Shared Variables for instance. With Expression Parser external variables are implemented with VI Registers.

Piecewise defined function support	✓	-	-	-	-
Custom periodic function support	✓	-	-	-	-
Verbose error messages[5]	✓	-	-	-	-
Performance[6]	10x – 40x	1x	2x – 2000x	2x – 1000x	20x – 1000x

Table 1 - Comparison of LabVIEW math evaluation methods

Table 1 tells us a number of important things:

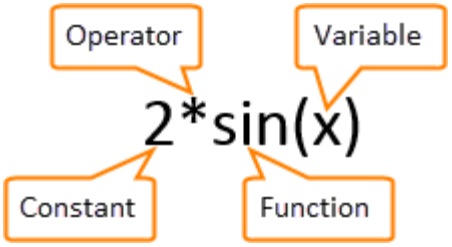
- Expression Parser is in all aspects a much better alternative to its built-in alternative, the Formula Parsing VIs.
- Expression Parser is very fast, typically millions of evaluations per second, and thus comparable in performance to LabVIEW function blocks.
- Expression Parser is much easier to set up, than the built-in options, thanks to its detailed error messages if you happen to do anything wrong.

[5] Formula Parsing VIs and the Formula and Expression nodes are very imprecise when it comes to error reporting. Typically there are no syntax location given in the error message, but just a statement like “Not a valid function”. Couple that with a sometimes awkward and limited syntax, the built-in methods can be difficult to debug. Expression Parser is much more precise in its error output, telling you exactly which part of the expression is at fault.

[6] Performance varies a lot depending on which math functions are being evaluated. What matters is that Expression Parser generally is 10-40 times faster than the comparable built-in solution (Formula Parsing VIs). All the statically defined methods can be a lot faster, but they can actually perform worse in some cases as well. Typically the more complex the math the better Expression Parser performs compared to the built-in solutions.

3 GPower Expression Parser

The GPower Expression Parser toolset is developed as a replacement of the Formula Parsing VIs, with much better performance, an easier to use API, and much higher flexibility. It will evaluate expression strings containing constants, variables, operators and functions. Syntax and constraints on each of these are described in detail in chapter 5 “Expression string syntax”.



This toolset evaluates mathematical *expressions*, it doesn’t evaluate *equations*, so there aren’t any assignment operator in these expression strings. Thus you can parse and evaluate an expression string like “2+x”, but not the equation “y=2+x”.

3.1 License

A separate Software License Agreement governs this software, its wording in full can be found in the enclosed ‘ExprParser_License.txt’ document.

3.2 Requirements

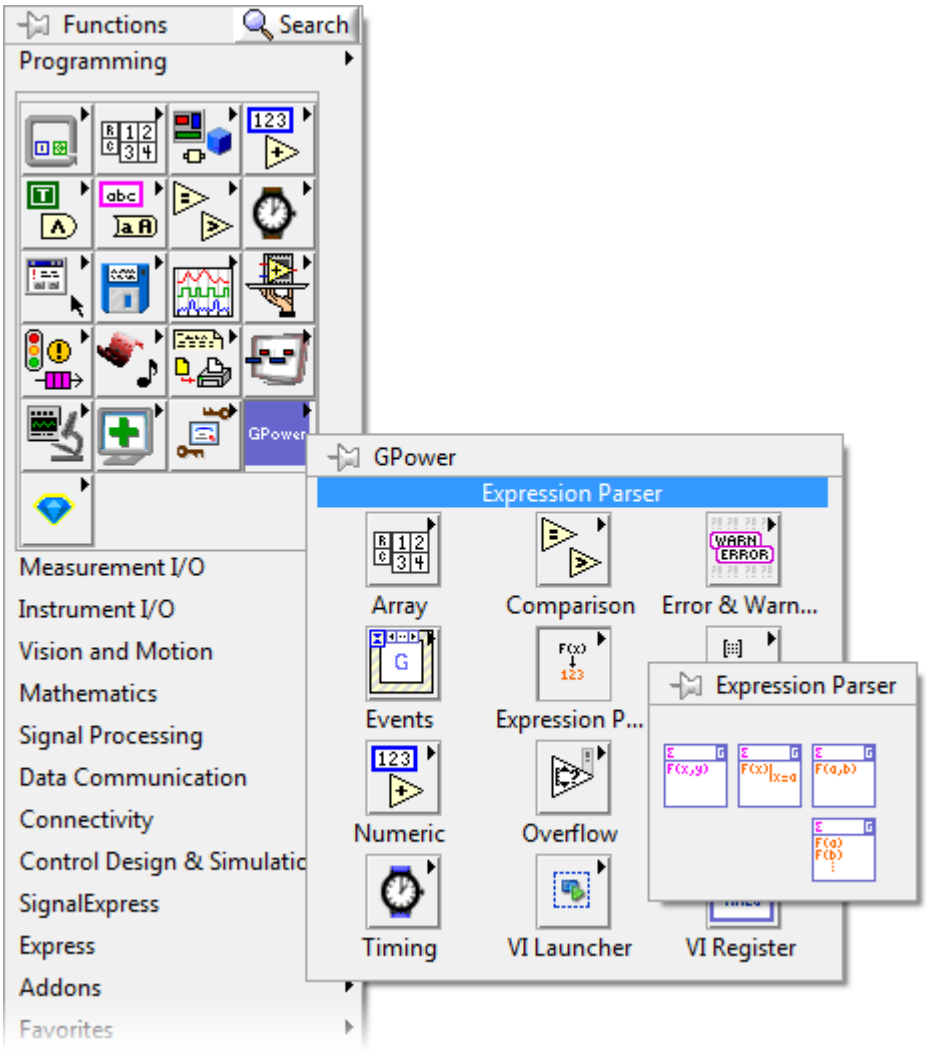
- Installation must be done with VI Package Manager (VIPM) 2017 or later.
- LabVIEW Full or Professional is required, version 2013 or later.
- The following GPower toolsets are dependencies of the Expression Parser toolset, and are thus required for it to work (a .vipc-file for offline installation is available with all dependencies included, and when installing Expression Parser directly from VIPM these dependencies will automatically be installed as well): Array, Comparison, Error & Warning, Events, Math, Numeric, Overflow, String, Timing, VI Launcher, and VI Register.

3.3 Installation & activation

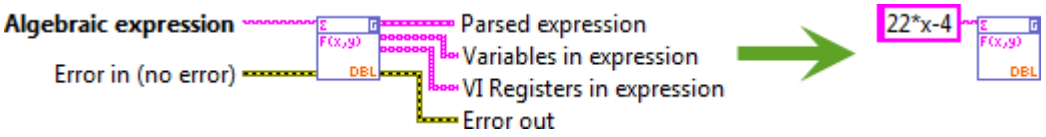
Installation and activation is described in the Expression Parser Getting Started Guide.

4 Palette functions

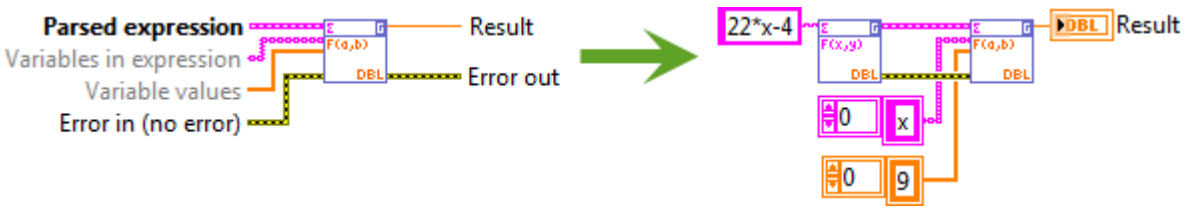
You find your installed Expression Parser toolset in the same palette as all your other GPower toolsets:



First step on your block diagram is always to parse your math expression, i.e. to convert it from string representation to a data form that the Evaluation VI can process:



When you have parsed your expression, you can evaluate it with numeric input in place of any variables in the expression:



4.1 Data types

Both the Parse VIs and the Evaluation VIs operate with a specific data type. This selected data type governs four parts of the parsing and evaluation process:

When parsing

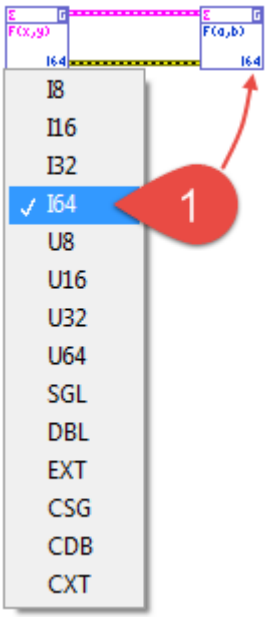
- The data type which constants in the expression string are parsed as.

When evaluating

- The data type you can input variable values as.
- The data type that VI Registers are read as.
- The data type and algorithm that each operator and function selects when evaluated.

All 14 numeric data types in LabVIEW are supported by this toolset, although not all functions are supported by all data types (you can find the specifics on this in chapter 10.1 “Operators & Functions”).

You need to select already at the parsing VI which data type you want to work with, and when you wire the parsed expression to the evaluation VI that will (by default) automatically change to the same data type as well:



5 Expression string syntax

In general, there are a number of characters that are *allowed* in expression strings in specific contexts, such as “+”, “6”, “(”, and “A”. You will be introduced to these in the following sections. There are also a few characters that are *never* allowed in expression strings, such as “¤” and “§”. These characters are in general just the ones that we don’t mention in the following sections.

The space character is generally not allowed in expression strings, the only exception being in VI Register names, which can have spaces in them.

5.1 Constants

Expression strings support three types of real constants:

- **Floating-point constants**
  Straight numbers that appear in the expression string, such as “7” and “8.002”. Scientific notation, such as “9.1E-6”, is supported, as are both dot and comma as decimal separator.

  The parser parses these from string to EXT, and then if necessary casts them to another data type (e.g. SGL or U8). Overflow that happens in the cast to a lower data type will yield a parse error. Non-integer floating-point values cast to integer at the parsing stage will be rounded to nearest integer without any error.

- **Integer constants**
  Unsigned integer constants of up to 64 bits can be embedded in the expression string starting with "0x" and ending with a lowercase base specifier of either "b" (binary), "d" (decimal), "h" (hexadecimal), or "o" (octal). Omitting the base specifier is also supported in which case the integer constant is assumed to be hexadecimal. Hex digits A-F must be uppercase.

  Examples: “0x12AAh”, “0x11101011b”, “0x45d”

  The parser parses these from string to U64, and then if necessary casts them to another data type (e.g. SGL or U8). As with floating-point constants overflow will yield a parse error.

- **Constant keywords**
  Certain reserved keywords will be parsed into a constant value, for instance CONSTe and CONSTpi. See “Table 2 - Function and operator support across data types” for a complete list of built-in constants. These constant keywords are case insensitive and may not be used as variable names.

Complex constants aren’t supported as compound numbers (like “4+5*i”, in which case the “i” will be treated as a variable). Instead you must enter a complex constant using the CONSTi keyword, e.g. “4+5*CONSTi”. The CONSTi keyword is only supported in complex mode.

5.2 Variables

Any string of characters…

- starting with a supported character* and optionally continuing with supported characters* or numbers,
- not a constant keyword,
- not the argument of a VIR() function,
- and not ending in a left-parenthesis (since then it's a function name),

…is considered a variable.

* Supported characters in variable names (besides numbers after the first character) are:

- Letters A_Z and a-z.
- The characters #, $, @, and _ (underscore).
- The ASCII characters with codes from 127 to 255.

Thus, in the expression "2*a-L10+5" both "a" and "L10" are considered variables. The Parse VI will return a list of variable names it found in the expression string. Numeric values must be input for each variable name at the Evaluation VI.

Variables are case sensitive.

Note that it's allowed to have the same name as both a variable and a VI Register name in the same expression string, but such two are *not the same identity* (more on VI Registers in chapter 5.5 "VI Register support"). Thus "abc*VIR(abc)" is allowed, but the two instances of "abc" are not the same – the first occurrence is a variable and the other is a VI Register name. The Parse VI will in this case return "abc" both in the variables list and in the VI Registers list.

5.3 Operators

A number of unary and binary operators are supported:

- Binary (infix) operators: +, -, *, /, ^, && (logical AND), || (logical OR), & (bitwise AND), | (bitwise OR), << (left shift), >> (right shift), != (not equal), ==, <, <=, >, >=.
- Unary (prefix) operators: - (negative sign), ~ (bitwise NOT).
- Unary (postfix) operators: ! (factorial), % (percentage).

A couple of rules apply to operators in this toolset:

- Implied multiplication isn't supported, so do "2*x" and not "2x" when multiplying 'x' by 2.
- Negative sign is supported on constants, variables, and functions, but unary positive sign is not supported. Thus "-x" is allowed, but "+x" on its own will yield a syntax error.

See chapter 10.1 "Operators & Functions" for more details on each operator.

5.4 Functions

This toolset currently supports around 170 functions in various categories such as arithmetic, powers and logarithms, trigonometry, rounding and parts, logical operations, integer operations, tests and conditionals, probability and statistics, and special functions. Chapter 10.1 “Operators & Functions” lists them all.

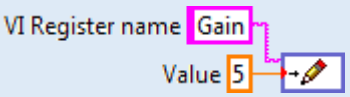
- A function is defined as any function name followed by parentheses enclosing the function arguments. Semicolons separate the function arguments: “function_name(arg1;arg2;…;argn)”.
- Function names are case insensitive, so that “sin(x)” means the same as “SIN(x)” and “Sin(x)”.
- N-ary functions (functions that take a variable number of arguments) will ignore non-specified arguments instead of error out on them, thus “MAX(2;;;6;;5)” is the same as “MAX(2;6;5)”.

5.5 VI Register support

The Expression Parser toolset supports VI Register values inside the expressions, thus enabling you to work with externally controlled variables.

It’s beyond this document to describe in detail what VI Registers are, but in brief the GPower VI Register toolset is a technology that gives you high performance local-like variables with global scope – without having to define any controls (contrary to locals), any files on disk (contrary to globals), or any project items (contrary to Shared Variables for instance). You can write, read, register for events and much more on VI Registers.

A write with a VI Register looks like this in your code:



More info is available in the VI Register help.

When you write values to VI Registers in other parts of your LabVIEW code, you can have expressions evaluated by this Expression Parser toolset depend on these values. The syntax for this is “VIR(name)”, for instance “2*VIR(Gain)”, which will read the value of the ‘Gain’ VI Register and multiply it by 2.

- **VI Register names**
  All characters in any combination are valid in a VI Register name, so “VIR(2*+p/6)” for instance is valid and means one read of the VI Register with the (somewhat odd) name “2*+p/6”. Mind that you can’t really use the right parenthesis character “)” in a VI Register name, as that would make the parser think the VIR() function statement ended earlier than intended (and cause a syntax error on the rest of the expression string).
- **Each VI Register read is atomic**
  Each “VIR(name)” sub-expression reads the VI Register on its own, so evaluating “2*VIR(name)” will result in **one** read of the VI Register, while “VIR(name)+VIR(name)” for instance will result in **two** distinct reads of the same VI Register, with the possibility of two different values of the same VI Register (if a write to that VI Register happened in between those reads).
- **Buffered vs. unbuffered VI Registers**
  A VI Register is defined as either buffered or unbuffered (current value). If you use a buffered VI Register in your expression each VI Register read will pop a new value from the register buffer. If

you use an unbuffered VI Register each VI Register read will just peek the current value from the register.

- **Uninitialized VI Registers**
  If the VI Register does not hold any value when you execute an expression evaluation, the evaluation will use a value of zero instead and issue a warning. This will happen if a buffered VI Register has an empty buffer, or when an unbuffered VI Register hasn't yet been written any value to.

5.6 Order of operations

Standard order of operations apply with unary operators taking highest precedence. The latter means for instance that the expression string "-3^2" is interpreted as "(-3)^2".

6 Numeric considerations

6.1 Overflow when parsing

Numeric overflow can happen during parsing of constants, and the outcome depends on the selected data type:

- Overflow in floating point (SGL, DBL, EXT, CSG, CDB or CXT) parsing of a constant will not yield an error but a value of infinity for that constant instead, e.g. “4e5000” → “Inf”.
- Overflow in integer parsing of a constant will throw an error (as integer data types does not have any built-in way to express infinity), e.g. “150” → “Overflow when parsing 150 as I8”.

6.2 Overflow when evaluating

Numeric overflow can happen during expression evaluation, and the outcome depends on the selected data type:

- Overflow during floating point evaluation (SGL, DBL, EXT, CSG, CDB or CXT) will result in a value of infinity, e.g. “110^802” → “Inf”.
- Overflow during integer evaluation will abort the rest of the evaluation and yield an overflow error, e.g. “40*11” → “Overflow: *.” (for I8 evaluation).

Remember that overflow in intermediate steps of the evaluation are just as big a problem as overflow in the final result. Sometimes you can re-arrange an expression to make it less likely to overflow *during* the evaluation, for instance by doing a division before a multiplication instead of the other way around.

Example: The following two expressions are mathematically equivalent and will evaluate to 55 which fit nicely within the data range of an I8 [-128;127]:

$$A)\ \frac{40 * 11}{8} \qquad B)\ \frac{40}{8} * 11$$

However, expression A) will overflow since the intermediate result of $40 * 11 = 440$ won’t fit in the I8. Expression B) will evaluate all the way to 55 without error, as all intermediate results fit within the I8.

Any expression re-arrangement might introduce other inaccuracies though, for instance loss of precision or even numeric underflow, so take care designing your math expressions.

6.3 Rounding of floating point values

Keep in mind the intrinsic limitations of the ANSI/IEEE 754-1985 floating point data types (SGL, DBL and EXT) in regards to number of digits that can be expressed. If any given round-off is significant or not depends on your application of course, but in addition to the finite number of significant digits (approximately 6-20 depending on data type and platform) it’s just as important to remember that the floats do not represent exact decimals. Here is the value 3.1 displayed with 20 digits of precision:

SGL: 3,09999990463256836000 DBL: 3,10000000000000009000 EXT: 3,10000000000000000000

The SGL represents 3.1 up to 7 significant digits, which isn't bad considering. Just remember that a SGL won't ever represent 3.1 exactly even if you used a mathematical rounding function on it. And that can have an impact all the way through your calculations and to the final result you get displayed.

6.4 Logical evaluation

This toolset operates with numeric values only, Booleans aren't supported as data type. Therefore, comparisons and tests that do Boolean logic return 1 for True and 0 for False. E.g. the expression "1<8" will evaluate to 1, and thus an expression like "(1<8)+(9>5)" will evaluate to 2.

Note that shorthand notation often used in literature of limits such as "2<x<9" is not evaluated as "x must be between 2 and 9 to yield True". The correct mathematical syntax for such an expression is "2<x && x<9".

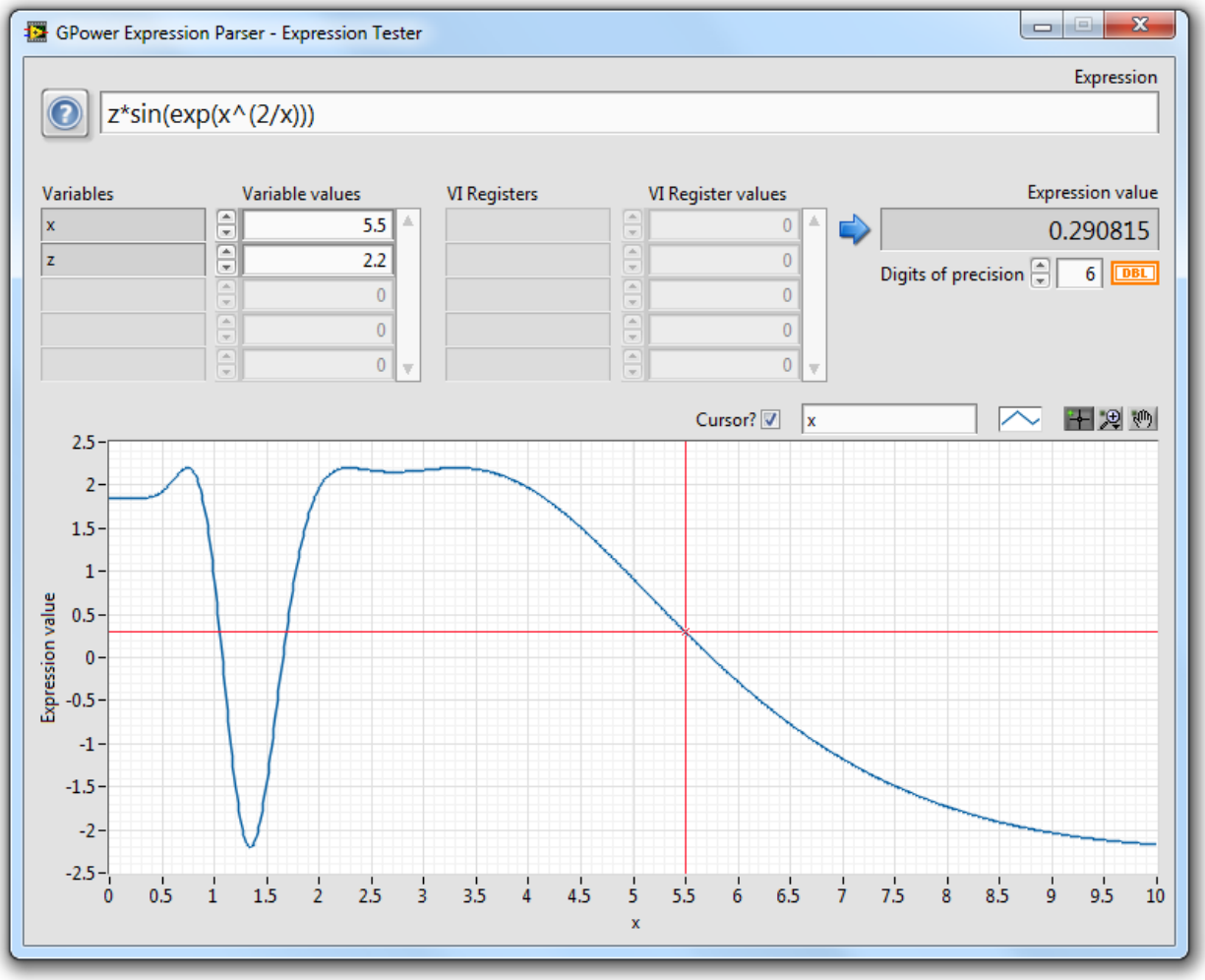
Logical operators and functions will assign False to both real and complex 0, and True to any numeric value different from 0.

6.5 Trigonometric evaluation

Trigonometric functions are always evaluated in radians mode.

7 Expression Tester

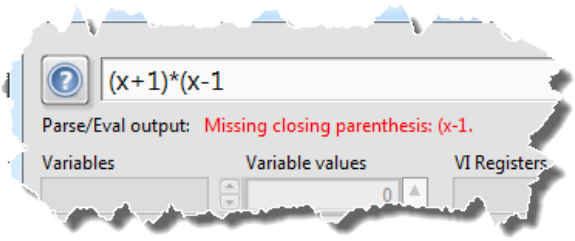
To experiment with expression syntax and to verify how different expressions evaluate with varying numeric input and with different data types, the Expression Parser toolset includes an Expression Tester application:



You can find this Expression Tester in the LabVIEW Tools menu:

Tools >> GPower >> Expression Tester…

The Expression Tester can be set to parse and evaluate with any of the 14 supported data types, which will be reflected in the variables, the VI Registers, and the expression value fields. The graph window however always uses DBL values, so in the graph you can run into not being able to view very large values even if you select the EXT or CXT data types (since the numeric range of those exceed DBL).



The Expression Tester will for instance highlight syntax errors, explain overflow conditions, and let you explore numeric ranges in a graph as you type expressions.

This will help you get the expression just right before committing it to code.

7.1 VI Registers in the Expression Tester

The Expression Tester application runs as an isolated environment, so VI Registers set outside the Expression Tester won't be read by the Expression Tester application. The only way to set VI Register values for the Expression Tester is with its own "VI Register values" control on its front panel.

8 Getting the most out of this toolset

Please study this section to understand when to use the Expression Parser toolset and which exactly its strengths are. There are two main use cases where the Expression Parser toolset is a perfect fit:

1) **Selecting between math algorithms**
   If your program needs to select between several different algorithms at runtime, using built-in methods you'd typically implement a case structure with math functions in them:

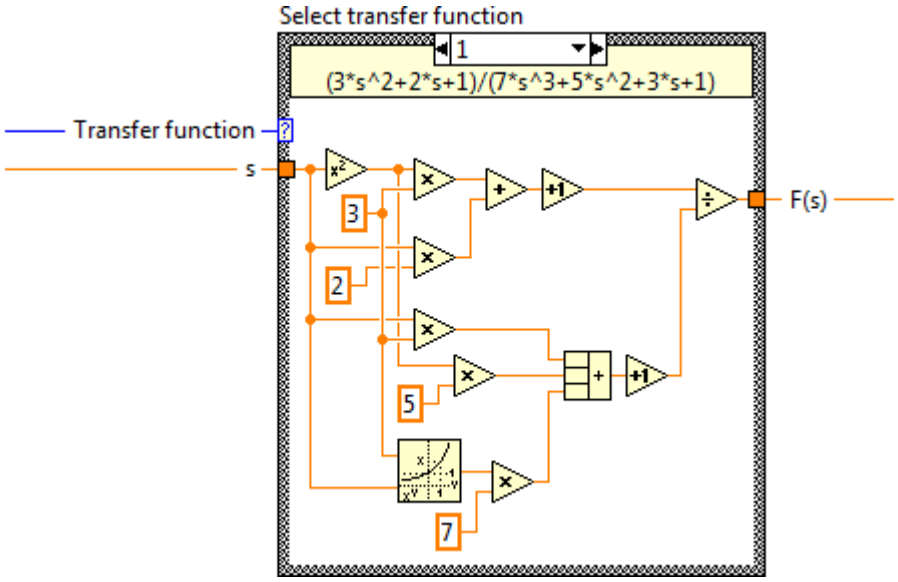


Figure 1 - A typical way to select between static algorithms

Figure 1 above requires careful documentation on the block diagram, the math code is hard to understand, and code maintenance is error-prone – especially you run the risk of evolving mismatches between code and documentation.

Consider this solution using Expression Parser instead:

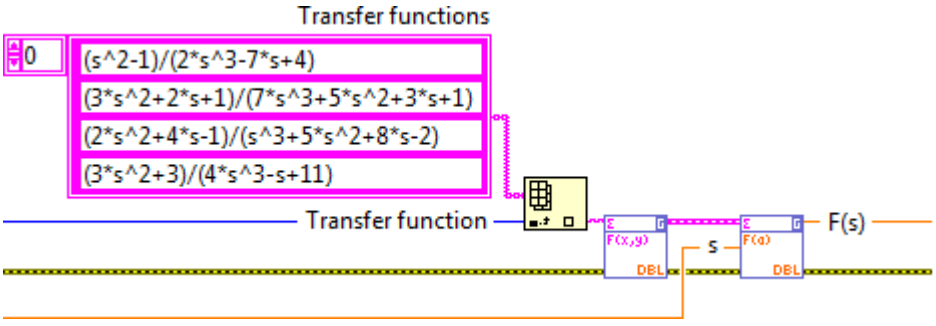


Figure 2 - Selecting between algorithms with Expression Parser

In Figure 2 the implementation *is* the documentation, all the transfer functions can be seen on the block diagram at the same time, and the code is much easier recognizable as a choice between a selection of math functions. You get error handling thrown in for free as well.

2) **Inputting math expressions at runtime**

When your math expression isn't known at edit time you'll have to parse it at runtime. Perhaps you read the math expression from a file, or perhaps the user enters it on the UI. For dealing with this situation your only built-in choice is using the Formula Parsing VIs:

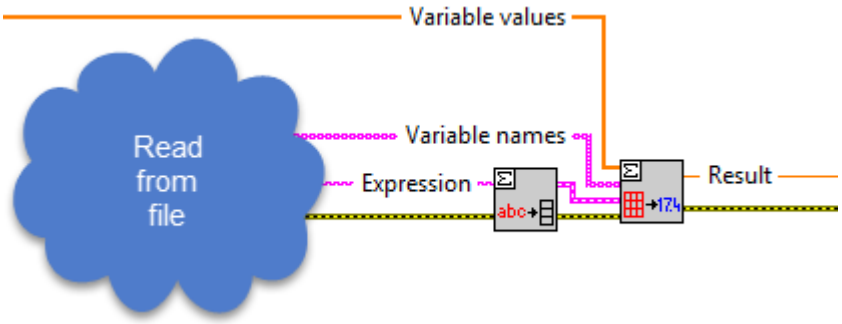


Figure 3 - Using the built-in Formula Parsing VIs

As section 2.1 "Method comparison" shows you the Formula Parsing VIs come with many limitations. For much better function support, better error handling, and better performance, you should use the Expression Parser toolset for these kinds of situations:

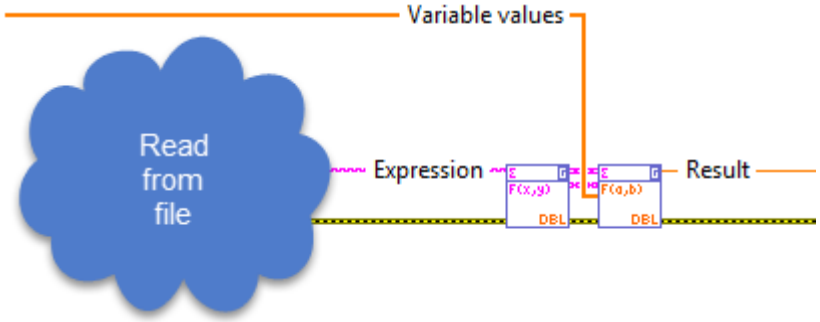


Figure 4 - Using the Expression Parser toolset

8.1 Variable substitution

If a variable is constant for a sufficient number of evaluations, you can potentially increase evaluation performance by substituting the constant value for that variable:

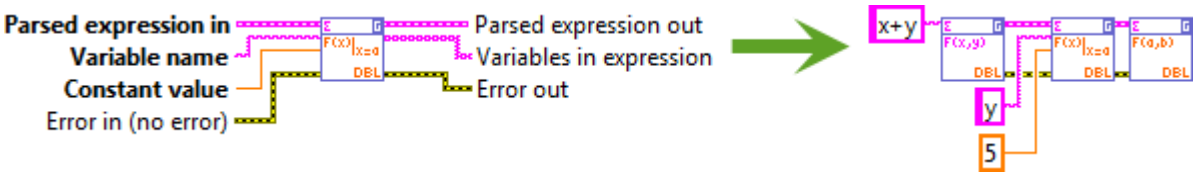


Figure 5 - Substituting variables can improve evaluation performance

Substituting a variable will trigger a simplification routine to run on the parsed expression, which among other things will calculate all constant terms within the expression. For instance, if you have the expression '2*x-y/4' and substitute 'y=8', the resulting expression will be '2*x-2' which is faster to evaluate than '2*x-8/4'.

Substitution and subsequent simplification takes more time to perform than inputting a variable of course, so don't just replace ordinary variable input with substitution in your code. Only substitute

variables when you can spare the CPU cycles to do so in your program, i.e. when you know that you'll do many subsequent evaluations with that constant value in your expression.

8.2 Single variable evaluation

If your expression contains exactly one variable you can evaluate it with the EvalSingleVar function. It doesn't matter what the variable name is, only that there is exactly one. This mode of evaluation won't be faster, it's just a bit simpler on the block diagram:

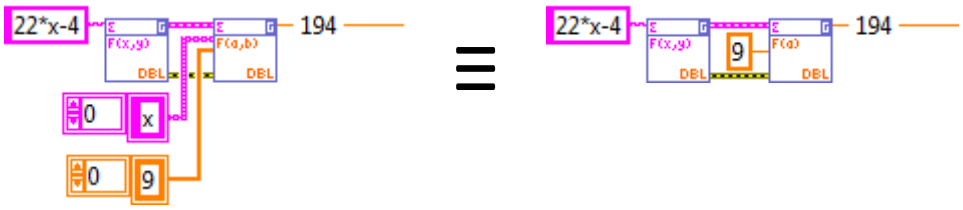


Figure 6 - Ordinary variable evaluation (left) vs. single variable evaluation (right)

8.3 Evaluating multiple values

Calculating a numeric value from an expression string takes two steps: 1) **Parsing** the expression string into a data structure, and 2) plugging in variable values and **evaluating** the data structure into a result.

The parsing step usually takes up much more time than the evaluation step. If you are just calculating one result you can't get around doing one parsing step followed by one evaluation step. But if you're calculating many results from the same expression it's important that you do the parsing step only once, and then repeat only the evaluation step for each result. Keep the parsing VI outside a loop like this for instance:

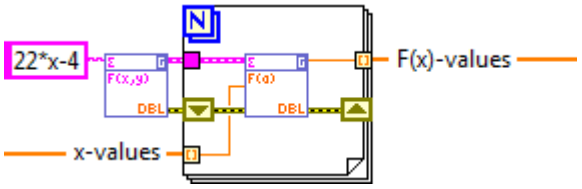
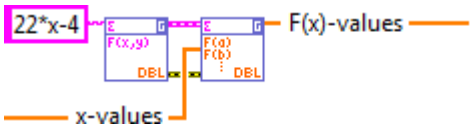


Figure 7 - Parse only once and then evaluate repeatedly

If you already have all the x-values as an array the EvalSingleVar VI lets you even dispose of the For-loop in this case, as EvalSingleVar is polymorphic and will also accept an array of values:



8.4 Examples

An example-VI that showcases the main features of this Expression Parser toolset is installed into the <LabVIEW>\examples\GPower\ExprParser\ folder. This example can also be found through the NI Example Finder in the LabVIEW menu at Help >> Find Examples… (select 'Directory Structure' view in the Example Finder to make the GPower folder visible).

9 Special functions

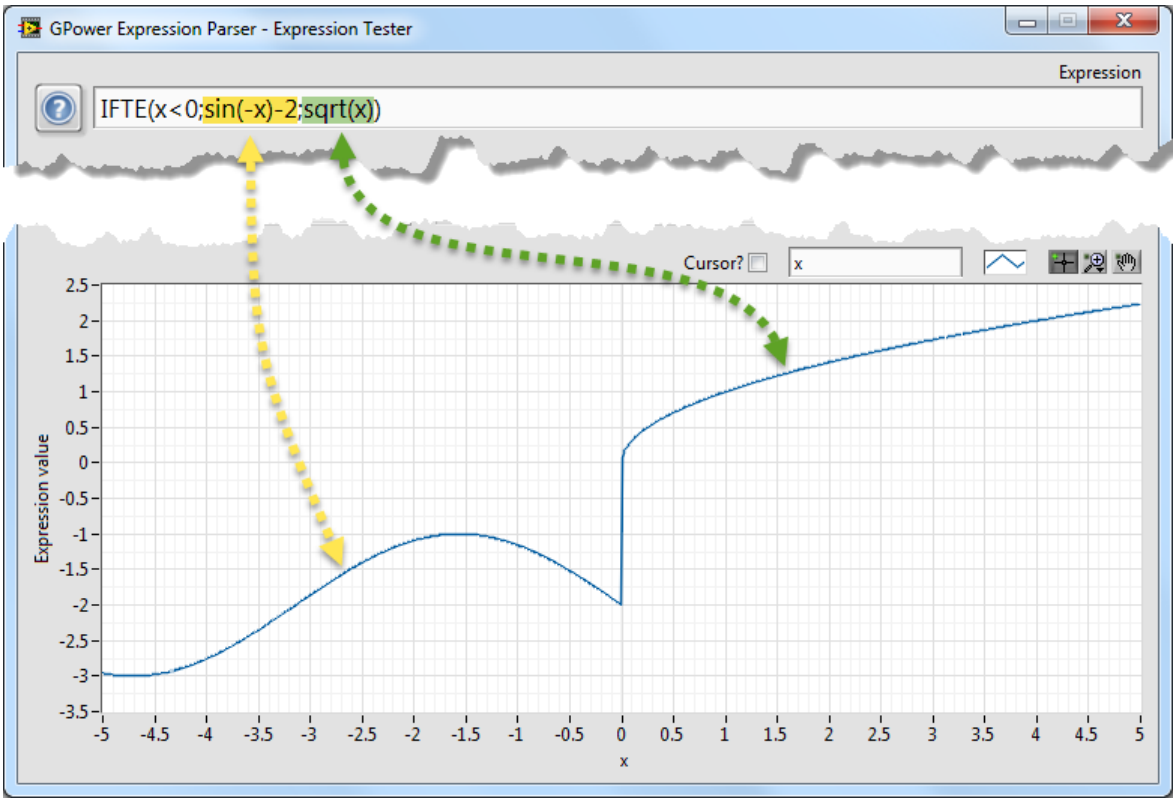
Most functions in the GPower Expression Parser toolset are well known and have straightforward syntax, like SIN(x) and EXP(x) for instance. These are all listed in section 10.1 “Operators & Functions”. A few functions deserve a more detailed mention though:

9.1 Piecewise defined functions

Piecewise defined functions are functions that have different function definitions in different parts of the independent variable range, for instance:

$$f(x) = \begin{cases} \sin(-x) - 2, & x < 0 \\ \sqrt{x}, & x \geq 0 \end{cases}$$

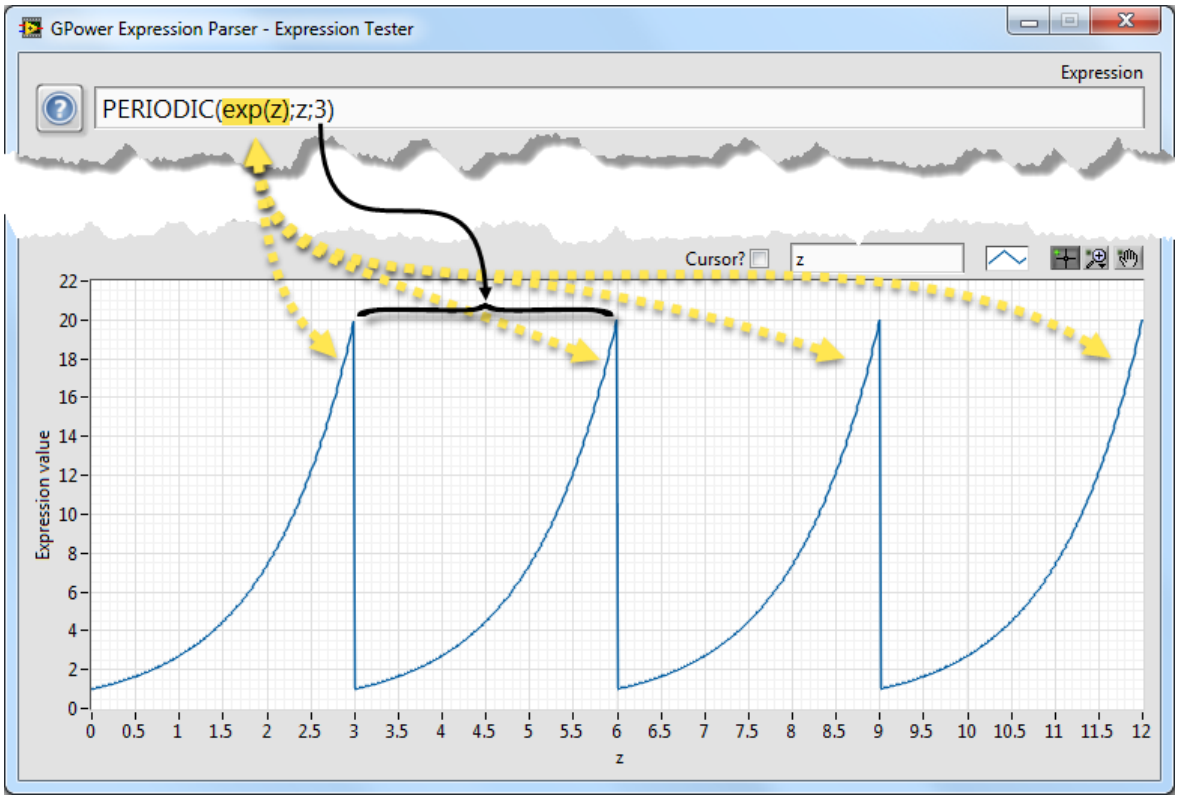
With the Expression Parser toolset you build piecewise defined functions with the IFTE function (IFTE means **IF**-**T**hen-**E**lse). The syntax is IFTE(<condition>;<true-expression>;<false-expression>):



For piecewise defined functions with more than two different function definitions, you just nest multiple IFTE-functions inside each other.

9.2 Custom periodic functions

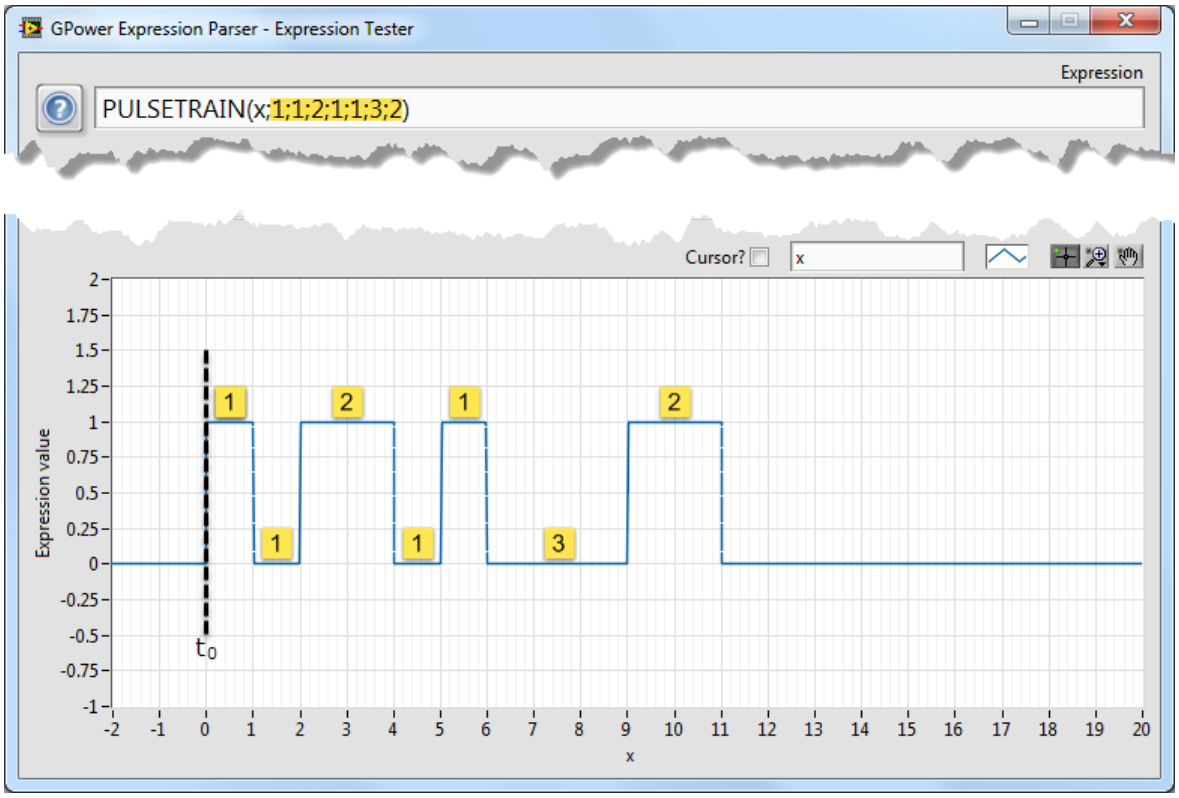
There are many common periodic functions; SIN(x) for instance which is periodic with 2*π, or TAN(x) which is periodic with π. With the Expression Parser toolset you can build your own custom periodic functions, the syntax is PERIODIC(<function>;<function independent variable>;<periodicity>):



9.3 Pulse trains

A pulse train is a typically asymmetric square wave often used for describing a digital signal. Such a digital signal could be a trigger, a filter or modulation, or even a protocol describing some data communication.

The PULSETRAIN function in this Expression Parser toolset lets you define such a pulse train with an amplitude of 1, which starts at t_0 and has a number of alternating level pulses which you specify the width(s) of. The syntax is PULSETRAIN(<independent variable>;<high-time>;<low-time>;<high-time>;<low-time>;…):



You can combine the PULSETRAIN and PERIODIC functions to generate a waveform of some arbitrary repeating digital signal for instance.

10 Reference

10.1 Operators & Functions

Table 2 below lists the complete function and operator support depending on evaluation mode (data type). Some functions have restrictions on their argument value ranges, exceeding these will usually result in numeric overflow. Special cases have a note associated with them beside the checkmark, with a matching explanation just after the table.

Function/Operator			Evaluation mode support			
			Signed integer I8, I16, I32, I64	Unsigned integer U8, U16, U32, U64	Real floating-point SGL, DBL, EXT	Complex floating-point CSG, CDB, CXT
Arithmetic						
+	Infix	Addition.	√	√	√	√
-	Infix	Subtraction.	√	√	√	√
-	Prefix	Negative sign.	√		√	√
*	Infix	Multiplication.	√	√	√	√
/	Infix	Division.	√	√	√	√
!	Postfix	Factorial for integers, Gamma(n+1) for floating-point numbers.	√	√	√	√ [1]
%	Postfix	Percentage, same as x/100.	√	√	√	√ [1]
%CH(x;y)		Percent change from x to y.	√	√	√	√ [1]
%T(x;y)		Percent total y is of x.	√	√	√	√ [1]
ARG(x)		Angle of imaginary number x.	√	√	√	√
CONJ(x)		Conjugation of imaginary number x.	√	√	√	√
INV(x)		1/x.	√	√	√	√
MOD(x;y)		x modulo y.	√	√	√	√
MULT(x;y;z...)		Multiplies all arguments.	√	√	√	√
NEG(x)		Negation.	√		√	√
QUOT(x;y)		Integer quotient of x/y.	√	√	√	√ [1]
RAND()		Random number between 0 and 1.	√	√	√	√
RAND(x;y)		Random number between x and y.	√	√	√	√
REM(x;y)		Remainder of x/y.	√	√	√	√ [1]
SIGN(x)		Sign of x.	√	√	√	√
SUM(x;y;z...)		Sums all arguments.	√	√	√	√
Powers & logarithms						
^	Infix	Power operator.	√	√	√	√
EXP(x)		Exponential function.			√	√
EXPM(x)		EXP(x)-1, more precise for small x than EXP(x).			√	√

LN(x)	Natural logarithm.			√	√
LNP1(x)	LN(x+1), more precise for small x than LN(x).			√	√
LOG(x) or LOG10(x)	Common (base 10) logarithm.			√	√
LOGN(x;n)	Base n logarithm of x.			√	√
SQRT(x)	Square root.			√	√
XROOT(x;y)	yth root of x.			√	√
Trigonometric functions					
ACOS(x)	Arc cosine.			√	√
ACOSH(x)	Hyperbolic arc cosine.			√	√
ACOT(x)	Arc cotangent.			√	√
ACOTH(x)	Hyperbolic arc cotangent.			√	√
ACSC(x)	Arc cosecant.			√	√
ACSCH(x)	Hyperbolic arc cosecant.			√	√
ASEC(x)	Arc secant.			√	√
ASECH(x)	Hyperbolic arc secant.			√	√
ASIN(x)	Arc sine.			√	√
ASINH(x)	Hyperbolic arc sine.			√	√
ATAN(x)	Arc tangent.			√	√
ATANH(x)	Hyperbolic arc tangent.			√	√
COS(x)	Cosine.			√	√
COSH(x)	Hyperbolic cosine.			√	√
COT(x)	Cotangent ≡ 1/TAN(x).			√	√
COTH(x)	Hyperbolic cotangent.			√	√
CSC(x)	Cosecant ≡ 1/SIN(x).			√	√
CSCH(x)	Hyperbolic cosecant.			√	√
SEC(x)	Secant ≡ 1/COS(x).			√	√
SECH(x)	Hyperbolic secant.			√	√
SIN(x)	Sine.			√	√
SINC(x)	SIN(x)/x.			√	√
SINH(x)	Hyperbolic sine.			√	√
SINHC(x)	SINH(x)/x.			√	√
TAN(x)	Tangent.			√	√
TANC(x)	TAN(x)/x.			√	√
TANH(x)	Hyperbolic tangent.			√	√
TANHC(x)	TANH(x)/x.			√	√
Rounding & parts					
ABS(x)	Absolute value of real number x, or magnitude of imaginary number x.	√	√	√	√
CEIL(x)	Round x up to nearest integer.	√	√	√	√
FLOOR(x)	Round x down to nearest integer.	√	√	√	√

FP(x)		Fractional part of x.	√	√	√	√
IM(x)		Imaginary part of x.	√	√	√	√
IP(x)		Integer part of x.	√	√	√	√
MANT(x)		Base 10 mantissa of x.			√	√ [1]
RE(x)		Real part of x.	√	√	√	√
RND(x;y)		Round x to y decimal places. If y is negative x is rounded to nearest tens (y=-1), hundreds (y=-2), thousands (y=-3) etc.	√	√	√ [4 (y only)]	√ [1 (y only), 4 (y only)]
TRNC(x;y)		Truncate x at y decimal places. If y is negative x is truncated to nearest tens (y=-1), hundreds (y=-2), thousands (y=-3) etc.	√	√	√ [4 (y only)]	√ [1 (y only), 4 (y only)]
XPON(x)		Base 10 exponent of x.	√	√	√	√
Logical operations						
&&	Infix	Logical AND operator.	√	√	√	√
||	Infix	Logical OR operator.	√	√	√	√
AND(x;y;z...)			√	√	√	√
NAND(x;y;z...)			√	√	√	√
NOR(x;y;z...)			√	√	√	√
NOT(x)			√	√	√	√
OR(x;y;z...)			√	√	√	√
XNOR(x;y)			√	√	√	√
XOR(x;y)			√	√	√	√
Integer operations						
~	Prefix	Bitwise NOT operator.	√	√	√ [3]	√ [1, 3]
&	Infix	Bitwise AND operator.	√	√	√ [3]	√ [1, 3]
|	Infix	Bitwise OR operator.	√	√	√ [3]	√ [1, 3]
<<	Infix	Left shift operator, x << y shifts x y bits to the left.	√	√	√ [3 (x only), 4 (y only)]	√ [1, 3 (x only), 4 (y only)]
>>	Infix	Right shift operator, x >> y shifts x y bits to the right.	√	√	√ [3 (x only), 4 (y only)]	√ [1, 3 (x only), 4 (y only)]
ASR(x;y)		Arithmetic right shift function, x is shifted y bits to the right while preserving the sign bit.	√	√	√ [2 (x only), 4 (y only)]	√ [1, 2 (x only), 4 (y only)]
RL(x;y)		Rotate x y bits to the left.	√ [4 (y only)]	√ [4 (y only)]	√ [2 (x only), 4 (y only)]	√ [1, 2 (x only), 4 (y only)]
RR(x;y)		Rotate x y bits to the right.	√ [4 (y only)]	√ [4 (y only)]	√ [2 (x only), 4 (y only)]	√ [1, 2 (x only), 4 (y only)]
SL(x;y)		Shift x y bits to the left.	√	√	√ [3 (x only), 4 (y only)]	√ [1, 3 (x only), 4 (y only)]
SR(x;y)		Shift x y bits to the right.	√	√	√ [3 (x only), 4 (y only)]	√ [1, 3 (x only), 4 (y only)]
Tests & conditionals						
!=	Infix	'Not equal?' test.	√	√	√	√
==	Infix	'Equal?' test.	√	√	√	√
<	Infix	'Less than?' test.	√	√	√	√ [1]
<=	Infix	'Less than or equal?' test.	√	√	√	√ [1]
>	Infix	'Greater than?' test.	√	√	√	√ [1]
>=	Infix	'Greater than or equal?' test.	√	√	√	√ [1]
IFTE(f(x);a;b)		If f(x) evaluates to logically True then return a else return b. See	√	√	√	√

	section 9.1 “Piecewise defined functions” for a detailed explanation.				
MAX(x;y;z...)		√	√	√	√ [1]
MIN(x;y;z...)		√	√	√	√ [1]
Probability and statistical functions					
GEOMETRICMEAN(x;y;z...)	Geometric mean.			√	√
HARMONICMEAN(x;y;z...)	Harmonic mean.			√	√
MEAN(x;y;z...)	Arithmetic mean.			√	√
MEDIAN(x;y;z...)	Statistical median.			√	√
PERCENTILE(p;x;y;z...)	Percentile p of the dataset.			√	√ [1]
PSDEV(x;y;z...)	Population standard deviation.			√	√
PVAR(x;y;z...)	Population variance.			√	√
RMS(x;y;z...)	Root-mean-square.			√	√
SDEV(x;y;z...)	Sample standard deviation.			√	√
TRIMMEDMEAN(p;x;y;z...)	Trims p percent of each end of the dataset, then arithmetic mean of the rest.			√	√ [1] (p only)
VAR(x;y;z...)	Sample variance.			√	√
Special functions					
AIRYAI(x)	Airy Ai function.			√	√ [1]
AIRYBI(x)	Airy Bi function.			√	√ [1]
BESSELI(x;v)	Modified Bessel function of the first kind.			√	√ [1]
BESSELJ(x;v)	Bessel function of the first kind.			√	√ [1]
BESSELK(x;v)	Modified Bessel function of the second kind.			√ [2] (v only)	√ [1, 2] (v only)
BESSELY(x;v)	Bessel function of the second kind.			√	√ [1]
BETA(x;y)				√	√ [1]
BETA(x;y;a)	Incomplete beta function.			√	√ [1]
CHI(x)	Hyperbolic cosine integral.			√	√ [1]
CI(x)	Cosine integral.			√	√ [1]
COMB(n;k) or NCR(n;k)	Number of combinations when taking n things k at a time.	√	√	√	√ [1]
DAWSON(x)	Dawson's integral.			√	√ [1]
DELTA(x)	Kronecker delta function (1 if x=0, else 0).	√	√	√	√
DELTA(x;y;z...)	Kronecker delta function (1 if all arguments are equal, else 0).	√	√	√	√
DIGAMMA(x)				√	√ [1]
ELLIPTICC(x;k)	Jacobi elliptic function cn.			√	√ [1]
ELLIPTICD(x;k)	Jacobi elliptic function dn.			√	√ [1]
ELLIPTICE(k)	Legendre complete elliptic integral of the second kind.			√	√ [1]
ELLIPTICE(k;a)	Legendre incomplete elliptic integral of the second kind.			√	√ [1]
ELLIPTICF(k;a)	Legendre incomplete elliptic integral of the first kind.			√	√ [1]
ELLIPTICK(k)	Legendre complete elliptic integral of the first kind.			√	√ [1]
ELLIPTICS(x;k)	Jacobi elliptic function sn.			√	√ [1]

EN(x;n)	Exponential integral				
ERF(x)	Error function.			√	√ [1]
FRESNELC(x)	Fresnel cosine integral.			√	√ [1]
FRESNELS(x)	Fresnel sine integral.			√	√ [1]
GAMMA(x)	Gamma function.			√	√ [1]
GAMMA(x;a)	Incomplete gamma function.			√	√ [1]
GCD(x;y)	Greatest common divisor between x and y.	√	√	√	√ [1]
HANKEL1(x;v)	Hankel function of the first kind.				√ [1]
HANKEL2(x;v)	Hankel function of the second kind.				√ [1]
HYPERGAUSS(x;a;b;c)	Gauss hypergeometric function.			√	√ [1]
HYPERKUMMER(x;a;b)	Kummer hypergeometric function.			√	√ [1]
HYPERTRICOMI(x;a;b)	Tricomi hypergeometric function.			√	√ [1]
KELVINBEI(x;v)	Kelvin Bei function.			√ [2 (v only)]	√ [1, 2 (v only)]
KELVINBER(x;v)	Kelvin Ber function.			√ [2 (v only)]	√ [1, 2 (v only)]
KELVINKEI(x;v)	Kelvin Kei function.			√ [2 (v only)]	√ [1, 2 (v only)]
KELVINKER(x;v)	Kelvin Ker function.			√ [2 (v only)]	√ [1, 2 (v only)]
LCM(x;y)	Least common multiple between x and y.	√	√	√	√ [1]
LNGAMMA(x)				√	√ [1]
PERIODIC(f(x);x;p)	Turns f(x) periodic with periodicity p. See section 0 “ Custom periodic functions” for a detailed explanation.	√	√	√	√ [1 (p only)]
PERM(n;k) or NPR(n;k)	Number of permutations when taking n things k at a time.	√	√	√	√ [1]
PULSETRAIN(x;a;b;c;...)	From x=0 alternating 1 and 0 in sections a, b, c etc. wide. See section 0 “ Pulse trains” for a detailed explanation.	√	√	√	√ [1]
RAMP(x)	0 for x<0, else x.	√	√	√	√ [1]
RECTANGLE(x)	1 for 0<=x<1, else 0.	√	√	√	√ [1]
SAWTOOTHWAVE(x)	Amplitude -1 to 1, period 1.			√	√ [1]
SHI(x)	Hyperbolic sine integral.			√	√ [1]
SI(x)	Sine integral.			√	√ [1]
SPHERICALBESSELJ(x;v)	Spherical Bessel function of the first kind.			√ [2 (v only)]	√ [1, 2 (v only)]
SPHERICALBESSELY(x;v)	Spherical Bessel function of the second kind.			√ [2 (v only)]	√ [1, 2 (v only)]
SPHERICALHANKEL1(x;n)	Spherical Hankel function of the first kind.				√ [1, 2 (n only)]
SPHERICALHANKEL2(x;n)	Spherical Hankel function of the second kind.				√ [1, 2 (n only)]
SPI(x)	Spence’s integral.			√	√ [1]
SQUAREWAVE(x)	Amplitude 0 to 1, period 2.	√	√	√	√ [1]
STEP(x)	Heaviside step function (0 if x<0, else 1).	√	√	√	√ [1]
STRUVEH(x;v)	Struve function.			√	√ [1]
TRIANGLE(x)	(x+1) for -1<=x<0, (1-x) for 0<=x<=1, else 0.			√	√ [1]
TRIANGLEWAVE(x)	Amplitude -1 to 1, period 1.			√	√ [1]
ZETA(x)	Riemann zeta function.			√	

Variable access					
VIR(name)	Read the VI Register named “name”.	√	√	√	√
Built-in constants (prepend the CONST keyword, e.g. CONSTpi)					
ag	Alladi-Grindstead constant.			√	√
apery	Apéry’s constant.			√	√
artin	Artin’s constant.			√	√
backhouse	Backhouse’s constant.			√	√
barban	Barban’s constant.			√	√
baxter	Baxter’s four-coloring constant.			√	√
bernstein	Bernstein’s constant.			√	√
bloch	Bloch constant.			√	√
brun	Brun’s constant.			√	√
cahen	Cahen’s constant.			√	√
catalan	Catalan’s constant.			√	√
conway	Conway’s constant.			√	√
debruijn	de Bruijn’s constant.			√	√
delian	Delian constant 2^(1/3).			√	√
e	Euler’s number.			√	√
eb	Erdós-Borwein constant.			√	√
em	Euler-Mascheroni constant.			√	√
ft	Feller-Tornier constant.			√	√
fibonacci	Fibonacci factorial constant.			√	√
fo	Flajolet-Odlyzko constant.			√	√
fr	Fransén-Robinson constant.			√	√
freiman	Freiman’s constant.			√	√
gauss	Gauss’s constant.			√	√
gd	Golomb-Dickman constant.			√	√
gelfond	Gelfond’s constant.			√	√
gieseking	Gieseking’s constant.			√	√
gk	Glaisher-Kinkelin constant.			√	√
gkw	Gauss-Kuzmin-Wirsing constant.			√	√
gohs	Goh-Schmutz constant.			√	√
gold	The golden ratio (1+SQRT(5))/2.			√	√
goldc	Golden ratio conjugate 2/(1+SQRT(5)).			√	√
gompertz	Gompertz constant.			√	√
gs	Gelfond-Schneider constant 2^SQRT(2).			√	√
hardhex	Hard hexagon entropy constant.			√	√
hardsq	Hard square entropy constant.			√	√
hbm	Heath-Brown-Moroz constant.			√	√
heptanacci	Heptanacci constant.			√	√

hexanacci	Hexanacci constant.			√	√
hm	Hall-Montgomery constant.			√	√
i	The unit imaginary number SQRT(-1).				√
inf	Positive infinity.			√	√
infn	Negative infinity.			√	√
kb	Kepler-Bouwkamp constant.			√	√
khinchin	Khinchin's constant.			√	√
kl	Komornik-Loreti constant.			√	√
lehmer	Lehmer’s constant.			√	√
lemniscate	Lemniscate constant.			√	√
lengyel	Lengyel’s constant.			√	√
levy	Lévy constant			√	√
lieb	Lieb’s square ice constant.			√	√
lochs	Lochs’s constant.			√	√
lr	Landau-Ramanujan constant.			√	√
merten	Merten’s constant.			√	√
mills	Mills’ constant.			√	√
murata	Murata’s constant.			√	√
nan	NaN or Not-A-Number.			√	√
niven	Niven’s constant.			√	√
norton	Norton’s constant.			√	√
parabolic	Universal parabolic constant.			√	√
paris	Paris constant.			√	√
pell	Pell constant.			√	√
pentanacci	Pentanacci constant.			√	√
pi	π.			√	√
plastic	Plastic constant.			√	√
pogson	Pogson’s ratio.			√	√
porter	Porter’s constant.			√	√
rabbit	Rabbit constant.			√	√
ramanujan	Ramanujan constant.			√	√
recipfib	Reciprocal Fibonacci constant.			√	√
reciplucas	Reciprocal Lucas constant.			√	√
robbins	Robbins constant.			√	√
rs	Ramanujan-Soldner constant.			√	√
rutherford	Rutherford constant.			√	√
sh	Stolarsky-Harborth constant.			√	√
shallit	Shallit constant.			√	√
sierpinski	Sierpinski constant.			√	√
silverc	Silver constant.			√	√

silverr	Silver ratio.			√	√
somos	Somos's quadratic recurrence constant.			√	√
sqrt2	Pythagoras's constant SQRT(2).			√	√
sqrt3	Theodorus's constant SQRT(3).			√	√
tetranacci	Tetranacci constant.			√	√
thue	Thue constant.			√	√
tm	Thue-Morse constant.			√	√
totient	Totient constant.			√	√
tp	Takeuchi-Prellberg constant.			√	√
tribonacci	Tribonacci constant.			√	√
twinprim	Twin primes constant.			√	√
vallee	Vallée constant.			√	√
varga	Varga's constant.			√	√
w1	The complex half period of the Weierstrass elliptic function in the eigenharmonic case.				√
w2	The real half period of the Weierstrass elliptic function in the eigenharmonic case.			√	√
wallis	Wallis's constant.			√	√
weierstrass	Weierstrass constant.			√	√
wg	Wilbraham-Gibbs constant.			√	√
wyler	Wyler's constant.			√	√
zs	Zolotarev-Schur constant.			√	√

Table 2 - Function and operator support across data types

[1)] Expects real arguments (SGL for CSG, DBL for CDB, and EXT for CXT). Error if overflow or non-zero imaginary part.
[2)] Expects signed integer arguments (I16 for SGL, I32 for DBL, and I64 for EXT). Error if overflow or non-zero fractional part.
[3)] Expects unsigned integer arguments (U16 for SGL, U32 for DBL, and U64 for EXT). Error if overflow or non-zero fractional part.
[4)] Supports a smaller than full range integer argument, for instance I16 for EXT data type or I32 for U64 data type. Error if overflow or non-zero fractional part.

10.2 Error and warning table

This toolset can return a number of errors and warnings as shown in Table 3 below (only code 402802 is a warning, the rest are errors):

Code	Message	Typical cause
When parsing		
402780	Bad binary integer value: <integer>.	Characters other than “0” and “1” were found in a binary unsigned integer constant.
402781	Bad decimal integer value: <integer>.	Characters other than “0” through “9” were found in a decimal unsigned integer constant.
402782	Bad hexadecimal integer value: <integer>.	Characters other than “0” through “9” and “A” through “F” were found in a hexadecimal unsigned integer constant.
402783	Bad integer constant specifier: <specifier>. Should be "0x...".	A number other than 0 was encountered in front of x, for instance “9x…”. Implied multiplication isn’t supported, and variable names may not start with a number.
402784	Bad number of arguments for function <function name>: Expected <number>, found <number>.	Too many or too few arguments for a given function, like “SIN(a;b)” (the sine function expects one argument, here there are two).
402785	Bad octal integer value: <integer>.	Characters other than “0” through “7” were found in an octal unsigned integer constant.
402786	Expected number, found empty string.	Typically when the parser encounters a prefix negation operator (a minus sign that is not the infix minus operator) but finds no numbers after it, as in “RAND(-;5)”.
402787	Ill-formatted number: <number>.	For instance a number with more than one decimal point: “4.5.96”.
402788	Implied multiplication not supported: <expression>(.	For instance a number just before a parenthesis: “2(x+2)”. If you intended multiplication do explicit multiplication instead: “2*(x+2)”.
402789	Integer overflow: <integer>. Value must be 2^64-1 or less.	The parser parses all unsigned integer constants in the expression as U64, and then casts them if necessary to a lower data type (signed 8-bit integer, if you have selected to parse as I8 for instance). If an overflow occurs in the first parsing to U64 the parser will return this overflow error.
402790	Missing closing parenthesis: <expression>.	The parser found a left-parenthesis without a matching right-parenthesis, e.g. “(x+2”.
402791	Overflow when parsing <number> as <data type>.	The parser parses all numbers in the expression as EXT, and then casts them if necessary to a lower data type (double, if you have selected to parse as DBL for instance). If an overflow occurs in that type cast the parser will return this overflow error.
402792	Syntax error: <expression>.	Incomplete expression or missing symbol, like “x+” or “7!4”.
402793	Unsupported character in expression: <character> (ASCII: < code>).	A few characters are not supported in expressions, for instance the ASCII control characters with codes 0-31.
402794	Unsupported function name: <function name>.	Any name just before a set of parentheses, which does not translate into a supported function name. Implied multiplication is not supported, thus in “x(y+2)” ‘x’ is assumed to be an (in this case unsupported) function name.
402795	Unsupported operator: <operator name>.	Any character string in the expression that consists of normal operator characters, but does not translate into a supported operator, like “x++56”.
When evaluating		
402796	Function or operator <function/operator name> expected integer argument, found floating-point value: <number>.	Some functions and operators require one or more of their arguments to be integer (see section 10.1 “Operators & Functions” about which ones). For instance the bitwise AND operator “a&b” requires both its arguments a and b to be integer. When evaluating ‘&’ in DBL format the required integer format is I32 for instance, so if either a or b have non-zero fractional parts the evaluator will return this argument error.

402797	Function or operator <function/operator name> expected real argument, found complex value: <number>.	Some functions and operators require one or more of their arguments to be real valued (see section 10.1 "Operators & Functions" about which ones). For instance the bitwise OR operator "a|b" requires both its arguments a and b to be real. When evaluating '|' in CDB format the required real format is I32 for instance, so if either a or b have non-zero imaginary parts the evaluator will return this argument error.
402798	Overflow: <function/operator name>.	The function or operation resulted in numeric overflow.
402799	Overflow when converting <number> to <data type> for function or operator <function/operator name>.	Some functions and operators require one or more of their arguments to be integer within a limited range (see section 10.1 "Operators & Functions" about which ones). For instance the shift-left function "SL(x;y)" when evaluated in SGL mode requires x to be within I16 range and y to be within I8 range. If either is not the evaluator will return this overflow error.
402800	Undefined variable name: <variable name>.	The evaluator found a variable name in the expression that wasn't specified in the 'Variables in expression' input array (and thus didn't have a numeric value associated with it).
402801	Unspecified value for variable <variable name>.	Missing value for a variable name in the 'Variable values' input array. You must supply a numeric value for each variable when evaluating an expression.
402802	VI Register <register name> had no value. Default value used.	(Warning) The VI Register held no value when it was needed for evaluation, thus zero was used instead. In case of a unbuffered VI Register no value had ever been written to it, and in case of a buffered VI Register the buffer was currently empty.
When parsing or evaluating		
402803	Not supported in this mode: <keyword>.	Some keywords are not supported in real mode, in integer mode, or in unsigned mode. See section 10.1 "Operators & Functions" about which keywords are supported when.

Table 3 - Error and warning messages

In addition to the above list the Expression Parser toolset can also return errors from the GPower Math toolset if conditions apply.

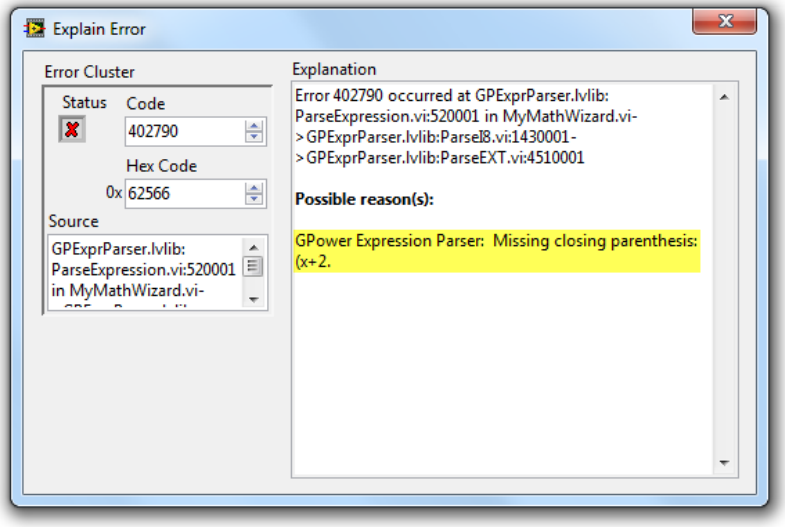


Figure 8 - Error example