Babel: Creating Math Expressions

What is Babel?

Babel is keyboard-based mathematical expression language that will be used to describe math functions during problem formulation. Simple math functions can be described with +, –, *, /, etc. However, interacting with variables and more complex math functions requires the use of subtle and non-obvious operators, functions, and symbols. Babel provides a set of keywords and functions that allow you to describe complex functions and interact with variables.

Usage Overview

Some general characteristics of babel expressions is as follows:

- Similar to many modern programming languages, whitespace characters such as tabs, spaces, and newlines are ignored. This allows the author to format code in the most readable way without changing the meaning of the expression.
- Babel supports the most common operators and functions, such as `+`, `-`, `*`, and `/`. A complete list is provided below, as well as keywords for functions that are non-obvious such as `sin` for the sine function and `ln` for the natural logarithm. A complete list of these functions and operators is in the next four sections.
- Variable names must consist of only letters, number, and underscores. A variable name cannot start with a number. Variables and the names of functions are case sensitive.
- Babel uses IEEE 754 defined 64-bit precision floating point numbers. This means that for most cases it preserves values well, but all mathematic expressions are subject to rounding error, particularly when the domain mixes very large and very small values and transcendental functions.
- Variables may be created and used across multiple statements, using var as the first word and semi-colons to separate multiple lines. The last line must be an expression, not a statement.
- Babel expressions are evaluated according to common mathematic order of operations, with functions taking the same precedence as brackets. It is generally advisable that if any portion of an expression is unclear, the user may add brackets to clarify what should be evaluated first.

- A note on exponentiation: negation has a higher precedence than exponentiation, which is similar to Excel but dissimilar from MATLAB. Thus, `-3^2` is interpreted as `(-3)^2 = 9`, and is NOT interpreted as `-(3^2) = -9`.

Babel expressions are only the right-hand side of a mathematical function. For example:

Figure 1: Babel Expression Interface

Note that because the value `f1` is entered in the *Function Name* text box, it is implied that the value of `f1` is the result of evaluating the expression below for the given values of `x1` and `x2`. Thus, it is both unnecessary and invalid to write `f1 = x1 + cos(x2)`

Multiple Statements

Babel supports the creation of temporary variables and their use in multiple statements.

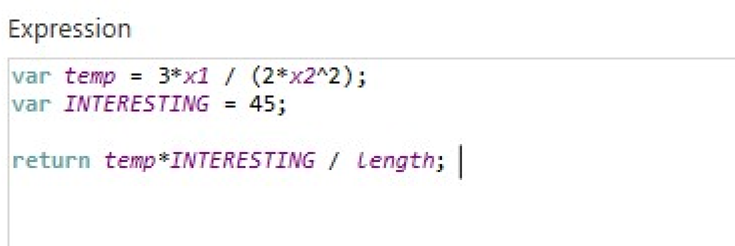


Figure 2 Example of three babel statements

Variables created in this way can be accessed locally within the expression block but not by any other expressions. They cannot be accessed as dynamic variables with `var[expr]`.

Static Variable Access

Variables can be accessed simply by typing their names. For example, assuming you had declared the variables `X1`, `X2`, and `x3` (note the case-sensitivity here), the following are legal babel expressions:

```
    X1
X2 + x3
```

Dynamic Variable Access

In some cases, it may be preferable to access variables as numbered entities in a list rather than by their names. We call this dynamic variable access. Babel exposes this functionality through an implied list or array of variables called `var`. When writing a Babel expression, one can retrieve a value from this list through the index operator: [$expr_I$]. The index expression $expr_I$ is an expression that will retrieve $expr_I$ th variable from the list of variables as ordered in the problem setup screen. Generally, use of var takes the form:

var [$expr_I$]

Where $expr_I$ is an expression that generates an *index* into the list of variables. In the simplest case, $expr_I$ can be a literal integer, like 4, giving var[4], which would be interpreted as the value of the 4th variable declared on the problem setup screen. If the user writes a more complex expression for $expr_I$, the result is rounded to the nearest integer.

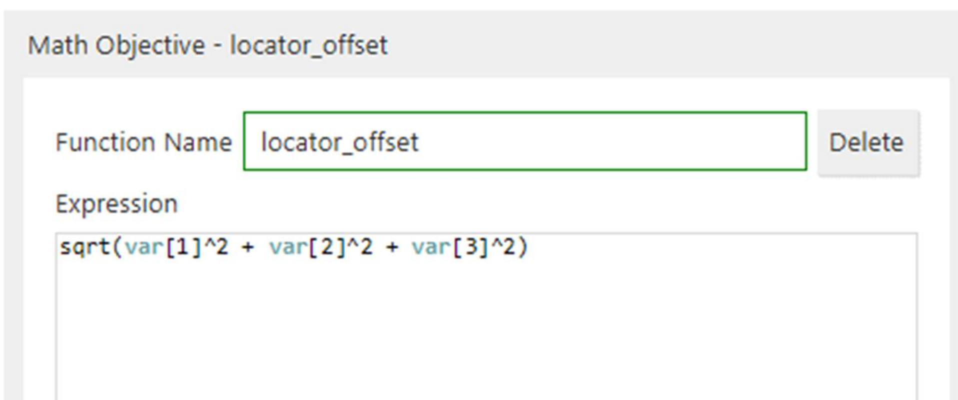
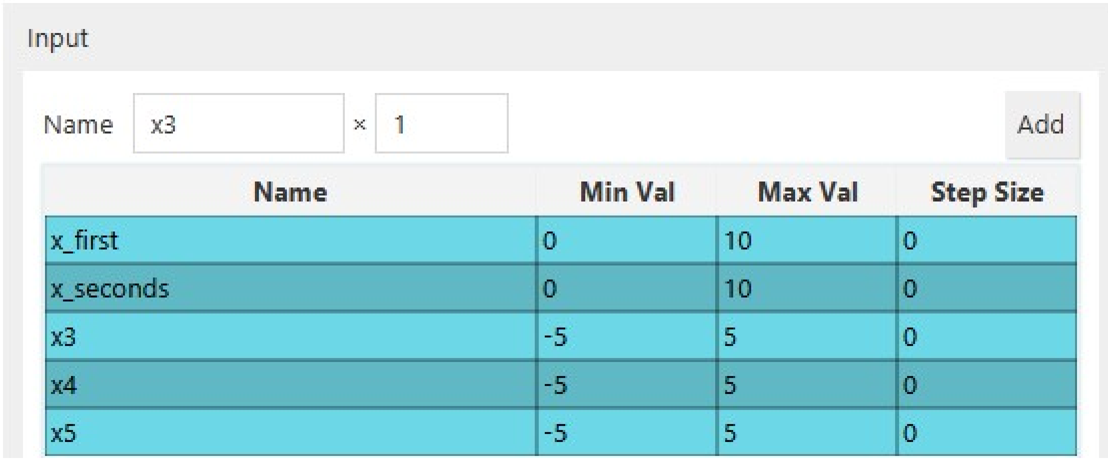


Figure 3: Function Containing a Dynamic Variable Access (var) Expression

Given the following example setup:



Input

Name x3 × 1 Add

Name	Min Val	Max Val	Step Size
x_first	0	10	0
x_seconds	0	10	0
x3	-5	5	0
x4	-5	5	0
x5	-5	5	0

Table 1: Example uses of Var

`var[1]`	Retrieves the value for x_first when being evaluated
`var[3]`	Retrieves the value for x3 when being evaluated
`var[ 1 + 4 ]`	Retrieves the value for x5 when being evaluated. This example illustrates that the value inside the square brackets may be any arbitrary expression.
`var[ceil(ln(x_first)) + 2]`	Retrieves the value: • x3 if x_first is between 0 and 2.718 (e) • x4 if x_first is between 2.718 and 7.38 • x5 if x_first is between 7.38 and 10 This example illustrates that functions may be performed on indexes to achieve some kind of distribution. Care must be taken in making sure the result of the expression is greater than 1 and less than or equal to the count of the variables.

<pre>var [
  min(
    ceil(
      x_first ^ X_second,
    ),
    50
)
]
// one may use whitespace
// to format as they see
fit
// for readability</pre>
	Retrieves the value:
	• x_first if x_first raised to X_second is less than or equal to 1.0 (given the bounds of the variables it is not possible for (x_first)$^{X\_Second}$ to be less than 0).
	• X_second if x_first raised to X_second is between 1.0 and 2.0
	• x3 if x_first raised to X_second is between 2.0 and 3.0
	• (theoretical) x50 if x_first raised to X_second any value greater than 49.

It is illegal to write var[0], or var[*expr*] where *expr* evaluates to a value that is less than 1 or greater than the number of variables declared in the problem setup.

The main use case for var *is for* sum *and* prod, *see* **Loops with `sum` and `prod`**

Binary Operators

Many of the common binary operators are supported within babel.

Table 2: list of binary operators

$expr_L$ + $expr_R$	Addition, evaluates to the sum of $expr_L$ and $expr_R$
$expr_L$ - $expr_R$	Subtraction, evaluates to the difference between $expr_L$ and $expr_R$
$expr_L$ * $expr_R$	Multiplication, evaluates to the product of $expr_L$ and $expr_R$
$expr_L$ / $expr_R$	Division, evaluates to the quotient of $expr_L$ and $expr_R$
$expr_L$ ^ ($expr_{R1}$) $expr_L$ ^ $expr_{R2}$	Exponentiation, evaluates to the $expr_L$ raised to the power of $expr_R$. Note 1: Exponentiation of decimal values is supported. Negative base numbers raised to decimal exponentiation will trigger the error flow resulting in `NaN`. Note 2: Negation of $expr_L$ as a higher precedence than exponentiation, meaning -3^2 is interpreted as (-3)^2 which is 9.
$expr_L$ % $expr_R$	Modulus, the remainder from dividing $expr_L$ by $expr_R$
$expr_L$ < $expr_R$	Strictly-less, true if $expr_L$ is less than (but not equal to) $expr_R$
$expr_L$ <= $expr_R$	Less-or-equal, true if $expr_L$ is less than or equal to $expr_R$
$expr_L$ > $expr_R$	Strictly-greater, true if $expr_L$ is greater than (but not equal to) $expr_R$.
$expr_L$ >= $expr_R$	Greater-or-Equal, true if $expr_L$ is greater than or equal to $expr_R$

Note that the boolean (true/false) operations are only supported for constraint expressions and only at the top-level. Thus, the expression `x1 > (x2 + 2)` is valid for a constraint function, but the expression `(x1 > x2) + 2` is not valid in any context.

Binary Functions

Babel supports two binary functions:

Table 3: List of Binary Functions

min($expr_L$, $expr_R$)	The lowest value resulting from $expr_L$ or $expr_R$
max($expr_L$, $expr_R$)	The highest value resulting from the $expr_L$ or $expr_R$
log($expr_B$, $expr_N$)	The logarithm of $expr_N$ with base $expr_B$

Unary Operators

Babel supports negation as a unary operator

Table 4: List of Unary Operators

`-`*`expr`*	The negated value of the expression, equivalent to `-1 * `*`expr`*

Unary Functions

Many common math concepts are expressed as unary functions in Babel.

Table 5: List of Unary Functions

`sin(`*`expr`*`)`	the sine value of *`expr`*, with *`expr`* assumed to be in radians
`cos(`*`expr`*`)`	the cosine value of *`expr`*, with *`expr`* assumed to be in radians
`tan(`*`expr`*`)`	the tangent value of *`expr`*, with *`expr`* assumed to be in radians
`asin(`*`expr`*`)`	the arc-sine value of *`expr`*, with *`expr`* assumed to be in radians
`acos(`*`expr`*`)`	the arc-cosine value of *`expr`*, with *`expr`* assumed to be in radians
`atan(`*`expr`*`)`	the arc-tangent value of *`expr`*, with *`expr`* assumed to be in radians
`sinh(`*`expr`*`)`	the hyperbolic tangent value of *`expr`*
`cosh(`*`expr`*`)`	the hyperbolic cosine value of *`expr`*
`tanh(`*`expr`*`)`	the hyperbolic tangent value of *`expr`*
`cot(`*`expr`*`)`	the cotangent value of *`expr`*, with expr assumed to be in radians
`ln(`*`expr`*`)`	the natural logarithm of *`expr`* (log base e of *`expr`*)
`log(`*`expr`*`)`	the decimal logarithm of *`expr`* (log base 10 of *`expr`*)
`abs(`*`expr`*`)`	the absolute value of *`expr`*
`sqrt(`*`expr`*`)`	the square root of *`expr`*
`cbrt(`*`expr`*`)`	the cube root of *`expr`*
`sqr(`*`expr`*`)`	the value of *`expr`* to the power of 2
`cube(`*`expr`*`)`	the value of *`expr`* to the power of 3
`ceil(`*`expr`*`)`	the lowest integer value greater than *`expr`*
`floor(`*`expr`*`)`	the highest integer value lower than *`expr`*
`sign(`*`expr`*`)`	the sign of *expr*

Loops with sum **and** prod

It is often desirable to take the sum or product of a set of values, applying a uniform function to each value first. This is done frequently in mathematics and typically denoted by capitol sigma and capitol pi.

For example:

$$\sum_{i=1}^{50}[(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2]$$

We can express this in babel as

```
sum(
  1,
  50,
  i -> (
    (var[2*i - 1]^2 – var[2*i])^2
    + (var[2*i-1] - 1)^2
  )
)
```

The two functions within babel that perform aggregation (ie ‘loops’) are `sum` and `prod`:

Table 6: sum and prod details

sum(*expr*$_L$, *expr*$_U$, *indexAlias* -> *expr*$_A$)	Adds the results from running *expr*$_A$ once for each integer value *indexAlias* in the inclusive range [*expr*$_L$, *expr*$_U$] together.
prod(*expr*$_L$, *expr*$_U$, *indexAlias* -> *expr*$_A$)	Multiplies the results from running *expr*$_A$ once for each integer value *indexAlias* in the inclusive range [*expr*$_L$, *expr*$_U$] together.
expr$_L$ is the lower bound expression, and *expr*$_U$ is the upper bound expression	

For each evaluation of *$expr_A$*, a new variable with the name specified in *indexAlias* is given a value in the range. The *variable* (**not** expression) defined in *indexAlias* is only available to *$expr_A$*, and must follow the variable name guidelines. In this sense, `sum` and `prod` allow us to specify a new temporary variable that may be given many different values on any one objective or constraint evaluation.

A typical value for *indexAlias* is simply the `i`, for example:

```
sum(1, 10, i -> x1 * i)
```

indexAlias can be any valid variable name. It is often best to pick a name that makes sense in the context of the model.

EG, if the 7th through 12th variables were TRUSS lengths, it might make sense to use the variable TRUSS_ID:

```
prod(7, 12, TRUSS_ID ->
  ciel(var[TRUSS_ID] * sqrt(desity_var))
)
```

It can also be a mathematically-understood unicode text such as β, for example:

```
sum(1, ceil(sqrt(target)), β -> x1 + var[ciel(sqrt(β)) + x4]))
```

It is illegal to write `sum(5, 4, …)` or, more generally, sum(*$expr_L$*, *$expr_U$*, ...) where *$expr_L$* evaluates to a value that is less than *$expr_U$*.

It is legal to write a sum or product that is expressed over a zero range, that is a range in which the upper and lower bounds are equal.

- `sum(`*$expr_L$*`,` *$expr_U$*`,` *indexAlias* -> *$expr_A$*`)`, where *$expr_L$* is equal to *$expr_U$*, (eg `sum(x1 + 2, 2 + x1, …)`), and will evaluate to the value 0.0.
- Similarly, `prod`*(exprL*`,` *$expr_U$*`,` *indexAlias* -> *$expr_A$*`)` (eg `prod(5, 5, …)`) will evaluate to 1.0.

Example: Arithmetic Series

Mathematics:

$$\sum_{i=1}^{10} i$$

Babel with sum/prod:

```
sum(1, 10, i -> i)
```

Long form Babel:

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

Example: Geometric Series

Mathematics:

$$\sum_{nextTerm=}^{\lceil \log_2 highValu \quad \rceil} \frac{1}{2^{nextTerm}}$$

Babe with sum/prod:

```
sum(1, ceil(ln(highValue)/ln(2)), nextTerm -> 1 / 2^nextTerm)
```

Long form Babel:

```
1 / 2^1 + 1/2^2 + 1/2^3 + [...] + 1/2^(ceil(ln(highValue)/ln(2)) - 1) +
1/2^(ciel(ln(highValue)/ln(2)))
```

Known Constants

There are some constants which are already in the Babel library, so a user doesn't have to type out their value manually.

Table 8: List of Constants

pi	3.141592653589793
e	2.718281828459045