

Developer

/ Documentation

/ Unreal Engine ▾

/ Unreal Engine 5.4 Documentation

/ Programming and Scripting

/ Blueprints Visual Scripting

/ Specialized Blueprint Node Groups

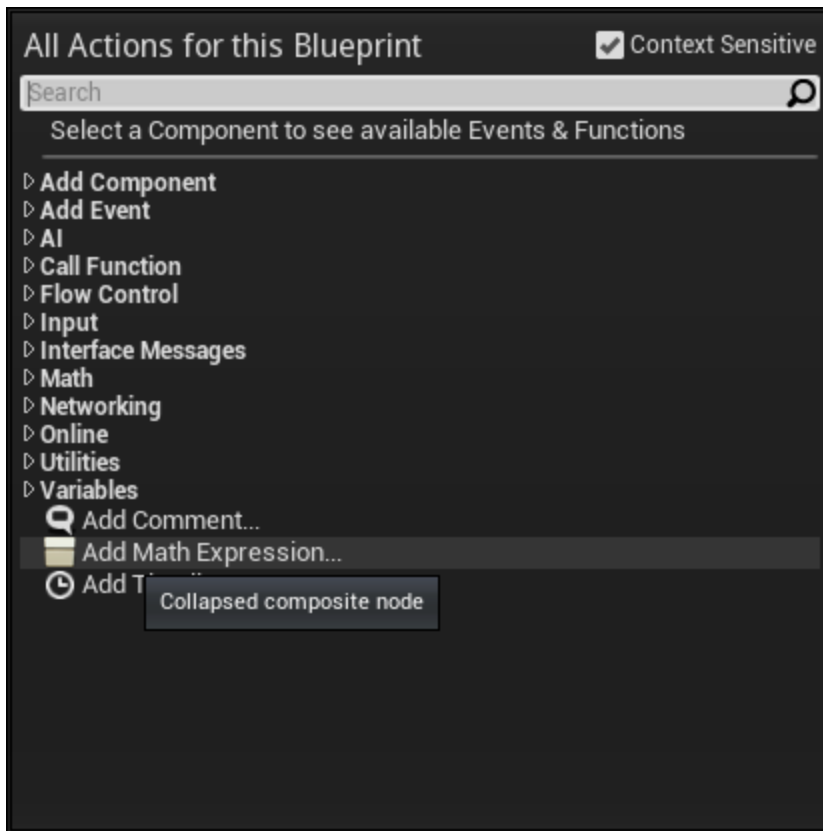
/ Math Expression Node

Math Expression Node

The math expression node allows you to type in a math expression and builds the appropriate sub-graph to create that expression.



To create a Math Expression node, **Right-click** in the graph and select **Add Math Expression...** from the context menu.



The Math Expression node acts like a collapsed graph. It is a single node that you can **Double-click** to open the sub-graph that makes up its functionality. Initially, the name/expression is blank. Whenever you rename the node, then the new expression is parsed and a new sub-graph is generated.

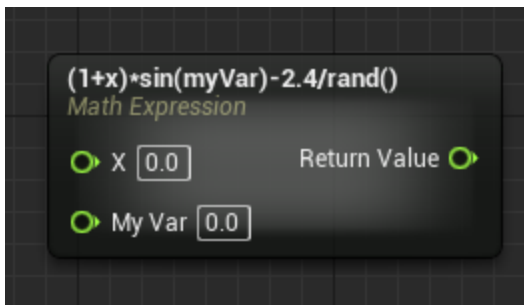
To rename the node and update the sub-graph, **Right-click** on the node and select **Rename**.

For example, you could rename the Math Expression node with the following expression:

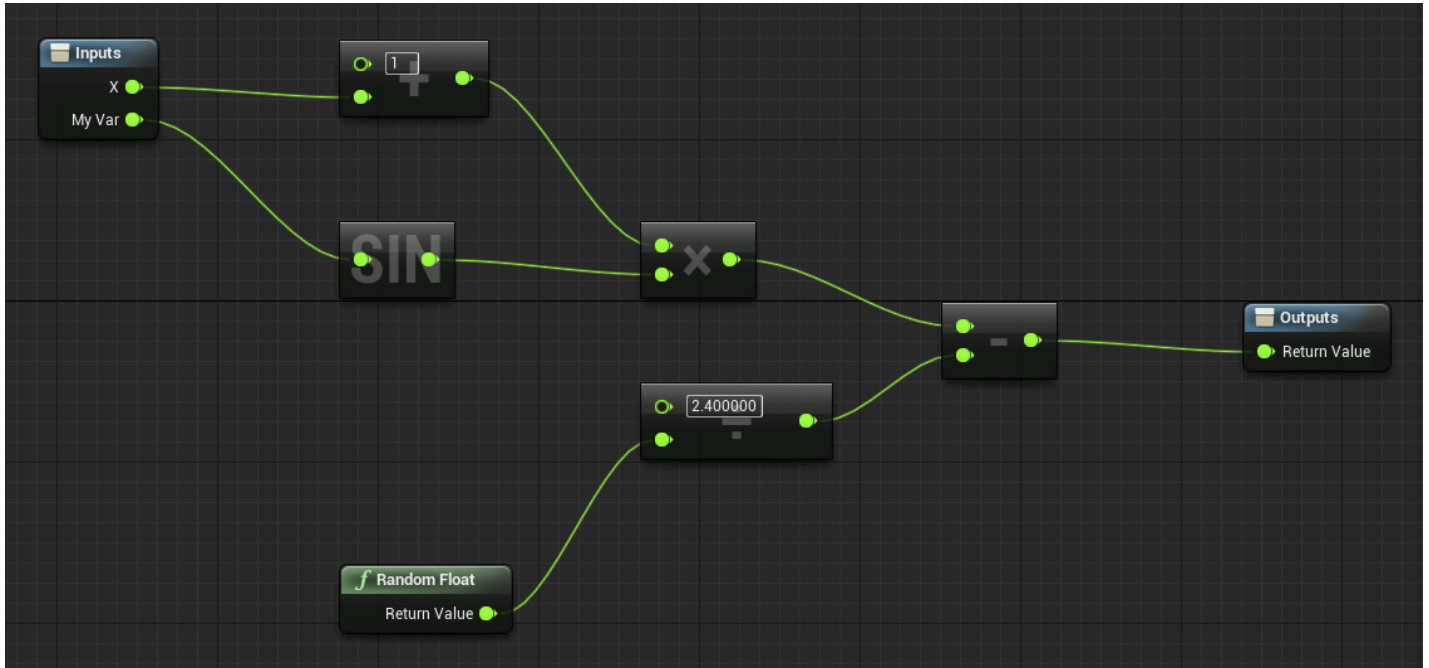
```
1 (1+x)*sin(myVar)-2.4/rand()  
2
```

 Copy full snippet

This would update the Math Expression node to have two float inputs, **X** and **MyVar**, and one float output.



If you **Double-click** the node with this expression, you would see the below sub-graph:



The sub-graph is created by following the below rules:

1. Alphabetic names (tokens beginning with a letter) should be turned into either variables, input pins on the Math Expression node, or function calls. From the example above: **x**, **sin**, **myVar**, and **rand**.
 - a. If an alphabetic name matches an existing variable in the Blueprint, then it will become a variable get node (i.e. if the Blueprint already had a **myVar** variable).
 - b. If an alphabetic name is followed by an open parenthesis, like **sin(** and **rand(**, then it will be turned into a function node.
 - c. If the previous two states are not met, then the alphabetic name is turned into a float input on the Math Expression node.
 - d. The name will also be used as an input if it matches the name of an existing pin on the node.
2. Numerical constants are always turned into pin inputs. They never create a node themselves, but instead are used to fill out input fields on other nodes.
3. Mathematical operators (like +, *, -) are turned into function nodes.

4. Like in mathematics, expressions inside of parentheses will take precedence, and be evaluated first.

Variables

Variable naming is fairly flexible, but it is important to remember the following points:

- Variables can have numbers in them, but they cannot start with a number.
- Variables cannot have the same name as a hidden Blueprint variable.
- Be sure that you are working with the correct type of variable. For example, **boolVar+1.5** is not a valid expression.

Order of Operations

Operations follow the below order of operations (highest priority first):

1. Parentheses
2. Factorials
3. Exponents
4. Multiplication and division
5. Addition and subtraction

Available Blueprint Functions

Blueprint pure functions that are in a coded function library should all be available. These include:

- Trig functions (sin/cos/tan, asin, acos, etc.):

```
sin(x)
```

 Copy full snippet

- Clamping functions (min, max, clamp, etc.):

```
clamp(y,0,20)
```

 Copy full snippet

- Rounding functions (floor, fceil, round, etc.):

```
round(z)
```

 Copy full snippet

- Exponential functions (square, power, sqrt, loge, e/exp, etc.)

Certain functions have multiple names that are commonly used. To account for that, there are several aliases that map to the same function/node. Some examples are:

- Power aliases: power, pow
- Trig arc functions (asin/arcsin, acos/arccos, etc.)




Since you are typing in functions rather than connecting pins, make sure to enter the correct number and type of parameters. The Math Expression node will display an error if there is a mismatch.

Basic Struct Types

There are some basic structures that we use quite often in math as well: vectors, transforms, etc. You can make and operate on these types in the expression easily.

- The vector keyword spawns a MakeVector node:

```
vector(x,y,z)
```

 Copy full snippet

- The rotator keyword spawns a MakeRotator node:

```
rotator(x,y,z)
```

 Copy full snippet

- The transform keyword spawns a MakeTransform node:

```
1 transform(vec(x,y,z), rot(p,y,r), vec(6,5,4))  
2
```

 Copy full snippet

Just like there are aliases for certain functions, there are vector and rotator aliases.


- Vector aliases

```
vector, vec, vect
```

 Copy full snippet

- Rotator aliases

```
1 rotator, rot  
2
```

 Copy full snippet

Supported Operators

The following operators should all be supported, and the logical and comparison operators can be combined to create complex expressions.

- Multiplicative:

```
*, /, %
```

 Copy full snippet


- Additive:

```
+, -
```

 Copy full snippet

- Relational:

```
<, >, <=, >=
```

 Copy full snippet

- Equality:

```
==, !=
```

 Copy full snippet

- Logical:

```
1 ||, &&, ^  
2
```

 Copy full snippet

Unsupported Operators

Currently, the following operators are unsupported.

- Conditional:

```
?:
```

 Copy full snippet

- Bitwise:

```
~, &, <<, >>
```

 Copy full snippet

- Unary prefix:

```
+, -, ++, --, ~, !, etc.
```

 Copy full snippet

- Postfix:

```
++, --, [], etc.
```

 Copy full snippet