# Maya Math Nodes Documentation

**Serguei Kalentchouk**

**Sep 27, 2018**

# User Documentation

1 Node Reference 1

2 Expression Language 9

CHAPTER 1

# Node Reference

## 1.1 Overview

The nodes are designed with the following principles in mind:

- nodes perform a single operation
- nodes have a single output attribute
- nodes are strongly typed

---

**Note:** In order to achieve consistency and streamlined workflow, there are a few nodes that duplicate existing Maya functionality.

---

The node library tries to adhere to the following set of rules when it comes to choosing the node and attribute names:

- node names are prefixed with `math_`
- nodes are named with affirmative action verbs, ex: `Add`, `Multiply`
- the *get* action verb is implied, ex: `GetDotProduct` is `DotProduct`
- nodes are assumed to operate on doubles by default, ex: `Add`, `Multiply`
- mixed type operations are reflected in the name, ex: `AddVector`, `MultiplyVectorByMatrix`
- conversion nodes have following format *OutputFromSource*, ex: `RotationFromMatrix`
- attributes are generally named `input` and `output`
- if multiple inputs are required they are enumerated, ex: `input1`, `input2`
- for clarity other attribute names are allowed, ex: `translation`, `alpha`, `axis`, `min`

## 1.2 Node List

### 1.2.1 Absolute

**description** Computes absolute value

**type variants** AbsoluteAngle, AbsoluteInt

**expression** abs(x)

### 1.2.2 Acos

**description** Computes arccosine

**expression** acos(x)

### 1.2.3 Add

**description** Computes sum of two values

**type variants** AddAngle, AddInt, AddVector

**expression** x + y

### 1.2.4 AndBool

**description** Gets logical *and* of two values

**type variants** AndInt

### 1.2.5 AngleBetweenVectors

**description** Computes angle between two vectors

**expression** anglebetween(x, y)

### 1.2.6 Asin

**description** Computes arcsine

**expression** asin(x)

### 1.2.7 Atan

**description** Computes arctangent

**expression** atan(x)

### 1.2.8 Atan2

**description** Computes arctangent of *x / y*

**expression** atan(x, y)

### 1.2.9 Average

**description** Computes average value

**type variants** AverageAngle, AverageInt, AverageMatrix, AverageQuaternion, AverageRotation, AverageVector

### 1.2.10 AxisFromMatrix

**description** Gets basis vector from matrix for a given axis

**expression** axis(x, axis)

### 1.2.11 Ceil

**description** Computes the smallest integer value greater than or equal to input

**type variants** CeilAngle

**expression** ceil(x)

### 1.2.12 Clamp

**description** Computes the value within the given min and max range

**type variants** ClampAngle, ClampInt

**expression** clamp(x, min, max)

### 1.2.13 Compare

**description** Compute how the two values compare to each other

**type variants** CompareAngle

**expression** compare(x, y)

### 1.2.14 CosAngle

**description** Computes the cosine of angle

**expression** cos(x)

### 1.2.15 CrossProduct

**description** Computes the cross product of two vectors

**expression** cross(x, y)

### 1.2.16 Divide

**description** Computes the quotient of two values

**type variants** DivideAngle, DivideAngleByInt, DivideByInt

**expression** x / y

### 1.2.17 DotProduct

**description** Computes the dot product of two vectors

**expression** dot(x, y)

### 1.2.18 DistancePoints

**description** Computes the distance between two points or matrices

**type variants** DistanceTransforms

**expression** distance(x, y)

### 1.2.19 Floor

**description** Computes the largest integer value less than or equal to input

**expression** floor(x)

### 1.2.20 InverseMatrix

**description** Computes the inverse of value

**type variants** InverseQuaternion, InverseRotation

**expression** inverse(x)

### 1.2.21 Lerp

**description** Computes linear interpolation between two values

**type variants** LerpAngle, LerpMatrix, LerpVector

**expression** lerp(x, y, alpha)

### 1.2.22 MatrixFromTRS

**description** Computes a matrix from translation, rotation and scale

### 1.2.23 Max

**description** Gets the largest of the two values

**type variants** MaxAngle, MaxInt

**expression** max(x, y)

### 1.2.24 MaxElement

**description** Gets the largest value in array

**type variants** MaxAngleElement, MaxIntElement

**expression** maxelement(x, y)

### 1.2.25 Min

**description** Gets the smallest of the two values

**type variants** MaxAngle, MaxInt

**expression** min(x, y)

### 1.2.26 MinElement

**description** Gets the smallest value in array

**type variants** MinAngleElement, MinIntElement

**expression** minelement(x, y)

### 1.2.27 ModulusInt

**description** Computes the remainder of the two values

**expression** x % y

### 1.2.28 Multiply

**description** Computes the product of two values

**type variants** MultiplyAngle, MultiplyAngleByInt, MultiplyByInt, MultiplyInt, MultiplyMatrix, MultiplyQuaternion, MultiplyRotation, MultiplyVector, MultiplyVectorByMatrix

**expression** x * y

### 1.2.29 Negate

**description** Computes the negation of value

**type variants** NegateAngle, NegateInt, NegateVector

**expression** negate(x)

### 1.2.30 NormalizeVector

**description** Computes normalized vector

**expression** normalize(x)

### 1.2.31 NormalizeArray

**description** Normalize array of values

**expression** normalizearray(x)

### 1.2.32 NormalizeWeightsArray

**description** Normalize array of weight values

### 1.2.33 OrBool

**description** Gets logical *or* of two values

**type variants** OrInt

### 1.2.34 Power

**description** Computes the value raised to power of the exponent

**expression** power(x, exp)

### 1.2.35 QuaternionFromMatrix

**description** Gets quaternion from matrix or rotation

**type variants** QuaternionFromRotation

**expression** quat(x, rot_order)

### 1.2.36 Round

**description** Computes rounded value

**type variants** RoundAngle

**expression** round(x)

### 1.2.37 RotationFromMatrix

**description** Gets rotation from matrix or quaternion

**type variants** RotationFromQuaternion

**expression** rot(x, rot_order)

### 1.2.38 ScaleFromMatrix

**description** Gets scale from matrix

### 1.2.39 Select

**description** Toggles output

**type variants** SelectAngle, SelectInt, SelectMatrix, SelectQuaternion, SelectRotation, SelectVector

**expression** select(x, y, state)

### 1.2.40 SelectArray

**description** Toggles array output

**type variants** SelectAngleArray, SelectIntArray, SelectMatrixArray, SelectVectorArray

### 1.2.41 SinAngle

**description** Computes sin of angle

**expression** sin(x)

### 1.2.42 SlerpQuaternion

**description** Comptues slerp interpolation between two quaternions

**expression** slerp(x, y)

### 1.2.43 Subtract

**description** Computes the difference between two values

**type variants** SubtractAngle, SubtractInt, SubtractVector

**expression** x - y

### 1.2.44 Sum

**description** Computes the the sum of values

**type variants** SumAngle, SumInt, SumVector

### 1.2.45 TanAngle

**description** Computes tangent of angle

**expression** tan(x)

### 1.2.46 TranslationFromMatrix

**description** Get translation from matrix

### 1.2.47 TwistFromMatrix

**description** Computes twist around axis from matrix or rotation

**type variants** TwistFromRotaiton

**expression** twist(x, axis, rot_order)

### 1.2.48 VectorLength

**description** Computes length of vector

**expression** length(x)

### 1.2.49 VectorLengthSquared

**description** Computes squared length of vector

**expression** lengthsquared(x)

### 1.2.50 WeightedAverage

**description** Computes the weighted average value

**type variants** WeightedAverageAngle, WeightedAverageInt, WeightedAverageMatrix, WeightedAverageQuaternion, WeightedAverageRotation, WeightedAverageVector

### 1.2.51 XorBool

**description** Gets logical *xor* of two values

**type variants** XorInt

CHAPTER 2

# Expression Language

## 2.1 Overview

Even simple math expressions often require relatively large node networks, which are tedious to create by hand. While this process can be scripted, the code is likewise tedious to write and makes it difficult to see the logic at a glance.

To help alleviate these issues, Maya Math Nodes plugin provide a simple expression language that can be used to describe a series of mathematical operations inline, which can then be interpreted to generate a math node network for you. For example:

```
# project vector to plane
eval_expression('node.t - (vec(0, 1, 0) * dot(node.t, vec(0, 1, 0)))', 'projectToPlane
↪')
```

## 2.2 Data Types

The language supports the following data types:

**numeric** float and int types are supported, ex: `-1, 0, 1.0`

**string** string literals are used to reference Maya attributes, ex: `node.attribute[0]`, note that there are no quatation marks around the string literals!

**complex** complex types such as vector, matrix, rotation, and quaternion are specified by using cast functions, ex: `vec(0, 1, 0)`

## 2.3 Operators

The language supports a limited set of arithmetic operators: `+, -, *, /, %,`

## 2.4 Conditionals

The language supports the following relational operators: `==`, `!=`, `>`, `<`, `>=`, `<=`

These are used in combination with ternary conditional expression: `a == b ? true : false`

## 2.5 Functions

The language supports calling functions with arguments. These functions map directly to the node operators available in the plugin.

For example `Absolute` node is made available through the `abs()` function call. Please see the *Node Reference* for the mapping between node type and function name.

The function arguments correspond with node attributes. For example the `Clamp` node has two input attributes, therefore the `clamp(arg1, arg2)` function will take two arguments.

Likewise, array arguments are also supported with the following syntax: `minelement([1, 2, 3])`.

Output array arguments can also be index using the `[]` operator.

### 2.5.1 Cast Functions

Several functions that output complex data types can take constant values as input.

**mat** `mat(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1)` can be used to specify constat matrix value

**rot** `rot(0, 1, 0)` can be used to specify constant rotation value, `rot()` also maps to several math nodes and can take other arguments, ex: `rot(node.matrix, 0)`

**quat** `quat(0, 0, 0, 1)` can be used to specify constant quaternion value, `quat()` also maps to several math nodes and can take other arguments, ex: `quat(node.rotation, 0)`

**vec** `vec(1, 0, 0)` can be used to specify a constant vector value

> **Warning:**
>
> Currently, some nodes do not have expression bindings!
> See *Node Reference* section for details.

**Note:** Default and keyword arguments are currently not supported!

## 2.6 Evaluation Order

Expressions are evaluated left to right with the following operator precedence, listed from lowest to highest:

| Operator | Description |
|---|---|
| +, - | Addition and subtraction |
| *, /, % | Multiplication, division, remainder |
| <, <=, >, >=, !=, == | Comparisons |
| func() | Function call |
| (...) | Grouping |

## 2.7 Type Resolution

The operators and functions are mapped to specific Maya nodes shipped with the plugin, and because the node library is strongly typed the parser needs to make a determination about types using the following rules:

- for operators, the left operand is used to determine primary type
- for conditional expressions, the true value is used to determine primary selector type
- for functions, the first argument is used to determine primary type
- if operand or argument is literal numeric type then casting to another numeric type is allowed

## 2.8 Name Generator

The expression evaluator will create Maya nodes procedurally and therefore needs a mechanism to generate unique names consistently.

This is achieved with the `NameGenerator` class. To customize this behavior you can create your own implementation, with the only requirement that it implements `get_name(str: node_type) -> str` method.

## 2.9 Evaluator

The public API for this module consist of a single function:

`eval_expression(str: expression, str: base_node_name='', NameGenerator: name_generator=None) -> str`

The return value is the path to the output attribute of the last node in the generated node network that will have the result value computed for the expression. This value can then be passed to subsequent expressions to chain them together.

## 2.10 Examples

```
from maya_math_nodes import eval_expression

# get twist value for roll joint
eval_expression('twist(ctrl.worldMatrix[0], 0, 0) * 0.5', 'roll')

# get toe pivot value for foot roll
eval_expression('ctrl.roll > ctrl.break ? ctrl.roll - ctrl.break : 0', 'toeroll')
```

(continues on next page)

(continued from previous page)

```
# compute some pole vector with offset
eval_expression('cross(axis(ctrl.matrix, 0), vec(0, 1, 0)) * 2', 'pole')
```

Maya Math Nodes is a plugin for Autodesk Maya that provides a set of atomic nodes to perform various common math operations. The purpose of these nodes is to streamline the creation of complex and highly performant rigging systems.

To see the list of nodes made available by the plugin, please refer to the *Node Reference* section.

Additionaly, this plugin provides a simple expression language that can be used to describe a series of mathematical operations inline, which can then be interpreted to generate a math node network for you, see *Expression Language* section for details.

---

**Note:** At this time there are no distributable binaries available for download. However, it is fairly easy to build it directly from the source code.

---