McGill School of Architecture
Advanced Construction

Introduction to Grasshopper for Rhinoceros

Maria Mingallon
Msc (Emtech AA) MEng CEng MICE
maria.mingallon@arup.com

Schedule

Tuesday

6.30 - 7.30pm

Introduction
Data Storage:
- **Objects, Scalars, Data Trees**
- Examples: scalar_intervals, scalar_operators, Function_Spiral

7.30 - 8.30pm

Data Storage:
- **Vectors**
- Examples: Unit Vectors & Attractor scaled boxes

Wednesday

6.30 - 7.30pm

Curves (NURBS)
- **Definition & type of curves**
- **Curve Analysis (Curve Domain)**

7.30 - 8.30pm

Surfaces (NURBS)
- **Definition & Surface Analysis**
- **Manipulation of Curves & Surfaces**
- Example: Surface Connect Example & Panelisation of a surface

Thrusday

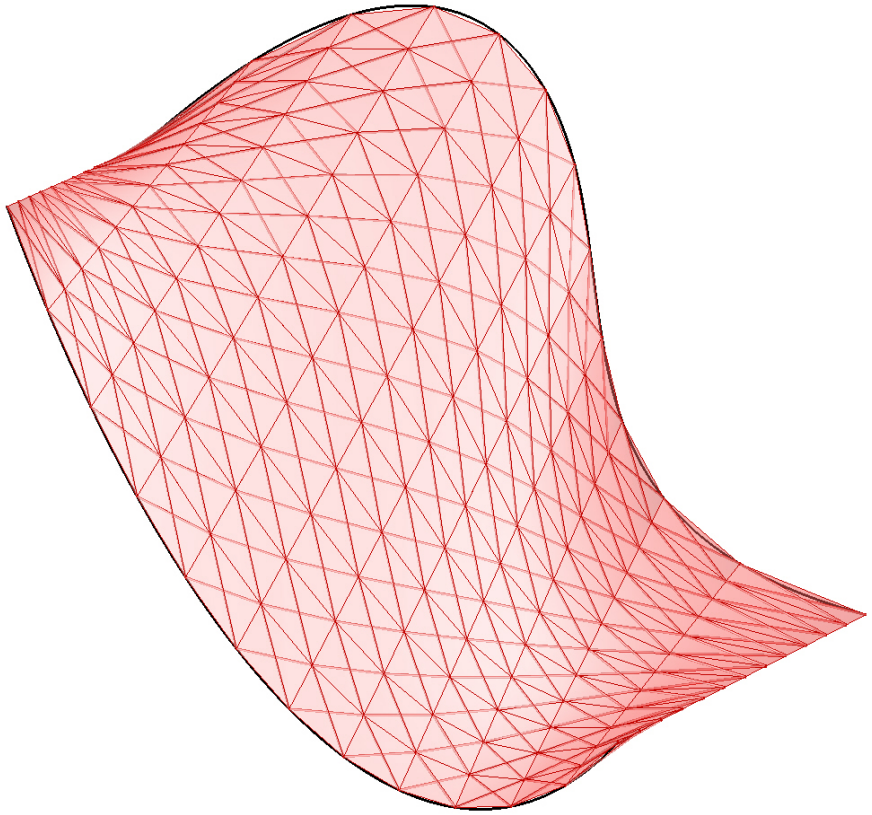
6.30 - 7.30pm

Surfaces (NURBS):
- **Manipulation of Curves & Surfaces II:**
- Example: Panelisation with non-uniform grids

7.30 - 8.30pm

Vb.net Introduction:
- **Introduction**
- Example: Hexagonal Grid Panelisation

What can Grasshopper do for you?



Surface DiaGrid

Paneling a Surface

1. Data Storage

1.1 Objects

a) Params:

- Import and store data
- Used to introduced objects previously created in Rhino Interface

e.g. point, surface, plane, curve, etc

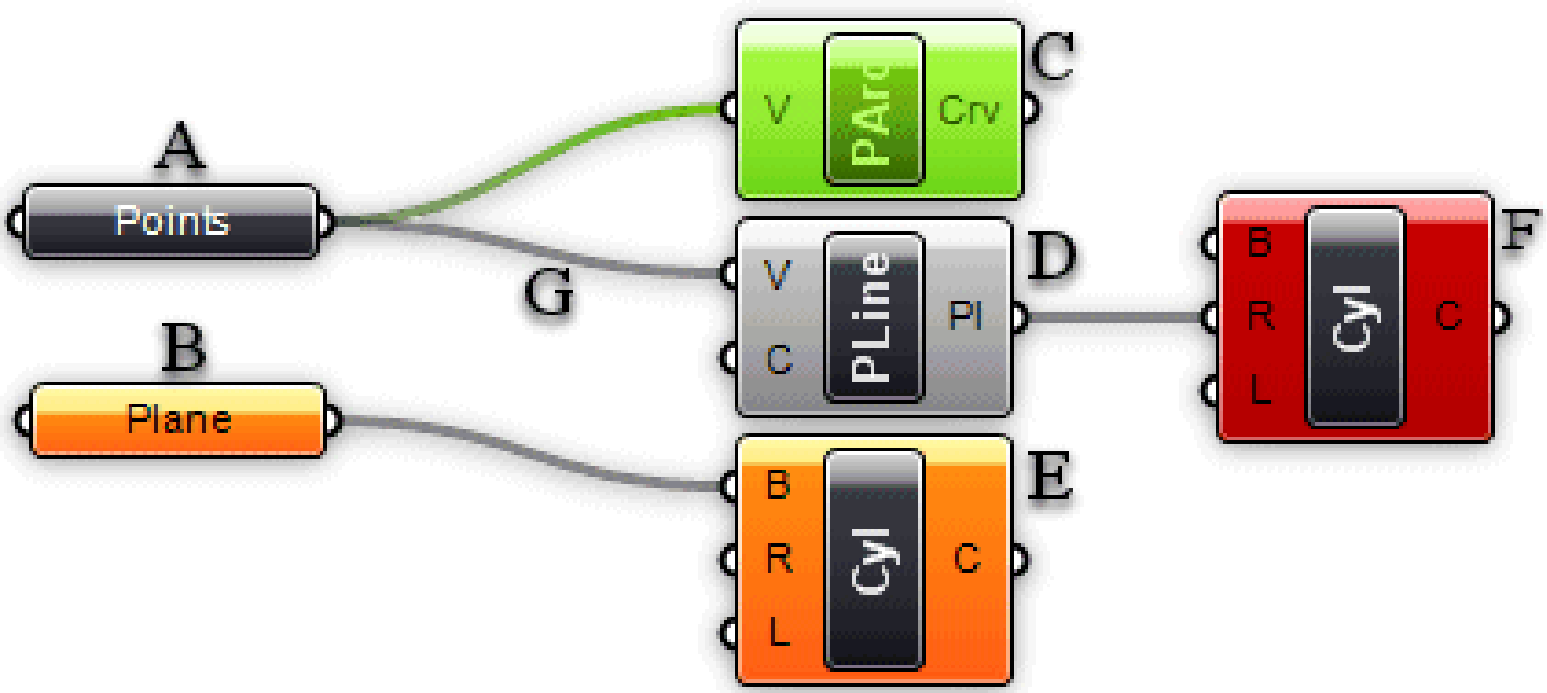
b) Components:

- Generate and store data
- Created within Grasshopper

e.g. logic, scalar, vector, curve, surface, mesh, etc

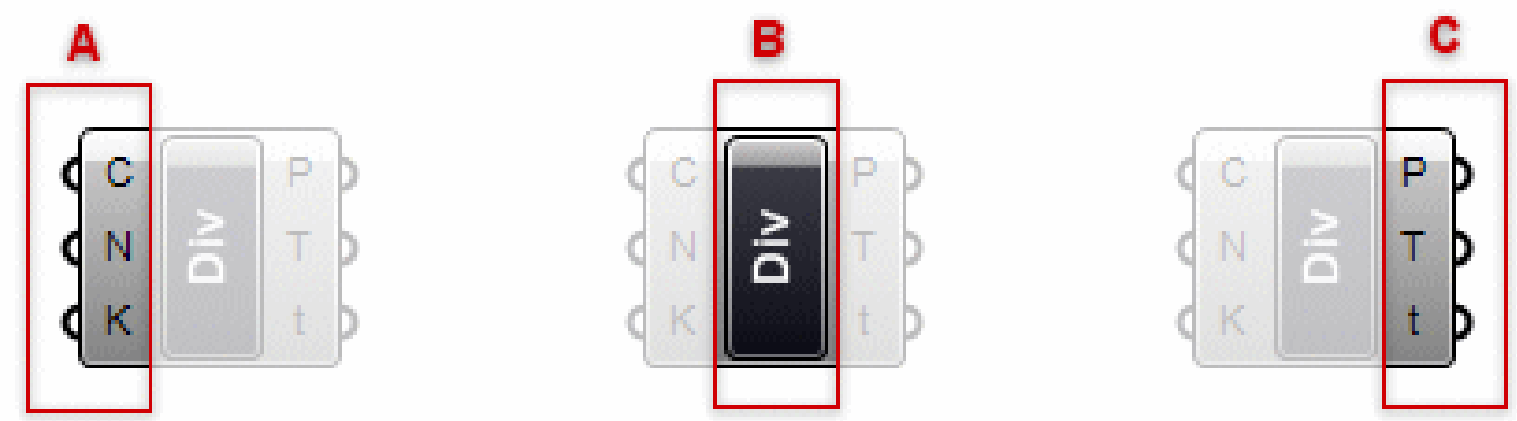
1. Data Storage

1.1 Objects



1. Data Storage

1.1 Objects



A: Input parameters
B: Name of Component
C: Output parameters

1. Data Storage

1.2 Scalars

A) Constants: Used to represent a constant value (e.g. Pi, Golden Ratio, etc).

B) Operators: used for mathematical operations such as Add, Subtract, Multiply, etc. Example: scalar_operators.ghx

C) Intervals: used to divide numeric extremes (or domains) into interval parts. Apart from Intervals, Ranges and Series are included within this group. Example: scalar_intervals.ghx

D) Utility: evaluate two or more numerical values. Examples: conditional_statements.ghx and If_Elsetest.ghx

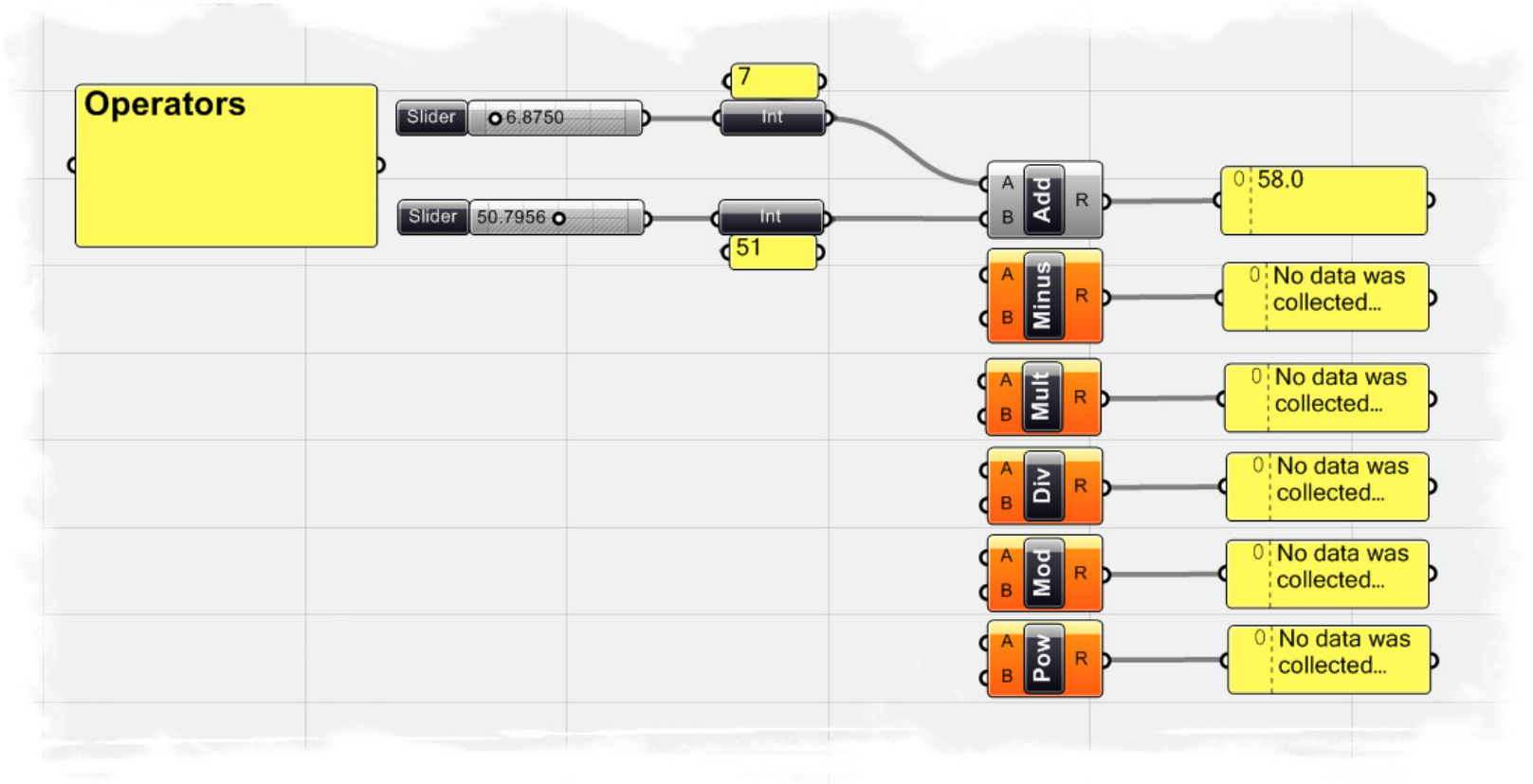
- Conditional Statements: Equality, Similarity, Larger than, Smaller than
- Booleans: 'If/Else' statements in grasshopper

E) Polynomials: used to define a polynomial function. Example: Function_Spiral.ghx

F) Trigonometry: used to define a trigonometric function such as sine, cosine, tangent, etc. Example: Trigonometric_curves.ghx

1. Data Storage

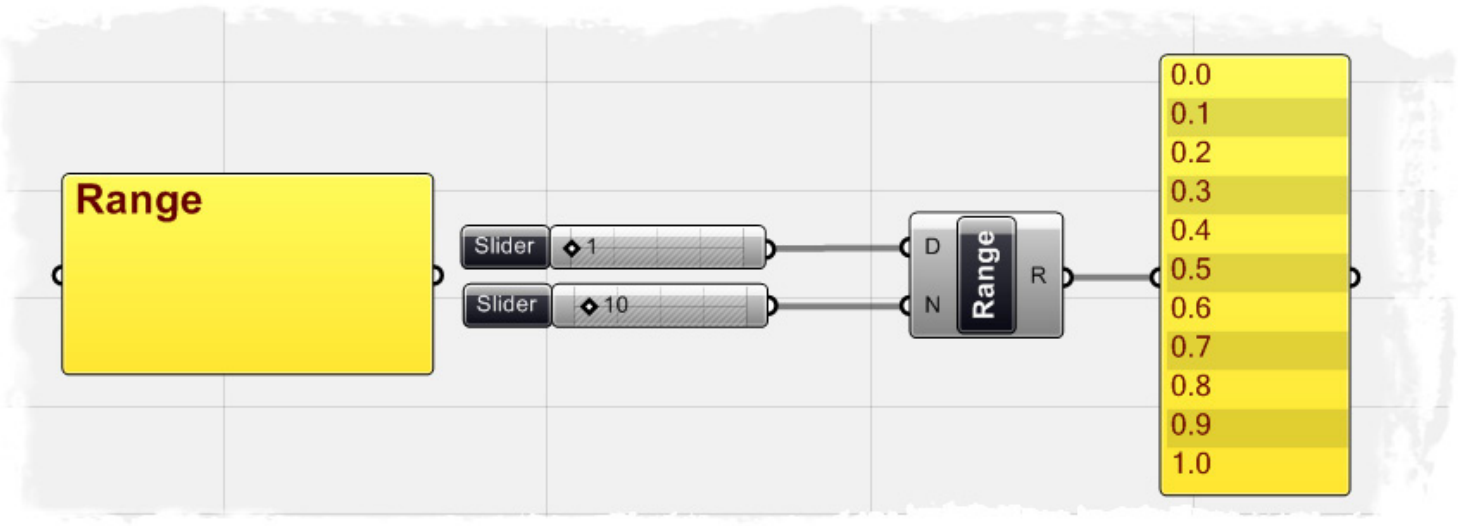
1.2 Scalars :
(B) Operators

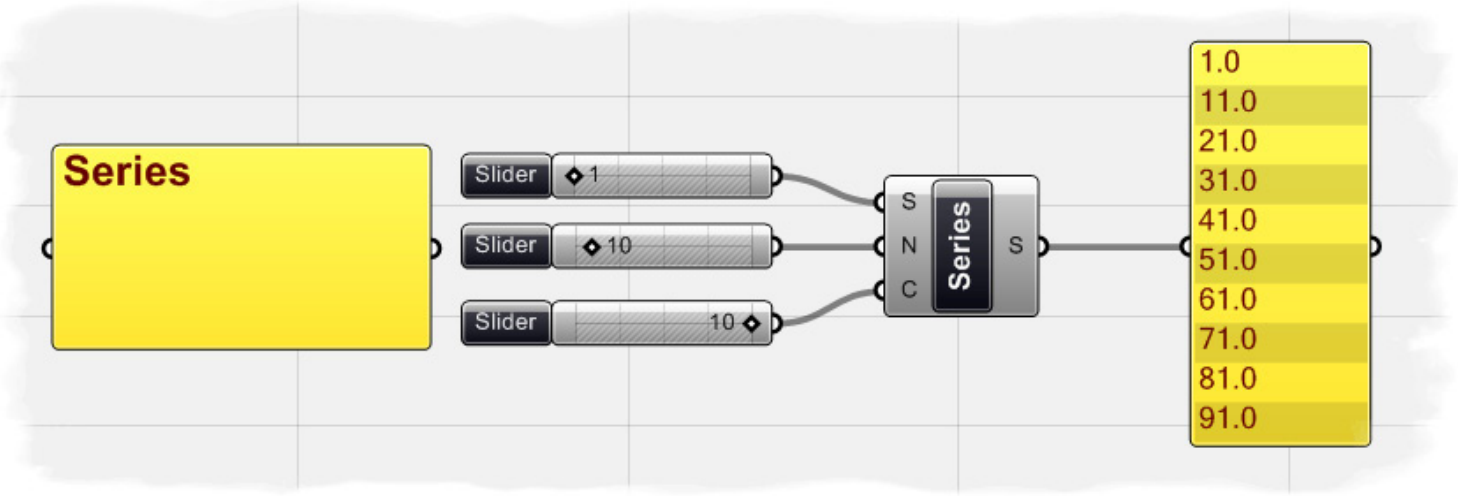


1. Data Storage

1.2 Scalars :

(C) Intervals
Range
Series

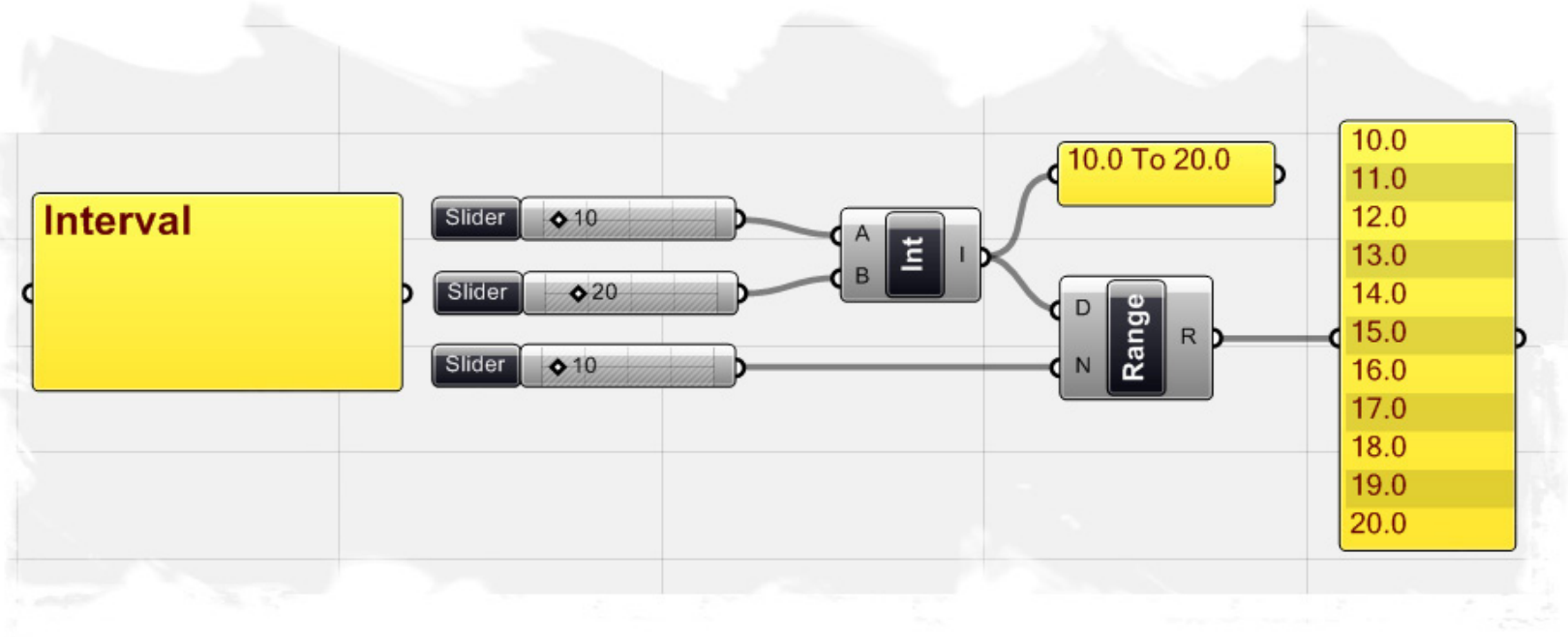




1. Data Storage

1.2 Scalars :

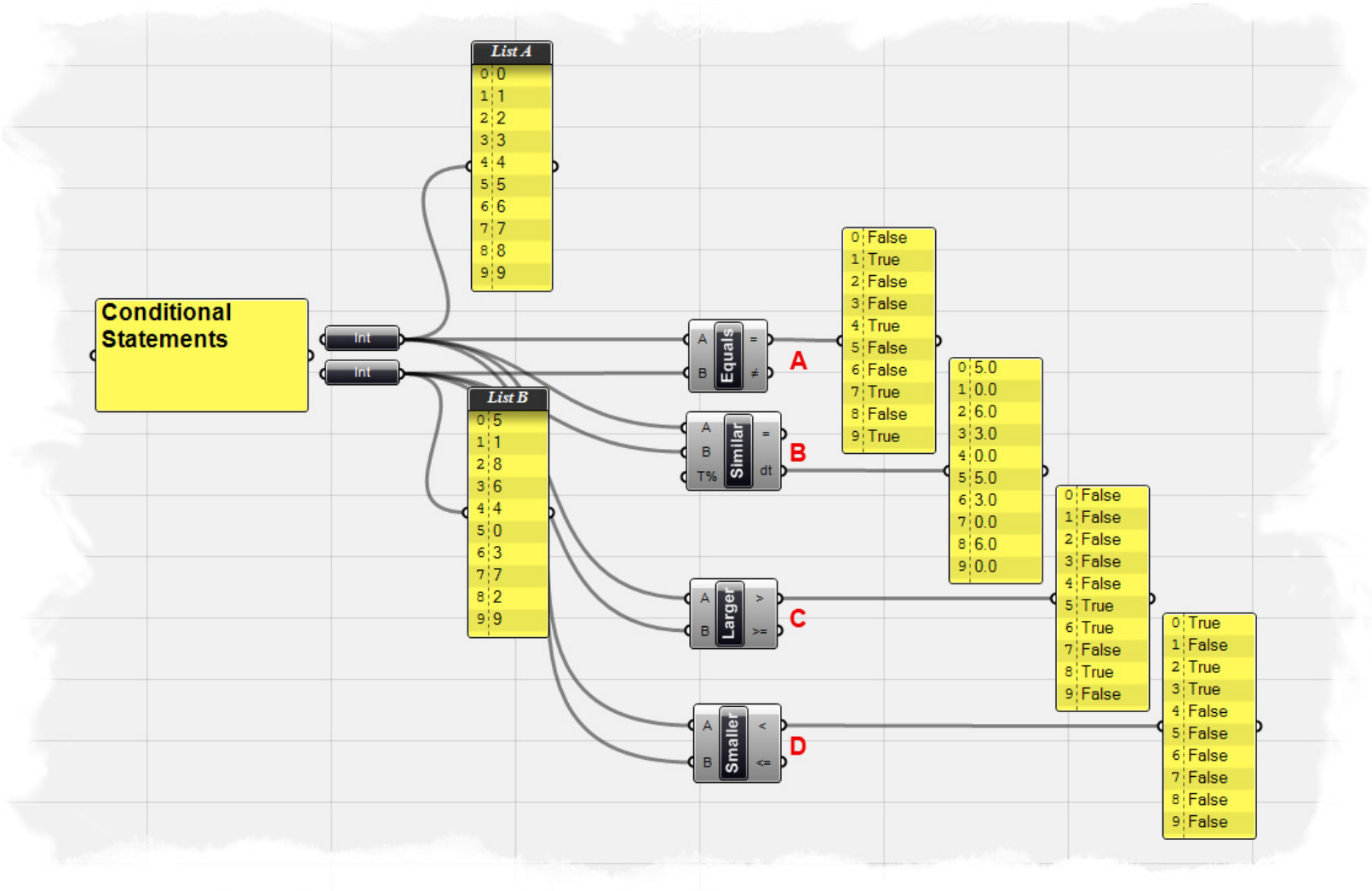
(C)
Intervals
Range
& Series



1. Data Storage

1.2 Scalars:

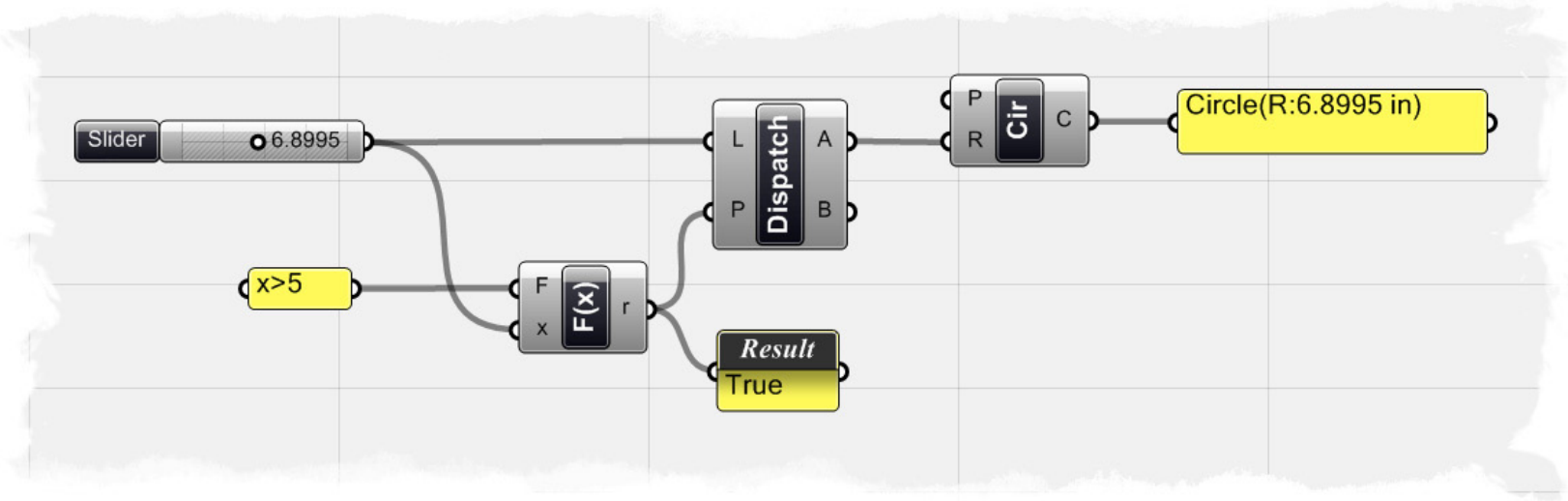
(D) Conditional Statements :



1. Data Storage

1.2 Scalars:

(D) Conditional Statements :

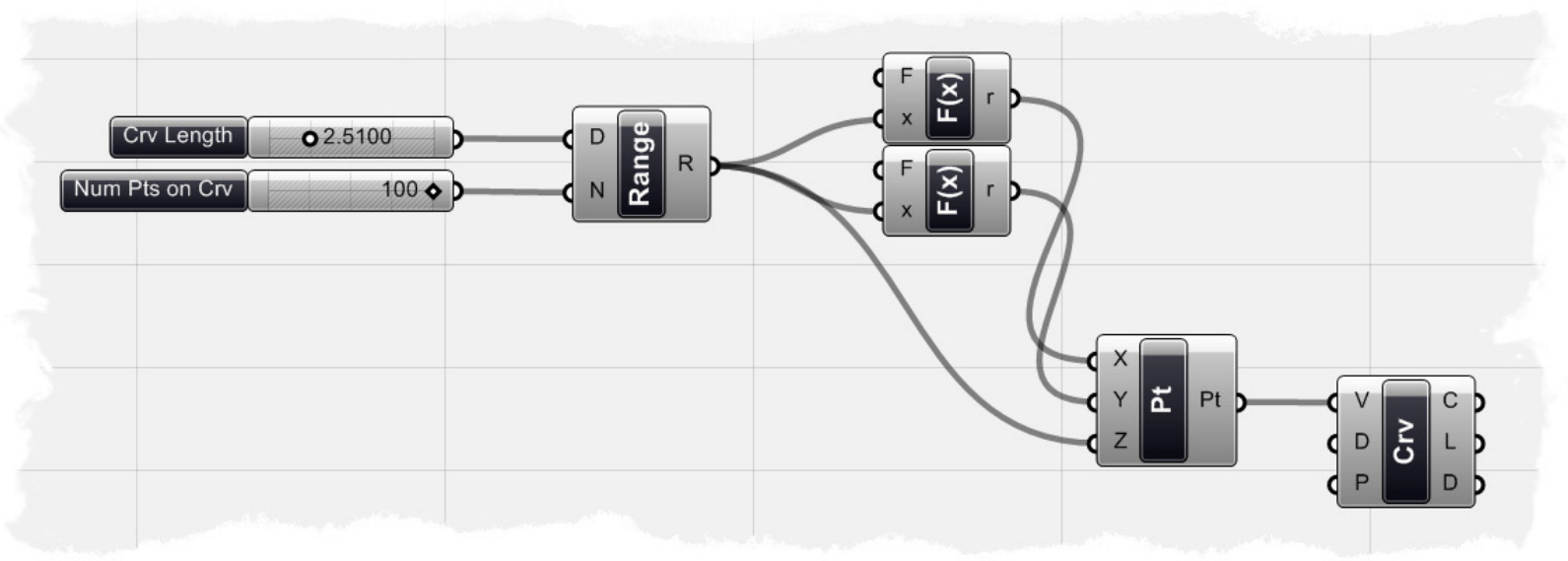


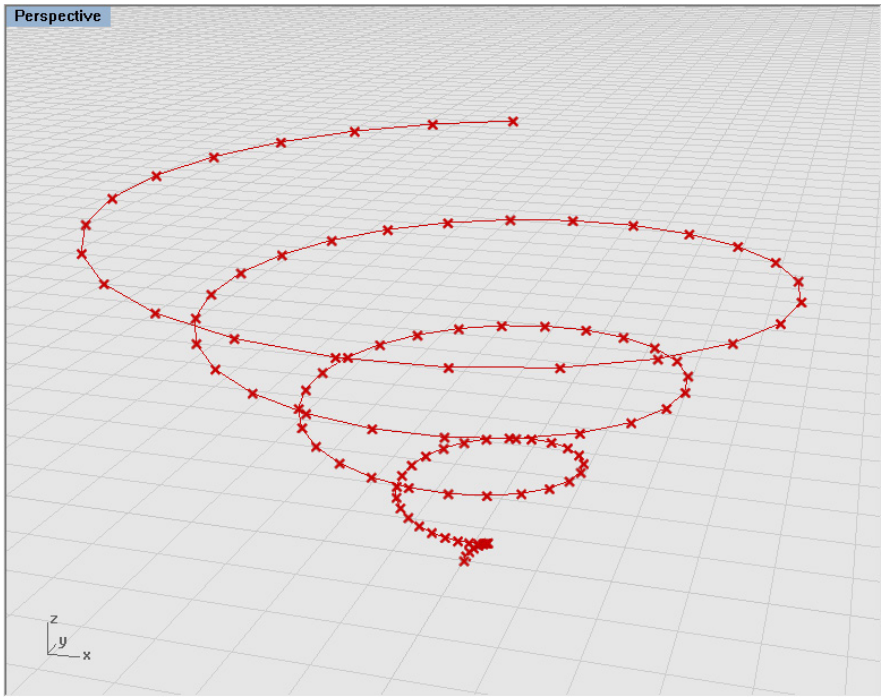
1. Data Storage

1.2 Scalars:

(E) Polynomials:

Spiral Function



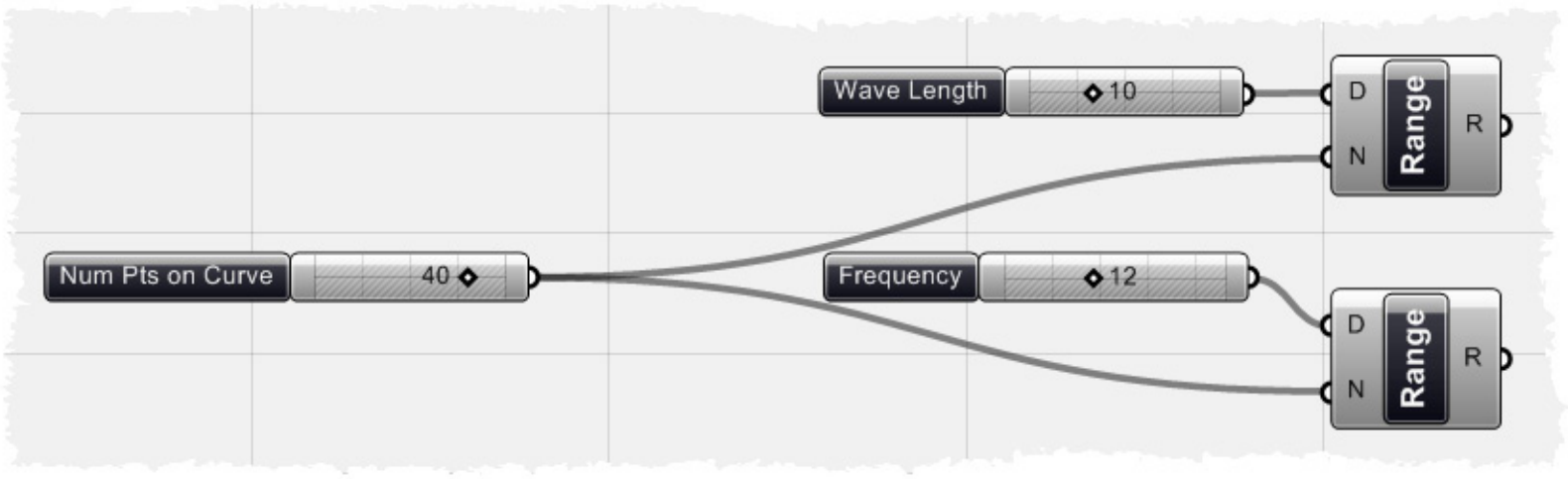


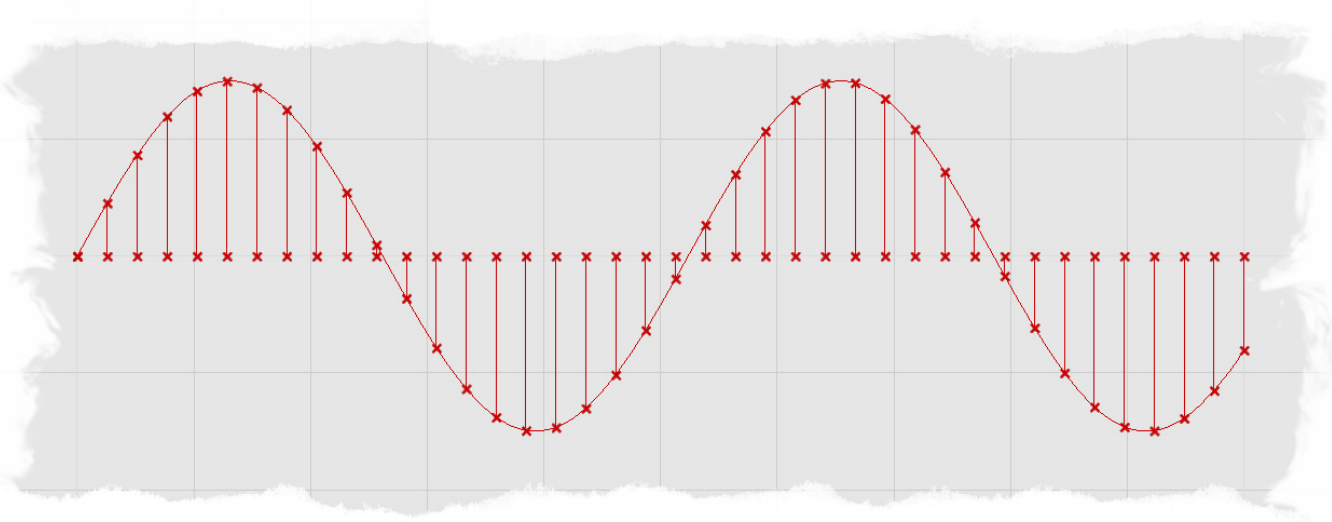
1. Data Storage

1.2 Scalars:

(F)

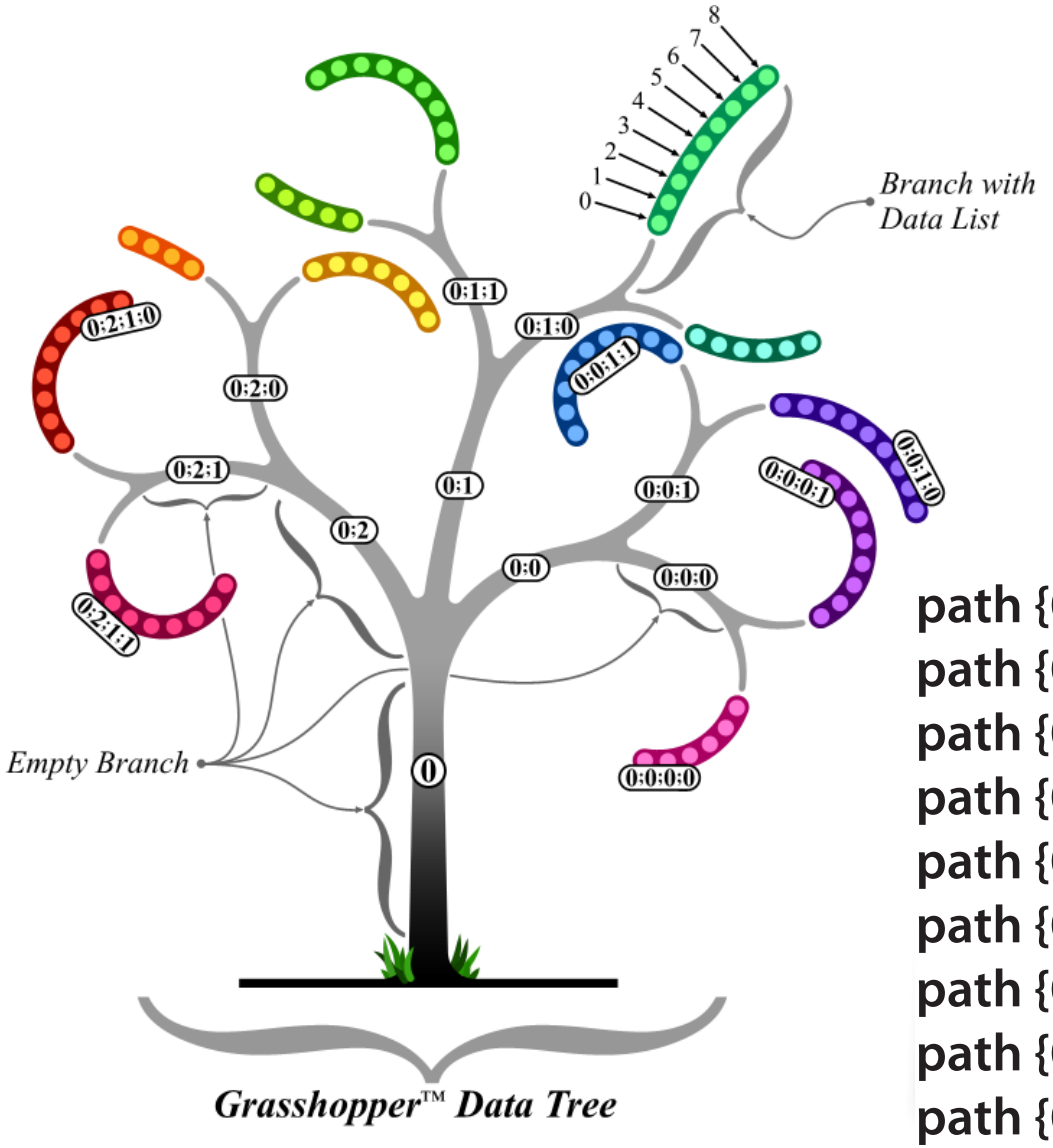
Trigonometric:





1. Data Storage

1.3 Data Trees



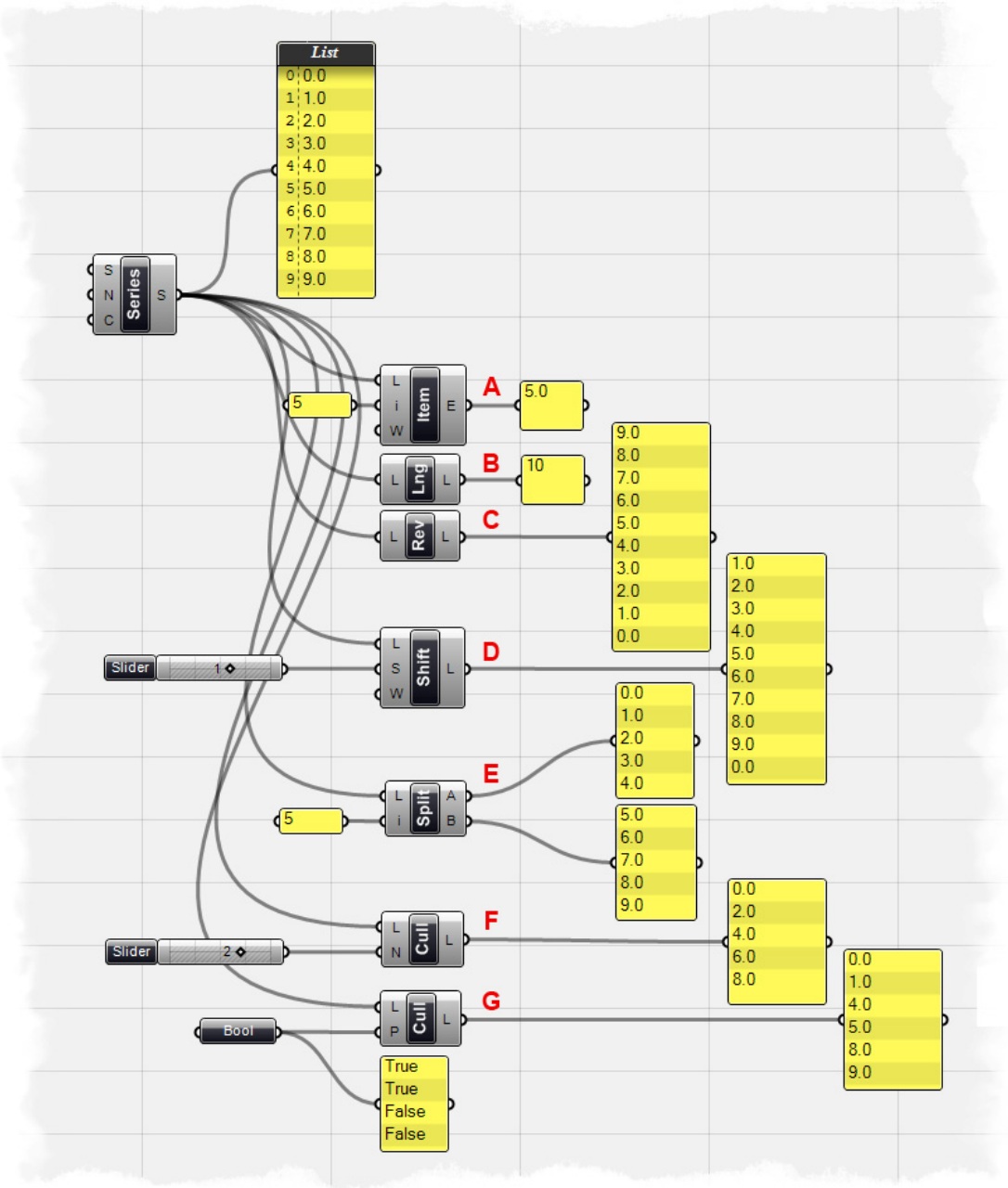
path {0;0;0;0} (N=6)
path {0;0;0;1} (N=9)
path {0;0;1;0} (N=9)
path {0;0;1;1} (N=9)
path {0;1;0;0} (N=6)
path {0;1;0;1} (N=9)
path {0;1;1;0} (N=9)
path {0;1;1;1} (N=5)
path {0;2;0;0} (N=7)
path {0;2;0;1} (N=4)
path {0;2;1;0} (N=9)
path {0;2;1;1} (N=9)

1. Data Storage

1.3 Data Trees

Lists & data management

- List Item
- List length
- List Reverse
- List Shift
- Split list
- Cull nth
- Cull Pattern
- Weaving Data

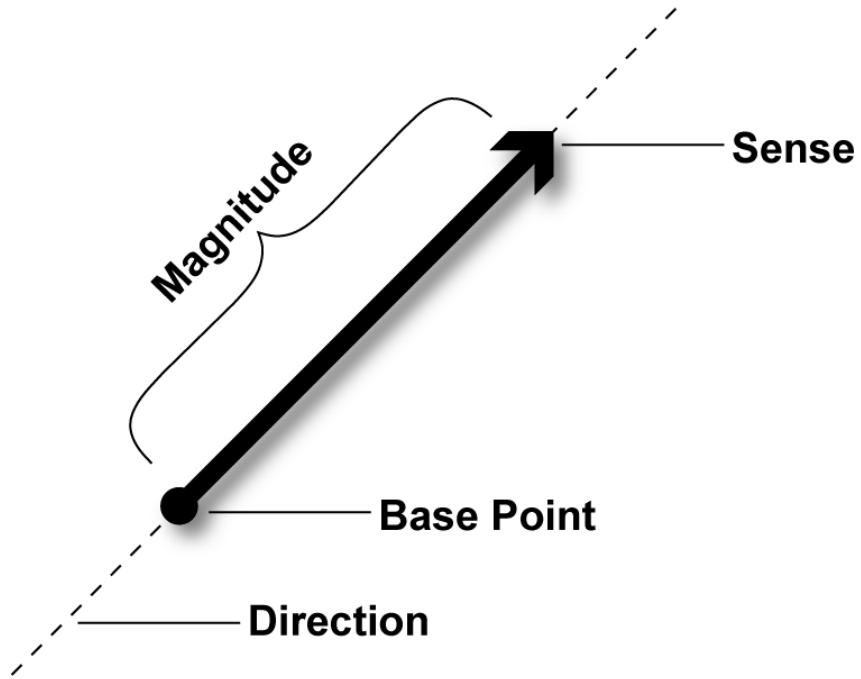


Vector Definition:

magnitude: length of the vector; distance between the start and end points (scalar value)

direction: the line ($-\infty$ to $+\infty$) on which the vector is based

sense: whether vector is oriented towards $-\infty$ or $+\infty$



1. Data Storage

1.4 Vectors:

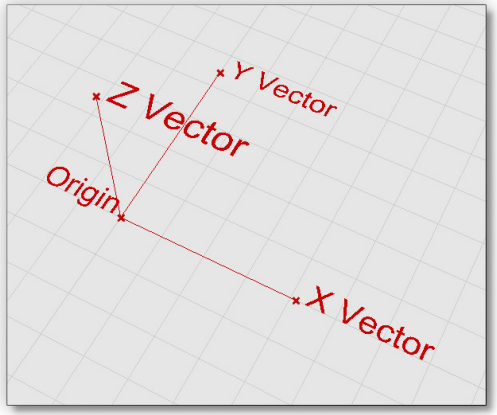
Vector Unit

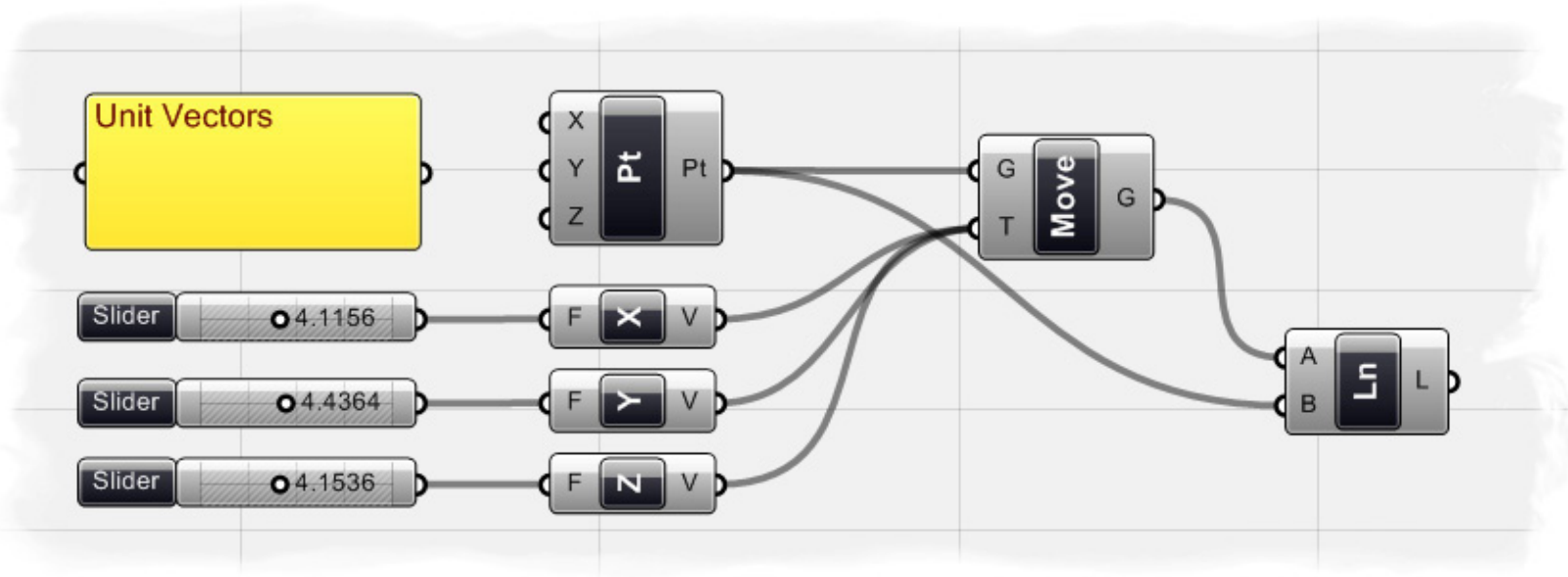
Vector Unit:

magnitude: 1 (the unity)

direction: x, y or z cartesian system axis

sense: cartesian system agreed senses

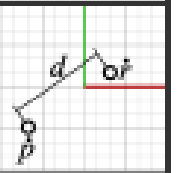
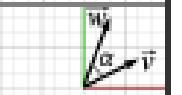
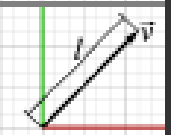




1. Data Storage

1.4 Vectors:

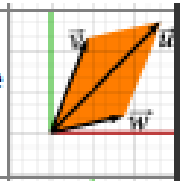
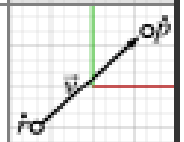
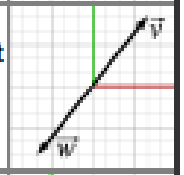
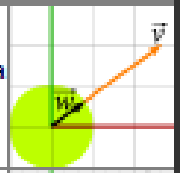
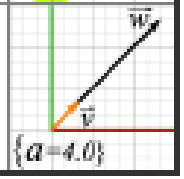
Manipulation

Component	Location	Description	Example
	Vector/Point/**Distance**	Compute the Distance between two points (A and B inputs)	
	Vector/Point/**Decompose**	Break down a point into its X, Y, and Z components	
	Vector/Vector/**Angle**	Compute the angle between two vectors Output computed in Radians	
	Vector/Vector/**Length**	Compute the length (amplitude) of a vector	
	Vector/Vector/**Decompose**	Break down a vector into its component parts	

1. Data Storage

1.4 Vectors:

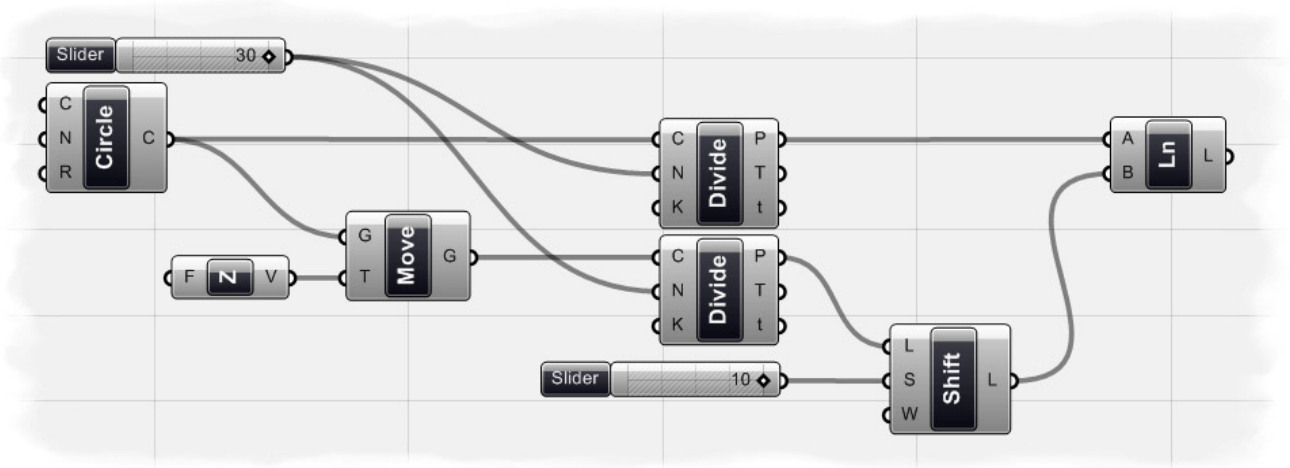
Manipulation

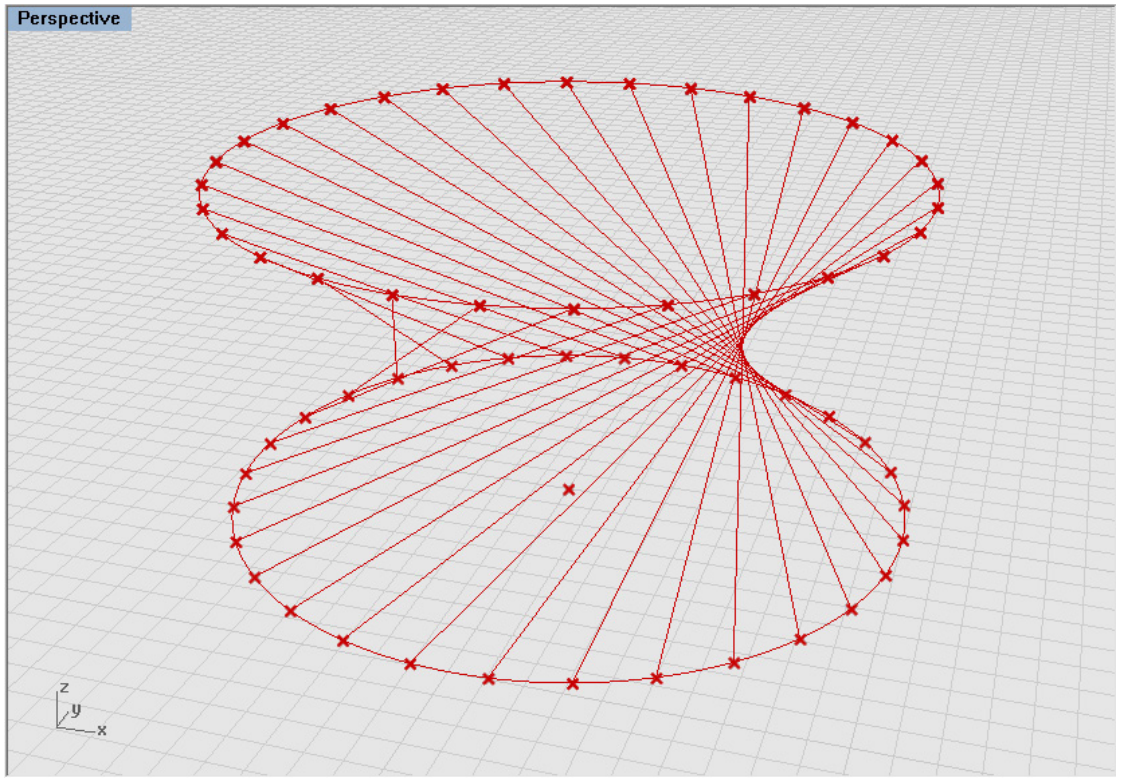
	Vector/Vector/**Summation**	Add the components of vector 1(A input) to the components of vector 2 (B input)	
	Vector/Vector/**Vector2pt**	Creates a vector from two defined points	
	Vector/Vector/**Reverse**	Negate all the components of a vector to invert the direction. The length of the vector is maintained	
	Vector/Vector/**Unit Vector**	Divide all components by the inverse of the length of the vector. The resulting vector has a length of 1.0 and is called the unit vector. Sometimes referred to as 'normalizing'	
	Vector/Vector/**Multiply**	Multiply the components of the vector by a specified factor	

1. Data Storage

1.4 Vectors:

Manipulation of Vectors & Data Trees Example ‘Shift Data’



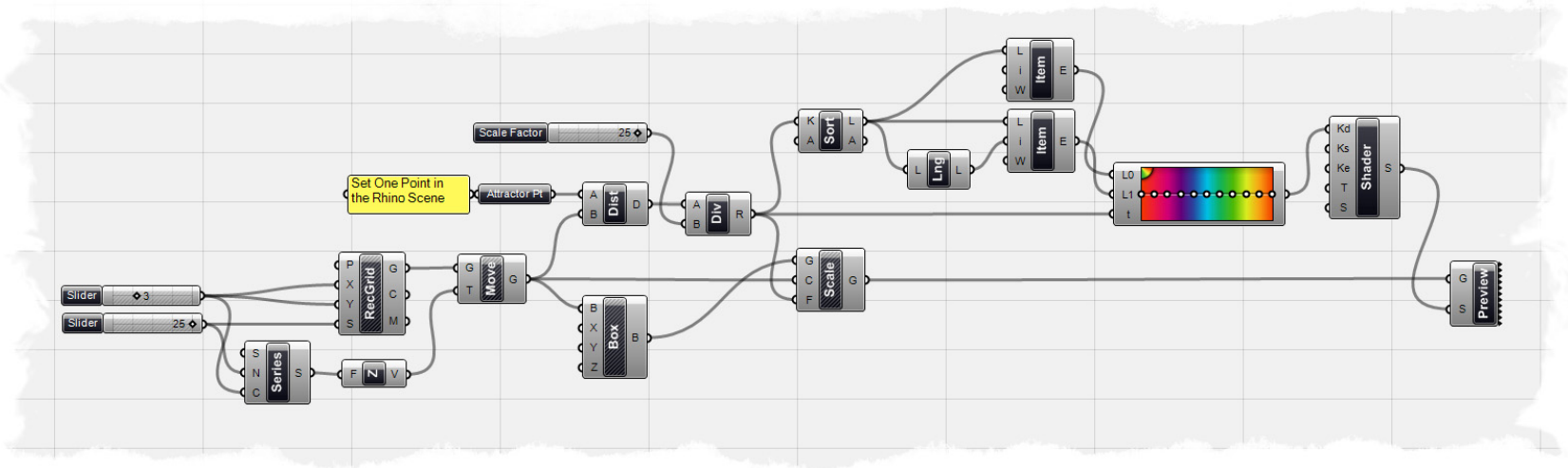


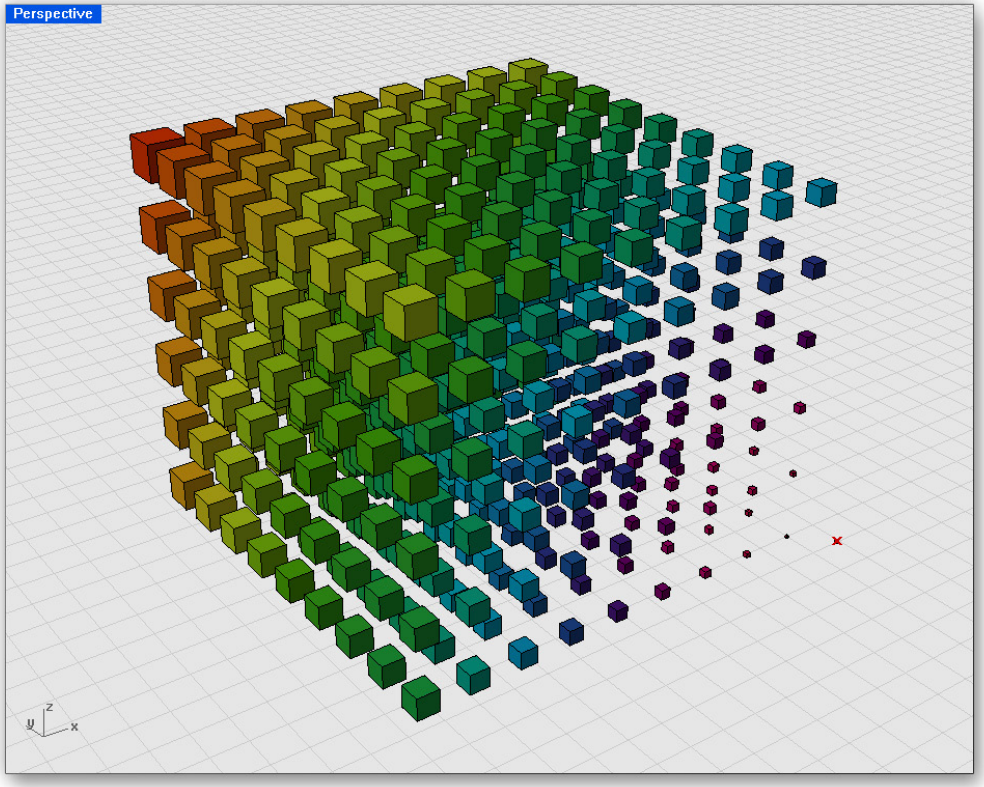
1. Data Storage

1.4 Vectors:

Manipulation of Vectors & Data Trees & Grids of Points.

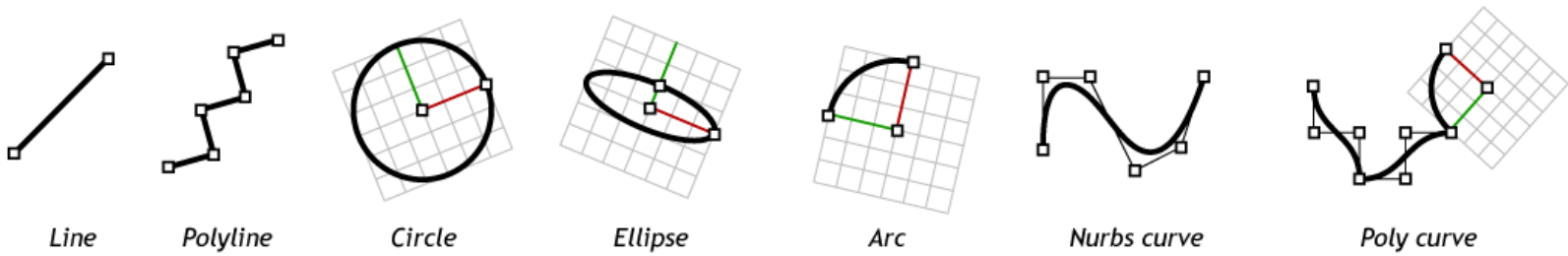
Example ‘Scaled Boxes’



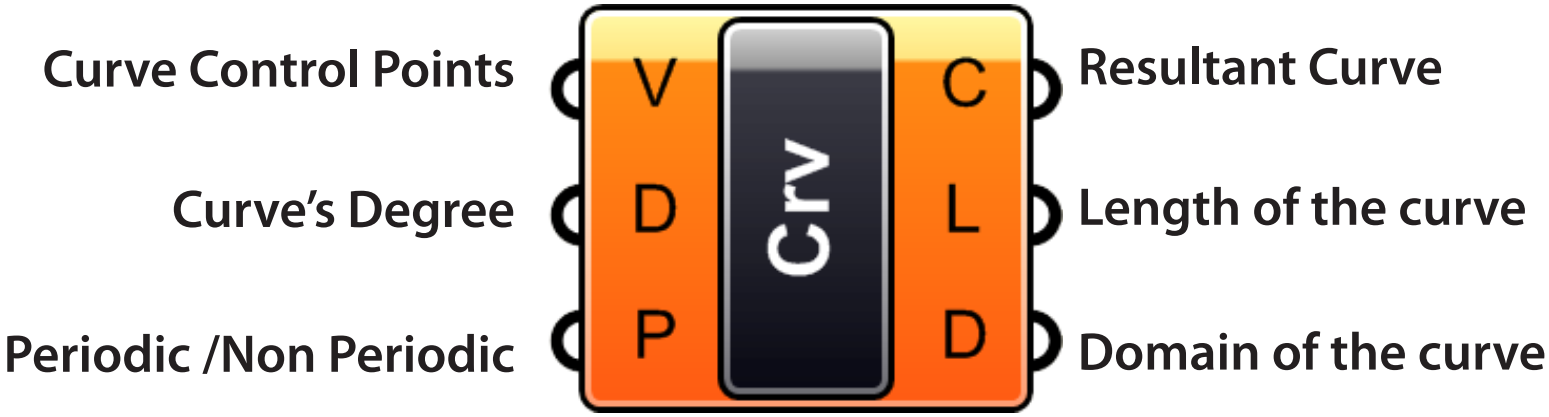


2. Curves

2.1 Definition & Type of Curves



NURBS Curve

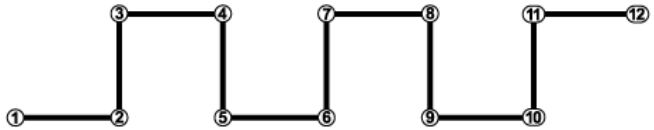


2. Curves

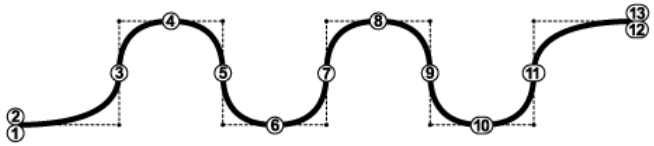
2.1 Definition & Type of Curves

Degree of a NURBS curve

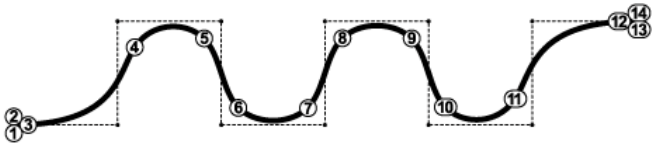
NURBS curve knot vectors as a result of varying degree



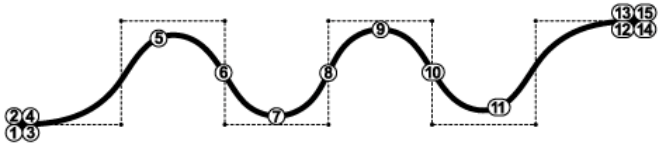
A D^1 nurbs curve behaves the same as a polyline. It follows from the knotcount formula that a D^1 curve has a knot for every control point. Thus, there is a one-to-one relationship.



A D^2 nurbs curve is in fact a rare sighting. It always looks like it is over-stressed, but the knots are at least in straightforward locations. The spline intersects with the control polygon halfway each segment. D^2 nurbs curves are typically only used to approximate arcs and circles.

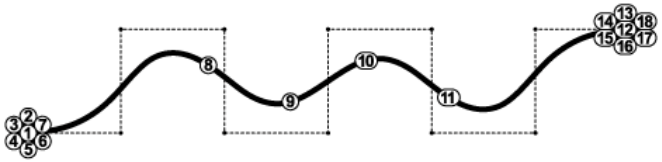


D^3 is the most common type of nurbs curve and -indeed- the default in Rhino. You are probably very familiar with the visual progression of the spline, eventhough the knots appear to be in odd locations.

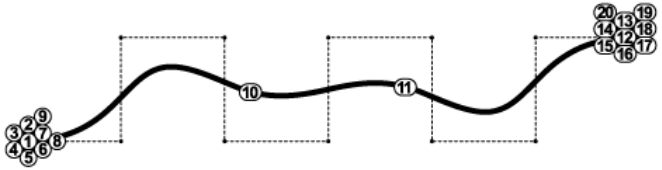


D^4 is technically possible in Rhino, but the math for nurbs curves doesn't work as well with even degrees. Odd numbers are usually preferred.

D^5 is also quite a common degree. Like the D^3 curves it has a natural, but smoother appearance. Because of the higher degree, control points have a larger range of influence.



D^7 and D^9 are pretty much hypothetical degrees. Rhino goes all the way up to D^{11}, but these high-degree-splines bear so little resemblance to the shape of the control polygon that they are unlikely to be of use in typical modeling applications.



2. Curves

2.1 Definition & Type of Curves

Other Types:

Interpolated Curve

Kinky Curve

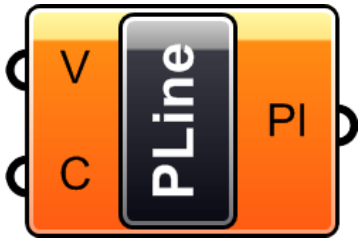
Polyline

PolyArc

Curve through control points

Similar to interpolated; kinnks at each vertex; requires angle (A) of kink

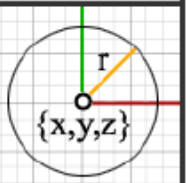
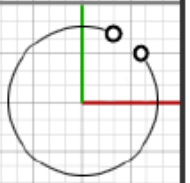
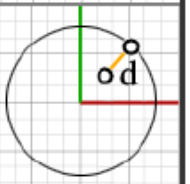
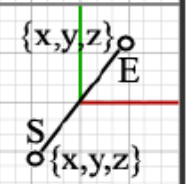
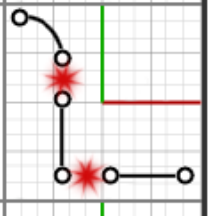
Curve made of straight segments



Curve made of arched segments

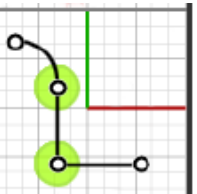
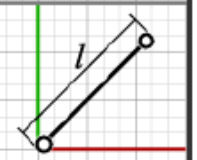
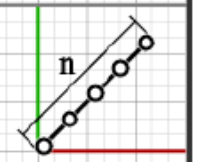
2. Curves

2.2 Curve Analytics

Component	Location	Description	Example
	Curve/Analysis/**Center**	Find the center point and radius of arcs and circles	
	Curve/Analysis/**Closed**	Test if a curve is closed or periodic	
	Curve/Analysis/**Closest Point**	Find the closest point on a curve to any sample point in space	
	Curve/Analysis/**End Points**	Extract the end points of a curve.	
	Curve/Analysis/**Explode**	Decompose a curve into its component parts	

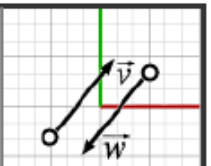
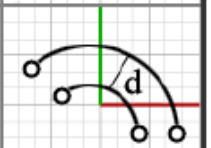
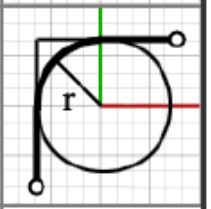
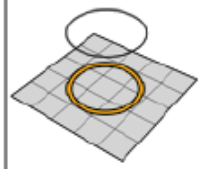
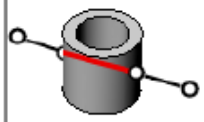
2. Curves

2.2 Curve Analytics

	Curve/Utility/**Join Curves**	Join as many curve segments together as possible	
	Curve/Analysis/**Length**	Measure the length of a curve	
	Curve/Division/**Divide Curve**	Divide a curve into a equal length segments	
	Curve/Division/**Divide Distance**	Divide a curve with a preset distance between points	
	Curve/Division/**Divide Length**	Divide a curve with a segments with a preset length	

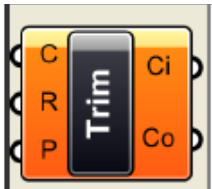
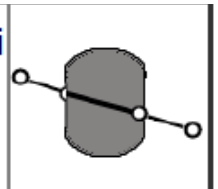
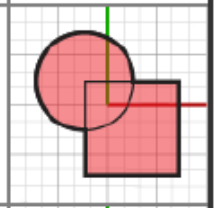
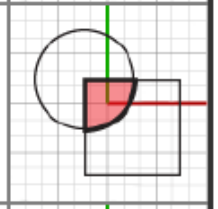
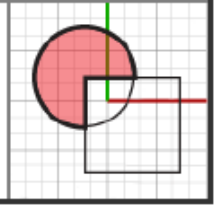
2. Curves

2.2 Curve Analytics

	Curve/Utility/**Flip**	Flip the direction of a curve using an optional guide curve	
	Curve/Utility/**Offset**	Offset a curve with a specified distance	
	Curve/Utility/**Fillet**	Fillets the sharp corners of a curve with an input radius	
	Curve/Utility/**Project**	Project a curve onto a Brep (a Brep is a set of joined surfaces like a polysurface in Rhino)	
	Intersect/Region/**Split with Brep(s)**	Split a curve with one or more Breps	
	Intersect/Region/**Trim with Brep(s)**	Trim a curve with one or more Breps. The **Ci** (Curves Inside) and **Co** (Curves Outside) outputs indicate the direction in which you would like the trim to occur.	

2. Curves

2.2 Curve Analytics

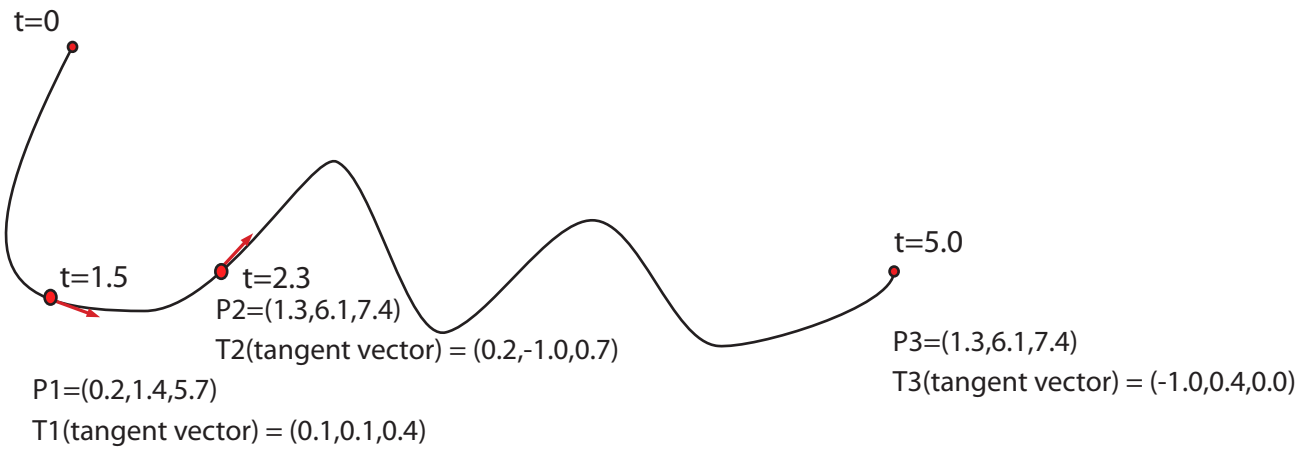
	Intersect/Region/**Trim with Region(s)**	Trim a curve with one or more Regions. The **Ci** (Curves Inside) and **Co** (Curves Outside) outputs indicate the direction in which you would like the trim to occur.	
	Intersect/Boolean/**Region Union**	Finds the outline (or union) of two planar closed curves	
	Intersect/Boolean/**Region Intersection**	Finds the intersection of two planar closed curves	
	Intersect/Boolean/**Region Difference**	Finds the difference between two planar closed curves	

2. Curves

2.2 Curve Analytics

Curve Divide & Curve Domain

Curve Domain

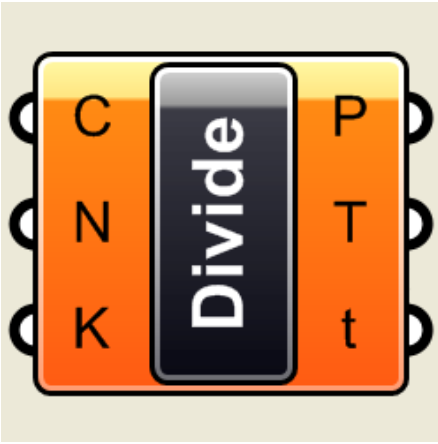


Divide curve by number of Divisons

Curve

Number of Divisions

Divide at Kinks?



3D Points at Division

Tangent at Divisions

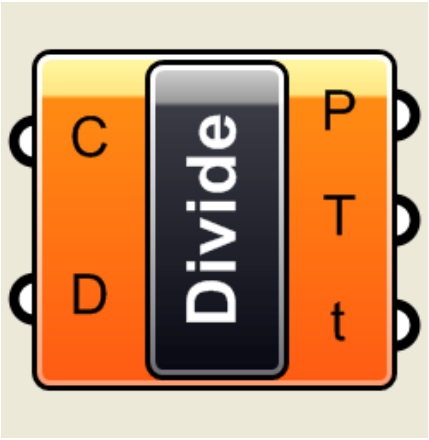
Domain parameter at divisions

2. Curves

2.2 Curve Analytics

Curve Domain Other Options:

Divide curve by Distance

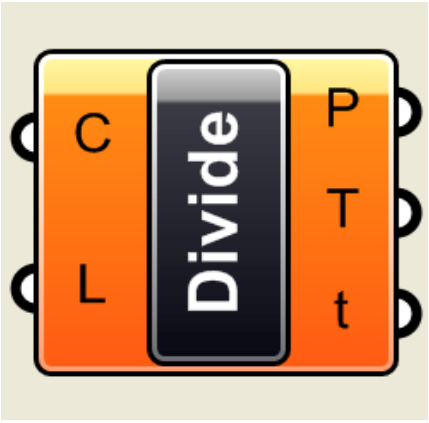


3D Points at Division

Tangent at Divisions

Domain parameter at divisions

Divide curve by length



3D Points at Division

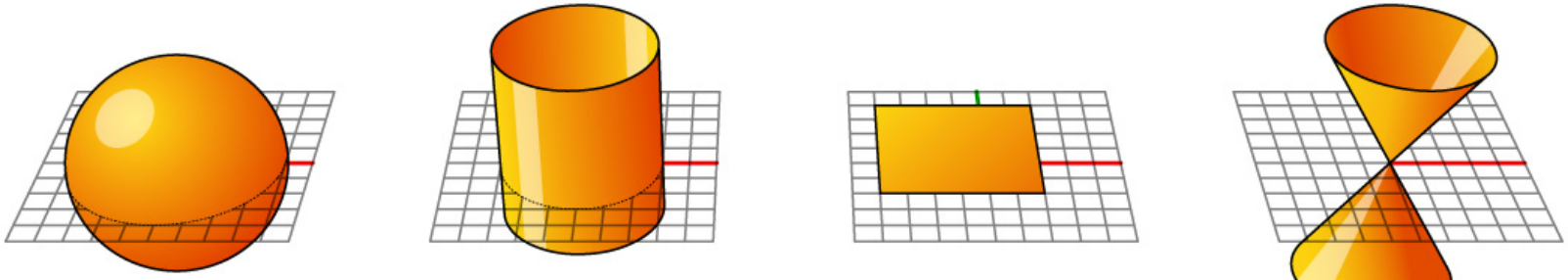
Tangent at Divisions

Domain parameter at divisions

3. Surfaces

3.1 Type of Surfaces

Primitives



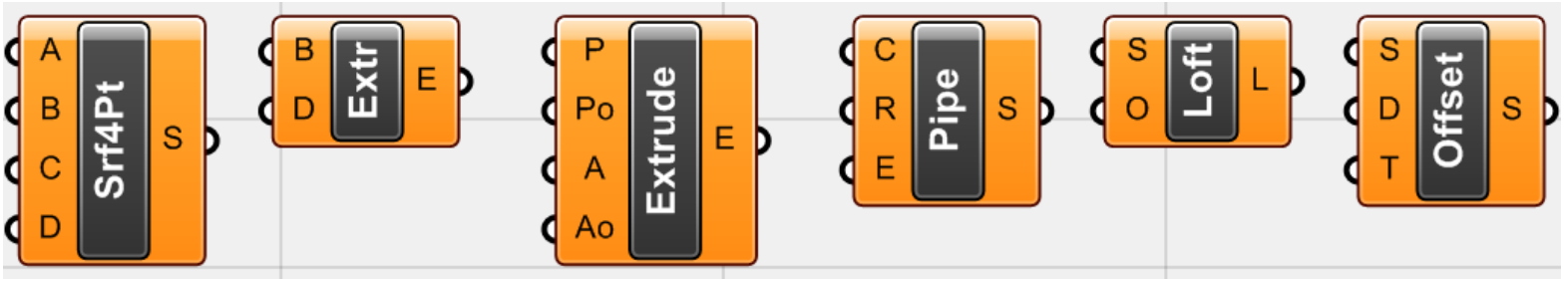
Sphere primitive
{Plane; Radius}

Cylinder primitive
{Plane; Radius; Height}

Plane primitive
{Plane; Width; Height}

Cone primitive
{Plane; Radius; Height}

Free-Form



Surface from 4 corner points

Extrude Curves

Extrude linear

Pipe

Loft Curves

Offset Surface

3. Surfaces

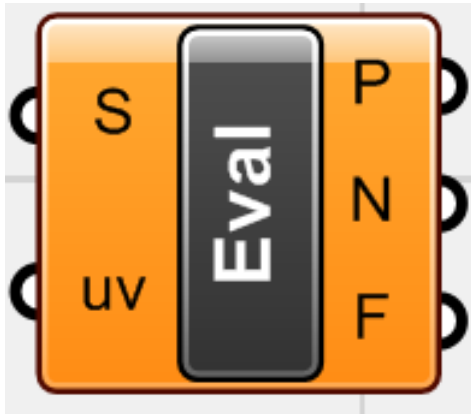
3.1 Surface Analytics:

Surface Domain

Tangent Plane and Normal Vector

Surface

u and v coordinates to evaluate



3D Points at u,v coordinates

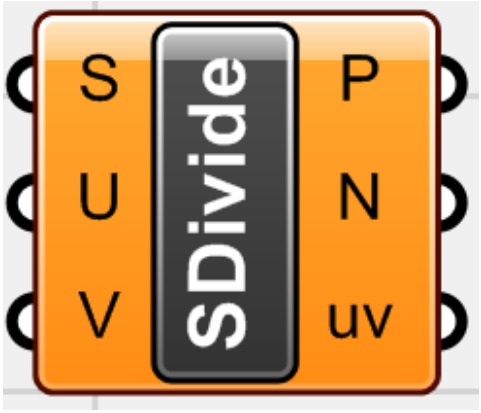
Normal Vectors at u,v

Tangent Plane (Frame)

Surface Divide (to obtain coordinates in u,v directions)

Surface

Divisions in V



3D Points at u,v coordinates

Normal Vectors at u,v

u,v coordinates from division of surface

3. Surfaces

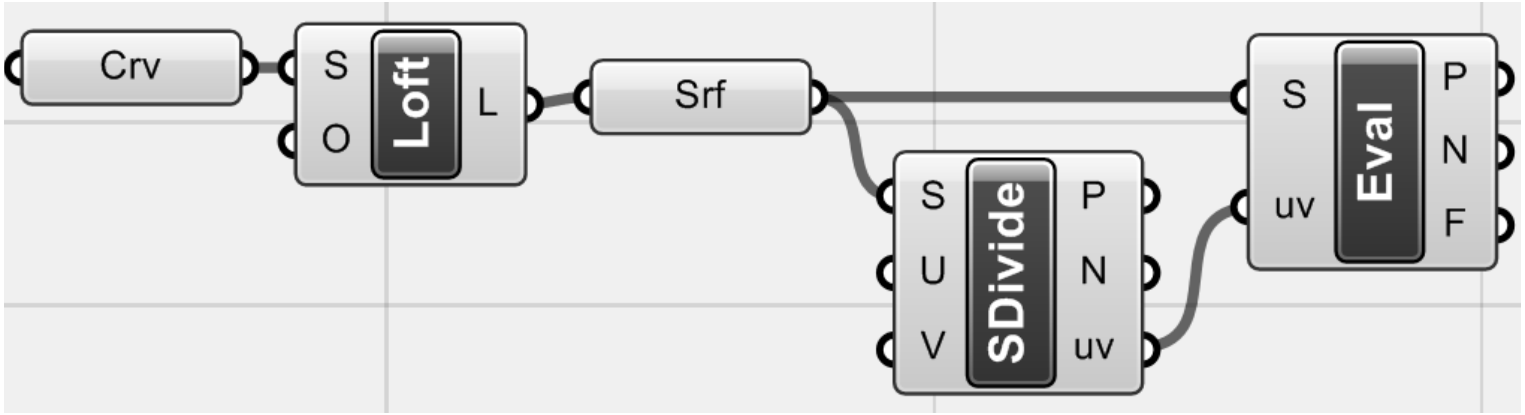
3.1 Surface Analytics:

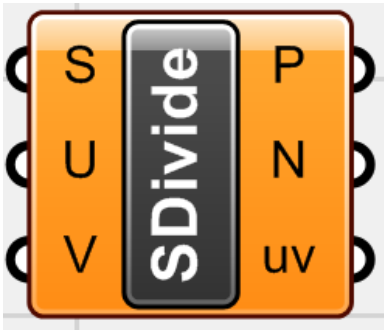
Surface Domain

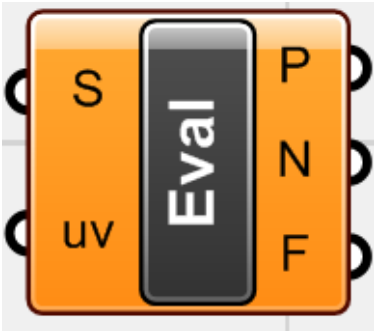
Tangent Plane and Normal Vector

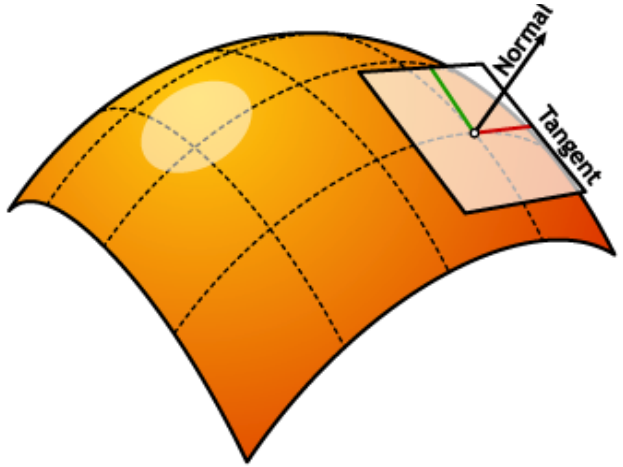
Normal Vectors & Tangent Planes:

Surface Divide + Evaluate Surface





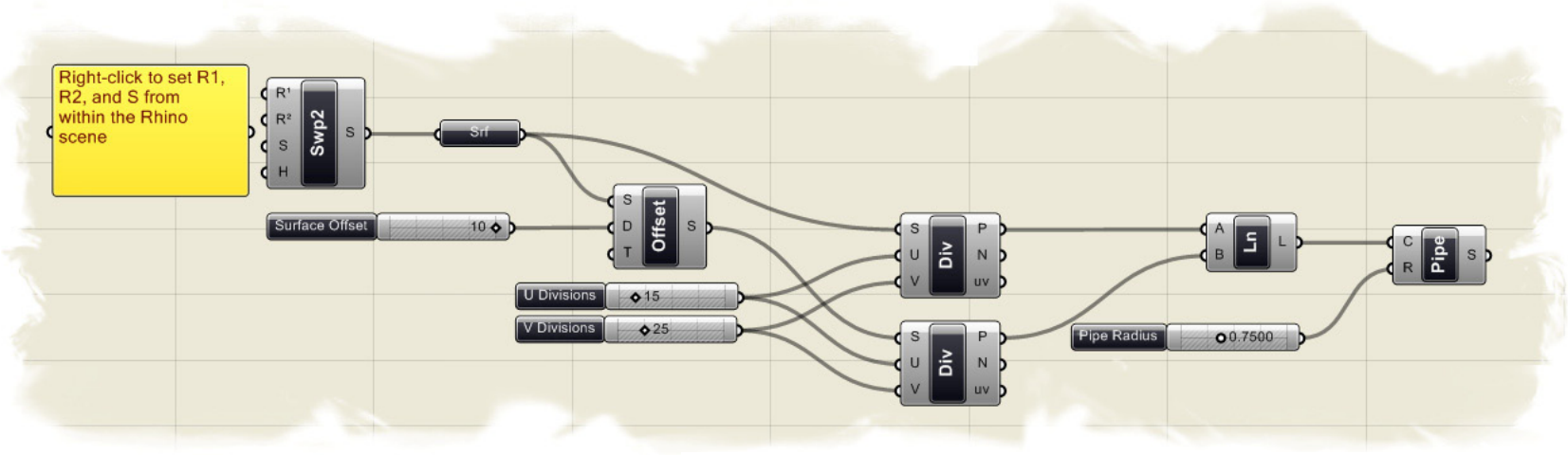


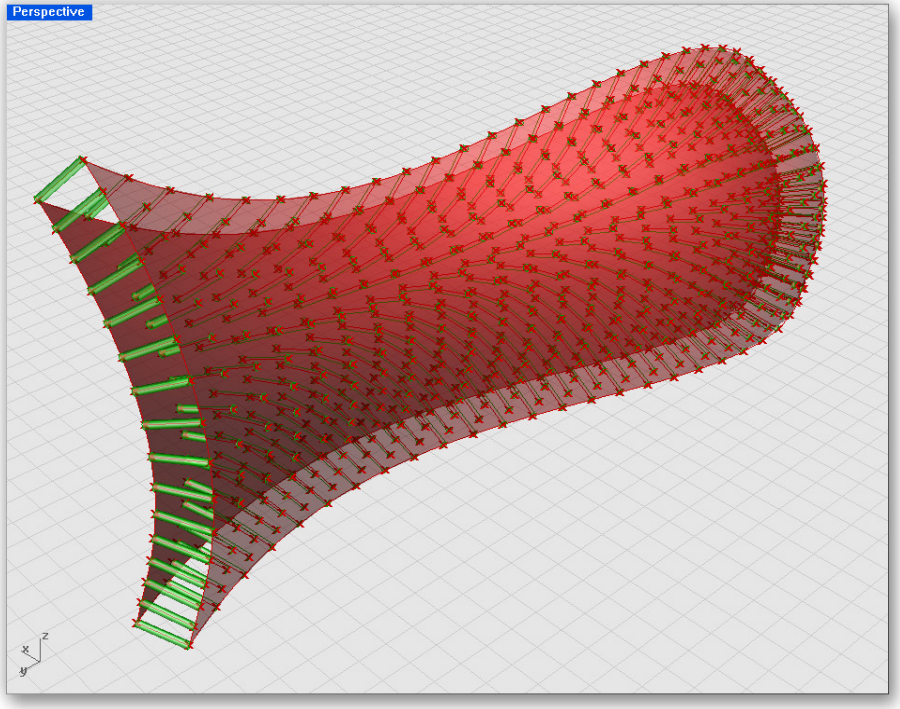


3. Surfaces

3.1 Surface Analytics:

Example Surface Connect

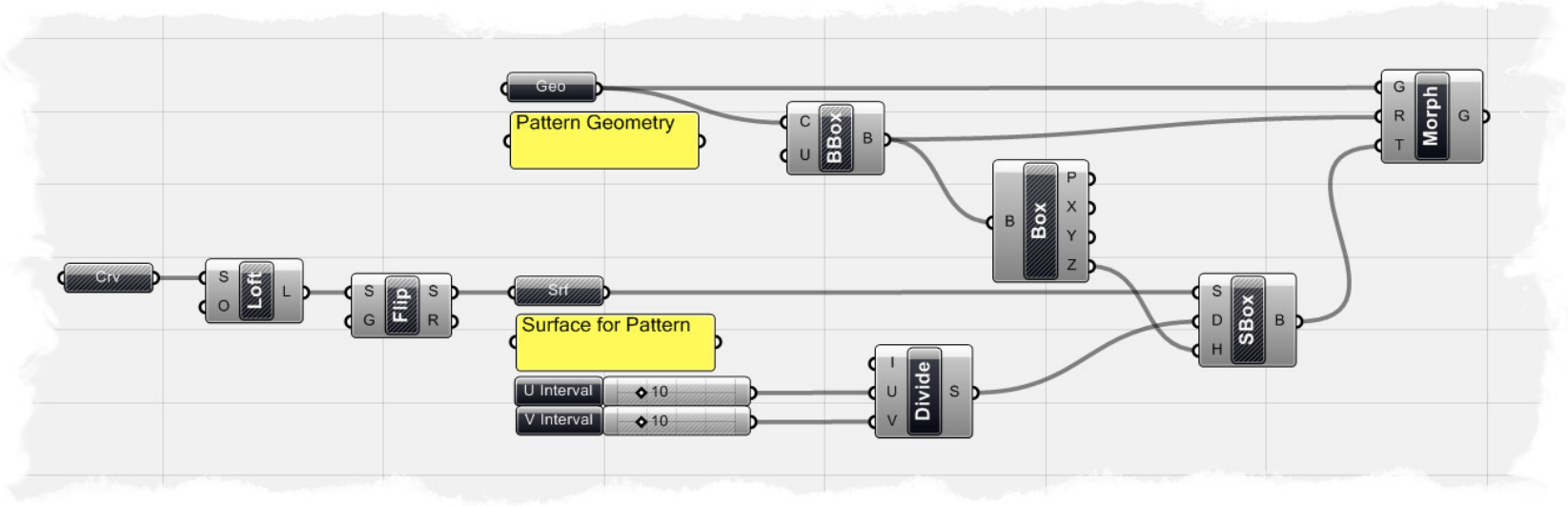


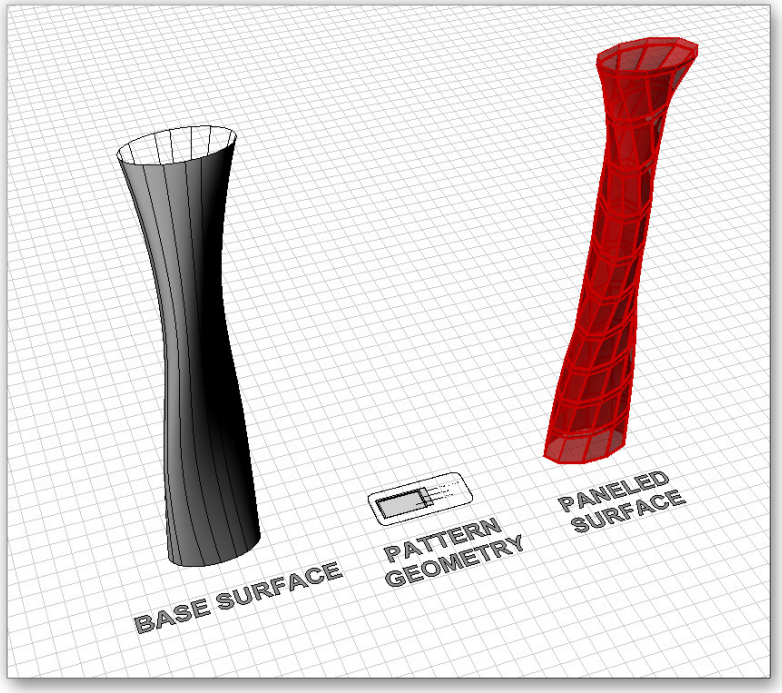


3. Surfaces

3.1 Surface Analytics:

Example Panelisation

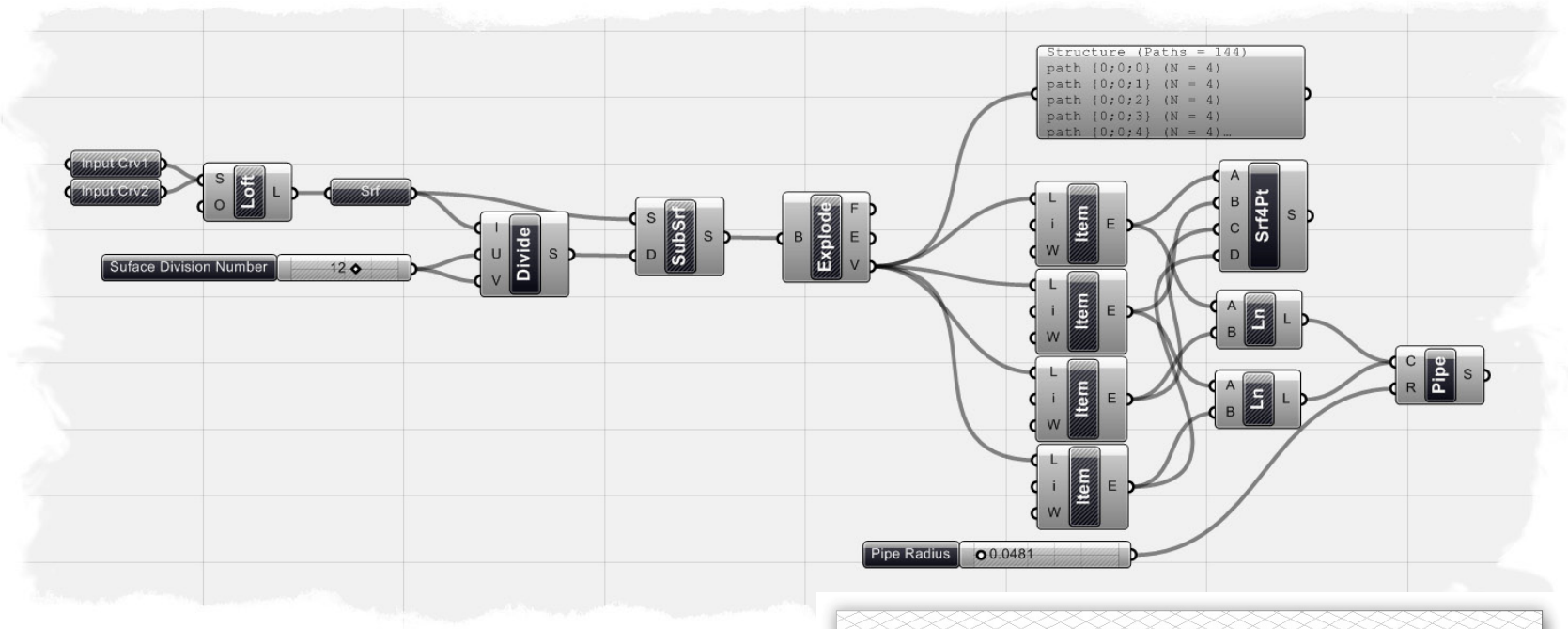


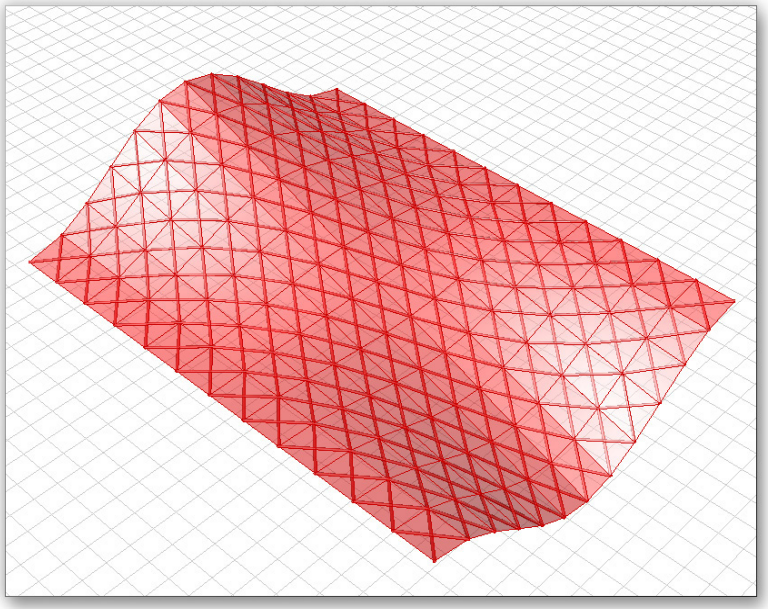


3. Surfaces

3.1 Surface Analytics:

Surface Diagrid (Uniform)



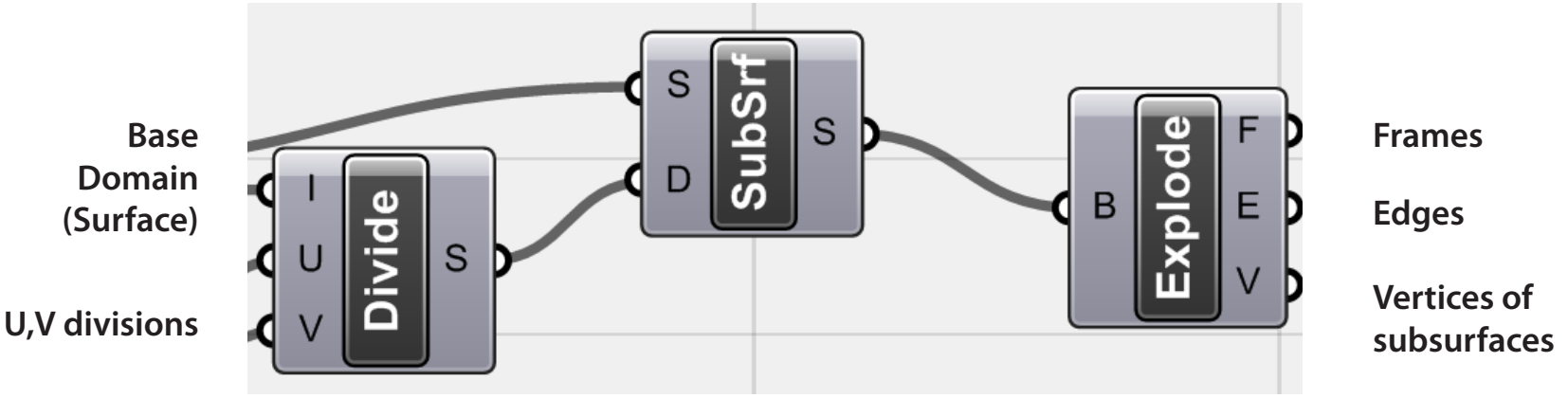


3. Surfaces

3.1 Surface Analytics:

Surface Diagrid (Uniform)

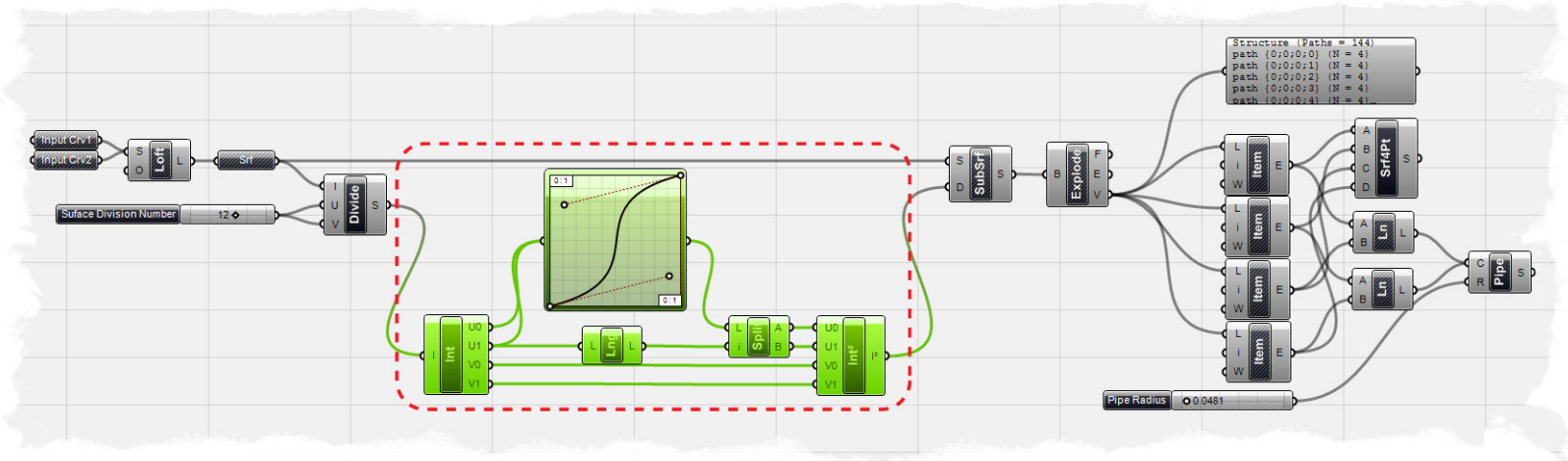
Divide + SubSurface + Explode

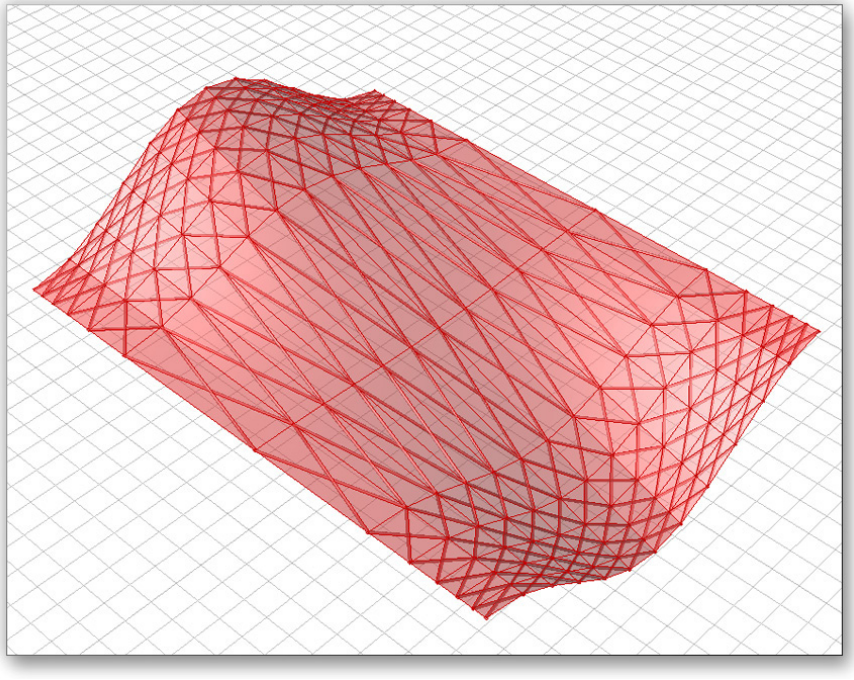


3. Surfaces

3.1 Surface Analytics:

Surface Diagrid (Non-uniform)





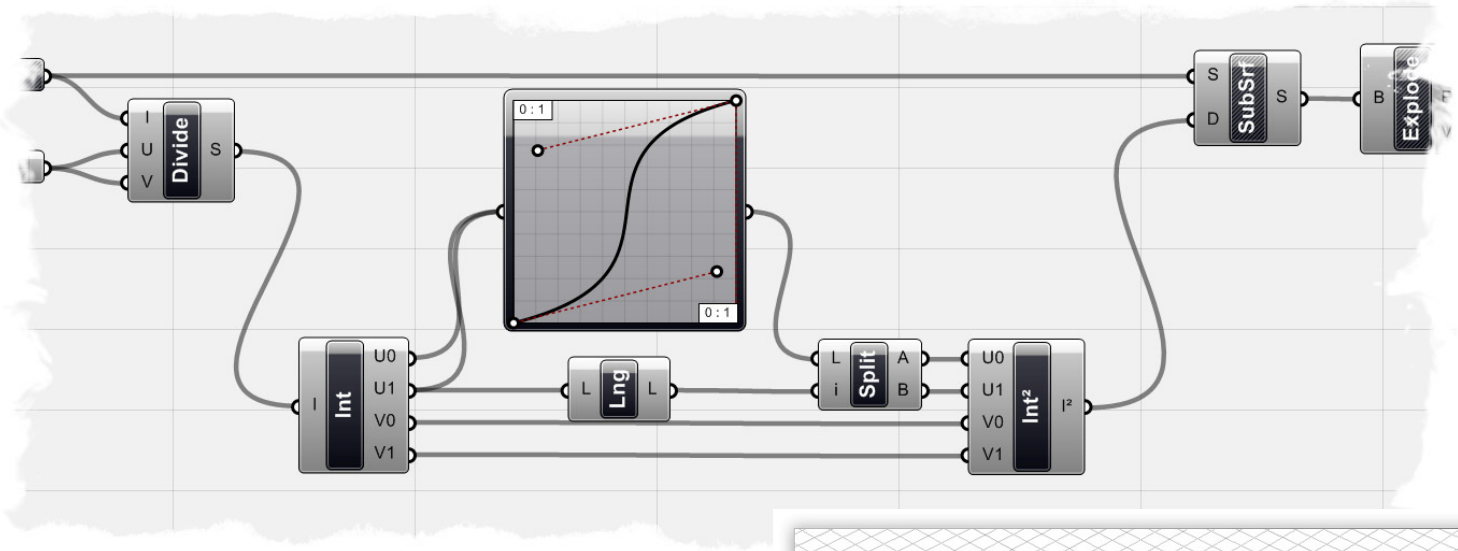
3. Surfaces

3.1 Surface Analytics:

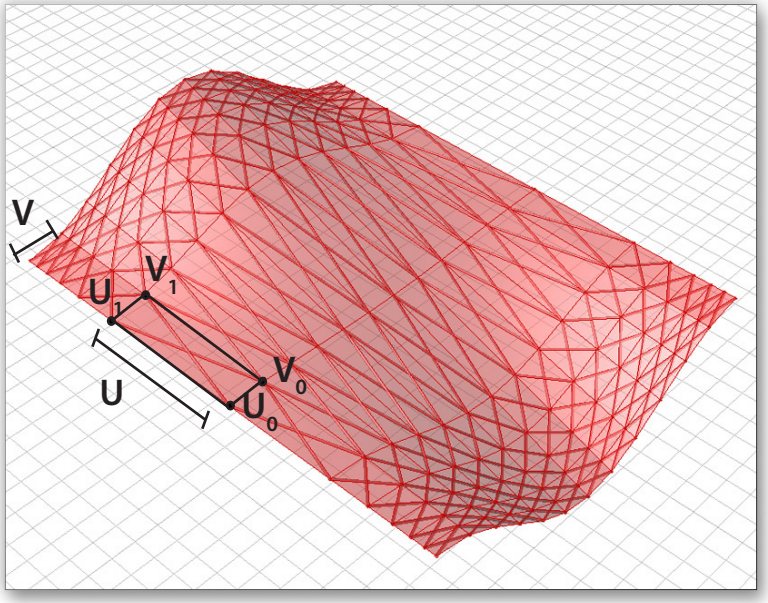
Surface Diagrid (Non-uniform)

Non-Uniform Distortion:

Alter u & v coordinate grid by enlarging or reducing the length of the u,v spaces

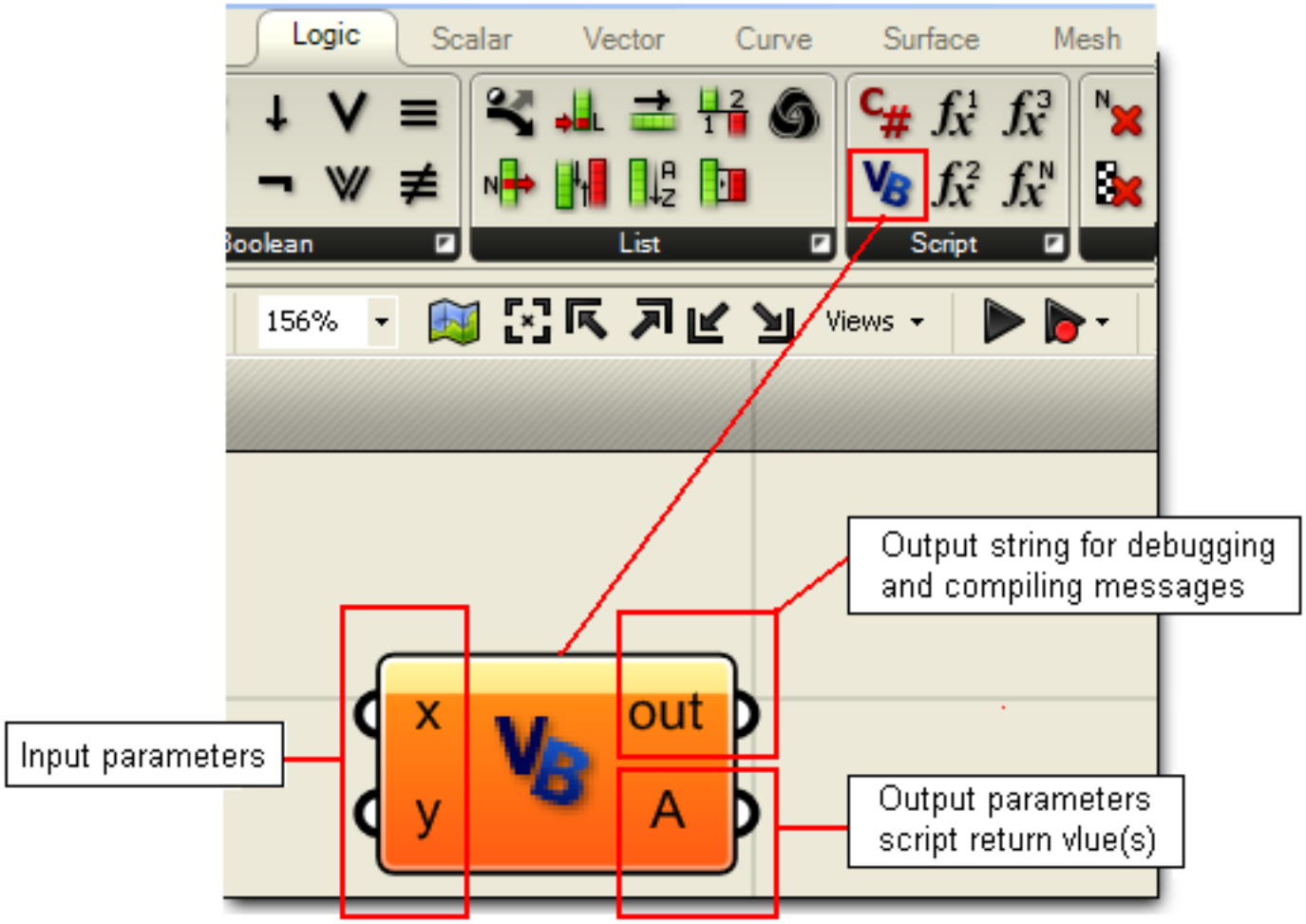


U divisions = larger
V divisions = mantained



4. Introduction vb.net

See Appendix A



4. Introduction vb.net

4.1 Variables

Vb.net vs Vbscript:

1) Variables

When declaring variables, it is necessary to indicate the type of variable
Int32, double, boolean, string, list, list of 3D points, array, array of 3D points, etc

```
Dim x as Int32 = 10
'If you print the value of x at the point, then you will get 10
x = 20
'From now on, x will return 20

Dim x as Double = 20.4
 'Dim keyword means that we are about to define a variable
Dim b as Boolean = True
'Double, Boolean and String are all examples of base or
Dim name as String = "Joe"
' system defined types
```

Create an array:

```
'Array of things
Dim things_list As New List( of String )
things_list.Add( "Thing1" )
things_list.Add( "Thing2" )
things_list.Add( "Thing3" )
```

4. Introduction vb.net

4.2 Conditional Statements

Vb.net vs Vbscript:

2) Conditional Statements

Similar to Vbscript

```
'Single line if statement doesn't need End If: condition=(x<y), code=(x=x+y)
If x < y Then x = x + y
'Multiple line needs End If to close the block of code
If x < y Then
x = x + y
End If

If x < y Then
x = x + y 'execute this line then go to the step after "End If"
Else If x > y Then
x = x – y 'execute this line then go to the step after "End If"
Else
x = 2*x 'execute this line then go to the step after "End If"
End If
```

4. Introduction vb.net

4.3 Loops

Vb.net vs Vbscript:

3)Loops

Similar to Vbscript

```
'Array of places
Dim places_list As New List( of String )
places_list.Add( "Paris" )
places_list.Add( "NY" )
places_list.Add( "Beijing" )

'Loop index
Dim i As Integer
Dim place As String
Dim count As Integer = places_list.Count()

'Loop starting from 0 to count -1 (count = 3, but last index of the places_list = 2)
For i=0 To count-1
place = places_list(i)
Print( place )
Next
```

4. Introduction vb.net

4.4 Operators

Vb.net vs Vbscript:

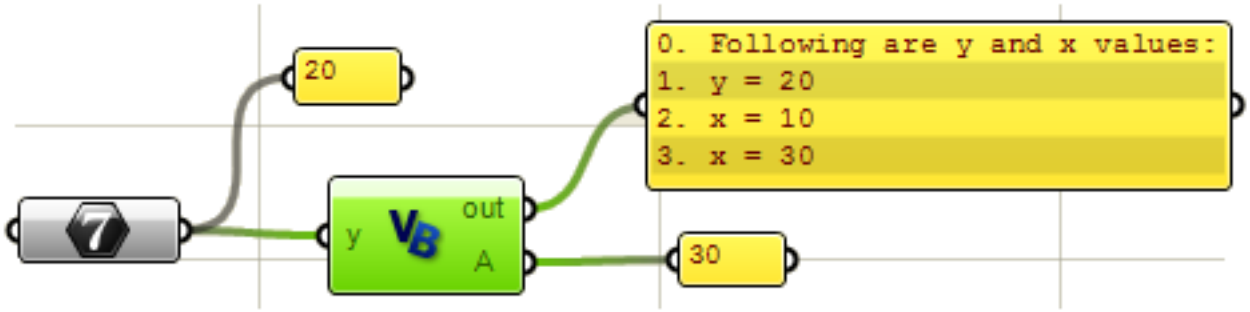
4) Operators

Similar to Vbscript

Type	Operator	Description
Arithmetic Operators	^	Raises a number to the power of another number.
	*	Multiplies two numbers.
	/	Divides two numbers and returns a floating-point result.
	\	Divides two numbers and returns an integer result.
	Mod	Divides two numbers and returns only the remainder.
	+	Adds two numbers or returns the positive value of a numeric expression.
	-	Returns the difference between two numeric expressions or the negative value of a numeric expression
Assignment Operators	=	Assigns a value to a variable
	^=	Raises the value of a variable to the power of an expression and assigns the result back to the variable.
	*=	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.
	/=	Divides the value of a variable by the value of an expression and assigns the floating-point result to the variable.
	\=	Divides the value of a variable by the value of an expression and assigns the integer result to the variable.
	+=	Adds the value of a numeric expression to the value of a numeric variable and assigns the result to the variable. Can also be used to concatenate a String expression to a String variable and assign the result to the variable.
	-=	Subtracts the value of an expression from the value of a variable and assigns the result to the variable.
	&=	Concatenates a String expression to a String variable or property and assigns the result to the variable or property.
Comparison Operators	<	Less Than
	<=	Less or equal
	>	Greater than
	>=	Greater or equal
	=	Equal
	<>	Not equal
Concatenation Operators	&	Generates a string concatenation of two expressions.
	+	Concatenate two string expressions.
Logical Operators	And	Performs a logical conjunction on two Boolean expressions
	Not	Performs logical negation on a Boolean expression
	Or	Performs a logical disjunction on two Boolean expressions
	Xor	Performs a logical exclusion on two Boolean expressions

4. Introduction vb.net

Example




```
Sub RunScript(ByVal y As Integer)
  'Print variables values
  Print("Following are y and x values:")

  'Print input value (y)
  Print("y = " & y)

  'Declare x as an integer variable
  Dim x As int32 = 10

  'Print x initial value
  Print("x = " & x)

  'Set x value to be whatever was there plus input y
  x = x + y
  Print("x = " & x)

  'Assign x to output
  A = x
End Sub
```

4. Introduction vb.net

Appendix

Appendix

12 An Introduction to Scripting

Grasshopper functionality can be extended using scripting components to write code using VB DotNET or C# programming languages. There will probably be support for more languages in the future. User code is placed inside a dynamically generated class template which is then compiled into an assembly using the CLR compiler that ships with the DotNET framework. This assembly exists purely in the memory of the computer and will not be unloaded until Rhino exits.

The script component in Grasshopper has access to Rhino DotNET SDK classes and functions which are what plugin developers use to build their plug-ins for Rhino. As a matter of fact, Grasshopper is a Rhino plugin that is completely written as a DotNET plugin using the very same SDK that accessible to these scripting components!

But why bother using the script components to start with? Indeed, you may never need to use one, but there are some cases when you might need to. One would be if you like to achieve a functionality that is otherwise not supported by other Grasshopper components. Or if you are writing a generative system that uses recursive functions such as fractals.

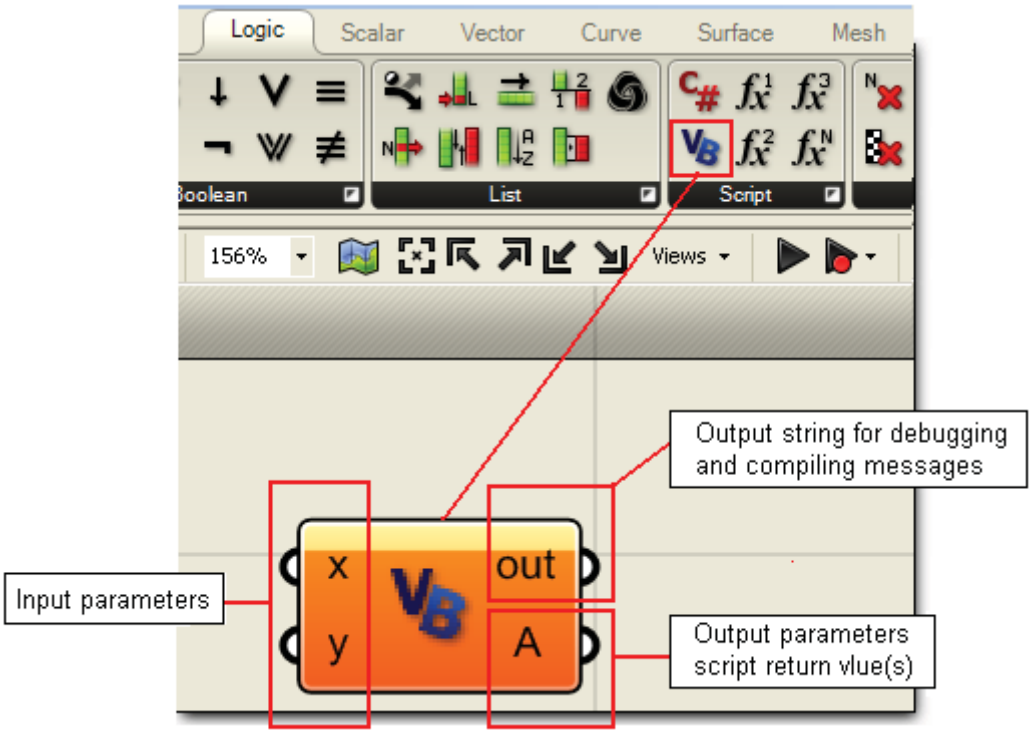
This Primer gives a general overview of how to use the scripting component in Grasshopper using VB DotNET programming language. It includes three sections. The first is about the script component interface. The second includes a quick review of VB DotNET language. The next section talks about Rhino DotNET SDK, geometry classes and utility functions. At the end, there is a listing of where you can go for further help.

13 The Scripting Interface

13.1 Where to find the Script Components

VB DotNet Script component is found under logic tab. Currently there are two script components. One is for writing Visual Basic code and the other for C#. There will no doubt be other scripting languages supported in the future.

To add a script component to the canvas, drag and drop the component icon.



The default script component has two inputs and two outputs. The user can change names, types and number of inputs and outputs.

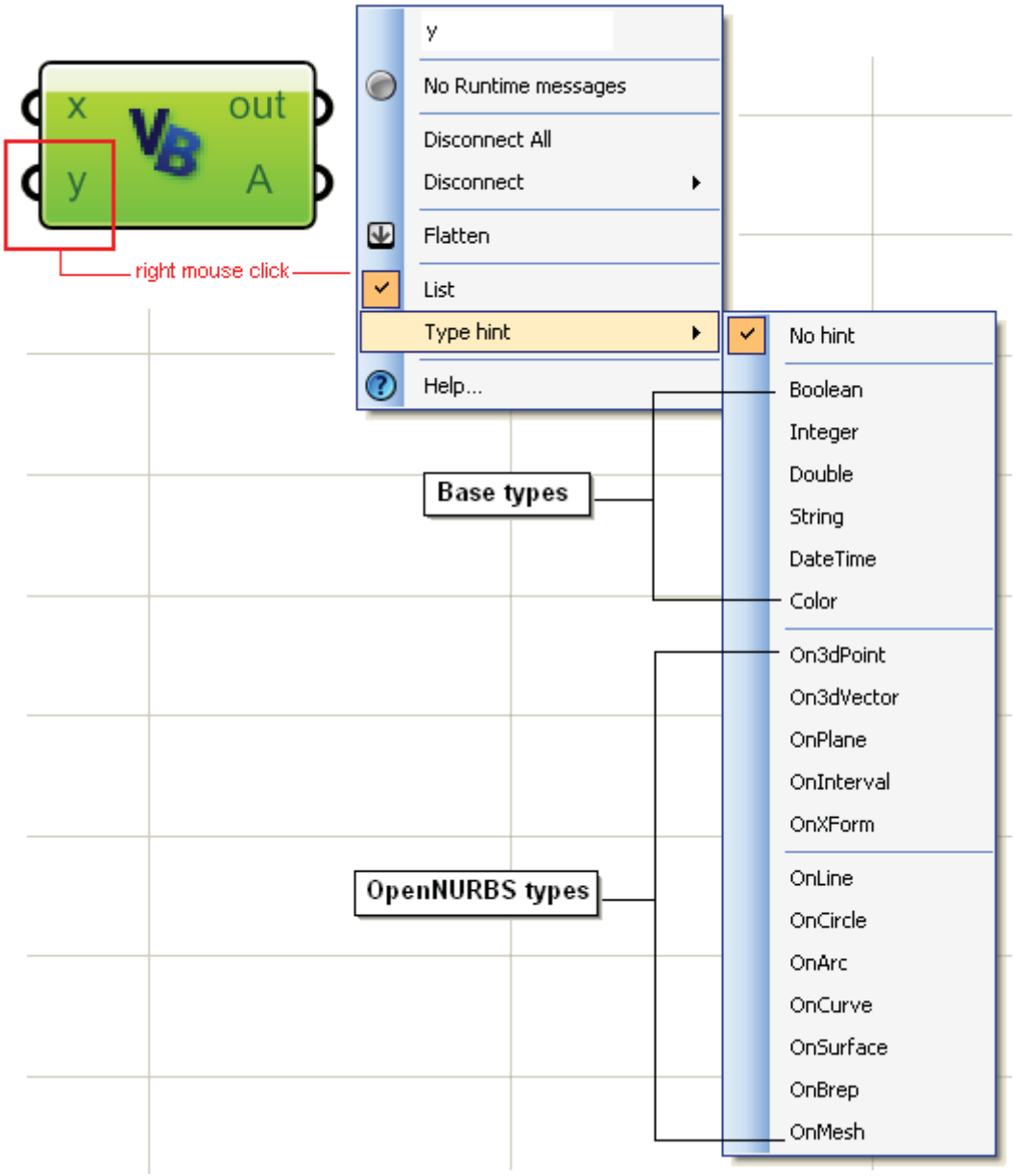
- **X**: first input of a generic type (object).
- **Y**: second input of a generic type (object).
- **Out**: output string with compiling messages.
- **A**: Returned output of type object.

13.2 Input Parameters

By default, there are two input parameters: x and y. It is possible to edit parameters names, delete or ad to them and also assign a type. If you right mouse click on any of the input parameters, you will see a menu that has the following:

- **Parameter name**: you can click on it and type new name.

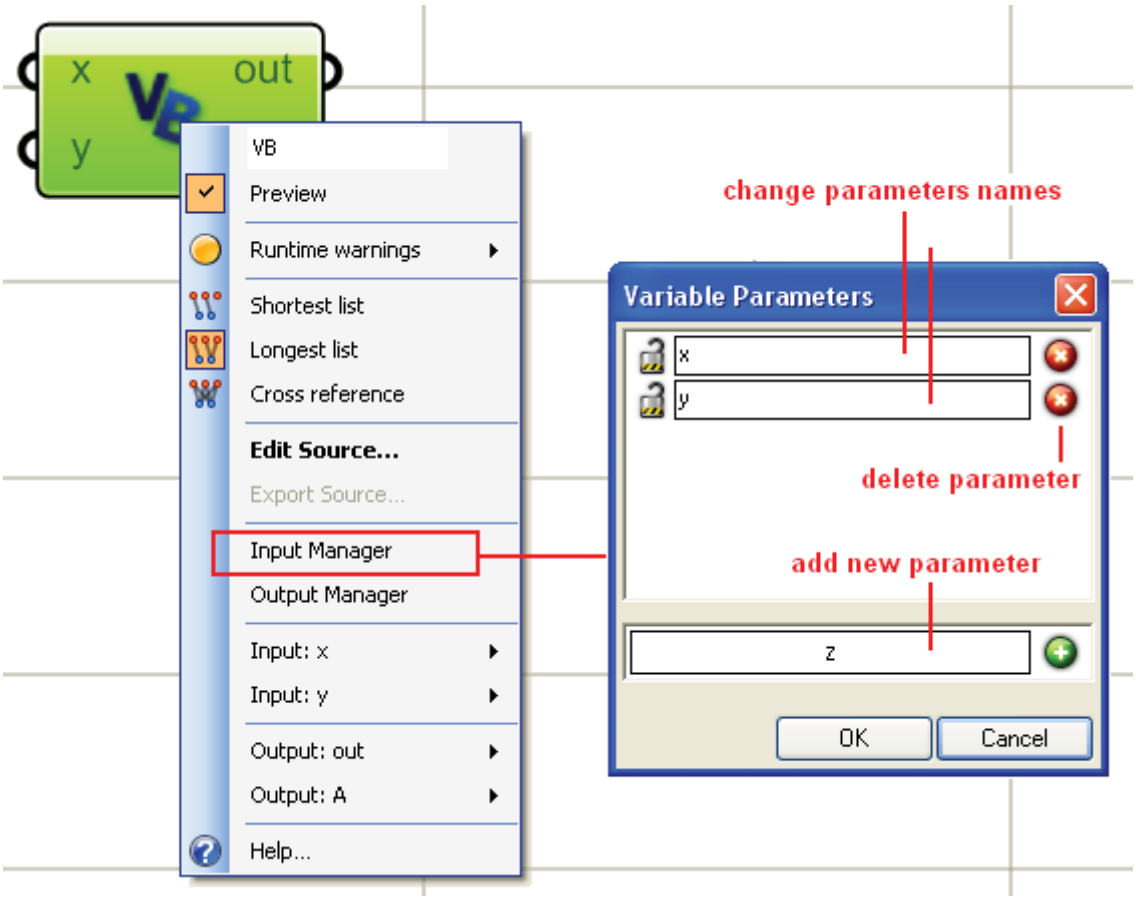
- **Run time message**: for errors and warnings.
- **Disconnect** and **Disconnect All**. Work the same as other Grasshopper components.
- **Flatten**: to flatten data. In case of a nested list of data, it converts it to single array of elements.
- **List**: to indicate if the input is a list of data.
- **Type hint**: Input parameters are set by default to the generic type “object”. It is best to specify a type to make the code more readable and efficient. Types that start with “On” are OpenNURBS types.



Input parameters can also be managed from the main component menu. If right mouse click in the middle of the component, you get a menu that has input and output details.

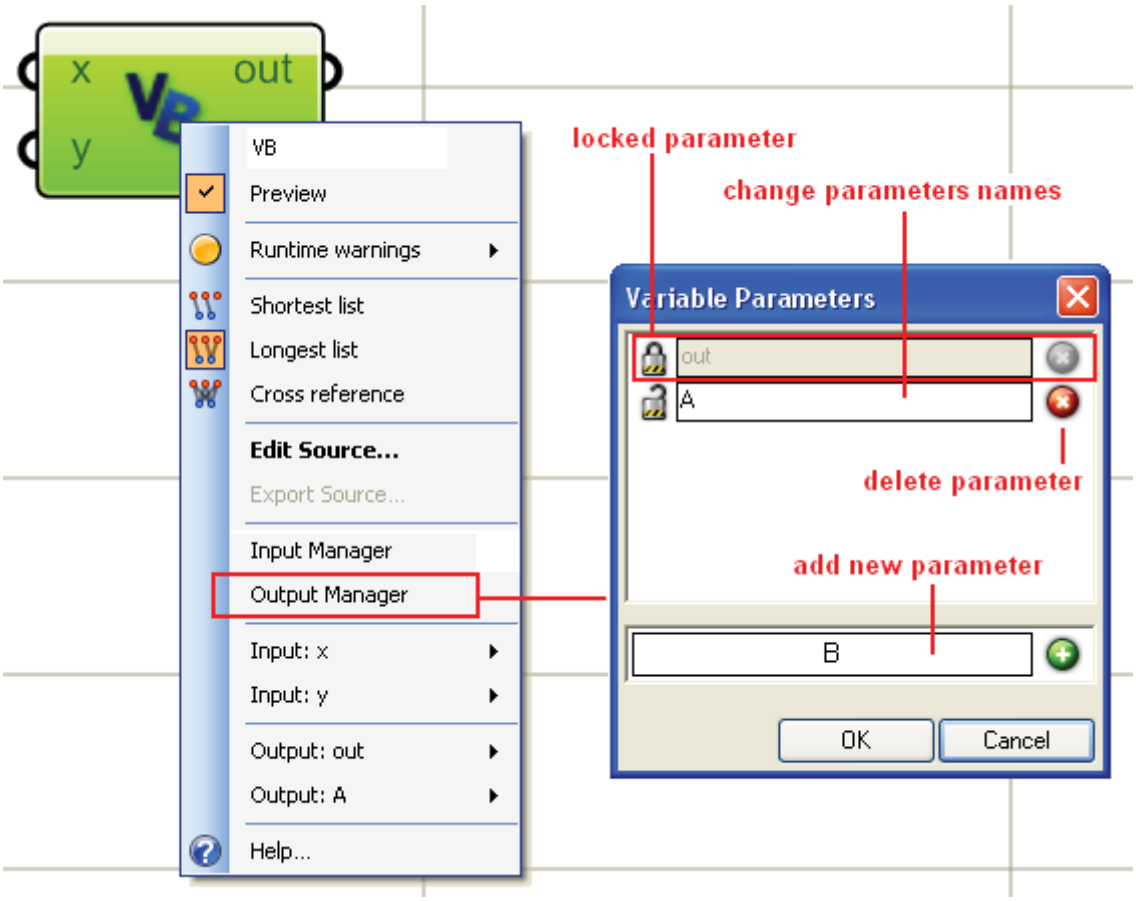
You may use this menu to open input manager and change parameters names, add new ones or delete as shown in the image.

Note that your scripting function signature (input parameters and their types) can only be changed through this menu. Once you start editing the source, only function body can be changed and not the parameters.



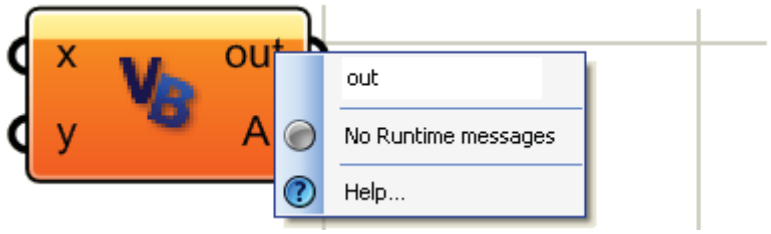
13.3 Output Parameters

You can also define as many outputs or returns as you wish using the main component menu. Unlike input parameters, there are no types associated with output. They are defined as the generic system type “object” and the function may assign any type, array or not to any of the outputs. Following picture shows how to set outputs using the output manager. Note that the “out” parameter cannot be deleted. It has debugging string and user debugging strings.

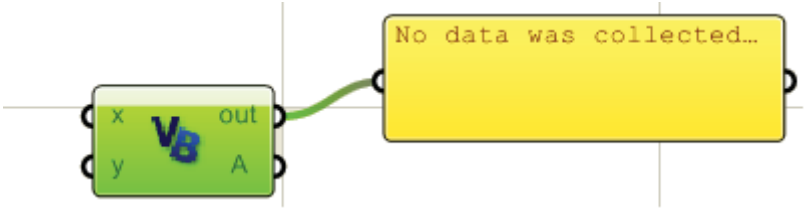


13.4 Out Window and Debug Information

Output window which is called “out” by default is there to provide debug information. It lists all compiling errors and warnings. The user can also print values to it from within the code to help debug. It is very useful to read compiling messages carefully when the code is not running correctly.



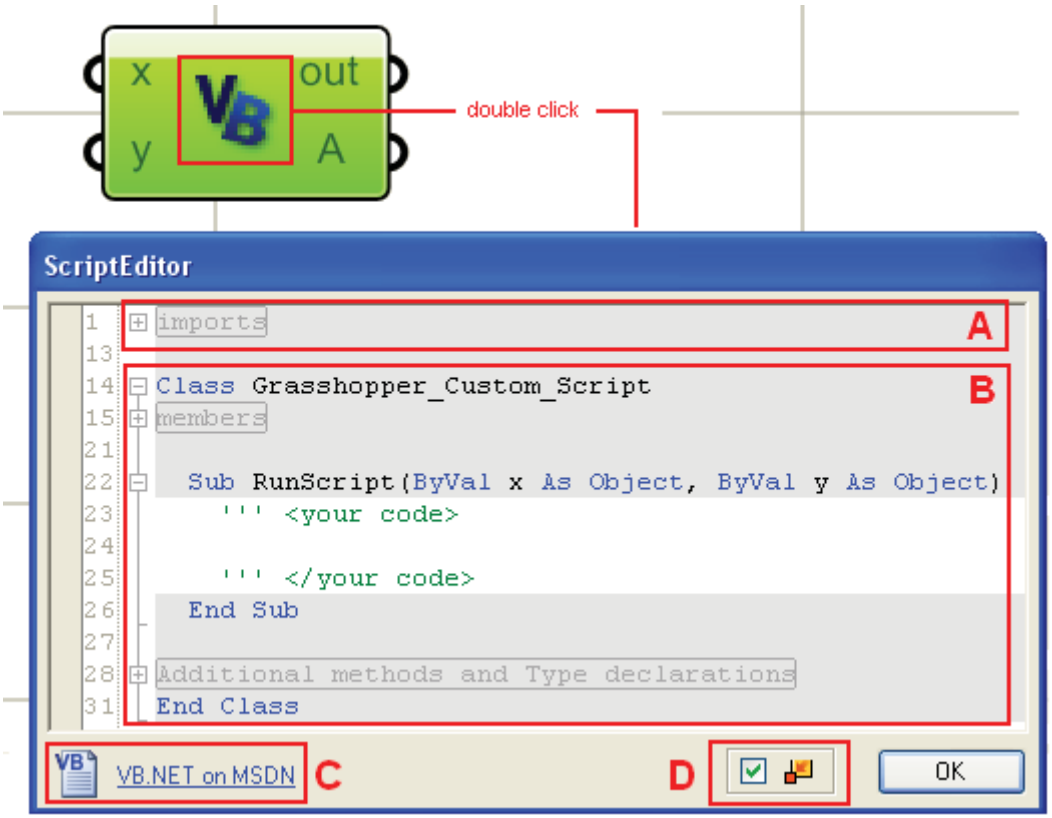
It is a good idea to connect the output string to a text component to see compiling messages and debug information directly.



13.5 Inside the Script Component

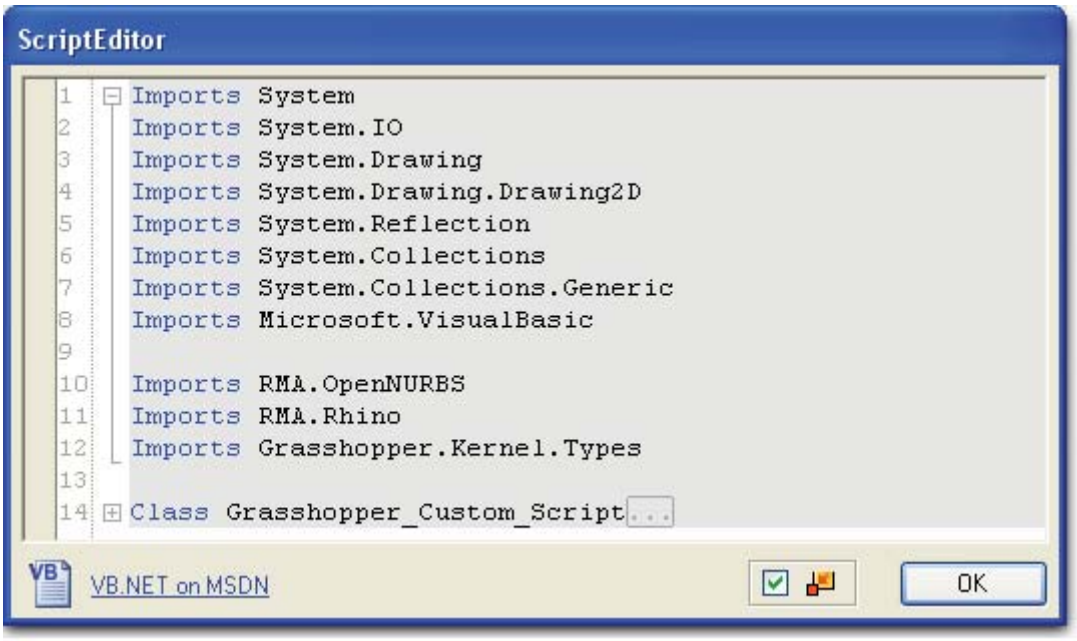
To open your script component, double click on the middle of the script component or select “Edit Source…” from the component menu. The script component consists of two parts. Those are:

A: Imports
B: Grasshopper_Custom_Script class.
C: Link to Microsoft developer network help on VB.NET.
D: Check box to activate the out of focus feature of scripting component.



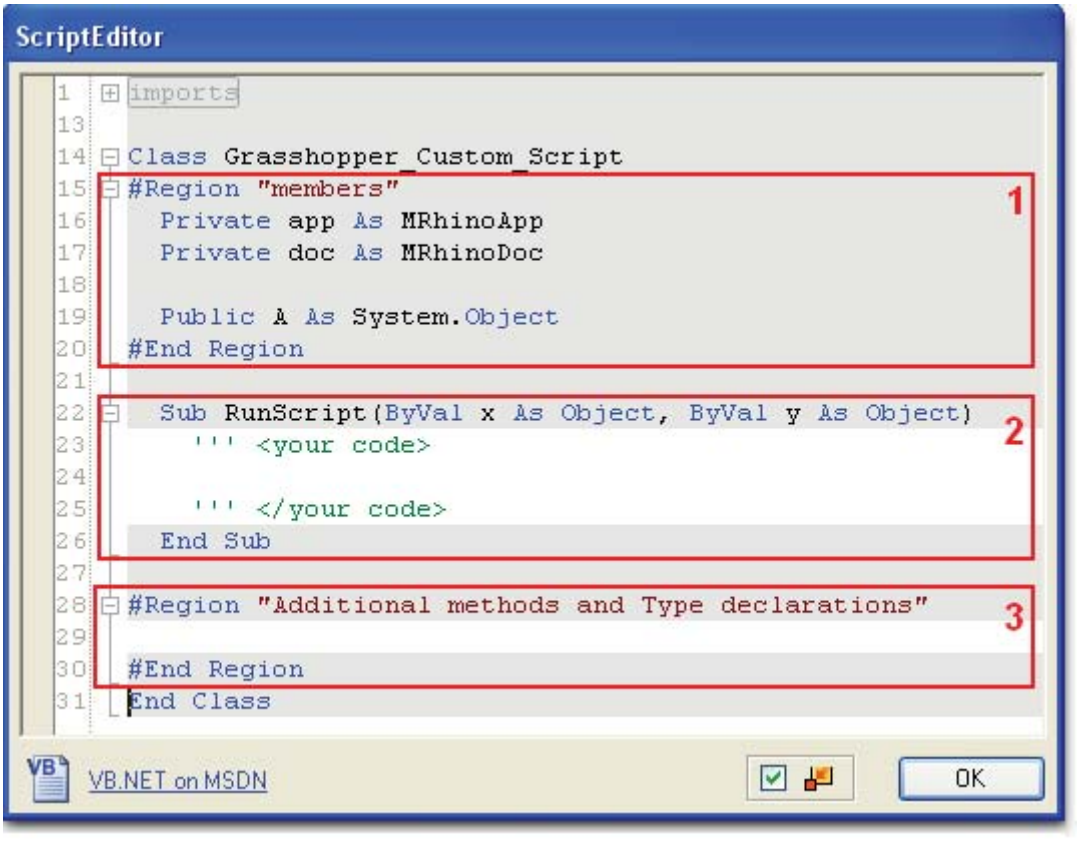
A: Imports

Imports are external dlls that you might use in your code. Most of them are DotNET system imports, but there is also the two Rhino dlls: RMA.openNURBS and RMA.Rhino. These include all rhino geometry and utility functions. There are also the Grasshopper specific types.



B: Grasshopper_Custom_Script Class

Grasshopper_Custom_Script class consists of three parts:

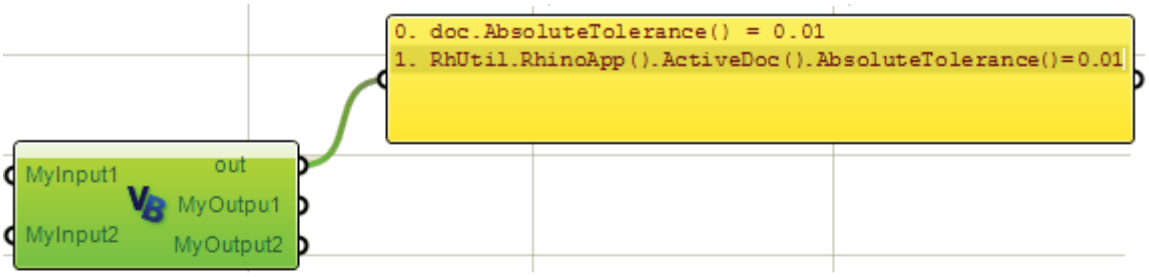


1. **Members**: This includes two references; one to the current rhino application (app) and the other to the active document (doc). Rhino application and document can also be accessed directly using RhinoUtil. Members region also includes return values or the output of the script function. Return is defined as a generic system type and the user cannot change that type.

2. **RunScript**: This is the main function that the user writes their code within.

3. **Additional methods and type declarations**: this is where you may put additional functions and types.

The following example shows two ways to access document absolute tolerance. The first is through using the document reference that comes with the script component (doc) and the second is through *RhUtil* (Rhino Utility Functions). Notice that when printing the tolerance value to the output window, both functions yield same result. Also note that in this example there are two outputs (MyOutput1 and myOutput2). They are listed in the members region of the script class.




```
Class Grasshopper_Custom_Script
#Region "members"
  Private app As MRhinoApp
  Private doc As MRhinoDoc

  Public MyOutput1, MyOutput2 As System.Object
#End Region

  Sub RunScript(ByVal MyInput1 As Object, ByVal MyInput2 As Object)
    ''' <your code>
    Dim tol As Double

    tol = doc.AbsoluteTolerance()
    Print(" doc.AbsoluteTolerance() = " & tol)

    tol = RhUtil.RhinoApp().ActiveDoc().AbsoluteTolerance()
    Print(" RhUtil.RhinoApp().ActiveDoc().AbsoluteTolerance()=" & tol)

    ''' </your code>
  End Sub
```

14 Visual Basic DotNET

14.1 Introduction

There are plenty of references about VB.NET available over the Internet and in print. The following is meant to be a quick review of the essentials that you will need to use in your code.

14.2 Comments

It is a very good practice to comment your code as much as possible. You'll be surprised how fast you will forget what you did! In VB.NET, you can use an apostrophe to signal that the rest of the line is a comment and the compiler should ignore. In Grasshopper, comments are grey color. For example:

```
'This is a comment… I can write anything I like!
'Really… anything
```

14.3 Variables

You can think of variables as containers of data. Different variables have different sizes depending on the type of the data they accommodate. For example an int32 variable reserves 32 bits in memory and the name of the variable in the name of that container. Once a variable is defined, the rest of the code can retrieve the content of the container using the name of that variable.

Let's define a container or variable called "x" of type Int32 and initialize it to "10". After that, let's assign the new integer value "20" to x. This is how it will looks in VB DotNET:

```
Dim x as Int32 = 10
'If you print the value of x at the point, then you will get 10
x = 20
'From now on, x will return 20
```

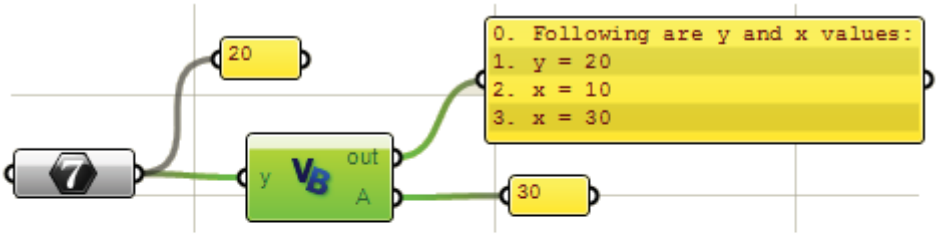
Here are other examples of commonly used types:

```
Dim x as Double = 20.4          'Dim keyword means that we are about to define a variable
Dim b as Boolean = True         'Double, Boolean and String are all examples of base or
Dim name as String = "Joe"      ' system defined types
```

The following Grasshopper example uses three variables:

x: is an integer variable that is defined inside the code.
y: is an integer variable passed to the function as an input.
A: is the output variable.

The example prints variable values to the output window throughout the code. As mentioned before, this is a good way to see what is happening inside your code and debugging to hopefully minimize the need to an external editor.




```
Sub RunScript(ByVal y As Integer)
  'Print variables values
  Print("Following are y and x values:")

  'Print input value (y)
  Print("y = " & y)

  'Declare x as an integer variable
  Dim x As int32 = 10

  'Print x initial value
  Print("x = " & x)

  'Set x value to be whatever was there plus input y
  x = x + y
  Print("x = " & x)

  'Assign x to output
  A = x
End Sub
```

Assigning meaningful variable names that you can quickly recognize will make the code much more readable and easier to debug. We will try to stick to some good coding practices throughout the examples in this chapter.

14.4 Arrays and Lists

There are many ways to define arrays in VB.NET. There are single or multi-dimensional arrays. You can define size of arrays or use dynamic arrays. If you know up front the number of elements in the array, you can declare defined-size arrays this way:

```
'One dimensional array
Dim myArray(1) As Integer
myArray (0) = 10
myArray (1) = 20
```

```
'Two-Dimensional array
Dim my2DArray (1,2) As Integer
my2DArray (0, 0) = 10
my2DArray (0, 1) = 20
my2DArray (0, 2) = 30
my2DArray (1, 0) = 50
my2DArray (1, 1) = 60
my2DArray (1, 2) = 70
```

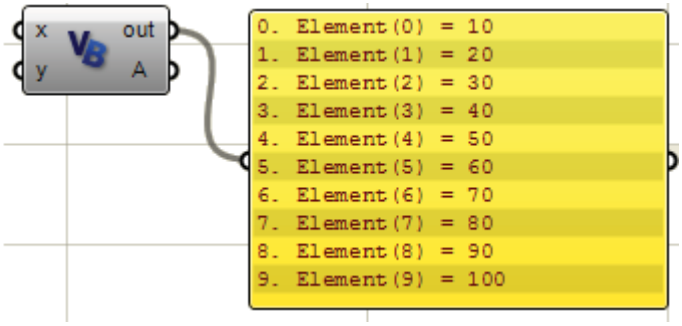
```
'Declare and assign values
Dim myArray() As Integer  = {10,20}
```

```
'Declare and assign values
Dim my2DArray(,)As Integer = {{10,20,30},{40,50,60}}
```

Keep in mind that arrays in VB.NET are zero based, so when you declare an array size to be (9) that means that the array has 10 elements. Same goes for multi-dimentional arrays.

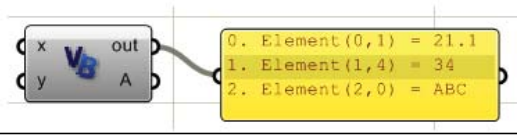
For single dimension dynamic arrays, you can declare a new “List” as shown in the following example and start adding elements to it.

```
Dim myList As New List(Of Integer)
For i As Integer = 1 To 10
  myList.Add(10 * i)
  Print("Element(" & i - 1 & ") = " & myList(i - 1))
Next
```

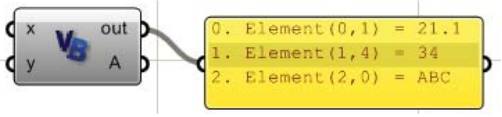


You can use nested List or ArrayList to declare dynamic multi-dimensional arrays of same or mixed types. Check the following example:

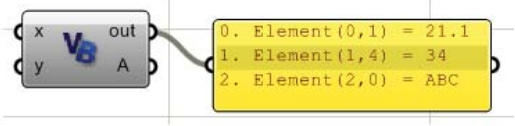
```
Dim myList As New ArrayList()
Dim myRow1() As Double = {20.1,21.1,22.1,23.1}
Dim myRow2() As Integer = {30,31,32,33,34}
Dim myRow3() As String = {"ABC", "DEF"}
```



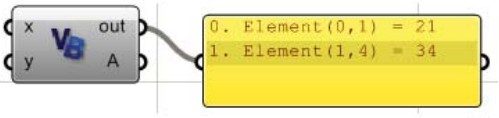

```
Dim myList As New ArrayList()
Dim myRow1 As New List( Of Double )
MyRow1.Add(20.1): MyRow1.Add(21.1): MyRow1.Add(22.1): MyRow1.Add(23.1)
Dim myRow2 As New List( Of Integer )
MyRow2.Add(30): MyRow2.Add(31): MyRow2.Add(32): MyRow2.Add(33): MyRow2.Add(34)
Dim myRow3 As New List( Of String )
MyRow3.Add("ABC"): MyRow3.Add("DEF")
```




```
Dim myList As New List(Of Object)
Dim myRow1 As New List( Of Double )
MyRow1.Add(20.1): MyRow1.Add(21.1): MyRow1.Add(22.1): MyRow1.Add(23.1)
Dim myRow2 As New List( Of Integer )
MyRow2.Add(30): MyRow2.Add(31): MyRow2.Add(32): MyRow2.Add(33): MyRow2.Add(34)
Dim myRow3 As New List( Of String )
MyRow3.Add("ABC"): MyRow3.Add("DEF")
```




```
Dim myList As New List(Of List(Of Integer))
Dim myRow1 As New List( Of Integer )
MyRow1.Add(20): MyRow1.Add(21): MyRow1.Add(22): MyRow1.Add(23)
Dim myRow2 As New List( Of Integer )
MyRow2.Add(30): MyRow2.Add(31): MyRow2.Add(32): MyRow2.Add(33): MyRow2.Add(34)
```



Add Lists and Print:

```
myList.Add(myRow1)
myList.Add(myRow2)
myList.Add(myRow3)

Print("Element(0,1) = " & myList(0)(1))
Print("Element(1,4) = " & myList(1)(4))
Print("Element(2,0) = " & myList(2)(0))
```

14.5 Operators

There are many built-in operators in VB.NET. They operate on one or more operands. Here is a table of the common operators for quick reference:

Type	Operator	Description
Arithmetic Operators	**^**	Raises a number to the power of another number.
	*****	Multiplies two numbers.
	/	Divides two numbers and returns a floating-point result.
	****	Divides two numbers and returns an integer result.
	Mod	Divides two numbers and returns only the remainder.
	+	Adds two numbers or returns the positive value of a numeric expression.
	-	Returns the difference between two numeric expressions or the negative value of a numeric expression
Assignment Operators	**=**	Assigns a value to a variable
	^=	Raises the value of a variable to the power of an expression and assigns the result back to the variable.
	***=**	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.
	/=	Divides the value of a variable by the value of an expression and assigns the floating-point result to the variable.
	\=	Divides the value of a variable by the value of an expression and assigns the integer result to the variable.
	+=	Adds the value of a numeric expression to the value of a numeric variable and assigns the result to the variable. Can also be used to concatenate a String expression to a String variable and assign the result to the variable.
	-=	Subtracts the value of an expression from the value of a variable and assigns the result to the variable.
	&=	Concatenates a String expression to a String variable or property and assigns the result to the variable or property.
Comparison Operators	**<**	Less Than
	<=	Less or equal
	>	Greater than
	>=	Greater or equal
	=	Equal
	<>	Not equal
Concatenation Operators	**&**	Generates a string concatenation of two expressions.
	+	Concatenate two string expressions.
Logical Operators	**And**	Performs a logical conjunction on two Boolean expressions
	Not	Performs logical negation on a Boolean expression
	Or	Performs a logical disjunction on two Boolean expressions
	Xor	Performs a logical exclusion on two Boolean expressions

14.6 Conditional Statements

You can think of conditional statements as blocks of code with gates that get executed only when the gate condition is met. The conditional statement that is mostly used is the "if" statement with the following format "**IF<condition> Then <code> End IF**".

```
'Single line if statement doesn't need End If: condition=(x<y), code=(x=x+y)
If x < y Then x = x + y

'Multiple line needs End If to close the block of code
If x < y Then
   x = x + y
End If
```

It is also possible to use an "**Else If … Then**" and "**Else**" to choose n alternative code block to execute. For example:

```
If x < y Then
   x = x + y          'execute this line then go to the step after "End If"
Else If x > y Then
   x = x – y          'execute this line then go to the step after "End If"
Else
   x = 2*x            'execute this line then go to the step after "End If"
End If
```

There is also the "Select Case" statement. It is used to execute different blocks of code based on the value of an expression ("index" in the example). For example:

```
Select Case index
   Case 0             'If index=0 the execute next line, otherwise go directly to the next case
      x = x * x
   Case 1
      x = x ^ 2
   Case 2
      x = x ^ (0.5)
End Select
```

14.7 Loops

Loops allow repeating the execution of the code block inside the body of the loop again and again as long as the loop condition is met. There are different kinds of loops. We will explain two of them that are most commonly used.

"For … Next" Loop

This is the most common way of looping. The structure of the loop looks like:

```
For < index = start_value> To <end_value> [Step <step_value>]

   'For loop body starts here
    [ statements/code to be executed inside the loop]

  [ Exit For ]   'Optional: to exit the loop at any point

    [ other statements]
```

```
    [ Continue For ]     'optional: to skip executing the remaining of the loop statements.

    [ other statements]

    'For loop body ends here (just before the "Next")
    'Next means: go back to the start of the for loop to check if index has passed end_value
    'If index passed end_value, then exit loop and execute statements following "Next"
    'Otherwise, increment the index by "step_value"
Next
[ statements following the For Loop ]
```

The following example uses a loop to iterate through an array of places:

```
'Array of places
Dim places_list As New List( of String )
places_list.Add( "Paris" )
places_list.Add( "NY" )
places_list.Add( "Beijing" )

'Loop index
Dim i As Integer
Dim place As String
Dim count As Integer = places_list.Count()

'Loop starting from 0 to count -1 (count = 3, but last index of the places_list = 2)
For i=0 To count-1
   place = places_list(i)
   Print( place )
Next
```

If looping through objects in an array, you can also use the *For…Next* loop to iterate through elements of the array without using an index. The above example can be rewritten as follows:

```
Dim places_list As New List( of String )
places_list.Add( "Paris" )
places_list.Add( "NY" )
places_list.Add( "Beijing" )

For Each place As String In places_list
   Print( place )
Next
```

"While … End While" Loop

This is also a widely used loop. The structure of the loop looks like:

```
While < some condition is True >

    'While loop body starts here
    [ statements to be executed inside the loop]

  [ Exit While ]          'Optional to exit the loop

    [ other statements]

    [ Continue While ]  'optional to skip executing the remaining of the loop statements.
```

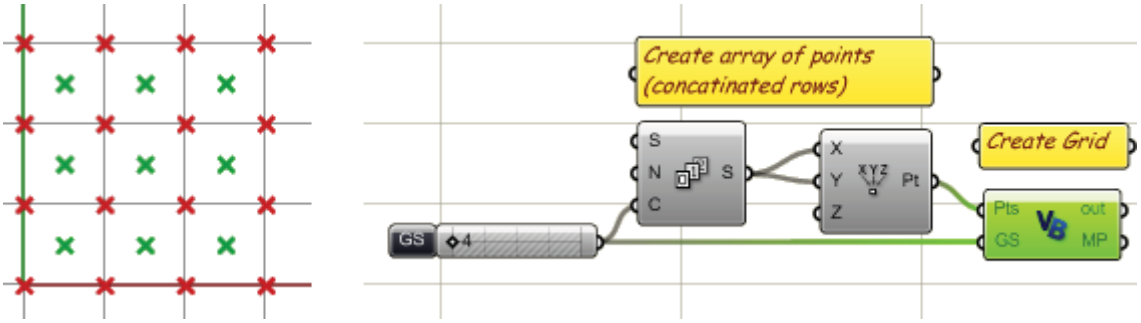
```
    [ other statements]

    'While loop body ends here
    'Go back to the start of the loop to check if the condition is still true, then execute the body
    'If condition not true, then exit loop and execute statements following "End While"

End While
[ statements following the While Loop ]
```

Here is the previous places example re-written using while loop:

```
Dim places_list As New List( of String )
places_list.Add( "Paris" )
places_list.Add( "NY" )
places_list.Add( "Beijing" )

Dim i As Integer
Dim place As String
Dim count As Integer = places_list.Count()

i = 0
While i < count              '(i<count) evaluates to true or false
   place = places_list(i)
   Print( place )
   i = i + 1
End While
```

14.8 Nested Loops

Nested loops are a loop that encloses another loop in its body. For example if we have a grid of points, then in order to get to each point in the list, we need to use a nested loop. The following example shows how to turn a single dimensional array of points into a 2-dimensional grid. It then processes the grid to find cells mid points.



The script has two parts:

- First convert a single dimension array to 2-dimension array we call "Grid".
- Second process the grid to find cells mid points.

In both parts we used nested loops. Here is the script definition in Grasshopper:

```
Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)

    'Create a grid of points
    Dim Grid As New ArrayList()

    Dim i As Integer
    Dim j As Integer

    'Nested loop to covert 1D array to 2D grid
L1  For i = 0 To Pts.Count() - 1 Step GS
      'Declare a row of points
      Dim Row As New List( Of On3dPoint )
L2    For j = i To i + GS - 1
        'Get a reference od the point
        Dim pt As On3dPoint
        pt = Pts(j)

        'Add point to the row
        Row.Add(pt)
      Next
      'Add row to the grid
      Grid.Add(Row)
    Next

    'Process the grid to find mid points of cells
    Dim mid_points As New List( Of On3dPoint )
L1  For i = 1 To Grid.Count() - 1
      'Get first and second rows
      Dim Row0 As List( Of On3dPoint )
      Row0 = Grid(i - 1)
      Dim Row1 As List( Of On3dPoint )
      Row1 = Grid(i)

L2    For j = 1 To Row0.Count() - 1
        Dim mid_pt As New On3dPoint
        mid_pt = (Row0(j-1) + Row0(j) + Row1(j-1) + Row1(j)) / 4
        mid_points.Add(mid_pt)
      Next
    Next

    'Assign mid point to output
    MP = mid_points
End Sub
```

14.9 Subs and Functions

RunScript is the main function that all script components use. This is what you see when you open a default script component in Grasshopper:

```
Sub RunScript(ByVal x As Object, ByVal y As Object)
        '''<your code…>
End Sub
```

Sub… End Sub: Are keywords that enclose function block of code
"**RunScript**": is the name of the sub
"**(…)**": Parentheses after the sub name enclose the input parameters
"**ByVal x As Object,…**": Are what is called input parameters

Each input parameter needs to define the following:

- **ByRef** or **ByVal**: Specify if a parameter is passed by value or by reference.
- **Name** of the parameter.
- Parameter **type** preceded by **"As"** keyword.

Notice that input parameters in Grasshopper RunScript sub are all passed by value (***ByVal*** key word). That means they are copies of the original input and any change of these parameters inside the script will not affect the original input. However if you define additional subs/functions inside your script component, you can pass parameters by reference (***ByRef***). Passing parameters by reference means that any change of these parameter inside the function will change the original value that was passed when the function exits.

You maybe able to include all your code inside the body of RunScript, however, you can to define external subs and functions if needed. But why use external functions?

- To simplify the main function code.
- To make the code more readable.
- To isolate and reuse common functionality.
- To define specialized functions such as recursion.

What is the difference between a **Sub** and **Function** anyway? You define a Sub if you don't need a return value. Functions allow you to return one result. You basically assign a value to that function name. For example:

```
Function AddFunction( ByVal x As Double, ByVal y As Double )
        AddFunction = x + y
End Function
```

That is said, you don't have to have a function to return a value. Subs can do it through input parameters that are passed "ByRef". In the following, "**rc**" is used to return result:

```
Sub AddSub( ByVal x As Double, ByVal y As Double, ByRef rc As Double )
        rc = x + y
End Sub
```

Here is how the caller function looks like using the Function and the Sub::

```
Dim x As Double = 5.34
Dim y As Double = 3.20
Dim rc As Double = 0.0
'Can use either of the following to get result
rc = AddFunction( x, y )     'Assign function result to "rc"
AddSub( x, y, rc )           'rc is passed by refernce and will have addition result
```

In the nested loops section, we illustrated and example that created a grid from a list of points then calculated mid points. Each of these two functionalities is distinct enough to

separate in an external sub and probably reuse in future code. Here is a re-write of the grid example using external functions.

```
  Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)

    'Create a grid of points
    Dim Grid As New ArrayList()

    'Call grid function
1   Call CreateGrid(Pts, Grid, GS)

    'Call mid points function
    Dim mid_points As New List( Of On3dPoint )
2   Call FindMidPoints(Grid, mid_points)

    'Assign mid point to output
    MP = mid_points
  End Sub

#Region "Additional methods and Type declarations"

  'Function to convert 1d array to 2d array
1 Sub CreateGrid( ByVal Pts As List(...

  'Function to find grid mid points
2 Sub FindMidPoints(ByVal Grid As ArrayList, mid_points As List(...

#End Region
End Class
```

Here is how each of the two subs looks when expanded:

```
#Region "Additional methods and Type declarations"

  'Function to convert 1d array to 2d array
  Sub CreateGrid( ByVal Pts As List(Of On3dPoint), ByRef Grid As ArrayList, ByVal GS As Integer )

    Dim i As Integer
    Dim j As Integer

    For i = 0 To Pts.Count() - 1 Step GS
      'Declare a row of points
      Dim Row As New List( Of On3dPoint )
      For j = i To i + GS - 1
        'Get a reference od the point
        Dim pt As On3dPoint
        pt = Pts(j)

        'Add point to the row
        Row.Add(pt)
      Next
      'Add row to the grid
      Grid.Add(Row)
    Next

  End Sub
```

```
'Function to find grid mid points
Sub FindMidPoints(ByVal Grid As ArrayList, mid_points As List(Of On3dPoint ))

  Dim i As Integer
  Dim j As Integer

  For i = 1 To Grid.Count() - 1
    'Get first and second rows
    Dim Row0 As List( Of On3dPoint )
    Row0 = Grid(i - 1)
    Dim Row1 As List( Of On3dPoint )
    Row1 = Grid(i)

    For j = 1 To Row0.Count() - 1
      Dim mid_pt As New On3dPoint
      mid_pt = (Row0(j - 1) + Row0(j) + Row1(j - 1) + Row1(j)) / 4
      mid_points.Add(mid_pt)
    Next
  Next

End Sub
#End Region
```

14.10 Recursion

Recursive functions are special type functions that call themselves until some stopping condition is met. Recursion is commonly used for data search, subdividing and generative systems. We will discuss an example that shows how recursion works. For more recursive examples, check Grasshopper wiki and the Gallery page.

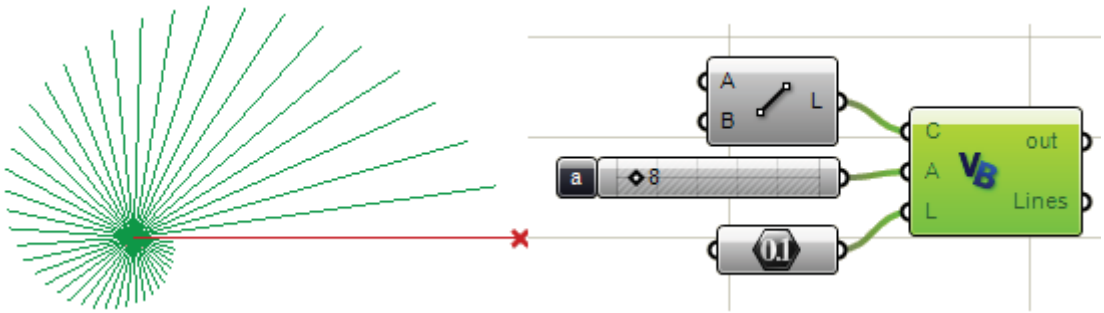
The following example takes smaller part of an input line and rotates it by a given angle. It keeps doing it until line length becomes less than a minimum length.

Input parameters are

- Staring line (C).
- Angle in radians (A). Slider shows angle in degrees, but converts it to radians.
- Minimum length (L) – As a stopping condition.

Output is:

- Array of lines.



We will solve same example iteratively as well as recursively for the sake of comparison.

Recursive solution. Note that inside the “DivideAndRotate” sub there are:

- Stopping condition to exit the sub.
- A call to the same function (recursive function call themselves).
- “AllLines” (array of lines) is passed by reference to keep adding new lines to the list.

```
Sub RunScript(ByVal C As OnLine, ByVal A As Double, ByVal L As Double)

  'Declare all lines
  Dim AllLines As New List( Of OnLine )

  'Call recursive function
  Call DivideAndRotate(Line, AllLines, A, L)

  'Assign return value
  Lines = AllLines
End Sub

#Region "Additional methods and Type declarations"

  Sub DivideAndRotate(ByVal Line As OnLine,
                      ByRef AllLines As List(Of OnLine),
                      ByVal angle As Double,
                      ByVal MinLength As Double)

    'Check the stopping condition
    If Line.Length() < MinLength Then Exit Sub

    'Take a portion of the line
    Dim new_line As New OnLine(Line)
    Dim end_pt As New On3dPoint
    end_pt = new_line.PointAt(0.95)

    new_line.To = end_pt

    'Rotate
    new_line.Rotate(angle, OnUtil.On_zaxis, Line.from)

    AllLines.Add(new_line)

    'Call self
    Call DivideAndRotate(new_line, AllLines, angle, MinLength)

  End Sub

#End Region
```

This is the same functionality using iterative solution using a “while” loop:

```
Sub RunScript(ByVal C As OnLine, ByVal A As Double, ByVal L As Double)

    'Declare all lines
    Dim AllLines As New List( Of OnLine )

    'Find current length
    Dim current_L As Double = C.Length()

    Dim new_line As OnLine
    new_line = C

    'Loop until length is less than min length
    While current_L > L
      'Generate the new line
      new_line = DivideAndRotate(new_line, A)

      'Add to list
      AllLines.Add(new_line)

      'Stopping condition
      current_L = new_line.Length()
    End While

    'Assign return value
    Lines = AllLines
  End Sub

#Region "Additional methods and Type declarations"

 Function DivideAndRotate(ByVal L As OnLine, ByVal A As Double) As OnLine

    'Take a portion of the line
    Dim new_line As New OnLine(L)
    Dim end_pt As New On3dPoint
    end_pt = new_line.PointAt(0.95)

    new_line.To = end_pt

    'Rotate
    new_line.Rotate(A, OnUtil.On_zaxis, L.from)

    'Function return
    DivideAndRotate = new_line

 End Function

#End Region
```

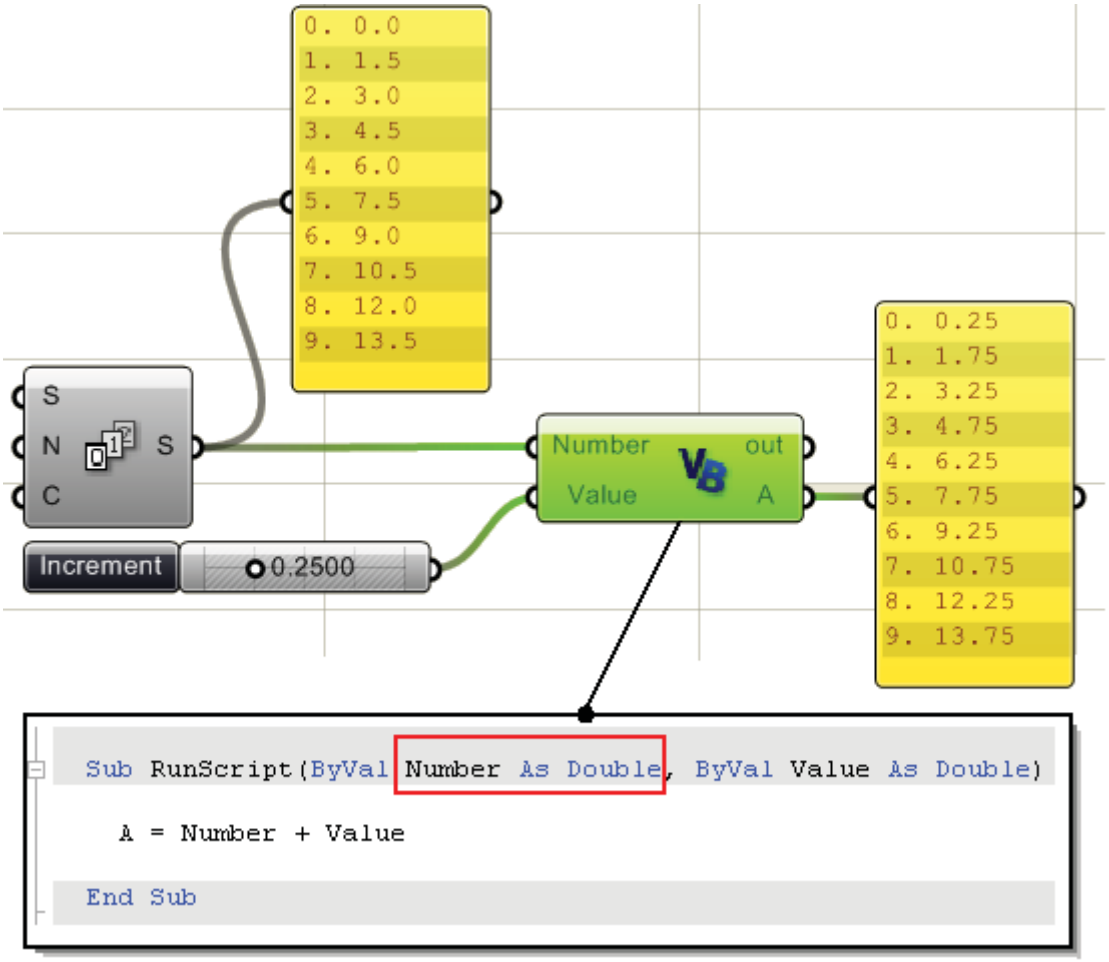
14.11 Processing Lists in Grasshopper

Grasshopper script component can process list of input in two ways:

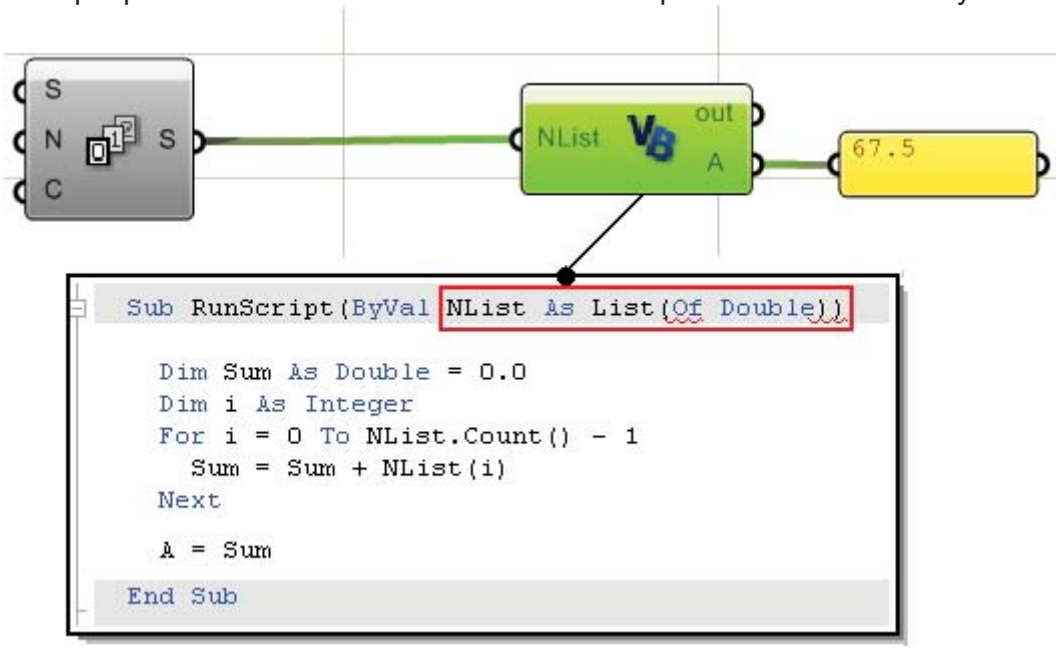
1. Process one input value at a time (component is called number of times equal to number of values in the input array).
2. Process all input values together (component is called once).

If you need to process each element in a list independently from the rest of the elements, then using first approach is easier. For example if you have a list of numbers that you like to increment each one of them by say “10”, then you can use the first approach. But if you need a sum function to add all elements, then you will need to use the second approach and pass the whole list as an input.

The following example shows how to process a list of data using the first methods possessing one input value at a time. In this case the RunScript function is called 10 times (once for each number). Key thing to note is the “Number” input parameter is passed as a “Double” as opposed to “List(of Double)” as we will see in the next example.



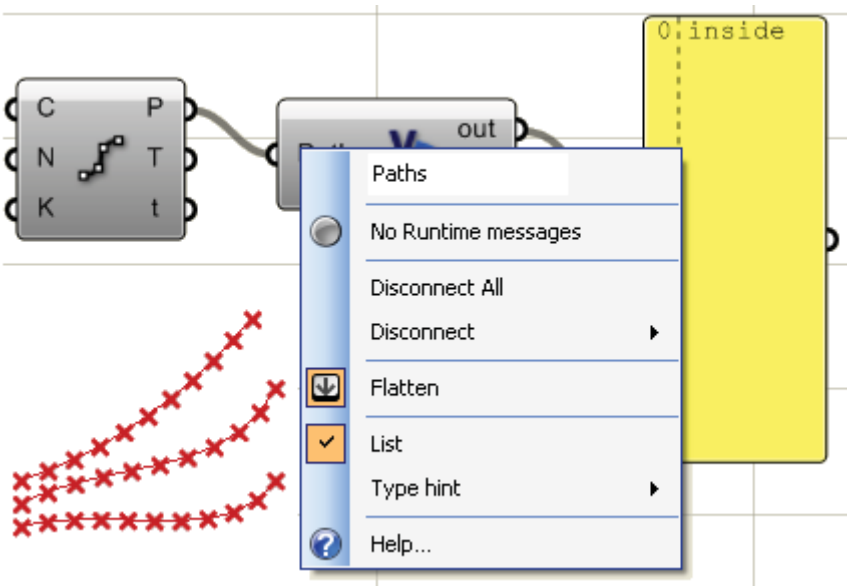
In the following, we input the list of numbers. You can do that by right mouse click on the input parameter and check “List”. The RunScript function is called only once.



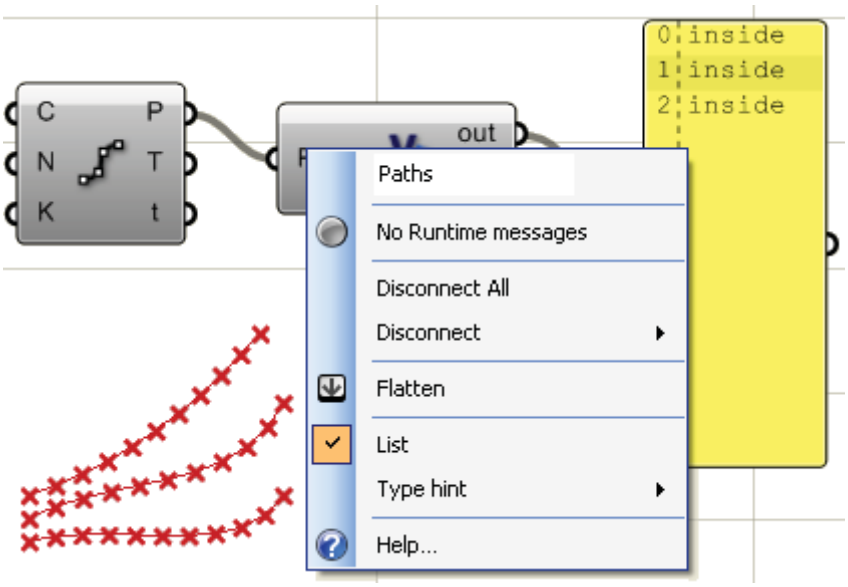
14.12 Processing Trees in Grasshopper

Trees (or multi-dimensional data) can be processed one element at a time or one branch (path) at a time or all paths at ones. For example if we divide three of curves into ten segments each, then we get a structure of three braches or paths, each with a eleven points. If we use this as an input then we get the following:

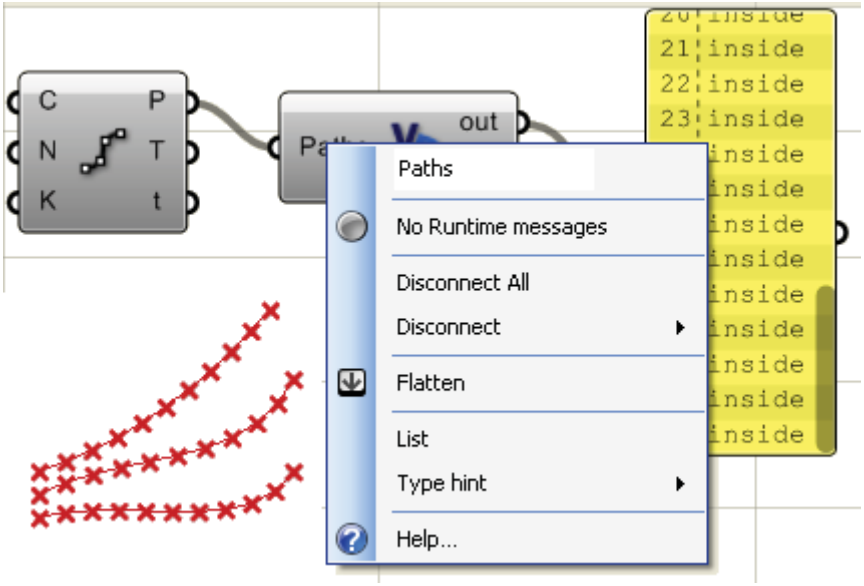
A: If both “Flatten” and “List” are checked, then the component is called once and flat list of all points is passed:



B: If “List” only is checked, then the component is called three times in each time, a list of divide points of one curve is passed:



C: When nothing is checked, then the function is called once for each divide points (the function is called 33 in this case). The same thing happens if only “Flatten” is checked:



This is the code inside the VB component. Basically just print the word “inside” to indicate that the component was called:

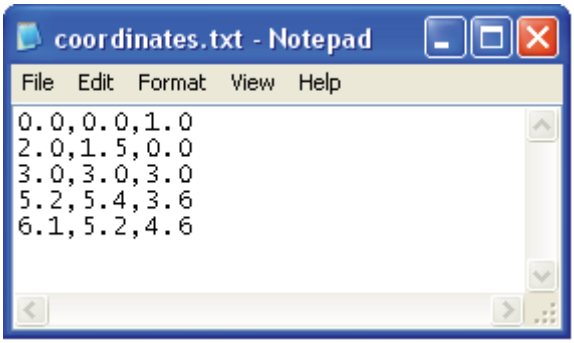
```
Sub RunScript(ByVal Paths As Object)

  Print("inside")

End Sub
```

14.13 File I/O

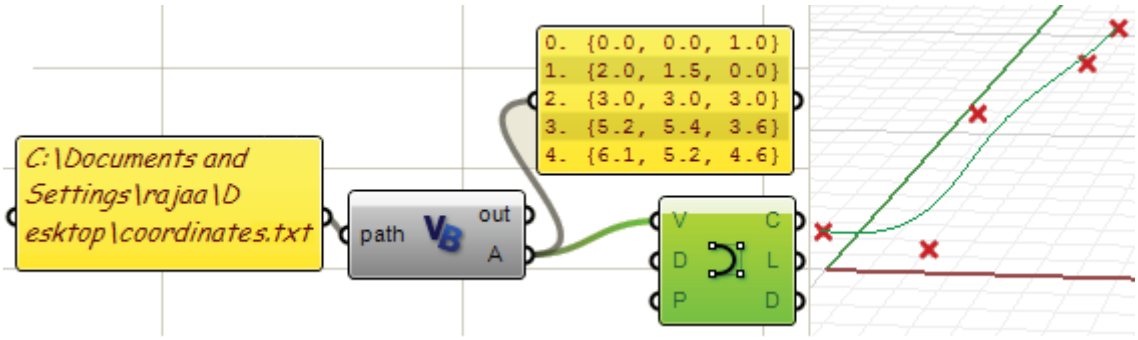
There are many ways to read from and write to files in VB.NET and many tutorials and documents are available in the internet and printed material. In general, reading a file involves the following:

- Open the file. Generally you need a path to point to.
- Read the string (whole string or line by line).
- Tokenize the string using some delimiter character(s).
- Cast each token (to double in this case).
- Store result.

We will illustrate a simple example that parses points from a text file. Using the following text file format, we will read each line as a single point and use the first value as x coordinate, the second as y and the third as z of the point. We will then use these points as curve control points.



The VB component accepts as an input a string which is the path of the file to read and output On3dPoints.



Here is the code inside the script component. There are few error trapping code to make sure that the file exits and has content:

```
Sub RunScript(ByVal path As String)

  'Check if file exists
  If (Not IO.File.Exists(path)) Then
    Print("Exit without reading")
    Return
  End If

  'Read the file
  Dim lines As String() = IO.File.ReadAllLines(path)

  'Check that file is not empty
  If (lines Is Nothing) Then
    Print("File has no content")
    Return
  End If

  'Declare list of points
  Dim pts As New List(Of On3dPoint)

  'Loop through lines
  For Each line As String In lines
    'Tokenize line into array of strings separated by ","
    Dim parts As String() = line.Split(",".ToCharArray())

    'Make sure that each line has exactly 3 values
    If UBound(parts) <> 2 Then Continue For

    'Convert each coordinate from string to double
    Dim x As Double = Convert.ToDouble(parts(0))
    Dim y As Double = Convert.ToDouble(parts(1))
    Dim z As Double = Convert.ToDouble(parts(2))

    pts.Add(New On3dPoint(x, y, z))
  Next

  A = pts

End Sub
```

15 Rhino .NET SDK

15.1 Overview

Rhino .NET SDK provides access to OpenNURBS geometry and utility functions. There is a help file that comes when you download .NET SDK. It is a great resource to use. This is where you can get it from:
http://en.wiki.mcneel.com/default.aspx/McNeel/Rhino4DotNetPlugIns.html

In this section, we will focus on the part of the SDK dealing with Rhino geometry classes and utility functions. We will show examples about how to create and manipulate geometry using Grasshopper VB script component.

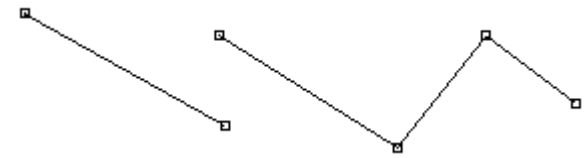
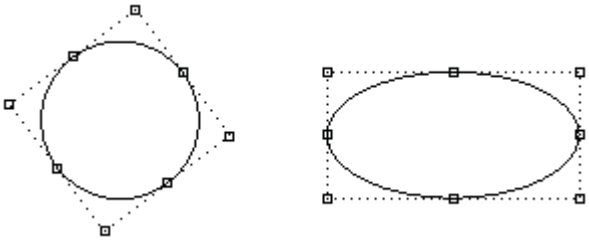
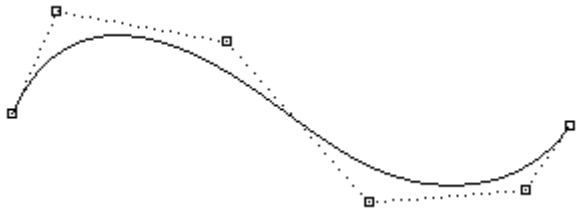
15.2 Understanding NURBS

Rhino is a NURBS modeler that defines curves and surfaces geometry using Non-Uniform Rational Basis Spline (or NURBS for short). NURBS is an accurate mathematical representation of curves and surfaces that is highly intuitive to edit.

There are many books and references for those of you interested in an in-depth reading about NURBS (http://en.wikipedia.org/wiki/NURBS). A basic understanding of NURBS is necessary to help you use the SDK classes and functions more effectively.

There are four things that define a nurbs curve. Degree, control points, knots and evaluation rules:

Degree
It is a whole positive number that is usually equal to 1,2,3 or 5. Rhino allows working with degrees 1-11. Following are few examples of curves and their degree:

	**Lines** and **polylines** are degree 1 nurbs curves. Order = 2 (order = degree + 1)
	**Circles** and **ellipses** are examples of degree 2 nurbs curves. They are also rational or non-uniform curves. Order = 3.
	Free form **curves** are usually represented as degree 3 nurbs curves. Order = 4 Degree 5 is also common, but the rest are pretty much hypothetical.

Control points
Control points of a NURBS curve is a list of at least (degree+1) points. The most common way to change the shape of a nurbs curve is through moving its control points. Control points have an associated number called a **weight**. With a few exceptions, weights are positive numbers. When a curve's control points all have the same weight (usually 1), the curve is called non-rational and. We will have an example showing how to change the weights of control points interactively in Grasshopper.

Knots or knot vector
Each NURBS curve has a list of numbers associated with it that is called a knot vector. Knots are a little harder to understand and set, but luckily there are SDK functions that do the job for you. Nevertheless, there are few things that will be useful to learn about the knot vector:

Knot multiplicity
Number of times a knot value is duplicated is called the knot's multiplicity. Any knot value can not be duplicated more than the curve degree. Here are few things that would be good to know about knots.

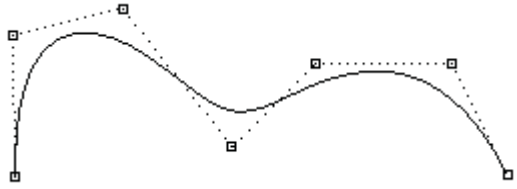
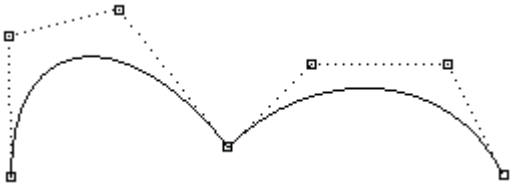
Full multiplicity knot is a knot duplicated number of times equal to curve degree. Clamped curves have knots with full multiplicity at the two ends of the curve and this is why end control points coincide with curve end points. If there were full multiplicity knot in the middle of the knot vector, then the curve will go through the control point and there would be a kink.

Simple knot: is a knot with value appearing only once.

Uniform knot vector satisfies 2 conditions:

1. Number of knots = number of control points + degree – 1.
2. Knots start with a full multiplicity knot, is followed by simple knots, terminates with a full multiplicity knot, and the values are increasing and equally spaced. This is typical of clamped curves. Periodic curves work differently as we will see later.

Here are two curves with identical control points but different knot vectors:

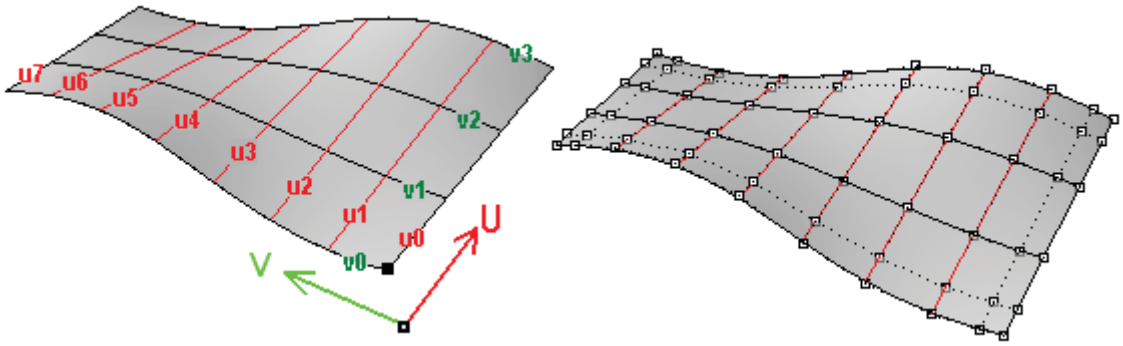
	Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,2,3,5,5,5)
	Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,1,1,4,4,4) ***Note**: Full knot multiplicity in the middle creates a kink and the curve is forced to go through the associated control point.*

Evaluation rule
The evaluation rule uses a mathematical formula that takes a number and assigns a point. The formula involves the degree, control points, and knots.

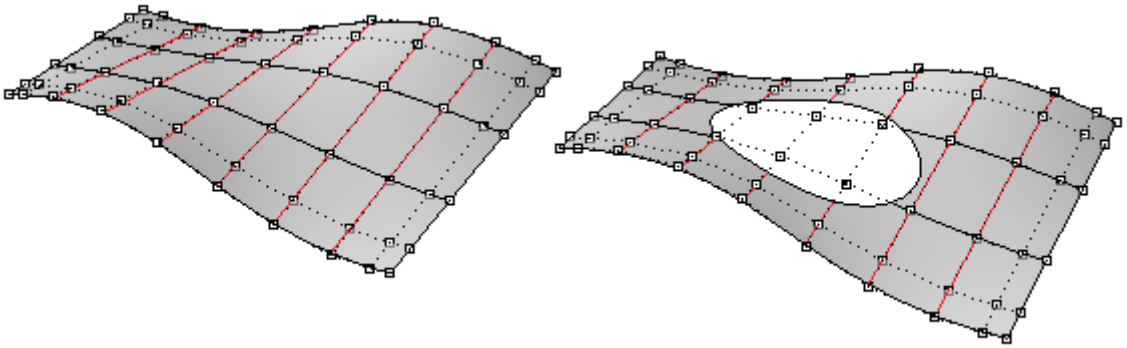
Using this formula, there are SDK functions that take a curve parameter, and produce the corresponding point on that curve. A parameter is a number that lies within the curve domain. Domains are usually increasing and they consist of two numbers: minimum domain parameter (m_t(0)) that usually the start of the curve and maximum (m_t(1)) at the end of the curve.

NURBS Surfaces

You can think of nurbs surfaces as a grid of nurbs curves that go in two directions. The shape of a NURBS surface is defined by a number of control points and the degree of that surface in each one of the two directions (u- and v-directions). Refer to *Rhino Help Glossary* for more details.



NURBS surfaces can be trimmed or untrimmed. Think of trimmed surfaces as using an underlying NURBS surface and closed curves to cut a specific shape of that surface. Each surface has one closed curve that define the outer border (*outer loop*) and as many non-intersecting closed inner curves to define holes (*Inner loops*). A surface with outer loop that is the same as that of its underlying NURBS surface and that has no holes is what we refer to as an untrimmed surface.

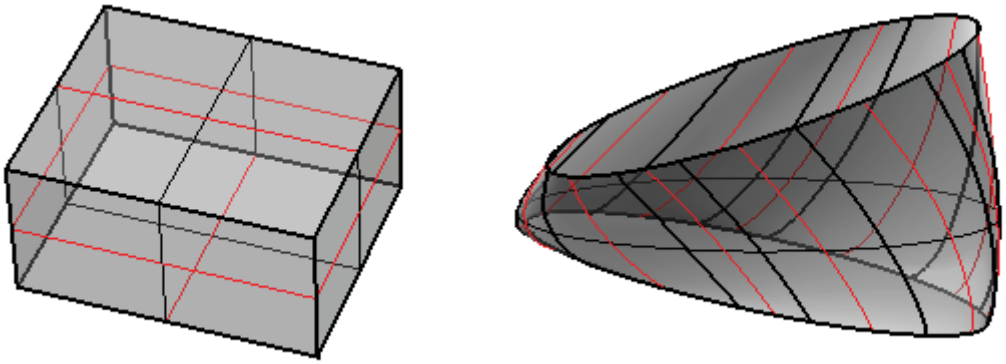


The surface on the left is untrimmed. The one on the right is the same surface trimmed with an elliptical hole. Note that the NURBS structure of the surface doesn't change when trimming.

Polysurfaces

A polysurface consists of more than one (usually trimmed) surfaces joined together. Each of the surfaces has its own parameterization and uv directions don't have to match. Polysurfaces and trimmed surfaces are represented using what is called boundary representation (BRep for short). It basically describes surfaces, edges and vertices geometry with trimming data and relationships among different parts. For example it describes each face, its surrounding edges and trims, normal direction relative to the surface, relationship with neighboring faces and so on. We will describe BReps member variable and how they are hooked together in some detail later.

OnBrep is probably the most complex data structures in OpenNURBS and it might not be easily digested, but luckily there are plenty of tools and global functions that come with Rhino SDK to help create and manipulate BReps.



15.3 OpenNURBS Objects Hierarchy

The SDK help file show all classes hierarchy. Here is a dissected subset of classes related to geometry creation and manipulation that you will likely use when writing scripts. I have to warn you, this list is very incomplete. Please refer to the help file for more details.

OnObject (*all Rhino classes are derived from OnObject*)

- **OnGeometry** (*class is derived from or inherits OnObject*)
  - OnPoint
    - OnBrepVertex
    - OnAnnotationTxtDot
  - OnPointGrid
  - OnPointCloud
  - OnCurve (*abstaract class)*
    - OnLineCurve

            - OnPolylineCurve
            - OnArcCurve
            - OnNurbsCurve
            - OnCurveOnSurface
            - OnCurveProxy
                - OnBrepTrim
                - OnBrepEdge
        - OnSurface (*abstract classt*)
            - OnPlaneSurface
            - OnRevSurface
            - OnSumSurface
            - OnNurbsSurface
            - OnProxySurface
                - OnBrepFace
                - OnOffsetSurface
        - OnBrep
        - OnMesh
        - OnAnnotation

- **Points and Vectors** *(not derived from OnGeometry)*
    - On2dPoint (good for parameter space points)
    - On3dPoint
    - On4dPoint (good for representing control points with x,y,z and w for weight)
    - On3dVector

- **Curves** *(not derived from OnGeometry)*
    - OnLine
    - OnPolyline (is actually derived from OnPointArray)
    - OnCircle
    - OnArc
    - OnEllipse
    - OnBezierCurve

- **Surfaces** *(not derived from OnGeometry)*
    - OnPlane
    - OnSphere
    - OnCylinder
    - OnCone
    - OnBox
    - OnBezierSurface

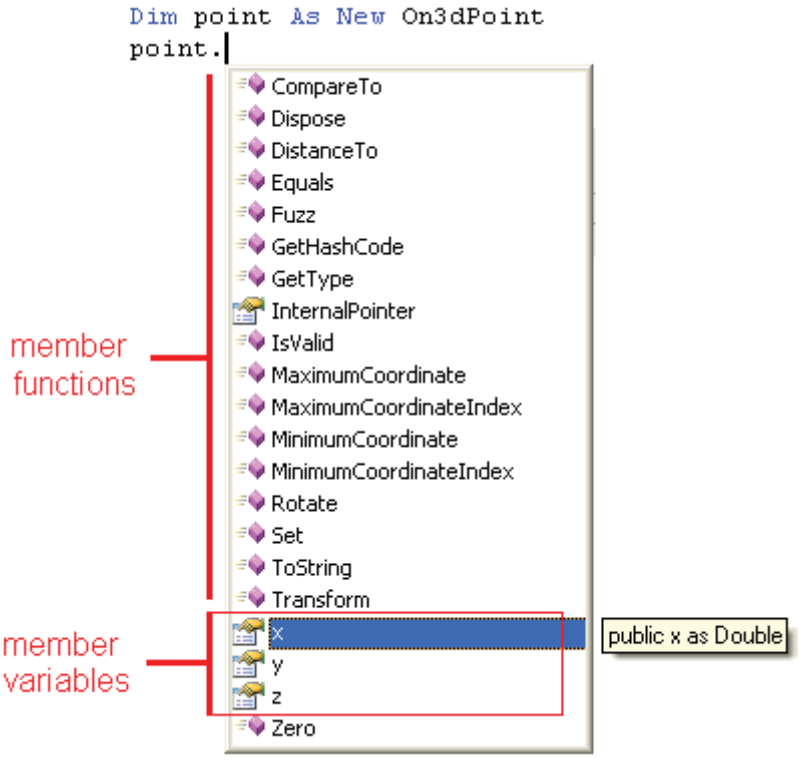
- **Miscellaneous**
    - OnBoundingBox (For objects bounding box calculation)
    - OnInterval (Used for curve and surface domains)
    - OnXform (for transforming geometry objects: move, rotate, scale, etc.)
    - OnMassProperties (to calculate volume, area, centroid, etc)

15.4 Class structure

A typical class (which is a user-defined data structure) has four main parts:

- **Constructor**: This is used to create an instance of the class.
- **Public member variables**: This is where class data is stores. OpenNURBS member variables usually start with "**m_**" to quickly isolate.
- **Public member functions**: This includes all class functions to create, update and manipulate class member variable or perform certain functionality.
- **Private members**: these are class utility functions and variables for internal use.

Once you insatiate a class, you will be able to see all class member functions and member variable through the auto-complete feature. Note that when you roll-over any of the function or variables, the signature of that function is shown. Once you start filling function parameters, the auto-complete will show you which parameter you are at and its type. This is a great way to navigate available function for each class. Here is an example from On3dPoint class:



Copying data from an exiting class to a new one can be done in one or more ways depending on the class. For example, let's create a new On3dPoint and copy the content of an existing point into it. This is how we may do it:

```
Use the constructor when you instantiate an instance of the point class
Dim new_pt as New On3dPoint( input_pt )

Use the "= operator" if the class provides one
Dim new_pt as New On3dPoint
new_pt = input_pt
```

```
You can use the "New" function if available
Dim new_pt as New On3dPoint
new_pt.New( input_pt )

There is also a  "Set" function sometimes
Dim new_pt as New On3dPoint
new_pt.Set( input_pt )

Copy member variables one by one.  A bit exhaustive method
Dim new_pt as New On3dPoint
new_pt.x =  input_pt.x
new_pt.y =  input_pt.y
new_pt.z =  input_pt.z

OpenNURBS geometry  classes provide "Duplicate" function that is very efficient to use
Dim new_crv as New OnNurbsCurve
new_crv = input_crv.DuplicateCurve()
```

15.5 Constant vs Non-constant Instances

Rhino .NET SDK provide two sets of classes. The first is constant and it classes names are preceded by an "I"; for example *IOn3dPoint*. The corresponding non-constant class which is what you will mostly need to use has same name without the "I"; for example *On3dPoint*. You can duplicate a constant class or see its member variables and some of the functions, but you cannot change its variables.

Rhino .NET SDK is based on Rhino C++ SDK. C++ programming language offers the ability to pass constant instants of classes and is used allover the SDK functions. On the other hand, DotNET doesn't have such concept and hence the two versions for each class.

15.6 Points and Vectors

There are many classes that could be used to store and manipulate points and vectors. Take for example double precision points. There are three types of points:

Class name	Member variables	Notes
On2dPoint	**x** as Double **y** as Double	Mainly used for parameter space points. The "**d**" in the class name stand for double precision floating point number. There are other points classes that has "**f**" in the name that use single precision.
On3dPoint	**x** as Double **y** as Double	Most commonly used to represent points in three dimensional coordinate space
On4dPoint	**x** as Double **y** as Double **z** as Double **w** as Double	Used for grip points. Grips have weight information in addition to the three coordinates.

Points and vectors operations include:

Vector Addition:

```
Dim add_v As New On3dVector = v0 + v1
```

Vector Subtraction:

Dim subtract_vector As New On3dVector = v0 – v1

Vector between two points:

Dim dir_vector As New On3dVector = p1 – p0

Vector dot product (if result is positive number then vectors are in the same direction):

Dim dot_product As Double = v0 * v1

Vector cross product (result is a vector normal to the 2 input vectors)

Dim normal_v As New On3dVector = OnUtil.ON_CrossProduct(v0, v1)

Scale a vector:

Dim scaled_v As New On3dVector = factor * v0

Move a point by a vector:

Dim moved_point As New On3dPoint = org_point + dir_vector

Distance between 2 points:

Dim distance As Double = pt0.DistanceTo(pt1)

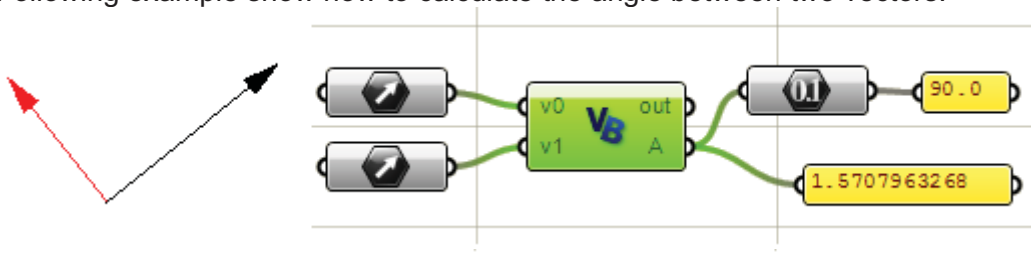
Get unit vector (set vector length to 1):

v0.Unitize()

Get vector length:

Dim length As Double = v0.Length()

Following example show how to calculate the angle between two vectors.



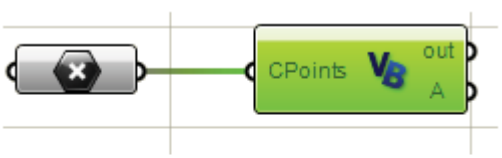

```
Sub RunScript(ByVal v0 As On3dVector, ByVal v1 As On3dVector)

    ' Unitize the input vectors
    v0.Unitize()
    v1.Unitize()
    Dim dot As Double = OnUtil.ON_DotProduct(v0, v1)

    ' Force the dot product of the two input vectors to
    ' fall within the domain for inverse cosine, which
    ' is -1 <= x <= 1. This will prevent runtime
    ' "domain error" math exceptions.
    If (dot < -1.0) Then dot = -1.0
    If (dot > 1.0) Then dot = 1.0

    A = System.Math.Acos(dot)

End Sub
```

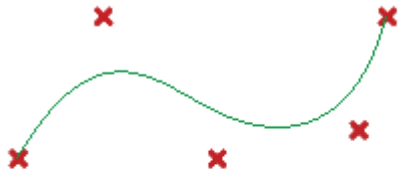
15.7 OnNurbsCurve

In order to create a nurbs curve, you will need to provide the following:

- Dimension, which is typically = 3.
- Order: Curve degree + 1.
- Control points (array of points).
- Knot vector (array of numbers).
- Curve type (clamped or periodic).

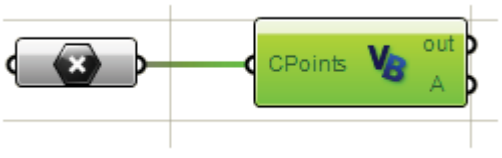
There are functions to help create the knot vector, as we will see shortly, so basically you need to decide on the degree and have a list of control points and you are good to go. The following example creates a clamped curve.

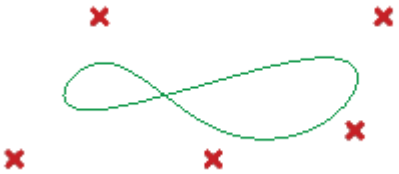




```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim nc As New OnNurbsCurve

  'Create open (Clamped) Nurbs Curve
  nc.CreateClampedUniformNurbs(dimension, order, CPoints.ToArray())

  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

For smooth closed curves, you should create periodic curves. Using same input control points and curve degree, the following example shows how to create a periodic curve.



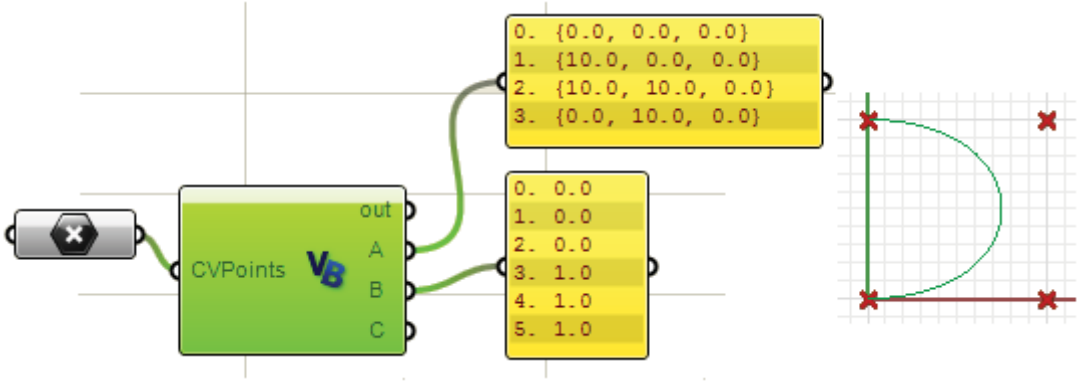


```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim nc As New OnNurbsCurve

  'Create closed (Periodic) Nurbs Curve
  nc.CreatePeriodicUniformNurbs(dimension, order, CPoints.ToArray())

  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

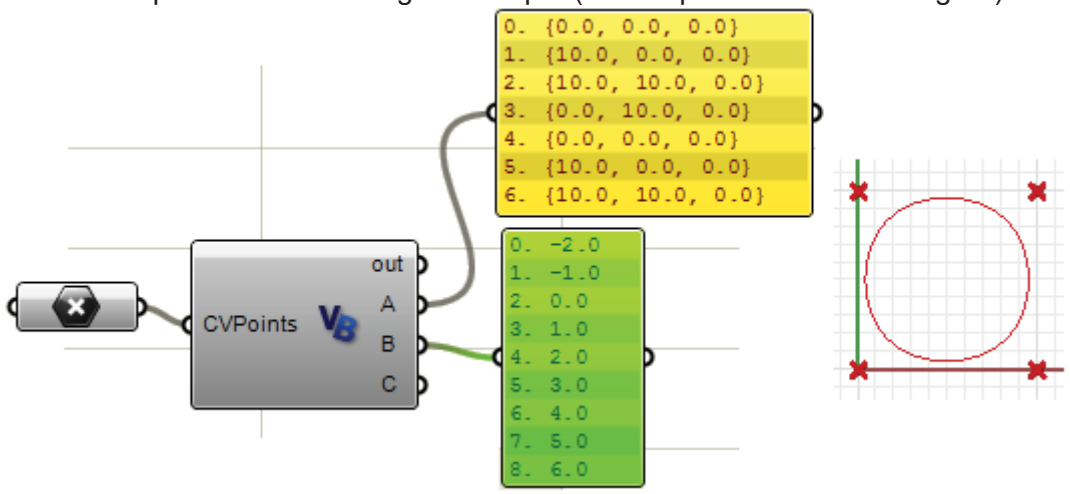
Clamped vs periodic NURBS curves

Clamped curves are usually open curves where curve ends coincide with end control points. Periodic curves are smooth closed curves. The best way to understand the differences between the two is through comparing control points.

The following component creates clamped NURBS curve and outputs:

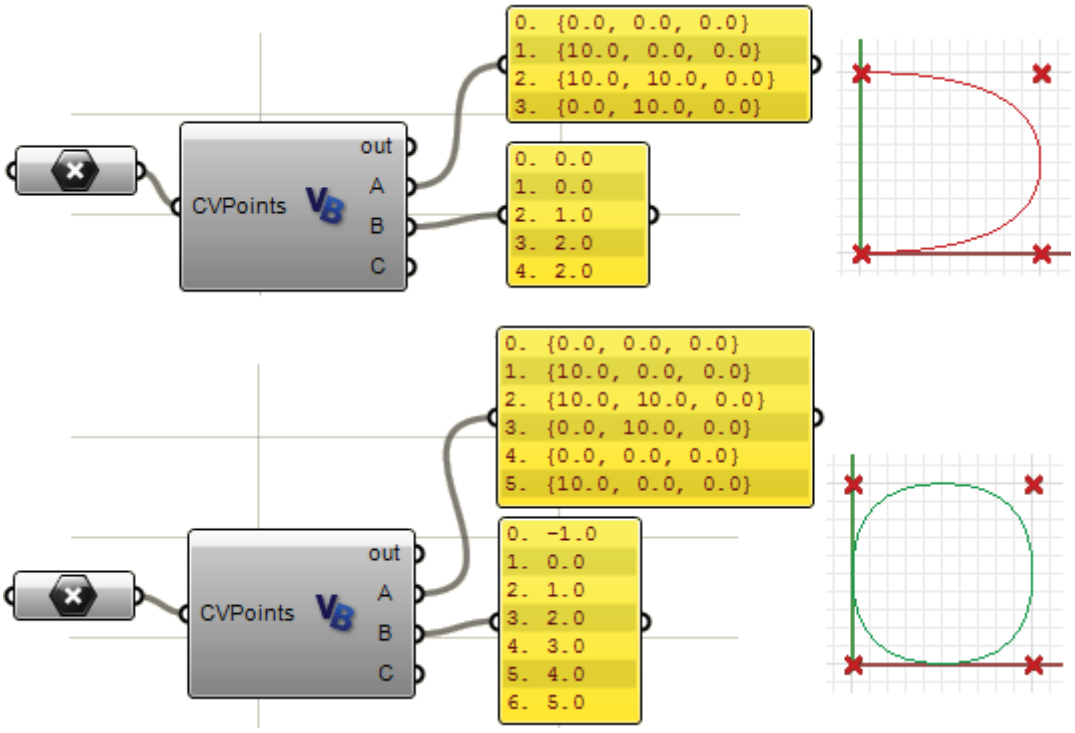


Here is the periodic curve using same input (control points and curve degree):



Note that the periodic curve turned the four input points into seven control points (4+degree)" and while the clamped curve used four control points only. The knot vector of the periodic curve used only simple knots while the clamped curve start and end knots have full multiplicity.

Here are same examples but with degree 2 curves. As you may have guessed, number of control points and knots of periodic curves change when degree changes.

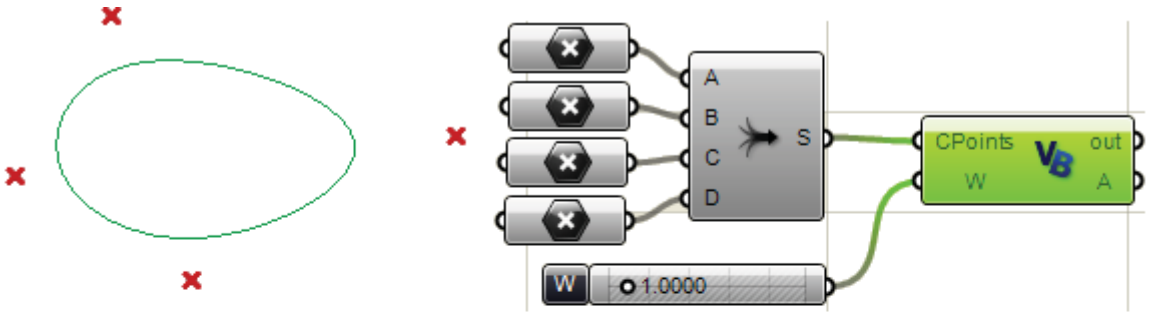


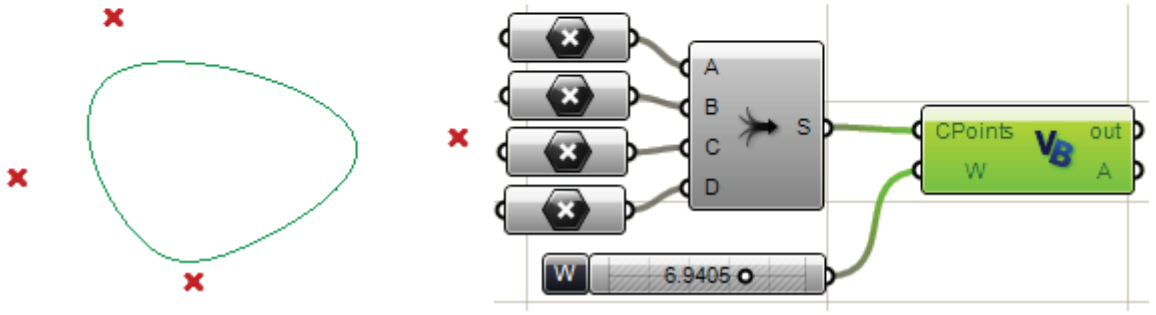
This is the code used to navigate through CV points and knots in the previous examples:

```
'Output control points
Dim count As Double = nc.CVCount()
Dim i As Integer
Dim cvs As New List( Of On3dPoint )
For i = 0 To count - 1
  Dim cv As New On3dPoint(0, 0, 0)
  nc.GetCV(i, cv)
  cvs.Add(cv)
Next

'Output knots
Dim knots As New List( Of Double )
count = nc.KnotCount()
For i = 0 To count - 1
  knots.Add(nc.Knot(i))
Next
```

Weights

Weights of control points in a uniform nurbs curve are set to 1, but this number can vary in rational nurbs curves. The following example shows how to modify weights of control points interactively in Grasshopper.





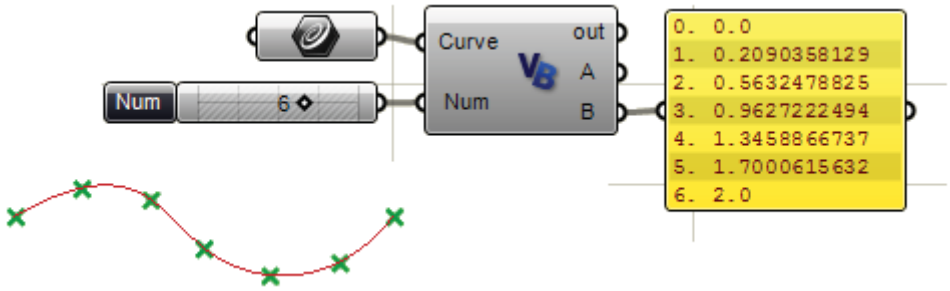

```
Sub RunScript(ByVal CPoints As List(Of On3dPoint), ByVal W As Double)

  Dim i As Integer

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim cv_count As Integer = CPoints.Count
  Dim nc As New OnNurbsCurve
  nc.CreatePeriodicUniformNurbs(dimension, order, CPoints.ToArray())
  nc.MakeRational()

  'Assign weights
  Dim cv As New On3dPoint
  For i = 0 To cv_count - 1
    nc.GetCV(i, cv)
    cv = cv * W
    nc.SetCV(i, cv)
    nc.SetWeight(i, W)
  Next


  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

Divide NURBS curve

Dividing a curve into a number of segments involves the following steps:

- Find curve domain which is the parameter space interval.
- Make a list of parameters that divide the curve into equal segments.
- Find points on the 3d curve.

The following example shows how to achieve that. Note that there is a global function under RhUtil name space that divides a curve by number of segments or arc length that you can use directly as we will illustrate later.




```
Sub RunScript(ByVal Curve As OnCurve, ByVal Num As Integer)

  Dim min As Double = Curve.Domain().Min()
  Dim max As Double = Curve.Domain().Max()

  'Find the step value
  Dim step_value As Double = (max - min) / (Num - 1)

  Dim Points As New List( Of on3dPoint )

  Dim t_list(Num) As Double
  For i As Integer = 0 To Num
    t_list(i) = i / Num
  Next

  If (Curve.GetNormalizedArcLengthPoints(t_list, t_list)) Then
    For i As Integer = 0 To Num
      Dim pt As On3dPoint = Curve.PointAt(t_list(i))
      Points.Add(pt)
    Next
  End If


  A = Points
  B = t_list

End Sub
```

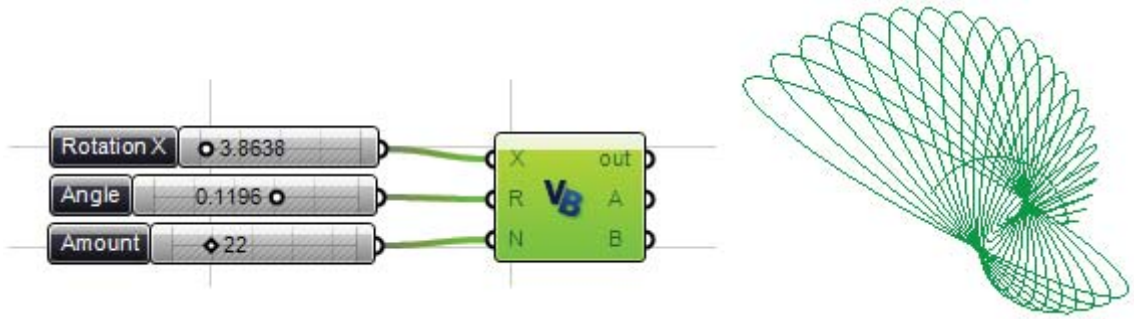
15.8 Curve Classes not Derived from OnCurve

Although all curves can be represented as NURBS curves, it is useful sometimes to work with other types of curve geometry. One reason is because they mathematical

representation hat is easier to understand than NURBS and are typically more light-weighted. It is relatively easy to get the NURBS form of those curves not derived from OnCurve when you need one. Basically you need to convert to a corresponding class. The following table shows the correspondence:

Curves Types	OnCurve Derived Types
OnLine	OnLineCurve
OnPolyline	OnPolylineCurve
OnCircle	OnArcCurve or OnNurbsCurve (use GetNurbsForm() member function)
OnArc	OnArcCurve or OnNurbsCurve (use GetNurbsForm() member function)
OnEllipse	OnNurbsCurve (use GetNurbsForm() member function)
OnBezierCurve	OnNurbsCurve (use GetNurbsForm() member function)

Here is an example that uses OnEllipse and OnPolyline classes:



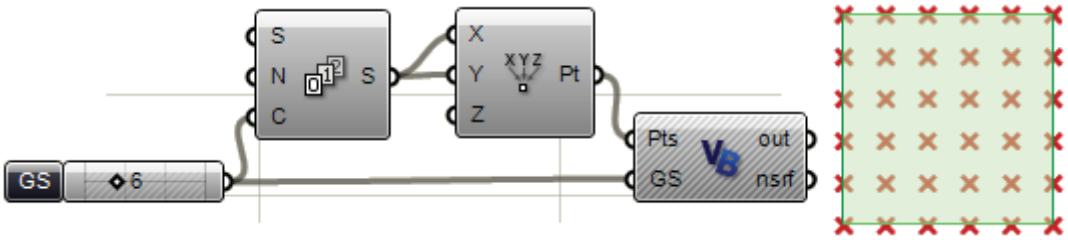

```
Sub RunScript(ByVal X As Object, ByVal R As Object, ByVal N As Object)
  'Declare a new list of OpenNURBS circles
  Dim c_list As New List(Of OnEllipse)

  'Declare list of lines
  Dim p_list As New On3dPointArray

  For i As Int32 = 1 To N
    'Declare a new circle
    Dim c As New OnEllipse(OnUtil.On_xy_plane, i / 2, i)
    'Rotate the circle
    C.Rotate(R * i, New On3dVector(0, 1, 0), New On3dPoint(X, 0, 0))
    'Add the circle to the list
    c_list.Add(c)
    'Add center point
    p_list.Append(C.Center())
  Next

  Dim polyline As New OnPolyline(p_list)
  'Assign the list to the output value A
  A = c_list
  'Assign polyline to output value B
  B = polyline
End Sub
```

15.9 OnNurbsSurface

Similar to what we discussed for OnNurbsCurve class, to create a OnNurbsSurface you will need to know:

- Dimension, which is typically = 3.
- Order in u and v direction: degree + 1.
- Control points.
- Knot vector in u and v directions.
- Surface type (clamped or periodic).

The following example creates a nurbs surface from a grid of control points:



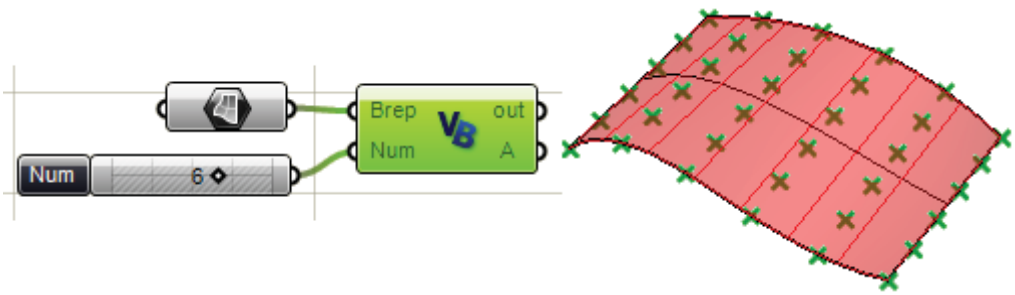

```
Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)
  'Create a grid of points
  Dim Grid As New ArrayList()
  'Call grid function
  Call CreateGrid(Pts, Grid, GS)
  'Call create nurbs surface function
  Dim ns As OnNurbsSurface
  ns = CreateNS(Grid, GS)

  'Assign mid point to output
  nsrf = ns

End Sub
```

```
Sub CreateGrid( ByVal Pts As List(Of On3dPoint),
                ByRef Grid As ArrayList, ByVal GS As Integer )
  Dim i As Integer
  Dim j As Integer
  For i = 0 To Pts.Count() - 1 Step GS
    'Declare a row of points
    Dim Row As New List( Of On3dPoint )
    For j = i To i + GS - 1
      'Get a reference od the point
      Dim pt As On3dPoint
      pt = Pts(j)
      'Add point to the row
      Row.Add(pt)
    Next
    'Add row to the grid
    Grid.Add(Row)
  Next
End Sub
```

```
Function CreateNS(ByVal cvpoints As ArrayList,
                  ByVal GS As Integer) As OnNurbsSurface

  Const Degree As Integer = 3

  'Make the surface
  Dim orderU As Integer = Degree + 1
  Dim orderV As Integer = Degree + 1

  Dim ns As New OnNurbsSurface
  ns.Create(3, False, orderU, orderV, GS, GS)

  'Add cv points
  Dim i As Integer
  Dim j As Integer
  Dim pt As On3dPoint
  For i = 0 To GS - 1
    For j = 0 To GS - 1
      pt = cvpoints(i)(j)
      ns.SetCV(i, j, pt)
    Next
  Next

  'Set knots for open surface
  ns.MakeClampedUniformKnotVector(0)
  ns.MakeClampedUniformKnotVector(1)

  CreateNS = ns
End Function
```

Another common example is to divide a surface domain. The following example divides surface domain into a equal number of points in both direction (number of points must be greater than one to make sense) and it does the following:

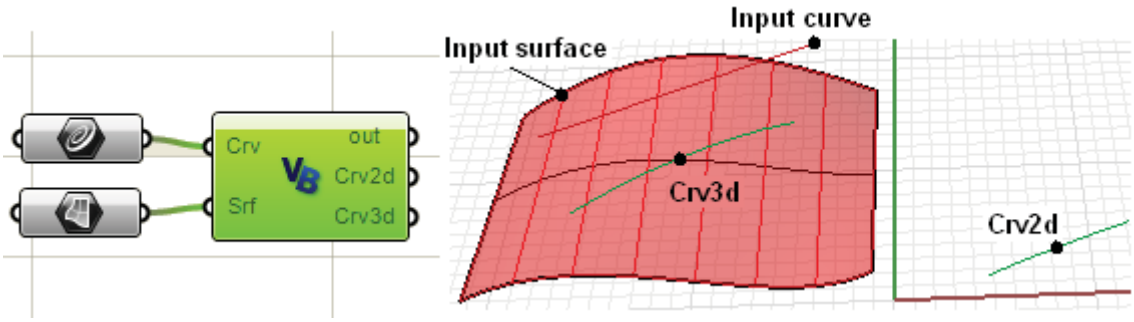
- normalize surface domain (set domain interval to 0-1)
- Calculate step value using number of points.
- Uses a nested loop to calculate surface points using u and v parameters.



```
Sub RunScript(ByVal Brep As OnBrep, ByVal Num As Integer)
  'Find step - Num must be > 1
  Dim StepValue As Double = 1 / (Num - 1)

  Dim nSrf As New OnNurbsSurface
  nSrf = Brep.Face(0).NurbsSurface

  'Normalize domain in u and v directions
  nSrf.SetDomain(0, 0, 1)
  nSrf.SetDomain(1, 0, 1)

  Dim Points As New List( Of on3dPoint )
  Dim i As Double = 0
  Dim j As Double = 0
  For i = 0 To 1 Step StepValue
    For j = 0 To 1 Step StepValue
      Dim Pt As New On3dPoint
      Pt = nSrf.PointAt(i, j)
      Points.Add(Pt)
    Next
  Next

  A = Points
End Sub
```

OnSurface class has many functions that are very useful to manipulate and work with surfaces. The following example shows how to pull a curve to a surface.

There are 2 outputs in the Grasshopper scripting component. The first is the parameter space curve (flat representation of the 3d curve in world xy plane) relative to surface domain. The second is the curve in 3d space. We got the 3d curve through “pushing” the 2d parameter space curve to the surface.



```
Sub RunScript(ByVal Crv As OnCurve, ByVal Srf As OnBrep)

  'Get pulled curve in 2D parameter space
  Dim pull_crv As OnCurve
  pull_crv = Srf.m_S(0).Pullback(Crv, doc.AbsoluteTolerance())

  'Get the pulled curve in 3D space
  Dim push_crv As OnCurve
  push_crv = Srf.m_S(0).Pushup(pull_crv, doc.AbsoluteTolerance())

  'Output both curves
  Crv2d = pull_crv
  Crv3d = push_crv

End Sub
```

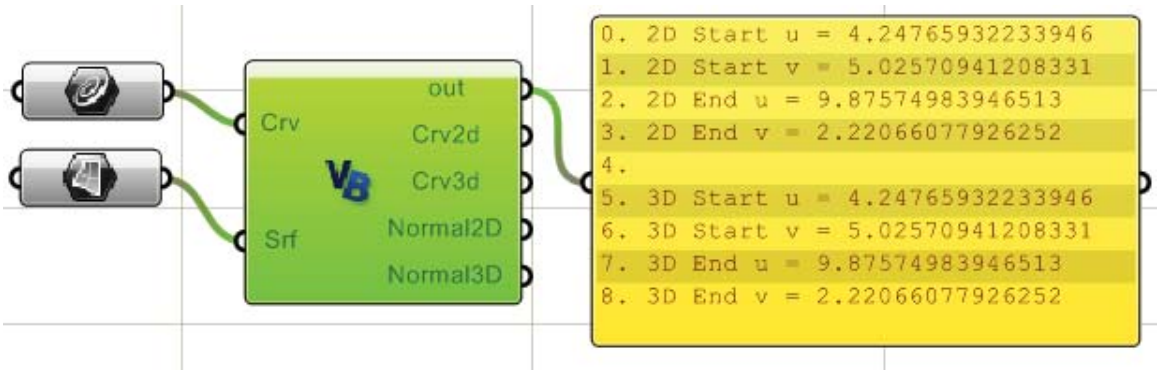
Using the previous example, we will calculate normal vector of the start and end points of the pulled curve. Here are 2 ways to do that:

- Use pulled 2D curve start and end 2D points and this would be start and end points on surface in parameter space.
- Or use pushes 3D curve end points, find closest point to surface and use resulting parameters to find surface normal.

```
Sub GetEndNormals2D(ByVal crv2d As OnCurve,
                    ByVal srf As OnSurface,
                    ByRef EndVectors2D As List(Of On3dVector ))
  Dim start_normal As On3dVector
  Dim end_normal As On3dVector

  'find start and end points in parameter space
  Dim start2d As New On2dPoint
  start2d = crv2d.PointAtStart()
  Dim end2d As New On2dPoint
  end2d = crv2d.PointAtEnd()

  'Output parameters
  'Surface parameters are the x and y of the 2d curve end points
  Print("2D Start u = " & start2d.x)
  Print("2D Start v = " & start2d.y)
  Print("2D End u = " & end2d.x)
  Print("2D End v = " & end2d.y)
  Print("")

  'Call surface normal function
  start_normal = srf.NormalAt(start2d.x, start2d.y)
  end_normal = srf.NormalAt(end2d.x, end2d.y)

  EndVectors2D.Add(start_normal)
  EndVectors2D.Add(end_normal)
End Sub
```

```
Sub GetEndNormals3D(ByVal crv3d As OnCurve,
                    ByVal srf As OnSurface,
                    ByRef EndVectors3D As List(Of On3dVector ))
  'Declare start and end normal
  Dim start_normal As On3dVector
  Dim end_normal As On3dVector

  'Find start and end points in parameter space
  Dim start3d As New On3dPoint
  start3d = crv3d.PointAtStart()
  Dim end3d As New On3dPoint
  end3d = crv3d.PointAtEnd()

  'Declare parameters
  Dim u As Double
  Dim v As Double

  'Get surface closest point
  srf.GetClosestPoint(start3d, u, v)
  start_normal = srf.NormalAt(u, v)

  'Output start parameters
  Print("3D Start u = " & u)
  Print("3D Start v = " & v)

  srf.GetClosestPoint(end3d, u, v)
  end_normal = srf.NormalAt(u, v)

  'Output end parameters
  Print("3D End u = " & u)
  Print("3D End v = " & v)

  EndVectors3D.Add(start_normal)
  EndVectors3D.Add(end_normal)
End Sub
```

This is the component picture showing output of parameter value at the end points using both functions. Notice that both methods yield same parameters as expected.

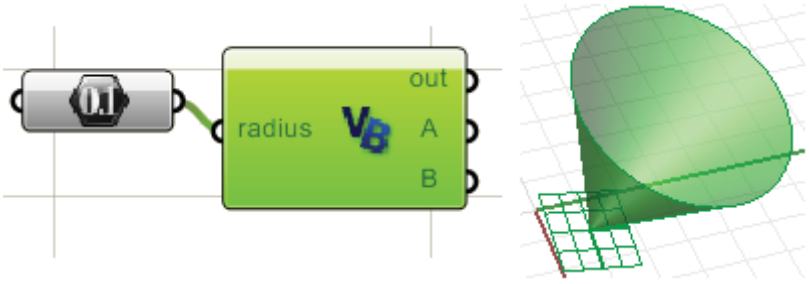


15.10 Surface Classes not Derived from OnSurface

OpenNURBS provides surface classes not derived from OnSurface. They are valid mathematical surface definitions and can be converted into OnSurface derived types to use in functions that take OnSurface. Here is a list of surface classes and their corresponding OnSurface derived classes:

Basic Surface Types	OnSurface derived Types
OnPlane	OnPlaneSurface or OnNurbsSurface (use OnPlane.GetNurbsForm() function)
OnShpere	OnRevSurface or OnNurbsSurface (use OnShpere.GetNurbsForm() function)
OnCylinder	OnRevSurface or OnNurbsSurface (use OnCylinder.GetNurbsForm() function)
OnCone	OnRevSurface or OnNurbsSurface (use OnCone.GetNurbsForm() function)
OnBezierSurface	OnNurbsSurface (use GetNurbsForm() member function)

Here is an example that uses OnPlane and OnCone classes:



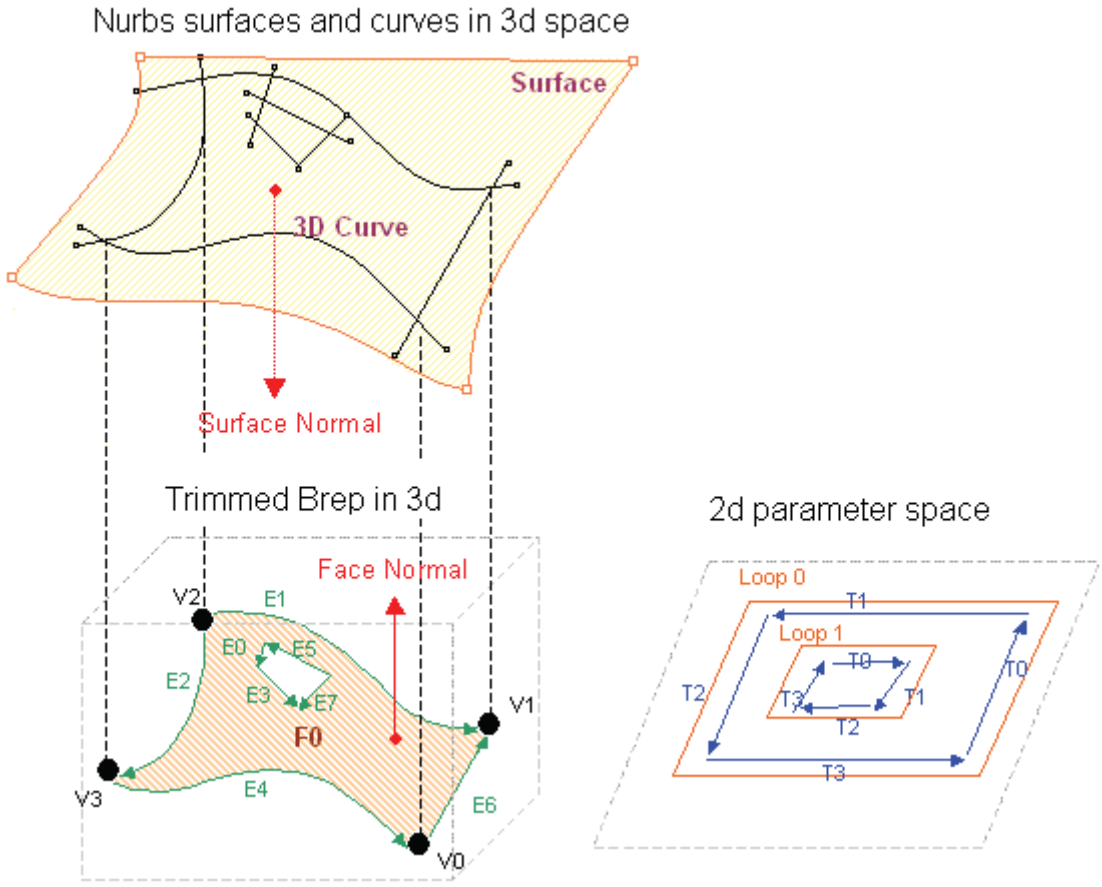

```
Sub RunScript(ByVal radius As Double)
  'Create a plane from origin and normal
  Dim plane As New OnPlane
  Dim origin As New On3dPoint(1, 1, 0)
  Dim normal As New On3dVector(1, 1, 3)
  plane.CreateFromNormal(origin, normal)

  'Define height value
  Dim height As Double = 5
  'Create cone
  Dim cone As New OnCone(plane, height, radius)

  'Assign output parameter
  A = cone
  B = plane
End Sub
```

15.11 OnBrep

Boundary representation (B-Rep) is used to unambiguously represent objects in terms of their boundary surfaces. You can think of OnBrep as having three distinct parts:

- Geometry: 3d geometry of nurbs curves and surfaces. Also 2D curves of parametric space or trim curves.
- 3D Topology: faces, edges and vertices. Each face references one nurbs surface. The face also knows all loops that belong to that face. Edges reference 3d curves. Each edge has a list of trims that use that edge and also the two end vertices. Vertices reference 3d points in space. Each vertex also have a list of edges they lie on one of their ends.
- 2D Topology: 2d parametric space representation of faces and edges. In parameter space, 2d trim curves go either clockwise or anti-clockwise depending on whether they are part of an outer or inner loop of the face. Each valid face will have to have exactly one outer loop but can have as many inner loops as it needs (holes). Each trim reference one edge, 2 end vertices, one loop and the 2D curve.

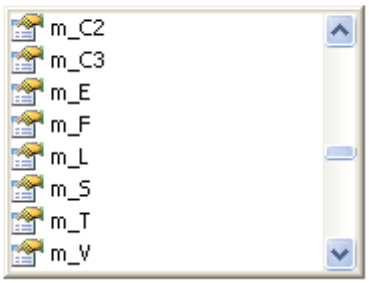
The following diagram shows the three parts and how they relate to each other. The top part shows the underlying 3d nurbs surface and curves geometry that defines a single face brep face with a hole. The middle is the 3d topology which includes brep face, outer edges, inner edges (bounding a hole) and vertices. Then there is the parameter space with trims and loops.



OnBrep member variables

OnBrep Class member variables include all 3d and 2d geometry and topology information. Once you create an instance of a brep class, you can view all member functions and variables. The following image shows member functions in the auto-complete. The table lists data types with description.

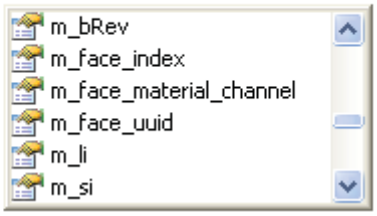
```
Dim brep As New OnBrep
brep.
```



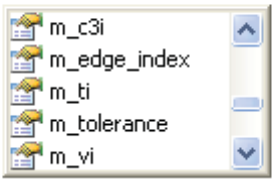
Topology members: describe relationships among different brep parts	
OnBrepVertexArray **m_V**	Array of brep vertices (OnBrepVertex)
OnBrepEdgeArray **m_E**	Array of brep edges (OnBrepEdge)
OnBrepTrimArray **m_T**	Array of brep trims (OnBrepTrim)
OnBrepFaceArray **m_F**	Array of brep faces (OnBrepFace)
OnBrepLoopArray **m_L**	Array of loops (OnBrepLoop)
Geometry members: geometry data of 3d curves and surfaces and 2d trims	
OnCurveArray **m_C2**	Array of trim curves (2D curves)
CnCurveArray **m_C3**	Array of edge curve (3D curves)
ONSurfaceArray **m_S**	Array of surfaces

Notice that each of the OnBrep member functions is basically an array of other classes. For example m_F is an array of references to OnBrepFace. OnBrepFace is a class derived from OnSurfaceProxy and has variable and member functions of its own. Here are the member variables of OnBrepFace, OnBrepEdge, OnBrepVertex, OnBrepTrim and OnBrepLoop classes:

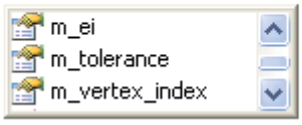
```
Dim brep_face As New OnBrepFace
brep_face.
```



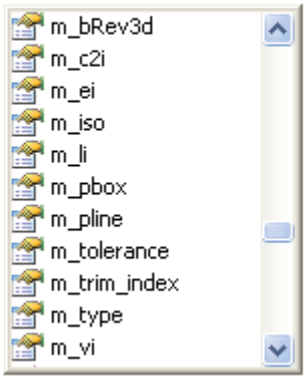
```
Dim brep_edge As New OnBrepEdge
brep_edge.
```



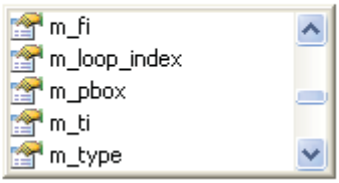

```
Dim brep_vertex As New OnBrepVertex
brep_vertex.
```



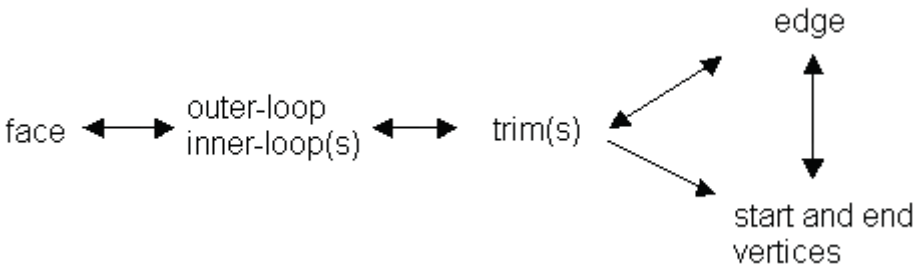

```
Dim brep_trim As New OnBrepTrim
brep_trim.
```



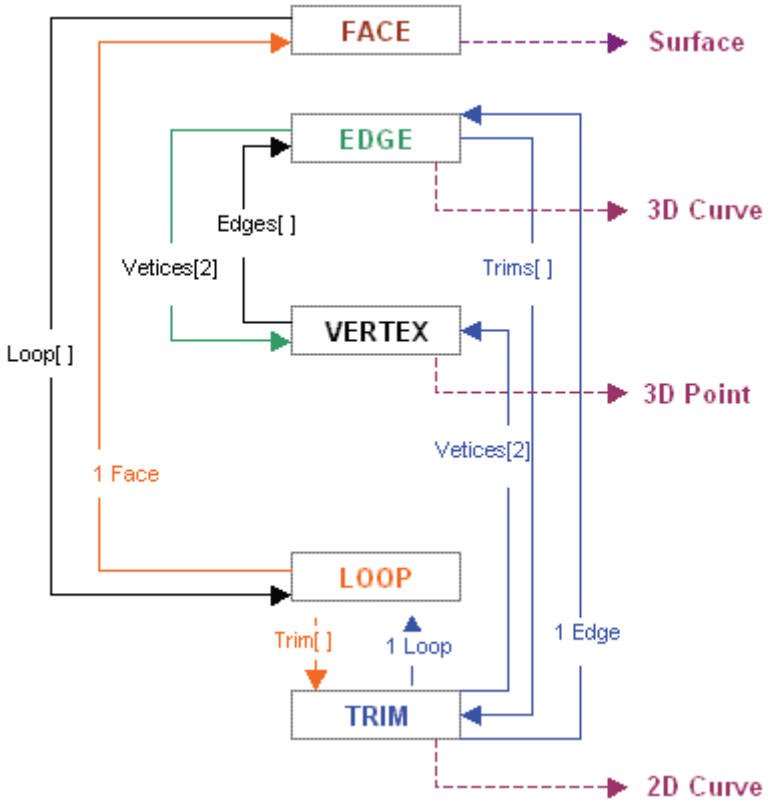

```
Dim brep_loop As New OnBrepLoop
brep_loop.
```



The following diagram shows OnBrep member variables and how they reference each other. You can use this information to get to any particular part of the brep. For example each face knows its loops and each loop has a list of the trims and from a trim you can get to the edge it is hooked to and the 2 end vertices and so on.



Here is another more detailed diagram of how brep parts are hooked together and how to get from one part o the other.



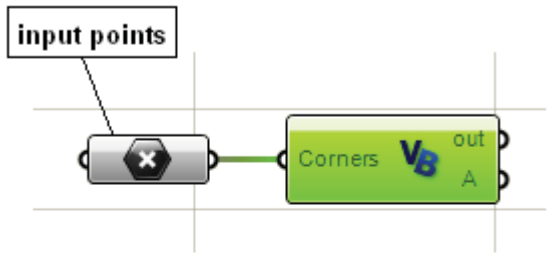
We will spend the coming few examples showing how to create an OnBrep, navigate different parts and extract brep information. We will also show how to use few of the functions that come with the class as well as global functions.

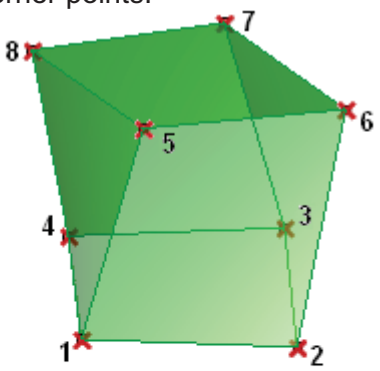
Create OnBrep

There are few ways to create a new instance of an OnBrep class:

- Duplicate existing brep
- Duplicate or extract a face in an exiting brep
- Use Create function that takes an OnSurface as an input parameter. There are 5 overloaded function using different types of surfaces:
    - from SumSurface
    - from RevSurface
    - from PlanarSurface
    - from OnSurface
- Use global utility functions.
    - From OnUtil such as ON_BrepBox, ON_BrepCone, etc.
    - From RhUtil such as RhinoCreatEdgeurface or RhinoSweep1 among others.

This is an example to create a brep box from corner points.





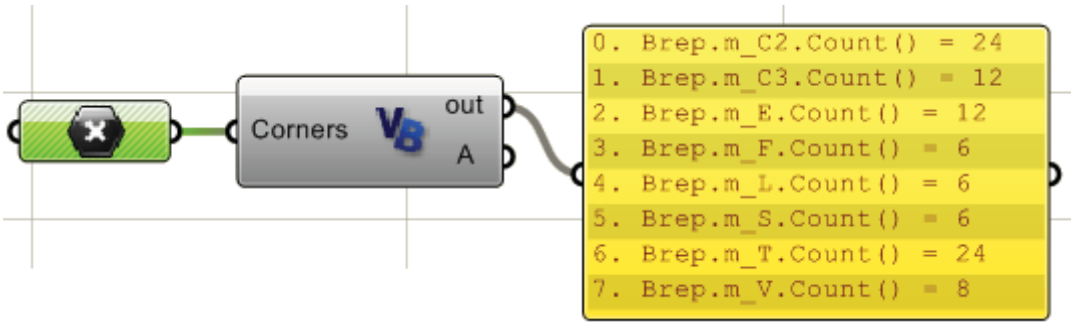

```
Sub RunScript(ByVal Corners As List(Of On3dPoint))

  ' Build the brep from corners
  Dim Brep As OnBrep = OnUtil.ON_BrepBox(Corners.ToArray())

  A = Brep
End Sub
```

Navigate OnBrep data

The following example shows how to extract vertices' points of a brep box

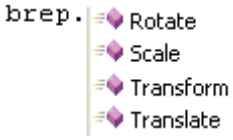
```
Sub RunScript(ByVal Corners As List(Of On3dPoint))

  ' Build the brep from corners
  Dim Brep As OnBrep = OnUtil.ON_BrepBox(Corners.ToArray())

  Dim myCorners As New List( Of On3dPoint )
  Dim v As OnBrepVertex
  Dim i As Integer

  For i = 0 To Brep.m_V.Count() - 1
    'get reference to OnBrepVertex
    v = Brep.m_V(i)

    'Get vertex point (loction)
    Dim pt As New On3dPoint
    pt = v.point

    'Add point to array
    myCorners.Add(pt)
  Next

  A = Brep
  B = myCorners
End Sub
```

The following example shows how to get the number of geometry and topology parts in a brep box (faces, edges, trims, vertices, etc).



Transform OnBreps

All classes derived from OnGeometry inherit four transformation functions. The first three are probably the most commonly used which are Rotate, Scale and Transform. But there is also a generic “Trabsform” function that takes a 4x4 transformation matrix defined with OnXform class. We will discuss OnXform in the next section.

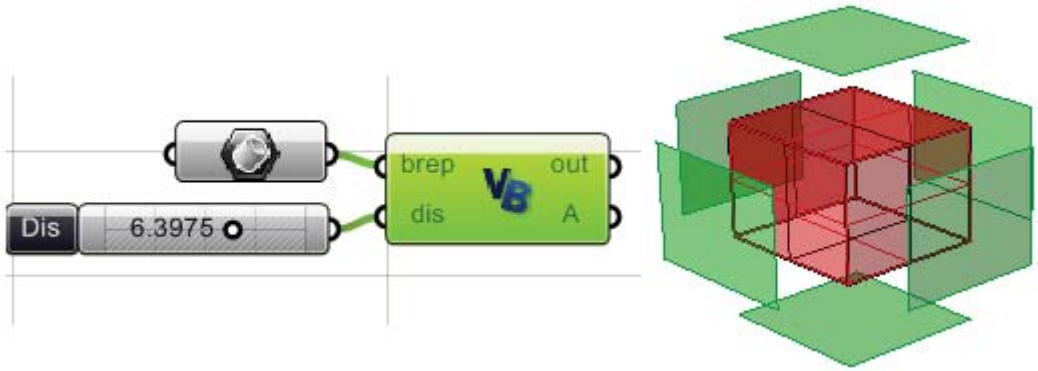


Edit OnBrep

Most of OnBrep class member functions are expert user tools to create and edit breps. There are however many global function to Boolean, intersect or split breps as we will illustrate in a separate section.

There is an excellent detailed example available in McNeel’s wiki DotNET samples that creates a brep from scratch and should give you a good idea about what it takes to create a valid OnBrep from scratch.

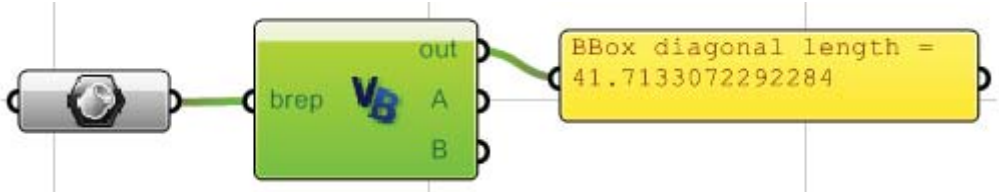
Here is an example that extracts OnBrep faces and move them away from the brep center. The example uses bounding box center.



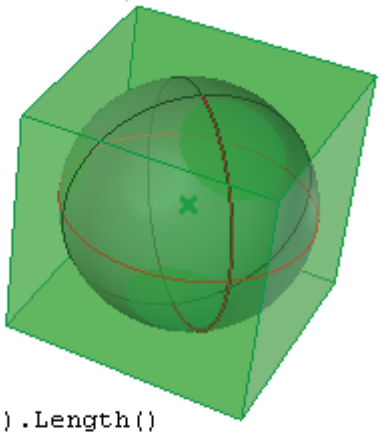
```
Sub RunScript(ByVal brep As OnBrep, ByVal dis As Double)

  Dim faces As New List(Of OnBrep)

  'Loop through brep faces to extract them
  For fi As Integer = 0 To brep.m_F.Count() - 1
    'Decalre new brep
    Dim face As New OnBrep
    face = brep.DuplicateFace(fi, False)

    'Add to faces array
    faces.Add(face)
  Next

  'Find brep bounding box center
  Dim center As New On3dPoint
  center = brep.BoundingBox().Center()

  'Loop through faces and move away from center by dis
  Dim dir As New On3dVector
  For i As Integer = 0 To faces.Count() - 1
    Dim face As OnBrep
    face = faces(i)

    'Find ceneter of each extracted face
    Dim face_center As On3dPoint
    face_center = face.BoundingBox().Center()

    'Find translation vector
    dir = face_center - center
    dir.Unitize()
    dir *= dis

    'Move face away from center
    face.Translate(dir)
  Next

  'Assign output
  A = faces


End Sub
```

Other OnBrep member functions

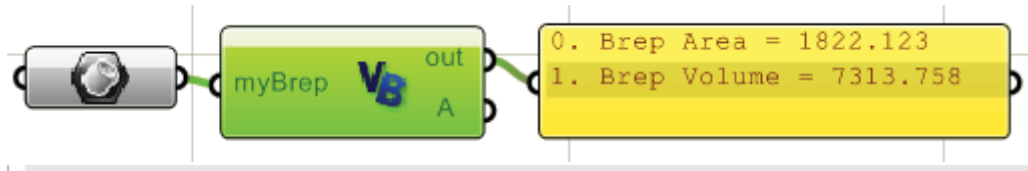
OnBrep class has many other functions that are either inherited from a parent class or are specific to OnBrep class. All geometry classes, including OnBrep, have a member function called “BoundingBox()”. One of OpenNURBS classes is OnBoundingBox which gives useful geometry bounding information. See the following example that finds a brep bounding box, its center and diagonal length.



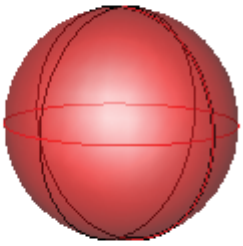

```
Sub RunScript(ByVal brep As OnBrep)

  'Find brep bounding box
  Dim bbox As New OnBoundingBox
  bbox = brep.BoundingBox()

  'Find bounding box ceneter
  Dim center As New On3dPoint
  center = bbox.Center()

  'Print bounding box diagonal length
  Dim length As Double = bbox.Diagonal().Length()
  Print("BBox diagonal length = " & length)

  A = bbox
  B = center

End Sub
```



Another area that is useful to know about is the mass properties. OnMassProperties class and few of its functions is illustrated in the following example:

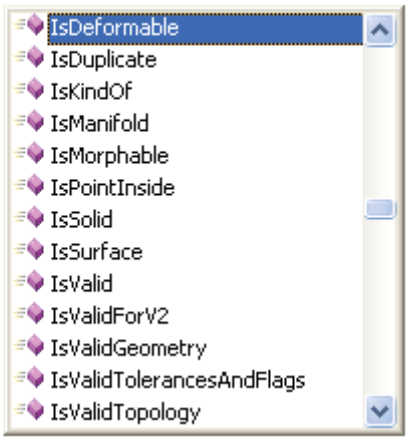



```
Sub RunScript(ByVal myBrep As Object)

  'Find and print brep area
  Dim a_mass As New OnMassProperties
  myBrep.AreaMassProperties(a_mass)
  Dim area As Double = a_mass.Area()
  Print("Brep Area = " & area)

  'Find and print brep volume
  Dim v_mass As New OnMassProperties
  myBrep.VolumeMassProperties(v_mass)
  Dim vol As Double = v_mass.Volume()
  Print("Brep Volume = " & vol)

End Sub
```

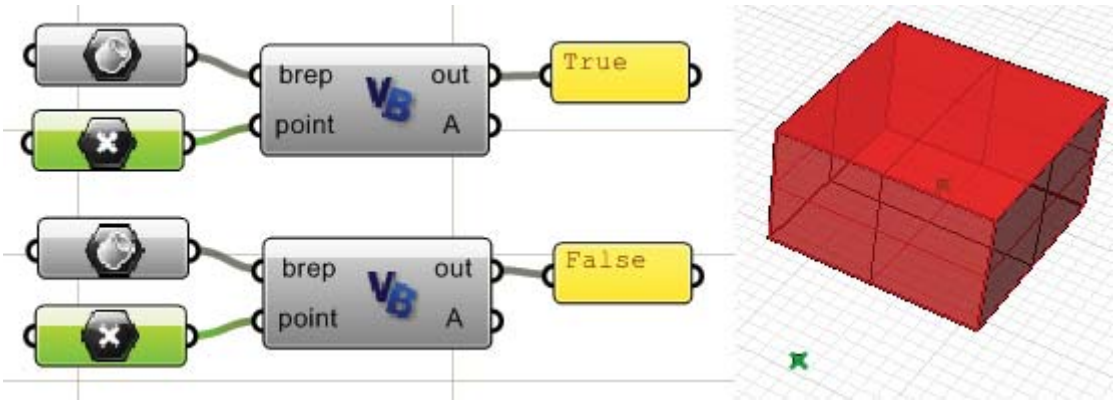


There are few functions that start with “Is” that usually retunes a Boolean (true or false). They inquire about the brep instance you are working with. For example if you like to know whether the brep is closed polysurface, then use OnBrep.IsSolid() function. It is also useful to check if the brep is valid or has valid geometry. Here is a list of these inquiring functions in OnBrep class:

```
Dim brep As New OnBrep
brep.Is…
```



Following example checks if a given point is inside a brep:



Here is the code to check if a point is inside:

```
Sub RunScript(ByVal brep As OnBrep, ByVal point As On3dPoint)
  'Test if input point is inside brep
  Dim tol As Double = doc.AbsoluteTolerance()
  Dim stricktly_inside As Boolean = True
  Dim is_inside As Boolean

  'Call brep function to test the point
  is_inside = brep.IsPointInside(point, tol, stricktly_inside)

  Print(is_inside)

End Sub
```

15.12 Geometry Transformation

OnXform is a class for storing and manipulating transformation matrix. This includes, but not limited to, defining a matrix to move, rotate, scale or shear objects. OnXform's m_xform is a 4x4 matrix of double precision numbers. The class also has functions that support matrix operations such as inverse and transpose. Here is few of the member functions related to creating different transformations.

```
Dim xform As New OnXform
xform.
```

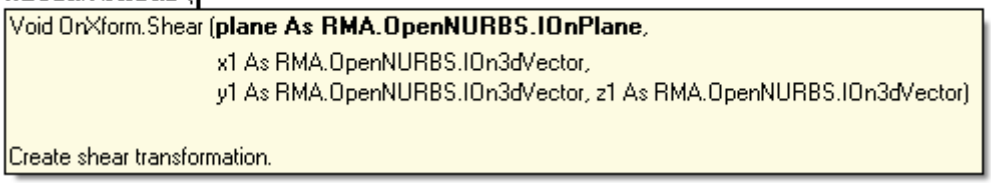
One nice auto-complete feature (available to all functions) is that once a function is selected, the auto-complete shows all overloaded functions. For example, Translation accepts either three numbers or a vector as shown in the picture.

```
Dim xform As New OnXform
xform.Translation(
```

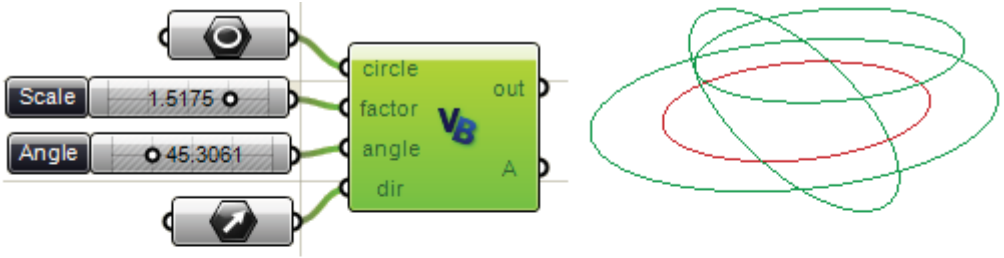
Here are few more OnXform functions:

```
Dim xform As New OnXform
xform.PlanarProjection(
```


```
Dim xform As New OnXform
xform.Shear(
```



The following example takes an input circle and outputs three circles. The first is scaled copy of the original circle, the second is rotated circle and the third is translated one.



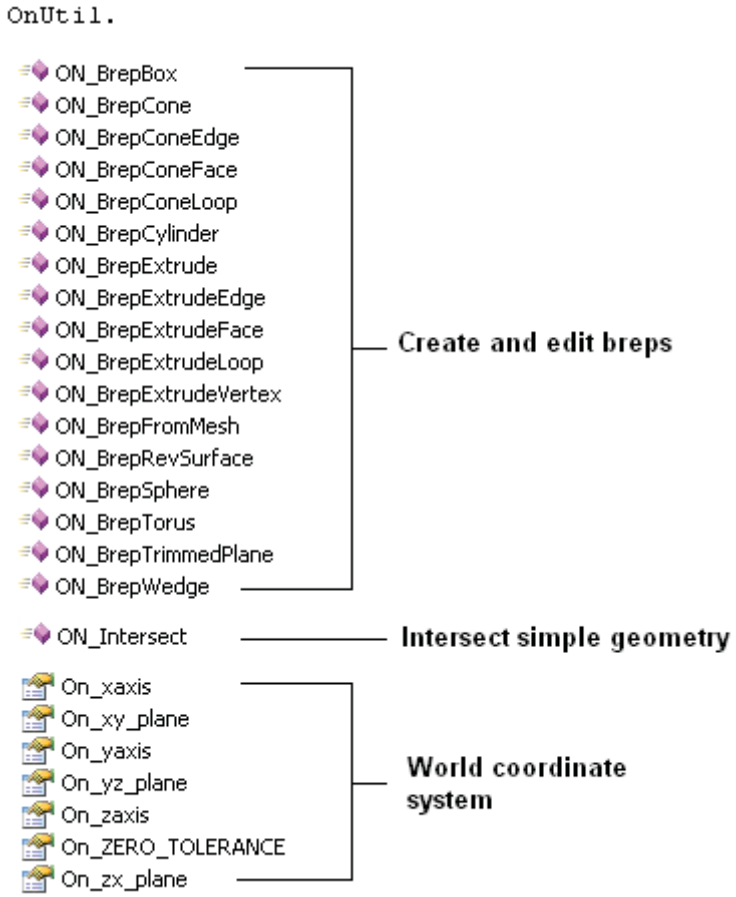

```
Sub RunScript(ByVal circle As OnCircle,
              ByVal factor As Double,
              ByVal angle As Double, ByVal dir As On3dVector)

    Dim circles As New List( Of OnCircle)

    'Scaled circle
    Dim scale As New OnXform
    scale.Scale(OnUtil.On_origin, factor)
    Dim s_circle As New OnCircle(circle)
    s_circle.Transform(scale)
    circles.Add(s_circle)

    'Rotated circle
    Dim rotate As New OnXform
    rotate.Rotation(angle, OnUtil.On_yaxis, OnUtil.On_origin)
    Dim r_circle As New OnCircle(circle)
    r_circle.Transform(rotate)
    circles.Add(r_circle)

    'Moved circle
    Dim move As New OnXform
    move.Translation(dir)
    Dim m_circle As New OnCircle(circle)
    m_circle.Transform(move)
    circles.Add(m_circle)

    'Assign output
    A = circles

End Sub
```

15.13 Global utility functions

Aside from member functions that come within each class, Rhino .NET SDK provides global functions under OnUtil and RhUtil name spaces. We will give examples using few of these functions.

OnUtil

Here a summary of functions available under OnUtil that is related to geometry:



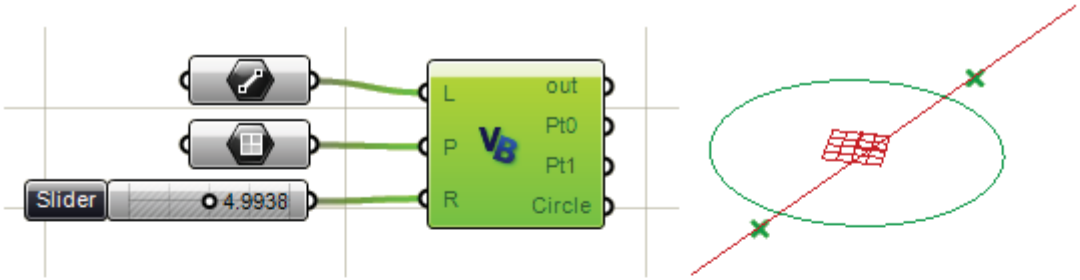
OnUtil intersections

ON_Intersect utility function has 11 overloaded functions. Here is a list of intersected geometry and the return values (the preceding "I" like in "IOnLine" means a constant instance is passed):

Intersected geometry	output
IOnLine with IOnArc	Line parameters (t0 & t1) and Arc points (p0 & p1)
IOnLine with IOnCircle	Line parameters (t0 & t1) and circle points (p0 & p1)
IOnSphere with IOnShere	OnCircle
IOnBoundingBoc with IOnLine	Line parameters (OnInterval)
IOnLine with IOnCylinder	2 points (On3dPoint)
IOnLine with IOnSphere	2 points (On3dPoint)
IOnPlane with IOnSphere	OnCircle
IOnPlane with IOnPlane with IOnPlane	On3dPoint

IOnPlane with IOnPlane	OnLine
IOnLine with IOnPlane	Parameter t (Double)
IOnLine with IOnLine	Parameters a & b (on first and second line as Double)

Here is an example to show the result of intersection a line and plane with a sphere:




```
Sub RunScript(ByVal L As OnLine, ByVal P As OnPlane, ByVal R As Double)

  Dim point0 As New On3dPoint
  Dim point1 As New On3dPoint
  Dim circle0 As New OnCircle

  'Declare the sphere
  Dim sphere As New OnSphere(OnUtil.On_origin, R)

  'Intersect line with sphere
  OnUtil.ON_Intersect(L, sphere, point0, point1)

  'Intersect plane with sphere
  OnUtil.ON_Intersect(P, sphere, circle0)

  'Assign output
  Pt0 = point0
  Pt1 = point1
  Circle = circle0
End Sub
```

RhUtil

Rhino Utility (RhUtil) has many more geometry related functions. The list expands with each new release based on user's requests. This is snapshot of functions related to geometry:

`RhUtil.`

Points

- RhinoArePointsCoplanar
- RhinoPointInPlanarClosedCurve
- RhinoProjectPointsToBreps
- RhinoIsPointInBrep
- RhinoIsPointOnFace

Curve

- RhinoConvertCurveToPolyline
- RhinoCurveBrepIntersect
- RhinoCurveFaceIntersect
- RhinoDivideCurve
- RhinoDoCurveDirectionsMatch
- RhinoExtendCrvOnSrf
- RhinoExtendCurve
- RhinoExtendLineThroughBox
- RhinoExtrudeCurveStraight
- RhinoExtrudeCurveToPoint
- RhinoFairCurve
- RhinoFitCurve
- RhinoFitLineToPoints
- RhinoInterpCurve
- RhinoInterpolatePointsOnSurface
- RhinoMakeCubicBeziers
- RhinoMakeCurveClosed
- RhinoMakeCurveEndsMeet
- RhinoMergeCurves
- RhinoOffsetCurve
- RhinoOffsetCurveOnSrf
- RhinoPlanarClosedCurveContainmentTest
- RhinoPlanarCurveCollisionTest
- RhinoProjectCurvesToBreps
- RhinoPullCurveToMesh
- RhinoRebuildCurve
- RhinoRemoveShortSegments
- RhinoRepairCurve
- RhinoShortPath
- RhinoSimplifyCurve
- RhinoSimplifyCurveEnd

Surface

- RhinoChangeSeam
- RhinoCreateSurfaceFromCorners
- RhinoExtendSurface
- RhinoFitSurface
- RhinoIntersectSurfaces
- RhinoMakeG1Surface
- RhinoRailRevolve
- RhinoRebuildSurface
- RhinoRepairSurface
- RhinoRetrimSurface
- RhinoRevolve
- RhinoSrfControlPtGrid
- RhinoSrfPtGrid

Mesh

- RhinoMeshBooleanDifference
- RhinoMeshBooleanIntersection
- RhinoMeshBooleanSplit
- RhinoMeshBooleanUnion
- RhinoMeshBox
- RhinoMeshCone
- RhinoMeshCylinder
- RhinoMeshObjects
- RhinoMeshOffset
- RhinoMeshPlane
- RhinoMeshSphere
- RhinoRepairMesh
- RhinoSplitDisjointMesh
- RhinoUnifyMeshNormals

Brep

- RhinoBooleanDifference
- RhinoBooleanIntersection
- RhinoBooleanUnion
- RhinoBrepCapPlanarHoles
- RhinoBrepClosestPoint
- RhinoBrepGet2dProjection
- RhinoBrepGet2dSection
- RhinoBrepSplit
- RhinoCreate1FaceBrepFromPoints
- RhinoCreateEdgeSrf
- RhinoIntersectBreps
- RhinoJoinBrepNakedEdges
- RhinoJoinBreps
- RhinoMakePlanarBreps
- RhinoMergeAdjoiningEdges
- RhinoMergeBrepCoplanarFaces
- RhinoMergeBreps
- RhinoRepairBrep
- RhinoSplitBrepFace
- RhinoStraightenBrep
- RhPlanarRegionBoolean
- RhPlanarRegionDifference
- RhPlanarRegionIntersection
- RhPlanarRegionUnion
- RhinoSdkLoft
- RhinoSdkLoftSurface
- RhinoSweep1
- RhinoSweep2

Utility

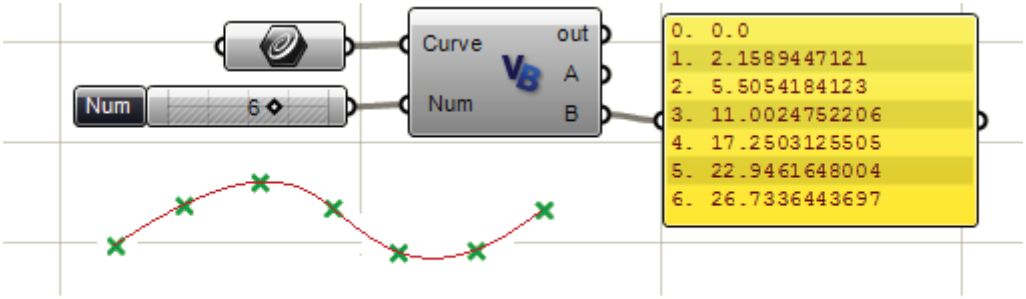
- RhinoActiveCPlane
- RhinoApp
- RhinoFitPlaneToPoints
- RhinoPlaneThroughBox
- RhinoProjectToPlane
- RhinoTriangulate3dPolygon

RhUtil Divide curve

It is possible to divide a curve by number of segments or length on curve using the utility function RhUtil.RhinoDivideCurve. This is a breakdown of function parameters:

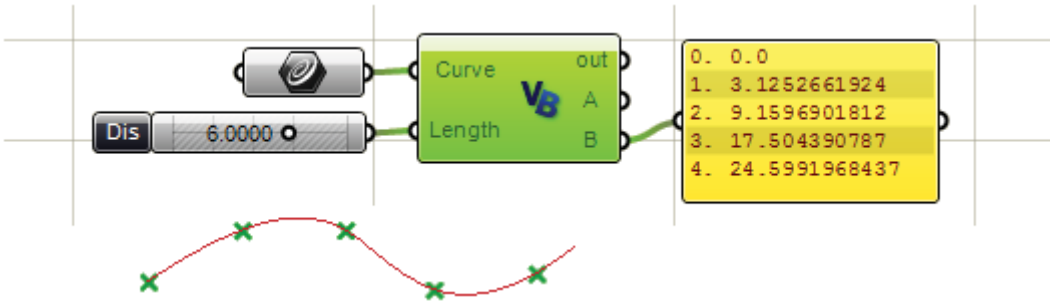
RhinoDivideCurve: Function name.
Curve: constant curve to divide
Num: number of segments.
Len: Curve length to divide by.
False: flag to reverse curve (can be set to True or False)
True: include end point (can be set to True or False)
crv_p: list of divide points
crv_t: list off divide points parameters on curve

Divide curve by number of segments example:




```
Sub RunScript(ByVal Curve As OnCurve, ByVal Num As Integer)

    Dim crv_p As New On3dPointArray
    Dim crv_t As New Arraydouble

    'A utility function to divide by number or curve length
    RhUtil.RhinoDivideCurve(Curve, Num, 0, False, True, crv_p, crv_t)

    A = crv_p
    B = crv_t

End Sub
```

Divide curve by arc length example:



```
Sub RunScript(ByVal Curve As OnCurve, ByVal Len As Double)

  Dim crv_p As New On3dPointArray
  Dim crv_t As New Arraydouble

  'A utility function to divide by number or curve length
  RhUtil.RhinoDivideCurve(Curve, 0, Len, False, True, crv_p, crv_t)

  A = crv_p
  B = crv_t

End Sub
```

RhUtil Curve through points (interpolate curve)

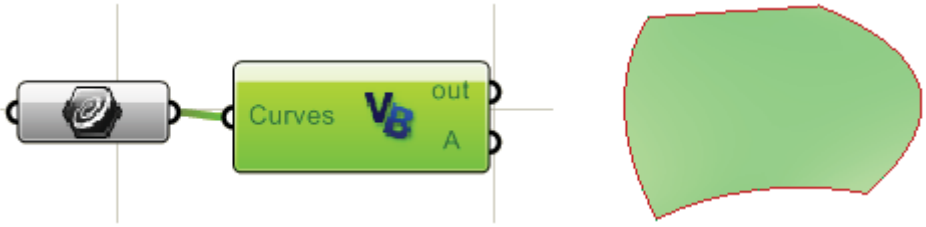
RhinoInterpCurve: Function name.
3: curve degree
pt_array: points to create a curve through
Nothing: Start tangent.
Nothng: End tangent
0: Uniform knots

The following example takes as an input a list of On3dPoints and outputs a nurbs curve that goes through these points.

```
Sub RunScript(ByVal Points As List(Of On3dPoint))

  Dim pt_array As New ArrayOn3dPoint
  Dim i As Integer

  For i = 0 To Points.Count() - 1
    pt_array.Append(Points(i))
  Next

  'Create an interpolated nurbs curve
  Dim crv As New OnNurbsCurve
  crv = RhUtil.RhinoInterpCurve(3, pt_array, Nothing, Nothing, 0)

  If( crv.IsValid() ) Then
    'Set return vlue to list
    A = crv
  End If

End Sub
```

RhUtil Create edge surface

The input in the following example is a list of four curves and the output is an edge surface.




```
Sub RunScript(ByVal Curves As List(Of OnCurve))

  Dim nc_list(3) As OnNurbsCurve
  For i As Integer = 0 To 3
    nc_list(i) = New OnNurbsCurve()
  Next

  ' Get nurb form of each curve
  For i As Integer = 0 To 3
    Curves(i).GetNurbForm(nc_list(i))
  Next

  ' Create the edgesurface
  Dim Brep As OnBrep = RhUtil.RhinoCreateEdgeSrf(nc_list)

  A = Brep
End Sub
```

16 *Help*

Where to find more information about Rhino DotNET SDK

There is a lot of information and samples in McNeel Wiki. Developers actively add new material and examples to it all the time. This is great resource and you can find it here:
http://en.wiki.mcneel.com/default.aspx/McNeel/Rhino4DotNetPlugIns.html

Forums and discussion groups

Rhino community is very active and supportive. Grasshopper discussion forum is a great place to start.
http://grasshopper.rhino3d.com/

You can also post questions to the Rhino Developer Newsgroup. You can access the developer newsgroup from McNeel's developer's page:
http://www.rhino3d.com/developer.htm

Debugging with Visual Studio

For more complex code that is hard to debug from within Grasshopper script component, you can use Visual Studio Express that Microsoft provides for free or the full version of developer studio.

Details of where to get the express visual studio and how to use it are found here:
http://en.wiki.mcneel.com/default.aspx/McNeel/DotNetExpressEditions

If you have access to the full version of Visual Studio, then it is even better. Check the following page to help you get set up:
http://en.wiki.mcneel.com/default.aspx/McNeel/Rhino4DotNetPlugIns.html

Grasshopper Scripting Samples

Grasshopper Gallery and discussion forum has many examples of scripted components that you will probably find useful.