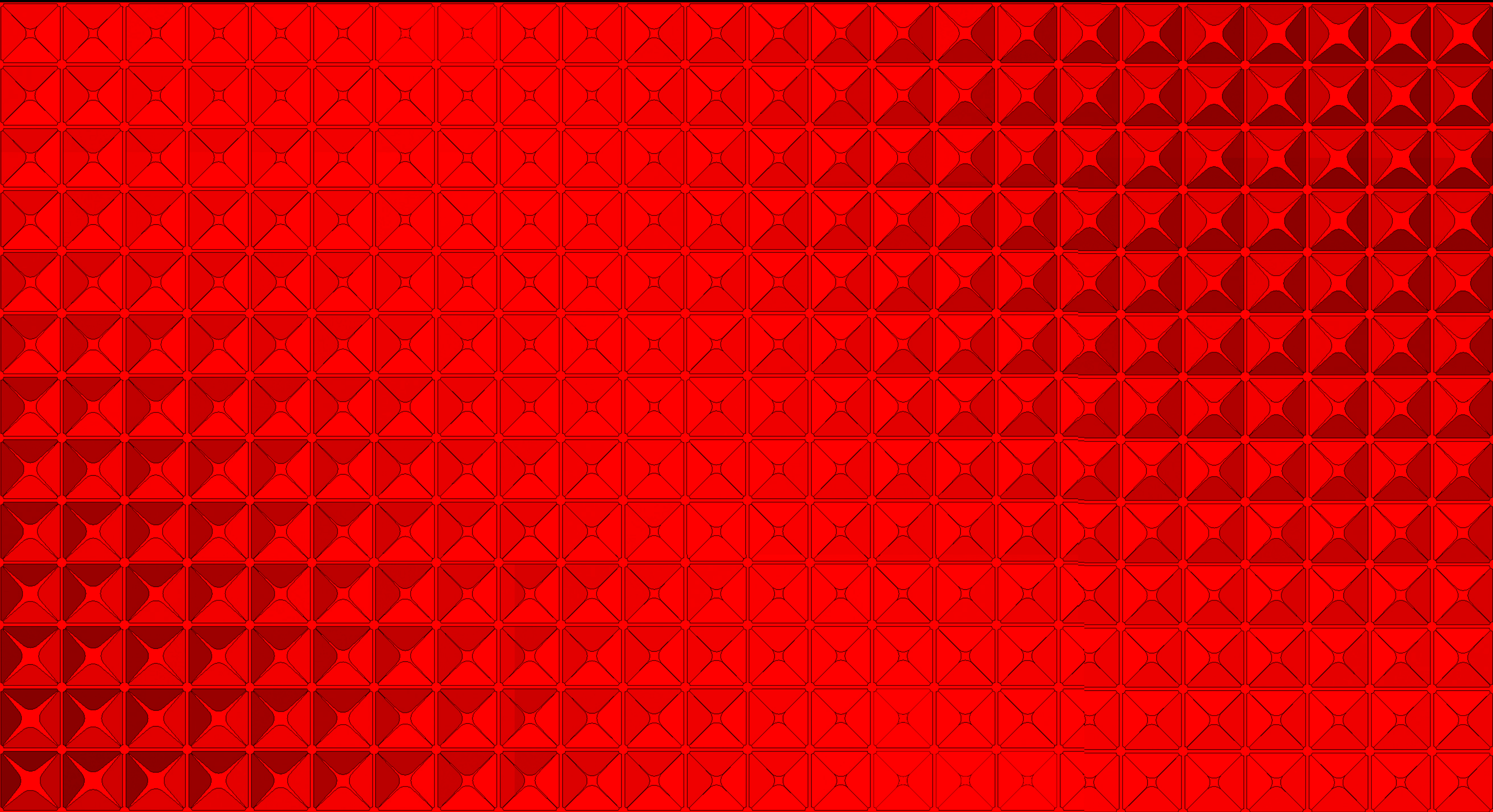
GRASSHOPPER PRIMER

FOR VERSION 0.6.0007
BY ANDREW PAYNE & RAJAA ISSA



The Grasshopper Primer, Second Edition – for version 0.6.0007

Introduction

Welcome to the wonderful new world of Grasshopper. This is the second edition of this primer and it wouldn't have been possible without the tremendous contribution from Rajaa Issa. Rajaa is a developer at Robert McNeel and Associates and is the author of several other Rhino plugins including ArchCut and the ever popular PanelingTools. This revision provides a much more comprehensive guide than the first edition, with over 70 new pages dedicated to creating your own customized scripting components.

The release of this manual coincides with two events; the first being the new release of Grasshopper version 0.6.0007 which proves to be a giant upgrade to the already robust Grasshopper platform. Existing users will find subtle, and some not so subtle changes to how data is stored in the current version; making some previous definitions outdated or even unusable. It is the hope that this primer will help new and existing users navigate many of the changes to the software system. The second event that overlaps with this publication is the conference titled FLUX: Architecture in a Parametric Landscape that is being held at the California College of the Arts. The event explores contemporary architecture and design through its relationship with changes in design technologies such as parametric modeling, digital fabrication, and scripting. Among other things, the event will consist of an exhibition and a series of workshops dedicated to parametric software systems. I am honored to be teaching an Introduction to Grasshopper modeling, while Rajaa Issa and Gil Akos will be handling the Advanced Grasshopper modeling and VB.Net Scripting workshop.

We have included a lot of new information in this primer and we hope that it will continue to be a great resource for those wanting to learn more about the plugin. However, one of the greatest assets this software has going for it is you, the user, because when more people begin to explore and understand parametric modeling, it helps the entire group. I would encourage each person who reads this primer to join the growing online community and post your questions to the forum, as there is almost always someone willing to share a solution to your problem. To learn more, visit http://www.grashopper.rhino3d.com.

Thanks and good luck!

Andrew Payne
LIFT architects
www.liftarchitects.com

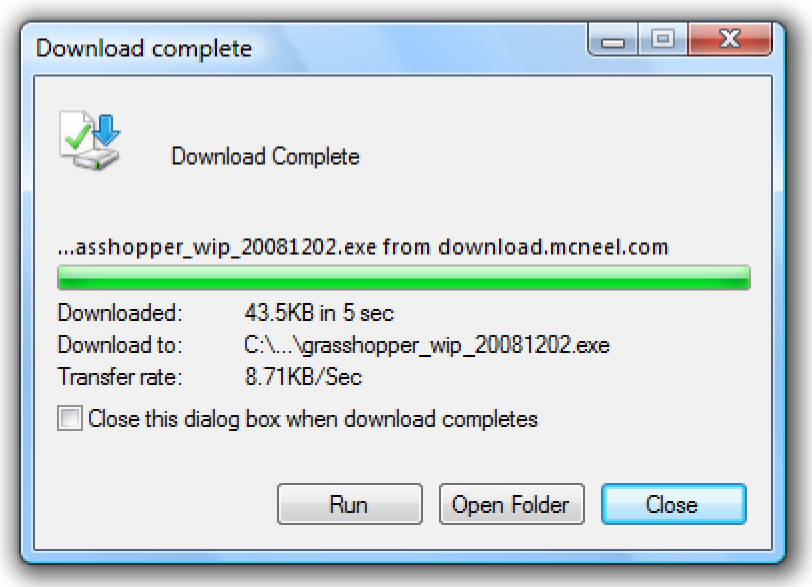
Rajaa Issa
Robert McNeel and Associates
http://www.rhino3d.com/

Table of Contents

1 *Getting Started*

Installing Grasshopper

To download the Grasshopper plug-in, visit **http://grasshopper.rhino3d.com/**. Click on the **Download** link in the upper left hand corner of the page, and when prompted on the next screen, enter your email address. Now, right click on the download link, and choose **Save Target As** from the menu. Select a location on your hard drive (note: the file cannot be loaded over a network connection, so the file must be saved locally to your computer's hard drive) and save the executable file to that address.


Download complete

Download Complete

...asshopper_wip_20081202.exe from download.mcneel.com

Downloaded: 43.5KB in 5 sec
Download to: C:\...\grasshopper_wip_20081202.exe
Transfer rate: 8.71KB/Sec

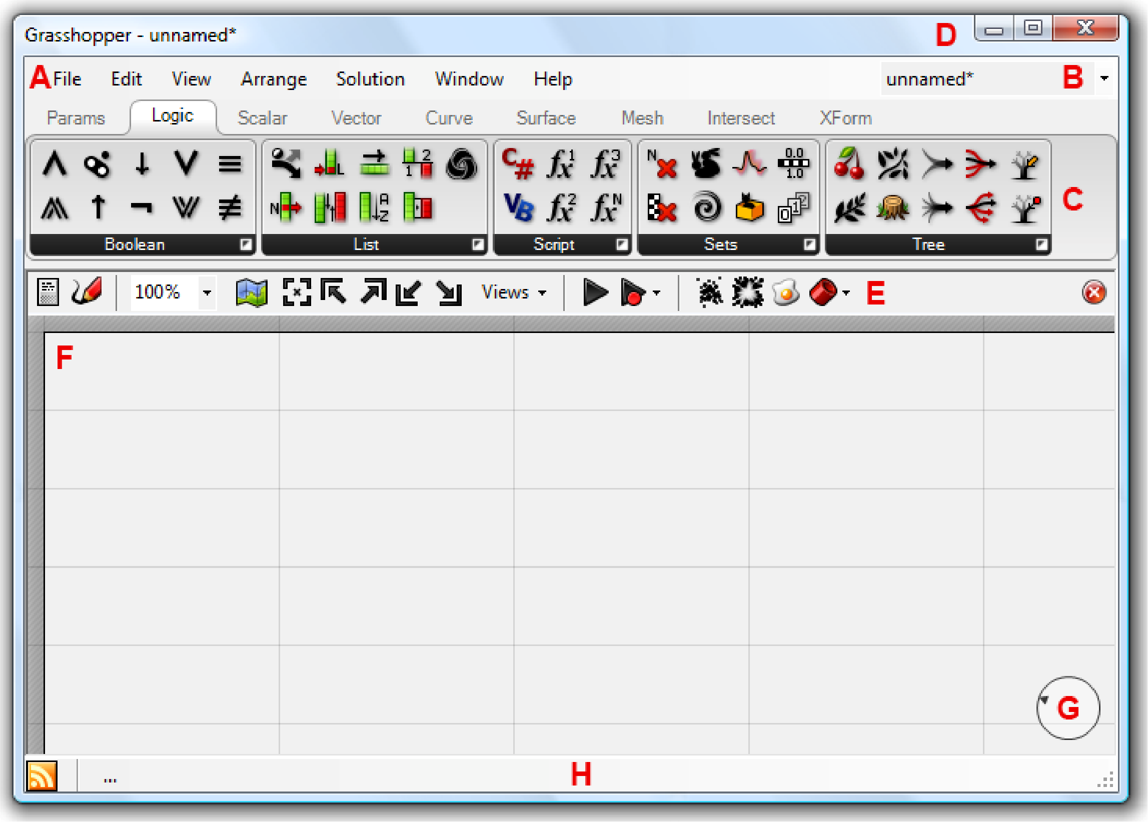
□ Close this dialog box when download completes

Select **Run** from the download dialogue box follow the installer instructions. (note: you must have Rhino 4.0 with SR4b or higher already installed on your computer for the plug-in to install properly)

2 *The Interface**

The Main Dialog

Once you have loaded the plug-in, type "Grasshopper" in the Rhino command prompt to display the main Grasshopper window:



This interface contains a number of different elements, most of which will be very familiar to Rhino users:

A. The Main Menu Bar

The menu is similar to typical Windows menus, except for the file-browser control on the right **B**. You can quickly switch between different loaded files by selecting them through this drop-down box. Be careful when using shortcuts since they are handled by the active window. This could either be Rhino, the Grasshopper plug-in or any other window inside Rhino. Since there is no undo available yet you should be cautious with the Ctrl-X, Ctrl-S and Del shortcuts.

B. File Browser Control

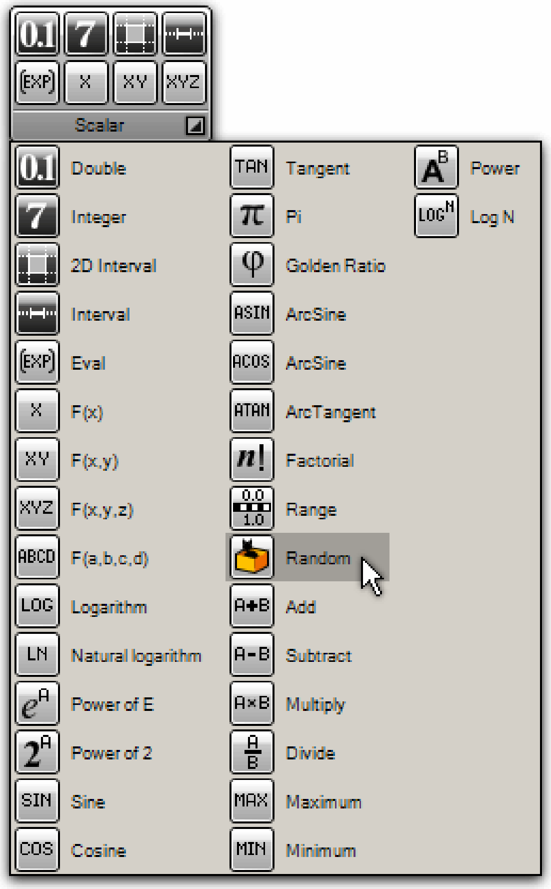
As discussed in the previous section, this drop down menu can be used to switch between different loaded files.

* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryPluginInterfaceExplained.html

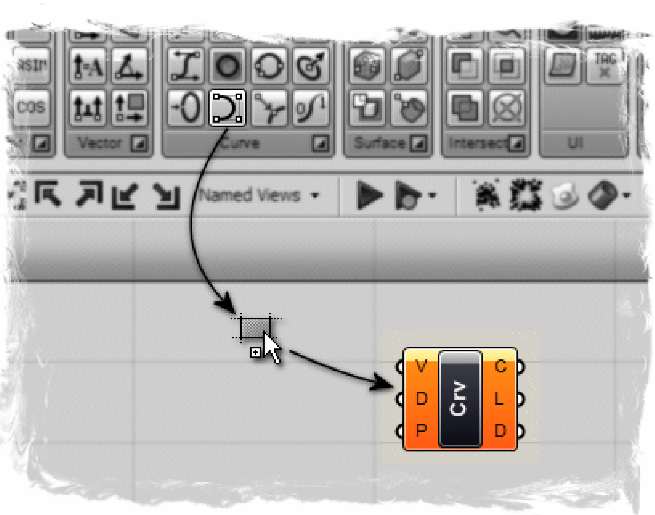
C. Component Panels

This area exposes all component categories. All components belong to a certain category (such as "Params" for all primitive data types or "Curves" for all curve related tools) and all categories are available as unique toolbar panels. The height and width of the toolbars can be adjusted, allowing for more or fewer on-screen buttons per category.

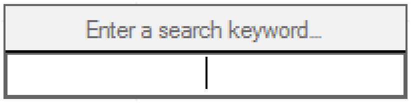
The toolbar panels themselves contain all the components that belong to that category. Since there are a potentially large number of these, it only shows the N most recently used items. In order to see the entire collection, you have to click on the bar at the bottom of the Panel:



This will pop up the category panel, which provides access to all objects. You can either click on the objects in the popup list, or you can drag directly from the list onto the canvas. Clicking on items in the category panel will place them on the toolbar for easy future reference. **Clicking on buttons will not add them to the Canvas!** You **must** drag them onto the Canvas in order to add them:



You can also find components by name, by double-clicking anywhere on the canvas; launching a pop-up search box. Type in the name of the component you are looking for and you will see a list of parameters or components that match your request.

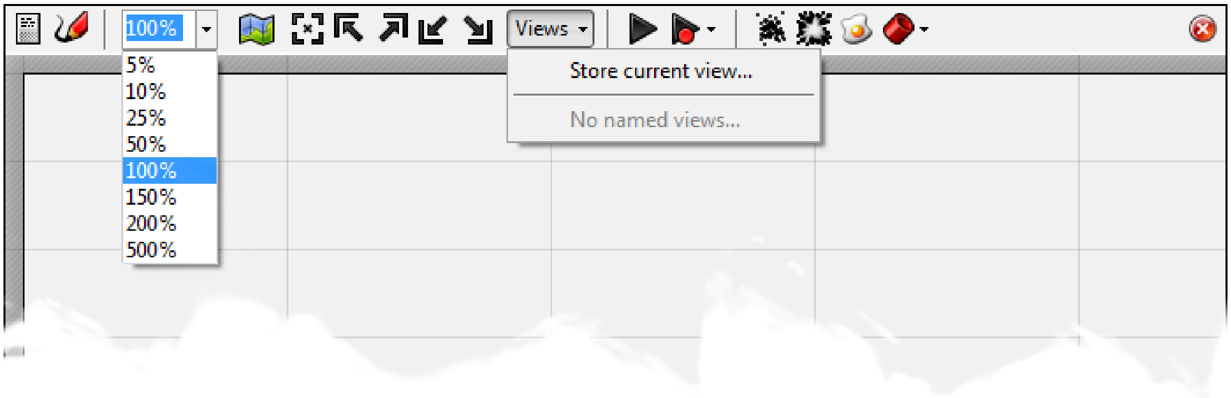


The Window Title Bar: D

The Editor Window title bar behaves different from most other dialogs in Microsoft Windows. If the window is not minimized or maximized, double clicking the title bar will fold or unfold the dialog. This is a great way to switch between the plug-in and Rhino because it minimizes the Editor without moving it to the bottom of the screen or behind other windows. Note that if you close the Editor, the Grasshopper geometry previews in the viewports will disappear, but the files won't actually be closed. The next time you run the _Grasshopper command, the window will come back in the same state with the same files loaded.

The Canvas Toolbar: E

The canvas toolbar provides quick access to a number of frequently used features. All the tools are available through the menu as well, and you can hide the toolbar if you like. (It can be re-enabled from the View menu).



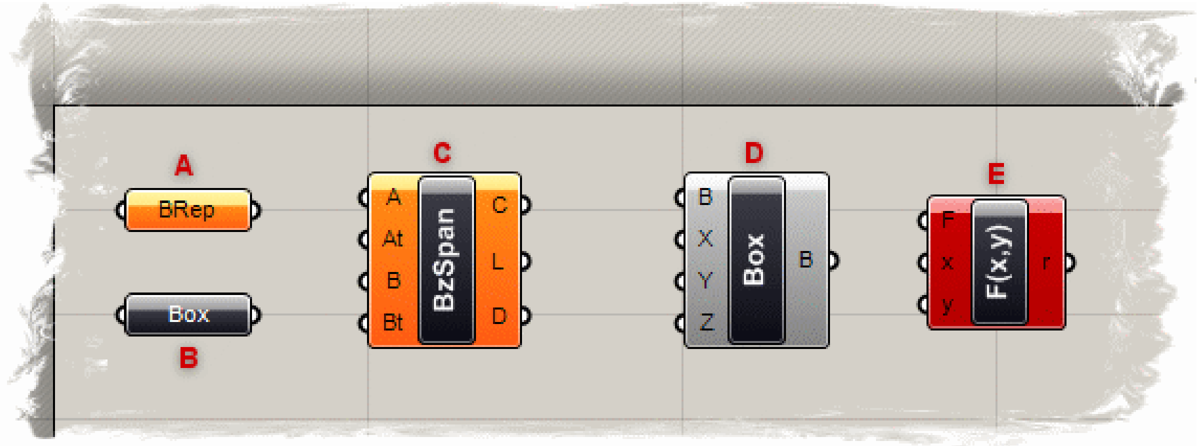
The canvas toolbar exposes the following tools (from left to right):

1. Definition properties editor
2. Sketch tool:
   The sketch tool works like most pencil-type tools of Photoshop or Window's Paint. Default controls of the sketch tools allow changes of line weight, line type, and color. However, it can be quite difficult to draw straight lines or pre-defined shapes. In order to solve this problem, draw out any sketch line onto the canvas. Right click on the line, and select "Load from Rhino", and select any pre-defined shape in your Rhino scene (This can be any 2d shape like a rectangle, circle, star...etc). Once you have selected your referenced shape, hit Enter, and your previously drawn sketch line will be reconfigured to your Rhino reference shape.
3. Zoom defaults
4. Navigation Map pops open a smaller diagrammatic window of the canvas allows you to quickly move around the canvas without having to scroll and pan. This feature is similar to the Navigator window in Photoshop.
5. Zoom Extents (will adjust the zoom-factor if the definition is too large to fit on the screen)
6. Focus corners (these 4 buttons will focus on the 4 corners of the definition)
7. Named views (exposes a menu to store and recall named views)
8. Rebuild solution (forces a complete rebuild of the history definition)
9. Rebuild events (by default, Grasshopper responds to changes in Rhino and on the Canvas. You can disable these responses though this menu)
10. Cluster compactor (turn all selected objects into a Cluster object) **Cluster objects are not finished yet and are more than likely going to be completely re-worked in the future. Caution using these in your current files.**

11. Cluster exploder (turn all selected clusters into loose objects) **Cluster objects are not finished yet and are more than likely going to be completely re-worked in the future. Caution using these in your current files.**
12. Bake tool (turns all selected components into actual Rhino objects)
13. Preview settings (Grasshopper geometry is previewed by default. You can disable the preview on a per object basis, but you can also override the preview for all objects. Switching off Shaded preview will vastly speed up some scenes that have curved or trimmed surfaces.
14. Hide button. This button hides the canvas toolbar, you can switch it back on through the View menu.

F: The Canvas

This is the actual editor where you define and edit the history network. The Canvas hosts both the objects that make up the definition and some UI widgets **G**.
Objects on the canvas are usually color coded to provide feedback about their state:



A) Parameter. A parameter which contains warnings is displayed as an orange box. Most parameters are orange when you drop them onto the canvas since the lack of data is considered to be a warning.

B) Parameter. A parameter which contains neither warnings nor errors.

C) Component. A component is always a more involved object, since it contains input and output parameters. This particular component has at least 1 warning associated with it. You can find warning and errors through the context menu of objects.

D) Component. A component which contains neither warnings nor errors.

E) Component. A component which contains at least 1 error. The error can come either from the component itself or from one of its input/output parameters. We will learn more about Component Structures in the following chapters.

All selected objects are drawn with a green overlay (not shown).

G: UI Widgets

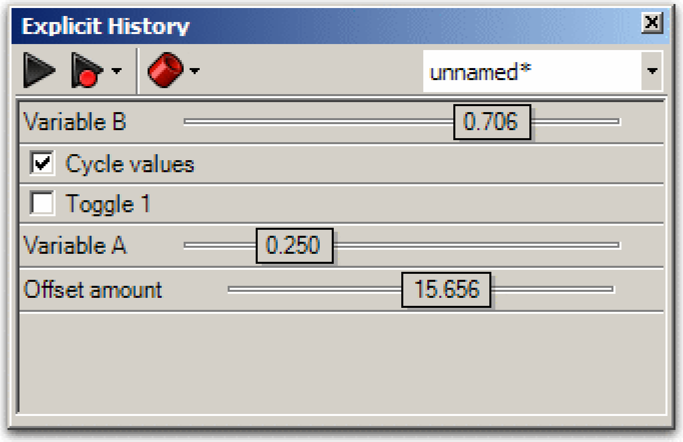
Currently, the only UI widget available is the Compass, shown in the bottom right corner of the Canvas. The Compass widget gives a graphic navigation device to show where your current viewport is in relation to the extents of the entire definition. The Widgets can be enabled/disabled through the View menu.

H: The Status Bar

The status bar provides feedback on the selection set and the major events that have occurred in the plug-in. Key information about whether or not you have any errors or warnings in your definition will be displayed here.
The square orange icon on the bottom left of the Status Bar is a live RSS reader of the Grasshopper forum. By clicking on this icon, a list of the most recent threads, linked to the Grasshopper user group website, will be displayed. Selecting any one of the threads, will take you directly to the discussion posted by one of the user group members. You can visit the Grasshopper user group website at:
http://grasshopper.rhino3d.com/

The Remote Control Panel:

Since the Grasshopper window is quite large, you may not want it on the screen all the time. Of course you can minimize or collapse it, but then you can't tweak the values anymore. If you want a minimal interface to the values inside the currently active definition, you can enable the Remote Panel. This is a docking dialog that keeps track of all sliders and boolean switches (and possibly other values as well in future releases):



Variable B 0.706

☑ Cycle values

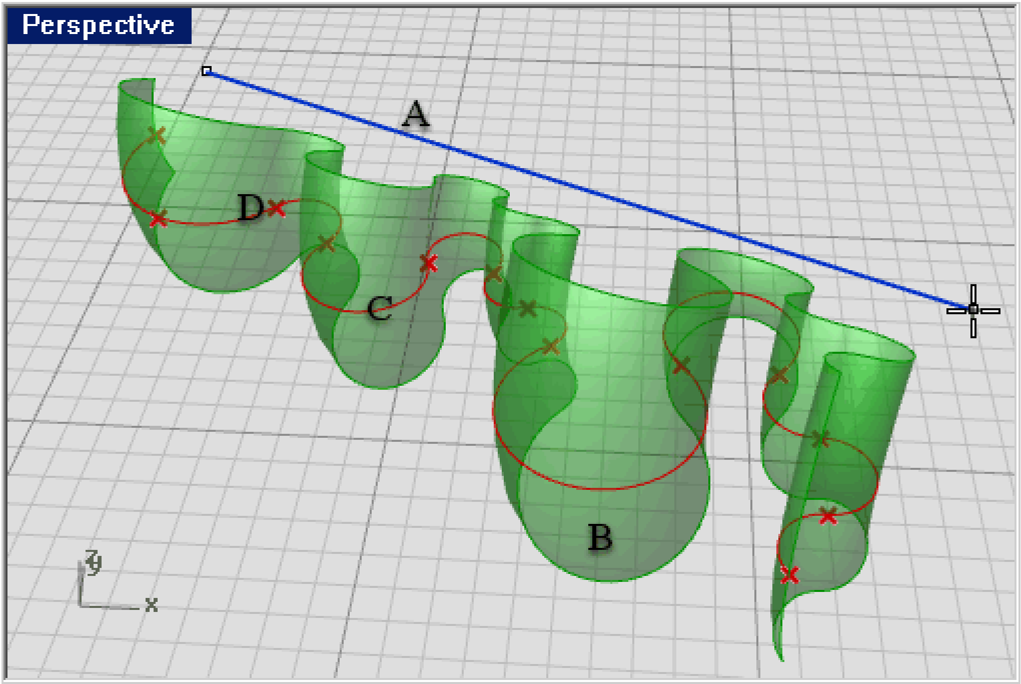
□ Toggle 1

Variable A 0.250

Offset amount 15.656

The Remote panel also provides basic preview, event and file-switching controls. You can enable the panel through the View menu of the Main window, or through the _GrasshopperPanel command.

Viewport Preview Feedback:



A) **Blue** feedback geometry means you are currently picking it with the mouse.

B) **Green** geometry in the viewport belongs to a component which is currently selected.

C) **Red** geometry in the viewport belongs to a component which is currently unselected.

D) **Point geometry** is drawn as a cross rather than a rectangle to distinguish it from Rhino point objects.

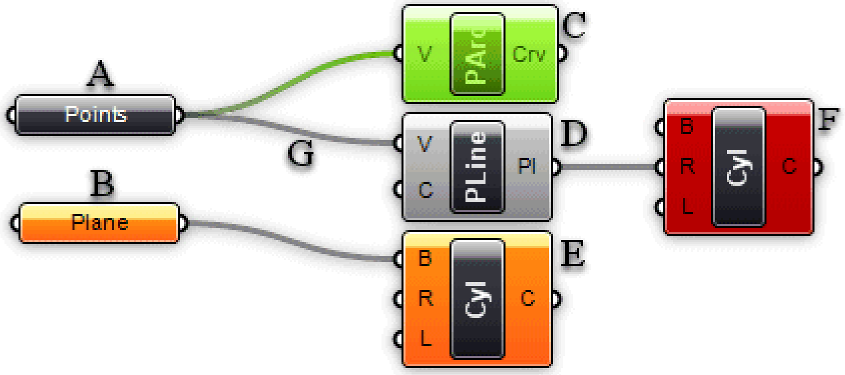
3 Grasshopper Objects*

Grasshopper Definition Objects

A Grasshopper definition can consist of many different kinds of objects, but in order to get started you only need to familiarize yourself with two of them:

- Parameters
- Components

Parameters contain data, meaning that they **store** stuff. Components contain actions, meaning that they **do** stuff. The following image shows some of the possible objects you are likely to encounter in a Grasshopper definition:



A) A parameter which contains data. Since there is no wire coming out the left side of the object, it does not inherit its data from elsewhere. Parameters which do not contain errors or warnings are thin, black blocks with horizontal text.

B) A parameter which contains no data. Any object which fails to collect data is considered suspect in an Explicit History Definition since it appears to be wasting everyone's time and money. Therefore, all parameters (when freshly added) are orange, to indicate they do not contain any data and have thus no functional effect on the outcome of the History Solution. Once a parameter inherits or defines data, it will become black.

C) A selected component. All selected objects have a green sheen to them.

D) A regular component.

E) A component containing warnings. Since a component is likely to contain a number of input and output parameters, it is never clear which particular object generated the warning by just looking at the component. There may even be multiple sources of warnings. You'll have to use the context menu (see below) in order to track down the problems. **Note that warnings do not necessarily have to be fixed.** They may be completely legit.

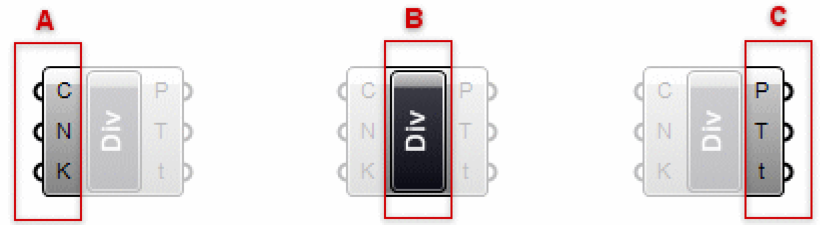
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryPluginObjectsExplained.html

F) A component containing errors. Similar to warnings, it is not possible to see where the error was generated in a component. You'll need to use the context menu (see below). Note that a component which contains both warnings and errors will appear red, the error color takes precedence over the warning color.

G) A connection. Connections always appear between an output and an input parameter. There is no limit to how many connections any particular parameter may contain, but it is not allowed to create a setup with cyclical/recursive connections. Such a recursion is detected and the entire Solution is short-circuited when it occurs, resulting in an error message in the first component or parameter that was detected to be recursive. For more information on connections, see chapter about Data Inheritance.

Component Parts

A component usually requires data in order to perform its actions, and it usually comes up with a result. That is why most components have a set of nested parameters, referred to as Input and Output parameters respectively. Input parameters are positioned along the left side, output parameters along the right side:

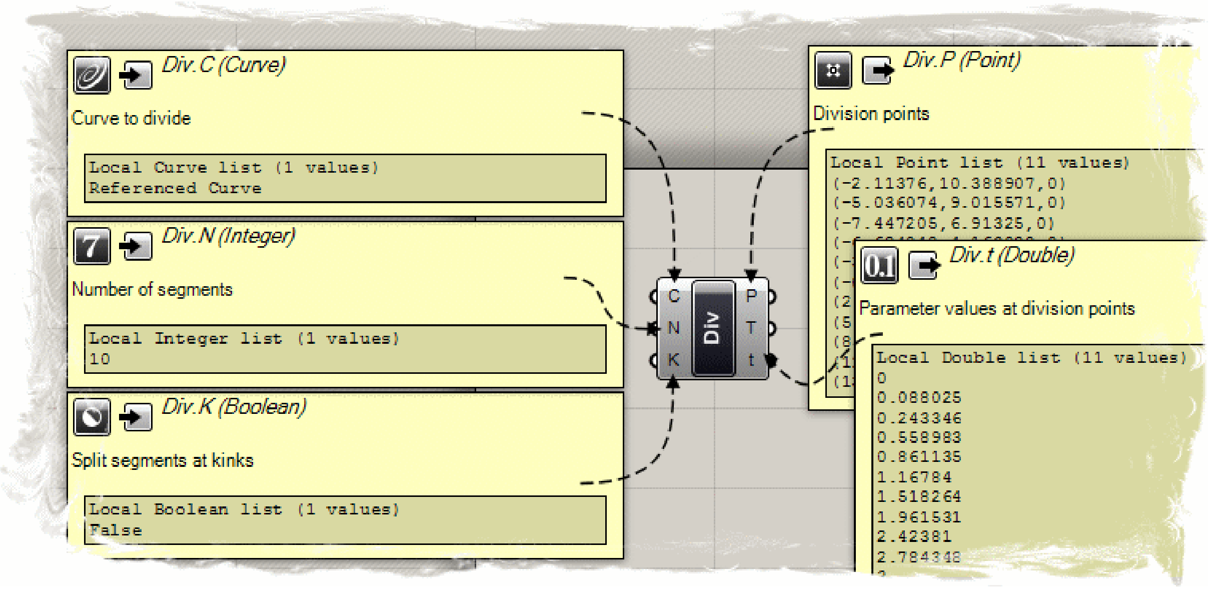


A) The three input parameters of the Division component. By default, parameter names are always extremely short. You can rename each parameter as you please.

B) The Division component area (usually contains the name of the component)

C) The three output parameters of the Division component.

When you hover your mouse over the individual parts of a Component object, you'll see different tooltips that indicate the particular type of the (sub)object currently under the mouse. Tooltips are quite informative since they tell you both the type and the data of individual parameters:



Div.C (Curve)

Curve to divide

```
Local Curve list (1 values)
Referenced Curve
```

Div.N (Integer)

Number of segments

```
Local Integer list (1 values)
10
```

Div.K (Boolean)

Split segments at kinks

```
Local Boolean list (1 values)
False
```

Div.P (Point)

Division points

```
Local Point list (11 values)
(-2.11376,10.388907,0)
(-5.036074,9.015571,0)
(-7.447205,6.91325,0)
```

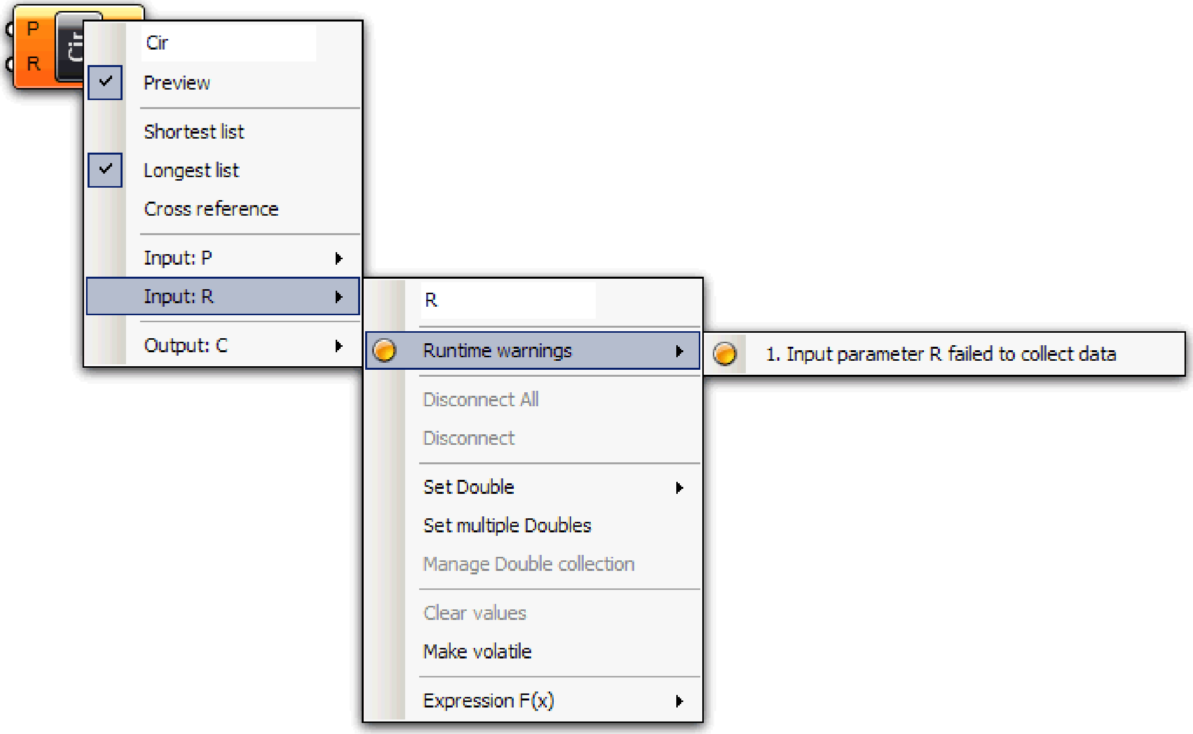
Div.t (Double)

Parameter values at division points

```
Local Double list (11 values)
0
0.088025
0.243346
0.558983
0.861135
1.16784
1.518264
1.961531
2.42381
2.784348
```

Using Context Popup Menus

All objects on the Canvas have their own context menus that expose most of the features for that particular component. Components are a bit trickier, since they also expose (in a cascading style) all the menus of the sub-objects they contain. For example, if a component turns orange it means that it, or some parameter affiliated with the component, generated a warning. If you want to find out what went wrong, you need to use the component context menu:



1. Input parameter R failed to collect data

Here you see the main component menu, with the cascading menu for the "R" input parameter. The menu usually starts with an editable text field that lists the name of the object in question. You can change the name to something more descriptive, but by default all names are extremely short to minimize screen-real-estate usage. The second item in the menu (Preview flag) indicates whether or not the geometry produced/defined by this object will be visible in the Rhino viewports. Switching off preview for components that do not contain vital information will speed up both the Rhino viewport framerate and the time taken for a History Solution (in case meshing is involved). If the preview for a parameter or a component is disabled, it will be drawn with a faint white hatch. Not all parameters/components can be drawn in viewports (numbers for example) and in these cases the Preview item is usually missing.

The context menu for the "R" input parameter contains the orange warning icon, which in turn contains a list (just 1 warning in this case) of all the warnings that were generated by this parameter.

4 Persistent Data Management*

Types of Data

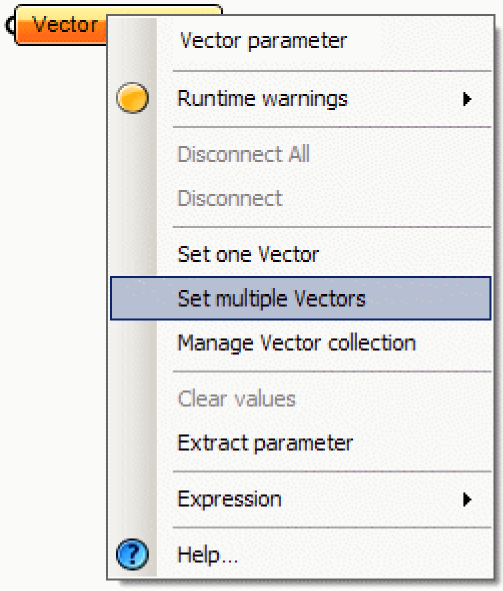
Parameters are only used to store information, but most parameters can store two different kinds; Volatile and Persistent data. Volatile data is inherited from one or more source parameters and is destroyed (i.e. recollected) whenever a new solution starts. Persistent data is data which has been specifically set by the user. Whenever a parameter is hooked up to a source object the persistent data is ignored, but not destroyed.

(The exception here are output parameters which can neither store permanent records nor define a set of sources. Output parameters are fully under the control of the component that owns them.)

Persistent data is accessed through the menu, and depending on the kind of parameter has a different manager. Vector parameters for example allow you to set both single and multiple vectors through the menu.

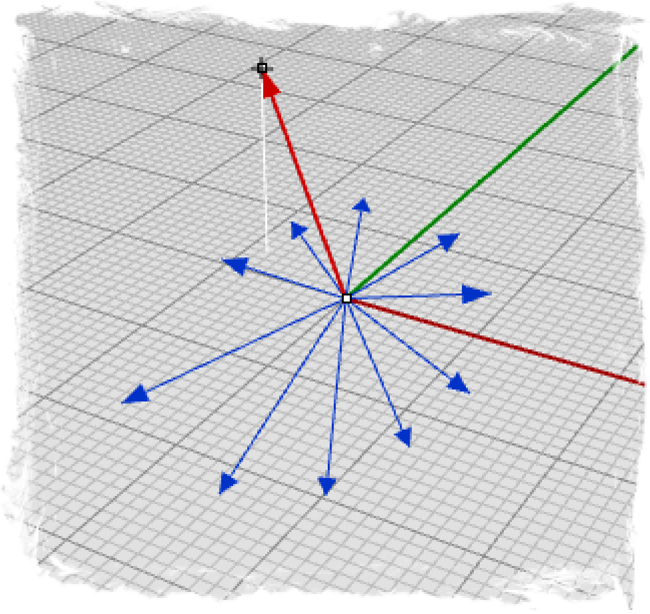
But, let's back up a few steps and see how a default Vector parameter behaves. Once you drag+drop it from the Params Panel onto the canvas, you will see the following:

The parameter is orange, indicating it generated a warning. It's nothing serious, the warning is simply there to inform you that the parameter is empty (it contains no persistent records and it failed to collect volatile data) and thus has no effect on the outcome of a history solution. The context menu of the Parameter offers 2 ways of setting persistent data: single and multiple:

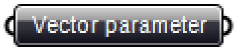


* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryPersistentDataRecordManagement.html

Once you click on either of these menu items, the Grasshopper window will disappear and you will be asked to pick a vector in one of the Rhino viewports:



Once you have defined all the vectors you want, you can press Enter and they will become part of the Parameters Persistent Data Record. This means the Parameter is now no longer empty and it turns from orange to black:



At this point you can use this parameter to 'seed' as many objects as you like with identical vectors.

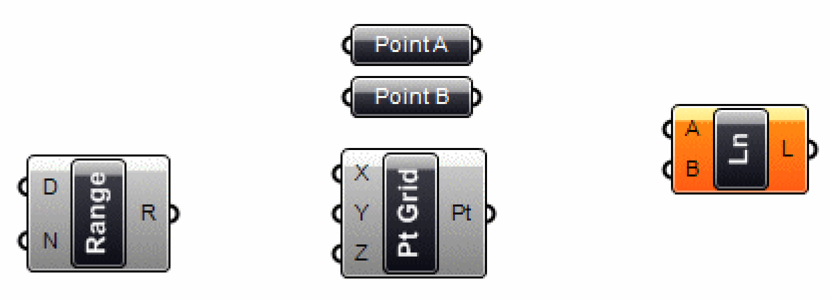
5 Volatile Data Inheritance*

Data Inheritance

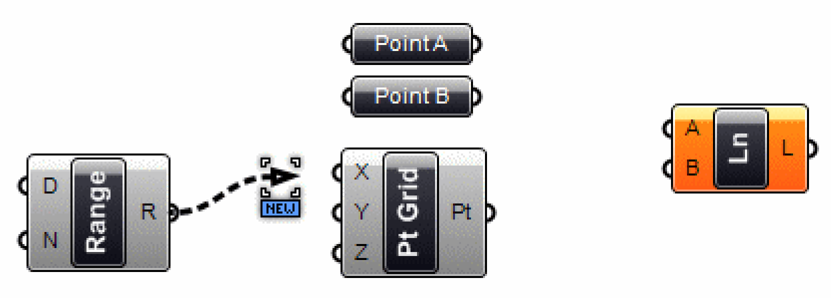
Data is stored in parameters (either in Volatile or Persistent form) and used in components. When data is not stored in the permanent record set of a parameter, it must be inherited from elsewhere. Every parameter (except output parameters) defines where it gets its data from and most parameters are not very particular. You can plug a double parameter (which just means that it is a number with some decimal number) into an integer source and it will take care of the conversion. The plug-in defines many conversion schemes but if there is no translation procedure defined, the parameter on the receiving end will generate a conversion error. For example, if you supply a Surface when a Point is needed, the Point parameter will generate an error message (accessible through the menu of the parameter in question) and turn red. If the parameter belongs to a component, this state of red-ness will propagate up the hierarchy and the component will become red too, even though it may not contain errors of itself.

Connection management

Since Parameters are in charge of their own data sources, you can get access to these settings through the parameter in question. Let's assume we have a small definition containing three components and two parameters:

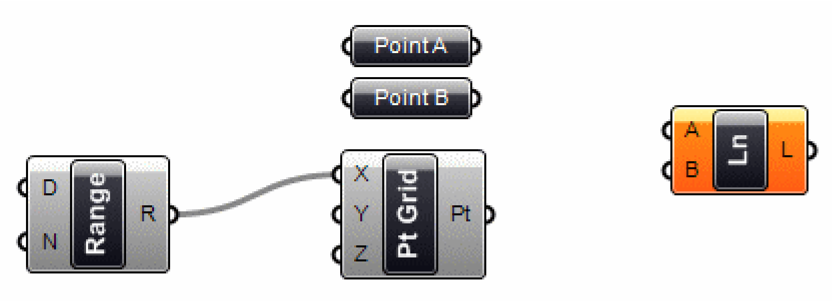


At this stage, all the objects are unconnected and we need to start hooking them up. It doesn't matter in what order we do this, but lets go from left to right. If you start dragging near the little circle of a parameter (what us hip people call a "grip") a connecting wire will be attached to the mouse:

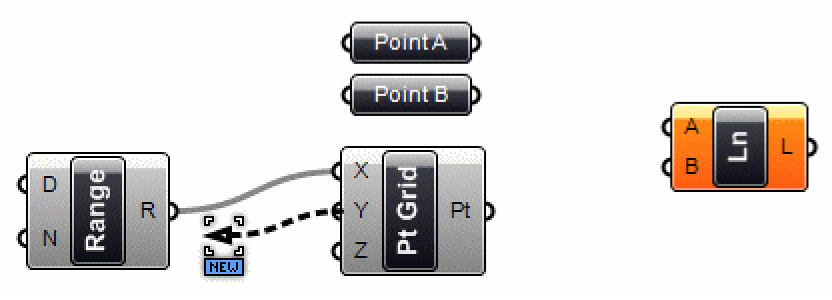


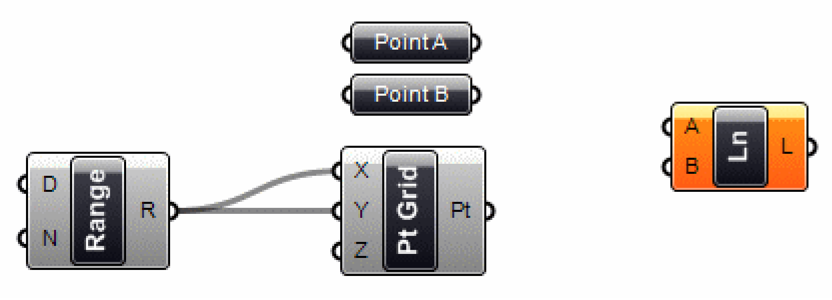
Once the mouse (with the Left Button still firmly pressed) hovers over a potential target Parameter, the wire will attach and become solid. This is not a permanent connection until you release the mouse button:

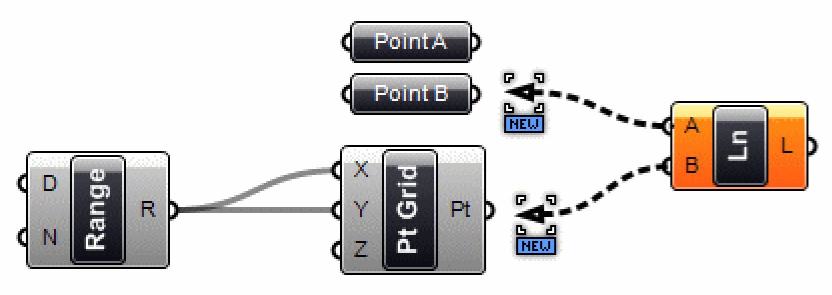
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryVolatileDataInheritance.html

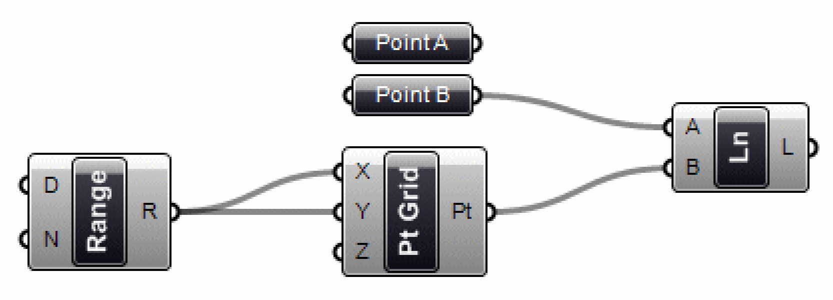


We can do the same for the "Y" parameter of the PtGrid component and the "A" and "B" parameters of the Line component: Click+Drag+Release...

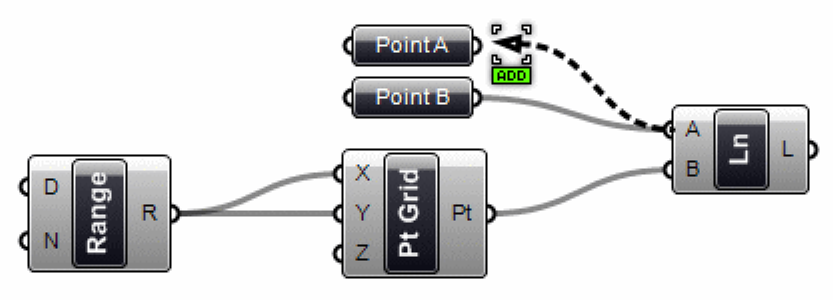


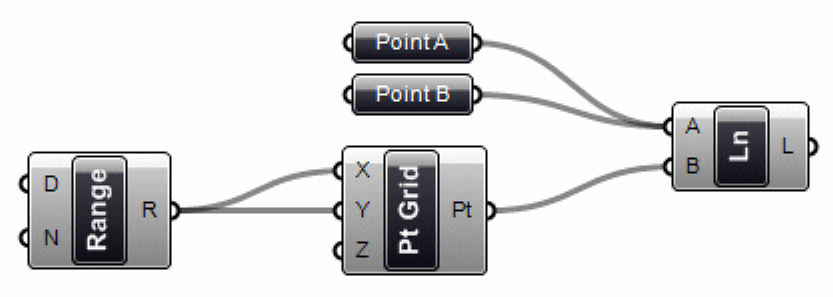






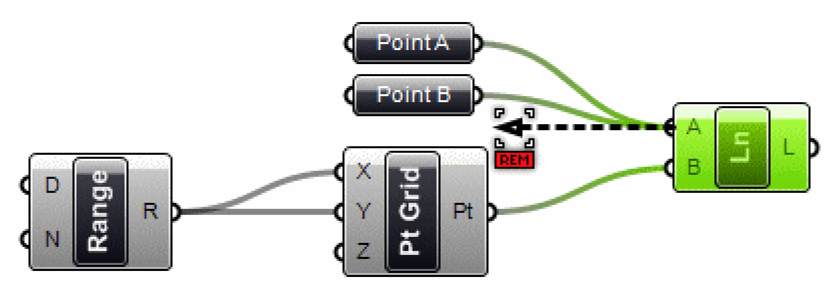
Note that we can make connections both ways. But be careful, by default a new connection will erase existing connections. Since we assumed that you will most often only use single connections, you have to do something special in order to define multiple sources. If you hold Shift while dragging connection wires, the mouse pointer will change to indicate addition behavior:

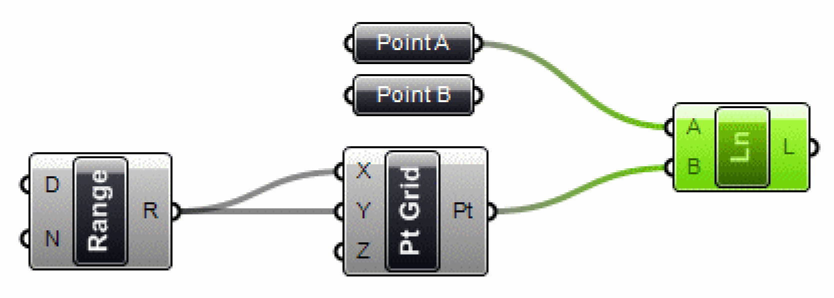




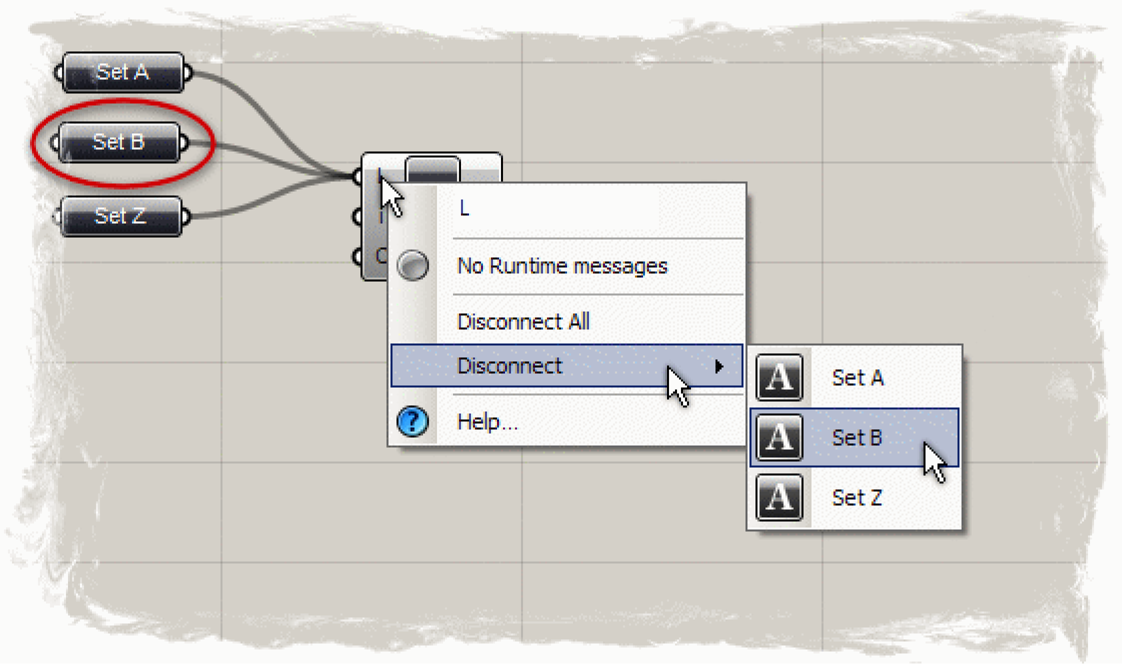
If the "ADD" cursor is active when you release the mouse button over a source parameter, that parameter will be added to the source list. If you specify a source parameter which is already defined as a source, nothing will happen. You cannot inherit from the same source more than once.

By the same token, if you hold down Control the "REM" cursor will become visible, and the targeted source will be removed from the source list. If the target isn't referenced, nothing will happen.

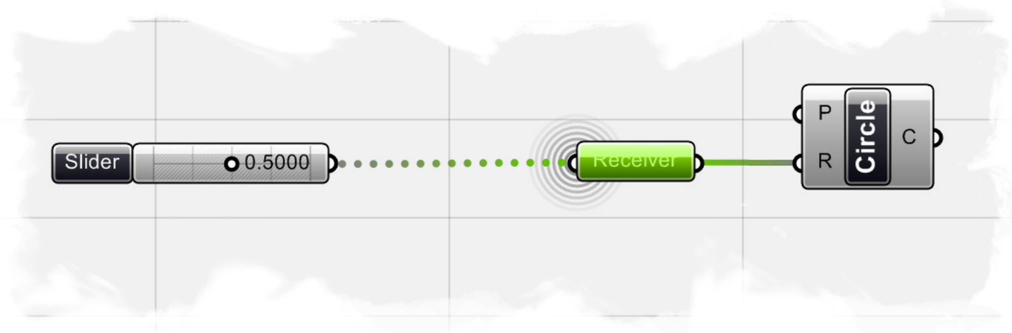




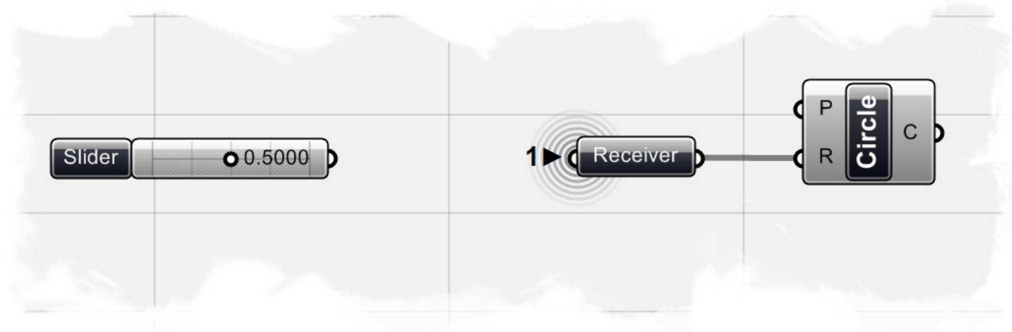
You can also disconnect (but not connect) sources through the parameter menu:



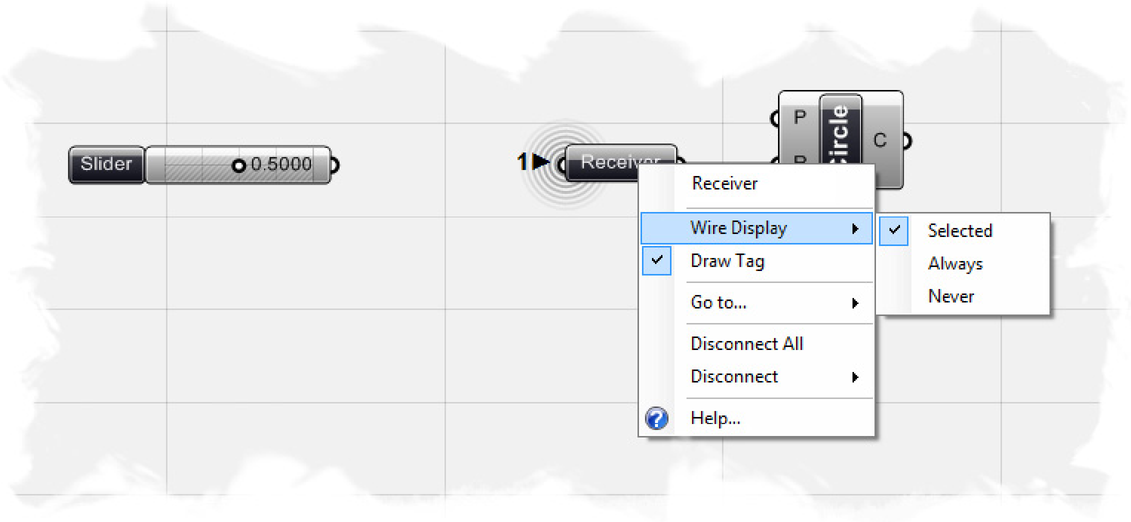
Grasshopper also has the capability to transfer information wirelessly through the use of a receiver, found under the Special subcategory of the Params tab. You can make connections to the receiver, just as you would with any other component. However, as soon as you release the left mouse button from the connection, the wire automatically disappears. This happens because the receivers default setting is set to only shows its dashed connection wires when the receiver is selected. You can right-click on the receiver and set the wire connections to show only when the receiver is "selected", or to "always" or "never" show the connection wires. You can connect the output of the receiver to as many other components as needed.



Here, the dashed connection wire is shown because the receiver component is selected



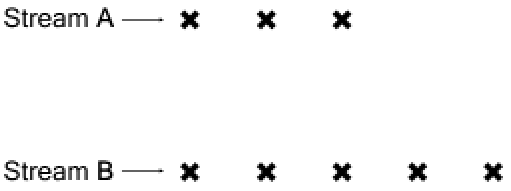
The number 1 before the input of the receiver component indicates that there is one connection being fed into its input. However, since the receiver component is not selected, the connection wire no longer appears (but the information is still transferred).



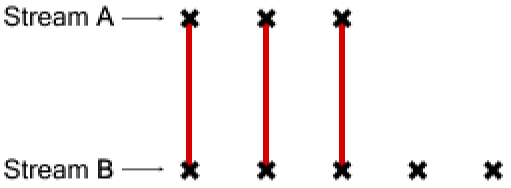
6 Data Stream Matching*

Data matching

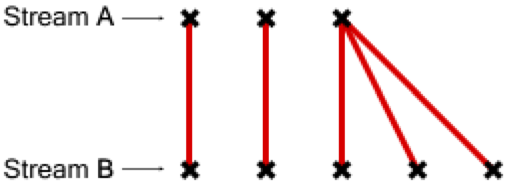
Data matching is a problem without a clean solution. It occurs when a component has access to differently sized inputs. Imagine a component which creates line segments between points. It will have two input parameters which both supply point coordinates (Stream A and Stream B). It is irrelevant where these parameters collect their data from, a component cannot "see" beyond its in- and output parameters:



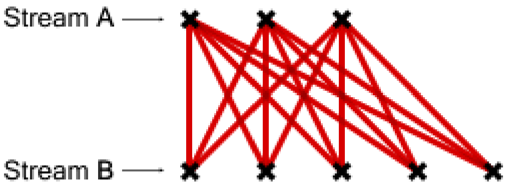
As you can see there are different ways in which we can draw lines between these sets of points. The Grasshopper plug-in currently supports three matching algorithms, but many more are possible. The simplest way is to connect the inputs one-on-one until one of the streams runs dry. This is called the "Shortest List" algorithm:



The "Longest List" algorithm keeps connecting inputs until all streams run dry. This is the default behavior for components:



Finally, the "Cross Reference" method makes all possible connections:

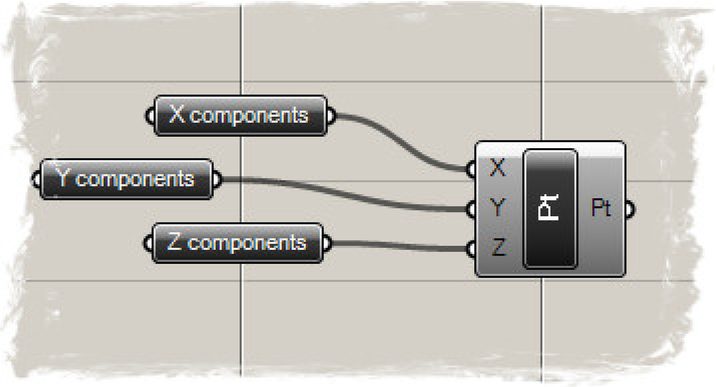


This is potentially dangerous since the amount of output can be humongous. The problem becomes more intricate as more input parameters are involved and when the volatile data inheritance starts to multiply data, but the logic remains the same.

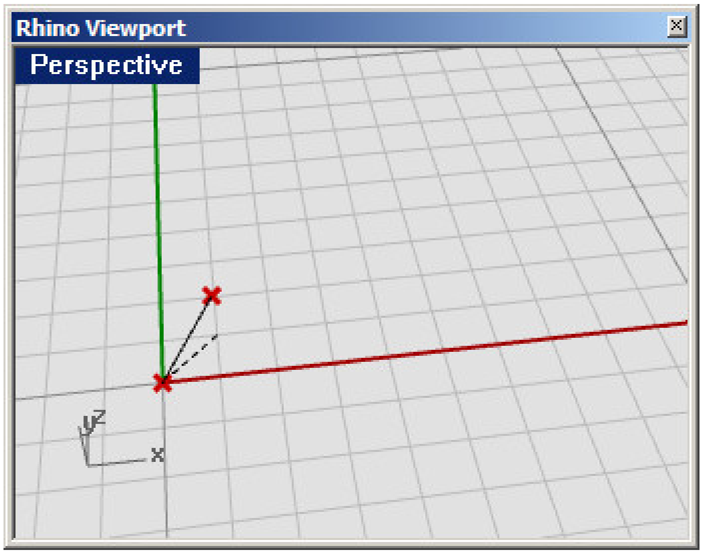
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryDataStreamMatchingAlgorithms.html

Imagine we have a point component which inherits its x, y and z values from remote parameters which contain the following data:

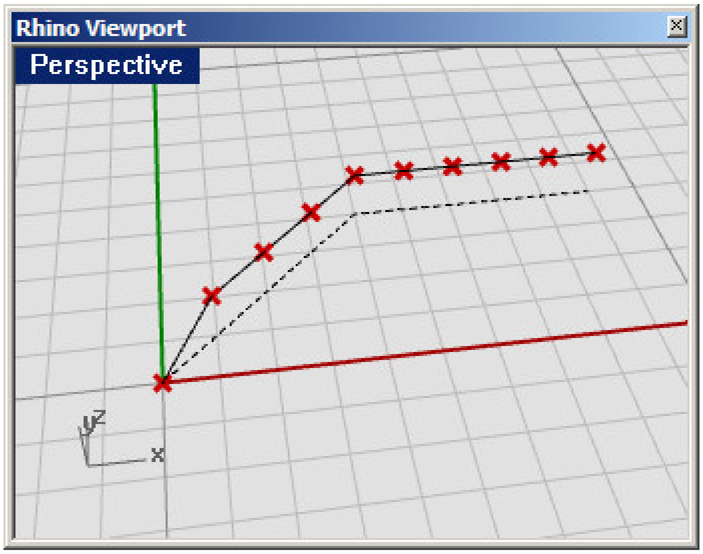
X coordinate: {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0}
Y coordinate: {0.0, 1.0, 2.0, 3.0, 4.0}
Z coordinate: {0.0, 1.0}



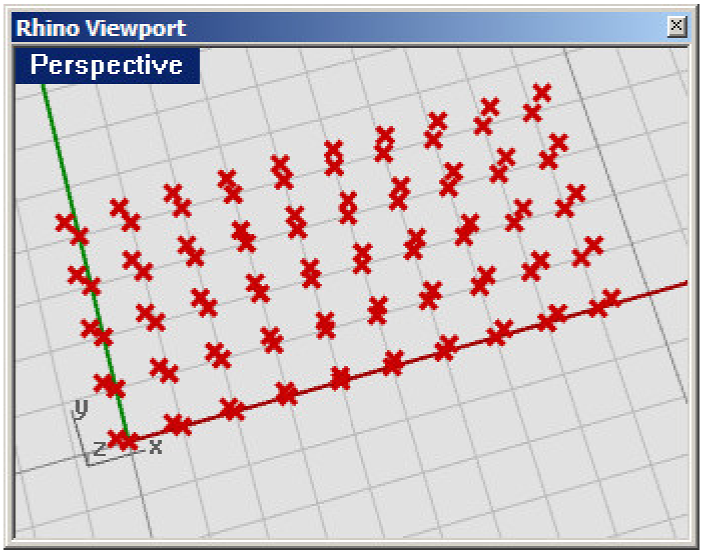
If we combine this data in the "Shortest List" fashion, we get only two points since the "Z coordinate" contains only two values. Since this is the shortest list it defines the extent of the solution:



The "Longest List" algorithm will create ten points, recycling the highest possible values of the Y and Z streams:



"Cross Reference" will connect all values in X with all values in Y and Z, thus resulting in 10×5×2 = a hundred points:



Every component can be set to obey one of these rules (the setting is available in the menu by right clicking the component icon).

Note the one big exception to this behavior. Some components EXPECT to get a list of data in one or more of their input fields. The polyline component, for example, creates a polyline curve through an array of input points. More points in the input parameter will result in a longer polyline, not in more polylines. Input parameters which are expected to yield more than one value are called List Parameters and they are ignored during data matching.

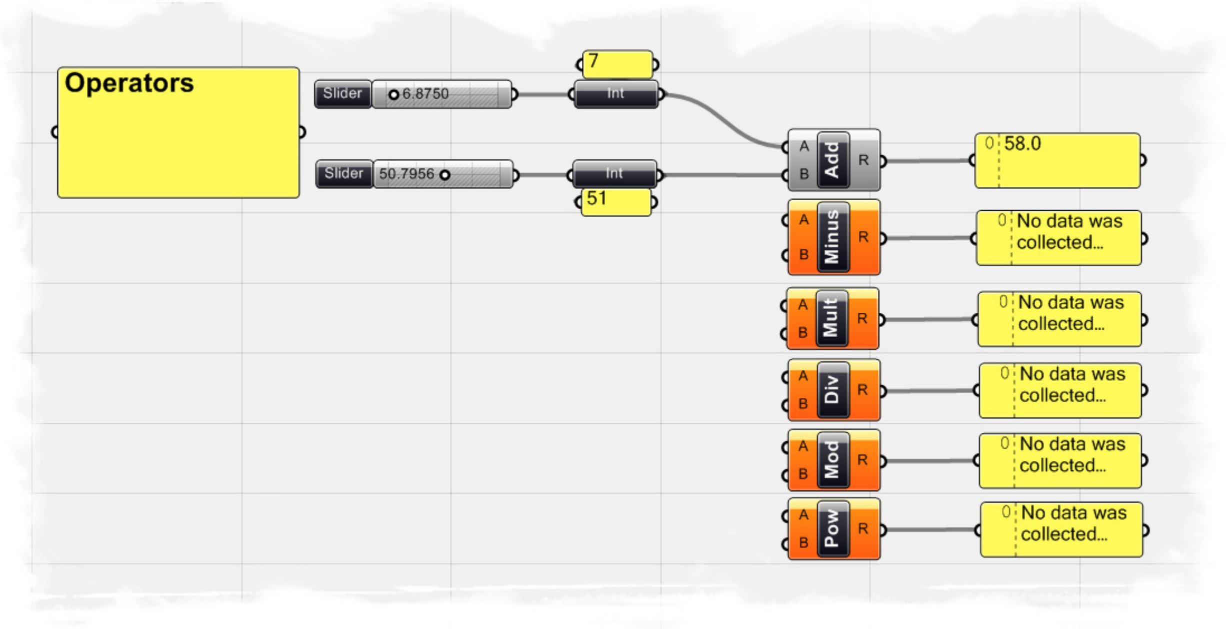
7 *Scalar Component Types*

Scalar Component Types are typically used for various mathematical operations and consist of:
A) **Constants**. Returns a constant value such as Pi, Golden Ratio, etc...
C) **Intervals**. Used to divide numeric extremes (or domains) into interval parts. There are many components under the Intervals tab that allow you to create or decompose a number of different interval types.
D) **Operators**. Used in mathematical operations such as Add, Subtract, Multiply, etc…
E) **Polynomials**. Used to raise a numeric value by some power.
F) **Trigonometry**. Returns typical trigonometric values such as Sine, Cosine, and Tangent, etc…
G) **Utility** (Analysis). Used to evaluate of two or more numerical values.

7.1 Operators

As was previously mentioned, Operators are a set of components that use algebraic functions with two numeric input values, which result in one output value. To further understand Operators, we will create a simple math definition to explore the different Operator Component Types.

Note: To see the finished version of this definition, **Open** the file **Scalar_operators.ghx** found in the Source Files folder that accompanies this document. Below is a screen shot of the completed definition.



To create the definition from scratch:

- Params/Special/Numeric Slider – Drag and drop a numeric slider component to the canvas
- Double click the slider to set:
    - Lower limit: 0.0

    - Upper limit: 100.0
    - Value: 50.0 (note: this value is arbitrary and can be modified to any value within the upper and lower limits)
- Select the slider and type Ctrl+C (copy) and Ctrl+V (paste) to create a duplicate slider
- Params/Primitive/Integer – Drag and drop two Integer components onto the canvas
- Connect slider 1 to the first Integer component
- Connect slider 2 to the second Integer component

    The slider's default value type is set to Floating Point (which results in a decimal numeric value). By connecting the slider to the Integer component, we can convert the floating point value to an Integer, or any whole number. When we connect a Post-It panel (Params/Special/Panel) to the output value of each Integer component, we can see the conversion in real-time. Move the slider to the left and right and notice the floating point value be converted to a whole number. Of course, we could have simplified this step by just setting the slider type to Integers.

- Scalar/Operators/Add – Drag and drop an Add component to the canvas
- Connect the first Integer component to the Add-A input
- Connect the second Integer component to the Add-B input
- Params/Special/Panel – Drag and drop a Post-it panel to the canvas
- Connect the Add-R output to the Post-it panel input

    You can now see the summation value of the two integers in the Post-it panel.

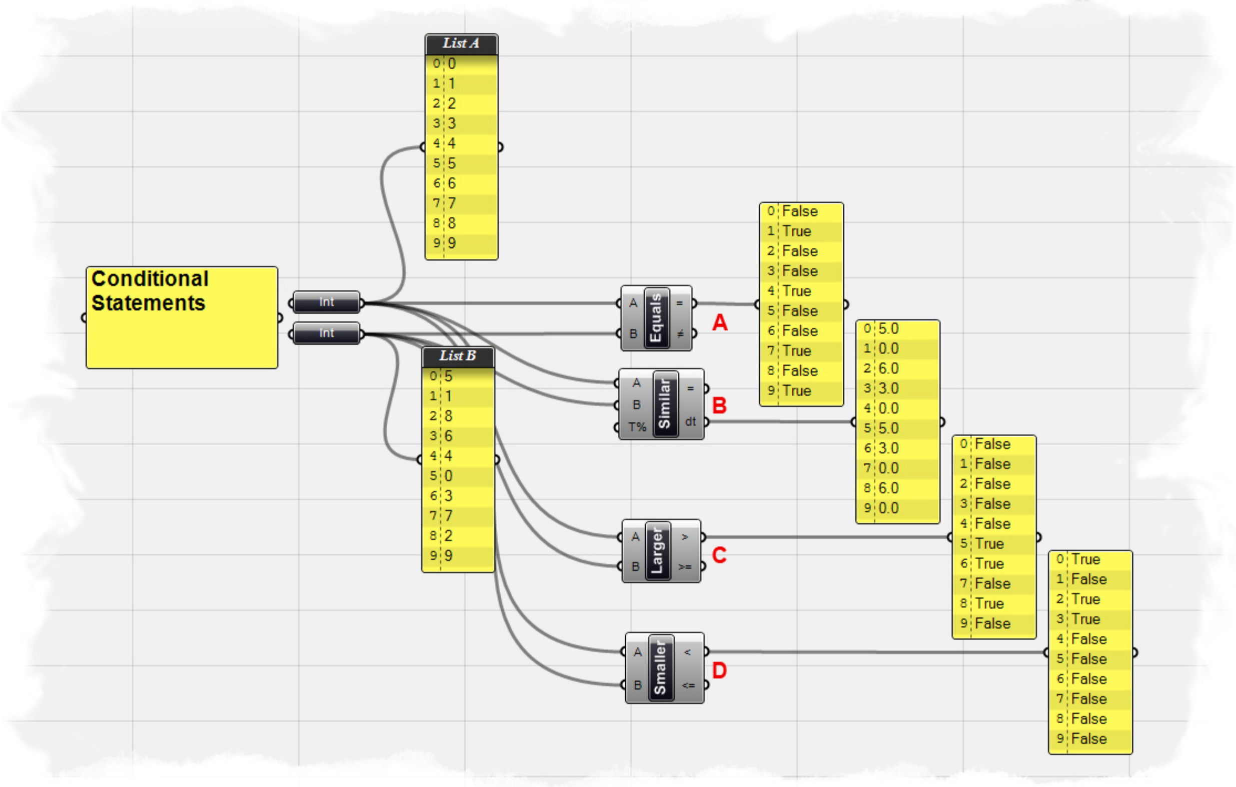
- Drag and drop the other remaining Scalar Operators onto the Canvas:
    - Subtraction
    - Multiplication
    - Division
    - Modulus
    - Power
- Connect the first Integer component to each of the Operator's-A input value
- Connect the second Integer component to each of the Operator's-B input value
- Drag and drop a five more Post-it panels onto the canvas and connect one panel to each Operator's output value

The definition is complete and now when you change each of the slider's values you will see the result of each Operator's action in the Post-it panel area.

7.2 Conditional Statements

You probably noticed that there were a few components under the Operators subcategory of the Scalar tab that we didn't cover in the last section. That's because there are 4 components (new to version 0.6.0007) that act somewhat differently than the mathematical operators, in that, they compare two lists of data instead of performing a algebraic expression. The four components are **Equality**, **Similarity**, **Larger Than**, and **Smaller Than** and are explained in further detail below.

Note: To see the finished version of this definition, **Open** the file **Conditional Statements.ghx** found in the Source Files folder that accompanies this document. Below is a screen shot of the completed definition.



```
List A
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
```

```
List B
0 5
1 1
2 8
3 6
4 4
5 0
6 3
7 7
8 2
9 9
```

```
0 False
1 True
2 False
3 False
4 True
5 False
6 False
7 True
8 False
9 True
```

```
0 5.0
1 0.0
2 6.0
3 3.0
4 0.0
5 5.0
6 3.0
7 0.0
8 6.0
9 0.0
```

```
0 False
1 False
2 False
3 False
4 False
5 True
6 True
7 False
8 True
9 False
```

```
0 True
1 False
2 True
3 True
4 False
5 False
6 False
7 False
8 False
9 False
```

A) Equality component takes two lists and compares the first item of List A and compares it to the first item of List B. If the two values are the same, then a True boolean value is created; conversely if the two values are not equal, then a False boolean value is created. The component cycles through the lists according to the set data matching algorithm (default is set to Longest List). There are two outputs for this component. The first returns a list of boolean values that shows which of the values in the list were equal to one another. The second output returns a list that shows which values were not equal to one another - or a list that is inverted from the first output.

B) Similarity component evaluates two lists of data and tests for similarity between two numbers. It is almost identical to the way the Equality component compares the two lists, with one exception... in that it has a percentage input that defines the ratio of list A that list B is allowed to deviate before inequality is assumed. The Similarity component also has an output that determines the absolute value distance between the two input lists.

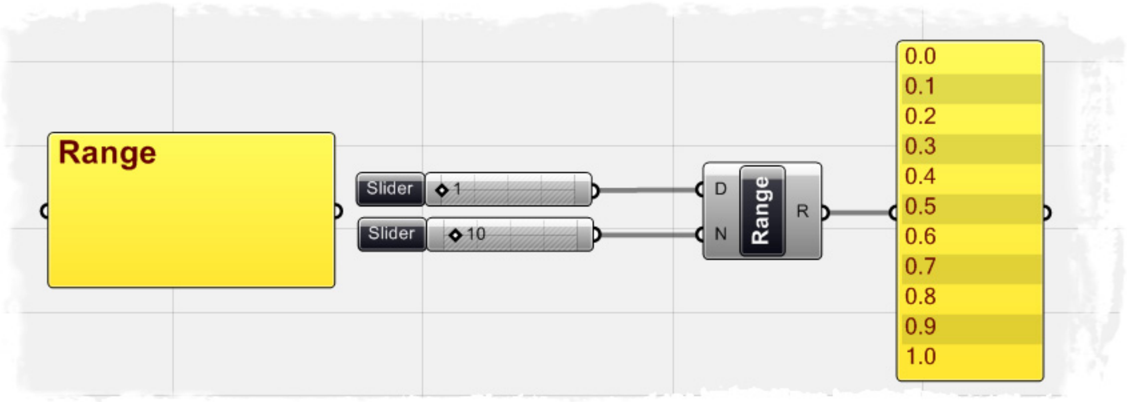
C) Larger Than component will take two lists of data and determine if the first item of List A is greater than the first item of List B. The two outputs allows you determine if you would like to evaluate the two lists according to a greater than (>) or greater than and equal to (>=) condition.

D) Smaller Than component performs the opposite action of the Larger Than component. The Smaller Than component determines if list A is less than list B and returns a list of boolean values. Similarly, the two outputs let you determine if you would like to evaluate each list according to a less than (<) or less than and equal to (<=) condition.

7.2 Range vs. Series vs. Interval

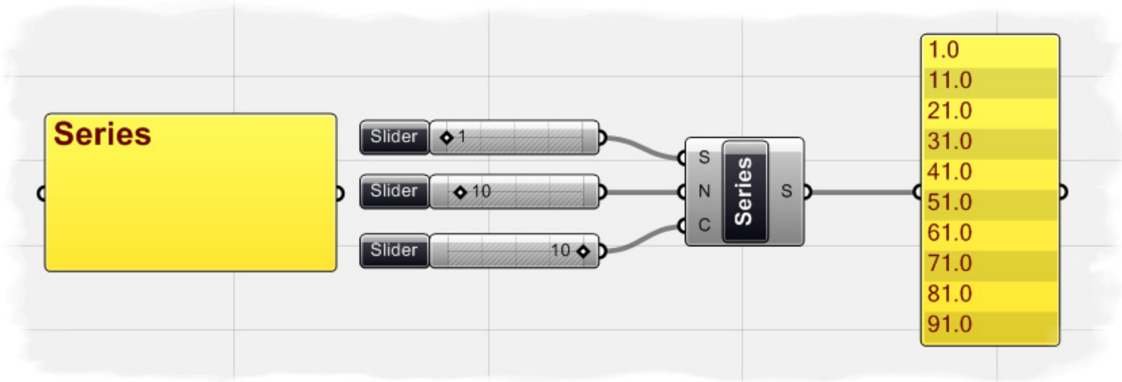
The Range, Series, and Interval components all create a set of values between two numeric extremes; however the components operate in different ways.

Note: To see the finished version of the following examples, **Open** the file **Scalar_intervals.ghx** found in the Source Files folder that accompanies this document.



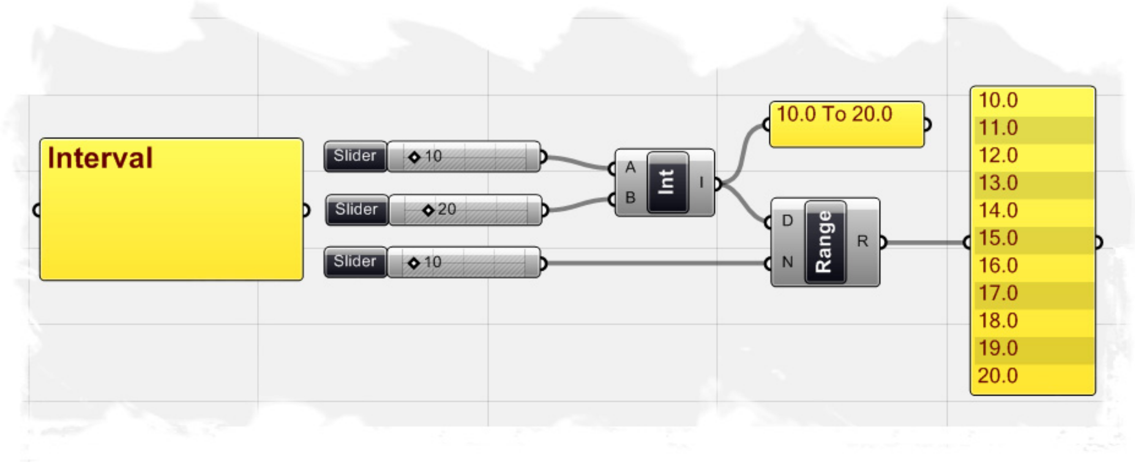
```
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
```

The Range component creates a list of evenly spaced numbers between a low and a high value called the domain of numeric range. In the example above, two numeric sliders are connected to the input values of the Range component. The first slider defines the numeric domain for the range of values. In this example, the domain has been defined from zero to one, since the slider is set to 1. The second slider defines the number of steps to divide the domain, which in this case has been set to 10. Thus, the output is a list of 11 numbers evenly divided between 0 and 1. (note: The second slider, set to 10, is defining the number of divisions between 0 and 1, which ultimately creates 11 numbers, not 10)



```
1.0
11.0
21.0
31.0
41.0
51.0
61.0
71.0
81.0
91.0
```

The Series component creates a set of discreet numbers based on a start value, step size, and the number of values in the Series. The series example shows three numeric sliders connected to the Series component. The first slider, when connected to the Series-S input defines the starting point for the series of numbers. The second slider, set to 10, defines the step value for the series. Since, the start value has been set to 1 and the step size has been set to 10, the next value in the series will be 11. Finally, the third slider defines the number of values in the series. Since this value has also been set to 10, the final output values defined in the series shows 10 numbers, that start at 1 and increase by 10 at each step.



The Interval component creates a range of all possible numbers between a low and high number. The Interval component is similar to the numeric domain which we defined for the Range component. The main difference is that the Range component creates a defualt numeric domain between 0 and whatever input value has been defined. In the Interval component, the low and the high value can be defined by the A and B input values. In the example below, we have defined a range of all possible values between 10 and 20, set by the two numeric sliders. The output value for the Interval component now shows 10.0 To 20.0 which reflects our new numeric domain. If we now connect the Interval-I output to a Range-D input, we can create a range of numbers between the Interval values. As was the case in the previous Range example, if we set the number of steps for the Range to 10, we will now see 11 values evenly divided between the lower Interval value of 10.0 and the upper Interval value of 20.0. (Note: There are multiple ways to define an interval, and you can see several other methods listed under the Scalar/Interval tab. We have merely defined a simple Interval component, but we will discuss some of the other interval methods in the coming chapters)

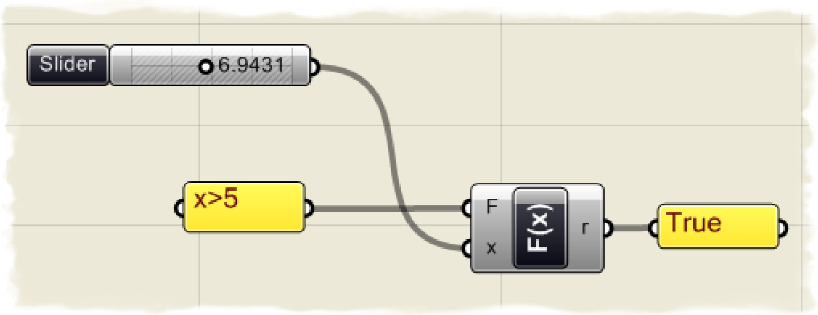
7.3 Functions & Booleans

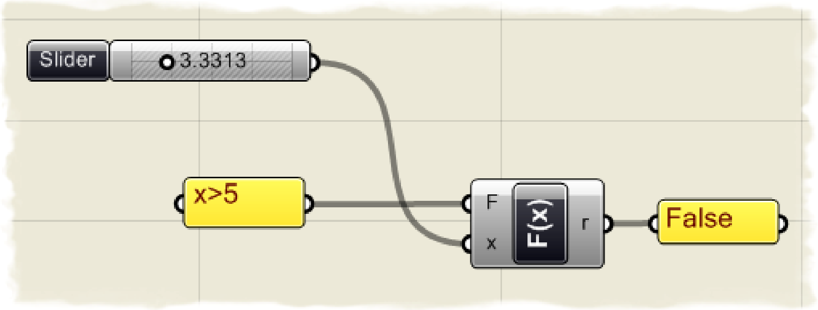
Almost every programming language has a method for evaluating conditional statements. In most cases the programmer creates a piece of code to ask a simple question of “what if?” What if the 9/11 terrorist attacks had never happened? What if gas cost $10/gallon? These are important questions that represent a higher level of abstract thought. Computer programs also have the ability to analyze “what if?” questions, and take actions depending on the answer to the question. Let’s take a look at a very simple conditional statement that a program might interpret:

If the object is a curve, delete it.

The piece of code first looks at an object and determines a single boolean value for whether or not it is a curve. There is no middle ground. The boolean value is **True** if the object is a curve, or **False** if the object is not a curve. The second part of the statement performs an action dependant on the outcome of the conditional statement, in this case, if the object is a curve, then delete it. This conditional statement is called an **If/Else statement**; If the object meets certain criteria, do something; else, do something else.

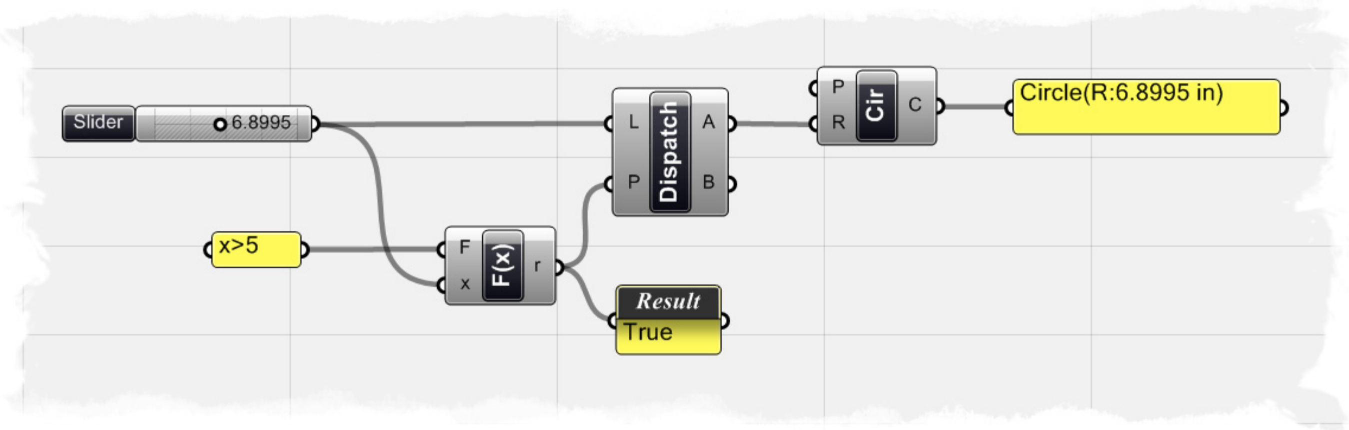
Grasshopper has the same ability to analyze conditional statements through the use of Function components.



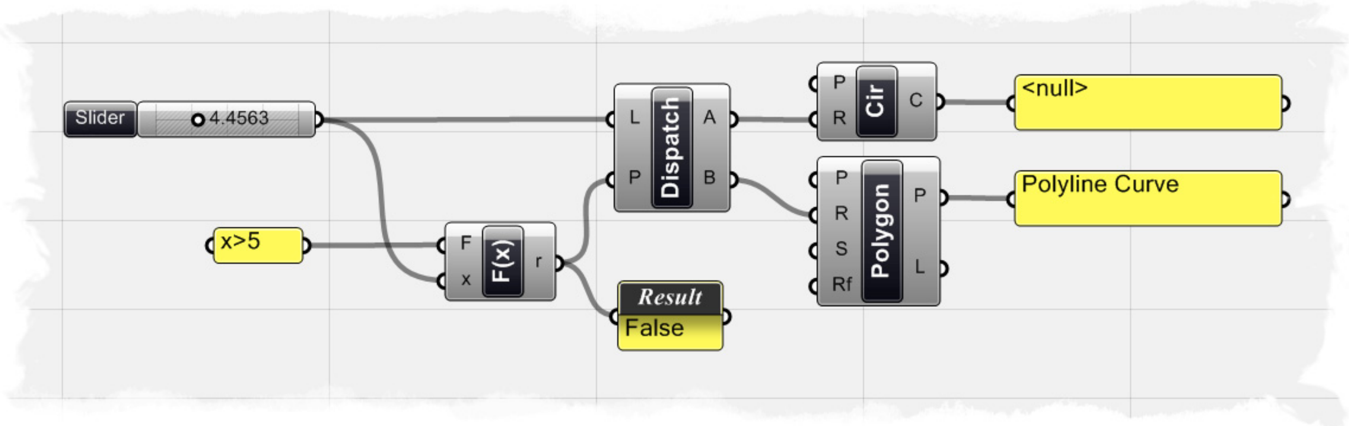


In the example above, we have connected a numeric slider to the x-input of a single variable Function component (Logic/Script/F1). Additionally, a conditional statement has been linked to the F-input of the Function, defining the question, “Is x greater than 5?” If the numeric slider is set above 5, then the r-output for the Function shows a True boolean value. If the numeric slider drops below 5, then the r-output changes to a False value.

Once we have determined the boolean value of the function, we can feed the True/False pattern information into a Dispatch component (Logic/List/Dispatch) to perform a certain action. The Dispatch component works by taking a list of information (in our example we have connected the numeric slider information to the Dispatch-L input) and filters the information based on the boolean pattern result of the single variable function. If the pattern shows a True value, the list information will be passed to the Dispatch-A output. If the pattern is False, it passes the list information to the Dispatch-B output. For this example, we have decided to create a circle ONLY if the slider value is greater than 5. We have connected a Circle component (Curve/Primitive/Circle) to the Dispatch-A output, so that a circle with a radius specified by the numeric slider will be created only if the boolean value passed into the Dispatch component is True. Since no component has been linked to the Dispatch-B output; if the boolean value is False, then nothing happens and a circle will not be created.



We can further this definition a little bit, by connecting a N-sided Polygon (Curve/Primitive/Polygon) curve to the Dispatch B output, making sure to connect the wire to the Polygon R-input to define the polygon radius. Now, if the numeric slider falls below 5, then a five sided polygon with a radius defined by the number slider will be created at the origin point. If we take the slider value above 5, then a circle will be created. By this method, we can begin to create as many **If/Else** statements as needed to feed information throughout our definition.



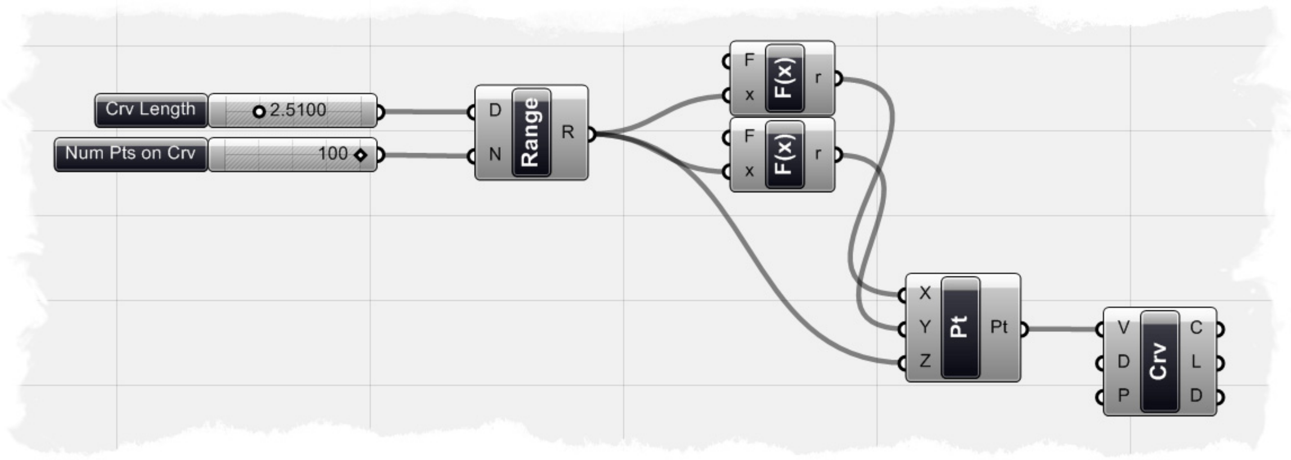
Note: To see the finished version of the circle boolean test example, **Open** the file **If_Else test.ghx** found in the Source Files folder that accompanies this document.

7.4 Functions & Numeric Data

The Function component is very flexible; that is to say that it can be used in a variety of different applications. We have already discussed how we can use a Function component to evaluate a conditional statement and deliver a boolean value output. However, we can also use Function components to solve complex mathematical algorithms and display the numeric data as the output.

In the following example, we will create a mathematical spiral similar to the example David Rutten provided in his *Rhinoscript 101* manual. For more information about Rhinoscript or to download a copy of the manual, visit http://en.wiki.mcneel.com/default.aspx/McNeel/RhinoScript101.html

Note: To see the finished version of the mathematical spiral example, **Open** the file **Function_spiral.ghx** found in the Source Files folder that accompanies this document. Below is a screen shot of the completed definition.

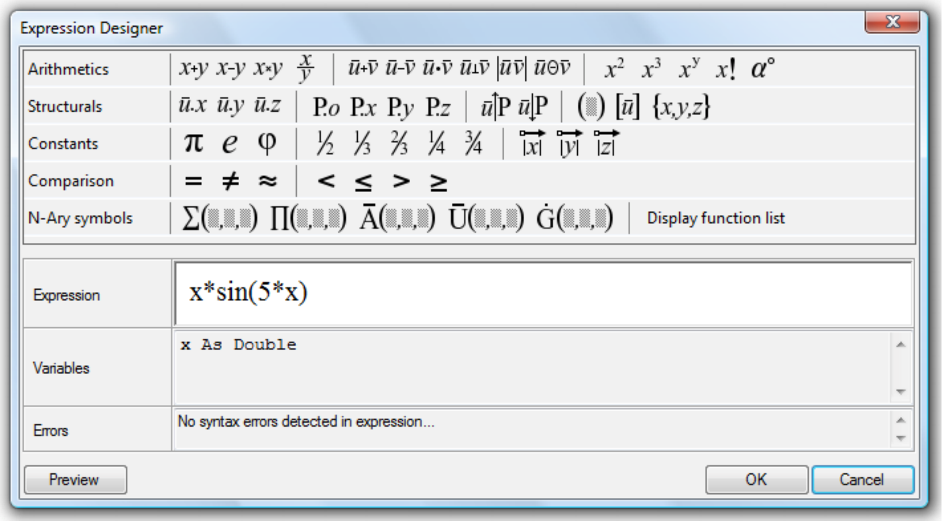


To create the definition from scratch:

- Logic/Sets/Range - Drag and drop a Range component onto the canvas
- Params/Special/Slider - Drag and drop two numeric sliders onto the canvas
- Right-click the first slider and set the following:
  - Name: Crv Length
  - Slider Type: Floating Point (this is set by defualt)
  - Lower Limit: 0.1
  - Upper Limit: 10.0
  - Value: 2.5
- Now, right-click on the second slider and set the following:
  - Name: Num Pts on Crv
  - Slider Type: Integers
  - Lower Limit: 1.0
  - Upper Limit: 100.0
  - Value: 100.0
- Connect the Crv Length slider to the Range-D input
- Connect the Num Pts on Crv slider to the Range-N input

  We have just created a range of 101 numbers that are evenly spaced that range from 0.0 to 2.5, which we can feed into our Function components.

- Logic/Script/F1 - Drag and drop a single variable Function component onto the canvas
- Right-click the F-input of the Function component and open the Expression Editor.
- In the Expression Editor dialogue box, type the following equation:
  - **x*sin(5*x)**
    If you have entered the algorithm into the editor correctly, you should see a statement that says, "No syntax errors detected in expression" under the errors rollout.
  - Click OK to accept the algorithm



Expression: x*sin(5*x)

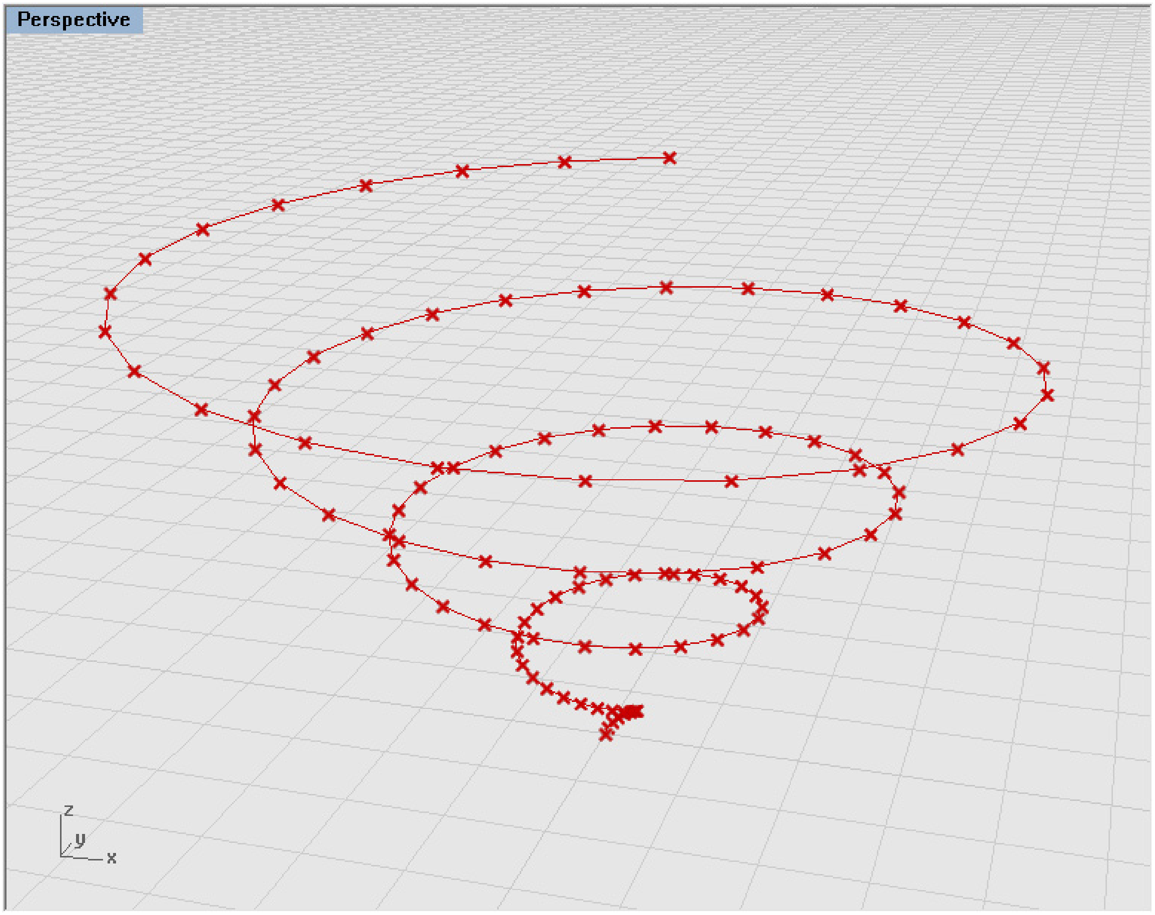
Variables: x As Double

Errors: No syntax errors detected in expression...

- Select the Function component and hit Ctrl+C (copy) and Ctrl+V (paste) to create a duplicate of this Function
- Right-click the F-input of the duplicated Function component and open the Expression Editor
- In the Expression Editor dialogue box, type the following equation:
  - **x*cos(5*x)**
    ***Note:** the only difference in this equation is that we have replaced the sine function with the cosine function.*
  - Click OK to accept the algorithm
- Connect the Range-R output to the x-input to both Function components
  We have now fed the 101 numbers created by the Range component into the Function component, which solves a mathematical algorithm and outputs a new list of numeric data. You can hover your mouse over the r-output of each function to see the result of each equation.
- Vector/Point/Point XYZ - Drag and drop a Point XYZ component onto the canvas
- Connect the first Function-r output to the X-input of the Point component
- Connect the second Function-r output to the Y-input of the Point component
- Connect the Range-R output to the Z-input of the Point component
  Now, if you look at the Rhino viewport, you will see a set of points forming a spiral. You can change the two numeric sliders at the beginning of the definition to change the number of points on the spiral or the length of the spiral.

- Curve/Spline/Curve - Drag and drop a Curve component onto the canvas
- Connect the Point-Pt output to the Curve-V input

  We have created a single curve that passes through each point of the spiral. We can right-click on the Curve-D input to set the Curve degree; a 1 degree curve will create straight line segments between each point and will insure that the curve actually passes through each point. A 3 degree curve will create a smooth Bezier curve where the points of the spiral will act as control points for the curve, however the line will not actually pass through each point.



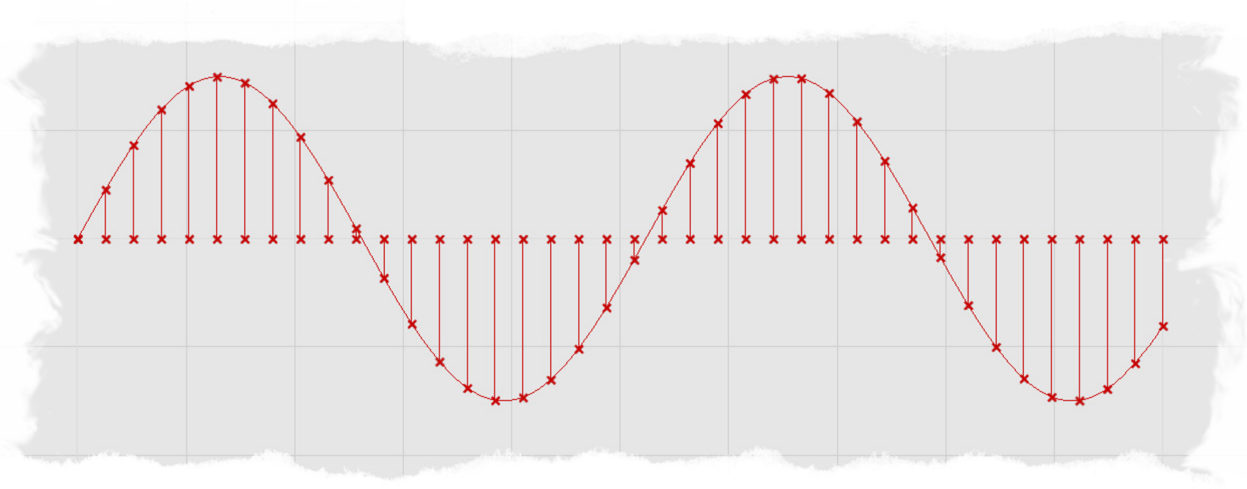
Note: To see a video tutorial of this example, please visit Zach Downey's blog at: http://www.designalyze.com/2008/07/07/generating-a-spiral-in-rhinos-grasshopper-plugin/

7.5 Trigonometric Curves

We have already shown that we can use Function components to evaluate complex formulas to create spirals and other mathematical curves, however Grasshopper also has a set of trigonometric components built into the scalar component family. Trigonometric functions, like sine, cosine, and tangent are important tools for mathematicians, scientists, and engineers because they define a ratio between two sides of a right triangle containing a specific angle, called Theta. These functions are important in Vector analysis, which we will cover in later chapters. However, we can also use these functions to define periodic phenomena, like sinusoidal wave functions often found in nature in the form of ocean waves, sound waves, and light waves. In 1822, Joseph Fourier, a French mathematician, discovered that sinusoidal waves can be used as simple building blocks to 'make up' and describe nearly any periodic waveform. The process is called Fourier analysis.

In the following example, we will create a sinusoidal wave form where the number of points on the curve, the wavelength, frequency, and amplitude can be controlled by a set of numeric sliders.

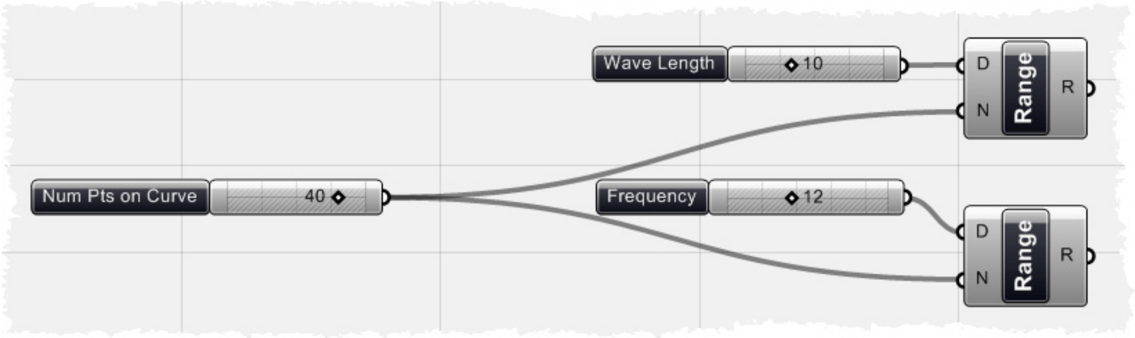
Note: To see the finished definition used to create the sine curve below, **Open** the file **Trigonometric_curves.ghx** found in the Souce Files folder that accompanies this document.



To create the definition from scratch:

- Params/Special/Slider - Drag and drop 3 numeric sliders onto the canvas
- Select the first slider and set the following parameters:
    - Name: Num Pts on Curve
    - Slider Type: Integers
    - Lower Limit: 1
    - Upper Limit: 50
    - Value: 40
- Select the second slider and set the following parameters:
    - Name: Wave Length
    - Slider Type: Integers
    - Lower Limit: 0
    - Upper Limit: 30
    - Value: 10

- Select the third slider and set the following parameters:
  - Name: Frequency
  - Slider Type: Integers
  - Lower Limit: 0
  - Upper Limit: 30
  - Value: 12
- Logic/Sets/Range - Drag and drop 2 Range components onto the canvas
- Connect the Wave Length slider to the first Range-D input
- Connect the Frequency slider to the second Range-D input
- Connect the Num Pts on Curve slider to both Range-N inputs

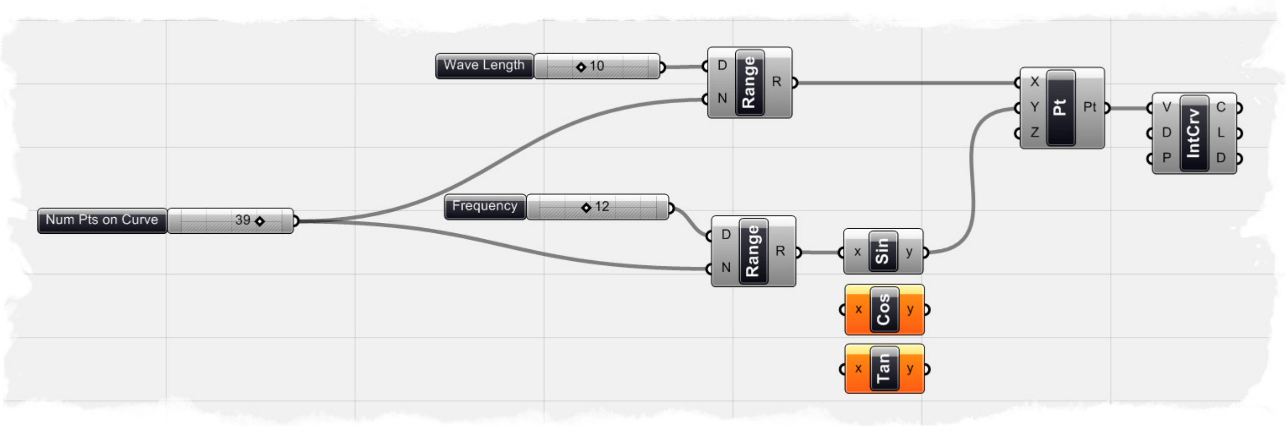


Your definition should look like the image above. We have now created two lists of data; the first is a range of evenly divided numbers from 0 to 10, and the second is a list of evenly divided numbers ranging from 0 to 12.

- Scalar/Trigonometry/Sine - Drag and drop a Sine component onto the canvas
- Connect the second Range-R output to the x-input of the Sine component
- Vector/Point/Point XYZ - Drag and drop a Point XYZ component onto the canvas
- Connect the first Range-R output to the X-input of the Point XYZ component
- Connect the y-output of the Sine component to the Y-input of the Point XYZ component

  If you look at the Rhino viewport, you should see a series of points in the shape of a sine wave. Since the first Range output is being fed directly into the X-input of the Point component, without first being fed into a trigonometric function component, our x value of the points are constant and evenly spaced. However, our sine component is being fed into the Y-input of the Point component, so we see a change in the y value of our points; which ultimately form a wave pattern. You can now change any of the numeric sliders to change the shape of the wave pattern.
- Curve/Spline/Interpolate - Drag and drop an Interpolated Curve component onto the canvas
- Connect the Point-Pt output to the Curve-V input

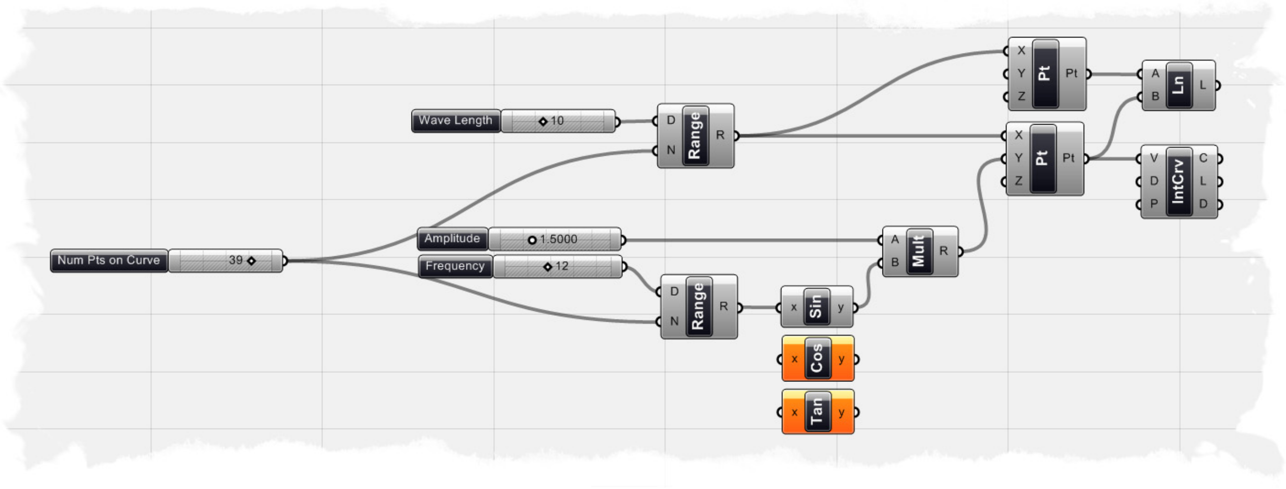
At this point, your definition should look like the screen shot below. We have set up our definition to use the Num Pts on Curve, Wave Length, and Frequency sliders, but we can set up one last slider to control the amplitude of the sine curve.



- Params/Special/Slider - Drag and drop another number slider onto the canvas
- Right-click on this new slider and set the following:
    - Name: Amplitude
    - Slider Type: Floating Point
    - Lower Limit: 0.1
    - Upper Limit: 5.0
    - Value: 2.0
- Scalar/Operators/Multiplication - Drag and drop a Multiplication component onto the canvas
- Connect the Amplitude slider to the Multiplication-A input
- Connect the y-output of the Sin component to the Multiplication-B input
- Connect the Multiplication-r output to the Y-input of the Point XYZ component

    This connection should replace your existing connection, which had been previously connected to the Sine component output. The Amplitude slider is just multiplying the Sine values by a scale factor. Since the slider is driving the Y-value of our Sine curve, when you increase the Amplitude value you will in turn be increasing the amplitude of the curve.
- Vector/Point/Point XYZ - Drag and drop another Point XYZ component onto the canvas
- Connect the first Range-R output to the X-input of the new Point XYZ component
- Curve/Primitive/Line - Drag and drop a Line component onto the canvas
- Connect the first Point-Pt output to the Line-B input
- Connect the second Point-Pt output to the Line-A input

    In t*he last part of definition, we have created a second set of evenly spaced points along the X-axis, which correspond to the same x-coordinates of the Sine curve. The Line component creates a line segment between the first list of points, the ones that create the sine curve, and the second list of points which define the X-axis. The new lines give you a visual reference of the vertical displacement in the wave form pattern. Your definition should look like the image below.*

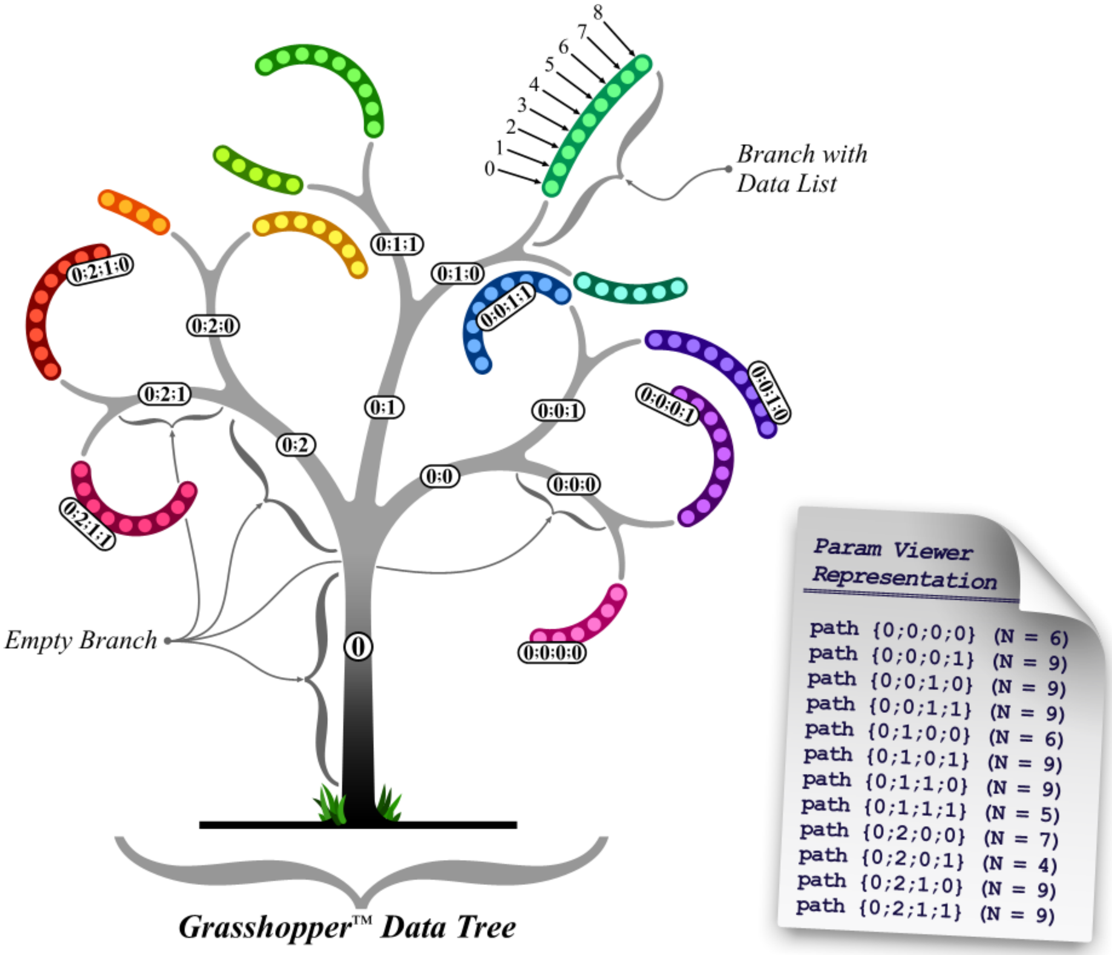


We have shown how to create a sine wave curve, however you can generate other sinusoidal curves, like a Cosine or Tangent wave forms, by replacing the Sine component with a Cosine or Tangent component found under the Scalar/Trigonometry tab.

Note: To see a video tutorial of this example, please visit David Fano's blog at: http://designreform.net/2008/06/01/rhino-3d-sine-curve-explicit-history/

8 The Garden of Forking Paths

In all versions of Grasshopper released prior to version 0.6.00xx, data inside a parameter was stored in a single list; and as such, did not need a list index. However, there has been a complete overhaul of how data is stored in Grasshopper, and now its possible to have multiple lists of data inside a single parameter. Since multiple lists are available, there needed to be a way to identify each individual list. Below is an image, created by David Rutten, that represents a reasonably complex, yet well structured tree diagram.

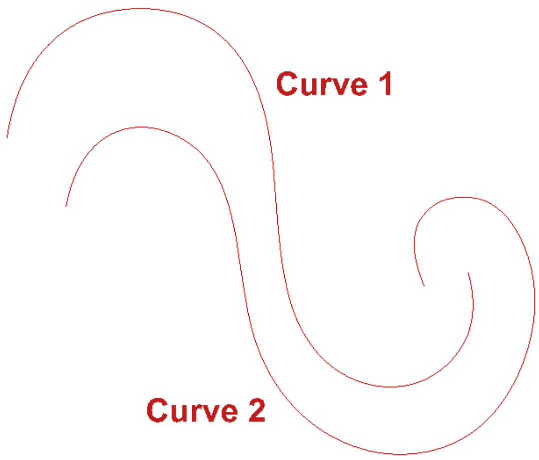


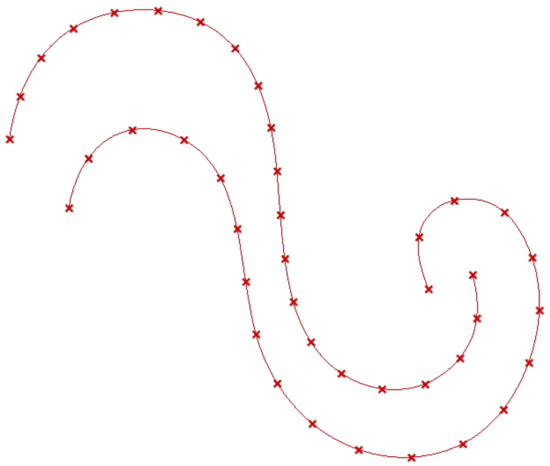
```
Param Viewer
Representation

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)
```

In the image above, there is a single master branch (you could call this a trunk, but since it's possible to have multiple master branches, it might be a bit of a misnomer) at path {0}. This path contains no data, but does have 3 sub-branches. Each of these sub-branches inherit the index of the parent branch {0} and add their own sub-index (0, 1 and 2 respectively). It would be wrong to call this an "index", because that implies just a single number. It is probably better to refer to this as a "path", since it resembles a folder-structure on the disk. Each of these sub-branches are again host to 2 sub-sub-branches and they again contain no data themselves. The same is true for the sub-sub-branches. Once we reach nesting level 4, we finally encounter some data (lists are represented as colorful thick lines, data items are bright circles). Every sub-sub-sub-branch (or level 4 branch) is a terminus branch, meaning they don't subdivide any further.

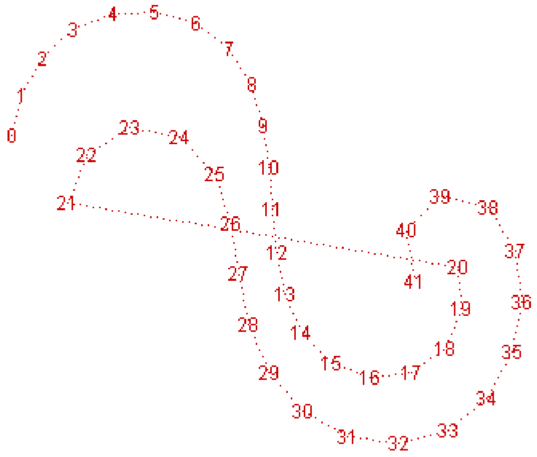
Each data item is thus part of one (and only one) branch in the tree, and each item has an index that specifies its location within the branch. Each branch has a path that specifies its location within the tree.

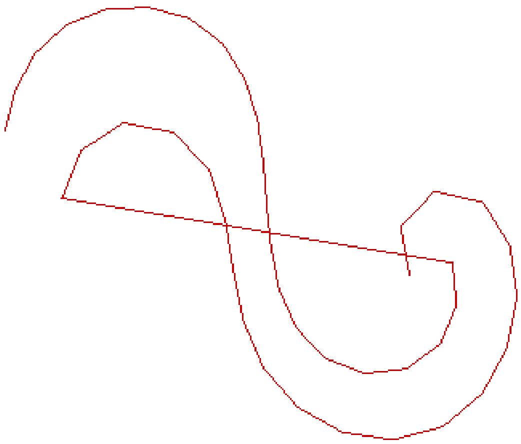
To further examine tree structures, let's take a look at a very simple example. We'll start with two Curves which we reference into Grasshopper. We'll then use a single Divide Curve (Curve/Division/Divide Curve) component to divide both curves into 20 individual segments; ultimately giving us 21 points on each curve (but a list of 42 points). Let's now feed all of those points into a Polyline component (Curve/Spline/Polyline), which will create a new line through the division points.



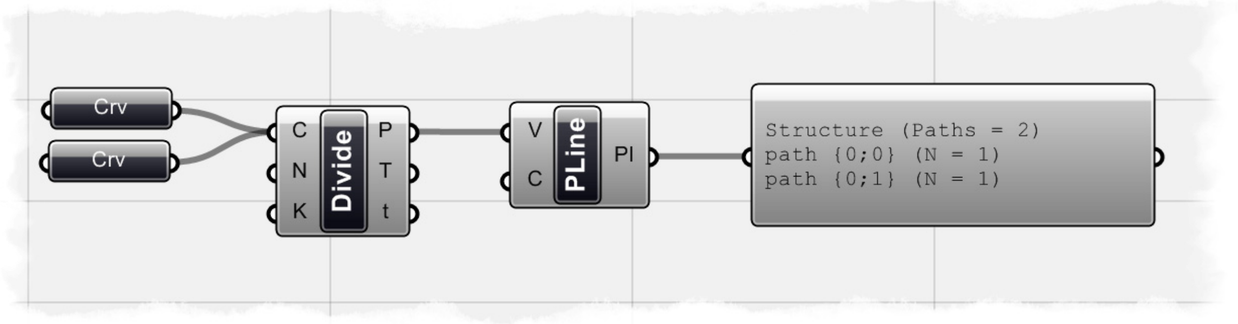


If we had been using Grasshopper version 0.5 or before, the Polyline curve would draw only 1 line through our entire point list (which has 42 points as a result of the Divide Curve component). This is because all of the points are stored in 1 list (with no branches or paths), so the component is programmed to draw a Polyline through all of the points in the list. If we were using Grasshopper version 0.5, the point index and the resulting Polyline would have looked like this:



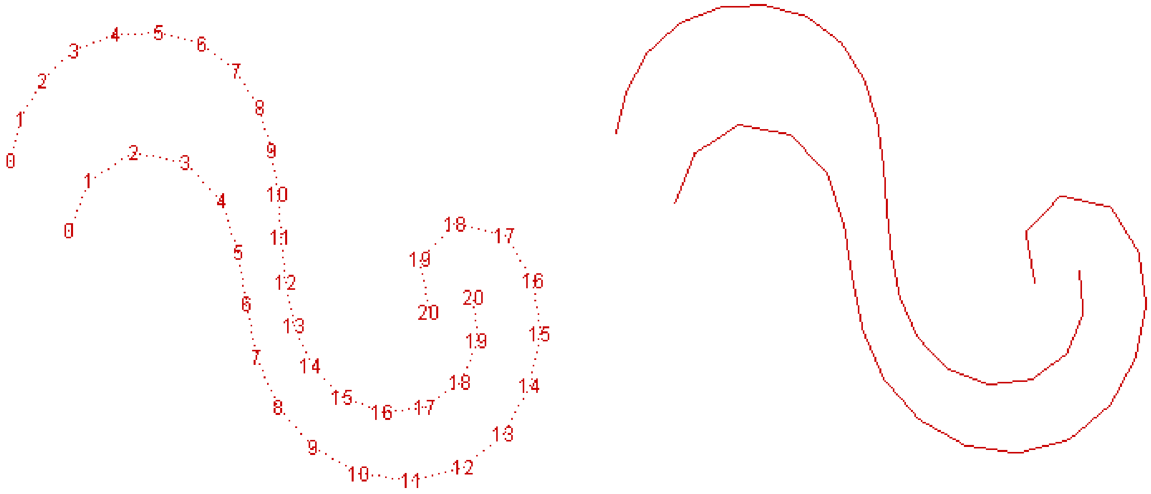


However, since we now know that Grasshopper has the capabilities to incorporate paths and branches, we can now use that index to control how the Polyline component behaves. If we follow the exact same steps as before, only using Grasshopper version 0.6.00xx or higher, our Polyline component will create 2 polylines because it recognizes that there are 2 paths that have each been divided into 20 segments. We can check out the tree structure by using the Parameter Viewer (Params/Special/Param Viewer). Below is a screenshot of this example, which shows that our structure has 2 paths (0;0) and (0;1) which each have 1 resultant curve at their terminus.



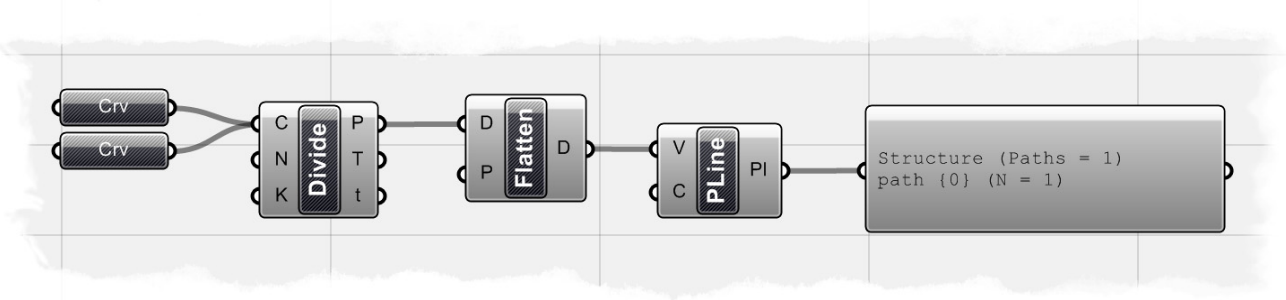
```
Structure (Paths = 2)
path {0;0} (N = 1)
path {0;1} (N = 1)
```

The point index of the 2 polylines and the resultant curves should now look like this:



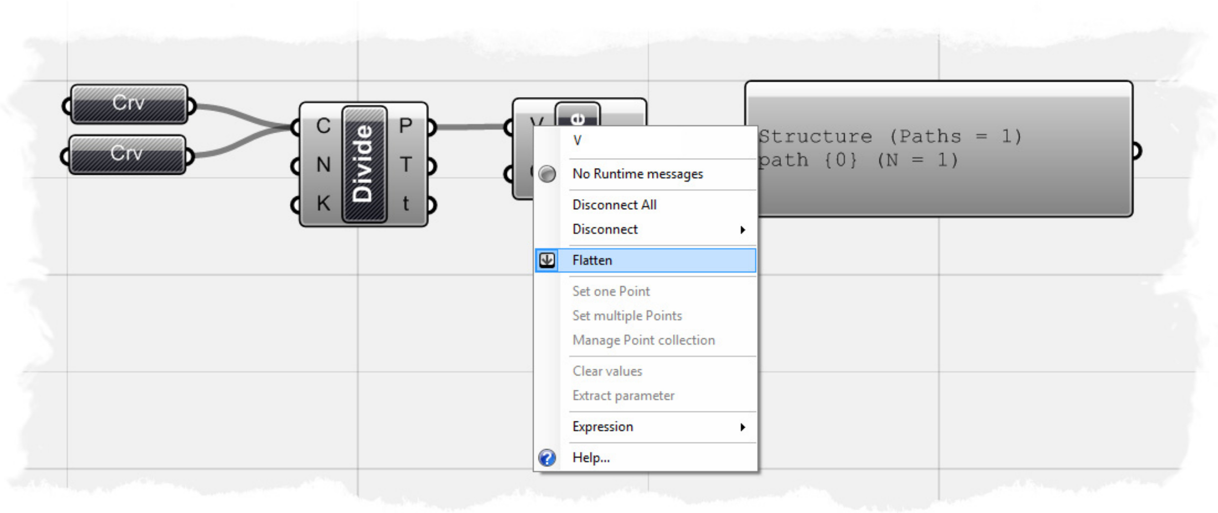
We now know that our definition has 2 paths, but what if we really only wanted 1 path and thus 1 resultant polyline. Grasshopper has a number of new tree structure components, found under the Tree subcategory of the Logics tab, which can help you control your tree structure. We can use a Flatten Tree component (Logic/Tree/Flatten Tree) to collapse an entire tree structure into 1 list, much like the data would have been displayed in version 0.5 or before.

You can see in the image below, that when we flatten our tree, our structure now only has 1 path, with 1 resultant curve stored in the terminus... ultimately giving us the same single polyline curve that we would have gotten if we had used an earlier version of Grasshopper.



```
Structure (Paths = 1)
path {0} (N = 1)
```

You can also simplify this definition by flattening your structure inside the Polyline component, and by-passing the Flatten component. To do this, simply right-click on the V-input of the Polyline component and select the "Flatten" toggle. By doing this, you should notice that your tree structure (which we can view through the Parameter Viewer) changes from 2 paths back to 1.



```
Structure (Paths = 1)
path {0} (N = 1)
```

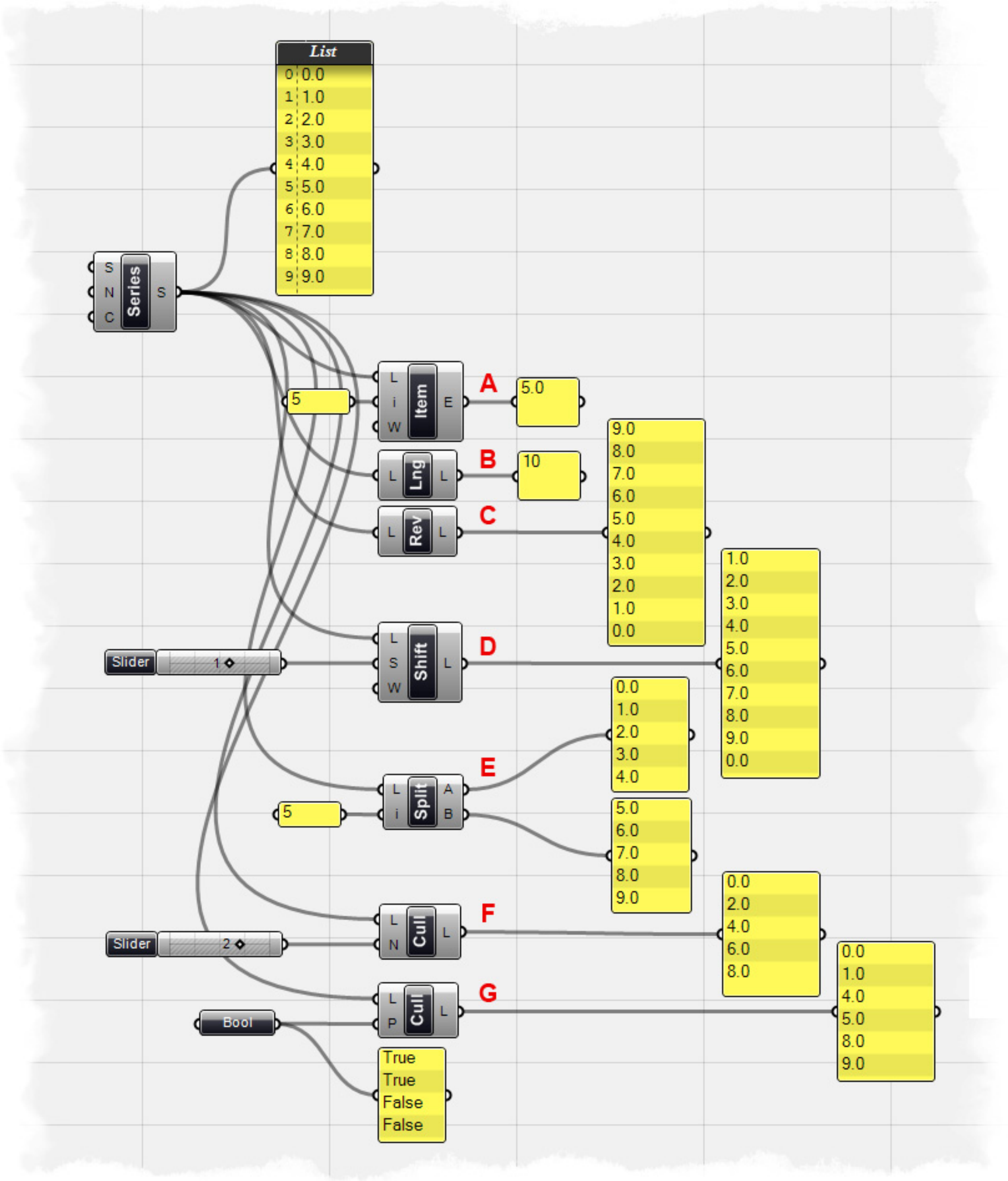
Note: To see the finished definition used to create this example, **Open** the file **Curve Paths.ghx** found in the Source Files folder that accompanies this document. You will also need to open the corresponding Rhino file found in the same folder called **Curve Paths_base file.3dm**.

8.1 Lists & Data Management

It's helpful to think of Grasshopper in terms of *flow*, since the graphical interface is designed to have information flow into and out of specific types of components. However, it is the DATA (such as lists of points, curves, surfaces, strings, booleans, and numbers, etc.) that define the information flowing in and out of the components; such that understanding how to manipulate list data is critical to understanding the Grasshopper plug-in.

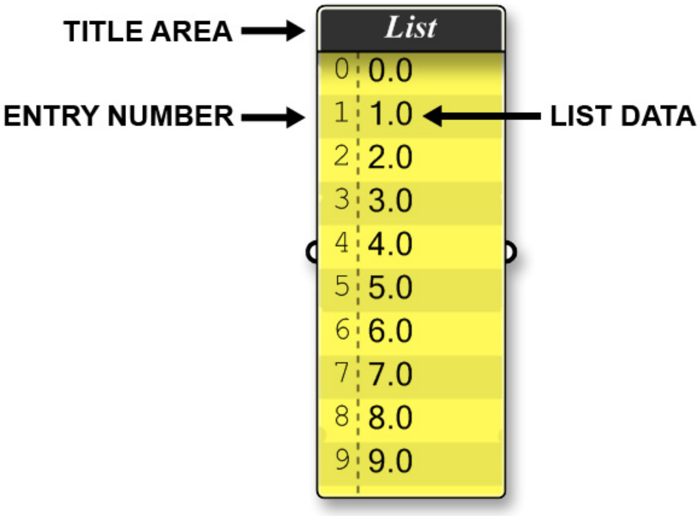
Below is an example of how you can control a list of numeric data using a variety of list components found under the Lists subcategory of the Logics tab.

Note: To see the finished definition seen below, **Open** the file **List Management.ghx** found in the Source Files folder that accompanies this document.



To start, we have created a Series component, with a starting value of 0.0, a step value of 1.0, and a count of 10. Thus, the Post-it panel connected to the Series-S output shows the following list of numbers: 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, and 9.0.

Note: The Post-it panel's default setting is to show the list index number before the numeric data, so that each panel entry actually looks like this:



```
List
0 0.0
1 1.0
2 2.0
3 3.0
4 4.0
5 5.0
6 6.0
7 7.0
8 8.0
9 9.0
```

These can be toggled on/off by right-clicking on the Post-it panel and changing the Entry Numbers preview to either on or off. However, we will leave the Entry Numbers turned on for this example, as it will allow us to see exactly what list index number is assigned to each value.

A) The numeric data is then fed into a **List Item** component (Logic/List/List Item), which is used to retrieve a specific entry within the list. When accessing individual items in a list, one has to use an index value. The first item in any list is always stored at location zero, the second item at location 1 and so on and so forth. Typically, if you start to access a list at index -5, an error will occur since no such location exists. We have connected the Series-S output into the List Item-L input. Additionally, we fed an integer into the List Item-i input, which defines which list index number we would like to retrieve. Since we have set this value to 5.0, the List Item output will show the numeric data associated with the 5th entry number, which in this case is also 5.0.

B) The **List Length** component (Logic/List/List Length) essentially evaluates the number of entries in the list and outputs the last entry number, or the length of the List. In this example, we have connected the Series-S output to the List Length-L input, showing that there are 10 values in the list.

C) We can invert the order of the list by using a **Reverse List** component (Logic/List/Reverse List). We have input an ascending list of numbers into the Reverse List component, whereby the output shows a descending list from 9.0 to 0.0.

D) The **Shift** component (Logic/List/Shift List) will either move the list up or down a number of increments dependent on the value of the shift offset. We have connected the Series-S output into the Shift-L input, while also connecting a numeric slider to the

Shift-S input. We have set the numeric slider type to integers so that the shift offset will occur in whole numbers. If we set the slider to -1, all values of the list will move down by one entry number. Likewise, if we change the slider value to +1, all values of the list will move up by one entry number. We can also set the wrap value, or the Shift-W input, to either True or False by right-clicking the input and choosing Set Boolean. In this example, we have a shift offset value set to +1, so we have a decision to make on how we would like to treat the first value. If we set the wrap value to True, the first entry will be moved to the bottom of the list. However, if we set the wrap value to False, the first entry will be shifted up and out of the list, essentially removing this value from the data set.

E) The **Split List** component does just what you think it might do... it splits a list into two smaller lists. It has a splitting index, which is just an integer where it will split the list. In this case, we have told the Split List component to split it after the 5th entry item, thus our output will look like this: List A - 0.0, 1.0, 2.0, 3.0, 4.0 List B - 5.0, 6.0, 7.0, 8.0, 9.0.

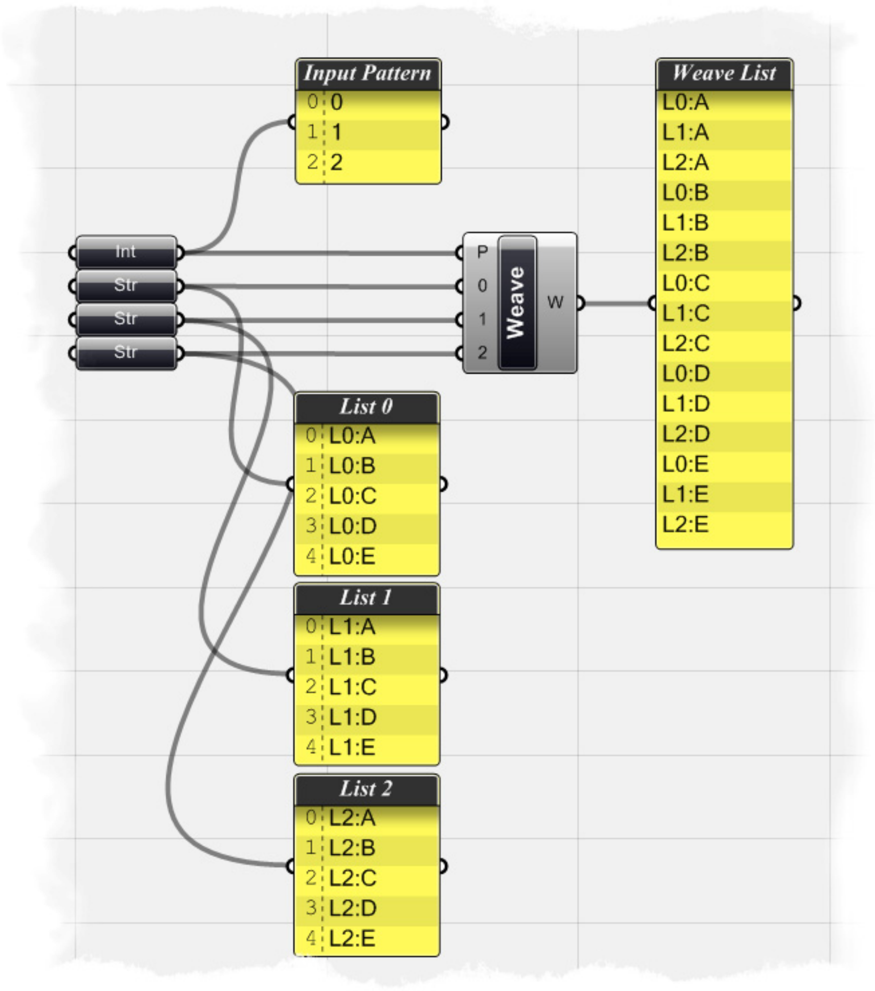
F) The **Cull Nth** component (Logic/Sets/Cull Nth) will remove every Nth data entry from the list, where N is defined by a numeric integer. In this example, we have connected a numeric slider to the Cull Nth-N input. We have set our slider to 2.0, such that the Cull Nth component will remove every other entry from the list. The Cull Nth-L output reveals a new culled list where every odd entry has been deleted: 0.0, 2.0, 4.0, 6.0, and 8.0. If we change the numeric slider to 3.0, the Cull Nth component will remove every third number from the list so that the output would be: 0.0, 1.0, 3.0, 4.0, 6.0, 7.0, and 9.0.

G) The **Cull Pattern** component (Logic/Sets/Cull Pattern) is similar to the Cull Nth component, in that it removes items from a list based on a defined value. However, in this case, it uses a set of boolean values that form a pattern, instead of numeric values. If the boolean value is set to True, the data entry will remain in the list; whereas a false value will remove the data entry from the set. In this example, we have set the boolean pattern to: True, True, False, False. Since there are only 4 boolean values and our list has 10 entries, the pattern will be repeated until it reaches the end of the list. With this pattern, the output list will look like: 0.0, 1.0, 4.0, 5.0, 8.0, and 9.0. The Cull Pattern component kept the first two entries (0.0 and 1.0) and then removed the next two values (2.0 and 3.0). The component continued this pattern until reaching the end of the list.

8.2 Weaving Data

In the last section, we explained how we can use a number of different components to control how lists are managed in Grasshopper, but we can also use the **Weave** Component to control the order of a list. The Weave component can be found under the Lists subcategory of the Logics Tab (Logics/Lists/Weave).

Note: To see the finished definition seen below, **Open** the file **Weave Pattern.ghx** found in the Source Files folder that accompanies this document.



Input Pattern
```
0 0
1 1
2 2
```

Weave List
```
L0:A
L1:A
L2:A
L0:B
L1:B
L2:B
L0:C
L1:C
L2:C
L0:D
L1:D
L2:D
L0:E
L1:E
L2:E
```

List 0
```
0 L0:A
1 L0:B
2 L0:C
3 L0:D
4 L0:E
```

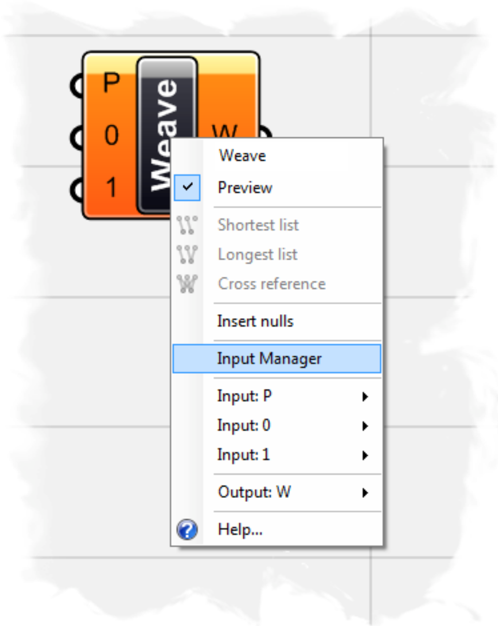
List 1
```
0 L1:A
1 L1:B
2 L1:C
3 L1:D
4 L1:E
```

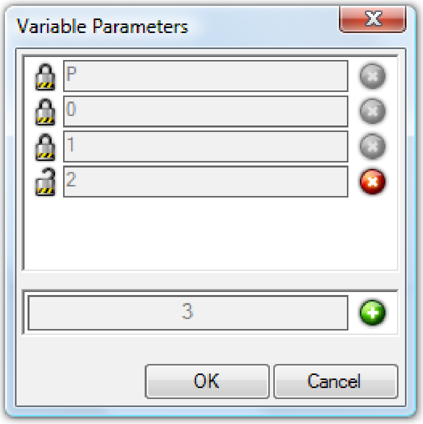
List 2
```
0 L2:A
1 L2:B
2 L2:C
3 L2:D
4 L2:E
```

To create the definition from scratch:

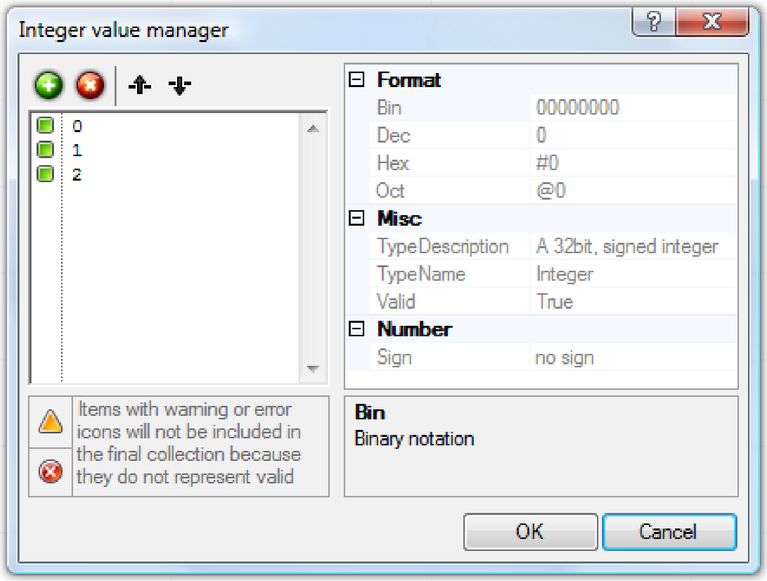
- Logic/List/Weave - Drag and drop a Weave component onto the canvas

  You should immediately see 3 input items. The first, or the P-input, is the weaving pattern and this will determine the order in which we weave our data. The next two input items, label 0 and 1 respectively, will allow us to input two lists to weave together. But, what if we want to weave more than two lists? Well, if you right-click on the middle of the Weave component, you can open the Input Manager and add as many lists as needed. When the Input Manager opens, click on the green plus button to add another list. Similarly, the red "X" next to each list number will remove that list from the component.





- Open the Input Manager and select the green plus button to add an additional list to the component. Your Input Manager should look like the image above.
- Params/Primitive/Integer - Drag and drop an Integer parameter onto the canvas
- Right-click the Integer parameter and select "Manage Integer Collection"
- By clicking on the green plus button at the top, you can add an integer into the collection. Add three numbers to the collection and change each value so that your list looks like this: 0, 1, 2. Your manager should look like the image below.



Format	
Bin	00000000
Dec	0
Hex	#0
Oct	@0
Misc	
TypeDescription	A 32bit, signed integer
TypeName	Integer
Valid	True
Number	
Sign	no sign

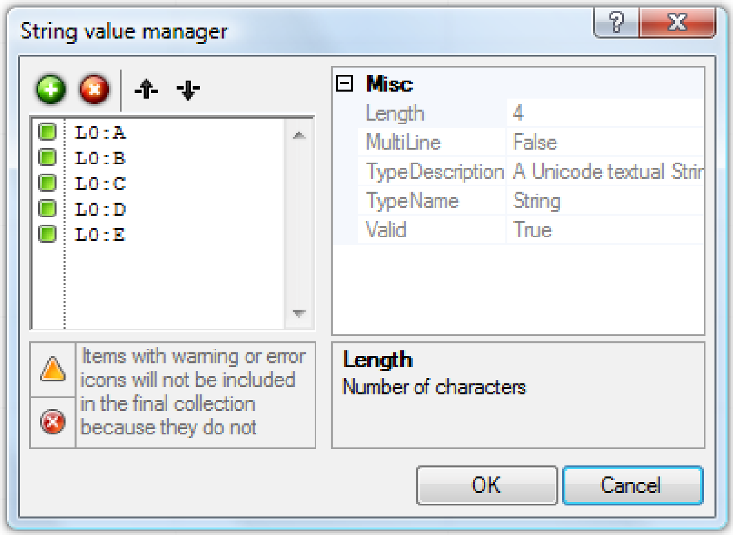
Bin
Binary notation

Items with warning or error icons will not be included in the final collection because they do not represent valid

This integer collection will determine our weaving pattern. By changing the order of these numbers, we can quickly change the order of our data set.

- Params/Primitive/String - Drag and drop a String component onto the canvas
- Right-click on the String component and select "Manage String Collection"

  Just like we did with the integer collection, we're going to add a list of strings that we want to weave together. This will be our first list of data, but we will copy and paste this two more times so that we have three lists to weave together. Add 5 strings to the collection so that your list looks like this: L0:A, L0:B, L0:C, L0:D, L0:E. The L0: prefix just tells us that these strings are in list zero and will help us be able to track the data once we weave them together. Your String Manager should look like the image below.



L0:A
L0:B
L0:C
L0:D
L0:E

Misc	
Length	4
MultiLine	False
TypeDescription	A Unicode textual Strir
TypeName	String
Valid	True

Items with warning or error icons will not be included in the final collection because they do not

Length
Number of characters

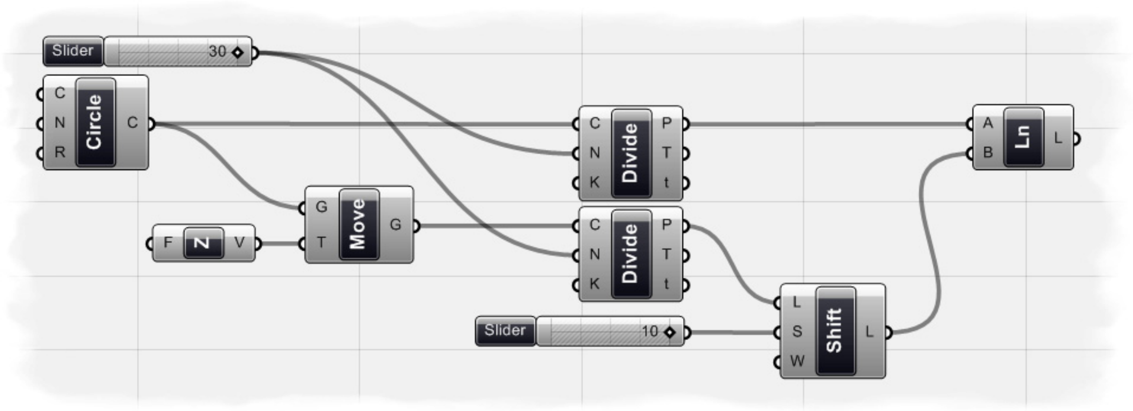
- Select the String parameter and hit Ctrl+C (copy) and Ctrl+V (paste) to duplicate **2 more** String parameters to the canvas
- Right-click the second String parameter and open the String Collection Manager. Change your string collection so that your list now reads like this: L1:A, L1:B, L1:C, L1:D, L1:E.
- Right-click the third String parameter and open the String Collection Manager. Change your string collection so that your list now reads like this: L2:A, L2:B, L2:C, L2:D, L2:E.
- Now, connect the Integer parameter - the one that is going to define our weave pattern - to the P-input of the Weave component
- Connect the first String parameter to the 0-input of the Weave component
- Connect the second String parameter to the 1-input of the Weave component
- Connect the third String parameter to the 2-input of the Weave component
- Params/Special/Post-it Panel - Drag and drop a Post-it Panel onto the canvas
- Connect the Weave-W output to the Post-it Panel so that we can look at the data.

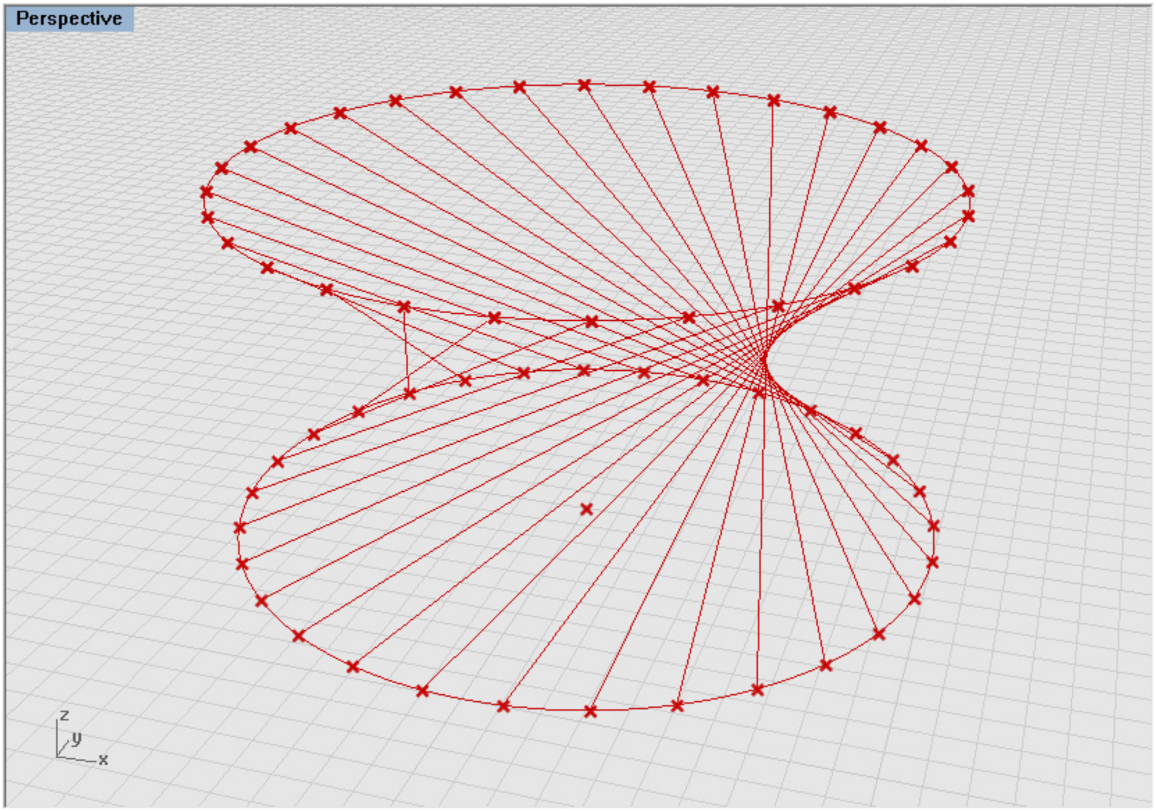
  The Post-it Panel should now show a list of data that has been weaved together according to our Integer collection. The first item in the list should be the first item from list 0. Accordingly, the second item in the list will be the first item of list 1. And so on so forth until it reaches the end of the list. Try changing the order of the Integer collection to change the order of your weaved list.

8.3 Shifting Data

We discussed in section 8.1 how we can use the **Shift** component to move all values in a list up or down depending on a shift offset value. Below is an example, created by David Rutten, to demonstrate how we can use the shift component on two sets of point lists of a circle. You can find more information on this example here: http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryShiftExample.html

Note: To see the finished definition of the following example, **Open** the file **Shift Circle.ghx** found in the Source Files folder that accompanies this document. Below is a look at the finished definition needed to generate the shifted circle example.





To create the definition from scratch:

- Curve/Primitive/Circle CNR - Drag and drop a Circle CNR (Center, Normal, and Radius) onto the canvas
- Right-click on the Circle-C input and click Set One Point
- In the Rhino dialogue box, type "0,0,0" and hit enter
- Right-click on the Circle-R input and Set Number to 10.0
- Vector/Constants/Unit Z - Drag and drop a Unit Z vector component onto the canvas
- Right-click on the F input of the Unit Z component and Set Number to 10.0
- X Form/Euclidean/Move - Drag and drop a Move component onto the canvas
- Connect the Unit Z-V output to the Move-T input
- Connect the Circle-C output to the Move-G input

  We have just created a circle whose center point is at 0,0,0 and has a radius of 10.0 units. We then used the move component to copy this circle and move the duplicated circle in the Z axis 10.0 units.
- Curve/Division/Divide Curve - Drag and drop **two** Divide Curve components onto the canvas
- Connect the Circle-C output to the first Divide-C input
- Connect the Move-G output to the second Divide-C input
- Params/Special/Slider - Drag and drop a numeric slider onto the canvas
- Select the slider and set the following parameters:
  - Slider Type: Integers
  - Lower Limit: 1.0
  - Upper Limit: 30.0
  - Value: 30.0
- Connect the numeric slider to both Divide Curve-N input components

  You should now see 30 points evenly spaced along each circle
- Logic/List/Shift List - Drag and drop a Shift List component onto the canvas
- Connect the second Divide Curve-P output to the Shift List-L input
- Params/Special/Slider - Drag and drop a numeric slider onto the canvas
- Select the new slider and set the following parameters:
  - Slider Type: Integers
  - Lower Limit: -10.0
  - Upper Limit: 10.0
  - Value: 10.0
- Connect the numeric slider output to the Shift List-S input
- Right-click the Shift List-W input and set the boolean value to True

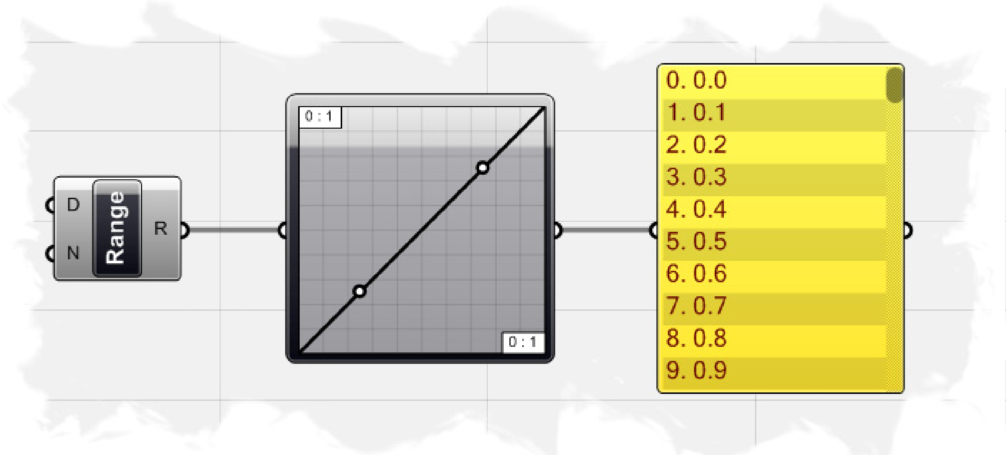
  We have shifted the points on the upper circle up the index list by 10 entries. By setting the wrap value to True, we have created a closed loop of data entries.
- Curve/Primitive/Line - Drag and drop a Line component onto the canvas
- Connect the first Divide Curve-P output to the Line-A input
- Connect the Shift-L output to the Line-B input

  We have created a series of line segments that connect the un-shifted list of points on the bottom circle to the shifted set of points on the upper circle. We can change the value of the numeric slider that controls the Shift Offset to see the line segments change between the un-shifted and shifted list of points.

8.4 Exporting Data to Excel

There are many instances where you may need to export data from Grasshopper in order to import the information into another software package for further analysis.

Note: To see the finished definition of the following example, **Open** the file **Stream Contents_Excel.ghx** found in the Source Files folder that accompanies this document.

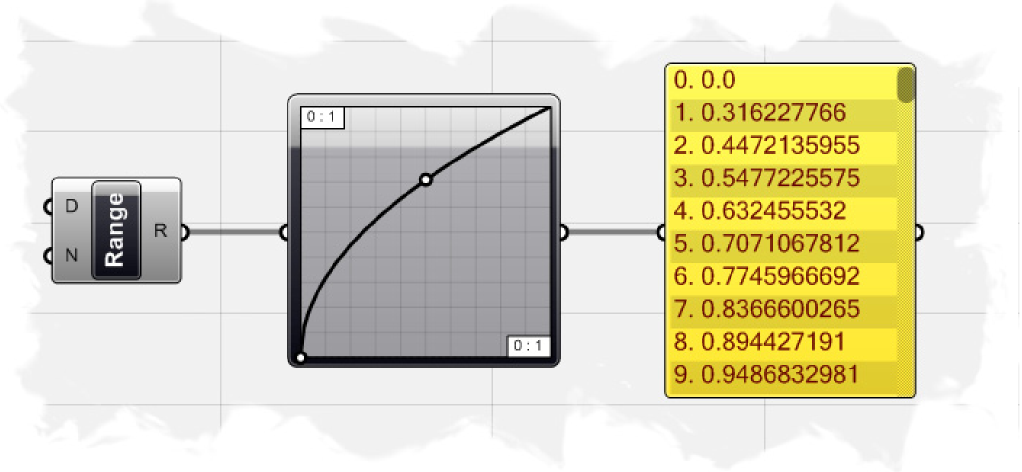


```
0. 0.0
1. 0.1
2. 0.2
3. 0.3
4. 0.4
5. 0.5
6. 0.6
7. 0.7
8. 0.8
9. 0.9
```

To start, we have dropped a **Range** component (Logic/Sets/Range) onto the canvas, and set the numeric domain from 0.0 to 10.0. By right-clicking on the Range-N input, we have set the number of steps to 100, so that our output list will show 101 equally spaced values between 0.0 and 10.0.

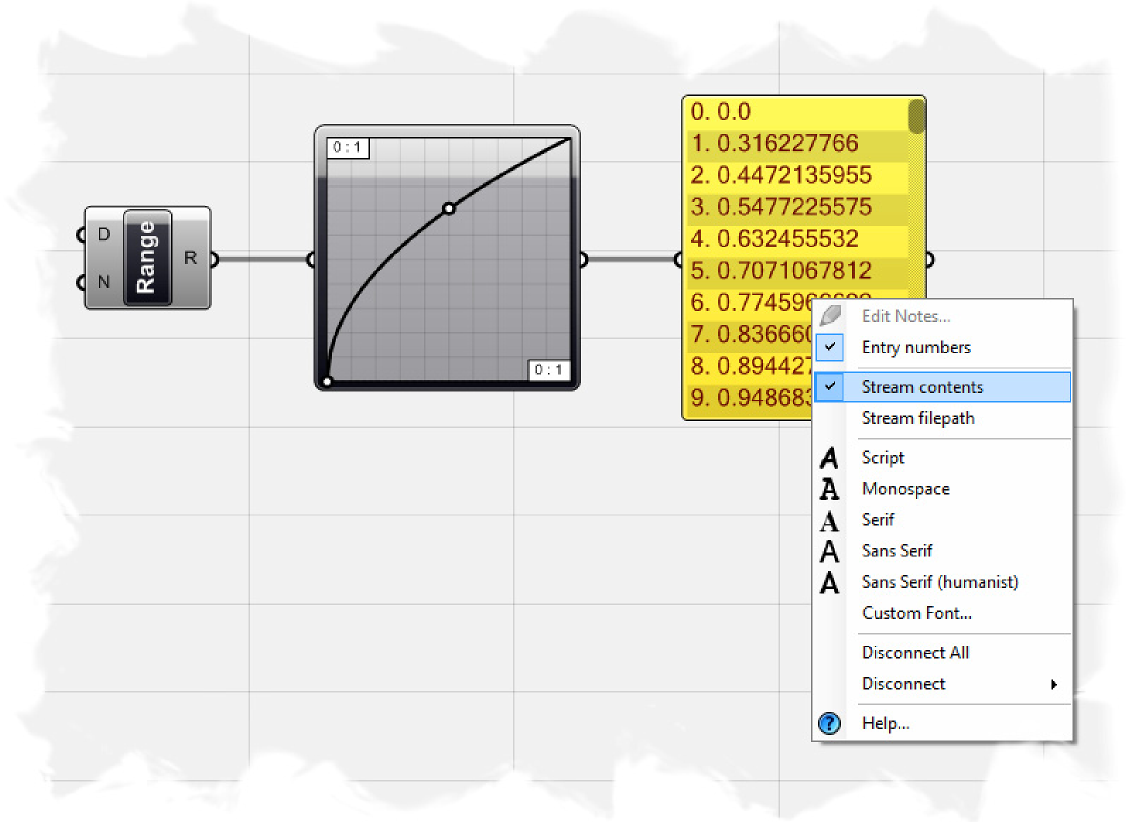
We then drag and drop a **Graph Mapper** component (Params/Special/Graph Mapper) onto the canvas. Right-click on the Graph Mapper component and set the Graph Type to **Linear**. Now connect the Range-R output to the Graph Mapper input. To finish the definition, drag and drop a **Post-it Panel** component onto the canvas and connect the Graph Mapper output to the Post-it Panel input.

Since the Graph Mapper type is set to Linear, the output list (shown in the Post-it Panel) displays a set of numeric data that ascends from 0.0 to 10.0 in a linear fashion. However, if we right-click on the Graph Mapper component and set the Graph Type to **Square Root**, we should see a new list of data that represents a logarithmic function.



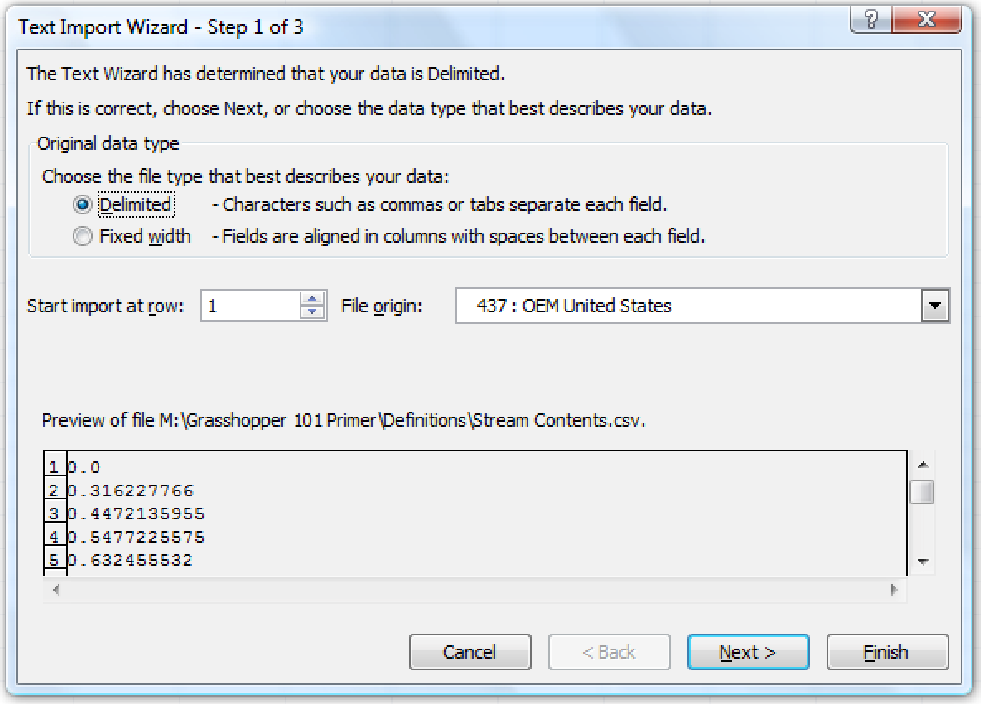
```
0. 0.0
1. 0.316227766
2. 0.4472135955
3. 0.5477225575
4. 0.632455532
5. 0.7071067812
6. 0.7745966692
7. 0.8366600265
8. 0.894427191
9. 0.9486832981
```

To export the list of data from the Post-it Panel, simply right-click on the panel and select **Stream Contents**. When prompted, find a location on your hard drive to **Save** the file with a unique file name. In our example, we will save the file to the following location: C:/Tutorials/Exporting Data/Stream_Contents.csv. There are a variety of file types you can use to save your data, including Text Files (.txt), Comma Separated Values (.csv), and Data Files (.dat) to name a few. I tend to use the Comma Separated Values file format because it was designed for storage of data structured in a table form, although many of the file formats can be imported into Excel. Each line in the CSV file corresponds to a row in the table. Within a line, fields are separated by commas, each field belonging to one table column. Our example only has one value per line, so we will not utilize the mutli-column aspect available with this file format, but it is possible to create complex spreadsheets by exporting the data correctly.



```
0. 0.0
1. 0.316227766
2. 0.4472135955
3. 0.5477225575
4. 0.632455532
5. 0.7071067812
6. 0.77459
7. 0.83666
8. 0.89442
9. 0.94868
```

We can now import the data into Microsoft Excel 2007. First launch the application and select the Data tab. Select the **Get External Data from Text** under this tab and find the Stream_Contents.csv file you saved on your hard drive. You will now be guided through the **Text Import Wizard** where you will be asked a few questions on how you would like to import the data. Make sure that the **Delimited** radial button is marked and select Next to proceed to Step 2.



Text Import Wizard - Step 1 of 3

The Text Wizard has determined that your data is Delimited.

If this is correct, choose Next, or choose the data type that best describes your data.

Original data type

Choose the file type that best describes your data:

● Delimited - Characters such as commas or tabs separate each field.

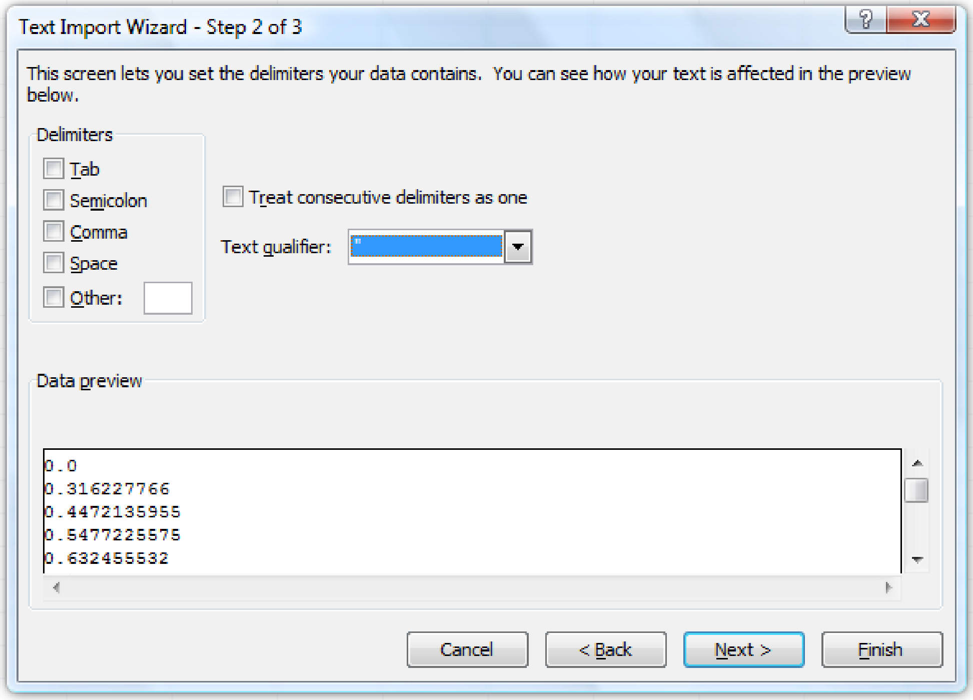
○ Fixed width - Fields are aligned in columns with spaces between each field.

Start import at row: 1 File origin: 437 : OEM United States

Preview of file M:\Grasshopper 101 Primer\Definitions\Stream Contents.csv.

```
1 0.0
2 0.316227766
3 0.4472135955
4 0.5477225575
5 0.632455532
```

Step 2 of the Text Import Wizard allows you to set the which types of Delimiters will define how your data is separated. A Delimiter is some character (such as a semi-colon, comma, or space) stored in the CSV file that indicates where the data should be split into another column. Since we only have numeric data stored in each line of the CSV file, we do not need to select any the specific delimiter characters. Select Next to proceed to Step 3.



Text Import Wizard - Step 2 of 3

This screen lets you set the delimiters your data contains. You can see how your text is affected in the preview below.

Delimiters

□ Tab

□ Semicolon

□ Comma

□ Space

□ Other:

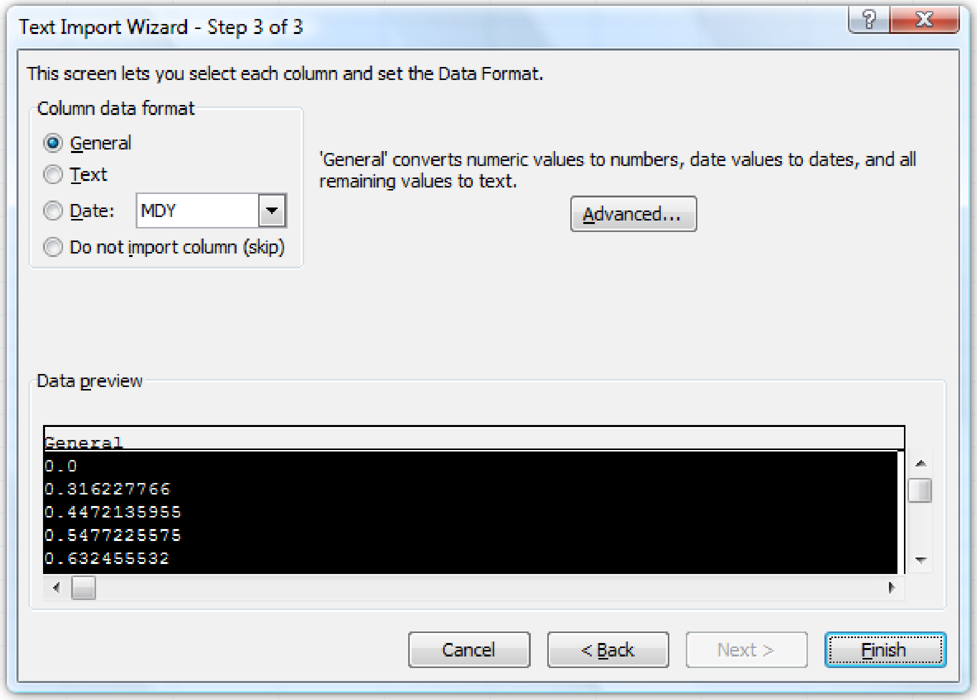
□ Treat consecutive delimiters as one

Text qualifier: "

Data preview

```
0.0
0.316227766
0.4472135955
0.5477225575
0.632455532
```

Step 3 of the Text Import Wizard allows you to tell Excel how you would like to format your data within Excel. General converts numeric data to numbers; Date converts all values to Dates (Day/Month/Year); and all remaining values are formatted as Text data. For our example, select General and hit Finish.



Text Import Wizard - Step 3 of 3

This screen lets you select each column and set the Data Format.

Column data format

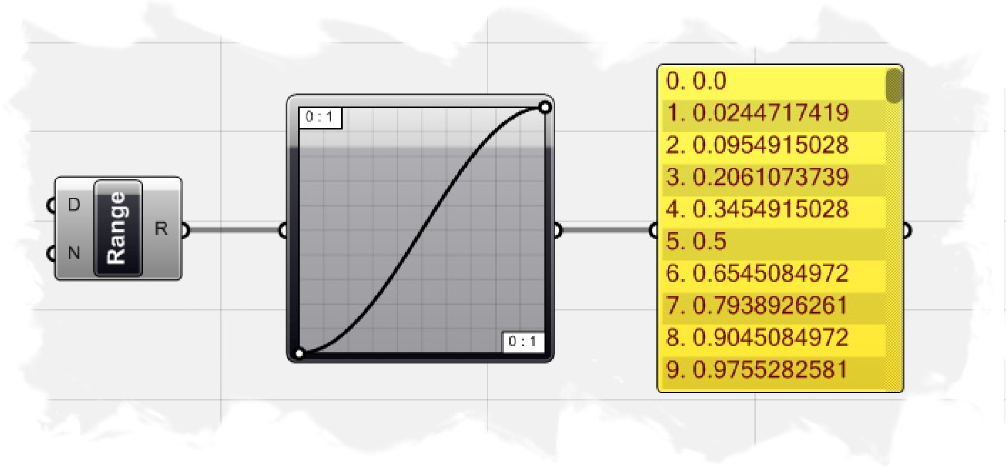
- ● General
- ○ Text
- ○ Date: MDY
- ○ Do not import column (skip)

'General' converts numeric values to numbers, date values to dates, and all remaining values to text.

Data preview

```
General
0.0
0.316227766
0.4472135955
0.5477225575
0.632455532
```

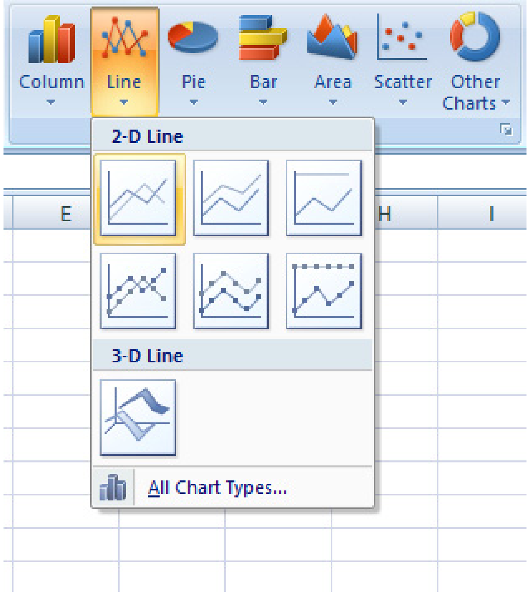
You will now be prompted as to which cell you would like to use to begin importing your data. We will use the default cell value of A1. You will now see all 101 values in the A column that correspond to the values within the Grasshopper Post-it Panel. The Grasshopper definition is constantly streaming the data, so any change we make to the list data will automatically update the CSV file. Go back to Grasshopper and change the Graph Mapper type to **Sine**. Note the list data change in the Post-it Panel.

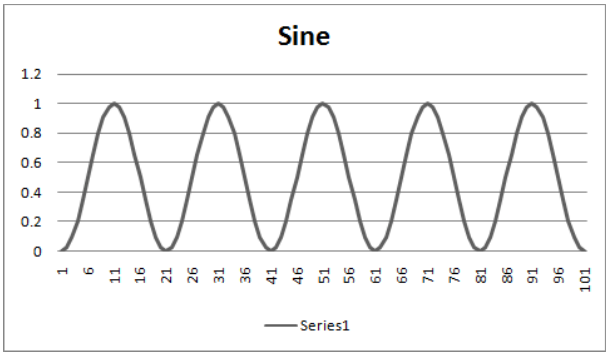


```
0. 0.0
1. 0.0244717419
2. 0.0954915028
3. 0.2061073739
4. 0.3454915028
5. 0.5
6. 0.6545084972
7. 0.7938926261
8. 0.9045084972
9. 0.9755282581
```

Switch back to Microsoft Excel and under the Data Tab, you will see another button that says **Refresh All**. Select this button, and when prompted, select the CSV file that you previously loaded into Excel. The list data in column A will now be updated.

Now select cells A1 through A101 (select A1, and while holding the shift button, select A101) and click on the **Insert** Tab at the top. Choose the **Line Chart** type and select the first 2D line chart icon.

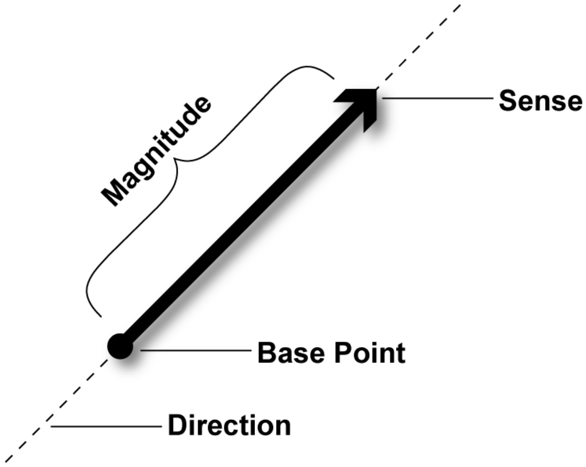




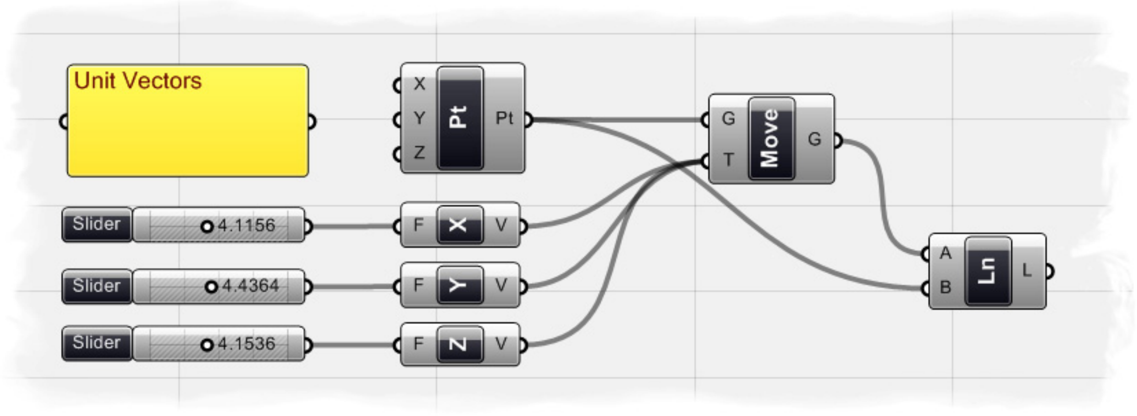
You will see a Line Chart that reflects the same shape shown in the Graph Mapper component in Grasshopper. You can make as many changes as you want within Grasshopper, and after hitting **Refresh All**, you will see the reflected changes in Excel.

9 Vector Basics

From physics, we know that a vector is a geometric object that has a magnitude (or length), direction, and sense (or orientation along the given direction). A vector is frequently represented by a line segment with a definite direction (often represented as an arrow), connecting a base point A with a terminal point B. The magnitude (or amplitude) of the vector is the length of the segment and the direction characterizes the displacement of B relative to A: how much one should move the point A to "carry" it to the point B.



In Rhino, vectors are indistinguishable from points. Both are represented as three doubles, where each double (or numeric variable which can store numbers with decimals) represents the X, Y, and Z coordinate in Cartesian space. The difference is that points are treated as absolutes, whereas vectors are relative. When we treat and array of three doubles as a point, it represents a certain coordinate in space. When we treat the array as a vector, it represents a certain direction. Vectors are considered relative because they only indicate the difference between the start and end points of the arrow, i.e. **vectors are not actual geometrical entities, they are only information.** This means that there is no visual indicator of the vector in Rhino, however we can use the vector information to inform specific geometric actions like translation, rotation, and orientation.



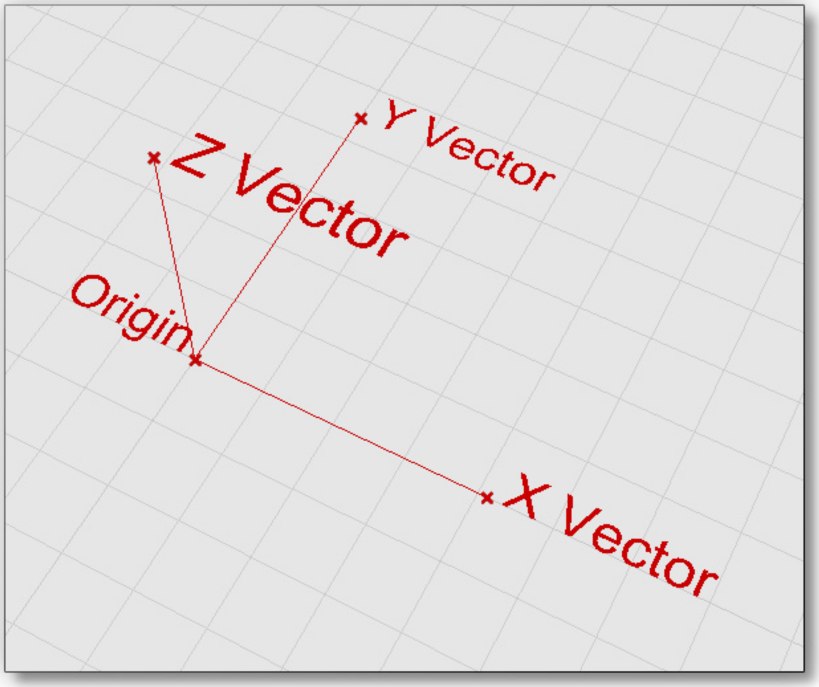
Unit Vectors

Slider 4.1156

Slider 4.4364

Slider 4.1536

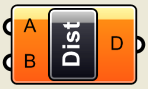
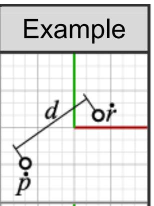
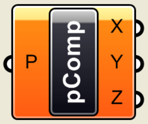
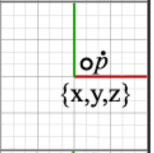
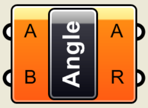
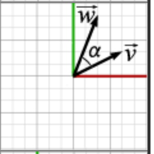
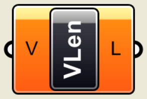
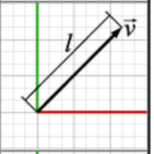
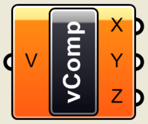
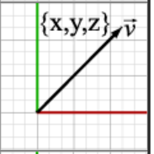
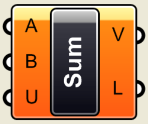
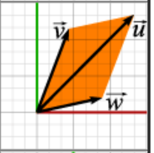
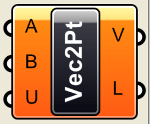
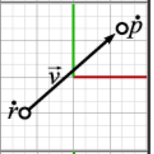
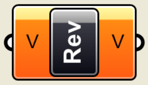
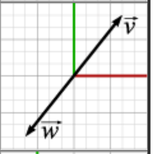
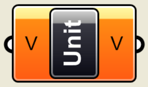
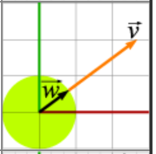
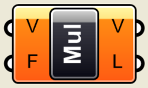
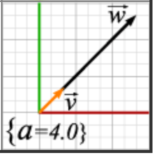
In the example above, we start by creating a point at the origin 0,0,0 using the Point XYZ component (Vector/Point/Point XYZ). We then connect the Point-Pt output to a Move-G input component to translate a copy of the point in some vector direction. To do this, we drag and drop a Unit X, Unit Y, and Unit Z component onto the canvas (Vector/Constants). These components specify a vector direction in one of the Orthagonal directions of the X, Y, or Z axes. We can specify the magnitude of the vector by connecting a numeric slider to the input of each Unit Vector component. By holding down the Shift button while connecting the Unit Vector outputs to the Move-T input, we are able to connect more than one component. Now if you look at the Rhino viewport, you will see a point at the Origin point, and three new points that have been moved in each of the X, Y, and Z axes. Feel free to change the value of any of the numeric sliders to see the magnitude of each vector change. To get a visual indicator of the vector, similarly to drawing an arrow, we can create a line segment from the Origin point to each of the translated points. To do this, drag and drop a Line component (Curve/Primitive/Line) onto the canvas. Connect the Move-G output to the Line-A input and the Point-Pt output to the Line-B input. Below is a screen shot of the Unit Vector definition.



Note: To see the finished definition of the example above, **Open** the file **Unit Vectors.ghx** found in the Source Files folder that accompanies this document.

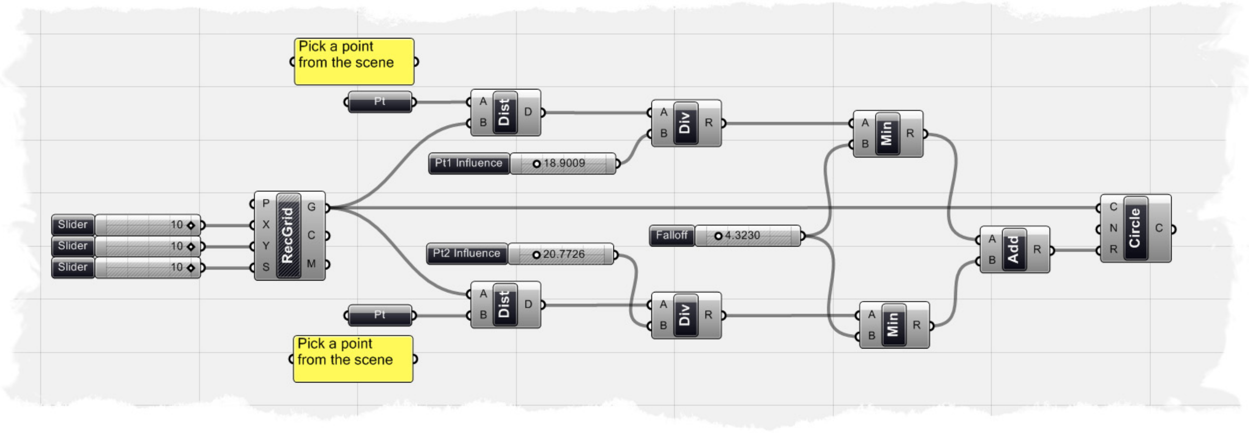
9.1 Point/Vector Manipulation

Grasshopper has an entire group of Point/Vector components which perform the basic operations of 'vector mathematics'. Below is a table of the most commonly used components and their functions.

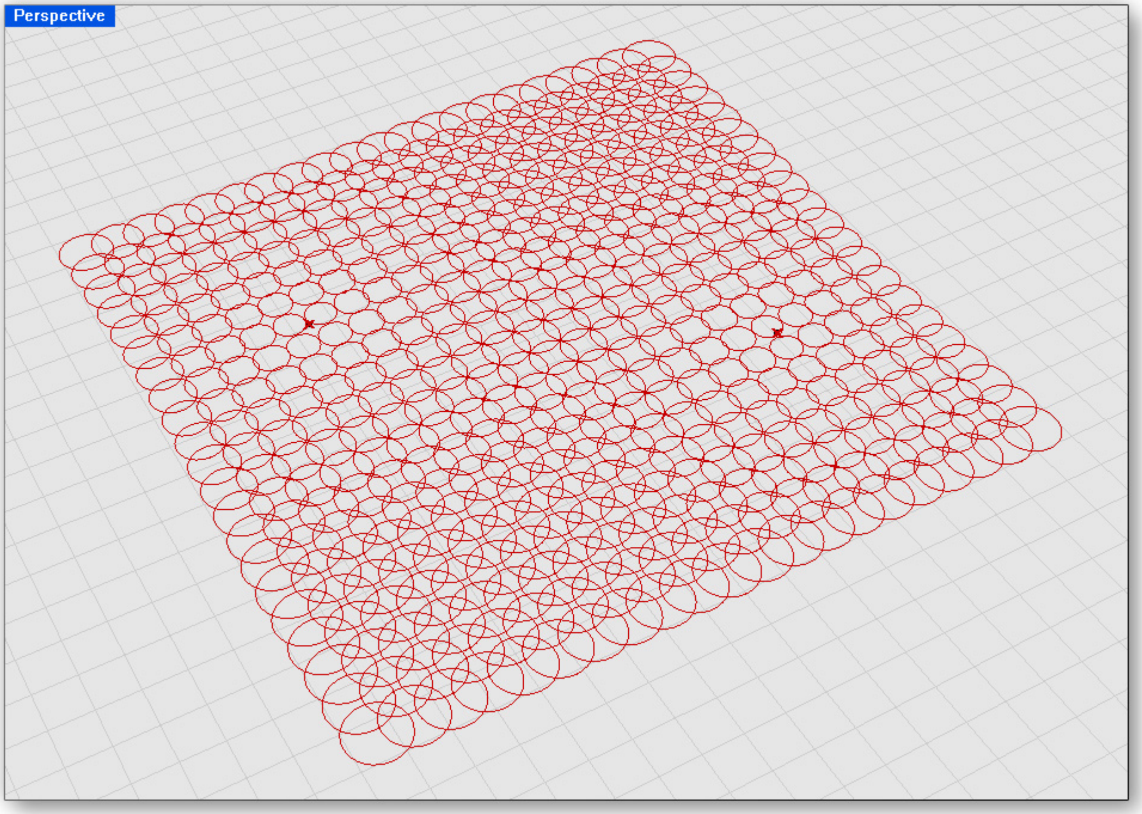
Component	Location	Description	Example
	Vector/Point/**Distance**	Compute the Distance between two points (A and B inputs)	
	Vector/Point/**Decompose**	Break down a point into its X, Y, and Z components	
	Vector/Vector/**Angle**	Compute the angle between two vectors Output computed in Radians	
	Vector/Vector/**Length**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

9.2 Using Vector/Scalar Mathematics with Point Attractors (Scaling Circles)

Now that we know some of the basics behind Scalar and Vector mathematics, lets take a look at an example that scales a grid of circles according to the distance from the center of the circle to a point.



Note: To see the finished definition of this example, **Open** the file **Attractor_2pt_circles.ghx** found in the Source Files folder that accompanies this document. Above is a look at the finished definition needed to generate the series of scaled circles below.



To create the definition from scratch:

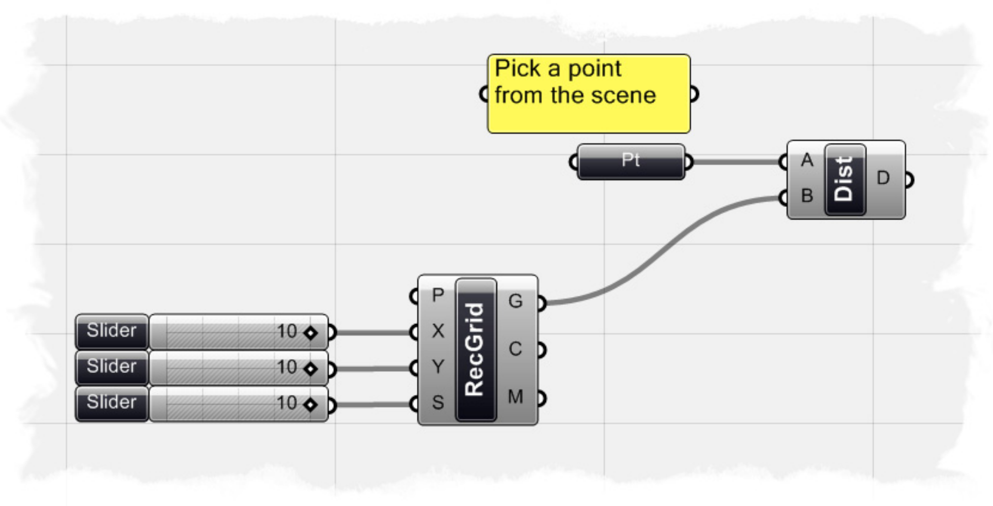
- Params/Special/Numeric Slider - Start by dragging and dropping three numeric sliders onto the canvas
- Right-click all three sliders to set the following:
    - Slider Type: Integers
    - Lower Limit: 0.0
    - Upper Limit: 10.0
    - Value: 10.0
- Vector/Point/Grid Rectangular - Drag and drop a Rectangular Point Grid component onto the canvas
- Connect the first slider to the Pt Grid-X input
- Connect the second slider to the Pt Grid-Y input
- Connect the third slider to the Pt Grid-S input

    The Rectangular Point Grid component create a grid of points, where the P-input is the origin of the grid (in our case we'll use 0,0,0). The Grid component creates a number of points in both the X and Y direction specified by the numeric sliders. However, notice that we have set both of these to 10.0. If you actually count the number of rows and columns, you'll find that there are 20 in each direction. This is because it creates the grid from a center point, and offsets the number of rows and column in each direction from this point. So, essentially you get double the number of X and Y points. The S-input specifies the spacing between each point.
- Params/Geometry/Point - Drag and drop a Point component onto the canvas

    This component is considered an implicit component because it uses persistent data as its input value (See Chapter 4 for more information about persistent data types). This component is different than the other Point XYZ component that we have used before, in that it does not create a point until you actually assign it a point from the scene. To do this, of course, there must already be a point object in your Rhino scene. ***These will be our attractor point.***
- In the Rhino scene, type "Point" in the dialogue box and place a Point object anywhere in your scene. In our case, we'll place the point in the Top viewport so that the point is in the XY plane.

    Now that we have created an attractor point in the scene, we can assign them to the Point component we just created inside Grasshopper.
- Right-click the Point component and select "Set One Point"
- When prompted, select the attractor point object that you created in the Rhino scene

    We have now created a Grid of points and we have also referenced in an attractor point from our scene. We can use a bit of vector mathematics to determine the distance from each grid point to the attractor point.
- Vector/Point/Distance - Drag and drop a Distance component onto the canvas
- Connect the attractor point output to the Distance-A input
- Connect the rectangular Grid Point-G output to the Distance-B input



The first step of our definition should look like the screenshot above. If we hover our mouse over each Distance-D output, we will see a list of numbers which correspond to how far each grid point is from the attractor point. We will use these values to determine each circle's radius, but first we must scale these numbers down to give us more appropriate radius dimensions.

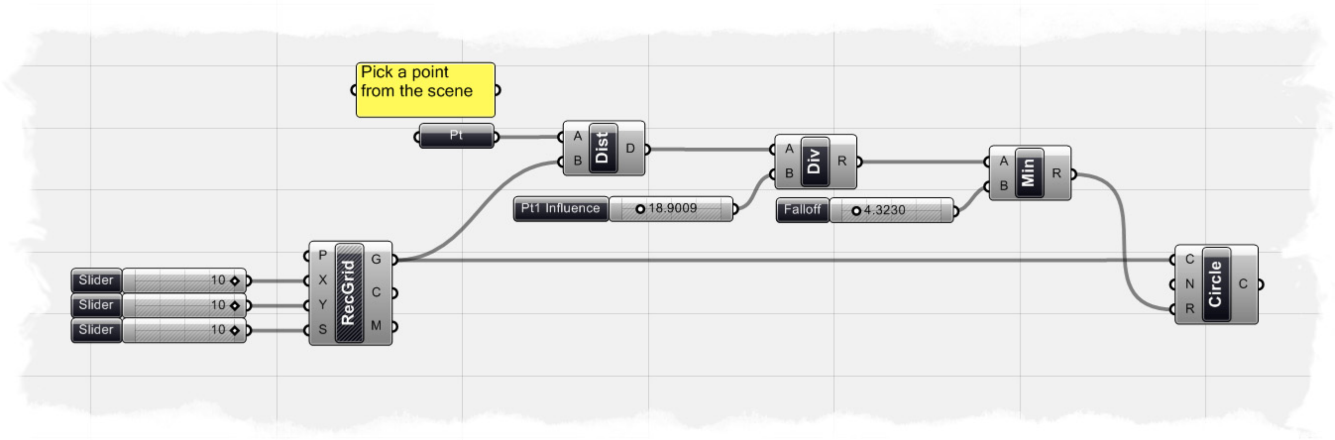
- Scalar/Operators/Division - Drag and drop a Division operator onto the canvas
- Connect the Distance-D output to the Division-A input
- Params/Special/Numeric Slider - Drag and drop a numeric slider onto the canvas
- Right-click the slider and set the following:
  - Name: Pt1 Influence
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 25.0
- Connect the Pt1 Influence slider to the Division-B input

  Since the distance values were quite large, we needed a scale factor to bring the numbers down to a more manageable value. We have used the numeric slider as division factor to give us a new set of output values that we can use for our circle radius. We could directly input these values into a Circle component, but to make our definition a little more robust, we can use some scalar math to determine a falloff distance. What this means, is that we can limit how large our circles become if they get too far away from the attractor point .
- Scalar/Utility/Minimum - Drag and drop a Minimum component onto the canvas
- Connect the Division-R output to the Minimum-A input
- Params/Special/Numeric Slider - Drag and drop a numeric slider onto the canvas
- Right-click the slider and set the following:
  - Name: Falloff
  - Slider Type: Floating Point

- Lower Limit: 0.0
- Upper Limit: 30.0
- Value: 5.0

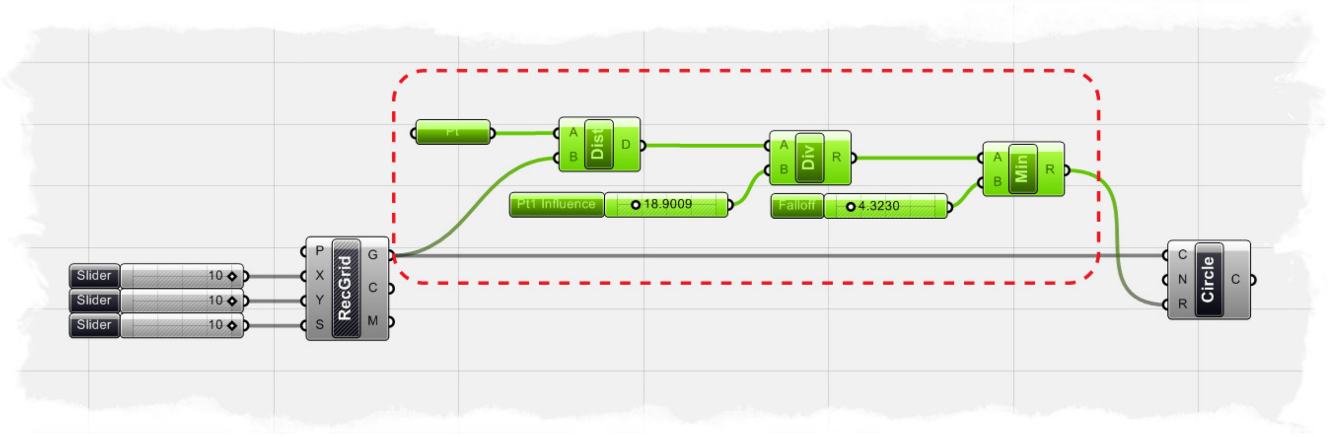
- Connect the Falloff slider to the Minimum-B inputs
- Curve/Primitive/Circle CNR - Drag and drop a Circle CNR (Center, Normal, and Radius) onto the canvas

  We would like the center point of each circle to be located at one of the grid points that we created at the beginning of the definition.
- Connect the Rectangular Point Grid-G output to the Circle-C input
- Connect the Minimum-R output to the Circle-R input
- Right-click on the Rectangular Point Grid component and turn the Preview off



Your definition should look like the image above. We have now create a set of circles that scale up and down depending on how far away each circle is from a single attractor point. But, what if we wanted to add an additional attractor point? Since we already have the definition set up, we merely need to copy and paste a few of our components to make this happen.

- Select the following components: Attractor Point parameter, Distance component, Pt 1 Influence slider, Division component, Falloff slider, and the Minimum component and hit Ctrl-c (copy) and Ctrl-v (paste) to duplicate these components



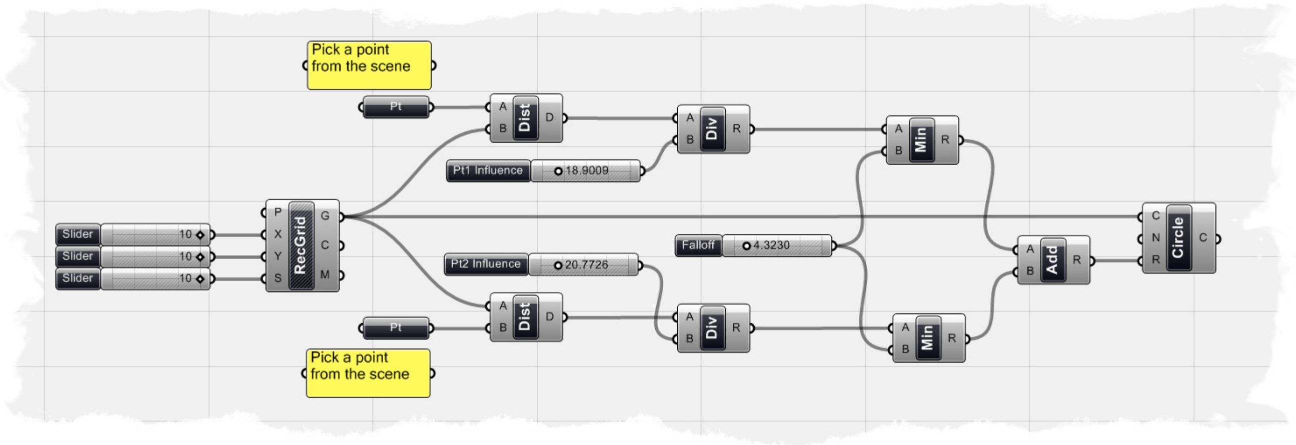
- Move the duplicated components down the canvas slightly, so that your component don't overlap each other

We need to assign another point attractor to our definition... but of course, to do this we need to make another point in our Rhino scene first.

- In the Rhino scene, type "Point" in the dialogue box and place a Point object anywhere in your scene. As we did before, place the point in the Top viewport so that the point is in the XY plane.
- Right-click the duplicated Point parameter and select "Set One Point"
- When prompted, select the attractor point object that you just created in the Rhino scene

We now have 2 rows of components that are evaluating the distance from the rectangular Point Grid and giving us information that we can use to control the radius of a circle. However, we need to combine the two lists that are the result of the Minimum component into a single list. To do this, we will use the scalar Addition component.

- Scalar/Operators/Addition - Drag and drop and Addition component onto the canvas
- Connect the first Minimum-R output to the Addition-A input
- Connect the second Minimum-R output to the Addition-B input
- Now, connect the Addition-R to the Circle-R input (**Note:** *This will replace the existing connection wire.)*

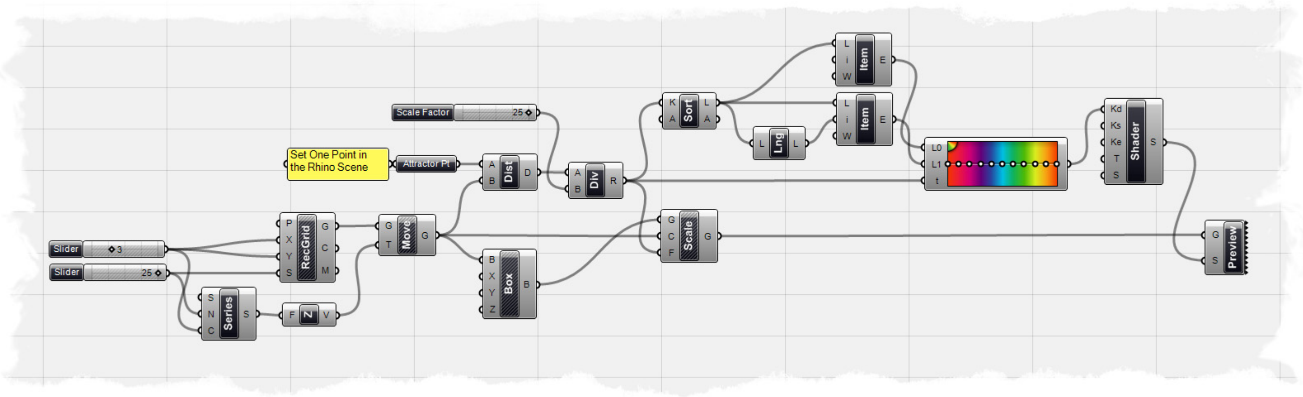


If completed correctly, your definition should look like the screenshot above. You may notice that I've deleted one of the Falloff sliders and I am controlling both of the Minimum components with a single slider. Try playing around with the Pt Influence and Falloff sliders to adjust how your circles scale according to the distance information.

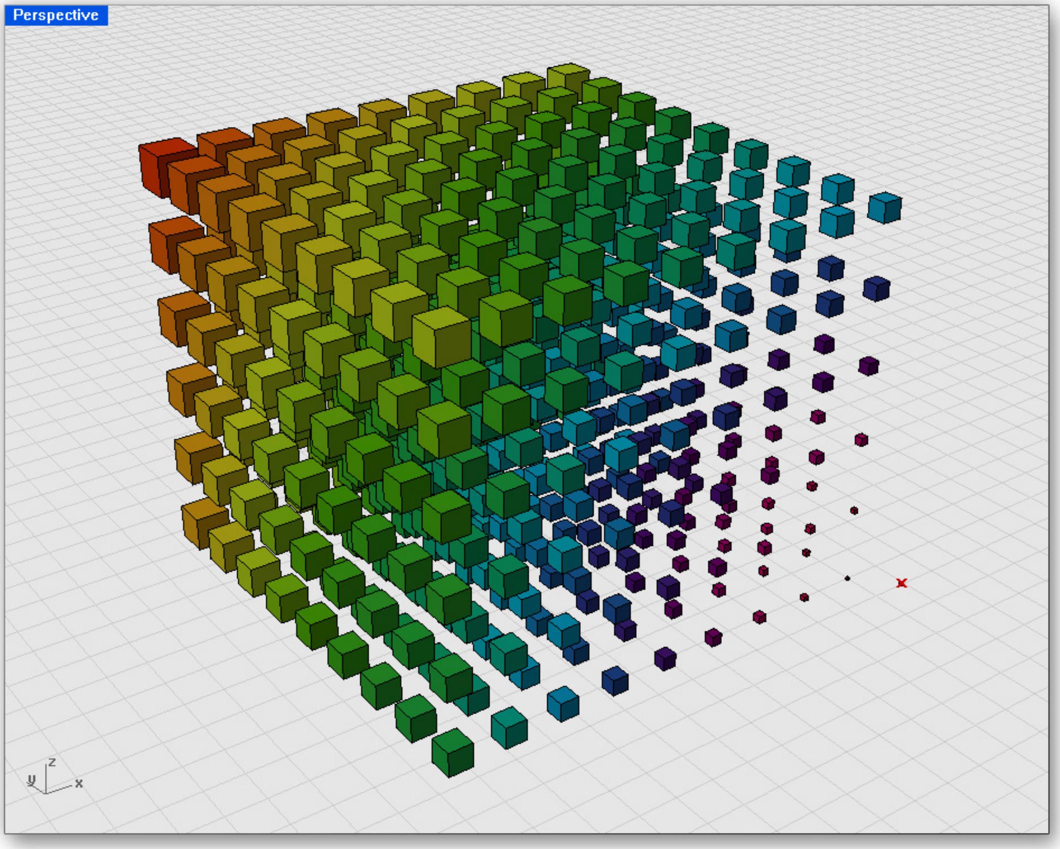
Note: To see a video tutorial of this example, please visit David Fano's blog at:
http://designreform.net/2008/07/08/grasshopper-patterning-with-2-attractor-points/

9.3 Using Vector/Scalar Mathematics with Point Attractors (Scaling Boxes)

We've already shown how we can use vector and scalar mathematics to determine a circles radius based on distances away from other point objects, but we can also use the same core components to scale objects and determine object coloring with Grasshopper's shader components. The following example definition has been used to generate the screen shot below, but let's start from the beginning and work through the definition step by step.



Note: To see the finished definition of this example, **Open** the file **Color Boxes.ghx** found in the Source Files folder that accompanies this document.



Step 1: Begin by creating a three dimensional point grid

- Params/Special/Numeric Slider – Drag and drop 2 sliders onto the canvas
- Right-click on the first slider and set the following:
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 10.0
  - Value: 3.0
- Right-click on the second slider and set the following:
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 25.0
  - Value: 25.0
- Vector/Point/Grid Rectangular – Drag and drop a rectangular Point Grid component onto the canvas
- Connect the first slider to *BOTH* the Point Grid-X & Y input components
- Connect the second slider to the Point Grid-S input

  You should see a grid of point in your scene, where the first slider controls the number of points in both the X and Y axis (remember because if offsets the points from a center point, there are always double the amount of rows and columns as the numeric slider input). The point spacing will be controlled by the second slider. We now need to copy this point grid in the Z-axis to form a three dimensional volume.
- Logic/Sets/Series – Drag and drop a Series component onto the canvas
- Connect the second slider to the Series-N input
- Connect the first numeric slider to the Series-C input

  Our Series component will count the number of copies of the point grid we will make in the Z-direction; however you may have already noticed we have a small error in our math. As was previously mentioned, our point grid was created from a center point, so even though we have set the number of points in both the X & Y axis to 3, we actually have 7 points in each direction. Since we ultimately would like to have a three dimensional cube of points, we need to create 7 copies in the Z axis to be consistent. In order to keep the count uniform, we will need to write a simple expression to double the Series Count.
- Right-click on the Series-C input and scroll down to the Expression Tab
- In the Expression editor, type in the following equation: (**C*2)+1**

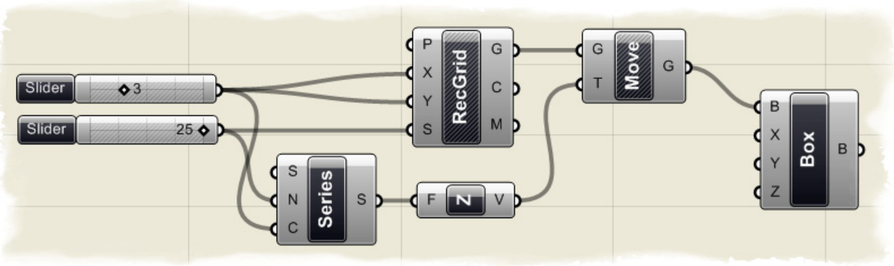
  We have now told the component to multiply Series-C input by a factor of two and then add 1. Since our original input was set to 3, our new Series Count will be 7.0
- Vector/Constants/Unit Z – Drag and drop a Unit Z vector component onto the canvas
- Connect the Series-S output to the Unit Z-F input

  If you hover your mouse over the Unit Z-V output you should see that we have defined 7 locally defined values where the Z value of each entry increases by 25.0, or the value of the second numeric slider. We will use this vector value to define the distance we would like to space each copy of our point grid.
- X Form/Euclidean/Move – Drag and drop a Move component onto the canvas
- Connect the Point Grid-G output to the Move-G input
- Connect the Unit Z-V output to the Move-T input

*If you look at the Rhino scene, you will notice that our points don't necessarily look much like a three dimensional cube of points. If anything, it looks like a ramp leading up to a plane of rectangular points. This is because the default data matching algorithm is set to "**Longest List**". If you right-click on the component, you can change the algorithm to "**Cross Reference**". Now, when you look at your scene you should see a three dimensional cube of points. (See Chapter 6 for more information about Data Stream Matching).*

- Surface/Primitive/Center Box – Drag and drop a Center Box component onto the canvas
- Connect the Move-G output to the Center Box-B input
- Right-click on the Point Grid and the Move component and set the Preview to off

  Below is a screen shot of how our first step of the definition should be set-up.



Step 2: Solve the Scalar and Vector Math

- Params/Geometry/Point – Drag and drop a Point component onto the canvas
- Right-click on the Point component and rename it to: "Attractor Pt"

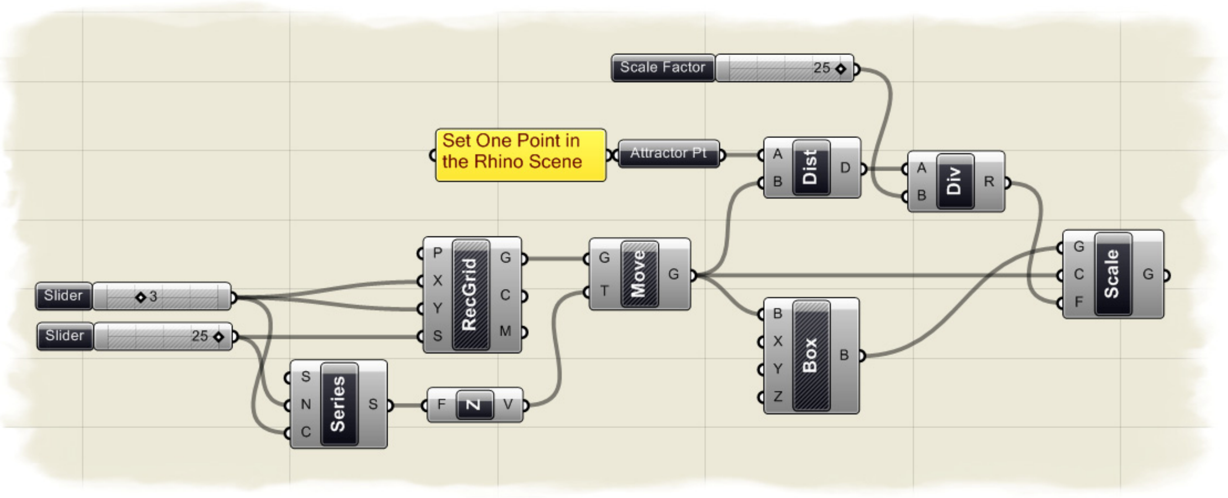
  Just like in the scaling circles example, we will need to assign the Attractor Pt component to a point pre-defined in the Rhino scene. To do this, we will first need to create a point.
- In the Rhino scene, type "Point" in the dialogue box and place a point anywhere in your scene
- Go back into Grasshopper, and right-click on the Attractor Pt component and select "Set One Point"
- When prompted, select the point that you just created in the Rhino scene

  You should now see a small red X over the point indicating that the Attractor Pt component has been assigned that point value in your scene. If you move the point anywhere in your scene, the Grasshopper component will automatically update its position.
- Vector/Point/Distance – Drag and drop a Distance component onto the canvas
- Connect the Attractor Pt output to the Distance-A input
- Connect the Move-G output to the Distance-B input

  If we were to hover our mouse over the Distance-D output, we would see a list of numeric values indicating each points distance away from the Attractor Pt. In order to use these values as a scale factor for our boxes, we will need to divide them by some number to bring the values down to the appropriate levels for our example.

- Scalar/Operators/Division – Drag and drop a Division component onto the canvas
- Connect the Distance-D output to the Division-A input
- Params/Special/Numeric Slider – Drag and drop a slider onto the canvas
- Right-click on the slider and set the following:
  - Name: Scale Factor
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 25.0
  - Value: 25.0
- Connect the Scale Factor slider to the Division-B input
- X Form/Affine/Scale – Drag and drop a Scale component onto the canvas
- Connect the Center Box-B output to the Scale-G input
- Connect the Division-R output to the Scale-F input
- Right-click the Center Box component and turn the Preview off

  Below is a screen shot of how the definition has been set up so far. If you look at the Rhino scene now, you should notice that all of your boxes have been scaled according to their distance away from the attractor point. We can take our definition one step further by adding color to our boxes to give us a visual representation of the scale factor.



Step 3: Assign a color value to each scaled box

- Logic/List/Sort List – Drag and drop a Sort List component onto the canvas

  In order to assign a color value based on each box's distance from the attractor point, we will need to know two numeric values; the closest point and the point that is furthest away. To do this, we must sort our list of distance value, and retrieve the first and last entry from the list.
- Logic/List/List Item – Drag and drop a List Item component onto the canvas
- Connect the Sort List-L output to the List Item-L input
- Right-click on the List Item-i input and Set the Integer value to 0.0

  This will retrieve the first entry in our list which will be the smallest distance value.
- Logic/List/List Length – Drag and drop a List Length component onto the canvas
- Connect the Sort List-L output to the List Length-L input

The List Length component will tell us how many entries are in our list. We can input this information into another List Item component to retrieve the last value in the list.

- Logic/List/List Item- Drag and drop another List Item component onto the canvas
- Connect the Sort List-L output to the second List Item-L input
- Connect the List Length-L output to the second List Item-i input

  If you hover your mouse over the second List Item-E output, you will see that the component has not retrieved the last value in the list. This is because list data in Grasshopper always stores the first value as entry number 0. So, if our list length is showing that there are 100 values in the list and our first number starts at 0, our last entry number will actually be number 99. We must add an expression to the second List Item-i input to subtract 1 value from the list length to actually retrieve the last item.
- Right-click on the second List Item-i input and select the Expression Editor
- In the editor dialogue box, enter the following equation: **i-1**

  Now, if you hover your mouse over the second List Item-E output you should see a numeric value which corresponds to the distance value that is farthest away from the attractor point.
- Params/Special/Gradient – Drag and drop a Gradient component onto the canvas
- Connect the first List Item-E output (the one associated with our closest distance value) to the Gradient-L0 input
- Connect the second List Item-E output (the one associated with our farthest distance value) to the Gradient-L1 input
- Connect the Division-R output to the Gradient-t input

  The L0 input defines what numeric value will represent the left side of the gradient, and in our example, the left side of the gradient will indicate the closest box to the attractor point. The L1 input defines what numeric value will represent the right side of the gradient, and we have set this to represent the value of the point farthest away from the attractor point. The t-input value for the Gradient component represents the list of values you would like to chart along the gradient range. We have chosen to input all of our scale factor values, so that the scale of each box will also be associated with a color along the gradient range.
- Vector/Color/Create Shader – Drag and drop a Create Shader component onto the canvas
- Connect the Gradient output to the Shader-Kd input

  The Shader component has a number of inputs to help define how you want your preview to look. Below is a brief description of how each input effect the resultant shader.

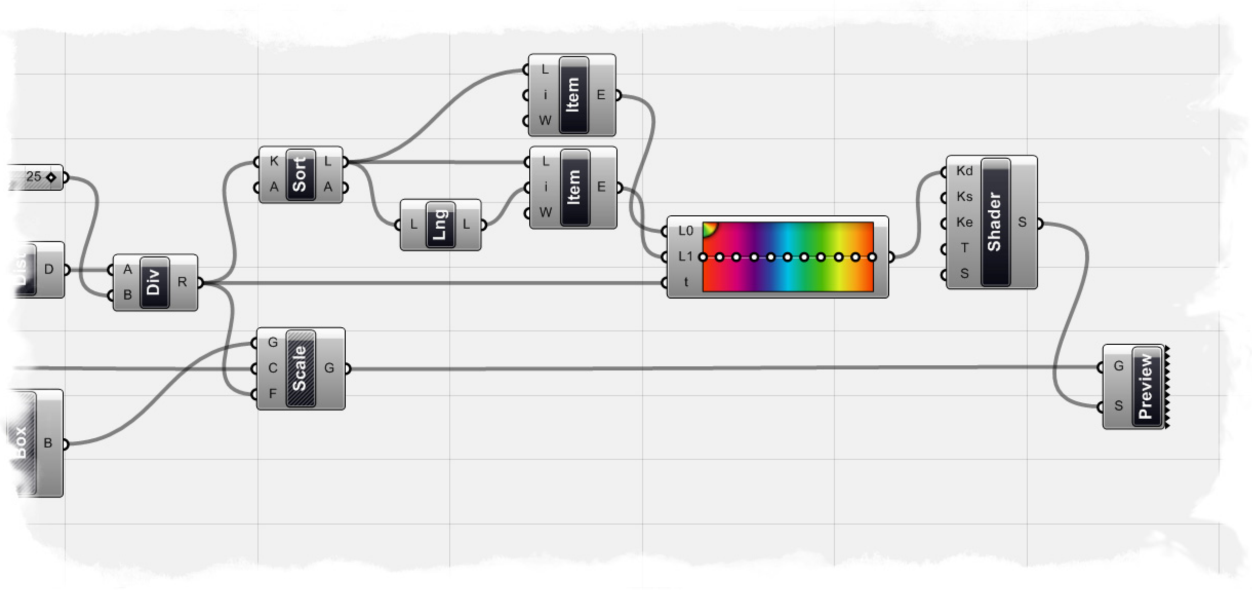
  > ***Kd:** Input defines the Diffuse Color of the shader. This will define the primary color of each object. The diffuse color is defined by three integer numbers that range from 0 – 255, and represent the Red, Green, and Blue values of a color.*
  > ***Ks:** Input defines the color of the Specular Highlight and requires an input of three integer values to define its RGB color.*
  > ***Ke:** Input defines the shader's Emmissivity, or the shader's self illumination color.*
  > ***T:** Input defines a shader's Transparency.*

***S:** Input defines the Shininess of the shader; where a value of 0 means the shader has no shininess, and value of 100 has maximum shininess.*

*We have connected the gradient slider to the Diffuse input of the Shader component so that our box's primary color will be represented by the gradient pattern. You can change the colors of the gradient by selecting one of the small white dots in the Gradient pattern and defining the Color In and the Color Out values. You can also drag the dot up and down the length of the gradient to control the location of where you would like the color to change values. Additionally, there are a number of preset gradient patterns loaded into the Gradient component, and you can set them by right-clicking on the gradient pattern and choosing one of the four preset patterns. Our example has the gradient pattern set to **Spectrum**.*

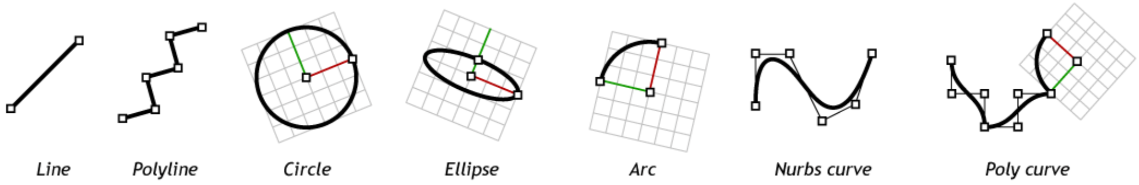
- Params/Special/Custom Preview – Drag and drop a Custom Preview component onto the canvas
- Connect the Scale-G output to the Custom Preview-G input
- Connect the Shader-S output to the Custom Preview-S input
- Right-click and turn the preview off for the Scale component

Below is a screen shot of how we have set up the third step of this definition. If you move the attractor point around in your Rhino scene, the scaled boxes and color information will automatically update.



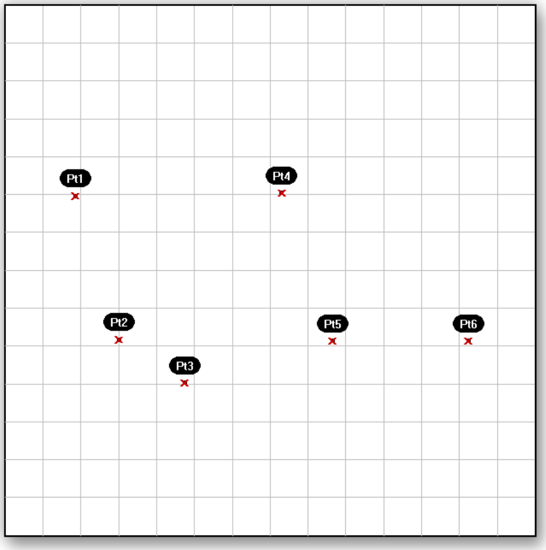
10 *Curve Types*

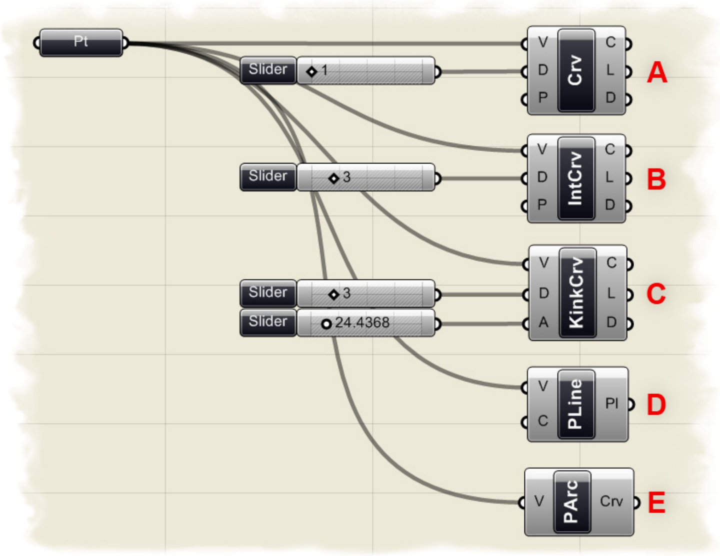
Since curves are geometric objects, they possess a number of properties or characteristics which can be used to describe or analyze them. For example, every curve has a starting coordinate and every curve has an ending coordinate. When the distance between these two coordinates is zero, the curve is closed. Also, every curve has a number of control-points, if all these points are located in the same plane, the curve as a whole is planar. Some properties apply to the curve as a whole, while others only apply to specific points on the curve. For example, planarity is a global property while tangent vectors are a local property. Also, some properties only apply to some curve types. So far we've discussed some of Grasshopper's **Primitive Curve Components** such as: lines, circles, ellipses, and arcs.



Grasshopper also has a set of tools to express Rhino's more advanced curve types like nurbs curves and poly curves. Below, is an example that will walk us through some of Grasshopper's **Spline Components** but first we will need to create a set of points that will define how our curves will act.

In the Source Files folder that accompanies this manual, **Open** the **Curve Types.3dm** file. In the scene, you will find 6 points placed on the X-Y plane. I have labeled them from left to right, like the image on the right, as this will be the order from which will pick them from within Grasshopper.

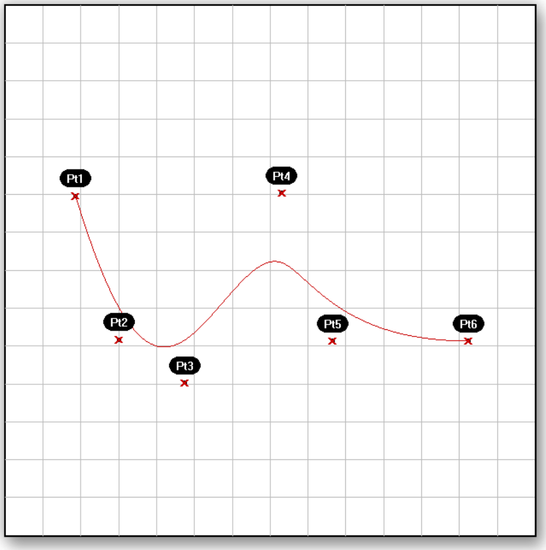




Now in Grasshopper, **Open** the file **Curve Types.ghx** in the Source Files folder that accompanies this document. You will see a Point component connected to several Curve components, each defining a curve using a different method. We will go through each component individually, but first we must assign the points in the Rhino scene to the Point component within Grasshopper. To do this, right-click on the point component and select, **Set Multiple Points**. When prompted select each of the 6 points making sure to select the points in the correct order, from left to right. As you are selecting the points, an implied connection line will be drawn on the screen in Blue to indicate your selections. When you've selected all 6 points, hit enter to return to Grasshopper. All 6 points should now have a small Red X on top of them indicating that this particular point has been assigned to the Grasshopper Point component.

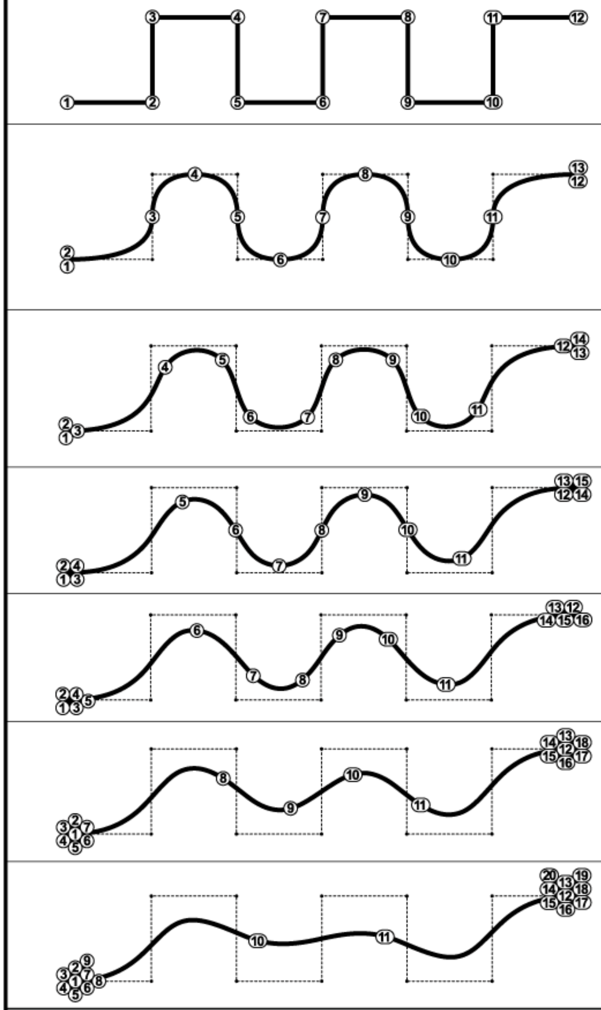
A) NURBS Curves (Curve/Spline/**Curve**)

Non-**U**niform **R**ational **B**asic **S**plines, or NURBS curves, are one of many curve definition types that are available in Rhino. In addition to the control points that help define the curves location (these are the 6 points we just selected in Rhino), NURBS curves also have specific properties like degree, knot-vectors, and weights. Entire books (or at least very lengthy papers) have been written about the mathematics behind NURBS curves, and I will not cover that here. However, if you would like a little more information about this topic, please visit: http://en.wikipedia.org/wiki/NURBS.



The **NURBS Curve-V input** defines the curve control points, and these can be described implicitly by selecting points from within the Rhino scene, or by inheriting volatile data from other components. The **NURBS Curve-D input** sets the degree of the curve. The degree of a curve is always a positive integer between and including 1 and 11. Basically, the degree of the curve determines the range of influence the control points have on a curve; where the higher the degree, the larger the range. The table on the following page is from David Rutten's manual, *Rhinoscript 101*, and illustrates how the varying degrees define a resultant 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.

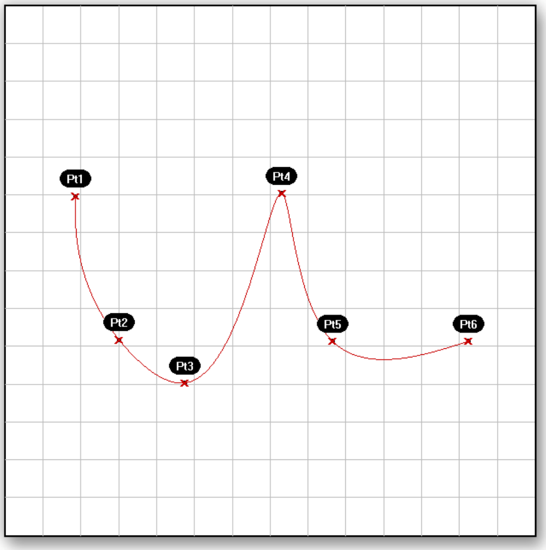
D^5 is also quite a common degree. Like the D^3 curves it has a natural, but smoother appearance. Because of the higher degree, control points have a larger range of influence.

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.

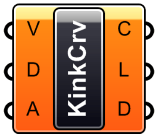
In our example, we have connected a numeric slider to the Curve-D input to define the degree of our NURBS curve. By dragging our slider to the left and right, we can visually see the change in influence of each control point. The **NURBS Curve-P input** uses a boolean value to define whether or not the curve should be periodic or not. A False input will create an open NURBS curve, whereas a True value will create a closed NURBS curve. The three output values for the NURBS Curve component are fairly self-explanatory, where the **C output** defines the resultant curve, the **L output** provides a numeric value for the length of the curve, and the **D output** defines the domain of the curve (or the interval from 0 to the numeric value of the curve degree).

B) Interpolated Curves (Curve/Spline/**Interpolate**)

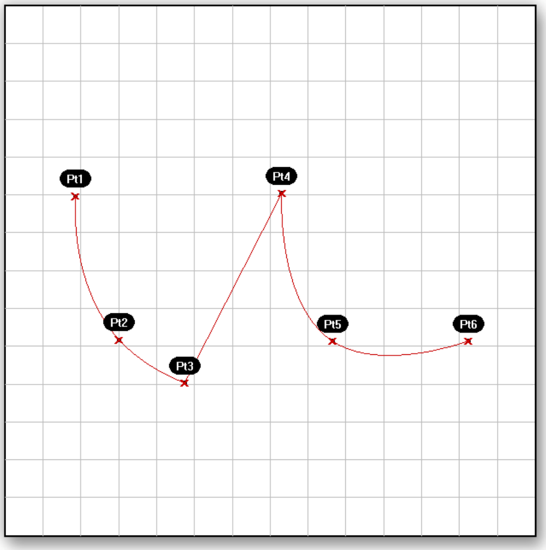
Interpolated curves behave slightly differently than NURBS curves, in that the curve passes through the control points. You see, it is very difficult to make NURBS curves go through specific coordinates. Even if we were to tweak individual control points, it would be an incredibly arduous task to make the curve pass through a specific point. Enter, Interpolated Curves. The **V-input** is for the component is similar to the NURBS component, in that, it asks for a specific set of points to create the curve. However, with the Interpolated Curve method, the resultant curve will actually pass through these points, regardless of the curve degree. In the NURBS curve component, we could only achieve this when the curve degree was set to one. Also, like the NURBS curve component, the **D-input** defines the degree of the resultant curve. However, with this method, it only takes odd numbered values for the degree input, so it is impossible to create a two degree Interpolated curve. Again, the **P-input** determines if the curve is Periodic. You will begin to see a bit of a pattern in the outputs for many of the curve components, in that, the **C**, **L**, and **D outputs** generally specify the resultant curve, the length, and the curve domain respectively.



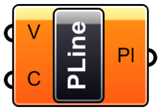
C) Kinky Curves



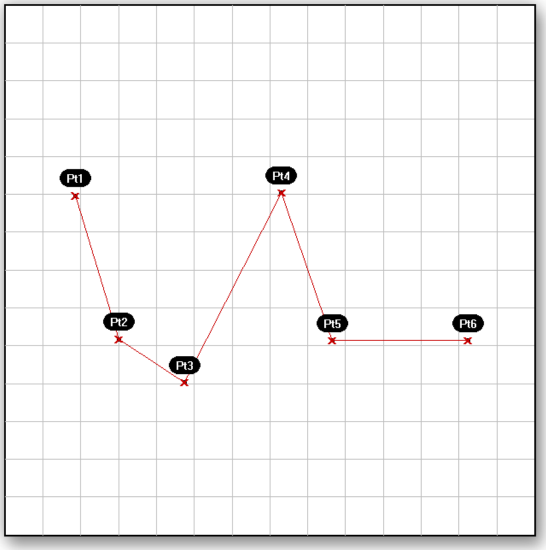
(Curve/Spline/**Kinky Curve**) Despite its name, a kinky curve is no more than a glorified Interpolated Curve. It has many of the attributes of the Interpolated Curve method mentioned in subsection B, with one small difference. The kinky curve component allows you the ability to control a specific angle threshold where the curve will transition from a kinked line, to a smooth interpolated curve. We have connected a numeric slider to the **A-input** of the Kinky Curve component to see the threshold change in real-time. It should be noted that the A-input requires an input in radians. In our example, there is an expression in the A-input to convert our numeric slider, which specifies an angle in degrees, into radians.



D) Polyline Curves (Curve/Spline/Polyline)

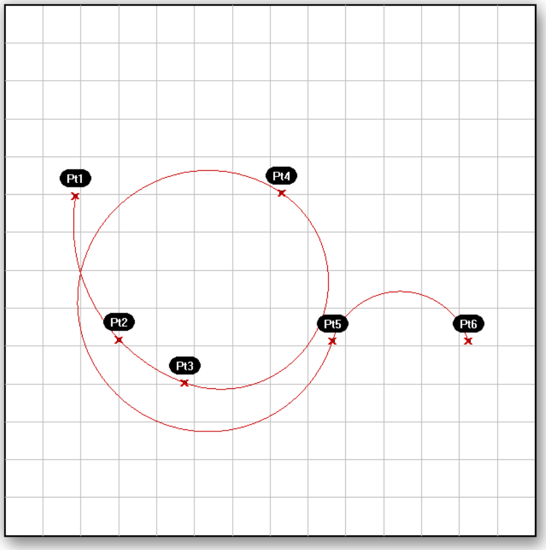


A polyline just may be one of the most flexible curve types available in Rhino. This is because a polyline curve can be made up of line segments, polyline segments, degree=1 NURBS curves, or any combination of the above. But, let's start with the basics behind the polyline. Essentially, a polyline is the same as a point array. The only difference is that we treat the points of a polyline as a series, which enables us to draw a sequential line between them. As was previously mentioned, a one degree NURBS curve, acts, for all intents and purposes, identically to a polyline. Since a polyline is a collection of line segments connecting two or more points, the resultant line will always pass through its control points; making it similar in some respects to an Interpolated Curve. Like the curve types mentioned above, the **V-input** of the Polyline component specifies a set of points that will define the boundaries of each line segment that make up the polyline. The C-input of the component defines whether or not the polyline is an open or closed curve. If the first point location does not coincide with the last point location, a line segment will be created to close the loop. The output for the Polyline component is slightly different than that of the previous examples, in that, the only resultant is the curve itself. You would have to use one of the other analytic curve components within Grasshopper to determine the other attributes of the curve.



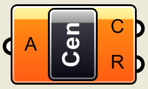
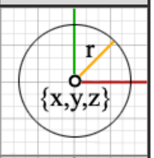
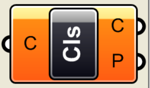
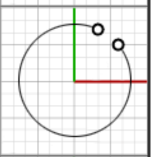
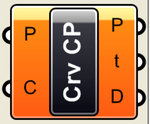
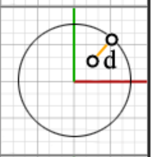
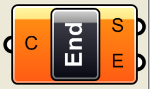
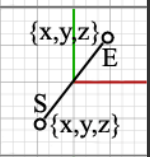
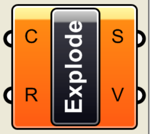
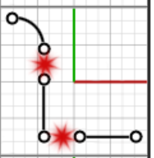
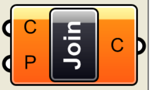
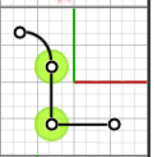
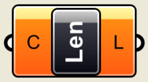
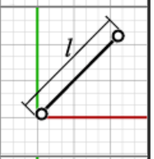
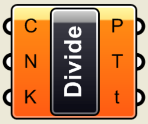
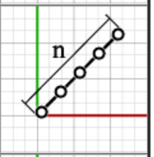
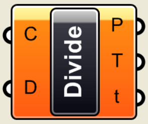
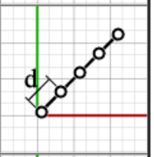
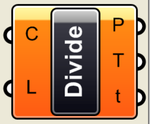
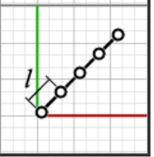
E) Poly Arc (Curve/Spline/Poly Arc)

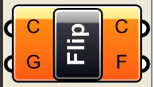
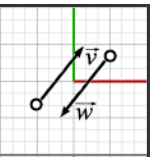
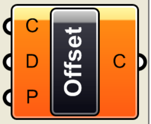
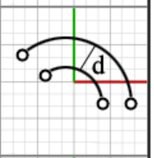
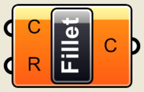
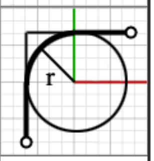
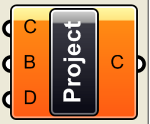
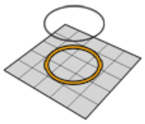
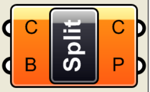
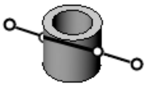
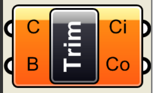
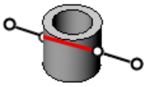
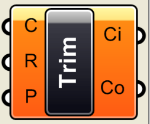
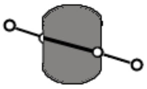
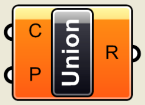
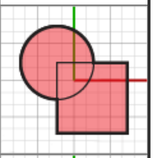
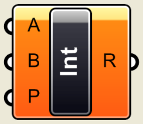
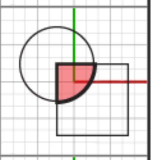
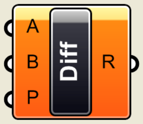
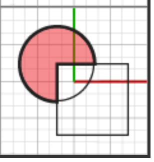
A poly-arc is almost identical in nature to the polyline, except that instead of straight line segments, the poly-arc uses a series of arcs connect each point. The poly-arc is unique, in that, it computes the required tangency at each control point in order to create one fluid curve where the transition between each arc is continuous. There are no other inputs, other than the initial point array, and the only output is the resultant curve.



10.1 Curve Analytics

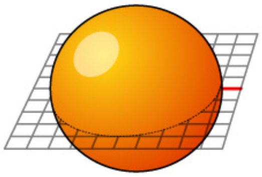
It would be quite difficult to create a tutorial that would utilize all of the analytic tools available in Grasshopper, so I have included a table to explain many of the most commonly used components.

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	
	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	

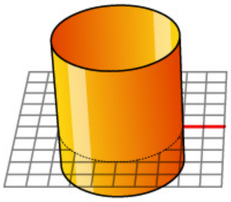
	Curve/Utility/**Flip**	Flip the direction of a curve using an optional guide curve	
	Curve/Utility/**Offset**	Offset a curve with a specified distance	
	Curve/Utility/**Fillet**	Fillets the sharp corners of a curve with an input radius	
	Curve/Utility/**Project**	Project a curve onto a Brep (a Brep is 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.	
	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	

11 Surface Types*

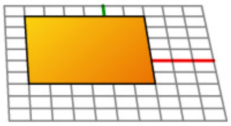
Apart from a few primitive surface types such as spheres, cones, planes and cylinders, Rhino supports three kinds of freeform surface types, the most useful of which is the NURBS surface. Similar to curves, all possible surface shapes can be represented by a NURBS surface, and this is the default fall-back in Rhino. It is also by far the most useful surface definition and the one we will be focusing on.



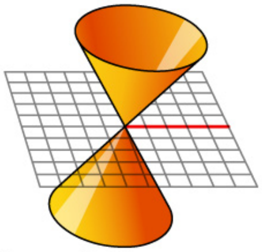
Sphere primitive
{Plane; Radius}



Cylinder primitive
{Plane; Radius; Height}

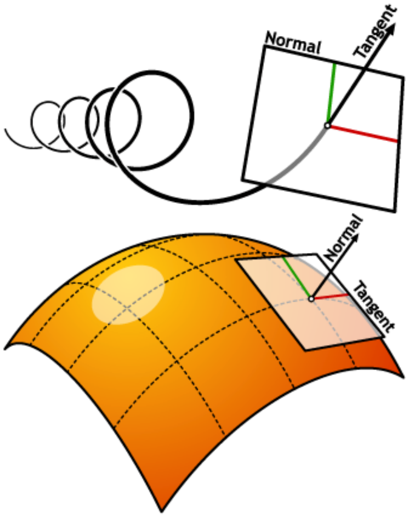


Plane primitive
{Plane; Width; Height}



Cone primitive
{Plane; Radius; Height}

NURBS surfaces are very similar to NURBS curves. The same algorithms are used to calculate shape, normals, tangents, curvatures and other properties, but there are some distinct differences. For example, curves have tangent vectors and normal planes, whereas surfaces have normal vectors and tangent planes. This means that curves lack orientation while surfaces lack direction. This is of course true for all curve and surface types and it is something you'll have to learn to live with. Often when writing code that involves curves or surfaces you'll have to make assumptions about direction and orientation and these assumptions will sometimes be wrong.



In the case of NURBS surfaces there are in fact two directions implied by the geometry, because NURBS surfaces are rectangular grids of {u} and {v} curves. And even though these directions are often arbitrary, we end up using them anyway because they make life so much easier for us.

Grasshopper handles NURBS surfaces similarly to the way that Rhino does because it is built on the same core of operations needed to generate the surface. However, because Grasshopper is displaying the surface on top of the Rhino viewport (which is why you can’t really select any of the geometry created through Grasshopper in the viewport until you bake the results into the scene) some of the mesh settings are slightly lower in order to keep the speed of the Grasshopper results fairly high. You may notice some faceting in your surface meshes, but this is to be expected and is only a result of Grasshopper’s drawing settings. Any baked geometry will still use the higher mesh settings.

* Source: Rhinoscript 101 by David Rutten
http://en.wiki.mcneel.com/default.aspx/McNeel/RhinoScript101

Grasshopper chooses to handle surface in two ways. The first, as we have already discussed, is through the use of NURBS surfaces. Generally, all of the Surface Analysis components can be used on NURBS surfaces, like finding the area or surface curvature of a particular surface. While there is a fair amount of complicated math involved, this would still be fairly easy to solve because the computer doesn't have to take into account the third dimension of a volume, like depth or thickness. But how does Grasshopper interpret three dimensional surfaces? Well, the developers at McNeel decided to throw us a bone, and create a second method from which we can control solid objects like we normally would in the Rhino interface. Enter… the Brep, or the Boundary Representation.

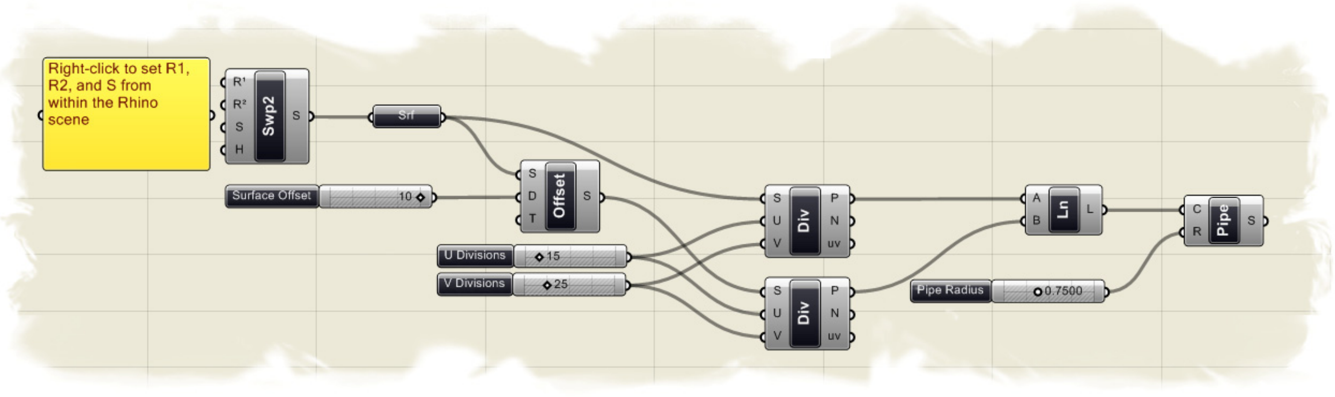
The Brep can be thought of as a three dimensional solid or similarly to the polysurface in Rhino. It is still made up of a collection of NURBS surface, however they are joined together to create a solid object with thickness, whereas a single NURBS surface theoretically has no thickness. Since Breps are essentially made up of surfaces that are joined together, some of the standard NURBS surface analysis components will still work on a Brep, while others will not. This happens because Grasshopper has a built in translation logic which tries to convert objects into the desired input. If a component wants a BRep for the input and you give it a surface, the surface will be converted into a BRep on the fly. The same is true for Numbers and Integers, Colors and Vectors, Arcs and Circles. There are even some fairly exotic translations defined, for example:

- Curve → Number (gives the length of the curve)
- Curve → Interval (gives the domain of the curve)
- Surface → Interval2D (gives the uv domain of the surface)
- String → Number (will evaluate the string, even if it's a complete expression)
- Interval2D → Number (gives the area of the interval)

There are more auto-conversion translations and whenever more data types are added, the list grows. That should be enough background on surface types to begin looking at a few different examples.

11.1 Surface Connect

The following example, created by David Fano from Design Reform, is an excellent tutorial that shows a range of surface manipulation techniques. In this example we'll cover the Sweep2Rail, Surface Offset, and Surface Division components by creating the model displayed below. To start, in Rhino **Open** the **SurfaceConnect.3dm** file found in the Source Files folder that accompanies this manual. In this file you'll find 3 curves (2 rails and a section curve) which will provide the framework needed for this example.



Right-click to set R1, R2, and S from within the Rhino scene

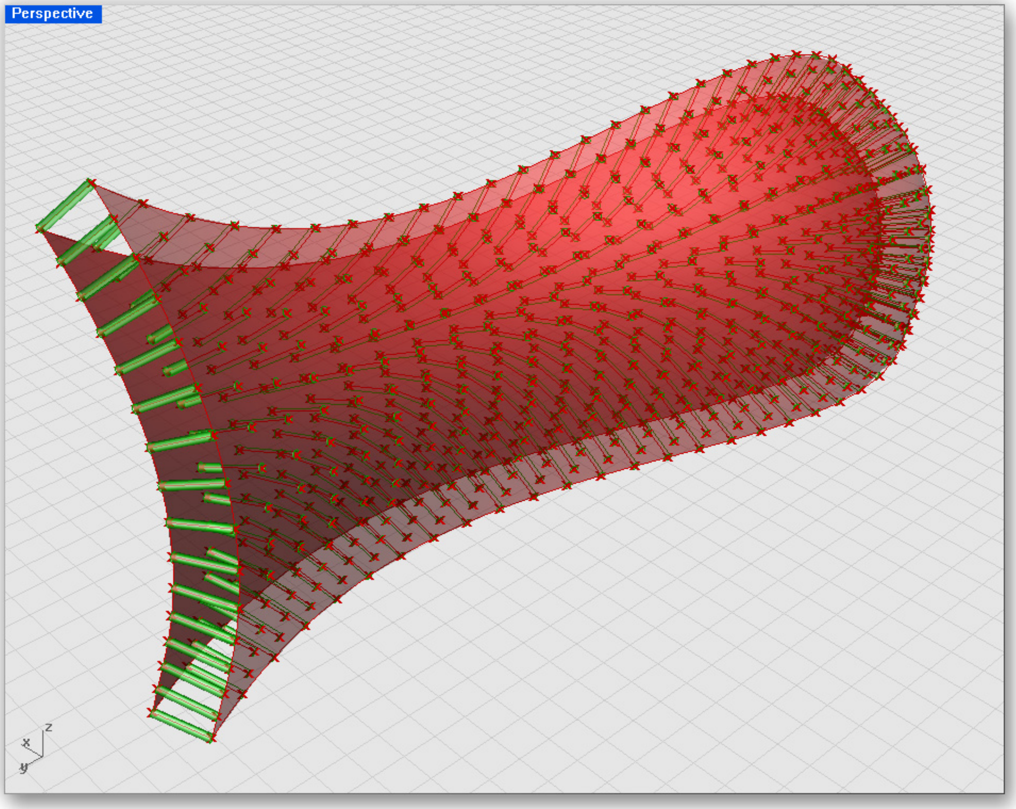
Surface Offset 10

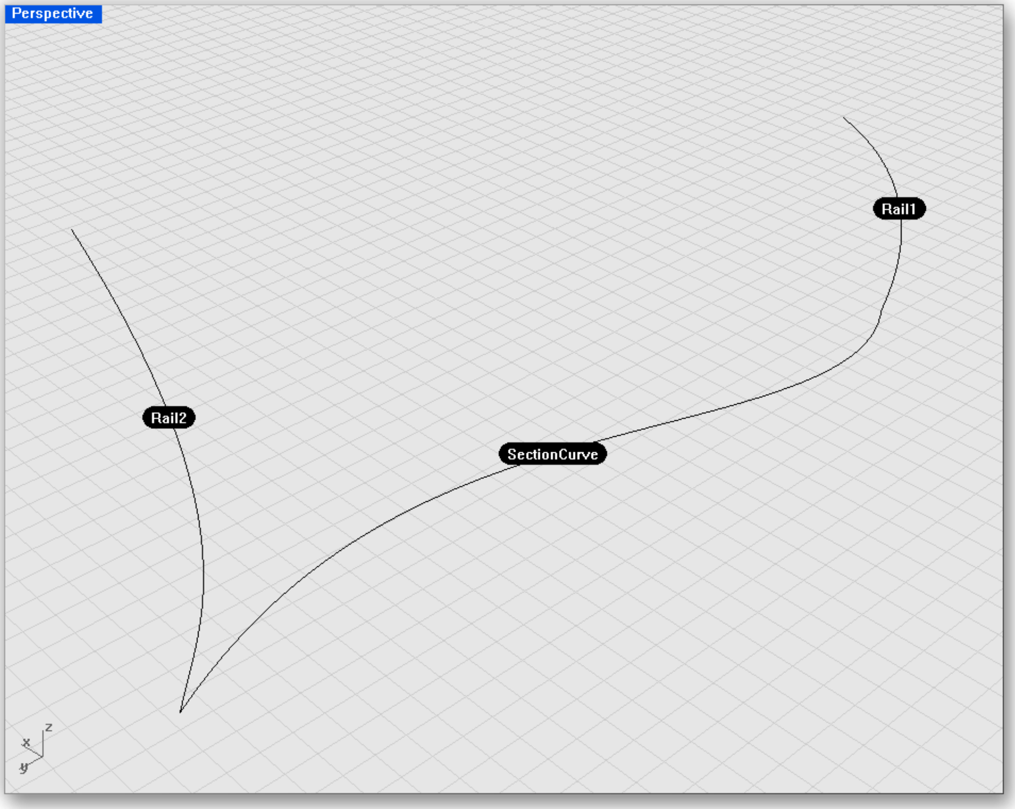
U Divisions 15

V Divisions 25

Pipe Radius 0.7500

Note: To see the finished definition of this example, **Open** in Grasshopper the file **SurfaceConnect.ghx** also found in the Source Files folder.





To create the definition from scratch:

- Surface/Freeform/Sweep2Rail - Drag and drop a Sweep 2 Rails component onto the canvas
- Right-click on the Sweep2Rail-R1 input and select "Set one Curve"
- When prompted, select the first rail curve in the scene (see image above)
- Right-click on the Sweep2Rail-R2 input and select "Set one Curve"
- When prompted, select the second rail curve in the scene (to state the obvious, see the image above)
- Right-click on the Sweep2Rail-S input and select "Set one Curve"
- Again, when prompted, select the section curve in the scene (no need to belabor the point here :)

  If all curves were properly selected, you should now see a surface spanning between each rail curve.
- Surface/Freeform/Offset - Drag and drop a Surface Offset component onto the canvas
- Connect the Sweep2Rail-S output to the Offset-S input
- Params/Special/Slider - Drag and drop a Numeric Slider onto the canvas
- Right-click on the slider and set the following:
  - Name: Surface Offset
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 10.0
  - Value: 10.0
- Connect the slider to the Surface Offset-D input

You should now see a new surface that is offset 10 units (or whatever value you have set for your Surface Offset slider) from the original surface.

- Surface/Utility/Divide Surface - Drag and drop **two** Divide Surface components onto the canvas
- Connect the Sweep2Rail-S output to the first Divide Surface-S input

  You should immediately see a set of points show up on your first Sweep2Rail surface. This is because the default U and V values for the Divide Surface component is set to 10. Basically, the Divide Surface component is creating 10 divisions in each direction on the surface, ultimately creating a grid of points on your surface. If you were to connect the points along each division line you would get the "isocurves" that form the internal framework of the surface.
- Connect the Surface Offset-S output to the Divide Surface-S input

  Again, a new set of grid points have been created on the offset surface.
- Params/Special/Slider - Drag and drop **two** numeric sliders onto the canvas
- Right-click the first slider and set the following:
  - Name: U Divisions
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 15.0
- Right-click the second slider and set the following:
  - Name: V Divisions
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 25.0
- Connect the U Divisions slider to **both** Divide Surface-U inputs
- Connect the V Divisions slider to **both** Divide Surface-V inputs

  The two sliders now control the number of divisions in each direction on both surfaces. Since both surfaces have the exact same number of points, and thus each point has the same index number, we will easily be able to connect the inner surface to the outer surface with a simple line.
- Curve/Primitive/Line - Drag and drop a Line component onto the canvas
- Connect the first Divide Surface-P output to the Line-A input
- Connect the second Divide Surface-P output to the Line-B input

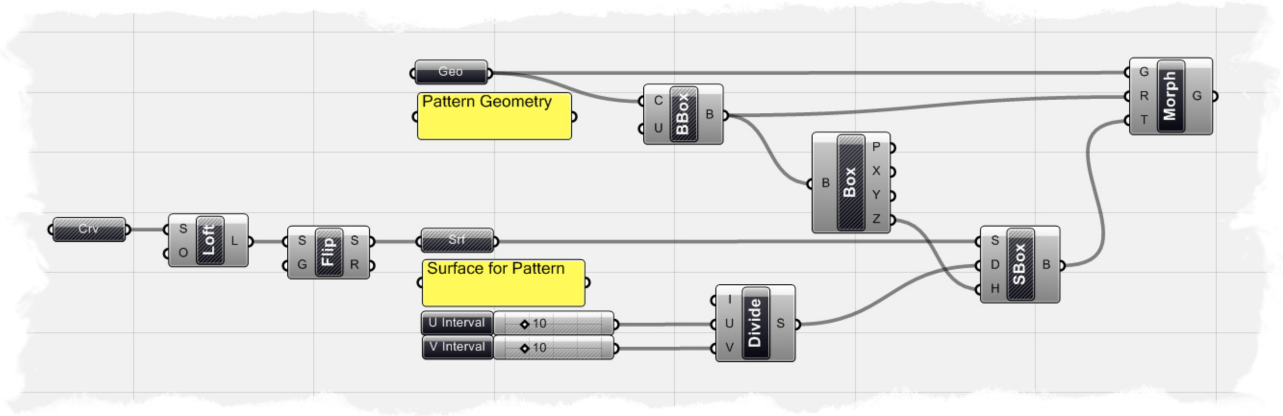
  It's as easy as that. You should now see a series of lines connecting each point of the inner surface to the corresponding point on the outer surface. We can take the definition a step further by giving each line some thickness.
- Surface/Freeform/Pipe - Drag and drop a Pipe component onto the canvas
- Connect the Line-L output to the Pipe-C input
- Params/Special/Slider - Drag and drop a Numeric Slider onto the canvas
- Right-click the slider and set the following:
  - Name: Pipe Radius
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 2.0
  - Value: 0.75
- Connect the Pipe Radius slider to the Pipe-R input

11.2 Paneling Tools

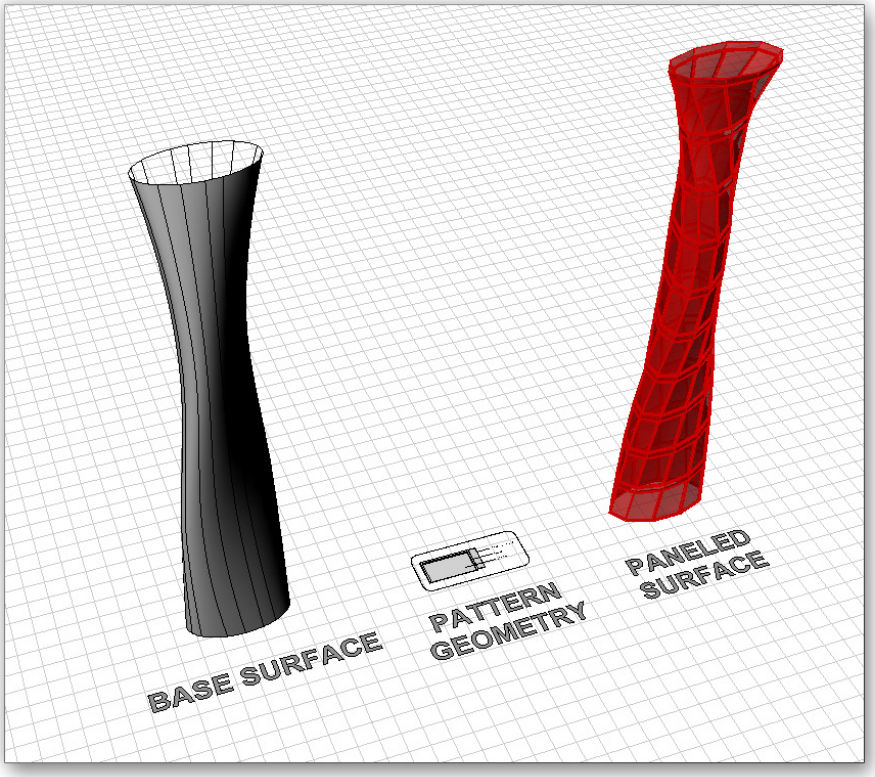
McNeel recently released an excellent free plug-in for Rhino called **Paneling Tools**. The plug-in, among other things, allows you the ability to propagate specific geometric modules over a given surface. Grasshopper also has some components that can be used to re-create the Paneling Tools method and in the following tutorial we'll cover how to subdivide a surface through the use of an Interval component, and we'll cover some of the Morphing components included in Grasshopper.

To find out more about the Paneling Tools plug-in, please visit: http://en.wiki.mcneel.com/default.aspx/McNeel/PanelingTools.html.

Below is a screen shot of the definition needed to copy a geometric pattern, such as a window mullion system, across the surface of a high-rise tower.



Note: To see the finished definition of this example, in Grasshopper **Open** the file **Paneling Tool.ghx** found in the Source Files folder that accompanies this document.



To create the definition from scratch, we'll first need a set of curves in which to Loft the tower surface. In Rhino, **Open** the file **Panel Tool_base.3dm** also found in the Source Files folder. You should find 4 ellipses, which we will use to help us define our surface, and some preset pattern geometry in the scene.

Let's start the definition by creating a lofted surface:

- Params/Geometry/Curve – Drag and drop a Curve component onto the canvas
- Right-click the curve component and select "Set Multiple Curves"
- When prompted, select the 4 ellipses in the scene, one by one, starting from the bottom most curve and working your way up to the top
- Hit enter after you have selected the last ellipse

    As we have done in some of our previous examples, we have now implicitly defined some Rhino geometry within Grasshopper.

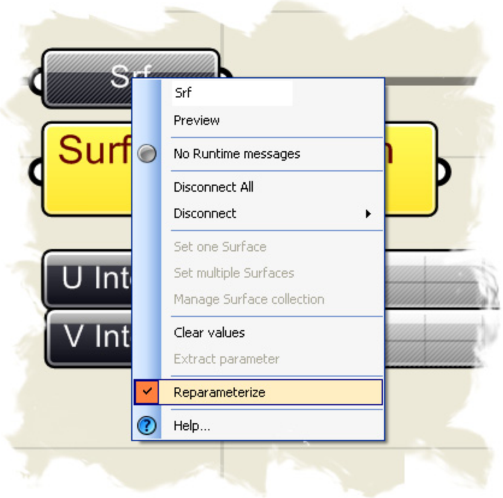
- Surface/Freeform/Loft – Drag and drop a Loft component onto the canvas
- Connect the Curve output to the Loft-S input

    If you right-click on the Loft-O input, you will find the typical loft options associated with the Rhino command. In our case, the default settings will suffice, but there might be a time when these setting would need to be adjusted to fit your particular need.

* **Optional Step:** It is almost impossible to tell whether or not your lofted surface is facing the right direction. However, through trial and error in setting up this tutorial, I noticed that my default lofted surface happened to face inward, thus causing all of my panels to face inward as well. So, we can use the Flip component to reverse the normals of our surface to face outward. We will put the component in our definition, but if you find that when we get to the end of the definition that all of your panels are facing the opposite direction than what you intended, all you will need to do is delete the Flip component and reconnect the appropriate wires.

- Surface/Utility/Flip – Drag and drop a Flip component onto the canvas
- Connect the Loft-L output to the Flip-S input
- Params/Geometry/Surface – Drag and drop a Surface component onto the canvas
- Connect the Flip-S output to the Surface component's input
- Right-click on the Surface component and select the check box next to the word **Reparameterize**

    Currently, our surface has a domain interval that ranges from zero (at the base of the surface) to some number representing the top of the surface. We really don't know care what that upper limit is, because we can reparameterize the surface. Meaning, we will reset the domain of the surface to be an interval from zero to one, where zero represents the base of the surface and one represents the upper limit of the surface. This is a crucial step, because our

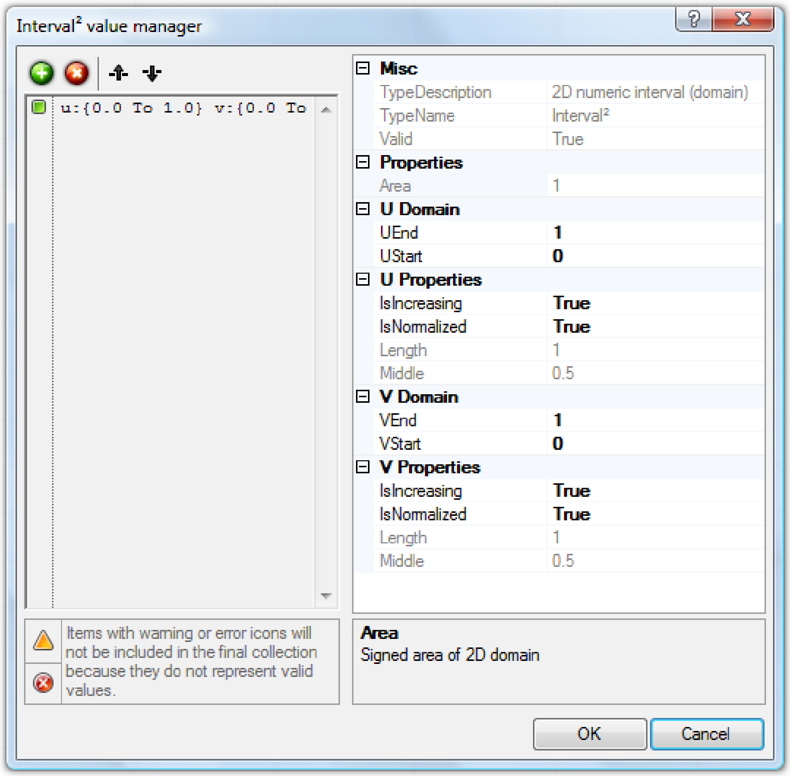


surface will not subdivide correctly if it is not reparameterized into an interval between zero and one.

- Scalar/Interval/Divide Interval^2 – Drag and drop a Divide Two Dimensional Interval (what a mouth full) component onto the canvas

  We'll first need to set up our interval range, before we subdivide our surface.
- Right-click the I-input of the Divide Interval component and select "Manage Interval Collection"
- Click the green plus button to add an interval to the collection

  By default, the interval is set to: u:{0.0 To 0.0} v:{0.0 To 0.0}, but we need the interval to range from 0 to 1 in both the U and V direction.
- Change the value of U-End to 1.0
- Change the value of V-End to 1.0



u:{0.0 To 1.0} v:{0.0 To

Items with warning or error icons will not be included in the final collection because they do not represent valid values.

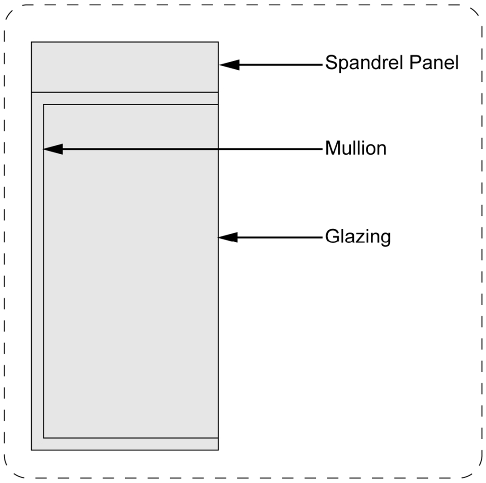
Misc	
TypeDescription	2D numeric interval (domain)
TypeName	Interval²
Valid	True
Properties	
Area	1
U Domain	
UEnd	1
UStart	0
U Properties	
IsIncreasing	True
IsNormalized	True
Length	1
Middle	0.5
V Domain	
VEnd	1
VStart	0
V Properties	
IsIncreasing	True
IsNormalized	True
Length	1
Middle	0.5

Area
Signed area of 2D domain

Your Interval Collection Manager should look like the image above. Click OK to accept the interval. Now, if you hover your mouse over the Divide Interval-I input, you will now see that both our U and V base intervals range from zero to one, which also corresponds to our reparameterized surface.

- Params/Special/Slider - Drag and drop **two** numeric sliders onto the canvas
- Right-click on the first slider and set the following:
  - Name: U Interval
  - Slider Type: Integers
  - Lower Limit: 5.0
  - Upper Limit: 30.0
  - Value: 10.0
- Right-click on the second slider and set the following:

- Name: V Interval
  - Slider Type: Integers
  - Lower Limit: 5.0
  - Upper Limit: 30.0
  - Value: 10.0
- Connect the U Interval slider to the Divide Interval-U input
- Connect the V Interval slider to the Divide Interval-V input
- Xform/Morph/Surface Box - Drag and drop a Surface Box component onto the canvas
- Connect the Surface component output to the Surface Box-S input
- Connect the Divide Interval-S output to the Surface Box-D input
- Right-click on the Curve, Loft, and Surface components and turn their 'Preview' off



We have now subdivided our surface into 100 areas based on our U and V interval sliders. What is happening is that, we originally created an interval that ranged from zero to one, which did correspond to the same interval value of our surface. Then, we divided that interval 10 times in the U direction and 10 times in the V direction, which ultimately created 100 unique interval combinations. You can change the U and V values on the sliders to control the amount of subdivision in each direction of your surface. Now, let's take a step back and create a geometric pattern that we can propagate across each new subdivision. In the scene, you will find some a window system consisting of a spandrel panel, a window mullion, and a glazing panel. We will use these three geometric instances to create a facade system on our surface.

- Params/Geometry/Geometry - Drag and drop a Geometry component onto the canvas
- Right-click on the Geometry component and select "Set Multiple Geometries"
- When prompted, select the spandrel panel, mullion, and the glazing panel from the scene
- Hit enter after selecting all three Breps
- Surface/Primitive/Bounding Box - Drag and drop a Bounding Box component onto the canvas
- Connect the Geometry component output to the Bounding Box-C input

We're using the Bounding Box component for two reasons. First, the Bounding Box component will help us determine the height of our geometric pattern. Since we have only used rectangular boxes as our pattern, the height would be pretty easy to determine. However, if you had chosen a more organic shape to copy across your surface, the height would be much more difficult to resolve. We'll use this height information as an input value for our Surface Box component. Secondly, we'll use *the Bounding Box as a reference Brep for the BoxMorph component which we will discuss in a moment.*

- Right-click on the U-input of the Bounding Box component and set the boolean value to True.

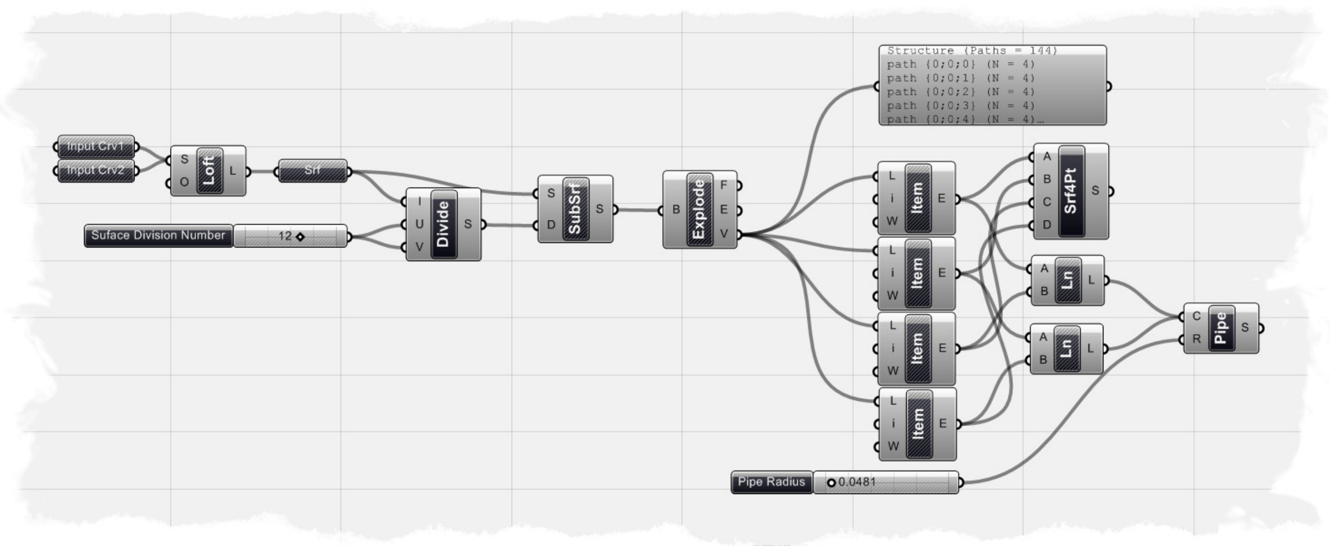
  This is an important step, as this will tell our bounding box to create a single box for all 3 Brep objects.
- Surface/Analysis/Box Components - Drag and drop a Box Components onto the canvas
- Connect the Bounding Box-B output to the Box Components-B input
- Connect the Box Components-Z output to the Surface Box-H input

  We're on the home stretch now.
- Xform/Morph/Box Morph - Drag and drop a Box Morph component onto the canvas
- Connect the Pattern Geometry output to the Box Morph-G input
- Connect the Bounding Box-B output to the Box Morph-R input
- Connect the Surface Box-B output to the Box Morph-T input
- Right-click on the Morph Box component and set the data matching algorithm to **Cross Reference**
- Right-click on the Surface Box component and turn the 'Preview' off

  Phew. If your brain isn't about to explode, I'll try to explain the last part of the definition. We've input the pattern geometry into the Morph Box component, which replicates that pattern over each subdivision. We've used the Bounding Box of our window system as our reference geometry, and we input the 100 box subdivisions as our target boxes to replicate our geometry. If you followed all of the steps correctly, you should be able to change your base surface, pattern geometry, and the U and V subdivisions to control any number of panels on a surface.

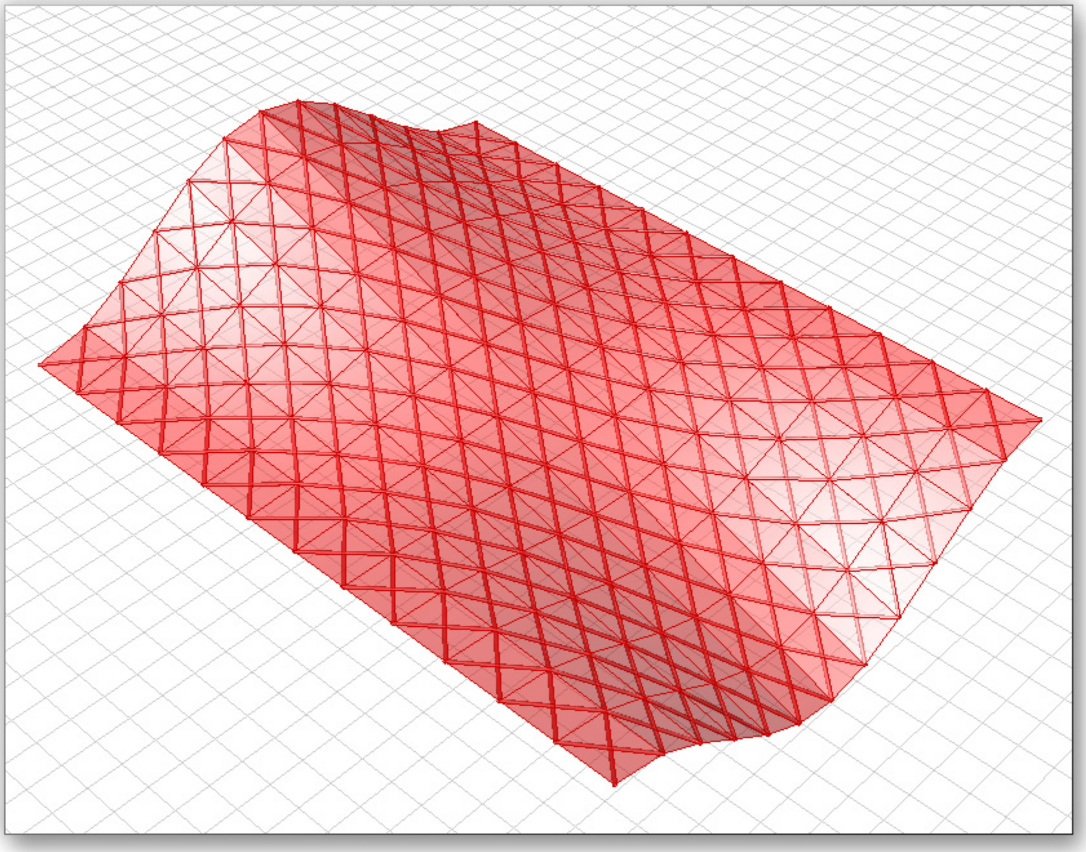
11.3 Surface Diagrid

We've already shown how we can use the Paneling Tools definition to create façade elements on a surface, but the next example will really show how we can manipulate the information flow to create a structural diamond grid, or diagrid on any surface. To begin, let's **Open** the **Surface Diagrid.3dm** file in Rhino. In the scene, you will find two mirrored cosine curves which will be the boundaries of our lofted surface.



```
Structure (Paths = 144)
path {0;0;0} (N = 4)
path {0;0;1} (N = 4)
path {0;0;2} (N = 4)
path {0;0;3} (N = 4)
path {0;0;4} (N = 4)…
```

Note: To see the finished definition of this example, in Grasshopper **Open** the file **Surface Diagrid.ghx** found in the Source Files folder that accompanies this document.



Let's start the definition from scratch:

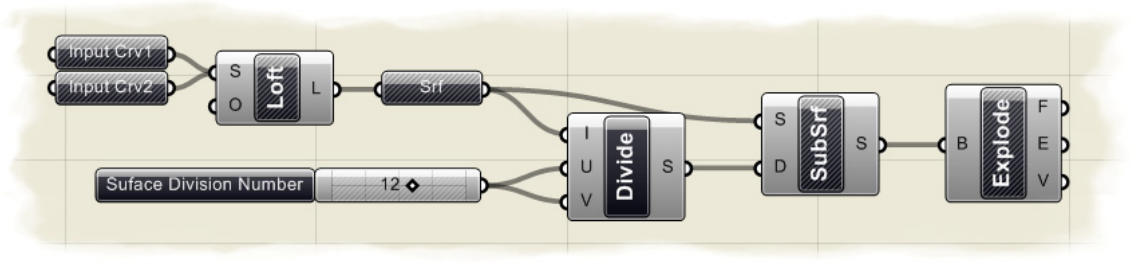
- Params/Geometry/Curve – Drag and drop **two** Curve components onto the canvas
- Right-click the first Curve component and rename it to "Input Crv1"
- Right-click the Input Crv1 component and select "Set One Curve"
- When prompted, select one of the curves in the Rhino scene
- Right-click the second Curve component and rename it to "Input Crv2"
- Right-click the Input Crv2 component and select "Set One Curve"
- When prompted, select the other curve in the Rhino scene
- Surface/Freeform/Loft – Drag and drop a Loft component onto the canvas
- Connect the Input Crv1 component to the Loft-S input
- While holding the Shift key, connect the Input Crv2 component to the Loft-S input

  You should now see a lofted surface between the two input curves inside your Rhino scene.
- Params/Geometry/Surface – Drag and drop a Surface component onto the canvas
- Connect the Loft-L output to the Surface component input
- Scalar/Interval/ Divide Interval^2 – Drag and drop a Divide Two Dimensional Interval component onto the canvas

  Just like in the last example, we are going to subdivide our surface into smaller surfaces. To do this, we must create an interval in both the U and V direction from which to subdivide our surface
- Connect the Surface component output to the Divide Interval-I input
- Params/Special/Numeric Slider – Drag and drop a slider onto the canvas
- Right-click the slider and set the following:
  - Name: Surface Division Number
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 20.0
  - Value: 12.0
- Connect the Slider to both the U and the V inputs of the Divide Interval component
- Surface/Utility/Isotrim – Drag and drop an Isotrim component onto the canvas
- Connect the Surface component output to the Isotrim-S input
- Connect the Divide Interval-S output to the Isotrim-D input
- Right-click the Loft and Surface component and turn the 'Preview' off

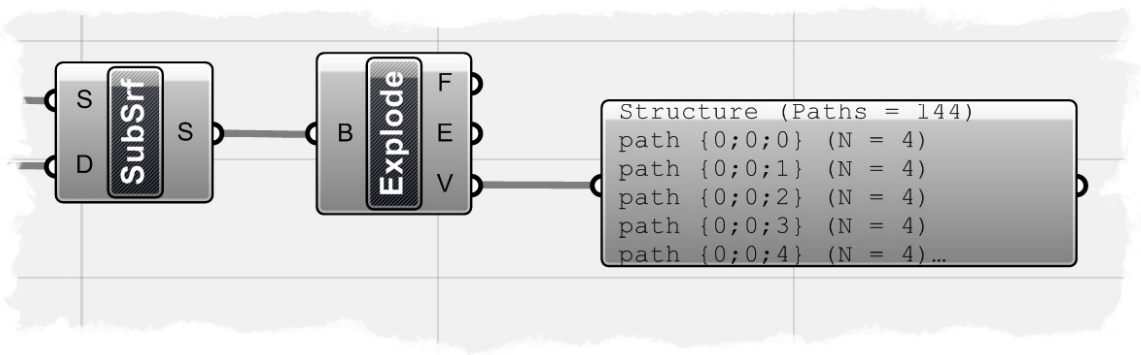
  You should now see a set of subdivided surface corresponding to the subdivision value you set in the numeric slider. Because we have connected only one slider to both of the U and V inputs for the interval, you should see the subdivisions change equally as you move the number slider to the right and left. You can add an additional slider here if you would like to control the divisions separately.
- Surface/Analysis/Brep Components – Drag and drop a Brep Components onto the canvas

  This component will break down a series of Breps into their component elements, such as Faces, Edges, and Vertices. In this case, we want to know the positions of each corner point so that we can begin to make diagonal connections between each subdivision.
- Connect the Isotrim-S output to the Brep Components-B input



Our definition, so far, should look like the image above. We have essentially subdivided our surface into smaller sub-surfaces and exploded each sub-surface to get the position of each surface corner point. In our next step, we will look at our tree structure an index each of the corner points.

- Param/Special/Parameter Viewer - Drag and drop a Parameter Viewer component onto the canvas
- Connect the Brep Components-V output to the Parameter Viewer input
  The Parameter Viewer will help us examine our tree structure. The structure for this particular example tells us that we have 144 paths, with each terminus branch containing 4 data entries (in our case, these are the four corner points of each subdivided surface). We can use the List Item component to go into each path, or sub-path and retrieve a specific data entry. To create our diagrid, we want to know where the first, second, third, and fourth point of each subdivided surface is in Cartesian space.



```
Structure (Paths = 144)
path {0;0;0} (N = 4)
path {0;0;1} (N = 4)
path {0;0;2} (N = 4)
path {0;0;3} (N = 4)
path {0;0;4} (N = 4)…
```

- Logic/List/List Item - Drag and drop **four** List Item component onto the canvas
- Connect the Brep Components-V output to all four of the List Item-L inputs
- Right-click on the i-input of the first List Item and Set the Integer to 0
- Right-click on the i-input of the second List Item and Set the Integer to 1
- Right-click on the i-input of the third List Item and Set the Integer to 2
- Right-click on the i-input of the fourth List Item and Set the Integer to 3
  You probably noticed that we increased the List Item index number by 1 each time. Since our tree structure has been setup and Grasshopper understands that we have 144 different paths, each with 4 item at the end of each path... all we need to do is go into each path and retrieve the first item, and then retrieve the second point, and so on and so forth. So the 4 List Item components are going to define each corner point of the sub-

surfaces, and now that we have them separated, we can connect opposing points with a line to create our diagrid.

- Curve/Primitive/Line - Drag and drop **two** Line components onto the canvas
- Connect the first List Item-E output to the first Line-A input
- Connect the third List Item-E output to the first Line-B input
- Connect the second List Item-E output to the second Line-A input
- Connect the fourth List Item-E output to the second Line-B input

At this point, you should see a set of lines on your surface that will represent the diagrid structure.

- Surface/Freeform/Pipe - Drag and drop a Pipe component onto the canvas
- Connect the first Line-L output to the Pipe-C input
- While holding the Shift key down, connect the second Line-L output to the Pipe-C input

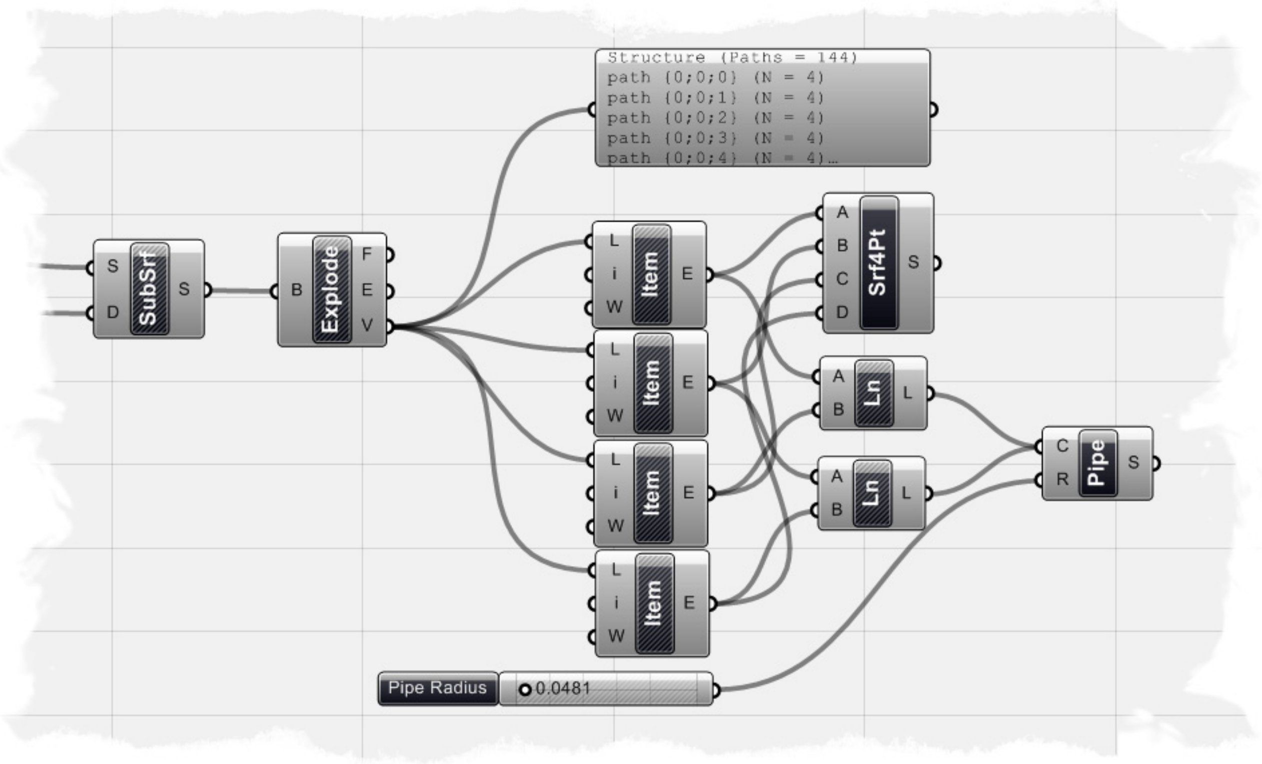
To create a variable structural thickness we will use a slider to control the Pipe radius

- Params/Special/Number Slider – Drag and drop a numeric slider onto the canvas
- Right-click on the slider and set the following:
    - Name: Pipe Radius
    - Slider Type: Floating Point
    - Lower Limit: 0.0
    - Upper Limit: 1.0
    - Value: 0.05
- Connect the Pipe Radius slider to the Pipe-R input

Lastly, we will create a flat surface between each of the 4 corner points. We are doing this, because the sub-surfaces we originally created are curved and for this exercise we would like to create a faceted surface with a diagrid structural system.

- Surface/Freeform/4Point Surface – Drag and drop a 4 Point Surface component onto the canvas
- Connect the first List Item-E output to the 4 Point Surface-A input
- Connect the second List Item-E output to the 4 Point Surface-B input
- Connect the third List Item-E output to the 4 Point Surface-C input
- Connect the fourth List Item-E output to the 4 Point Surface-D input

Make sure you turn the Preview off for all components except for the 4 Point Surface component and the Pipe component.



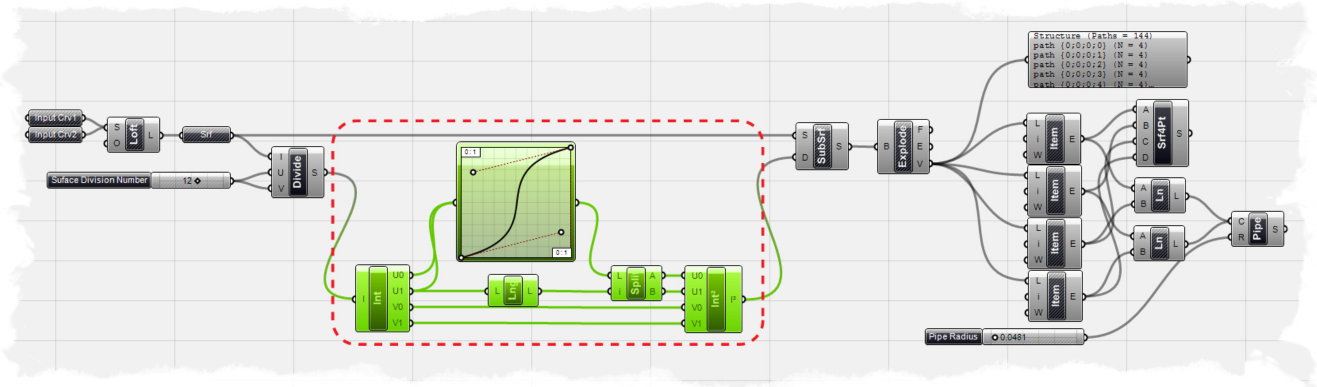
```
Structure (Paths = 144)
path {0;0;0} (N = 4)
path {0;0;1} (N = 4)
path {0;0;2} (N = 4)
path {0;0;3} (N = 4)
path {0;0;4} (N = 4)…
```

The last part of your definition should look like the image above. This definition will work on any type of single surface and you can replace the Loft part of the definition at the beginning with more complicated surface generation methods.

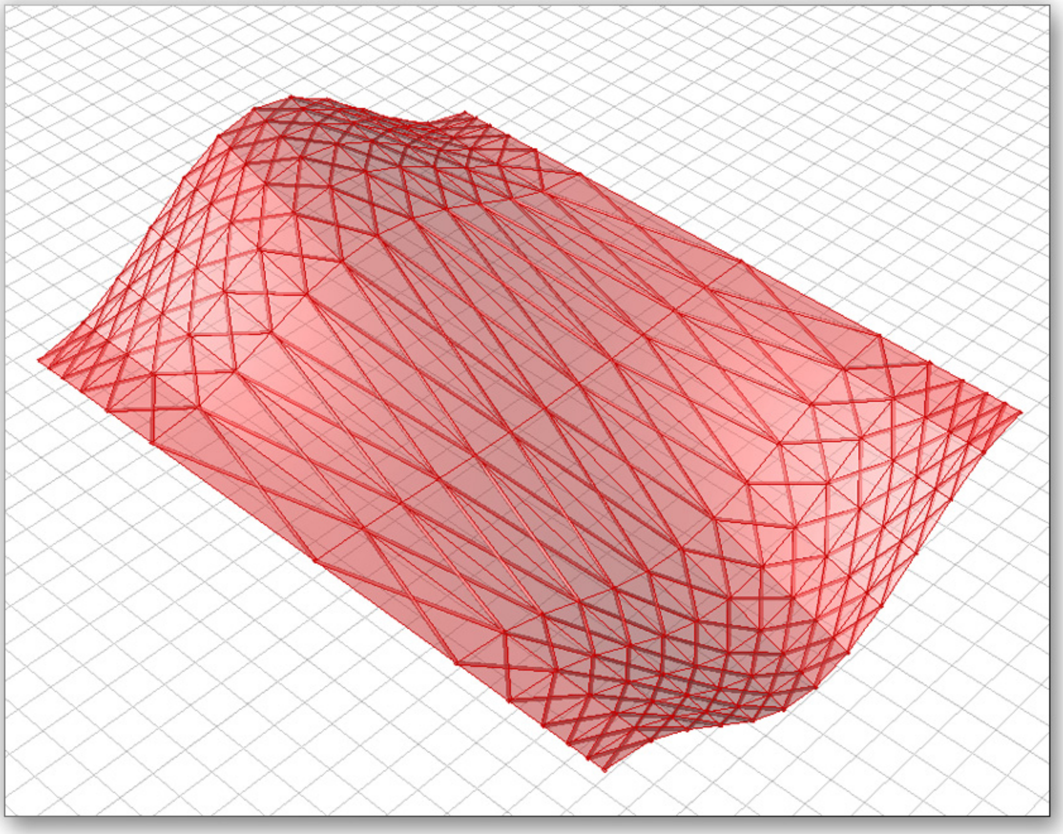
11.4 Uneven Surface Diagrid

We showed in the last example how we can subdivide a surface to create a uniformly spaced diagrid structure. The diagrid was uniformly spaced because we created an evenly spaced interval... however we can use a few new components and a Graph Mapper to control our interval collection, and also the diagrid spacing. This tutorial will build off of the previous one, so I am assuming that you have already built the evenly spaced diagrid definition. Below is a screenshot of the finished definition for the unevenly spaced diagrid, and the components we are going to add to the existing definition are shown in green.

Note: To see the finished definition of this example, in Grasshopper **Open** the file **Uneven Surface Diagrid.ghx** found in the Source Files folder that accompanies this document.

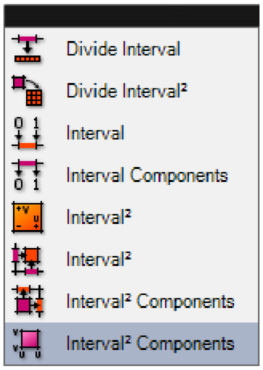


```
Structure (Paths = 144)
path {0;0;0;0} (N = 4)
path {0;0;0;1} (N = 4)
path {0;0;0;2} (N = 4)
path {0;0;0;3} (N = 4)
path {0;0;0;4} (N = 4)…
```



In the last definition, we connected the Divide Interval component to the Isotrim component. We will start our definition by disconnecting that connection.

- Right-click on the Isotrim-D input and select "Disconnect All"
  Because we are going to be inserting a number of new components, its probably a good idea to select everything downstream of the Isotrim component and move it farther to the right side of the canvas.
- Scalar/Interval/Interval Components - Drag and drop an Interval Components (the one that decomposes the interval into 4 numbers) onto the canvas

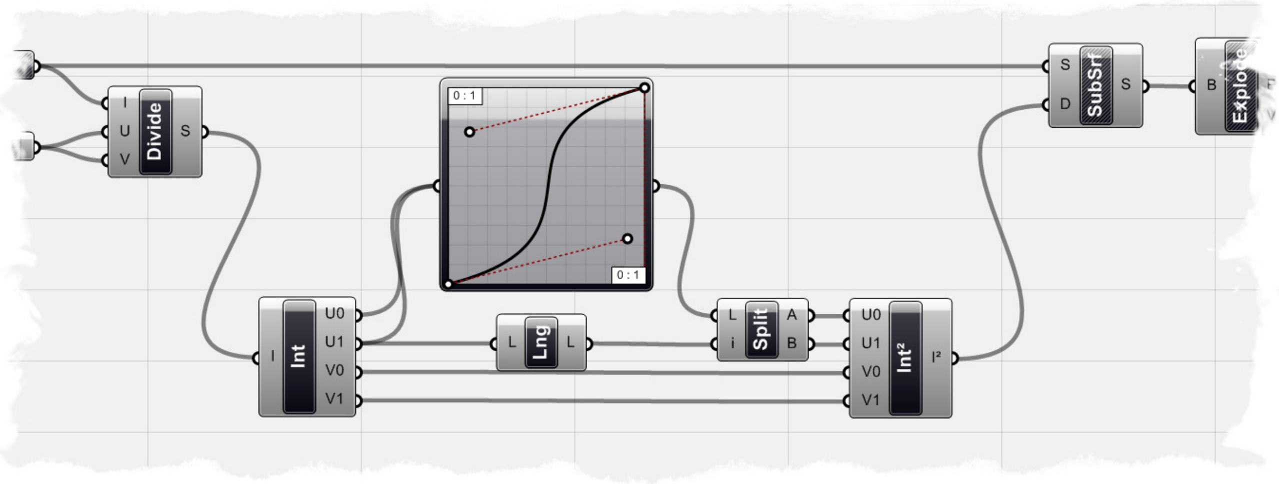


- Connect the Divide Interval-S output to the Interval Components-I input
- Params/Special/Graph Mapper - Drag and drop a Graph Mapper component onto the canvas
- Connect the U0-output to the input of the Graph Mapper
- Holding the shift key, connect the U1-output to the input of the Graph Mapper
- Right-click on the Graph Mapper and choose a graph type
  I think the bezier graph type works best in this situation, but to each their own. You can adjust the graph by moving the bezier handles around the Graph Mapper component. Your probably asking yourself why we're doing all of this. Well, essentially what we've done is decompose our evenly spaced interval into a list where we can access each of our U and V values. We've fed our U list data into the Graph Mapper which will reorganize the list according to a specific graph type. Ultimately we're going to take this reorganized data and reassemble it back into a new interval that we can feed back into the Isotrim component.
- Logic/List/List Length - Drag and drop a List Length component onto the canvas
- Connect the U1-output to the List Length-L input
- Logic/List/Split List - Drag and Drop a Split List component onto the canvas
- Connect the output of the Graph Mapper to the L-input of the Split List component
- Connect the List Length-L output to the Split List-i input
- Scalar/Interval/Interval 2d - Drag and drop a two dimensional Interval component onto the canvas
- Connect the Split List-A output to the U0-input of the two dimensional Interval component
- Connect the Split List-B output to the U1-input of the two dimensional Interval component
- Connect the V0 and V1 outputs of the decomposed interval to the V0 and V1 inputs (respectively) of the two dimensional Interval component
  Since we connect two lists into the input of the Graph Mapper, we needed to split the list in two, so that we could feed the correct list data back into the two dimensional interval. We've connect all of the U values into the Graph Mapper, so our U subdivision will be controlled by the bezier curve. Since we've connect the V0 and V1 directly from the decomposed interval to the two dimensional interval, no change will happen in that direction. However, you could follow the same steps and connect another Graph

Mapper component to the V0 and V1 lists and control the V subdivisions in the same manner. Now, all we need to do is to connect our newly created interval collection to the Isotrim component.

- Connect the two dimensional Interval-I^2 output to the Isotrim-D input

  Since the end portion of the definition is the same, our graph type will now control the spacing of our subdivisions; subsequently driving the diagrid. You can adjust the bezier curve to increase the amount of structure needed to support your surface. The middle portion of your definition should now look like the image below.



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.

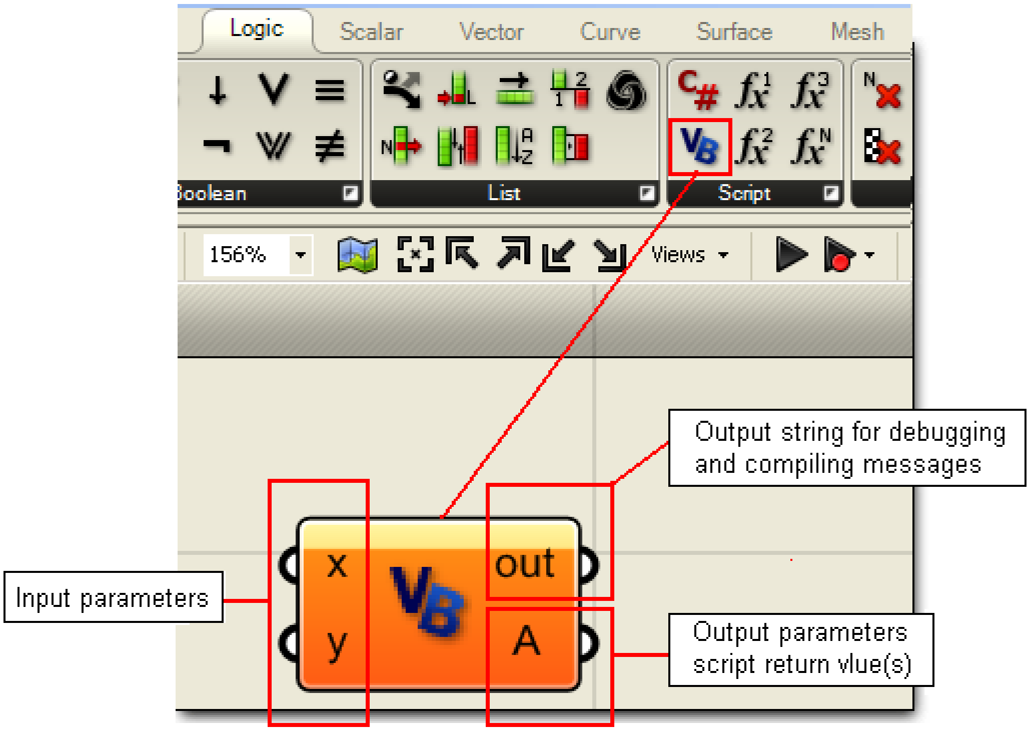
This Primer gives a general overview of how to use the scripting component in Grasshopper using VB DotNET programming language. It includes three sections. The first is about the script component interface. The second includes a quick review of VB DotNET language. The next section talks about Rhino DotNET 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.


Logic
Scalar
Vector
Curve
Surface
Mesh
Boolean
List
Script
156%
Views
Output string for debugging and compiling messages
Input parameters
x
y
out
A
Output parameters script return vlue(s)

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

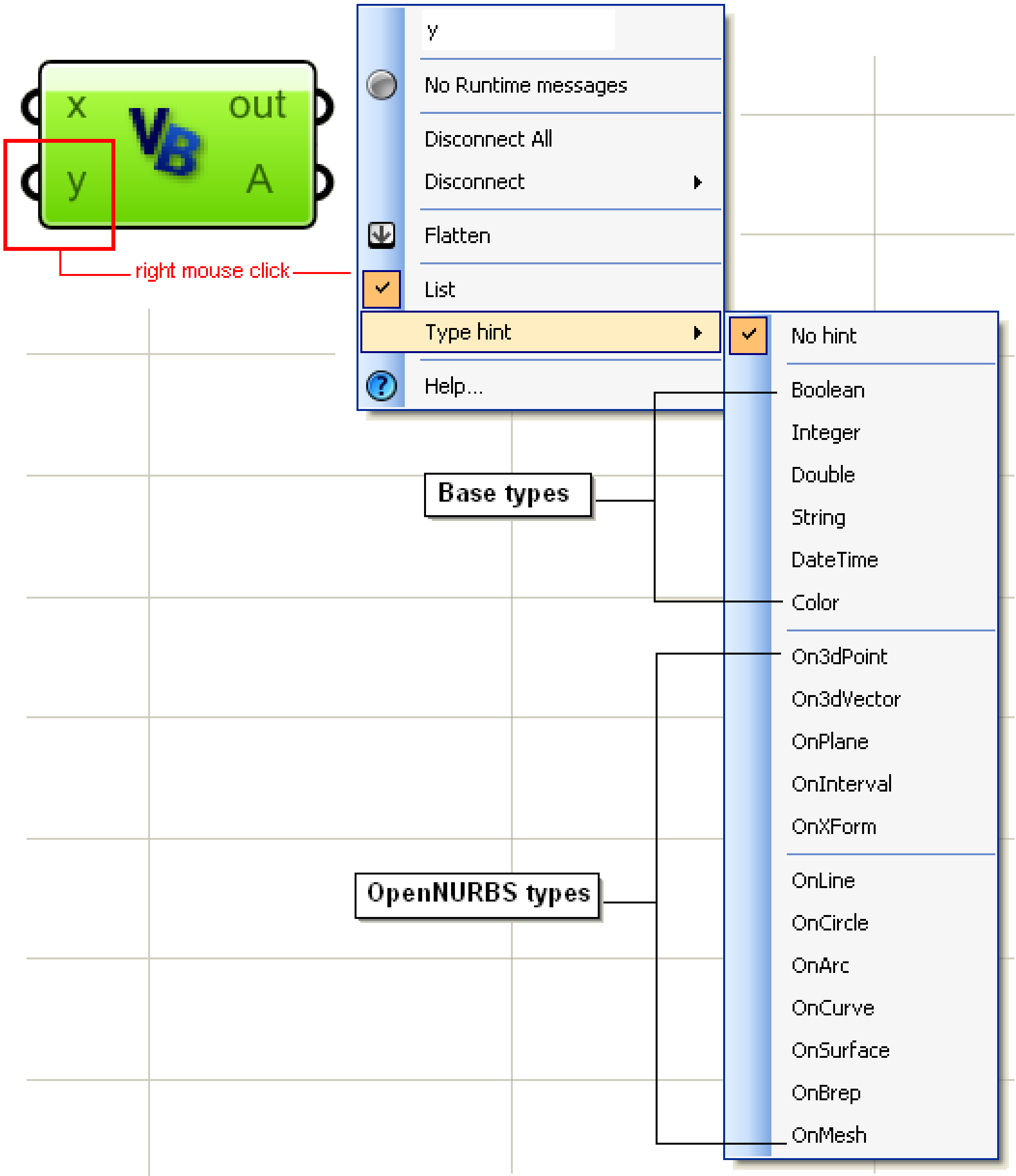
- **X**: first input of a generic type (object).
- **Y**: second input of a generic type (object).
- **Out**: output string with compiling messages.
- **A**: Returned output of type object.

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.

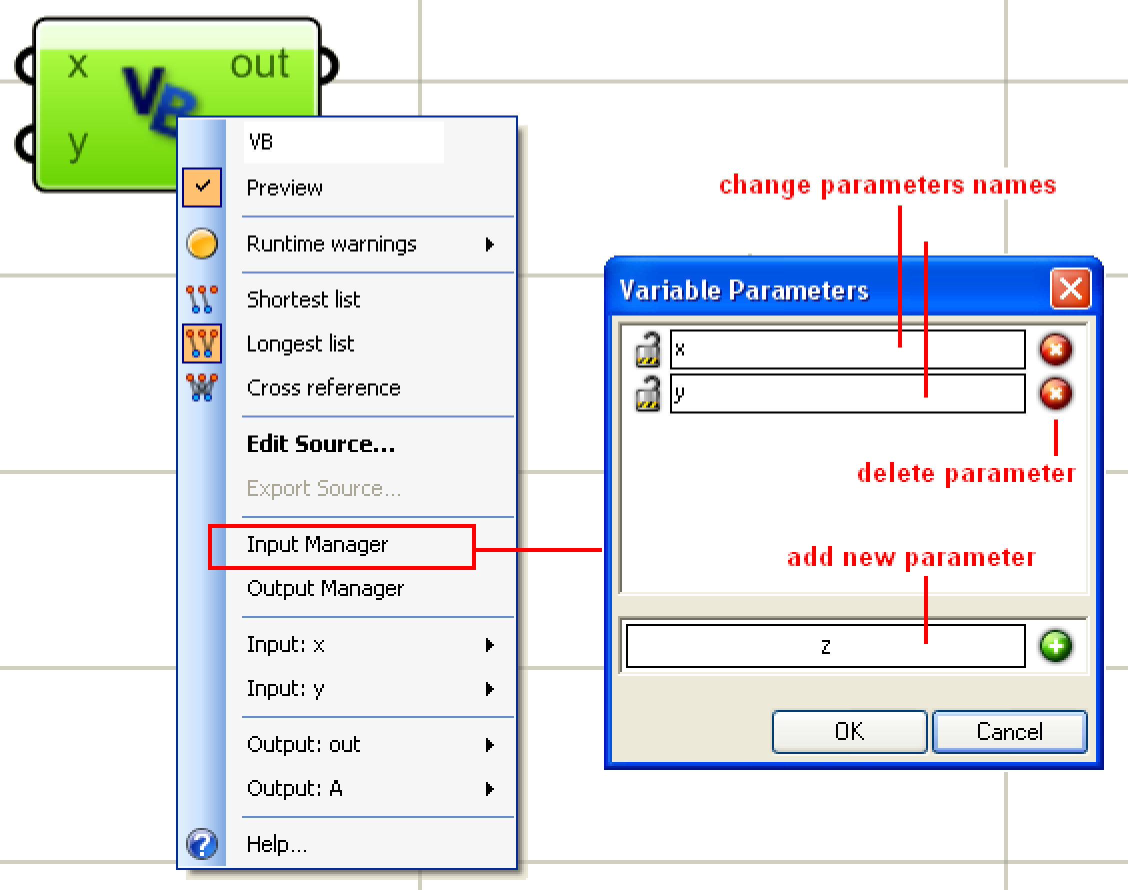
- **Run time message**: for errors and warnings.
- **Disconnect** and **Disconnect All**. Work the same as other Grasshopper components.
- **Flatten**: to flatten data. In case of a nested list of data, it converts it to single array of elements.
- **List**: to indicate if the input is a list of data.
- **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.

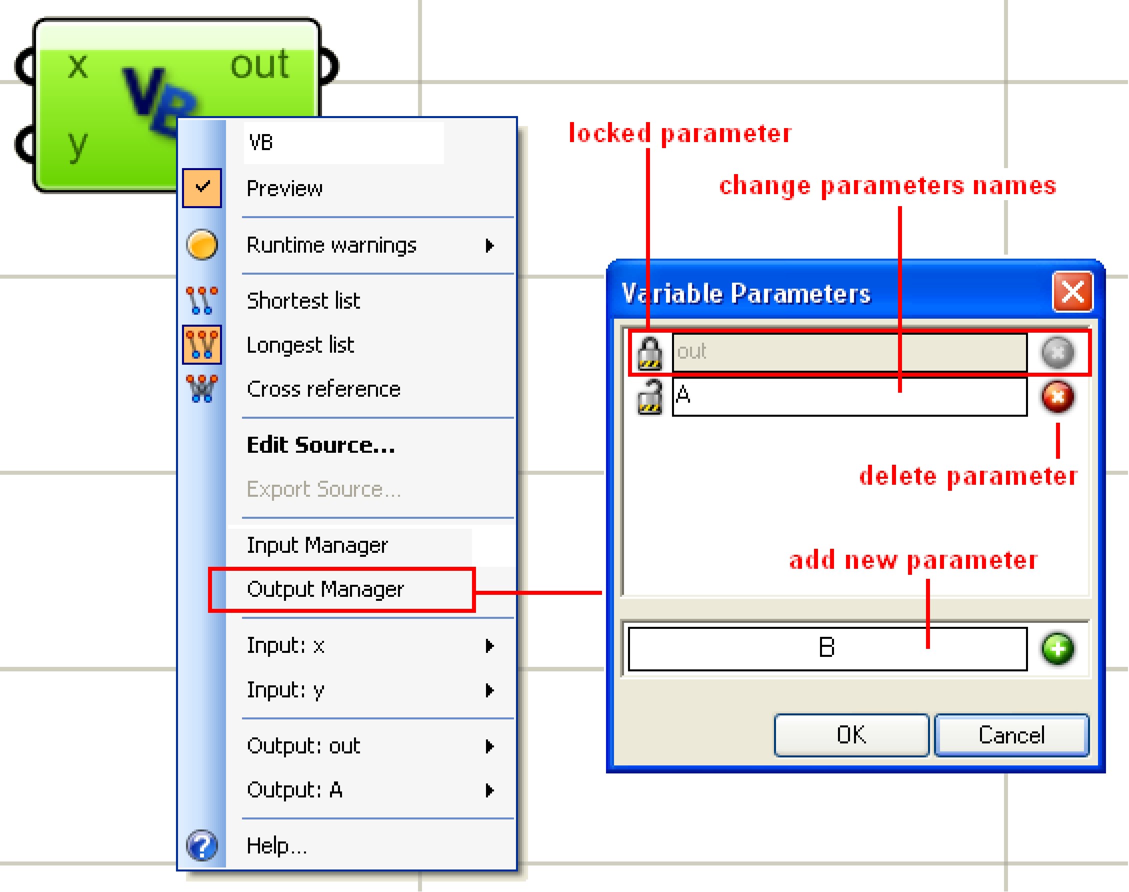
You may use this menu to open input manager and change parameters names, add new ones or delete as shown in the image.

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.



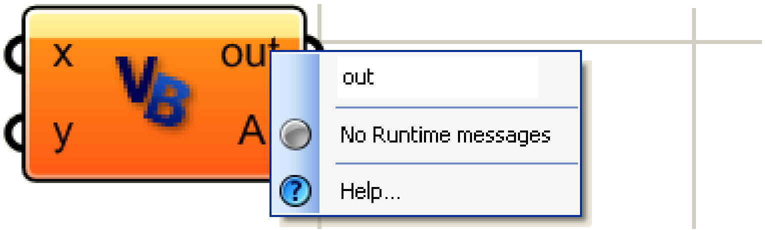
13.3 Output Parameters

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.

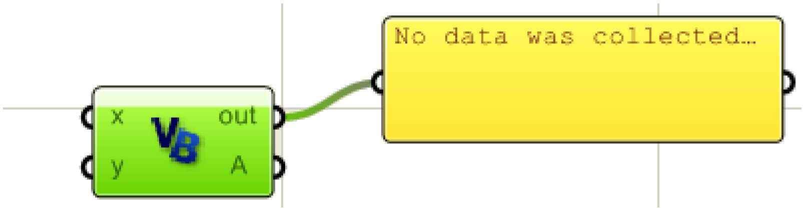


13.4 Out Window and Debug Information

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.

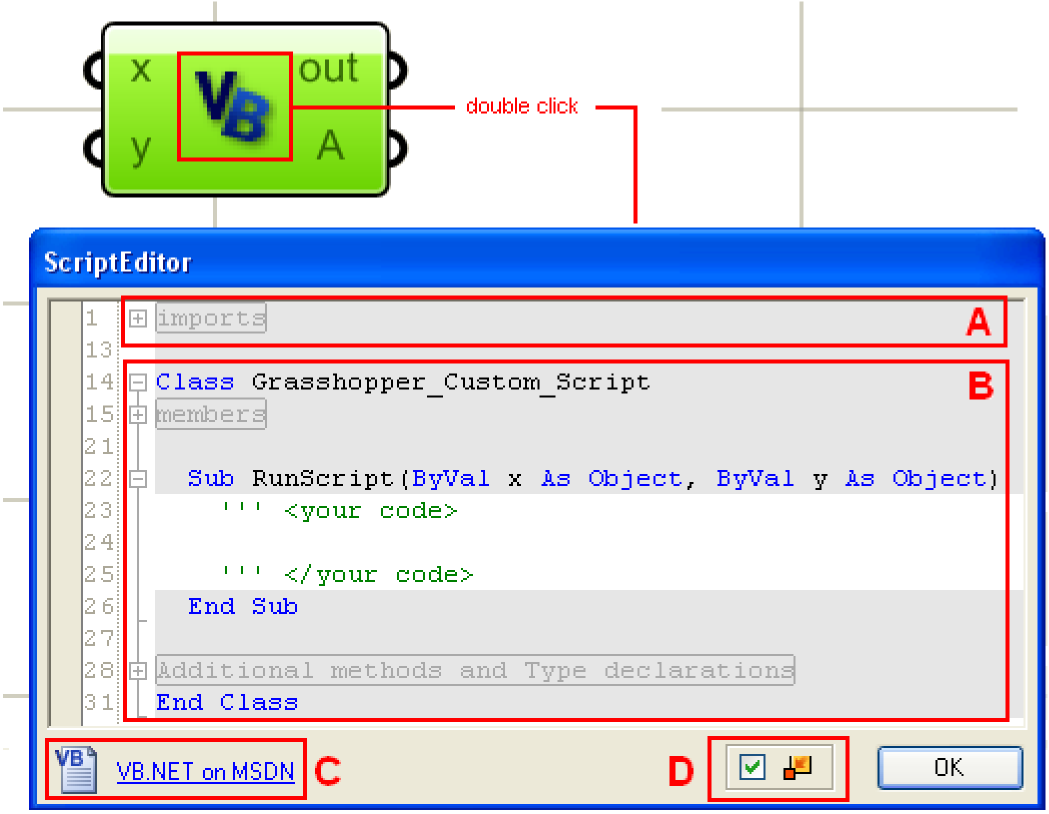


```
No data was collected…
```

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:

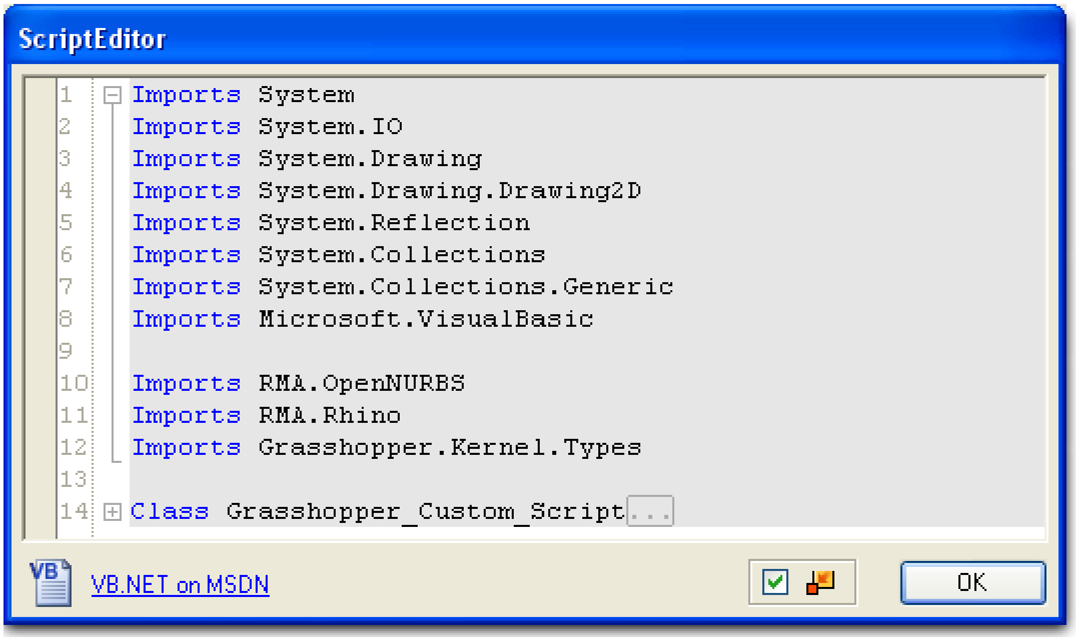
A: Imports
B: Grasshopper_Custom_Script class.
C: Link to Microsoft developer network help on VB.NET.
D: Check box to activate the out of focus feature of scripting component.



```
imports
Class Grasshopper_Custom_Script
members

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

    ''' </your code>
  End Sub

Additional methods and Type declarations
End Class
```

A: Imports

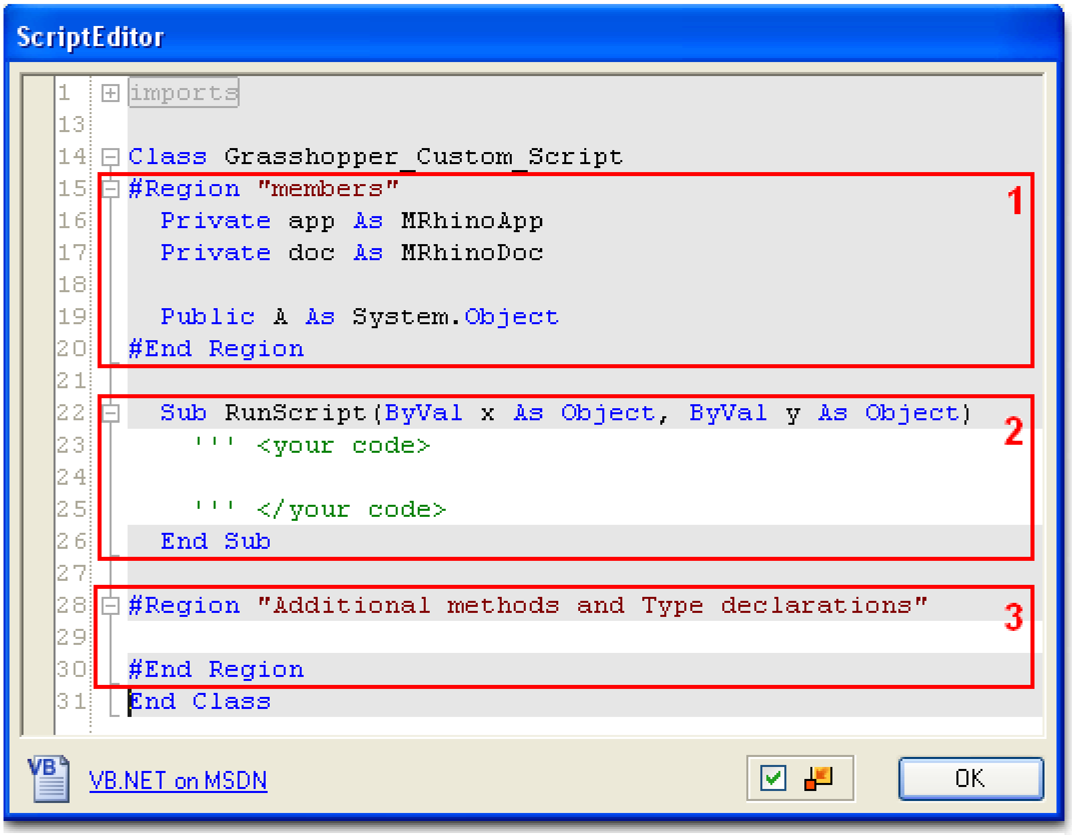
Imports are external dlls that you might use in your code. Most of them are DotNET system imports, but there is also the two Rhino dlls: RMA.openNURBS and RMA.Rhino. These include all rhino geometry and utility functions. There are also the Grasshopper specific types.



```
Imports System
Imports System.IO
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Reflection
Imports System.Collections
Imports System.Collections.Generic
Imports Microsoft.VisualBasic

Imports RMA.OpenNURBS
Imports RMA.Rhino
Imports Grasshopper.Kernel.Types

Class Grasshopper_Custom_Script...
```

B: Grasshopper_Custom_Script Class

Grasshopper_Custom_Script class consists of three parts:



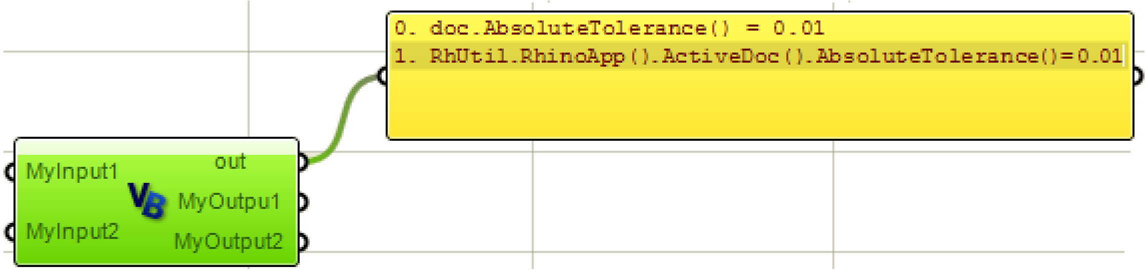
```
imports

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

  Public A As System.Object
#End Region

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

    ''' </your code>
  End Sub

#Region "Additional methods and Type declarations"

#End Region
End Class
```

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.

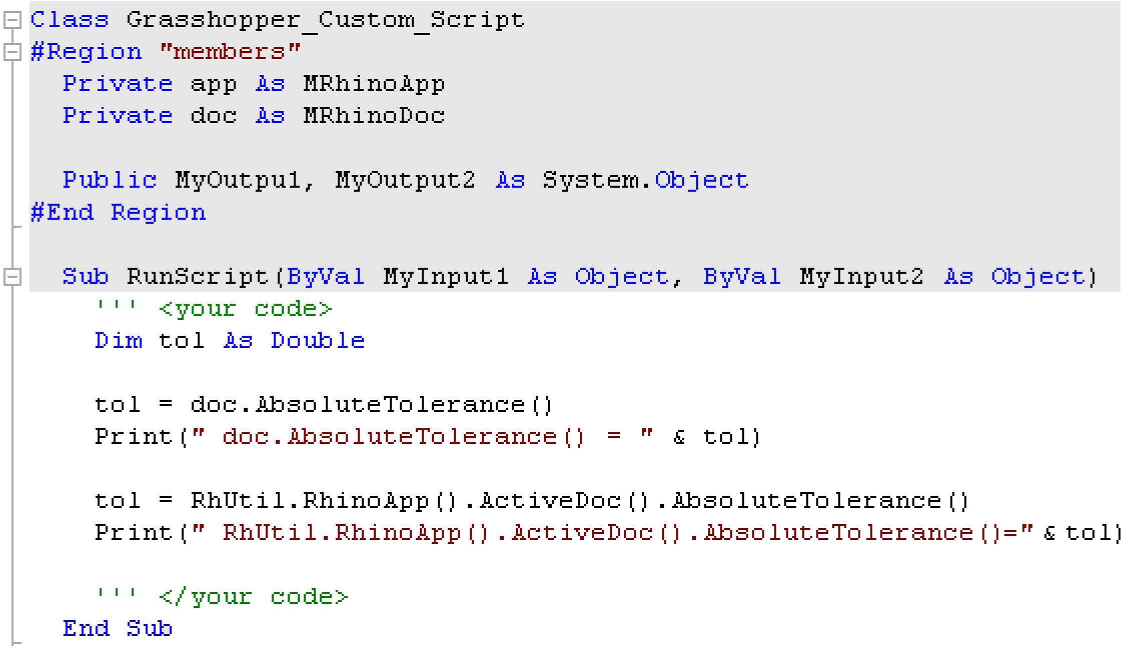
2. **RunScript**: This is the main function that the user writes their code within.

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.



```
0. doc.AbsoluteTolerance() = 0.01
1. RhUtil.RhinoApp().ActiveDoc().AbsoluteTolerance()=0.01
```



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

  Public MyOutpu1, 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
```

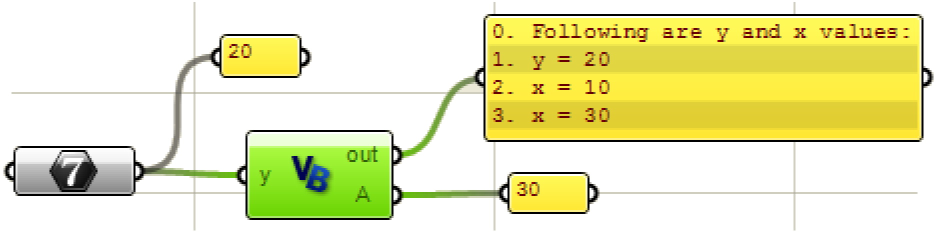
Here are other examples of commonly used types:

```
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
```

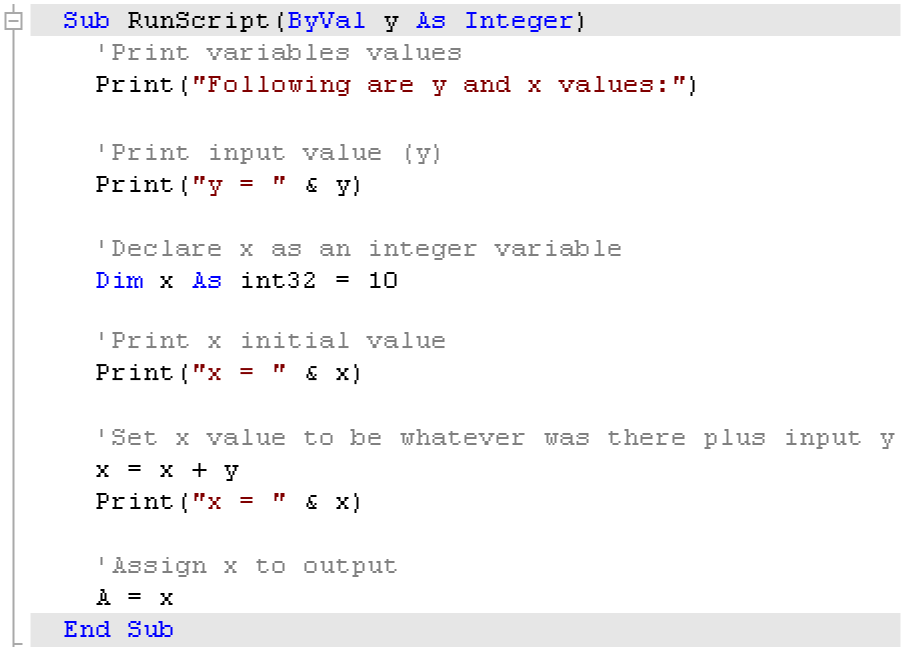
The following Grasshopper example uses three variables:

- **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.



```
0. Following are y and x values:
1. y = 20
2. x = 10
3. x = 30
```



```
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:

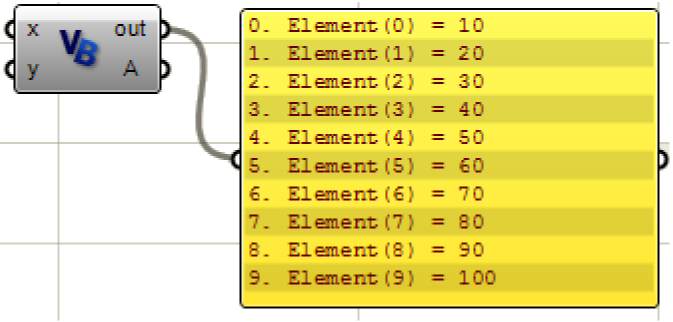
```
 'One dimensional array
Dim myArray(1) As Integer
myArray (0) = 10
myArray (1) = 20

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

```
'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 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.

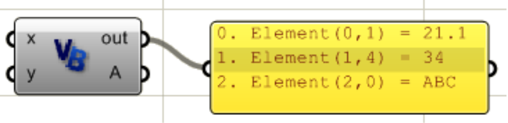
```
Dim myList As New List(Of Integer)
For i As Integer = 1 To 10
  myList.Add(10 * i)
  Print("Element(" & i - 1 & ") = " & myList(i - 1))
Next
```



```
0. Element(0) = 10
1. Element(1) = 20
2. Element(2) = 30
3. Element(3) = 40
4. Element(4) = 50
5. Element(5) = 60
6. Element(6) = 70
7. Element(7) = 80
8. Element(8) = 90
9. Element(9) = 100
```

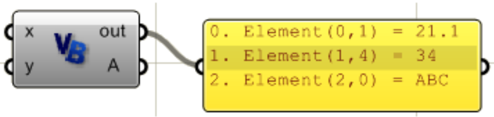
You can use nested List or ArrayList to declare dynamic multi-dimensional arrays of same or mixed types. Check the following example:

```
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"}
```



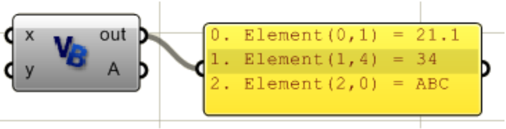
```
0. Element(0,1) = 21.1
1. Element(1,4) = 34
2. Element(2,0) = ABC
```

```
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")
```



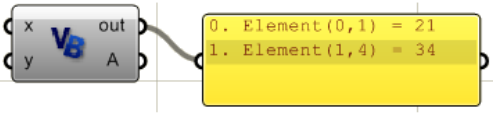
```
0. Element(0,1) = 21.1
1. Element(1,4) = 34
2. Element(2,0) = ABC
```

```
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")
```



```
0. Element(0,1) = 21.1
1. Element(1,4) = 34
2. Element(2,0) = ABC
```

```
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)
```



```
0. Element(0,1) = 21
1. Element(1,4) = 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:

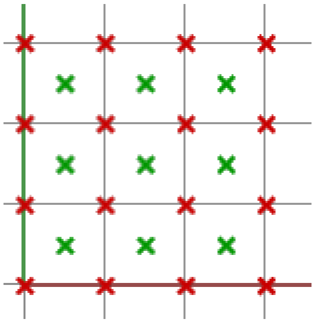
```
While < some condition is True >

    ‘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 ]
```

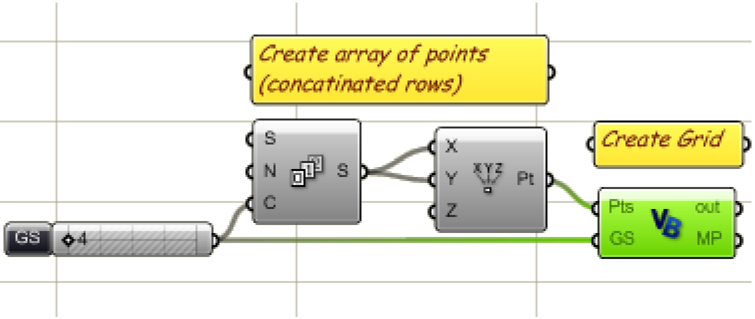
Here is the previous places example re-written using while loop:

```
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.

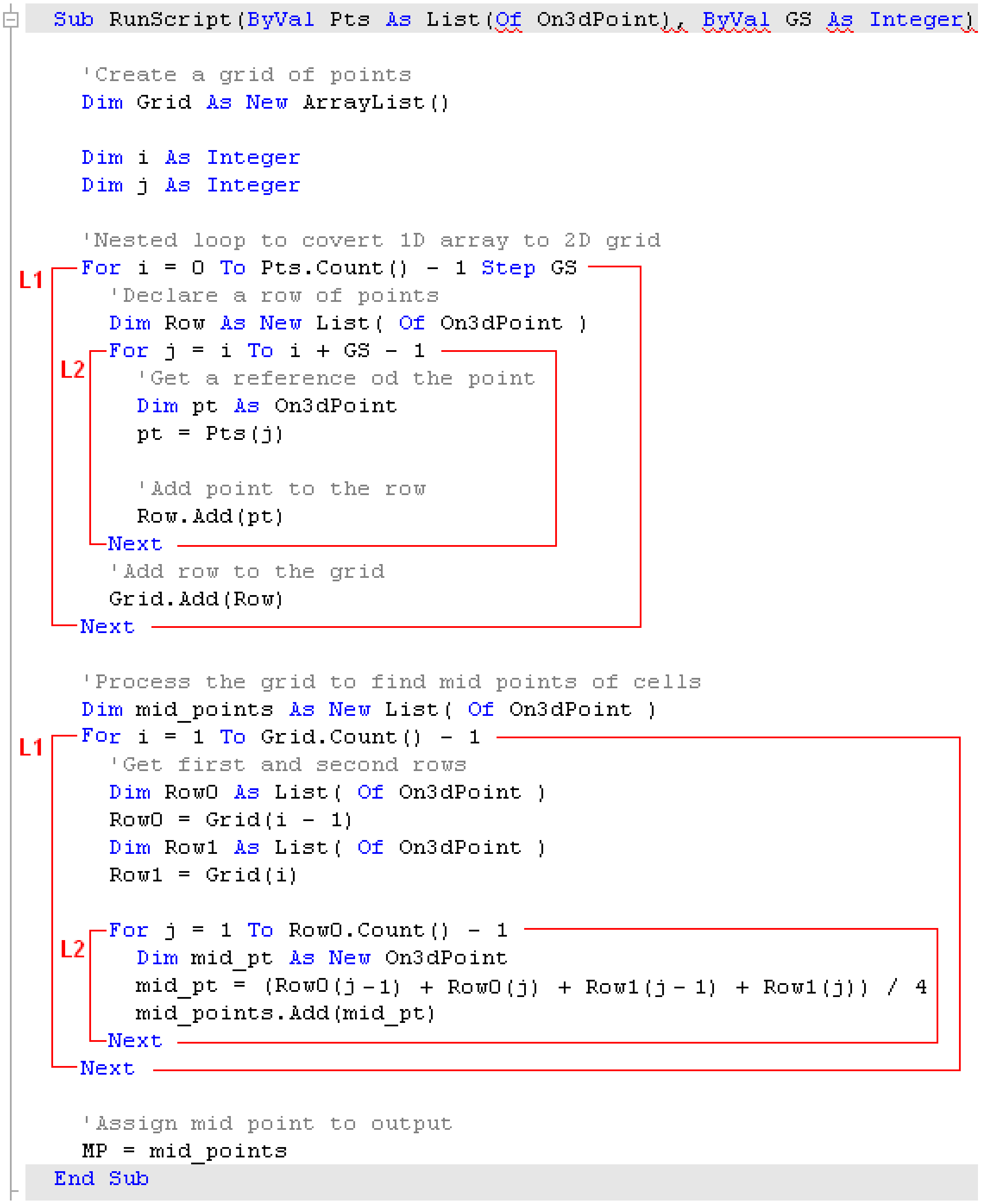




The script has two parts:

- 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
    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

    'Process the grid to find mid points of cells
    Dim mid_points As New List( Of On3dPoint )
    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

    '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
```

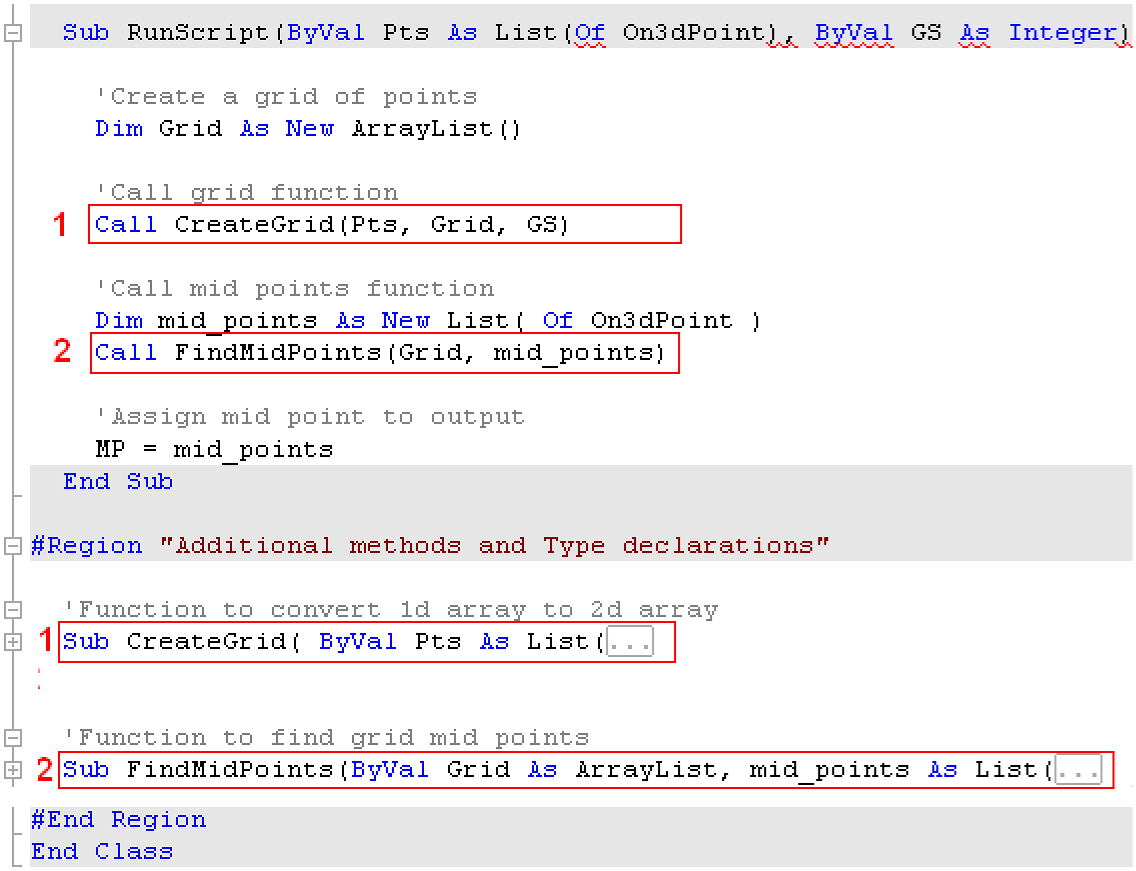
That is said, you don’t have to have a function to return a value. Subs can do it through input parameters that are passed “ByRef”. In the following, “**rc**” is used to return result:

```
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::

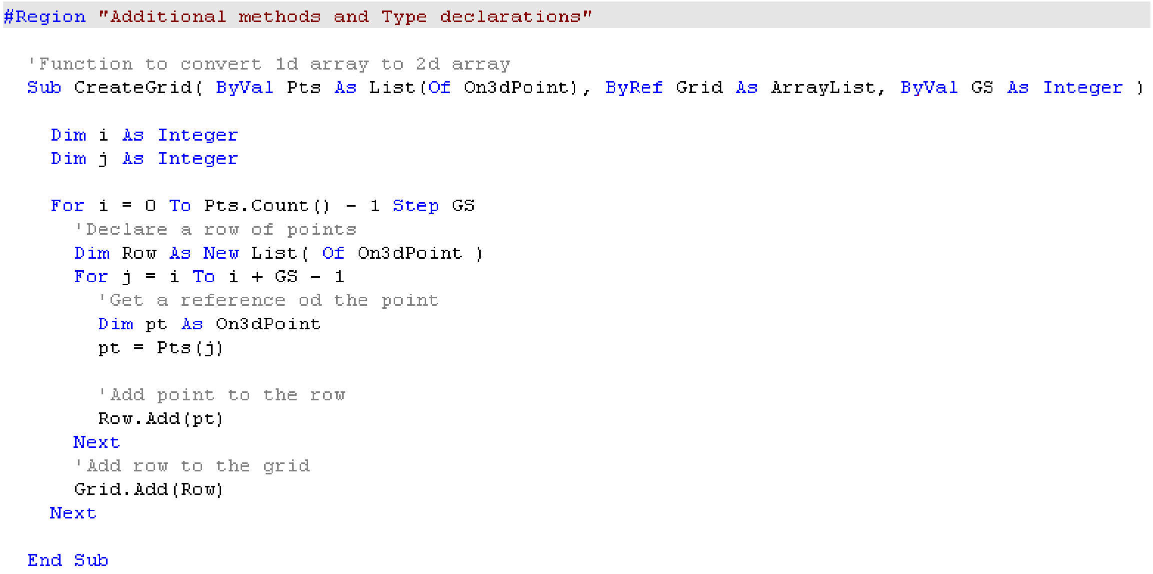
```
Dim x As Double = 5.34
Dim y As Double = 3.20
Dim rc As Double = 0.0
‘Can use either of the following to get result
rc = AddFunction( x, y )      ‘Assign function result to “rc”
AddSub( x, y, rc )            ‘rc is passed by refernce and will have addition result
```

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.

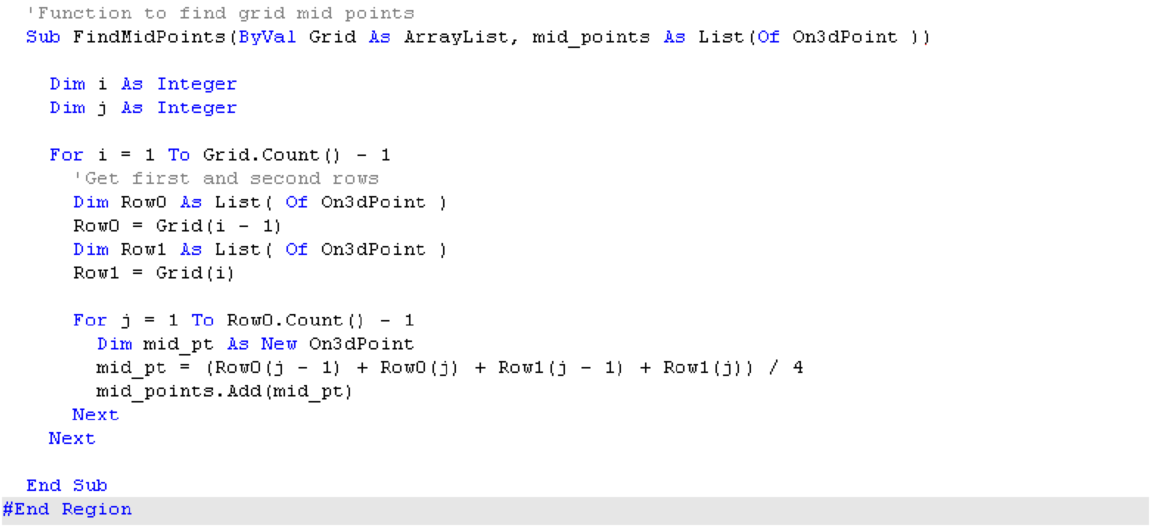


```
 Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)

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

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

    'Call mid points function
    Dim mid_points As New List( Of On3dPoint )
    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
  Sub CreateGrid( ByVal Pts As List(...

  'Function to find grid mid points
  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.

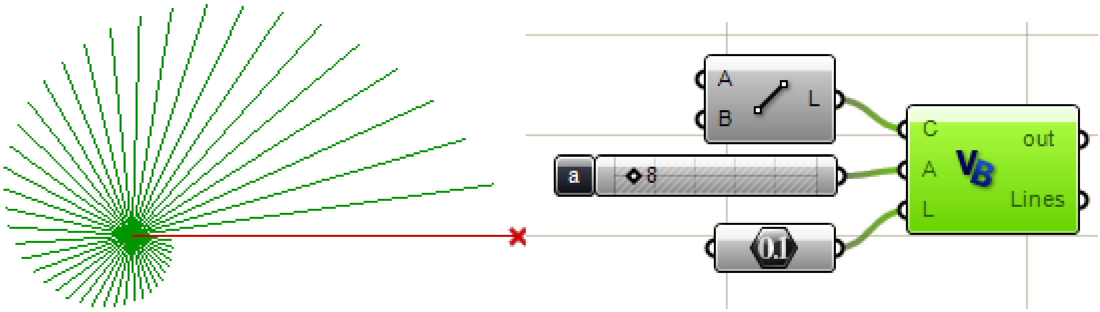
The following example takes smaller part of an input line and rotates it by a given angle. It keeps doing it until line length becomes less than a minimum length.

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:

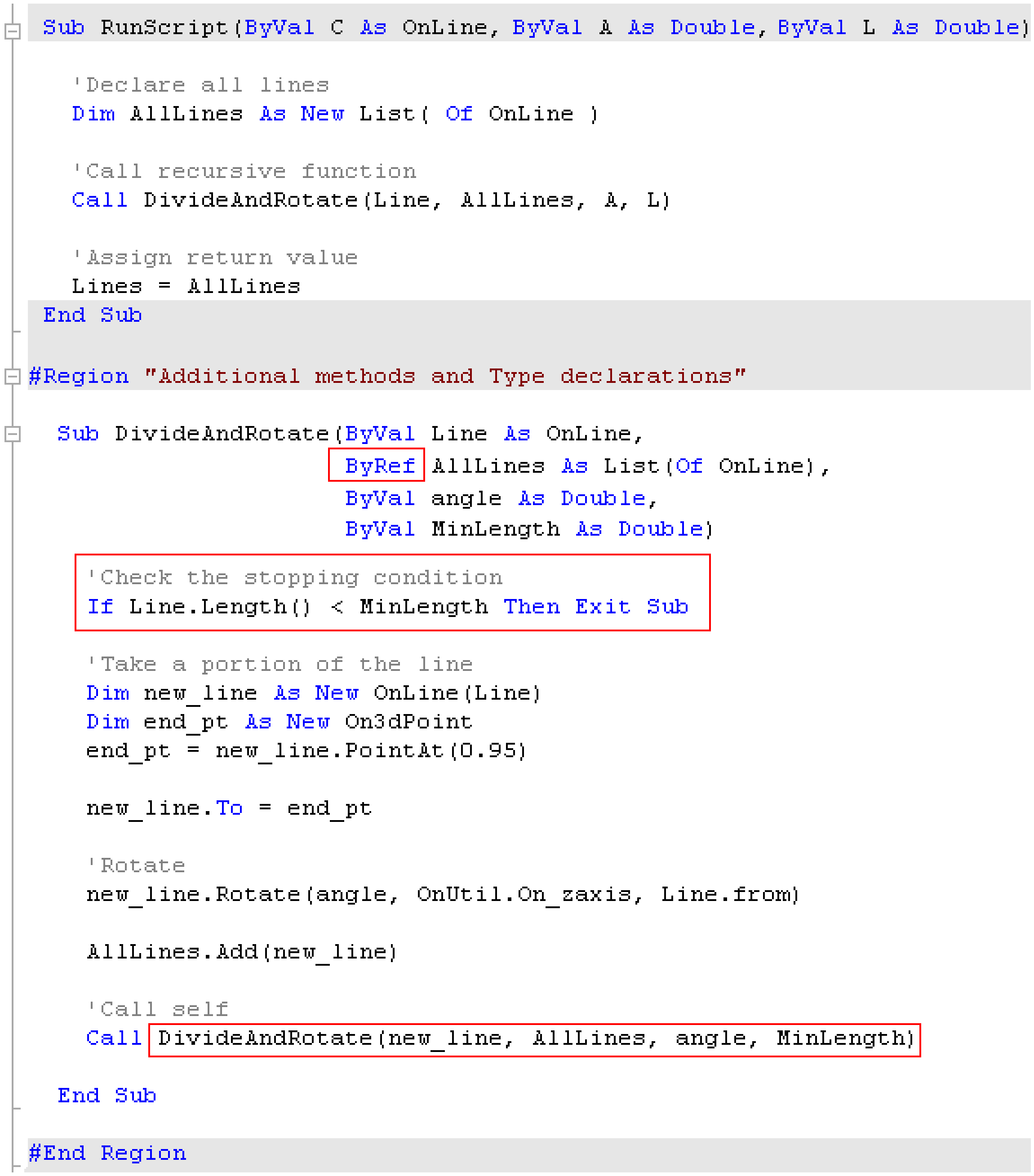
- Array of lines.



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

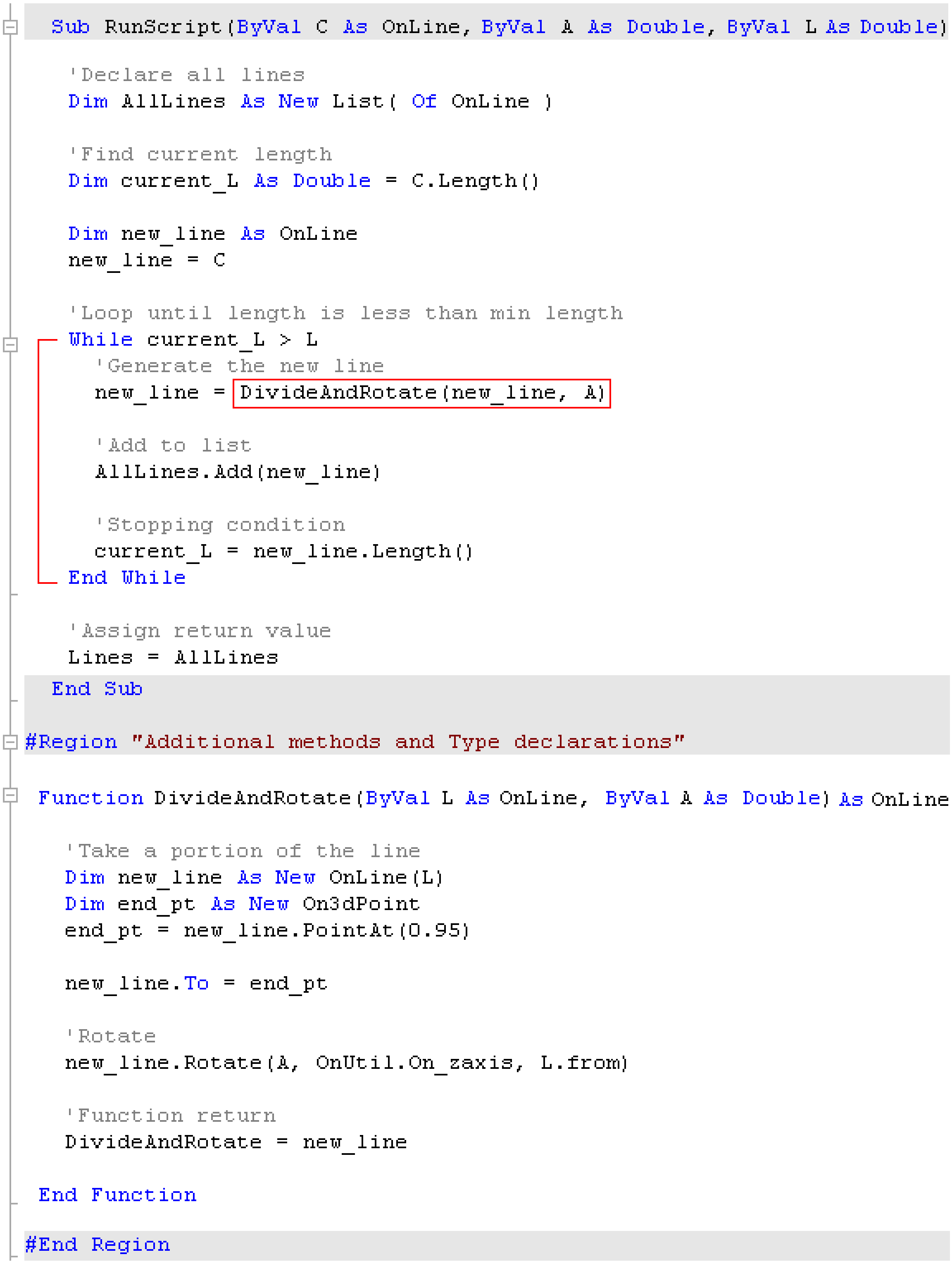
Recursive solution. Note that inside the “DivideAndRotate” sub there are:

- 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
```

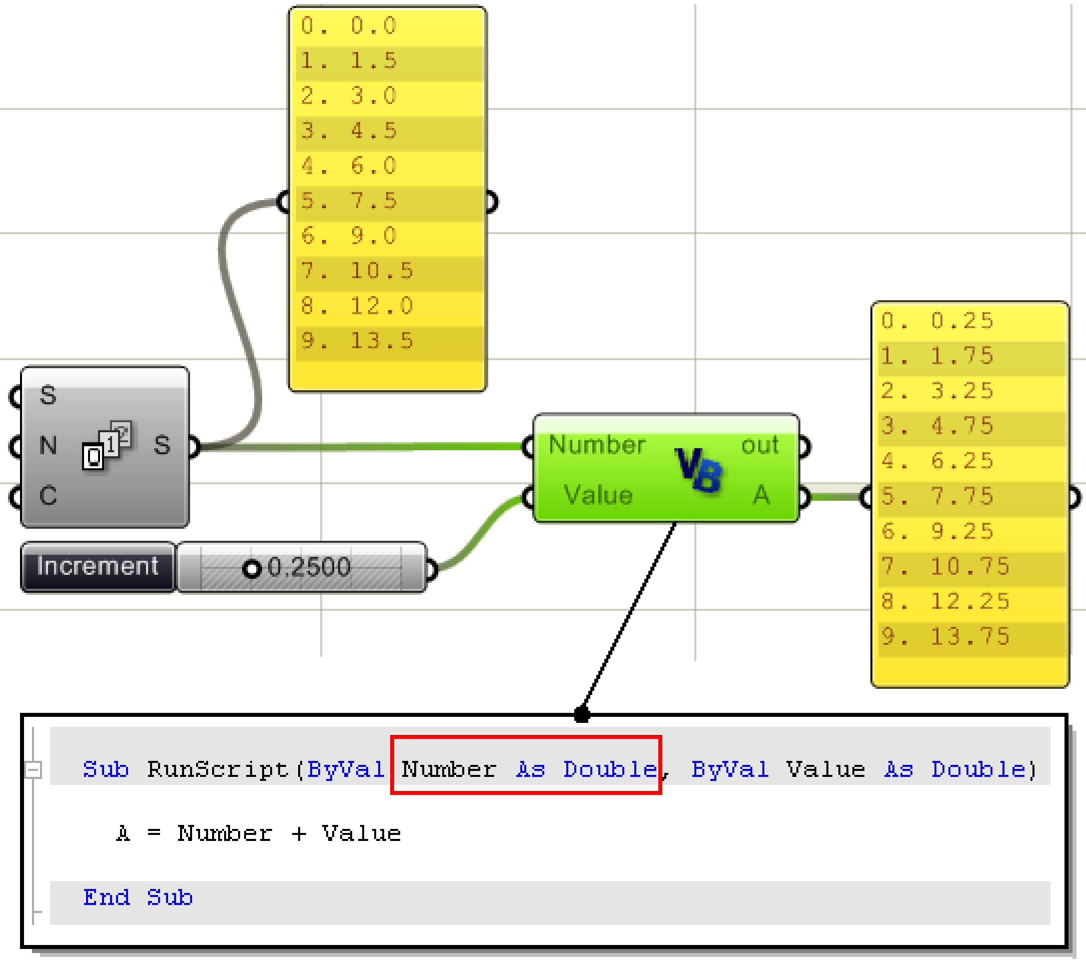
14.11 Processing Lists in Grasshopper

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.

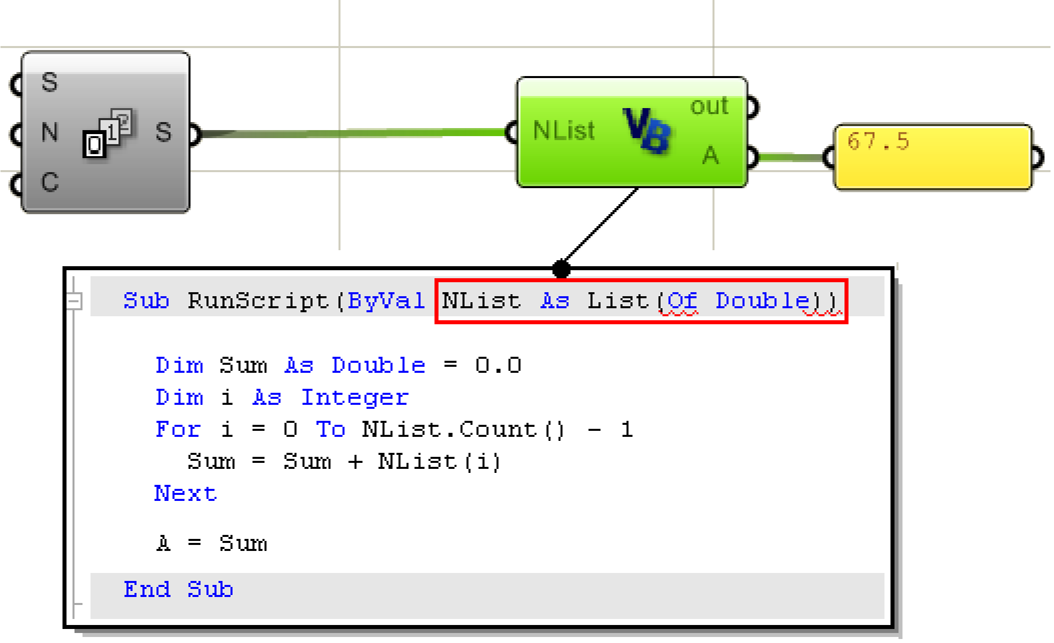


```
0. 0.0
1. 1.5
2. 3.0
3. 4.5
4. 6.0
5. 7.5
6. 9.0
7. 10.5
8. 12.0
9. 13.5
```

```
0. 0.25
1. 1.75
2. 3.25
3. 4.75
4. 6.25
5. 7.75
6. 9.25
7. 10.75
8. 12.25
9. 13.75
```

```
Sub RunScript(ByVal Number As Double, ByVal Value As Double)
  A = Number + Value
End Sub
```

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.

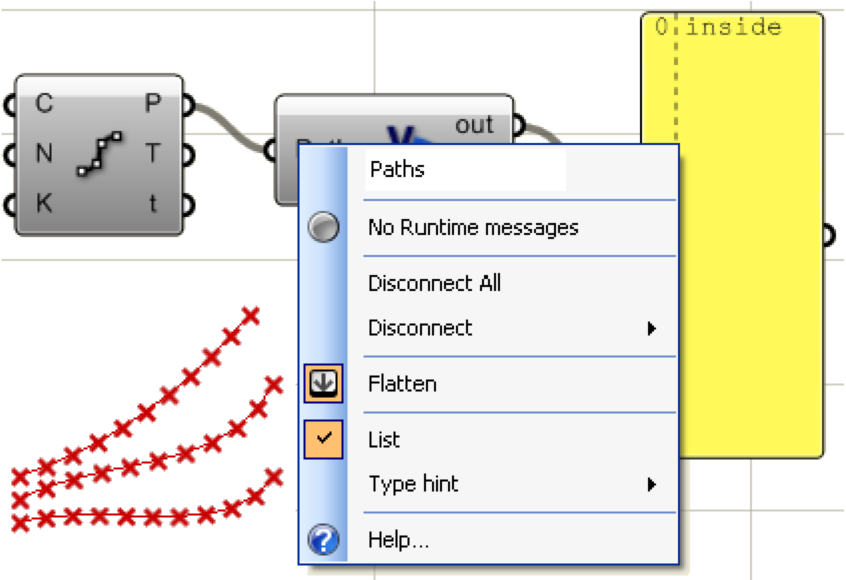


```
Sub RunScript(ByVal NList As List(Of Double))

    Dim Sum As Double = 0.0
    Dim i As Integer
    For i = 0 To NList.Count() - 1
      Sum = Sum + NList(i)
    Next

    A = Sum
End Sub
```

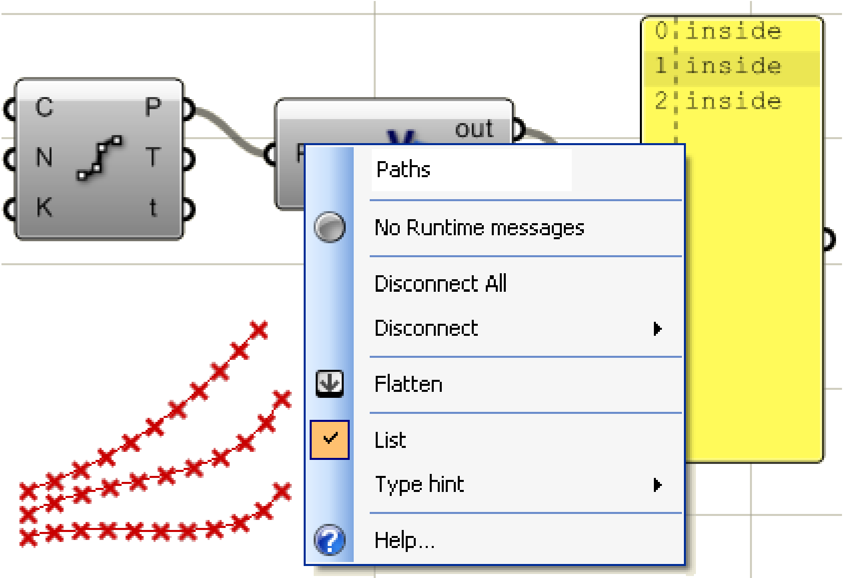
14.12 Processing Trees in Grasshopper

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:

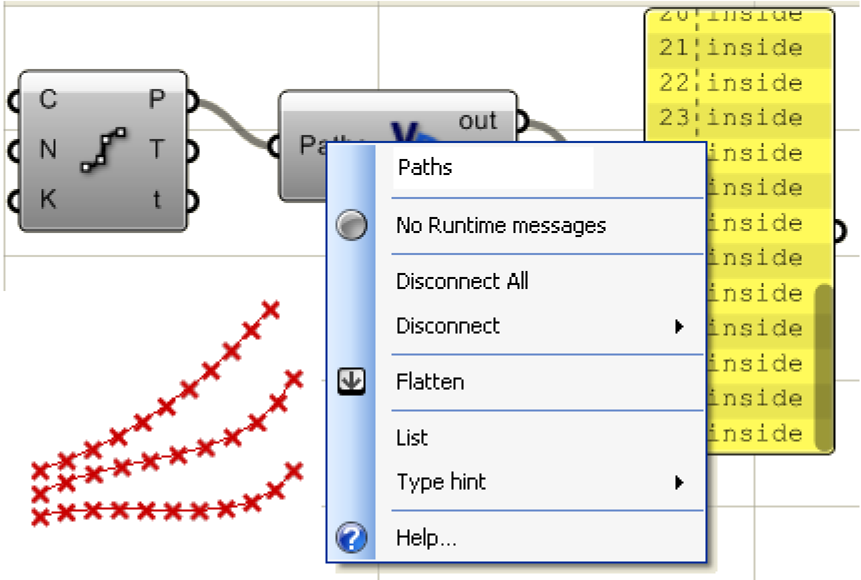


B: If “List” only is checked, then the component is called three times in each time, a list of divide points of one curve is passed:



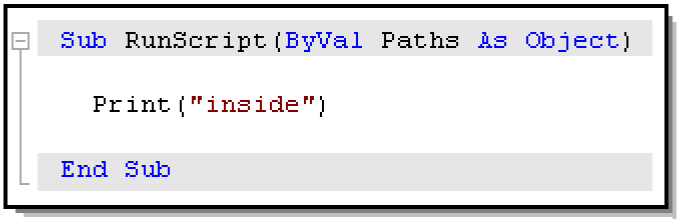
```
0 inside
1 inside
2 inside
```

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:



```
20 inside
21 inside
22 inside
23 inside
   inside
   inside
   inside
   inside
   inside
   inside
   inside
   inside
   inside
```

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



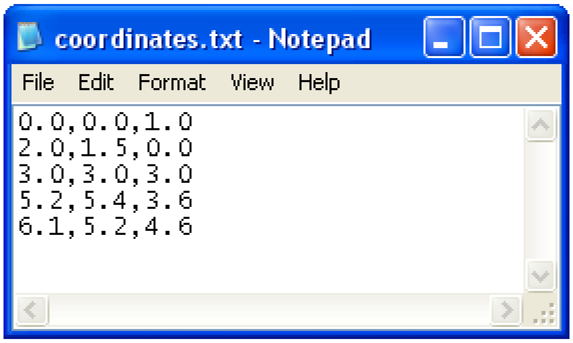
```
Sub RunScript(ByVal Paths As Object)

  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:

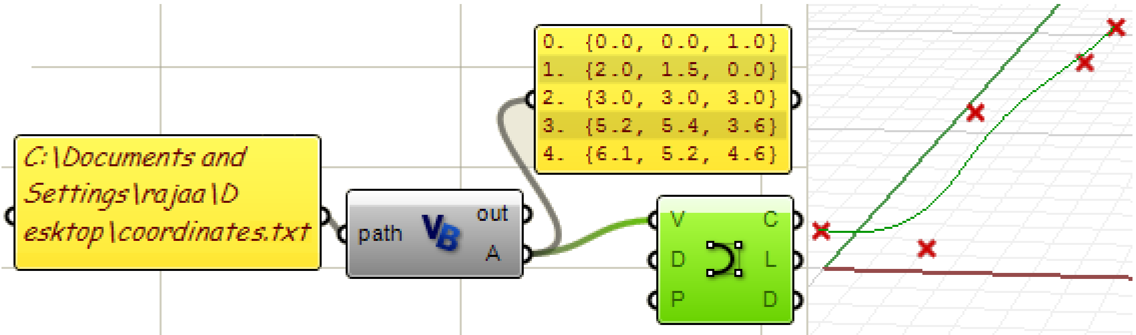
- Open the file. Generally you need a path to point to.
- Read the string (whole string or line by line).
- Tokenize the string using some delimiter character(s).
- Cast each token (to double in this case).
- Store result.

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.



```
0.0,0.0,1.0
2.0,1.5,0.0
3.0,3.0,3.0
5.2,5.4,3.6
6.1,5.2,4.6
```

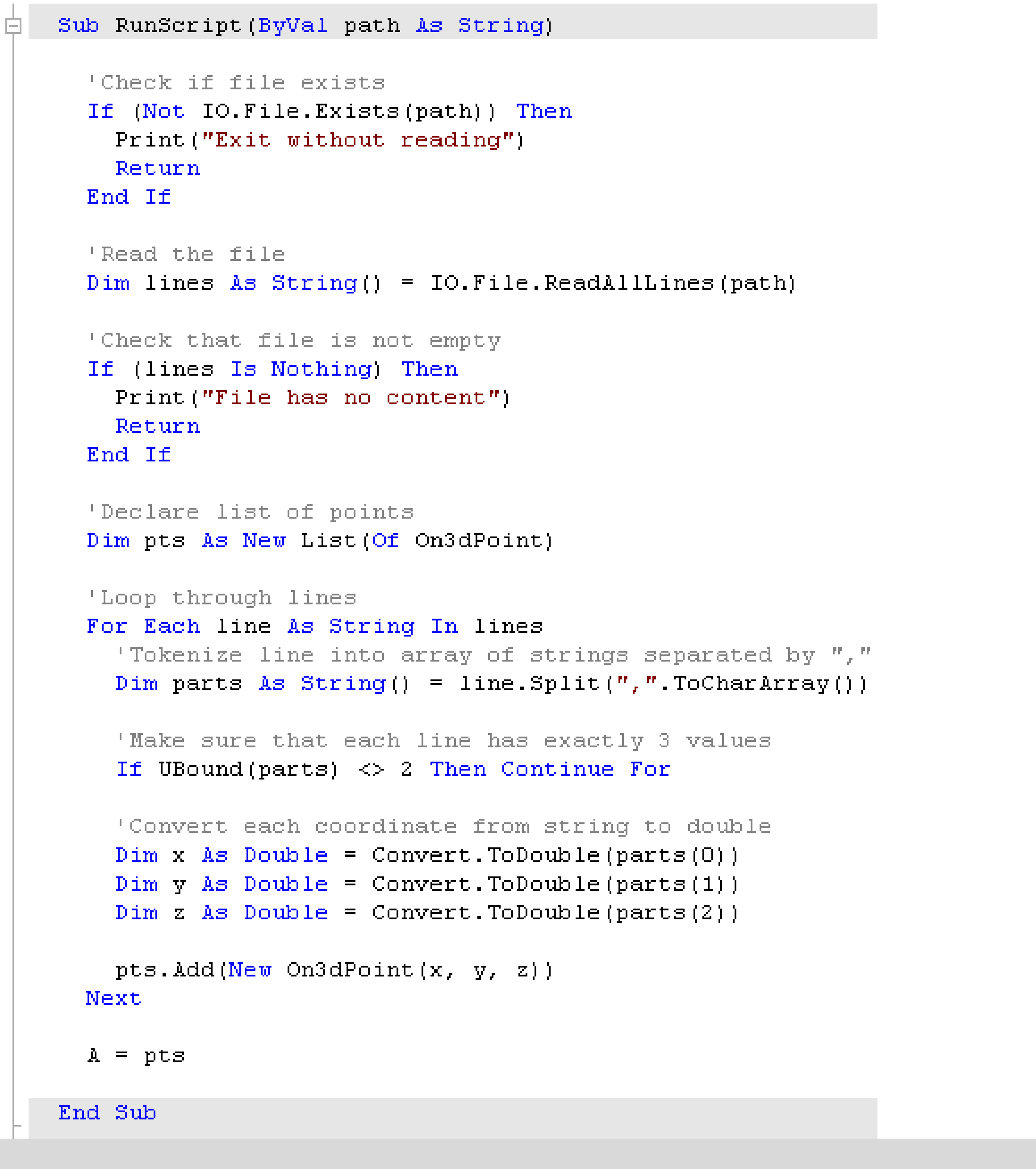
The VB component accepts as an input a string which is the path of the file to read and output On3dPoints.



C:\Documents and Settings\rajaa\Desktop\coordinates.txt

```
0. {0.0, 0.0, 1.0}
1. {2.0, 1.5, 0.0}
2. {3.0, 3.0, 3.0}
3. {5.2, 5.4, 3.6}
4. {6.1, 5.2, 4.6}
```

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.

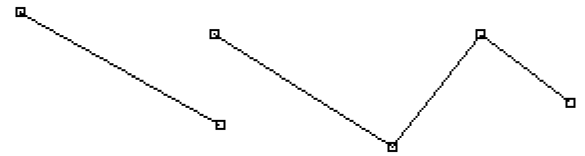
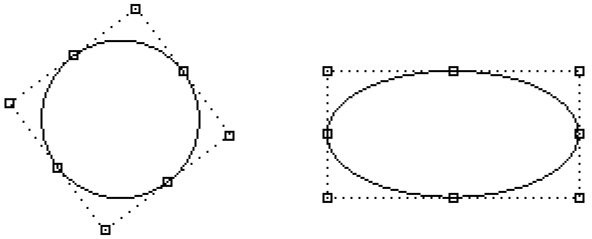
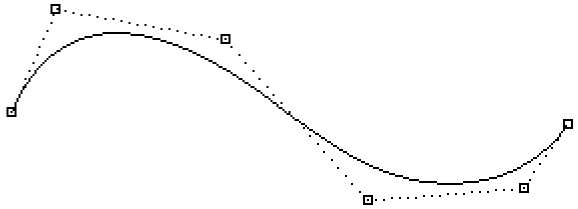
15.2 Understanding NURBS

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.

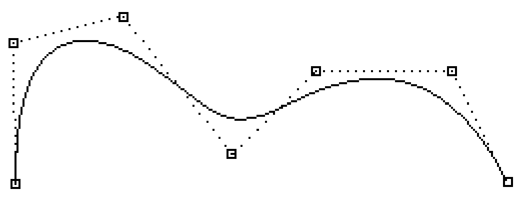
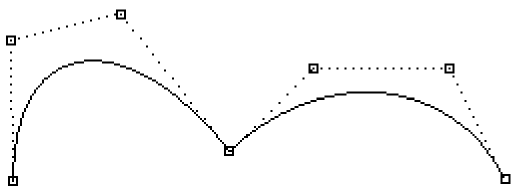
Full multiplicity knot is a knot duplicated number of times equal to curve degree. Clamped curves have knots with full multiplicity at the two ends of the curve and this is why end control points coincide with curve end points. If there were full multiplicity knot in the middle of the knot vector, then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

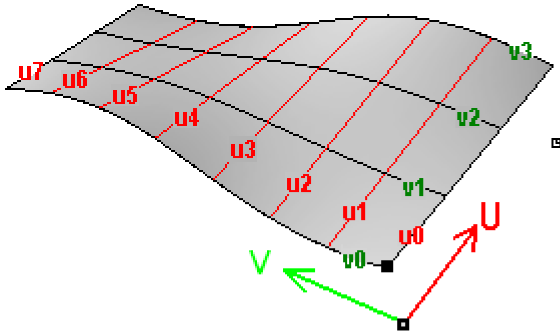
	Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,2,3,5,5,5)
	Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,1,1,4,4,4) ***Note***: *Full knot multiplicity in the middle creates a kink and the curve is forced to go through the associated control point.*

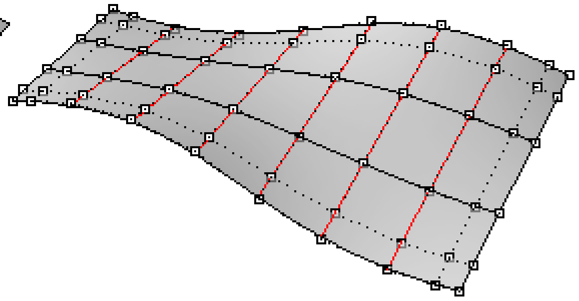
Evaluation rule
The evaluation rule uses a mathematical formula that takes a number and assigns a point. The formula involves the degree, control points, and knots.

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.

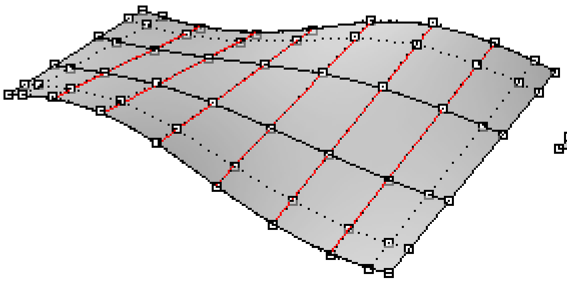
NURBS Surfaces

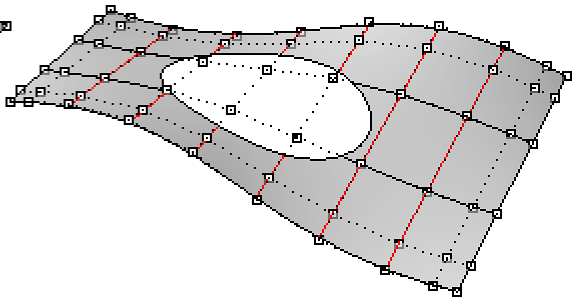
You can think of nurbs surfaces as a grid of nurbs curves that go in two directions. The shape of a NURBS surface is defined by a number of control points and the degree of that surface in each one of the two directions (u- and v-directions). Refer to *Rhino Help Glossary* for more details.





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.



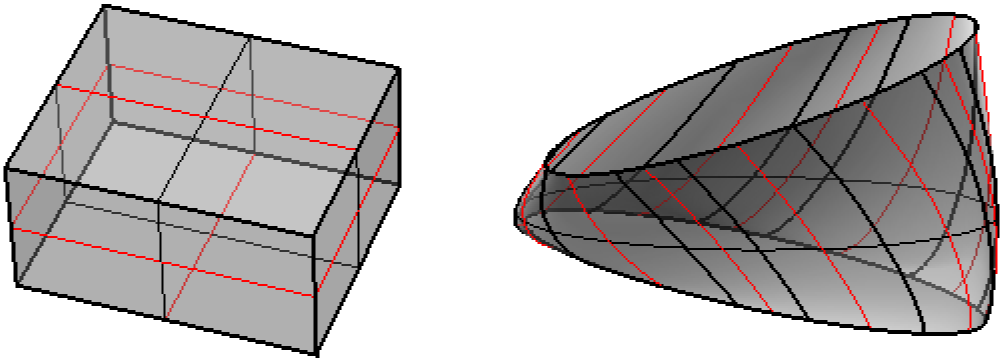


The surface on the left is untrimmed. The one on the right is the same surface trimmed with an elliptical hole. Note that the NURBS structure of the surface doesn't change when trimming.

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

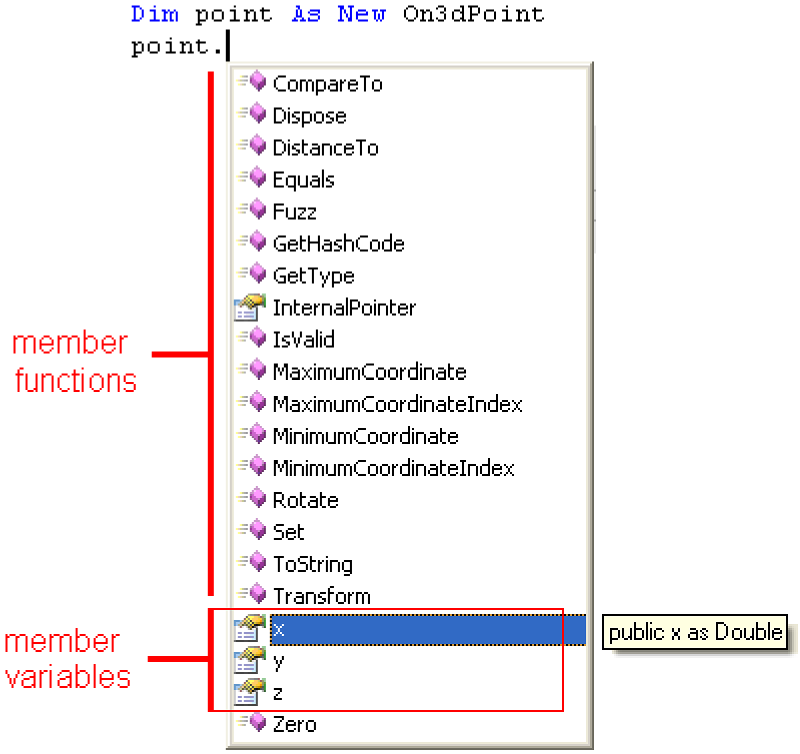
- **Miscellaneous**
    - OnBoundingBox (For objects bounding box calculation)
    - OnInterval (Used for curve and surface domains)
    - OnXform (for transforming geometry objects: move, rotate, scale, etc.)
    - OnMassProperties (to calculate volume, area, centroid, etc)

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:



```
Dim point As New On3dPoint
point.
```

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)

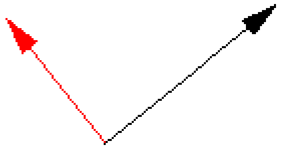
Get unit vector (set vector length to 1):

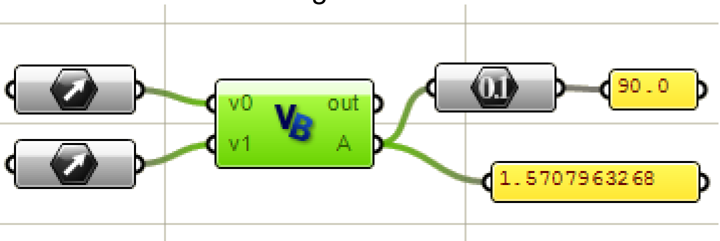
v0.Unitize()

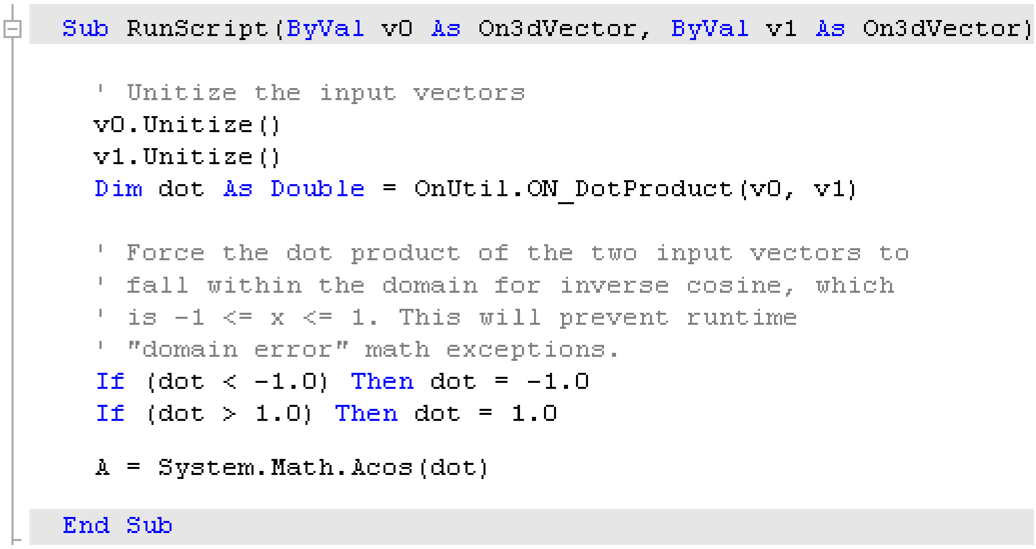
Get vector length:

Dim length As Double = v0.Length()

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







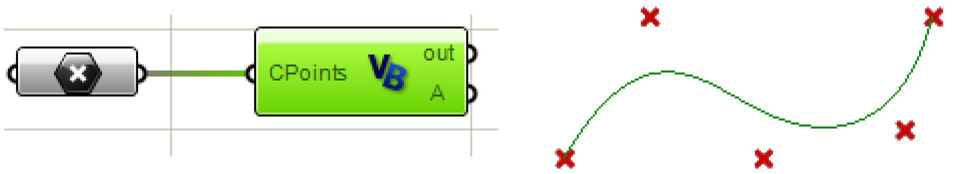
```
Sub RunScript(ByVal v0 As On3dVector, ByVal v1 As On3dVector)

  ' 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
```

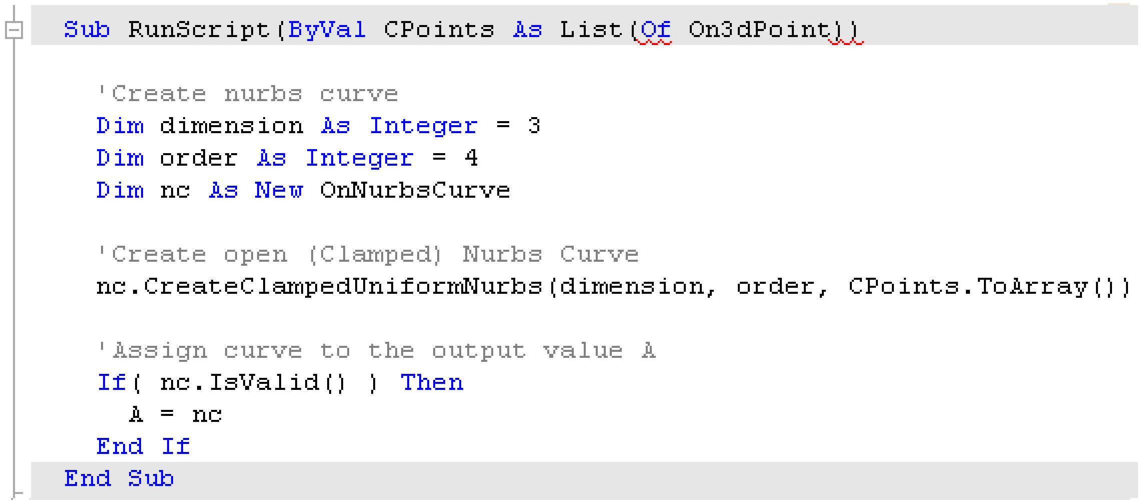
15.7 OnNurbsCurve

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).

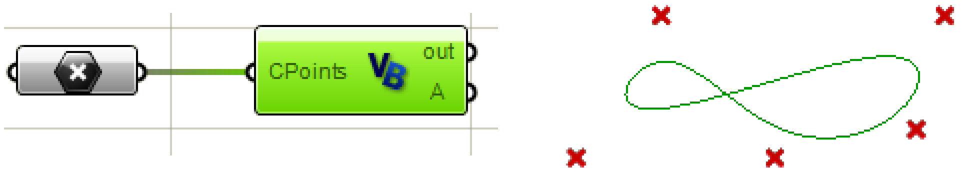
There are functions to help create the knot vector, as we will see shortly, so basically you need to decide on the degree and have a list of control points and you are good to go. The following example creates a clamped curve.

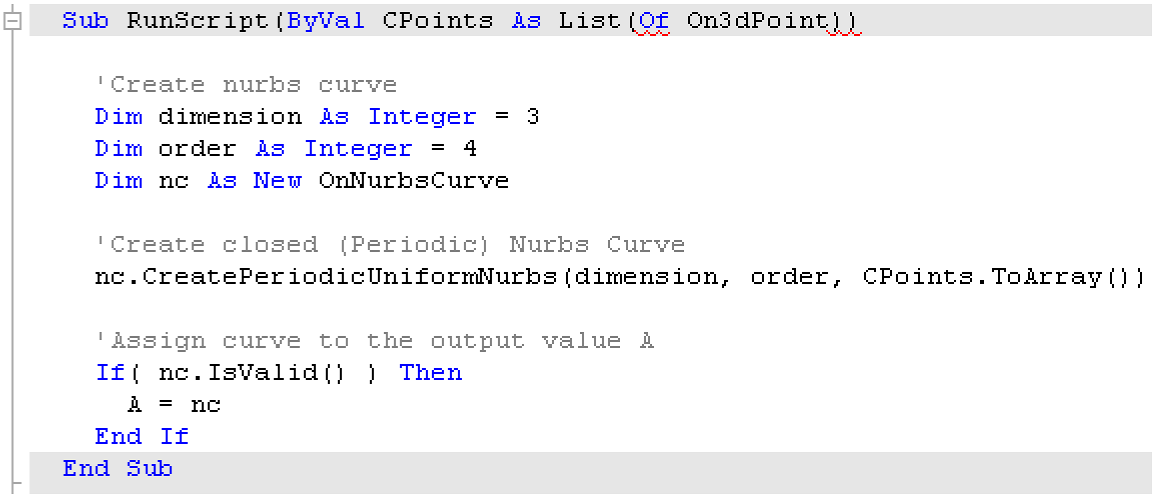




```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

  '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.



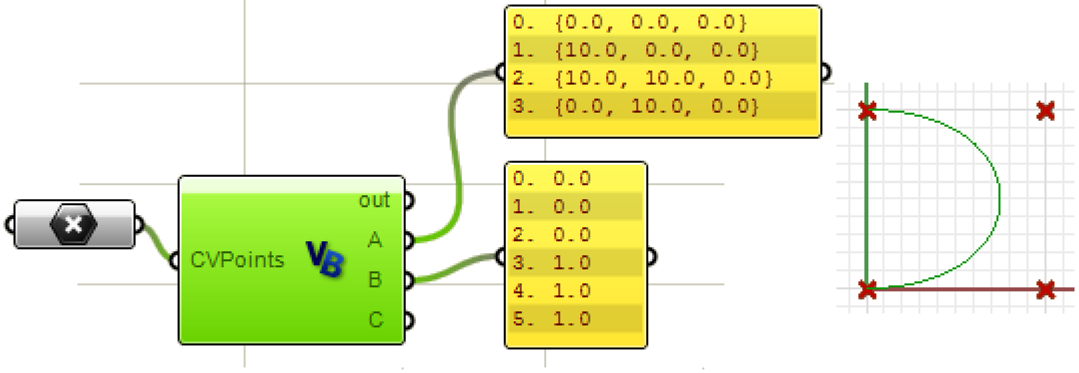


```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

    '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.

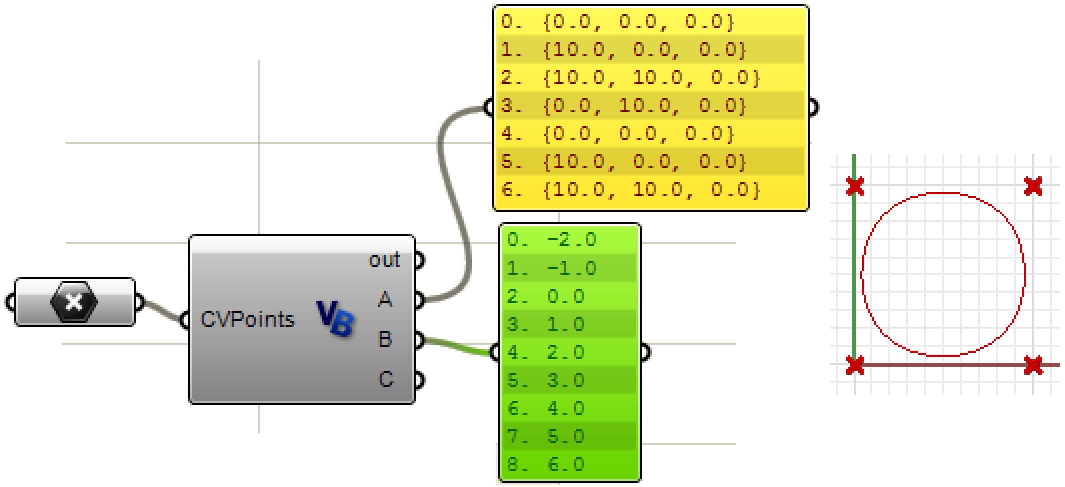
The following component creates clamped NURBS curve and outputs:



```
0. {0.0, 0.0, 0.0}
1. {10.0, 0.0, 0.0}
2. {10.0, 10.0, 0.0}
3. {0.0, 10.0, 0.0}
```

```
0. 0.0
1. 0.0
2. 0.0
3. 1.0
4. 1.0
5. 1.0
```

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

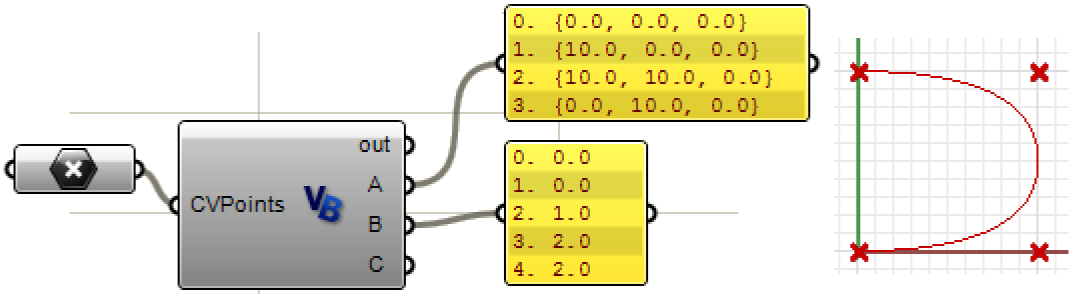


```
0. {0.0, 0.0, 0.0}
1. {10.0, 0.0, 0.0}
2. {10.0, 10.0, 0.0}
3. {0.0, 10.0, 0.0}
4. {0.0, 0.0, 0.0}
5. {10.0, 0.0, 0.0}
6. {10.0, 10.0, 0.0}
```

```
0. -2.0
1. -1.0
2. 0.0
3. 1.0
4. 2.0
5. 3.0
6. 4.0
7. 5.0
8. 6.0
```

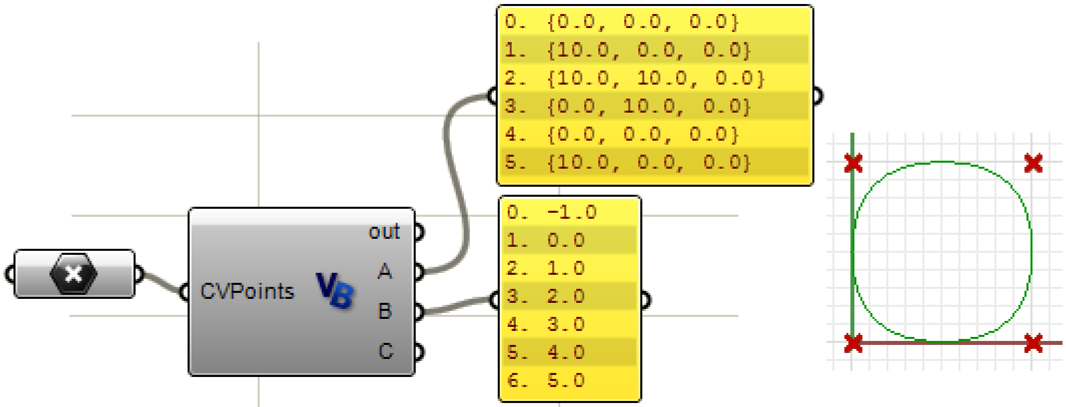
Note that the periodic curve turned the four input points into seven control points (4+degree)" and while the clamped curve used four control points only. The knot vector of the periodic curve used only simple knots while the clamped curve start and end knots have full multiplicity.

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.



```
0. {0.0, 0.0, 0.0}
1. {10.0, 0.0, 0.0}
2. {10.0, 10.0, 0.0}
3. {0.0, 10.0, 0.0}
```

```
0. 0.0
1. 0.0
2. 1.0
3. 2.0
4. 2.0
```



```
0. {0.0, 0.0, 0.0}
1. {10.0, 0.0, 0.0}
2. {10.0, 10.0, 0.0}
3. {0.0, 10.0, 0.0}
4. {0.0, 0.0, 0.0}
5. {10.0, 0.0, 0.0}
```

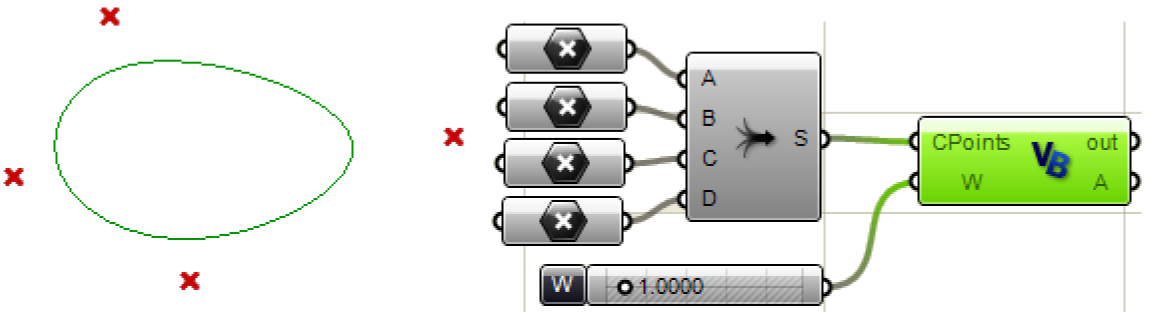
```
0. -1.0
1. 0.0
2. 1.0
3. 2.0
4. 3.0
5. 4.0
6. 5.0
```

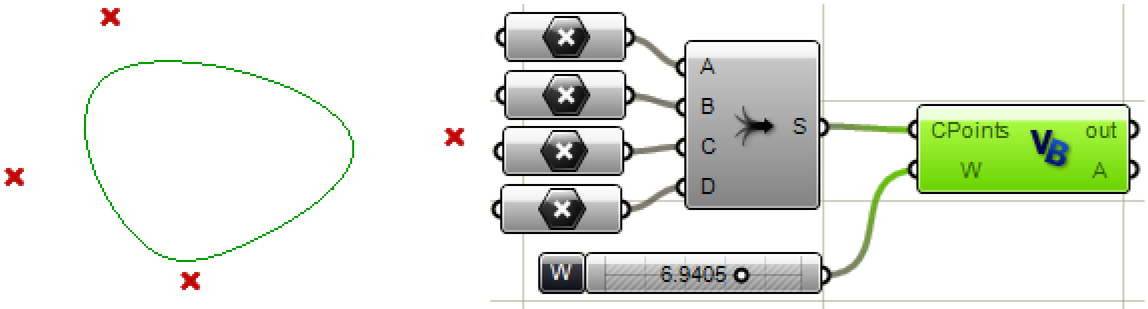
This is the code used to navigate through CV points and knots in the previous examples:

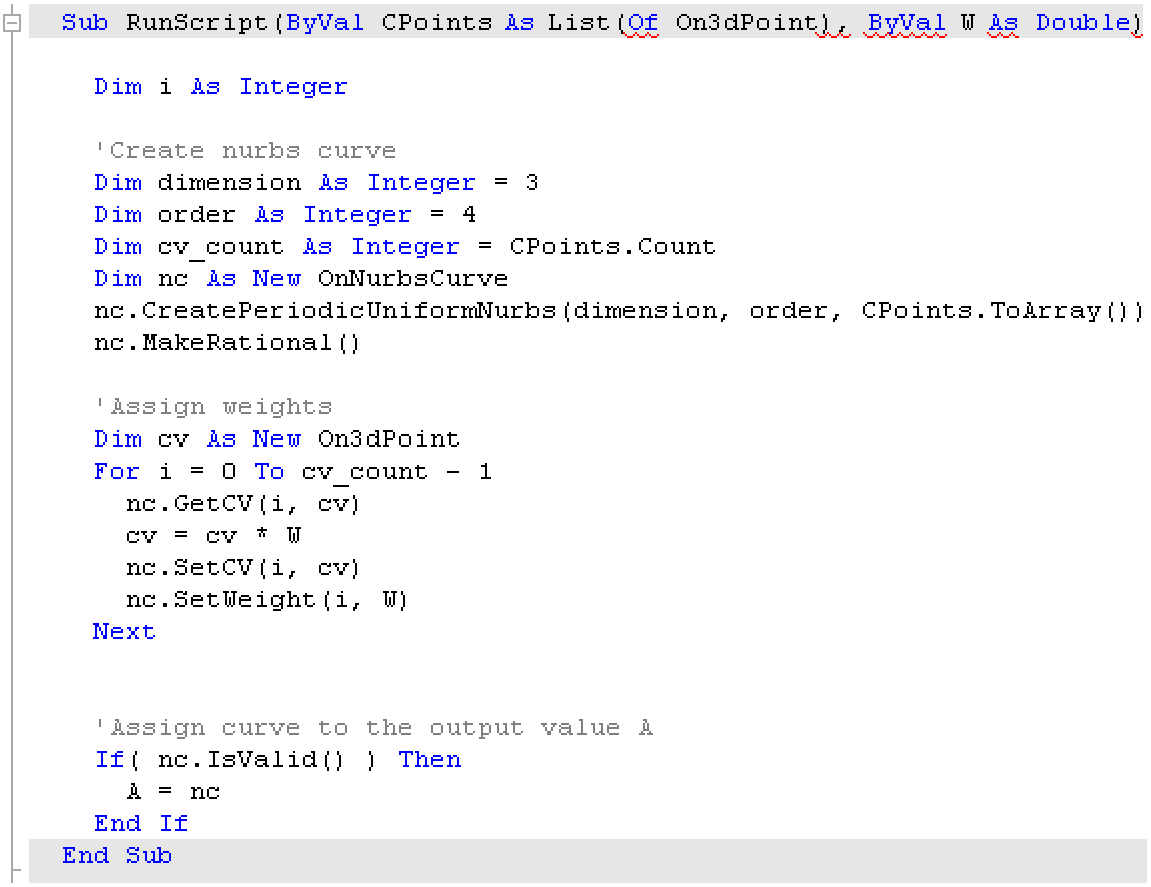
```
'Output control points
Dim count As Double = nc.CVCount()
Dim i As Integer
Dim cvs As New List( Of On3dPoint )
For i = 0 To count - 1
  Dim cv As New On3dPoint(0, 0, 0)
  nc.GetCV(i, cv)
  cvs.Add(cv)
Next

'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.







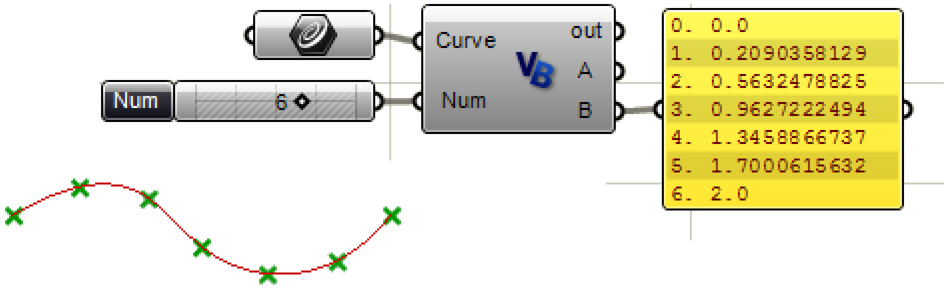
```
Sub RunScript(ByVal CPoints As List(Of On3dPoint), ByVal W As Double)

 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

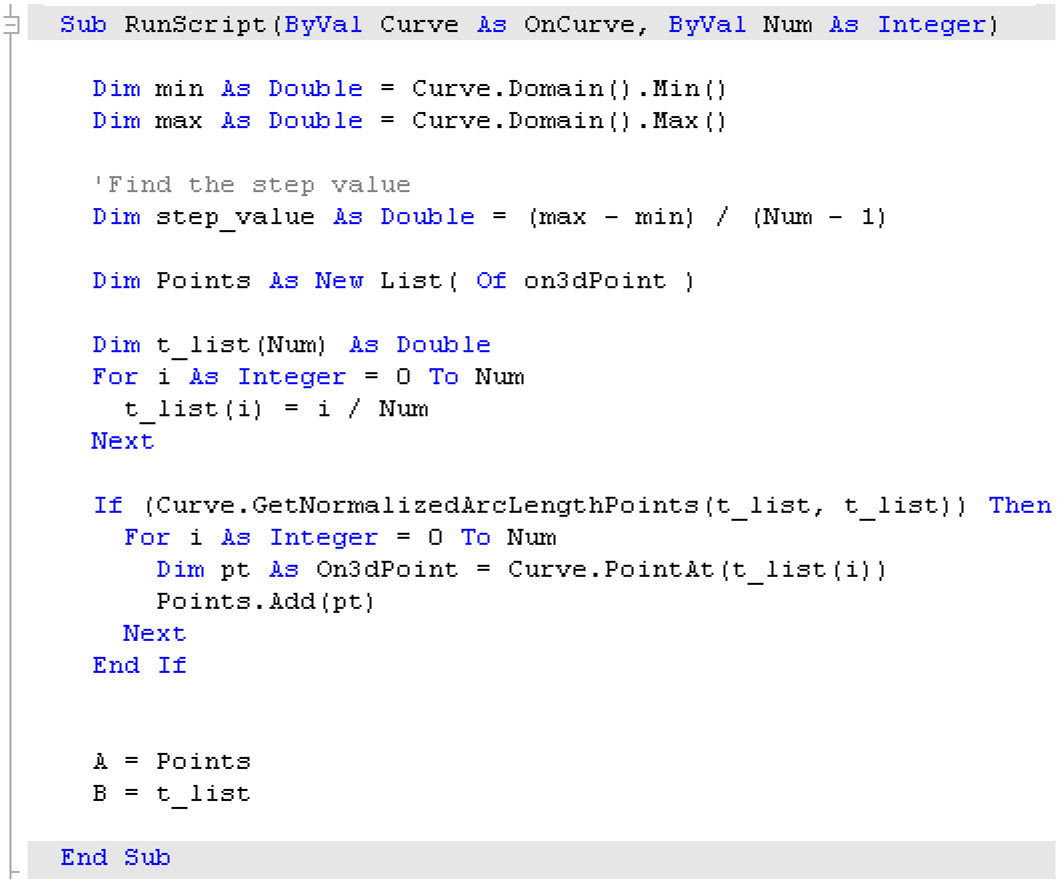
Dividing a curve into a number of segments involves the following steps:

- 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.



```
0.  0.0
1.  0.2090358129
2.  0.5632478825
3.  0.9627222494
4.  1.3458866737
5.  1.7000615632
6.  2.0
```



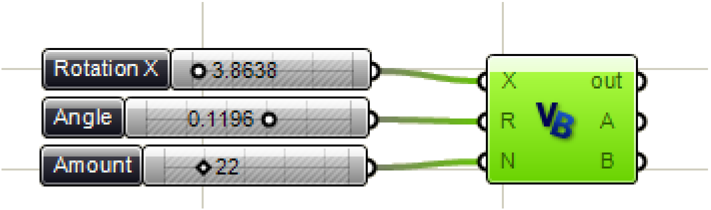
```
Sub RunScript(ByVal Curve As OnCurve, ByVal Num As Integer)

  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
```

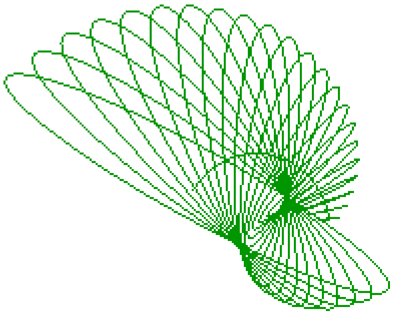
15.8 Curve Classes not Derived from OnCurve

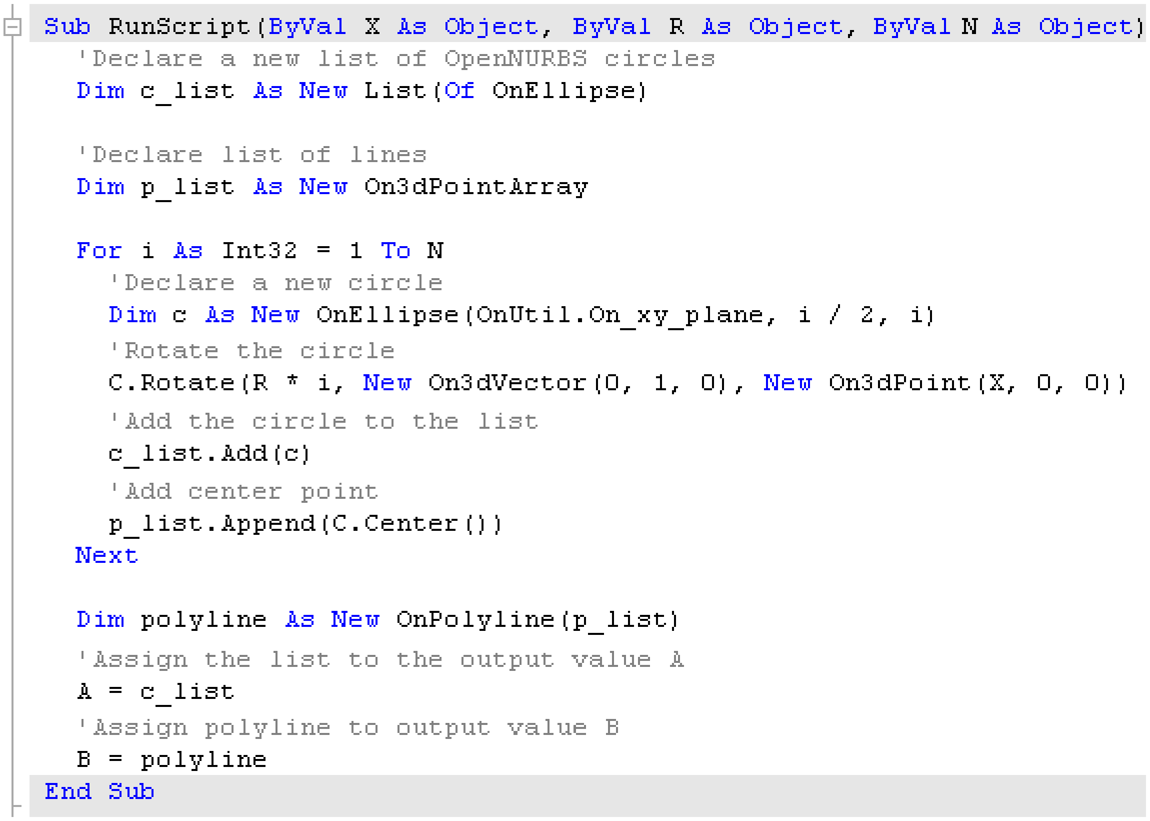
Although all curves can be represented as NURBS curves, it is useful sometimes to work with other types of curve geometry. One reason is because they mathematical representation hat is easier to understand than NURBS and are typically more light-weighted. It is relatively easy to get the NURBS form of 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:







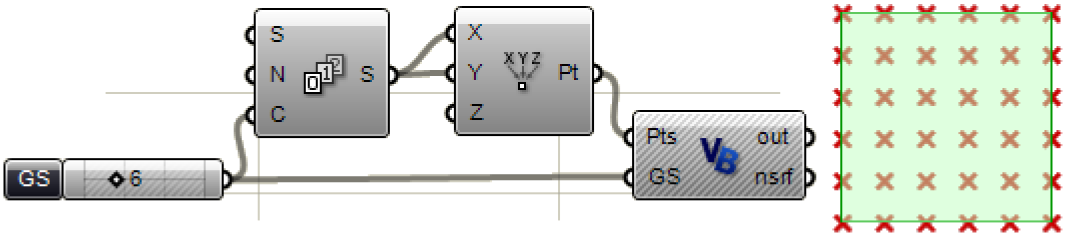
```
Sub RunScript(ByVal X As Object, ByVal R As Object, ByVal N As Object)
  'Declare a new list of OpenNURBS circles
  Dim c_list As New List(Of OnEllipse)

  '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
```

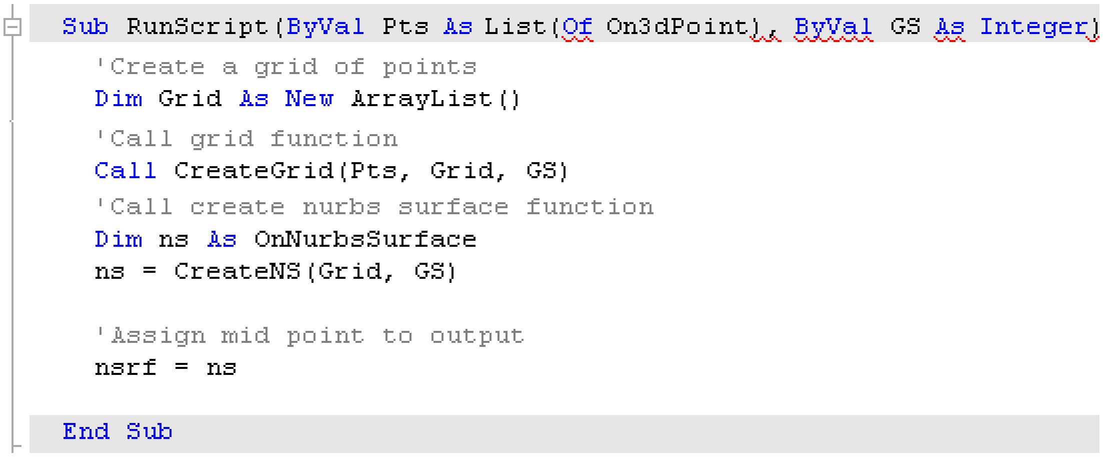
15.9 OnNurbsSurface

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

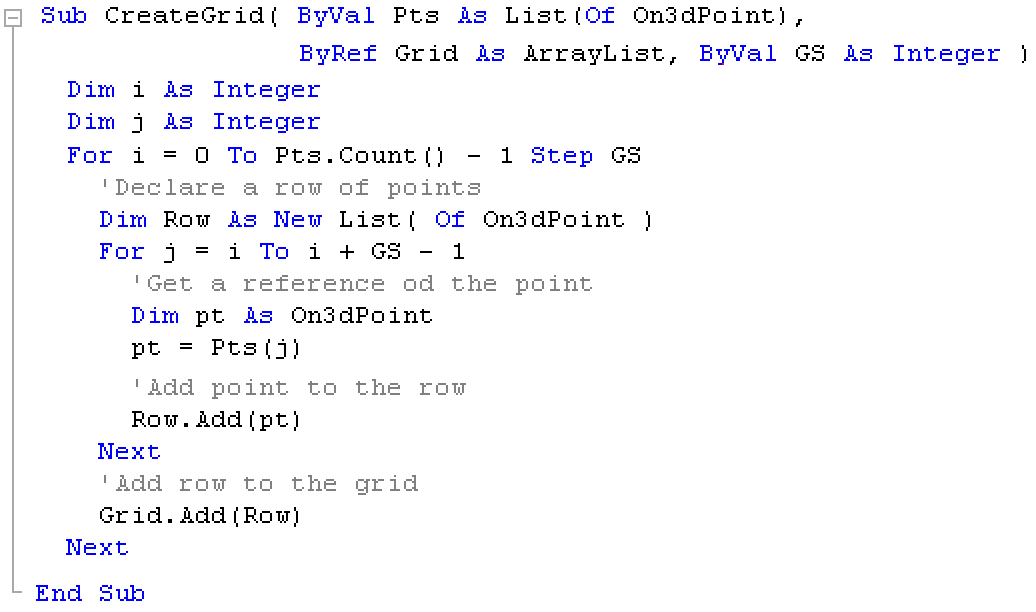
- Dimension, which is typically = 3.
- Order in u and v direction: degree + 1.
- Control points.
- Knot vector in u and v directions.
- Surface type (clamped or periodic).

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

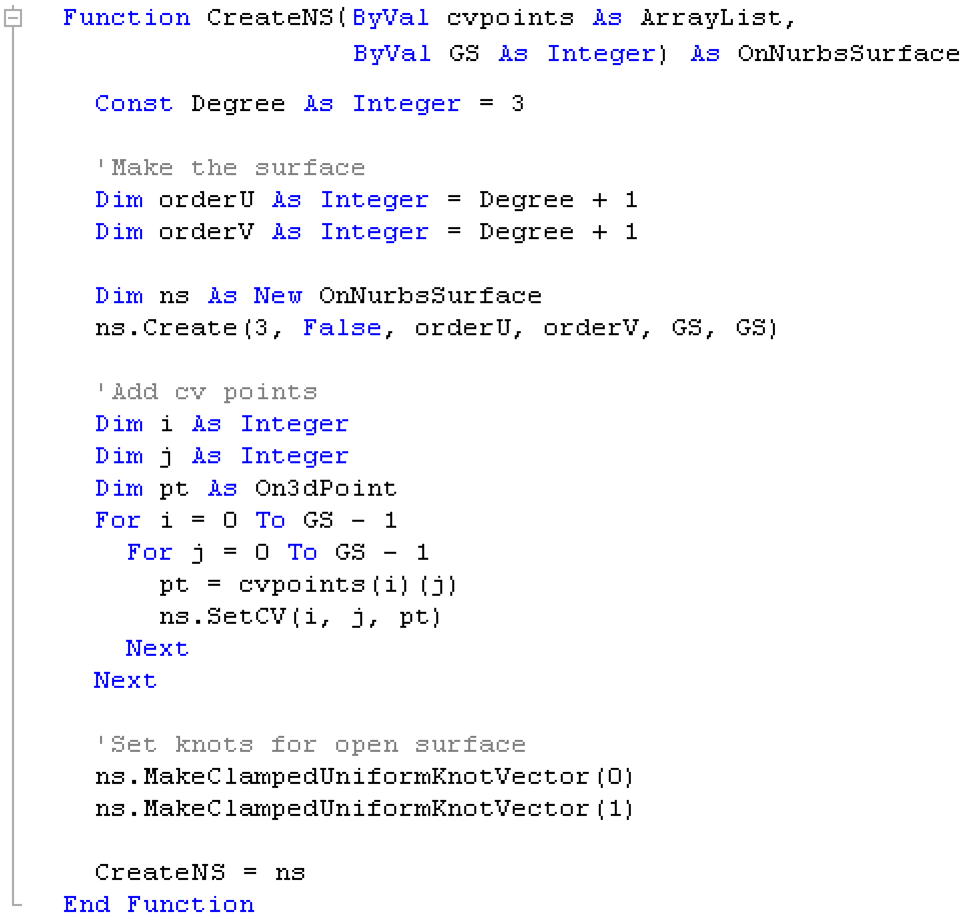




```
Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)
  'Create a grid of points
  Dim Grid As New ArrayList()
  'Call grid function
  Call CreateGrid(Pts, Grid, GS)
  'Call create nurbs surface function
  Dim ns As OnNurbsSurface
  ns = CreateNS(Grid, GS)

  'Assign mid point to output
  nsrf = ns

End Sub
```



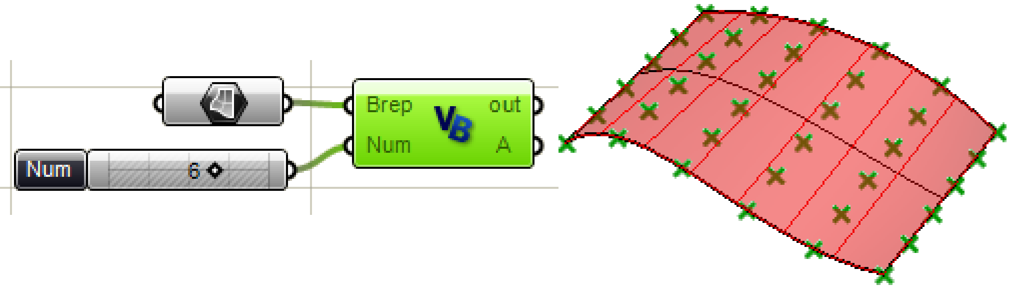
```
Sub CreateGrid( ByVal Pts As List(Of On3dPoint),
                ByRef Grid As ArrayList, ByVal GS As Integer )
  Dim i As Integer
  Dim j As Integer
  For i = 0 To Pts.Count() - 1 Step GS
    'Declare a row of points
    Dim Row As New List( Of On3dPoint )
    For j = i To i + GS - 1
      'Get 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
```

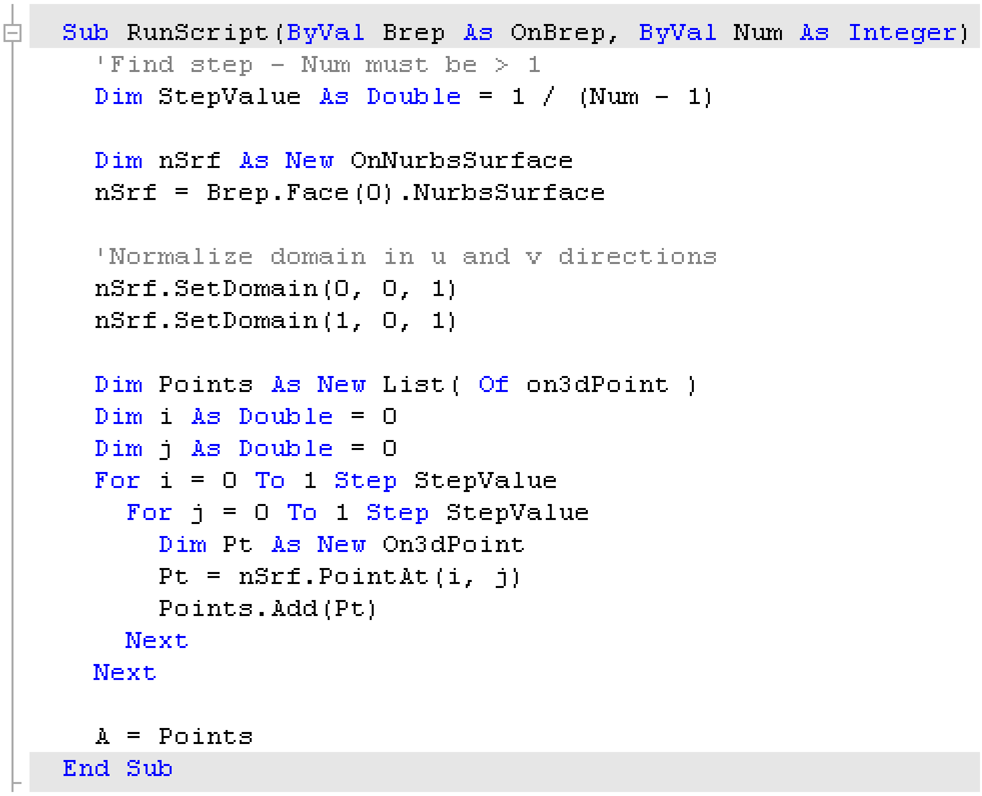


```
Function CreateNS(ByVal cvpoints As ArrayList,
                  ByVal GS As Integer) As OnNurbsSurface

  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.

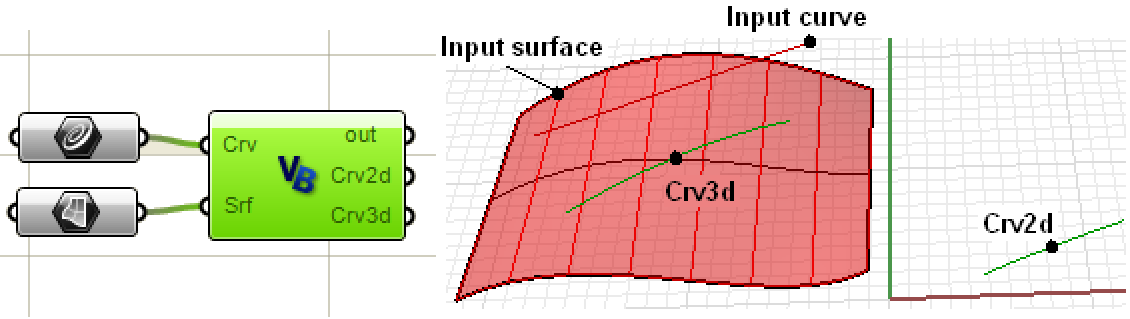


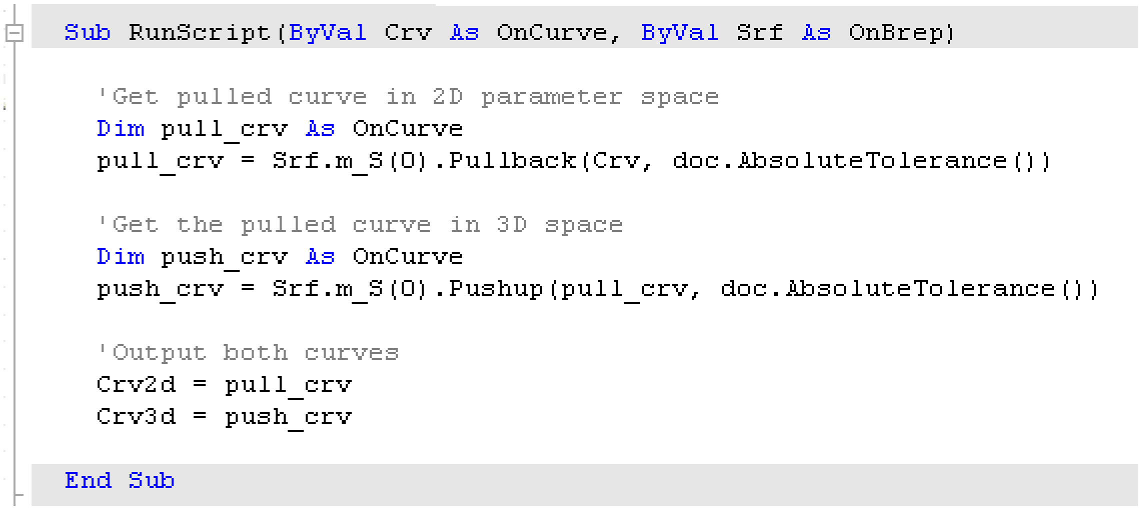


```
Sub RunScript(ByVal Brep As OnBrep, ByVal Num As Integer)
  'Find step - Num must be > 1
  Dim StepValue As Double = 1 / (Num - 1)

  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.

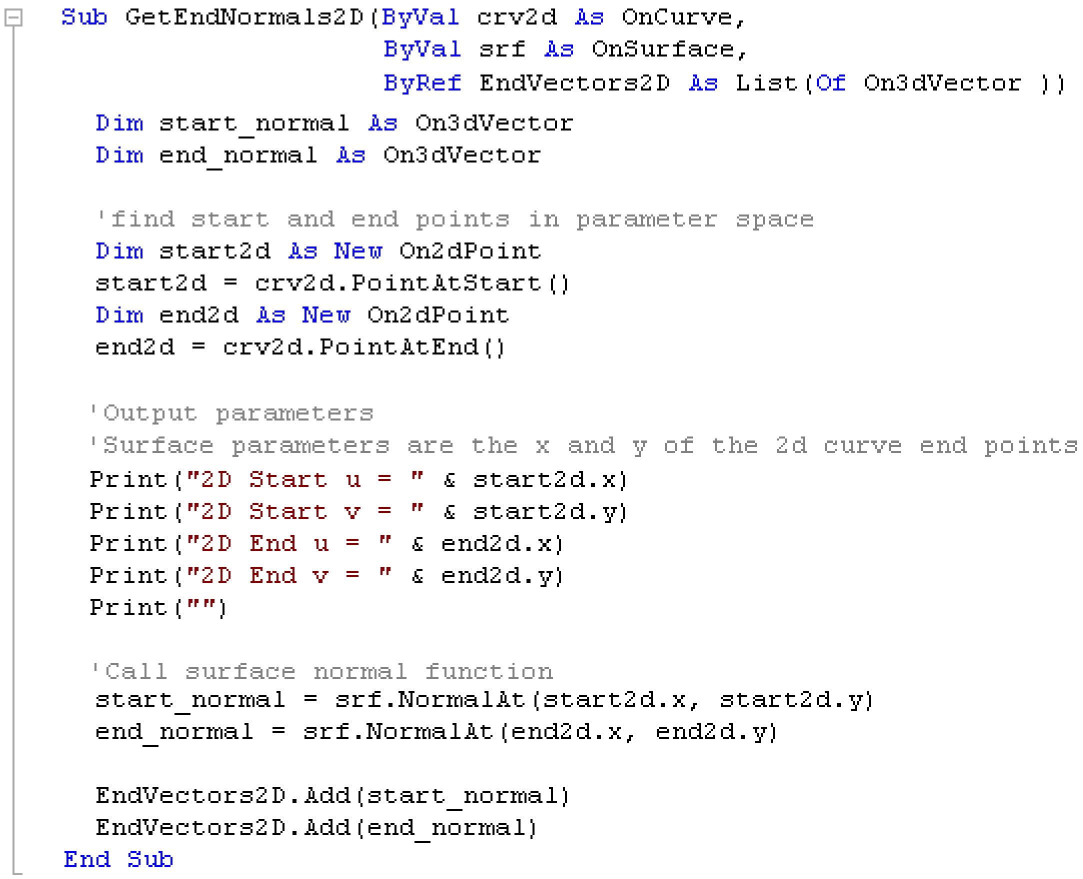




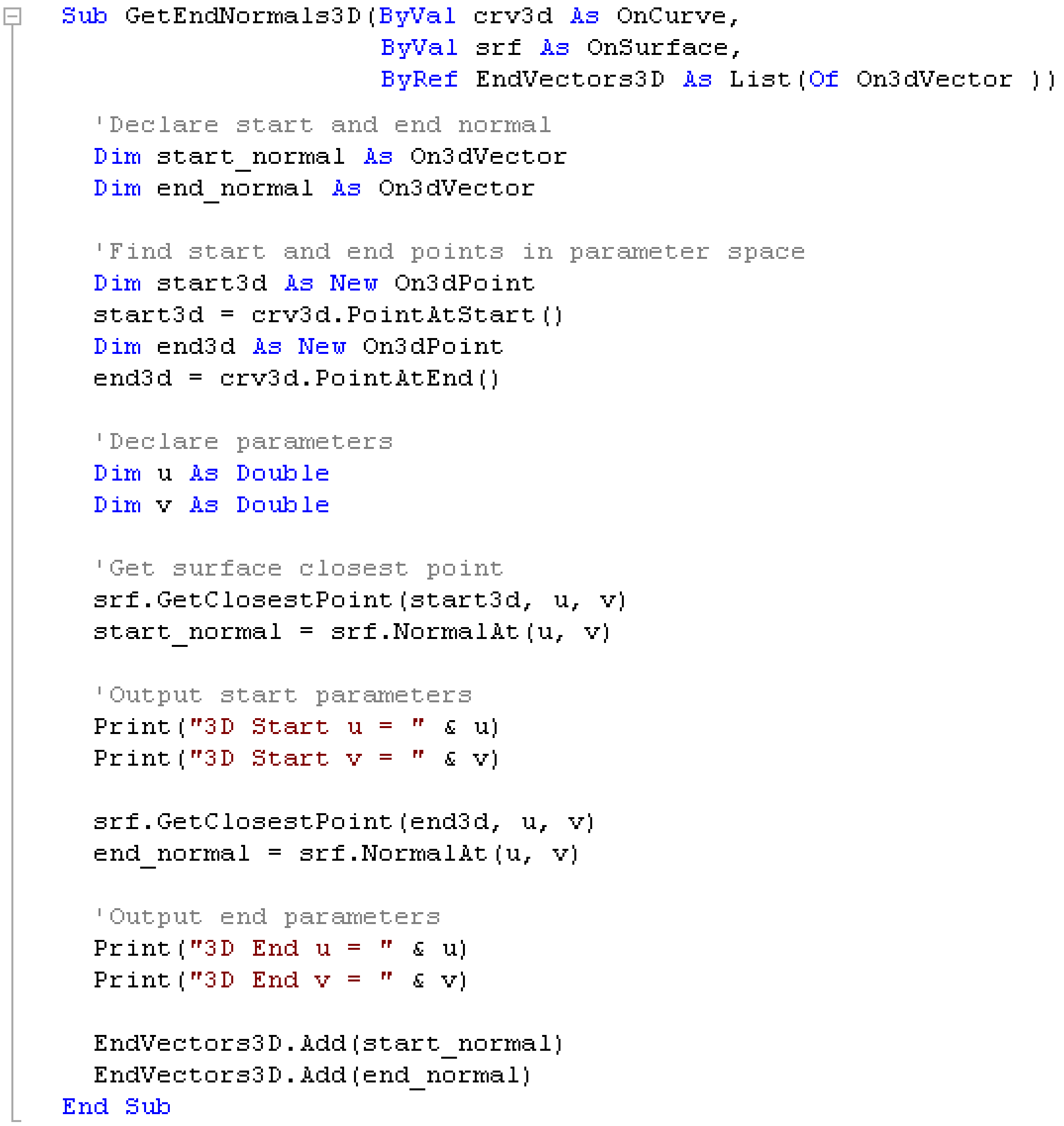
```
Sub RunScript(ByVal Crv As OnCurve, ByVal Srf As OnBrep)

  '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.

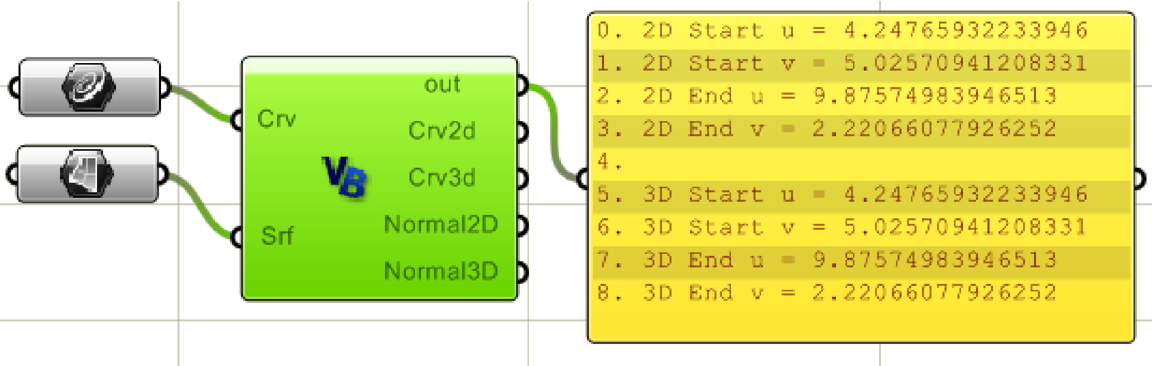


```
Sub GetEndNormals2D(ByVal crv2d As OnCurve,
                    ByVal srf As OnSurface,
                    ByRef EndVectors2D As List(Of On3dVector ))
  Dim start_normal As On3dVector
  Dim end_normal As On3dVector

  '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.



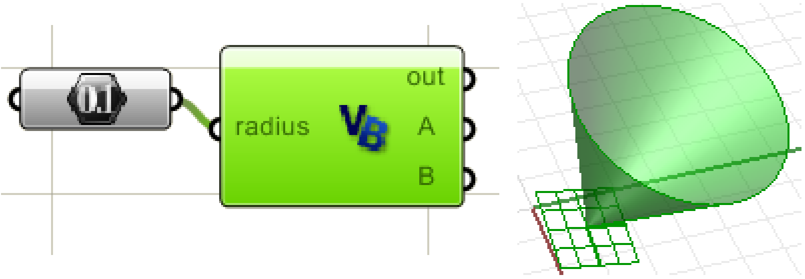
```
0. 2D Start u = 4.24765932233946
1. 2D Start v = 5.02570941208331
2. 2D End u = 9.87574983946513
3. 2D End v = 2.22066077926252
4.
5. 3D Start u = 4.24765932233946
6. 3D Start v = 5.02570941208331
7. 3D End u = 9.87574983946513
8. 3D End v = 2.22066077926252
```

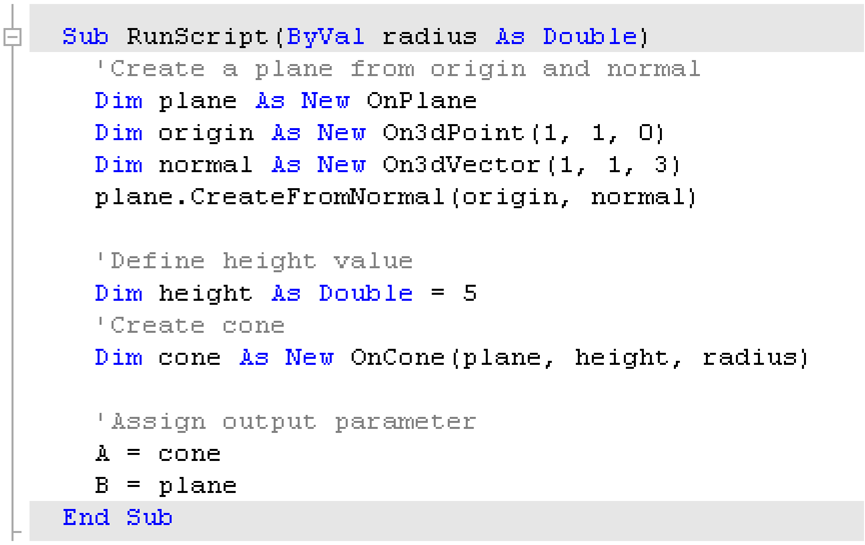
15.10 Surface Classes not Derived from OnSurface

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:





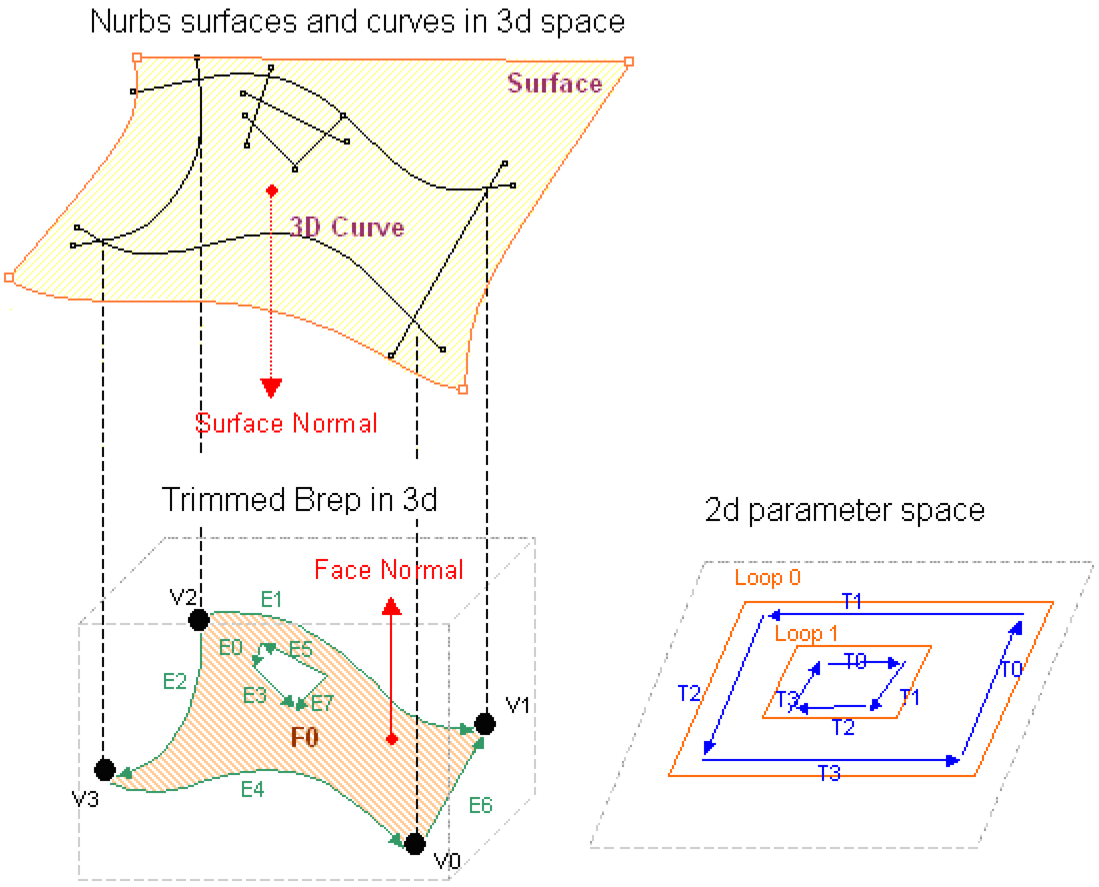
```
Sub RunScript(ByVal radius As Double)
  'Create a plane from origin and normal
  Dim plane As New OnPlane
  Dim origin As New On3dPoint(1, 1, 0)
  Dim normal As New On3dVector(1, 1, 3)
  plane.CreateFromNormal(origin, normal)

  '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:

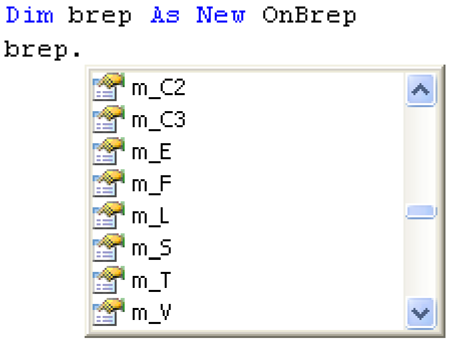
- Geometry: 3d geometry of nurbs curves and surfaces. Also 2D curves of parametric space or trim curves.
- 3D Topology: faces, edges and vertices. Each face references one nurbs surface. The face also knows all loops that belong to that face. Edges reference 3d curves. Each edge has a list of trims that 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

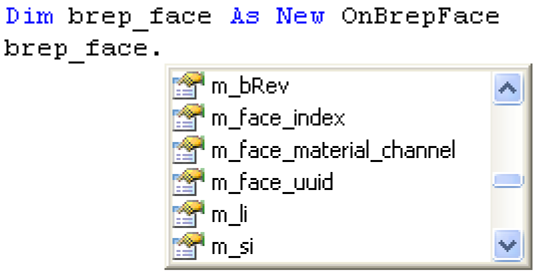
OnBrep Class member variables include all 3d and 2d geometry and topology information. Once you create an instance of a brep class, you can view all member functions and variables. The following image shows member functions in the auto-complete. The table lists data types with description.



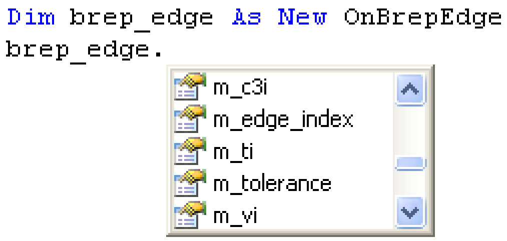
```
Dim brep As New OnBrep
brep.
```

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

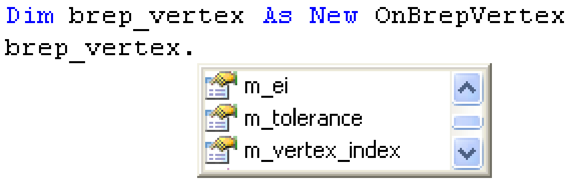
Notice that each of the OnBrep member functions is basically an array of other classes. For example m_F is an array of references to OnBrepFace. OnBrepFace is a class derived from OnSurfaceProxy and has variable and member functions of its own. Here are the member variables of OnBrepFace, OnBrepEdge,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



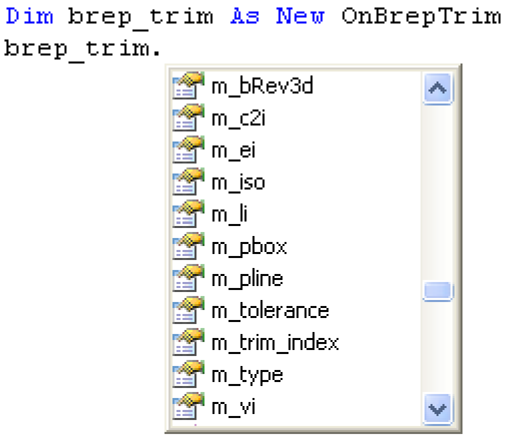
```
Dim brep_face As New OnBrepFace
brep_face.
```



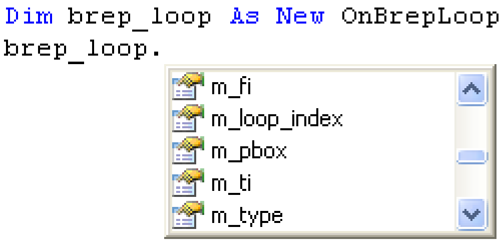
```
Dim brep_edge As New OnBrepEdge
brep_edge.
```



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

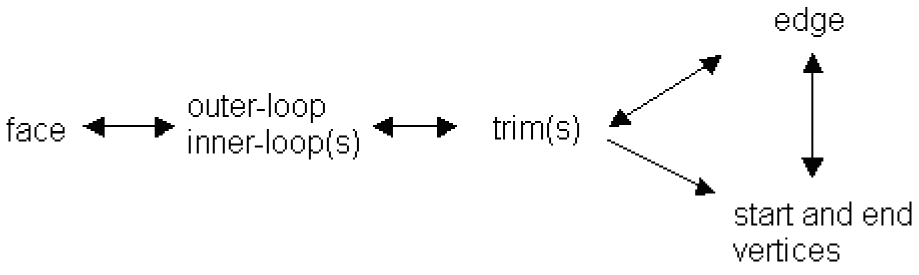


```
Dim brep_trim As New OnBrepTrim
brep_trim.
```

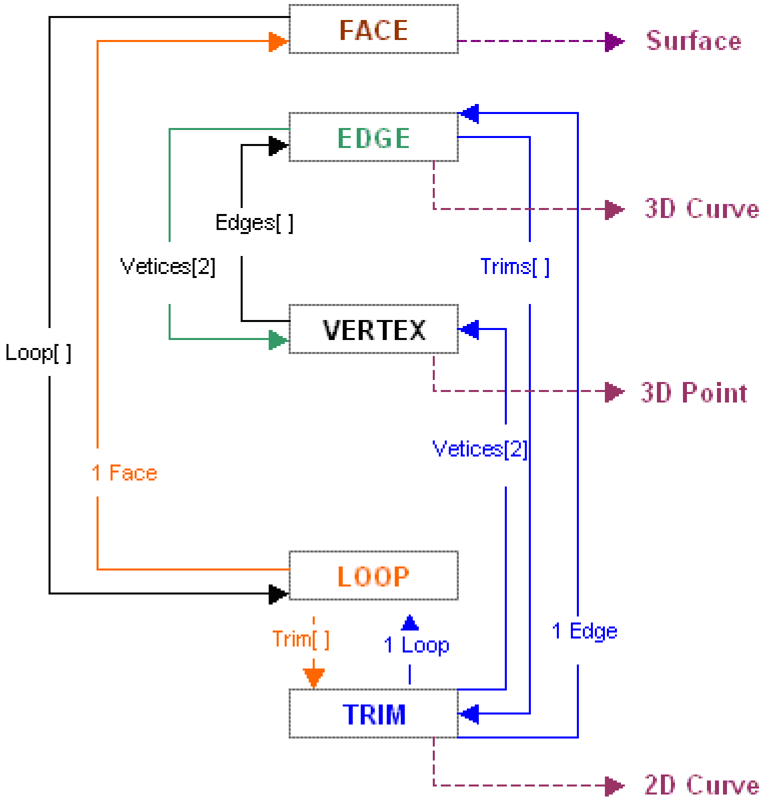


```
Dim brep_loop As New OnBrepLoop
brep_loop.
```

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.



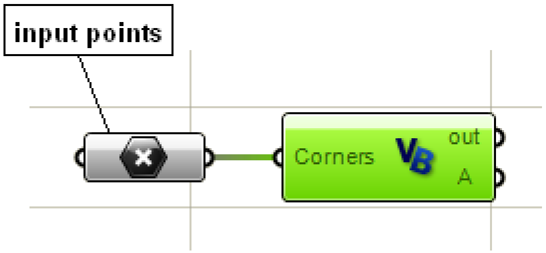
We will spend the coming few examples showing how to create an OnBrep, navigate different parts and extract brep information. We will also show how to use few of the functions that come with the class as well as global functions.

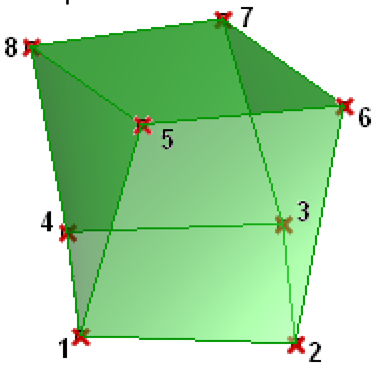
Create OnBrep

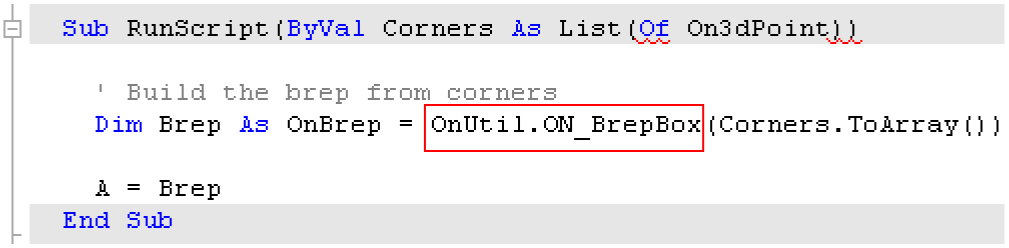
There are few ways to create a new instance of an OnBrep class:

- 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.



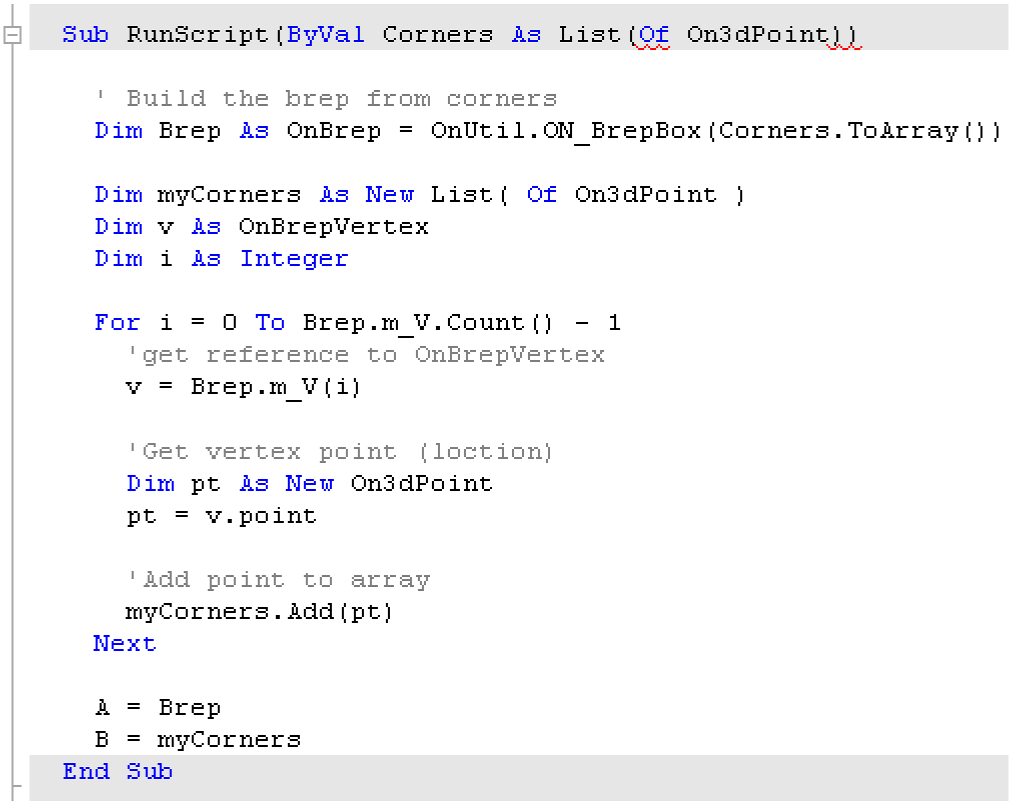




```
Sub RunScript(ByVal Corners As List(Of On3dPoint))

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

  A = Brep
End Sub
```

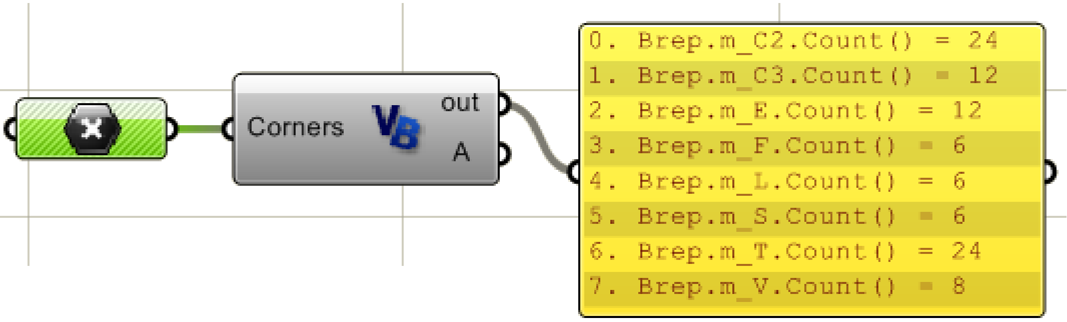
Navigate OnBrep data

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
```

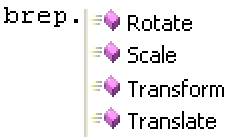
The following example shows how to get the number of geometry and topology parts in a brep box (faces, edges, trims, vertices, etc).



```
0. Brep.m_C2.Count() = 24
1. Brep.m_C3.Count() = 12
2. Brep.m_E.Count() = 12
3. Brep.m_F.Count() = 6
4. Brep.m_L.Count() = 6
5. Brep.m_S.Count() = 6
6. Brep.m_T.Count() = 24
7. Brep.m_V.Count() = 8
```

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.

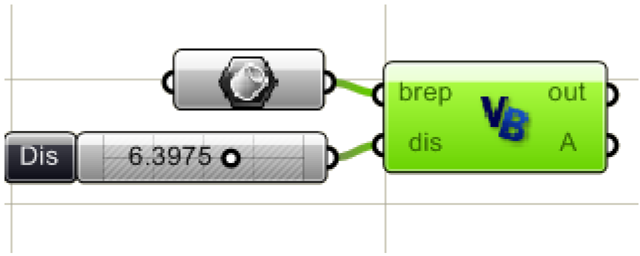


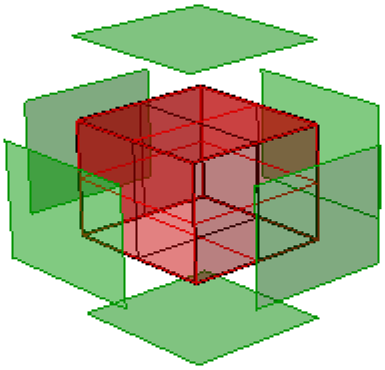
Edit OnBrep

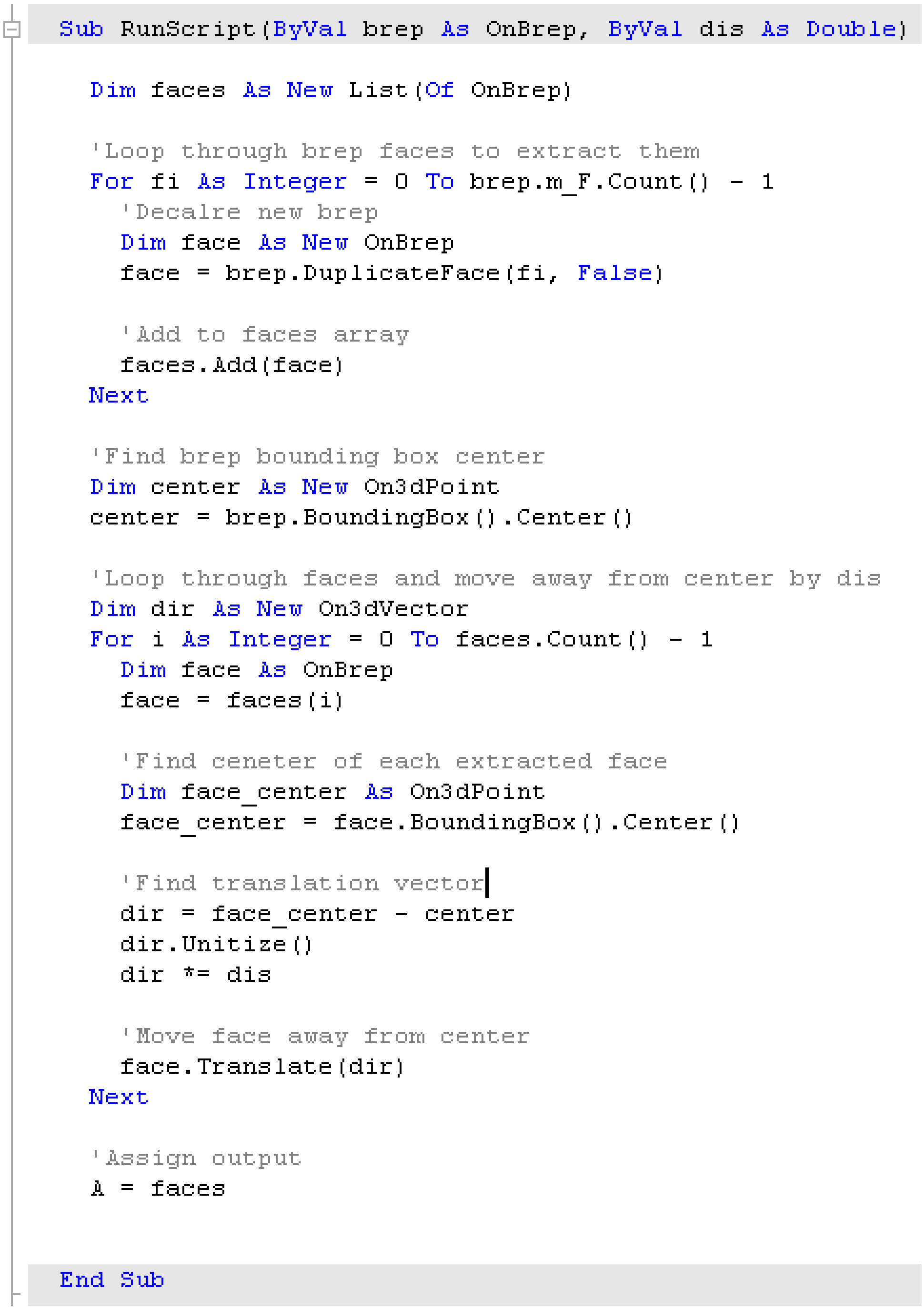
Most of OnBrep class member functions are expert user tools to create and edit breps. There are however many global function to Boolean, intersect or split breps as we will illustrate in a separate section.

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.



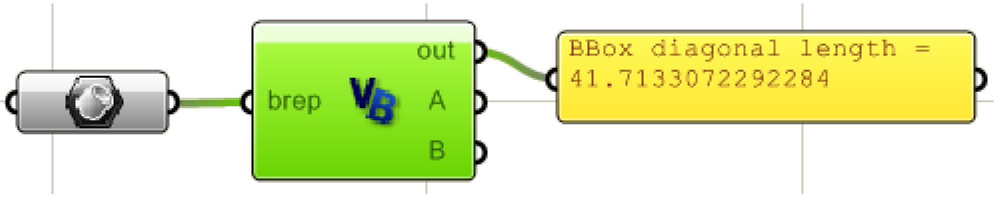




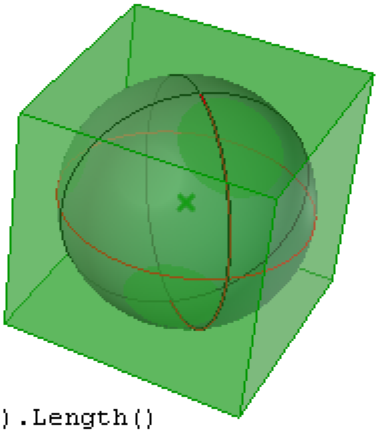
```
Sub RunScript(ByVal brep As OnBrep, ByVal dis As Double)

  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.

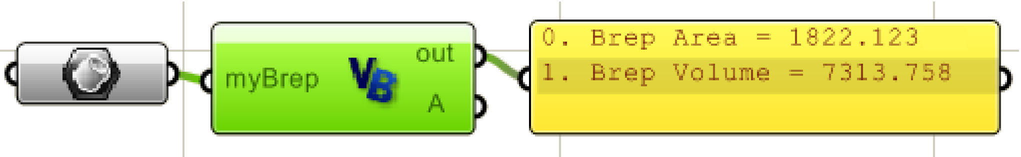


```
BBox diagonal length =
41.7133072292284
```



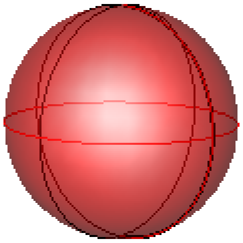
```
Sub RunScript(ByVal brep As OnBrep)

    '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:



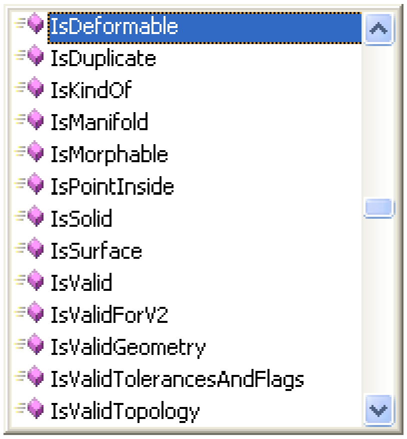
```
0. Brep Area = 1822.123
1. Brep Volume = 7313.758
```

```
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
```

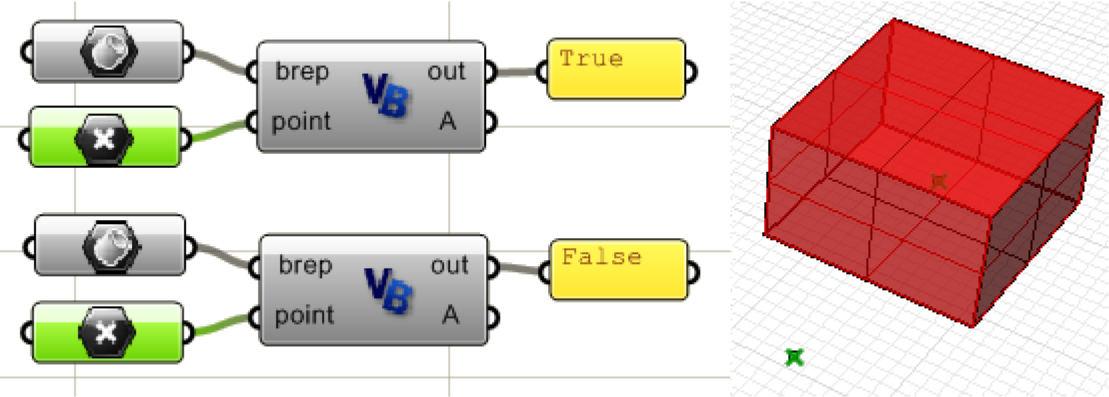


There are few functions that start with “Is” that usually retunes a Boolean (true or false). They inquire about the brep instance you are working with. For example if you like to know whether the brep is closed polysurface, then use OnBrep.IsSolid() function. It is also useful to check if the brep is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

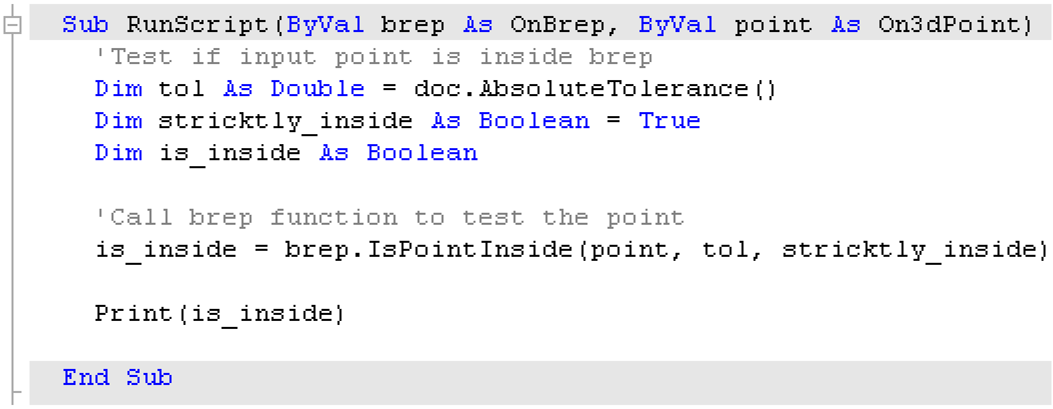
```
Dim brep As New OnBrep
brep.Is…
```



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



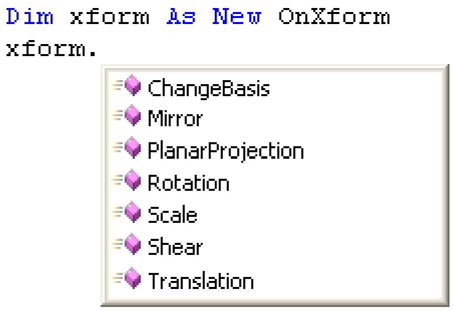
Here is the code to check if a point is inside:



```
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
```

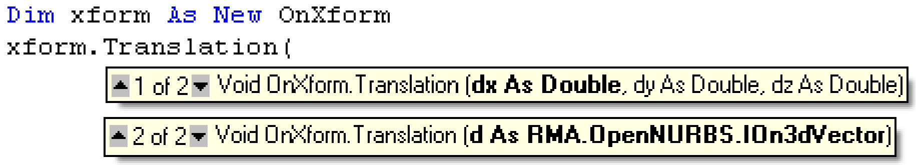
15.12 Geometry Transformation

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.

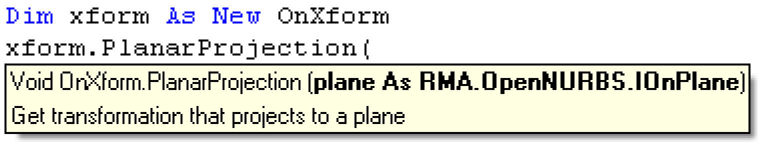


```
Dim xform As New OnXform
xform.Translation(
```

1 of 2 Void OnXform.Translation (**dx As Double**, dy As Double, dz As Double)

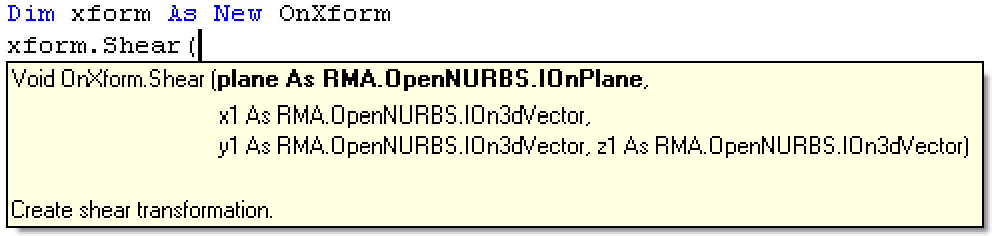
2 of 2 Void OnXform.Translation (**d As RMA.OpenNURBS.IOn3dVector**)

Here are few more OnXform functions:



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

Void OnXform.PlanarProjection (**plane As RMA.OpenNURBS.IOnPlane**)
Get transformation that projects to a plane

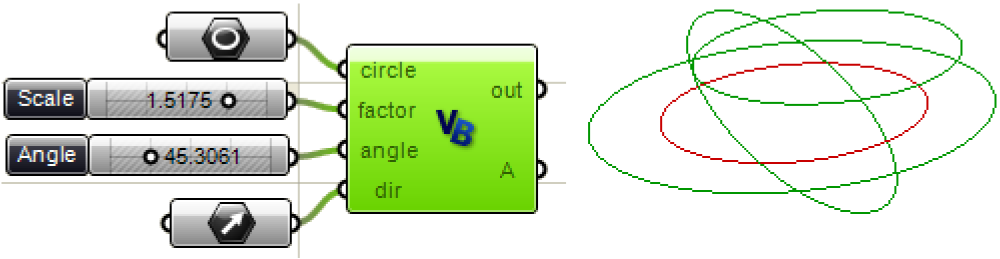


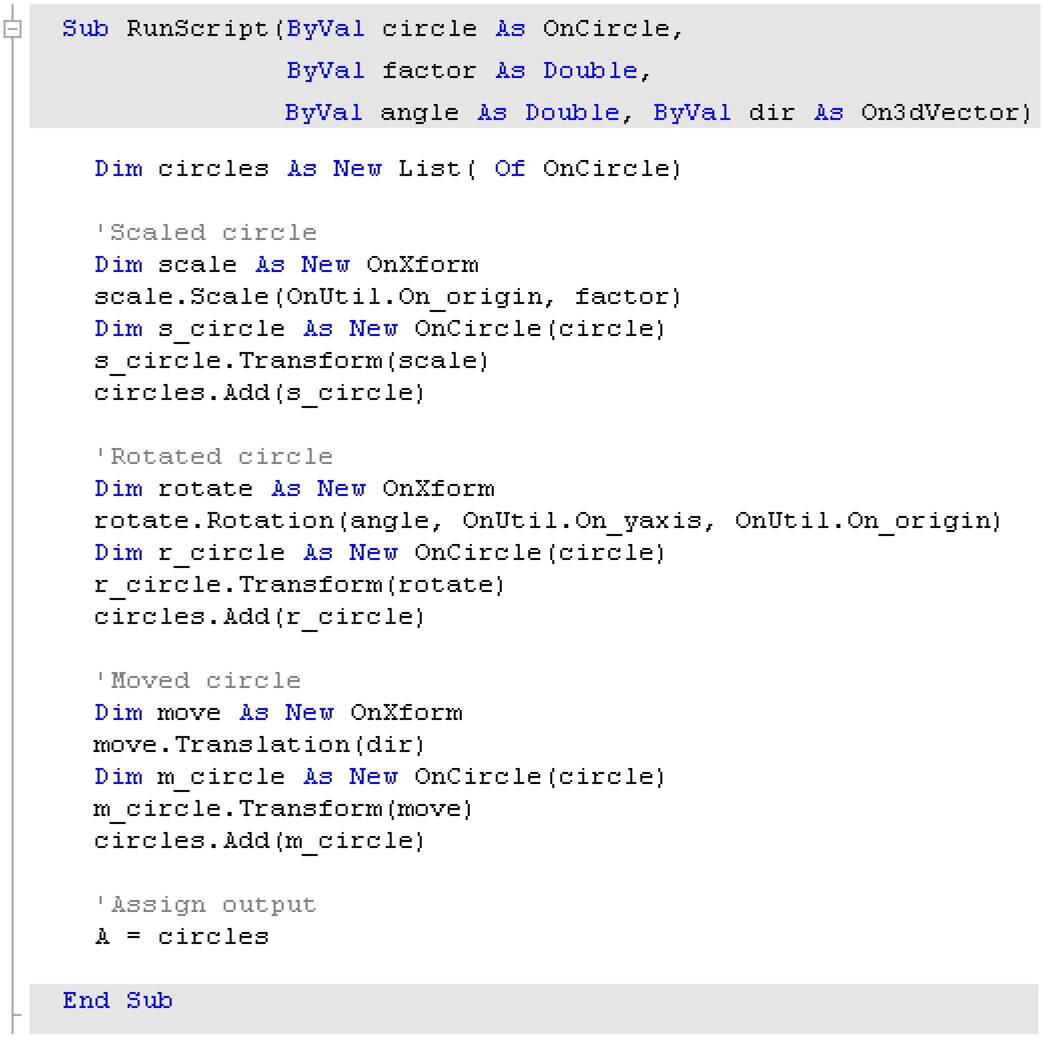
```
Dim xform As New OnXform
xform.Shear(
```

Void OnXform.Shear (**plane As RMA.OpenNURBS.IOnPlane**,
x1 As RMA.OpenNURBS.IOn3dVector,
y1 As RMA.OpenNURBS.IOn3dVector, z1 As RMA.OpenNURBS.IOn3dVector)

Create shear transformation.

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.





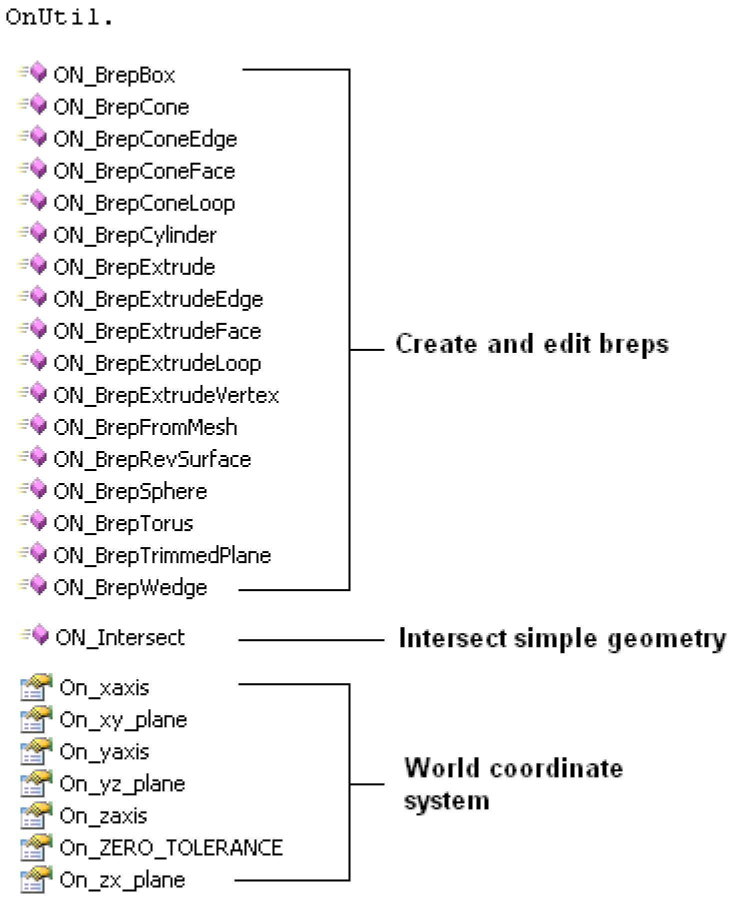
```
Sub RunScript(ByVal circle As OnCircle,
              ByVal factor As Double,
              ByVal angle As Double, ByVal dir As On3dVector)

  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.

ON_BrepBox
ON_BrepCone
ON_BrepConeEdge
ON_BrepConeFace
ON_BrepConeLoop
ON_BrepCylinder
ON_BrepExtrude
ON_BrepExtrudeEdge
ON_BrepExtrudeFace
ON_BrepExtrudeLoop
ON_BrepExtrudeVertex
ON_BrepFromMesh
ON_BrepRevSurface
ON_BrepSphere
ON_BrepTorus
ON_BrepTrimmedPlane
ON_BrepWedge

Create and edit breps

ON_Intersect

Intersect simple geometry

On_xaxis
On_xy_plane
On_yaxis
On_yz_plane
On_zaxis
On_ZERO_TOLERANCE
On_zx_plane

World coordinate system

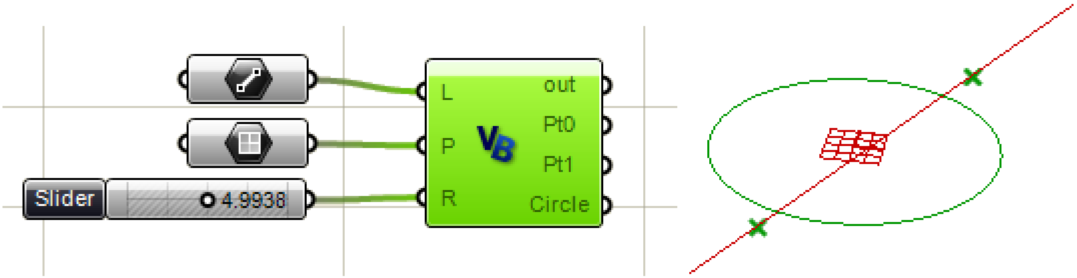
OnUtil intersections

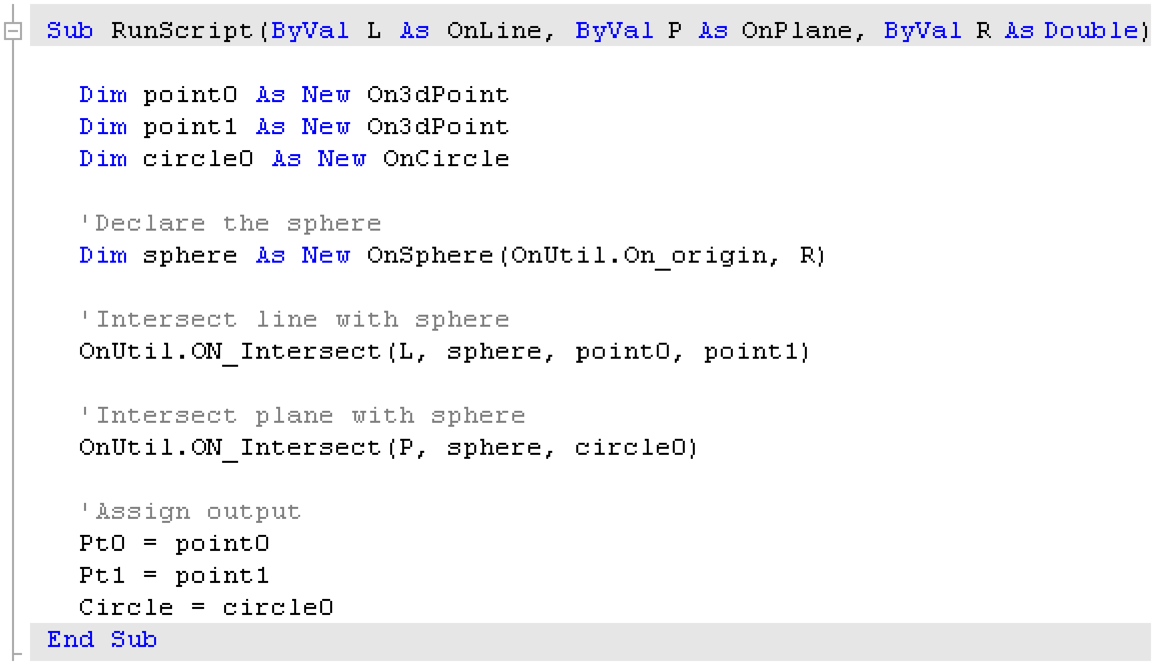
ON_Intersect utility function has 11 overloaded functions. Here is a list of intersected geometry and the return values (the preceding “I” like in “IOnLine” means a constant instance is passed):

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:



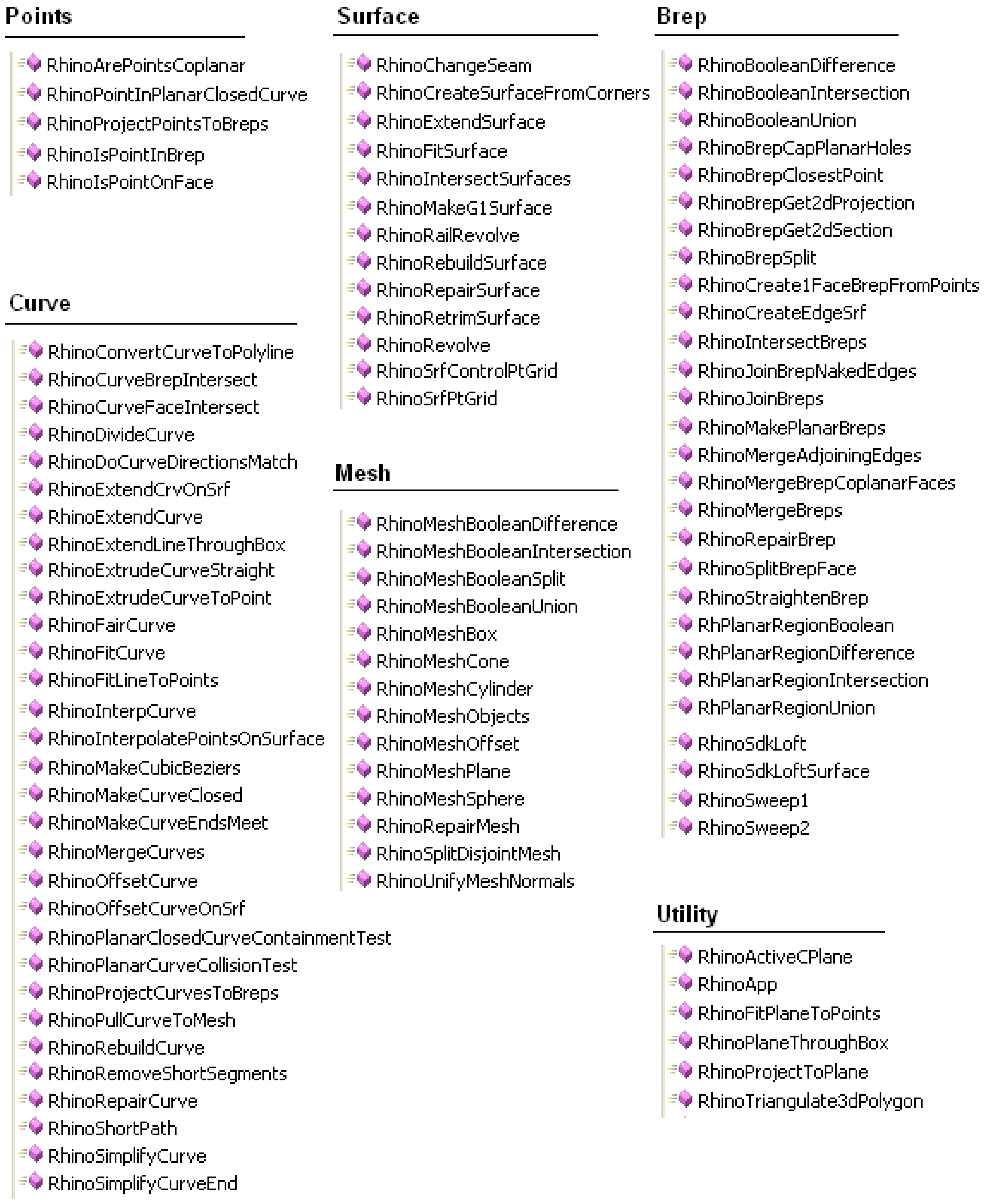


```
Sub RunScript(ByVal L As OnLine, ByVal P As OnPlane, ByVal R As Double)

  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

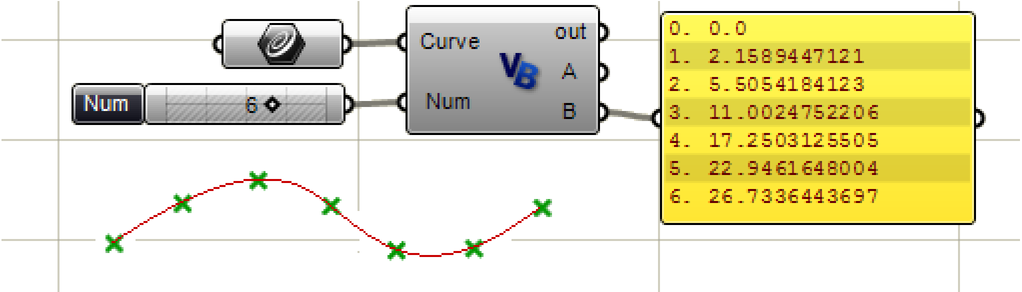
- RhinoActiveCPlane
- RhinoApp
- RhinoFitPlaneToPoints
- RhinoPlaneThroughBox
- RhinoProjectToPlane
- RhinoTriangulate3dPolygon

RhUtil Divide curve

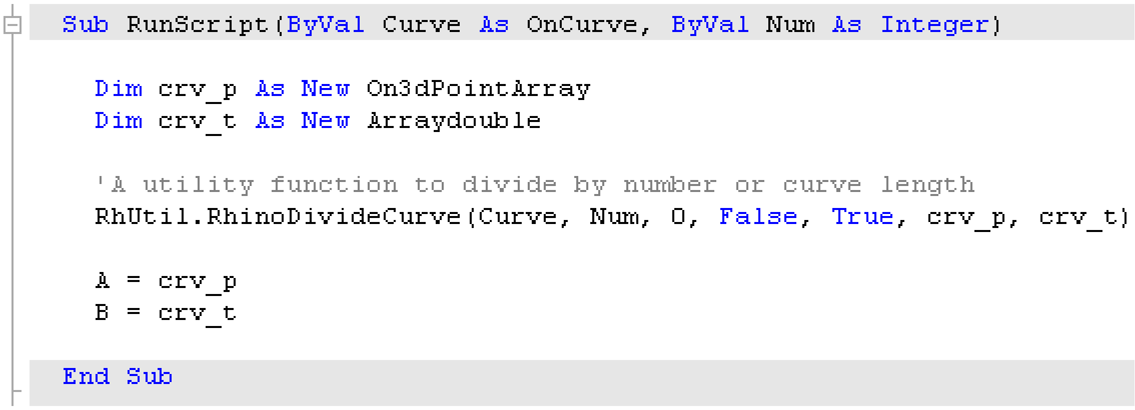
It is possible to divide a curve by number of segments or length on curve using the utility function RhUtil.RhinoDivideCurve. This is a breakdown of function parameters:

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:

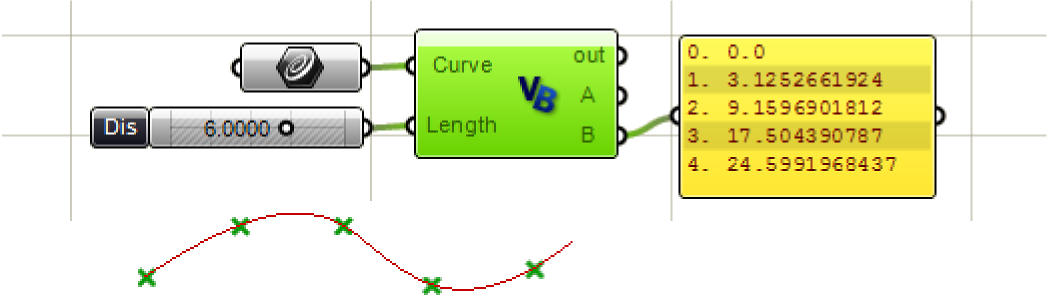


```
0. 0.0
1. 2.1589447121
2. 5.5054184123
3. 11.0024752206
4. 17.2503125505
5. 22.9461648004
6. 26.7336443697
```

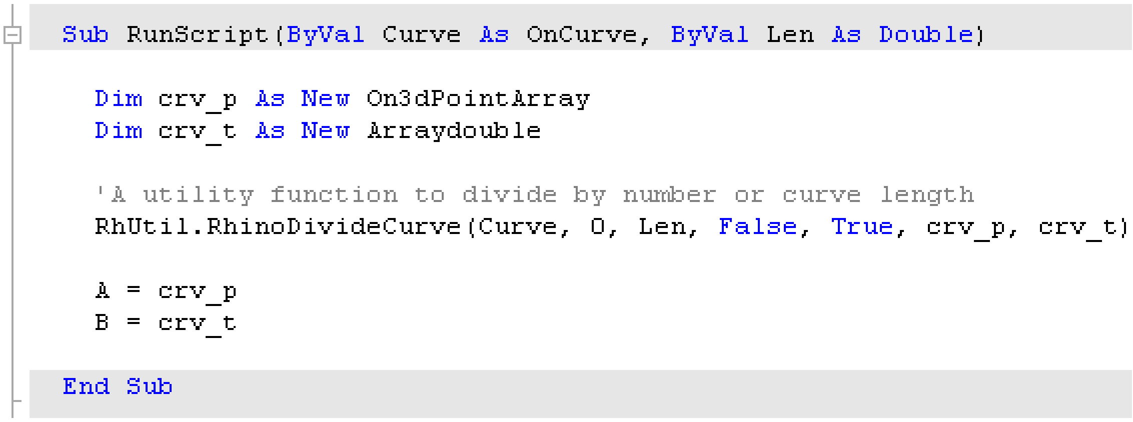


```
Sub RunScript(ByVal Curve As OnCurve, ByVal Num As Integer)

  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:



```
0. 0.0
1. 3.1252661924
2. 9.1596901812
3. 17.504390787
4. 24.5991968437
```

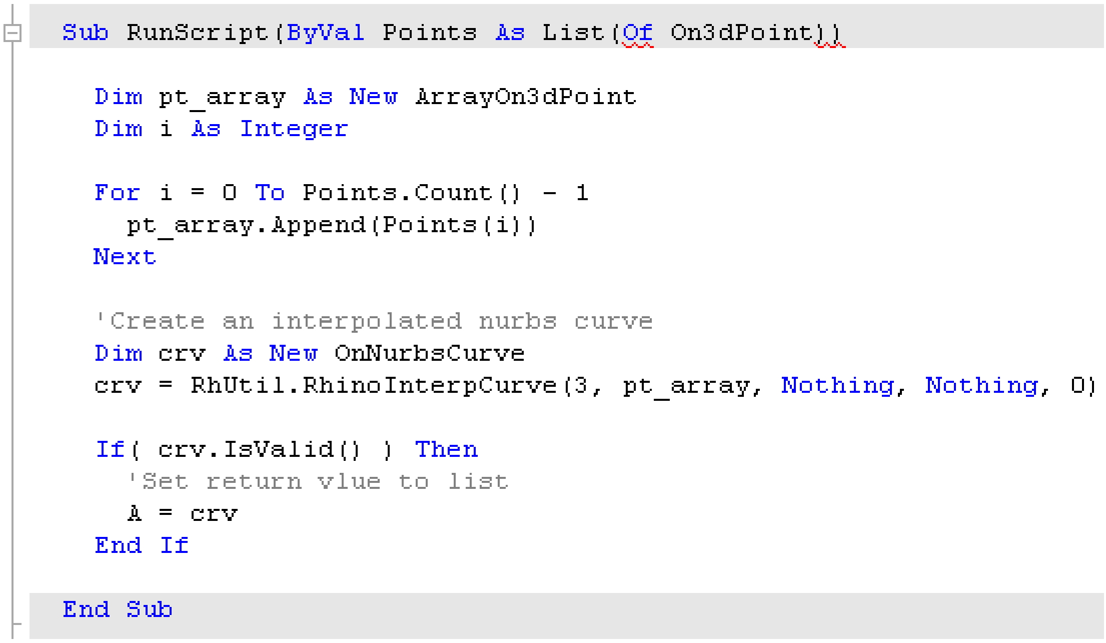


```
Sub RunScript(ByVal Curve As OnCurve, ByVal Len As Double)

  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

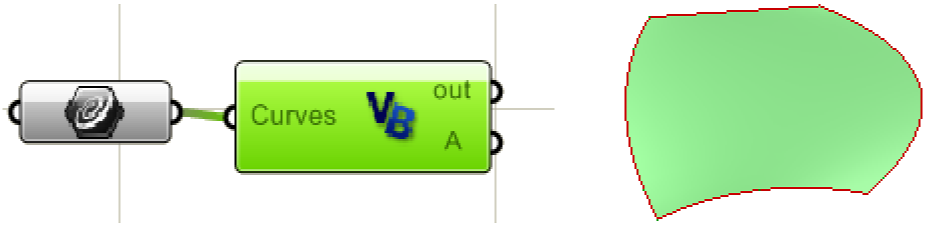
The following example takes as an input a list of On3dPoints and outputs a nurbs curve that goes through these points.

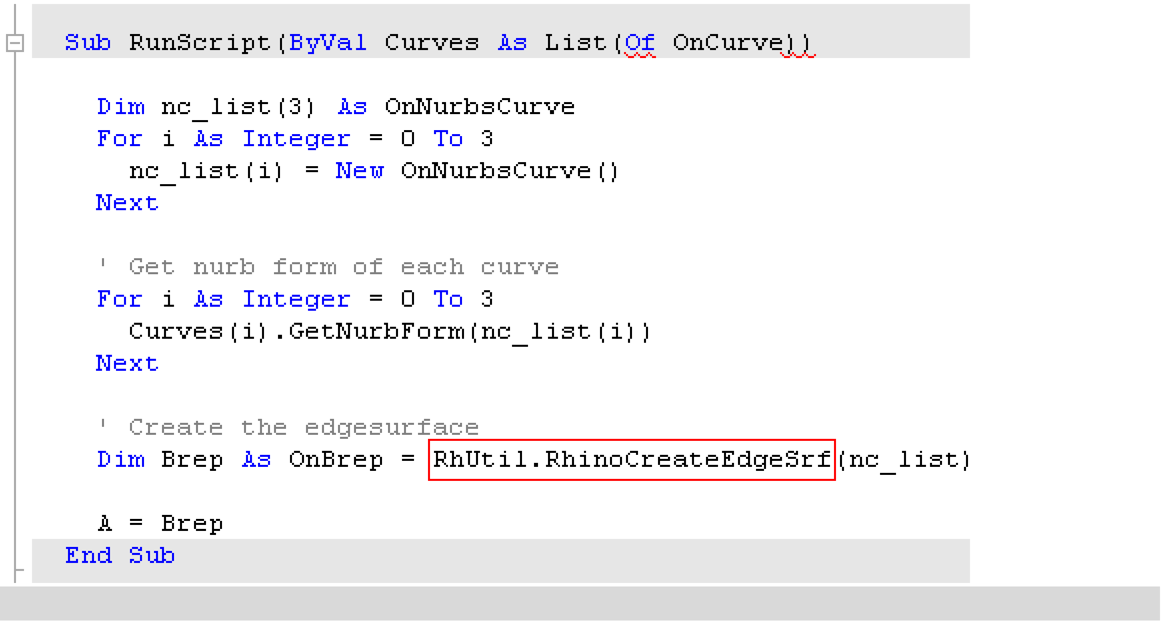


```
Sub RunScript(ByVal Points As List(Of On3dPoint))

  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.

Notes