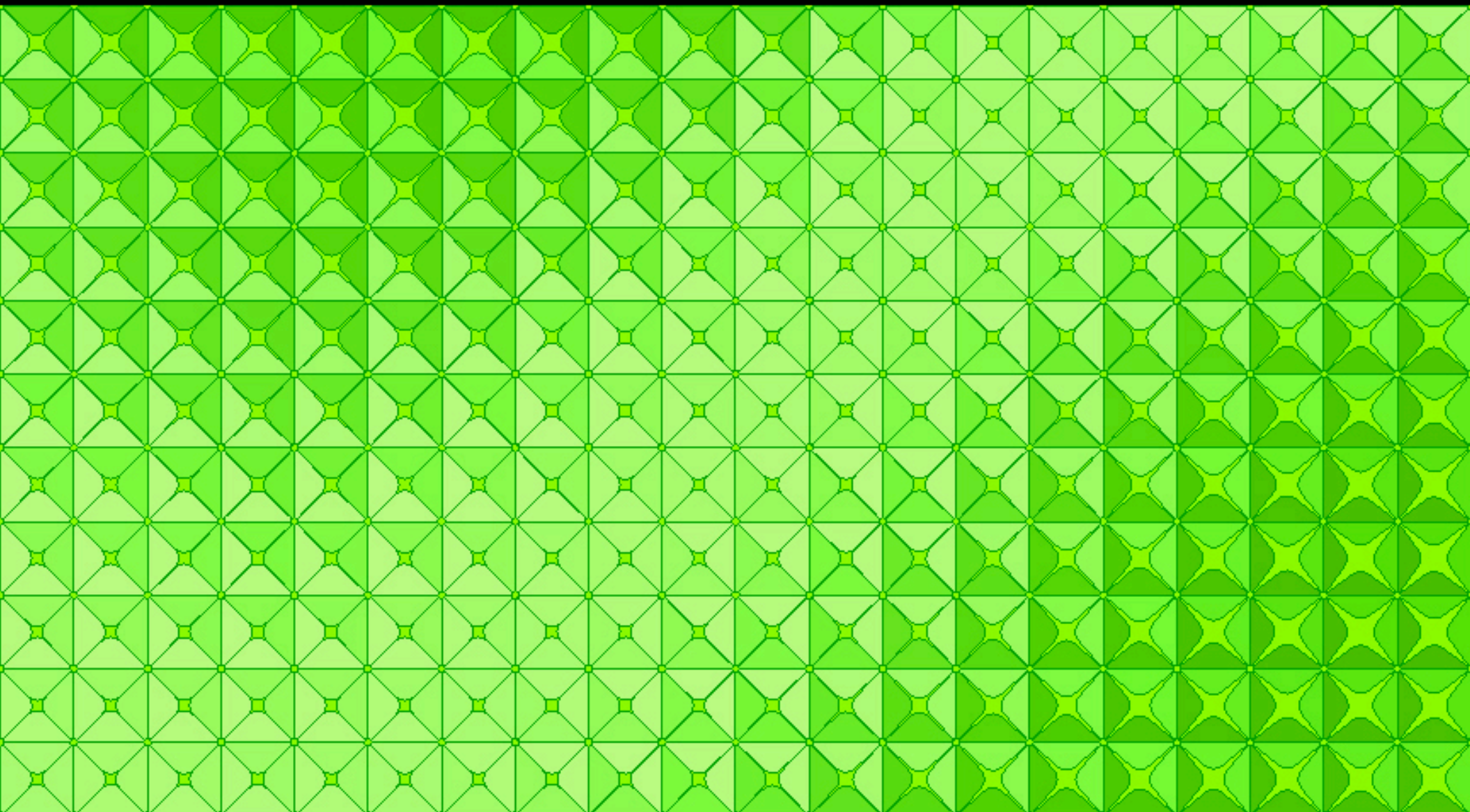
GRASSHOPPER
PRIMER
FOR RHINOCEROS 4.0 SR4



Introduction

Welcome to the wonderful new world of Grasshopper. This is the first printing of this primer and it was originally put together as a handout for a workshop I was teaching at the California College of the Arts. In compiling this manual, I found that there was a lot of information and online tutorials out there based on the Grasshopper plugin, but there wasn't *ONE* definitive guide where someone who was just beginning their trek into the unknown could get a good understanding on how to navigate this fantastic new tool.

I have to be honest. Some of the information and many of the tutorials were created by other great designers, and I cite my sources as needed. Often, there are links at the end of a tutorial to an online video that should be used as a complement to this manual. The first 20 pages or so came straight from the RhinoWiki page, which is a wonderful resource and should be consulted early and often in your search for more information about the plugin. Another great source of information is the Grasshopper Forum. Here, you'll find a growing online community dedicated to furthering this tool by sharing their questions, concerns, solutions, and definitions to help others out there just like you. I would encourage each person who reads this primer to join the group and post your questions online, as there is almost always someone willing to share a solution to your problem. To learn more, visit http://www.grashopper.rhino3d.com

One of the greatest things about this software is that it is always evolving. The developers often make changes and add new features, which may or may not conflict with older definitions. It can make it frustrating and even difficult to keep up with the speed at which things are moving. This primer was written to work with Grasshopper version 0.5.0099, which was released in December 2008, and I have no doubt that in even just a few short weeks it will be outdated. However, I will continue to try to keep this primer as up to date as possible and add more tutorials as time allows.

It is my hope that this manual will help beginners get a better understanding of the software, but I by no means expect it to become the Grasshopper Bible. There may very well be some small errors, here or there, within the primer and I would ask that you bring those to my attention should you see any inconsistencies in the material. It will only help the manual become more robust and in turn help others out there as well. That's about it. I believe it's time to jump into the material.

Thanks, and good luck!

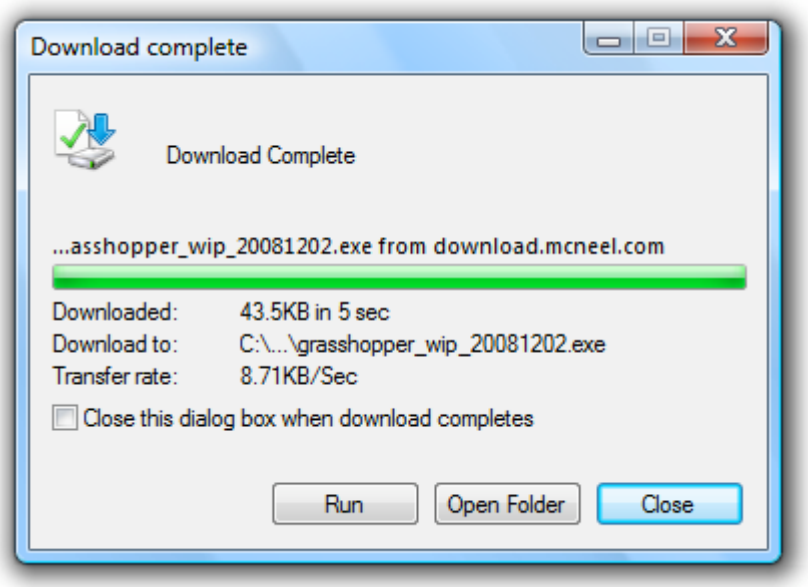
Andy Payne
LIFT architects
www.liftarchitects.com

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.

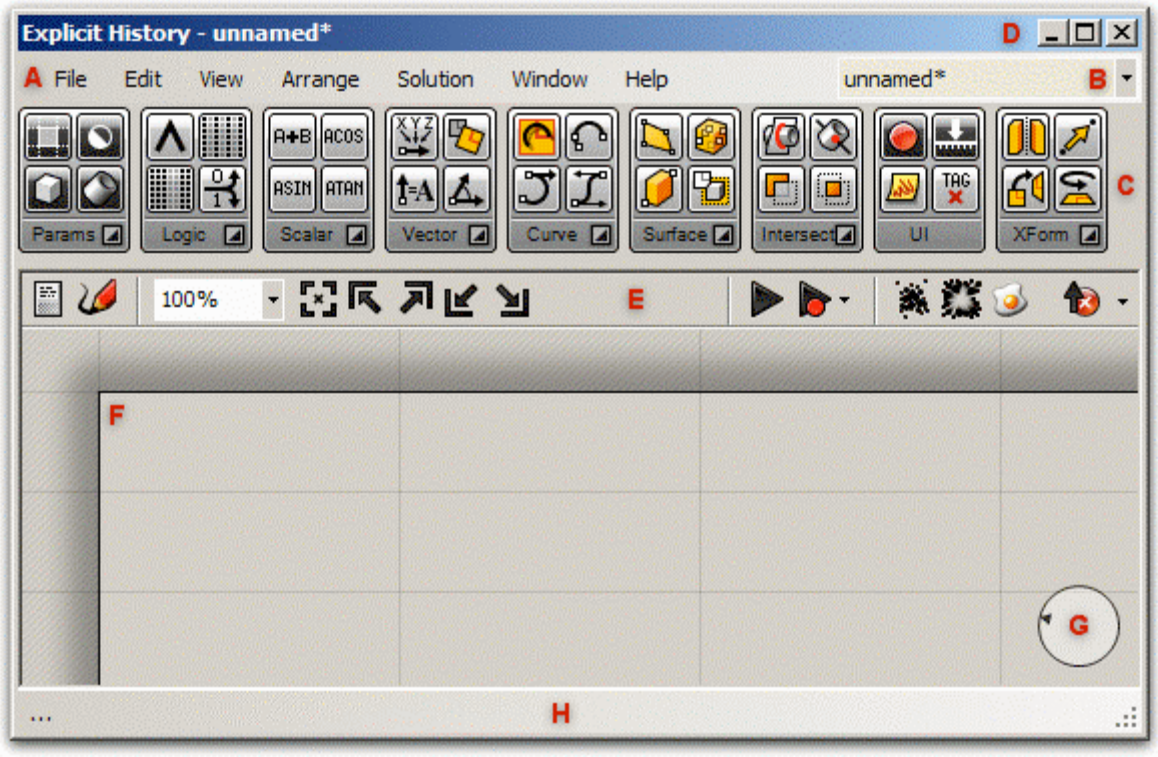


Select **Run** from the download dialogue box follow the installer instructions. (note: you must have Rhino 4.0 with the SR4b patch already installed on your computer for the plug-in to install properly)

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.

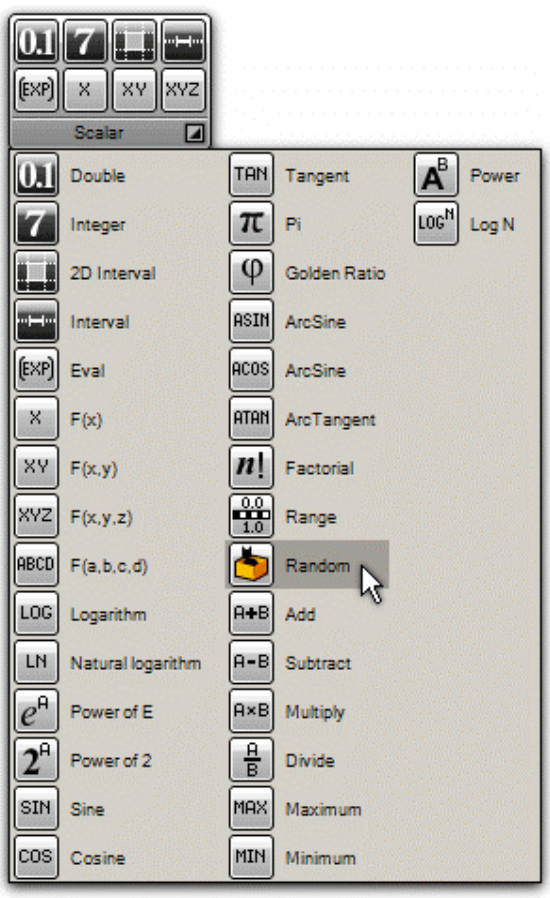
C. Component Panels

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

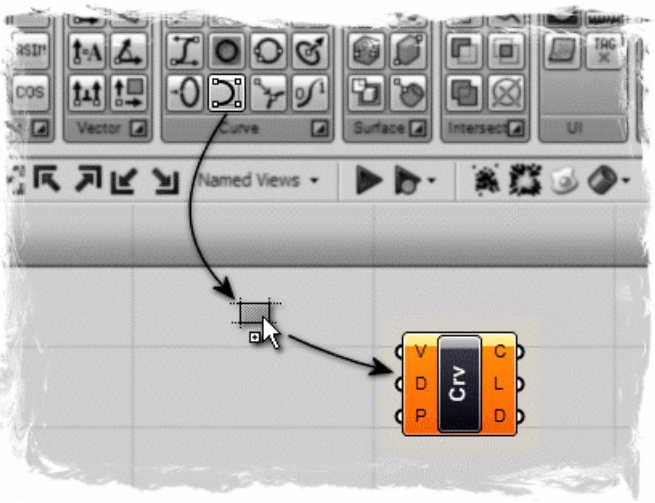
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryPluginInterfaceExplained.html

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.

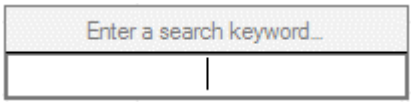
The toolbar panels themselves contain all the components that belong to that category. Since there are a potentially large number of these, it only shows the N most recently used items. In order to see the entire collection, you have to click on the bar at the bottom of the Panel:



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.



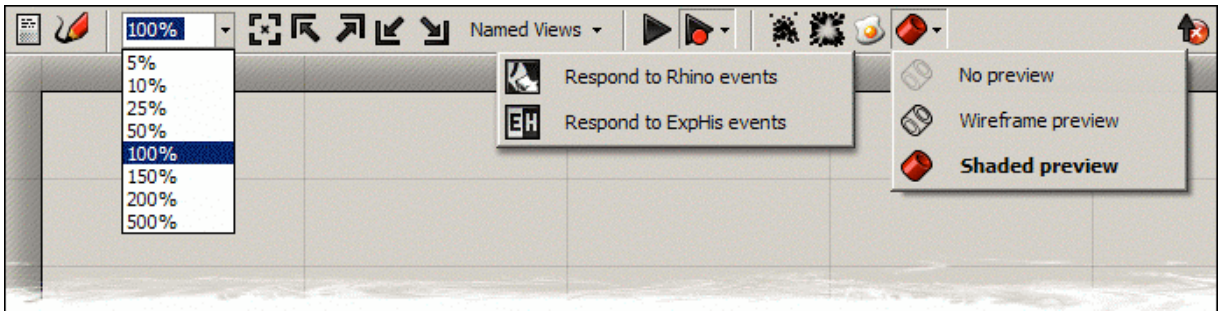
The Window Title Bar: D

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).

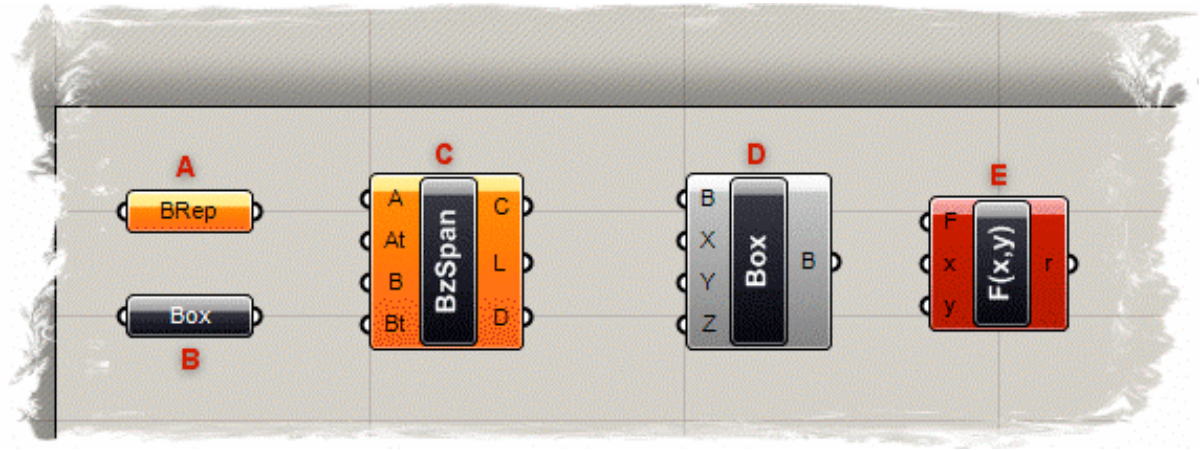


The canvas toolbar exposes the following tools (from left to right):

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. Zoom Extents (will adjust the zoom-factor if the definition is too large to fit on the screen)
5. Focus corners (these 4 buttons will focus on the 4 corners of the definition)
6. Named views (exposes a menu to store and recall named views)
7. Rebuild solution (forces a complete rebuild of the history definition)
8. Rebuild events (by default, Grasshopper responds to changes in Rhino and on the Canvas. You can disable these responses though this menu)
9. Cluster compactor (turn all selected objects into a Cluster object) **Cluster objects are not finished yet.**
10. Cluster exploder (turn all selected clusters into loose objects) **Cluster objects are not finished yet.**
11. Bake tool (turns all selected Explicit History geometry into actual Rhino objects)
12. 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
13. 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.

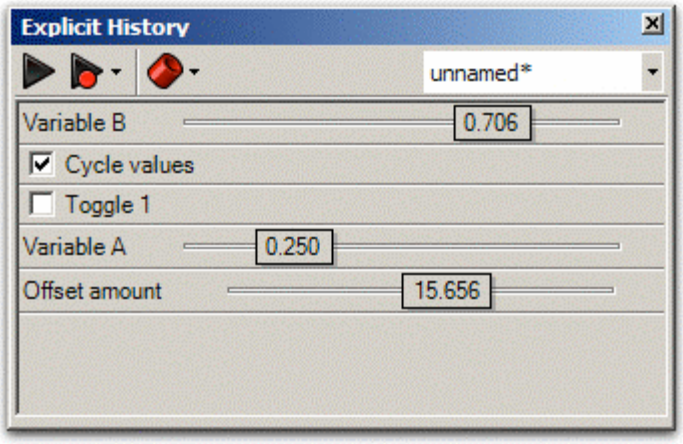
H: The Status Bar

The status bar provides feedback on the selection set (not yet implemented) and the major events that have occurred in the plug-in. You can see a list of all recent events by right clicking on the ellipsis symbol in the status bar.
The square orange icon on the bottom left of the Status Bar is a recent addition to the interface. 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/**

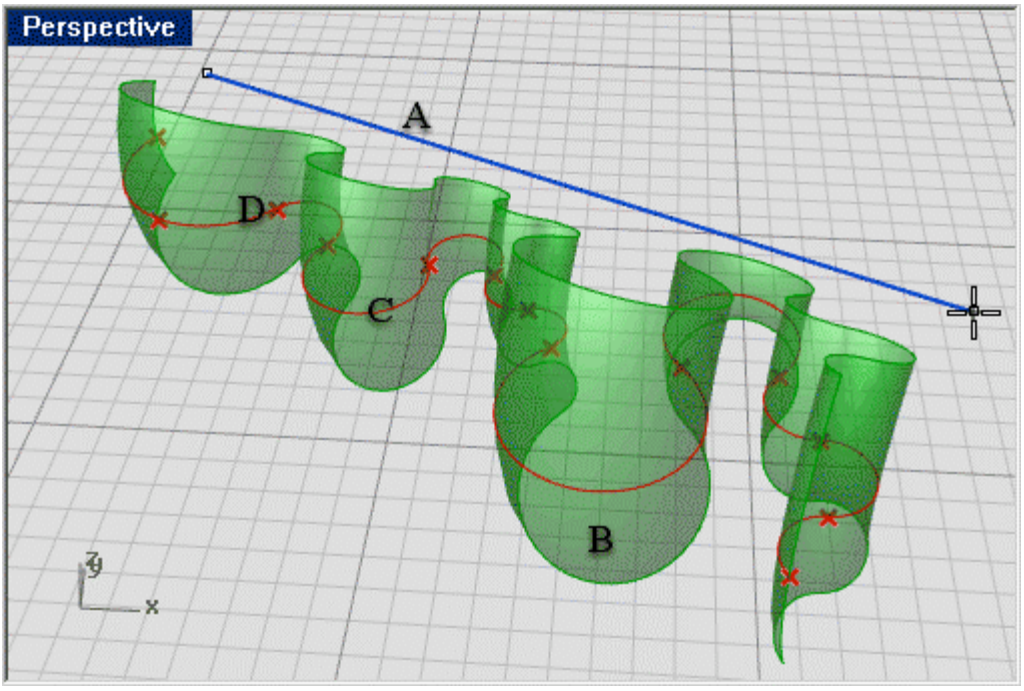
The Remote Control Panel:

Since the Explicit History 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):



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.

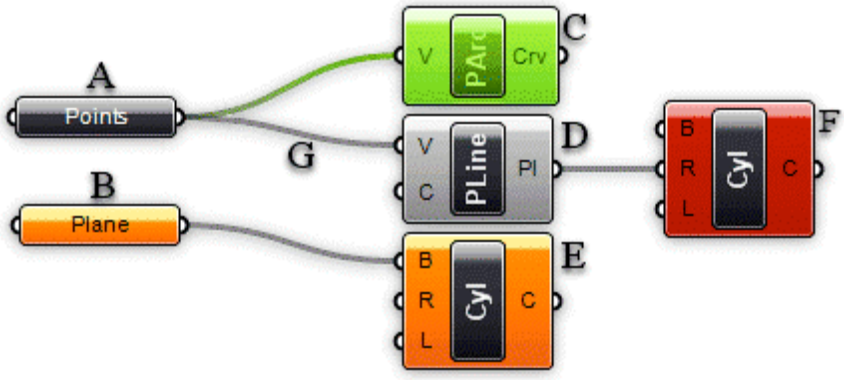
3 *Grasshopper Objects**

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.

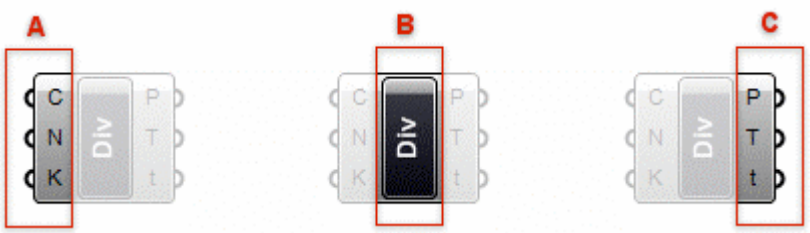
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryPluginObjectsExplained.html

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:

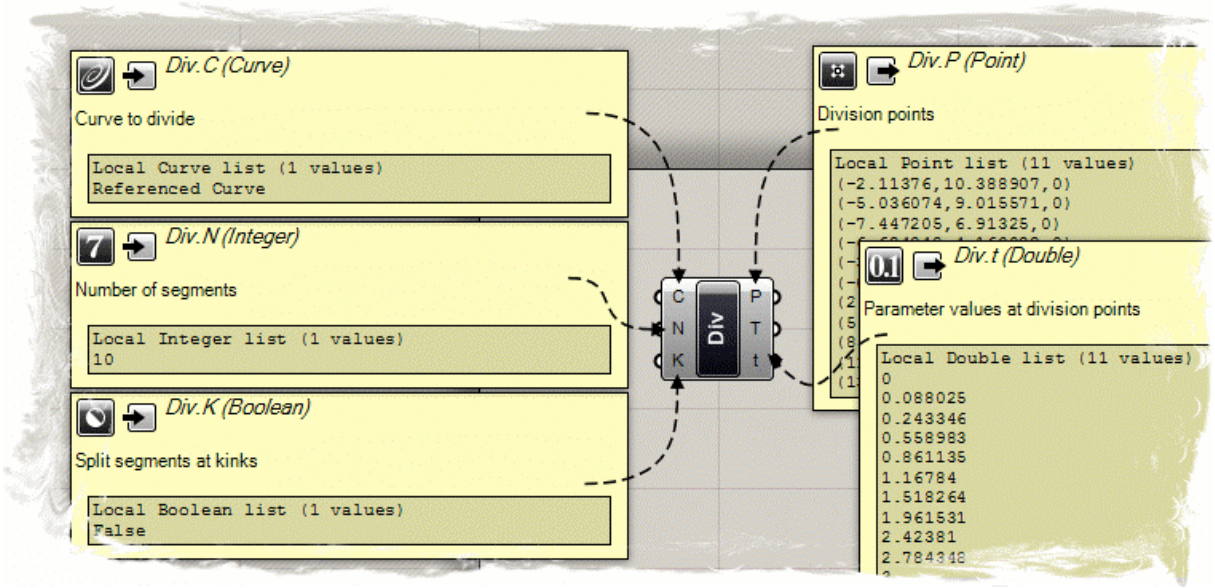


A) The three input parameters of the Division component. By default, parameter names are always extremely short. You can rename each parameter as you please.

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

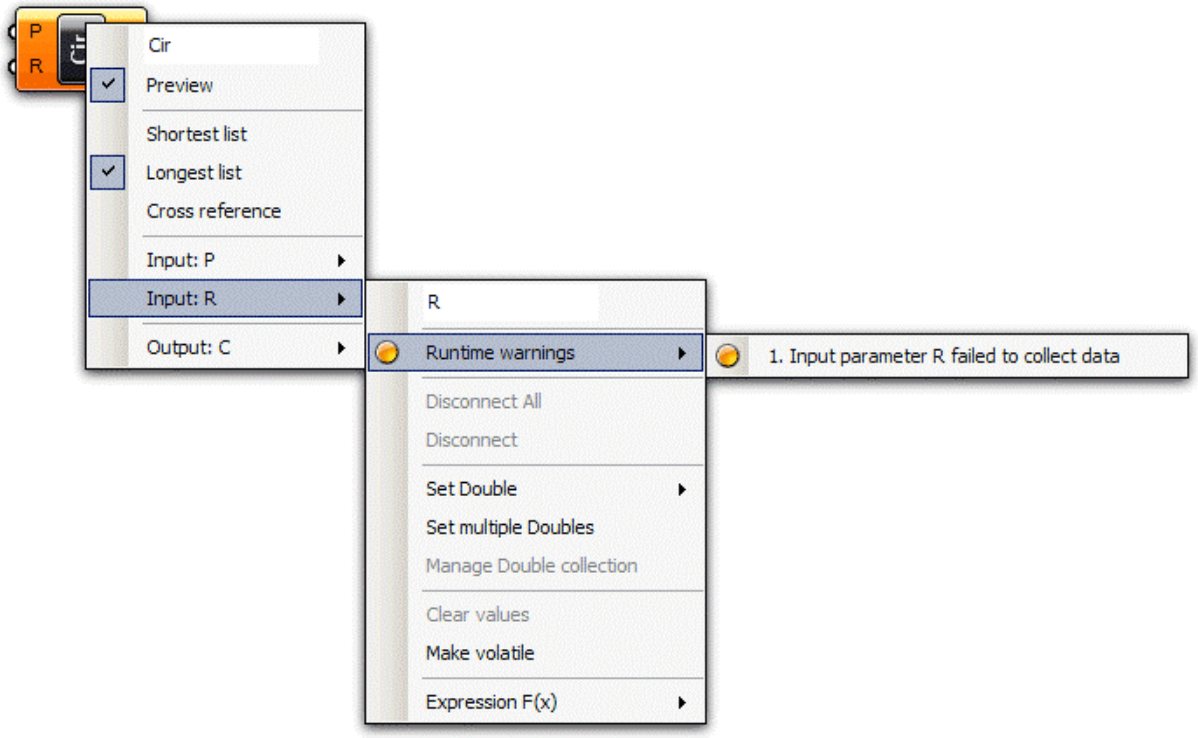
C) The three output parameters of the Division component.

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:



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:



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**

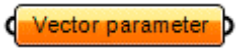
Types of Data

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.

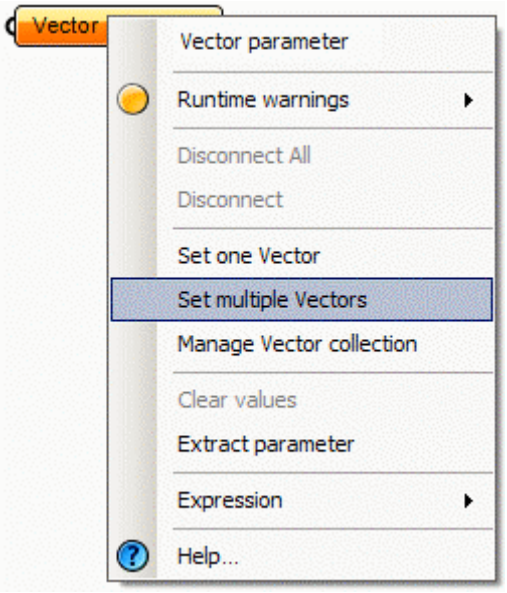
(The exception here are output parameters which can neither store permanent records nor define a set of sources. Output parameters are fully under the control of the component that owns them.)

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:

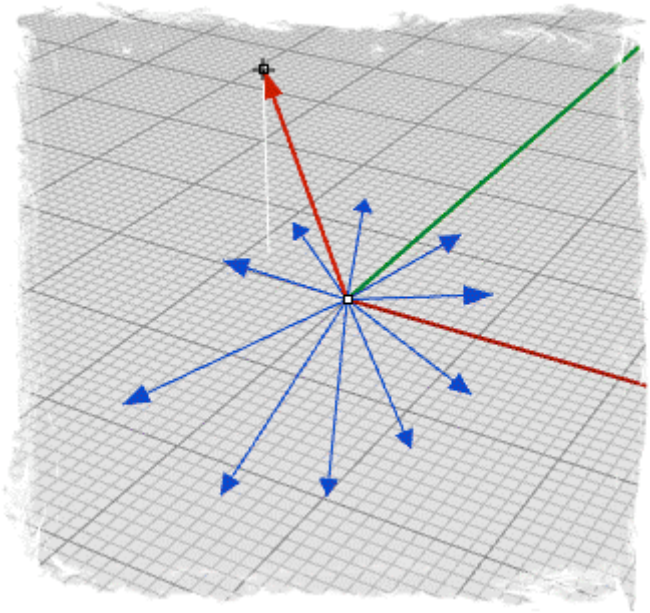


The parameter is orange, indicating it generated a warning. It's nothing serious, the warning is simply there to inform you that the parameter is empty (it contains no persistent records and it failed to collect volatile data) and thus has no effect on the outcome of a history solution. The context menu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

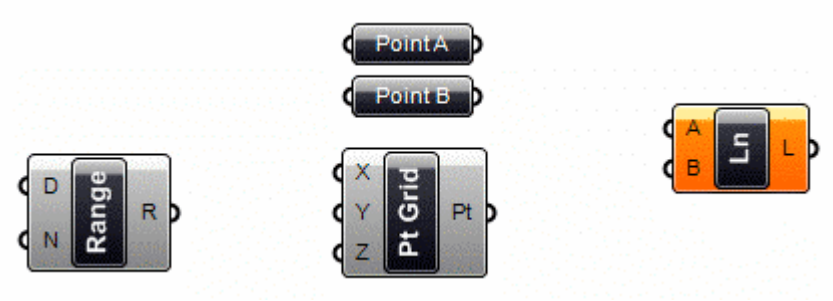
5 *Volatile Data Inheritance**

Data Inheritance

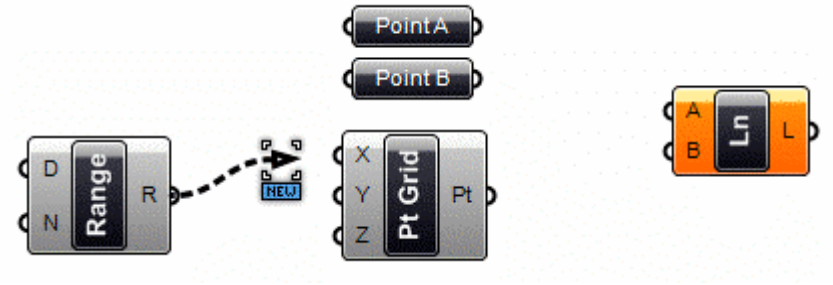
Data is stored in parameters (either in Volatile or Persistent form) and used in components. When data is not stored in the permanent record set of a parameter, it must be inherited from elsewhere. Every parameter (except output parameters) defines where it gets its data from and most parameters are not very particular. You can plug a double parameter 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:

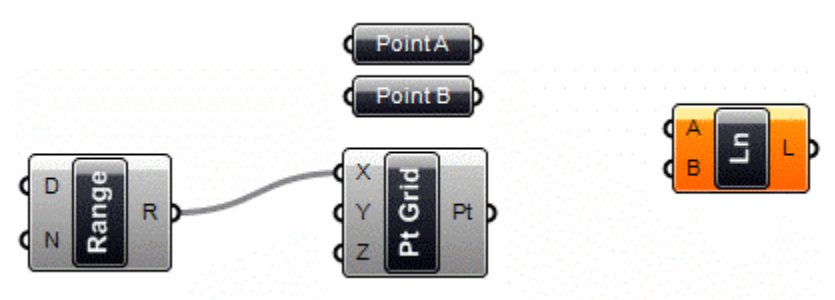


At this stage, all the objects are unconnected and we need to start hooking them up. It doesn't matter in what order we do this, but lets go from left to right. If you start dragging near the little circle of a parameter (what us hip people call a "grip") a connecting wire will be attached to the mouse:

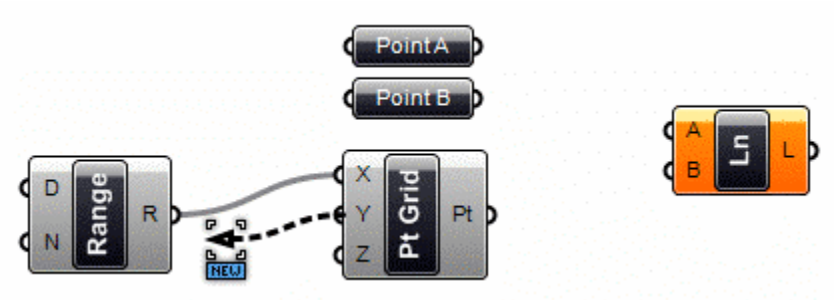


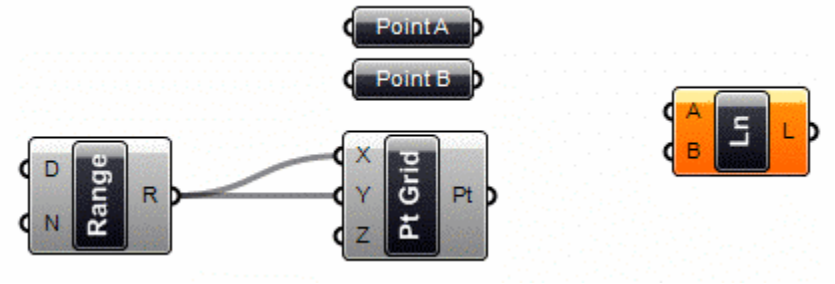
Once the mouse (with the Left Button still firmly pressed) hovers over a potential target Parameter, the wire will attach and become solid. This is not a permanent connection until you release the mouse button:

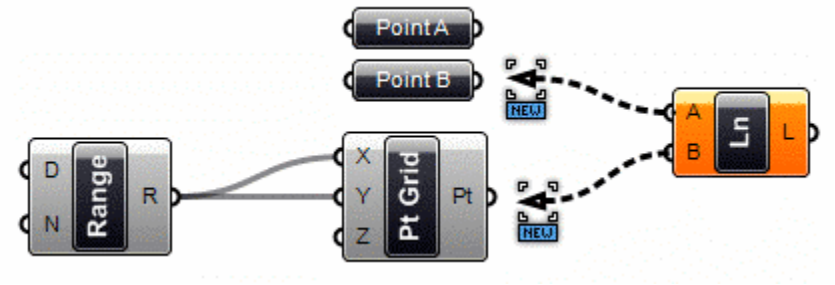
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryVolatileDataInheritance.html

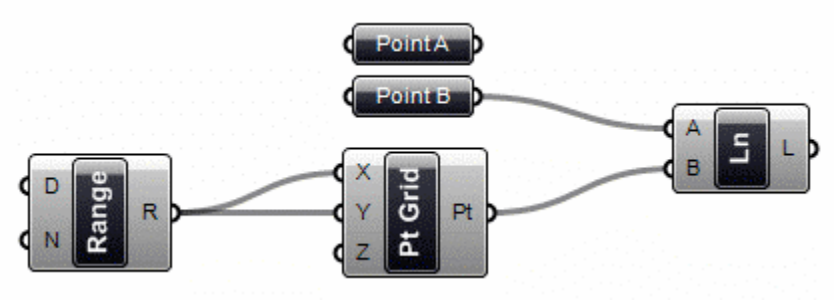


We can do the same for the "Y" parameter of the PtGrid component and the "A" and "B" parameters of the Line component: Click+Drag+Release...

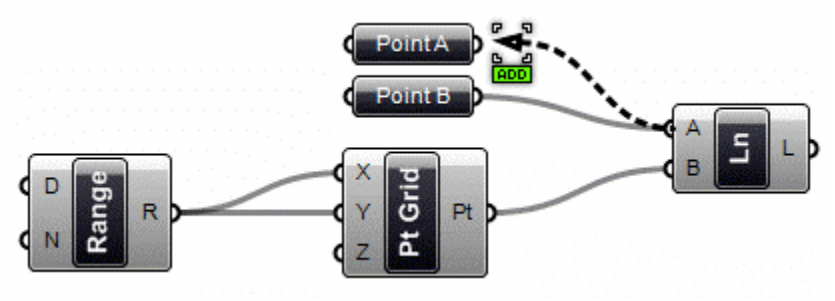


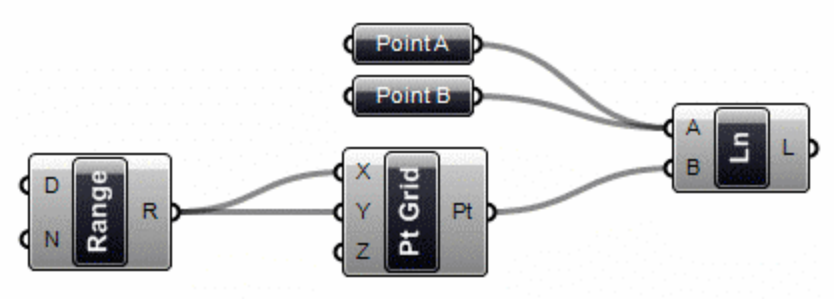






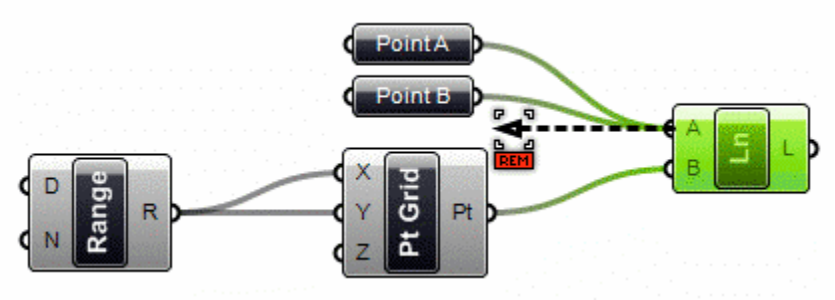
Note that we can make connections both ways. But be careful, by default a new connection will erase existing connections. Since we assumed that you will most often only use single connections, you have to do something special in order to define multiple sources. If you press Shift while dragging connection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

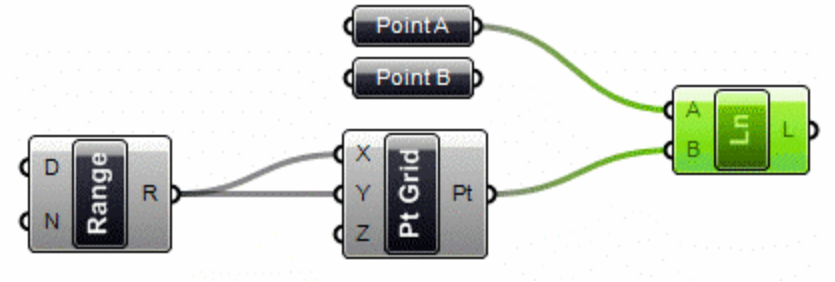




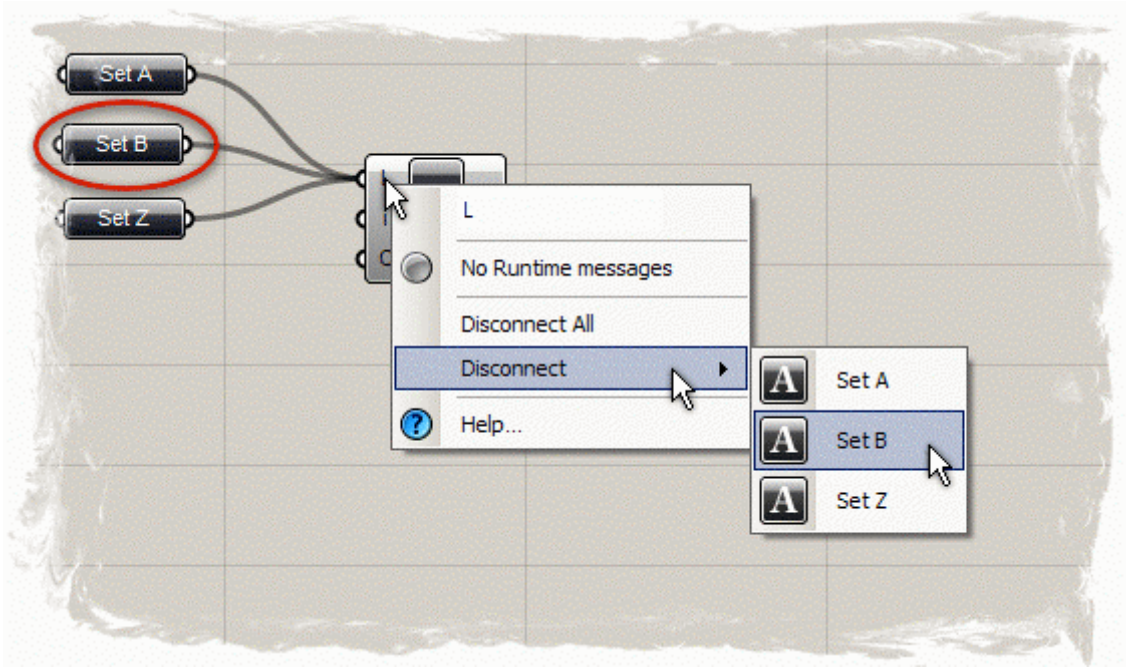
If the "ADD" cursor is active when you release the mouse button over a source parameter, that parameter will be added to the source list. If you specify a source parameter which is already defined as a source, nothing will happen. You cannot inherit from the same source more than once.

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:

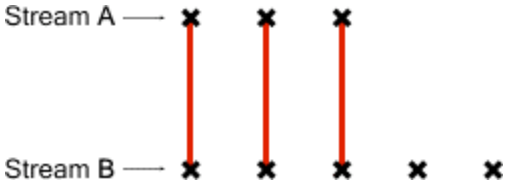


6 Data Stream Matching*

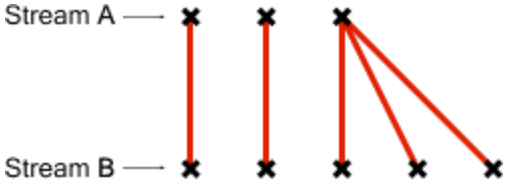
Data matching

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:

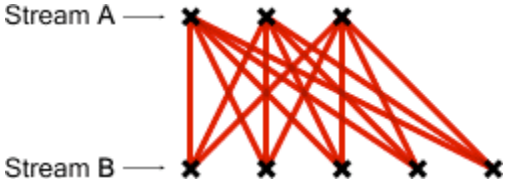
As you can see there are different ways in which we can draw lines between these sets of points. The Grasshopper plug-in currently supports three matching algorithms, but many more are possible. The simplest way is to connect the inputs one-on-one until one of the streams runs dry. This is called the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

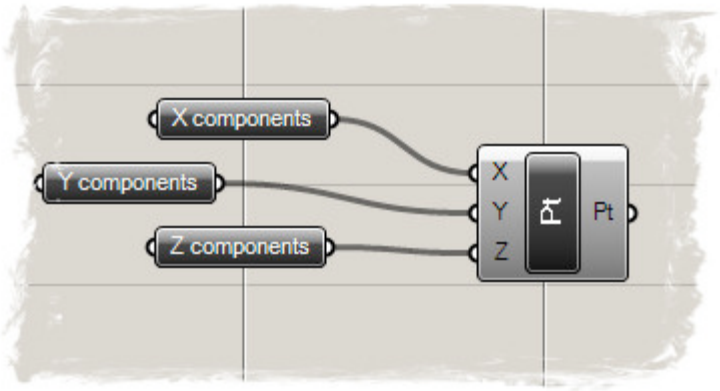


This is potentially dangerous since the amount of output can be humongous. The problem becomes more intricate as more input parameters are involved and when the volatile data inheritance starts to multiply data, but the logic remains the same.

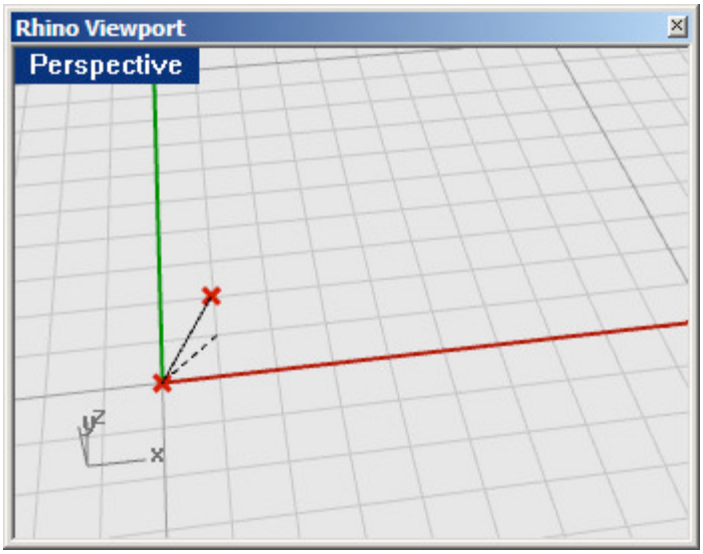
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryDataStreamMatchingAlgorithms.html

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

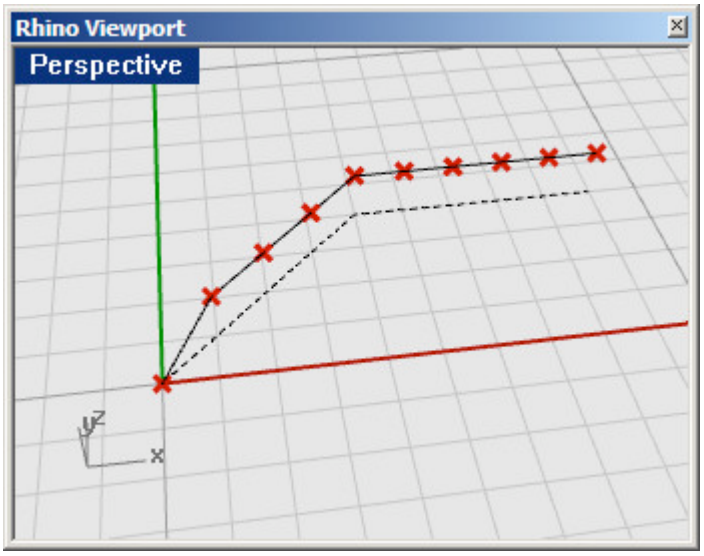
X coordinate: {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0}
Y coordinate: {0.0, 1.0, 2.0, 3.0, 4.0}
Z coordinate: {0.0, 1.0}



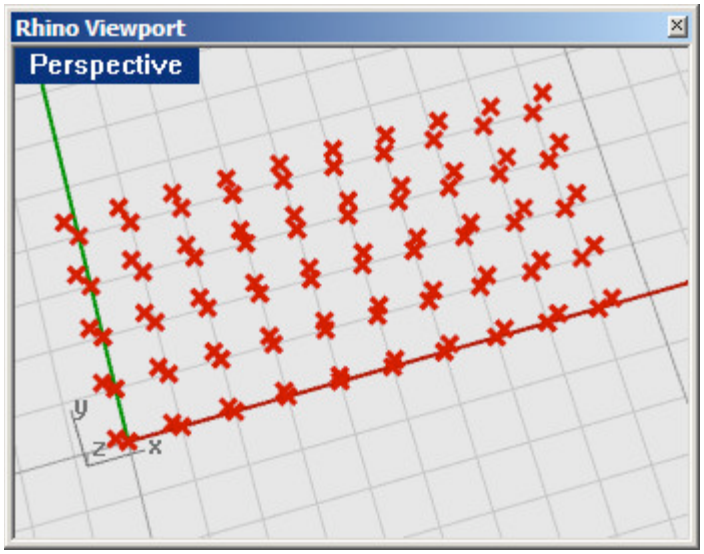
If we combine this data in the "Shortest List" fashion, we get only two points since the "Z coordinate" contains only two values. Since this is the shortest list it defines the extent of the solution:



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.

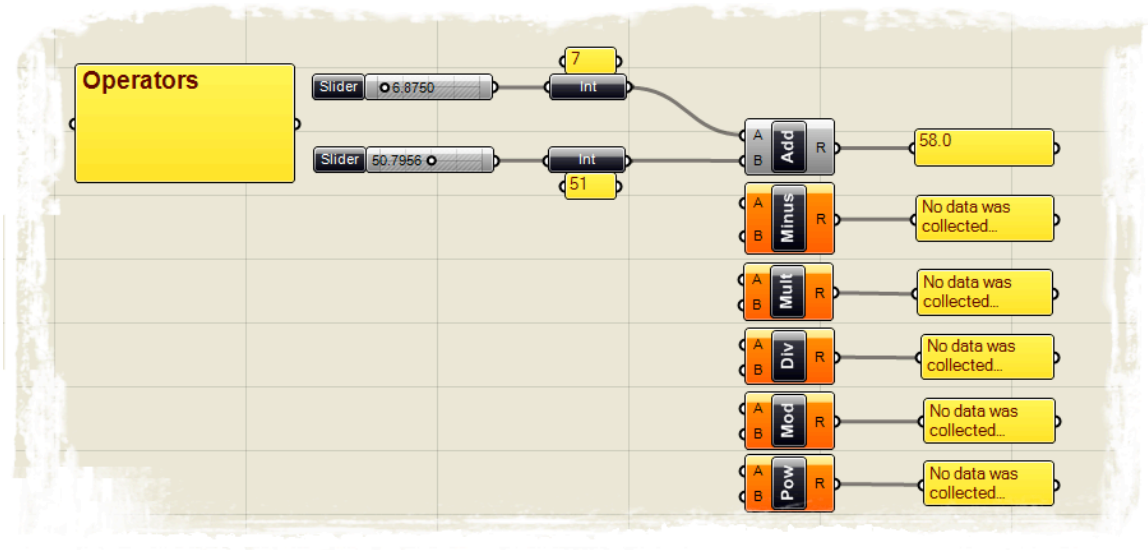
7 Scalar Component Types

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...
B) **Expressions**. Used to create single or multiple variable functions (algorithms).
C) **Intervals**. Used to divide two numeric extremes into interval parts.
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
- Right 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 Cntrl+C (copy) and Cntrl+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.
- 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

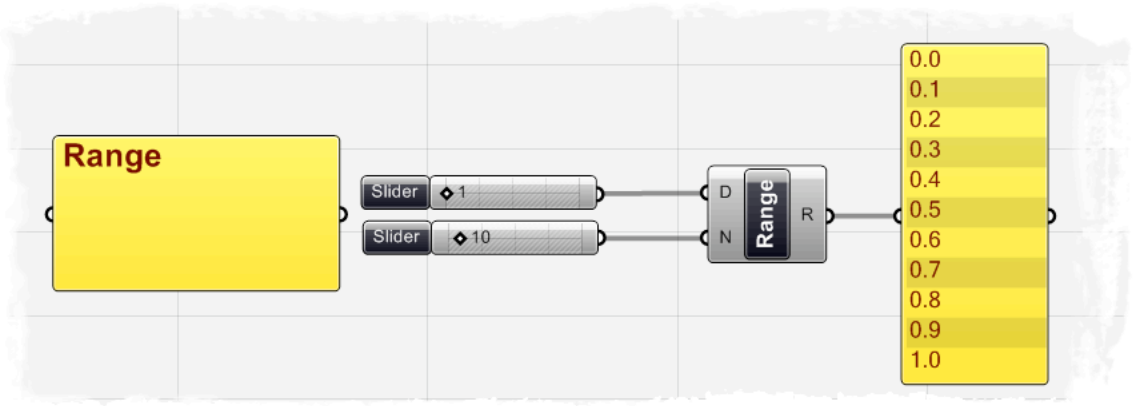
  You can now see the summation value of the two integers in the Post-it panel.
- Drag and drop the other remaining Scalar Operators onto the Canvas:
  - Subtraction
  - Multiplication
  - Division
  - Modulus
  - Power
- Connect the first Integer component to each of the Operator's-A input value
- Connect 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.

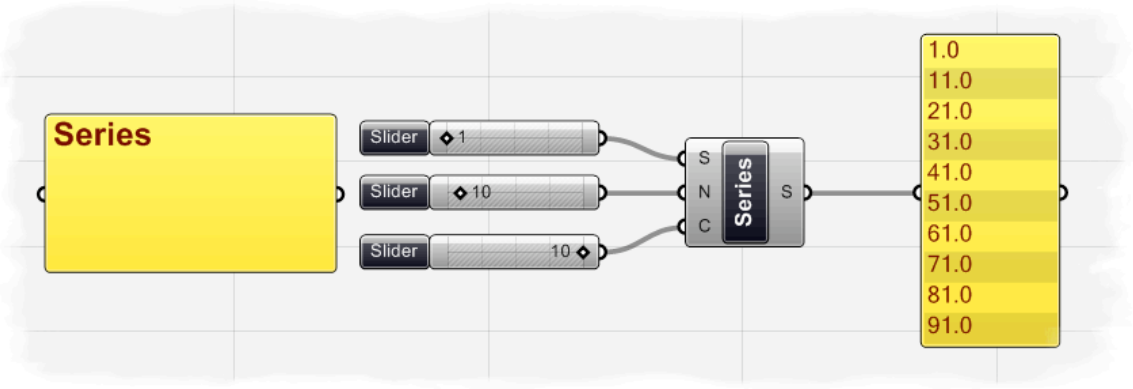
7.2 Range vs. Series vs. Interval

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.

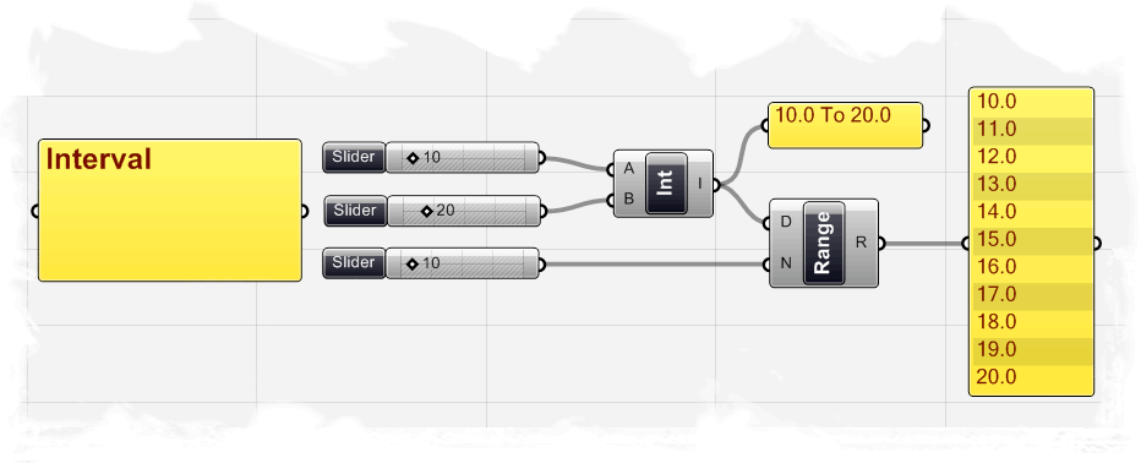


The Range component creates a list of evenly spaced numbers between a low and a high value called the domain of numeric range. In the example above, two numeric sliders are connected to the input values of the Range component. The first slider defines the numeric domain for the range of values. In this 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)



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

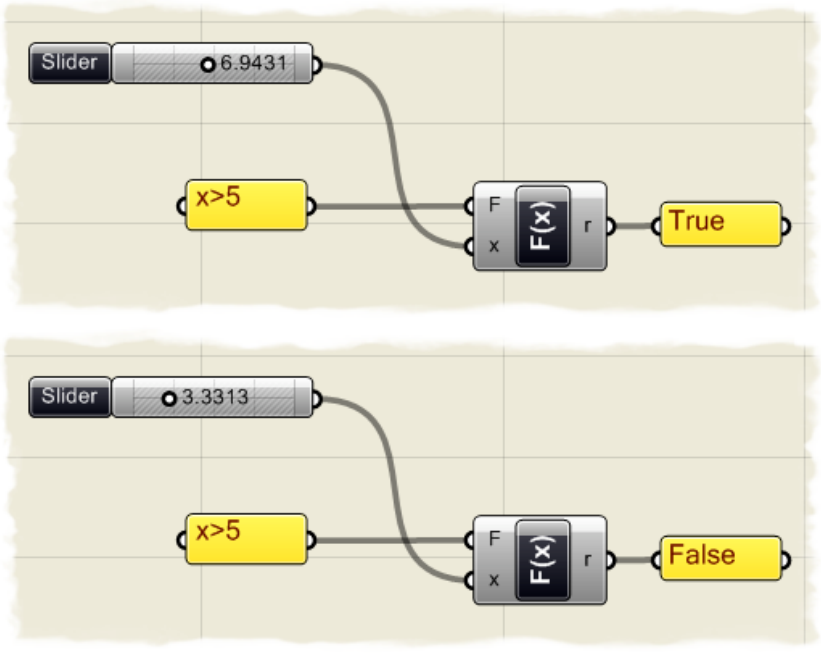
Almost every programming language has a method for evaluating conditional statements. In most cases the programmer creates a piece of code to ask a simple question of "what if?" What if the 9/11 terrorist attacks had never happened? What if gas cost $10/gallon? These are important questions that represent 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.

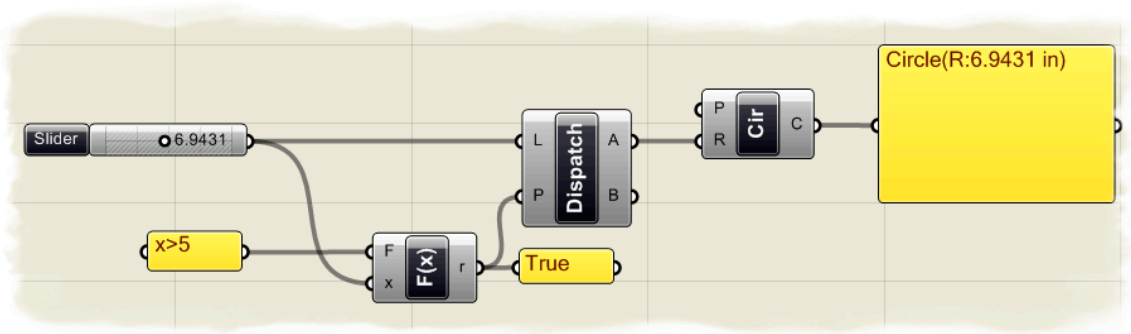
The piece of code first looks at an object and determines a single boolean value for whether or not it is a curve. There is no middle ground. The boolean value is **True** if the object is a curve, or **False** if the object is not a curve. The second part of the statement performs an action dependant 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.

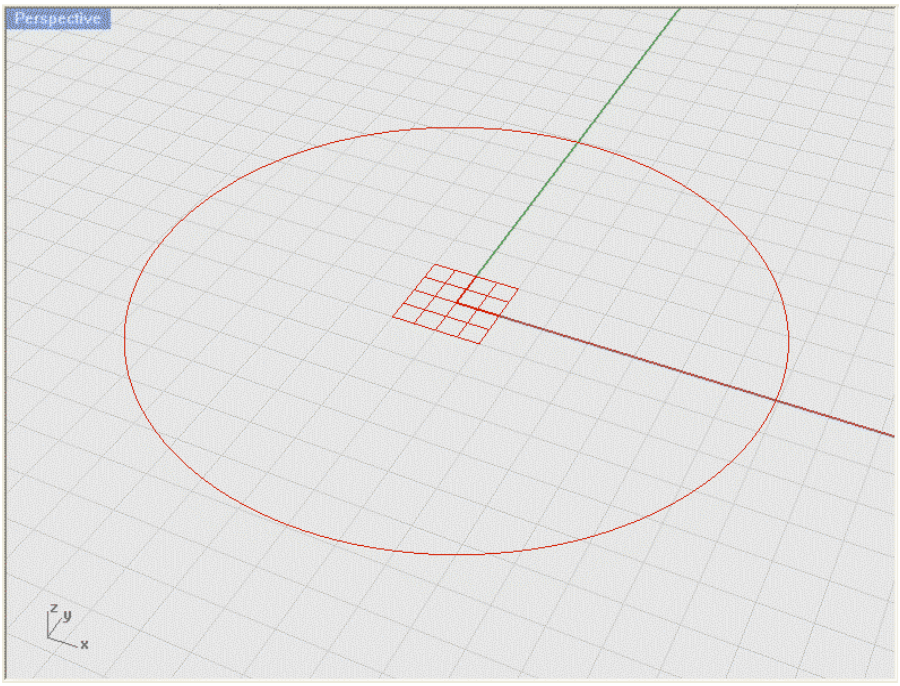
Grasshopper has the same ability to analyze conditional statements through the use of Function components.



In the example above, we have connected a numeric slider to the x-input of a single variable Function component (Scalar/Expressions/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 the Dispatch-P input component to perform a certain action. The Dispatch component works by taking a list of information, in the example below, we have connected the numeric slider information to the Dispatch-L input, and filters the information based on the boolean pattern of the Dispatch-P input. 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 x 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.





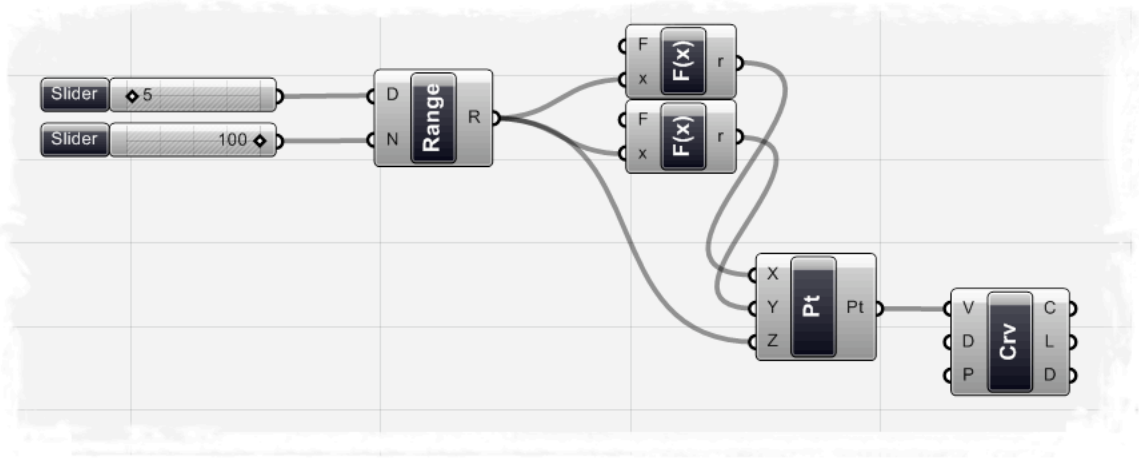
Note: To see the finished version of the circle boolean test example, **Open** the file **CircleBooleanTest.ghx** found in the Source Files folder that accompanies this document.

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.

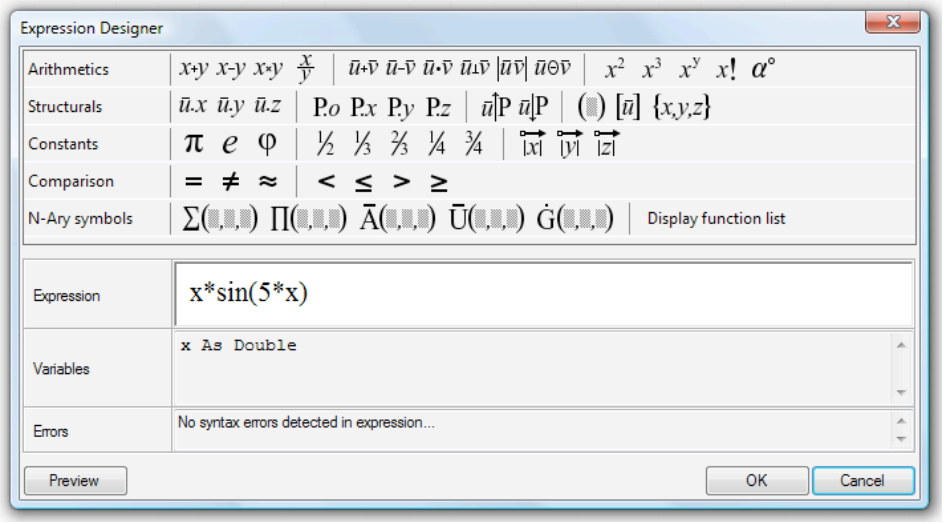
In the following example, we will create a mathematical spiral similar to the example David Rutten provided in his *Rhinoscript 101* manual. For more information about Rhinoscript or to download a copy of the manual, visit **http://en.wiki.mcneel.com/default.aspx/McNeel/RhinoScript101.html**

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 each slider and set the slider type to Integers
- Right-click each slider and set the Lower Limit to 1, and the Upper Limit to 100
- Set the first slider value to 5
- Set the second slider value to 100
- Connect the first slider to the Range-D input
- Connect the second slider to the Range-N input
  We have created a range of 101 numbers evenly spaced from 0.0 to 5.0, which we can feed into our Function components.
- Scalar/Expressions/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)
  - Click OK to accept the algorithm

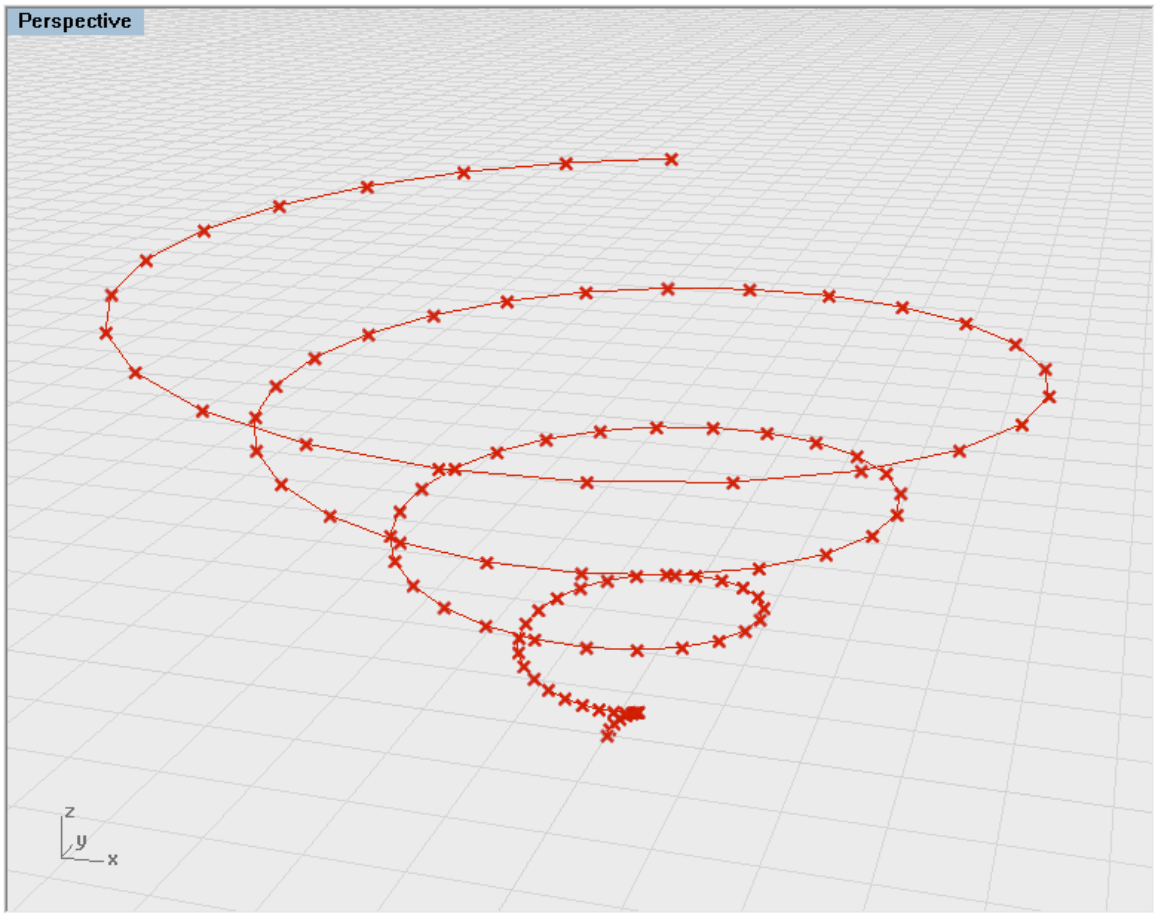


- Select the Function component and type Cntrl+C (copy) and Cntrl+V (paste) to create a duplicate Function
- Right-click the F-input of the duplicated Function component and open the Expression Editor
- In the Expression Editor dialogue box, type the following equation:
  - x*cos(5*x)
  - Click OK to 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.
- 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 number of rotations for each loop 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 value 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 actual line will not actually pass through each point.



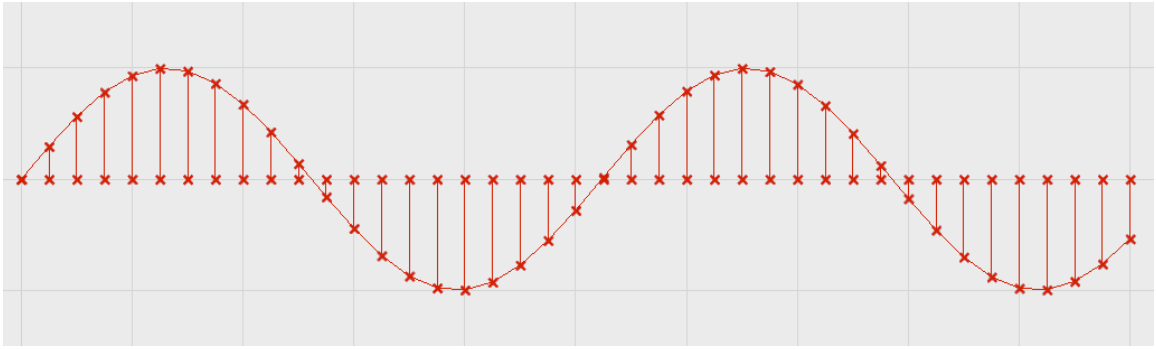
Note: To see a video tutorial of this example, please visit Zach Downey's blog at:
http://www.designalyze.com/2008/07/07/generating-a-spiral-in-rhinos-grasshopper-plugin/

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, and the frequency can be controlled by a set of numeric sliders.

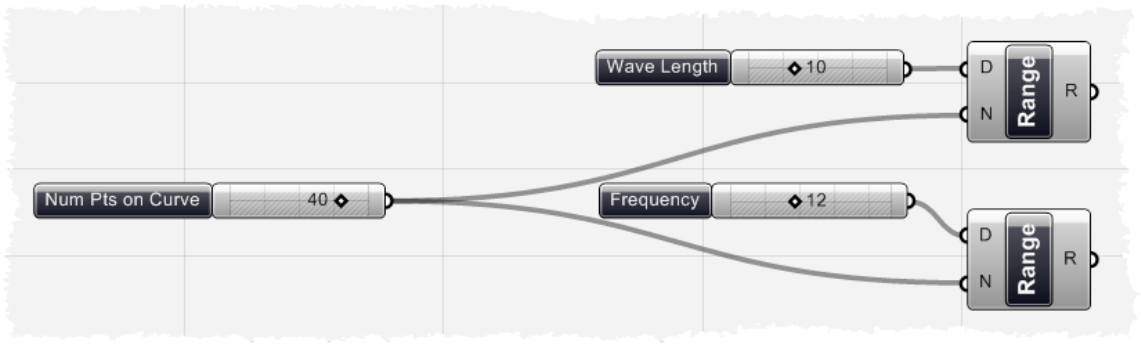
Note: To see the finished definition used to create the sine curve below, **Open** the file **Trigonometric_curves.ghx** found in the Souce Files folder that accompanies this document.



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, where 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.

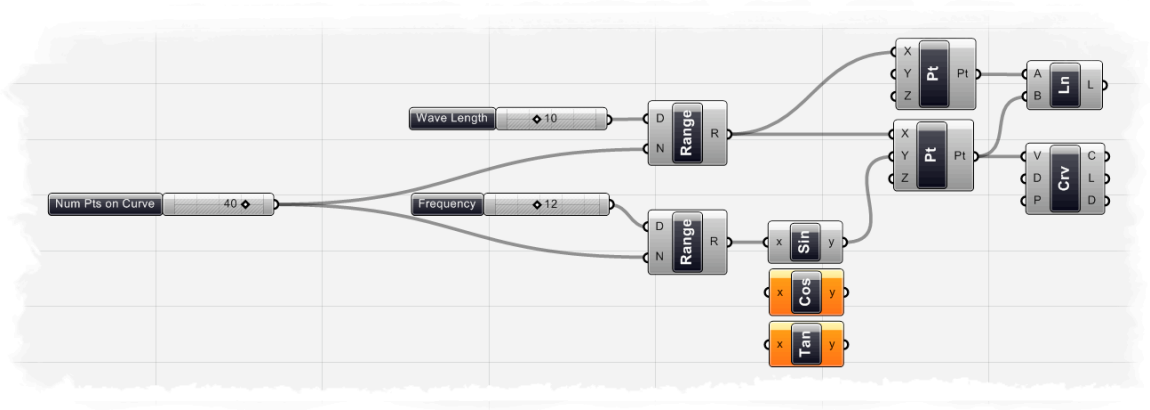
- Scalar/Trigonometry/Sine - Drag and drop a Sine component onto the canvas
- Connect the second Range-R output to the x-input of the Sine component
- Vector/Point/Point XYZ - Drag and drop a Point XYZ component onto the canvas
- Connect the first Range-R output to the X-input of the Point XYZ component
- 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/Curve - Drag and drop a Curve component onto the canvas
- Connect the Point-Pt output to the Curve-V input
- Right-click on the D-input of the Curve component and set the integer to 1.0, to make curve function as a one degree 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-*

components 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 we can generate other sinusoidal curves, like a Cosine wave forms, by replacing the Sine component with a Cosine component found under the Scalar/Trigonometry tab.



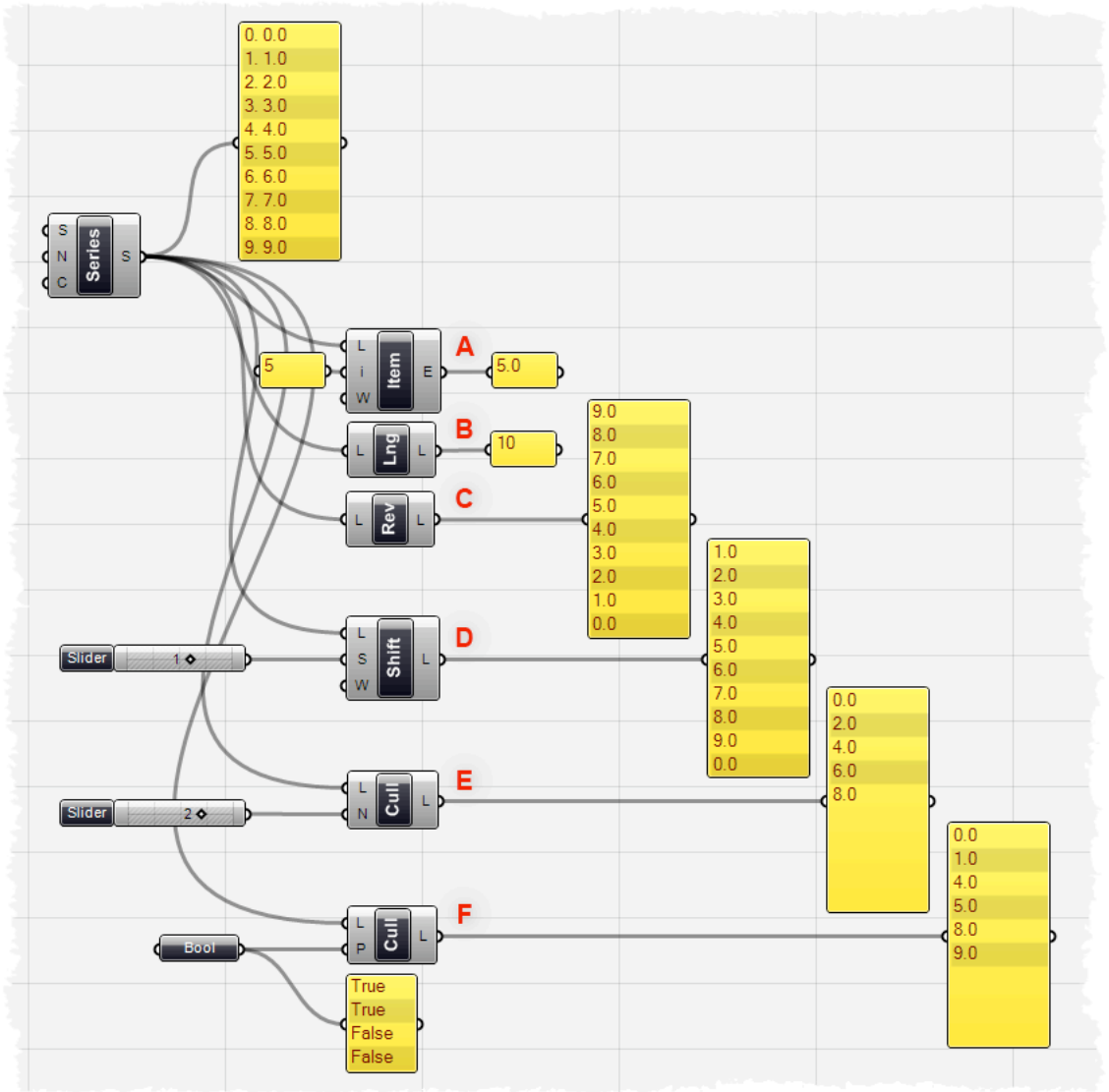
Note: To see a video tutorial of this example, please visit David Fano's blog at:
http://designreform.net/2008/06/01/rhino-3d-sine-curve-explicit-history/

8 *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.

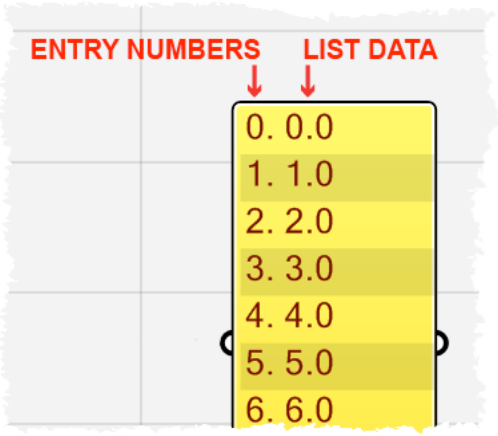
Below is an example of how to control a list of numeric data using a variety of list components.

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:



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.

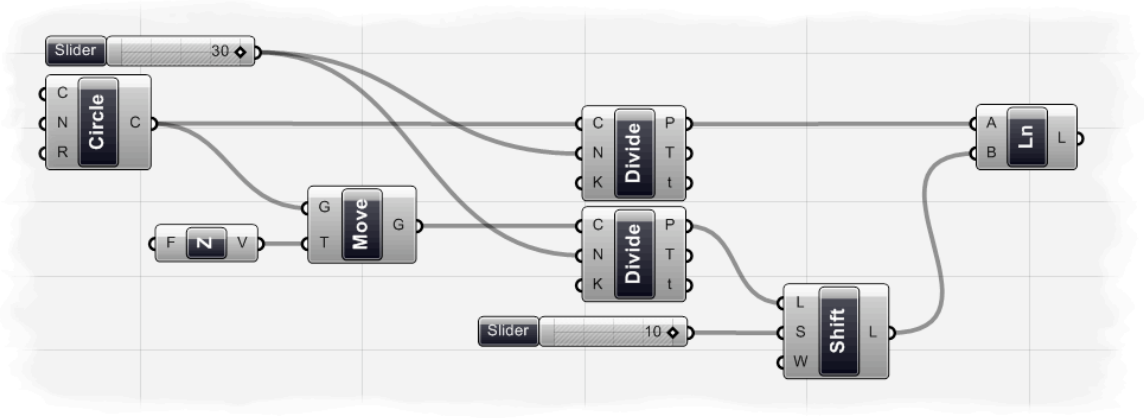
E) The **Cull Nth** component (Logic/Sets/Cull Nth) will remove every Nth data entry from the list, where N is defined by a numeric integer. In this example, we have connected a numeric slider to the Cull Nth-N input. We have set our slider to 2.0, such that the Cull Nth component will remove every other 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.

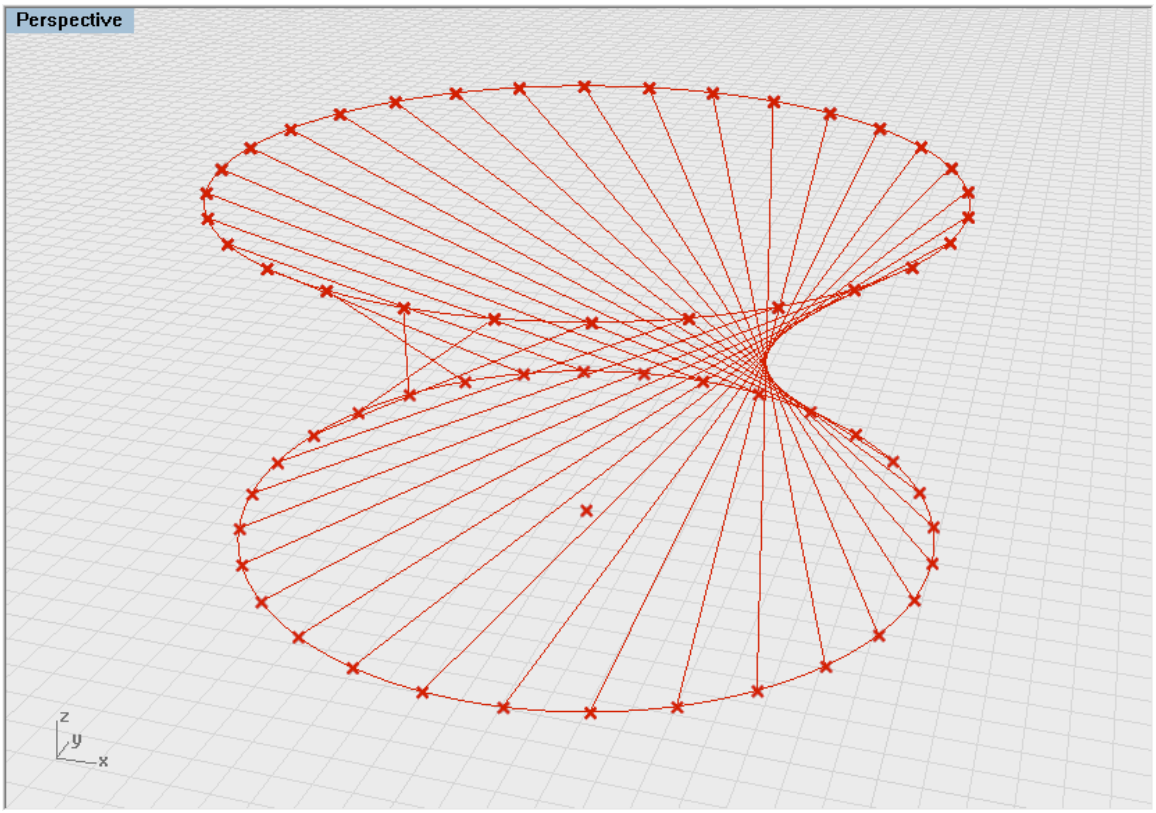
F) The **Cull Pattern** component (Logic/Sets/Cull Pattern) is similar to the Cull Nth component, in that it removes items from a list based on a defined value. However, in this case, it uses a set of boolean values that form a pattern, instead of numeric values. If the boolean value is set to True, 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.

8.1 Shifting Data

We discussed in the previous section 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 his 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

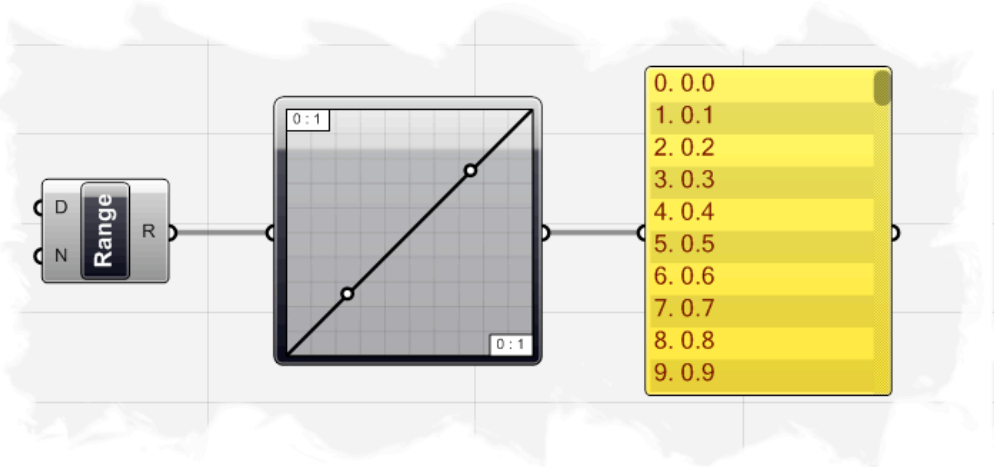
  We have shifted the points on the upper circle up the index list by 10 entries. By setting the wrap value to True, we have created a closed loop of data entries.
- Curve/Primitive/Line - Drag and drop a Line component onto the canvas
- Connect the first Divide Curve-P output to the Line-A input
- Connect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

8.2 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.

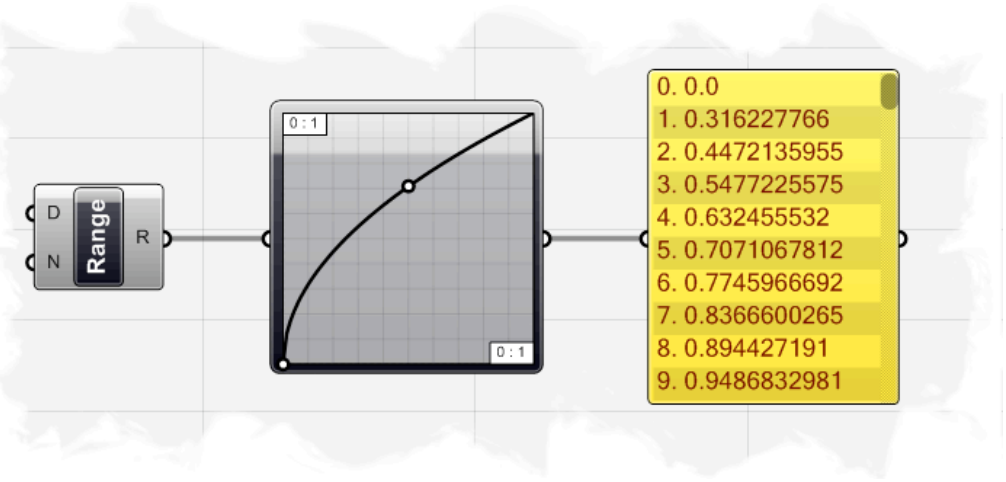
Note: To see the finished definition of the following example, **Open** the file **Stream Contents_Excel.ghx** found in the Source Files folder that accompanies this document.



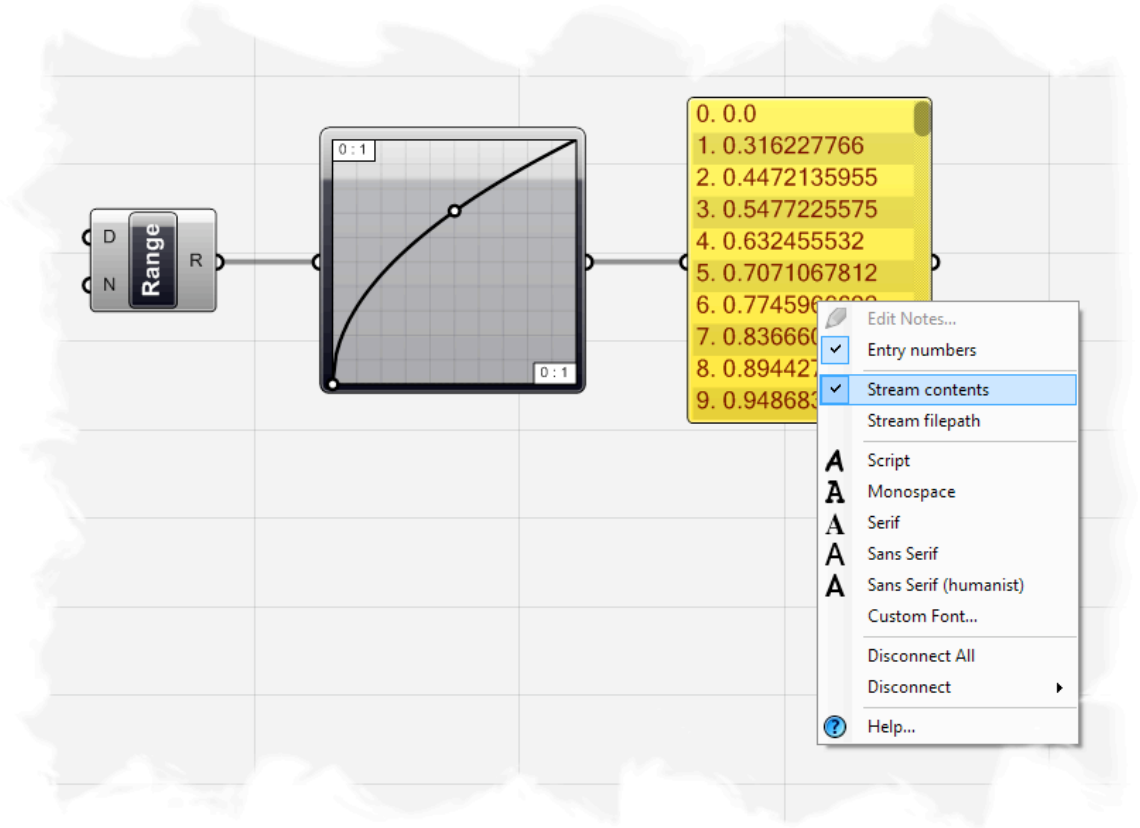
To start, we have dropped a **Range** component (Logic/Sets/Range) onto the canvas, and set the numeric domain from 0.0 to 10.0. By right-clicking on the Range-N input, we have set the number of steps to 100, so that our output list will show 101 equally spaced values between 0.0 and 10.0.

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.

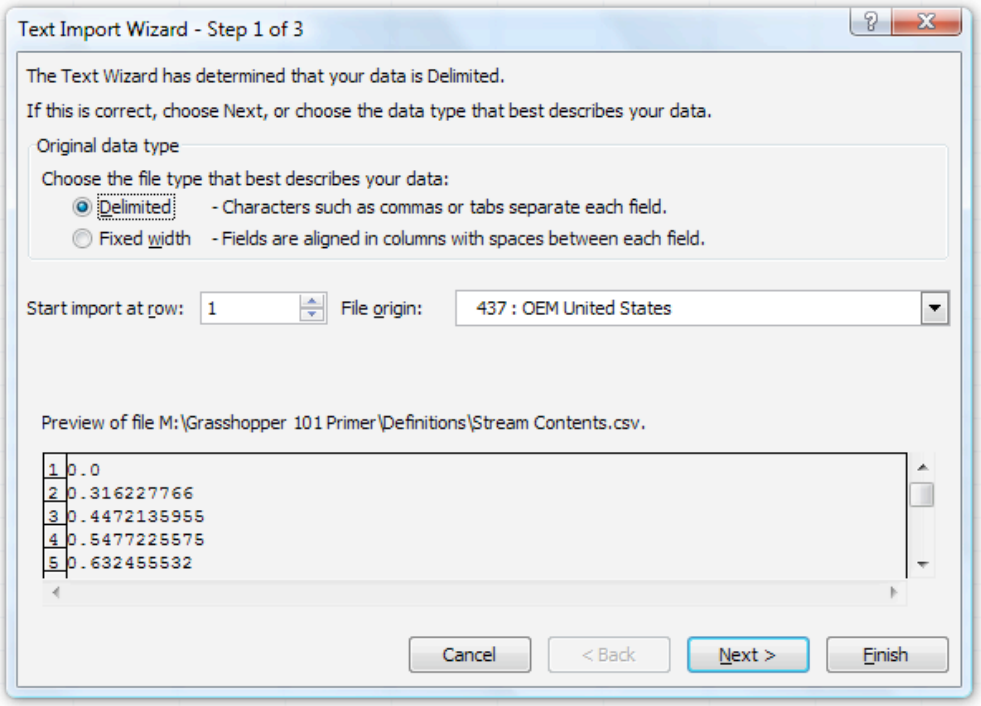
Since the Graph Mapper type is set to Linear, the output list (shown in the Post-it Panel) displays a set of numeric data that ascends from 0.0 to 10.0 in a linear fashion. However, if we right-click on the Graph Mapper component and set the Graph Type to **Square Root**, we should see a new list of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



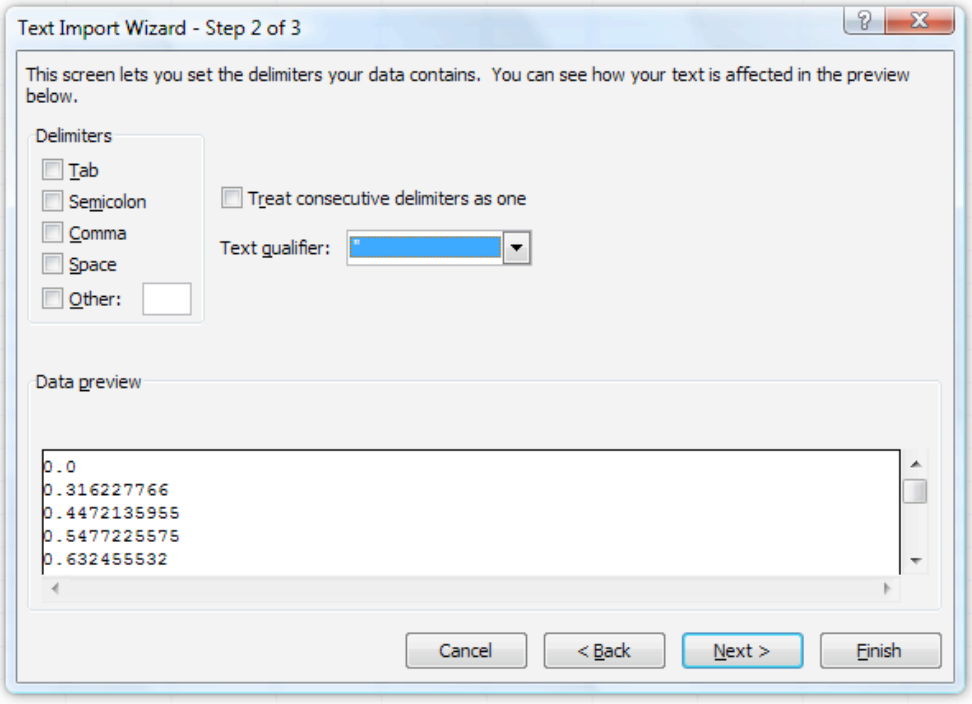
To export the list of data from the Post-it Panel, simply right-click on the panel and select **Stream Contents**. When prompted, find a location on your hard drive to **Save** the file with a unique file name. In our example, we will save the file to the following location: C:/Tutorials/Exporting Data/Stream_Contents.csv. 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.



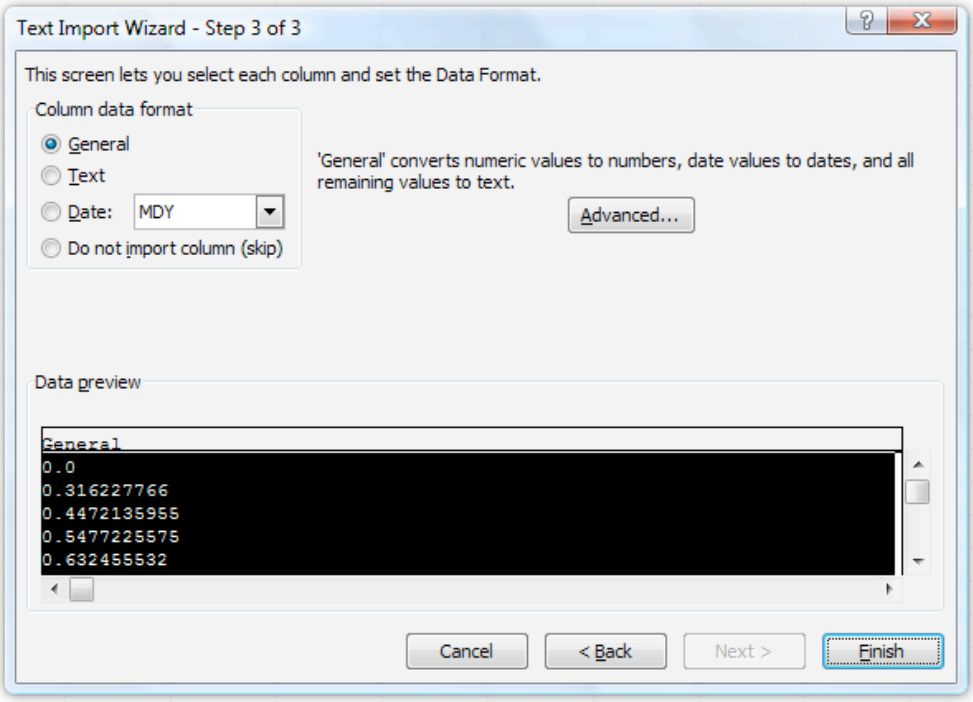
We can now import the data into Microsoft Excel 2007. First launch the application and select the Data tab. Select the **Get External Data from Text** under this tab and find the Stream_Contents.csv file you saved on your hard drive. You will now be guided through the **Text Import Wizard** where you 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.



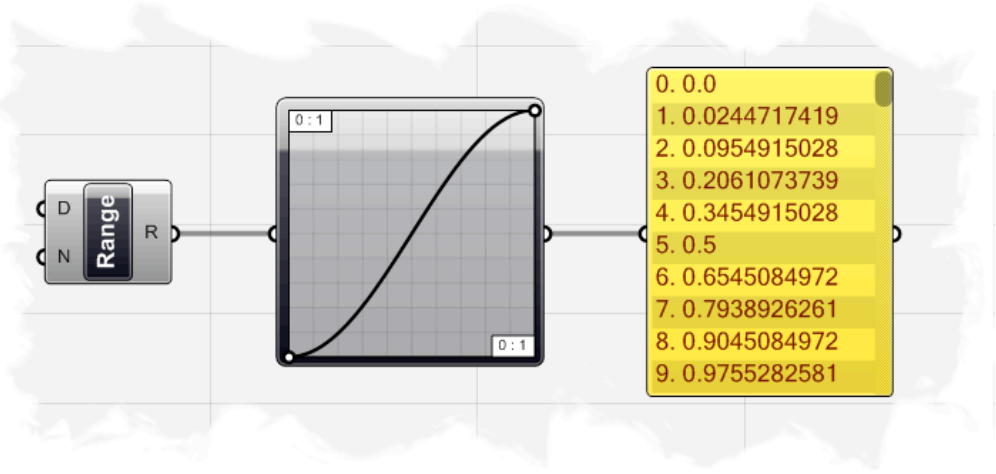
Step 2 of the Text Import Wizard allows you to set the which types of Delimiters will define how your data is separated. A Delimiter is some character (such as a semi-colon, comma, or space) stored in the CSV file that indicates where the data should be split into another column. Since we only have numeric 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.



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.

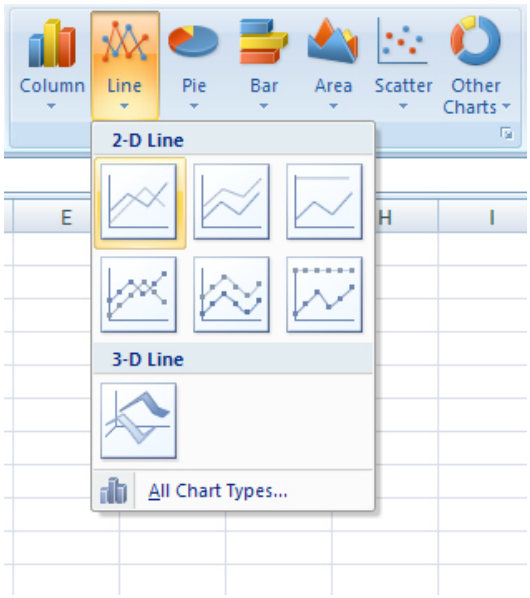


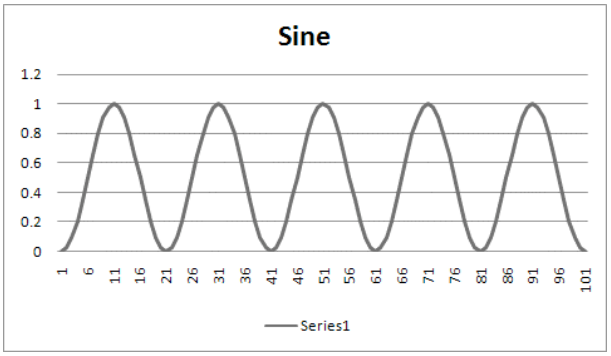
You will now be prompted as to which cell you would like to use to begin importing your data. We will use the default cell value of A1. You will now see all 101 values in the A column that correspond to the values within the Grasshopper Post-it Panel. The Grasshopper definition is constantly streaming 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.



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.

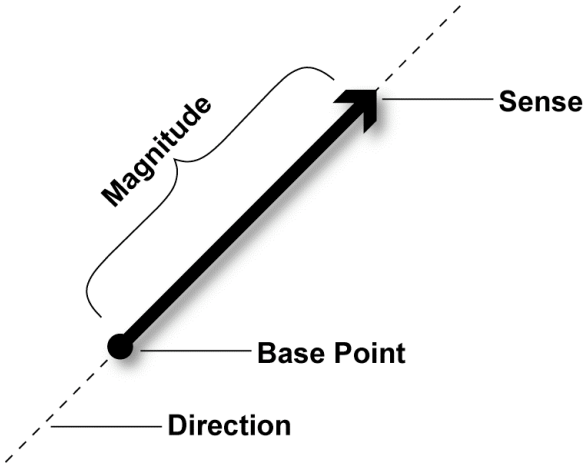




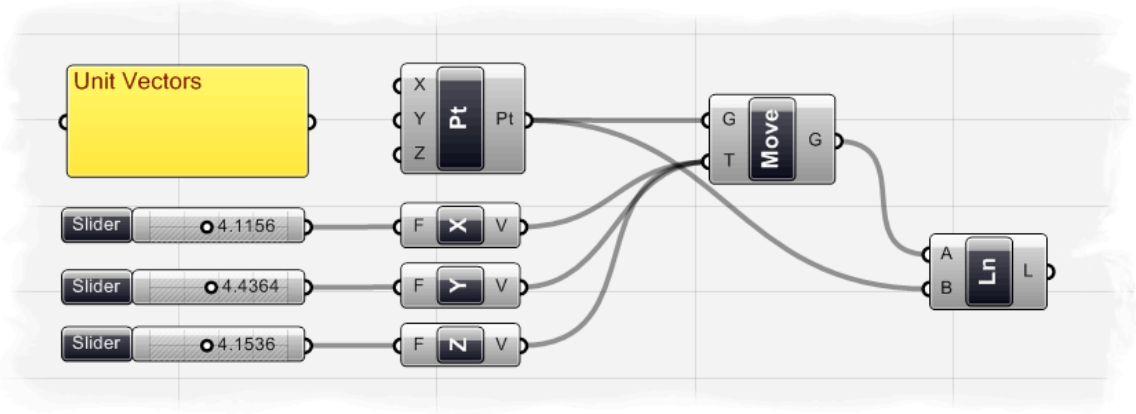
You will see a Line Chart that reflects the same shape shown in the Graph Mapper component in Grasshopper. You can make as many changes as you want within Grasshopper, and after hitting **Refresh All**, you will see the reflected changes in Excel.

9 *Vector Basics*

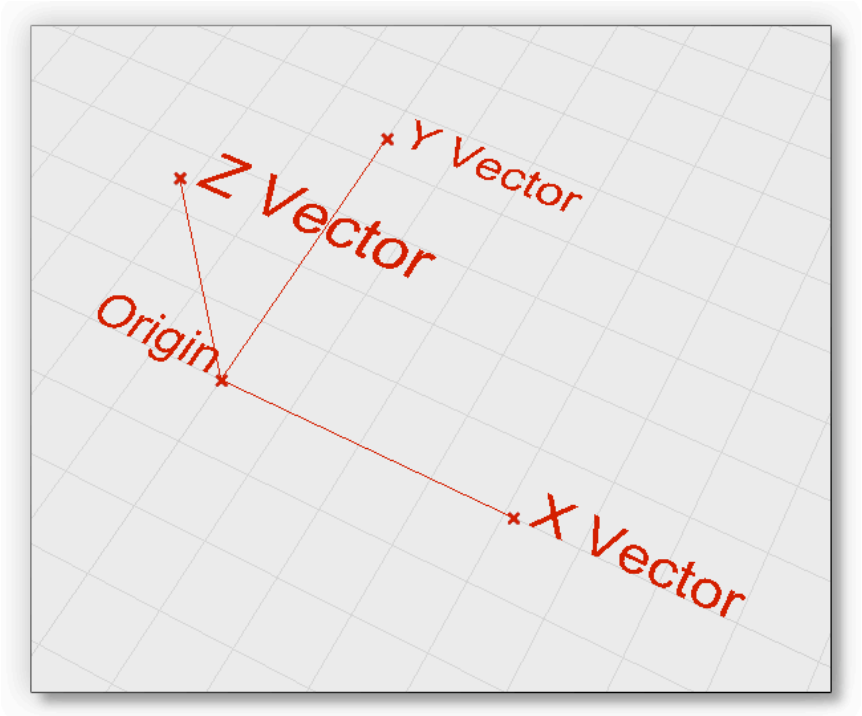
From physics, we know that a vector is a geometric object that has a magnitude (or length), direction, and sense (or orientation along the given direction). A vector is frequently represented by a line segment with a definite direction (often represented as an arrow), connecting a base point A with a terminal point B. The magnitude 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.



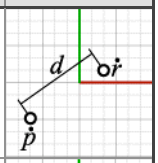
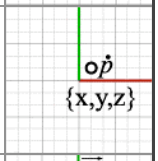
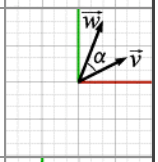
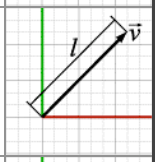
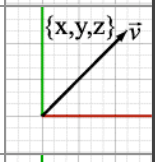
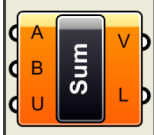
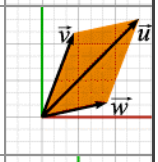
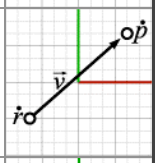
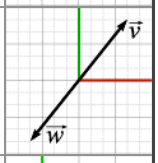
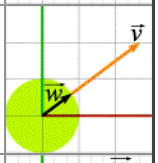
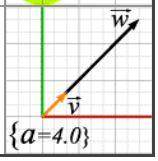
In the example above, we start by creating a point at the origin 0,0,0 using the Point XYZ component (Vector/Point/Point XYZ). We then connect the Point-Pt output to a Move-G input component to translate a copy of the point in some vector direction. To do this, we drag and drop a Unit X, Unit Y, and 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.



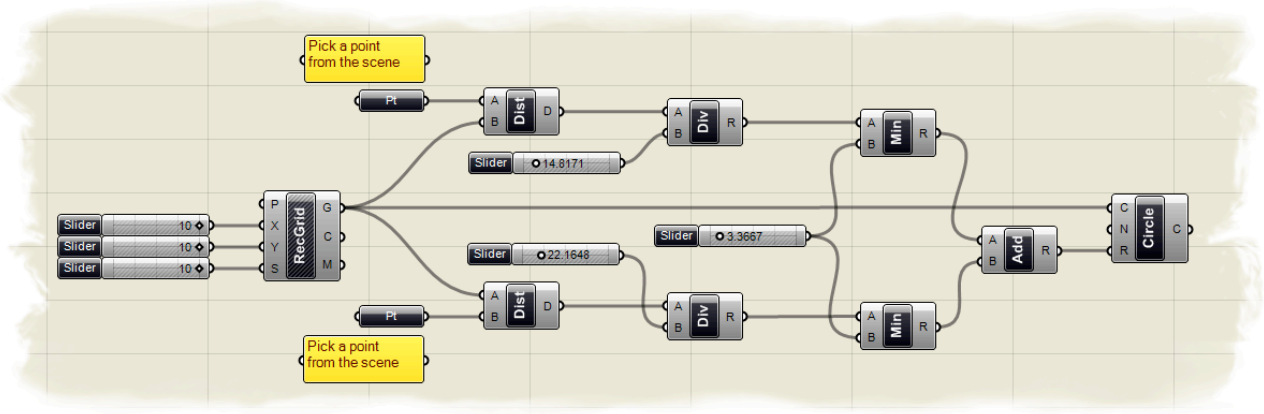
Note: To see the finished definition of the example above, **Open** the file **Unit Vectors.ghx** found in the Source Files folder that accompanies this document.

9.1 Point/Vector Manipulation

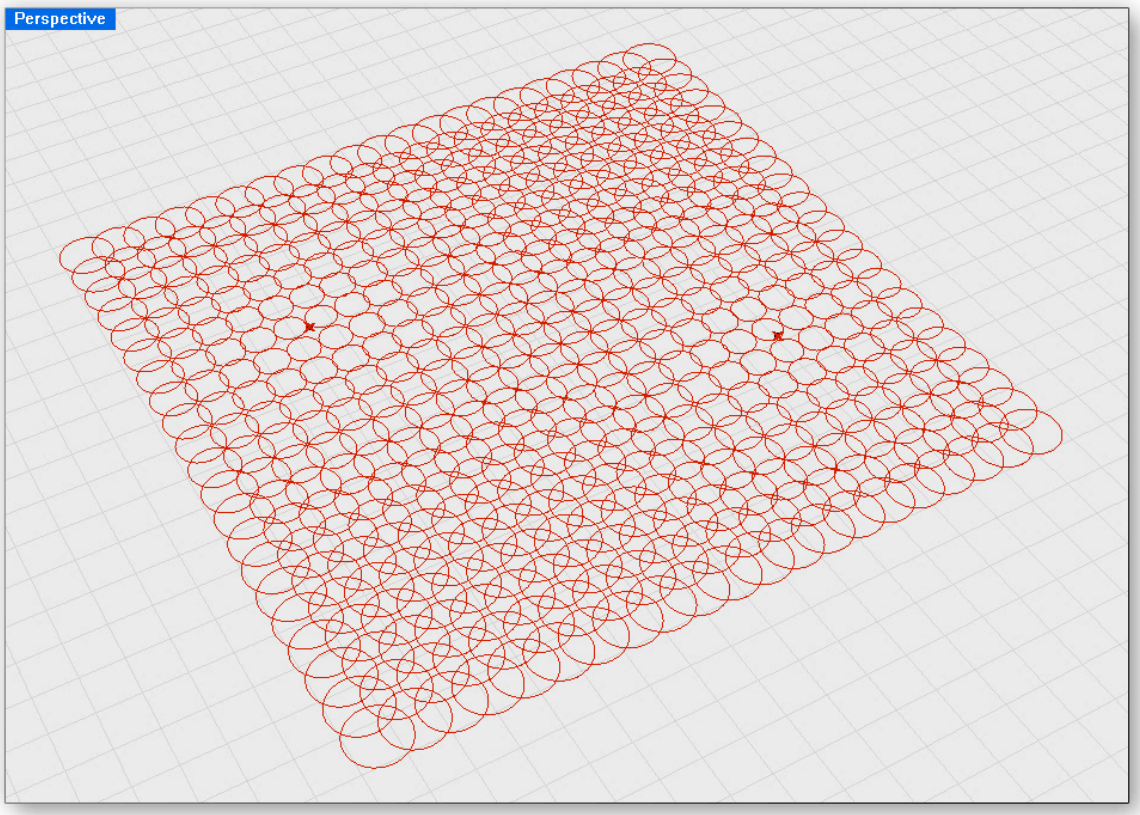
Grasshopper has an entire group of Point/Vector components which perform the basic operations of 'vector mathematics'. Below is a table of the most commonly used components and their functions.

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 on 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

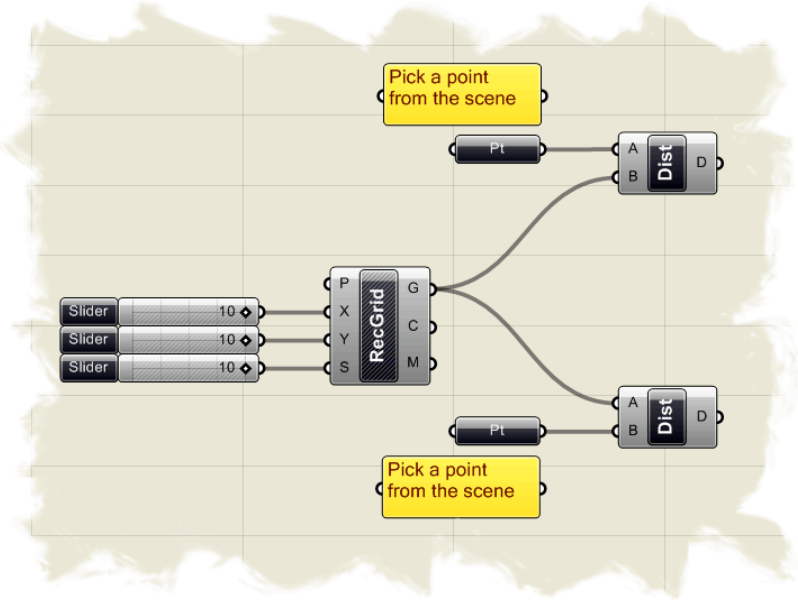
  The Rectangular Point Grid component create a grid of points, where the P-input is the origin of the grid (in our case we'll use 0,0,0). The Grid component creates a number of points in both the X and Y direction specified by the numeric sliders. However, notice that we have set both of these to 10.0. 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 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 two Point components 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 points.***
- In the Rhino scene, type "Point" in the dialogue box and place two Point objects anywhere in your scene. In our case, we'll place the points in the Top viewport so that each point is in the XY plane.

  Now that we have created two attractor point objects in the scene, we can assign them to the Point component we just created inside Grasshopper.
- Right-click each Point component and select "Set One Point"
- When prompted, select one of the attractor point objects that you created in the Rhino scene
- Repeat the previous two steps for the other Point component, making sure to pick the other attractor point for the second component

  We have now created a Grid of points and assigned 2 attractor point objects from our scene as Grasshopper components. We can use a bit of vector mathematics to determine the distance from each grid point to the attractor points.
- Vector/Point/Distance - Drag and drop two Distance components onto the canvas
- Connect the first attractor point output to the first Distance-A input
- Connect the rectangular Grid Point-G output to the first Distance-B input
- Connect the second attractor point output to the second Distance-B input

- Connect the rectangular Grid Point-G output to the second Distance-A 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 two Division operators onto the canvas
- Connect the first Distance-D output to the first Division-A input
- Connect the second Distance-D output to the second Division-A input
- Params/Special/Numeric Slider - Drag and drop two numeric sliders onto the canvas
- Right-click on the first slider and set the following:
  - Name: Pt1 Influence
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 25.0
- Right-click on the second slider and set the following:
  - Name: Pt2 Influence
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 25.0
- Connect the Pt1 Influence slider to the first Division-B input
- Connect the Pt2 Influence slider to the second Division-B input

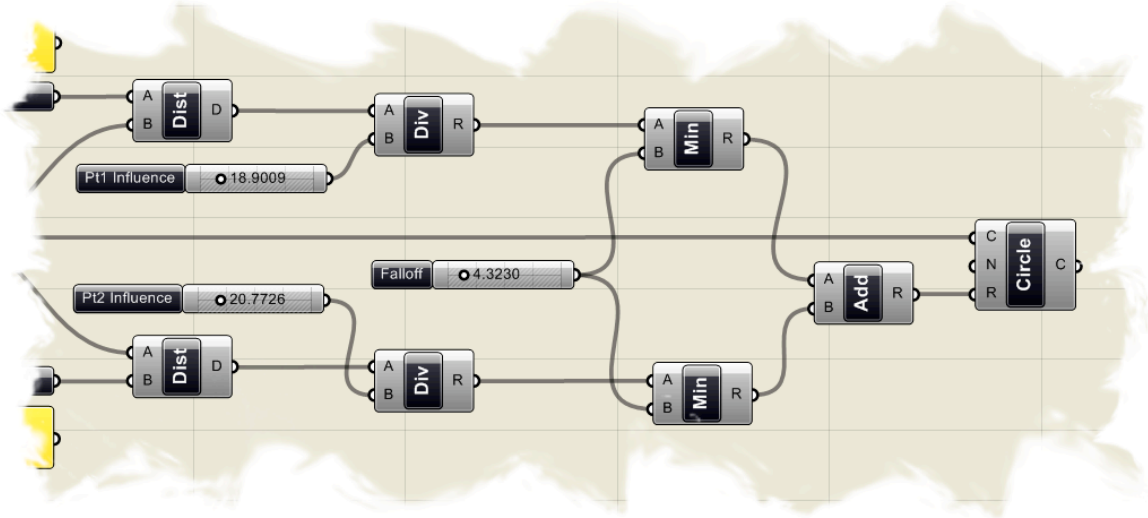
Since the distance values were quite large, we needed a scale factor to bring the numbers down to a more manageable value. We have used the numeric sliders as division factors to give us a new set of output values

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 determine a minimum number that we do not want our circle radius to fall below.

- Scalar/Utility/Minimum - Drag and drop two Minimum components onto the canvas
- Connect the first Division-R output to the first Minimum-A input
- Connect the second Division-R output to the second Minimum-A input
- Params/Special/Numeric Slider - Drag and drop a numeric slider onto the canvas
- Right-click on 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 *BOTH* Minimum-B inputs

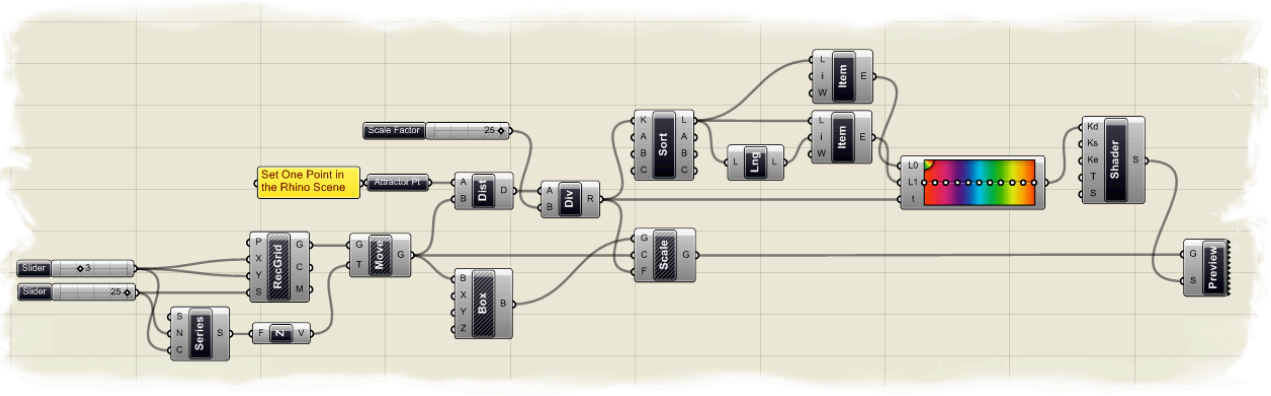
  Now, all there is left to do is to add the two lists together to form one list which will define the radius of each circle.
- Scalar/Operators/Addition - Drag and drop an Addition operator onto the canvas
- Connect the first Minimum-R output to the Addition-A input
- Connect the second Minimum-R output to the Addition-B input
- 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
- Also, connect the Addition-R output to the Circle-R input
- Right-click on the Rectangular Point Grid component and turn the Preview off

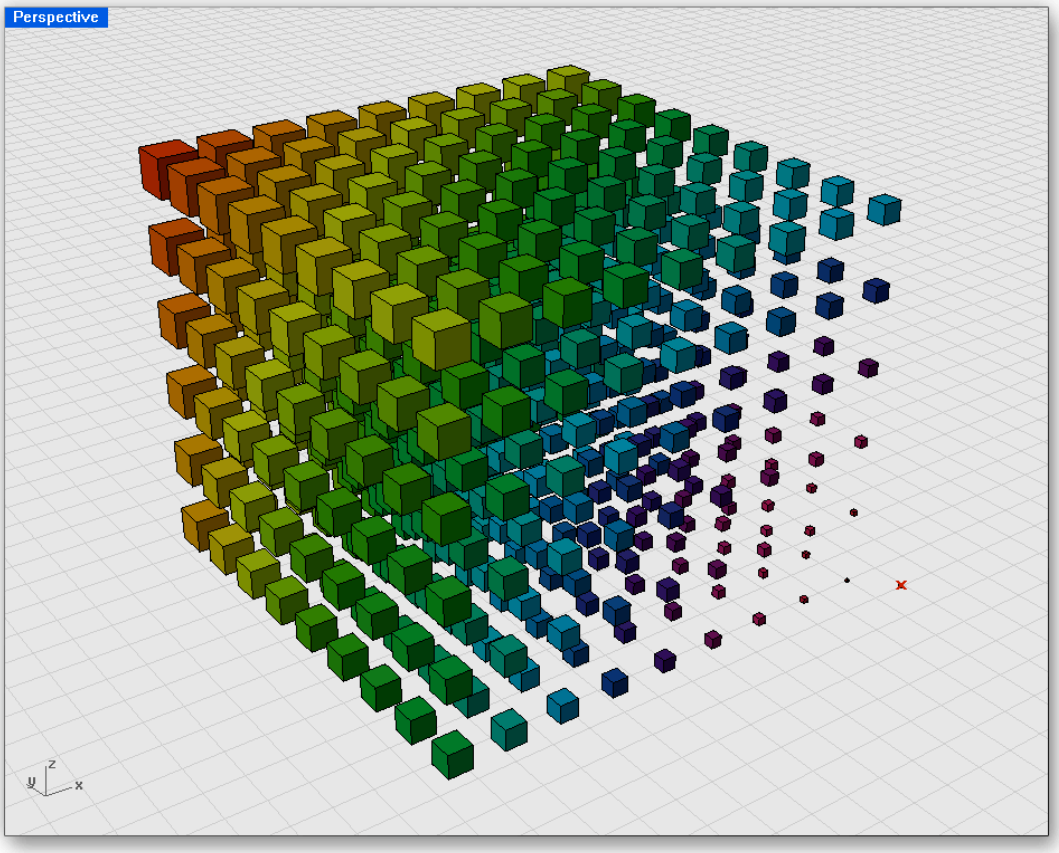


Note: To see a video tutorial of this example, please visit David Fano's blog at:
http://designreform.net/2008/07/08/grasshopper-patterning-with-2-attractor-points/

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 6 points in each direction. Since we ultimately would like to have a three dimensional cube of points, we need to create 6 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**

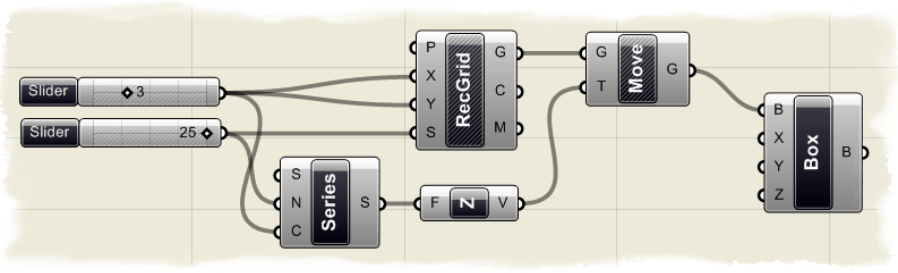
  We have now told the component to multiply Series-C input by a factor of two. Since our original input was set to 3, our new Series Count will be 6.0
- Vector/Constants/Unit Z – Drag and drop a Unit Z vector component onto the canvas
- Connect the Series-S output to the Unit Z-F input

  If you hover your mouse over the Unit Z-V output you should see that we have defined 6 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

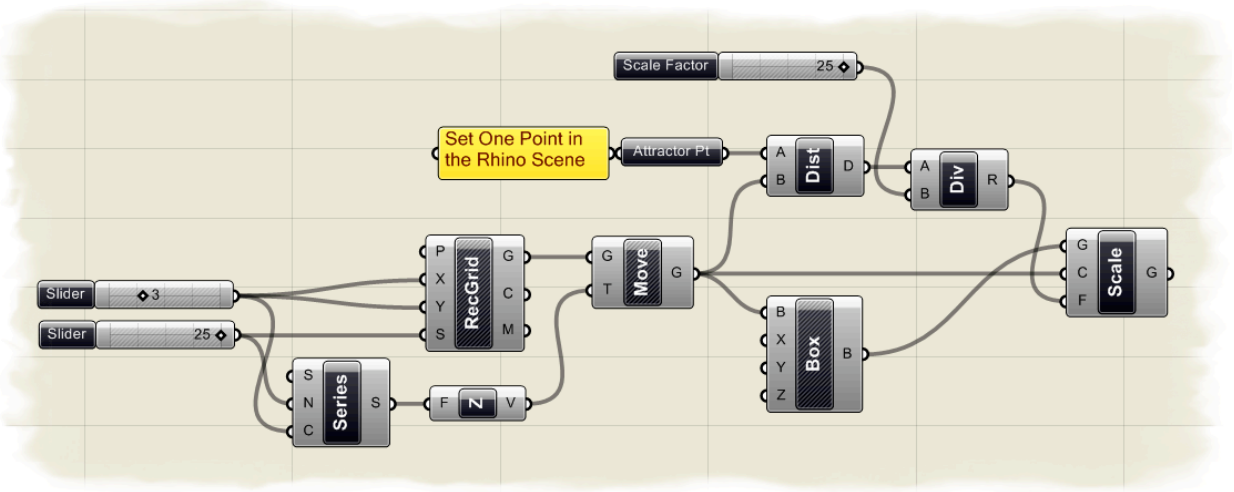
You should now see a small red X over the point indicating that the Attractor Pt component has been assigned that point value in your scene. If you move the point anywhere in your scene, the Grasshopper component will automatically update its position.

- 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: Assigning 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 away 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/Specia/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.*

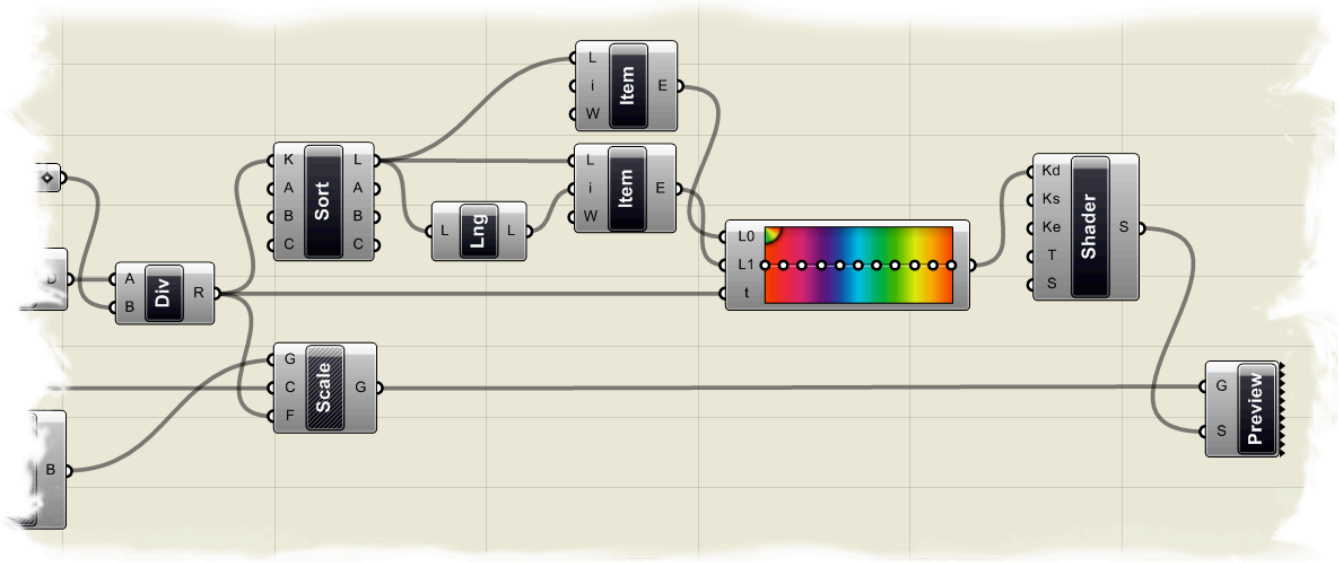
  ***T:** Input defines a shader's Transparency.*

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****.*

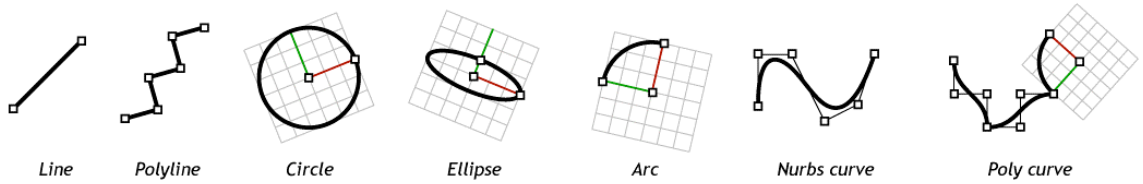
- Params/Special/Custom Preview – Drag and drop a Custom Preview component onto the canvas
- Connect the Scale-G output to the Custom Preview-G input
- Connect the Shader-S output to the Custom Preview-S input
- Right-click and turn the preview off for the Scale component

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.



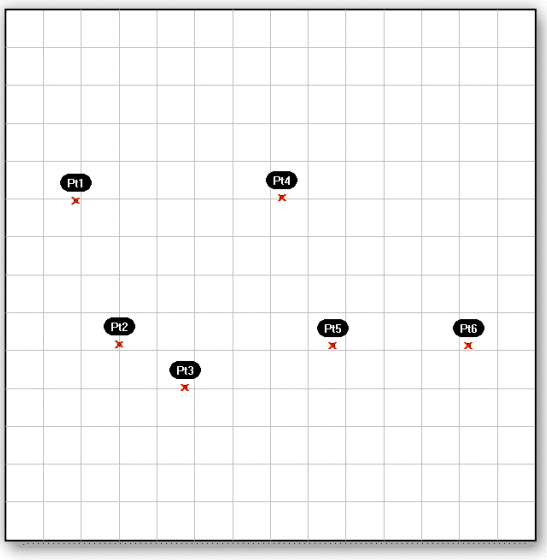
10 *Curve Types*

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.

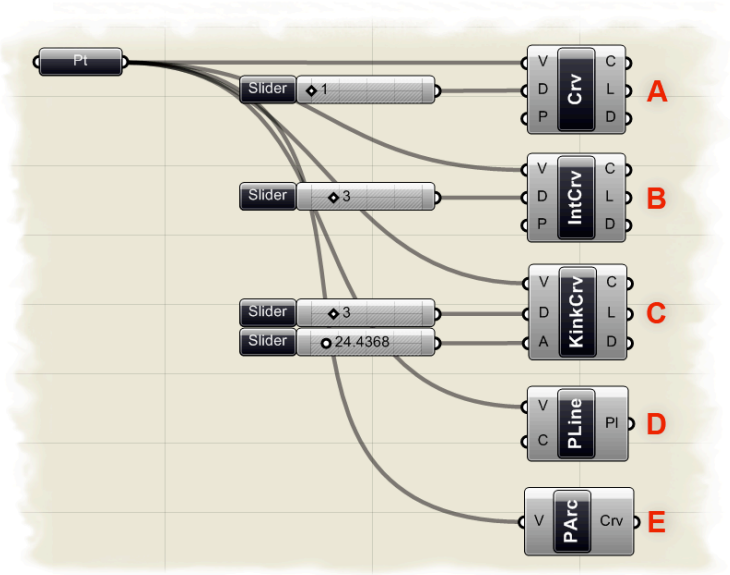


Grasshopper also has a set of tools to express Rhino's more advanced curve types like nurbs curves and poly curves. Below, is an example that will walk us through some of Grasshopper's **Spline Components** but first we will need to create a set of points that will define how our curves will act.

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.

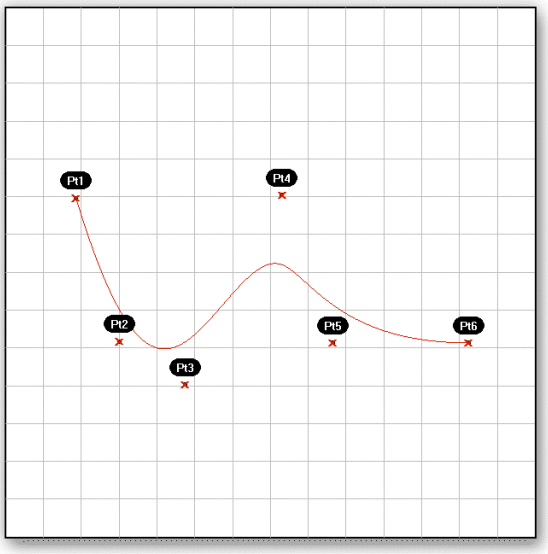


Now in Grasshopper, **Open** the file **Curve Types.ghx** in the Source Files folder that accompanies this document. You will see a Point component connected to several Curve components, each defining a curve using a different method. We will go through each component individually, but first we must 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.



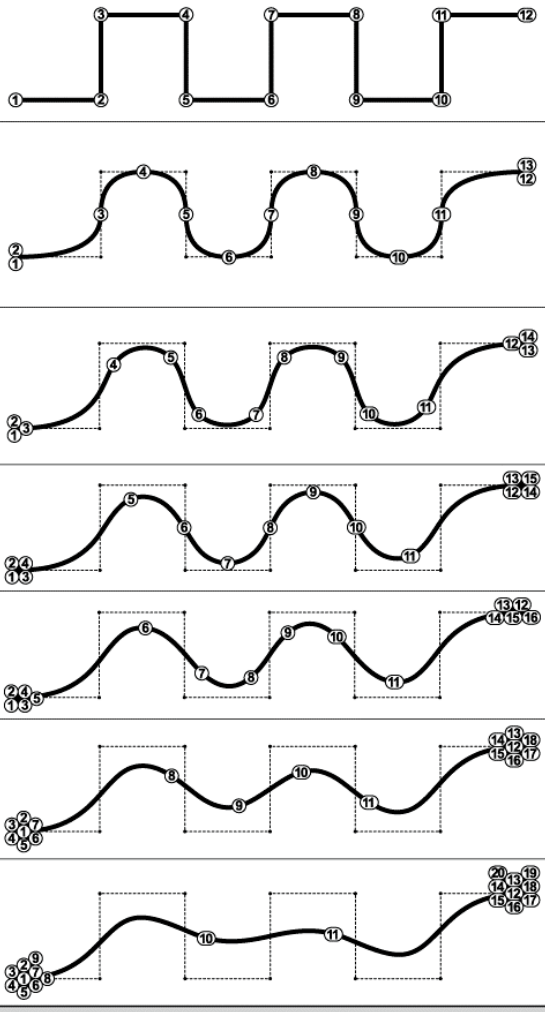
A) NURBS Curves (Curve/Spline/**Curve**)

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.

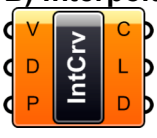
D^4 is technically possible in Rhino, but the math for nurbs curves doesn't work as well with even degrees. Odd numbers are usually preferred.

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.

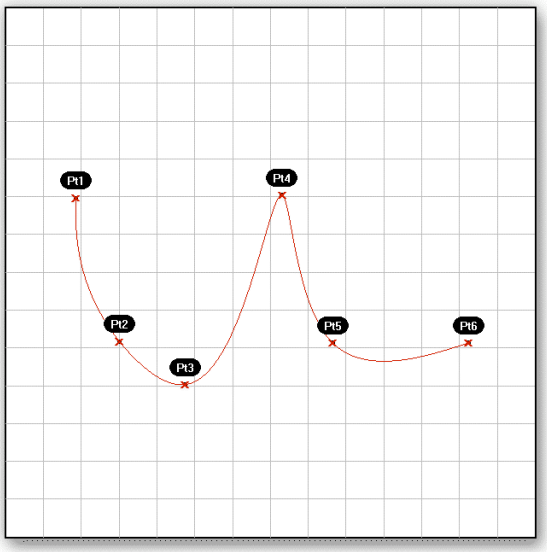
In our example, we have connected a numeric slider to the Curve-D input to define the degree of our NURBS curve. By dragging our slider to the left and right, we can visually see the change in influence of each control point. The **NURBS Curve-P input** uses a boolean value to define whether or not the 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**)



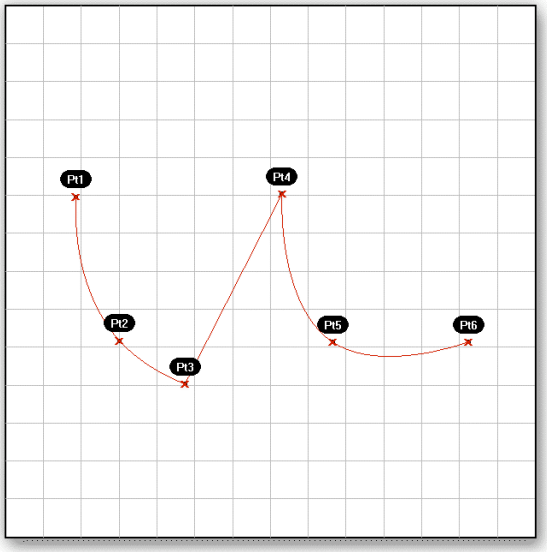
Interpolated curves behave slightly differently than NURBS curves, in that the curve passes through the control points. You see, it is very difficult to make NURBS curves go through specific coordinates. Even if we were to tweak individual control points, it would be an incredibly arduous task to make 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.

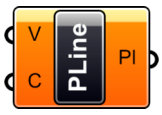


C) Kinky Curves

(Curve/Spline/**Kinky Curve**) Despite its risqué 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.

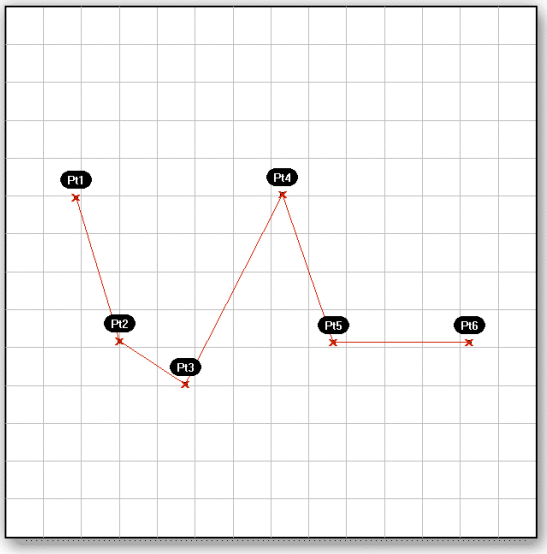


D) Polyline Curves (Curve/Spline/Polyline)

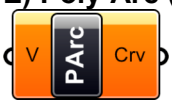


A polyline just may be one of the most flexible curve types available in Rhino. This is because a polyline curve can be made up of line segments, polyline segments, degree=1 NURBS curves, or any combination of the above. But, let's start with the basics behind the polyline. Essentially, a polyline is 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

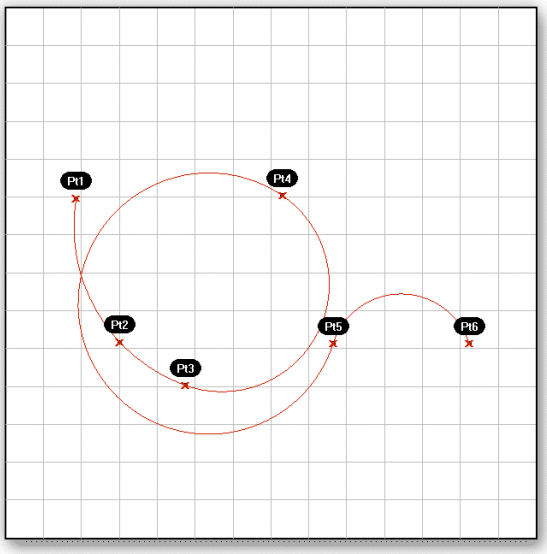
will define the boundaries of each line segment that make up the polyline. The C-input of the component defines whether or not the polyline is an open or closed curve. If the first point location does not coincide with the last point location, a line segment will be created to close the loop. The output 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)

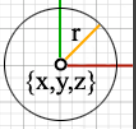
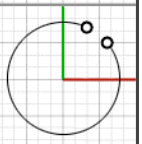
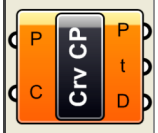
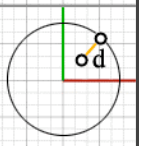
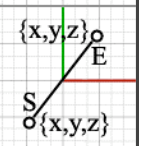
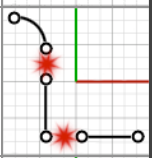
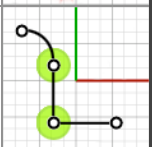
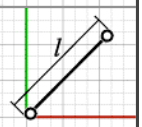
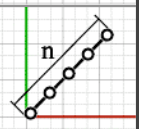
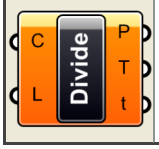
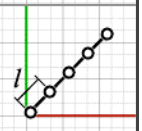


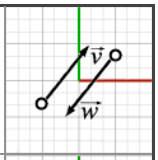
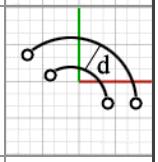
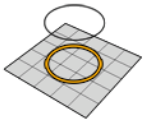
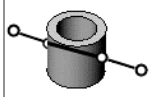
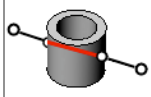
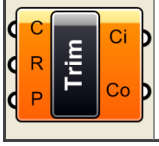
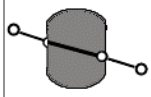
A poly-arc is almost identical in nature to the polyline, except that instead of straight line segments, the poly-arc uses a series of arcs connect each point. The poly-arc is unique, in that, it computes the required tangency at each control point in order to create one fluid curve where the transition 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

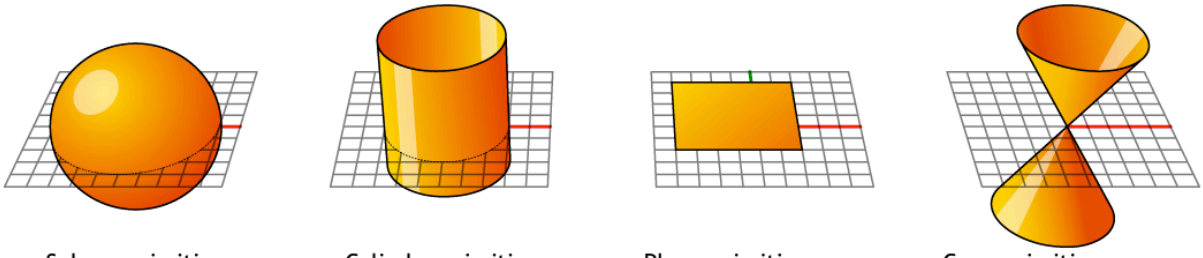
It would be quite difficult to create a tutorial that would utilize all of the analytic tools available in Grasshopper, so I have included a table to explain many of the most commonly used components.

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/**Project**	Project a curve onto a Brep (a Brep is a set of joined surfaces like a polysurface in Rhino)	
	Curve/Utility/**Split with Brep(s)**	Split a curve with one or more Breps	
	Curve/Utility/**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.	
	Curve/Utility/**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.	

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.



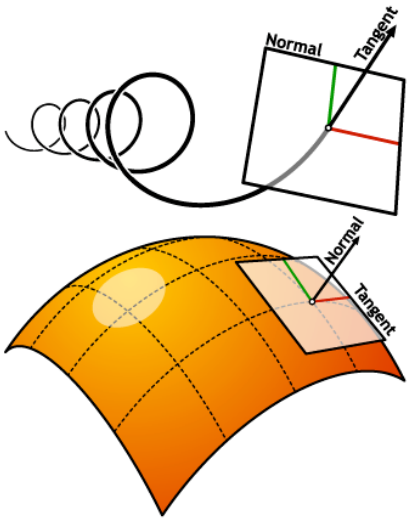
Sphere primitive
{Plane; Radius}

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.

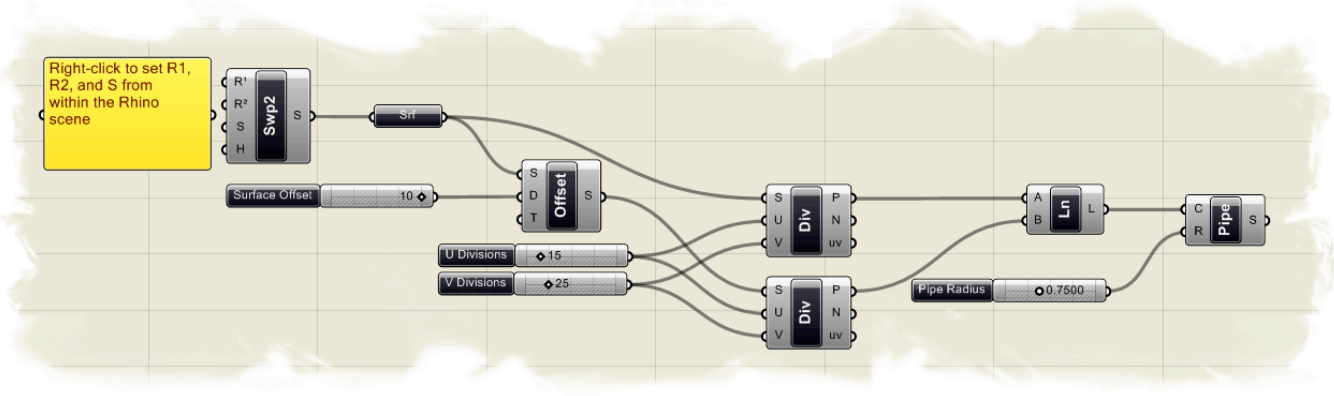
The Brep can be thought of as a three dimensional solid or similarly to the polysurface in Rhino. It is still made up of a collection of NURBS surface, however they are joined together to create a solid object with thickness, whereas a single NURBS surface theoretically has no thickness. Since Breps 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)

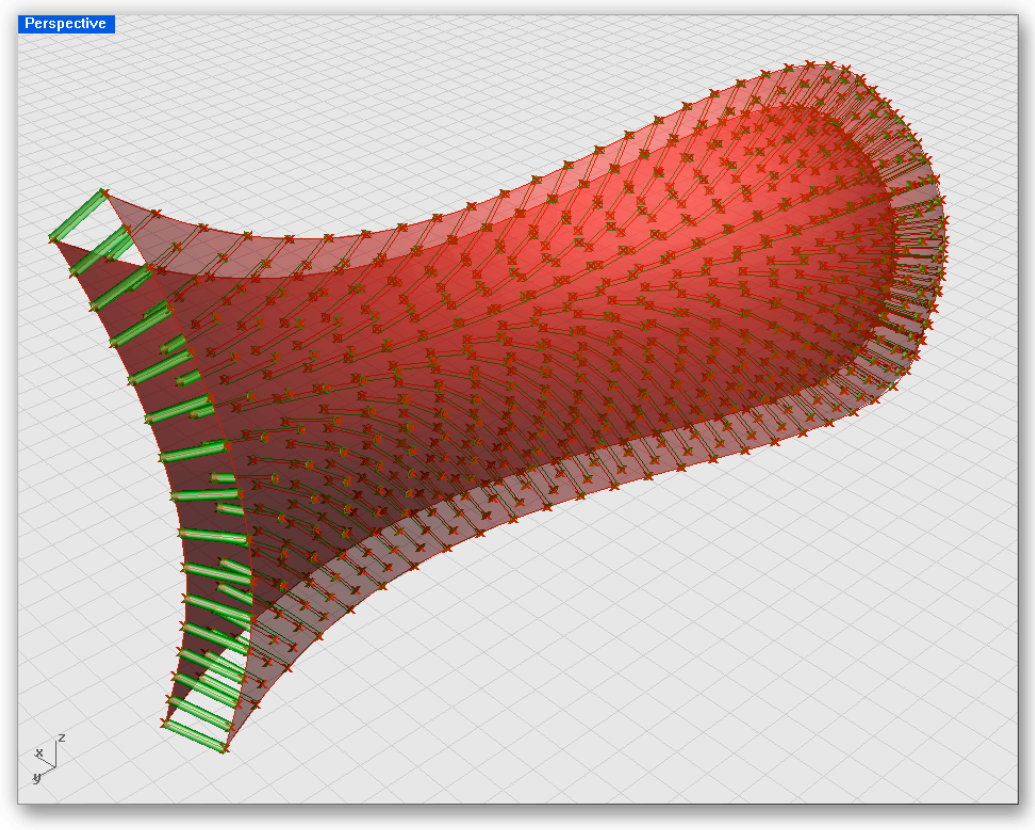
There are more auto-conversion translations and whenever more data types are added, the list grows. That should be enough background on surface types to begin looking at a few different examples.

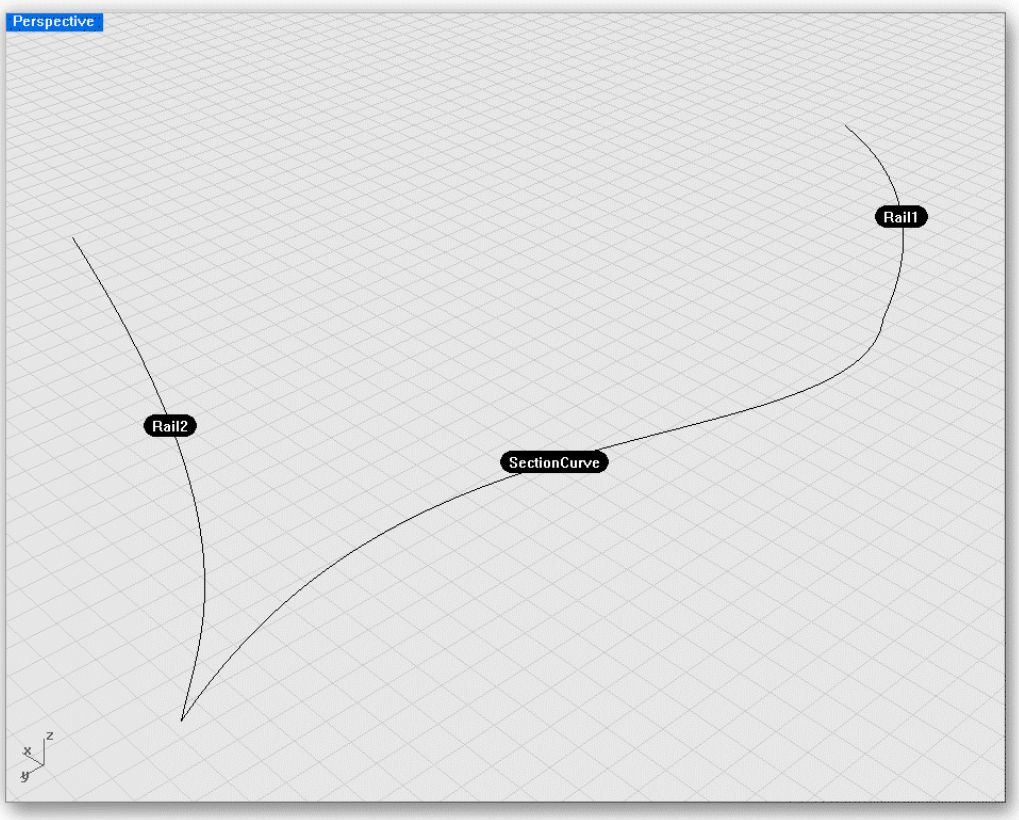
11.1 Surface Connect

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.



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/Specia/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

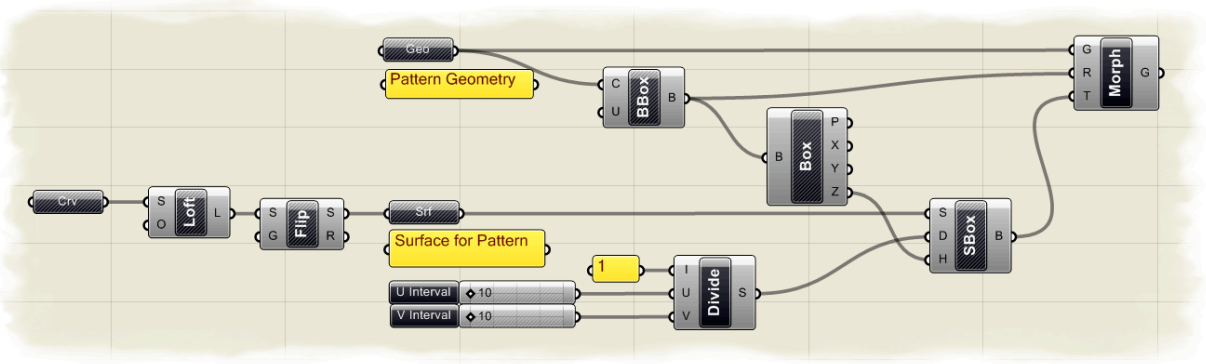
  It's as easy as that. You should now see a series of lines connecting each point of the inner surface to the corresponding point on the outer surface. We can take the definition a step further by giving each line some thickness.
- Surface/Freeform/Pipe - Drag and drop a Pipe component onto the canvas
- 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

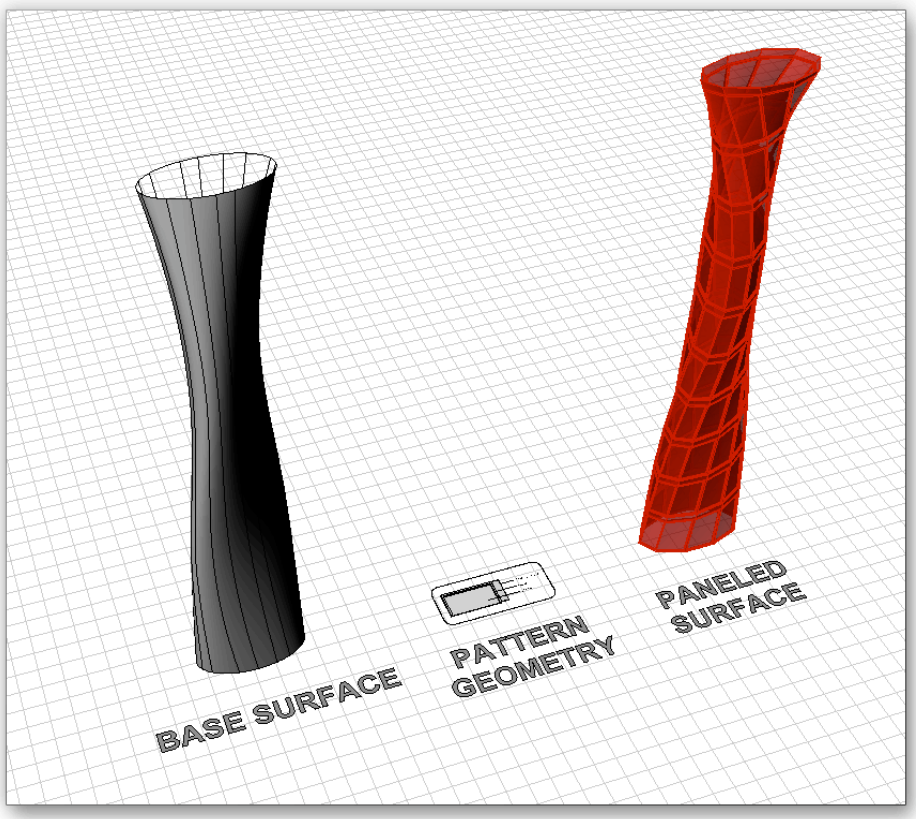
Mcneel recently released an excellent free plug-in for Rhino called **Paneling Tools**. The plug-in, among other things, allows you the ability to propagate specific geometric modules over a given surface. Grasshopper also has some components that can be used to re-create the Paneling Tools method and 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 new Morphing components included in GH version 0.5.0099.

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.

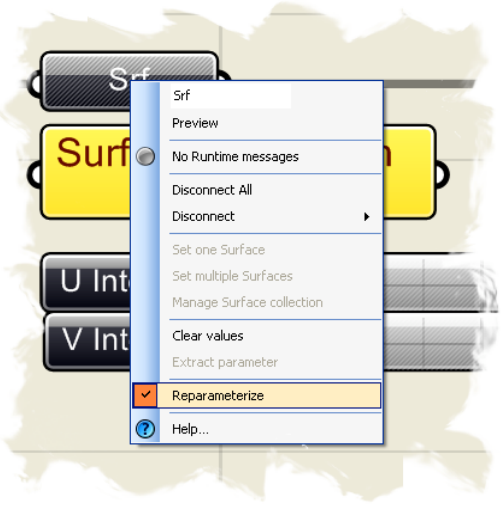
Let's start the definition by creating a lofted surface:

- 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 way with Grasshopper graphic interface. 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

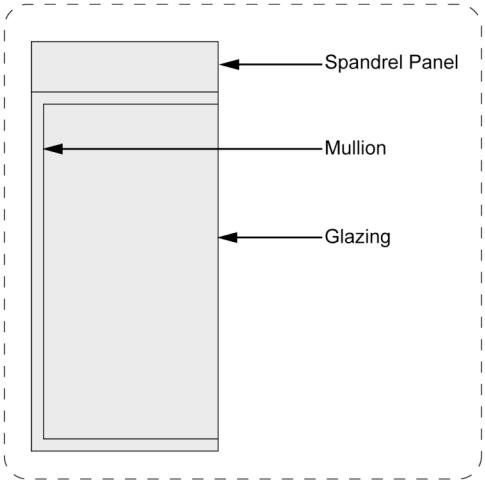


surface will not subdivide correctly if it is not reparameterized into an interval between zero and one.

- 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.
- Params/Primitive/Integer – Drag and drop and Integer component onto the canvas
- Right-click on the Integer component and set the value to 1.0
- Connect the Integer output to the Divide Two Dimensional Interval-I input

  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

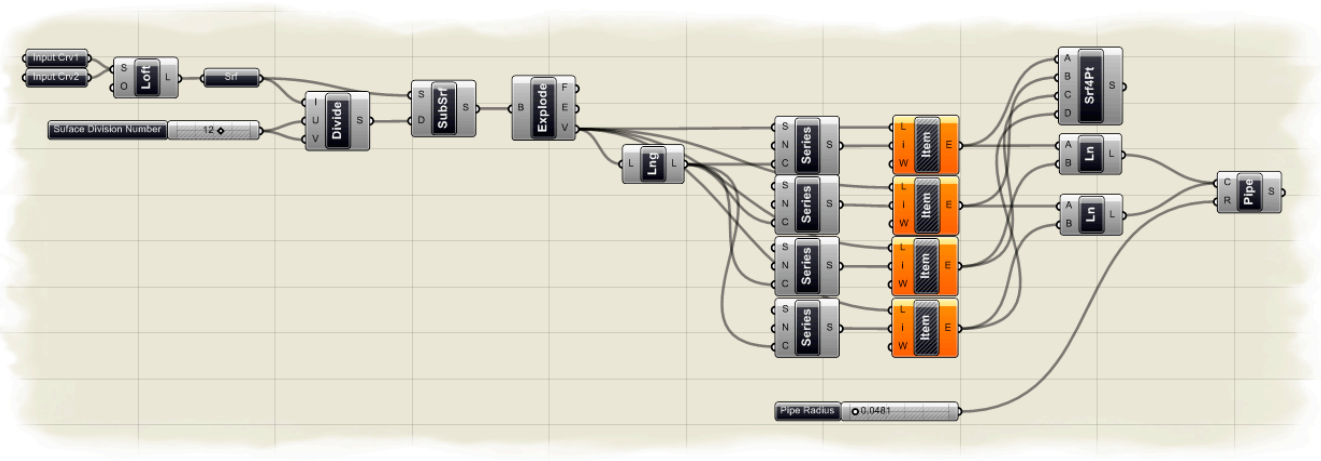
  We're using the Bounding Box component for two reasons. First, the Bounding Box component will help us determine the height of our geometric pattern. Since we have only used rectangular boxes as our pattern, the height would be pretty easy to determine. However, if you had chosen a more organic shape 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.
- 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 Surface Box component and turn the 'Preview' off

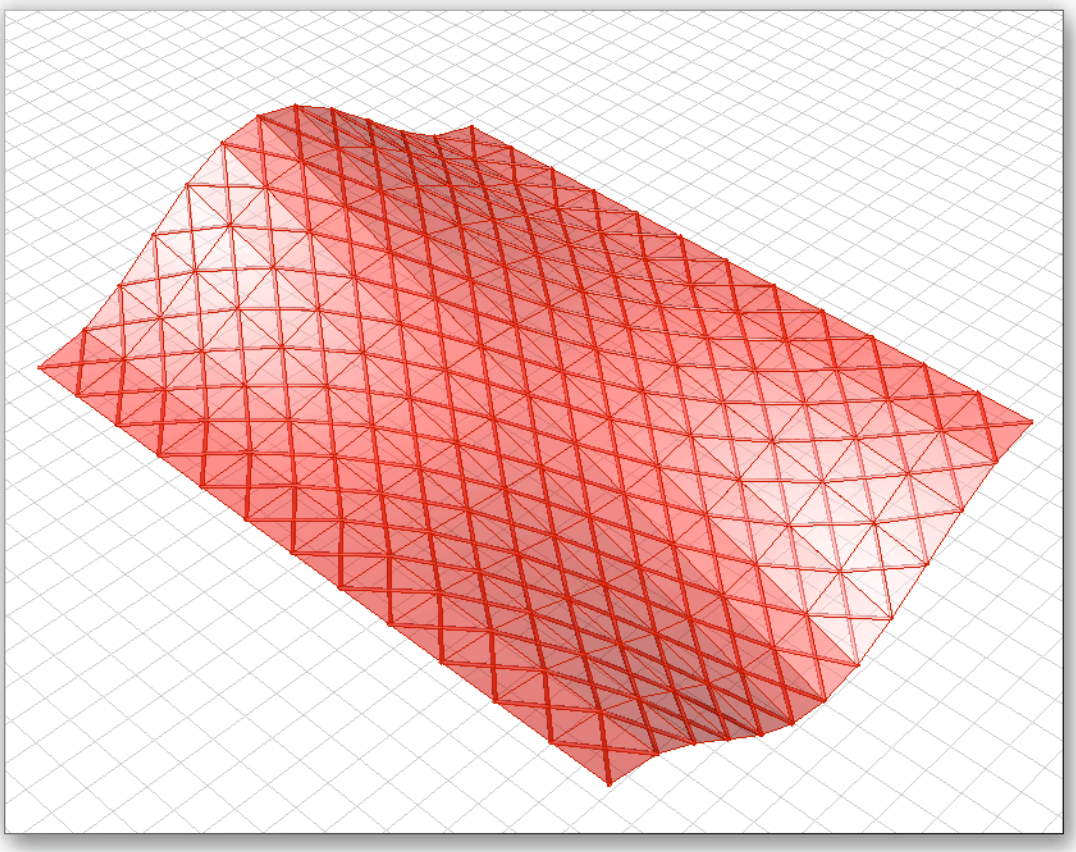
  Phew. If your brain isn't about to explode, I'll try to explain the last part of the definition. We've input the pattern geometry into the Morph Box component, which replicates that pattern over each subdivision. We've used the Bounding Box of our window system as our reference geometry, and we input 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.



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:

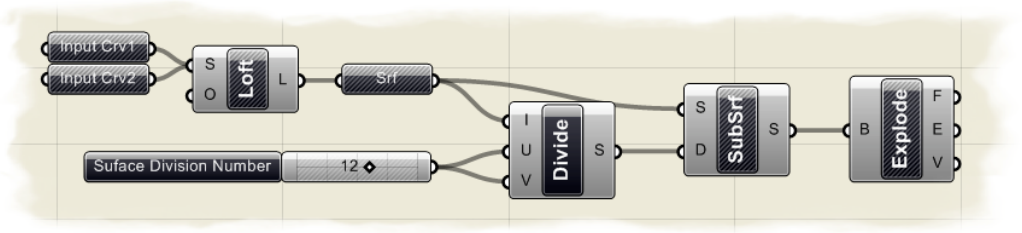
- Params/Geometry/Curve – Drag and drop **two** Curve components onto the canvas
- Right-click the first Curve component and rename it to "Input Crv1"
- Right-click the Input Crv1 component and select "Set One Curve"
- When prompted, select one of the curves in the Rhino scene
- Right-click the second 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.



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 index the list of points in order create a diagonal structural system.

- Logic/Lists/List Length – Drag and drop a List Length component onto the canvas
- Connect the Brep Components-V output to the List Length component
- Logic/Sets/Series – Drag and drop **four** Series components onto the canvas
- Right-click on the **first** Series component and set the following:
  - S-input: 0.0
  - N-input: 4.0
- Right-click on the **second** Series component and set the following:
  - S-input: 1.0
  - N-input: 4.0
- Right-click on the **third** Series component and set the following:
  - S-input: 2.0
  - N-input: 4.0
- Right-click on the **fourth** Series component and set the following:
  - S-input: 3.0
  - N-input: 4.0
- Connect the List Length-L output to **each** of the four Series-C inputs

  Those of you who often pay attention to the fine details probably noticed that we have four components whose start value increases by one for each new Series. There's a specific reason for this, but let's create a simple hypothetical situation to explain what's really going on. Let's say we have a surface where we have set the subdivisions to 2 in each direction, so that there are actually 4 sub-surfaces created. Each sub-surface will have 4 corner points, so we have a list of 16 points. However, we need to know how those 16 points are arranged in the list in order to make specific diagonal connections. Luckily for us, the Brep Components creates the list in order… meaning, it starts at the first sub-surface and evaluates the position of the four corner points first (let's say it goes in a clockwise direction) before moving to the second sub-surface and doing the whole procedure all over again. So, if we wanted the lower left hand point of each sub-surface, we would need to retrieve every fourth point from the lists (index numbers: 0, 4, 8, and 12). Likewise, since the Brep Components created the list going in a clockwise direction, if we wanted the upper left point of each sub-surface, we would need to retrieve every fourth point from the list but with a different start value (index numbers: 1, 5, 9, and 13). The same process works for the entire set of points. The four Series component we just created will create a list

of numbers that we can feed into a List Item component to retrieve each corner point of all of our sub-surfaces.

- Logic/Lists/List Item – Drag and drop **four** List Item components onto the canvas
- Connect the Brep Components-V output to **each** of the four List Item-L inputs
- Connect the first Series-S output to the first List Item-i input
- Connect the second Series-S output to the second List Item-i input
- Connect the third Series-S output to the third List Item-i input
- Connect the fourth Series-S output to the fourth List Item-i input
- Right-click on the Brep Component and turn the 'Preview' off

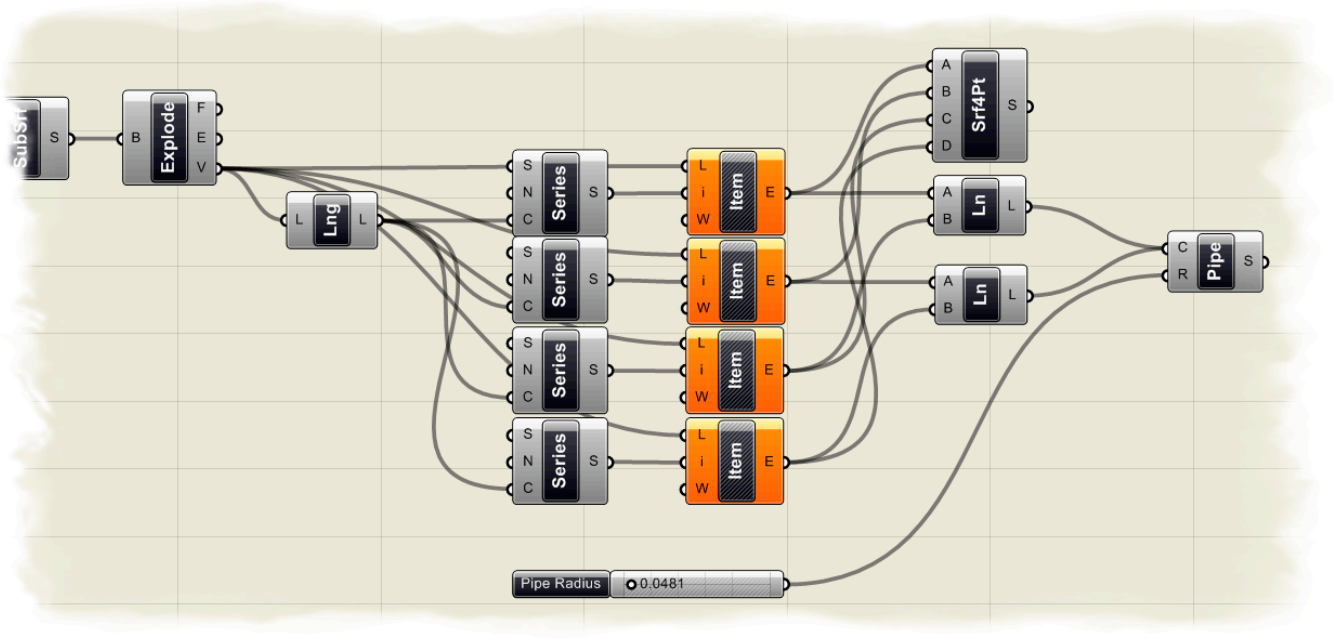
  Now each List Item component will correspond to one of the corner points of each sub-surface. You can test this by selecting one of the List Item components and watching which points are highlighted in Green in the Rhino scene. Now that we have four components that correspond to each corner point, we can simply use the Line component to join diagonally opposed points.
- Curve/Primitive/Line – Drag and drop a **two** Line components onto the screen
- 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

  Depending on how your lists were generated, you may or may not see a diagrid of line between your points. You may need to try a different combination of connections between List Item components and the Line components. In our Line components, we connected the first and the third List Item components; whereas we connected the second and fourth components to the second Line component. If you aren't seeing a diagrid, simply connect a different combination until you get the desired results.
- 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



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.

Notes