2_data
how to manage data in Grasshopper

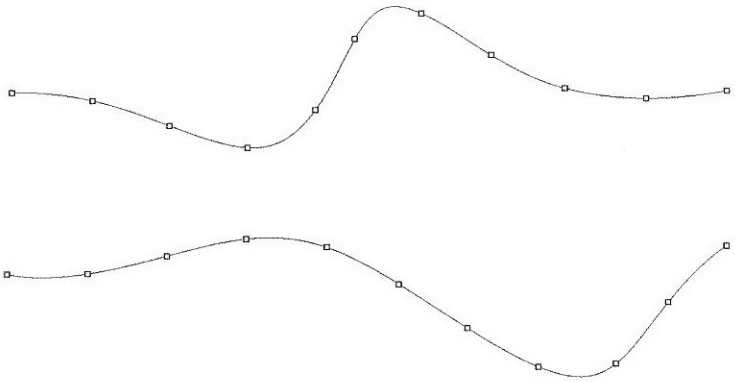
> "Our ability to do great things with data will make a real difference in every aspect of our lives".
>
> Jennifer Pahlka

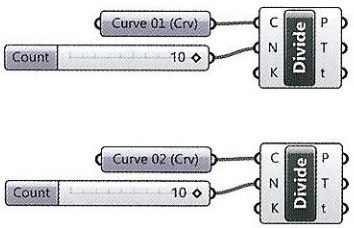
Digital modeling has traditionally been based on virtual manipulation, by the pseudo physical presence of the mouse and keyboard in the digital environment. Algorithmic modeling, conversely, is based on logic and is void of "physical" manipulation. Algorithms rely on the ability to establish conceptual associations between geometry and mathematics. In other words, **data is manipulated instead of digital objects**.

An algorithm can be imagined as a network of data streams, and in order to selectively operate on the network, is necessary to filter, divert and modify data to virtually manipulate geometry. Within this context mathematics and logic are the mouse and keyboard.

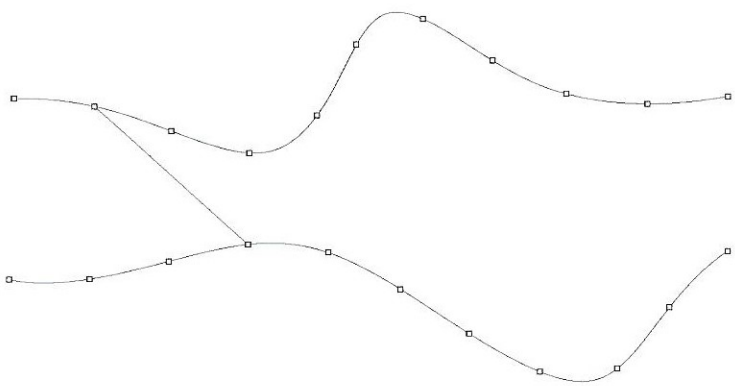
2.1 Filters

One of the consequence of the lack of "physical" interaction in Grasshopper is the seeming difficulty in selecting single items among a set of objects. In the following example two curves drawn in Rhino are imported into Grasshopper (by two *Curve* components renamed as Curve 01 and Curve 02) and then divided into ten parts. As result, eleven points are generated on each curve.

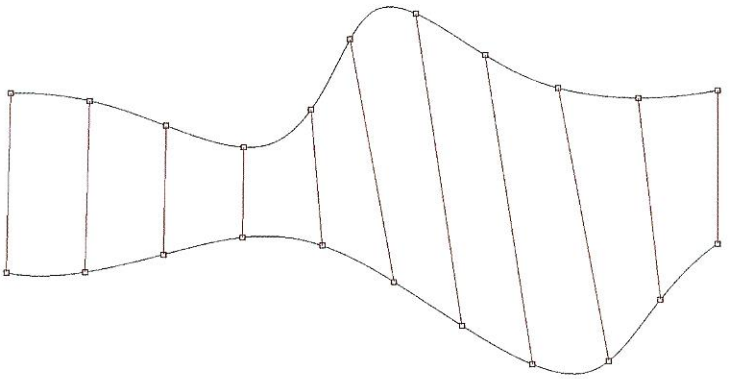


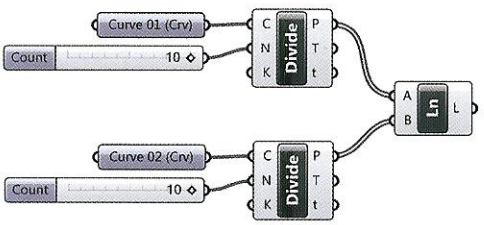


A simple operation such as connecting two arbitrary points among the set of 22 division points (see following image), seems impossible to get using the *Line* component, since we don't know yet the techniques to select single items.

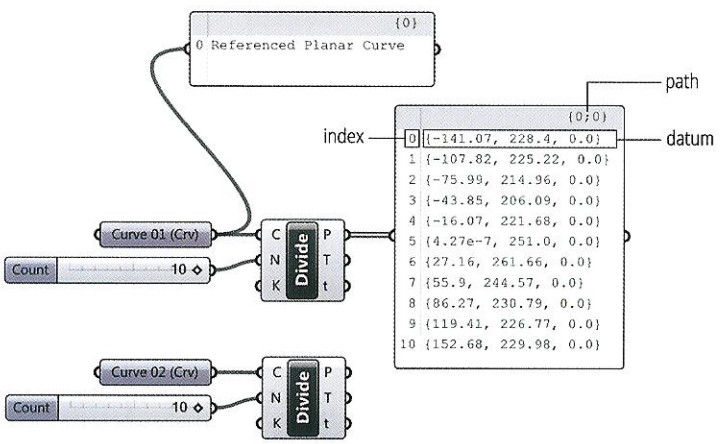


In fact, connecting a *line* component to the P-output of the two *Divide Curve* components will instruct Grasshopper to generate eleven lines connecting the entire set of corresponding points.



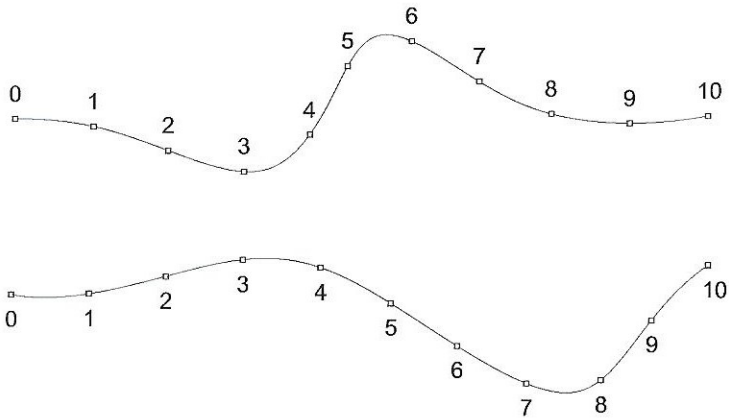


In order to analyze the output of a component a ***Panel*** (Params > Input) can be connected to a component's output to display the data structure. Panels are visualization tools that can be used to visualize output data.



Grasshopper structures data into paths i.e. hierarchical levels that contain **Lists**. Lists are further subdivided into items which are indexed with a natural number (0 to i) to reference an item's position

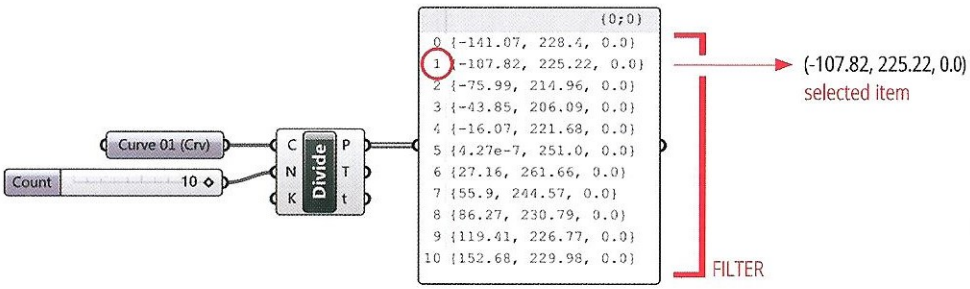
within the list. For example, the data output of the *Divide Curve* components slot (P), is a triplet of coordinates mathematically representing an {x,y,z} point. Each point or datum is referenced by an index, the first point in the list (-141.07, 228.4, 0.0) has an index of 0. Indices always start from 0 and count in integers to i. Paths will be discussed later (chapter 5).



Grasshopper sorts data based on the way data is set. For example, objects set from Rhino will be organized in Grasshopper in the same order they are selected from Rhino. In the case of *Divide Curve*, points are sorted according to the curve direction.

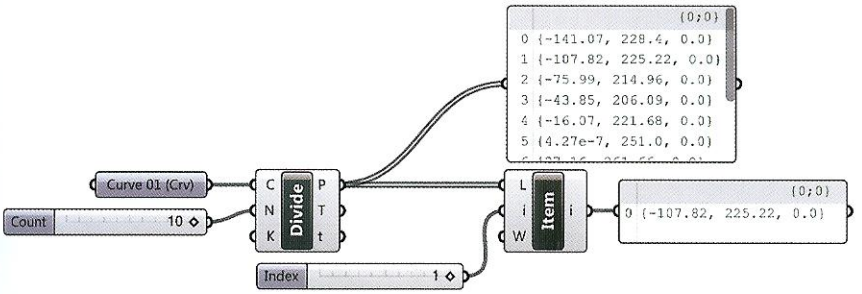
2.1.1 List Item: select one item from a list

Understanding lists enables users to easily select a specific datum within a list by referring to its index by a **filter**. Grasshopper has many components to filter and direct data.

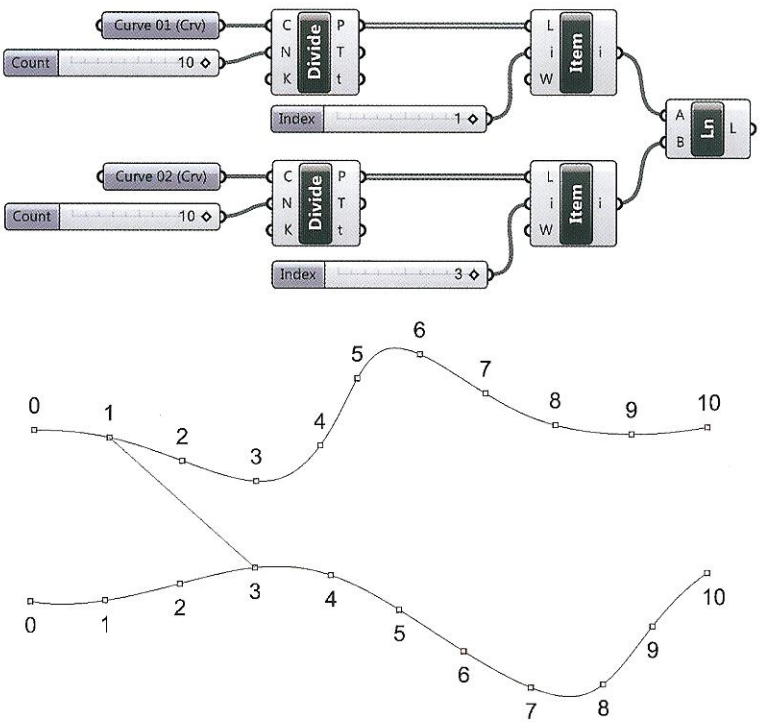


For example, the component ***List Item*** (Sets > List) is a filter to select a specific index item. To select

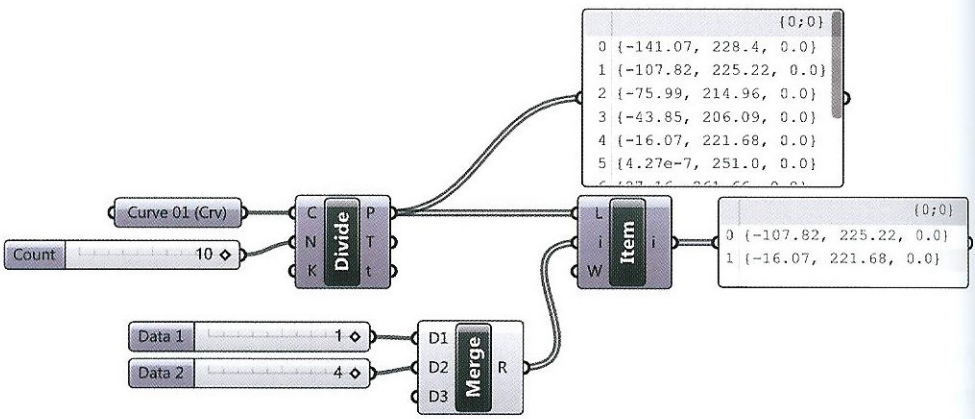
an item, connect a list to select from into the L-input of *List Item* and an integer to the item slot (i). The integer's value specifies the item that the filter selects.



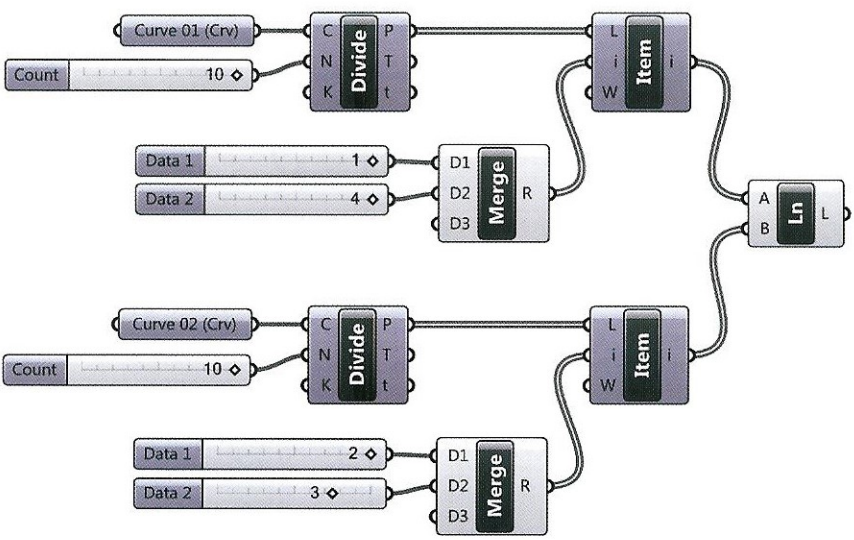
A Line can be generated between the *List Item* outputs (i) by connecting wires from the i-outputs to the A and B inputs of *Line*. The "Index" slider denotes the point selected from the list to be connected as endpoints of the line.

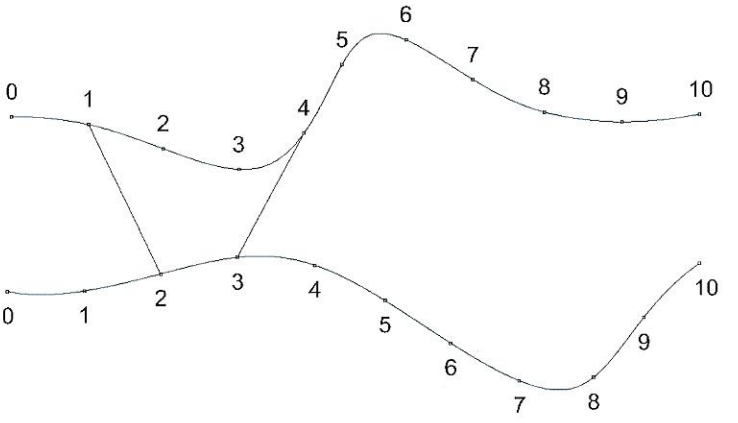


In order to select two or more items, several sliders can be connected to the i-input of the component *List Item*, by using the ***Merge*** component, or by pressing and holding the *shift* button on the keyboard while connecting multiple wires.



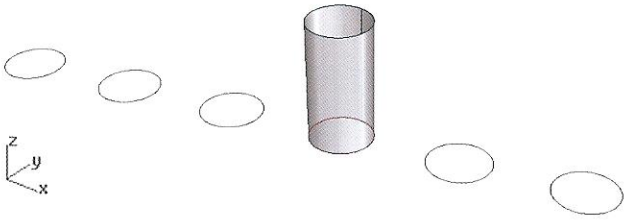
In this way, two lines can be generated by specifying two index items from the P-output using the *List Item* component.

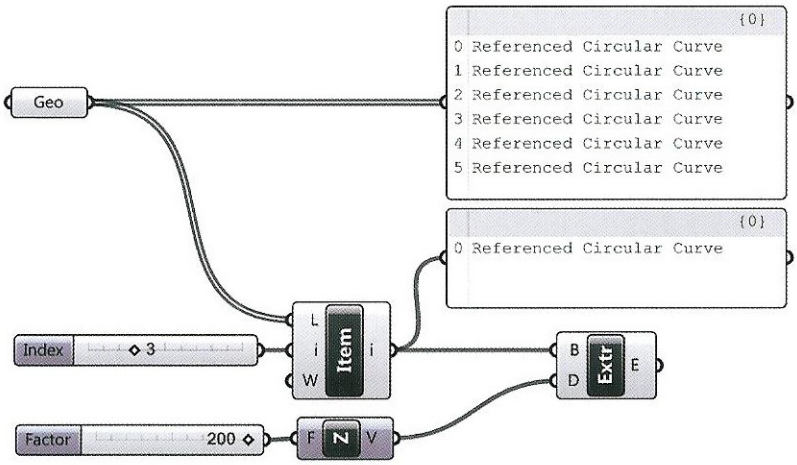




The component *List Item* can be used to filter any data formatted in a list. For instance, the *List Item* component can be used to select a specific circle within a list of circles.

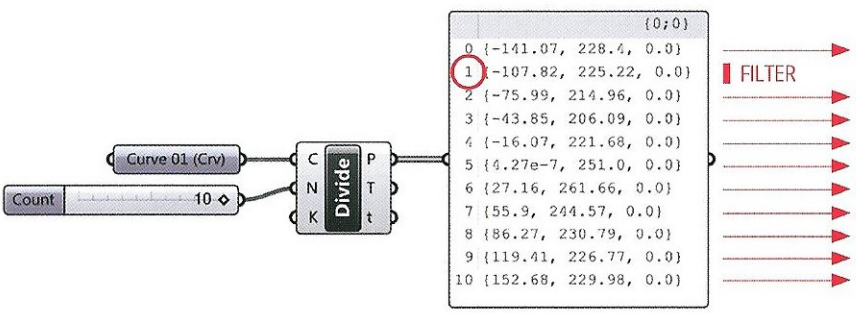
For example, using a *Geometry* container six circles are set from Rhino, using the *Set Multiple Geometries* option from the context menu. Grasshopper sorts the circles according to their position on the x-axis. In order to select and extrude the fourth circle, the index item 3 needs to be specified as the input of the *List Item.* The component *Extrude* (Surface > Freeform) is used to extrude the circle according to a distance and direction (D) specified by a vector. In this case the component *Unit Z* (Vector > Vector) specifies a vector in the z-direction with a magnitude of 1. Using scalar multiplication a specified magnitude is set using a number slider (factor).



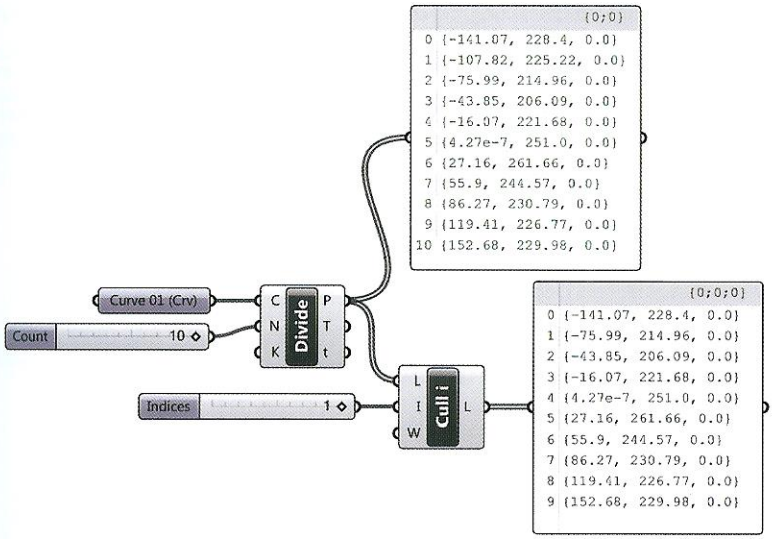


2.1.2 Cull Index: select all data except one item

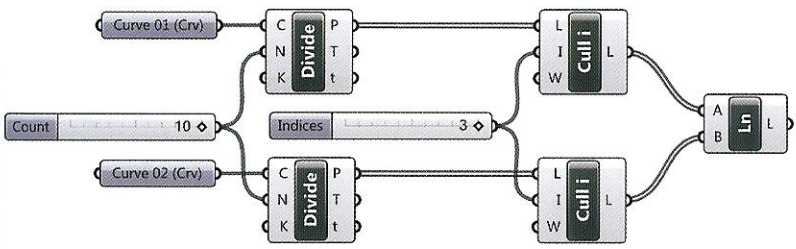
The component ***Cull Index*** (Sets > Sequence) performs the reverse fuction of the *List Item* component: *Cull Index* deletes specified index items from a list.

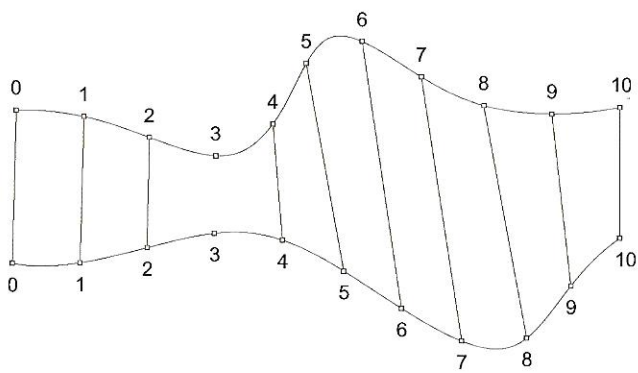


For instance, to cull index 1 from the list, a slider set to 1 is connected to the i-input of the *Cull Index* component (see following image). As a result, if the initial list has N items the output of *Cull Index* is a new list which hosts N minus the number of culled items.



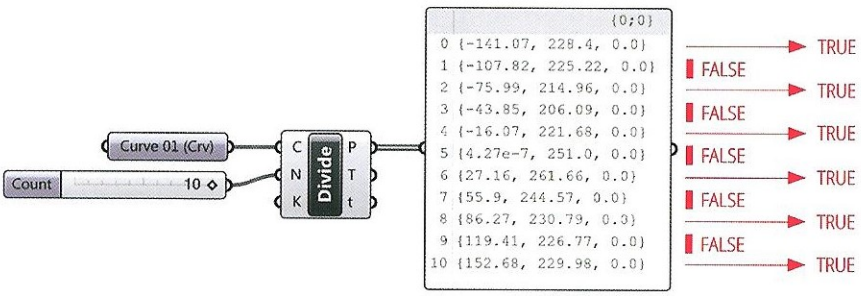
In the following example a set of lines is created between couples of corresponding points excluding the points which are identified by the index 3.



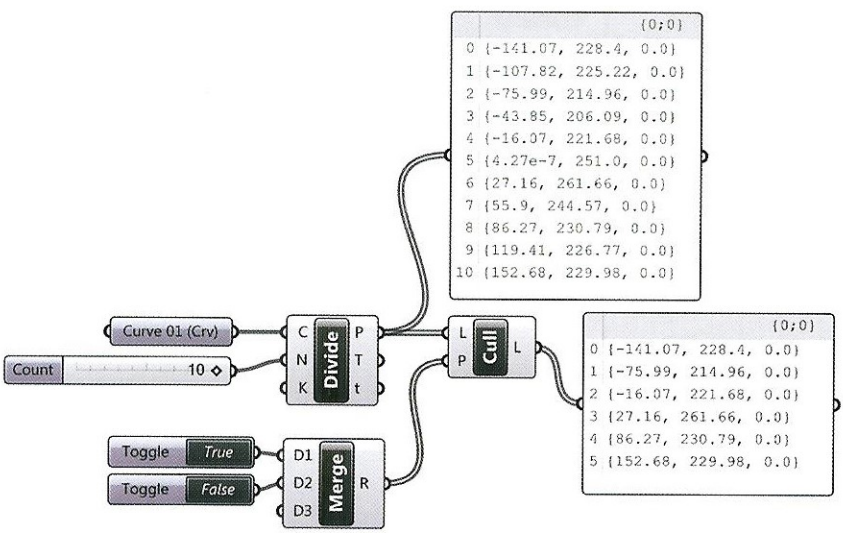


2.1.3 Cull Pattern: select items using a repeating model

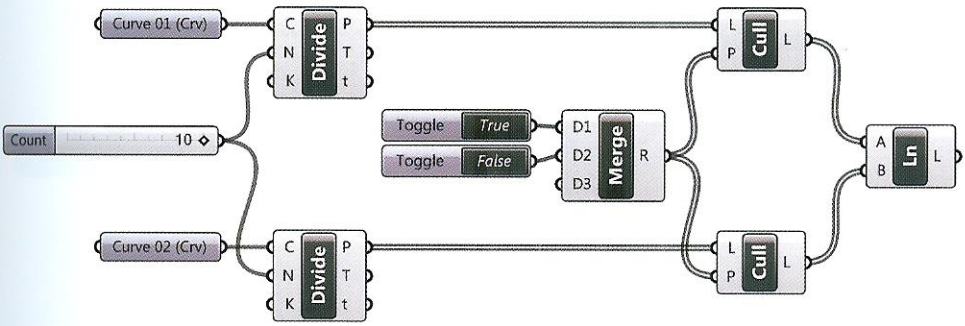
Repeating patterns of True and False values (***Boolean*** values) can be used to select and exclude, respectively, items from a list. The component ***Cull Pattern*** (Sets > Sequence) is used in Grasshopper to filter lists according to a repeating pattern of Boolean values.

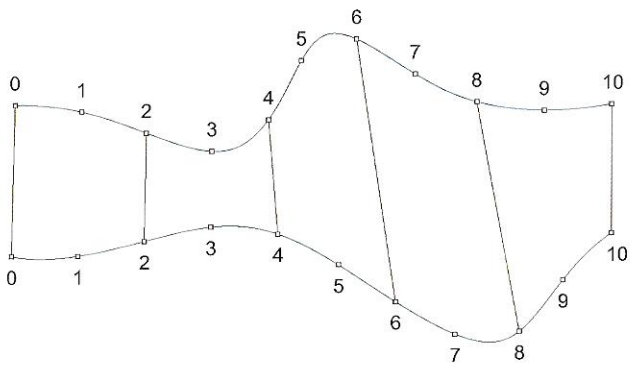


The component ***Boolean Toggle*** (Params > Input), can be used to define culling patterns. The component output can be changed by left clicking the True/False portion of the button, to declare either True or False respectively. Multiple *Boolean Toggle* components can be combined into a single list using the *Merge* component.



The declared True/False pattern is repeated for the length of the list, meaning if the pattern is (True, False) and the list has a length of 4 (contains index numbers 0,1,2,3) the pattern will repeat resulting in a culling pattern (True, False, True, False). Referring to the previous example, lines can be drawn between alternate couples of corresponding points.





Repeating models can be based on patterns greater than two declarations. As shown in the following image the *Cull Pattern* is based on a repetition of three Boolean values (True, True, False).

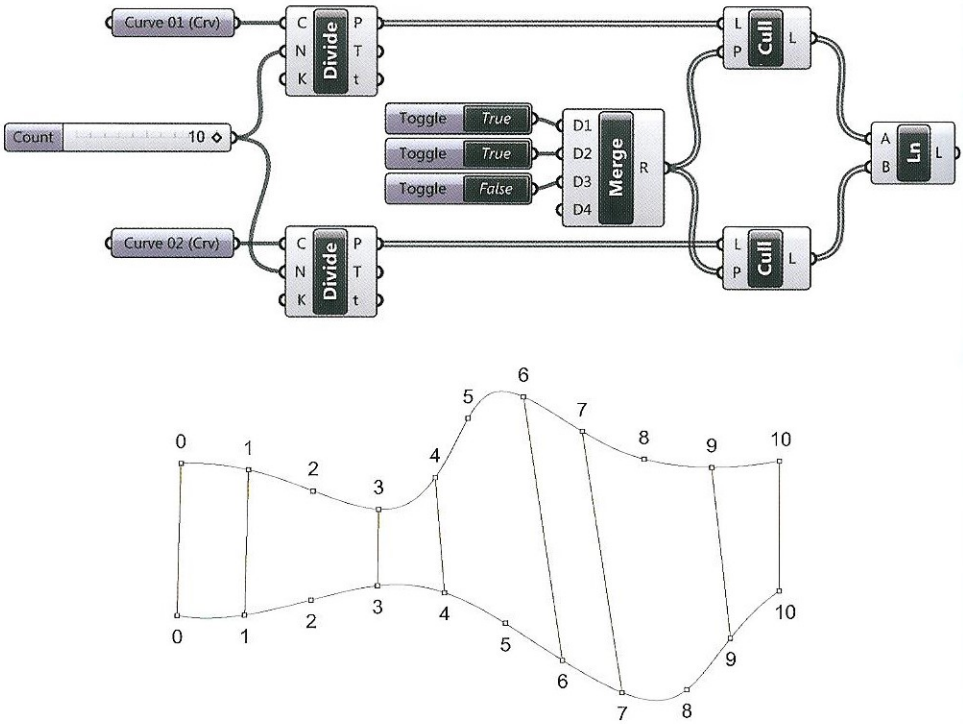
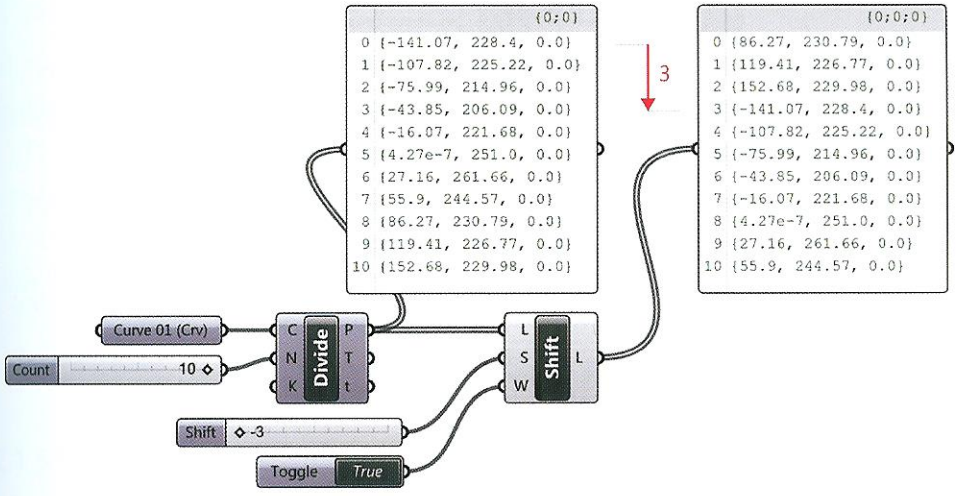


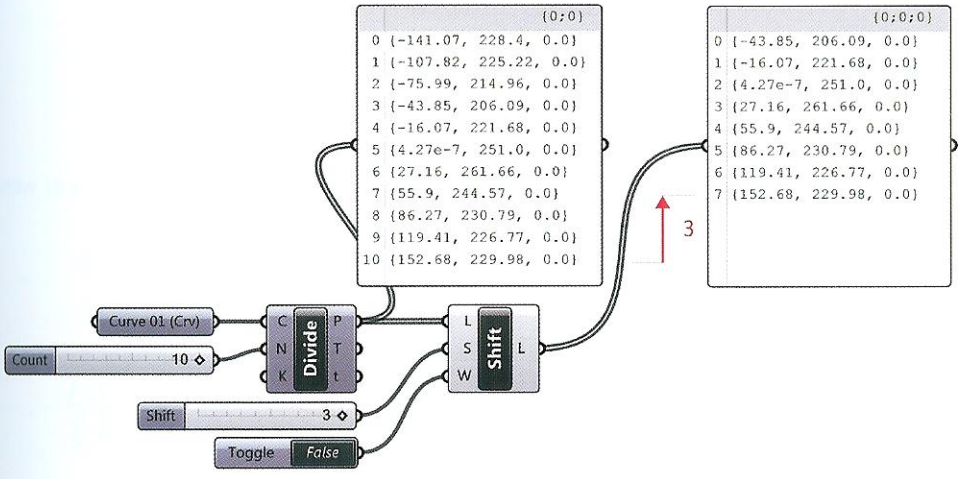
FIGURE 2.1
The image shows the result of *Cull Pattern* based on a repetition of three Boolean values: True, True, False.

2.1.4 Shift List: offset data in a list

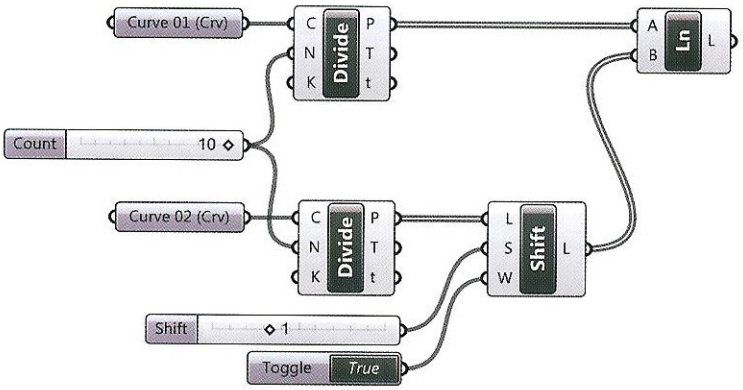
The component ***Shift List*** (Sets > List) shifts index numbers, upward or downward. The component's S-input is the shifting offset, the L-input is the list to shift, and the W-input (Wrap) is a Boolean Parameter which declares if the resulting shifting should be re-appended. Negative numbers shift the list downward and positive numbers upward. For example, If the shifting offset (S) is set to -3 and the wrap (W) is set to true, the list's original indexes are shifted 3 units towards the end of the list and the list is re-appended, meaning the bottom three indexes 8, 9, and 10 become indexes 0, 1, and 2 respectively.

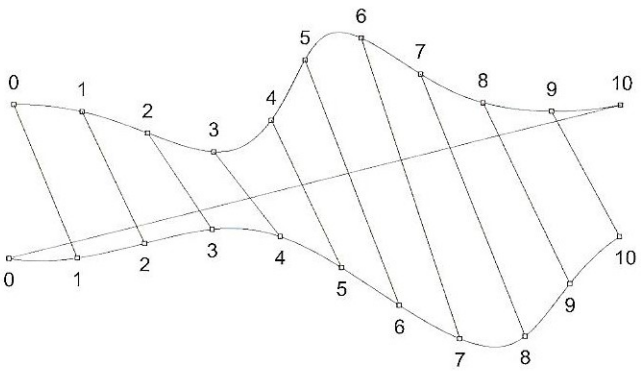


If the shifting offset is set to 3 and the wrap (W) is set to false, the original indexes are shifted 3 units toward the beginning of the list, and the first three values are culled.

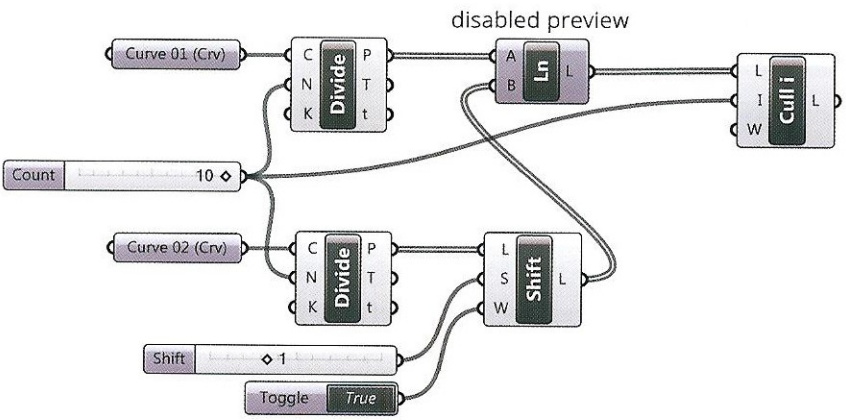


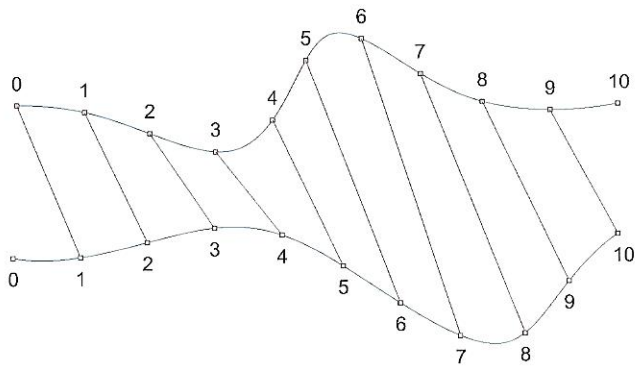
The following algorithm generates a set of lines connecting the generic point i on the first curve with the point i+1 on the second curve. The resulting output list of the *Shift List* component will be shifted 1 value towards the beginning of the list and the list will be re-appended with respect to the input list, meaning the index 10 of *Curve 01* will be connected to index 0 of *Curve 02.*





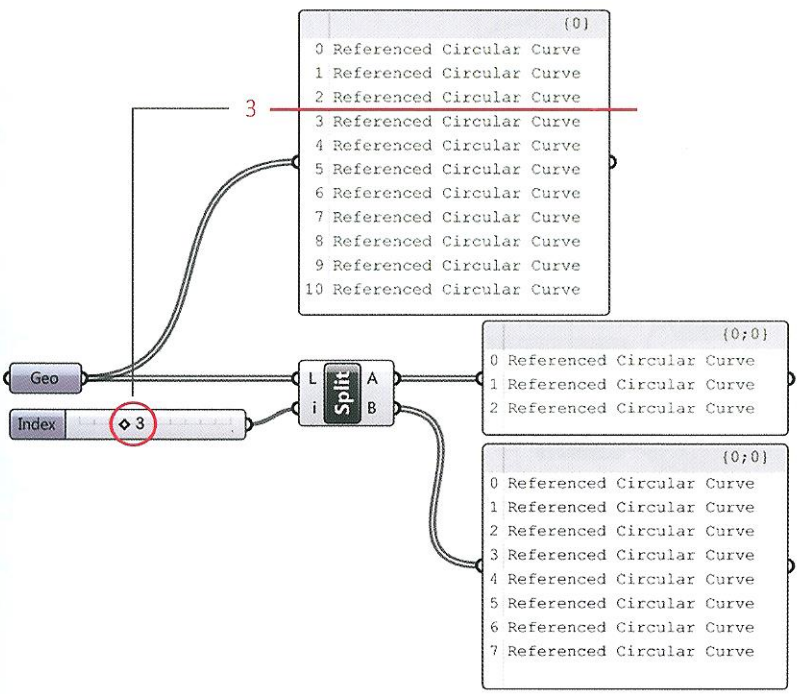
The *Cull Index* component can be used to delete the line connecting the index 10 of *Curve 01* with index 0 of *Curve 02*, by specifying the index number.



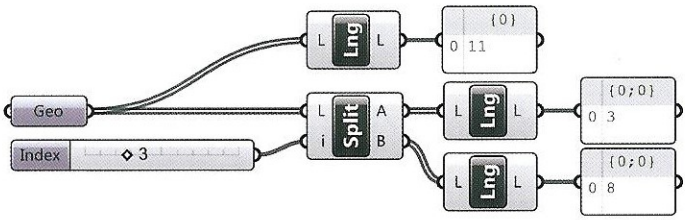


2.1.5 Split List / List Length

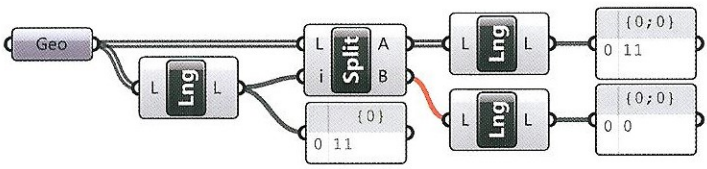
The ***Split List*** component (Sets > List) splits a single input list at a specific index into two lists A and B.



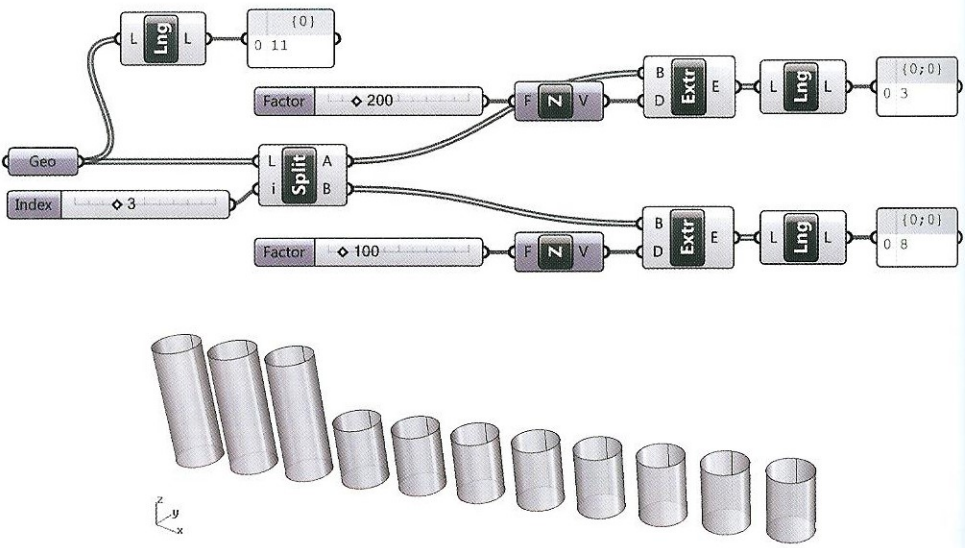
The Component ***List Length*** (Sets > List) calculates how many items there are in a list. To display the lists length, connect a *Panel* to the L-output of *List Length*.



If the splitting index is set equal to the *List Length*, List A will be the same as the initial list and List B will be empty.



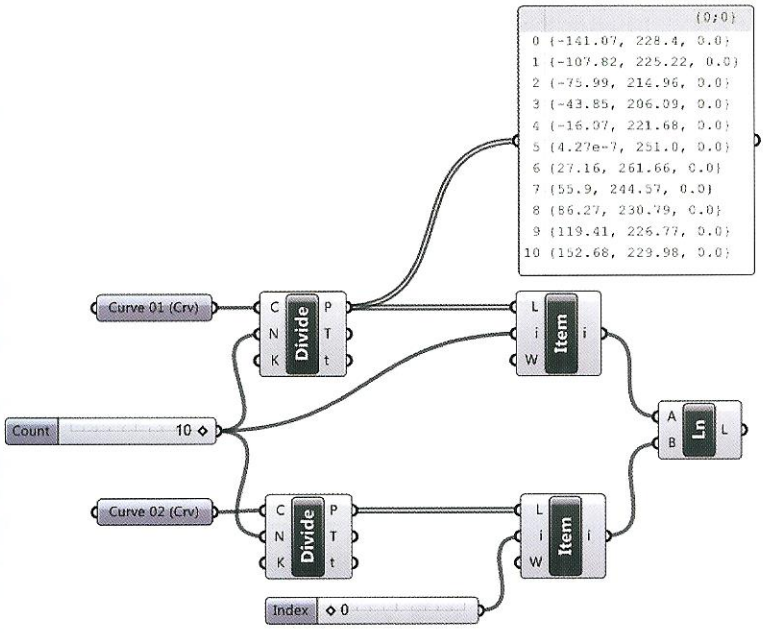
The *Split List* component can be used to carry out different operations on a set of geometries. For instance, after the set is split into two lists, list A is extruded a magnitude of 200 units in the *Unit Z* direction and list B is extruded a magnitude of 100 units in the same direction.



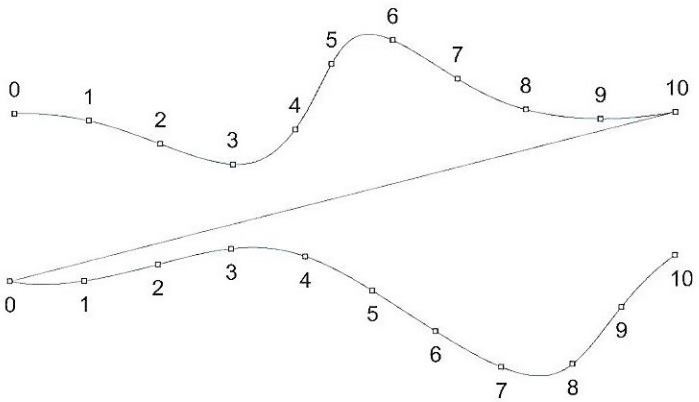
2.1.6 Reverse List

Frequently is useful to reverse the order of a list of data. Referring to the previous example, we want to create a line which connects the "last point" on the *Curve 01* with the "first point" on the *Curve 02*. In addition, such a rule must work even if changing the number of subdivisions through the N-input of *Divide Curve*.

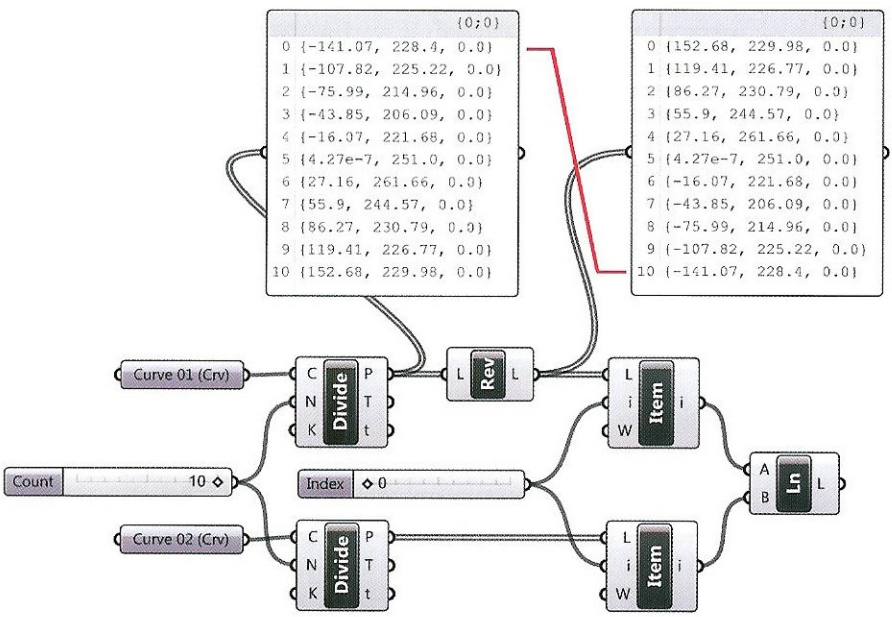
The "first point" can be easily found by a *List Item* component set to 0. The "last point", instead, depends on the number of subdivisions (N). To find this latter, we can rely on two main strategies. The first one consists in creating a "parametric link" using the same slider to "feed" the N-input and the i-input of the first *List Item*.

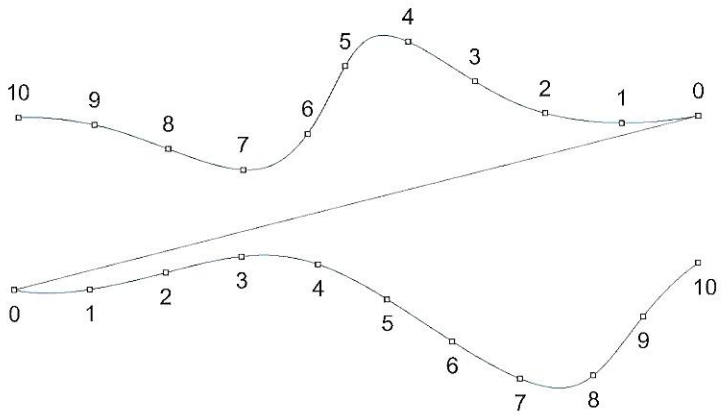


In this way, even if changing the number of subdivisions, the List Item will always select the "last point" on the Curve 01.



The second strategy is based on the component ***Reverse List*** (Sets > List) which reverses the order of a list (in this case the P-output list), meaning the index 10 is converted into 0, the index 9 into 1 etc. Every input slot embeds the functionality to reverse input data by right-clicking the slot and choosing *Reverse* from the contextual menu. Using *Reverse List*, the same slider set to 0 can feed the two *List Item* components.





2.2 Numerical sequences

The components *Series, Repeat Data, Random, and Range* create and manipulate numerical sequences to create lists that manage geometry.

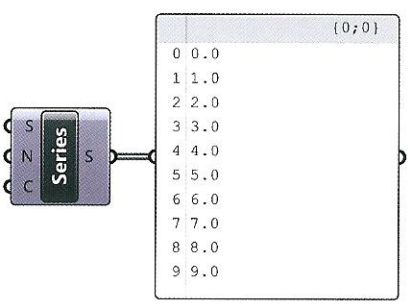
2.2.1 Series

The component ***Series*** (Sets > Sequence) generates numerical sequences according to:

- The first number in the series **(S)**;
- The step size for each subsequent number **(N)**;
- The number of steps in the series **(C)**.

The *Series* component default settings are: S=0, N=1, C=10, resulting in the sequence:

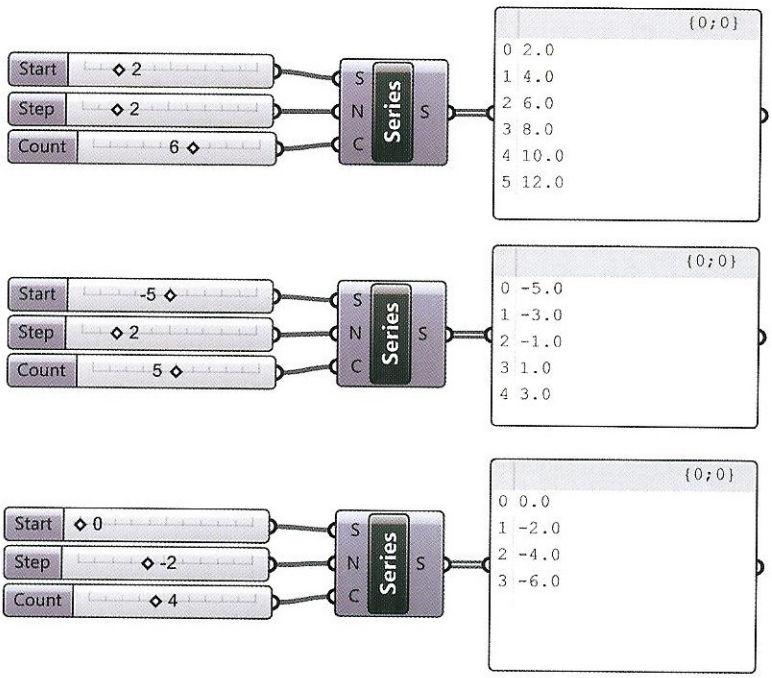
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9).



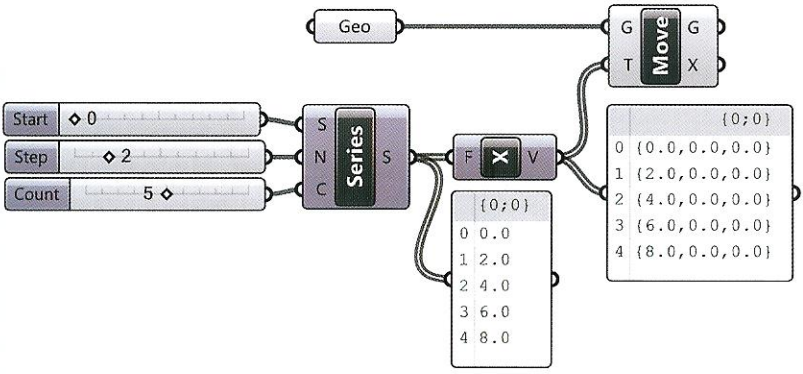
For instance, by setting the parameters to: S=2, N=2 and C=6, a series of 6 numbers is generated which starts from 2 and has a distance of 2 units between a number and its subsequent:

2, 4, 6, 8, 10, 12.

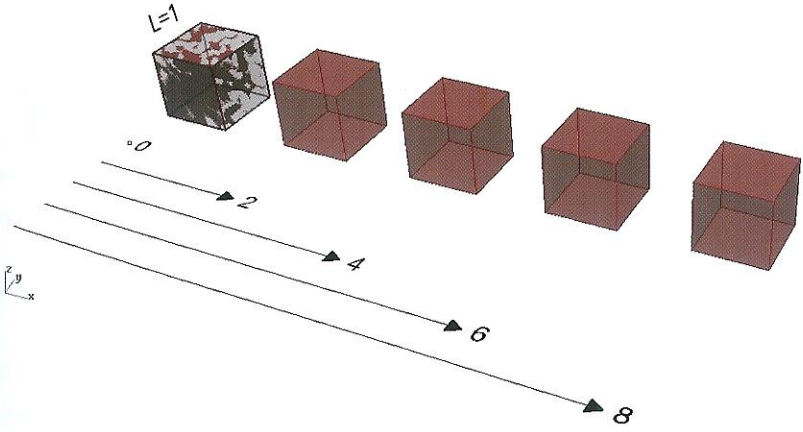
The *Series* component can create infinite unique numerical sequences that are increasing or decreasing based on the input parameters.



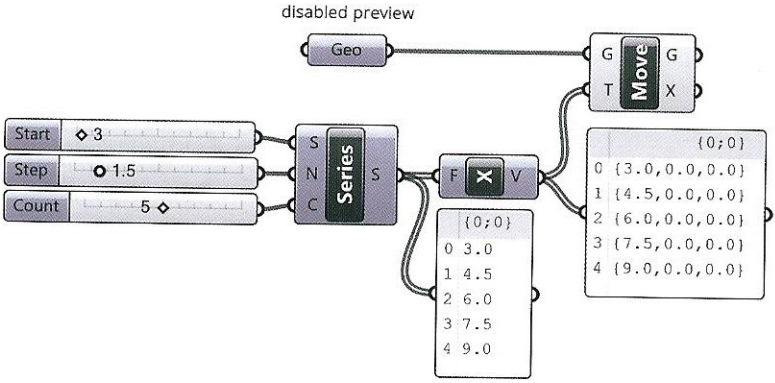
Series is often used along with transformation components, such as *Move*, in order to get multiple transformations. The component *Move* (Transform > Euclidian) translates a geometry (G) according to a vector (T). The *Series* component describes a list of scalar values which can be multiplied by a unit vector to define a translation vector (T) for the *Move* component. For example, a Rhino defined cube can be translated along the x-axis using the *Move* component in conjunction with *Series* component. The *Series* component generates the numerical sequence (0, 2, 4, 6, 8) of scalar values which are multiplied by a unit vector in the x direction resulting in the translation vectors (0x, 2x, 4x, 6x, 8x).



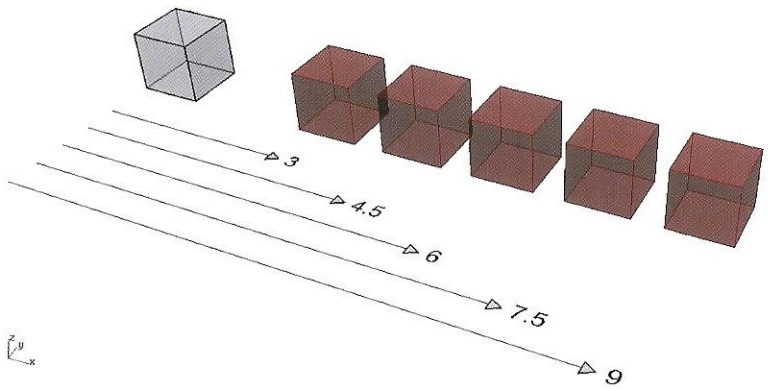
The translation vectors are connected to the T-input of the *Move* component, yielding a series of cubes translated *One Dimensionally*. The first translation vector is 0x, meaning the first translated geometry overlaps with the Rhino set geometry.



If you want to display just the translated cubes (output of *Move*), you have to disable the preview of *Geometry* as well as the preview of the initial object in Rhino (*Hide Objects* command).



The *Move* and *Series* components, used in combination, can create infinite unique vector sequences. The sequences can start at any value, have any step size and count to any length.

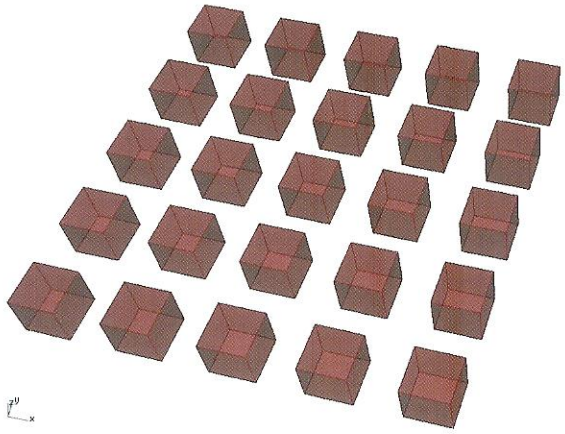


For instance, if (S) is equal to 3, (N) is equal to 1.5 and (C) is equal to 5, the sequence (3.0, 4.5, 6.0, 7.5, 9.0) will result. If the sequence is multiplied by the *Unit X* vector the translation vectors (3.0x, 4.5x, 6.0x, 7.5x, 9.0x) will be defined. Inputting the translation vectors into the T-input of the *Move* component defines the *One Dimension* translation. The first translation vector is 3.0x, meaning the first translated geometry will not overlap with the Rhino set geometry.

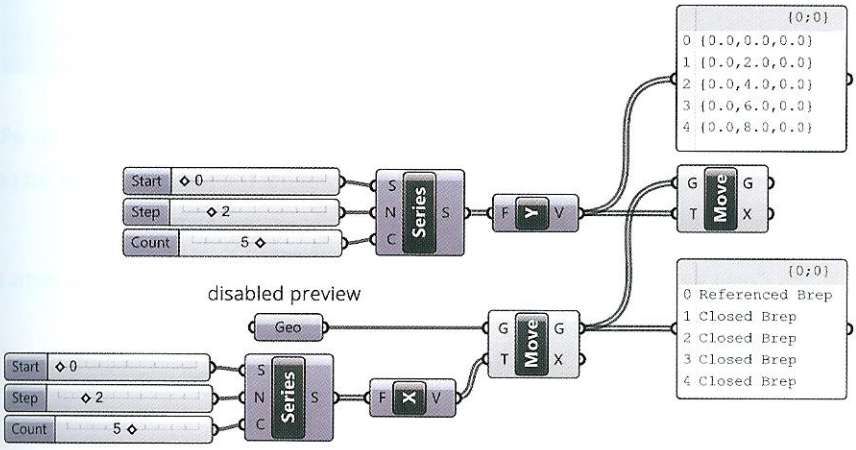
2.2.2 Data Matching

Usually a component receives two or more wires and as many data streams. When data enter into a component they are matched according to a default *logic* which must be understood in order to get specific results. An example will explain this concept.

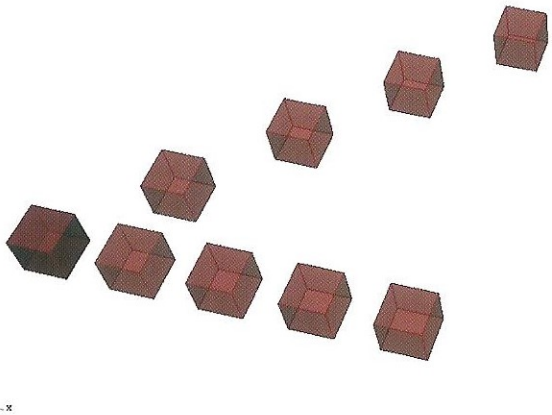
Starting from the previous algorithm we aim to get the following cube grid (based on a 5 x 5 grid).



As first attempt we can modify the algorithm by adding a new *Move* component connected to a *Series* in order to translate the five cubes (output by the first *Move*) according to the y-axis.

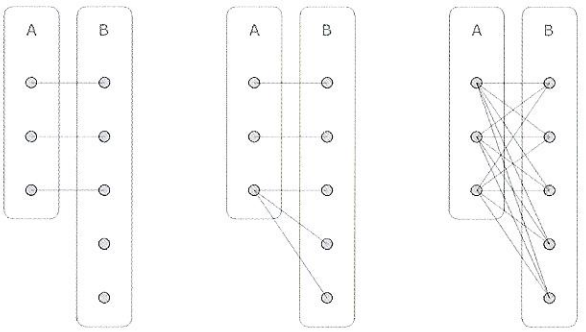


Nevertheless, the final output is quite different from a cube grid: the boxes will create a diagonal arrangement.



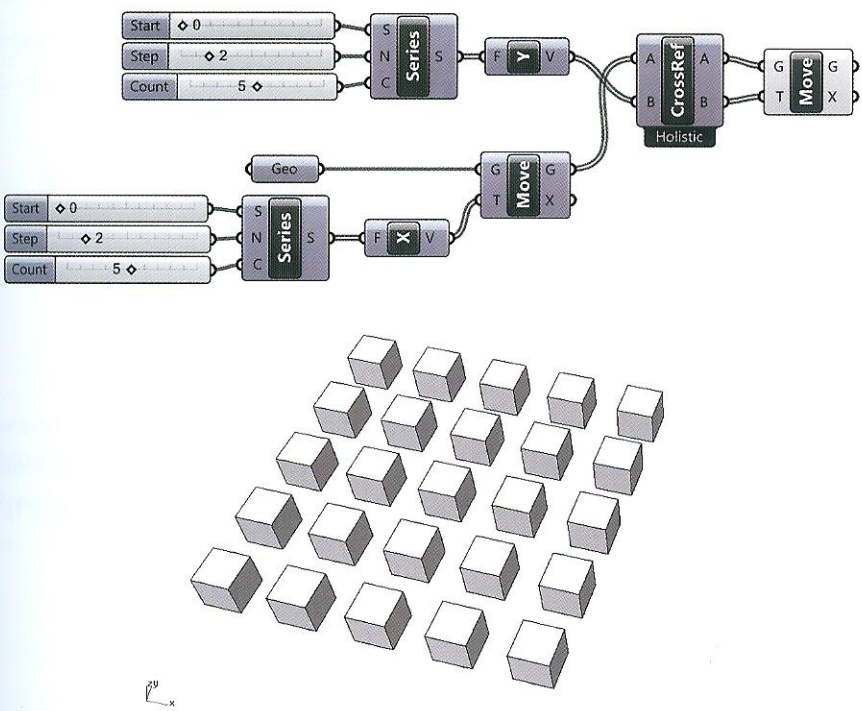
Such a result depends on a particular logic called "**Data Matching**" which, by default, matches – one by one – corresponding items from different lists entering into the same component.
In the example, the second *Move* component matches two flows, the first one transferring geometries (G-input), the second one transferring vectors (T-input). According to this logic the *Move* component moves the first geometry according to the first vector (0 length), the second geometry according to the second vector (whose length is 2) and so on.
In order to get the square grid, Grasshopper must move each cube according to all vectors. This implies that we have to manipulate the matching logic.
Grasshopper matches data according to three main methods:

- The **Shortest List** mode matches each item in the first list with the corresponding item in the second list. If the lists are not the same length the shorter list length will shrink the longer list to the same length.
- The **Longest List** mode (default method) matches each item in the first list with the corresponding item in the second list. If the lists are not the same length the last point in the shorter list will be connected to remaining points in the longer list.
- The **Cross Reference** mode matches each item of the first list with all the items of the second list.

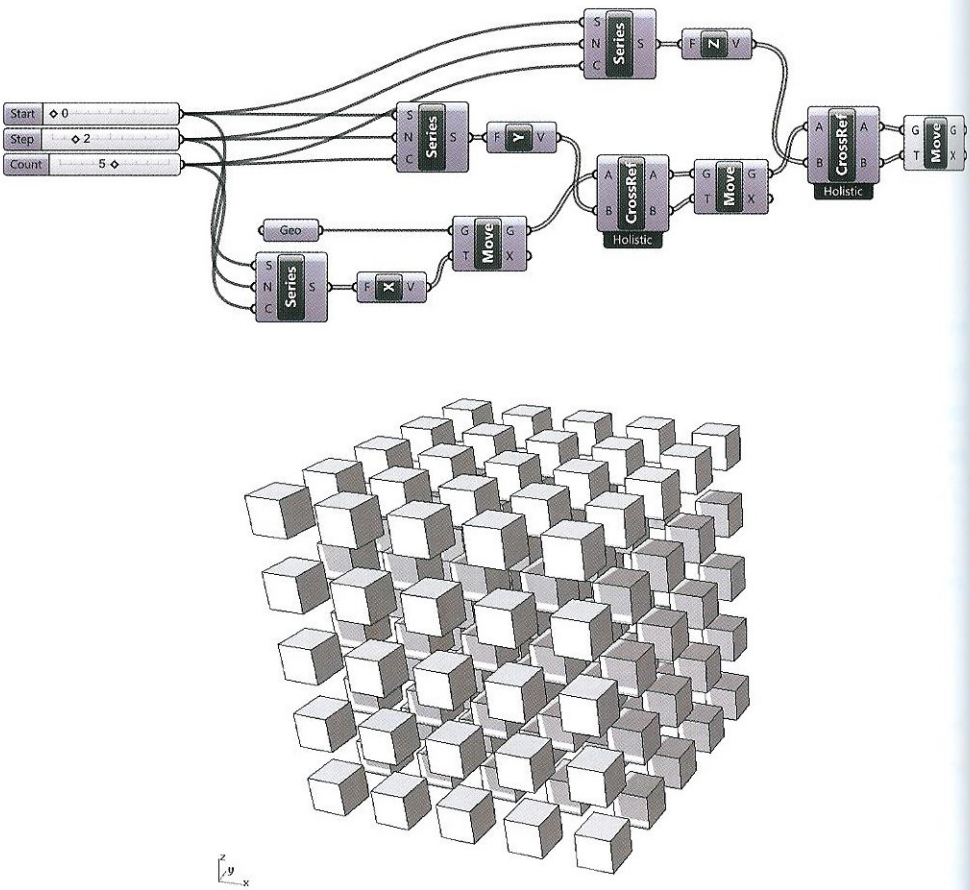


In order to modify the matching logic Grasshopper provides us of three specific components: *Shortest List*, *Longest List* and *Cross Reference* all hosted within the *List* panel of the *Sets* tab.

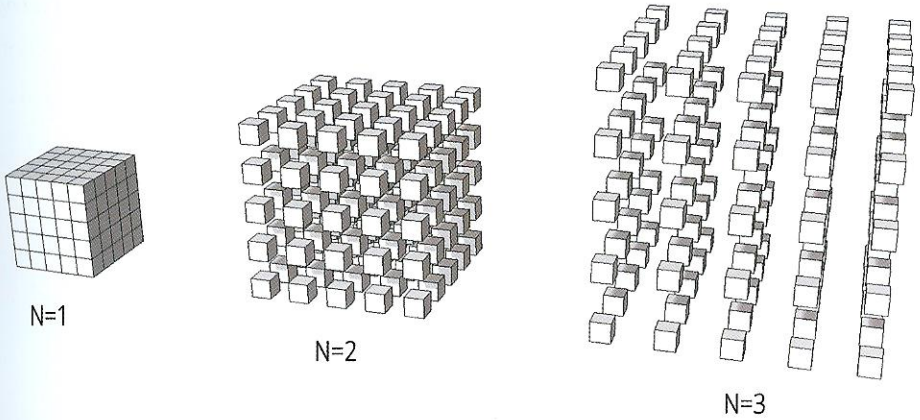
The component *Cross Reference* can be used, in the example, to create a two-dimensional array i.e. the desired box grid, as illustrated in the following image.



The component *Cross Reference* can also be used to create three-dimensional arrays, by connecting a z translation vector sequence and "cross referencing" the data with the second move component.

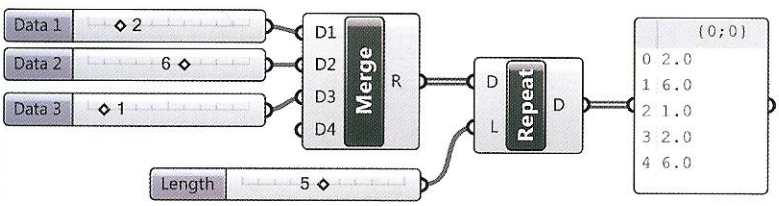


The algorithm uses the same three *Number Sliders* to feed the three *Series* components. In this way we can change the distance between cubes (in all the three directions) by acting on the slider connected to the N-input of the three *Series* (in our example the initial cube has a side length equal to 1).

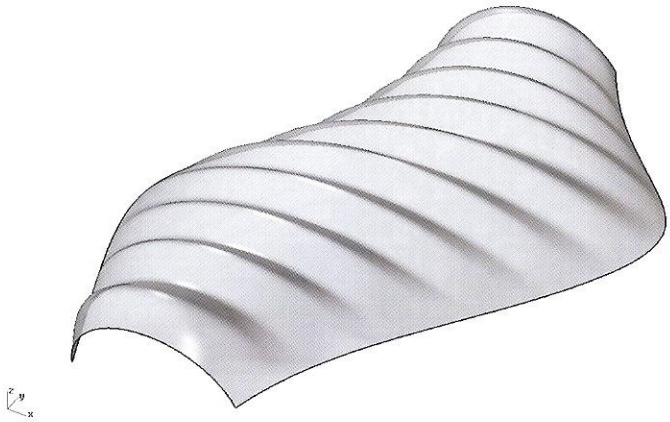


2.2.3 Repeat Data

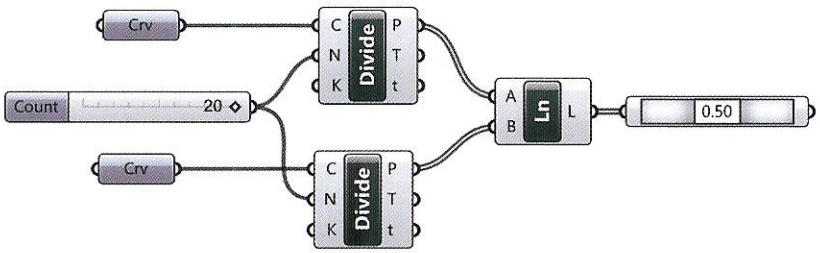
Another useful component is ***Repeat Data*** (Sets > Sequence) which extends a numeric sequence to a specified length by repeating the input data. The component's D-input collects a defined set of numbers, and the L-input defines the length of the output list. For example, if the numbers (2,6,1) are merged into a single list connected to the D-input of *Repeat Data*, the resulting list is (2, 6, 1, 2, 6) if the L-input is set to 5.

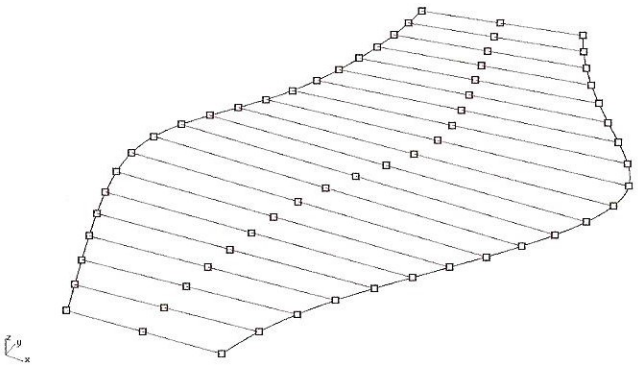


The output of the *Repeat Data* component can be used to define sequential geometry. To visualize how *Repeat Data* can manipulate geometry, a simple algorithm will be used to get the following surface.

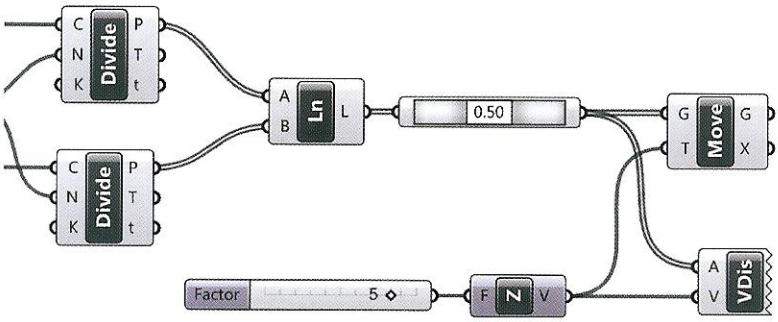


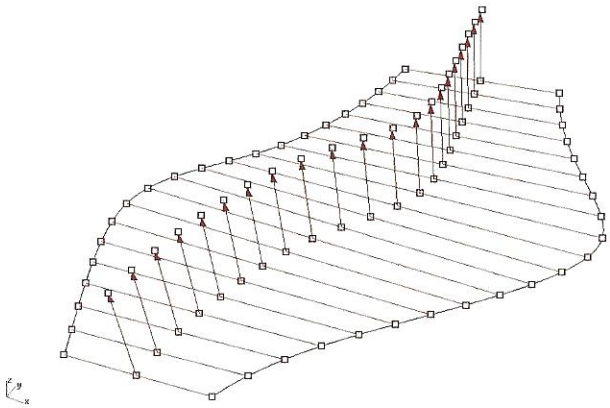
Two curves set from Rhino are divided using the *Divide Curve* component, and lines are connected to corresponding points in the lists using the *Line* component. The lines are evaluated using *Point on Curve* to calculate the midpoint of each line.



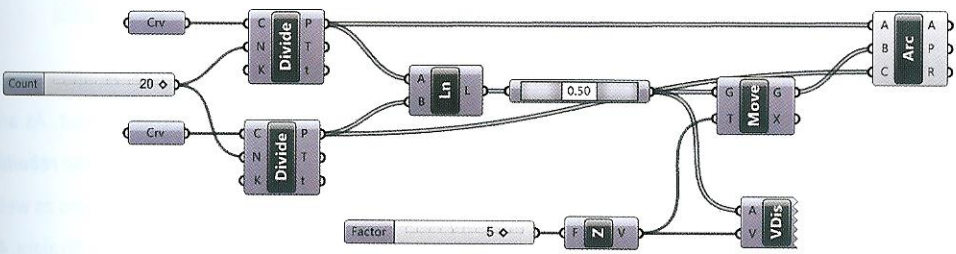


Each midpoint is moved by a translation z-vector (visualized by the *Vector Display* component), using the *Move* component.

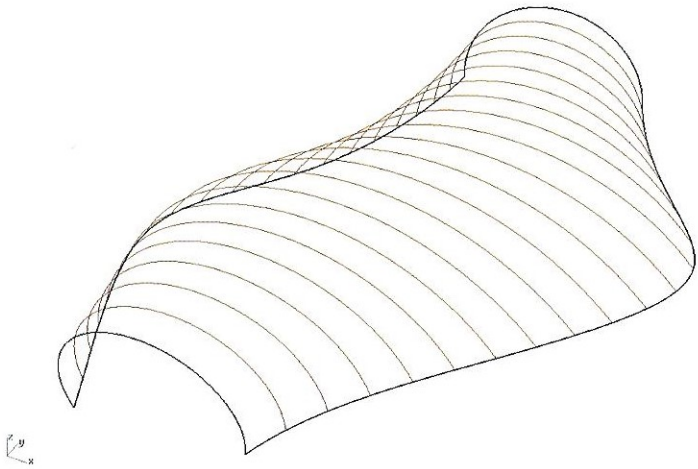




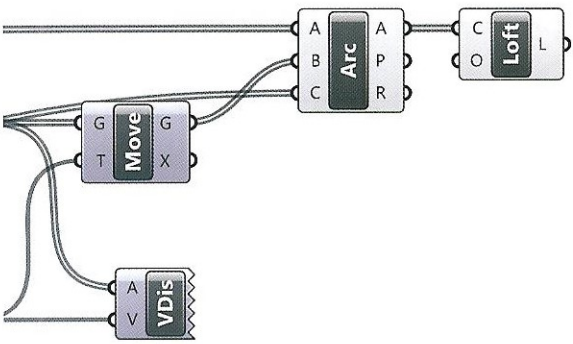
Then, the *Divide Curve* output (P) of *Curve 01*, the translated points, and the *Divide Curve* output (P) of *Curve 02*, are connected into the A, B, C slots of the component *Arc 3Pt* (Curve > Primitive) respectively.



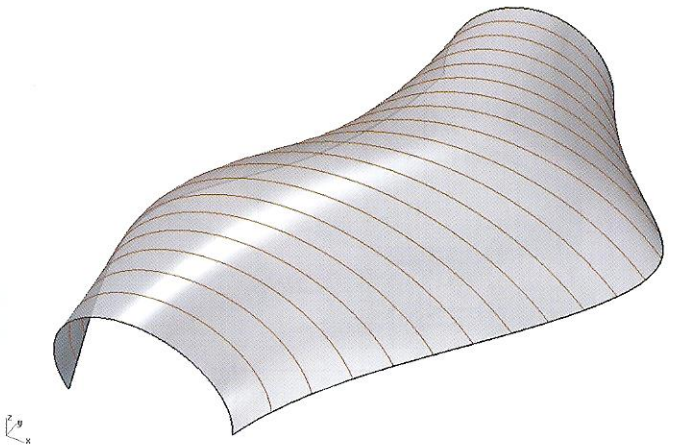
The following image displays the final set of arcs. We turned off the preview of all the components except *Arc 3Pt* (you can also see the two curves drawn in Rhino).



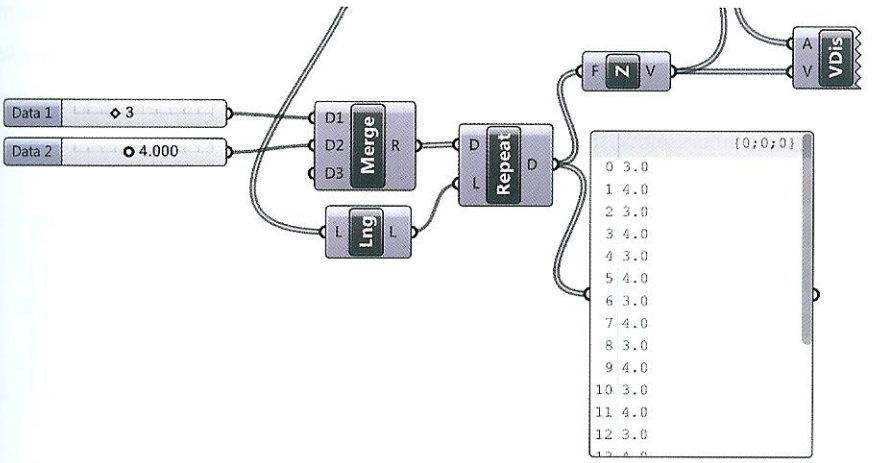
Lastly, the A-output of the *Arc 3Pt* component is connected to the C-input of the component ***Loft*** (Surface > Freeform) to generate a lofted surface, defined as a surface through a set of section curves.



The O-input of *Loft* is used to modify surface options. Options can be locally set or, in alternative, modified by the ***Loft Options*** component (Surface > Freeform) connected to the O-input. As an instance, when the number of section curves is not enough to create a "smooth" surface, the ***rebuild*** option (Rbd-input) must be set. *Loft Options* also allows to close the loft, to adjust the seams as well as select the loft type, which is defined by an integer (0 = Normal, 1 = Loose, 2 = Tight, 3 = Straight, 4 = Developable, 5 = Uniform).



The algorithm is now complete. To modify the surface, the lofted arcs can be manipulated using the *Repeat Data* component. Modifying an algorithm by changing the inputs is a common working method in Grasshopper. For example, if the *Number Slider* connected to the *Unit Z vector* component is replaced with a list of repeated numbers (with the *List Length* set equal to the number of items of the *Line* component) the lofted geometry will be updated, and express the change to the z-translation in its surface shape. The surface can be varied through infinite changes of the sequence input into the *Unit Z* component.



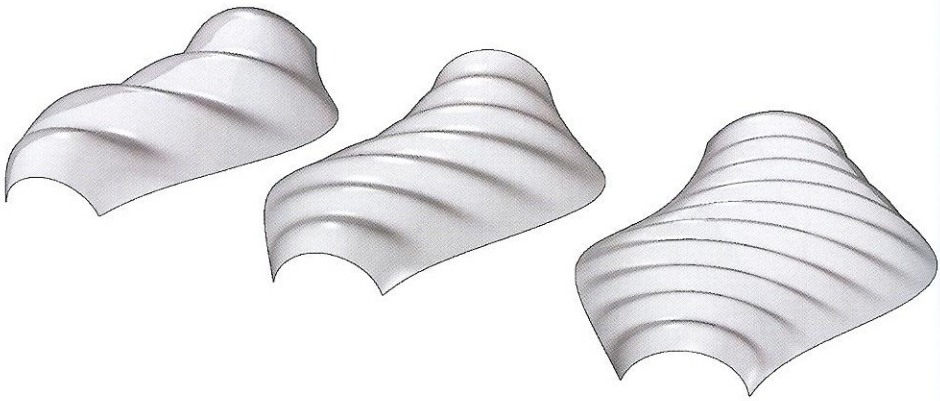
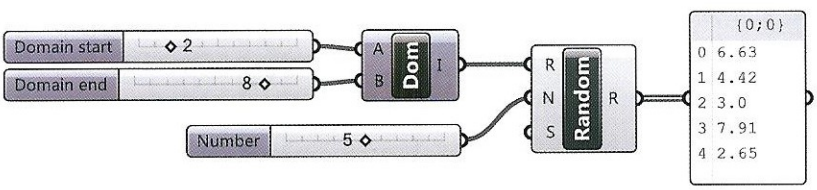


FIGURE 2.2
Shapes informed by data.

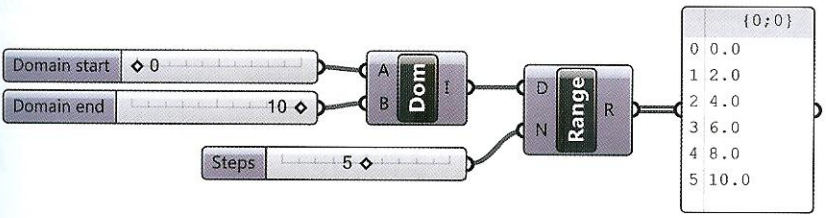
2.2.4 Random / Construct Domain

The component *Random* (Sets > Sequence) generates a list of (N) random numbers within a defined numeric domain (R). The S-input specifies the seed value of the component, if the seed value is changed and all other inputs remain the same a new list of random values will be generated. By default, the *Random* component generates real numbers. To specify integers right-click on the *Random* component and select the *Integer Numbers* option from the context menu and a black *Integers* flag will appear at the bottom of the component. The component *Construct Domain* (Maths > Domain) is used to define a numeric domain between two numeric extremes. For example, a domain defined between a minimum value of 2 and a maximum value of 8 is connected to a *Random* component specifying a list with five items is to be generated. The output of the *Random* component is five random numbers within the numeric domain.



2.2.5 Range

The component *Range* (Sets > Sequence) defines a range of numbers within a numeric domain (D) with (N) subdivisions. In other words, numeric domain is divided into equal parts. When the domain is divided into N parts N+1 values are defined.

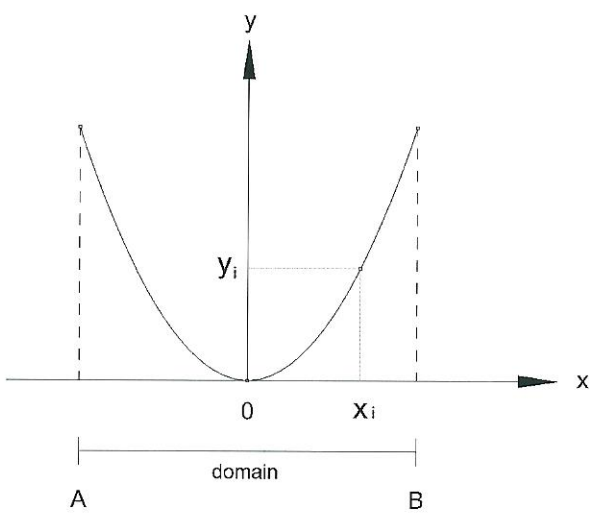


2.3 Mathematical Functions

The component *Evaluate F(X)* (Maths > Script) can be used to define numerical sequences that follow **mathematical functions**.

2.3.1 Functions of one variable

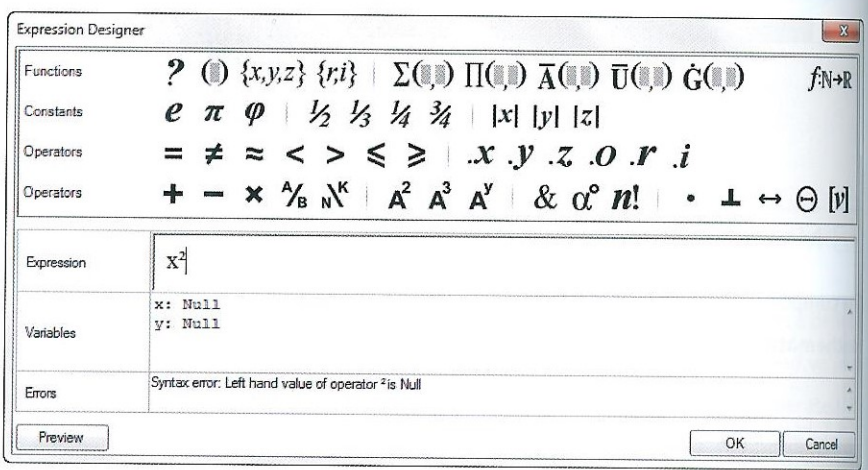
A function of one variable has as a **domain** of values in the x axis and a **codomain** of values in the y axis, meaning for each input value (xi) the function provides an output value (yi).



To build an algorithm with a mathematical function:

1. **Defining a function**

   The *Evaluate F(X)* component, can define a function by double-clicking on the node to display the *Expression Designer*. For instance, a parabola can be defined by the *explicit equation* $y=x^2$, so x^2 would be typed in the *Expression Designer*. The square operator can be found within the *Operators* library and a complete list of functions and their signatures are available by clicking the *f:N>R* button (at the upper right corner).



   Alternatively, a function can be defined by entering an equation in a *Panel* and connecting its output to the F-input slot of the *Evaluate* component.

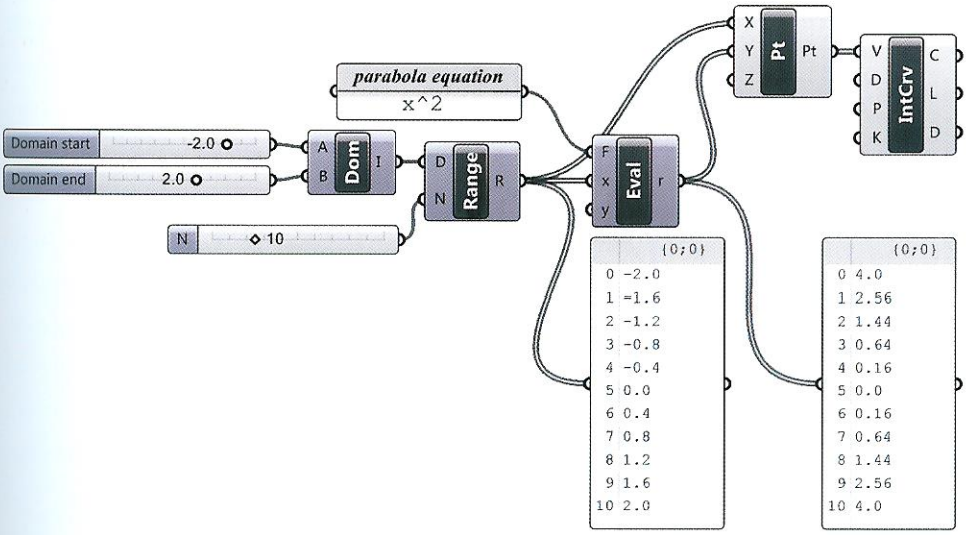
2. **Defining a domain**

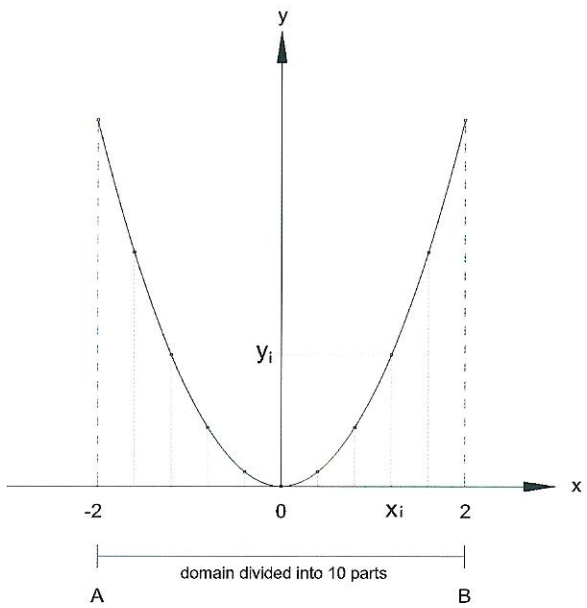
   In order for a function to return a result, the function requires numerical values as input. The component *Construct Domain* in conjunction with the component *Range* can generate numerical output. For example, if a domain [-2, 2] is connected to a range component with N set to 10, 11 values will be generated which are equally divided with respect to the domain. The output numerical values can be used as input for the *Evaluate F(X)* component.

The r-output of the *Evaluate* component is the codomain of the function. Examining the output of the *Evaluate* component, the list of numbers decreases until zero and then increases again after zero; the expected results of the function x^2.

Each point of a planar mathematical curve has two coordinates – x and y – where x is a value from the domain and y the output codomain. If the outputs of *Range* component (R) and the *Evaluate*

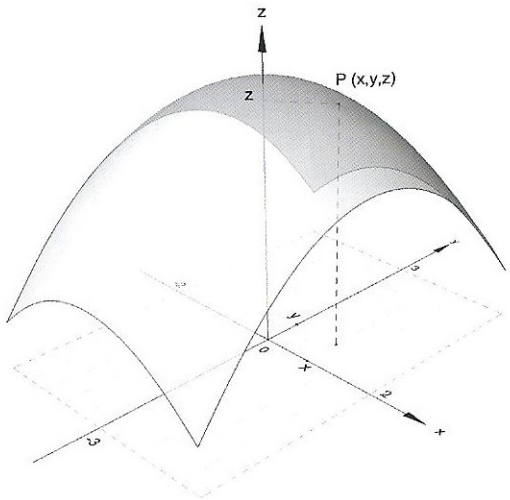
component (r) are connected to the *Construct Point* components (X) and (Y) slots respectively a set of points which define the graphical appearance of the function x^2 will be generated. A curve can be drawn through these points using the component *Interpolate Curve* (Curve > Spline).





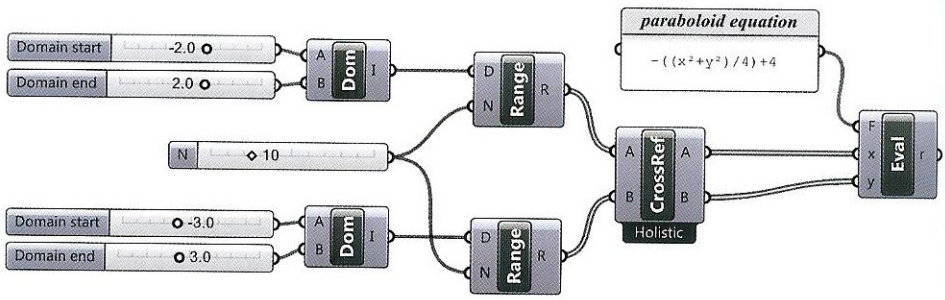
2.3.2 Functions of two variables

A function of two variables has a domain composed of a subset of points belonging to the XY-plane and a codomain of values in the z axis, meaning for each input value (xi,yi) the function provides an output value (zi).

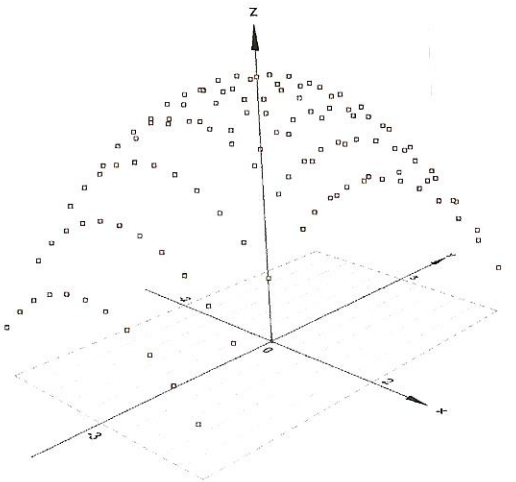


To generate a surface from a function of two variables: first, define an equation and second define the domain. Then construct points with respective (xi,yi,zi) output and interpolate a surface through the points. Functions of two variables have a bi-dimensional domain, meaning input values for (xi) and (yi) are required to be defined.

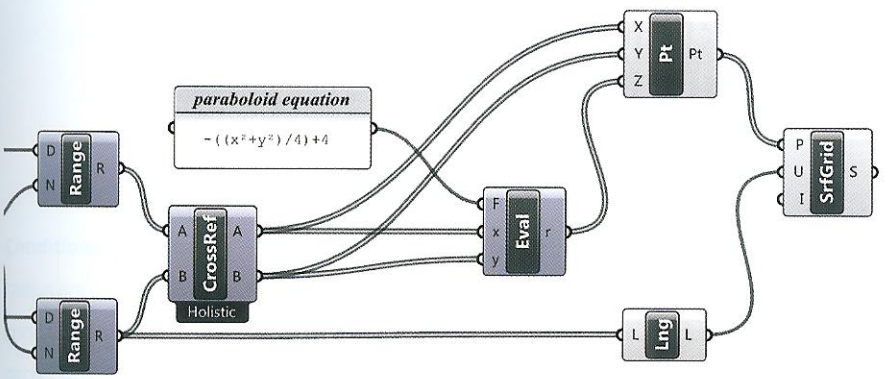
For example, a paraboloid can be defined by the equation: $z=-((x^2+y^2)/4)+4$. To generate a grid of points to define a paraboloid surface, the (xi) and (yi) input lists are required to be "cross referenced" before evaluating the function.



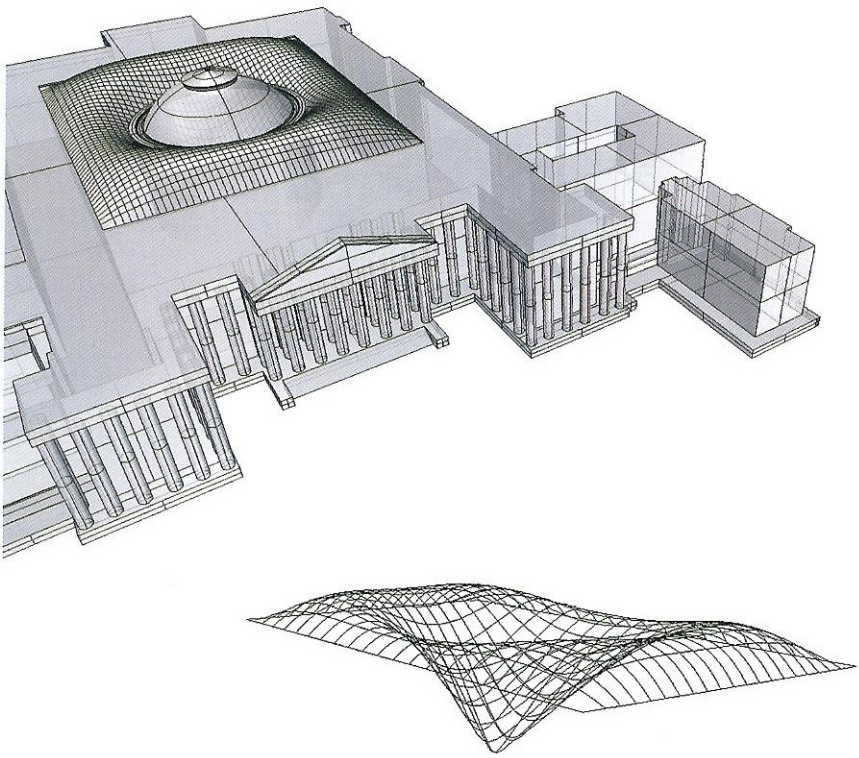
The *Evaluate* component generates a (zi) coordinate for each (xi,yi) input pair based upon the specified equation. The *Construct Point* component combines the data from the (xi), (yi), and (zi) output lists into a point (xi, yi, zi).



The component *SrfGrid* (Surface > Freeform) creates a surface from a grid of points (see 3.6). To generate the paraboloid surface the Pt-output of *Construct Point* is connected to to the P-input of the *SrfGrid* component, and the *List length* component is used to specify the number of points in the x or y direction to satisfy the (U) input of the *SrfGrid* component.



The following image illustrates how is possible to get the actual configuration of the **British Museum Great Court Roof** referring to the equations published by Chris JK Williams[7]. The roof's shape is generated by a sum of three equations of two variables.



$$z = \frac{h\left(1-\frac{x}{b}\right)\left(1+\frac{x}{b}\right)\left(1-\frac{y}{c}\right)\left(1+\frac{y}{d}\right)}{\left(1-\frac{ax}{rb}\right)\left(1+\frac{ax}{rb}\right)\left(1-\frac{ay}{rc}\right)\left(1+\frac{ay}{rd}\right)}$$

$$z = H\left(1-\frac{x}{b}\right)\left(1+\frac{x}{b}\right)\left(1-\frac{y}{c}\right)\left(1+\frac{y}{d}\right)\left(\frac{r}{a}-1\right)$$

$$z = \frac{\eta\left(\frac{r}{a}-1\right)}{\left(\begin{array}{l}\frac{\sqrt{(b-x)^2+(c-y)^2}}{(b-x)(c-y)} + \frac{\sqrt{(b+x)^2+(c-y)^2}}{(b+x)(c-y)} \\ + \frac{\sqrt{(b-x)^2+(d+y)^2}}{(b-x)(d+y)} + \frac{\sqrt{(b+x)^2+(d+y)^2}}{(b+x)(d+y)}\end{array}\right)}$$

NOTE 7

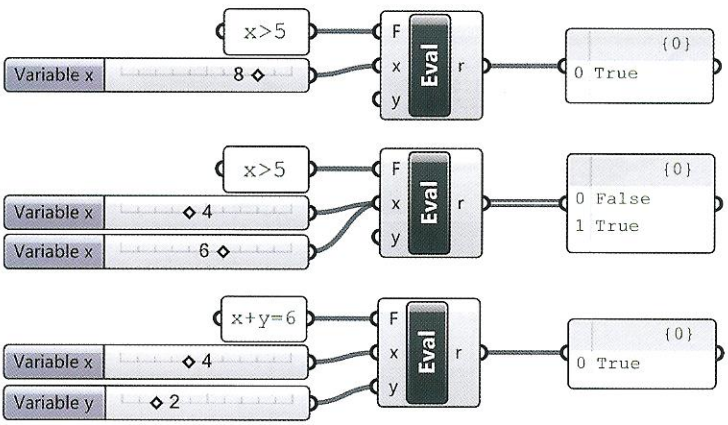
C.J.K. Williams, *The analytic and numerical definition of the geometry of the British Museum Great Court Roof*, (Mathematics & design, 2001), 434–440.

2.4 Conditions

The ***Evaluate*** component is not only useful for creating curves and surfaces from specific equations, but it also allows us to define **conditions** in Grasshopper. Often it's crucial that an algorithm performs different operations if a **condition** is met or not. A condition can be set in the F-input of *Evaluate* (in alternative, you can type the condition inside the *Expression Designer* editor) and it may include an arbitrary number of variables (x, y, ...).

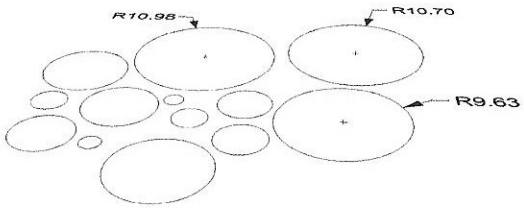
2.4.1 Logical operators/Boolean values

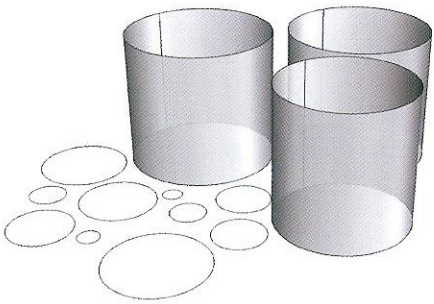
Conditions are build through variables and logical (comparison) operators such as: *Equality* (=), *Smaller Than* (<), *Larger Than* which can be found within the *Operators* panel of the *Maths* tab. Other operators (e.g. *inequality, smaller than or equal*) can be found within the *Expression Designer* editor. For example, the *Evaluate* component can be used to test if a set of numerical variables meets a defined condition. *Evaluate* returns **Boolean values**[8]: **True** if the condition is met, and **False** if the condition is not met.

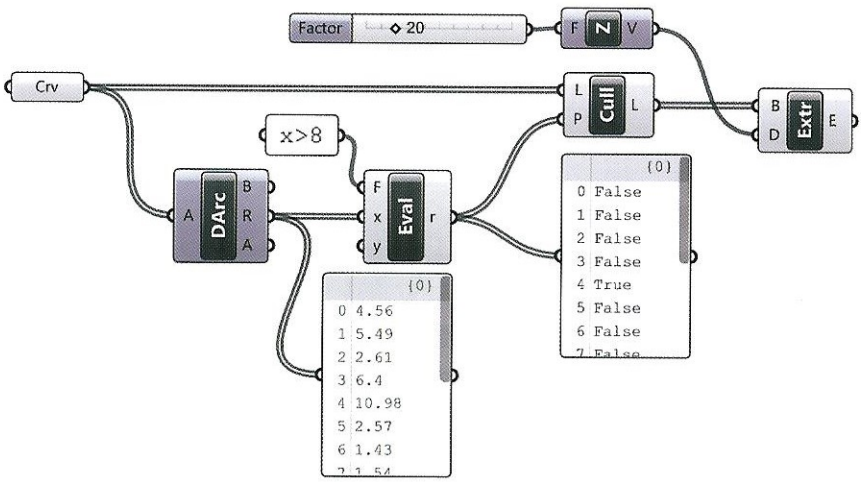


Conditional statements can be used to define *Cull Patterns*. For example, a set of circles whose radius are calculated using the component *Deconstruct Arc/DArc* (Curve > Analysis) are compared to the conditional statement R > 8, circles which satisfy this condition are extruded 20 units in the z direction, circles which do not satisfy this condition are culled.

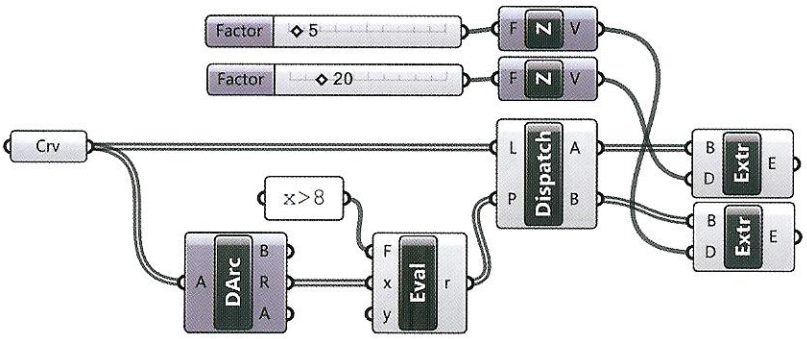
NOTE 8
Named after the English mathematician George Boole.

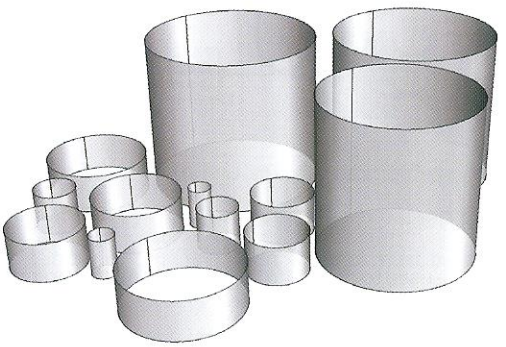




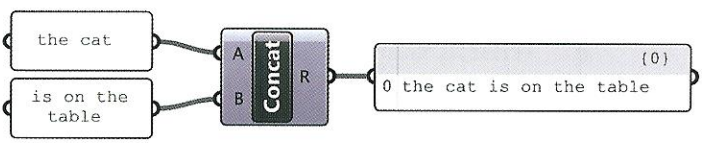


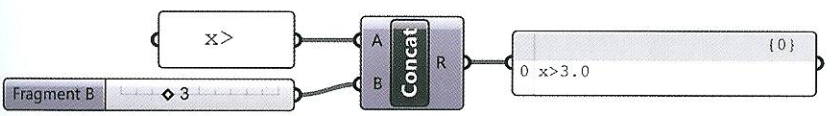
Alternatively, the component ***Dispatch*** (Sets > List) can be used to filter data with a corresponding Boolean values list. The *Dispatch* component returns two lists: **list A** contains data which returned True i.e. met the conditional statement, and a **list B** contains data which returned False i.e. data that did not meet the conditional statement. Referring to the previous example, *Dispatch* compares the radius of circles to the conditional statement (R>8) and sorts the circles so that circles which return True are extruded 20 units in the z direction and circles which return False are extruded 5 units in the z direction.



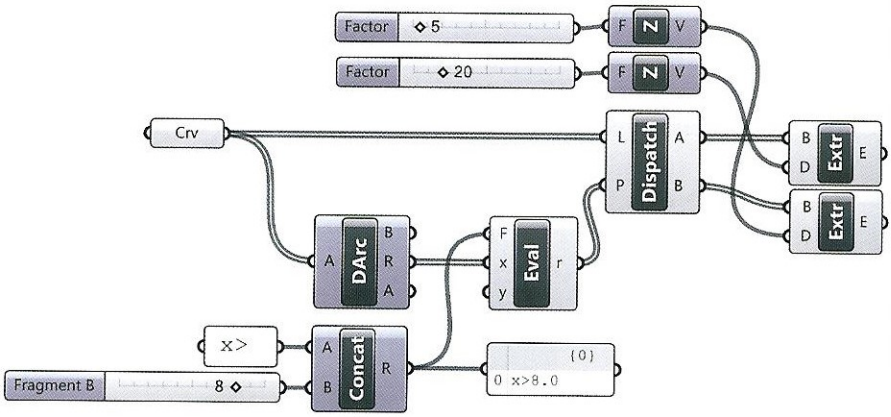


The component ***Concatenate*** (Sets > Text), can be used to join two or multiple fragments of text. For instance the statement "the cat is on the table" can be created by concatenating the panels "the cat" and "is on the table." *Concatenate* can be used to write a "parametric" condition by concatenating the text "x>" typed inside a *Panel* (do not press ENTER on your keyboard after typing "x>" but confirm with OK within the *Panel Properties* window) with a *Number Slider* or a generic number input.



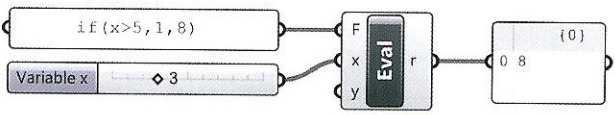


The *Concatenated* output can then be used as the F-input of the *Evaluate* component.

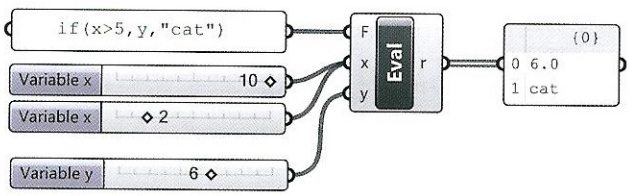


2.4.2 Conditional: if / then

The *Evaluate* component can be used to define if/then statements, meaning the conditional statement **if (test, A, B)** returns A if test is True, B if test is False.

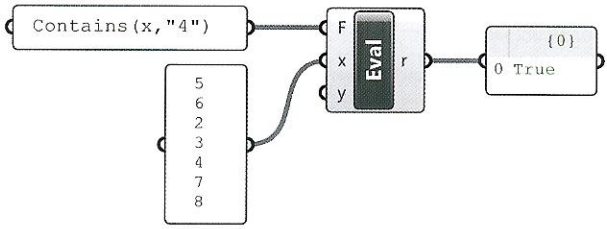


Inputs A and B do not have to be numbers. For instance, the text "cat" can be returned when the statement is False.

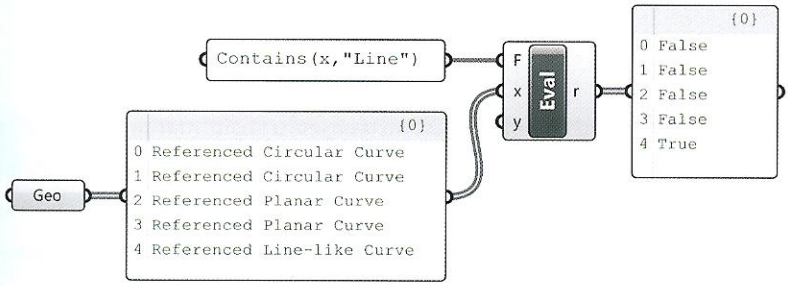


2.4.3 Other operators: Contains

The *Contains* operator **Contains (s, "p")** tests whether the string **p** is in the list of data **s**. The *Evaluate* component tests the operator and returns a Boolean value. The following example tests if "4" occurs within the list of numbers connected to the x-input, and yields True as a Boolean value.

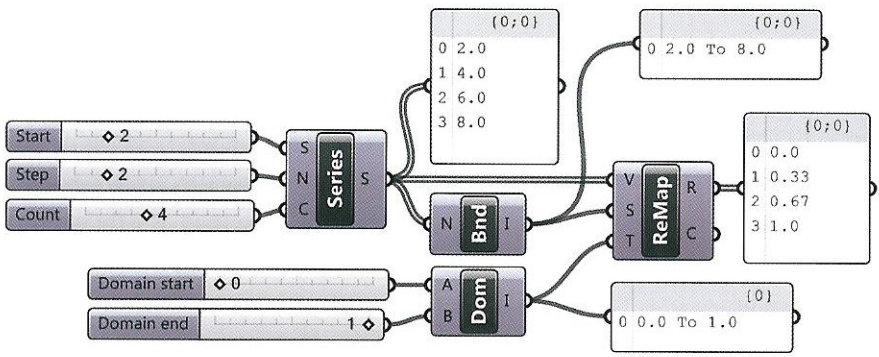


The operator *Contains* can also be used to find specific geometries within a list. For example, to test if a list of geometries (imported by a *Geometry* container) contains a line, the operator Contains (x,"line") can be set in the F-input of *Evaluate*. Then, a *Panel* must be used as a "bridge" between *Geometry* and the x-input of *Evaluate*. Panel's descriptions containing the text "Line" will result in a True statement and any other text will result in a False statement.



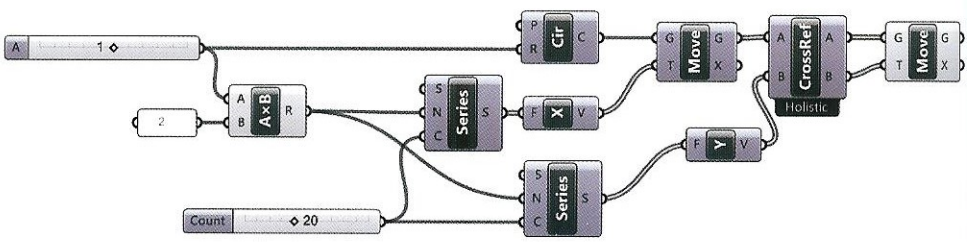
2.5 Remapping numbers / Attractors

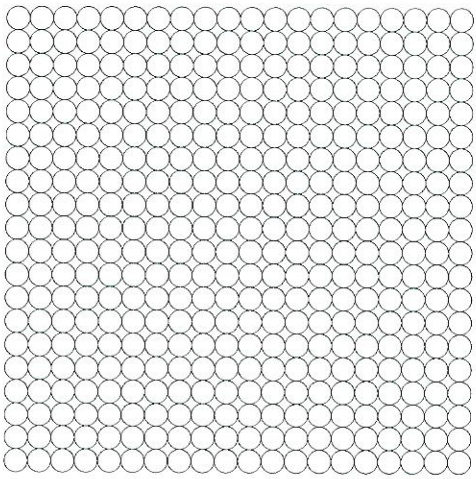
The component ***Remap Numbers*** (Maths > Domain) evaluates a list of numbers ranging from A to B, and resizes them proportionally to a new numeric domain A' to B'. The *Remap* component requires a list to remap (V), a source domain (S), and a target domain (T). The source domain of the numerical sequence can be found using the component *Bounds* (Maths > Domain), while the target domain is specified using the *Construct Domain* component. For example, the list of numbers (2, 4, 6, 8) whose source domain is [2,8] can be remapped to a new domain [0,1], yielding the list of values (0.0, 0.33, 0.67, 1.0).



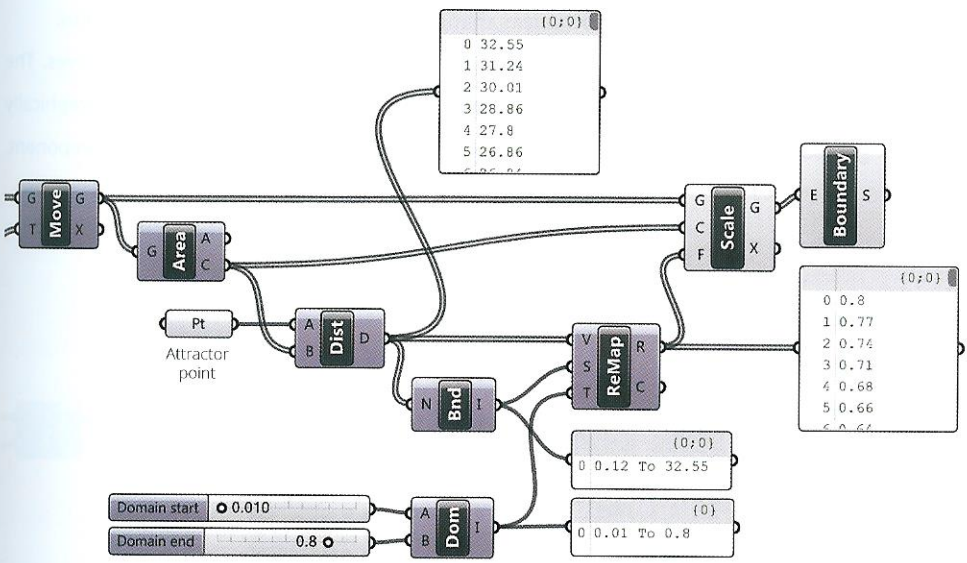
2.5.1 Attractors

The *Remap* component is used to perform **distance-based transformations**, utilizing "attractors." An **attractor** is a geometric entity: a point, a curve or another element, used to modify the geometry around it within defined limits. The impact an attractor has on other geometry depends on the distance between the defined attractor and the object it is manipulating. To visualize how a point attractor can scale a grid of circles a simple algorithm will be used as an example.



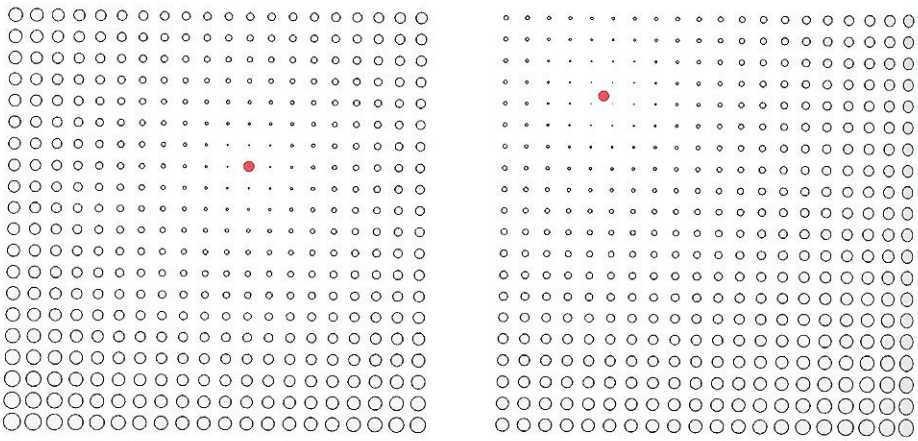


The component ***Circle*** (Primitive > Curve) generates a circle with the center at point {0,0,0} and a radius of 1. The defined circle is two dimensionally arrayed using *Move* and *Series* in conjunction, creating a grid of circles.



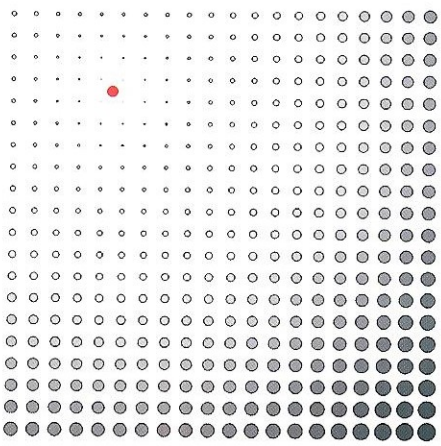
A point set from Rhino is used as the attractor mechanism by measuring the distance between the attractor-point and the centroid of each circle returning a list of numeric values. To scale the circles a scaling factor between 0 and 1 is required. The *Remap* component is used to remap the actual

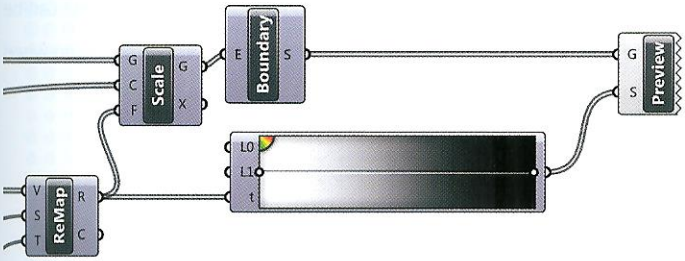
distances (between the attractor point and each centroid) proportionally to a new domain between (0,1]. A scaling factor of zero would yield null results; consequently a value close to zero is used.



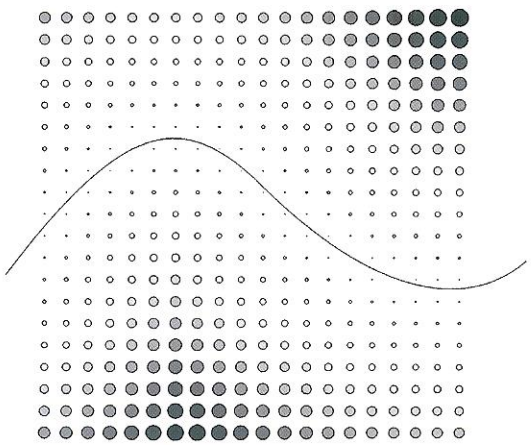
The component ***Scale*** (Transform > Euclidean), scales geometry (G) according to a center of scaling (C) and a scale-factor (F). The remapped domain (0,1] is used as the scaling factors. As the attractor points position moves, the respective distances change and the scaling associatively updates.

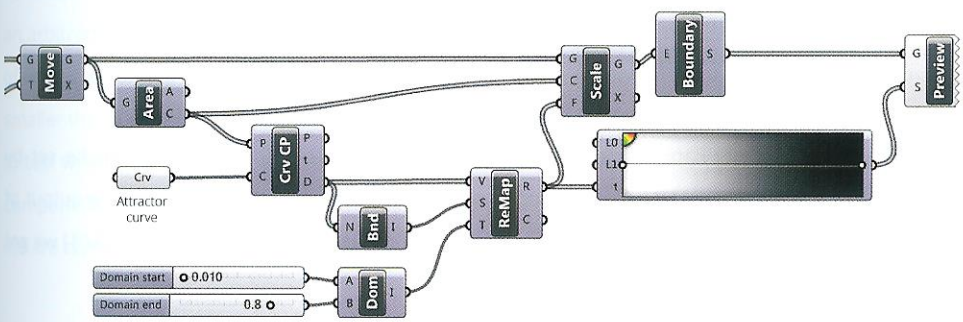
The component ***Boundary*** (Surface > Freeform) creates planar surfaces from closed curves. The component ***Gradient*** (Params > Input) can be used to display the respective scaling factors graphically by a color gradation. To change the *Gradient* component's color preset: right-click on the component, select preset from the sub-menu, and specify a color gradient.



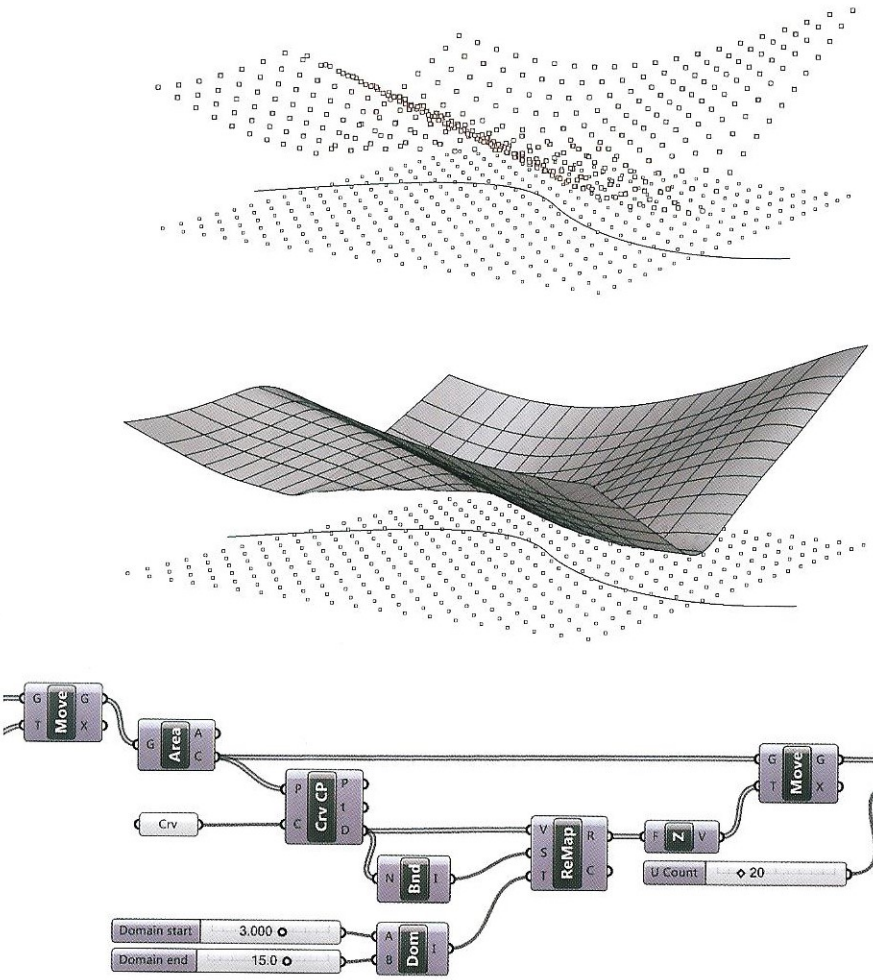


A curve can also be used as an attractor by means of the component ***Curve Closest Point*** (Curve > Analysis). The *Curve Closest Point* component measures the distance (D) between each circle's center and the closest point on the curve. The scaling factors can be visualized by using the *Gradient* component.

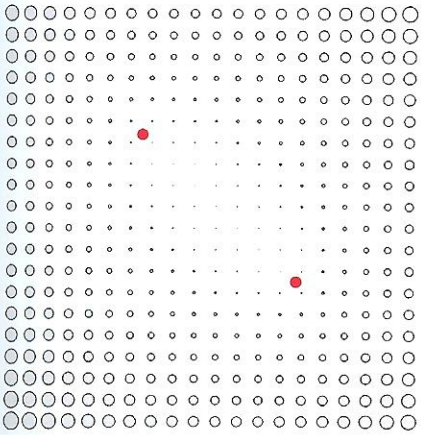




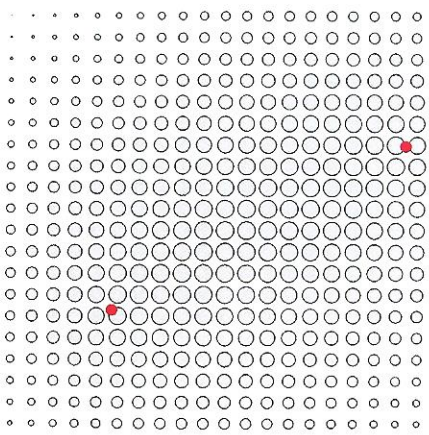
The remapped values can also be used to move the centroid points. For instance, the points can be moved in the z-direction. Changing the remapped domain to [3,15] emphasis the translation when scalar multiplication is carried out within the *Unit Z* component.



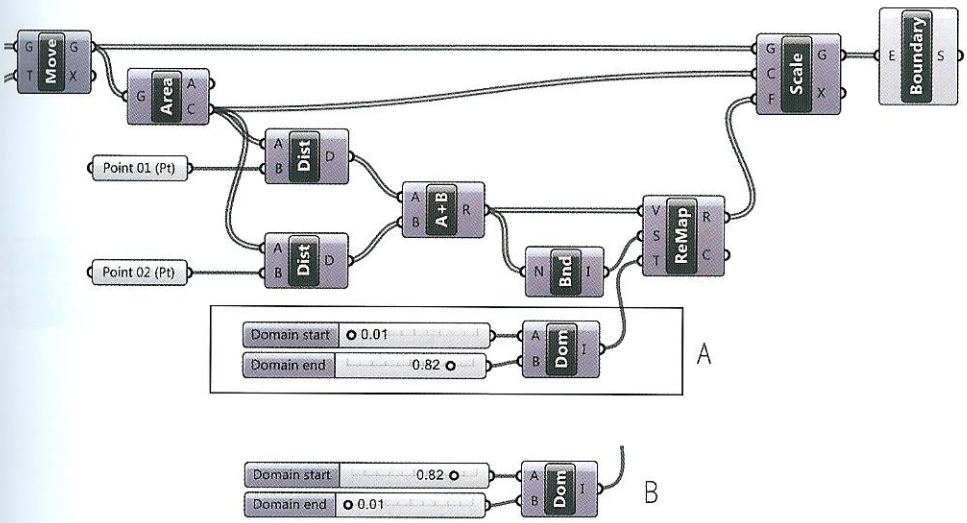
Multiple attractor points can also be defined. For example, two attractor-points can be set by summing their relative distances to each circles centroid. The list to remap (V) will be the output of the component *Addition* (Maths > Operators). If we invert the domain's extremes [0.82, 0.01] we get an opposite scale effect (image B).



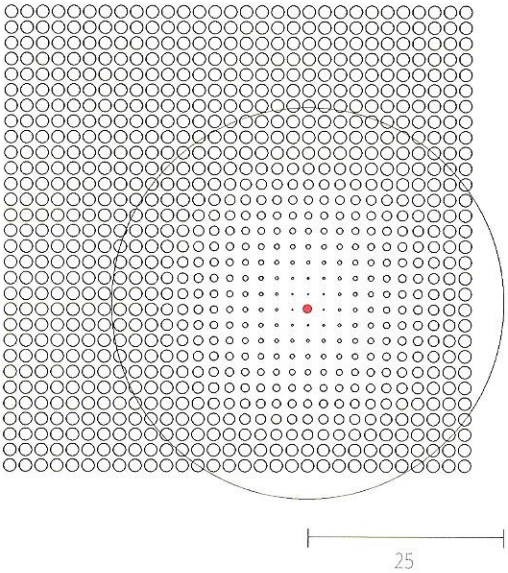
A

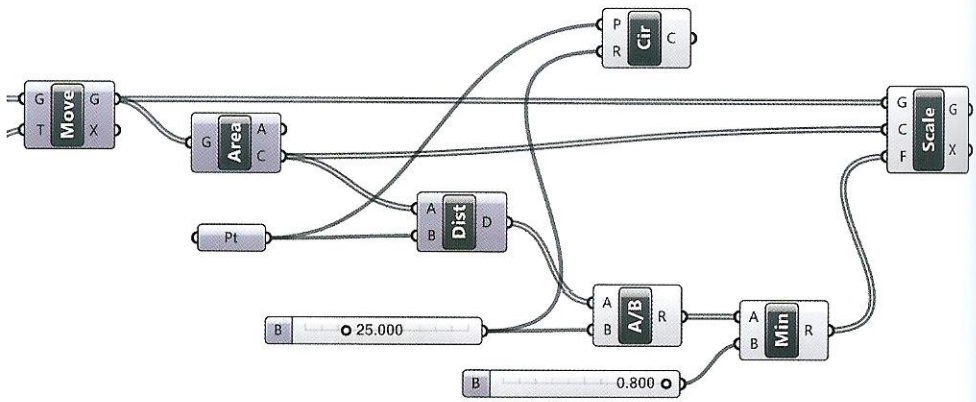


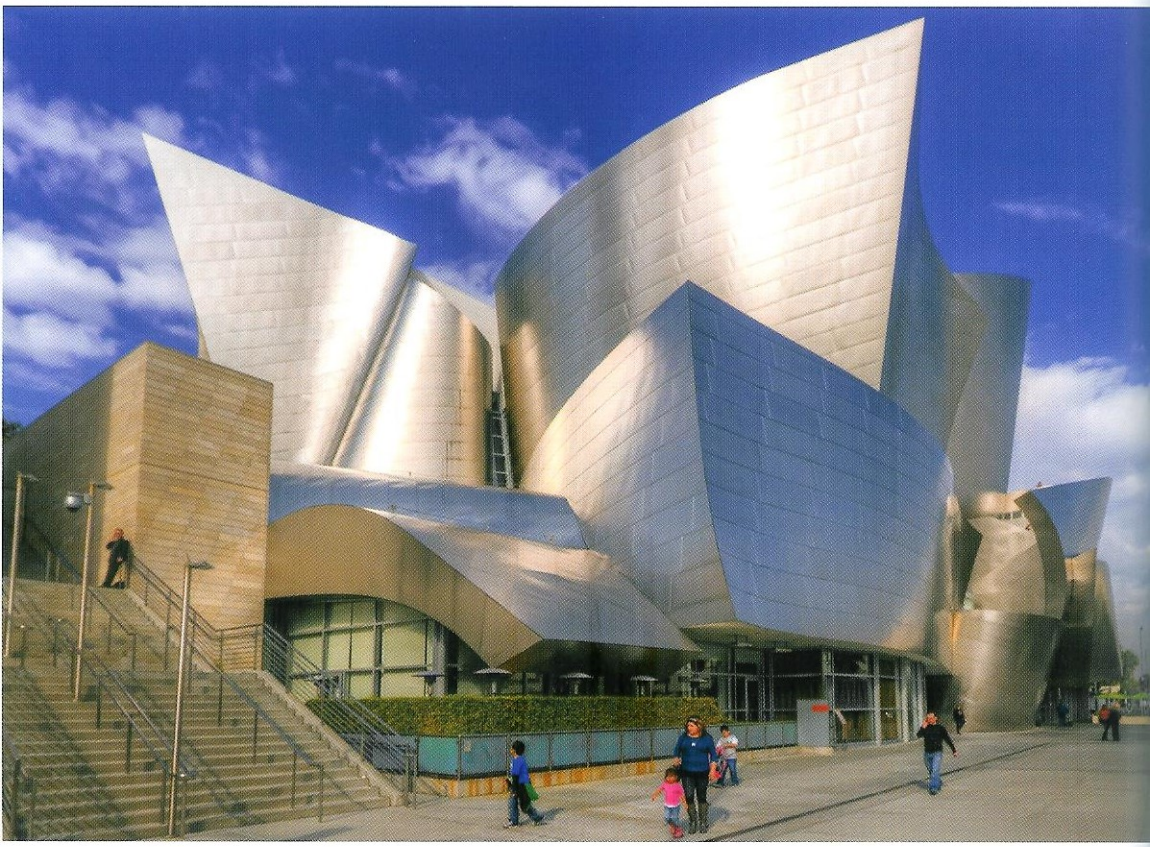
B



To **limit** the effect of an attractor the distance between the point and the centroid can be divided by an arbitrary number, which becomes the *operating range* of the attractor. The division returns a list of values which are connected to the F-input of the *Scale* component. The scaling factors must be smaller than 1 to reduce the original geometry. The component ***Minimum*** (Maths > Util) can be used to test whether a list of numbers are larger than a comparison number (B). Values that satisfy the component are replaced with the minimum value (B). For instance, distances that are greater than 0.8 are replaced with 0.8 as a scale factor.


25


Move
Area
Pt
Dist
Cir
Scale
A/B
Min
25.000
0.800



Frank Owen Gehry, Walt Disney Concert Hall building.
Picture by Pedro Szekely.