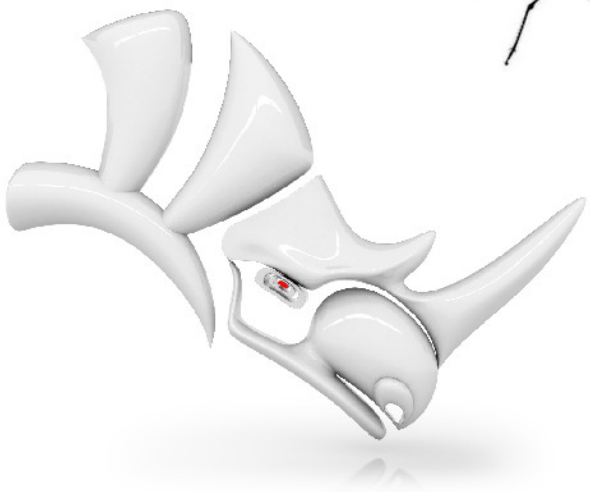
PanelingTools

for **Grasshopper**



PanelingTools helps designers create paneling solutions from concept to fabrication.

Development of the PanelingTools plug-in for Rhino started in 2008. PanelingTools facilitates conceptual and detailed design of paneling patterns using NURBS and mesh geometry. PanelingTools is closely integrated with the Rhinoceros environment using standard Rhino geometry. PanelingTools also extends RhinoScript, Python for completely customized paneling and Grasshopper for parametric modeling.

I hope using PanelingTools will be a fun and useful experience. I am always happy to hear from you and learn how you are using PanelingTools and how to improve it. If you have any questions or suggestions to further its development, feel free to contact me.

Rajaa Issa
Robert McNeel & Associates
Rhinoceros Development team
rajaa@mcneel.com

Technical Support

Suggestions, bug reports, and comments are very much encouraged. Please share your stories, examples, and experiences with us. Post questions to our discussion forum http://www.grasshopper3d.com/group/panelingtools or e-mail us directly.

Visit http://www.rhino3D.com/support.htm for more details, or feel free to contact the developer, Rajaa Issa.

Getting Started with PanelingTools

PanelingTools for Grasshopper is under active development. New functionality is added frequently, and like any other McNeel product, your feedback is very important and continuously shapes and steers the development.

Download and Install

To download PanelingTools for Rhino and Grasshopper, go to http://v5.rhino3D.com/group/panelingtools, and click on the **Download** button. All PanelingTools instructions, documentation, and discussions are available there.

The Main Menu

When you install PanelingTools, a new **PanelingTools** menu item is added to the Rhino menu bar. You can access all of the PanelingTools commands from there. Once you open Grasshopper, a new PanelingTools tab is added to the menu. It includes all PanelingTools for Grasshopper components.

Toolbars

In addition to the menu, a set of toolbars is installed for PanelingTools for Rhino.

To load the PanelingTools toolbars

1. From the **Tools** menu, click **Toolbar Layout**.
2. Under **Files**, click **PanelingTools**, and in the **Toolbars** list check **PanelingTools**.

Overview of Paneling Elements

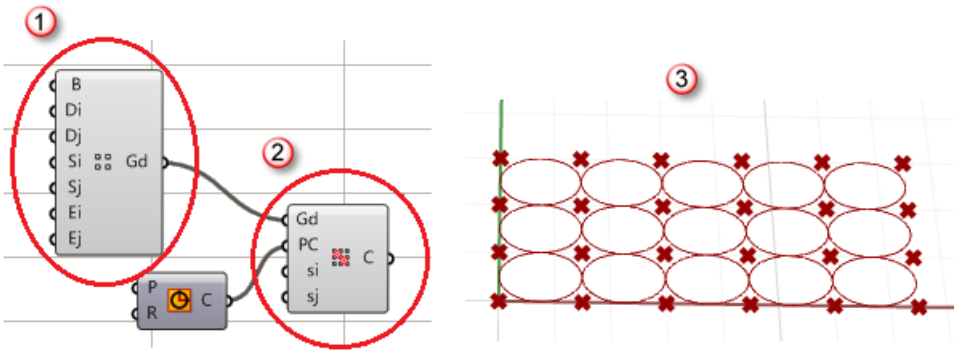
Paneling is typically done in two steps:

1. Create a grid.

   Create a rectangular paneling grid of points. Creating a paneling grid results in points that can be manipulated with any Grasshopper standard components or PT grid utility components.

2. Populate the grid with paneling elements.

   Populate a pattern or modules of curves, surfaces, and polysurfaces. Generating the paneling creates patterns and applies the patterns to a valid paneling grid of points. The resulting paneling is standard Rhino geometry in the form of curves, surfaces, or meshes. To further process panels (with the **Unroll**, **Offset**, **Pipe**, or **Fin** commands, for example) use paneling utility components and other Grasshopper components.



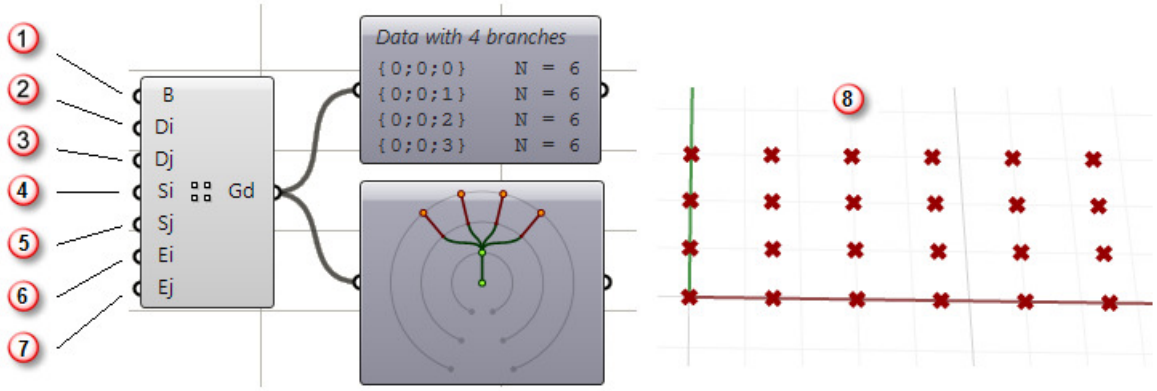
Grid component (1), paneling component (2), generated paneling (3).

The two-step process gives more flexibility and better control over the result. Normally, the initial grid is generated interactively and is a good indicator of scale. The grid can be generated using the many grid-generating commands or with scripting. The grid can be directly edited and refined before any paneling is applied. Panels can be created using user-defined patterns that connect grid points or free-form patterns.

Create Paneling Grids

A paneling grid is simply a tree structure of Rhino point objects. Each paneling point is assigned a name consisting of its row and column location in the grid.

We will explain all grid creation components later, but for example, a typical output of a single grid is arranged as follows:



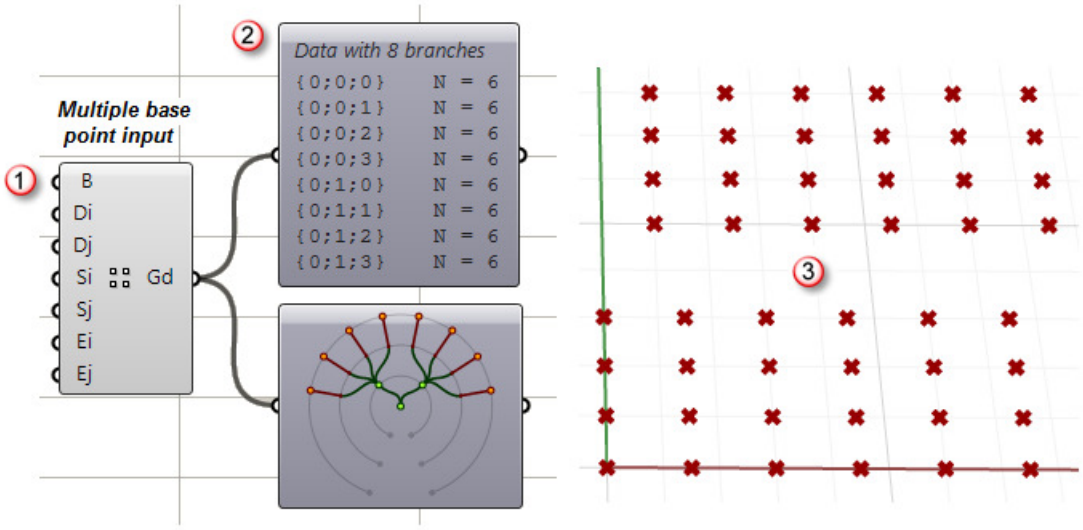
Grid base point (1), Row direction (2), column direction (3), distance between rows (4), distance between columns (5), number of rows (6), number of columns (7), output grid (8).

Paneling grids can be generated in many different ways. The following is an overview of these methods.

In the above figure, note the following:

1. Paneling grids are represented using GH tree structure.
2. Branches represent grid rows.
3. Leaves represent points in each row (branches do not have be equal in length).

One tree structure can hold a number of grids as in the following:

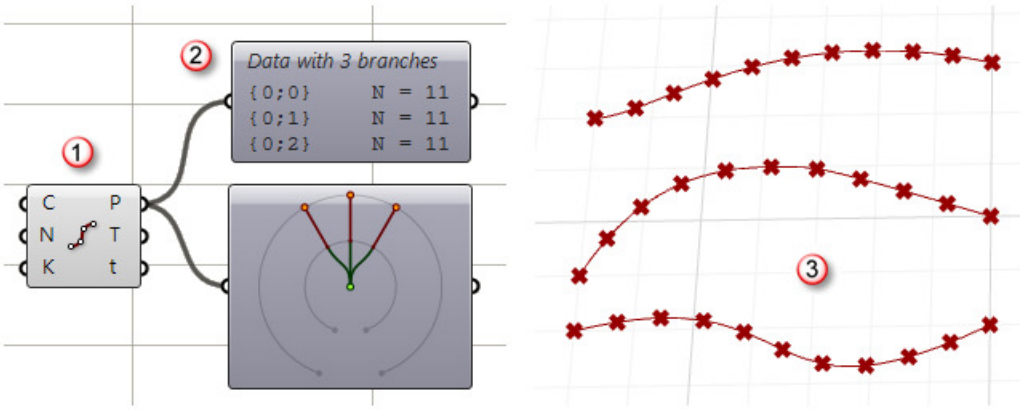


Input two base points (1), Two main branches for the two grids (2), generated grids (3).

Paneling grids can be generated in many different ways. The following is an overview of these methods.

Create Grids with Grasshopper Standard Components

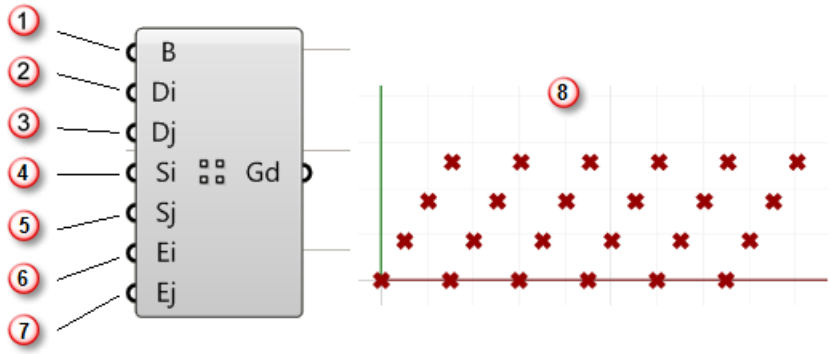
You can use grids created within GH environment as long as the output is a simple tree structure of points where branches represent rows of points. For example, DivideCrv component with multiple curve input creates such structure:



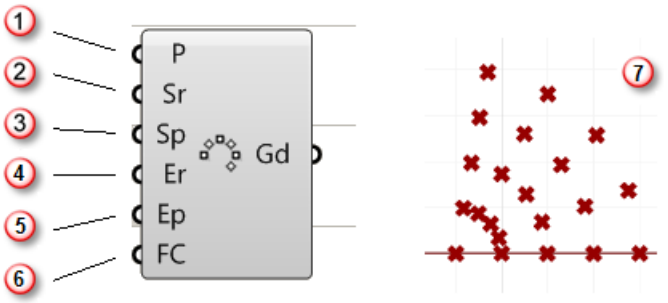
GH divide curve component (1), points stored in a tree structure (2), valid paneling grid (3).

Create Grids with PanelingTools Components

PT components make available a variety of ways to create paneling whether from scratch or through using reference geometry. The simplest components are the rectangular or polar grids as in the following:



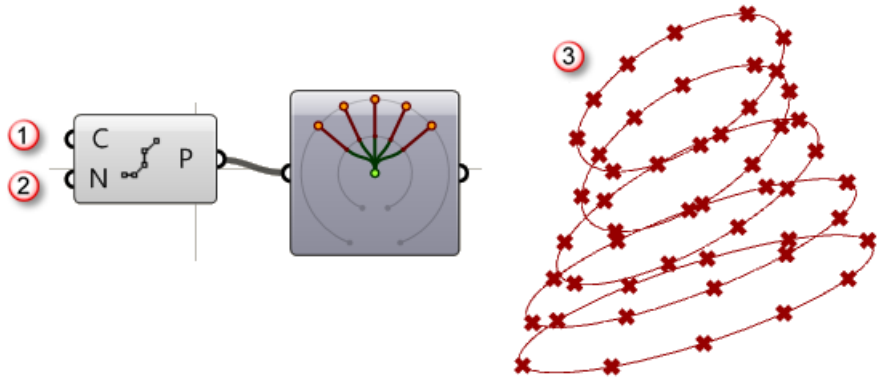
Parallel grid base point (1), Row direction (2), column direction (3), distance between rows (4), distance between columns (5), number of rows (6), number of columns (7), output grid (8).



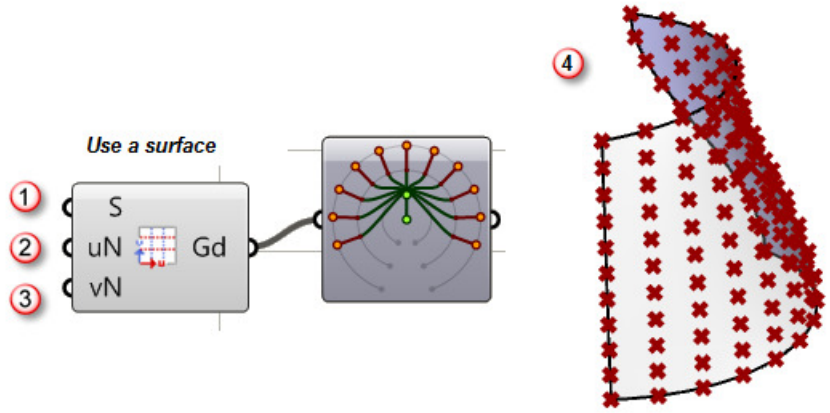
Polar grid base plane (1), distance between points in radial direction (2), angle between points in polar direction (3), number of points in radial direction (4), number of points in polar direction (5), close circle (ignore angle input) (6), output grid (7).

Create a Grid with Reference Geometry

Grids can be based on existing geometry such as curves or surfaces. For example we might have an array of curves that we would like our grid to follow. Also we might want to use a surface or a polysurface. There is a variety of grid creation components in PanelingTools that can help with that.



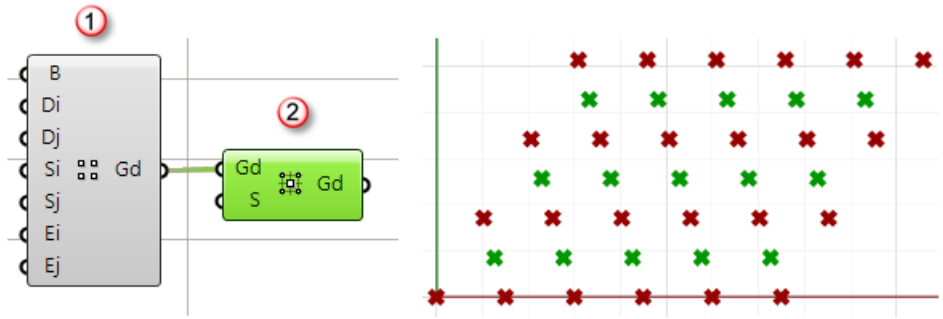
Input curves (1), number of points in each row (2), output grid (3).



Input surface (1), number of points in u direction (2),number of points in v direction (3), output grid (4).

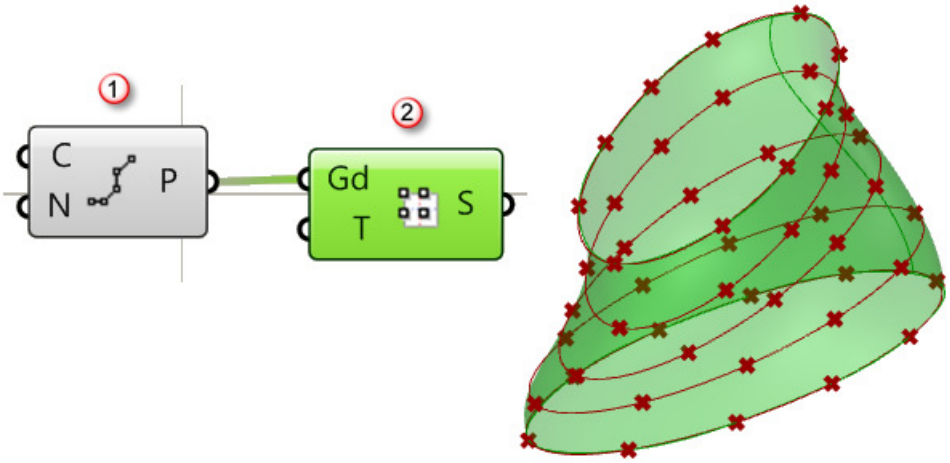
Grid Utility

PanelingTools provides an array of components that help manipulate the grid as a whole. For example, you might need to flip the direction of the grid (change base point or swap row and columns), or edit some row directions. Maybe close the grid in some direction or extend in another. Many other functions are easier to handle through grid editing components than to do manually. For example the following is a component that helps extract center grid from an existing grid.



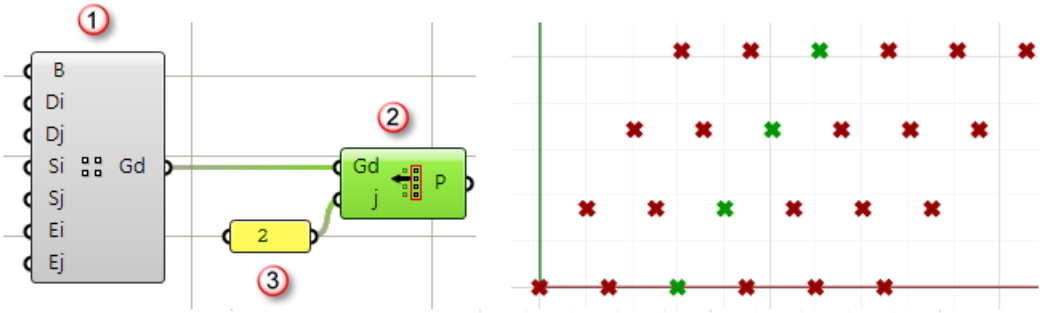
Planar grid component (1), center grid component (2).

Or you might want to create a surface that goes through grid points:



Grid from curves component (1), surface from grid component (2).

Or maybe extract certain columns or rows from a grid:

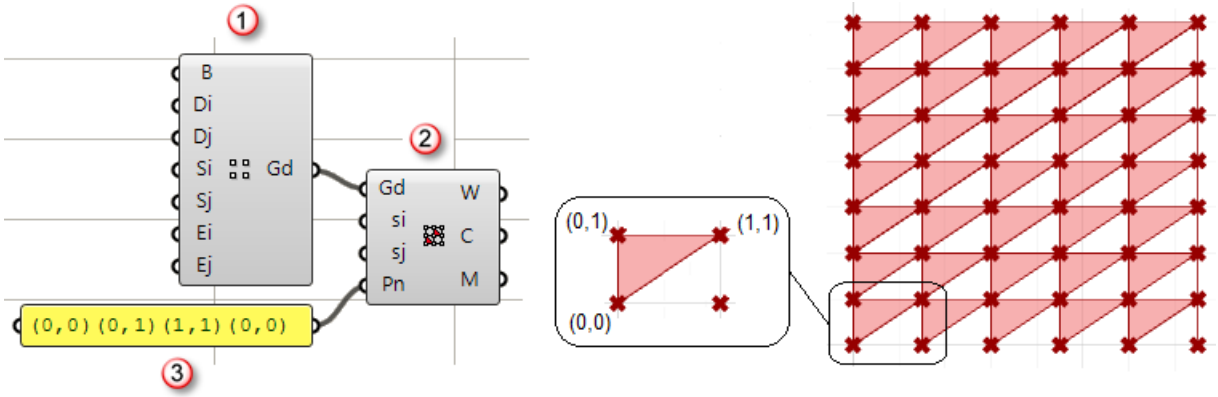


Planar grid component (1), extract grid column component (2), index of extracted column (3).

Create Paneling Patterns

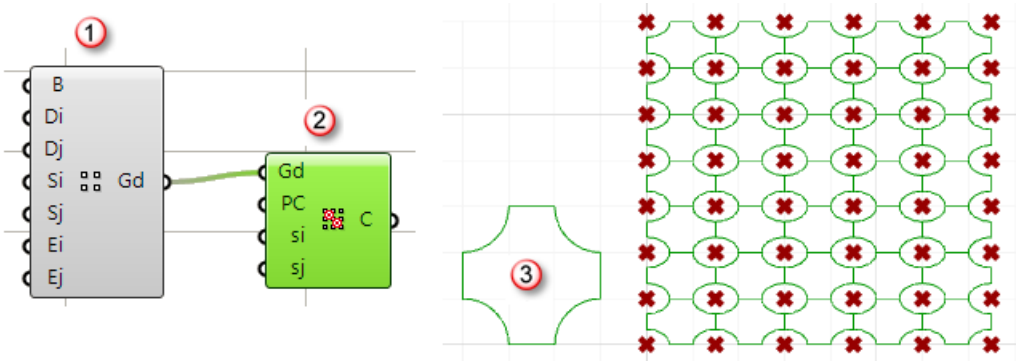
Paneling in the context of PanelingTools plugin refers to the process of mapping geometry or modules to a rectangular grid. Paneling can be either along one grid to generate 2D patterns or between 2 bounding grids to generate 3D patterns. There are three main methods to panel:

- Connect grid points to create edges, surfaces, or mesh faces of the intended pattern. This approach is the fastest and can cover a wide variety of patterns. You can also use base surfaces to pull the geometry.



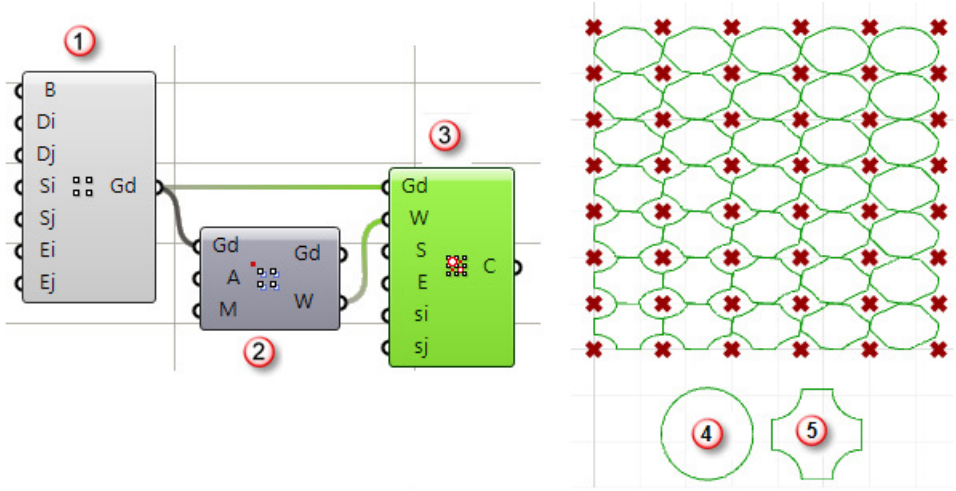
Planar grid component (1), paneling by connections component (2), unit connection string (3).

- Morph a unit module and distribute it over unit paneling grid. This approach can be more time consuming, but allows for rich development of free-form patterns that do not conform to grid points.



Planar grid component (1), morph 2D component (2), 2D module (3).

- Morph a unit module in a variable way along the grid, depending on design constraints.



Planar grid component (1), point attractor component (2), morph 2D variable component (3), start module (4), end module (5).

Attractors as a Design Element

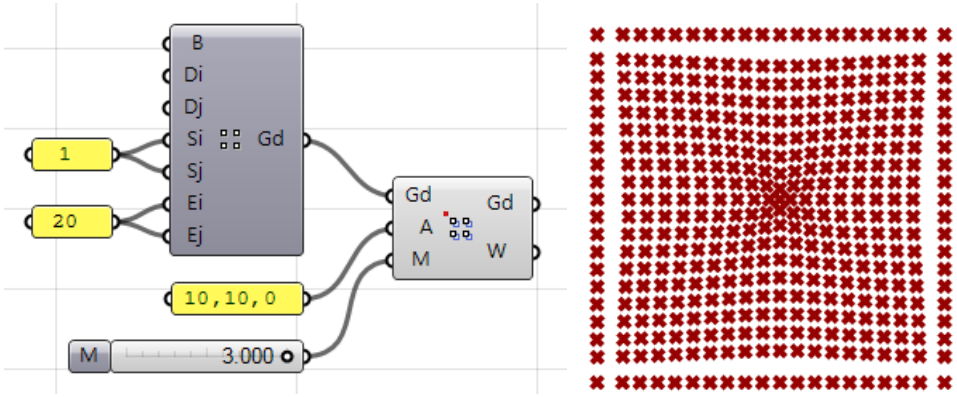
In parametric modeling, if you use an attractor (points, curves, etc.) to shuffle grids or create variable paneling, it becomes very easy to examine the effect of changing the attractor location and have the whole model update. PanelingTools for Grasshopper supports various ways to shuffle grid point locations or distribute variable components based on attractors. Attractors can be points, curves, surface curvature or other methods. Attractor components calculate the weights of corresponding input grid and output the attracted point grid as well as the weights grid. Weights range between 0 and 1 reflecting the degree of attraction for each point in the grid. If the weight is "0", then it means that the corresponding grid point is not affected by the attraction. On the other hand, a weight of "1" means that the corresponding grid point will be most affected. Here is a list of different attraction methods available:

Attraction Method	Description
Points	Use points to attract towards, or away from them
Curves	Use curves to attract towards or away from them
Mean Curvature	Follow the surface mean curvature
Gauss Curvature	Follow the surface Gaussian curvature

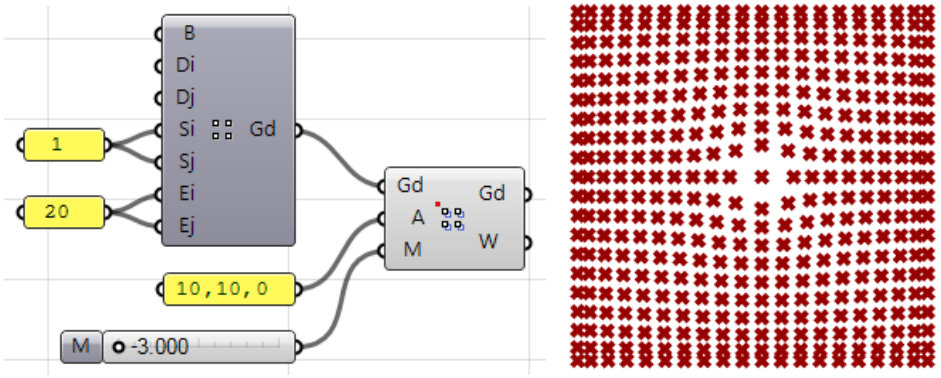
Vector	Attract relative to angle with predefined direction vector
Random	Randomly attract
Weights	Use explicit map of attraction values (0-1) per grid point

Point Attractors

PanelingTools offer a point attractor component where the user can input a grid of points, attractor point(s) and a magnitude; and gets shuffled grids of points and weights. If the magnitude value is positive, then points are attracted towards the input point(s) while if the magnitude is negative, then attracts away. The boundary of the grid is always maintained.

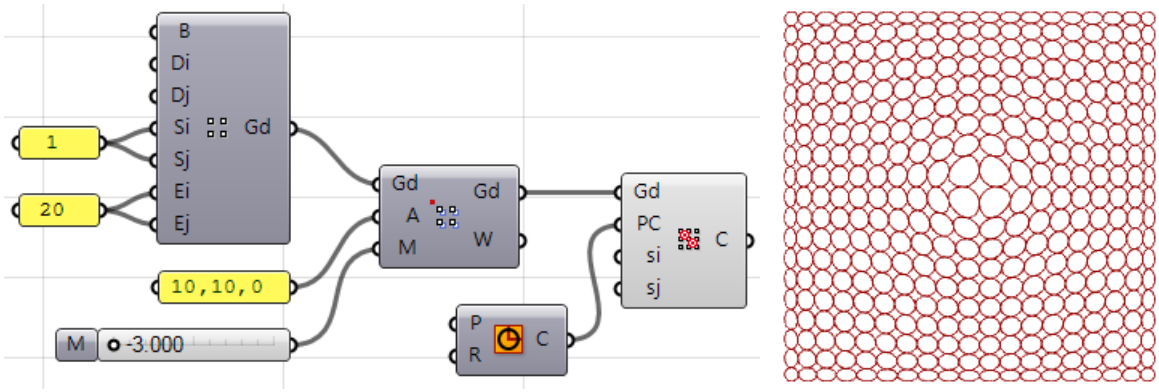


The grid points attract towards the center attraction point when magnitude (M) is positive value.



The grid points attract away from the center attraction point when magnitude (M) is negative value.

When you have a shuffled grid, populated uniform module will be variable in size because it will occupy the whole cell.

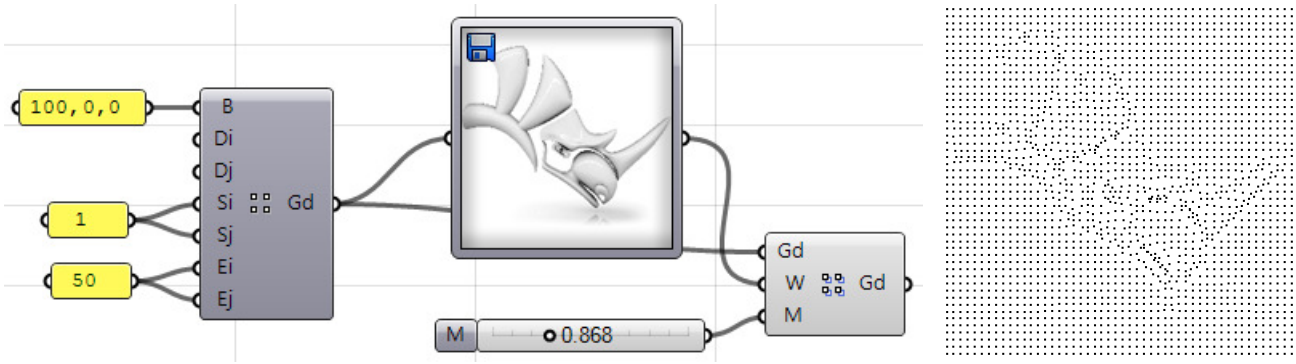


Use attracted grid as input to populate a circle using ptMorph2D component.

Bitmap Attractors

It is possible to use an image to attract or change locations of grid points. In the following example, I used the Rhino logo to attract the paneling grid. Points were moved depending on its greyscale value. The darker the sampled points are, the father they move.

In the GH definition, notice that the "M" value represents the Magnitude or the amount of attraction. It can be adjust to increase or decrease the movement of grid points.

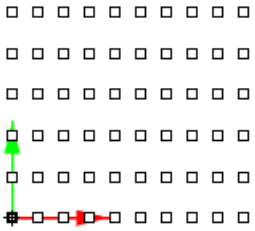


Selecting and Baking Grids

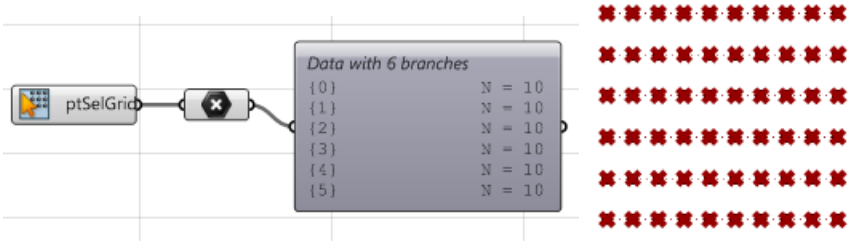
Grid can be generated using PanelingTools plugin for Rhino (outside Grasshopper). Those grids can be selected as an input in Grasshopper. Likewise, grids generated inside GH can be baked back to Rhino and can be used by the PanelingTools command in Rhino.

Select a Grid Example

First, let's create a paneling grid inside Rhino. From the "PanelingTools" menu in Rhino, go to "Create Paneling Grid" then select "Array". You can use the default values in the command options or change to create the desired grid. In this case, we have 10 points in the x direction and 6 points in the y direction.

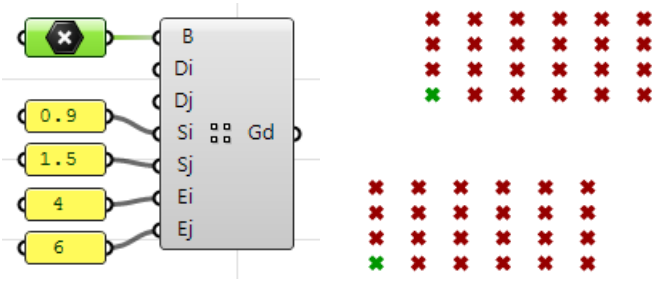


Suppose we need to use this grid as input in our Grasshopper definition. To do that, use ptSelGrid component, then click on the component icon. You will be prompted in Rhino to select the grid. Select the grid we just created and press enter. You'll notice that selected points are organized into 6 rows with 10 elements in each row.

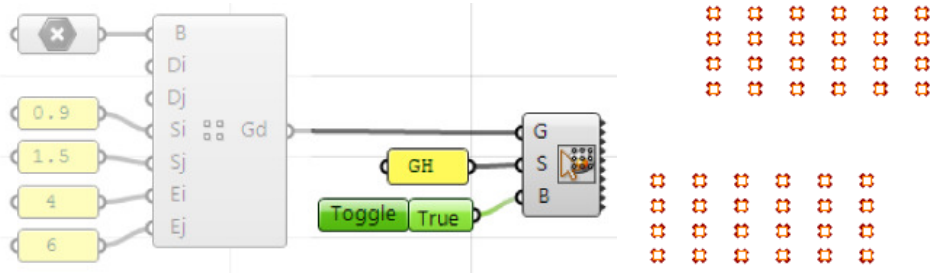


Bake a Grid Example

You can bake grids created inside Grasshopper into Rhino in a format that can be used by PanelingTools commands in Rhino. For example, create a grid using ptPlanar component in Grasshopper. For the base point input (B), pick two points in order to generate two grids. Set shift in I direction (Si) to "0.9", shift in j direction (Sj) to "1.5", number of points in I direction (Ei) to 4 and number of points in j dir (Ej) to 6 as in the following:



Use ptBake component to bake the two grids into Rhino by setting the toggle into "True". Make sure to set back to "False" so that you do not get multiple bakes whenever the solution is recalculated.



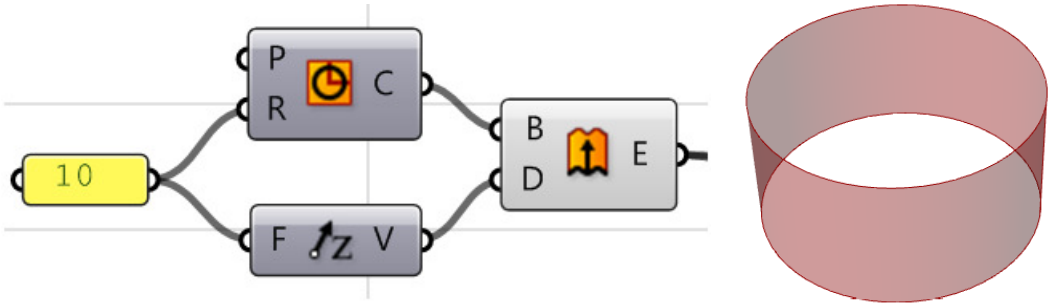
Tutorials

This section introduces number of tutorials that are meant to show how PanelingTools for Grasshopper is used to create parametric paneling solutions. It should give you a general idea about the context in which these tools may be used.

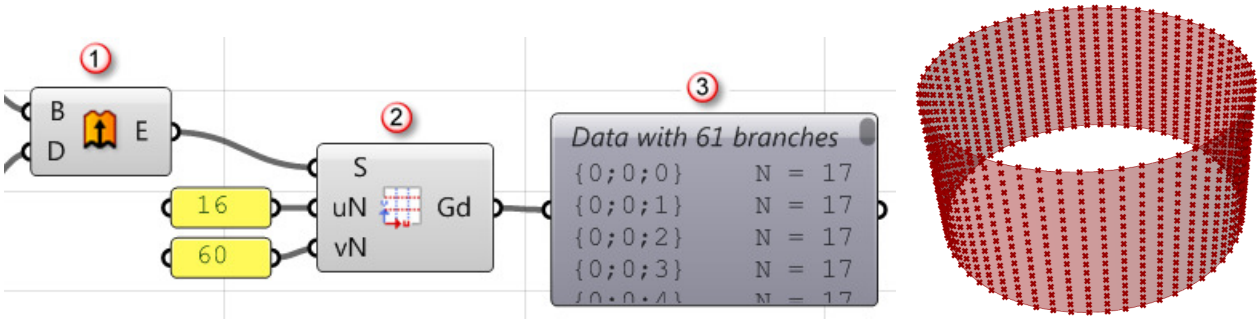
Diamond Panels

This tutorial introduces two different methods to create diamond panels. The first creates diamond panels by converting a rectangular grid into a diamond grid then paneling the new grid. The second keeps the rectangular grid but defines a connecting pattern for the diamond panels. Both are valid approaches and you can choose the one that works best with your solution flow.

First we create a hollow cylinder using GH standard components.

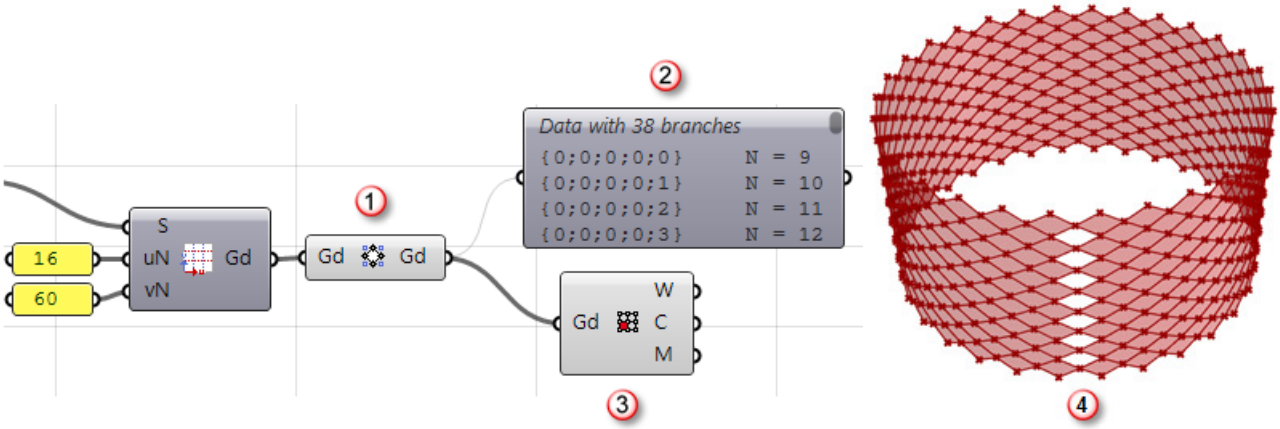


We then create a rectangular grid of points using ptSrfDomNum component. Notice that grid distribution follows the iso direction of the underlying surface. In this example, the "u" direction of the surface is vertical and hence the row directions are vertical too. Each row has "16" spans (or "17" points). Columns are in the circular direction and each column has "60" spans (or "61" points). The first and last points in each of the 17 columns overlap because the surface (cylinder in this case) is closed surface.



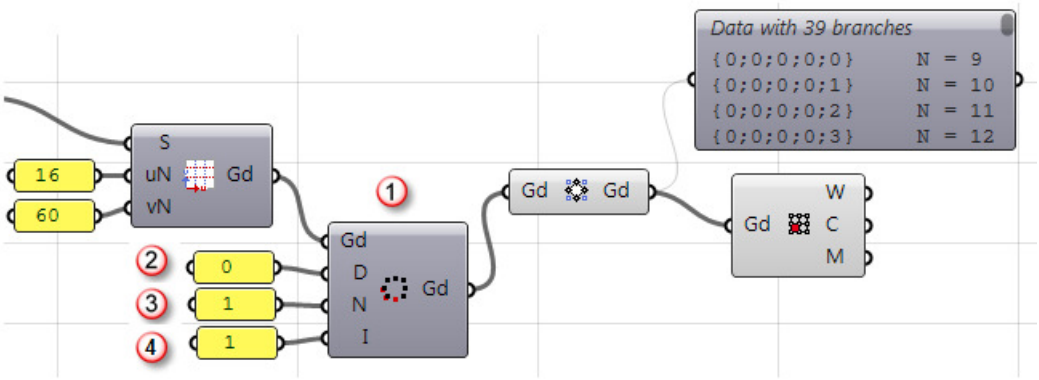
Cylinder surface from extrude (1), create a grid by surface domain number (2), result grid has 61 rows or branches of points, each has 17 points (3).

Following the first method of creating diamond panels, you can directly use ptToDiamond component to extract a new rectrangular grid in the diagonal direction and then use ptCeluate component to get the panels.

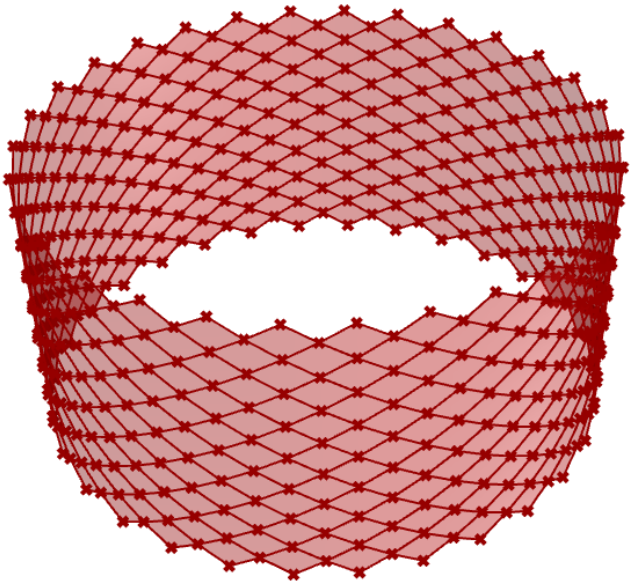


Convert to diamond grid (1), rows of the diamond grid have variable number of elements (2), create panels(3), generated panels in preview (4).

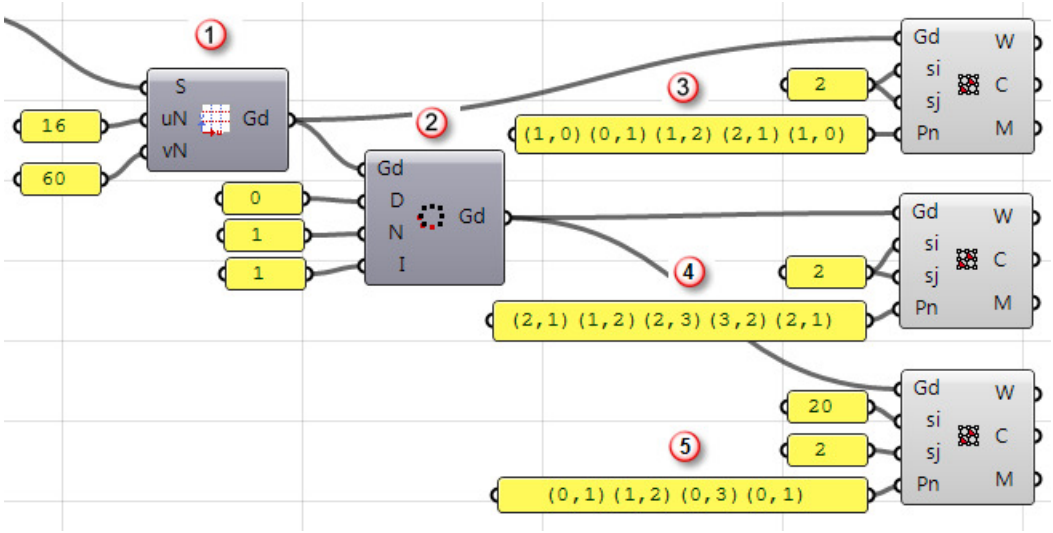
Notice, there is an apparent missing panels along the seam. This is because the pattern effectively ran out of grid points to cover. To deal with this situation, you need to wrap the grid to have one extra row that overlaps the second row. Keep in mind, that the first and last rows already overlap; we just need one extra row. There is a component in PT-GH that helps with that called ptWrap.

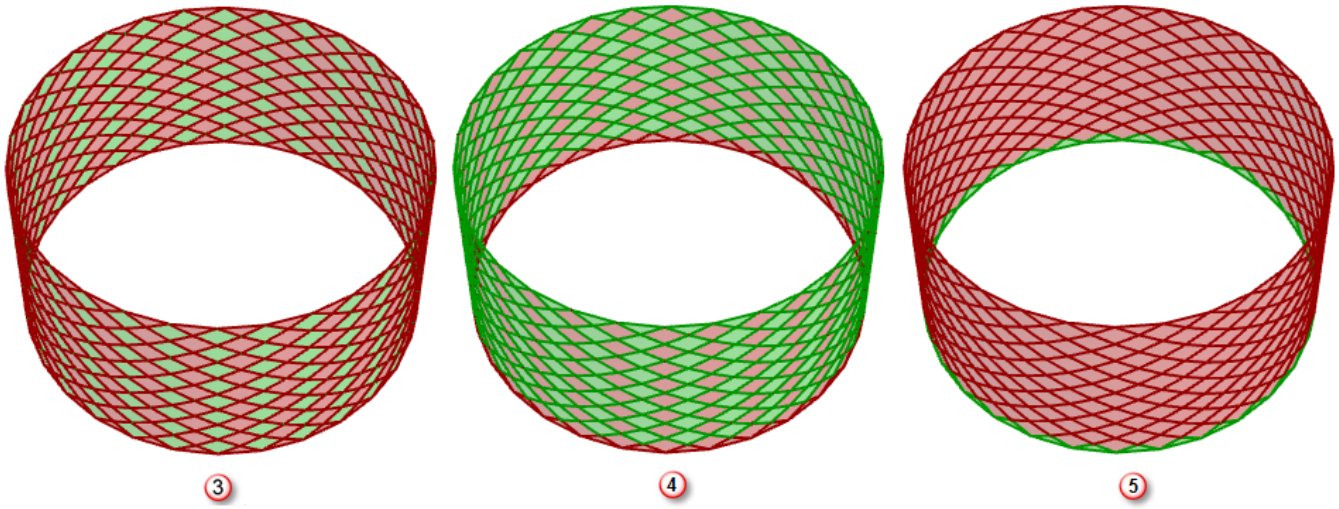


Wraps the grid (1), wrap direction (0= wrap extra rows, 1=wrap extra columns)(2), number of rows/columns to wrap(3), starting index to start wrapping from (4).



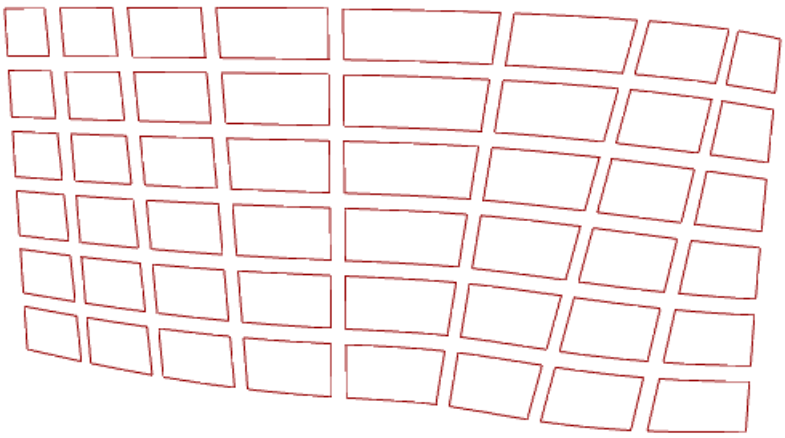
Another approach to creating a diamond panels is to use the ptMPanel component. We still need to create the grid and wrap it one extra row. The following illustrates how the definition works. Each of the three ptMPanel components used accept a grid (Gd), shift in u and v directions (si & sj) and a string (Pn) that represent the (u,v) unit points each pattern connects.



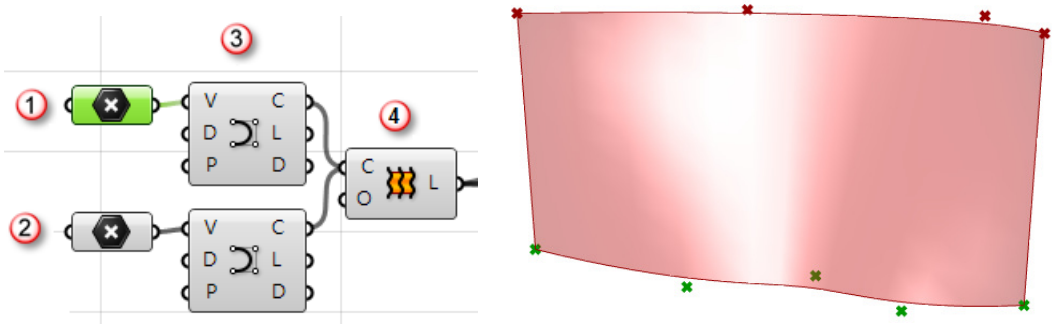


Generate grid using surface uv domain by number (1), wraps the grid (2), first group of diamond panels(3), second group of diamond panels (4), third group of panels along the edge(5).

Fixed Gaps between Panels

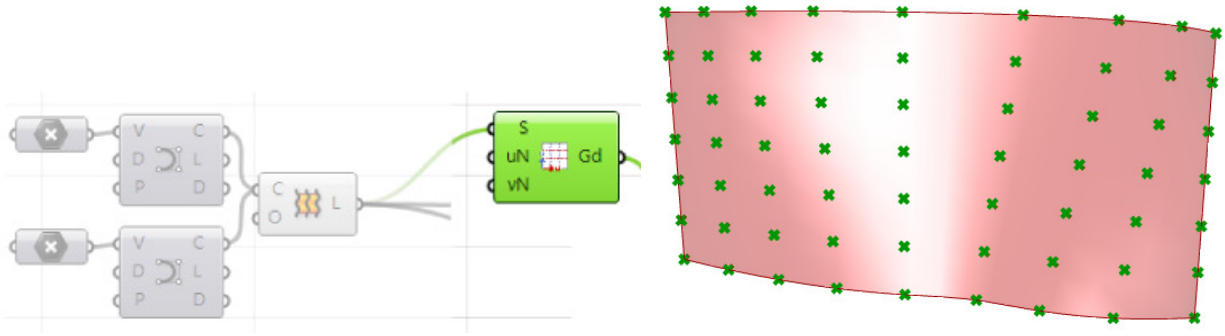


This tutorial shows how to achieve a fixed gap between panels that are based on free form surface. Start the GH definition with creating a free form loft surface from two NURBS curves.

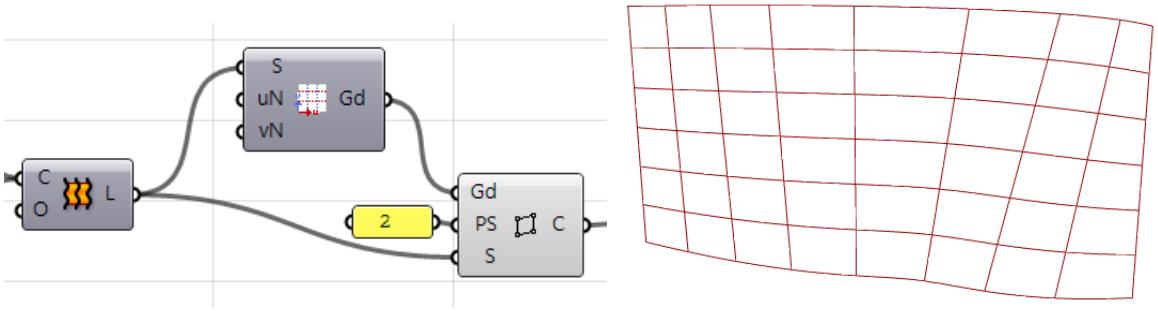


Control points of first loft curve (1), points of second loft curve (2), curves (3), loft surface (4).

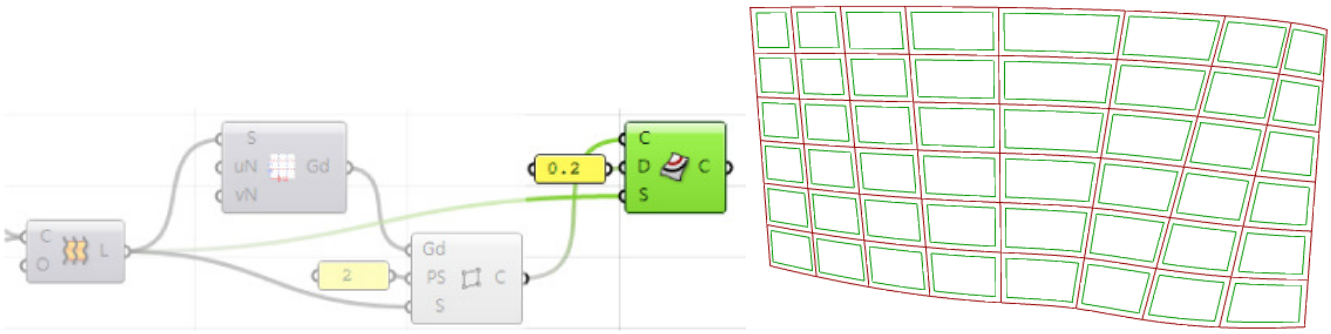
Next, create a grid on surface using ptSrfDomNum component.



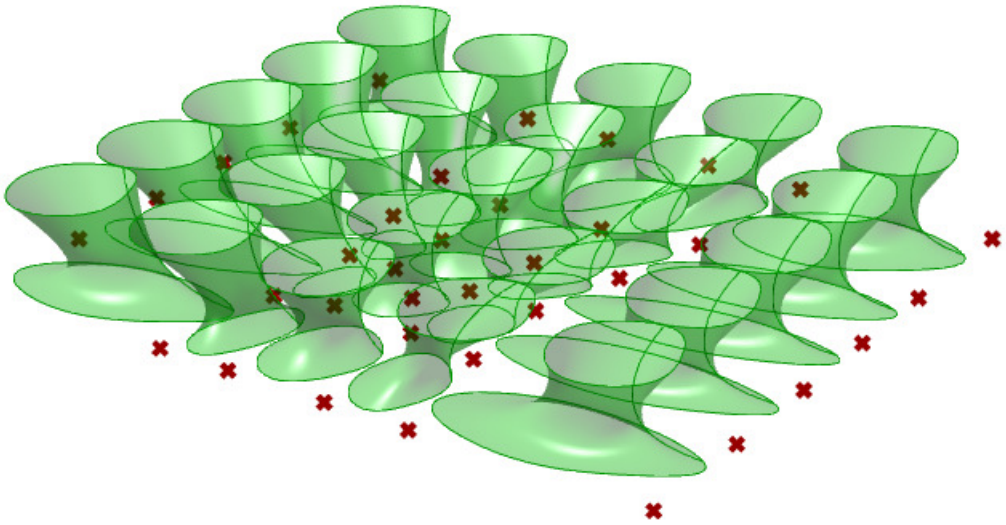
In order to get the panels as a polycurve outline, use ptPanel component



The last step is to use Grasshoppe OffsetS component to offset panel outline by fixed distance on the surface.



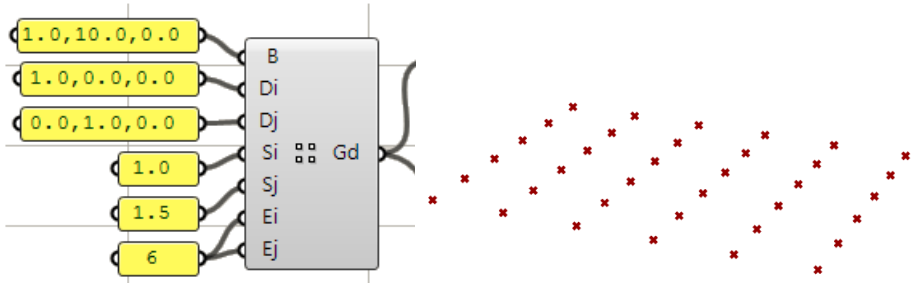
Loft Morphed Curves



This tutorial shows the following:

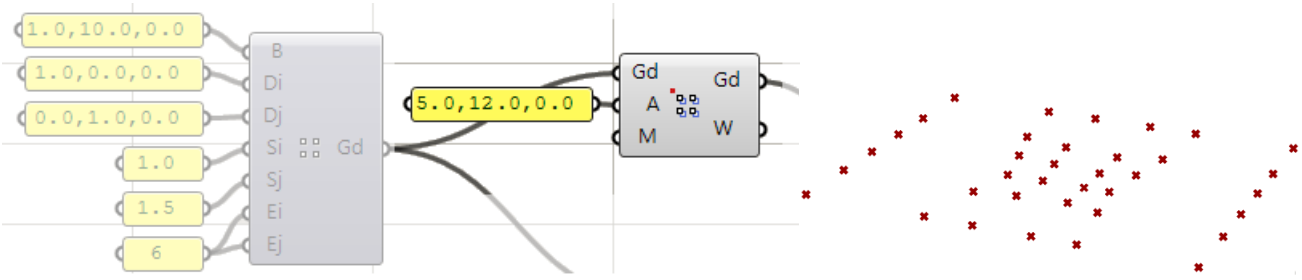
- How to create attracted grid.
- How to morph module curves in 3D space.
- How to create 3D modules from morphed curves.

Start the GH definition with creating a rectangular grid using ptPanar component.



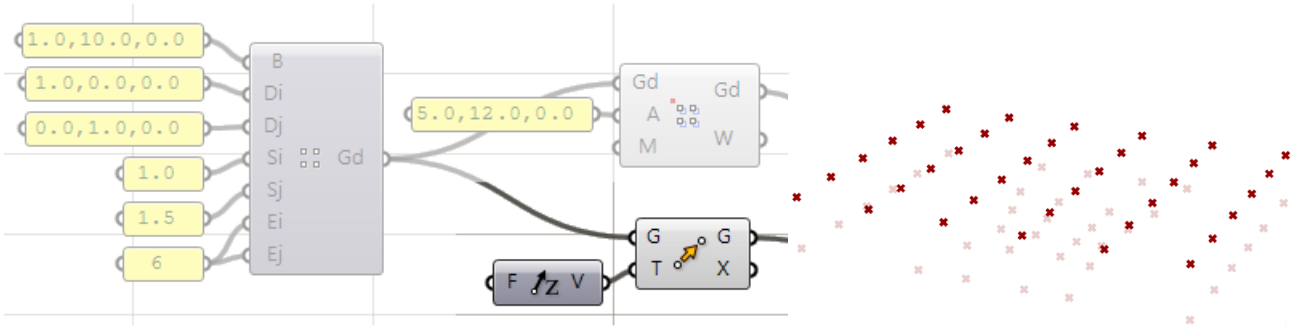
***B** (base point) = (1.0,10.0,0), **Di** (row dir) = (1.0,0.0,0.0), **Dj** (col dir) = (0.0,1.0,0.0), **Si** (row spacing) = 1.0, **Sj** (col spacing) = 1.5, **Ei** (row number) = 6, **Ej** (col number) = 6.*

Next add an attractor point and change grid points to attract towards the attractor point.



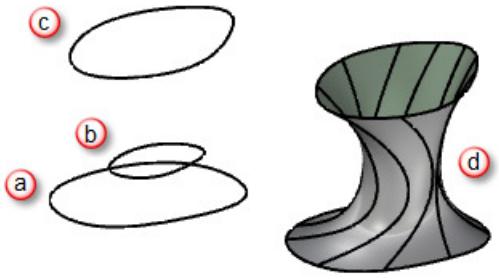
***Gd** = grid to be attracted, **A** (attractor point) = (5.0,12.0,0.0), **M** (Magnitude of attraction) = 1 (default), **Gd** (output) = attracted grid, **W** = weights grid (attraction degree for each grid point 0-1).*

We need a second bounding grid to populate our module in between. Copy the original planar grid in the Z direction.



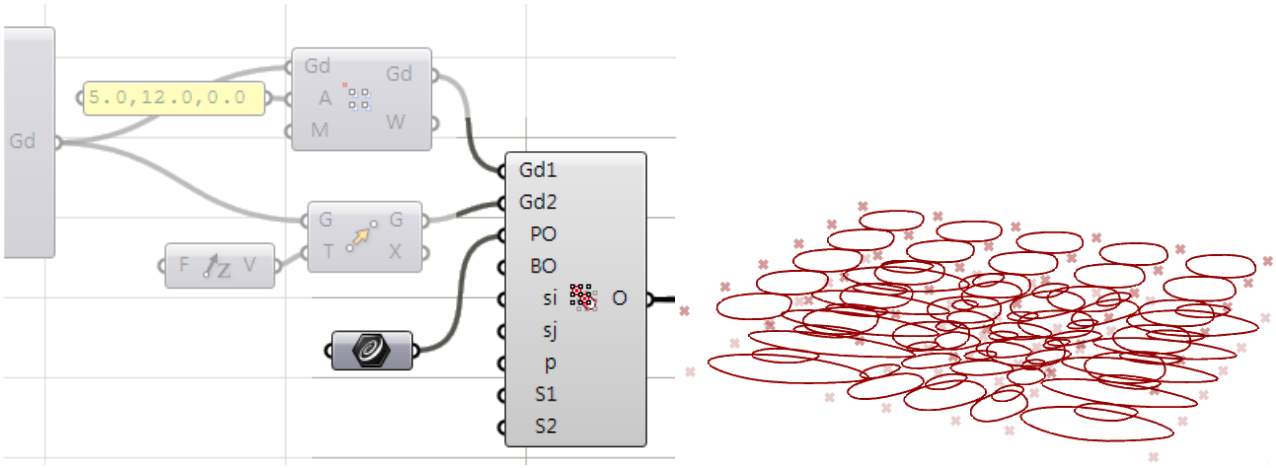
***G** = input geometry, **T** = input vector.*

Next create the module curves in Rhino. In this case, we use three curves to define a loft surface. We will morph the curves rather than the lofted surface because it is faster and more efficient.



First module curve (a), second module curve (b), third module curve (c), lofted surface (d).

Reference the module curves in the next step to morph between our two bounding grids using the 3D morphing component.



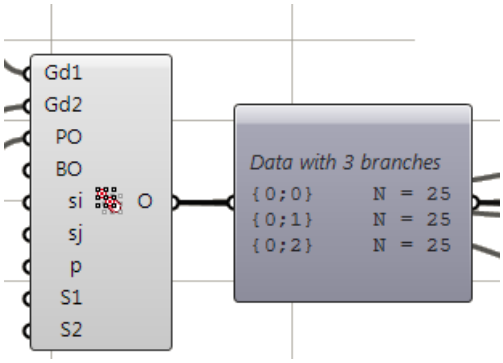
***Gd1** = first bounding grid, **Gd2** = second bounding grid, **PO** = pattern objects, **BO** (optional) = bounding objects for the pattern objects, **si** = shift in the i direction = 1 (default), **sj** = shift in the j direction = 1 (default), **p** = pull for smooth morphing = false (default), **S1**(optional) = grid1 surface, **S2** (optional) = grid2 surface.*

Note that the output curves from the 3D morphing component are organized into 3 branches.

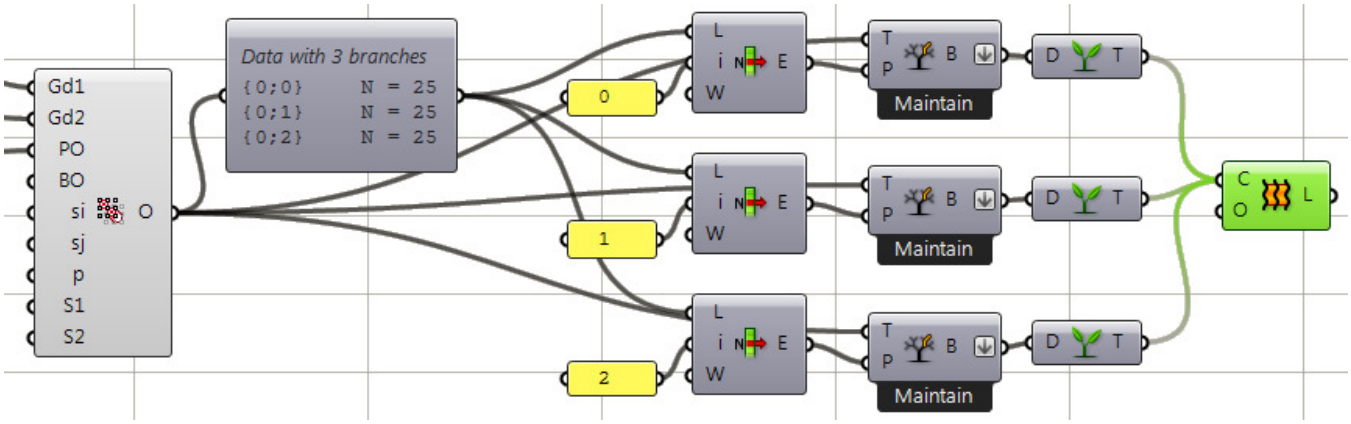
{0;0} holds morphed "a" curves.

{0;1} holds morphed "b" curves.

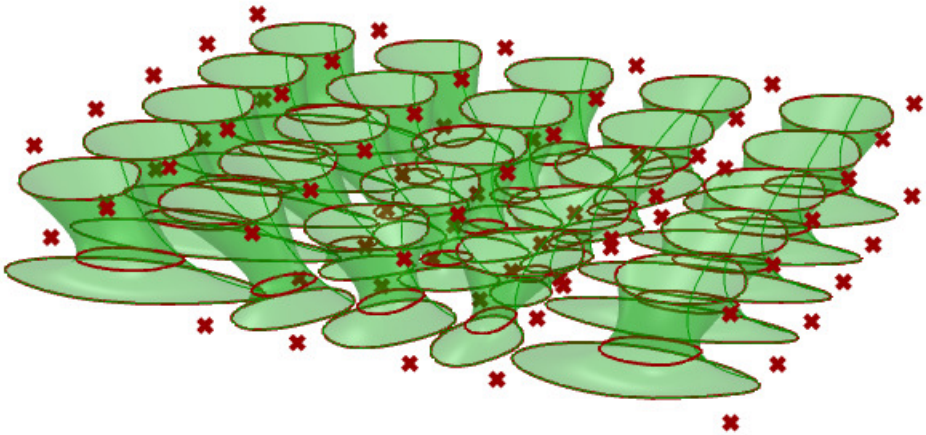
{0;2} holds morphed "c" curves.



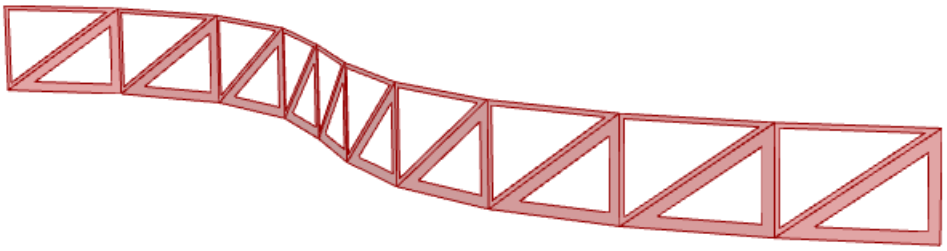
In order to loft morphed curves, we need to separate the three branches before feeding them into the GH "Loft" component. You can do that by separating the branches of the tree, then graft each branch before feeding into the GH "Loft" component as in the following:



This how the lofted modules look:



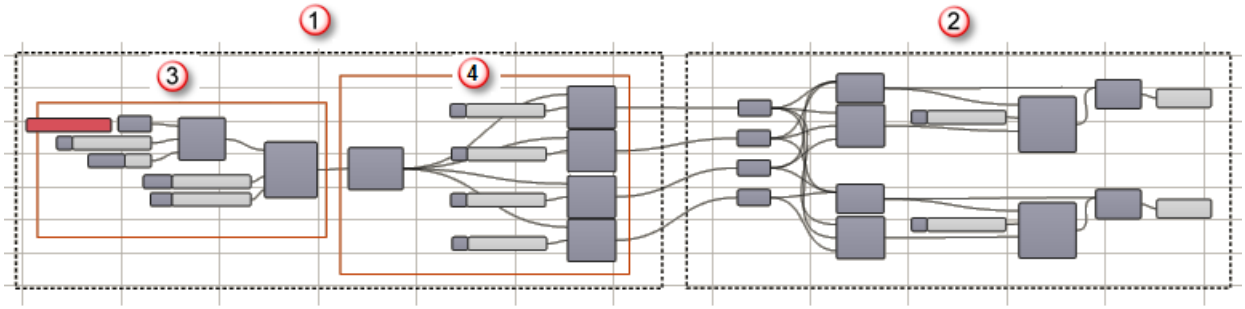
Parametric 2D Truss



This tutorial shows how to create a parametric truss that is based on a curve. It is based on David Fano's truss tutorial. The main advantages of using PanelingTools Add-On (PT-GH) over GH standard components are:

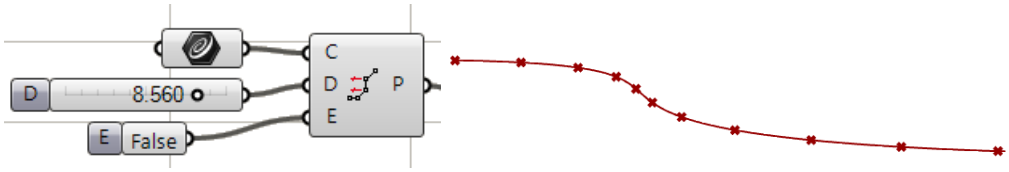
- System logic is easier to understand, create and edit.
- System logic is more flexible. It is not restricted to surfaces and their iso-curve directions which greatly limit user control over dimensions and orientation of truss components.
- The truss component logic is based on points, rather than surfaces, which is lighter.

The overall definition is structured into two parts. The system logic (1) and the component logic (2). The component logic uses standard GH components based on four corner points. The system logic defines a rectangular grid of cells using PT-GH components.



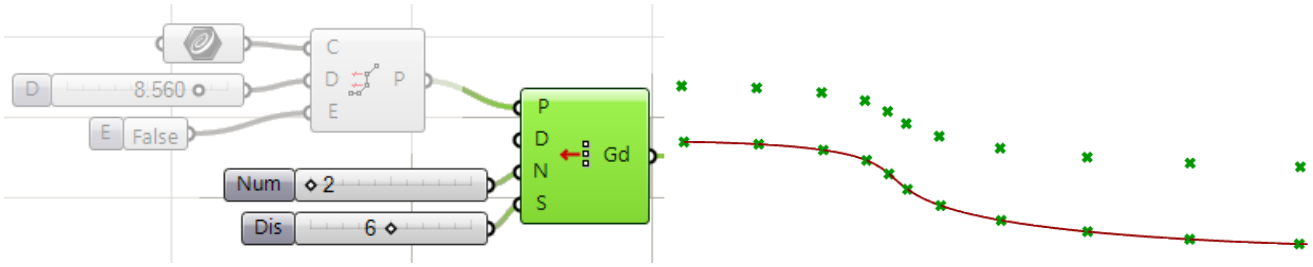
System logic (1), component logic (2), create system grid (3), extract components corners in the system (4).

To define the system logic, first we need to create a grid. In this case our grid is based on a curve[1]. First step is to create a reference a curve in Rhino, then divide the curve by distance which represents the width of the truss.



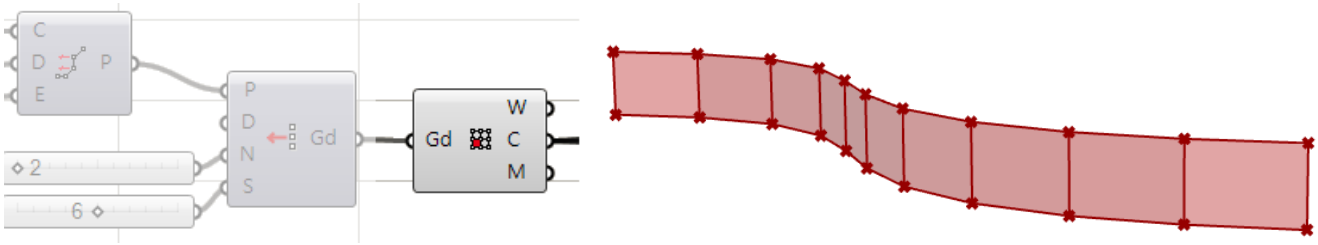
[1] *There is a variety of ways to generate the basic grid of cells using grid tab in PT-GH or simply by feeding a tree structure of points using GH standard components such as divide curve components.*

Now that the curve is divided, we generate the grid using the **Planar Extrude** grid component under **Grid** tab of the **PanelingTools** menu. Grid components in PT-GH generate two dimensional grids of points and organize them into a simple GH tree structure where each branch contains a list of points representing grid rows.

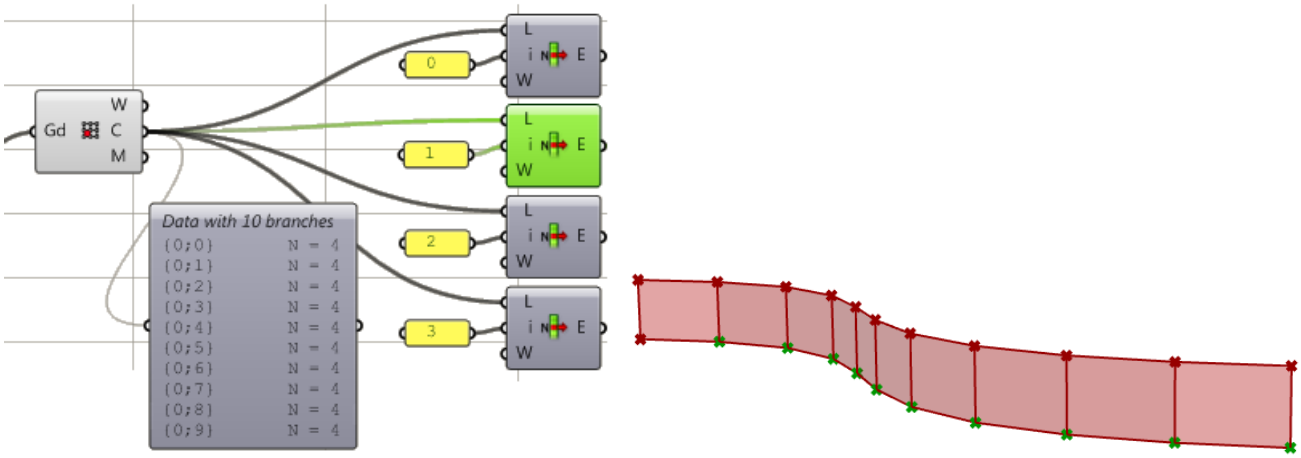


Next we need to extract individual cells of the grid. To do that we use the **Cellulate a Grid** component. This component in under **Panel2D** tab. It outputs three components:

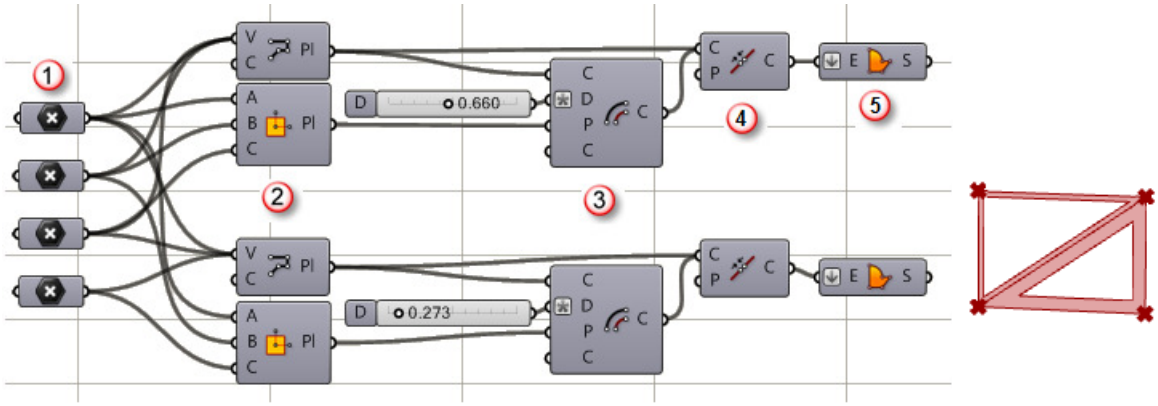
- **W** (Wires): a list of all edges.
- **C** (Cells): a list of the four corners of each cell (this is what we need here).
- **M** (Meshes): a list of mesh faces of all cells.



We have 10 cells, each has 4 corners. We need to get a separate list of each corner to feed into our component logic. We used GH list component to separate into 4 lists of corners.

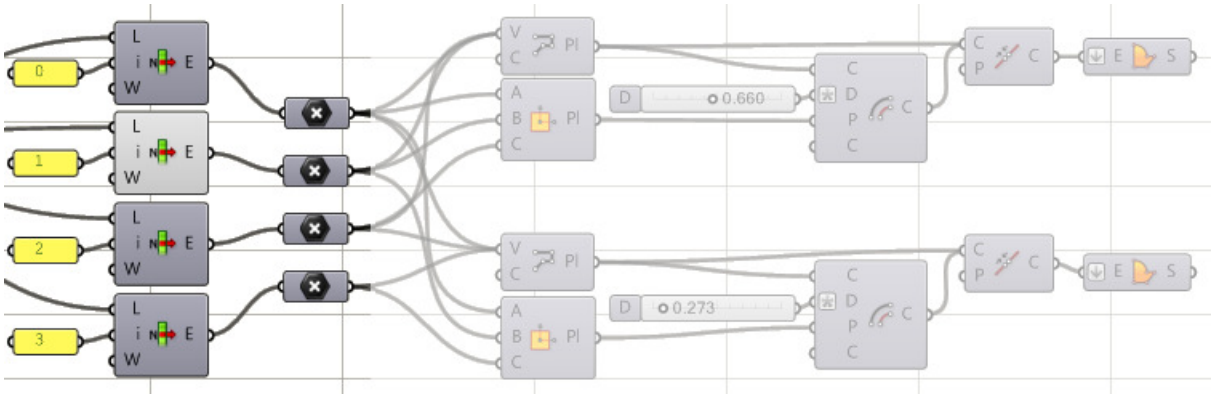


Next, create the component logic of truss units based on 4 points. Basically divide into two triangles. Each triangle can have its own thickness and creates a trimmed planar surface.

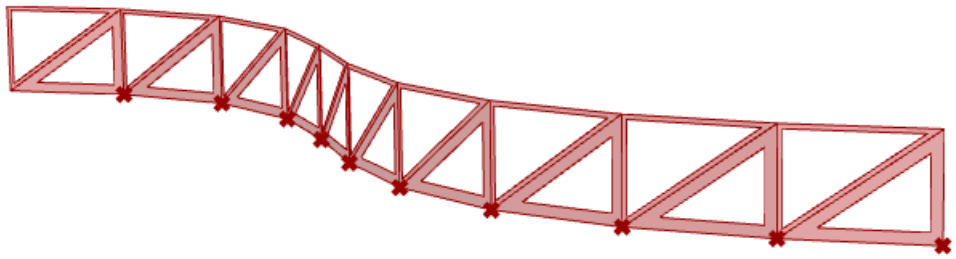


Truss unit corners (1), upper and lower triangle polylines (2),offset triangles by distance specified in the slider(3), join (4), create planar surface.

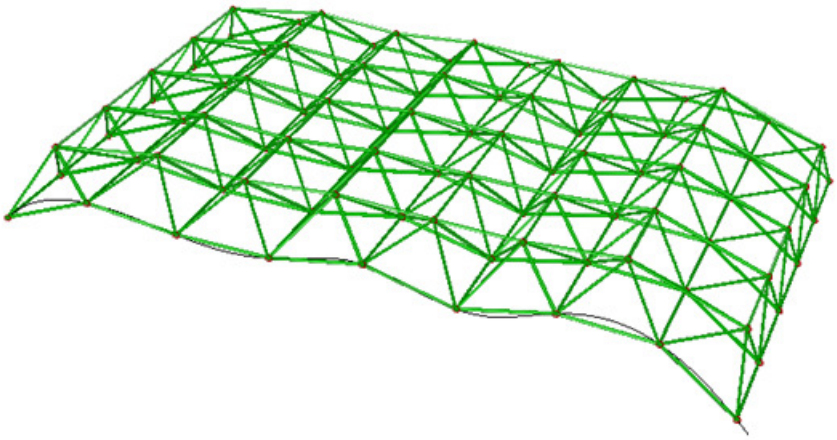
Finally, hook the system points into the custom truss component logic that is based on 4 points.



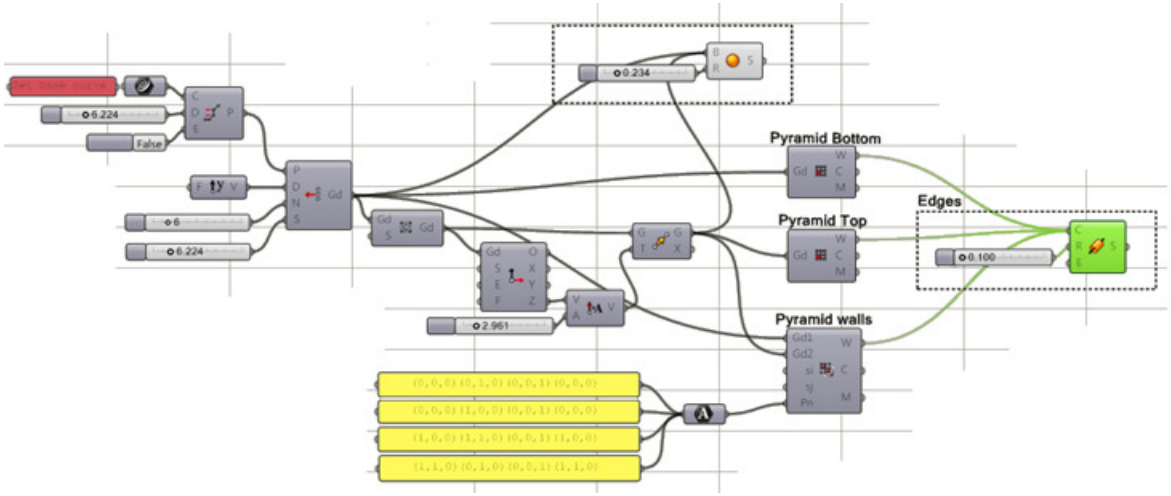
This is how the final truss looks like.



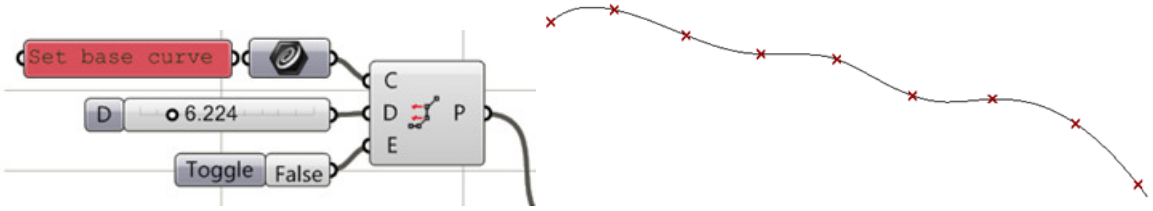
Parametric Space Frame



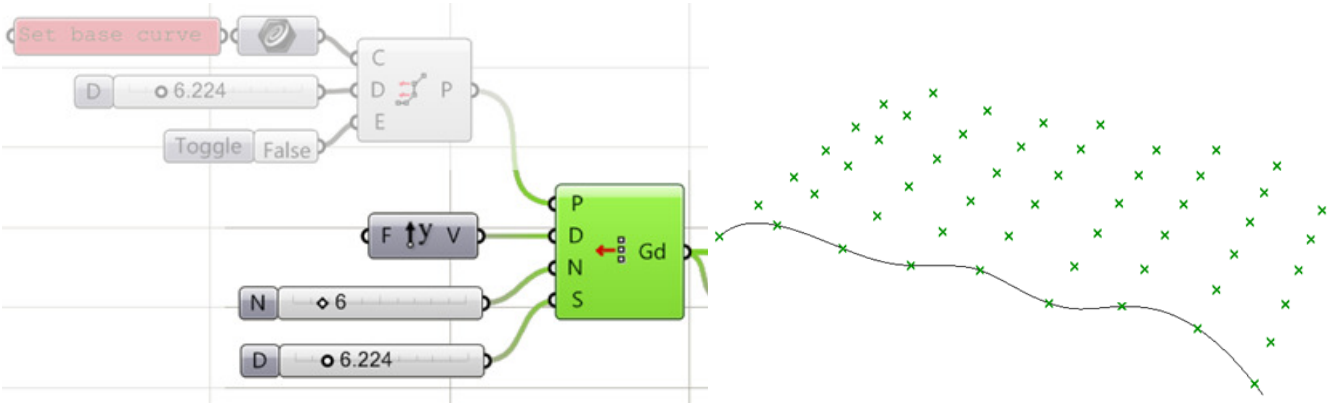
This tutorial shows how to create a parametric space frame based on a curve. While it is possible to create similar definition using standard Grasshopper components, using PanelingTools for GH makes the definition easier to write, read and edit. It also enables better control over dimensions and shape. This is how the final definition looks like:



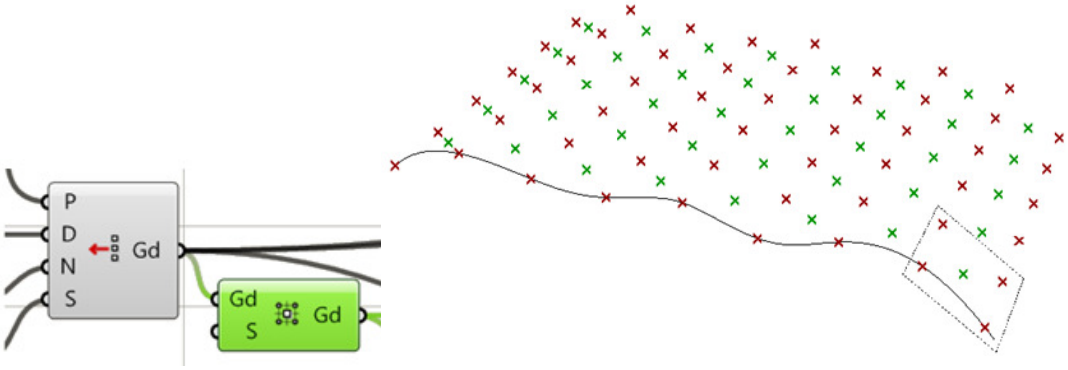
First step is to divide a given curve by distance that represents the width of the base cells of the space frame.



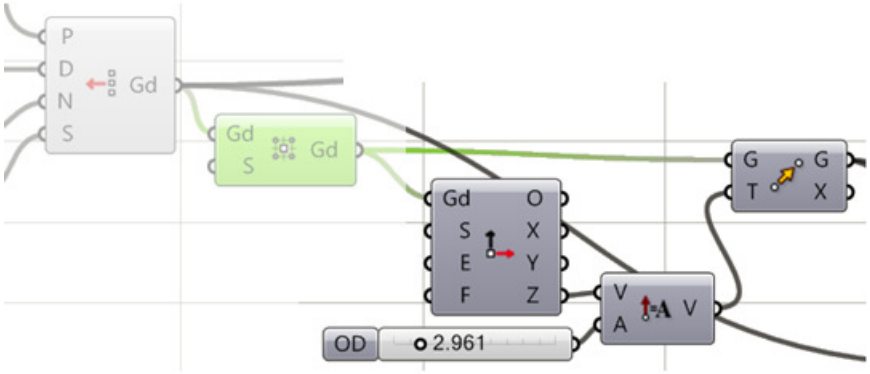
Now that the curve is divided, we generate the grid using the **Planar Extrude** grid component under **Grid** tab of the **PanelingTools** menu.



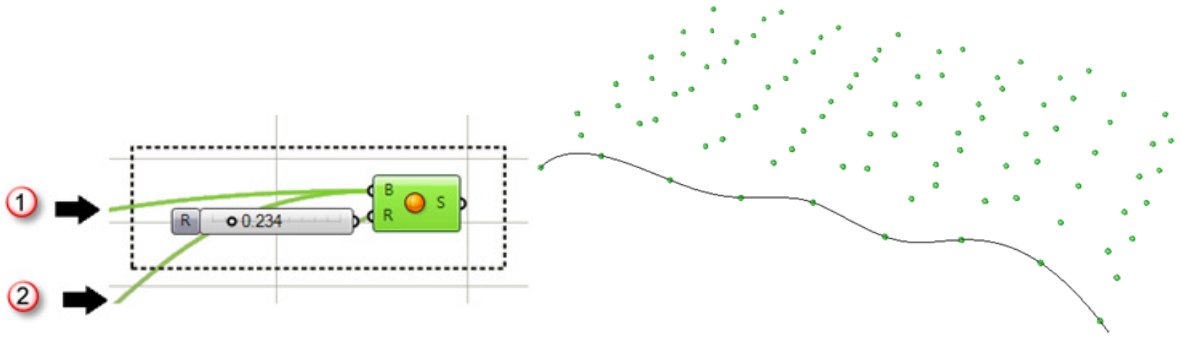
Next we need to extract center grid then move in the normal direction of that center grid using the center grid component in PT-GH.



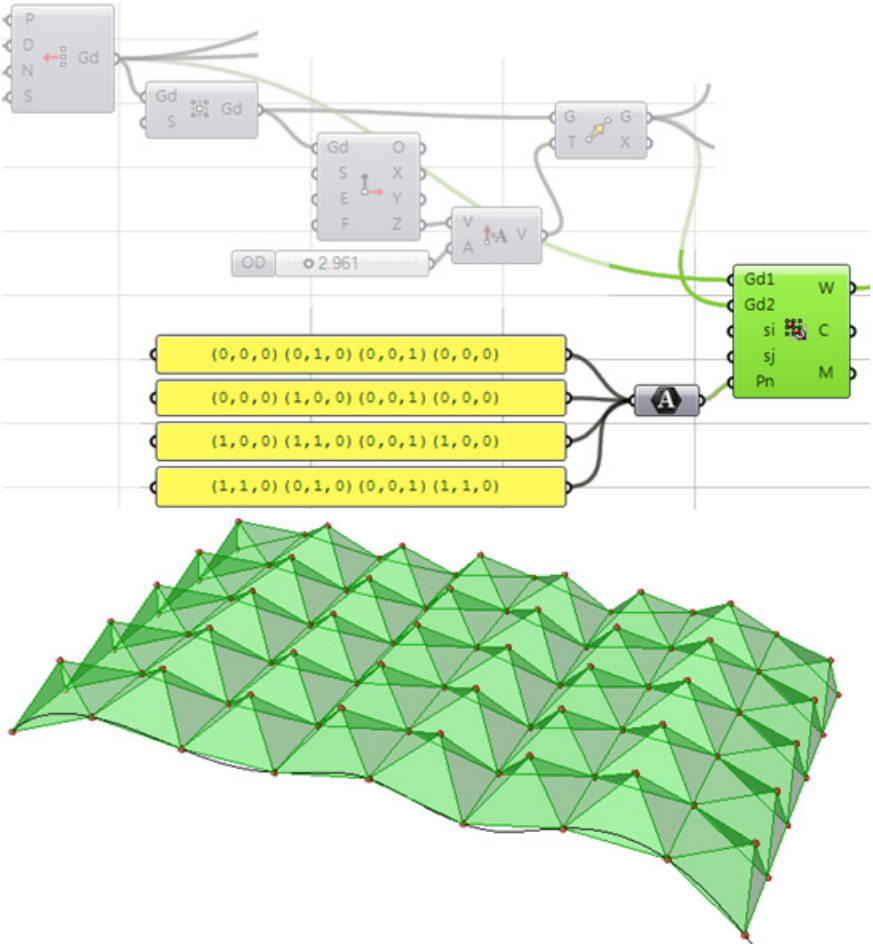
In order to move in normal direction, use the coordinate component extracts origin of each grid cell and x, y and z directions. Use the center grid to extract the normal direction that we can then use to create the top bounding grid of the space frame.



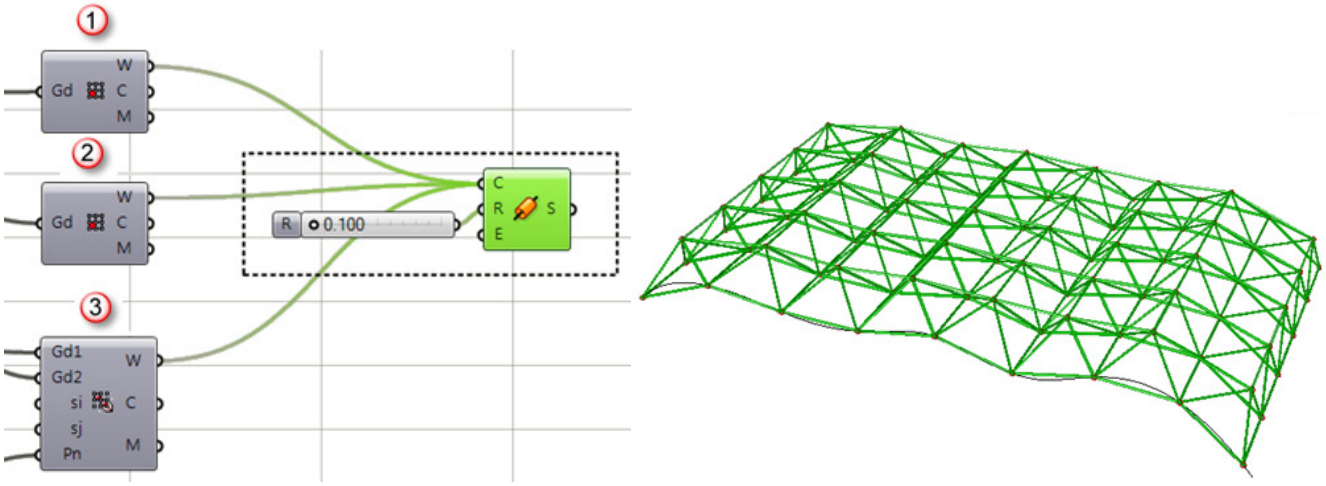
Next we create small spheres to mark the joints. Use GH Sphere component and input both the base grid and moved center grid.



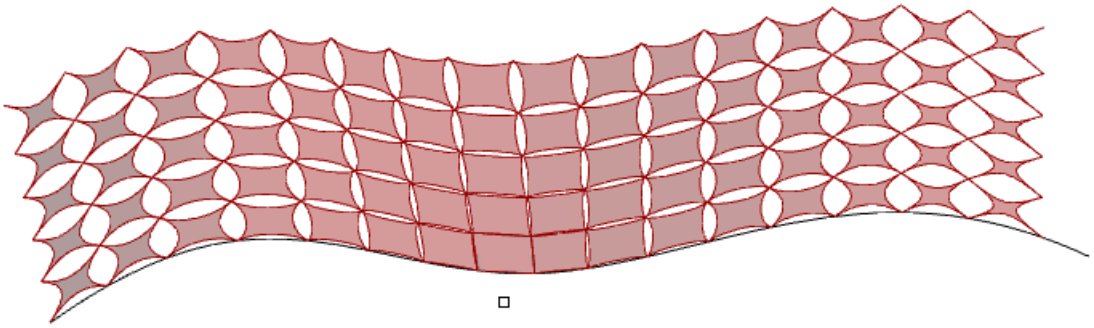
Use cellulate component in PT-GH to generate the bottom and top wires. Wall wires and faces of the space frame is created with the managed 2D panel component where each face is defined with a string. The string defines indices of grid to connect. These connections are repeated throughout the grid.



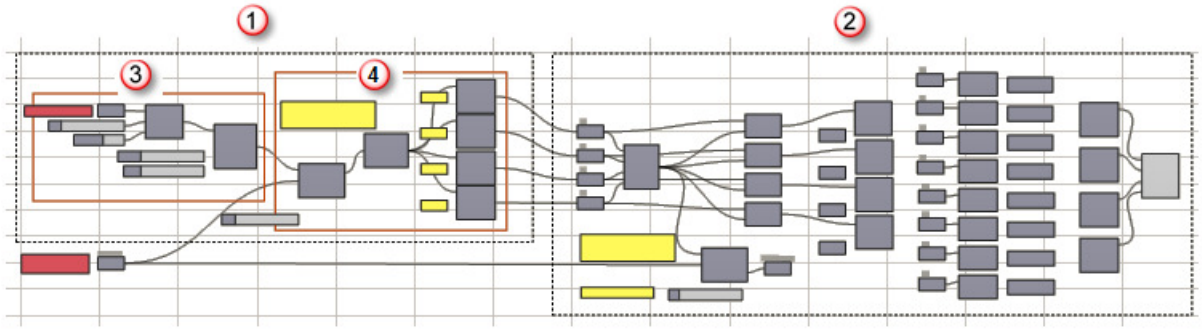
Wires from the top, bottom and walls are then used to generate edge pieces of the space frame using the cylinder GH component



Variable Wall

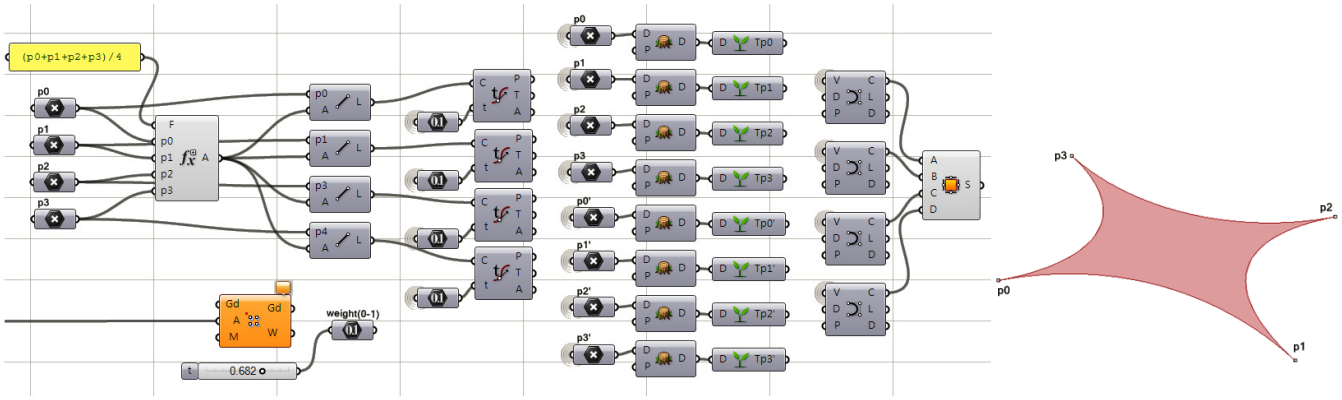


This tutorial shows how to create a parametric wall with variable system and variable components based on a curve. The wall component logic uses standard GH components based on four corner points. It also uses attractor component from PT-GH to create variable component. The system logic defines a rectangular grid of cells using PT-GH. The system has variable cell size using PT-GH attractor component.

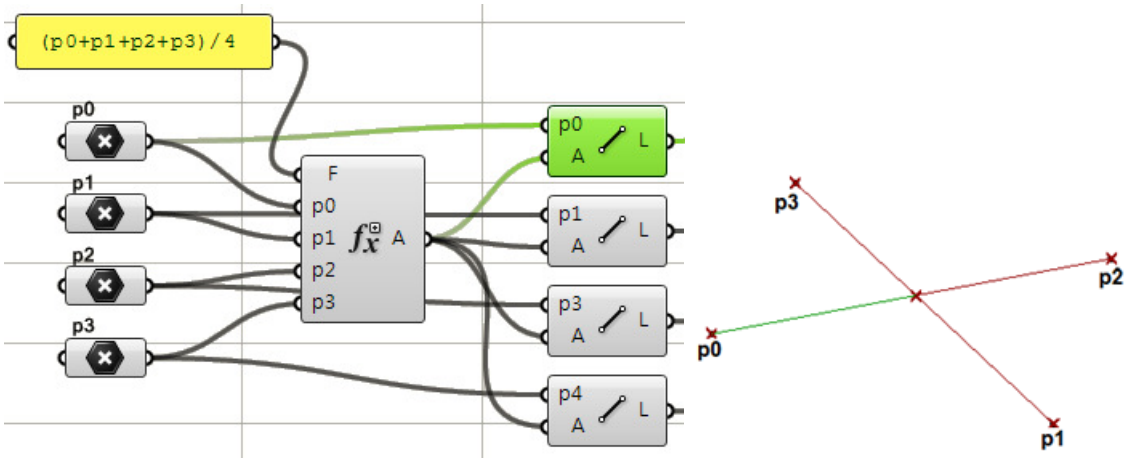


System logic (1), component logic (2), create system grid (3), extract components corners in the system (4).

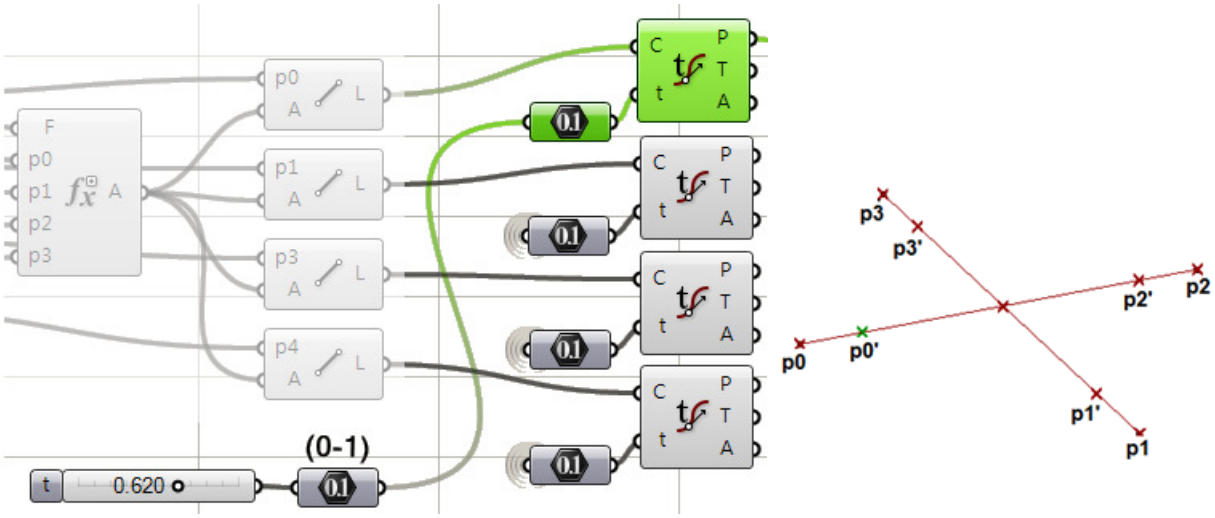
Let us start with building the component logic. The following image shows an overview of the component logic based on 4 corner points and weights that controls the shapes of edge curves. Keep in mind, that once the component logic is hooked to the system logic, there will be a list of corner points instead of just one. This is why we need extra steps like flattening and grafting the lists to get correct results.



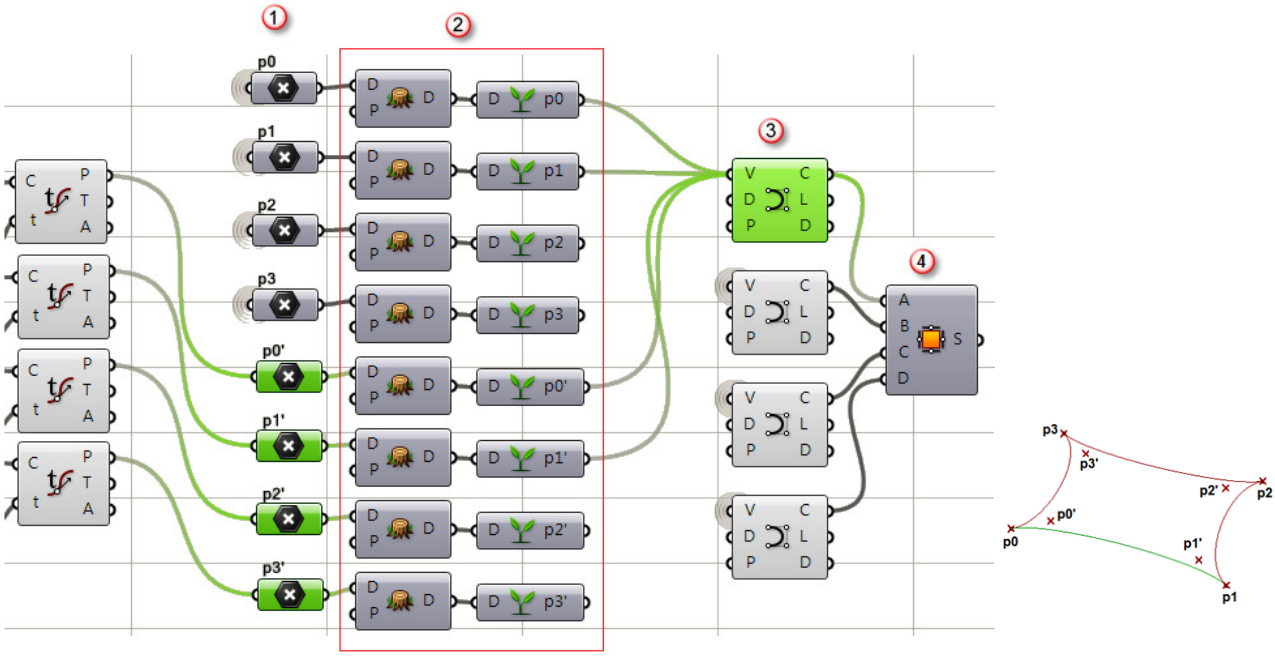
The component logic defines four corner points and mid-point, then connect each of the corner points with the center using line component.



Define points on each of the lines that fall within the line domain. Here we are using constant number. You can also hook a number slider between 0-1 to see the effect of changing the weight on the final component.

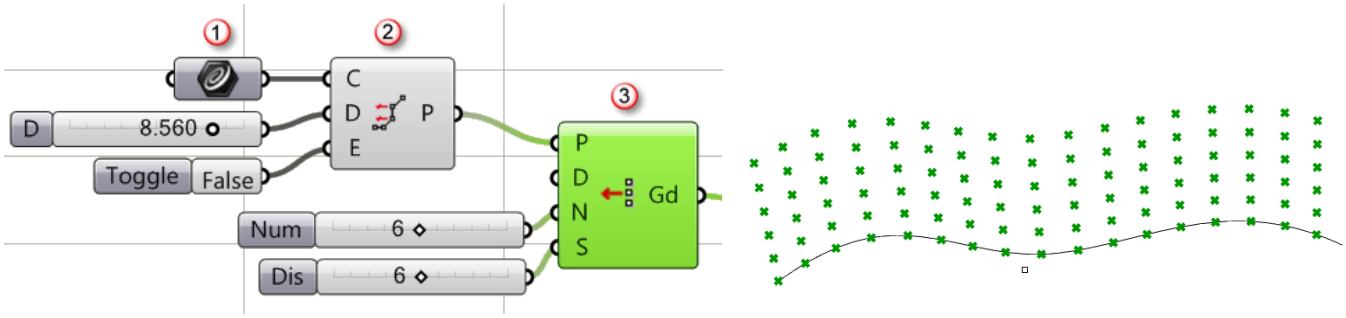


Next, create a curve using corner and line points as illustrated. This concludes the creation of the parametric component. Each component represents a cell in the wall system that we will build next.



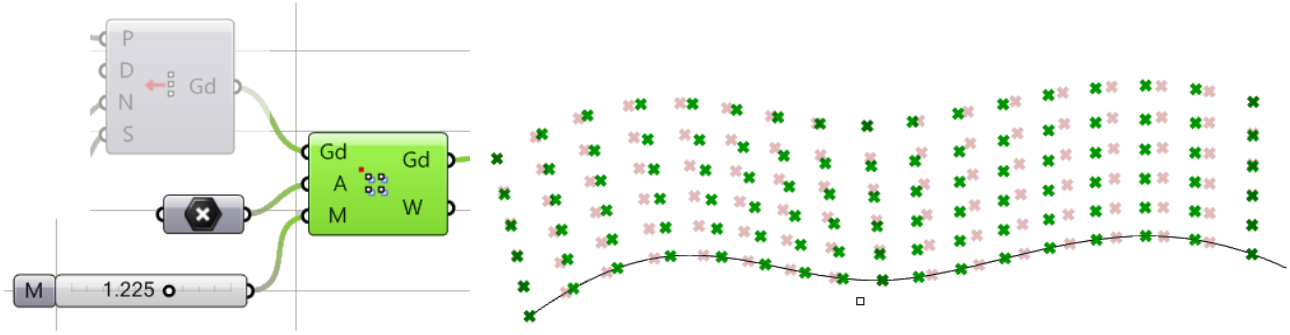
All 8 component points (1), flatten and graft necessary when hook to system (2), create edge curves (3), create edge surface (4).

Next we create the wall system, which is based on a curve created in Rhino and referenced by GH. You can use any of the curve divide components to get the list of points and feed into the linear extrude grid components.

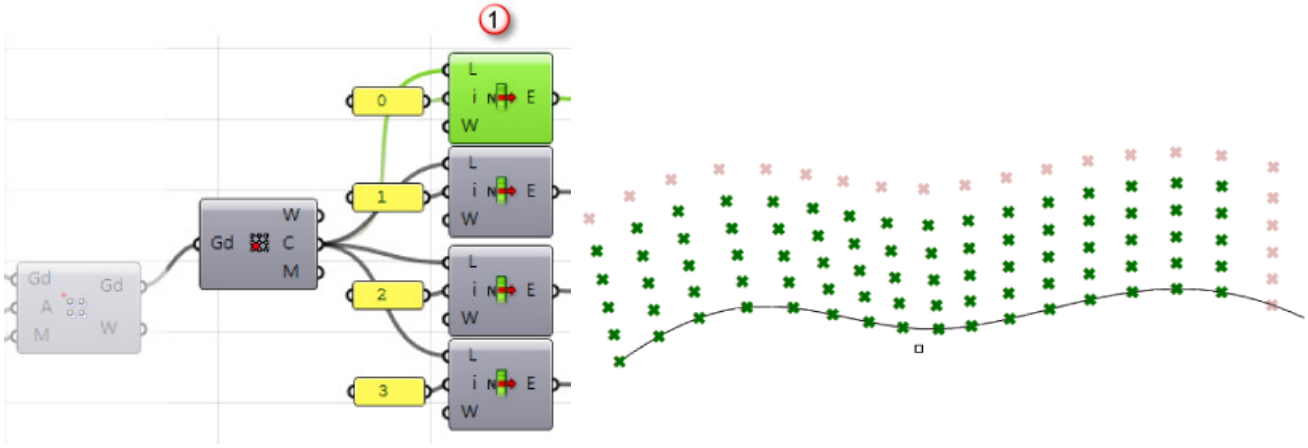


Input base curve (1), Divide curves by distance (2), Grid from extrude (3).

Next, we use an attractor point to vary cell size and basically shuffle grid points. Green points are the ones shifting towards the attractor point.

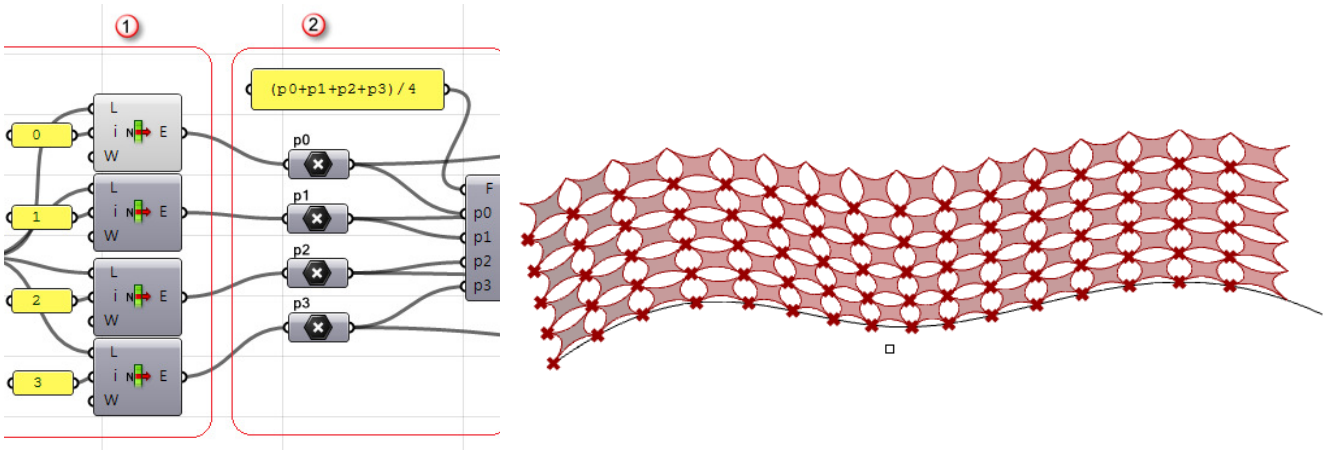


We need to extract lists of corner points of the system cells to feed into our component logic. To do that, use the cellulite component in PT-GH and separate each of the corners as illustrated.



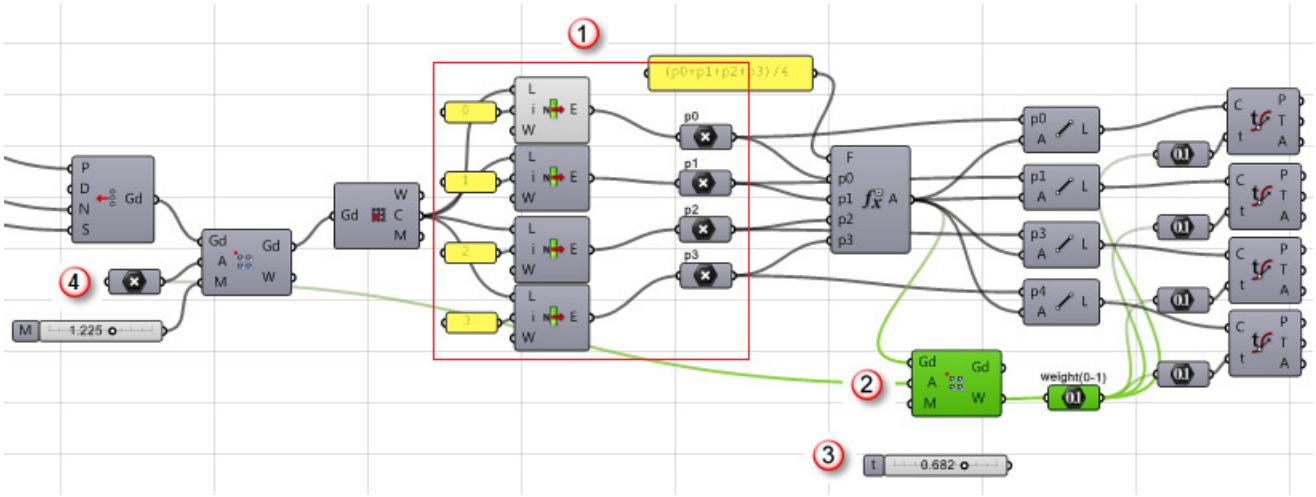
Separate corners into 4 lists (1).

Once we hook the system into the component logic, we get components populated over the system.



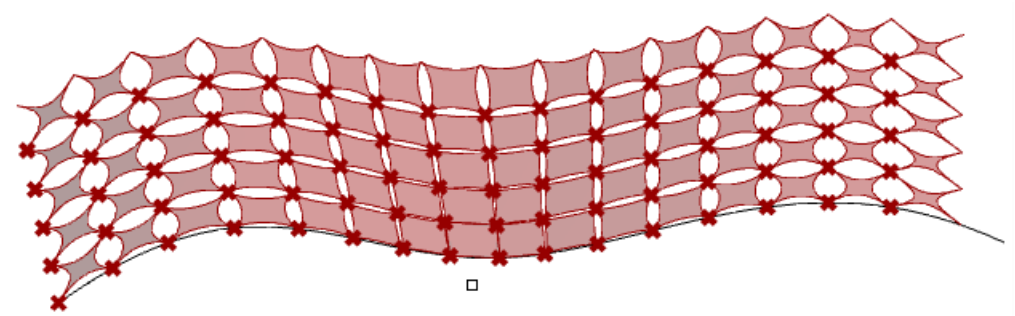
System logic (1), component logic (2).

In order to make the component distribution variable, we can feed variable weights based on distance from the attractor point.

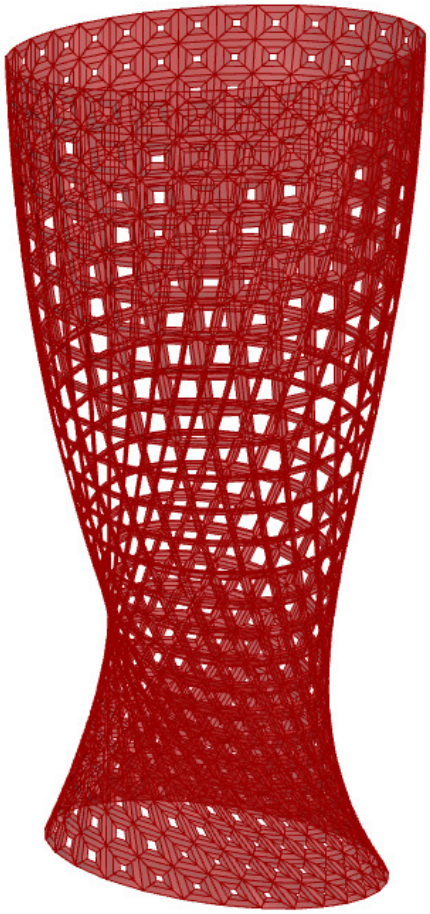


Connect the system logic to the component logic (1), variable weights (2), constant weights (3), attractor point (4).

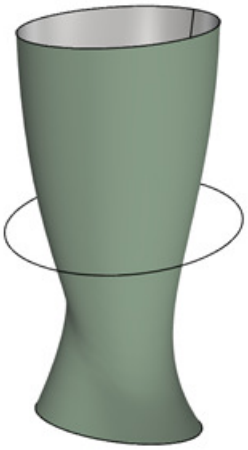
Here is the final result when using variable weights.



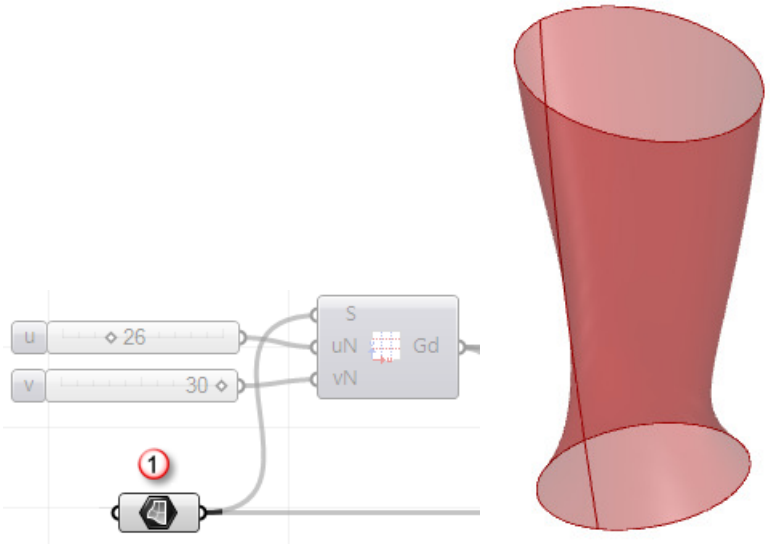
Tower



This tutorial distributes a list of variable mesh-based modules on the tower skin based on a curve attractor. The modules have variable opening size and are distributed so that modules with the biggest opening attract closer to the curve. The modules were modeled in Rhino, but you can choose to parametrically define a module inside grasshopper and control the aperture using the attractors.
The first step is to module the geometry elements in Rhino. In this case, we have the tower surface, attractor curve and the module-list.

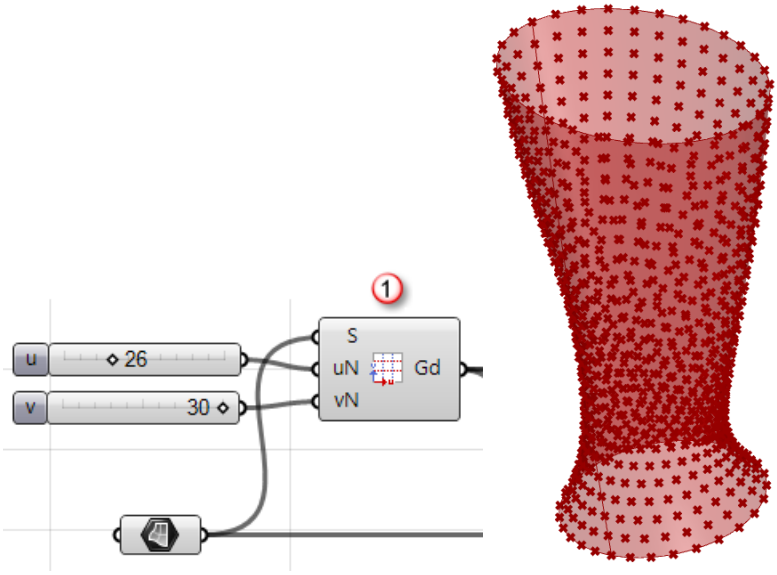


We start the definition with a surface reference component and select the tower surface from Rhino.



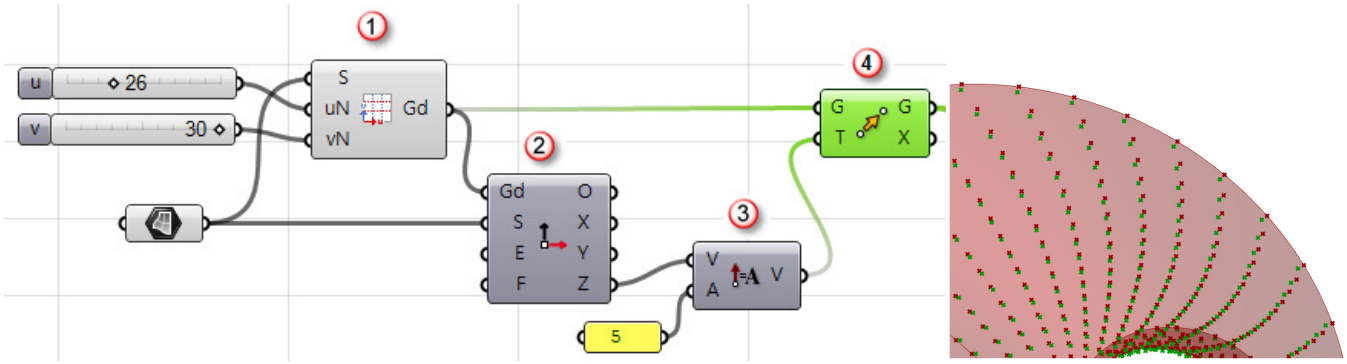
Reference surface parameter in GH (1).

The next step in to feed the surface into a grid by surface PanelingTools component. In this case I chose the grid by domain number.



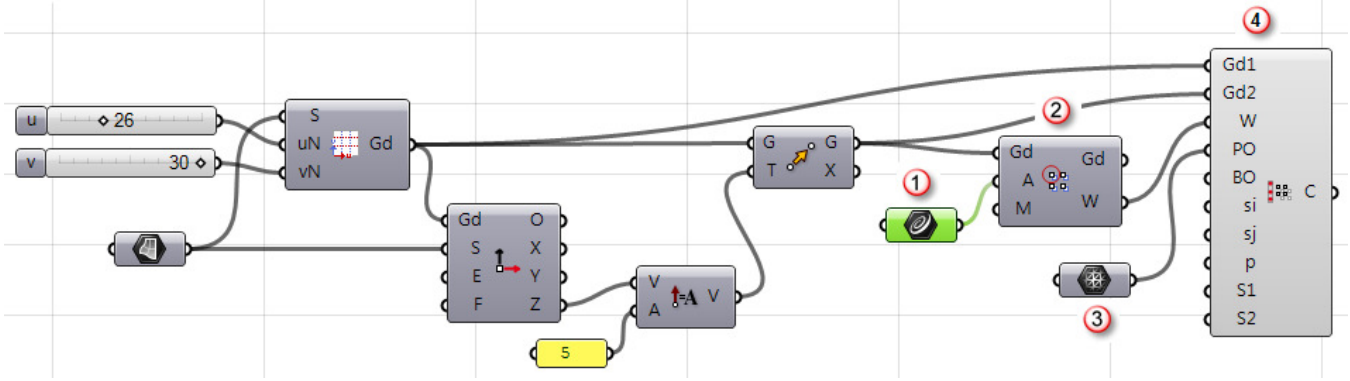
ptDomNum component to generate the grid (1).

Once we have the grid, we need to offset in a direction normal to the surface. To do that, we need to calculate the normal direction at each grid point. ptCoordinate component in PanelingTools takes care of that.



Input grid (1), ptCoordinate component calculates the x, y, z direction of each grid point relative to input surface (2), Specify offset amount (3), Move grid point in normal direction (4).

The last step involves creating attraction field (grid of weights) to feed into the paneling component.



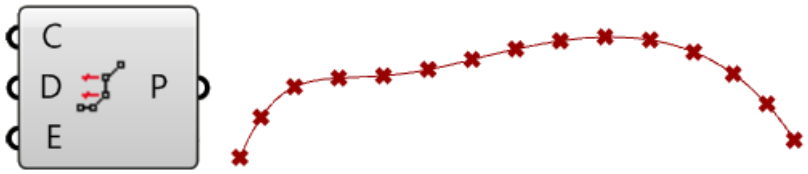
Attractor curve (1), attraction by curve component to calculate weights at each grid point (2), input list of modules (3), distribute the list of components on the grid using attraction values (4).

Components: Curve

These components divide a NURBS curve using various controls.

ptDivideDis

The **ptDivideDis** component calculates divide points on a curve or list of curves by specified distance with an option to add end point(s).



Input

C: Curve(s) to divide.

D: Distance between points.

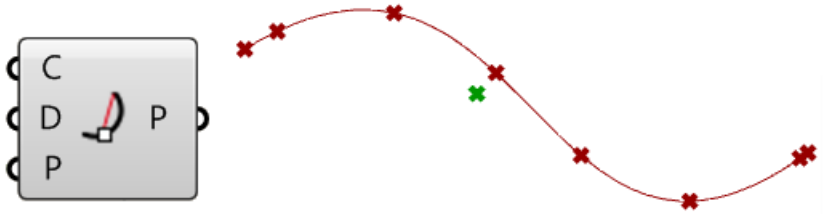
E: Option to add end point if set to "True".

Output

P: List of divide points.

ptDivideDisRef

The **ptDivideDisRef** component divides curves by distance with reference point. Reference point is used to control the location of divide points.



Input

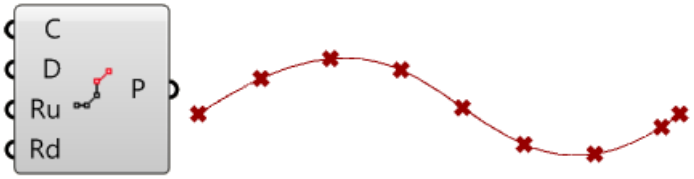
C: Curve(s) to divide.

D: Distance between points.

P: Reference point. Calculate the closest point on-curve to the reference point, then divide the two sides of the curve by distance.

ptDivideLen

The **ptDivideLen** component calculates divide points on a curve or list of curves by specified length on curve with an option to round the distance up or down. If rounded, the curve is divided into equal lengths.



Input

C: Curve(s) to divide.

D: length on curve.

Ru: Round the length up if set to "True".

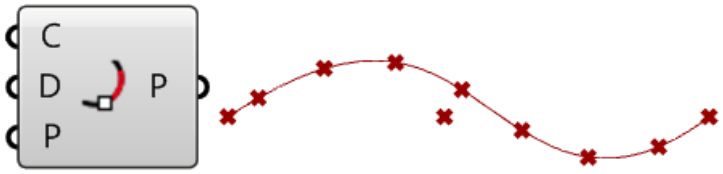
Rd: Round the length down if set to "True".

Output

P: List of divide points.

ptDivideLenRef

The **ptDivideLenRef** component calculates divide points on a curve or list of curves by specified length on curve. Reference point is used to control the location of divide points.



Input

C: Curve(s) to divide.

D: length on curve.

P: Reference point. Calculate the closest point on-curve to the reference point, then divide the two sides of the curve by distance.

Output

P: List of divide points.

ptDivideNum

The **ptDivideNum** component calculates divide points on a curve or list of curves by specified number.

Input

C: Curve(s) to divide.

Number: number of spans. An input of "10" generates "11" points.

Output

P: List of divide points.

ptDivideParam

The **ptDivideParam** component calculates divide points on a curve f list of curves from a list of parameters.

Input

C: Curve(s) to divide.

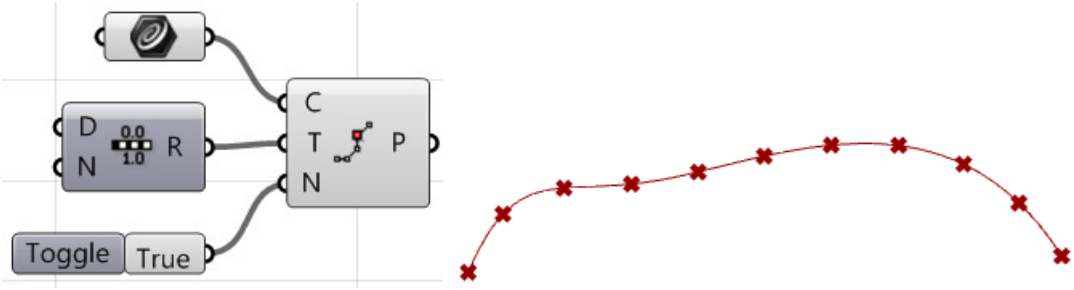
Number: number of spans. An input of "4" generates "5" points.

Output

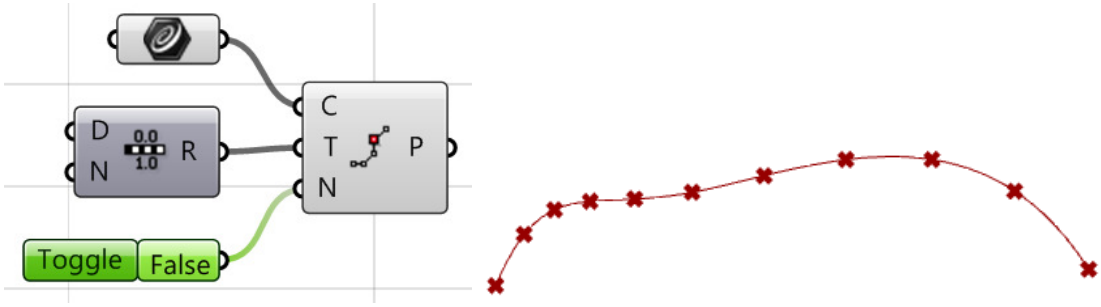
P: List of divide points.

Example

When "N" or normalize is set to true, curve(s) are divided to equal spans if distance between parameter values are equal regardless of the curve parameterization or domain.



When "N" is set to false, curve(s) are divided into equal spans in parameter space, which may not translate to equal spans in 3D space.

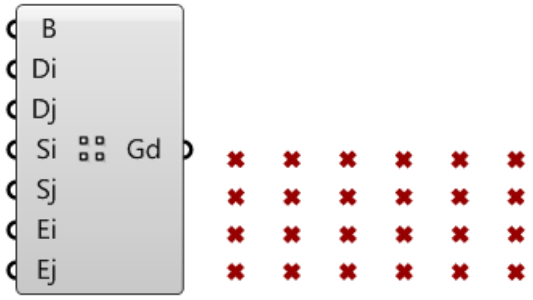


Components: Grid

These components generate paneling grids and organize in a tree structure where branches represent rows of points.

ptPlanar

The **ptPlanar** component creates parallel planar grids on the. The "u" and "v" directions of the grid do not have to be orthogonal.



Input

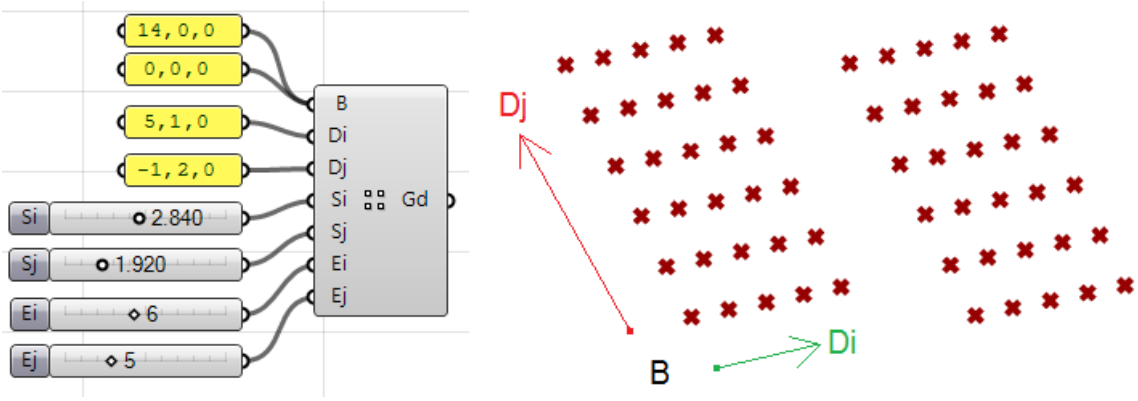
B: base point.
Di: direction of the rows.
Dj: direction of the columns.
Si: distance between rows.
Sj: distance between columns.
Ei: number of rows.
Ej: number of columns.

Output

Gd: grid.

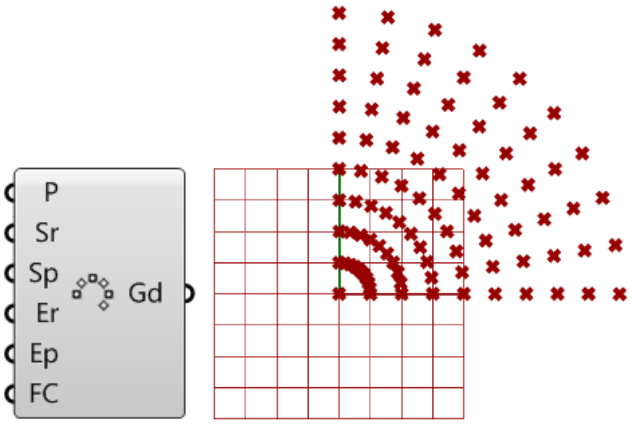
Example

Planar grids can have a user-defined row and column directions. Also you can input multiple base points to create more than one grid represented in one tree structure.



ptPolar2D

The **ptPolar2D** component creates polar planar grids.



Input

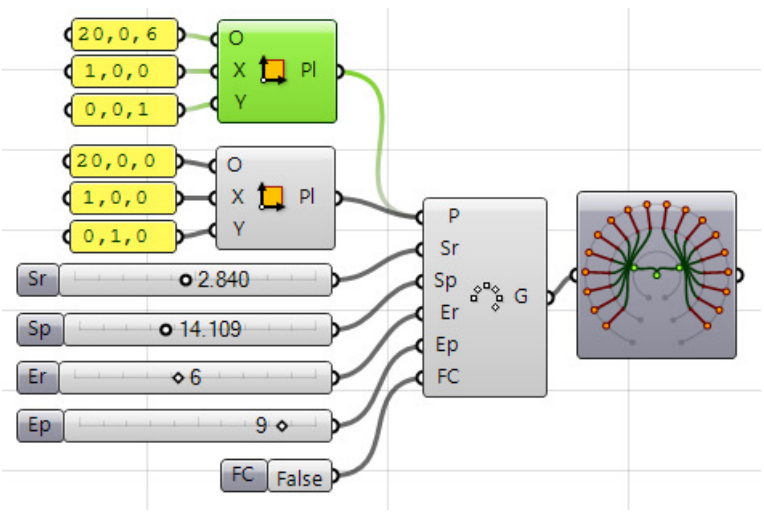
B: base plane.
Sr: distance between points in radial direction.
Sp: angle (in degrees) in polar direction.
Er: number of points in radial direction (number of columns).
Ep: number of points in polar direction (number of rows).
FC: create a full closed circle (ignores angle set in Sp).

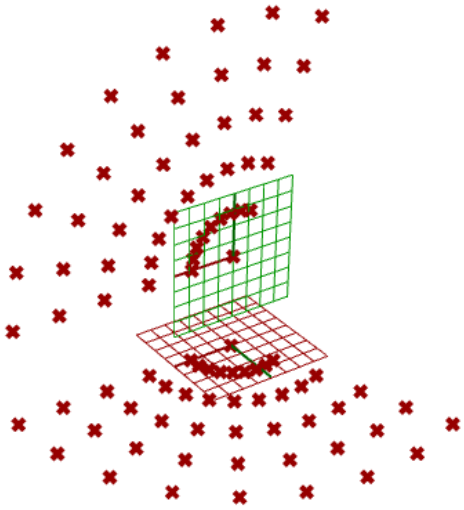
Output

Gd: grid.

Example

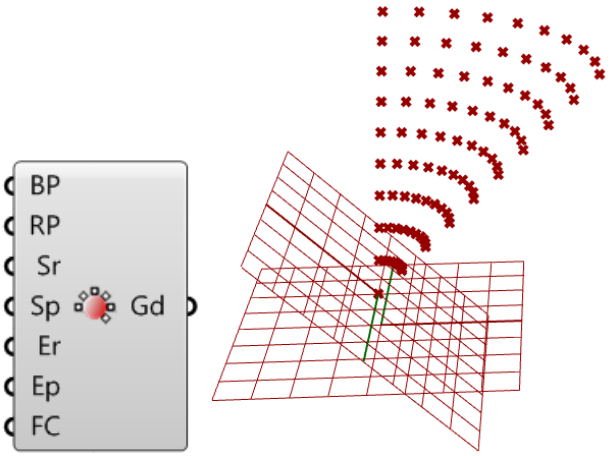
You can define plane direction and also input multiple planes.





ptPolar3D

The **ptPolar3D** component creates polar 3D grids. The component takes two input planes. The first defines the base plane and rotation axis. The second plane defines the grid base point and the first row direction.



Input

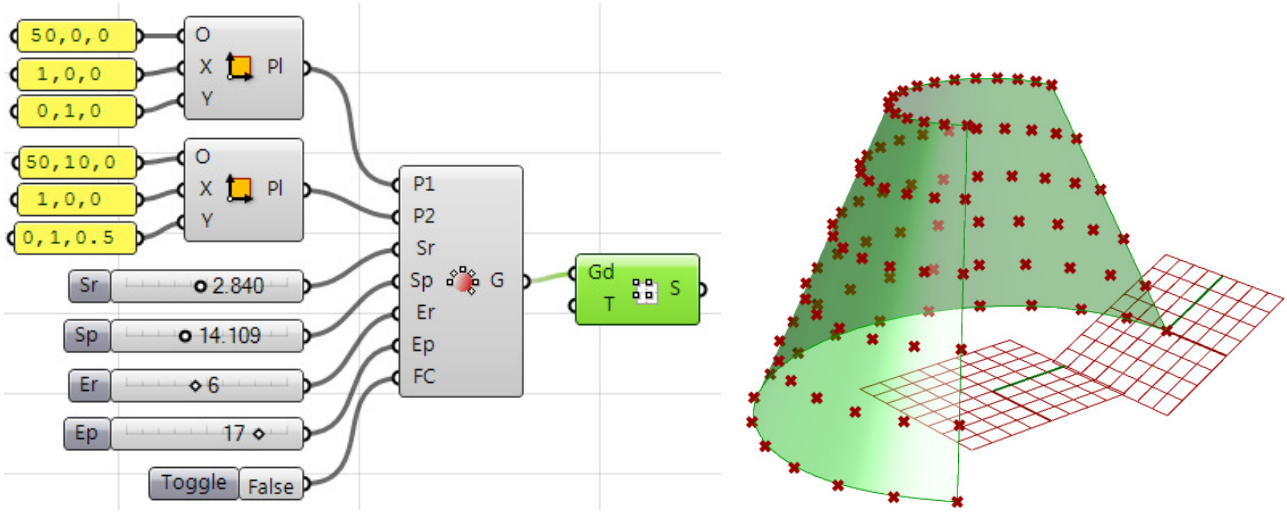
BP: base plane. Defines the base plane and rotation axis.
RP: revolve plane. Defines the grid base point and the first row direction.
Sr: distance between points in radial direction.
Sp: angle (in degrees) in polar direction.
Er: number of points in radial direction (number of columns).
Ep: number of points in polar direction (number of rows).
FC: create a full closed circle (ignores angle set in Sp).

Output

Gd: grid.

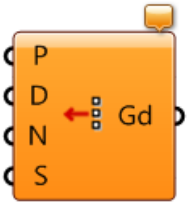
Example

You can define plane direction and also input multiple planes.



ptExtrudePlanar

The **ptExtrudePlanar** component creates extrude grids.



Input

P: list of points (represent first row).

D: extrude direction (represent columns direction).

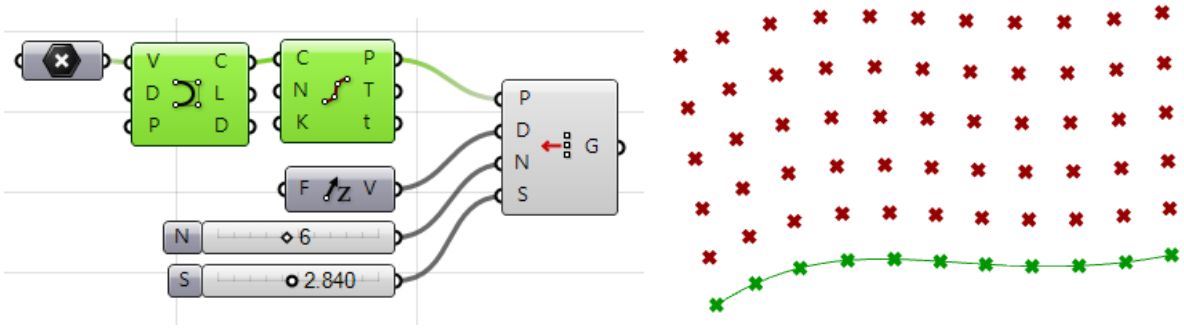
N: number of rows.

S: distance between rows.

Output

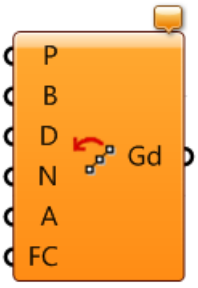
Gd: grid.

Example



ptExtrudePolar

The **ptExtrudePolar** component creates polar 3D grids. The component takes two input planes. The first defines the base plane and rotation axis. The second plane defines the grid base point and the first row direction.



Input

P: list of points (represent first row).

B: base point of the rotation axis.

D: rotation axis.

N: number of rows.

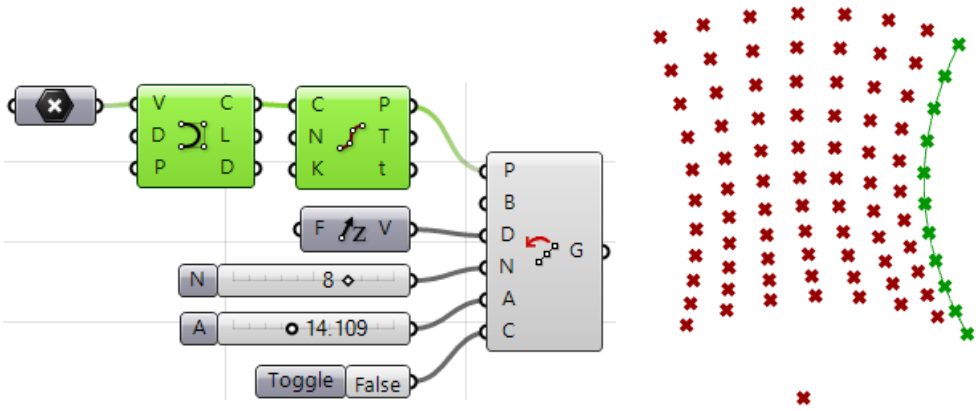
A: angle between rows.

FC: create a full closed circle (ignores angle set in A).

Output

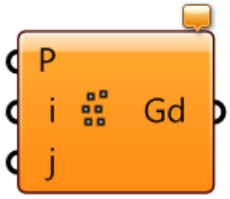
Gd: grid.

Example



ptCompose

The **ptCompose** component helps create grids from scratch. Its takes a list of points and two integer lists to define the (i.j) location of each point in the grid



Input

P: list of points.

i: list of i indices of corresponding points.

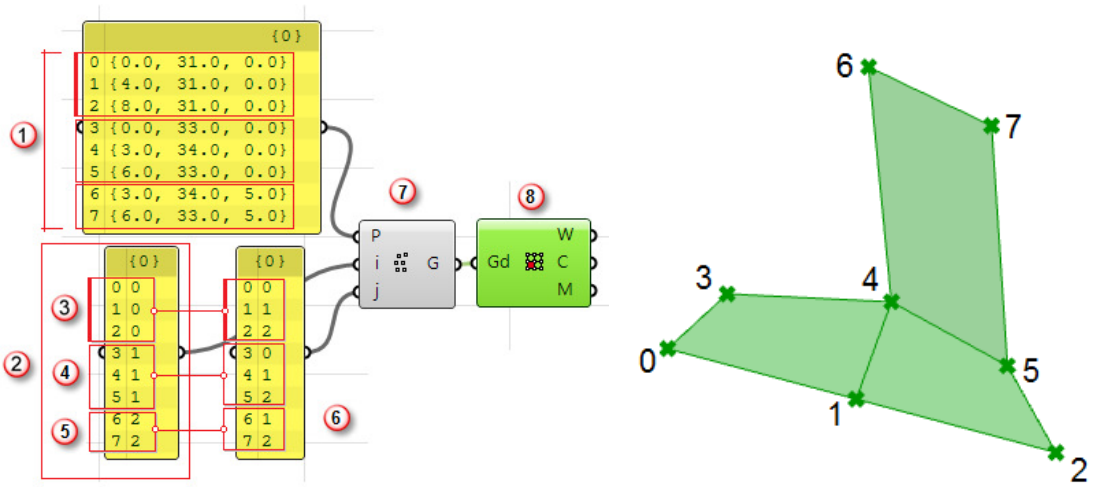
j: list of j indices of corresponding points.

Output

Gd: grid.

Example

The following example shows how to organize a list of 8 points into a grid. The user needs to define the index of each point in the grid (row, column location). For example the first three points make the first row in the grid. Notice that their row location (i-input) is set to “0”, and the column locations (j-input) are “0”, “1” & “2”.



List of points (1), row indices of the points (2), first row (3), second row (4), third row (5), column indices of the points (6), component to compose the grid (7), component to generate cells data (8).

ptComposeNum

The **ptComposeNum** component assumes that the resulting grid will have equal number of points in each row (rectangular grid). It takes a list of points and number of rows and outputs the grid. If the number of input points is not divisible by the input number of rows, then the remainder will make the last row of the grid.

Input

P: list of points.

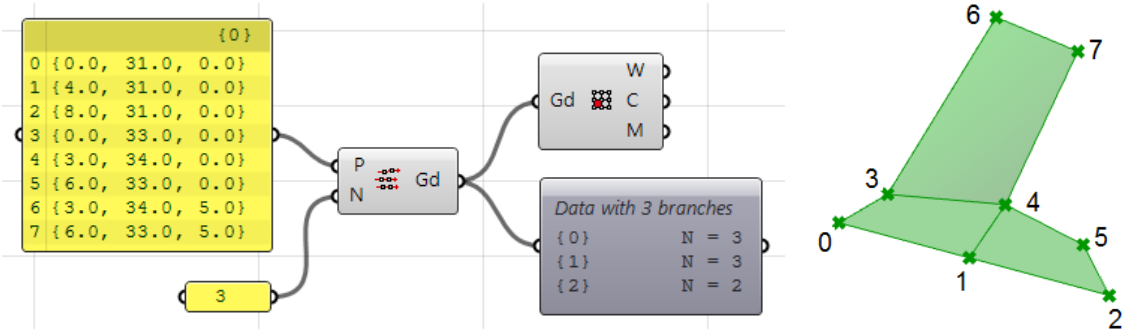
N: number of rows.

Output

Gd: grid.

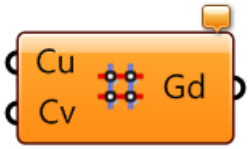
Example

The following example shows how to organize a list of 8 points into a grid that has 3 rows. It assumes that points are listed in order. That means that the first input point becomes the base point (or the first point of the first row). The second input point becomes the second point in the first row, and so on until all rows are filled. Note that the last row contain fewer points.



ptUVCrvs

The **ptUVCrvs** component generates a grid from intersecting two sets of curves. First set define the row direction of the grid and the second is the column direction.



Input

Cu: list of curves in the "u" (row) direction.

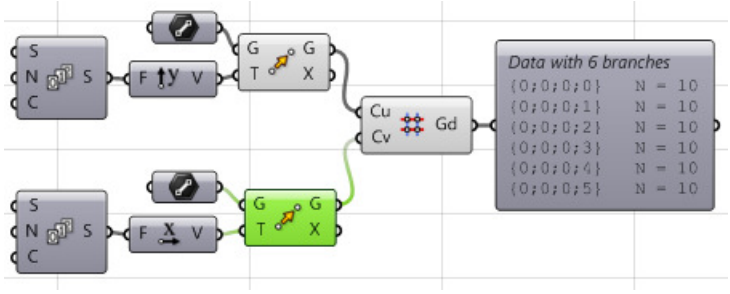
Cv: list of curves in the "v" (column) direction.

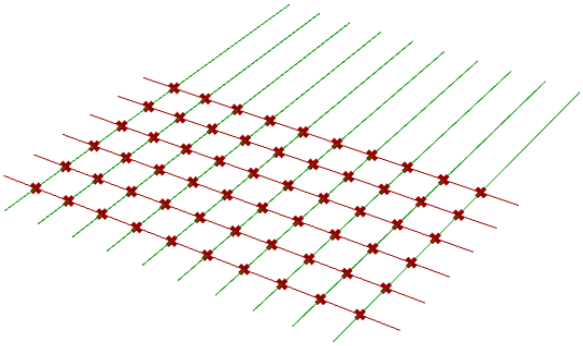
Output

Gd: grid.

Example

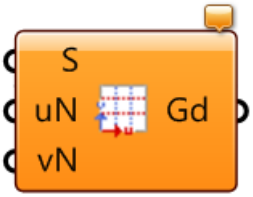
In this example, we generate two sets of lines in x and y directions and create the grid from the intersections. The generated grid has 6 rows and 10 columns. Intersecting curves do not have to be planar or linear.





ptSrfDomNum

The **ptSrfDomNum** component generates a grid using a surface and follows the surface domain direction.



Input

S: list of curves in the "u" (row) direction.

uN: number of spans in the "u" direction.

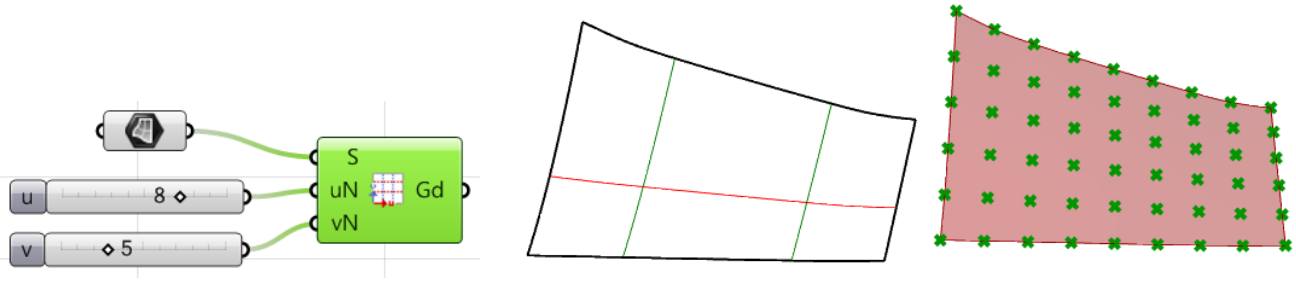
vN: number of spans in the "v" direction.

Output

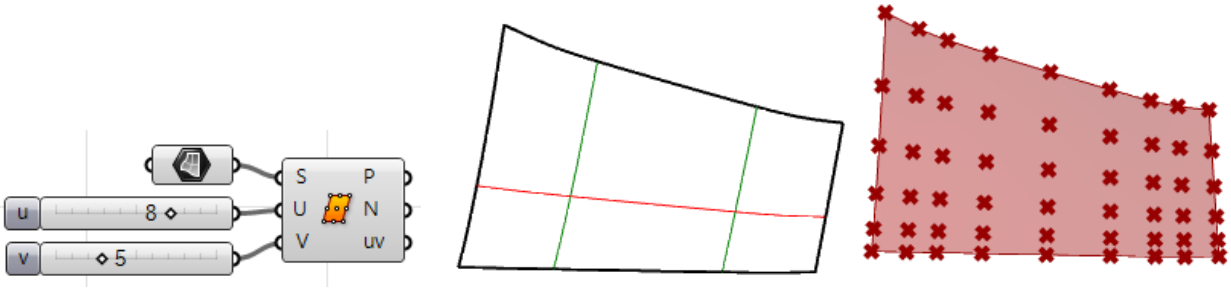
Gd: grid.

Example

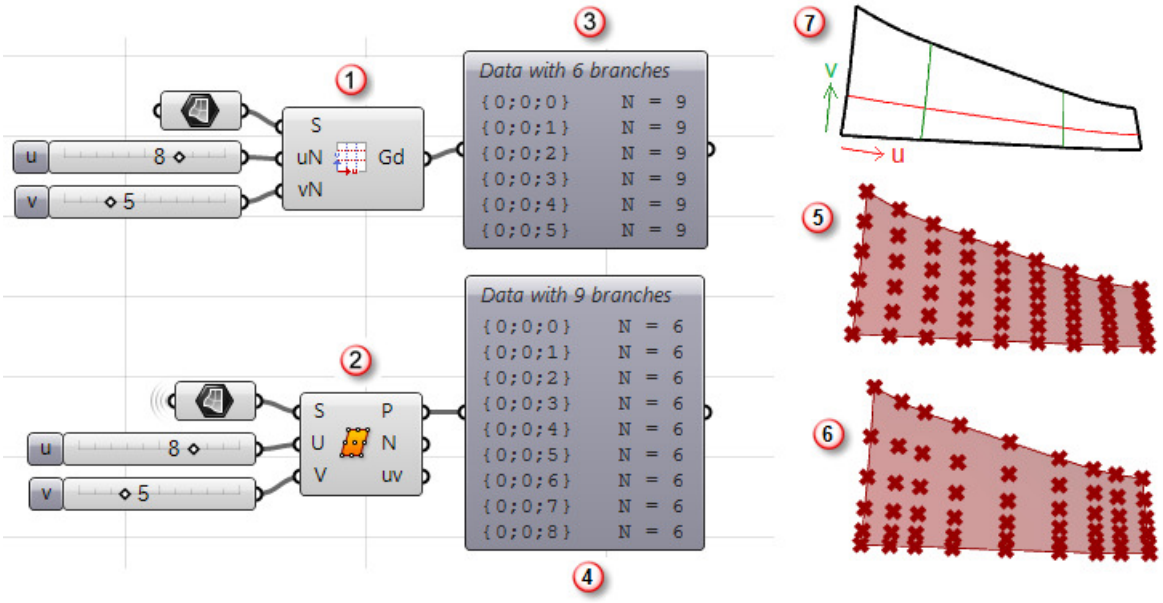
The surface in the image below is divided into 6 branches, each has 9 points. While points follow the iso direction of the surface, it is not affected by the surface parameterization and tries to divide by somewhat equal distances.



If you divide the same surface using Grasshopper surface divide (SDivide) component, you might have your points spaced unevenly. This is because the component actually divides the domain (in parameter space) into equal distances, but that does not necessarily translate into equal spacing in 3D space.



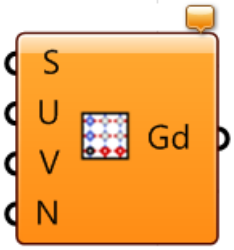
Another difference between the ptSrfDomNum component and GH SDive component is the way output is organized. The output tree from ptSrfDomNum arranges rows data in branches. In SDivide, it is arranged by columns.



Divide surface domain in PT (ptSrfDomNum) (1), Divide surface domain in GH (SDivide) (2), data structure of points from ptSrfDomNum (3), Data structure of points from SDivide (4), output grid from ptSrfDomNum (5), output grid from SDivide (6), input surface and its u & v directions (7).

ptSrfParam

The **ptSrfParam** component give you control over how to divide the surface domain using parameter space.



Input

S: Input surface.

U: parameter list (0-1) that divides the domain in u direction.

V: parameter list (0-1) that divides the domain in v direction.

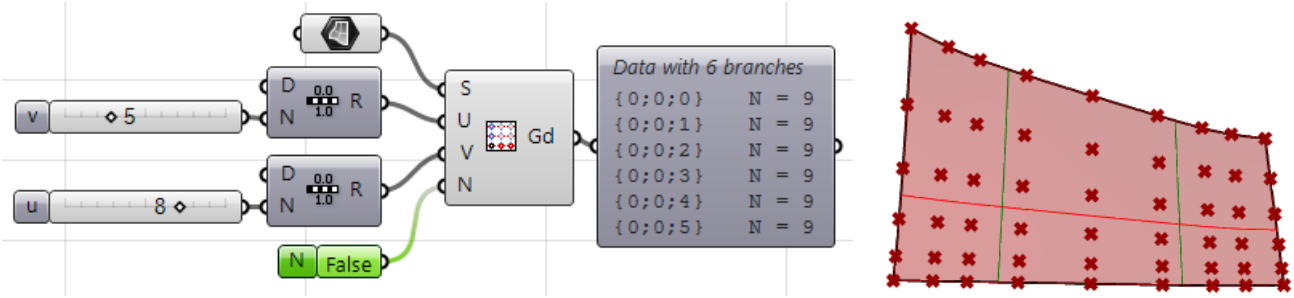
N: option to use normalized distance to achieve more even distribution.

Output

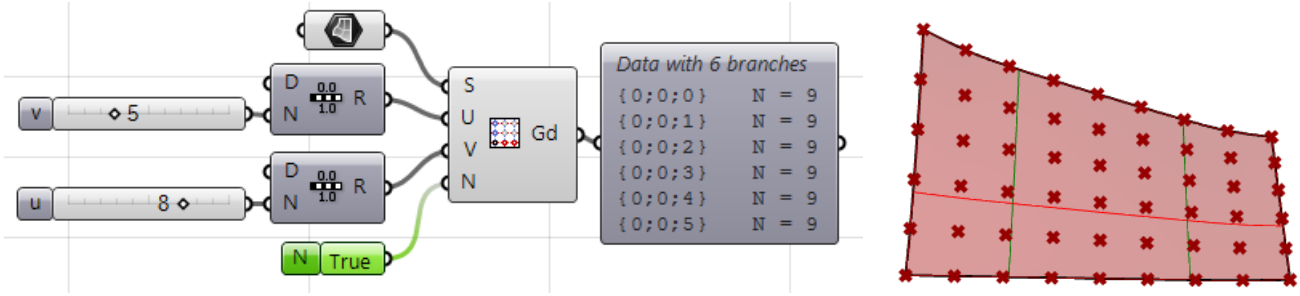
Gd: grid.

Example

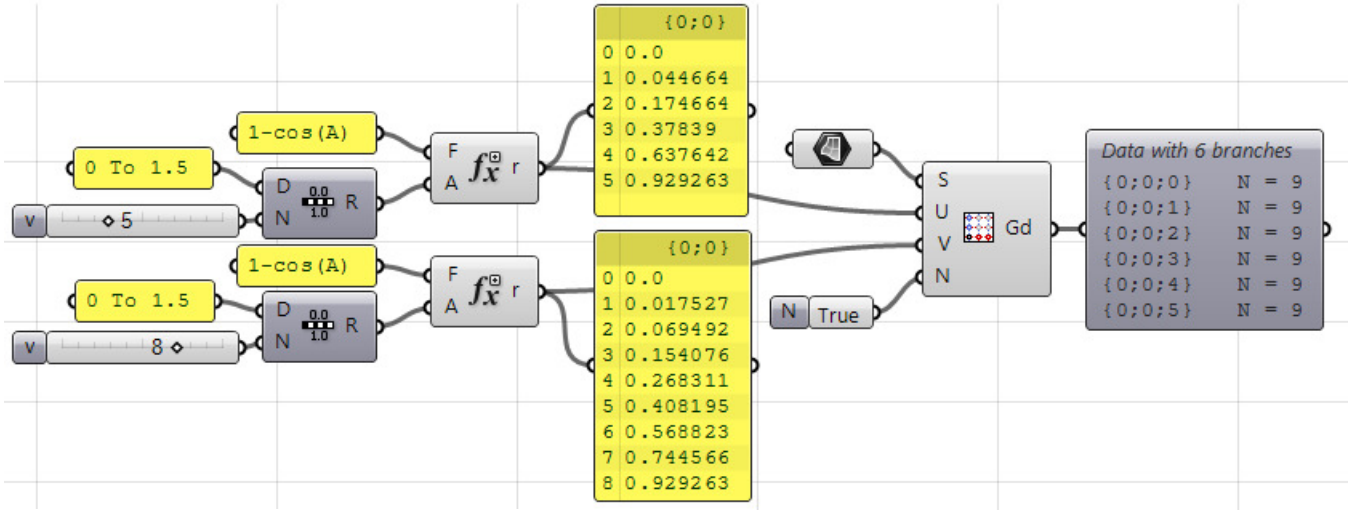
This component gives a full control over how the domain is divided. You can set the "N" to "False" if you desire the result to follow the surface parameterization. This achieves similar result to GH SDivide component explained in the previous section, except that output data is organized with rows as branches, while SDivide organize it with columns as branches.

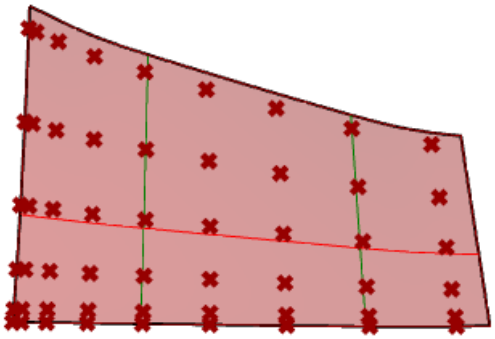


You can also achieve a result similar to that of ptSrfDomNum explained in the previous section with relatively even distances. You can do that when you set "N" to "True".



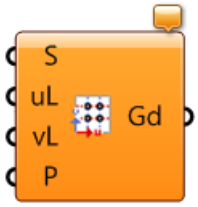
The greatest value in this component is when you have variable distances that you like to divide against. In this case, I set the "N" to "True" to get more predictable outcome that is not affected by how the input surface is parameterized. This is how the definition and outcome looks.





ptSrfDomLen

The **ptSrfDomLen** component give you control over how to divide the surface domain using parameter space.



Input

S: Input surface.

uL: length on surface in u direction.

vL: length on surface in v direction.

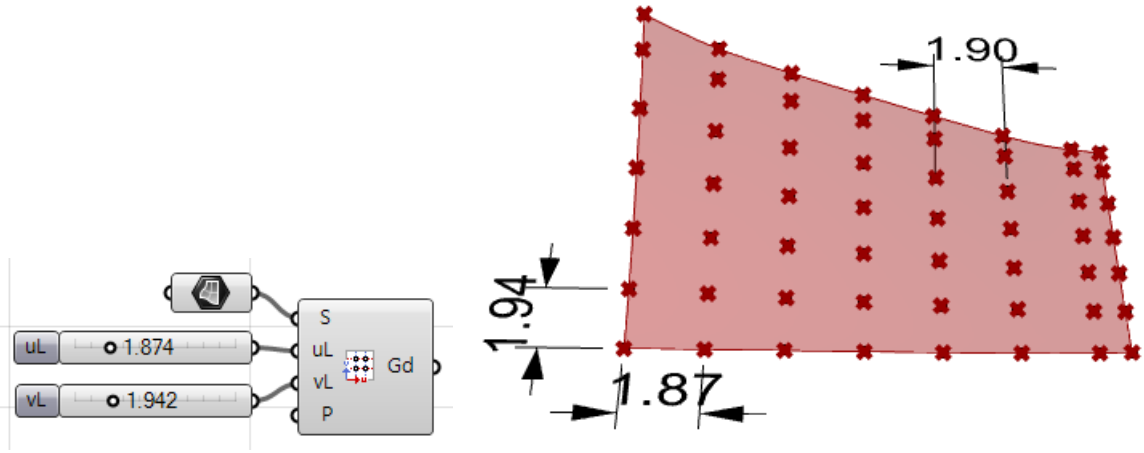
P: reference point.

Output

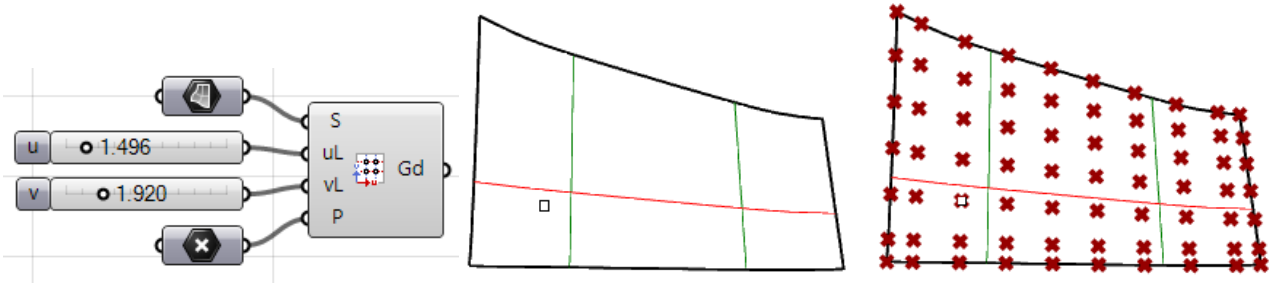
Gd: grid.

Example

For a planar or extrude surface, the distances are exact across the surface, but if the surface is curved in two directions, then distances will vary. The base u and v isos are divided exactly by the specified distance. The rest of the surface will probably have other distances depending on the shape of the surface. It is simple impossible to follow surface domain and yet maintain constant distances on surface for free-form surfaces.

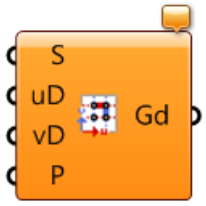


It is also possible to control the location of the grid on the surface by setting the "P" input parameter to a point on the surface. It is worth noting that the exact divide distance is achieved on the isocurves that go through that reference point.



ptSrfDomChord

The **ptSrfDomChord** component is very similar to the component discussed above (ptSrfDomLen). The only difference is that distance between points is set to the 3D direct distance rather than length on curve. This component has similar options with the ability to set reference point to control the location of the grid



Input

S: Input surface.

uD: distance in u direction.

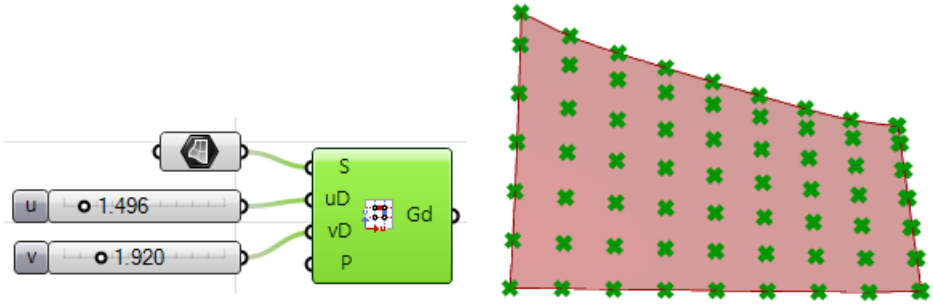
vV: distance in v direction.

P: reference point.

Output

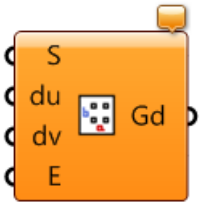
Gd: grid.

Example



ptSrfDis

The **ptSrfDis** component attempts to divide a surface by equal distances. The underlying algorithm is calculation intensive and it can take time to calculate. There is also a requirement to have the distances in u and v direction to be multiples of each other. The distance between points is maintained throughout the grid. In some cases and because of the surface curvature, the grid cannot cover the whole surface.



Input

S: Input surface.

du: distance in the u direction.

dv: distance in the v direction.

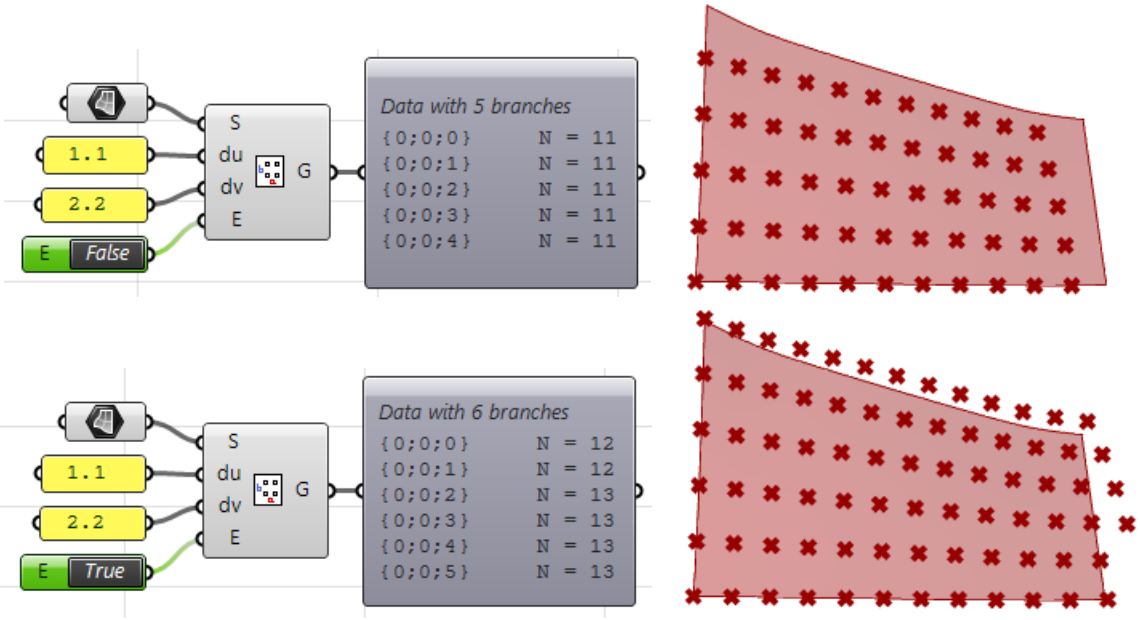
E: calculate on extended surface to achieve better coverage.

Output

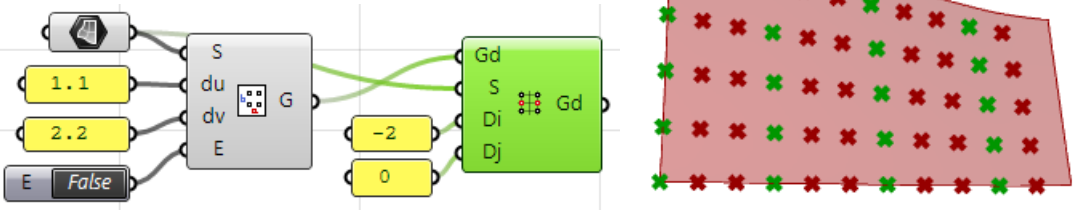
Gd: grid.

Example

The following example shows the difference when setting "E" input parameter to "False" or "True". Notice also that the v distance (dv) is a multiple of the u distance (du). If you need distances like "2.2" and "3.3", you can set the grid to be "2.2" and "1.1" and then decrease grid density (in the 1.1 direction) using ptDense component.



Notice that the v distance (dv) is a multiple of the u distance (du). If you need distances like "2.2" and "3.3", you can set the grid to be "2.2" and "1.1" and then decrease grid density (in the 1.1 direction) using ptDense component.

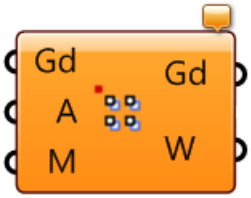


Components: Grid Attractors

These components adjust points locations in a grid based on various attractors.

ptPtsAtts

The **ptPtsAtts** component calculates new points' locations based on an attractor point or points. It also calculates the corresponding grid of weights, or influence. Attraction can be magnified by increasing the "M" or magnitude value in the input. The edge points of the grid always move within edge boundary keeping the outline of the grid intact.



Input

Gd: input grid.

A: attractor points.

M: magnitude or degree of influence. A negative input cause points to attract away from the attractor.

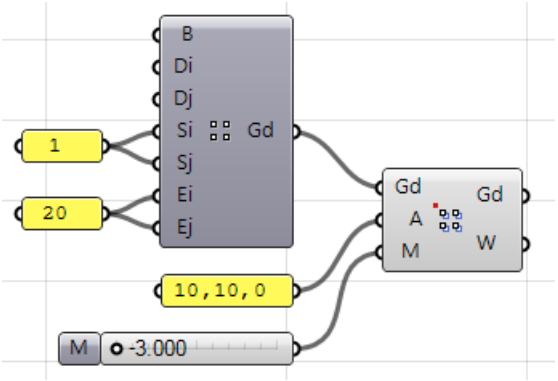
Output

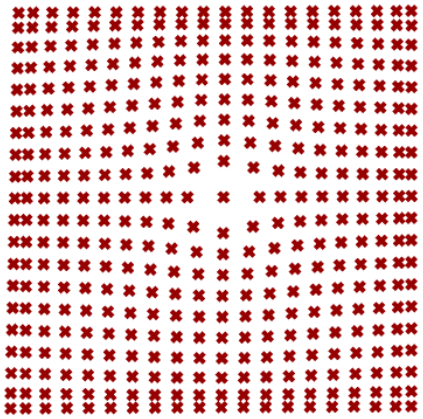
Gd: result grid after attraction.

W: grid of weights (0-1).

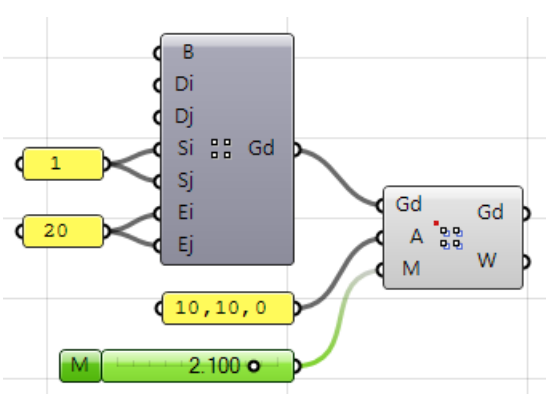
Example

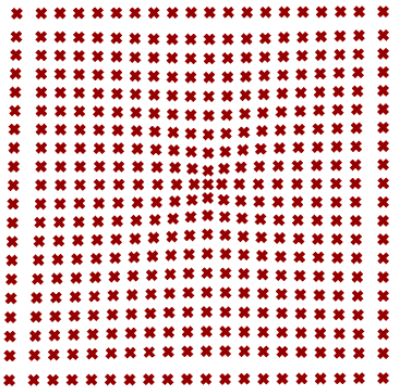
The following example generates a rectangular grid then assigns a point in the middle of the grid to be the attractor. The magnitude (M) is set to a negative value and hence attraction moves awat from the center.





If magnitude is set to a positive value, then attraction gravitate towards the attractors.





ptCrvsAtts

The **ptCrvsAtts** component calculates new grid points' locations based on attractor curve(s). It also calculates the corresponding grid of weights, or influence. Attraction can be magnified by increasing the "M" or magnitude value in the input. The edge points of the grid always move within edge boundary keeping the outline of the grid intact.

Input

Gd: input grid.

A: attractor curves.

M: magnitude or degree of influence. A negative input cause points to attract away from the attractor.

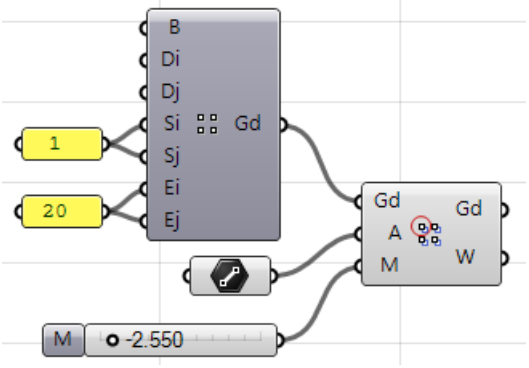
Output

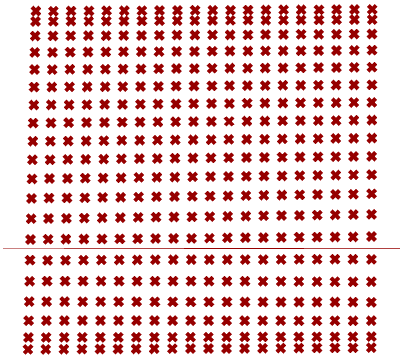
Gd: result grid after attraction.

W: grid of weights (0-1).

Example

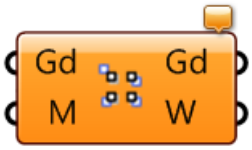
The following example generates a rectangular grid then defines a center line to be the attractor.





ptRandAtts

The **ptRandAtts** component calculates new grid points' locations randomly. Grid points will not collide or overlap.



Input

Gd: input grid.

M: magnitude or degree of influence.

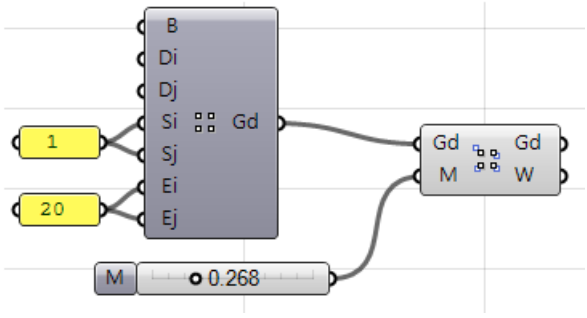
Output

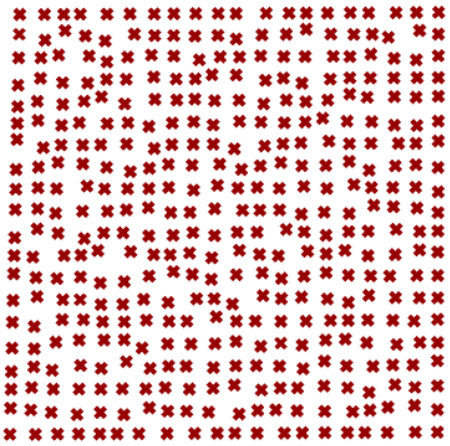
Gd: result grid after attraction.

W: grid of weights (0-1).

Example

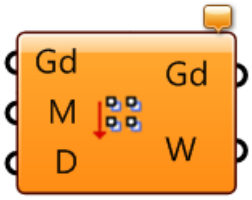
The following example generates a rectangular grid then shuffles the grid randomly. Notice that points are only shuffled within their local region and the shuffling did not produce overlapped regions.





ptDirAtts

The **ptDirAtts** component calculates the attraction field based on relative angles between the normal direction of each point (on the grid surface) and that of a user-defined direction. This can be useful if you need to modify the grid based on the sun direction or the viewing angle for example.



Input

Gd: input grid.

M: magnitude or degree of influence.

D: direction vector.

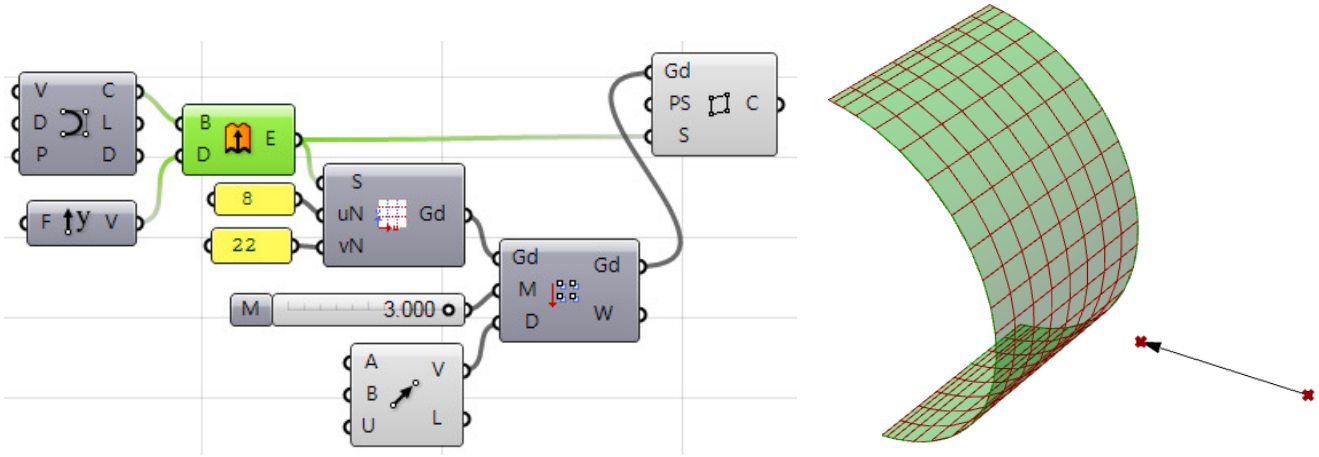
Output

Gd: result grid after attraction.

W: grid of weights (0-1).

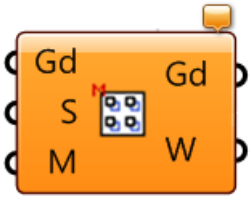
Example

This example shows how points are attracted using a direction curve. The definition first creates a surface then divides it by number to create the original grid. It then uses **ptDirAtt** component to create a more dense spacing towards the edges. This is because the normal direction of the points at the edge forms bigger angle with the input direction compared to points towards the middle of the surface.



ptMean

The **ptMean** component shuffles grid of points on a surface using surface Mean curvature.



Input

Gd: input grid.

S: surface.

M: magnitude or degree of influence.

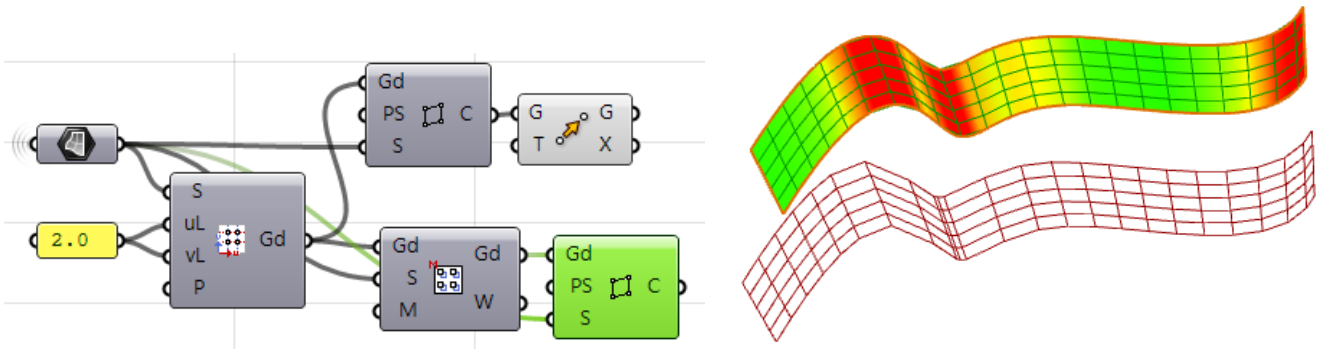
Output

Gd: result grid after attraction.

W: grid of weights (0-1).

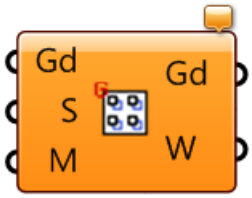
Example

In the top image, the surface is divided evenly using the underlying surface domain. The bottom part of the definition uses surface Mean curvature to attract towards the highest curvature. The



ptGauss

The **ptGauss** shuffles grid of points on a surface using surface Gaussian curvature.



Input

Gd: input grid.

S: surface.

M: magnitude or degree of influence.

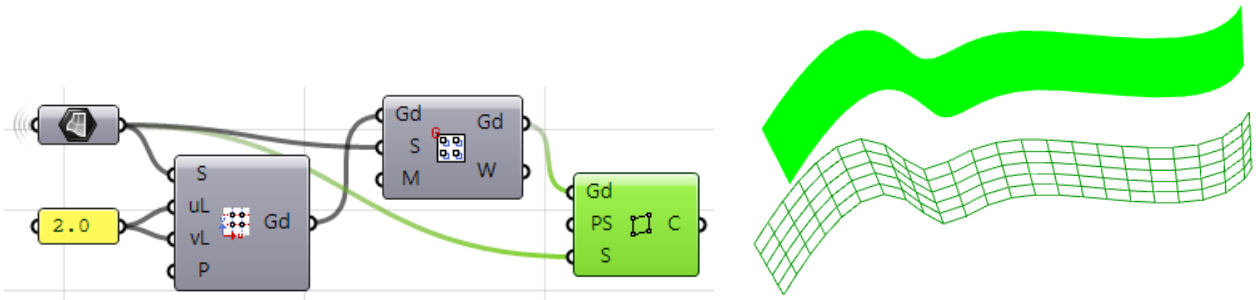
Output

Gd: result grid after attraction.

W: grid of weights (0-1).

Example

Using the same surface from the previous section, the analysis graph shows us a uniform curvature and this is why the attraction by Gaussian does not change the grid.



ptWeight

The **ptWeight** component allows the user to feed a weight field or grid and control attraction directly. This is useful when you have defined the amount of attraction you want for each grid point and want points to be shuffled accordingly.

Input

Gd: input grid.

W: grid of weights (0-1).

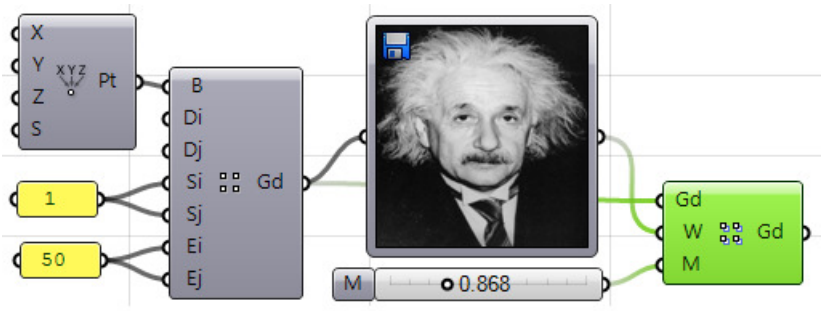
M: magnitude or degree of influence.

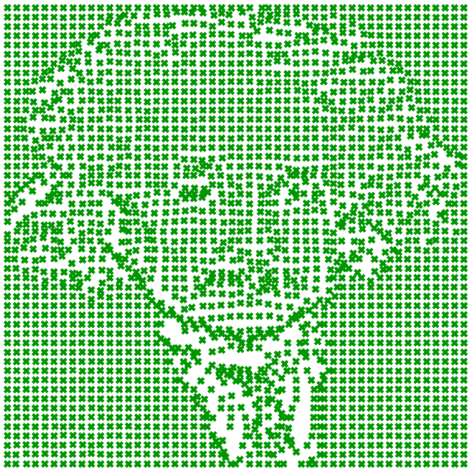
Output

Gd: result grid after attraction.

Example

The following example uses a bitmap image to generate weights and use that to shuffle points.





Components: Grid Utility

These components generate paneling grids and organize in a tree structure where branches represent rows of points.

ptCenter

The **ptCernter** component extracts the center grid of another input grid.

Input

Gd: input grid.

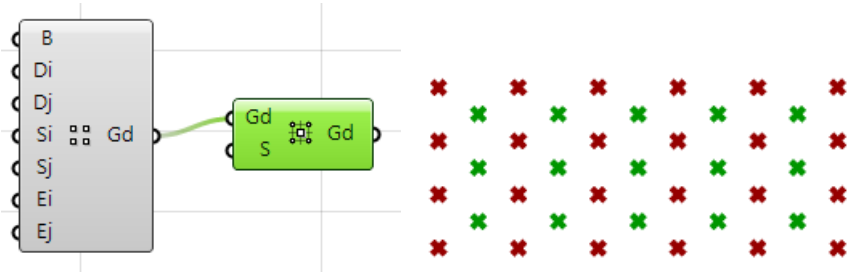
S: base surface [optional].

Output

Gd: center grid.

Example

Use a rectangular grid as an input. The output is a grid of the centers of the cells.



ptClean

The **ptClean** component removes null rows and null columns in the input grid.

Input

Gd: input grid.

Output

Gd: output grid.

ptToDiamond

The **ptToDiamond** component converts rectangular grids to diamond grids.

Input

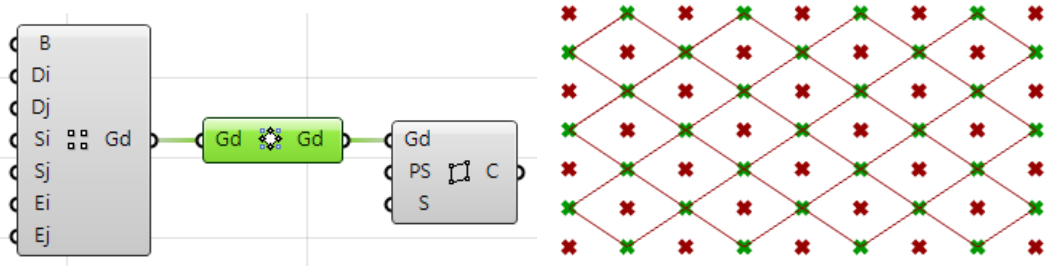
Gd: input grid.

Output

Gd: diamond grid.

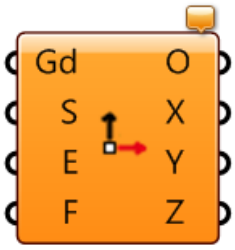
Example

Use a rectangular grid as an input. The output is a diamond grid.



ptCoordinate

The **ptCoordinate** Calculates grid (or cells) coordinates which is the origin, x, y and z vectors for each grid/cell point. The x and y vectors are not normalized and their length equals the distance between grid points..



Input

Gd: input grid.

S: input surface [optional].

E: calculate coordinates for end points [optional – set to "True" by default].

F: flip the z direction [optional – set to "False" by default].

Output

O: origin points.

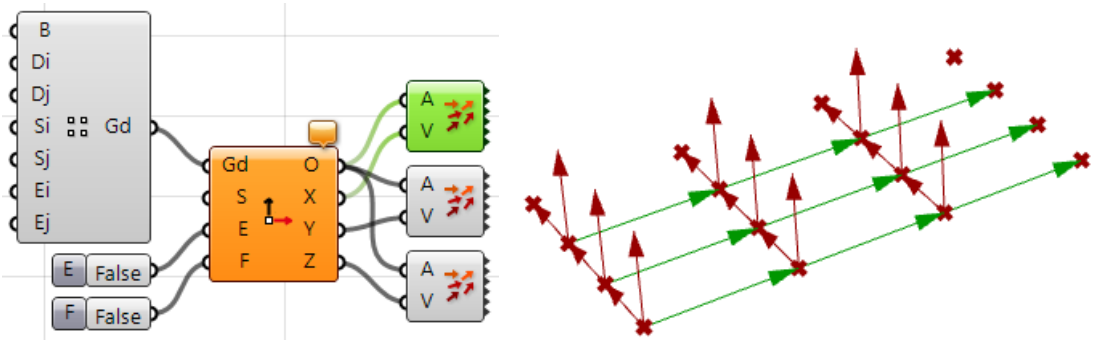
X: x vectors.

Y: y vectors.

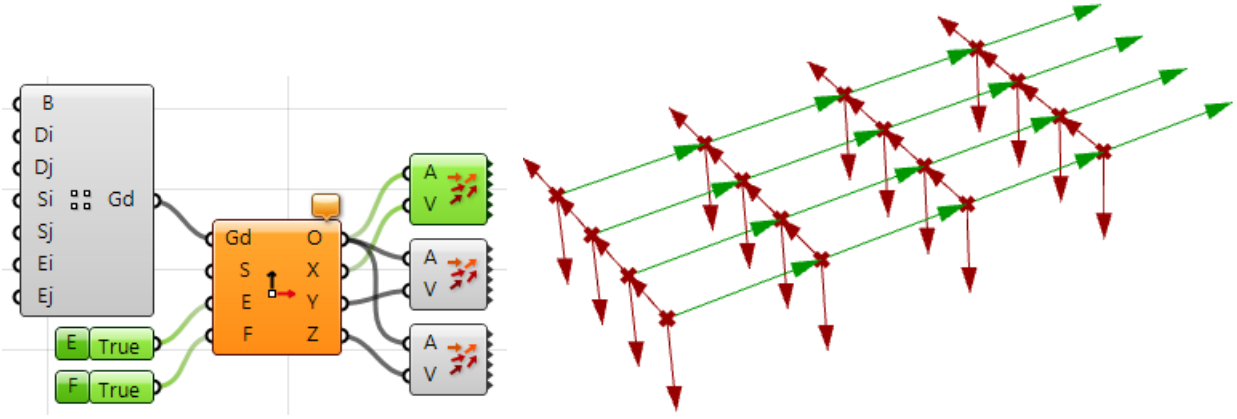
Z: z vectors.

Example

The example shows the coordinates of each cell in the grid. If the "S" (surface) input parameter is not available, the component will calculate the surface using the input grid.

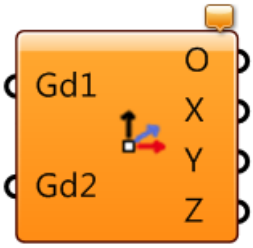


If the "E" input is set to "True", then the coordinates for end points is calculated. Also, there is an option to flip the z direction as in the image.



ptCoordinate3D

The **ptCoordinate3D** calculates the box cell coordinates between 2 grids which includes the origin, x, y and z vectors for each grid/cell point. The x, y and z vectors are not normalized and their length equals the distance between cell points.



Input

Gd1: first input grid.

Gd2: second input grid.

Output

O: origin points.

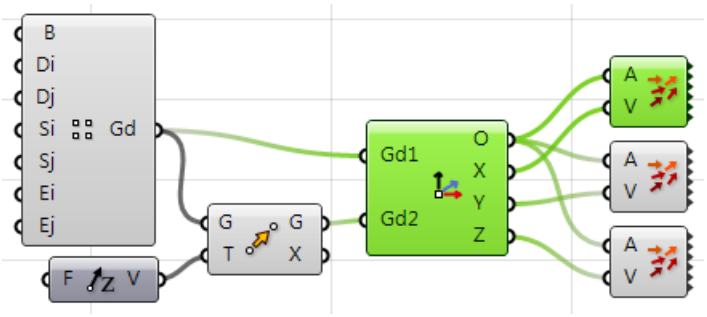
X: x vectors.

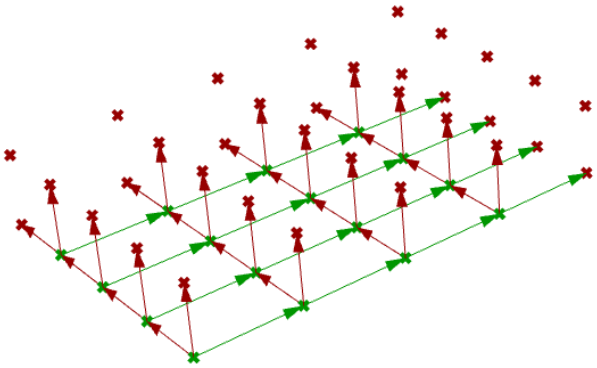
Y: y vectors.

Z: z vectors.

Example

The example creates a rectangular grid then copy in the "z" direction. The ptCoordinate3D component takes the 2 grids as input and calculates x, y and z vectors for each cell.





ptCol

The **ptCol** component help extract columns from the grid.

Input

Gd1: input grid.

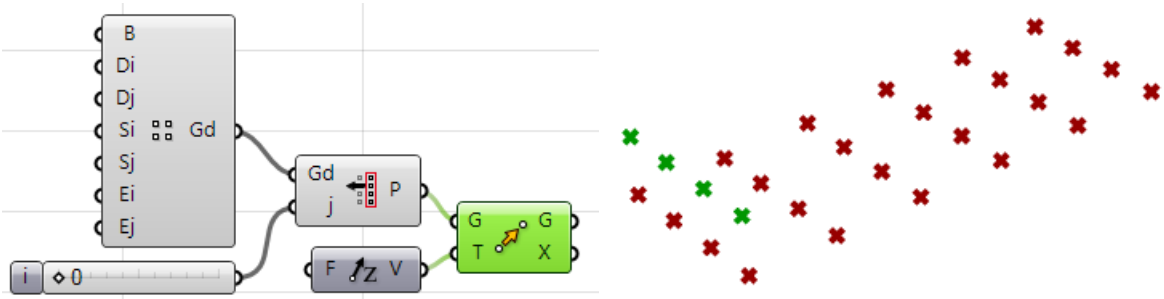
i: column index.

Output

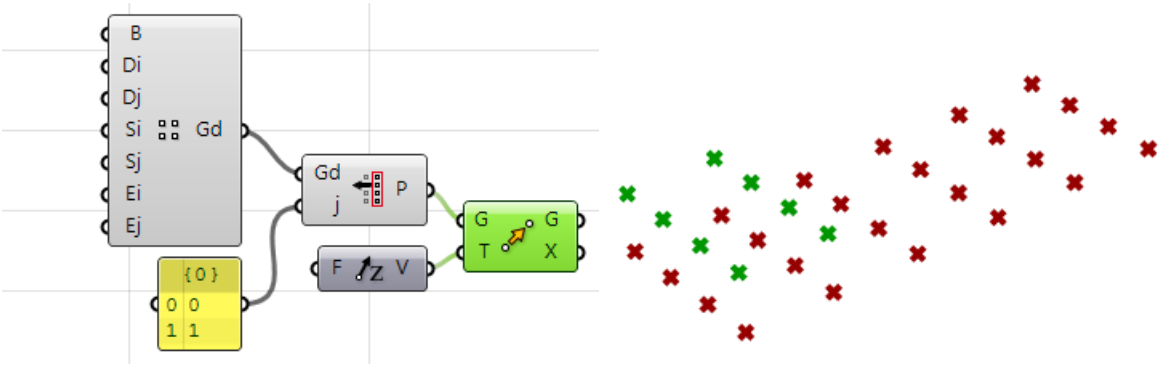
P: list or grid of points.

Example

This example extracts the first column of the grid then moves it in the z direction.



It is also possible to input a list of indices to extract multiple columns in the form of a grid as in the following.



PtIndices

The **ptIndices** takes as an input a grid of points and output 2 grids of integers representing the i and j indices of the points. This component can be used to tag points as in the example below.

Input

Gd: input grid.

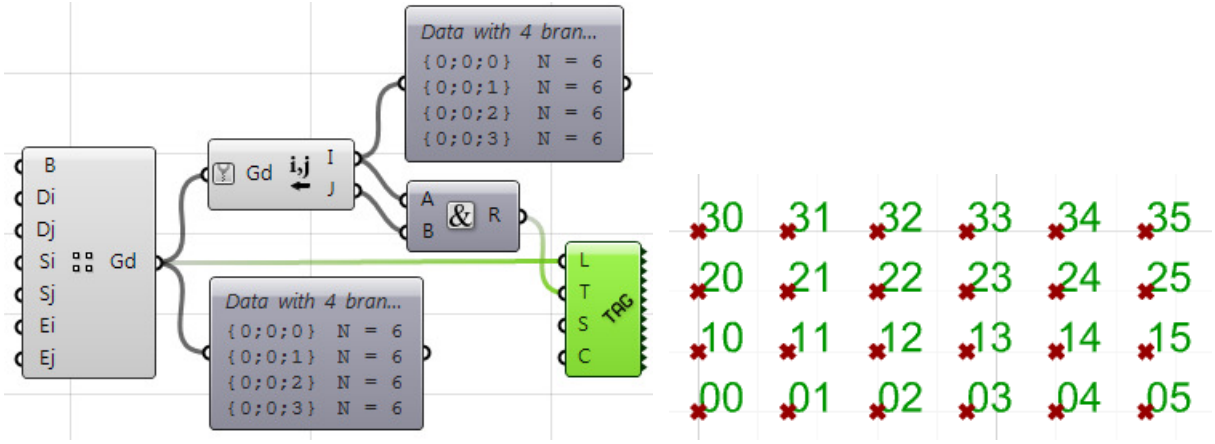
Output

I: grid of i indices.

J: grid of j indices.

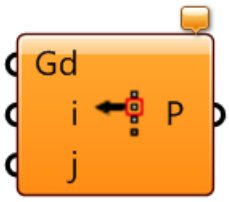
Example

The example uses ptIndices component to get the i,j indices of the points in a grid, then use the string concatenate component in GH to tag the points with their indices. You can notice in the definition, that the structure of the grid is the same as the indices.



PtItem

The **ptItem** extracts a point or list of points in a grid given their "i" and "j" indices.



Input

Gd: input grid.

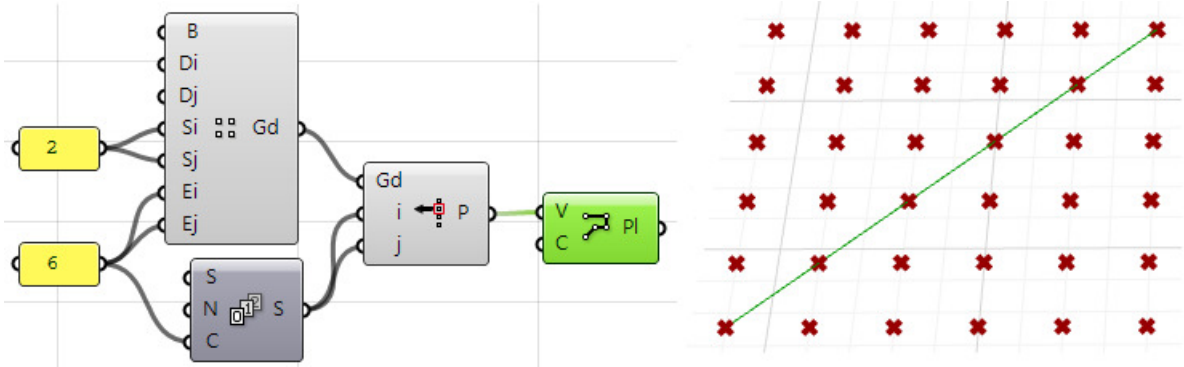
Output

i: item(s) i index.

j: item(s) j index.

Example

The example uses ptItem component to extract diagonal points of a square grid then draw a polyline through them.



PtRow

The **ptRow** extracts a row of points in a grid given the "i" index.

Input

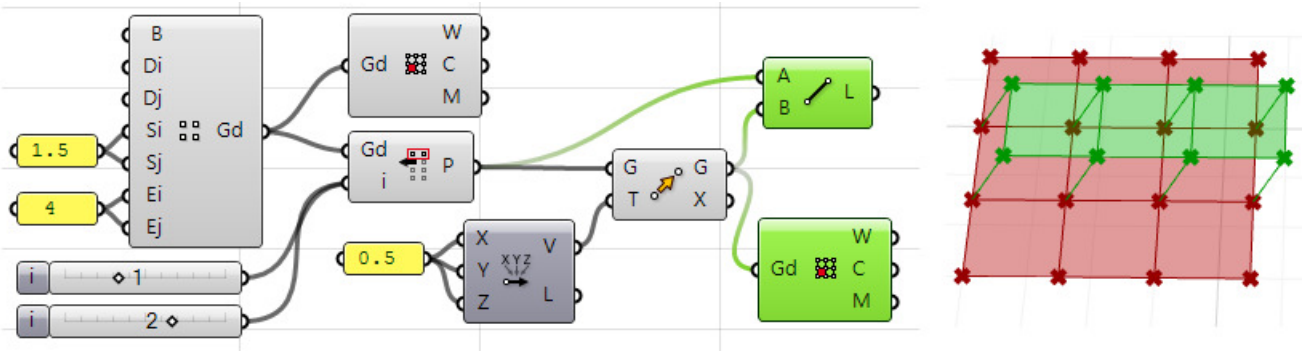
Gd: input grid.

Output

i: row index.

Example

The example extracts the second and third rows of the grid then move these points diagonally, connect to original rows with lines and generate cells out of original and extracted grids.



PtFlatten

The **ptFlatten** flattens a grid to a linear list of cells. This component helps reorganize the grid so that the order of cells is linear and the user has more control mapping a list of modules to that grid.

Input

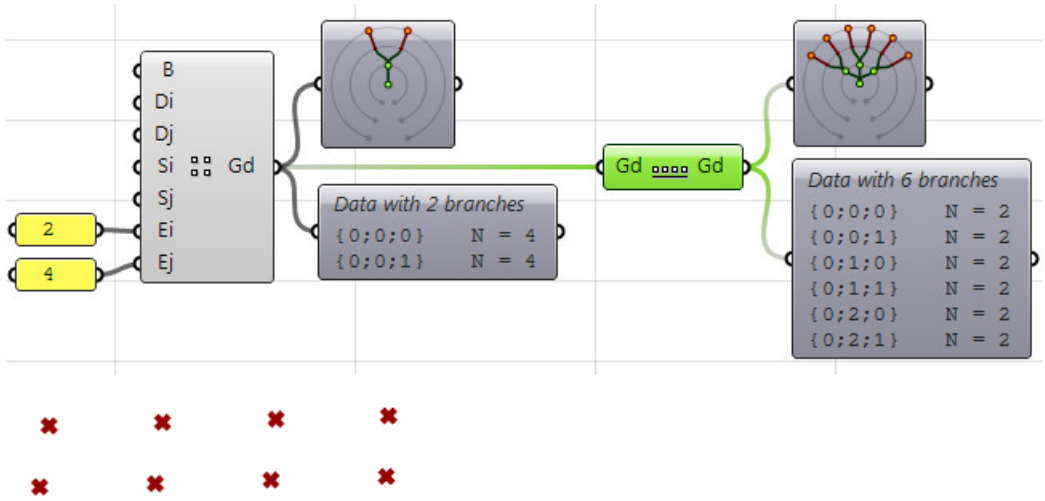
Gd: input grid.

Output

Gd: output grid of cells.

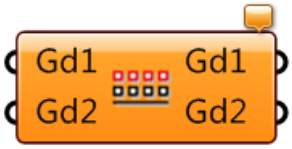
Example

The example shows how the grid is reformatted when put through the flatten component. The grid has 2 rows and 4 columns so the initial tree has 2 branches with 4 elements in each branch. Flattened grid creates sub-trees equal to the number of cells in the grid with each sub-trees holding 4 elements organized in two rows and two columns.



PtFlatten3D

The **ptFlatten3D** flattens two bounding grids into a linear list of bounding cells. This component helps reorganize the grids so that the order of cells is linear and the user has more control mapping a list of modules to grid 3D cells.



Input

Gd1: first input grid.

Gd2: second input grid.

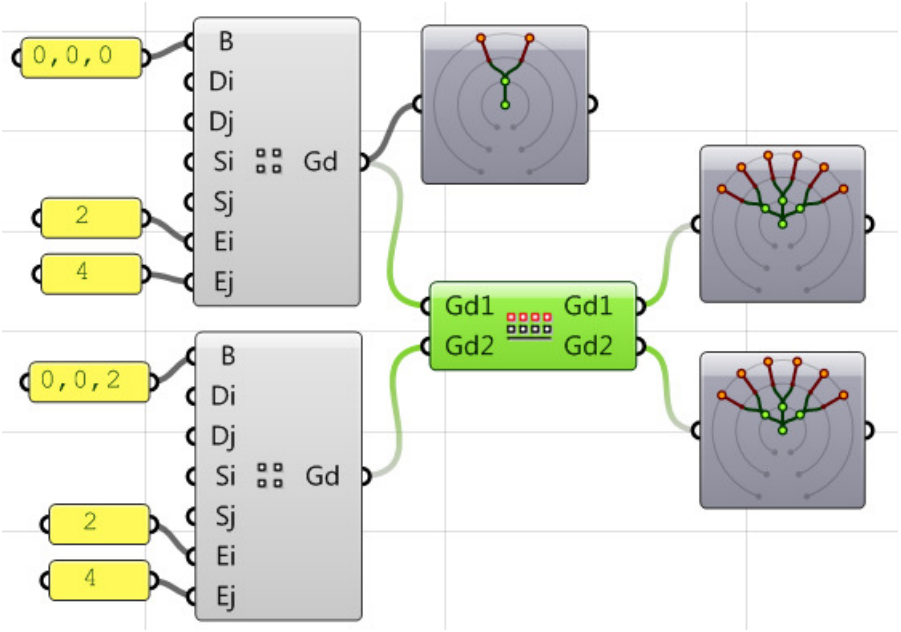
Output

Gd1: flattened first grid.

Gd2: flattened second grid.

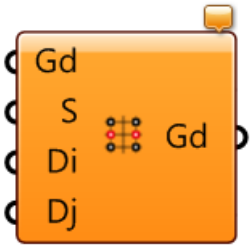
Example

The example shows how the grids are reformatted when put through the flatten-3D component. The input grids have 2 rows and 4 columns so the initial tree has 2 branches with 4 elements in each branch. Flattened grids create sub-trees equal to the number of cells in each grid with sub-trees holding 4 elements organized in two rows and two columns.



PtFDense

The **ptDense** changes grid density and either increase or decrease it. Input surface is optional to make sure added grid points lay on the surface.



Input

Gd: input grid.

S: grid surface.

Di: change is row density. This can positive, negative or zero.

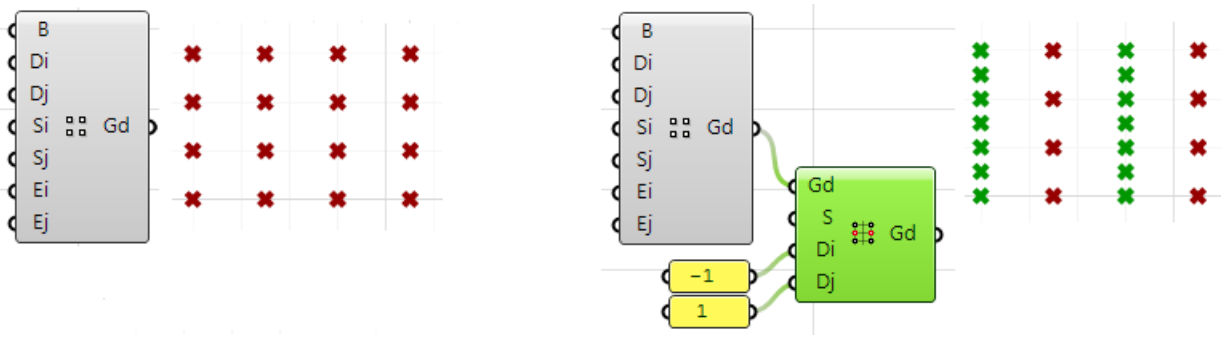
Dj: change is column density. This can positive, negative or zero.

Output

Gd1: output grid.

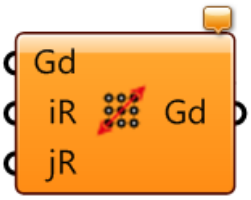
Example

You can increase or decrease the density of a grid. In the example Di is set to "-1" and that removes every other element in each row. Setting Dj=1, inserts an additional grid point in between each two column elements. The illustration shows the input grid before changing the density on the left and after on the right



PtDir

The **ptDir** helps reverse the grid row and column directions.



Input

Gd: input grid.

iR: reverse row direction when set to true.

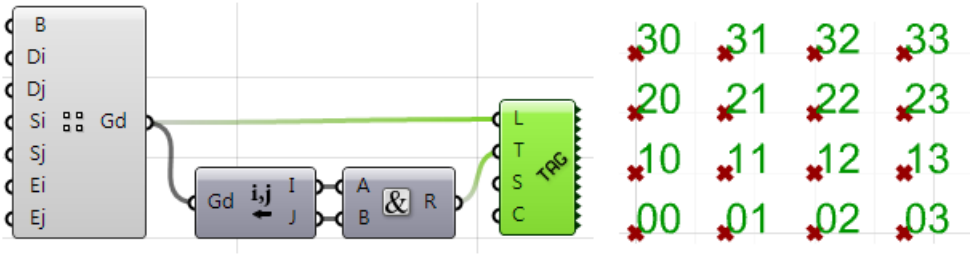
jR: reverse column direction when set to true.

Output

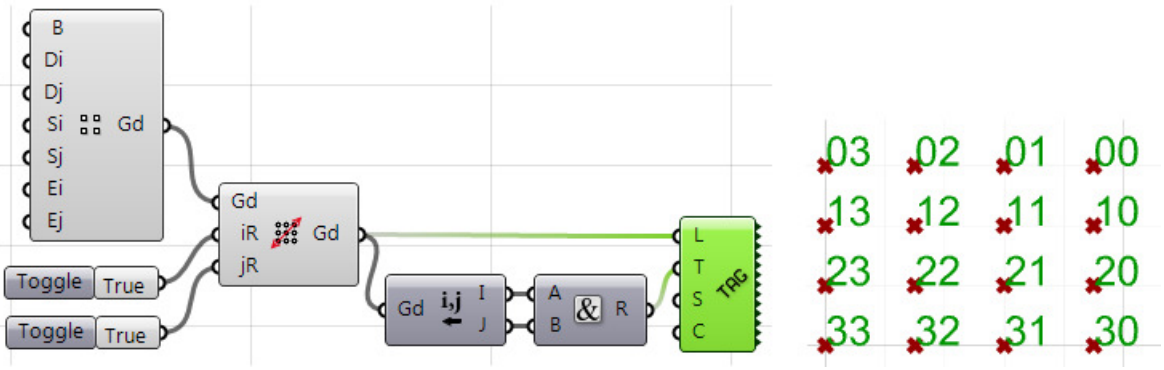
Gd: reversed grid.

Example

For example, the following rectangular grid is tagged to show the row-col location of each grid point. Notice that the base point (00) is in the lower left point.

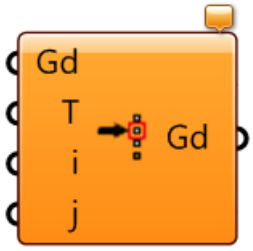


If we reverse the row and column directions, then the base point (00) becomes the top right point.



PtReplace

The **ptReplace** is used to replace one or more points in a grid.



Input

Gd: input grid.

T: list of one or more points.

i: corresponding row location of the point to be replaced.

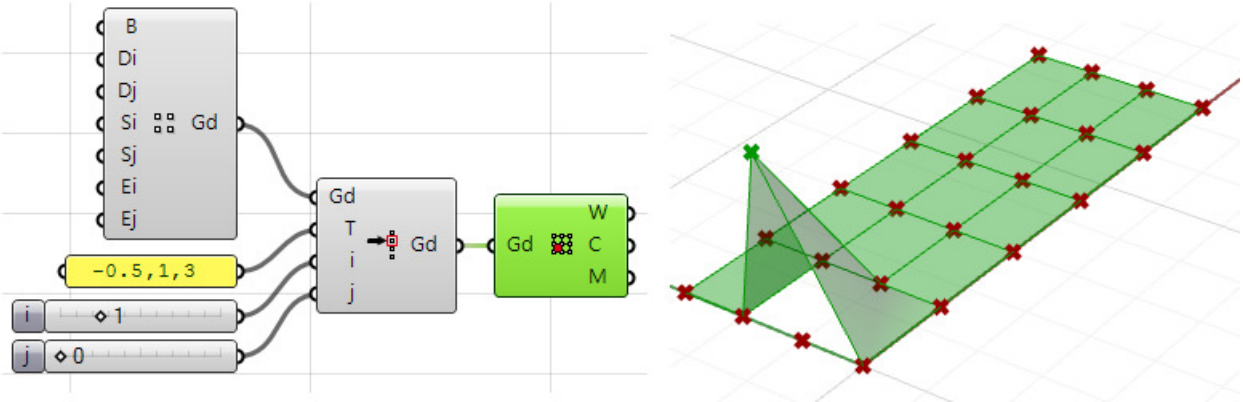
j: corresponding column location of the point to be replaced.

Output

Gd: output grid.

Example

This example shows how to replace one element in the grid. I added a paneling component (ptCellulate) component to show the mesh of the new grid.



PtSquare

The **ptSquare** turns a jagged grid of points into a rectangular grid filling the missing points with NULL points.

Input

Gd: input grid.

Output

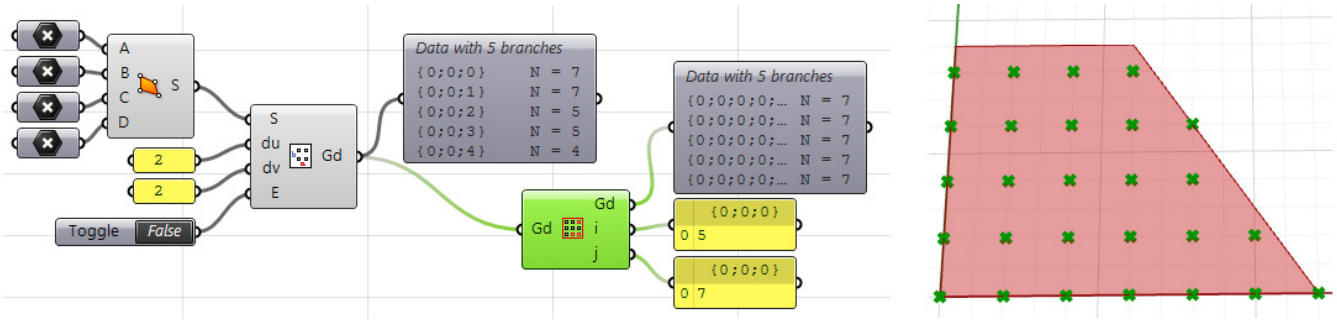
Gd: squared grid.

i: number of rows.

j: number of columns.

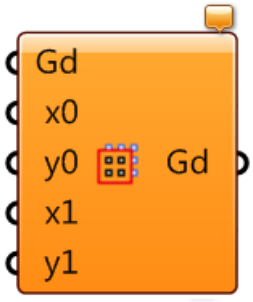
Example

This example shows how to replace one element in the grid. I added a paneling component (ptCellulate) component to show the mesh of the new grid.



PtSubGrid

The **ptSubGrid** helps extract part of the grid.



Input

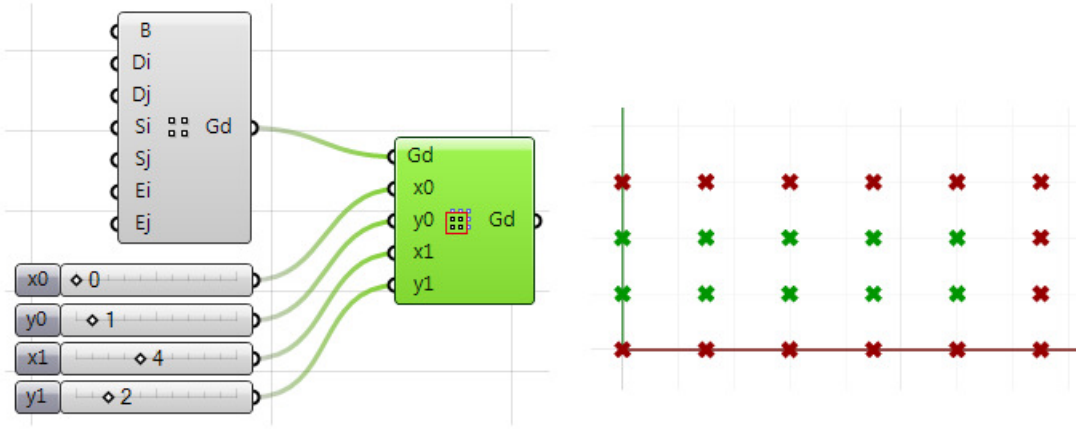
Gd: input grid.
X0: min x index for the sub grid.
Y0: min y.
X1: max x.
Y1: max y.

Output

Gd: sub grid.

Example

This example shows how extract a sub grid dynamically using sliders.



PtGridSrf

The **ptGridSrf** creates a NURBS surface out of a given grid. The surface can be set to either use grid points as surface control points or attempts to generate a surface that goes through grid points.The former suaually yields successfuk result more often.

Input

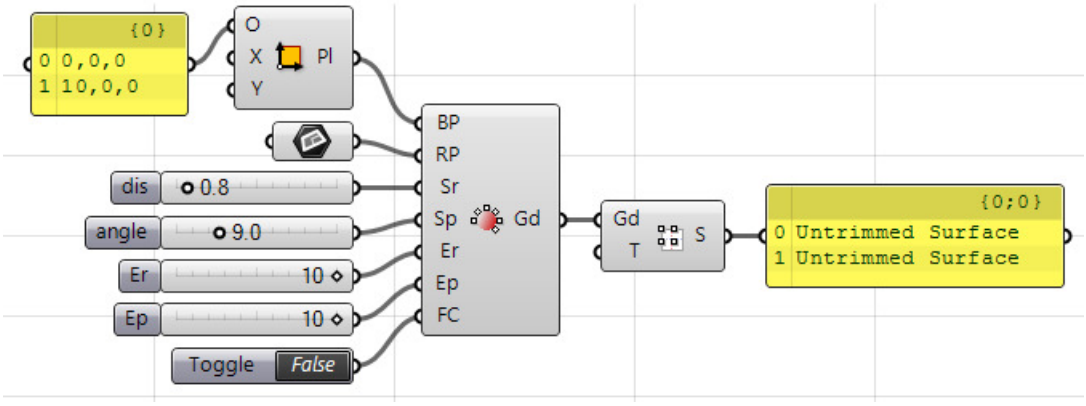
Gd: input grid.
T: when set to true, attempt to generate a surface that goes through the grid.

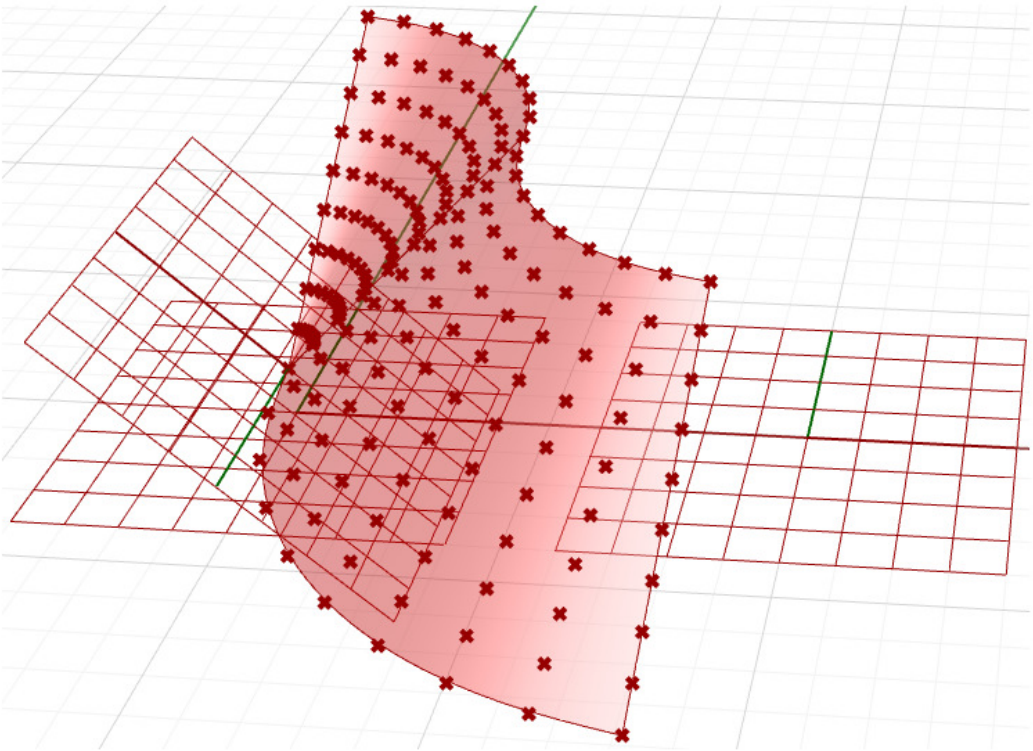
Output

S: surface.

Example

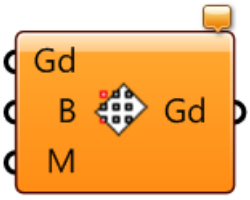
This example generates two surfaces using the two grids created by the polar 3d grid component





PtTrim

The **ptTrim** trims a grid using base brep. The user can choose to trim inside, outside or shift out points to the edge.



Input

Gd: input grid.

B: trim polysurface.

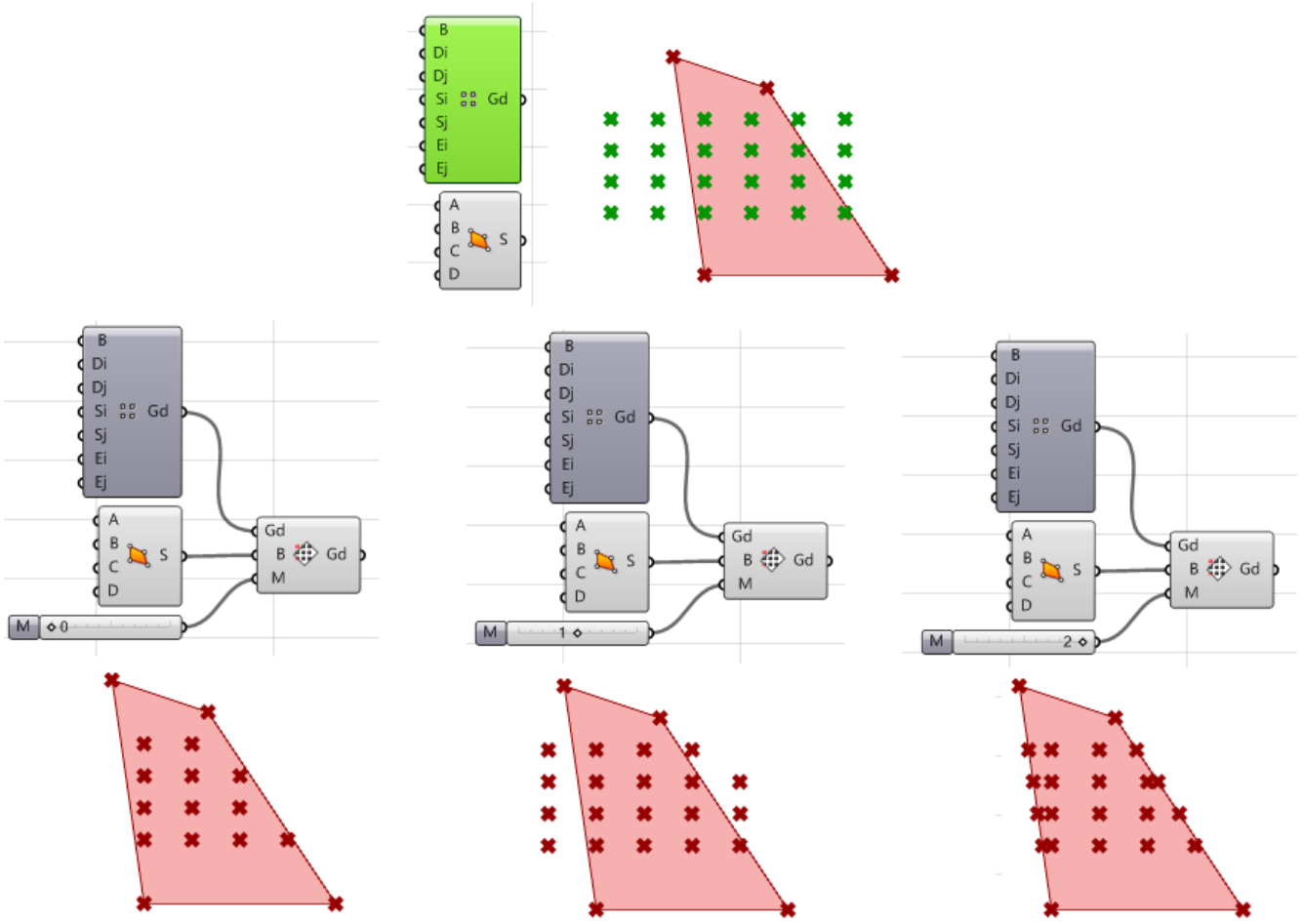
M: trim method (0=inside, 1=outside, 2=edge)

Output

Gd: output grid.

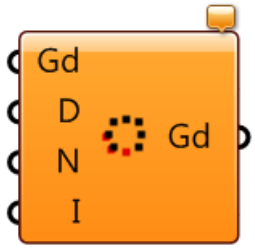
Example

This example shows different types of trimming against a reference surface.



PtWrap

The **ptWrap** copy in place rows or columns and append to the end of the grid. This is sometimes needed to close a grid or extend far enough to accommodate patterns that stretch more than two grid points.



Input

Gd: input grid.

D: 0=add rows, 1=add columns.

N: number of columns/rows to wrap

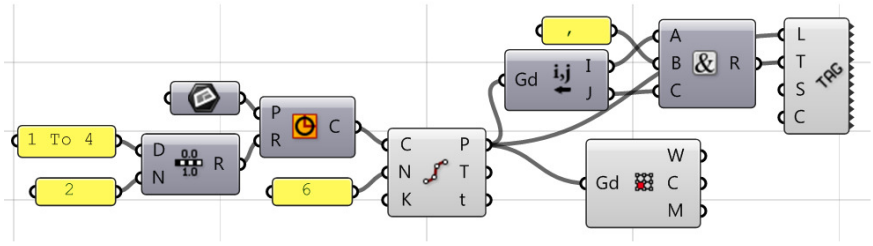
I: starting index of the row or column to copy in place.

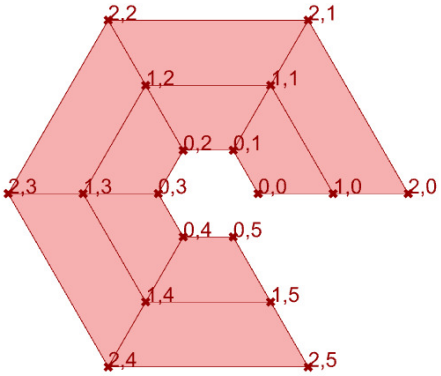
Output

Gd: output grid.

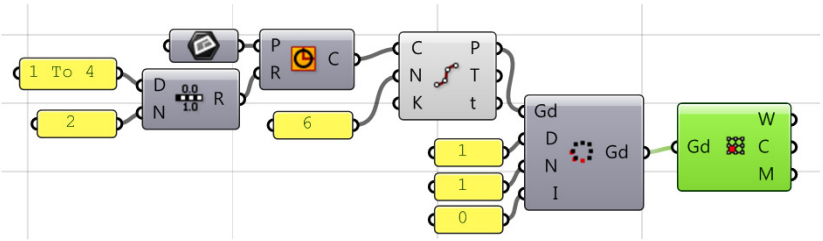
Example

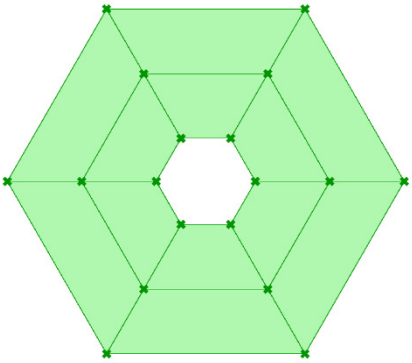
The first definition shows a gap when add panels. This is because the grid is not closed in the "v" or "y" direction. Meaning there need to have the first column of points to be appended to the end of the grid. For example, first row (labeled 0,0-0,1-…-0,5) need an additional point (0,6) that overlaps (0,0) to close that row. The same goes for the remaining rows.





Adding the ptWrap component helps append one more column at the end of the grid to close it.



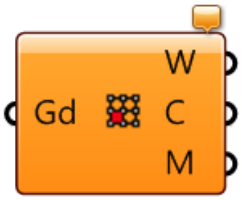


Components: Panel 2D

These components generate panels using one grid.

PtCell

The **ptCell** generates list of wires, borders and meshes of the paneling cells using a grid of points. It is possible to input multiple grids and the output will be organized in main branches that represent the number of input grids.



Input

Gd: input grid.

Output

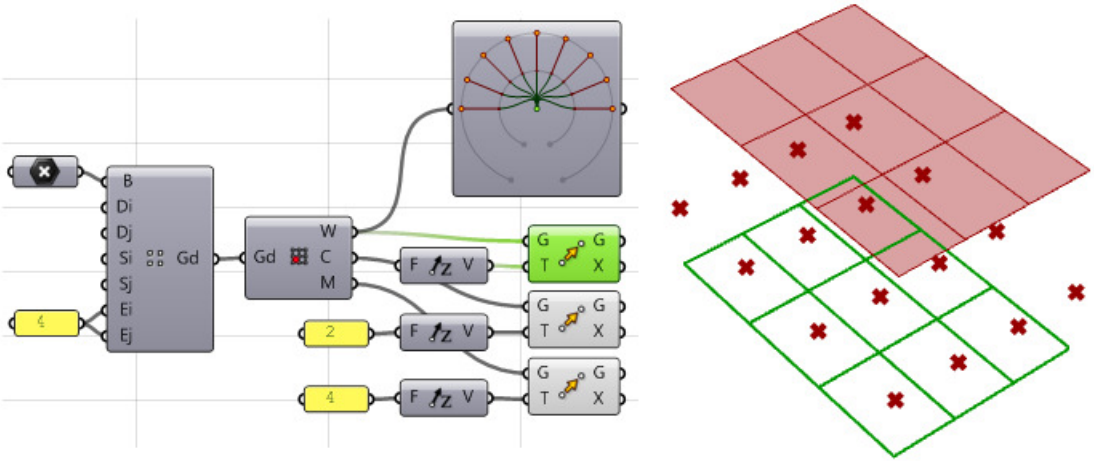
W: list of wires (4 wires per cell)

C: list of corners (4 points per cell)

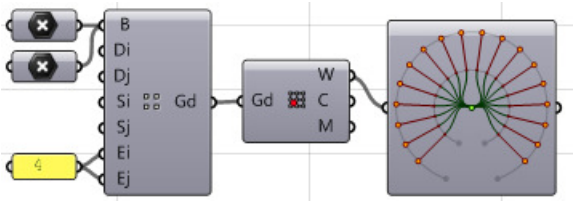
M: list of meshes (one mesh per cell)

Example

This example creates a 4x4 grid (9 cells). ptCellulate component generates lists of cells' wires, cells' corners and meshes.

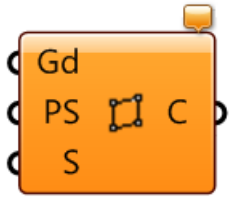


This example shows the output when the input is more than one grid.



PtBorders

The **ptBorders** creates a structure of borders(poly-curves). The output is organized in branches that represent the rows of the paneling structure.



Input

Gd: input grid.

PS: panels shape. 0=straight, 1=pull to the reference surface "S", 2=iso, 3=shortest path.

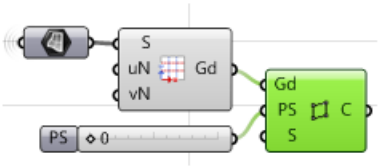
S: reference surface. One is created from the grid if none is provided.

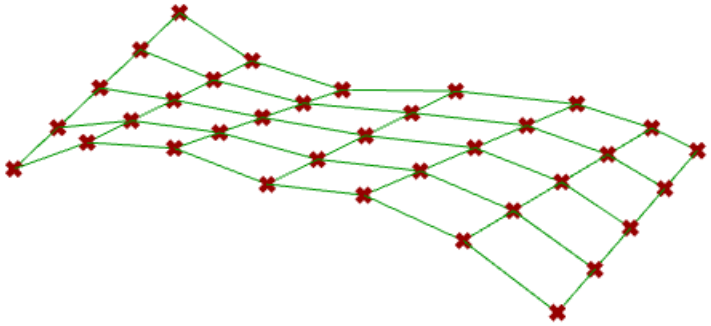
Output

C: output borders.

Example

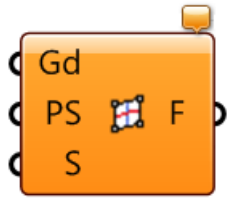
The output in this example has 6 branches with 4 curves (borders) in each branch.





PtFaces

The **ptFaces** creates a structure of faces. The output is organized in branches that represent the rows of the paneling structure.



Input

Gd: input grid.

PS: panels shape. 0=straight, 1=pull to the reference surface "S", 2=iso, 3=shortest path.

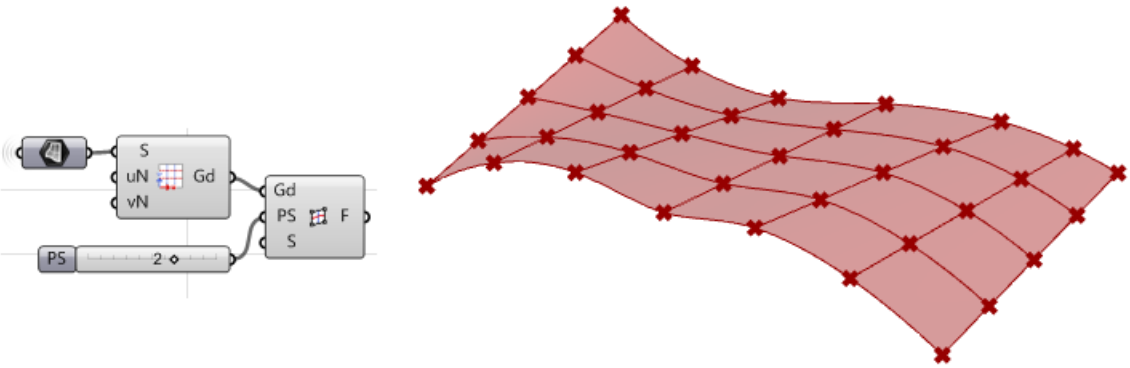
S: reference surface. One is created from the grid if none is provided.

Output

F: output faces.

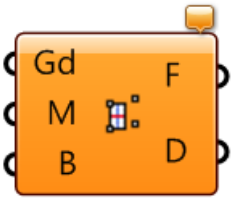
Example

The output in this example has 6 branches with 4 faces in each branch.



PtFlatFaces

The **ptFlatFaces** creates a structure of best-fit planar faces. The output is organized in branches that represent the rows of the paneling structure.



Input

Gd: input grid.

M: flattening method. 0=best fit planar faces, 1=fit plane through (1^{st},2^{nd},3^{rd}) corners, 2=fit through (2^{nd}, 3^{rd}, 4^{th}), 3=fit through (3^{rd}, 4^{th}, 1^{st}), 4=fit through (4^{th}, 1^{st}, 2^{nd}).

B: optional reference base polysurface.

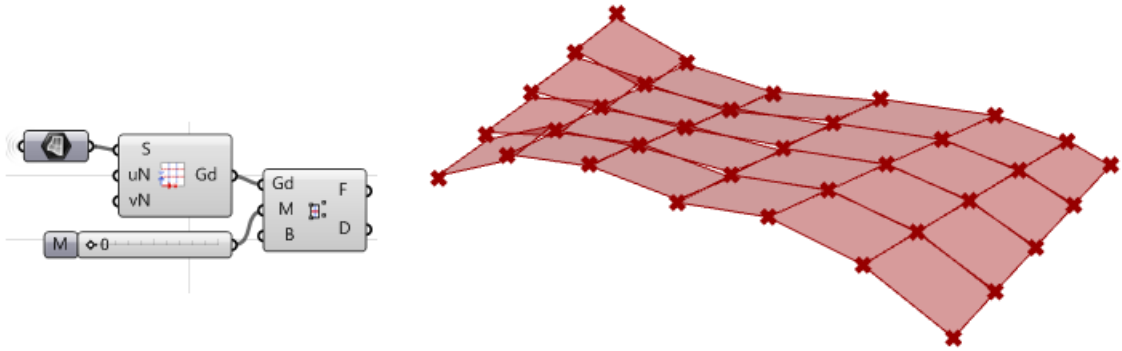
Output

F: output planar faces.

D: deviation map of output faces from the grid.

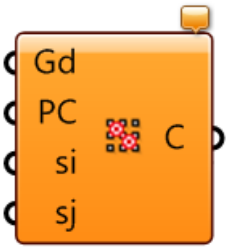
Example

The output in this example has 6 branches with 4 planar faces in each branch. Notice that faces might not join if the grid is free form, but they touch in at least one point and they do not overlap.



PtMorph2D

The **ptMorph2D** component distributes 2D modules over a given paneling grid.



Input

Gd: input grid.

PC: unit module (list of curves).

si: shift in the i direction (between columns).

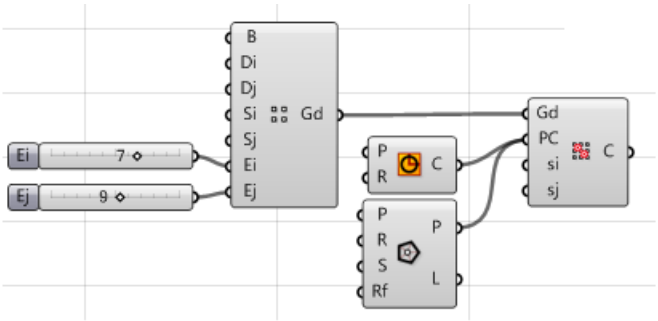
sj: shift in the j direction (between rows).

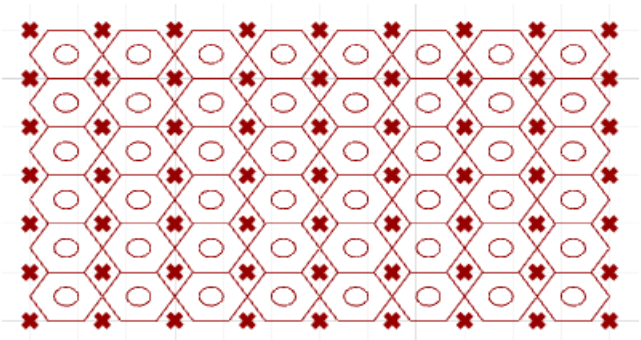
Output

Gd: list of output curves.

Example

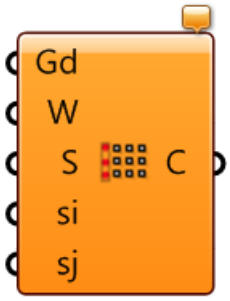
The module in this example consists of two curves. Those are distributed on the grid using the ptMorph2D component. Notice that the distributed unit module occupies the grid unit area bounded by 4 points.





PtMorph2DList

The **ptMorph2DList** is a variable distribution of a list of modules using attractors.



Input

Gd: input grid.

W: grid of normalized weights (values are 0-1) that has identical structure as the input points grid.

PC: list of pattern curves to be distributed following the weights map.

si: shift in the i direction (between columns).

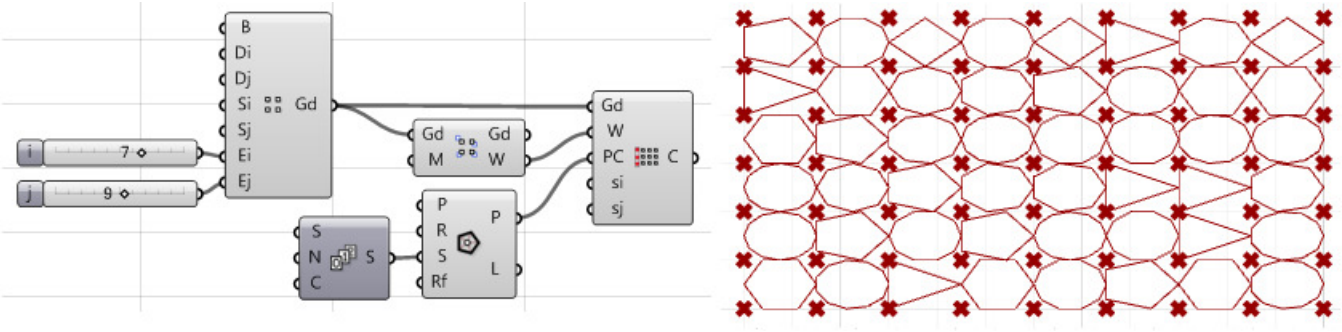
sj: shift in the j direction (between rows).

Output

C: list of output curves.

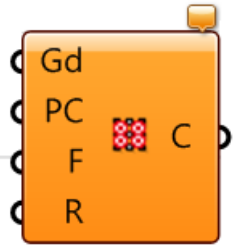
Example

The input consists of ten polygons (from 3 to 12 sides). Those are distributed to the grid randomly. The weights grid is generated using the random attractor (ptRandAtt).



PtMorph2DMap

The **ptMorph2DMap** component maps each module in a list, to one grid cell. There is an option to repeat last module to the remaining cells. This is useful to use if you have specific modules you would like to apply to specific location on the grid.



Input

Gd: input grid.

PC: list of pattern curves to be distributed in order.

F: flatter a grid into a list of cells (need to set to true if the grid was not flattened into a list of cells).

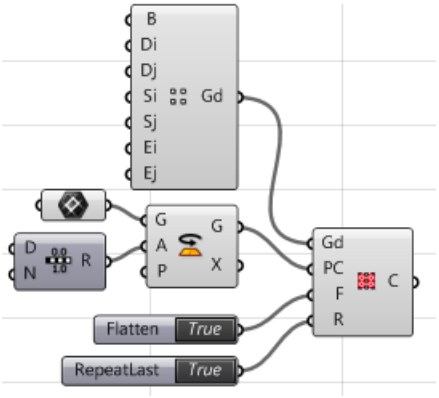
R: repeat the last module. If the list of modules is less than the number of cells, then use this option to map the last module to the rest of the grid cells.

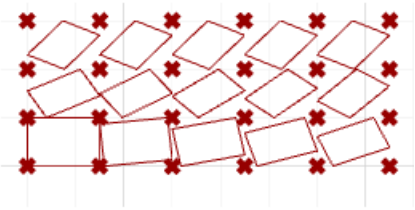
Output

C: list of mapped curves.

Example

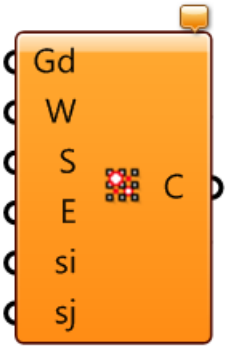
In the example, we created 11 rectangles rotated slightly from one to the next. The input grid has 15 cells. Rectangles are mapped to the grid cells in order the repeat the last one to cover the rest of the cells.





PtMorph2DMean

The **ptMorph2DMean** copy in place rows or columns and append to the end of the grid. This is sometimes needed to close a grid or extend far enough to accommodate patterns that stretch more than two grid points.



Input

Gd: input grid.

D: 0=add rows, 1=add columns.

N: number of columns/rows to wrap

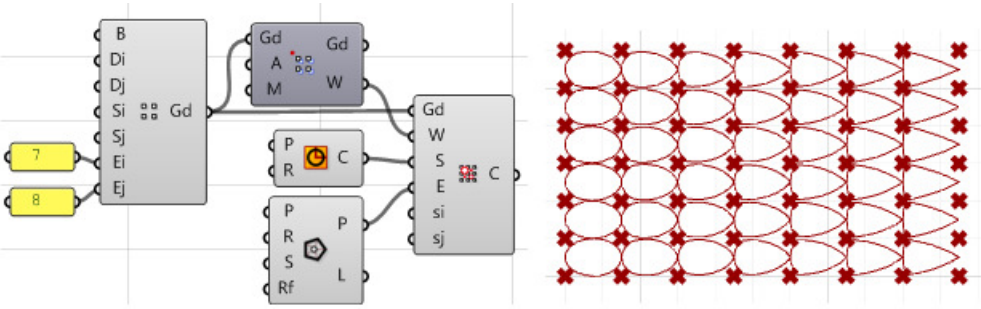
I: starting index of the row or column to copy in place.

Output

Gd: output grid.

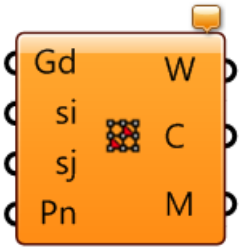
Example

Using a point attractor, this example generates tween shapes between a circle and a triangle and maps to the grid.



PtMPanel

The **ptMPanel** helps generate very fast and efficient pattern coverage through connecting grid points.



Input

Gd: input grid.

si: shift in the i direction (between columns).

sj: shift in the j direction (between rows).

Pn: pattern string.

Output

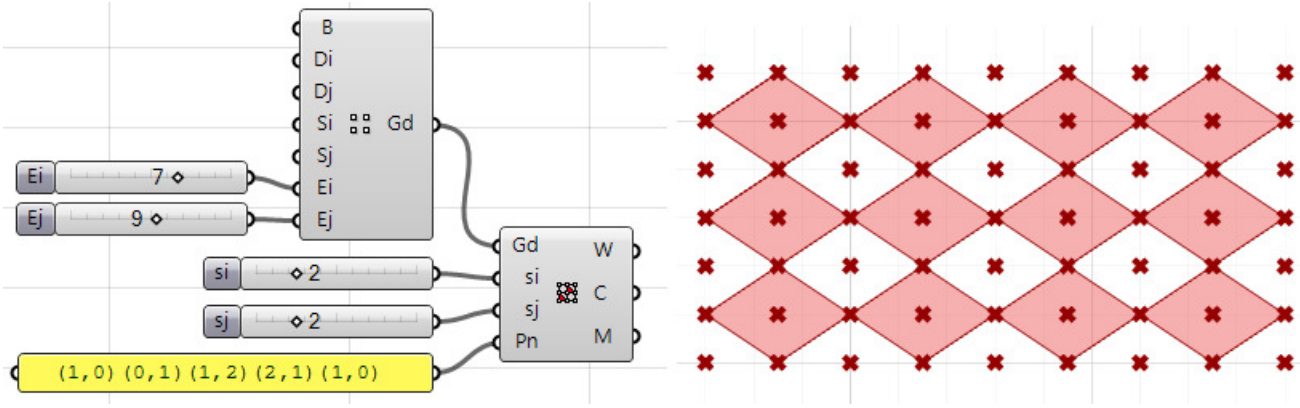
W: list of wires.

C: list of polycurves.

M: list of meshes.

Example

This example creates a diagrid using ptMPanel component. The string defines unit connections assuming that the base point of the grid has (0,0) index. Shift in "i" and "j" is set to "2" because the diamonds span over 2 unit grids.



Components: Panel 3D

These components generate 3D panels using two bounding grids.

Pt3DCell

The **pt3DCell** generates list of wires, cell corners and meshes of the 3D cell. The component takes two bounding grids and output list of wires connecting corresponding grid points, list of corner points (8 points for each cell) and a list of mesh boxes.

Input

Gd1: first input grid.

Gd2: second input grid.

Output

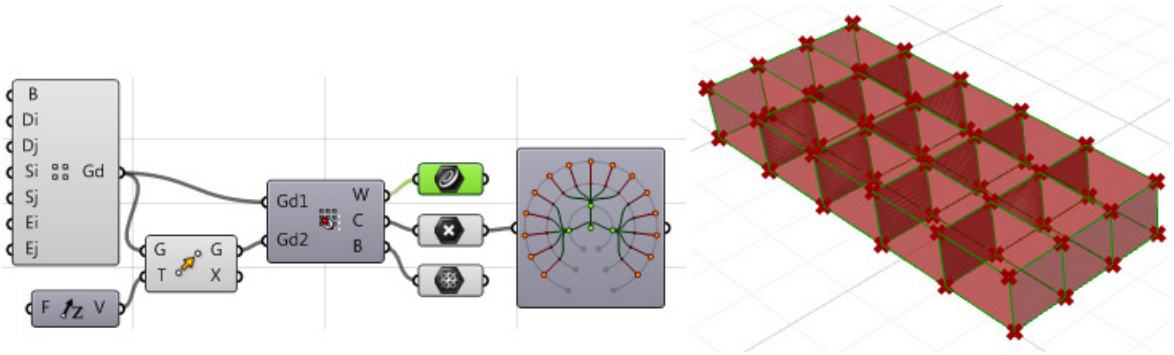
W: list of wires by rows

C: list of corners (8 points per cell)

M: list of mesh boxes by rows

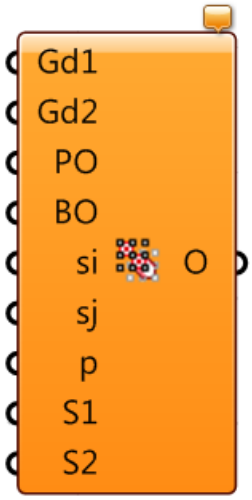
Example

The two bounding grids enclose 15 cells organized in three rows with five cells in each row. Output is organized by rows.



PtMorph3D

The **ptMorph3D** morphs 3D modules to 3D cells enclosed by two bounding grids.



Input

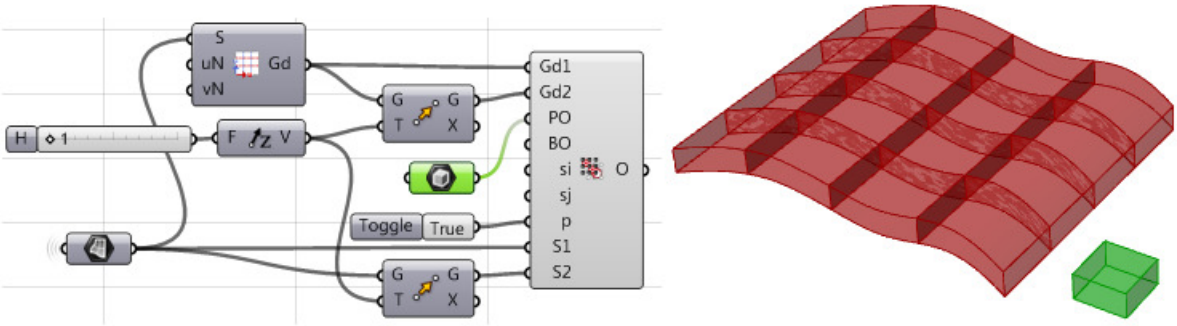
Gd1: first input grid.
Gd2: second input grid.
PO: Input pattern object.
BO: input bounding objects for pattern.
si: Shift in unit grid in i dir.
sj: Shift in unit grid in j dir.
p: if set to "True" then perform smooth morphing.
S1: (optional) first bounding surface (corresponds to Gd1).
S2: (optional) second bounding surface (corresponds to Gd2).

Output

O: output morphed objects.

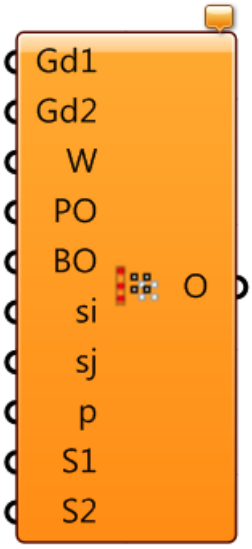
Example

This example morphs a box smoothly between two bounding grids and surfaces.



PtMorph3DList

The **ptMorph3DList** morphs 3D list of modules to 3D cells enclosed by two bounding grids.



Input

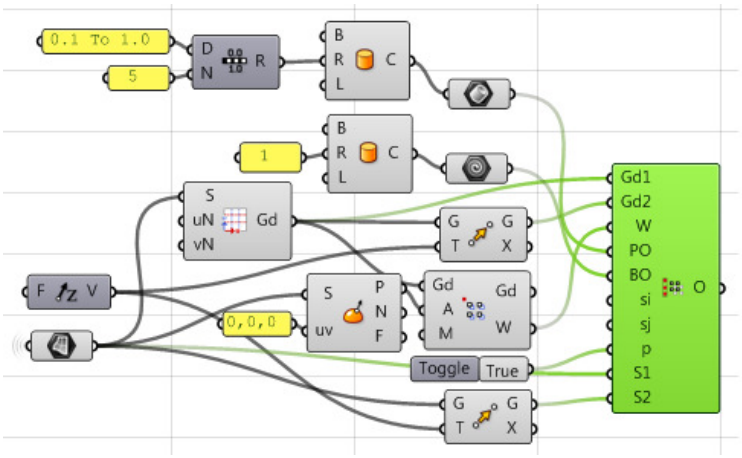
Gd1: first input grid.
Gd2: second input grid.
W: weight map.
PO: Input list of pattern object.
BO: input bounding objects for pattern.
si: Shift in unit grid in i dir.
sj: Shift in unit grid in j dir.
p: if set to "True" then perform smooth morphing.
S1: (optional) first bounding surface (corresponds to Gd1).
S2: (optional) second bounding surface (corresponds to Gd2).

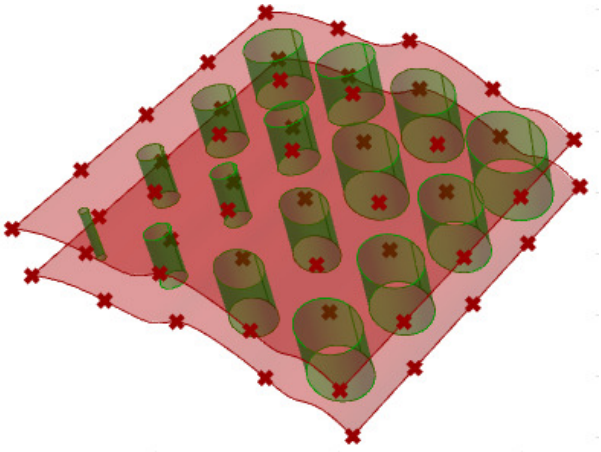
Output

O: output morphed objects.

Example

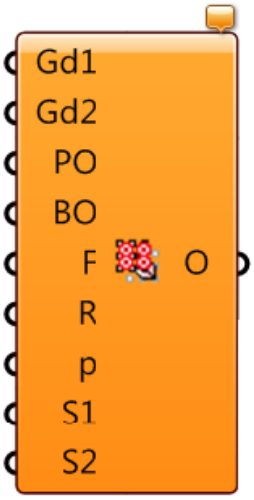
This example morphs a list of cylinders between two bounding grids and surfaces.





PtMorph3DMap

The **ptMorph3DMap** maps each module in a list to one 3D grid cell.



Input

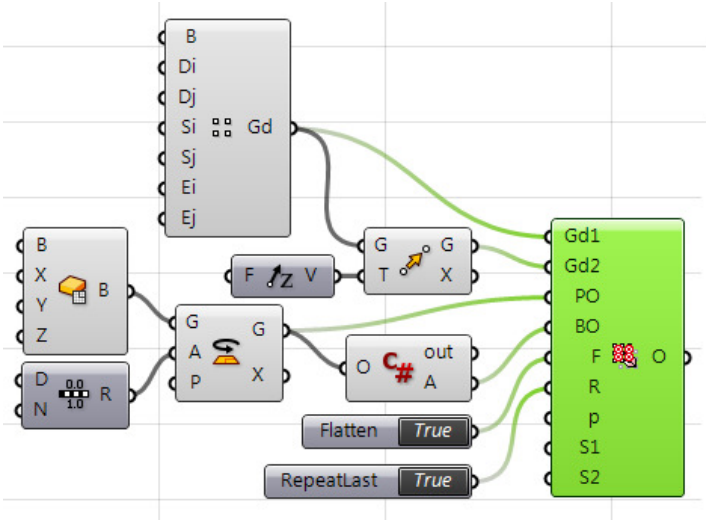
Gd1: first input grid.
Gd2: second input grid.
PO: Input list of pattern object.
BO: input bounding objects for pattern.
F: Flatten the grid cells into one list of cells to be able to map in order.
R: Repeat last object to remaining cells, if any.
p: if set to "True" then perform smooth morphing.
S1: (optional) first bounding surface (corresponds to Gd1).
S2: (optional) second bounding surface (corresponds to Gd2).

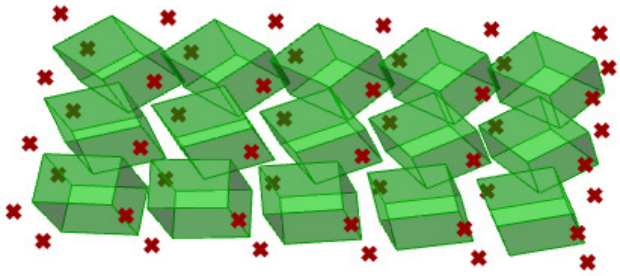
Output

O: output morphed objects.

Example

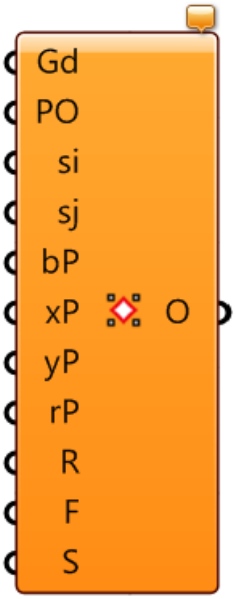
This example morphs a list of rotated boxes between two bounding grids and surfaces.





PtOrient

The **ptOrient** maps 2D or 3D modules to a grid. If pattern reference points are not provided, bounding box points of the pattern is used.



Input

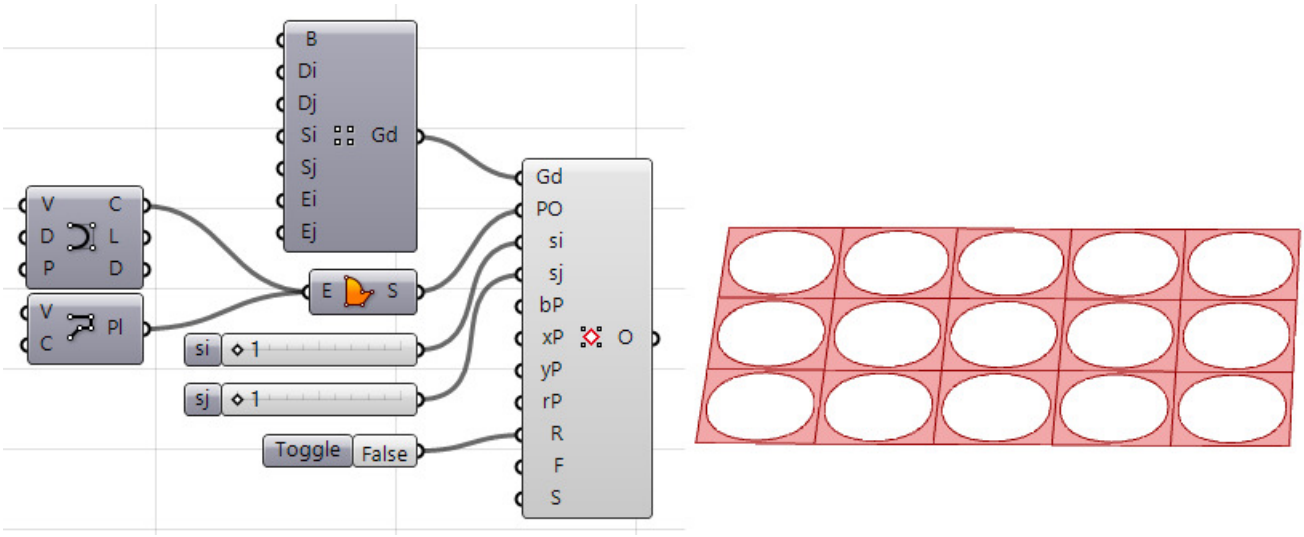
Gd: Input grid.
PO: Input pattern.
si: Shift in unit grid in i dir.
si: Shift in unit grid in i dir.
bP: Base point for the module.
xP: X direction reference point.
yP: Y direction reference point.
rP: 4Th reference point.
R: Rigid orient.
F: Flip.
S: (optional) base surface for smooth morph.

Output

O: output morphed objects.

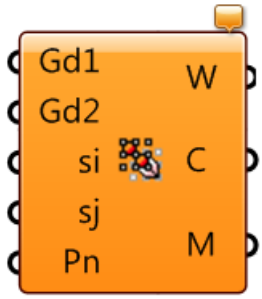
Example

This example morphs a trimmed surface to a grid.



PtMPanel3D

The **ptmPanel3D** creates 3D paneling using modules defined by connecting grid points.



Input

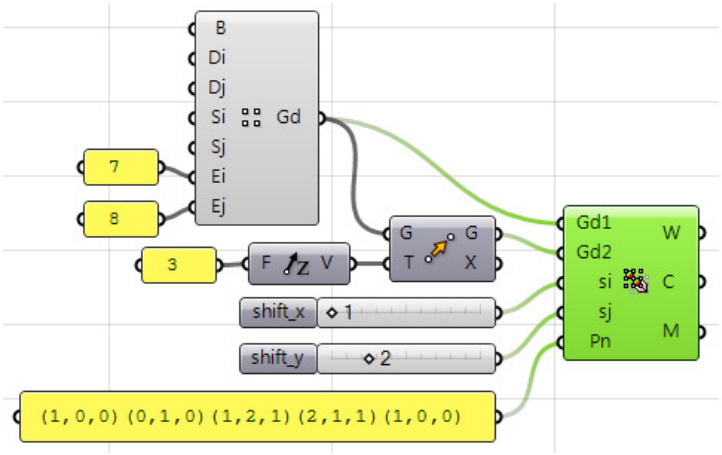
Gd1: First input grid.
Gd2: Second input grid.
si: Shift in unit grid in i dir.
si: Shift in unit grid in i dir.
Pn: Pattern string.

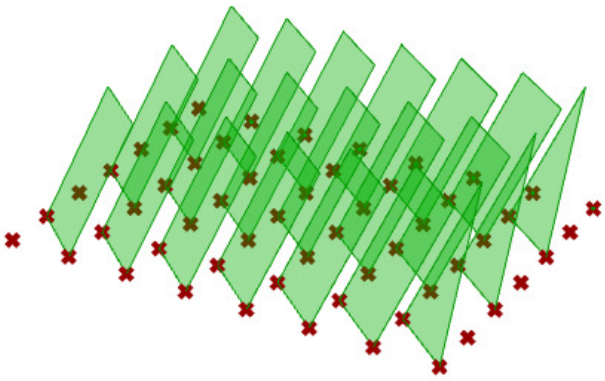
Output

W: Line wires.
C: Polylines.
M: Meshes.

Example

This example creates custom module then populate between two bounding grids.





Components: Panel Utility

These components help create special panels.

PtIsoE

The **ptIsoE** helps extract iso-edges on a given surface using linear edges. It uses the two end points of the input lines and generates an iso-curve on the surface using the surface UV directions. Both end points need to lie on the surface and align on the same surface iso.

Input

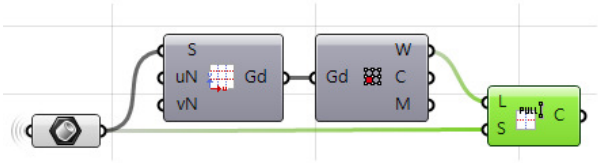
L: lines to extract the iso curves.
S: input surface.

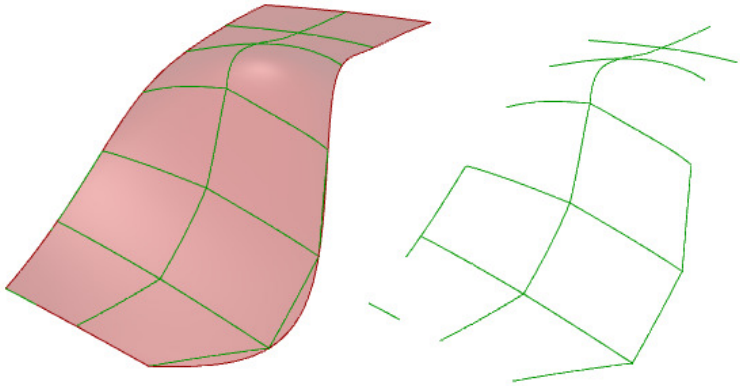
Output

C: resulting iso-curves.

Example

Pulled straight edges might not end up on surface and this is why you notice edge curves are missing or short.





PtPullE

The **ptPullE** pulls linear edges to a surface.

Input

L: lines to extract the iso curves.

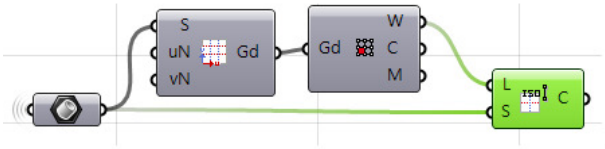
S: input surface.

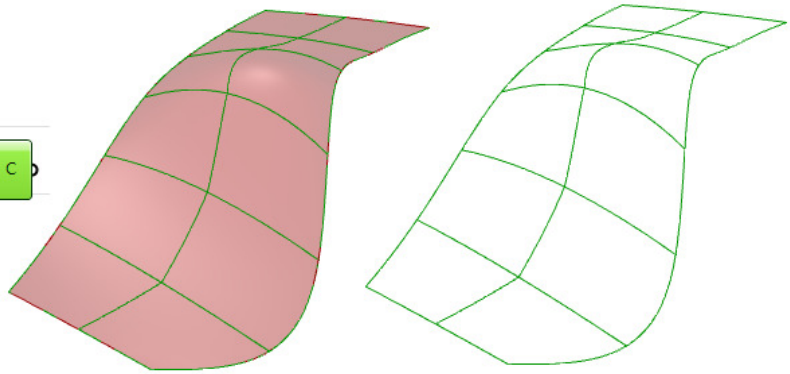
Output

C: resulting pulled curves.

Example

When the two end point of the line segments align with the surface UV, the result is clean continuous curves.





PtShortE

The **ptShortE** extracts shortest path on surface between the two end points of input linear edges.

Input

L: lines to extract the iso curves.

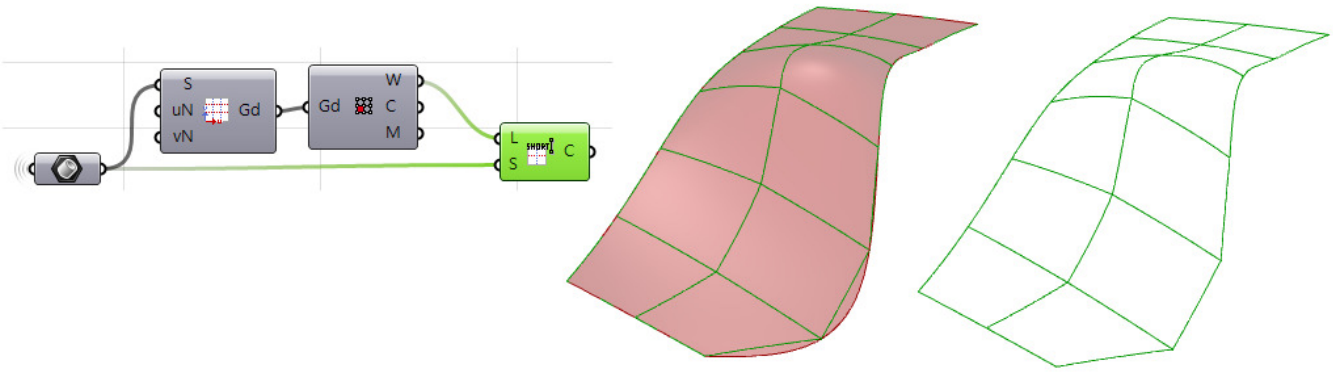
S: input surface.

Output

C: resulting short curves.

Example

If the two end points are within the surface domain, the shortest path will always results in a complete curve, but they will not necessarily follow the iso direction of the surface.



Components: Select and Bake Grids

PanelingTools is an integrated plugin for Rhino and Grasshopper. Sometimes it is useful to go back and forth between the Rhino and GH environments. The following components helps do just that..

PtSelGrid

Grids created and edited in Rhino can be brought to GH using **ptSelGrid** component.

Input

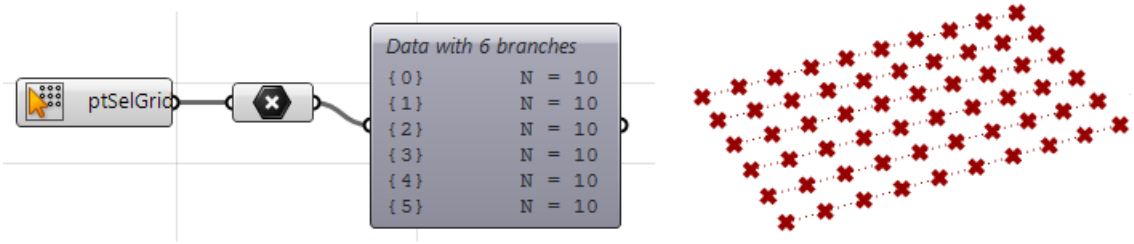
Click on the icon to select a paneling grid created using PT for Rhino

Output

Ordered grid in GH.

Example

Paneling grids created in Rhino sustain their structure when selected into GH as shown in the following.



PtBakeGrid

Bake grids from GH to Rhino.

Input

G: Input grid(s).

S: name of the grid (string).

B: Bake the grid when set to "true". Make sure to set back to "false" once baked.

Output

Paneling grid(s) baked to Rhino.

Example

You can bake more than one grid from GH to Rhino.