AUTODESK UNIVERSITY

CS21802: Dynamo for Construction Workflows

Brendan Nichols
The Beck Group
BrendanNichols@beckgroup.com
@BrendanANichols

Learning Objectives

- Learn construction-specific uses for Dynamo
- Learn some Dynamo basics
- Learn how to identify tasks Dynamo could be useful for
- Get an introduction to Dynamo software's online community and package manager

Description

We've all seen the Dynamo extension used for twisting dramatic towers, and we've even seen it used to recreate some of our favorite Star Wars characters. But if you're a general contractor or work in the construction industry, you may be wondering if Dynamo can do anything for you. In this class, we will demonstrate real construction problems that were solved with the Dynamo extension. You will see demonstrations analyzing slab flatness, cut-and-fill volume calculation, and installation simulation. With the skills and examples taught in this class, you will learn ways that Dynamo could be a game changer for your business too. This session features Dynamo Studio and Revit. AIA Approved

Your AU Expert

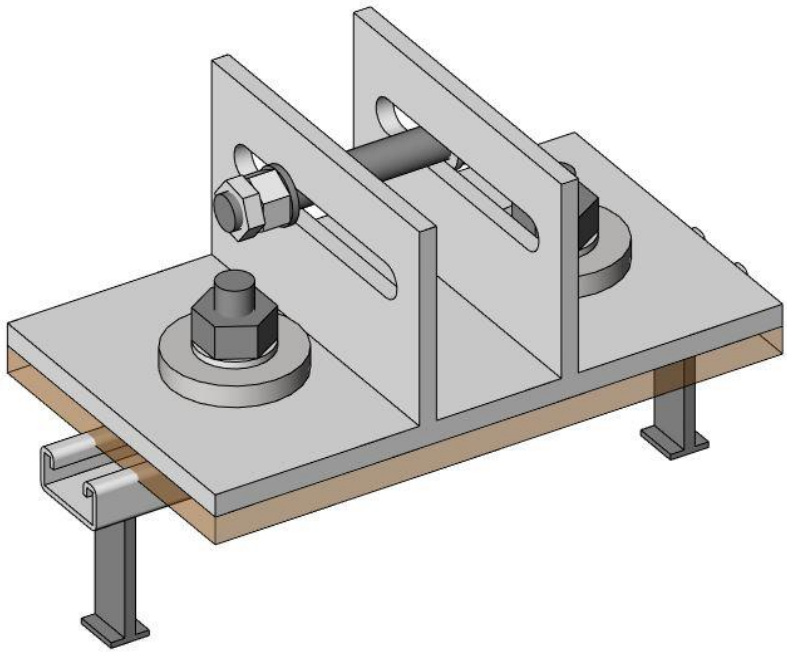
Brendan Nichols is a Senior Virtual Building Engineer at the Beck Group. Beck's integrated design-build model allows him to tackle difficult challenges across both Architecture and Construction. At the Beck Group, Brendan performs laser-scanning, building information modeling, and supports these processes and workflows throughout the company. After experiencing Beck's cross-disciplinary challenges, Brendan has written multiple internal Revit plugins to eliminate difficult or repetitive day to day tasks. He also enjoys writing scripts and Dynamo definitions to solve some of Beck's unique problems.

Halfen Detail Tolerance Simulation

For a legal case, I was asked to find a visual way to explain the tolerances in a Halfen embed connection and how they would impact the installation of a curtain wall system. Halfen embeds are typically cast in concrete and can be off of their desired location. The Halfen channel itself allows tolerance in one direction, while the rest of the attachments provide tolerance in two different directions with different methods.

A Revit family was made that could simulate these components and their adjustability. Dynamo was used to manipulate and animate these components to show the full range of flexibility in a way that legal staff could understand and present.

This section shows some of the very basic parts of Dynamo like manipulating parameters in families. It also introduces the package manager and the Dynamo community.



Halfen Connection Family

This family is designed in a way where the channel, and plates that attach the curtain wall to the concrete slab edge, can all move without changing the actual location of the curtain wall attachment. It has three instance parameters which control this movement:

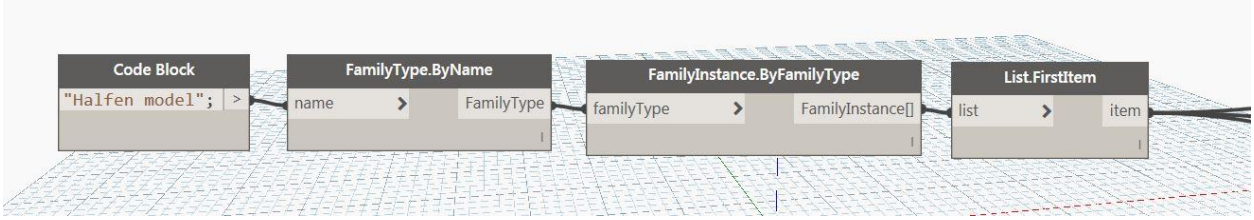
Halfen Vert- controls the vertical position of the Halfen channel and simulates how the system adapts to the channel being placed higher or lower in elevation than designed.
Halfen left- controls the left and right position of the Halfen channel and simulates how the system adapts to the channel being to the left or right of its design location.
Offset from Face- controls the in-out location of the Halfen channel. This simulates how the system reacts to the slab edge being off from its designed location

In our case, as the contractor, we were obligated to ensure that the Halfen channel and the slab edge were in the designed location plus or minus our tolerance. The subcontractor would then be responsible for the rest of the system that attaches to the curtain wall mullion.

This class will not spend time explaining how to make this family, but I wanted you to understand the types of parameters we will be driving with Dynamo.

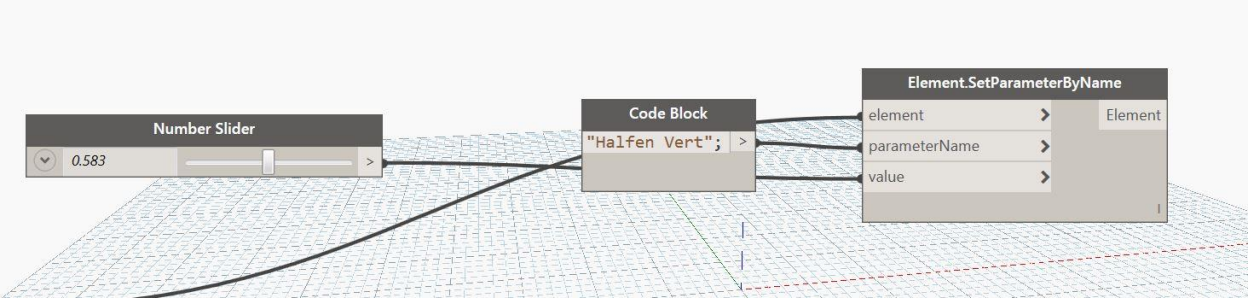
Driving from Dynamo

In Dynamo we can select an instance of this family that's placed in the project. One way of doing that is by using this script.



This script uses the name of the family to get a list of all instances of the family and then we take the first one from that list. It's very simple but is effectively like clicking an element in Revit. We could also use a select element node in Dynamo and then just click the element, however you would need to re-select the family if you used this script in a different file or if the original element was deleted.

Next we are going to look at the ways that we can control the parameters of this family through Dynamo. First, we need to pass the element above to a node called "*Element.SetParameterByName*". Then we also need to give that node a Parameter name and the intended value for that parameter. You can see that portion of the script below.



In this case we are driving the "Halfen Vert" parameter we discussed before and we are giving it a number to change that value to. This type of script can be used to modify almost any parameter from the dynamo environment and is something I'm always using.

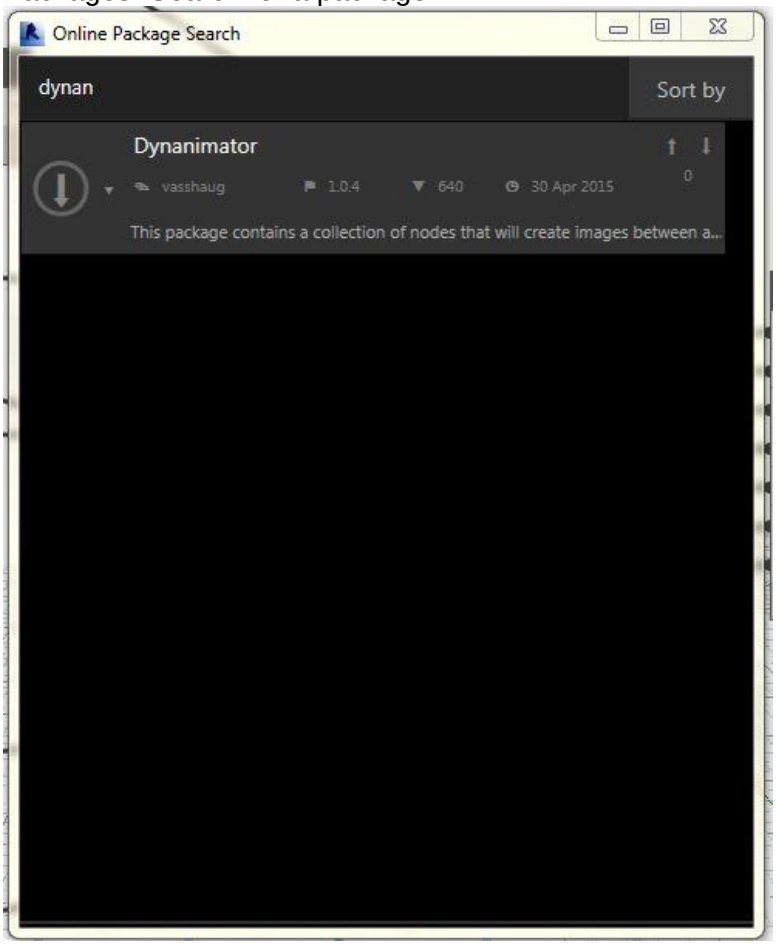
Animating in Dynamo

Revit doesn't come out of the box with any animation capabilities that I know of. And it's always something I wanted to do. Dynamo is here with a solution. I was keyed onto this node and package called "Dynanimator" created by Håvard Vasshaug. You can find the original blog post here:
https://vasshaug.net/2015/04/22/dynanimator-released-animating-data-changes-in-revit-with-dynamo/

He was using this node to do things like manipulate window size openings and see the effects on lighting inside of buildings. He's also used it to do some interesting mesh manipulations and other great stuff.

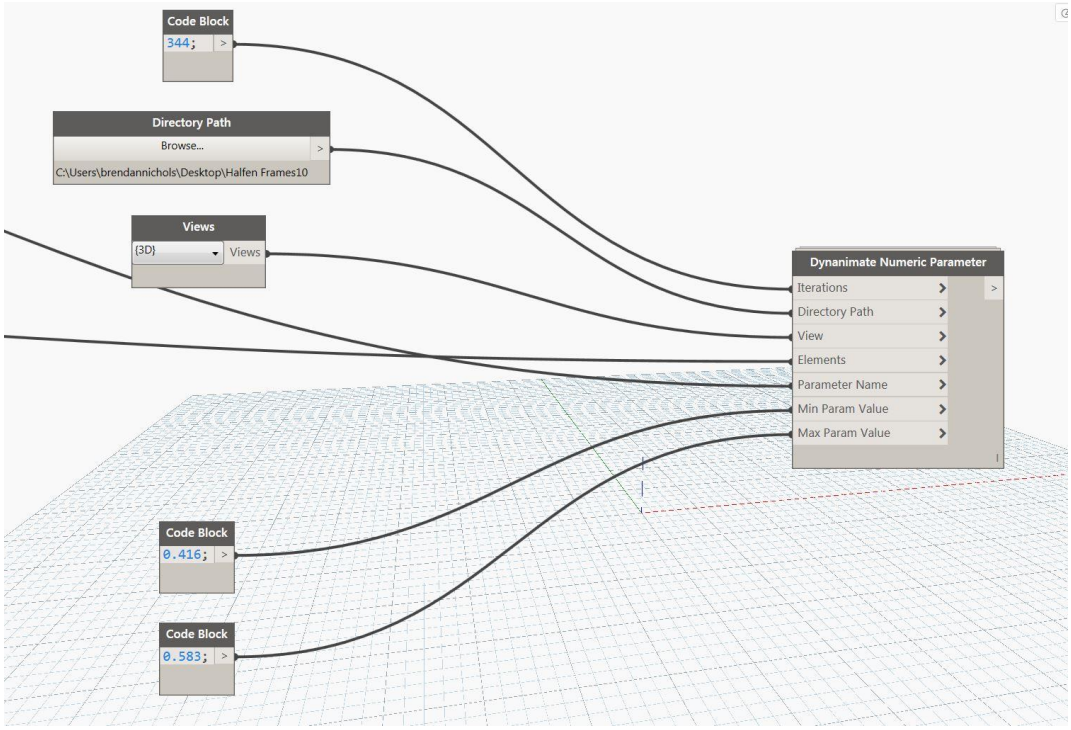
Dynamo Community

Dynamo has a very active and helpful community. You can create, find, and share Dynamo definitions directly through the Dynamo platform. I downloaded Håvard's Dynanimator package through the Dynamo package manager found under Packages>Search for a package.



From this interface you can search for any package another Dynamo user has shared on the package manager. This community and ability to share, really makes it easy to find solutions for problems that other people have already had. Most times, you don't need to re-invent the wheel to get the job done, instead you re-purpose what others have done for your own uses.

If you need any help doing this, the Dynamo forum is a great place to look for help https://forum.dynamobim.com/



Using Dynanimator

After downloading and installing the “Dynanimator” package, it will appear on the left toolbar along with all the standard Dynamo options. We will use the node called “*Dynanimate Numeric Parameter*”. This node goes in the place of the “*Element.SetParameterByName*” node that we used earlier. This node allows us to give a range of values to set the parameter to and exports a view of each iteration to a folder on your computer.

First change Dynamo’s run option to Manual. Then input the range of values you’d like to use for your parameter. Then input the number of iterations or frames you’d like to export. Keep in mind that if you plan to convert your frames to video, 30 frames typically equals 1 second of video. Then you need to set your path and set your view that you want to export. Using the standard view options in Revit you can really control the view to make sure you get the effect you are looking for. Then you can press “Run” and watch your views exporting. After your views are exported you can use any video editing software of your choice to convert these frames into a video.

Slab Flatness Analysis

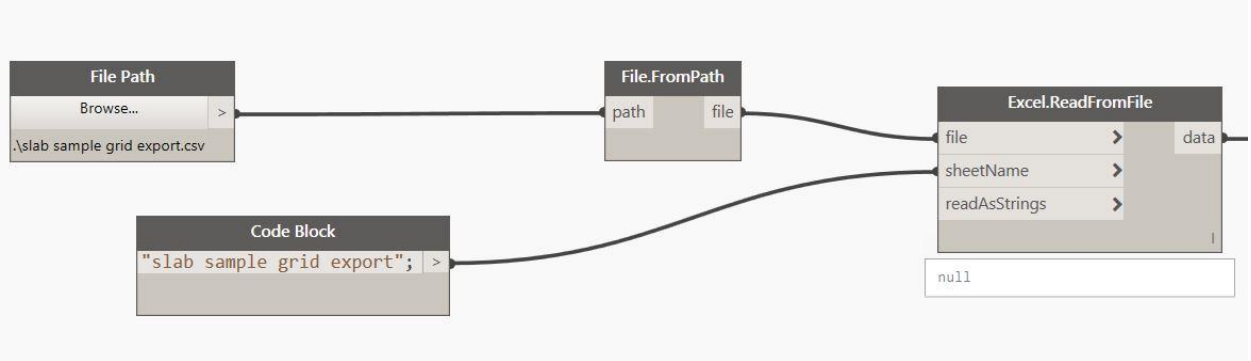
Often times after we pour slabs or demo an existing building down to the slabs, we need to add a topping slab to level the floors. Floor finishes like terrazzo are especially expensive and leveling out the elevation changes in the floor with concrete can be cheaper than just using terrazzo.

In these circumstances we like to use laser scanning to capture the existing or as-built conditions of the slab in order to analyze where the high and low spots in the slab are. We use laser scanning, but conventional surveying can also capture these deviations by measuring the elevation offsets on a consistent gridded distance. In the end we are both working with a list of spot coordinates showing the elevations of the slabs at different regular locations.

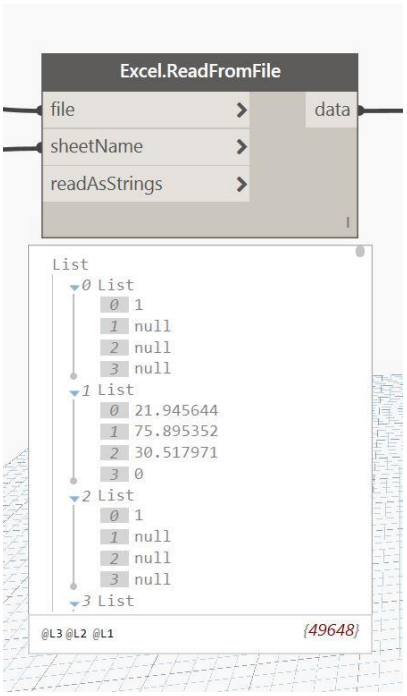
This topic will show list manipulation and cleanup steps taken through dynamo to remove erroneous data points. These data manipulations become critical in most of the cases where you use dynamo. It will also show how to place family instances from a list of coordinates and show some different visualization techniques to use on your data.

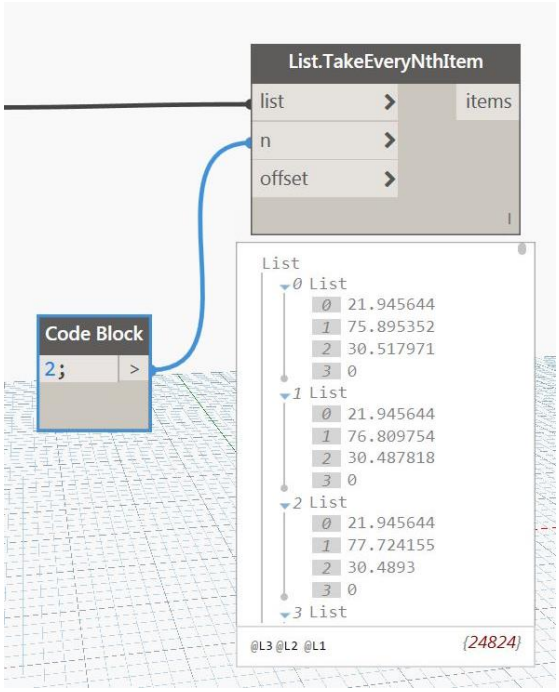
Reading from Excel file

To get our data from scans into Revit/Dynamo for analysis we export a text file that can be brought into excel. Dynamo is then used to clean the file of empty rows and sort the values into their proper positions.

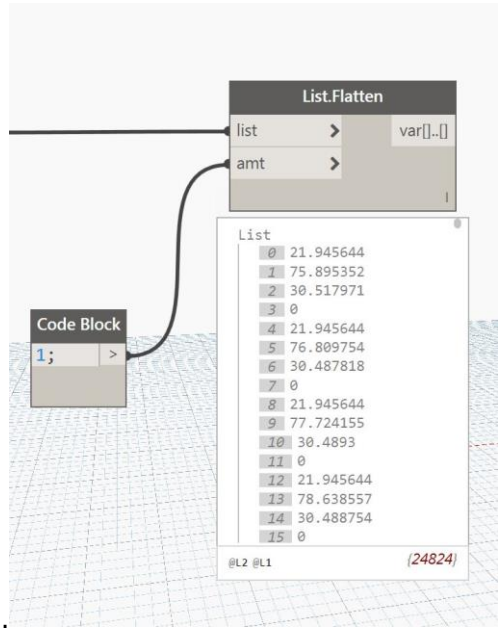


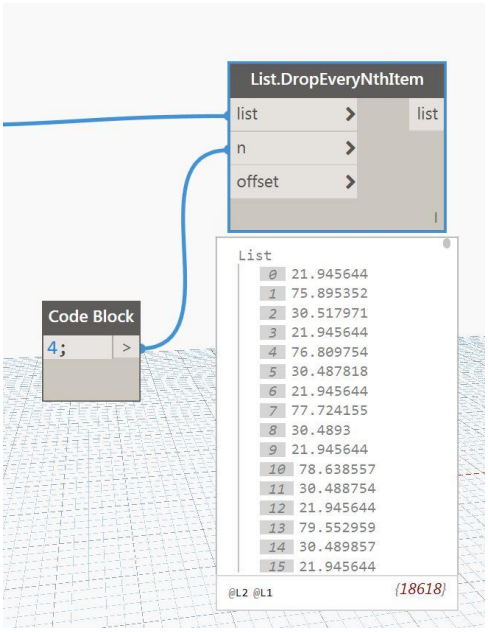
The script to read data out of excel is pretty simple. The node takes an excel file which can be navigated to using a file path node and then you need to specify the sheet name. After you read the excel file you will notice the data has some things we need to clean out.





On the left you can see the data readout of the excel file. The data structure is a series of nested lists. Every other list is full of null values, which we want to remove. We do this with the node on the right. *"List.TakeEveryNthItem"* selects every other list and returns them. Which is what we want.





.

Next we use the node on the left to flatten the lists. *"List.Flatten"* takes all the numbers and gets them out of their sublists. Which is something we need to do for the next step. You can see that every fourth value in this set is 0. That's because our pointcloud software expects there to be another value for each of these points which we didn't export this time and don't need for this analysis.

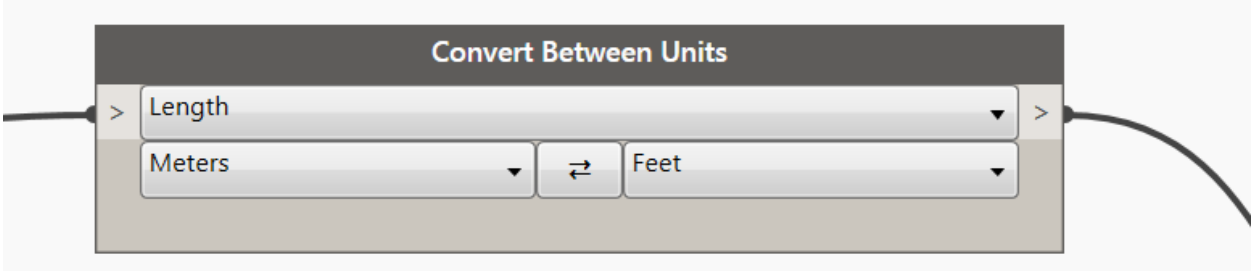
On the right you can see we use the node *"List.DropEveryNthItem"* which we use to remove every fourth item from the new list. Now our data is clean and ready for what's next!

Creating something from our cleaned data

At this point we have a list of numbers. Every three numbers is an XYZ coordinate. Each of these coordinates is a point on the slab that we are interested in. Now, we want to convert this list of numbers into actual geometry inside of Revit.

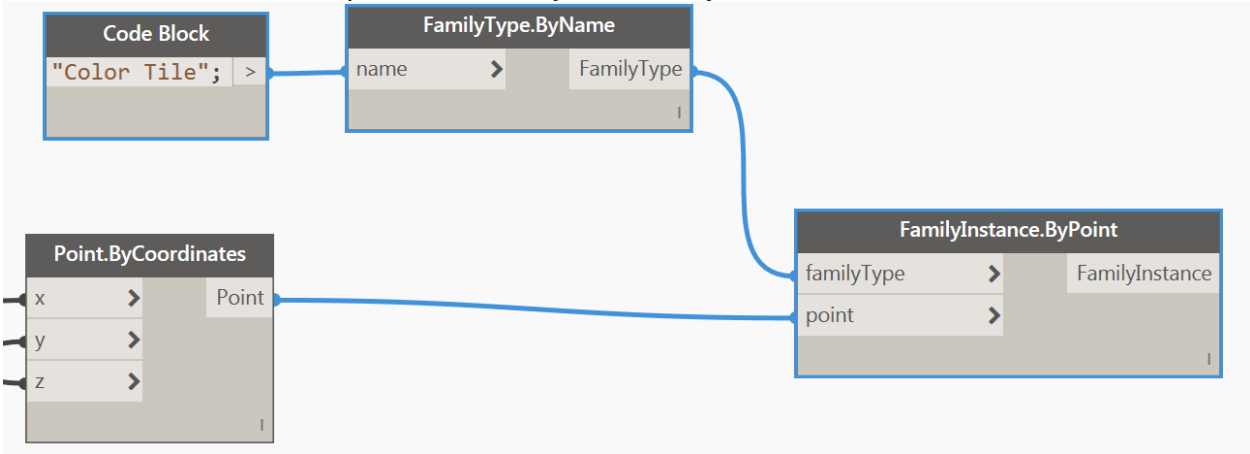
I use a family called “Color Tile” for this operation. It’s a very simple generic family. It’s point based and square. It has one parameter which controls the width of the square. Conceptually they are like pixels and we can make the pixels bigger or smaller depending on how big our sample is. So once this is loaded into Revit place one instance of it somewhere. This ensures that Dynamo can find the family to use it.

Now before we place try to place our families, we need to convert our units. Our Pointcloud software uses the Metric system and we need values in feet for our Revit project. We can easily convert this in Dynamo with a specifically designed “*ConvertBetweenUnits”* node.



This node has a few fairly obvious dropdown options for setting up your conversions.

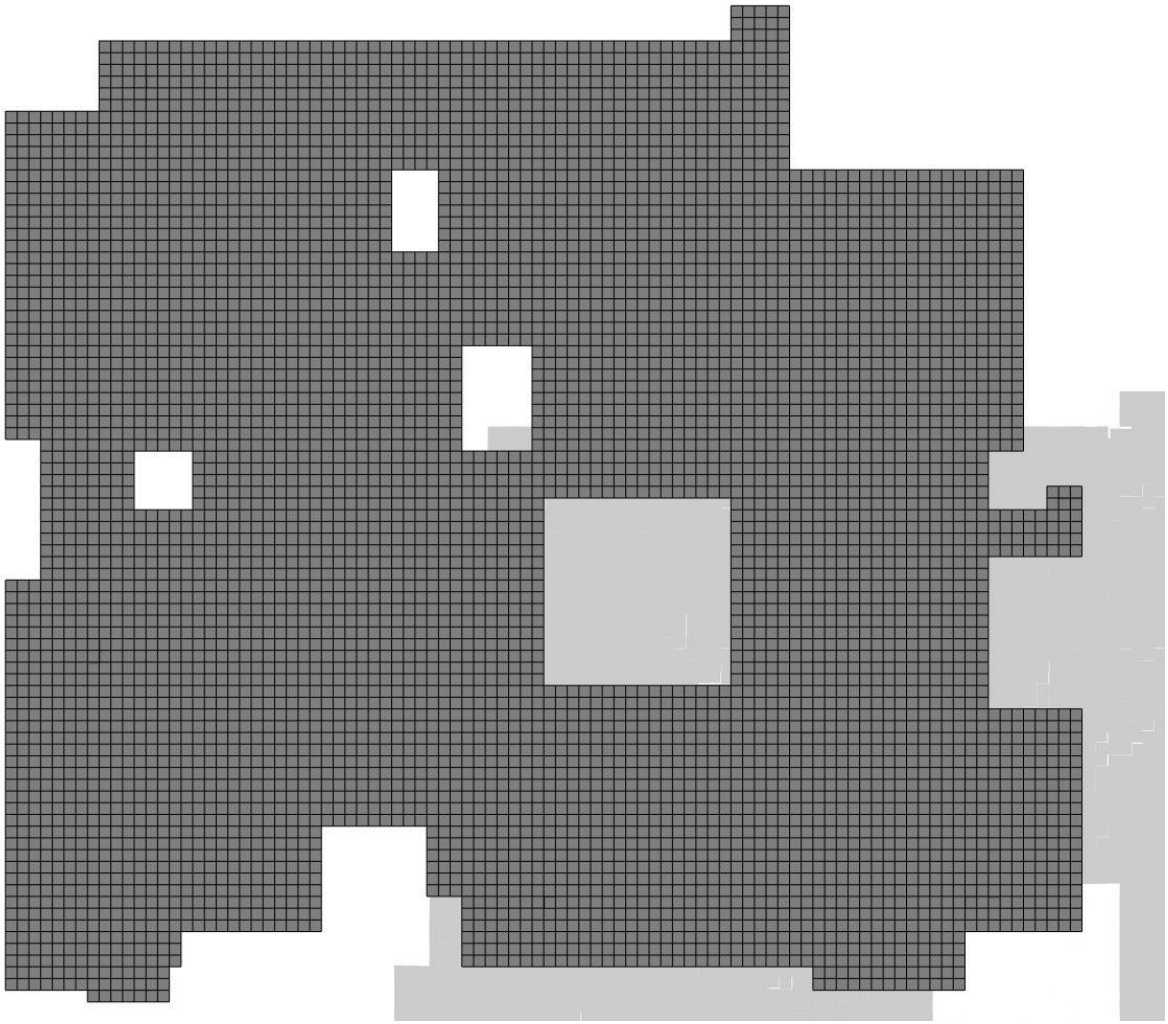
After our units have been converted we are ready to place our Sample tile family. Placing numerous family instances is one of the most practical and powerful uses for Dynamo. Doing this one time can save hours of manual tedious work whenever a problem like that pops up in your work, it’s a really good time to consider using Dynamo as your tool. The key node behind this is for our example is the “*FamilyInstance.ByPoint”* node.



This node takes two parameters. The first is a family type. We can get at a family type in Dynamo by passing the name of the family type into the “*FamilyType.ByName”* node. It is an important distinction in Dynamo that the node require very specific inputs. If we just passed the name of the family instance into this final node, it wouldn’t work.

The other input for the family instance node is a point. Again we are using the numbers we cleaned up earlier and inputting them into a “*Point.ByCoordinates*” node. This node converts them into an actual point and then we push that point into the final node. At this point, Dynamo and Revit will take a long time to place all of these instances. You need to let your computer spin while it does this. It may take an hour, but we are placing thousands of family instances into

the project. In the end it will look like this:



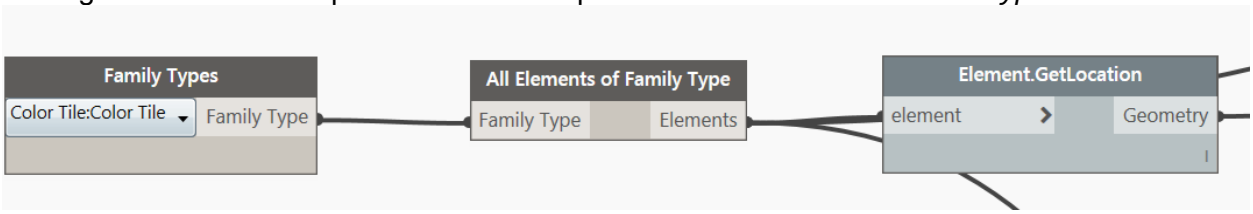
Which is really great. We have thousands of objects representing our slab. They match the surveyed points in location and accurately replicate the elevation changes of the slab. In this picture above, I've cleaned more of the erroneous points from the geometry. The pointcloud software extraction makes a few mistakes when extracting the points. These mistakes are quite difficult to remove computationally, however are very easy to remove visually in Revit. This is another case where Dynamo really makes a difference in our workflows and problem solving. It's so easy to switch between the type of things computers are good at doing and the things that humans are good at doing. If you can master when to use which approach, you can really accomplish some really interesting things.

Better Visualizing our Data

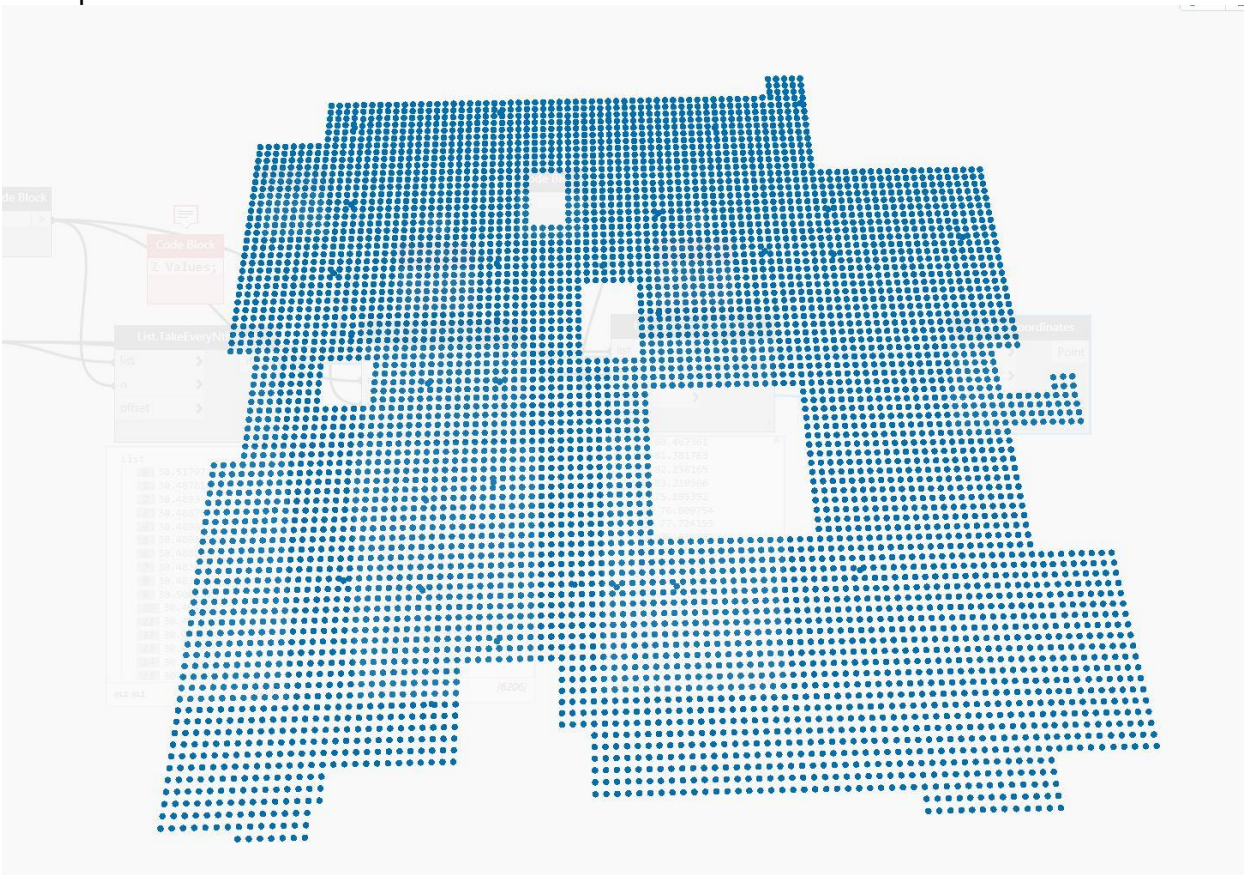
Now having these points represented with actual geometry is a really big step forward to understanding the changing slope of our slab. But Dynamo has some even better things we can do with this data. We'd really like to use color to make the changing elevation of the slabs even easier to understand.

We're going to move to a separate Dynamo script for this second part. We could continue working in the same file but I like to follow a software development idea when working in Dynamo as well. That idea is to separate concerns and make single use scripts. The idea behind that is that it's easier to see where things are going wrong and easier to re-use scripts. So our first script has the job of cleaning up data and inserting family instances. This second

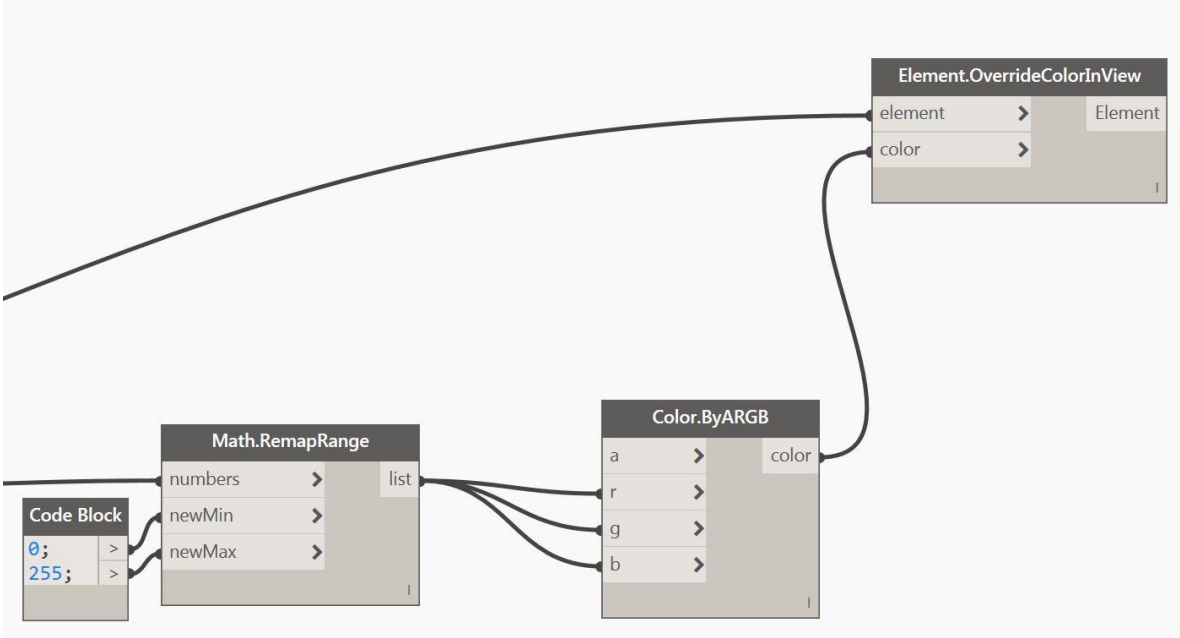
script will be tasked with coloring family instances based on elevation relative to one another. Also our operation in the first script is very computationally intensive, and we don't want to repeat it and slow ourselves down. However, that means we are going to have to "get at" those tiles again in our new script. Which is accomplished with the "*AllElementsOfType*" node.



This node takes a Family Type like we used earlier and gets all instances in the entire project. This time we are going to use the dropdown family type selector node to choose our "Color Tile" family that we have in our Revit file. If you put that node's output into the "*Element.GetLocation*" node we can see all of our points that we were working with earlier, will show up in our Dynamo workspace.

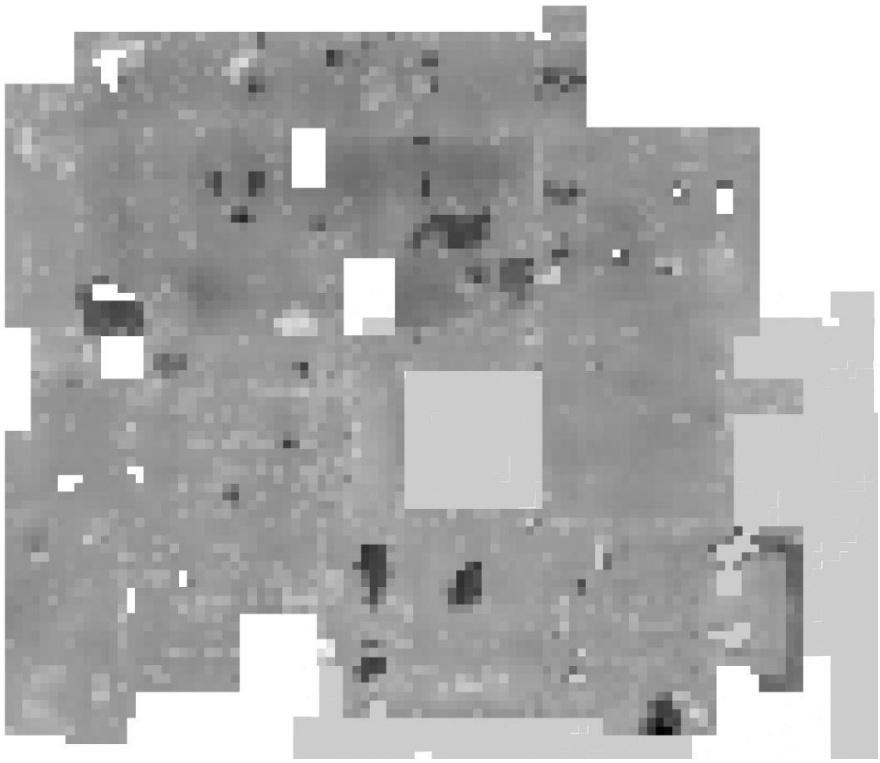


Now that we've re-accessed our family instances and our data point we can work on the part of the scrip that will add color to them. The node that will do that is "*Element.OverrideColorInView*" node. Which makes an instance and makes it any RGB color that we want in the specific view we want. It has two inputs, the first is our elements which we already have in Dynamo. The second is an RGB color object.



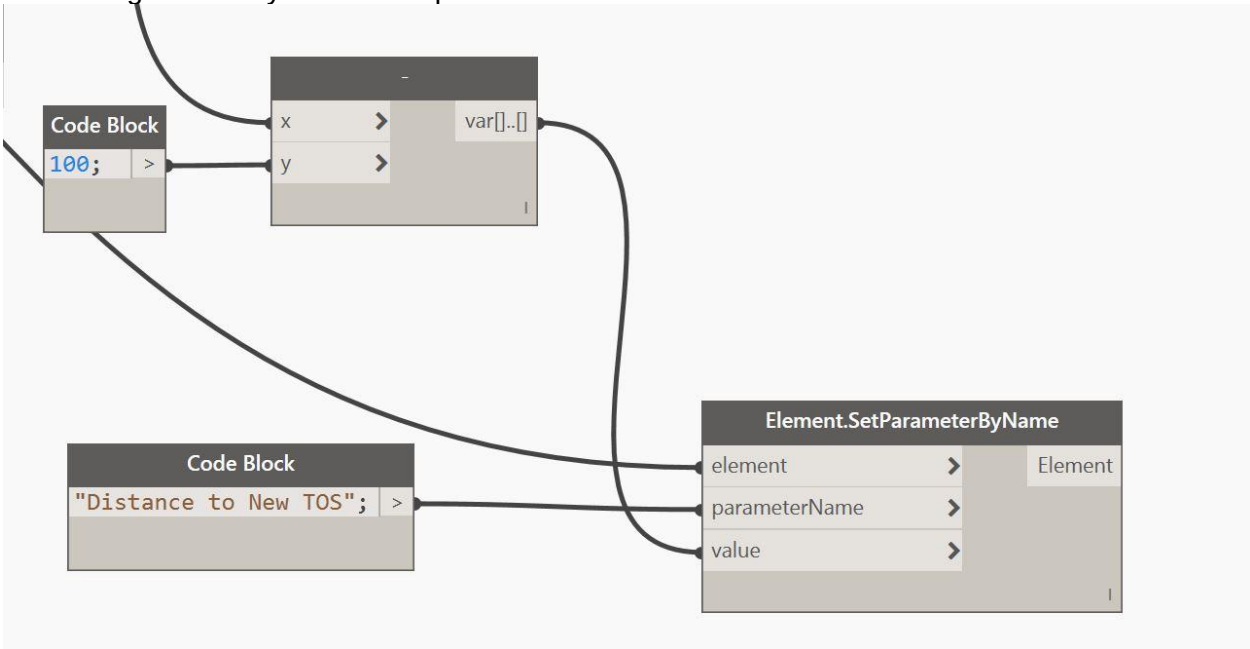
Creating that color in a programmatic way can be a bit tricky but also allows a lot of creativity. The way I'm going to do it is to use a "*Math.RemapRange*" node. Basically, can pass a list of numbers into it and re-distribute those numbers to a new minimum and maximum number. So our Z-values from our families can be mapped to 0-255 which allows for creating RGB colors.

Using a "*Color.ByARGB*" node we can make a color. With our re-mapped list of numbers and any other numbers we come up with we can really get creative with colors. When we map all red green and blue channels to the new mapped colors we get a greyscale map of the elevation changes. But we could also only manipulate one channel and create a gradient from green to black, or white to blue, etc.



Well How Much Concrete is it?

Now that we can all understand our data, we can place this image onto sheets and share it with our field staff. But we still haven't told them how much concrete do they need to top off this slab. We can again use Dynamo to help.



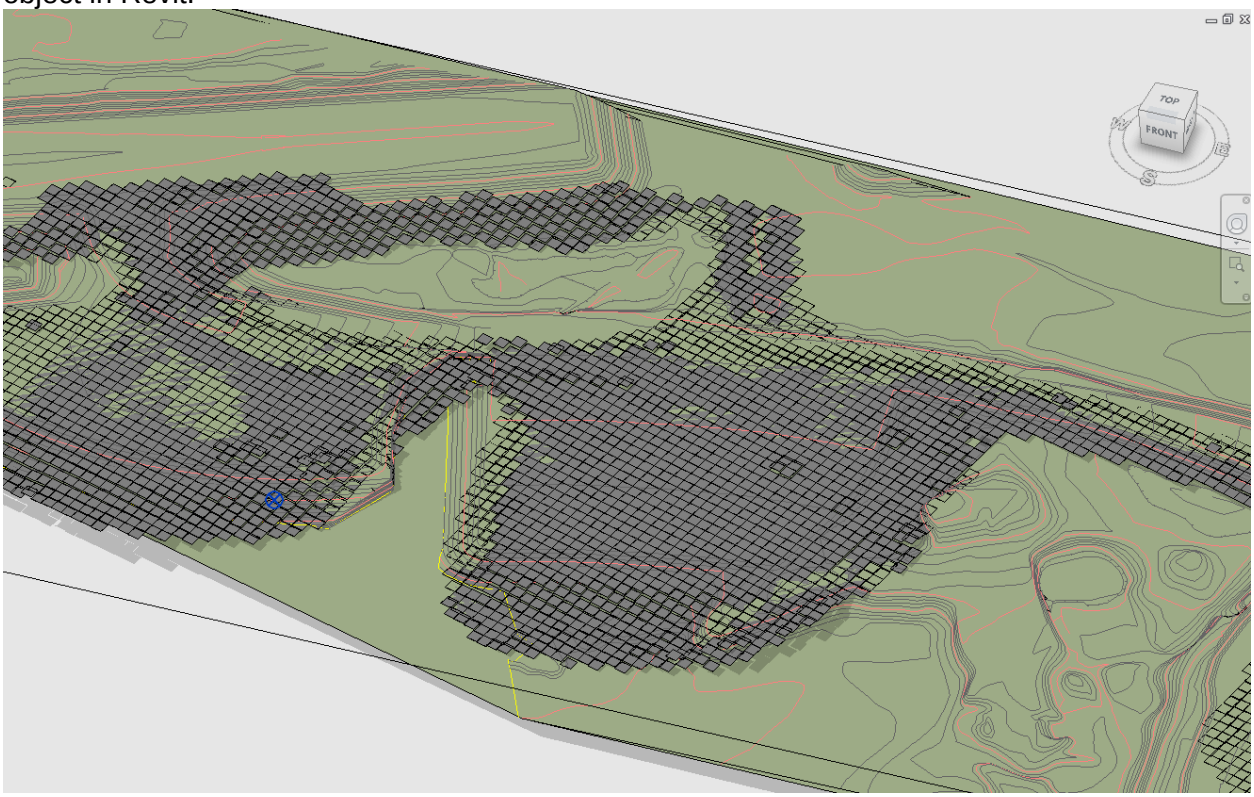
We're going to make use of Revit's scheduling abilities to schedule a parameter on each of these tiles. We're going to fill out this parameter in much the same way we did in the first example. However, rather than manipulating a parameter live we are going to store data in these parameters.

Each of our Color Tiles have a parameter called "Distance to New TOS" that we are using to store the value. Then we take the Z-value of each tile, use a subtraction node, and reduce the elevation by our designed elevation of 100. Now each tile has the distance it is from the designed elevation. With tags we could tag high and low spots using this parameter. With Revit Schedules and a little math we could calculate how much concrete we need.

Cut and Fill Analysis

A cut and fill analysis is quite similar to the slab flatness analysis we just performed. However, rather than choosing a new top of slab location arbitrarily, we are going to use a designed topography and see how far our existing needs to be modified to meet the design.

Again we started with a list of points that represent the existing topography. We also have the design topography which we brought in from Autocad and converted to a Topography object in Revit.



Now with Dynamo freshly opened we are going to import a DLL file that I created. This DLL file is for a custom node that I programmed in C#. The file is called ListPointProject.dll and we are going to use one of the nodes in the file for our next step.

Dynamo is really easy to add nodes and packages to if you are a developer. You can import Dynamo's library in C# and make use of all of the models and methods that are available to you in the Dynamo UI. However, you get to make use of the tooling and efficiencies of a traditional programming environment. If you aren't a developer, you can also contribute nodes of assembled existing nodes. This ability for everyone to contribute is probably a big reason why Dynamo is so successful.

After importing my DLL file, you will have a new dropdown on the left toolbar inside of Dynamo called *List Point Project.* We are going to use the node called "*CutAndFillHelpers.ClosestVerticeToPointZDiff*".

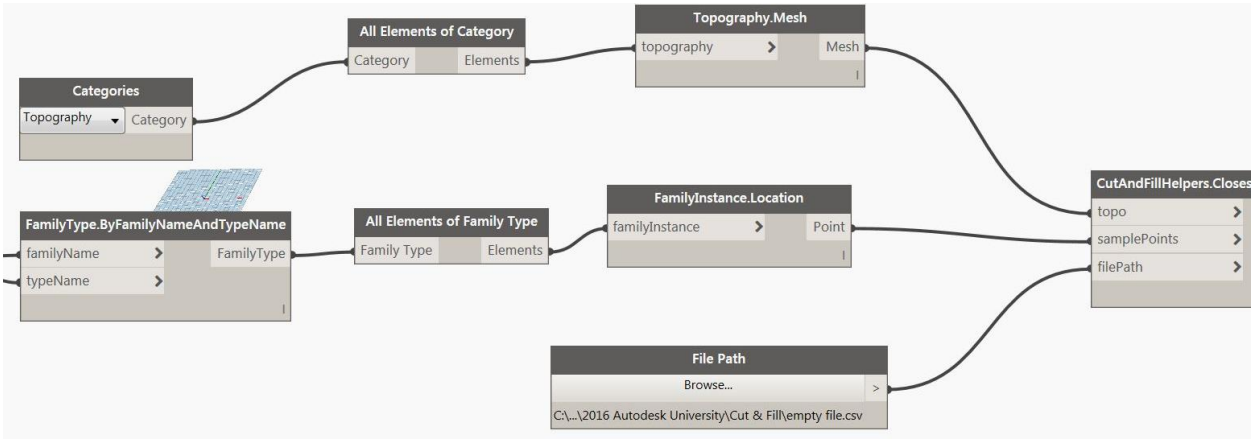
This node takes three inputs, a mesh extracted from topography, a set of sample points, and a file path object. It takes these inputs and writes a lot of information to a CSV file that we create. The function that its doing is something that would have been difficult to do with Dynamo nodes alone. Basically we need to take every one of those tiles, and find how far away it is from the topography object.

Takes one tile at a time and goes through each vertex on the mesh. It finds the closest vertex in the X and Y plane and calculates the difference in elevation. This is one way to estimate how much the existing topography needs to change in order to match the designed. This type of calculation involved “nested for-loops” and requires the computer to crunch a lot of data. It is probably possible to do this sort of thing exclusively in Dynamo, but was very challenging for me to think through. Writing this type of operation in C# is much more straightforward than it would have been in Dynamo and there are some performance improvements.

```
public static void ClosestVerticeToPointZDiff (Autodesk.DesignScript.Geometry.Mesh topo, Point[] samplePoints, String filePath )
{

    List<Point> meshPoints = topo.VertexPositions.ToList();
    //StreamWriter sr = new StreamWriter(@"C:\\Users\brendannichols\Documents\Projects\Trinity Forest\PointDistance.txt");
    StreamWriter sr = new StreamWriter(filePath);
    //List<Point> sortPoints = meshPoints.OrderBy(p => p.Z).ToList();
    for (int i =0; i < samplePoints.Length-1; i++)
    {
        Point samplePoint = samplePoints[i];
        Point moneyPoint = null;
        double currentDistance = 1000;
        for(int j = 0; j <meshPoints.Count-1; j++)
        {
            double xDist = (samplePoint.X - meshPoints[j].X);
            double yDist = (samplePoint.Y - meshPoints[j].Y);
            double distance = Math.Sqrt((xDist * xDist) + (yDist * yDist));
            if (distance < currentDistance)
            {
                currentDistance = distance;
                moneyPoint = meshPoints[j];
            }
        }
        string s = Convert.ToString(samplePoint.X) + "," + Convert.ToString(samplePoint.Y) + "," + Convert.ToString(samplePoint.Z) + "," + Conv
        sr.WriteLine(s);

    }
}
```



The rest of our script almost entirely stuff we've seen before. Except on the top, we are using a few nodes to get at our Topography mesh. We can use the categories node to select a category of elements and input that into the "*All Elements of Category"* node to get our topography. Then we use the node called "T*opography.Mesh*" to get to the geometry of the topography object. There are dozens of operators for meshes that don't exist for the higher level Topography object, so if you are ever working with topography you will probably need to do the same thing. Then we are putting those tiles into my node and giving a file path to an empty file to it as well. After you press run, this file will not be empty anymore. It will contain a series of rows, with column values of

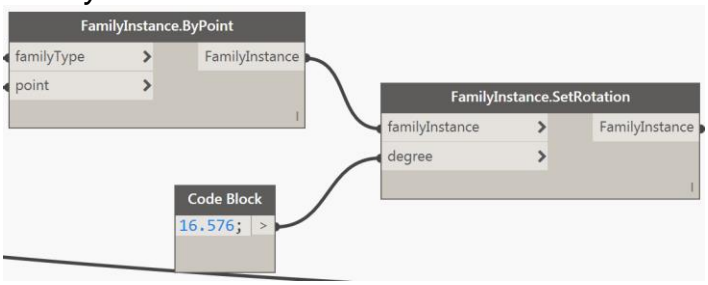
1- Tile X Value
2- Tile Y Value
3- Tile Z Value
4- Topography Vertex Z Value
5- Elevation difference
6- Distance from Tile to Vertex

This last value is a good indicator of how accurate the elevation difference calculated will be. If the Vertex and the Tile are very far apart then the Elevation difference will be less reliable.

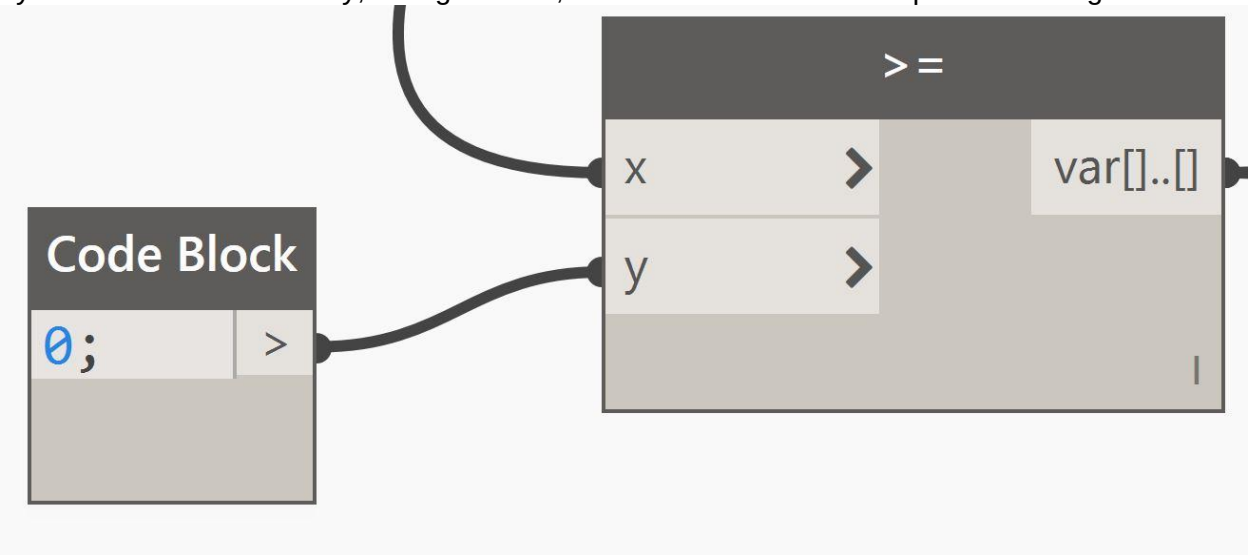
Better Visualizing our Data

Once again we are at the point where we have all the data for the solution of our problem. The issue is that we need to communicate it to others. This is really why this problem was a Dynamo problem and not a Programming problem. We get to use all of the tools and features available in Revit to document our solution.

Once again we are going to use a separate Dynamo file for this visualization. In the example files we are calling it "TilePlacer.dyn". We need to delete all of the existing tiles in Revit before running this script. Because this will re-create them. We are using the same techniques as shown in example 2 to place the tiles but we are doing if from the file we made in the last Dynamo file. With one modification, we are rotating the tiles with the "*FamilyInstance.SetRotation"* tile.



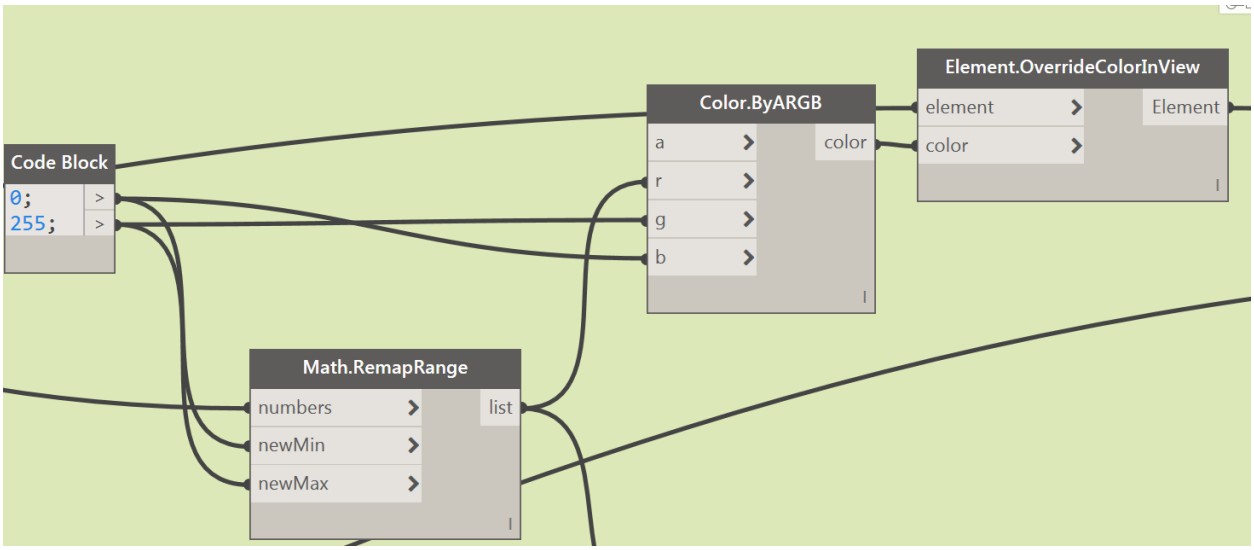
Now to visualize our cut and fill data, we would like to show the cut areas and the fill areas with different colors. This requires some conditional logic in Dynamo. We need to go through our data points and figure out which are which. This requires a technique in Dynamo called filtering by Boolean mask. Basically, we right a test, is the elevation difference positive or negative.



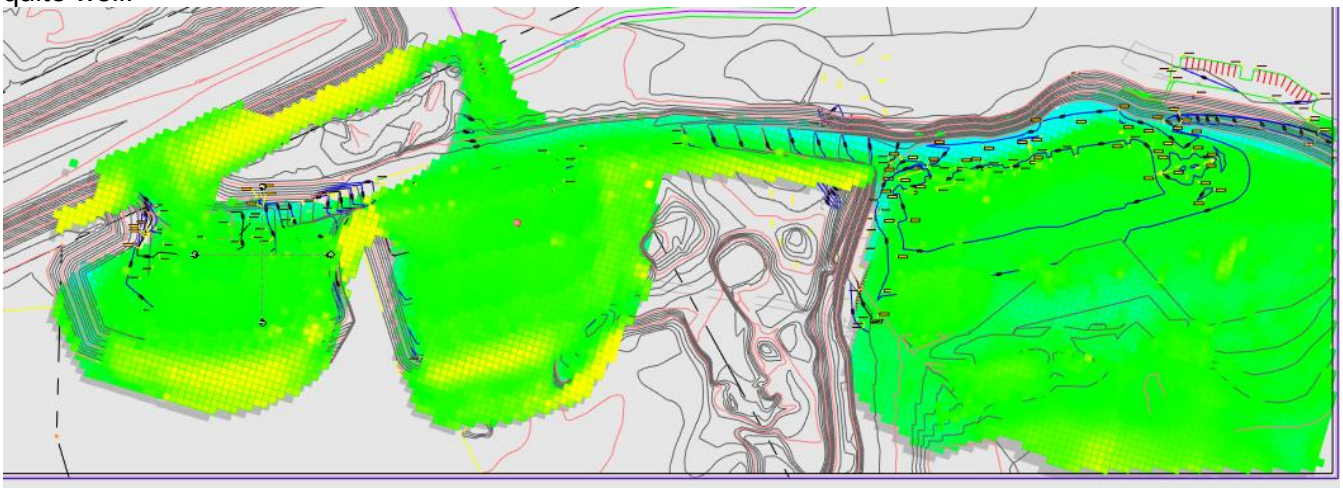
We select the correct columns from our earlier list and input them into the X value. And then input 0 into the Y value. If the elevation difference is positive it will return true and negative will return false. This new list of true false values can be used to filter another list. We use the "*Filter.ByBoolMask*" node to do that.

It splits our list of tiles into two separate lists. One for the positive values and one for the negative values. Now we can use these exports to color each differently.

We can use the same type of script we used in the second example for the coloration. But now we will need two of them to operate on each list separately. One of these lists we can vary with one color scheme, while the second list can have another.



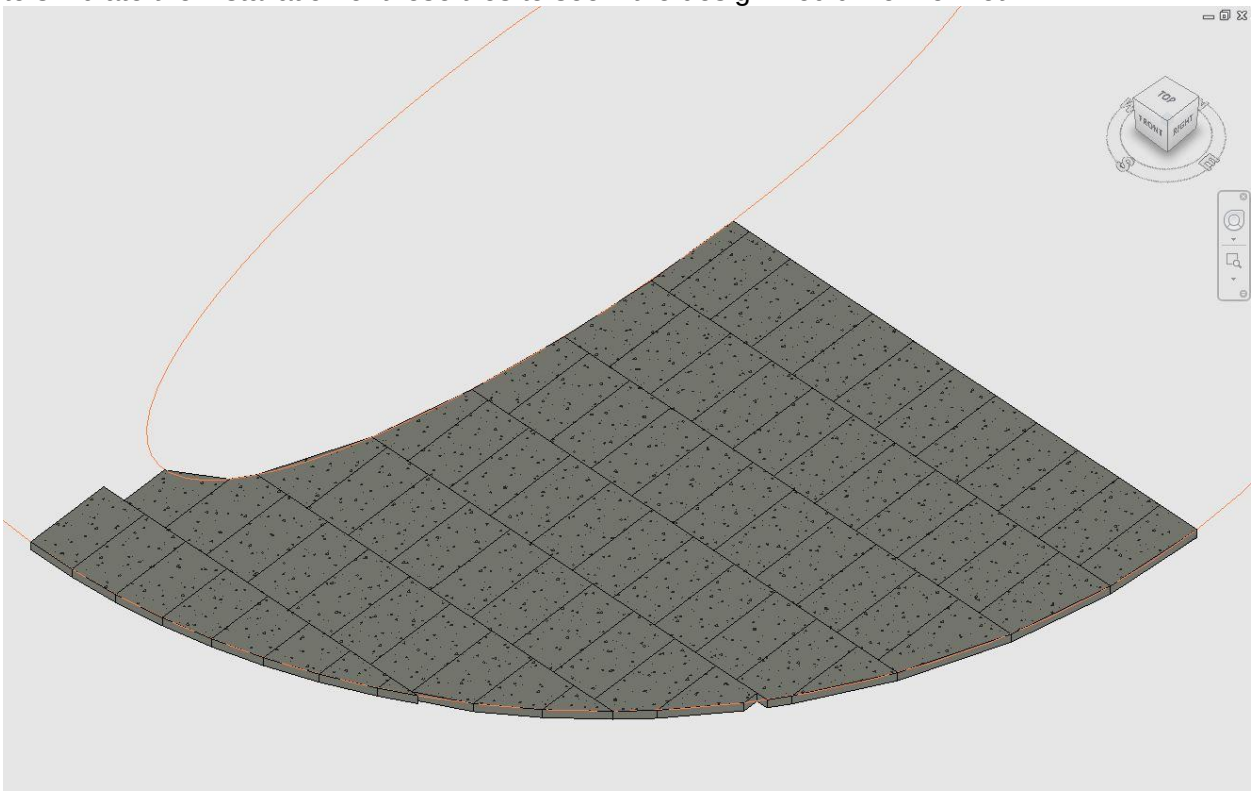
I chose to use green as a value closest to zero, or no cut and no fill. And then more blue as you need to fill more and more yellow as you need to cut more. I think this color option worked out quite well.



We can then use all the scheduling and annotation features of Revit to explain the analysis and data in greater detail.

Marble Tile Installation Simulation

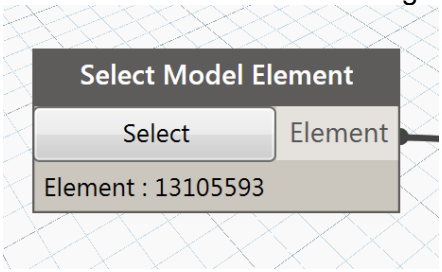
On one of our projects we were building a very high end fountain. The tiles that the water would run over were very expensive marble tiles and they were also an area where people had to be able to walk and kids had to play. The design left out critical slope information on these tiles and they had not been modeled by the Architect. We weren't sure if these large tiles would have fit the curvature and slope of the fountain without having some very sharp protruding corners. The project team was worried that they would have to cut the tiles in half to make them match the slope smoothly. Which would have been really terrible for the design intent. So we used dynamo to simulate the installation of these tiles to see if the design would work or not.



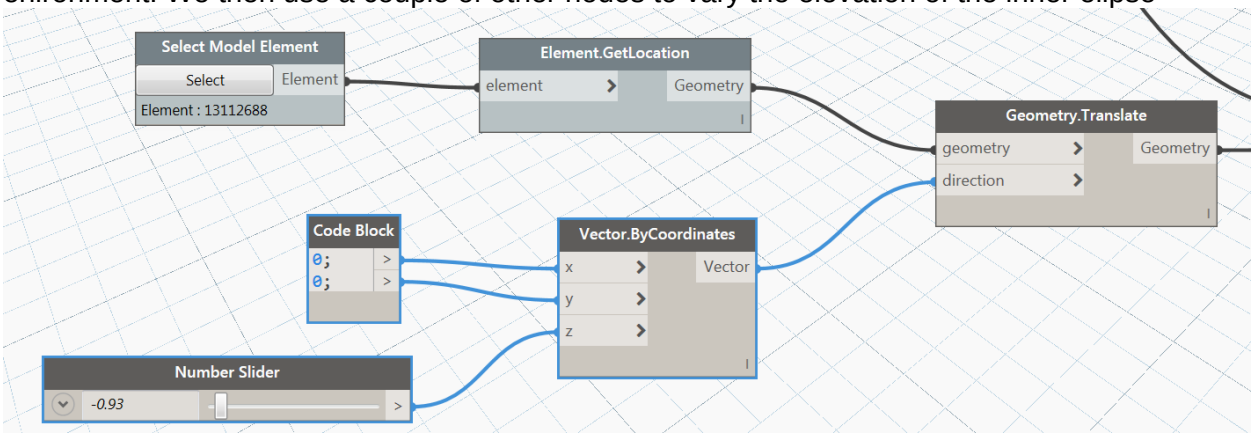
This is an image of the flatten tile pattern modeled in Revit. This is just the one quadrant that we tested representing the worst condition for sloping in the entire design. These are simply a series of floors approximating the shape and curvature of the designed fountain tiles.

The real challenge in this project is that the fountain has a form known as double curvature. Basically, it's shaped like the inside of a sphere. When you try to fit planar surfaces to this shape it's very difficult to reproduce the shape accurately. With smaller tiles or triangular tiles the problem is much easier to solve and the shape can be represented fairly accurately. Solving this type of problem is a very common challenge people use visual programming for. Lets look at how to solve this in Dynamo:

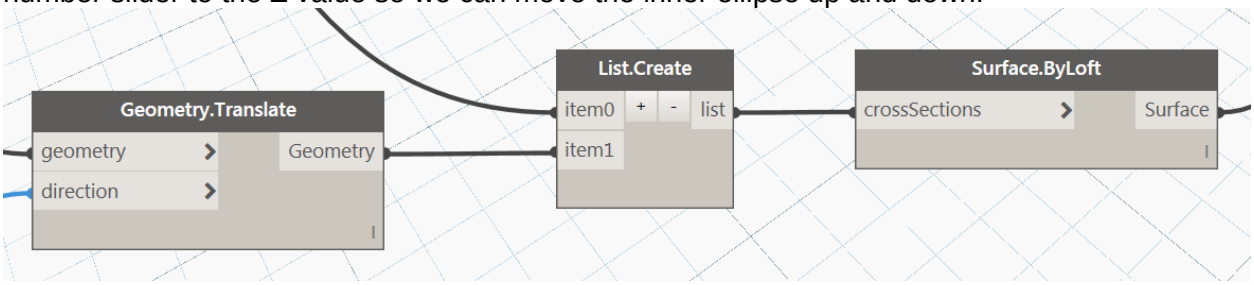
The first new node we are using is the "*Select Model Element*" node



This node allows you to manually select an element from the Revit environment and interact with it. I used them to bring the curves representing the fountaing footprint into the Dynamo enironment. We then use a couple of other nodes to vary the elevation of the inner elipse

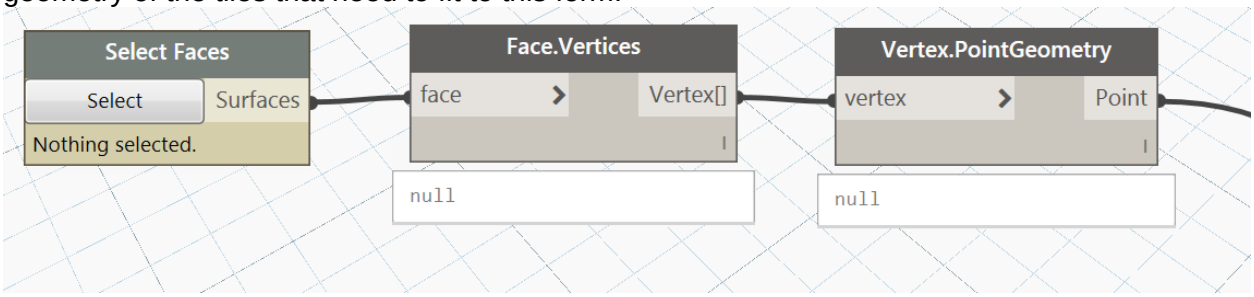


The "*Element.GetLocation*" node allows us to work with the geometry of the ellipse. We feed that into a "*Geometry.Translate*" node which allows us to vary the position of the geometry. Below that, we are building a vector to move the geometry. In this case we have hooked in a number slider to the Z value so we can move the inner ellipse up and down.



Then we make a list out of the inner and outer ellipses. We feed that into the "*Surface.ByLoft*" node. This node takes a series of shapes and will attempt to make a geometry that will transition from one to the other. Many times if you order your geometry incorrectly or don't have the right type of shapes this node will fail. However, when used correctly it can create shapes and forms that are very interesting and complex. Also when making lofts you are quickly heading into the territory of difficult to build structures and shapes. But it allows us to represent the "ideal" shape of the designed fountain accurately.

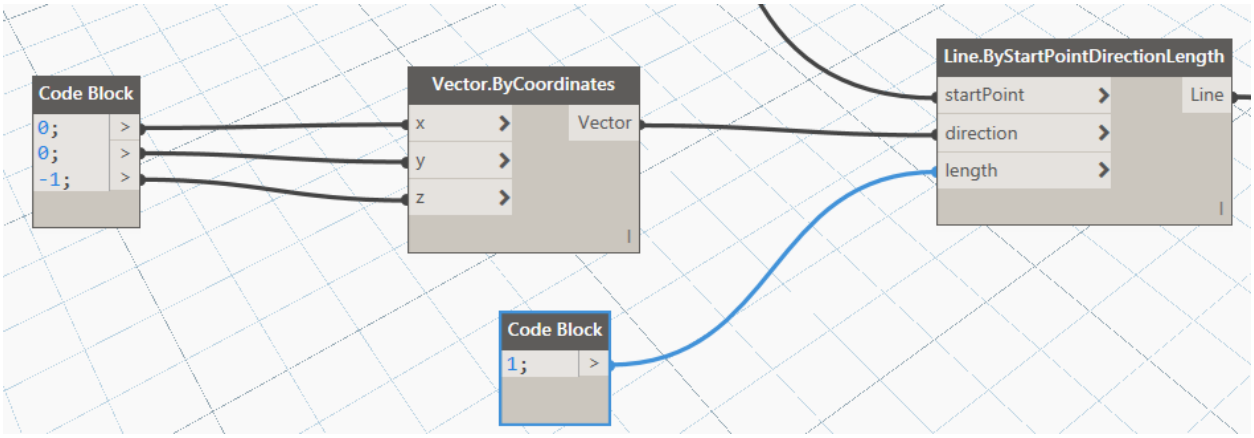
Now that we have the ideal form of the fountain, we need to bring in the shape and geometry of the tiles that need to fit to this form.



In this case we are going to use the "*Select Faces*" node. This node is similar to the select element node we used earlier but allows you to select multiple faces. Be sure to press the finish button on the toolbar after making a selection using this node.

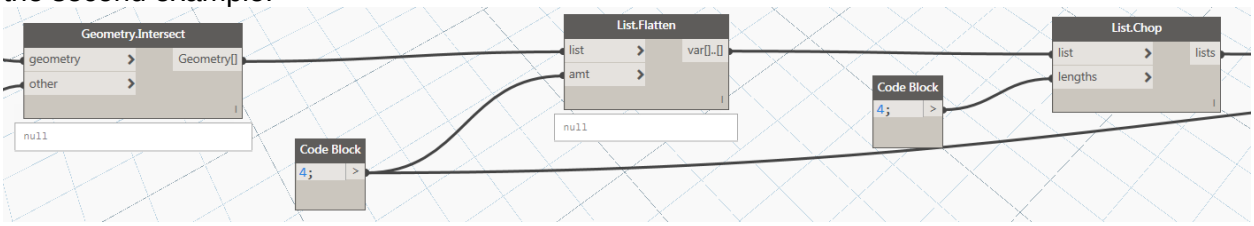
After we select the faces, we use the "*Face.Vertices*" node in order to get the corners of the faces. Then we use the "*Vertex.PointGeometry*" to convert these corners into points.

Eventually we need to draw these corners onto the curved surface of the fountain, we will use a "*Geometry.Intersect*" node in order to do that. However, first we need to make some type of Geometry out of these corner points that will intersect the fountain. To do that we use this series of nodes.



Again we are making a downward pointing vector and putting that into the "*Line.ByStartPointDirectionLength*". For this node, the vector serves as the direction, and then all we need to add is how long the line should be. It doesn't matter too much as long as it is intersecting the fountain.

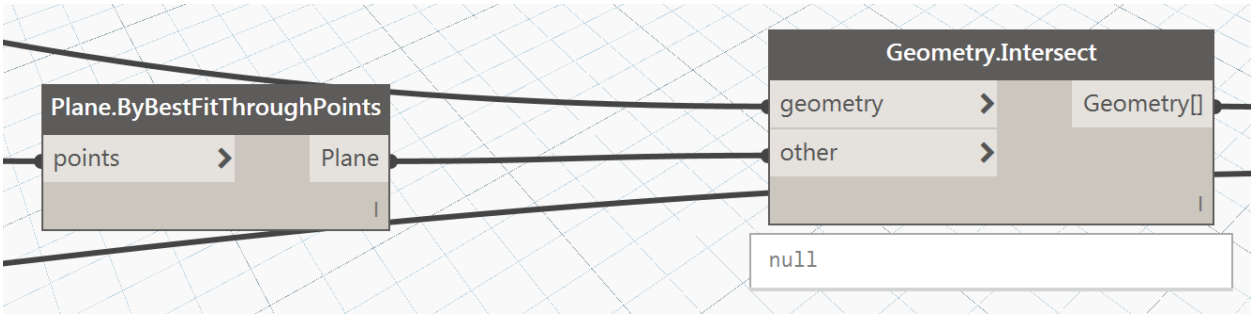
After drawing those lines, we feed them and the lofted geometry we made earlier into the "*Geometry.Intersect*" node mentioned earlier. And then we do a little more list cleanup similar to the second example.



At the end of the list cleanup, we are going to have a list of sub-lists. Each sub-list contains four points representing the location each tile intersects the fountain geometry. Which may seem like the solution we want but there is one more problem to solve before we can make new geometry with those points.

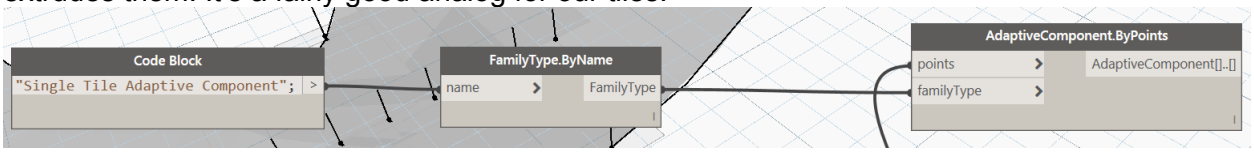
The problem is that these points are not on the same plane. Imagine if you pressed a book into the inside of a sphere. You could make three of the corners of the book touch the sphere but not all four. This principle is the same as what makes a chair rock on uneven ground. If the chair had only three legs it wouldn't rock. In our case, if our tiles had only three corners, or were triangular then they could match the surface of the fountain quite easily.

So now we need to come up with a solution, and Dynamo has an answer for us:



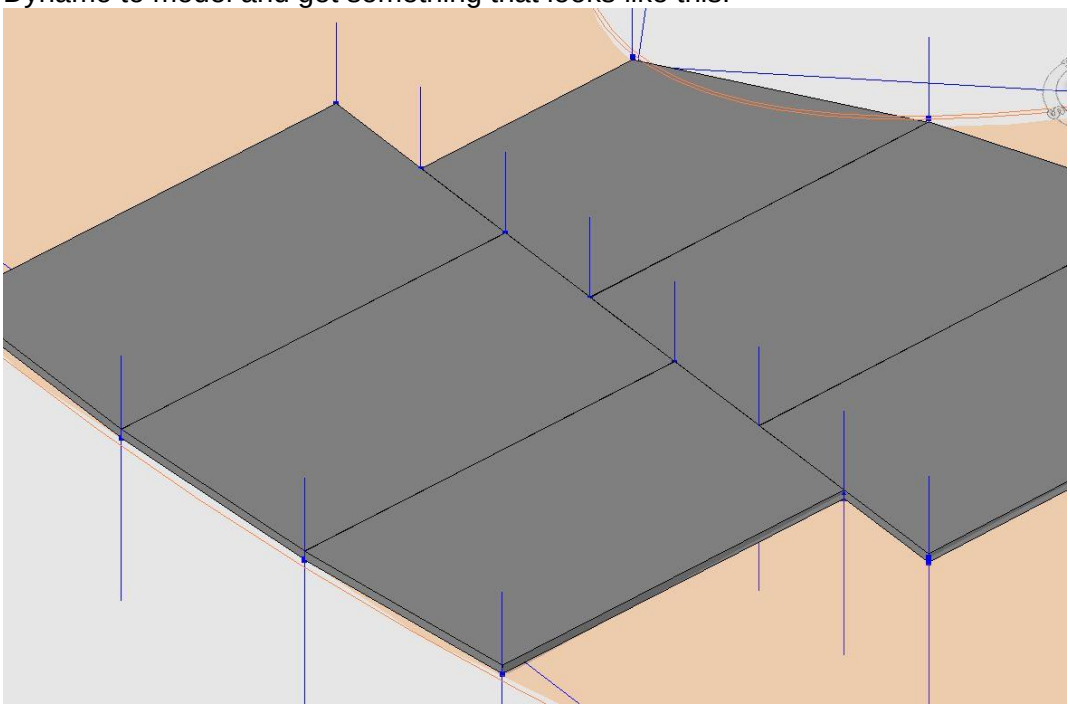
We use the "*Plane.ByBestFitThroughPoints*" node. Basically this will average the location of our four corners to a plane. It is sort of the digital shimmying you might do with your hands if you were installing these tiles yourself. After we know what plane the tile will sit on, we need to find the new corners. Again we use the "*Geometry.Intersect*" node but this time we intersect the planes we've just created with the lines we used earlier. The output of this is four points on the same plane on the corners of where the tiles were drawn to be. Which is our solution to this problem. However, we still haven't determined whether the design will work as drawn. We need to make these points into some sort of geometry inside of Revit. In this case, a typical family wouldn't work and instead I had to use an adaptive component.

This adaptive component takes four points and creates a rectangular shape and extrudes them. It's a fairly good analog for our tiles.



Placing an adaptive component in Revit is very similar to placing family instances. However instead of a single point for a location you need to input a list of points. It still has the same family type requirement. Both of those feed into the "*AdaptiveComponent.ByPoints*" node.

And that's it, with that node set up correctly, you can choose which tiles you want to use Dynamo to model and get something that looks like this:



In our scenario, it was determined that the differences between the tiles was not significant enough to cause any issues during installation. But because of this simulation we knew it to be the case way in advance and it set everyone's minds at ease. Below you can see a finished picture of the actual install.