AUTODESK **UNIVERSITY** 2014

Automation Prototyping

A Side-by-Side Comparison of Dynamo and the Revit API

Nuri Miller – Gehry Technologies, Inc.

SD5132-L The class will provide a walk-through of a sample project that is automated in the Dynamo visual programming language extension and using the Revit software API (application programming interface). This apples-to-apples comparison will showcase the best uses for each tool, helping attendees to draw their own conclusions as to how the Dynamo extension and/or the Revit software API may fit into their current practices. When is it appropriate to learn and implement the Revit software API? Is the Dynamo extension a flexible enough environment for project-based automation or is it a stepping-stone for the API? These are just some questions we will discuss and try to answer together while discovering some of the features of each tool.

LEARNING OBJECTIVES

At the end of this class, you will be able to:

- Discover the Dynamo extension implementation for project automation and how it compares to the Revit software API
- Understand Revit software's API functionality in relation to family placement and manipulation
- Understand when the use of the Dynamo extension may be most beneficial to your current practice
- Understand the Dynamo extension as a possible interface for prototyping Automation Solutions for eventual production in the Revit software API

ABOUT THE SPEAKER

Nuri joined the advanced technology consultancy of Frank Gehry in 2008. After launching a series of webinars for learning Gehry Technologies' parametric platform, Digital Project, Nuri led training seminars with leading design and construction firms. He has been heavily focused in the areas of knowledge transfer and capture throughout his time at GT. Beyond his earlier focus in design automation and direct to fabrication techniques, Nuri has been supporting collaborative systems for BIM project management and coordination in his role as a Technical Manager. He's always in search of repetition and patterns hiding within the frenzy of daily building design and construction. Nuri believes wholeheartedly that the AEC industry can be elevated through technology that takes on the burden of repetitious work and supports exploration of big data in building projects.

@nurimiller
www.nodelete.org
nmiller@gehrytech.com

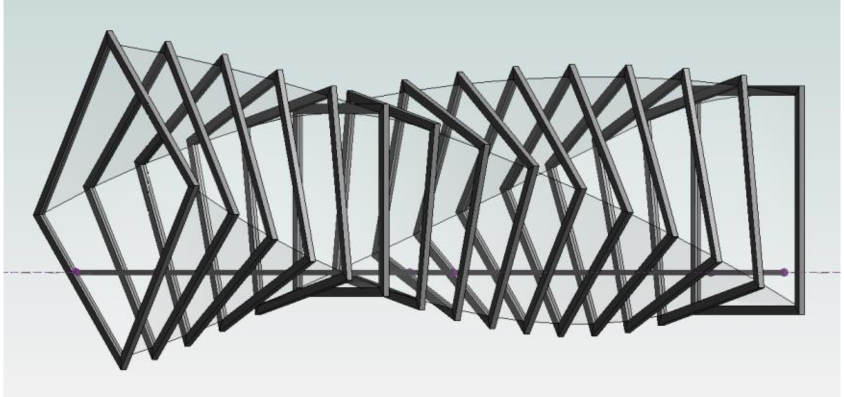
CONTENTS

SOME HELPFUL HINTS FOR DYNAMO

- *The search tool is key.* Dynamo nodes are well organized and follow a very logical hierarchy. Saying that it can still take a bit of time to navigate and so the search tool gets you there much faster most of the time.
- *Watch your output as you build.* When building your own workflows in Dynamo from scratch it's important to validate what you are building. You can do this by creating a Watch node and connecting it to any output. That's one option but Dynamo already has this functionality built-in to each node. Click the square in the lower right corner of any node to expand its output.
- *Double click to get code.* DesignScript is a unique computer language built from the ground up for designers of complex geometries which has now been incorporated into Dynamo. Resources for learning to write DesignScript are located under *dynamobim.org/learn*. Even small snippets of code can go a long way to streamlining any Dynamo script.
- *Sweep everything under the rug with Custom Nodes.* If your script has grown to a point where viewing it clearly in its entirety is challenging it may be time to consider custom nodes. Find logical groupings that can be isolated and create a new custom node for each. Establish the inputs and be clear on the outputs. Then you can replace a large tangle of Dynamo seaweed with simple and clean nodes you've built yourself.

EXAMPLE PROJECT OVERVIEW

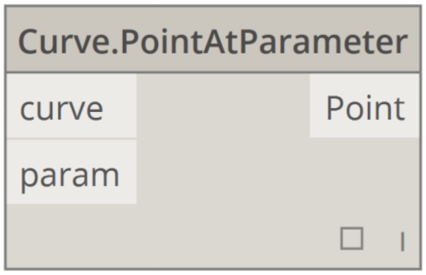
The example for this tutorial was originally developed as an introduction to parametric modeling in Digital Project on Gehry Technologies' gtwiki.org site. It's loosely based on a footbridge designed by Wilkinson Eyre Architects in London. The project's twisting form provides a useful vehicle for introducing parametric relationships and how a constraint solver deals with them. To understand how the relationships and constraints were developed as Revit Adaptive Components check out the post with YouTube video on the Dynamo blog.



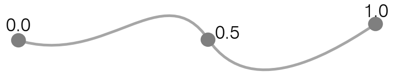
This tutorial's focus will be on automating the creation of Adaptive family instance which is typically referred to as *instantiation*. You will have an understanding of how each step of the visual script in Dynamo relates to a comparable set of routines written in Revit's .NET API. You can begin to make your own conclusions about how each may fit into your current workflows.

AUTOMATED INSTANTIATION THROUGH DYNAMO

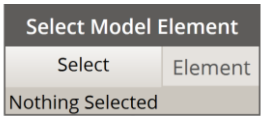
CREATING CONTEXT POINTS



The **PointAtParameter** is the basic mechanism for setting out the Adaptive Components locations. It creates equally spaced points along a curve based upon the curve parameterization (link). For a curve the parameter start value is 0.0 and end value is 1.0 with 0.5 being the center of its length.



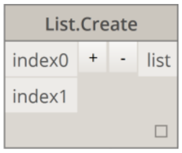
Inputs for ***curve***

1. 

   Type select model element + enter.

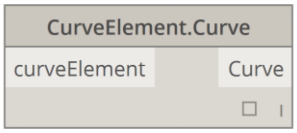
   The first step is to select the two context curves in the sample model file. Start by typing Select Model Element in Dynamo search to get the model element selection node.

2. Select the new Dynamo node and copy/paste to create a second node. Click the **Select** button on the node and then navigate to the 3D context model in Revit to choose the driving curve. Repeat this by assigning the second curve to the second node. The *Element ID* of each curve is now assigned to the nodes and should appear in the bottom left portion of each.

3. 

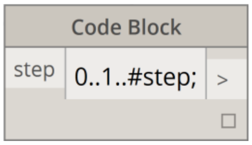
   Type create list + enter.

   Now route the two Select Model Element nodes connections.

4. 

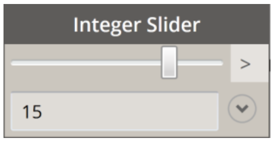
   Type curve element + select option under Revit > Elements > CurveElements > Query > Curve.

Inputs for ***param***

1. 

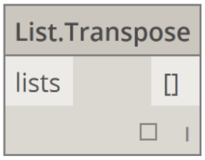
   ```
   0..1..#step;
   ```

   When you click twice in an empty portion of the Dynamo canvas a **Code Block** for DesignScript will automatically appear. Type 0..1..#step; into the input area of the Code Block. This tiny bit of code says, “Give me the set of numbers from 0 to 1 with a discrete number of steps”. This can be used in lieu of the Number Range node that is built into Dynamo.

2. 

Type integer + enter.

Output for ***Point***



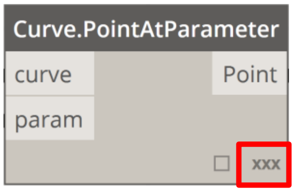
Type transpose + enter. This function will interchange the rows and columns. Before, the collection of points was made up of two lists. Those lists contain the point locations derived from the PointAtParamter function. In order to properly place the adaptive components we need a collection made up of each point pair from the two curves. Below illustrates this transformation.

	Parameter 0.1	Parameter 0.2	Parameter 0.3	…	Parameter 1.0
Curve 1	xyz	xyz	xyz	…	xyz
Curve 2	xyz	xyz	xyz	…	xyz

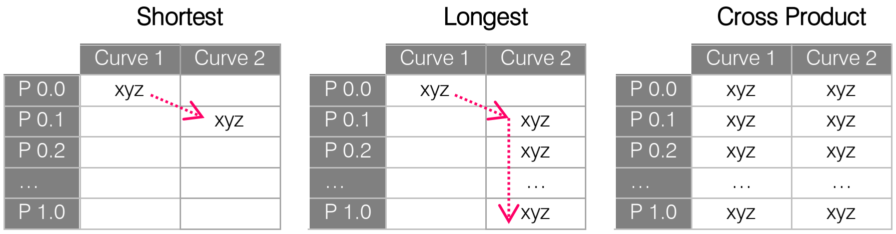
…becomes…

	Curve 1	Curve 2
Parameter 0.0	xyz	xyz
Parameter 0.1	xyz	xyz
Parameter 0.2	xyz	xyz
…	…	…
Parameter 1.0	xyz	xyz

Change the Lacing



Right click lower right corner to get pull-down menu with lacing options. By default lacing is set to **Shortest**. Instead select **Cross Product**. Also referred to as a Cartesian product (Wikipedia), this option gives you a set of all ordered pairs. Without doing this all possible pairs would not be available to use.



Shortest

	Curve 1	Curve 2
P 0.0	xyz	
P 0.1		xyz
P 0.2		
…		
P 1.0		

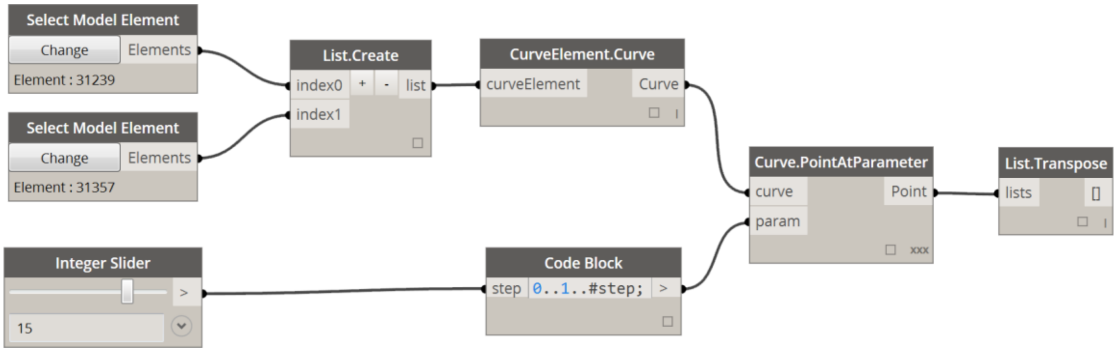
Longest

	Curve 1	Curve 2
P 0.0	xyz	
P 0.1		xyz
P 0.2		xyz
…		…
P 1.0		xyz

Cross Product

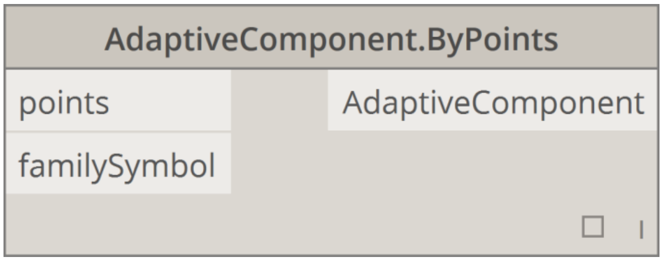
	Curve 1	Curve 2
P 0.0	xyz	xyz
P 0.1	xyz	xyz
P 0.2	xyz	xyz
…	…	…
P 1.0	xyz	xyz

Canvas Snapshot



Here is what you should see at this point in the process. Most of the basic logic is there. We just need to connect to the function which will place the Revit families. Simple enough.

PLACING THE ADAPTIVE FRAME FAMILY

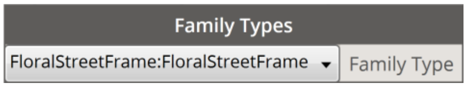


Type adaptive by points + enter. This Dynamo routine gives us the ability to instantiate the Adaptive Component by matching it's input points to the ones we've created on the input curves.

Input for ***points***

Connect the output of the List.Transpose node to get all the inputs points.

Input for ***familySymbol***

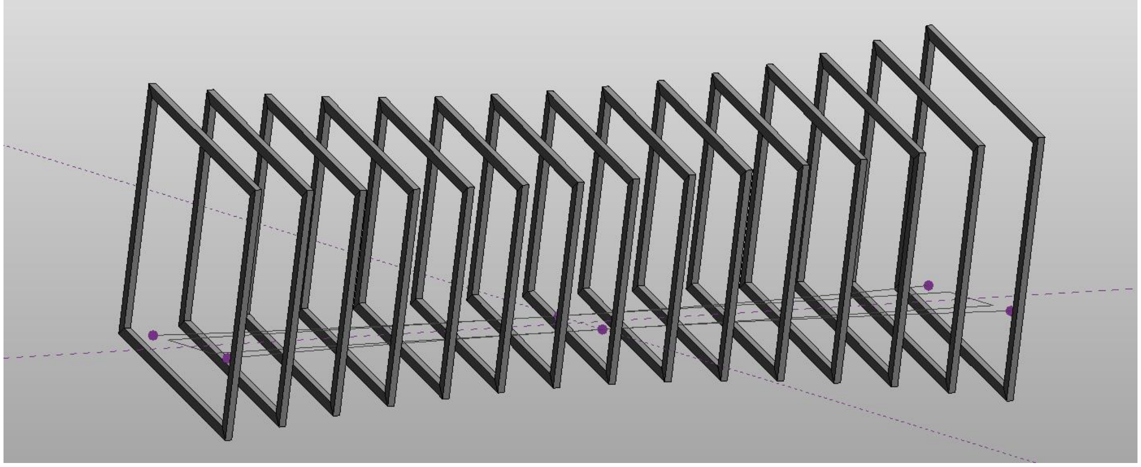


Type family types into Dynamo search and hit Enter to include the family type selection node. Click the pull down to select "FloralStreetFrame" from the list. Now make the connection to specify which family type to use.

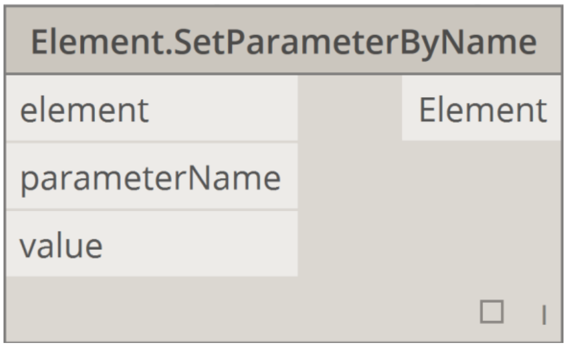
Test Run of the Script

There are now a minimum of components within the Dynamo script to see some family instances in Revit. Click the Run button to see the initial results.

You should see the frame instances at equal intervals along the input curves. Now the frames are just missing their rotation which can be controlled through the update of a family parameter. This will be shown in the next steps.



TWISTING THE FRAME



Type set parameter + enter. This will allow us to make the final adjustments to the Adaptive Component by setting the rotation parameter. Any type of family parameter can be read or written to through Dynamo.

Input for ***element***

Connect the output of the AdaptiveComponent.ByPoints node to pass the element objects to this node which will update the rotation parameter built into this Adaptive Component.

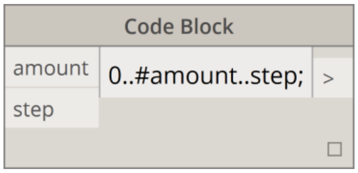
Input for ***parameterName***

Double click to create a Code Block and type “angle”; to tell this Dynamo node the parameter name you want to manipulate.

Input for value

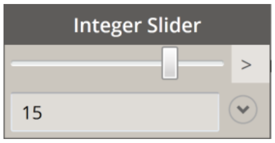
For this example the values will be a linear sequence of angles so that the frames create a smooth, twisting form.

1. Double click to create a **Code Block**. The code snippet for this Code Block will be similar to the number sequence one before. This time we will start the number sequence at 0 and establish a variable for the upper *amount* and another for the number of *steps* between.



```
0..#amount..step;
```

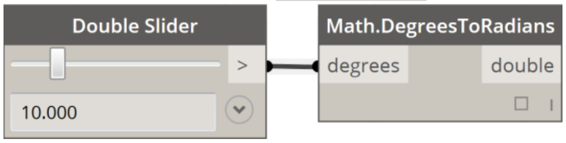
   a. The ***amount*** input will come from the same **Integer Slider** created before which establishes how many frames are instantiated.



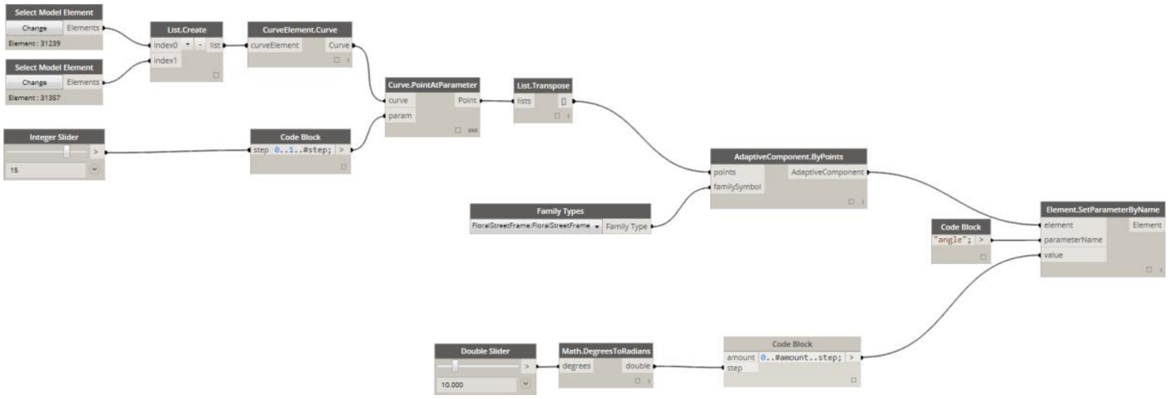
   b. The ***step*** input sets the degree amount per turn of the frame.

      i. Type slider to get the **Double Slider** node and set to a value of 10.000.

      ii. Start typing degrees to to get the **DegreesToRadians** node. This will convert the real number of 10.00 into radians, since angles in Revit are internally always in radians. (Note: An alternative approach would be to create a Code Block with the following code x*(Math.PI/180); to convert from degrees to radians.)



Canvas Snapshot

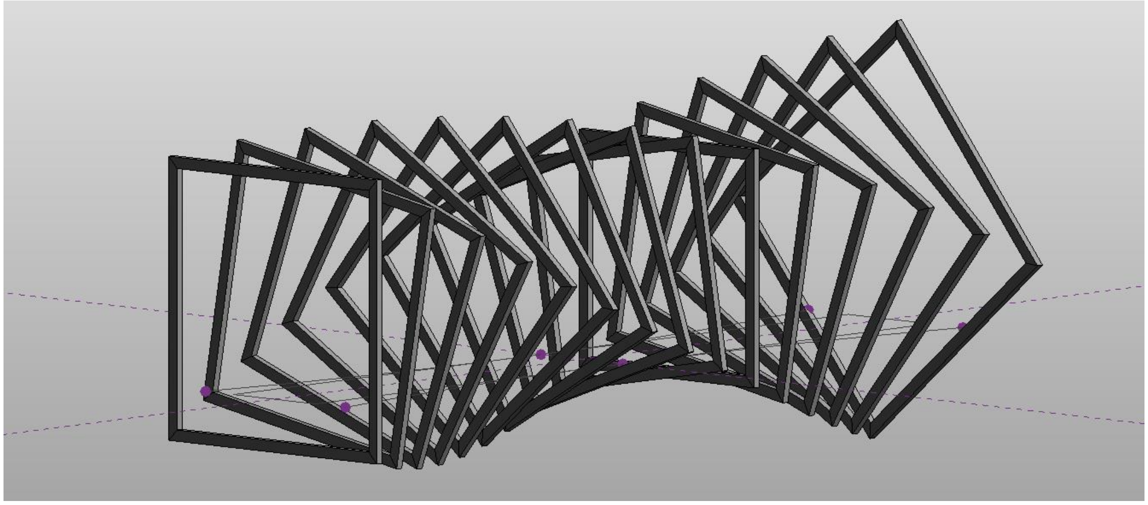


```
0..1..#step;
"angle";
0..#amount..step;
```

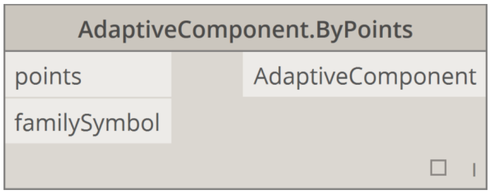
Here is the adaptive frame script is in its complete form. Get ready to press the **Run** button.

RUN THE FRAME SCRIPT

Run the script to see the final twisting frames. Try selecting one of the two original context curves, move an endpoint and re-run to see the frame update. If you want to see the script update automatically anytime a change occurs then check the **Run Automatically** box. Generally it's probably a good idea to keep this option un-checked to save on performance. At times it can be helpful to see live updates to test the automation routines you've designed or to do some design optioneering.

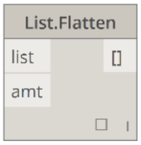


PLACING THE ADAPTIVE GLAZING FAMILY

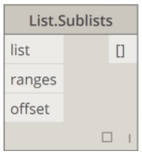


As before, we will be placing an adaptive component through input points on the curves. The difference, in this case, is that the glazing adaptive family takes four inputs points instead of two. The first task is to re-organize the list of points to support the four point family.

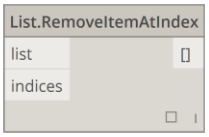
Input Point List

1. 

   Type list flatten + enter. Double click to add a **Code Block** and add 1;. Connect this to the **amt** input to indicate the layers of nesting to remove. This will take the collection of point pairs and create one long flat list of points.

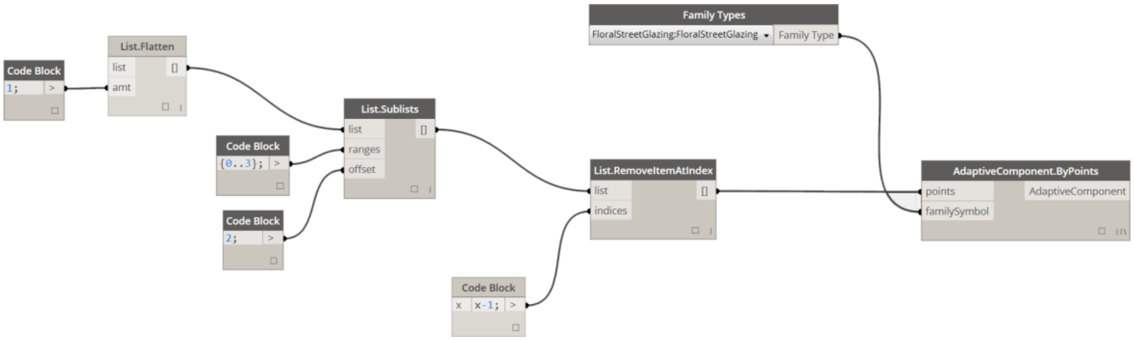
2. 

   Type sublists + enter. Connect the output of the previous node to the **list** input. Create a **Code Block** with {0..3}; and connect to the **ranges** input. This generates a set of numbers from 0 to 3 indicating the new range of point groups. Now the single flat list is a collection of points in groups of four. Add a **Code Block** with 2; and connect to **offset**.
   When this function goes through and groups the points the **offset** will tell it to shift by two positions every time. Therefore each grouping will have common points. Otherwise when you instantiate the glazing family there will be gaps between each instance.

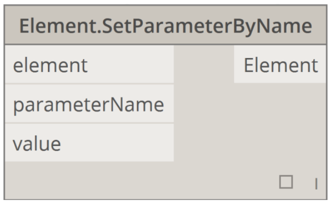
3. 

   Type list remove + enter. Connect the output of the Sublists done Add a **Code Block** with x-1; to tell Dynamo to stop the list creation just before the final set of points. Otherwise the last set of points would be a point pair and cause an error in the script.

Canvas Snapshot



GLAZING TWIST

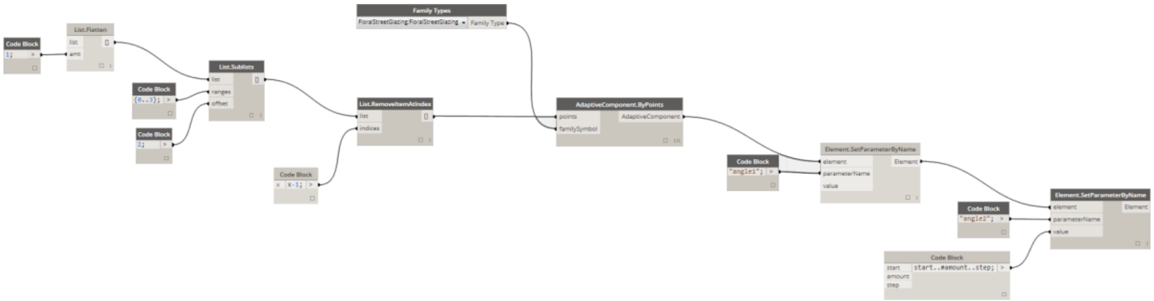


Type set parameter + enter. As before we need to define the angle parameter values. This time around there are two angle parameters instead of one since the glazing family is essentially the lofting of two frame shapes. Connect the output of the previous node to the element input.

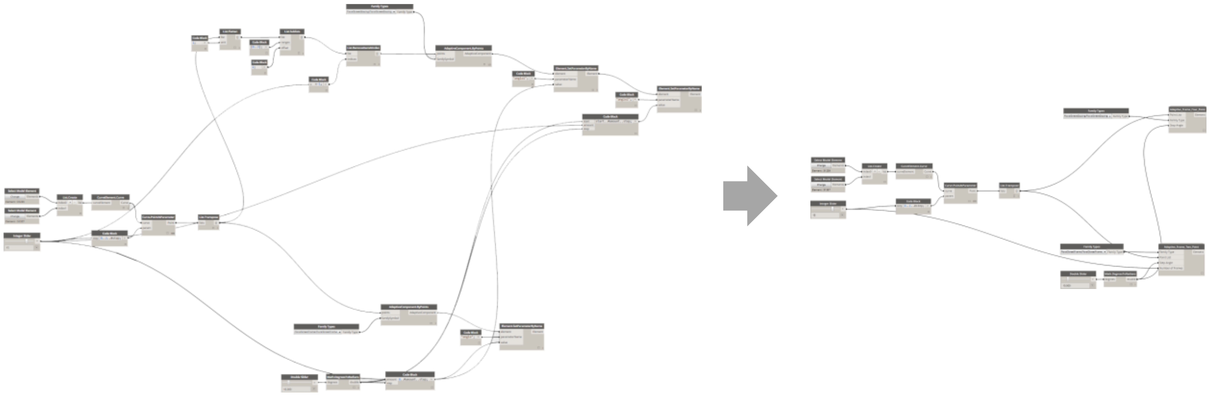
1. Add a **Code Block** with “angle1”; and connect to the **parameterName** input. Important! Now connect the list of rotations you previous made for the frame instantiation to the **value** input. You may need to zoom out and find that **Code Block** with the **DegreesToRadians** as input. Connect the output of that Code Block with says 0...#amount..setp; to the **value** input.

2. Copy/paste the nodes you created in the previous step. Disconnect the value input of the new **SetParameterByName** node. Connect the output of the previous **SetParameterByName** node to the **element** input of the new one. Add a **Code Block** with "angle2";.
3. For the value input create a new **Code Block** with start..#amount..step;. Connect the previous **DegreesToRadians** node to the **start** input and also the **step** input. The **amount** will come from the **Integer Slider** node.
4. Run Run the final script to see the results. If you get any errors please review the steps. Also you can refer to the complete Dynamo script provided.

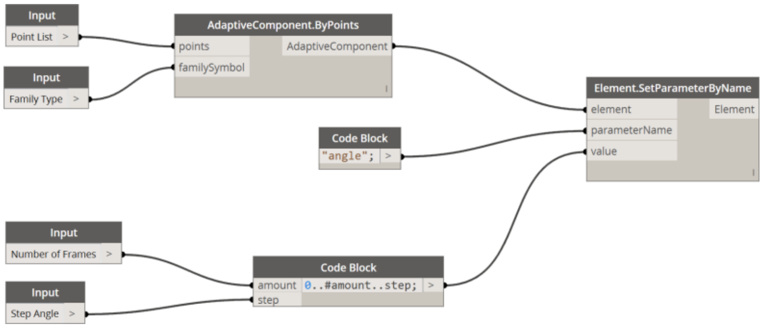
Canvas Snapshot



CUSTOM NODES



Implementing custom nodes consolidates this script by more than half. It went from a Dynamo script with 29 individual nodes down to 14. In the process if forces the script designer to group and organize so that others later on can manage the visual script more easily.



Select **Custom Node** under **File** > **New**. Choose a Name, add a Description and select the Category location so that your custom node shows up in a logical place in the node hierarchy. Type input + enter to define the inputs as shown above. Save the new DYF file and now when you return to your script it should appear in Dynamo.

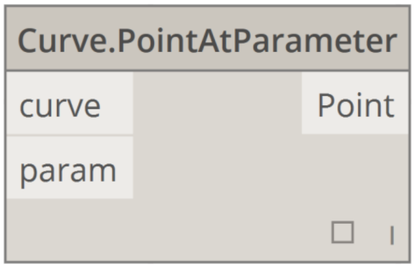
AUTOMATED INSTANTIATION THROUGH THE DOTNET REVIT API

We will start by examining the Revit API script I have provided and see how it relates to the Dynamo workflows we've just built. In some cases you'll see a one-to-one comparison. If you've never worked with it the API can seem a bit daunting. Hopefully these comparisons will help to digest all this new information and relate it back to something more familiar.

Go through the steps to create a new macro in Revit (link). Then copy/paste the script code provided. Make sure to select all lines below the section which reads #region Revit Macros generated code and replace them with the script for this class.

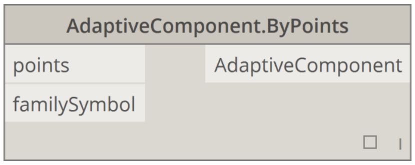
CORE FUNCTIONALITY COMPARED

The main functionality of this Dynamo script is made up of a few key routines. Each has a corresponding API code equivalent within the script provided.



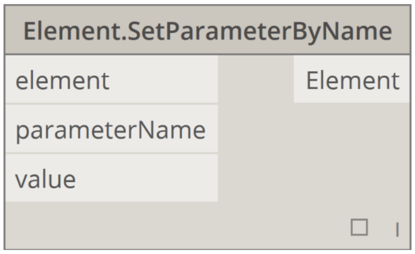
```
double curveParm = (double)i/numInstances;

pt.Position = curve.GeometryCurve
       .Evaluate(curveParm, true);
```

Revit has a function to evaluate the parameter of any curve and provide an XYZ point location. This has a direct correspondence with the Dynamo node we used. As mentioned before, sometimes Dynamo nodes and Revit API functions are directly equivalent.



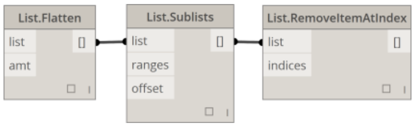
```
FamilyInstance instance = AdaptiveComponentInstanceUtils
        .CreateAdaptiveComponentInstance(doc, symbol);

ReferencePoint pt = doc
        .GetElement(placePointIds[j]) as ReferencePoint;

pt.Position = curve.GeometryCurve
        .Evaluate(curveParm, true);
```

In Revit through the API your first place an Adaptive Component in a default location. Then you collect e the point locations and set the input points of the Adaptive Component. You'll see we're using the same Evaluate routine. We are getting our point locations as we go instead of reading them from a point list passed to us from another routine. When you write code yourself you have this flexibility.



```
ParameterMapIterator it =  paramMap.ForwardIterator();

Parameter angleParm = it.Current as Parameter;

angleParm.Set(((angleDegStep*i)+angleDegStart)*degToRad);
```

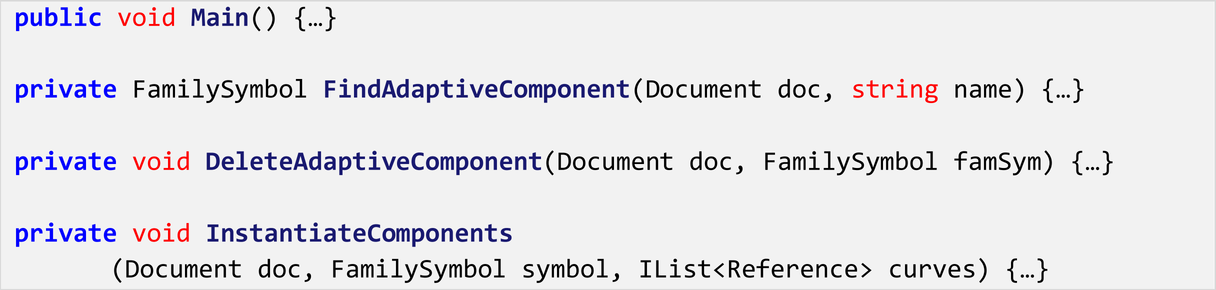
As in the Adaptive Component placement, for the rotation angles we are generating the radian values we need as we go. We don't need to take a list of angles as input. There's no need to even create a separate variable. Instead it is all included on the same line that sets the parameter value.



```
if (j < 2)
  curve = doc.GetElement(curves[j].ElementId) as CurveElement;

else {
  curve = doc.GetElement(curves[j-2].ElementId) as CurveElement;
  curveParm = (double)(i+1)/numInstances;
}
```

For the glazing Adaptive Component because it requires four input points we needed to go through several steps to reorganize the point list. Within the script this is taken care of through a single conditional statement. This statement says that if we are on the third or fourth input point (represented by the j variable) to shift the point evaluation. This is very similar to the approach we took in Dynamo but without manipulating lists of points.

ALL FUNCTIONS (PUBLIC AND PRIVATE)

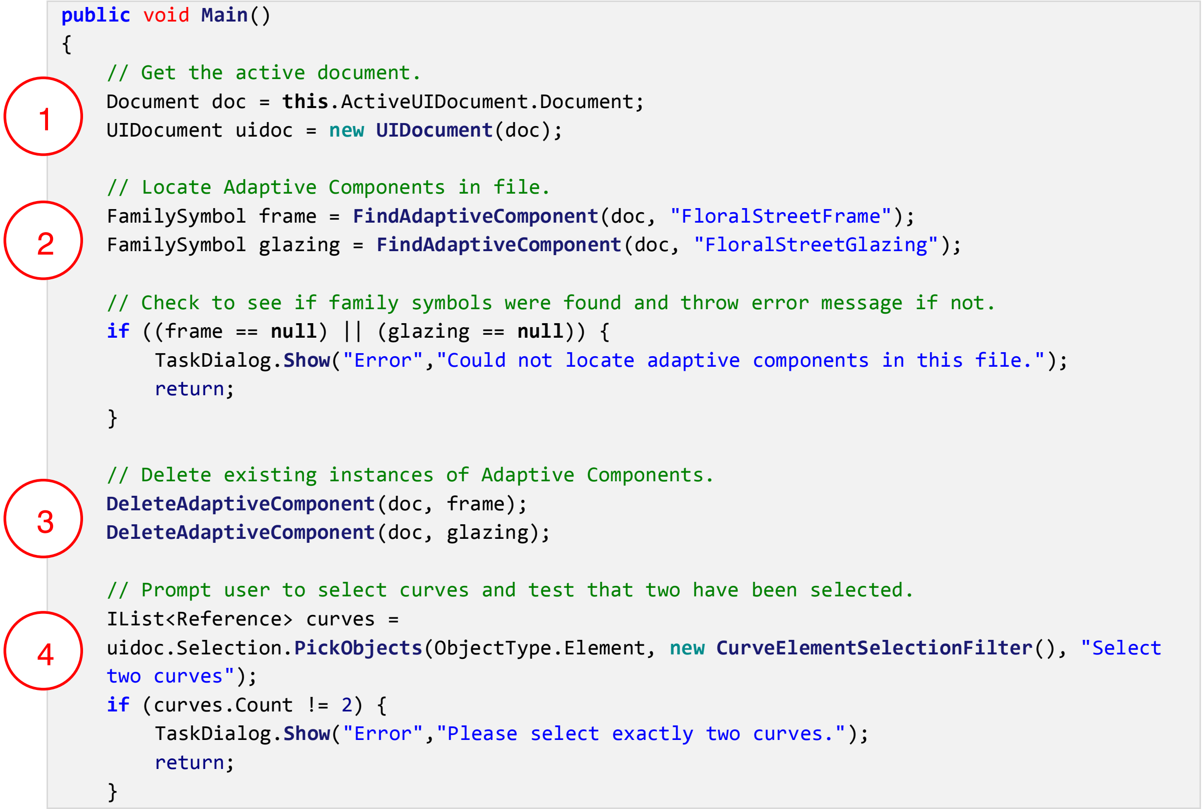


```
public void Main() {…}

private FamilySymbol FindAdaptiveComponent(Document doc, string name) {…}

private void DeleteAdaptiveComponent(Document doc, FamilySymbol famSym) {…}

private void InstantiateComponents
       (Document doc, FamilySymbol symbol, IList<Reference> curves) {…}
```

The entire script is made up of three private functions, which do most of the work, and one main public function to call them. They each have inputs and outputs defined in a similar way that the Dynamo nodes do. The main difference is that while the final Dynamo script was made up of 29 individual nodes the script is broken up into just three primary functions.

MAIN FUNCTION WORKFLOW

In stepping through this main function you will see there are number of things we will need to take care of ourselves that Dynamo was doing for us in the background. While writing scripts and plugins through the Revit API can provide significant flexibility and efficiency at also puts more of the burden on us to manage the working environment. This is something to keep in mind.



```
public void Main()
{
    // Get the active document.
    Document doc = this.ActiveUIDocument.Document;
    UIDocument uidoc = new UIDocument(doc);

    // Locate Adaptive Components in file.
    FamilySymbol frame = FindAdaptiveComponent(doc, "FloralStreetFrame");
    FamilySymbol glazing = FindAdaptiveComponent(doc, "FloralStreetGlazing");

    // Check to see if family symbols were found and throw error message if not.
    if ((frame == null) || (glazing == null)) {
        TaskDialog.Show("Error","Could not locate adaptive components in this file.");
        return;
    }

    // Delete existing instances of Adaptive Components.
    DeleteAdaptiveComponent(doc, frame);
    DeleteAdaptiveComponent(doc, glazing);

    // Prompt user to select curves and test that two have been selected.
    IList<Reference> curves =
    uidoc.Selection.PickObjects(ObjectType.Element, new CurveElementSelectionFilter(), "Select
    two curves");
    if (curves.Count != 2) {
        TaskDialog.Show("Error","Please select exactly two curves.");
        return;
    }
    // Instantiate the adaptive components.
    // Transactions are need whenever any change is made to the Revit model.
    Transaction t = new Transaction(doc, "Place Adaptive Component");
    t.Start();
5   InstantiateComponents(doc, frame, curves);
    InstantiateComponents(doc, glazing, curves);
    t.Commit();
}
```

(1) Get Active Document

Before anything can be pushed or prodded through the API you need to establish a connection with the active document. Dynamo goes through this process too. When you first fire up Dynamo it will state in the console Dynamo is now running on document: and then point to the active file. This gives it the starting point it needs to then do something.

(2) Find Adaptive Components

```
FilteredElementCollector col = new FilteredElementCollector(doc);
            col.OfCategory(BuiltInCategory.OST_GenericModel)
                .OfClass(typeof(FamilySymbol));

fs.Family.FamilyPlacementType == FamilyPlacementType.Adaptive

fs.Name == name
```

When locating the Adaptive Components we need there are few things we known including the fact that it will be a Generic Model and its placement type will be Adaptive. This helps us to locate the Family Types (or “Symbols” as they are referred to in the Revit API) we need to instantiate.

The FilteredElementCollector is heavily used. No matter the task you’ll most likely start by needing to find something. It could be a building element, view, sheet, etc. To keep things moving smoothly it’s important to be conscious of whether you are using Quick Filters or Slow Filters. Quick Filters can find elements without expanding them into memory unless they match the criteria. This keeps large search tasks moving lightning fast.

In this example we’re using both a Category and Class filter. They are both Quick Filters and so no reason for concern. To understand this topic further here is a section on “Applying Filters” from Autodesk’s documentation (link).

(3) Delete Existing Adaptive Components

Here we need to do a little house keeping. Without us even noticing it Dynamo already clears the slate every time we re-run a script. It used to be that Dynamo would store the IDs of the those elements it generate in the current session and clear them every time. Now, after a recent update, Dynamo keeps tabs on elements in generated in previous sessions and deals with those too.

```
FamilyInstanceFilter filter = new FamilyInstanceFilter(doc, famSym.Id);

ICollection<ElementId> familyInstances = collector.WherePasses(filter).ToElementIds();

doc.Delete(familyInstances);
```

The method we're employing here is to search for all existing instances of a Revit family type we define. It clears out all instances in the file. If we were concerned it might take some family instances with it we didn't want to delete we might want to take a different approach. Instead we might store the Element Ids in much the same fashion that Dynamo to know which to delete. For this example our current method is sufficient.

(4) Select Curves for Input

Every time the script runs we give the user the option to pick which inputs curves they want. When picking objects you need to be careful to limit what the user can choose. This can be done in two ways. In the first scenario you can let the user choose anything they see on the screen, check to see what they chose and then throw an error message if the objects are not the types you need (i.e. points, levels, solids, etc.). The other is to limit the choices upfront giving the user immediate feedback on what your script is looking for. This can be done through a class which inherits the ISelectionFilter. In the example below the allowed elements only return a result if the chosen element is a CurveElement. In this way the limitation is immediate and interactive.

```
    public class CurveElementSelectionFilter : ISelectionFilter
    {
        public bool AllowElement( Element e )
        {
          return e is CurveElement;
        }

        public bool AllowReference( Reference r, XYZ p )
        {
          return true;
        }
    }
```

(5) Instantiate the Adaptive Components

This portion of the script is probably the most similar to the Dynamo workflow. The main distinction would be that be that both the placement of points and setting the rotation occur together for each instance. This is a common difference between writing your own scripts and assembling a visual script. In visual scripting routines are generally broken up into their smallest functional parts for easy reuse. This means that managing collections of objects becomes an important task in visual scripting as one routine passes a collection to the next. Sometimes it's more manageable to do several operations at once and writing the code yourself allows that.

Below is a pseudocode (human readable version) of the InstantiateComponents function to illustrate how placing and rotating the family instance is setup as several inline operations instead of separate functions passing collections.

```
For each instance {
   Create a new adaptive component in the center of the model space;
   Set the adaptive component placement points based upon the calculated curve
   parameter position;
   Access the “angle” parameter and update;
}
```

SO WHAT DOES THIS MEAN FOR YOU?

Dynamo or the Revit API? Which one to choose?

Fortunately, you don’t need to make that choice. Dynamo and your own custom code are integral and can be used to complement each other. Furthering knowledge and experience in one can only help to better understand the other. Learning more about the API provides a backstory and in-depth understanding while Dynamo can be the agile, collaborative prototyping environment.

Whichever one you decide to focus on at any time depends on a few factors including:

- Packaging & Collaboration
- Customization
- Updating & Deprecation
- Code Management
- User Interface/Experience

As we wrap up the lab at Autodesk University 2014 we will discuss this further. There’s a lot to discuss on this topic and hopefully we will find opportunities to continue it further. It is only just now beginning to evolve. As efforts grow to introduce automation into people’s lives, e.g. ifttt.com or at code.org, the conversation is broadening and becoming more inclusive.