

CBW228260-L and AS229738-L

## Computational BIM Workshop—Beginner

Paul F. Aubin, Paul F. Aubin Consulting Services, Inc.

Kyle Martin, Gensler

### DESCRIPTION

Combine logic, geometry, math, and BIM (Building Information Modeling) with Dynamo visual programming. We'll use Dynamo to drive a Revit document and interoperate with other applications, data sources, and modeling tools. If you already have some experience with Revit and are looking to expand its capabilities with computation, this workshop is for you. Basic understanding of computer programming and scripting is helpful, but not required.

### SPEAKERS

**Paul F. Aubin**—Paul F. Aubin is best known as the author of many books and video training courses for Revit and other Autodesk tools. He has 28 years of experience in the Architectural industry and has worn many hats in that period, from designer, to CAD Manager, technologist, and trainer. He continues many of these in his current role as an independent Architectural consultant based in Chicago. Paul is the founder of ChiName; the Chicago Dynamo Community.

You can see Paul's **Dynamo** and **Revit** online training courses at:

<https://www.linkedin.com/learning/instructors/paul-f-aubin>

Be sure to check out: <https://www.linkedin.com/learning/dynamo-practical/>

**Kyle Martin**—Kyle Martin is a Digital Design Manager at Gensler in Boston where he is primarily responsible for BIM Management and Dynamo implementation. In addition, he oversees several initiatives related to computational design, digital fabrication, building performance, data visualization, and virtual reality. Martin is also an adjunct instructor of AEC technology courses at the Boston Architectural College and co-founder of design technology advocacy organizations that include Dynamo-litia Boston, ENCODE(Boston), the Design Technology Throwdown at Architecture Boston Expo, and the beyondAEC Symposium and Hackathon.

Some of the content in this paper was developed by last year's workshop instructors. Special thanks for their contributions.

Rachel Williams      twitter: @RachelWilliams

Sol Amour      twitter: @solamour

# COMPUTATIONAL BIM WORKSHOP



## AGENDA FOR WORKSHOP

Monday		Topic	Wednesday	
Start Time	Duration		Start Time	Duration
8:30 AM	0:15	<b>Welcome</b> (Introduce team, session and agenda)	1:00 PM	0:15
8:45 AM	0:10	<b>Getting Started</b> (Dynamo Player)	1:15 PM	0:10
8:55 AM	0:20	<b>Hello World!</b> (Dynamo Basics, Nodes, Wires, Library, Dynamo Geometry version, Revit version)	1:25 PM	0:20
9:15 AM	0:10	Code, Logic, Pseudo Code, Programming and Program Execution	1:45 PM	0:10
9:25 AM	0:20	<b>Placing a Single Family</b> (Basic selection, Errors, Ranges, Lists, Lacing)	1:55 PM	0:20
9:45 AM	0:10	Bonus items	2:15 PM	0:10
9:55 AM	0:15	<b>Morning Break (Monday Only)</b>		
10:10 AM	0:30	<b>Placing Multiple System Families</b> (Working within Revit hierarchy, System vs. Component families)		
10:40 AM	0:30	<b>Placing Loadable Families</b> (Array of points, introducing logic, using a package node)	2:25 PM	0:30
11:10 AM	0:35	<b>Importing and Exporting Data</b> (Using Excel) (Get and Set Parameters)	2:55 PM	0:35
11:45 AM	0:15	<b>Use Excel to Edit Revit</b>	3:30 PM	0:15
12:00 PM	1:00	<b>Lunch! (Monday Only)</b>		
		<b>Afternoon Break (Wednesday Only)</b>	3:45 PM	0:20
1:00 PM	0:30	<b>The "I" in BIM</b> (Performing Calculations on Model Data)	4:05 PM	0:30
1:30 PM	0:30	<b>Using CAD Data to build Revit Geometry</b> (Extract CAD, geometry analysis, element creation)	4:35 PM	0:30
2:00 PM	0:45	<b>A Curtain Wall Challenge</b> (Adaptive components, geometry analysis)	5:05 PM	0:45
2:45 PM	0:15	<b>Q&amp;A</b>	5:50 PM	0:10
3:00 PM		<b>Wrap up</b>	6:00 PM	

## INTRODUCTION

This session is for beginners. This means that we will start here with some of the basics. If you are already using Dynamo and creating your own graphs, then some or all of the examples that follow might be review for you. But if you are like us, there are always tidbits you can learn even when reviewing familiar material. So we encourage you to follow along regardless of your personal level of expertise.

## SOFTWARE VERSION

This session will use **Dynamo for Revit** and will use **Dynamo version 2.0.1** running in **Revit 2019.1** (see Figure 1). More information on Dynamo versions and installation in the appendix.

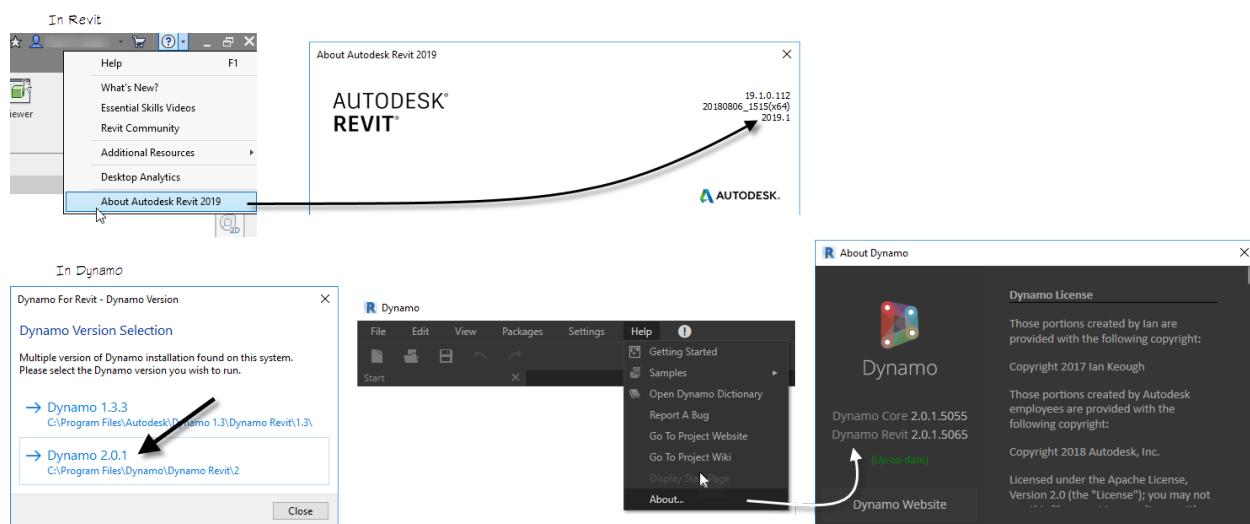


Figure 1

## LEARNING OBJECTIVES

We have a few goals for this session:

- Introduce visual scripting using Dynamo for Revit and just exactly what that jargon means.
- Dynamo for Revit allows you to perform tasks in Revit. It cannot invent new functionality that is not already possible in Revit. Usually this means that Dynamo is ideal for problems that would be tedious or time-consuming to do manually. For example, tasks that require little to no brain power but take thousands of clicks to execute, are the perfect place to automate using Dynamo.
- Dynamo should be thought of as a tool you use alongside of Revit. You do not have to perform a task only manually in the Revit UI or only with Dynamo. You can do some things directly in Revit and some things with Dynamo. In other words, use the right tool for the job. And Dynamo is a perfect complement to daily Revit workflows.
- We want you to understand how to structure a problem. This means clearly deciding what you want to do and then breaking it down into the component steps required to achieve it.
- Most Dynamo scripts follow a simple structure: Input, Process, Output. This means that you bring data and geometry into Dynamo. You ask Dynamo to do something to those

items; process or manipulate them in some way, and then you output the results; usually back to Revit.

## THIS IS A LAB

The session is **hands-on**. This means that we want you to follow along. Don't just sit back and watch. You will get MUCH more out of the session by participating and following along. Detailed steps are included here, so you should be able to keep up. But if you do get behind, there are several intermediate versions of the files provided throughout the handout. They look like this:

**SAMPLE FILE:** You can open a file completed to this point named: ***00\_Sample File.dyn***.

These files should help you keep pace with the session without getting too far behind. You can always perform the steps shown here later after the lab. So, it is highly recommended that you use the catch-up files instead of getting way behind in the live lab session. But we have lab monitors roaming around as well. **Feel free to raise your hand and ask for help.**

## APPENDIX, BONUS MATERIAL AND ADVANCED USERS

If you are comfortable with Dynamo already, some of the material in this session will already be familiar to you. So, feel free to stray off on your own and experiment freely. Take the time to experiment a little and break things! This is a great way to learn.

There is an appendix at the end. And some bonus content there and elsewhere in the handout that we will not get to in the live session. You can review that material after the session or even in the live session if you get ahead of the group.

## ADDITIONAL RESOURCES

There are many online resources that can help you on journey to learn more about Dynamo. There is a list in the appendix that you can consult before and after the session. But we would like to share a few useful resources with you right away:

[dynamobim.org/](http://dynamobim.org/) - the Dynamo home page.

<http://primer.dynamobim.org/en/> - an introductory tutorial on getting started with Dynamo.

[dictionary.dynamobim.com](http://dictionary.dynamobim.com) – this is a searchable database for Dynamo functionality.

[dynamonodes.com](http://dynamonodes.com) – all about dynamo nodes.

## GETTING STARTED

This is a hands-on lab. So, let's start right in and do some hands-on exercises. The first example will show us how to access Dynamo from within your Revit interface and use a premade script using the **Dynamo Player**.

## ACCESSING DYNAMO

To access Dynamo, you first launch Revit, open a project or family document and then click the Dynamo button on the Manage tab (see Figure 2). This will launch Dynamo in a separate

window. Dynamo requires an active Revit document<sup>1</sup>, and it will “tether” itself to this document<sup>2</sup>. So be sure to open a file first. It can be a project or a family. (Don’t do this yet, the hands-on steps begin with the next topic).

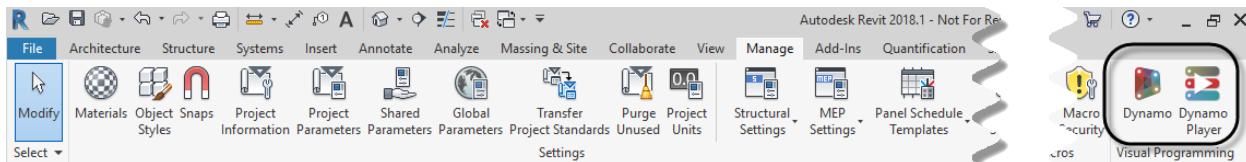


Figure 2

Working with two screens is recommended. If you don’t have two (such as here in the lab), you will want to position Revit so you can see the model behind the Dynamo window. You can also run Dynamo Player (using the button on the ribbon right next to Dynamo). We’ll do this in the first exercise.

### USING DYNAMO PLAYER TO LAUNCH A GRAPH

Dynamo Player allows existing Dynamo graphs to be run in Revit through a simple interface. This is a convenient way to share Dynamo graphs with your team and reuse them on multiple projects. Users do not need to have extensive experience with Dynamo to use scripts through Dynamo Player.

Let’s run our first Dynamo script from the Dynamo Player. To do so, we first need to launch Dynamo Player and then point it to the location where the scripts for today’s lab are stored.

1. Launch Revit 2019.1.
2. Create a new empty project file from the *Default Architectural template* file<sup>3</sup>.
3. On the Manage tab, on the Visual Programming panel, click the Dynamo Player button.

There are a few icons across the top.

4. Click the first icon to Browse to a folder containing Dynamo scripts.
5. Browse to: **[Local location of the lab datasets]**<sup>4</sup> and then click OK (see Figure 3).

<sup>1</sup> Dynamo for Revit requires Revit to run. There is a stand-alone version of Dynamo called Dynamo Studio which does not require Revit and runs as a separate stand-alone application. We will not be using Dynamo Studio in this lab.

<sup>2</sup> Be sure to save or synchronize your current Revit session before launching or running a Dynamo graph. If Revit or Dynamo crashes, it will bring down both.

<sup>3</sup> We are using the default United States Imperial project template in this lab. Other templates should work as well, but differences in settings and units may make for variations in your results and what is pictured here.

<sup>4</sup> We will instruct you in the lab where to locate the dataset on your lab computer. If you are using your own laptop, or following these instructions after AU, you can visit: <http://paulaubin.com/au/> to find this paper in PDF and a downloadable ZIP file containing the dataset files.

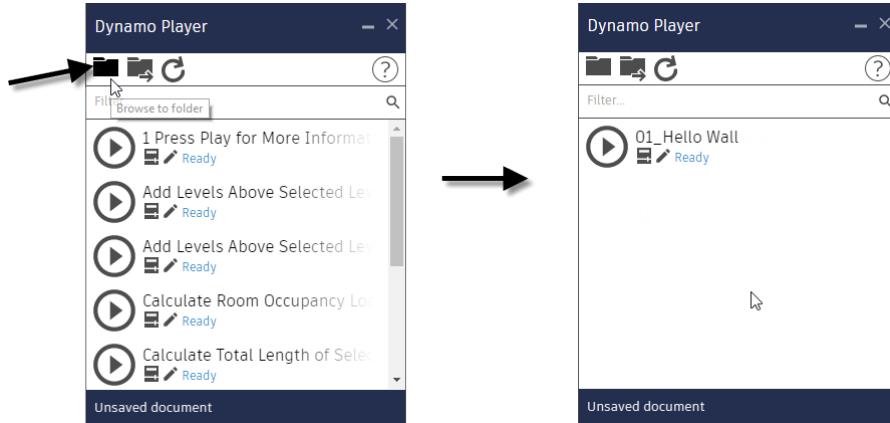


Figure 3

To run a graph from the Player, you simply click the play button next to the script you want to run.

6. Click the play button next to: **01\_Hello Wall** (see Figure 4).

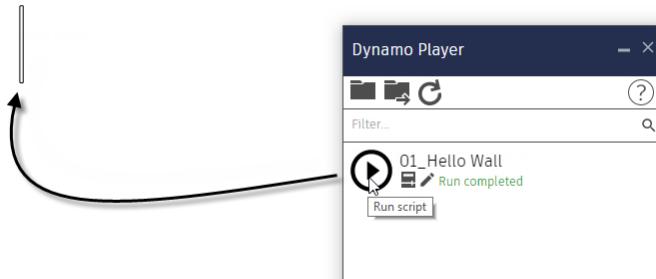


Figure 4

You will see a wall appear onscreen in the current Revit view window.

7. Select the wall onscreen in Revit and then on the Properties palette, look at the Comments field (see Figure 5).

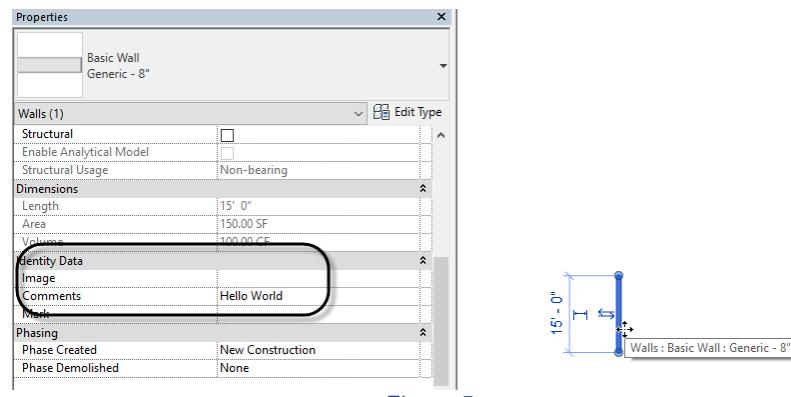


Figure 5

In computer programming, the classic first example is to write a program that displays the words "Hello World" onscreen. Think of this as the Revit version of that.

### EDIT DYNAMO PLAYER INPUTS

There are inputs in the script that control what the comments say and where the wall is drawn onscreen. We can edit those directly from Dynamo Player.

1. Click the small Edit inputs icon directly beneath the name of the script (see the left side of Figure 6).

This will display the inputs for the graph.

2. At minimum, change the: **Value of Comments** (optionally change some other inputs too).
3. Click the small arrow to return to the script name, then click the run button to run the script, again. Select the wall and check its Properties (see the right side of Figure 6).

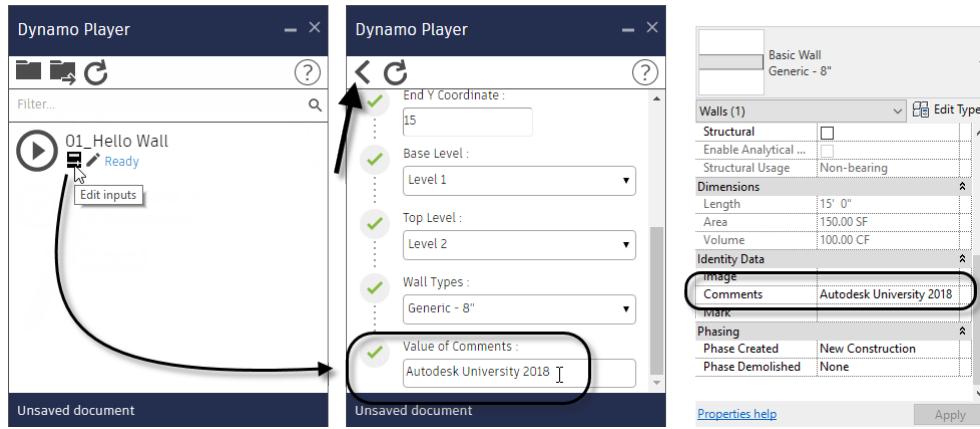


Figure 6

This graph has inputs for the X and Y values of the two endpoints of the wall, the wall type that is used, the base and top levels and of course the value of the comments field. You can change any of these. (To change levels, you would need to add some new levels in the Revit file). If you don't erase the wall, when you run a second time, it will create a new wall.

So now that you have seen a simple graph in action, let's close Dynamo Player, and launch Dynamo itself and create this graph from scratch.

4. Close Dynamo Player.
5. Delete the wall(s) created by Dynamo Player.

### RUNNING DYNAMO

Now that we've seen Dynamo Player and the results of an existing graph with inputs, let's try our hand at creating our own graph by launching the Dynamo application. We will recreate the graph we just ran piece by piece. Dynamo will launch in a separate window. Dynamo requires an active Revit document, and it will "tether" itself to this document. So be sure to have a file open first. It can be a project or a family. Working with two screens is recommended. If you don't have two, you will want to position Revit so you can see the model behind the Dynamo window.

1. From the Manage tab, launch Dynamo and then click the New button on the welcome screen, to create a new Dynamo file (called a "graph") (see Figure 7).

## Dynamo

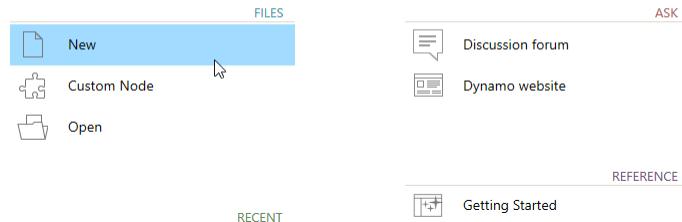


Figure 7

The Dynamo interface will load with menus across the top, the library on the left and the main graph editing window filling the remainder of the screen. Some navigation tools will appear in the upper-right. At the bottom of the screen is the execution options button (see Figure 8).

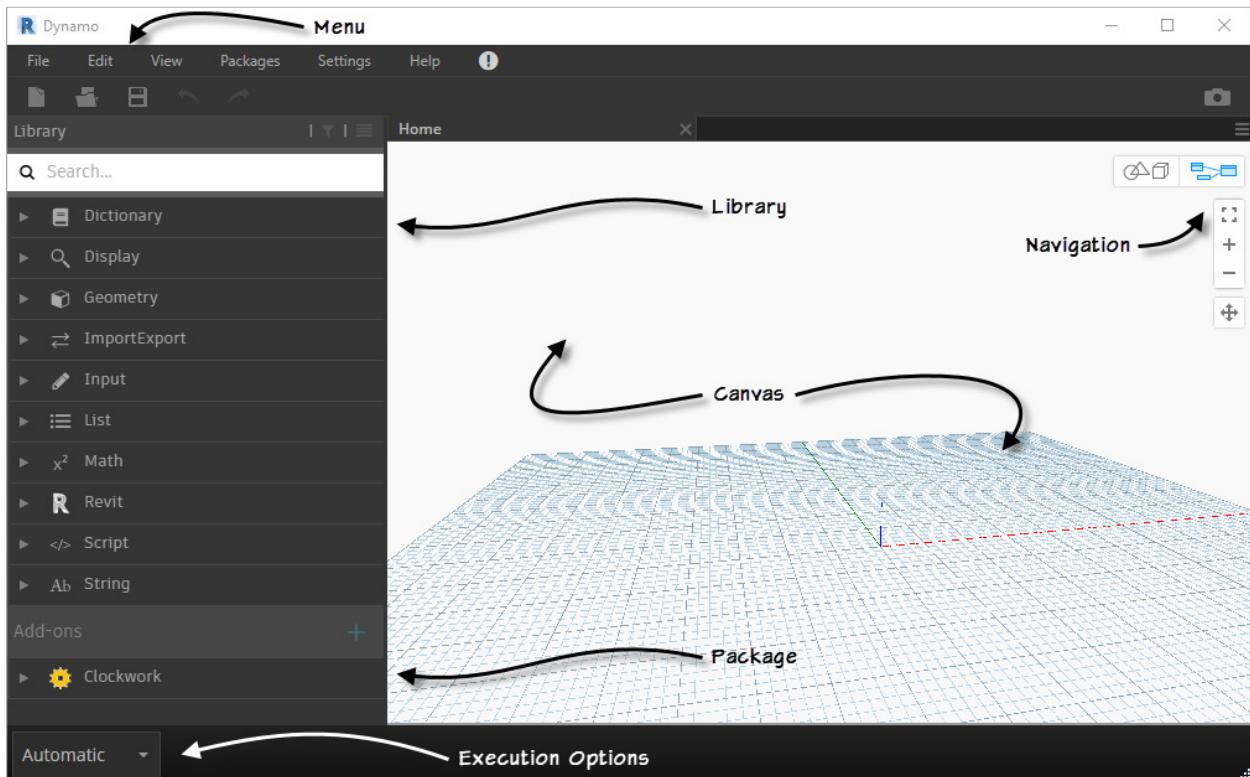


Figure 8

Your library may not match the figure. Dynamo is extensible, and many users of the community make their own custom nodes and make them available in “packages” to other users. Any installed packages will appear in your library panel beneath the Add-ons branch. In the figure, “Clockwork” is an installed package. The Add (+) button on the Add-ons branch can be used to add more packages to your install as can the Package menu on the menu bar.

## NODES AND WIRES

Creating a Dynamo script (or graph) uses “Nodes” and “Wires.” Nodes are prebuilt blocks of code that perform a single discrete action or function. Wires are used to connect nodes together and build the “flow” of the graph moving data from left to right and from node to node (see Figure 9).

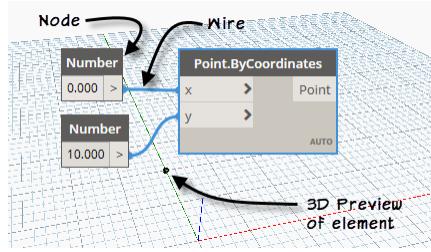


Figure 9

To place a node, simply locate it in the library and click on its name.

2. In the Library, click on: *Geometry*, then *Points*, then *Point* and then ***ByCoordinates (x,y)*** (see the left side of Figure 10).

The node will appear in the canvas. Notice also that a point appears onscreen in the model canvas beyond. This is because the inputs required by this node have default values. You can see an indication of the default value for any input by hovering your mouse directly over the input.

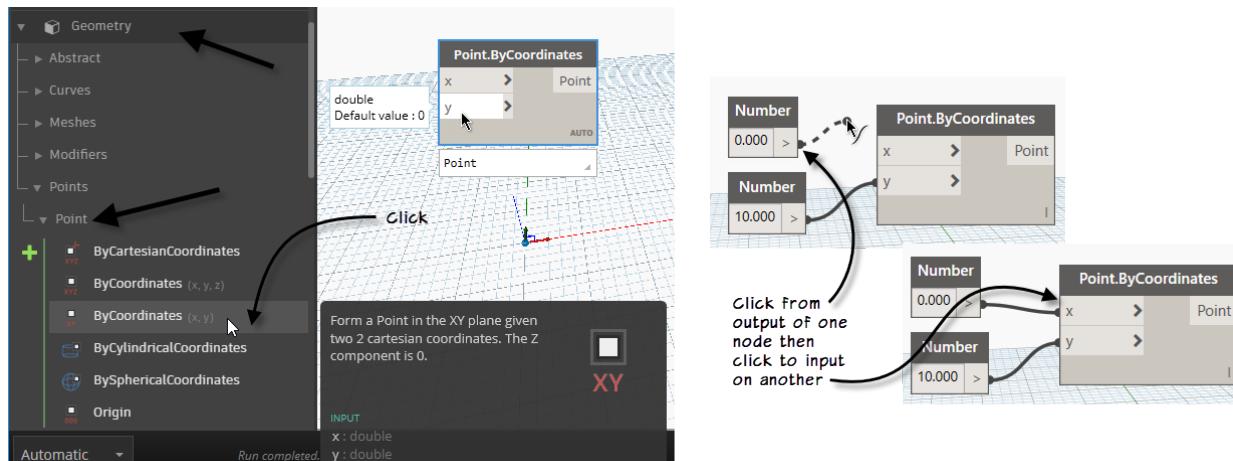


Figure 10

Ports appear on the left and right sides of nodes. Data in a Dynamo graph flows from left to right. The left side contains the “inputs” and the right side has the “outputs.” **A good way to get started is to think about what you want to do, find a node for that purpose, then simply look at what inputs it requires.** In this case, we are going to create a point. Points are used to locate geometry. This node we just added requires an X and Y input. These can be input with simple numerical values. **Making sure that you always feed the right kind of data into an input is critically important in Dynamo.** We’ll get into that much more later, but in this case, numerical input is required.

3. On the library, click on: *Input*, then *Basic*, then ***Number***. Click it again to add a second one.

4. Drag the two **Number** nodes by their titlebars to position them to the left side of the **Point.ByCoordinates** node.

To wire up your nodes, simple click on the port of one node, a wire will extend out of the port as you move your mouse. Click on another port to finish the wire and connect the two ports

5. Click on the output port (right side) of the first **Number** node and connect it to the **x** port of the **Point.ByCoordinates** node. Repeat for the second **Number** connecting it to the **y** port (see the right side of Figure 10).

Notice that the point onscreen is unchanged. This is because the **Number** nodes use the same default (0) as the **Point.ByCoordinates** node does.

6. Click in the **Number** node that feeds the **y** coordinate and change it to: **10**. Press ENTER.

Notice that the point will move accordingly.

Inputs can only have one wire. But multiple wires can be connected to outputs allowing output values to feed into several other locations of the graph. To see this, you can try connecting the output from one of the **Number** nodes to both inputs of the **Point.ByCoordinates** node. This would mean that the X and Y coordinates would always be the same. Reverse this when you are done experimenting so that we have one **Number** feeding each input separately.

## FINDING NODES

As we saw previously, you can browse the library by clicking on the category headings to see what is stored within them. But Dynamo contains LOTS of nodes. So, browsing this way can be time consuming. An alternative is to search for nodes. There is a Search field at the top of the library and you can also right-click in the canvas to get a search field on the context menu as well (see Figure 11).

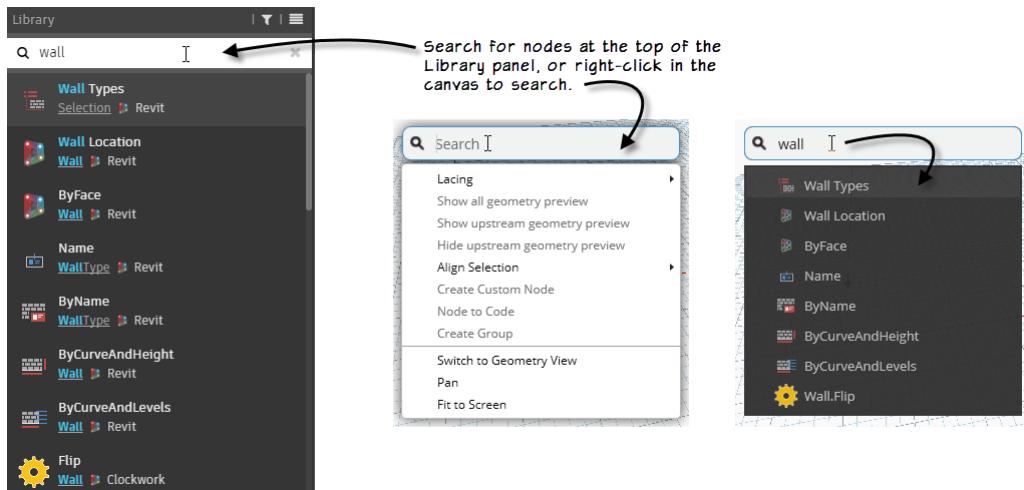


Figure 11

However, the trouble with searching if you are a beginner is knowing what to search for...

So, we recommend that you take some time initially to browse the library and get familiar with the included nodes and overall library structure. In the graphs that follow, we indicate where

each node is in the library for your convenience. You are free to search for them instead if you wish however.

## SUMMARY OF KEY CONCEPTS

Here is a summary of the most important concepts covered in the previous exercise:

- Dynamo files are referred to as “scripts” or “graphs.”
- Dynamo is accessed from within Revit and requires an active Revit document to run.
- Access Dynamo and Dynamo Player on the Manage tab.
- Dynamo Player runs pre-built Dynamo graphs without the need to open Dynamo.
- Dynamo Player gives access to input nodes included in the scripts allowing users to interact with the script and change input values.
- Dynamo graphs are built from left to right.
- Locate nodes (prebuilt bits of code that perform specific actions) in the library by browsing or searching.
- Wire nodes together by connecting outputs to inputs.
- Outputs can have more than one wire feeding into multiple inputs. Inputs can only contain a single wire.
- Graphs flow from left to right.

## HELLO WORLD

Once you are familiar with the overall interface, you will want to create your first graph. Programmers always like to show the “Hello World” example as an “ice-breaker.” In other words, something really simple; something that gives quick results.

**SAMPLE FILE:** You can open a file completed to this point named: **01\_Hello Wall\_A.dyn**. (In the Open dialog, uncheck “Open in Manual Execution Mode” before opening).

### NAVIGATION

You can navigate within your Dynamo canvas using the wheel on your mouse: hold in and drag to pan and roll it to zoom in or out. You can also use the icons on the right side. There are icons for Zoom to Fit, Zoom in, Zoom out and Pan.

### SIMPLE TEXT

Above we added numbers and a point. We’ll use these below. Now let’s work with some simple text.

1. On the library, click the *Input* branch and then click the *Basic* branch.

This branch contains several types of input nodes. These are nodes that you use to feed information to your graph.

2. Click the **String** node to add it to the canvas (see Figure 12).

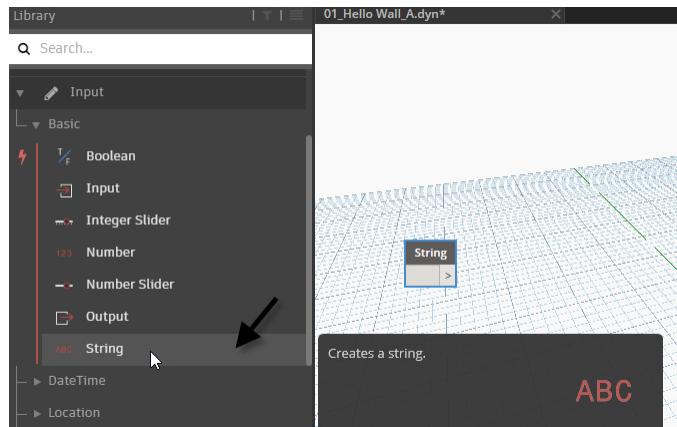


Figure 12

Within a **String** node you can input any text you like. Programmers like to refer to text as “strings.” So, you can type in anything you want into a **String** node. And even if you type in numbers, the graph will interpret them as text only. (We’ll see numerical input below).

3. In the **String** node, type: **Hello World**. To finish, click away from the node, don’t press ENTER. If you press ENTER, it will stay in the node and wrap to the next line.

Now let’s grab a “**Watch**” node. **Watch** nodes are useful to “report” the output at points along the graph.

4. To find it in the library, expand the *Display* branch and then the *Watch* branch. Click on the **Watch** node to add one.
5. Drag it to the right of the **String** node and then wire the output of the **String** to the input of the **Watch** (see Figure 13). (If necessary, click the Run button at the bottom).

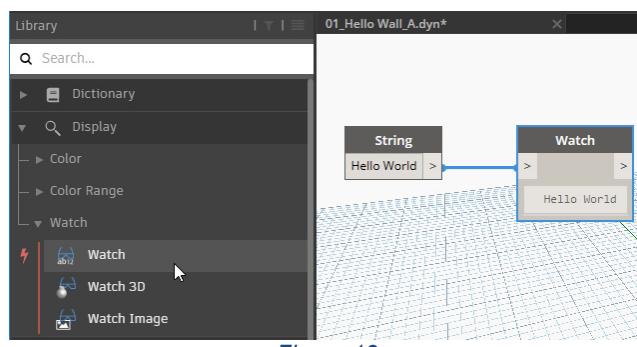


Figure 13

Well, that is a complete graph, just not very useful or exciting. Let’s vary it a bit.

6. Add another **String** node. Type: “Hello” in the first one and: “World” in the second.

The Watch will still be wired to the original node and will now only report “Hello.” To put the two together, remember that we cannot put two outputs into a single input. If you try to wire “World”

to the Watch, it will replace the wire from “Hello.” So instead, we need another node to combine their values first.

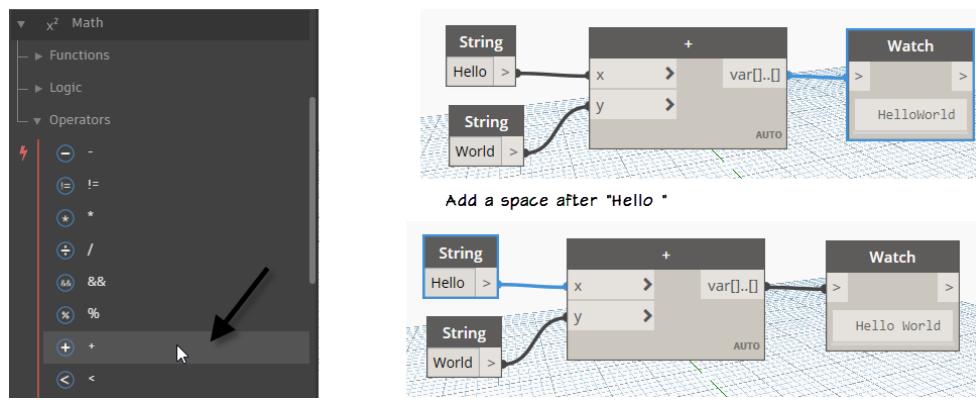
### COMBINING VALUES

So how do we combine them? Well there are lots of ways. Here’s a simple one.

7. On the *Math* branch of the library, expand *Operators* and then click the **+** node.

This has two inputs: **x** and **y**. These can be anything (kind of like algebra class...)

8. Wire “Hello” into **x** and “World” into **y**. Then wire the output of **+** to the input of the **Watch** node (see Figure 14).



Notice that it is quite literal and will report “HelloWorld” as the result. You can edit the first **String** node and add a space at the end of “Hello ” to fix that. The graph will update immediately. Again, this is not too exciting, but even this simple example shows how data flows through the graph. Let’s take it a little further. Since we have a **+** node, what about some numerical input?

**SAMPLE FILE:** You can open a file completed to this point named: **01\_Hello\_Wall\_B.dyn**. (In the Open dialog, uncheck “Open in Manual Execution Mode” before opening).

### NUMBERS

In the library, on the *Input > Basic* branch, click on the: **Number** node. Wire the **Number** node to the **y** input of the **+** node. Note the result in the **Watch**. In this case, Dynamo will happily combine text with numbers, but since one of the inputs is text, it will keep the result as text. But if you create a second **Number** node and wire it in, it will do the math instead (see Figure 15).

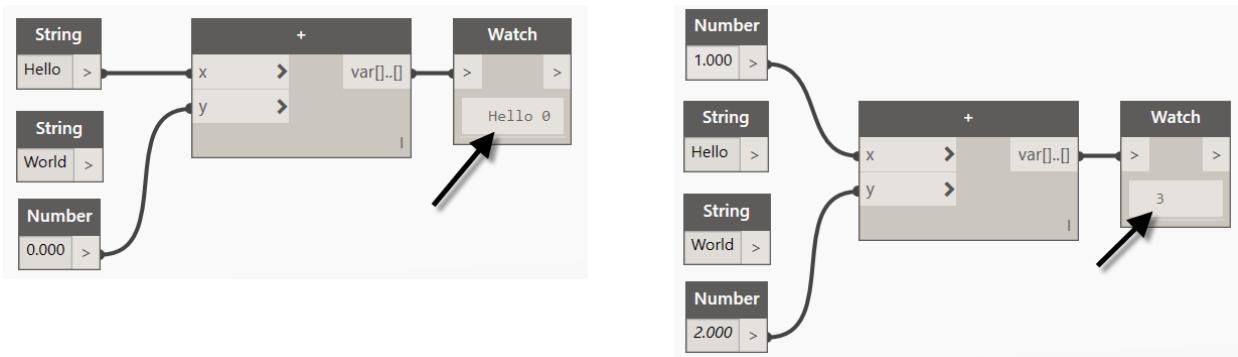


Figure 15

It is worth repeating that when we fed in text, the **+** node produced a text result, but when we fed in numbers, it produced a numerical result; but *only* when both inputs were numerical did it produce the numerical output. This is critically important. Dynamo is very sensitive the type of data being input. **Many errors encountered have to do with incompatible data types.** We will discuss this more later. But for now, please make note that understanding and properly matching of data types are a critical component to success in Dynamo.

**SAMPLE FILE:** You can open a file completed to this point named: **01\_Hello\_Wall\_C.dyn**. (In the Open dialog, uncheck “Open in Manual Execution Mode” before opening).

### MAKING POINTS

Using Dynamo to add two numbers may not be very exciting nor faster than your calculator, but there is plenty more that you can do with numbers. The “hello world” of making geometry would be making points; like the one we did above. So, let’s return to making points. The **Geometry > Points > Point** branch has a few options for making points (see Figure 16).

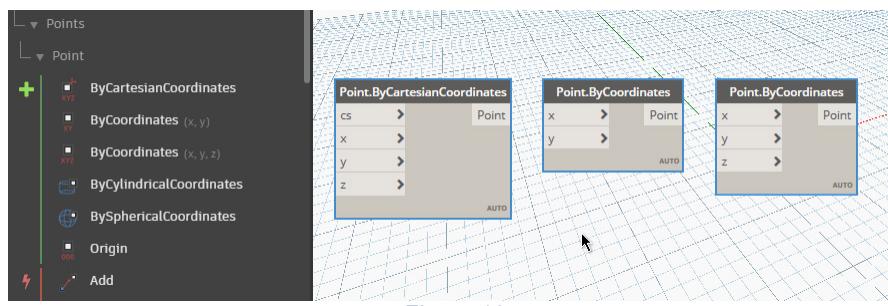


Figure 16

The main difference is in the inputs required. Let’s look at the **Point.ByCoordinates** that we used above (shown second from the right in Figure 16). It is the simplest one and only needs an **x** and **y** input. **If you hover your mouse over an input, it will typically tell you what it expects and if there is a default value.** In this case, simply placing this node will add a point to the canvas because the default value of both inputs is zero. So, you get a point at 0,0. However, when you wire the two **Number** nodes into the two point inputs, you can change their values and the point will move accordingly (see Figure 17).

You should still have the **Point.ByCoordinates** node with its two **Number** nodes from before.

9. Wire the output from the **Point.ByCoordinates** node to the existing **Watch** node.

10. Delete the **String** and + nodes.

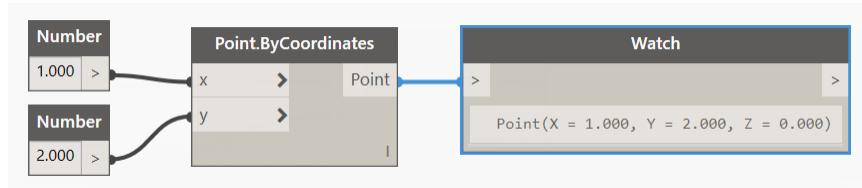


Figure 17

**SAMPLE FILE:** You can open a file completed to this point named: **01\_Hello Wall\_D.dyn**. (In the Open dialog, uncheck “Open in Manual Execution Mode” before opening).

### MAKING A LINE

What can we do with points? Lots of things (like draw lines for example). But you may not always know when Dynamo requires point inputs. The easiest way to tell what is needed is to place a node and look at the inputs required.

11. In the library, on the *Geometry > Curves > Line* branch, click **ByStartPointEndPoint**.

In mathematics, the definition of a line is the “curve” that represents the shortest distance between two points. When you add the line node, you will see that it has two inputs: the start and end points of the line. You can repeat the steps from above to place the additional nodes you need, or when you want to reuse parts of your graph, you can copy and paste. In this case, we have a single point. To make our line, we need another point. We can simply copy the existing point and then change the values to give us the inputs we need to make a line.

12. Drag a window around the **Point.ByCoordinates** and its inputs (include the **Watch** if you like), and then from the Edit menu, choose: **Copy** or press CTRL + C.

13. From the edit menu, choose: **Paste**, or press CTRL + V.

14. With the copies still selected, drag them to position them in a convenient location.

Since we copied a point, you now have two points in the same spot.

15. Change the number values of the copied point to move it (see Figure 18).

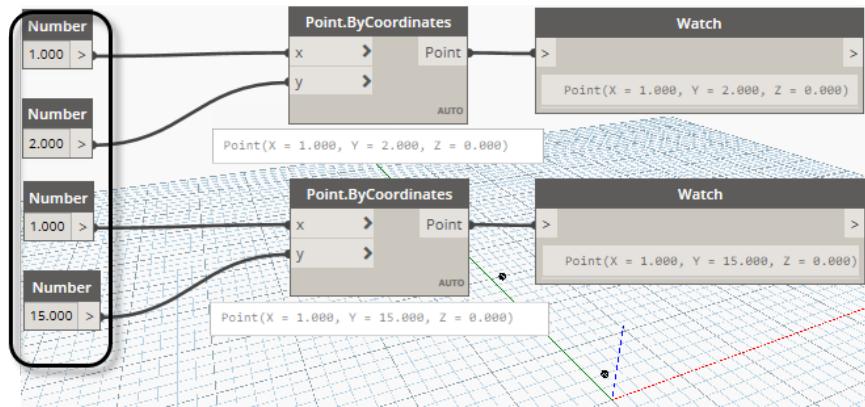


Figure 18

16. Wire these to the **Line.ByStartPointEndPoint** node to create a line (see Figure 19).

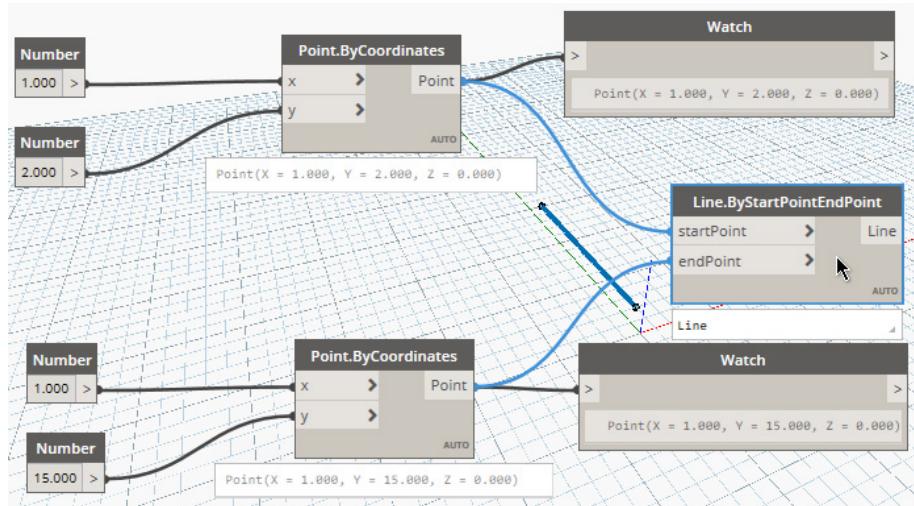


Figure 19

### To WATCH OR NOT TO WATCH...

The **Watch** nodes are not necessary here. It can be nice to include them as you are working so that you get feedback as you go. In this case however, you see the points in the 3D canvas. So the **Watch** nodes really don't add any value to the graph. You will also notice that **Watch** nodes have outputs. You can wire from those or go back to the original node and wire from their output. Notice that here, we have done the latter. This makes it easier to remove the **Watch** nodes later without forcing you to re-wire anything.

There is an alternative to Watch nodes. You can instead receive feedback directly from the node itself. When you hover your mouse over a node, it will show a little pop-up (called a "preview bubble") with the output of that node. There is a small pushpin icon that you can click to keep the preview bubble expanded even as you move your mouse away (see Figure 20).

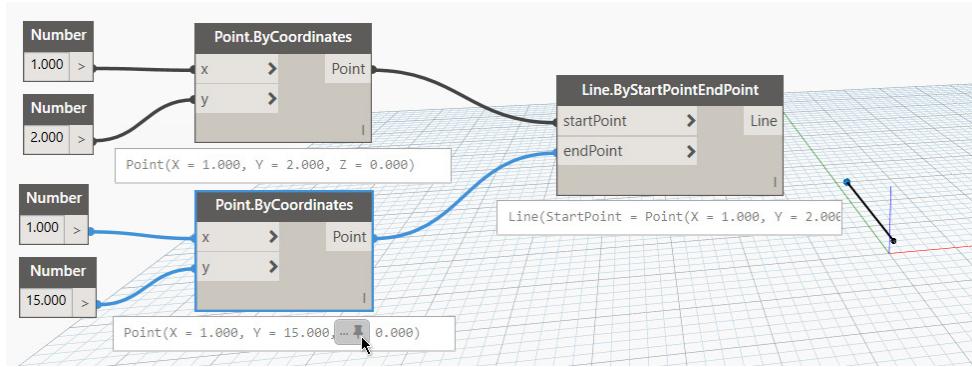


Figure 20

If you use the preview bubbles, you can eliminate the **Watch** nodes. Or you can use both. It is up to you.

**SAMPLE FILE:** You can open a file completed to this point named: **01\_Hello\_Wall\_E.dyn**. (In the Open dialog, uncheck "Open in Manual Execution Mode" before opening).

### DRAW A WALL

As noted before, when building your graph, think about what you want to create, add the nodes that will create what you need and then look to see what inputs are required. For example, if you started the previous exercise with the **Line** node, then look at its required inputs, that would have told you that you needed two points. This approach helps you figure out what will be required. Once you have a line, you can use it to make many things; like a Revit wall for example! There are two nodes available for adding walls. Both let us build a wall from this Dynamo line that we already have.

At this point we should stress that while you are potentially seeing a blue preview line in Revit, **we have not yet created any actual Revit geometry**. When you work in Dynamo, you will be frequently creating Dynamo geometry, then using that to build Revit geometry and vice-versa. **But they are too different things!** (See the next topic for more information on this concept).

17. in the *Revit* branch of the library under the: *Elements > Wall* branch, click the **Wall.ByCurveAndLevels** node.

This node has four inputs: **c** stands for “curve.” We also need two levels to set the height and a wall type. As noted previously, a useful habit to get into is to pause your mouse over the various inputs for a tooltip indicating more information on what that input expects (see Figure 22).

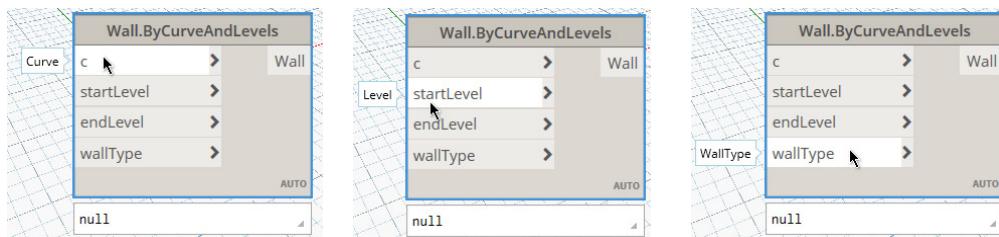


Figure 21

To make a wall, we need a path for it follow. In programmer speak this is called a “curve.” This is “curve” in the math class sense of the word; not necessarily “curvy.” By this definition, our line counts as a “curve.” As does anything with two endpoints; either a straight line or a curvy line and Dynamo still thinks of it as a curve<sup>5</sup>.

18. Starting with the curve, wire the output from our Line node above to the **c** input.

To satisfy the others, we need some more nodes.

19. From the *Revit > Selection* branch<sup>6</sup>, add: two **Levels** nodes and one **Wall Types**.

<sup>5</sup> Nearly any linear path with a start and end point can be thought of as a curve in Dynamo. This includes many types of curves, simple straight lines, arcs (part of a circle) to complex Bezier or nurbs curves. However, if you intend to make a Revit element from the curve, you must remember what is possible in Revit. Just because you can make a perfectly valid spline or elliptical curve in Dynamo, does not mean that Revit will allow you to use this path to make a wall. So, since walls in Revit can only be lines or circular arcs, you must limit your Dynamo curve to one of these shapes as well. (Sorry if you find that disappointing...)

<sup>6</sup> The reason we are using the **Levels** and **WallTypes** nodes from the *Selection* branch is because we want to use levels and wall types that already exist in our Revit project. You will find similar sounding nodes on the *Elements* branch, but these would create new levels or types. We don't want that here.

Each of these has a drop-down list from which you can make a selection.

20. From the first **Levels** node, choose: **Level 1** and wire it to the **startLevel** port.
21. Choose: **Level 2** from the next one and wire it to: **endLevel** and then choose any wall type and wire it to the **wallType** port (see Figure 22).

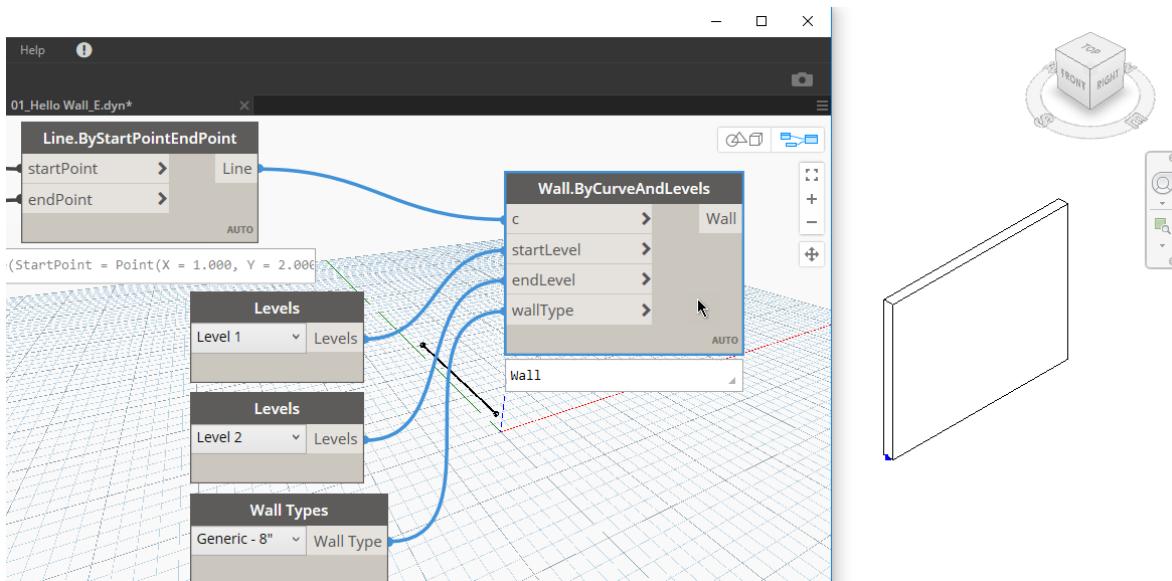


Figure 22

WooHoo. Look at that shiny new wall...

#### TYPES OF GEOMETRY

Notice that in Dynamo, we still only see the line. We hinted at this issue above. To see the wall, you must look at Revit. A 3D view shows it well. (You might have to shuffle your windows around onscreen to see everything). But the fact that the wall shows in Revit while only a line appears in Dynamo is a good indication that there is a difference between “Dynamo geometry” and “Revit geometry.” Dynamo geometry only appears in the Dynamo environment and is more abstract. You can use Dynamo geometry to create Revit geometry (as we have done here) and vice-versa. But you will not automatically get both. If you want to see a representation of the wall in Dynamo, you can do so by adding another node to the graph<sup>7</sup>.

22. Browse to: *Revit > Elements > Element* and click the **Geometry** node to add it to the canvas.
23. Wire the output of the **Wall** node into this node (see Figure 23).

<sup>7</sup> It can be tempting to add the **Geometry** node to the graph to see a nice preview directly within Dynamo. But do this sparingly as in many cases it can add significantly to the time it takes to process the graph.

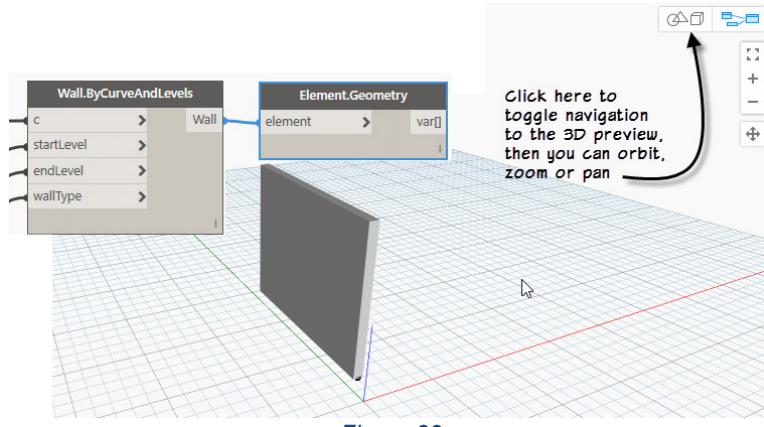


Figure 23

If you want to orbit, pan or zoom the 3D preview in Dynamo, use the small toggle icons at the top right to switch to navigation in the 3D view. Then switch back to the graph when you are done. You can also temporarily switch by holding down the ESC key and then using the mouse buttons and wheel to orbit, zoom and pan.

**SAMPLE FILE:** You can open a file completed to this point named: **01\_Hello Wall\_F.dyn**. (In the Open dialog, uncheck “Open in Manual Execution Mode” before opening).

### HELLO WALL

The original graph that we ran at the start (using Dynamo Player) contains one more piece. There was a value input into the comments on the wall we created. Two of the most useful nodes in the library allow us to “get” and “set” parameters on Revit elements. The “get” nodes retrieve a value from some parameter in the Revit model while the “set” nodes write a value to a specified parameter. We’ll try the “set” one now.

24. Browse to: *Revit > Elements > Element* and click the **SetParameterByName** node to add it to the canvas.

Notice that it takes three inputs: **element**, **parameterName** and **value**. The **element** input takes a Revit element like the wall we created with this graph. The **parameterName** input is a string and takes the name of any parameter of the element in the first input. In this instance, we want to write to the “Comments” parameter. Also keep in mind that you must be sure to type the name *exactly*. It is case sensitive. Finally, the **value** input takes the value you want to write. Since Comments is a text parameter in Revit, the value we input here will be text as well using a **String** node<sup>8</sup>.

25. On the *Input* library branch, expand *Basic* and then add two **String** nodes.
26. Position them to the left of the **SetParameterByName** node.
27. In the first **String** node, type: **Comments** and then click outside the node (do not press ENTER). Wire it to the **parameterName** input.

---

<sup>8</sup> Data type is very important in Dynamo. If you feed the wrong type of data to an input, it will fail. In this case it is a simple string. But in many cases it will be something more specific.

28. For the other **String** node, type in: **Hello World** (or any other message you like) and then wire it to the **value** input.

29. Wire the output from the **Wall.ByCurveAndLevels** node to the **element** input on the **SetParameterByName** node (see Figure 24).

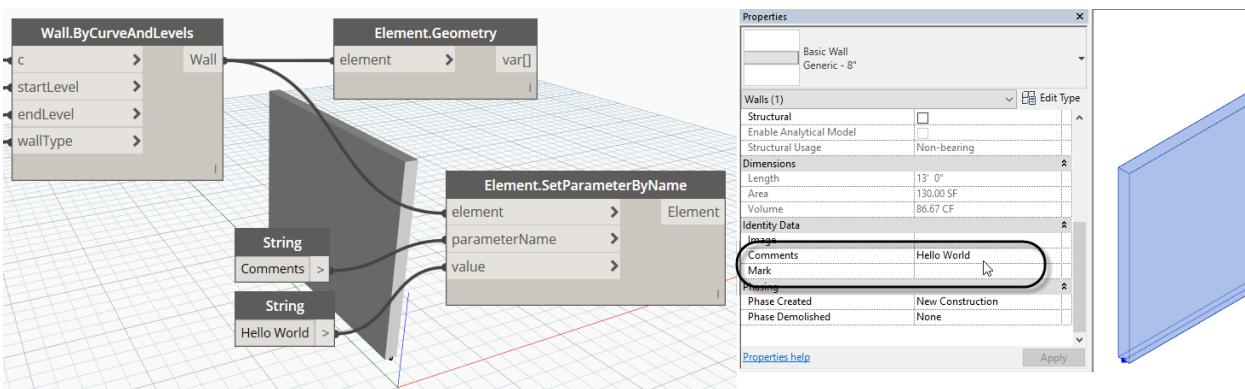


Figure 24

30. In Revit, select the wall and look at its Comments on the Properties palette.

**SAMPLE FILE:** You can open a file completed to this point named: **01\_Hello\_Wall\_G.dyn**.

## FINISHING TOUCHES

This graph is now complete and matches the one we ran at the start from Dynamo Player with one small exception. If you recall when we edited the inputs using Dynamo Player, they had user friendly names like: “Start X Coordinate” and “Base Level.” If you were to run the graph we just completed in Dynamo Player, the inputs would not have these names, but rather would simply display the generic names of the various nodes like: Number, String, etc.

If you intend to use a graph in Dynamo Player, you must designate which nodes are to be inputs and you can rename the nodes to make the input names more user friendly. To do this, right-click on a node and choose “Is Input.” To rename it, simply double click the titlebar on the node you want to change. This will let you type in a new name. If you have a node that you do not want to appear as an input in the Dynamo Player, right-click on it and uncheck “Is Input.” In our example, we would want to do this for the **String** node that feeds the **parameterName** input. Otherwise, users would be able to change the parameter that we are setting. This could result in errors if they do not input a valid value. An alternative to right-clicking the node and unchecking “Is Input” is to use a Code Block for the input instead. (More on Code Blocks follows below). If you rename a Watch node, it becomes an Output. You can use this to showcase volume/area etc.

31. Close Dynamo and the Revit project file when you are done. You can keep Revit running.

**Congratulations. You have completed your first Dynamo graph!**

## SUMMARY OF KEY CONCEPTS

Here is a summary of the most important concepts covered in the previous exercise:

- Inputs allow user input into the graph.
- Text input is referred to as a “String.”
- Strings can receive text or numeric input but will still be treated like text even if it is numerical.
- Matching data types correctly like text, number, Revit elements, etc. are critically important to properly functioning graphs.
- Make points by indicating coordinates.
- Make lines by designating two points.
- Watch nodes report the output of nodes to which they’re attached.
- An alternative to Watch nodes is Preview Bubbles.
- Dynamo geometry does not automatically create Revit geometry and vice-versa.
- Dynamo geometry is abstract and appears only in Dynamo.
- Revit geometry will not appear in Dynamo unless you add the appropriate nodes to your graph to show it (like **Element.Geometry** for example).
- Use Dynamo to input parameter values in Revit elements using the “Set” node.
- Rename nodes in Dynamo to change the name of inputs or outputs in Dynamo Player.

## VISUAL PROGRAMMING IS PROGRAMMING

All the previous examples can be considered “Hello World” examples in some form or another. They are very basic, introduce you to the interface of Dynamo and allow you to see immediate results. When you are ready to begin using Dynamo to do something practical, you will naturally be building graphs that are more complex. This is where training sessions like this one sometimes skip right over the middle part and dive into the deep end of the pool. However, this is the beginner workshop and our goal is to stay in the shallow end for the entire session! If you are a more advanced user, then you can try variations of the exercises along the way, but our focus will remain on building a good foundation and sticking with the basics.

What we will say about the “deep end” is that the possibilities are truly endless. Let’s make no mistake about it, **Dynamo IS programming**. To be precise, it is “visual programming.” This simply means that the Dynamo environment is designed to shield you from as much of the underlying text-based code as possible. So, we have the lovely nodes, pre-baked with useful code that do the heavy lifting for us... Mostly.

You drag and drop nodes and wire them up. In a nutshell, that’s visual programming. However, it is not as simple as all that most of the time. First, even when using prebuilt nodes, you still must understand programming concepts and **think like a programmer**. This is important to get the logic right. And, even though we have this library of nodes, what you will see very soon, (if you have not already seen it) is that many, if not most, Dynamo graphs and Dynamo programmers DO use code. This code is placed in **Code Block** nodes. Or for very advanced users: Python nodes.

## CODE BLOCKS

Certainly, you *can* do many useful things in Dynamo *without* having to use any code. That means that you can build a perfectly functional graph, that performs a useful procedure without the need for Code Blocks and manually typing out ANY of the code. So, if you are just a beginner, and not really ready for code of any kind, then don't worry. You can create useful Dynamo graphs that do not contain any Code Blocks. But, as you progress, you will find that Code Blocks are **incredibly** useful. Just not so much for beginners. The premise that we will follow here therefore, at least for the early part of this lab is that it is *rarely* a good idea for beginners to concern themselves with Code Blocks right away (**even if they can save time**). Your goal initially should be learning how to work in Dynamo first; before worrying about how to build graphs more quickly or efficiently. "Crawl before you walk or run" so to speak.

So now that you at least understand at a conceptual level what a **Code Block** is, you don't have to give them any more thought for now. We will revisit the issue later when it is appropriate for us to use Code Blocks, until then, don't worry about them.

## PSEUDO CODE

Writing computer code sounds so technical and complex. We often hear about thousands or even millions of lines of code and all that implies. But what is code really? Part of the problem is the word "code" in the first place. It literally implies secrecy and mystery. When something is in code, it must be decoded to be understood. If you are new to programming, this can certainly be intimidating. In computer terms however, the word code is simply a synonym for instructions. Each line of code tells the computer or device to perform some action. "Coding" is therefore the act of devising (or encoding) these instructions.

As we have already discussed, one of the virtues of Dynamo is that it mostly "shields us" from the code. Using it, we can take advantage of the power of programming *without* needing to write lines and lines of code<sup>9</sup>. But as noted in the "Visual Programming is Programming" topic above, you are still programming when you create Dynamo scripts. So, it is valuable to think about things the way a programmer would. To help you do this, we want you to think about some everyday tasks like: brushing your teeth, cooking a meal, baking a loaf of bread or making a cup of coffee. Some of these tasks are so common that we have internalized all the steps and rarely stop to think about them. Do you really think about brushing your teeth? Or do you just do it?

But this is exactly what you need to do if you want to be successful as a programmer. You must think about exactly what you are asking the software to do, what is required to achieve it, in what order and what outcome you want when the task is done. Most programs can be thought of in three parts: Input, Processing and Output. With this in mind, let's do a mind experiment (or you can sketch it out on paper). What is required to bake a loaf of bread? (see Figure 25).

---

<sup>9</sup> To be precise, you *can* write code in Dynamo, there is a full scripting language called Design Script which you can type into special nodes called Code Blocks. Code Blocks were covered briefly in the previous topic. You can also type in Python code for even more advanced coding operations when you are ready. The focus of our workshop is on the fundamentals and while some examples below will use **Code Block** nodes, their use will be minimal and limited more to simplifying graphs with shorthand rather than detailed and lengthy snippets of Design Script code. If you are interested in learning more about Design Script, Code Blocks, Python or any other more advanced coding, try a Google search or head over to [DynamoBIM.org](http://DynamoBIM.org) for plenty of resources.

First, what are the “inputs”? These would be the ingredients listed in the recipe. You will need flour, yeast, water, butter, etc. To “process” these ingredients, you follow the steps of the recipe to combine them in the correct quantities and sequence. You can’t knead the dough before you mix it. And you need to proof the yeast before you add it to the dough. And what about if “errors” occur? The reason you proof the yeast is to make sure it is still active. If it does not proof satisfactorily, you must scrap it and start again with fresh ingredients. If you don’t your bread will not rise and the whole project will be a failure. When writing code, or composing a visual program, you need to think of all the steps (no matter how trivial) required and (perhaps even more important) anticipate where there may be problems and allow contingencies for them (programmers refer to this as “error trapping”).

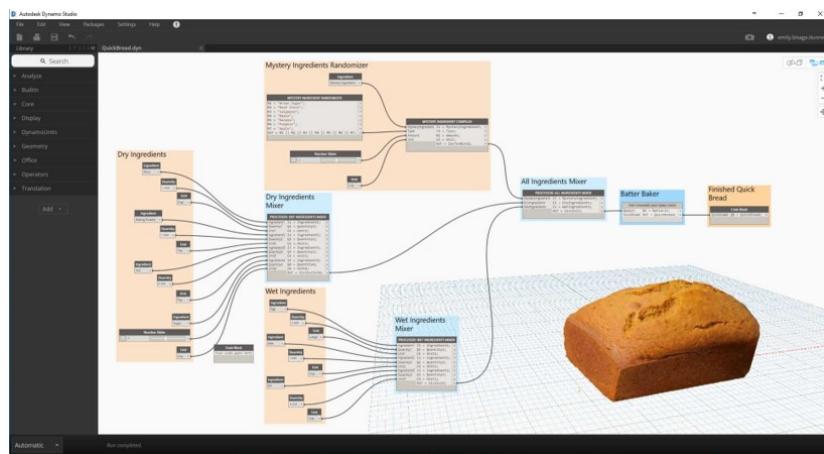


Figure 25

Once you have all the processing completed successfully, you are ready for the output. In the case of our bread, nothing spells successful output like that wonderful smell of a fresh baked bread that fills the room and that first slice while it is still warm... Mmmm, now I’m hungry..

## EXECUTION

“Execution” refers to the actual act of running your graph and allowing it to perform its function. At the bottom of the Dynamo screen, you have a pop-up button that controls how the graph will execute. New graphs default to “Automatic.” This means that as soon as you wire up a node, it will run the graph and perform its function. In smaller graphs without much happening, this can be a nice way to make the experience very interactive. However, as your graphs get more complex, you might want to set the execution to “Manual” instead. This makes a “Run” button appear, at which point the graph will only execute when you press the Run button (see Figure 26).



Figure 26

We also have the option to “Freeze” individual nodes to control which parts of the graph execute. This is a great way to execute just a portion of your workflow to ensure that things are working the way that you like before moving on downstream. This is typically called the “debugging” process in coding. To freeze a node, right-click it. All nodes connected to it downstream will also freeze (see Figure 27).

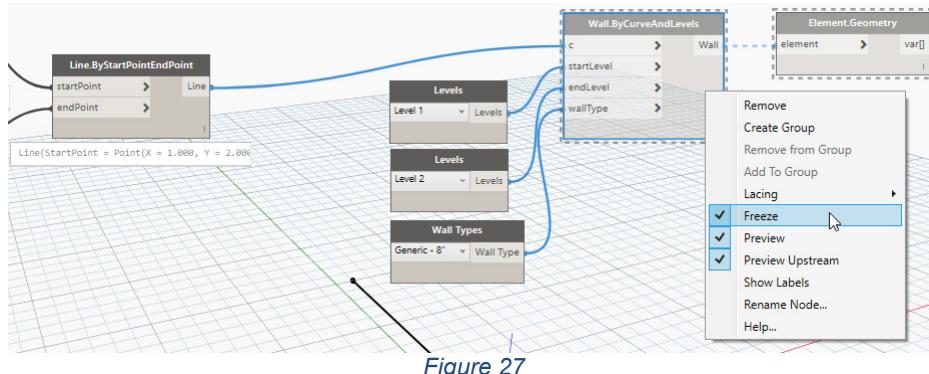


Figure 27

Finally, if you save a graph in automatic mode, you can still open it in manual mode. To do so, be sure to use the Open command (File menu, or on the start screen) and browse to the file. In the “Open” dialog, there will be a checkbox available at the bottom of the window to open the graph in “Manual Execution Mode” (see Figure 28).



Figure 28

In general, it is often desirable to open Dynamo files this way. This allows you to check the graph, ensure it is correct and optionally freeze certain parts of the workflow before execution.

## DOING SOMETHING PRACTICAL

So now that we have gotten some of the basics under our belt, what practical things can we do with Dynamo? A very common task to perform with Dynamo is placing elements in your Revit model. There are nearly limitless approaches to the task, and much of the specifics will depend on the precise goals and expected outcomes. But there will be some common threads.

Typically, you will use the selection nodes to choose which Revit item you want to place. Then you will often need to indicate where in the model to place the elements. And in many cases, you will want to manipulate the elements during placement as well. In the examples that follow we will showcase a few workflows to get you started with thinking about how to place elements in your Revit projects with Dynamo.

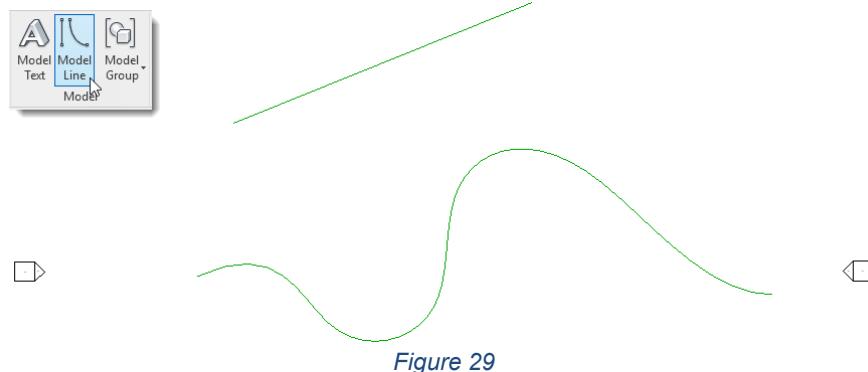
## PLACING A SINGLE FAMILY

Let's say that you wanted to place a collection of families along a path. This is easy to do directly in Revit if the path is straight or follows an arc; in those situations, you can use the Array command. But not so if it is irregular like an ellipse or a spline or if you have more than one path. In this example, we'll look at a graph that allows you to draw a spline in Revit, select it in Dynamo and then insert a variable quantity of elements along that selected path. We'll use trees here, but you can use this graph to place any non-hosted component family.

### FOLLOW A SPLINE PATH IN REVIT

The provided file has a curvy path and a straight line in it (see Figure 29). It was created from the default architectural template. The default template has an RPC tree family pre-loaded. You could use another family if you want, but wall-based, or other hosted families will not work in this example.

1. In Revit, open the file named: *02\_Place Families\_!Start.rvt*.



2. Launch Dynamo and then on the welcome page, click the New button.
3. Add the following nodes to your canvas:

Library Location	Node
Input > Basic	Number
Input > Basic	Number Slider
Revit > Selection	Select Model Element
Revit > Selection	Family Types
Revit > Elements > Element	Curves
Revit > Elements > FamilyInstance	ByPoint
Geometry > Curves > Curve	PointAtParameter

You can manually browse to the location indicated and click the node to add it to the canvas, or you can use the Search field instead and type in the node name to find them.

4. From the Edit menu, choose: **Cleanup Node Layout** (or press CTRL + L).
5. Position the nodes and wire them as indicated in Figure 30.

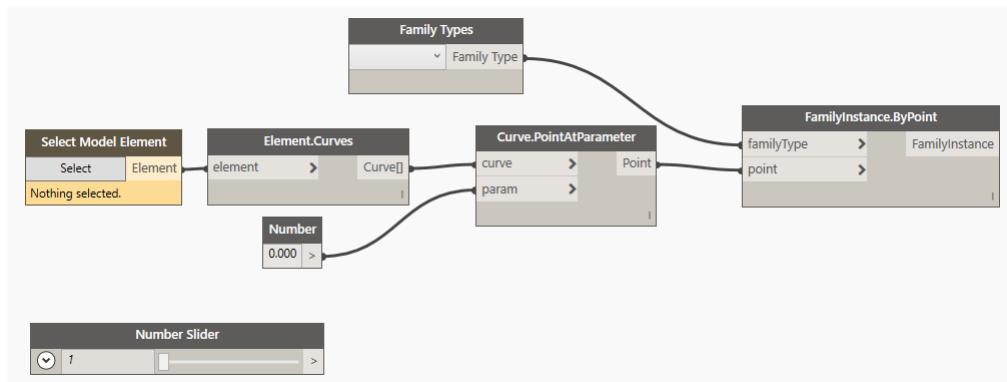


Figure 30

**SAMPLE FILE:** You can open a starter Revit file for this example named: **02\_Place Families !Start.rvt**  
and a starter Dynamo file named: **02\_Place Families\_A.dyn**

### ERRORS AND WARNINGS IN NODES

Notice that the **Select Model Element** node is yellow. This indicates that there is a warning associated with this node. There are other times when the condition is more severe and there is an error state of some kind. In this case, the node will turn red to indicate an error state. We don't have such a condition here, but you can see what one looks like in Figure 31.

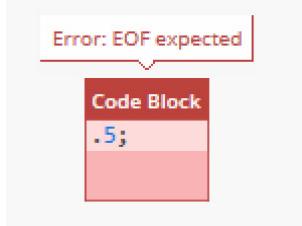


Figure 31

When you have not fed the proper information into a node, Dynamo will be unable to process it, therefore it will display the node in one of these ways to alert you. Here we have a simple warning state. The problem is simple; we have not selected a Revit element yet. The Select button on the **Select Model Element** node is used for this purpose.

*Make sure you can see both the Dynamo and Revit windows together onscreen. It will make the next step much easier. You can position them manually or use the Windows snap assist behavior to assist you. Drag the Revit window to the left side of the screen. It will snap to fill half the screen and then show all other open windows in the remainder of the screen. Choose the Dynamo window from the windows that appear on the other side of the screen. Dynamo and Revit will now each fill half the screen. You can then zoom the two windows to see their contents better.*

6. On the **Select Model Element** node, click the Select button and then in the Revit window, click on the spline element you created above.

The color of the **Select Model Element** node will change to gray indicating that it is now satisfied. The Revit element ID will appear beneath the Select button. If you have the Dynamo execution mode (noted above) set to: Automatic, the graph will process immediately and result in the **FamilyInstance.ByPoint** node going into a warning state. This time there is a small indicator above the node that you can highlight to get more information. If you do this, you see that the message says “cannot be null.” “Null” is programmer speak for “empty<sup>10</sup>.” Translated, we have not chosen a family from the drop-down on the **Family Types** node yet. So, it does not know which family to place (see Figure 32).

<sup>10</sup> This is not *completely* correct. There are also similar items such as: “empty,” “NaN,” “undefined,” “zero,” etc. Null in this context might be better thought of as: “I owe you” or a “dead” reference or pointer to something that is not currently there. So sometimes it means nothing, other times it is more of place holder. In this case it is simply reflecting the fact that have not chosen a family yet. So that input is empty.

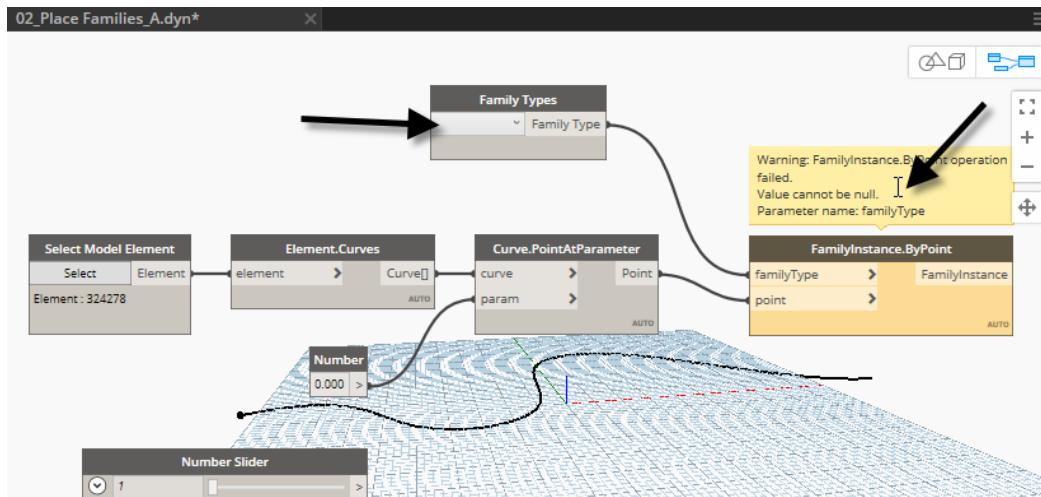


Figure 32

7. From the drop-down on the **Family Types** node, choose an RPC tree family (see Figure 33).

An RPC tree gives a nice result for this example, but feel free to choose another non-hosted model component family if you like.

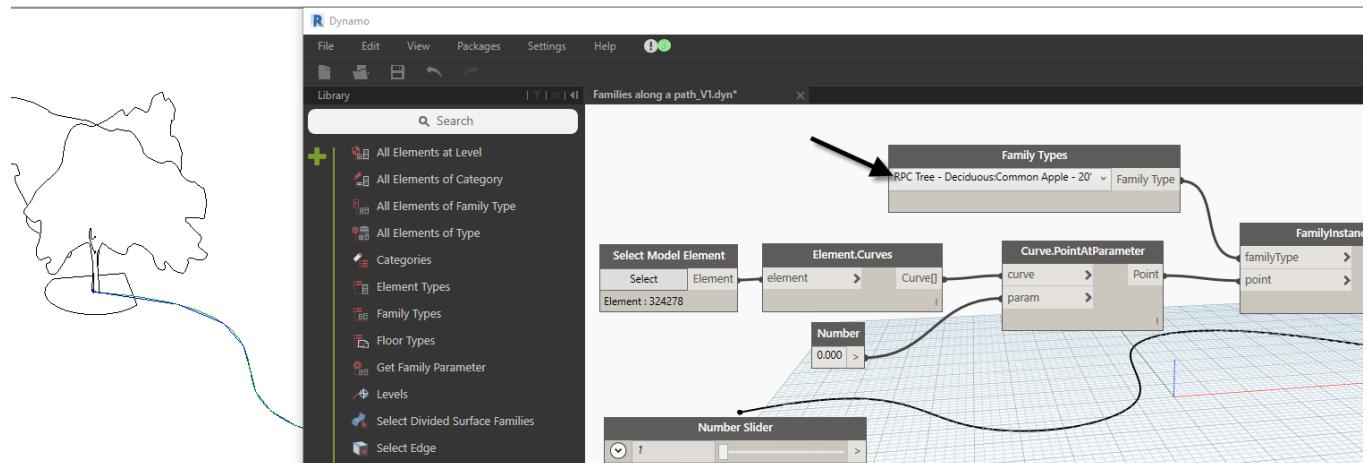


Figure 33

With the graph wired this way, we get one tree at the start of the spline. Also, as we noted above, the tree will appear in Revit, but not in the Dynamo canvas unless you add an **Element.Geometry** node to the graph. (You are welcome to do this if you like).

**SAMPLE FILE:** You can open a file completed to this point named: **02\_Place Families\_B.dyn**.<sup>11</sup>

<sup>11</sup> If your graph ran successfully, but you decide to open the sample file anyway, you will end up with two trees directly on top of one another. This is because when opening the sample file, it will add a new tree. However, in cases where you run the same graph again, it should replace the first instance (tree in this case) instead. Just be aware of this when working in Dynamo. Sometimes it is better to close everything and then reopen the files you need instead.

### ADJUST THE GRAPH AND THE RESULTING REVIT MODEL

So how did our tree know to be placed at the end of the spline? And how did it know which end? The key is the param input of the **Curve.PointAtParameter** node. Right now, we are feeding it a single number. You can input any number you want here within the allowable range. What is the allowable range? In the case of the “AtParameter” nodes, the “parameter” is a value in the range between zero and one. So, a value of zero in the **Number** node (like we have here) puts it at the curve start point, while **1** would put it at the curve endpoint. If you want the tree at other locations, you can try other inputs between these two extremes.

8. If your graph is not already set to Automatic execution, use the pop-up at the bottom-left of the Dynamo screen to change it to: **Automatic** now.
9. Change the value of the Number node to: **.5**.

The tree will move to the midpoint of the spline.

10. Change the value of the Number node to: **.75**.

The tree will move to a point three-quarters the way along the spline.

11. Change the value of the Number node back to: **0**.

### BUILDING A RANGE OF NUMBERS

This is a nice start, but our goal was to have multiple trees spaced evenly along the spline path. To do this, you need to feed in more than one value in the param port. There are many ways you can do this in Dynamo. Let’s do a very simple range of numbers.

12. In the library, on the *List > Generate* branch, add a **Range** node to the canvas.

You have three inputs here: start, end and step. Since our param input needs a range from zero to one, we will input those values into the **Range** node. Then we can step by any fractional amount we want. For example, if you start at zero, end at one and step by .5, you get three items in your range: {0, .5 and 1}. Step by .25, and you will get five: {0, .25, .5, .75 and 1}, etc. To input these values, we can use **Number** nodes or the **Number Slider** we added above. If you hover over each input, a tooltip will tell you the default value (if there is one). Doing this we see that the start input already defaults to zero, so we only need to change the end input.

13. Rewire the graph to incorporate the **Range** node as shown in Figure 34. Be sure to change the value of the **Number** node to: **1**.

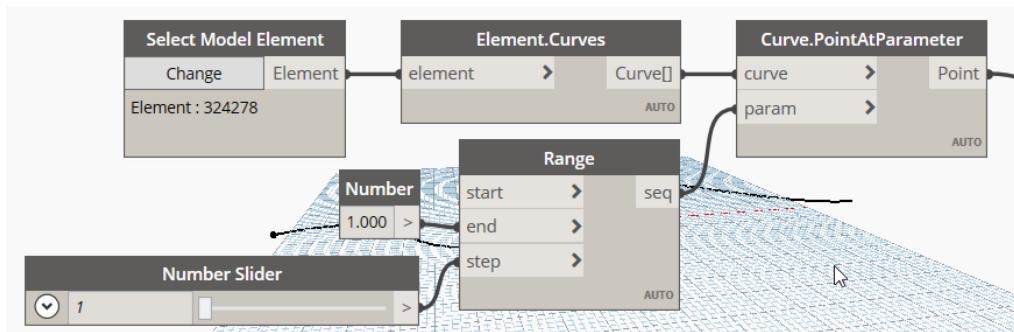


Figure 34

The **Number Slider** is nice because with the graph set to Automatic execution, you can drag the slider and have the graph update in real-time. However, we need to adjust the settings of the slider to make it work for the range we require.

14. Click the small chevron icon on the left of the slider to reveal its settings.
15. Set the Min to: **0** and the Max to: **1**. For the Step, try a value of: **0.1**. and then close the controls by clicking the chevron again (see Figure 35).

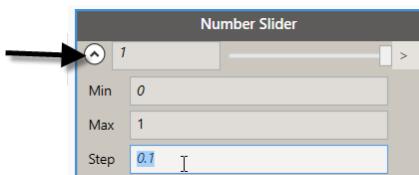


Figure 35

16. Drag the slider down to about **.1** or **.2**.

**SAMPLE FILE:** You can open a file completed to this point named: **02\_Place Families\_C.dyn**.

#### UNDERSTANDING OUTPUT

Unfortunately, it will appear as if nothing is happening. If you feed the slider directly into the **param** input of the **Curve.PointAtParameter** node, you will see the tree move along the curve, but you will still only have one tree. With the slider wired to the **Range** node, it remains at the start of the curve and does not move. The issue is that we are now feeding a list into the **param** input instead of a single value. Therefore, we must understand the underlying structure of the script we are building here a little better and how the data flows through it.

17. Hover over the **Element.Curves** node and when the Preview Bubble appears, move down slightly and click the small pushpin icon to keep it open.
18. Repeat for the **Range** node (see Figure 36).

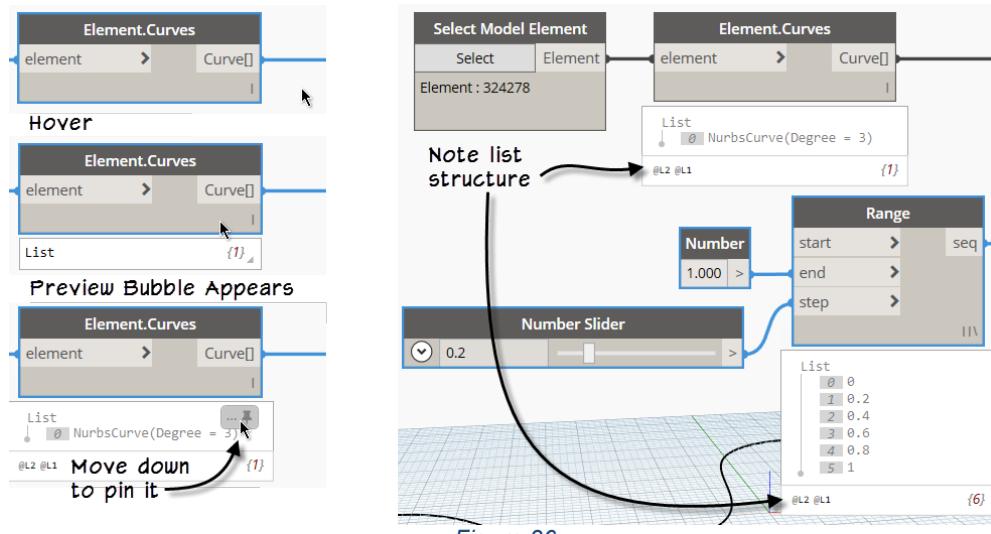


Figure 36

Taking an overly simplistic view of things, Dynamo is all about lists. Dynamo graphs are constantly generating, accessing and manipulating lists of data. The way those lists are structured is critical to getting the expected results from your scripts. Therefore, one of the most important and powerful concepts in Dynamo is list management. So, let's dig a little deeper into the simple list structure that we have created here to understand how to get the behavior we desire.

For both items, the top of the Preview Bubble says "List." Indicating that the output is a list of items. For the **Element.Curves** node, we have a list containing only one item. For the **Range** node we have a list of several items (in the figure there are five, but your quantity may vary depending on the value of your slider). At the bottom of each Preview Bubble, there is an indication of the list structure. It says: @L2 @L1. This is shorthand for "at level 1" and "at level 2." In this case, level 2 (@L2) is the list itself, where level 1 (@L1) is the items on that list. Lists can be structured several levels deep, so you might see lists with @L3 and @L4, etc. Such lists would be lists of lists! We will see some examples of this later. We will also learn how to address the items at each level of the list. But for now, we are most interested in recognizing that we have lists in the first place. And that even though there is one path, the **Element.Curves** node in this case is still generating a list. It is just a list that contains only one item.

Great, we have these two lists. But the fact that we have two lists does not exactly tell us why we are only getting one tree. To understand this, we must understand a new concept.

### UNDERSTANDING LACING

The issue here is simply that we are matching two lists with different quantities of items. The **Curve.PointAtParameter** node has a list with *one* curve. This is being matched to a list of parameter values with many items. Dynamo is matching them in the simplest way possible. The first item from each list; item [0] in this case<sup>12</sup>, is being matched. The rest are being ignored. This is why it appears to not be working correctly, even though in reality it is working perfectly!

Now perhaps you want this simple match. Perhaps not. Either way we can control what we get for outputs using a concept called: "Lacing." Lacing is Dynamo's way of matching the values on two or more lists to one another. The default lacing for most nodes is "**Auto**." This simply means that Dynamo chooses between the three other options automatically. In this case, Dynamo is automatically giving us the "**Shortest**" option for the **Curve.PointAtParameter** node. Shortest lacing will give you the fewest number of pairings. In this case, a list of one item matched to a list of six yields one pairing, and the simplest of the possible pairings, first item to first item (item [0] in each case). If we fed two curves into the curve input and ran the graph again, we would still get a single tree on each curve. However, the tree on the second curve would be placed at location number two (index [1]) from the second list. Let's try it.

19. From the *Revit > Selection* branch, click the **Select Model Elements** (plural)<sup>13</sup> node to add it.

---

<sup>12</sup> Dynamo uses "zero-based" numbering. Meaning that lists start at index zero, rather than index one. So, a list of five total items will be numbered: 0, 1, 2, 3, 4. This is common in the programming World, but if you are new to programming it might take a little getting used to.

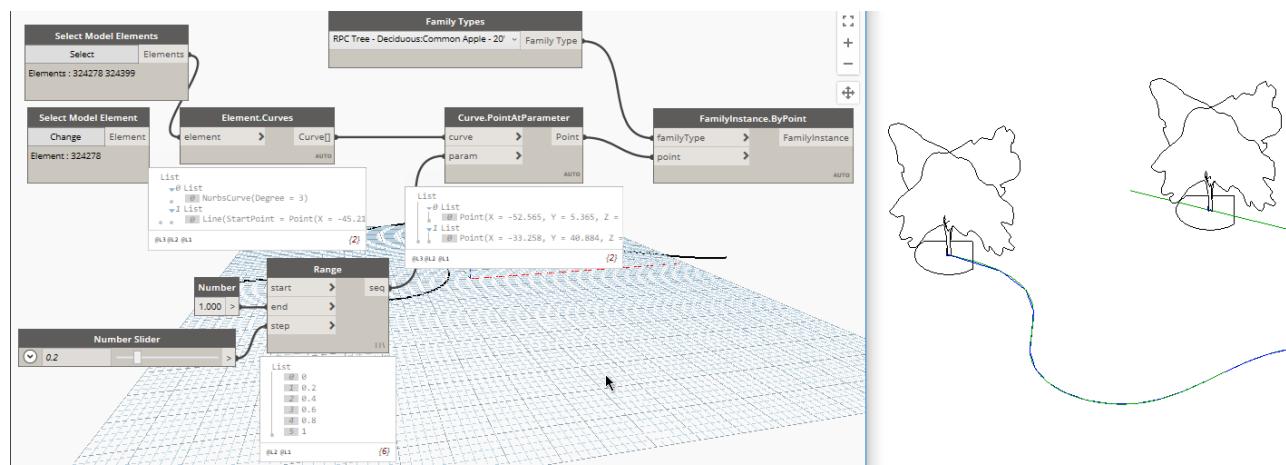
<sup>13</sup> There are two very similar selection nodes in the Revit library: the **Select Model Element** (singular) node allows only a single element to be selected. A **Select Model Elements** (plural) node allows a selection of two or more elements. You must use a window or crossing selection with the **Select Model Elements** (plural) node.

20. Position it next to the existing **Select Model Element** (singular) node and then wire it into the **element** port of the **Element.Curves** node.

If you are still set to Automatic execution, this will generate a warning state in all nodes and the tree will disappear in Revit as you will no longer have a Revit path selection.

21. On the **Select Model Elements** (plural) node, click the Select button and then using a crossing selection in Revit (drag a box across both elements from right to left) to select both the spline and line elements.

The graph will be satisfied again, and the warning state will disappear. Onscreen in Revit, a single tree will appear on each path. But notice that the one on the line is offset a bit from the end (see Figure 37).



This is because the default shortest lacing option is matching the first item on the first list to the first item on the second list. It is then matching the second item on each list. But because there are only two items on the first list, it stops there, and we get a total of two matches.

**SAMPLE FILE:** You can open a file completed to this point named: **02\_Place Families\_D.dyn**.

Note that in the sample file, an **Element.Geometry** node has been added to preview the results directly in Dynamo. This is optional. If you like you can delete it, or right-click it and Freeze it.

There are two other lacing options: **Longest** and **Cross Product**. Right-click on a node to change the setting. With longest, the paring process will be the same as shortest for the first items, but when the shorter list is exhausted, the final item on the shorter list will be repeated to ensure a total number of pairings equal to the longer list.

22. Right-click on the **Curve.PointAtParameter** node and choose: **Lacing>Longest** (see Figure 38).

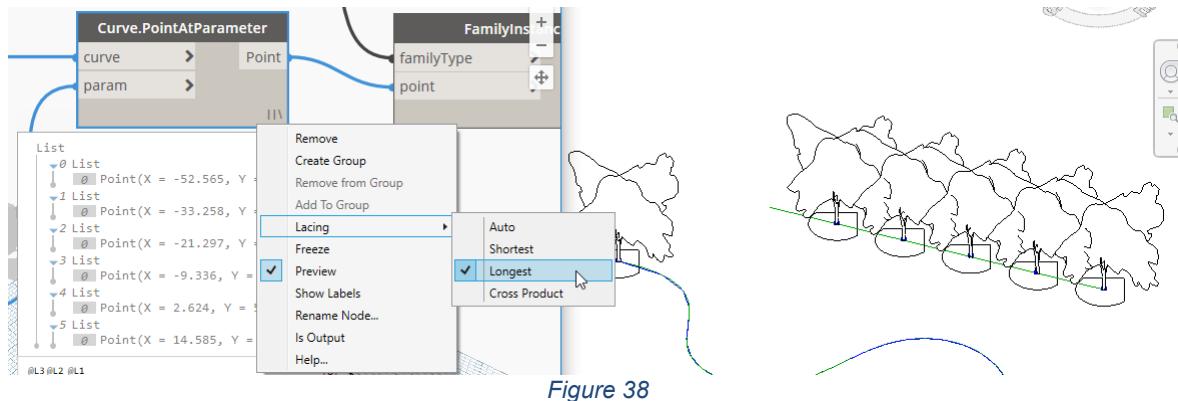


Figure 38

Sometimes the result can be unexpected. In this case, we end up with a several trees along the line, but only one on the spline. But if you think of how longest lacing is designed to behave, this is exactly what it is supposed to do. It only seems odd because our original goal was to repeat the tree along the spline path.

23. Wire the original **Select Model Element** node back into the **element** port of **Element.Curves**.

Notice that this give us exactly what we want! Well it is important to realize that this only works here because we went back to a single element on our first list. So, if you think of it the way Dynamo does, you have two lists. First, we match item [0] from list A with item [0] on list B. But then we run out of items on the first list. There was only one. So longest lacing says repeat the last item on the list with each of the subsequent items on the longer list. So, if you have only one path, then longest lacing will work. But there is one more option: Cross Product. Let's explore it.

24. Wire **Select Model Elements** back into **Element.Curves**.

25. Right-click on the **Curve.PointAtParameter** node and choose: **Lacing>Cross Product**.

The final option: Cross Product, matches *all* possibilities. So, you can end up with many pairings! In this case, if you want to add trees to all locations on both paths, then Cross Product would do the trick (see Figure 39).

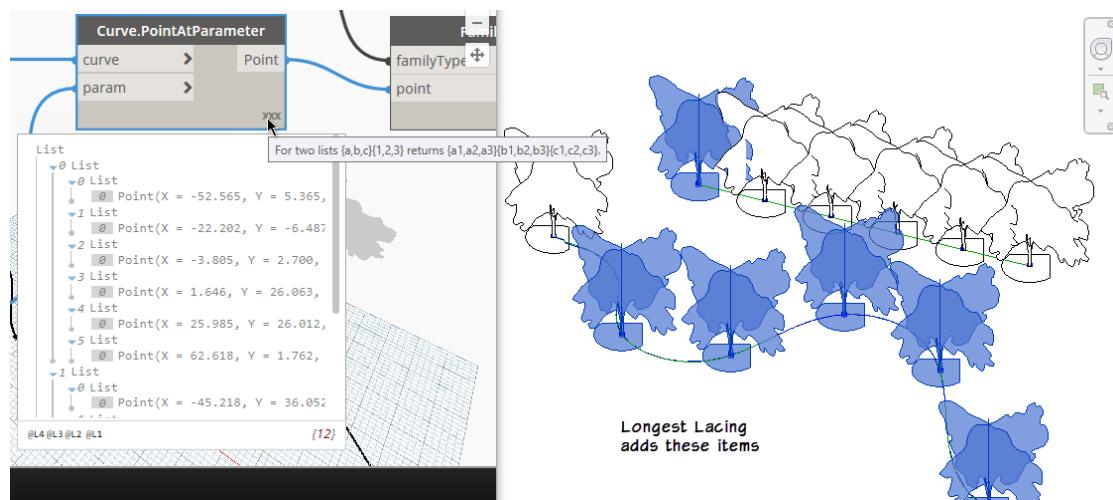


Figure 39

You may also want to take notice of the structure of the list that is generated by the **Curve.PointAtParameter** node in each case. The list structure with lacing set to: shortest is a list of lists. The parent list contains two sub-lists. Each of those lists contain a single point. With lacing at longest, we also get a list of lists. But this time there are six lists each with one point. When we go to cross product, we now have 3-level nested list. The top level is two lists that each contain six lists. Those six lists contain a single point each.

Once you have settled on the proper lacing to get your desired result, as you drag the slider, you will see the quantity of trees adjust as you drag the slider. You can also play with the limits on the slider to vary the resultant quantities.

**SAMPLE FILE:** You can open a file completed to this point named: **02\_Place Families\_E.dyn**.

26. Close the Dynamo graph and the Revit file. Keep Revit running.

## SUMMARY OF KEY CONCEPTS

Here is a summary of the most important concepts covered in the previous exercise:

- The simplest way to select Revit elements is using the **Select Model Element** or **Elements** nodes. Element (singular) select just one Revit element. Elements (plural) selects two or more.
- “AtParameter” nodes create points or divisions within a range from 0 to 1. You can feed in a single value in this range or a list of values.
- Nodes in an error state will turn red onscreen. Nodes that are in a warning state will turn yellow onscreen to indicate that they are not satisfied.
- Sliders are a convenient way to interactively change values. They work best in Automatic execution mode.
- In its simplest form, Dynamo is about managing lists of data.
- Create a list of items quickly using Ranges and Sequences.
- List matching is the process of selecting an item on one list and matching it up with the corresponding item on another list. Lacing options let you customize how this occurs.
- Shortest Lacing matches the first item on List A with the first item on list B and then moves to the second item on each list until it runs out of items on the shorter list. The remaining items on the longer list are ignored.
- Longest Lacing matches the first item on List A with the first item on list B and then moves to the second item on each list until it runs out of items on the shorter list. The remaining items on the longer list are matched to the last item on the shorter list.
- Cross Product Lacing matches every item on each list with every other item on the other list.

## BONUS MATERIALS

Above we briefly introduced the concept of Code Blocks in the “Visual Programming is Programming” topic. In that discussion we indicated that Code Blocks would not be critical to our success in the lessons here in the workshop. However, it was mentioned that Code Blocks are used by many Dynamo users and often provide a nice shortcut to many common tasks. You

will see Code Blocks often used for simplifying code, reducing the number of nodes required or speeding up the process of creating a graph.

In this exercise, we used a **Range** node to generate a list for the AtParameter values. Using a Code Block and some common Design Script<sup>14</sup> we can directly code sequences, ranges and other common code syntax. If you want to see some examples of creating ranges and sequences with Code Blocks, open the graph named:

**BONUS FILE:** Code Block samples for Ranges: [10\\_01\\_NumberRangesAndSequences.dyn](#).

The preceding exercise also looked at the concept of lists and lacing. As noted, a huge part of Dynamo is working with, understanding and managing lists. There are two bonus files that showcase these important concepts:

**BONUS FILE:** Lists and Lacing examples: [10\\_02\\_ListLacing.dyn](#) and [10\\_03\\_ListManagement.dyn](#)

## PLACING MULTIPLE SYSTEM FAMILIES

The previous example placed one or more instances of the same family and type. You may also wish to place more than one family or type. One of the places where Dynamo really excels is in performing repetitive tasks. In this next example let's assume you wanted to create a single project file that would serve as a library or "warehouse" file. Users can then open the project file, locate the item they want and more importantly see an instance of each element already inserted onscreen in the file. Then they can simply select the one they need and copy and paste it to their project. Many firms build these so-called warehouse files. But without a tool like Dynamo, it can be quite tedious to create one. In this example, we'll use Dynamo to create a simple wall type warehouse using out-of-the-box wall types.

After that, we'll create a second warehouse using out-of-the-box casework families. There will be similarities in each graph, but some significant differences as well. This is because walls are system families while casework are component (loadable) families. You are likely already familiar with some of the differences between system and non-system families. Well, they differ under the hood in the API as well. Dynamo is really like our bridge to the API, and while visual programming does shield us from much of the API issues and complexities, it does not conceal all of them. So, keep in mind that Dynamo can only perform tasks that are possible through the Revit API. It just provides a more accessible way to do so that is easier to learn for many users.

## REPURPOSE AN OLD GRAPH

Recalling the work we did in the "Hello Wall" examples above, we will be able to reuse much of the graph that we built there. Our wall warehouse will need to make points by coordinates, draw lines and then place walls on these "curves." Therefore, the easiest thing to do is save a copy of that graph to a new name. You can begin many Dynamo graphs this way. It saves time and allows you to maintain consistency. It can also be good to re-build them from scratch as well, as there is much that can be learned and reinforced by building a new graph for a familiar solution. So, either approach is perfectly fine, but here to save time, we'll reuse the old one.

---

<sup>14</sup> Design Script is the name of the coding language that Dynamo uses. All programming uses programming languages. Just like spoken language, a programming language has grammar (syntax), rules, vocabulary (commands), etc.

---

**Note:** If you did not close the previous files, do so now. It is also a good habit to quit Dynamo entirely and restart it before beginning a new graph.

---

1. In Revit, close all files and then open the file named: **03\_Wall Warehouse !Start.rvt**.
2. Launch Dynamo and then open the file named: **03\_Wall Warehouse\_A.dyn**.

This is a copy of the completed graph from the “Hello Wall” exercise above. Let’s start by deciding what we can reuse and what we will not need. In the exercise above, we fed in comments to the wall we created. We will not need to do that for this example. So, we can delete those nodes. The rest of the graph we will be able to reuse.

3. Select the **Element.SetParameterByName** node and the two **String** nodes that feed it and delete them.
4. From the File menu, choose: **Save As**.
5. Browse to the same location as the project file, name the new graph: **Wall Warehouse** and then click Save.

#### DOCUMENTING YOUR GRAPH

As you begin creating Dynamo graphs, you are encouraged to develop good habits early on. One such habit is to document your work as you go. Dynamo has notes and groups to help us with this task. These tools allow the author of a graph to provide documentation directly within the graph. Documenting your graphs is **highly** recommended. It does not matter if you use groups, notes or both. But do make sure that you use at least one of them and make comments about what you did in the graph. This helps both your recipients (to understand how the graph functions) and yourself (when you open the graph later and try to remember what you did). Adding notes and creating groups is simple to do. Let’s practice doing so in the graph we just saved<sup>15</sup>.

6. From the Edit menu, choose: **Create Note** (or press CTRL + W).

A note will appear onscreen called “New Note.” (It is usually in the middle of the screen).

7. Drag the new note above the first **Point.ByCoordinates** node.
8. Double-click directly on the note. In the dialog that appears, type: **Line Start Point** and then click the Accept button.
9. Create a second note, position above the other **Point.ByCoordinates** node, double-click it and change it to read: **Line End Point** (See Figure 40).

---

<sup>15</sup> It is also possible to double-click the title of a node and rename the node itself. This does provide another way to document the graph, however, it is preferable to use notes and groups instead. Once you rename a node, it is easy to forget what the node was originally. The exception is inputs for Dynamo Player.

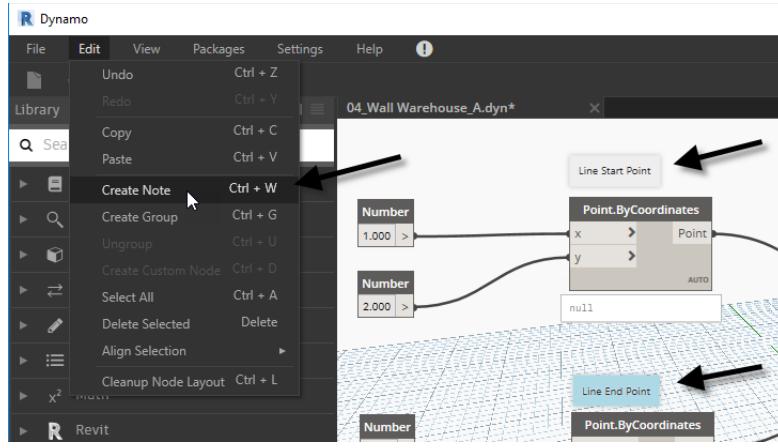


Figure 40

As you can see, notes are simple to add and provide an easy way to document your graph. Another way to organize your graph and make it more understandable is to create groups. You can select a collection of nodes and group them.

10. Drag a selection box around the **Point.ByCoordinates**, **Line.ByStartPointEndPoint**, the four **Number** nodes and the two notes (9 items in all).

11. From the edit menu, choose: **Create Group** or press CTRL + G.

A colored box will appear around the selected nodes. It will be titled: <Click here to edit the group title>.

12. Double-click anywhere on the group to rename it. Call it: **3. Create Points and Lines** and then click away from the group to accept the new name.

The name can be a full description if you like, or just a few words. We numbered it 3 here because we will end up with four groups total when we are finished and the numbers will help explain the flow of data through the graph. Right-click<sup>16</sup> a group to optionally customize its color and font size (see Figure 41).

---

<sup>16</sup> You can create groups using right-click also. But please note that the options on the right-click menu are context sensitive and will vary depending on whether you right-click a selection of elements or in the empty canvas.

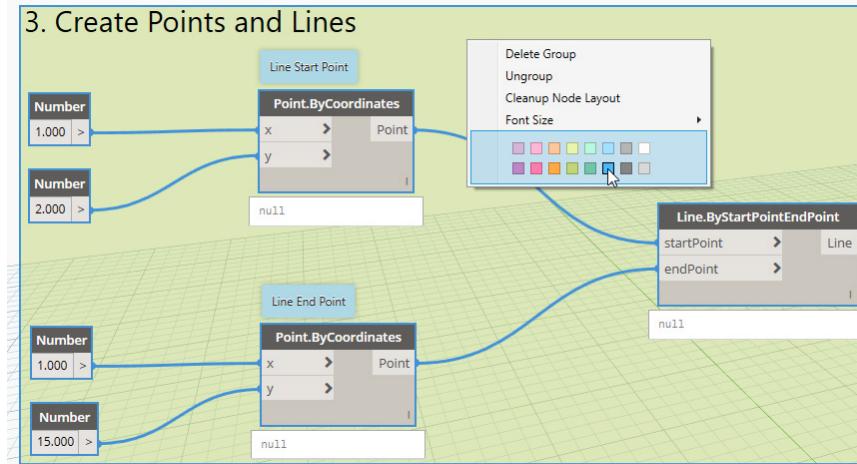


Figure 41

There are also options on the right-click to ungroup and delete. The “Cleanup Node Layout” option will move the nodes around to make them more compact and align them to each other. There are also several alignment options on the Edit menu under **Align Selection**. To use them, select at least two nodes and then choose your preferred alignment such as left, right, top, etc.

If you select a group, it will select everything in the group. This makes it easy to move the entire collection around. If you instead select just one or more elements within the group and move them, they will move independently and resize the group shading accordingly. Wires are unaffected by grouping. You can wire within a group or to items outside the group.

13. Select the remaining nodes (you can drag a box around them all) and press **CTRL + G** to group them.
14. Double-click and name the new group: **4. Create a wall for each type.**
15. Optionally change the color and adjust position.<sup>17</sup>
16. Save the graph.

**SAMPLE FILE:** You can open a file completed to this point named: **03\_Wall Warehouse.B.dyn**.

### CREATING A WALL WAREHOUSE

The graph we need to create is not too complicated. It will behave in nearly the same way as the Hello Wall example. The only difference is that we will create and place several walls instead of just one. We will need to supplement what we have here with nodes that get a list of all the wall types in the project, and then instead of creating one line, we need to create a line for each wall in the list.

Let's start with the lines. Right now, we have code that takes four numerical inputs to create two points and one line. To make several lines, we need several points. We could start copying and

<sup>17</sup> Many firms use standard color and naming conventions for their groups. One color for inputs, another for processing, another for output, etc. This makes scripts more consistent throughout the office and allows users to understand the overall structure on sight.

pasting the nodes, but that would be terribly inefficient. Instead let's think like a programmer and find a way to reuse what have in a more efficient way. Thinking back to the tree on the spline example that we just completed, let's remember an important concept in Dynamo: Most inputs can take either single values or lists! Therefore, what if instead of feeding in a single number to the **Point.ByCoordinates** node, we instead feed in a list? This is all that is required to create as many points as we require!

### CREATE THE CURVES (LINES)

What do we require to create several lines instead of just one? We are trying to create several walls, one of each type. Therefore, we will want a number of pairs of X, Y points equal to the number of walls we plan to create. To achieve this, we simply make sure that we feed in a list to at least one of the inputs. Let's make the X values vary and keep the Y values constant. This will draw our lines and walls along the X axis (horizontally). So, the x port is where we feed in our lists. They will both use the same list.

But we first need to create the list. We looked at the **Range** node above which gave us a series of numbers. **Range** uses a start and end value and then creates a list of values in between those two. That would require a calculation in this case to end up with the correct quantity. Instead, we'll use a **Sequence** node which allows us to input the desired quantity directly as one of its inputs.

17. To the left of the existing groups, add the following nodes to your canvas:

Library Location	Node
List > Generate	Sequence
Input > Basic	Number Slider

18. Feed the Number Slider into the step input.

The step input tells the Sequence how far to space each value. A step of 1 would create a sequence like: 1, 2, 3, 4, etc. A step of 3 would create: 3, 6, 9, 12, etc. In our graph, the step will be the desired X distance between each wall. We can configure the limits in the slider to give us a good range of values for this.

19. Click the small chevron icon on the left of the slider to reveal its settings.
20. Set the Min to: **5** and the Max to: **15**. For the Step, input: **0.5**. and then close the controls by clicking the chevron again.
21. Press CTRL + W to add a new note, position next to the slider and change it to read: **Use this slider to customize the spacing in the X direction.**
22. Select the slider, sequence and note and then press ctrl + g to group them.
23. Rename the group: **2. Establish the wall spacing** (see Figure 42).

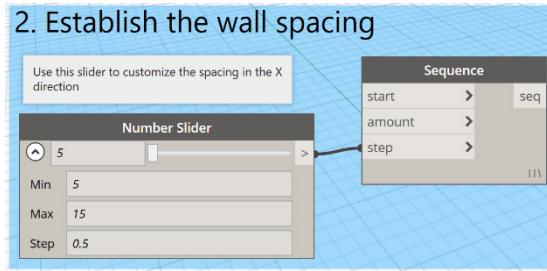


Figure 42

24. In group 3, delete the **Number** nodes feeding the **x** ports of each **Point.ByCoordinates** node (two total).

Do not delete the ones feeding the **y** ports.

25. In group 4, right-click the **Wall.ByCurveAndLevels** node and choose: **Freeze**.

This prevents this node and any node downstream from it from executing until you unfreeze it. This is a great way to test small portions of the graph without having to run the entire thing.

If not already set, change the execution mode to: Automatic.

26. Wire the **seq** port from the Sequence node into both **x** ports of the **Point.ByCoordinates** nodes.

Notice that we now have ten lines. But why did we get ten? If you hover your mouse over both the **start** and **amount** ports on the **Sequence** node, you will see that they default to 1 and 10. So our sequence starts at 1 and has 10 total items. Starting at 1 is fine. No need to change that. But we need the amount to match the number of walls we intend to create. To get that value however, we need to set up the last part of the graph; group 1.

**SAMPLE FILE:** You can open a file completed to this point named: **03\_Wall Warehouse\_C.dyn**.

**Note:** The **Range** and **Sequence** nodes provide simple ways to create these lists, but there are many more possibilities if you use Design Script code in a Code Block. The bonus file noted in the “Bonus Materials” topic above gives some examples. If you wanted to replace this Sequence with a Code Block, you would type something like: **1..#10..5**. If you substitute the numbers with text, they will become variables and create input ports. So: **1.#Quan..Spc** would be a sequence from 1 to a variable quantity (represented by: **Quan**) and with a step of **Spc** (see Figure 43).

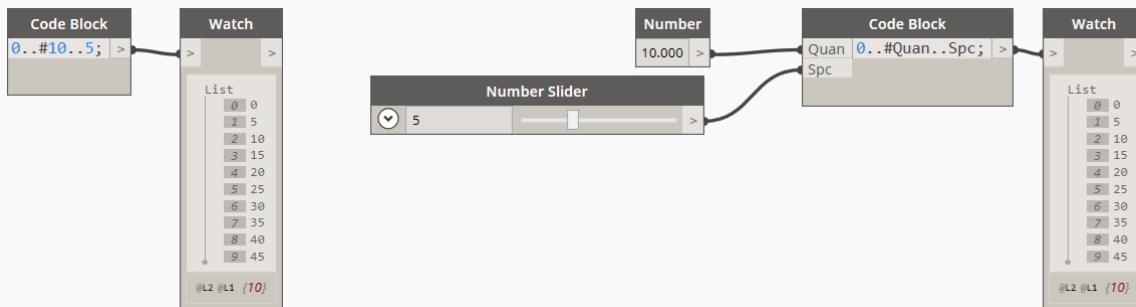


Figure 43

### MAKE A LIST OF ALL WALL TYPES

We need to make a list of all the wall types in the file. To do this, we wire an **Element Types** node into an **All Elements of Type** node. The drop-down list on the first node lists all element types in Revit. We simply choose: **WallType** to get a list of all wall types in the current project. To see the list, we can use the preview bubble or a **Watch** node. We can scroll through that list to see how many there are, but a **List.Count** node gives us this value and makes it available to downstream in the graph. And the nice thing about this is, if you add or delete types, the graph (and the count) will update.

27. To the left of the existing groups, add the following nodes to your canvas:

Library Location	Node
Revit > Selection	Element Types
Revit > Selection	All Elements of Type
List > Inspect	Count
Display > Watch	Watch (Optional)

28. Wire them up as shown in Figure 44.

29. From the **Element Types** node, select **WallType** from the drop-down list.<sup>18</sup>

30. Group the new nodes and name the group: **1. Create a list of all Wall Types**.

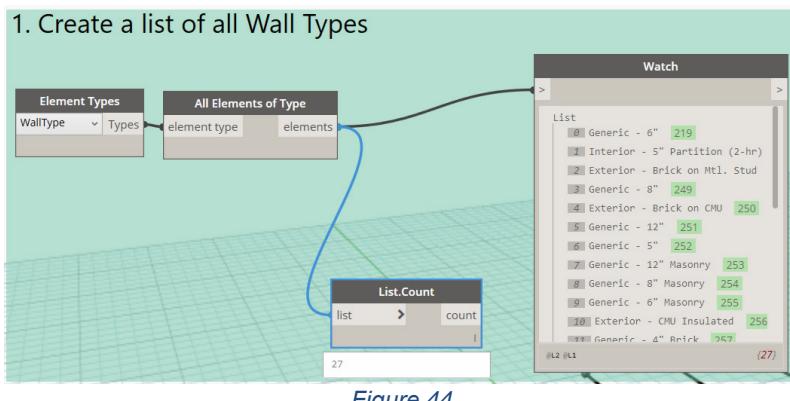


Figure 44

Your graph should still be set to Automatic.

31. Wire the count port from **List.Count** to the amount port of the **Sequence**.

This will execute immediately, and you will now have 27 lines instead of 10.

32. In group 4, right-click the **Wall.ByCurveAndLevels** node and choose: **Freeze** again.

<sup>18</sup> To select from the list quickly, click on the drop-down, then type the letter "w". This will jump to the first item whose name starts with "w." Then scroll from there and choose: **WallType**.

This will unfreeze the node and immediately execute it. You will now have 27 walls in Revit and if you have an **Element.Geometry** node, you will see them in Dynamo as well. However, currently all 27 walls use the same wall type. This is because the **Wall Types** from the original graph is still connected.

Wire the **elements** port from the *All Elements of Type* node in group 1 to the **wallType** input of the *Wall.ByCurveAndLevels* node in group 4.

There will be a slight pause as the graph updates and you should now have one of each wall type placed in Revit. Dynamo will still display simple representations however.

33. Select the **Wall Types** node in group 4 (Click directly on the node, not the group) and delete it.

**SAMPLE FILE:** You can open a file completed to this point named: **03\_Wall Warehouse\_D.dyn**.

### MAKING UPDATES

An important benefit of this type of graph is it is flexible and will keep up with changes in the Revit file. Since we are asking Dynamo to give us a list of all wall types in the Revit project, if you add a new wall type, it will immediately get added to the model the next time the graph executes. This is also true should you delete a wall type. To see this for yourself, expand the Families branch on Project Brower, expand walls, and then duplicate an existing wall. If Dynamo is still running and set to automatic, a new wall will appear onscreen immediately before you even rename it!

Naturally if you plan to make lots of edits, it might be prudent to close Dynamo or set it to manual update until you are finished in Revit, then run the graph again.

## SUMMARY OF KEY CONCEPTS

Here is a summary of the most important concepts covered in the previous exercise:

- A quick way to get started with a new graph is to save a copy of a similar one and modify it.
- Document your graphs as you work with notes and groups.
- Notes offer a simple piece of text that can contain any description or text you wish.
- Grouping nodes collects them in colored regions that can be named. This helps keep the graph more organized and easier to understand.
- Many inputs can take single values or a list. Using a list allows you to process multiple values through the node instead of having to make multiple copies.
- Freeze nodes as you work to run just selected portions of your code. Freezing affects the selected node and anything connected downstream.
- When you use selection nodes like select by category or all elements of type, the selection remains dynamic and will adjust to changes made in Revit in real time.
- You will use different nodes to select Revit system families than you will use for loadable families. **Element Types** will select system families, **Family Types** will select loadable families.

## PLACING MULTIPLE LOADABLE FAMILIES

The procedure for working with component (loadable) families differs a bit from system families. This is true in the Revit interface and certainly in the API; and by extension in Dynamo. For one, system families are already built-in to all Revit projects. Component families must be loaded. So, if you want to create a library file of families in a component category such as furniture or casework, you will need to ensure that you have some items of that category loaded. While there are ways to load from external files directly in Dynamo, in this workflow, we will start with a file that has some casework pre-loaded and focus our graph on placement of the families in the project. The starter file contains all the families from the out-of-the-box library that are contained in the *Casework\Base Cabinets* folder. This collection was chosen because they are free-standing and do not require wall hosts, which makes them easy to place with Dynamo.

The goal here is the similar to the previous example. We will ask Dynamo to collect a list of all the casework families and types in the project file. Then we'll place one of each in the model. We could do this similarly to how we did the walls and simply line them all up in a long row. But in this case, since there will be several casework families, each with one or more types, it might be more desirable to organize them in rows for each family and columns for the types. To achieve this, we'll need to pay close attention to how we acquire and process the lists in our graph.

### PACKAGES

Dynamo contains hundreds of built-in nodes exposing a great deal of functionality. But even with all the built-in nodes, you will still find times when the task you wish to perform is not available in the built-in nodes. This is when we can turn to Packages. Packages are custom nodes created by the community. However, before you can use them, you must install them.

Close Dynamo and Revit.

1. Open a web browser and navigate to: <https://provingground.io/tools/lunchbox/>
2. Click the Download Lunchbox for Dynamo and Grasshopper button and on the next page, the Download Lunchbox button.
3. Save it to your Desktop or Downloads folder and then run the installation.
4. When the installation is complete, launch Revit 2019.
5. In Revit, open the file named: ***04\_Casework Warehouse\_!Start.rvt***.
6. Launch Dynamo and then from the Packages menu and choose: **Search for a Package** (see the left side of Figure 45).
7. In the “Online Package Search” dialog that appears, type: **Clockwork** into the Search field.
8. When Clockwork for Dynamo appears, locate Clockwork for Dynamo 2.x and then click the arrow icon on the left to download and install it (see the middle of Figure 45).

9. Click OK in any messages that appear to confirm download and installation<sup>19</sup>.

10. Close the “Online Package Search” dialog.

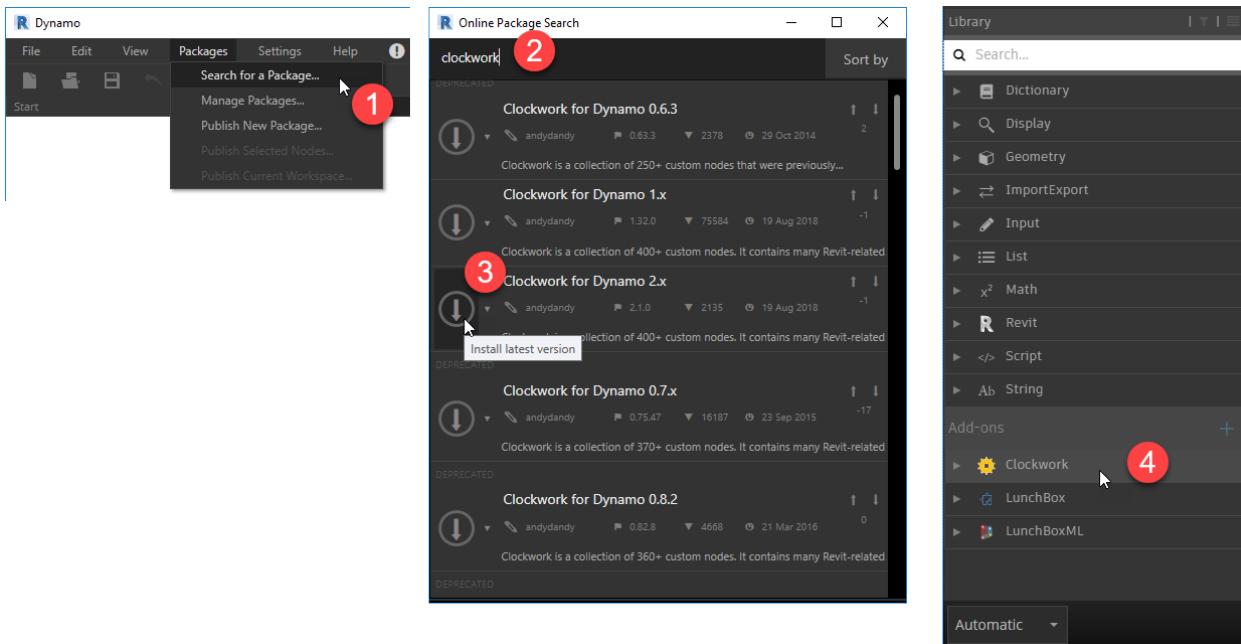


Figure 45

If you search for Lunchbox in the Package Manager, it does show up, but the install available does not properly install it for Dynamo 2.0.1. There is a note to this affect when you click on the Lunchbox item in Package Manager. This is why we installed it from the Provingground web page instead. Once installation is complete, you will see a new Add-ons branch in your library panel. It will show Clockwork and LunchBox (see the right side of Figure 45).

There are hundreds of Packages available that perform all sorts of useful functions. Feel free to explore and install others if you wish. Keep in mind that like any community driven initiative, the quality and utility of Packages will vary. A good strategy is to only install a Package if it has a node that fulfills a certain task you are trying to solve in your graph. If you are concerned about installing Packages, just take a few moments to do a Google search on the Package you are considering. The Dynamo community is very active and you will likely find many posts about the Package you are considering with users giving you the straight talk on the best ones available out there. And of course, you can always uninstall ones you no longer find useful. You can also learn more about packages at: <http://dynamopackages.com/>

<sup>19</sup> Clockwork will alert you that it has “dependencies.” It is safe to click OK here and dismiss that warning. However, when contemplating the installation of Packages, keep in mind that they are developed by individuals and third parties. It’s worth vetting them in a somewhat safe environment, or getting approval and guidance from your BIM manager or other IT support staff.

### CREATING A CASEWORK WAREHOUSE

Let's start a new graph and use the Clockwork package we just installed to get a list of all the casework types in the current file.

1. In Dynamo create a new graph.
2. Save the graph as: **Casework Warehouse**.
3. Add the following nodes to your canvas:

Library Location	Node
<b>Revit &gt; Selection</b>	<b>Categories</b>
<b>Clockwork &gt; Revit &gt; Selection &gt; Collectors</b>	<b>All Family Types Of Category</b>
<b>Revit &gt; Elements &gt; FamilyType</b>	<b>Family</b>
<b>List &gt; Organize</b>	<b>List.GroupByKey</b>
<b>Display &gt; Watch</b>	<b>Watch (Optional)</b>

4. Wire the output from the **Categories** node into the **category** input of the **All Family Types Of Category** node.
5. From the drop-down list on the **Categories** node, choose: **Casework** and then expand the preview bubble (see Figure 46).

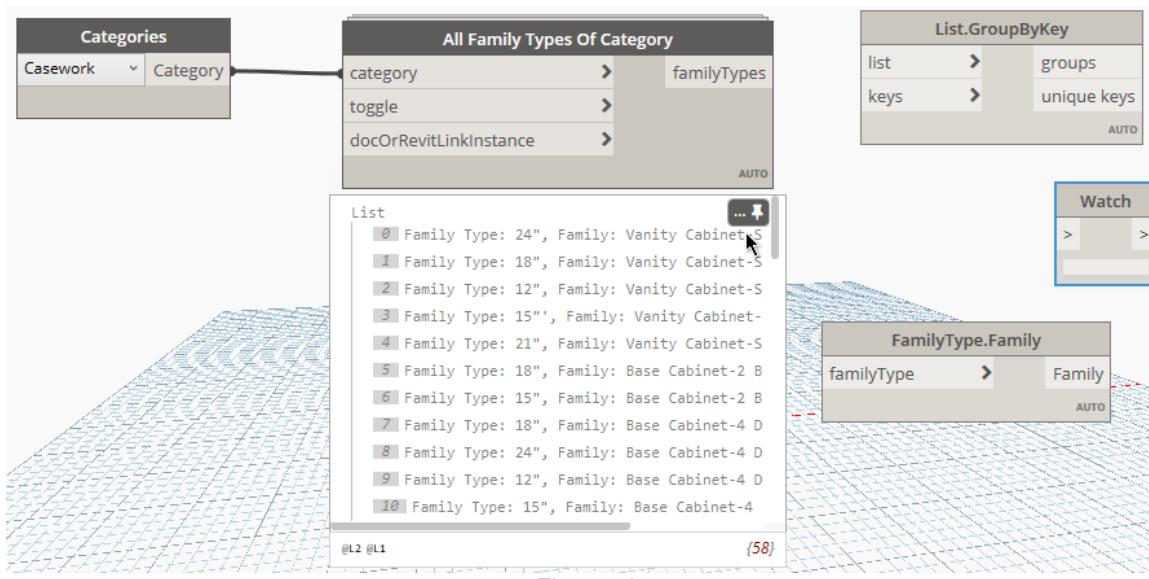


Figure 46

This gives us a list of all the casework types. But as you can see, this is a flat list with no structure. But since our goal is to place our casework in groupings by family, we need to process this list and introduce structure based on the family names. In other words, if you look at each item coming out of the Clockwork node, it lists the type name and then the family name. So, we will

extract the family names from this list using the **FamilyType.Family** node and then use them to create sublists based on each family. We can use **List.GroupByKey** for this purpose.

6. Wire the output of **All Family Types Of Category** node into the **FamilyType.Family** node.

If you expand the preview bubble you will see that it lists only the family names now for each entry. We can take this simplified list and use it to group the main list. The **keys** input on the **List.GroupByKey** node uses the values input to it to group the list fed into the **list** input. Here we are going to feed our original list into the **list** input and our simplified (families only) list into the keys. For each item on the list, it will look at the corresponding index on the key list. The output will be a structured list where the keys become sublists containing the items (family types) from the main list.

7. Feed the output of the **FamilyType.Family** node into the **keys** input of the **List.GroupByKey** node.

8. Feed the output of **All Family Types Of Category** node into the **list** input of the **List.GroupByKey** node.

9. Wire the **groups** output into the **Watch** node.

10. Select all the nodes onscreen and group them. Edit the name (see Figure 47).

1. Get all Casework Types

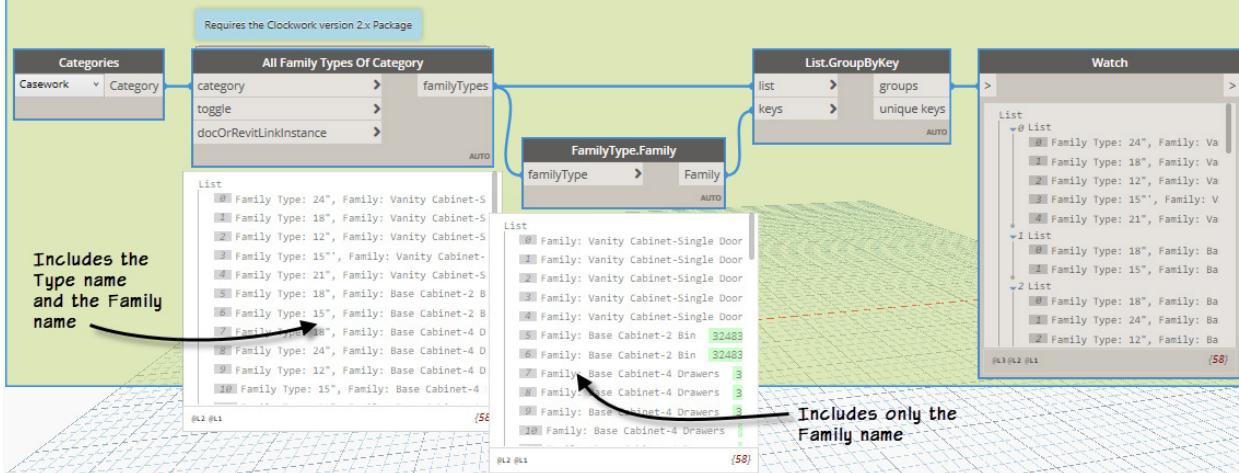


Figure 47

**SAMPLE FILE:** You can open this Dynamo file named: **04\_Casework Warehouse\_A.dyn**

## PLACE THE FAMILIES

Now that we have the list of families and types, let's add some nodes to place them.

11. Add the following nodes to your canvas:

Library Location	Node
Revit > Elements > FamilyInstance	ByPoint
Geometry > Points > Point	ByCoordinates (X,Y)

12. Wire the **Point.ByCoordinates** into the **FamilyInstance.ByPoint**.
13. Select both nodes and group them.
14. Wire the **groups** output (from the **List.GroupByKey** node) into the **familyType** input of **FamilyInstance.ByPoint** node (see Figure 48).

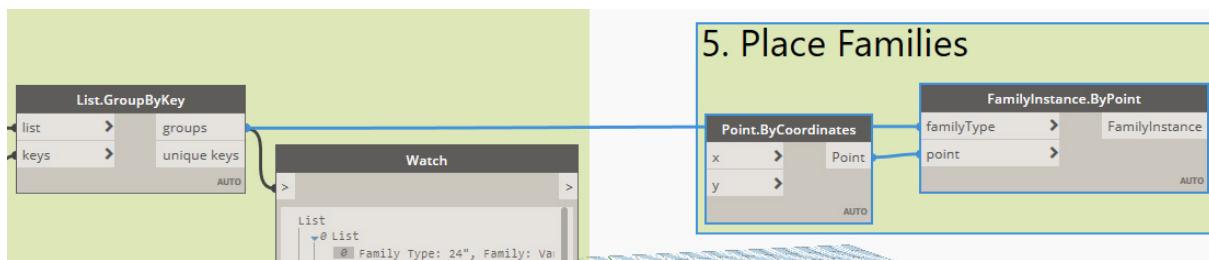


Figure 48

If you have the graph set to execute automatically, you will notice that in Revit all 58 families have been placed directly on top of each other. Naturally this is not quite the result that we desire. We need to build a list of X,Y coordinates in the desired row and column structure and feed that into the **Point.ByCoordinates** node to correct this.

**SAMPLE FILE:** You can open this Dynamo file named: **04\_Casework Warehouse\_B.dyn**

### CALCULATE THE X VALUES

As noted above, we'll place the families along the y-axis and their respective types along the x-axis.

15. Change the execution mode of the graph to Manual.
16. Add the following nodes to your canvas:

Library Location	Node
Revit > Elements > Element	GetParameterValueByName
Input > Basic	String

17. Wire the **String** into the **parameterName** input.
18. In the **String** node, type: **Width** (case sensitive) and then click outside the node (do not press ENTER) to accept it.
19. Group the two nodes and title the group: **2. Get Type Widths**.
20. Disconnect the wire from the **familyType** input on the **FamilyInstance.ByPoint** node and connect it to the **element** input on the **GetParameterValueByName** node instead.

21. Run the graph and then click the preview bubble (see Figure 49).

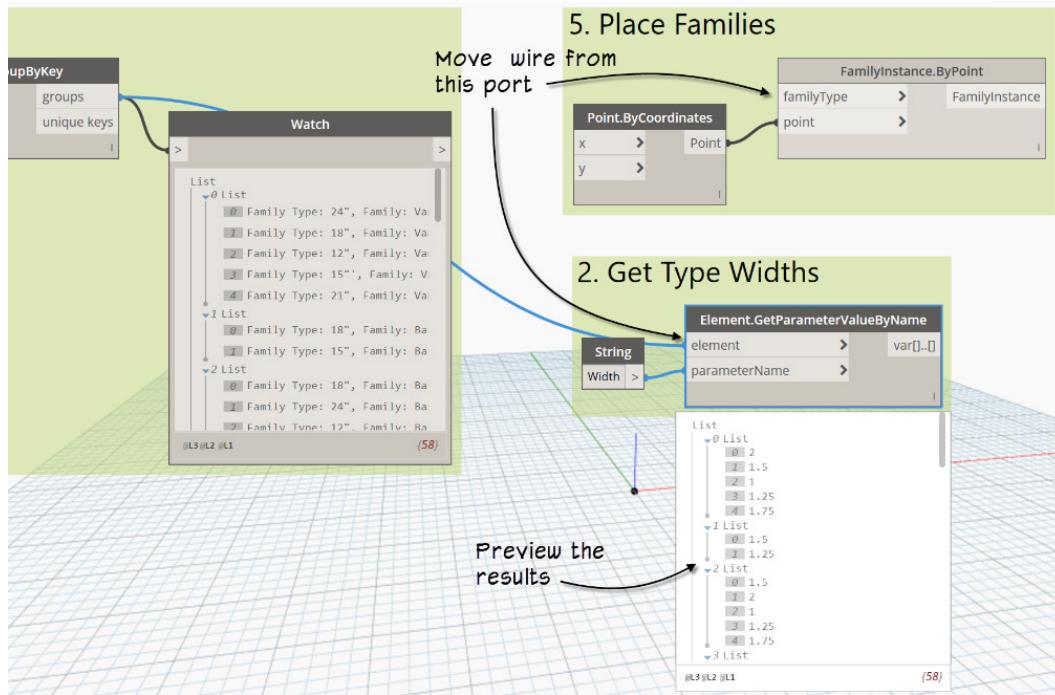


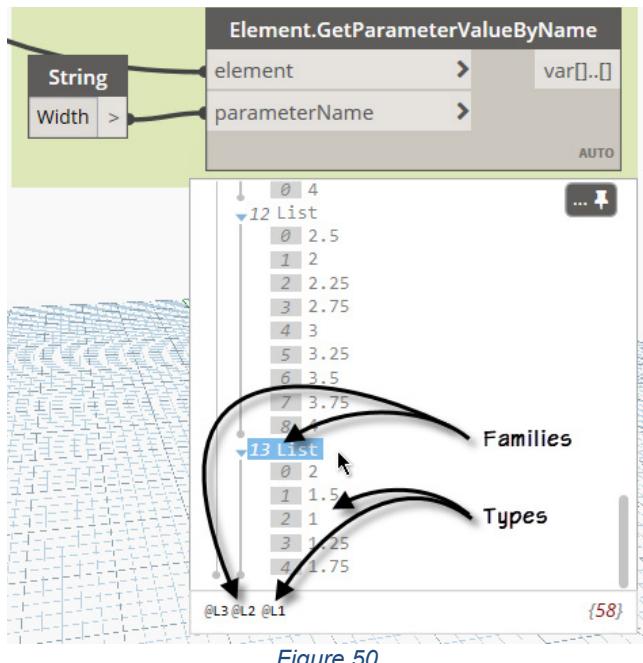
Figure 49

In an earlier example, we used the **SetParameterValueByName** node to write comments on elements in the Revit model. In that case, we were writing to an instance property. The **GetParameterValueByName** node that we are using here is very similar. The only difference is that it reads the value of the property rather than writing a new value as the set node does.

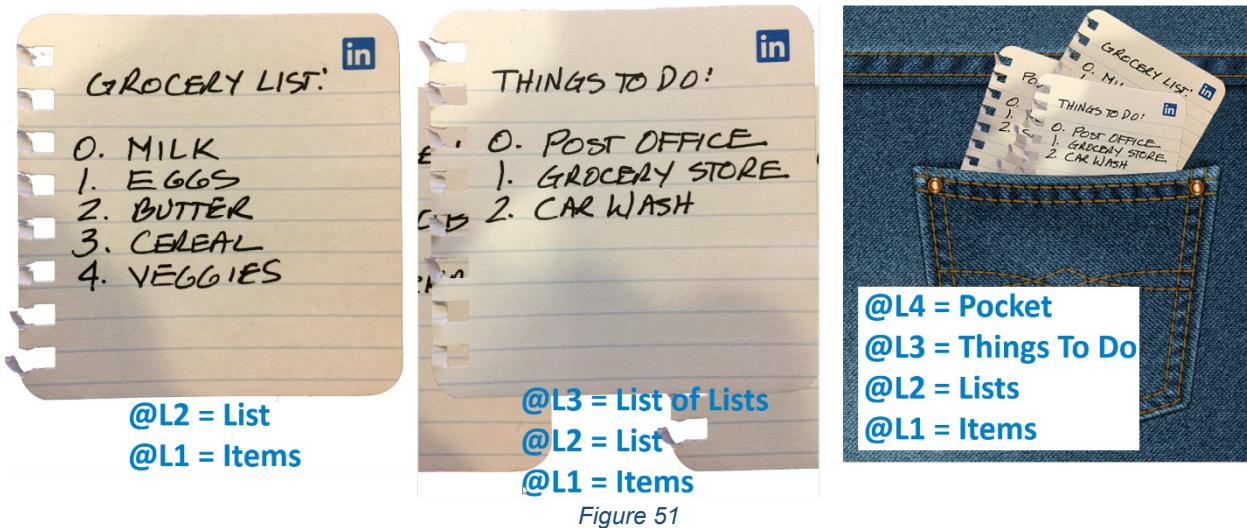
But there is another important difference in how we are wiring this graph. Notice that this time, we are working with a type property instead of an instance property. Both nodes (Get and Set) can operate on either instance or type properties. It is therefore very important for you to know if the property you are accessing is type or instance and then feed in the correct inputs accordingly; otherwise you will get unexpected or failed results.

#### LIST AT LEVEL

Look carefully at the structure of the list coming out of the **GetParameterValueByName** node. The output is a nested list, or a list of lists. There are 14 lists (list 0 through list 13) that each contain a variable number of elements. At the bottom of the preview bubble are three list level indicators: @3 @2 @1. The level 3 list is the main list itself. The level 2 list (with the 14 items) contains our families. The level 3 list contains the types for each family. The total number of types across all the nested lists is shown at the bottom right: 58 (see Figure 50).



Understanding the list structure is useful because you can access items at each level of the list. Think of it this way, when you need to do some errands, you often make a list so that you remember everything you need to do. So, let's say you are going to the grocery store. You might make a list of items you need to buy (see the left side of Figure 51). If this was the only place you needed to go, you might need only the one list. But let's say you had several stops to make. You might create a list for each place you plan to visit. Now you could just stuff all these lists in your pocket and off you go, or you can make a master list that showed each place you needed to stop on your errand run and then you could focus on just the appropriate "sub-list" while at each location (see the middle of Figure 51). And then you might just stuff all of those in your pocket: the ultimate "top-level" list (see the right side of Figure 51).



Using the “list at level” functionality, you can address items on each level of the list by simply indicating at which level of the list you want to operate. We will see examples of this next.

Remember that the numbers on the list are the widths of each type. But if we use these numbers directly the families will be inserted right next to each other with no space in between. Let’s add a little space in between each item. Also, let’s since these numbers will be used as the X coordinates, the first item on the list is actually the location of the second type. So, we need to add zero to the front of each list, and then to keep the quantity of each list accurate, we need to remove the last item from each list.

22. Add the following nodes to your canvas:

Library Location	Node
Input > Basic	Number Slider
Input > Basic	Number (add 2)
Math > Operators	+
List > Modify	AddItemToFront
List > Modify	DropItems

23. Wire and group the nodes as indicated in Figure 52.

24. Input the numerical values as indicated in the **Number Slider** and **Number** nodes as well and run the graph.

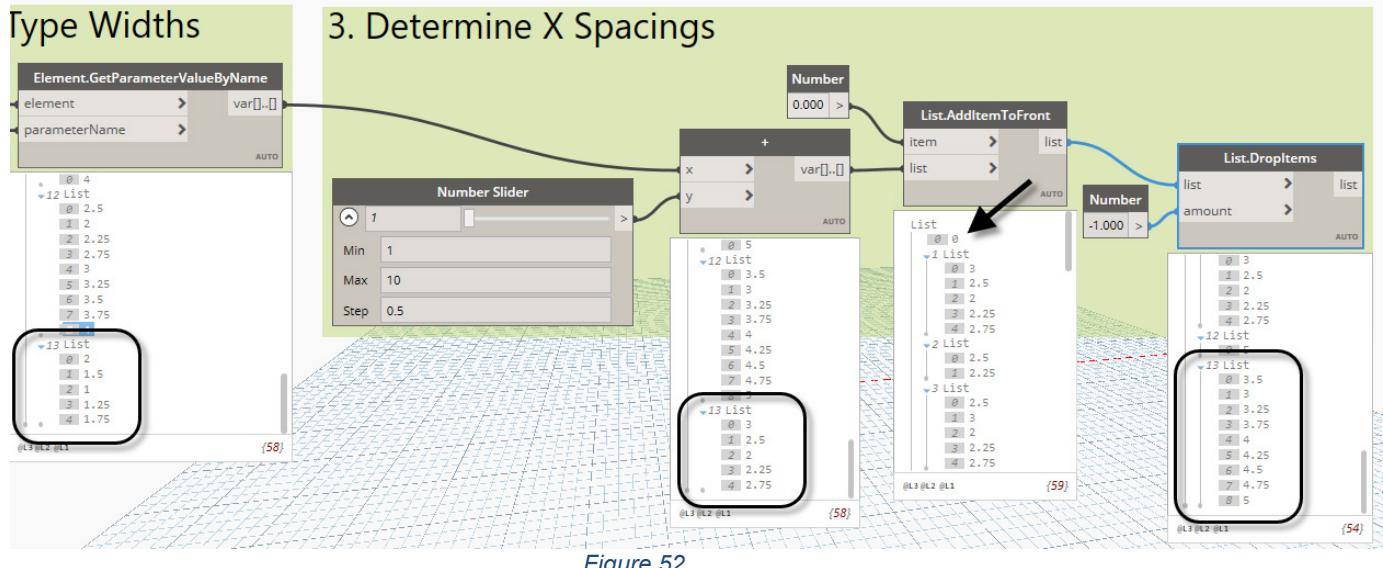


Figure 52

The slider allows you to add spacing to each X value interactively. This is added to the list of the widths (from the family types) with the + node. Compare the output from the **Element.GetParameterValueByName** to the + node. This is working correctly. Next, the **List.AddItemToFront** node adds zero to the top of the list, but we wanted zero added to the

front of each of the 14 sublists (at level 2), but we ended up with it's being added to the front of the main list at level 3 instead. Furthermore, the **List.DropItems** node removes one or more items from a list. Positive numbers remove from the start of the list, negative numbers remove from the end of the list. So, with a **-1** input it removes one item from the end of the list. We want this to compensate for the addition of zero to the front. But again, in this case it is removing at the wrong level – one of the 14 sublists, not the last item from each of the sublists as we require. We can fix both issues by employing the “List at Levels” option.

25. Click the small little triangle icon on the **list** input of the **List.AddItemToFront** node, and then check the “Use Levels” checkbox.
26. Repeat this for the **list** input on the **List.DropItems** node as well.

It will default to **@L2** which means apply the function to the level 2 list. This is what we want here so no need to change it further. If you need it to apply to a different level, click the small spinner icons to choose a different level.

27. Run the graph to see the effect.

Notice that we are now getting a zero entry as the first item on each sublist and that the last item of each sublist has been removed. You can look at the totals at the bottom of the preview bubbles to see how the lists change at each node (see Figure 53)

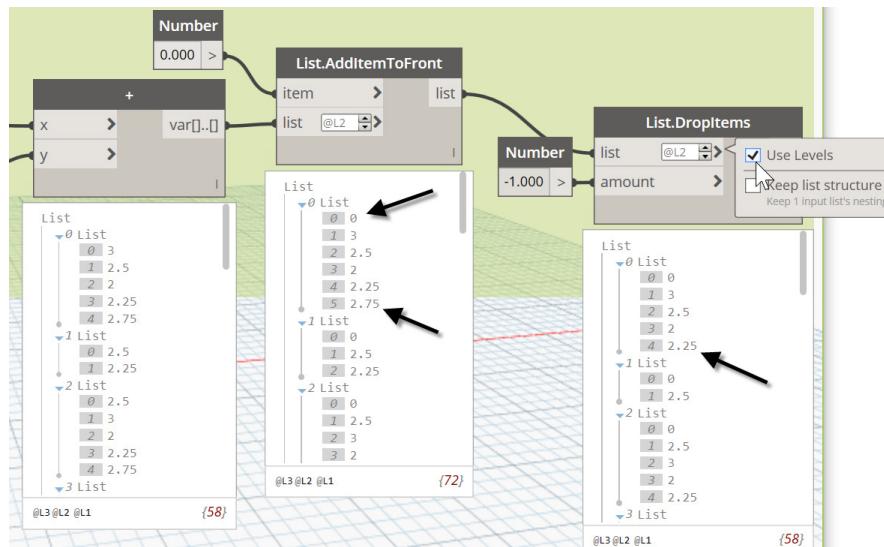


Figure 53

**SAMPLE FILE:** You can open a file completed to this point named: **04\_Casework Warehouse\_C.dyn**

28. Add the following nodes to your canvas:

Library Location	Node
Lunchbox > Math > Operators	Lunchbox Mass Addition
Display > Watch	Watch

29. Wire the **List.DropItems** output into **Lunchbox MassAddition** node.

30. Enable the At Levels for @L2 for the **Numbers** input.

31. Wire the **Partials** output into the **Watch** node.

We are using a Watch node here since there are two outputs. The preview bubble would show both outputs as a nested list, but with the Watch we can focus on just the one we need.

32. Select the **Lunchbox MassAddition** and the **Watch** node, hold the SHIFT key down and select group 3.

33. Right-click on either the **Mass Addition** or the **Watch** node, and choose: **Add To Group**.

34. Run the graph (see Figure 54).

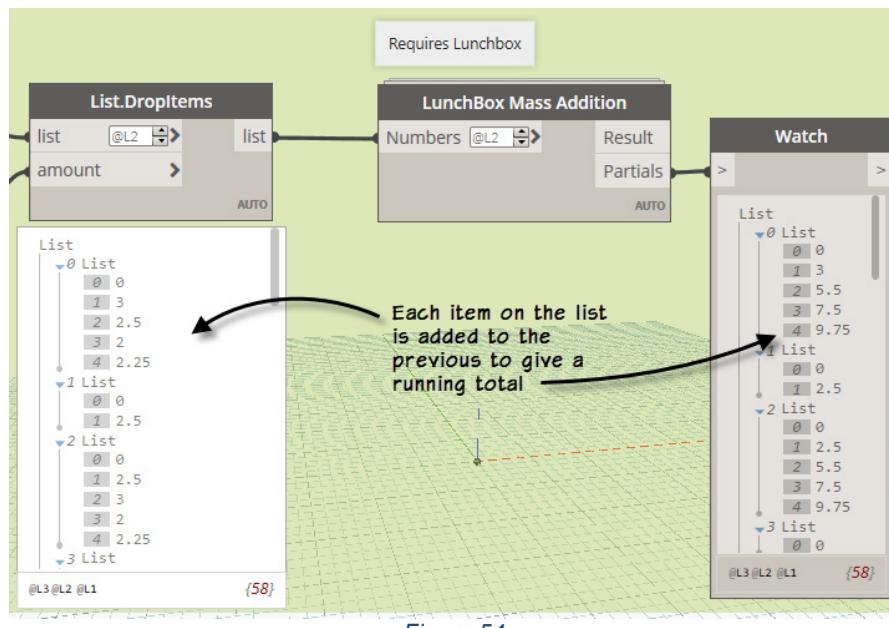


Figure 54

As you can see, we now have a running sum of the values on the list and these numbers can be used for the X coordinates of each family.

**SAMPLE FILE:** You can open a file completed to this point named: **04\_Casework Warehouse\_D.dyn**

#### CALCULATE THE Y VALUES

The Y values are a little easier. We will use the same Y value for all types within each family. The first family will be placed at zero and then we'll use a simple slider to add the spacing of the families placed above the first.

35. Add the following nodes to your canvas:

Library Location	Node
Input > Basic	Number Slider
Input > Basic	Number (add 2)
Math > Operators	-
Math > Operators	*
List > Generate	Range
List > Inspect	Count

36. Position, wire and group the nodes as indicated in Figure 55.

#### 4. Determine Y Spacings

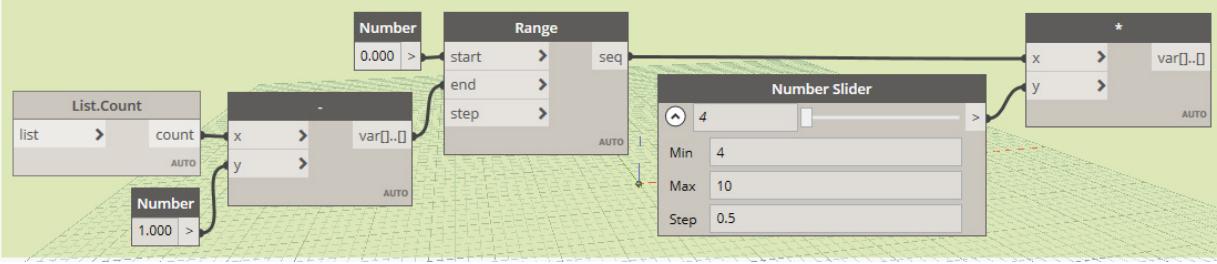


Figure 55

The **Range** node will give us a list of numbers<sup>20</sup>. Since we want to use this list to create our Y coordinates, we are starting it at zero with a **Number** input. The end of the range is a number equal to a count of our list of families minus one. The reason we need to subtract one is because we are starting at zero. If we don't subtract one, there will be one too many. We do not need to feed anything into the **step** port as it already defaults to: 1 which is what we need. To turn the values from the Range into coordinates, we are simply multiplying them by a constant value. The slider makes this value interactive. By setting the Min value of the slider to something like 4 we ensure that the smallest spacing between elements in the y direction will be larger than our biggest casework element.

37. Go back to the very beginning of the graph and wire the **groups** output from the **List.GroupByKey** node into the **List.Count**.

38. Run the graph (see Figure 56).

<sup>20</sup> Challenge: Try using a **Code Block** here instead. What format would the Design Script need to take to replace the Range and its input nodes?

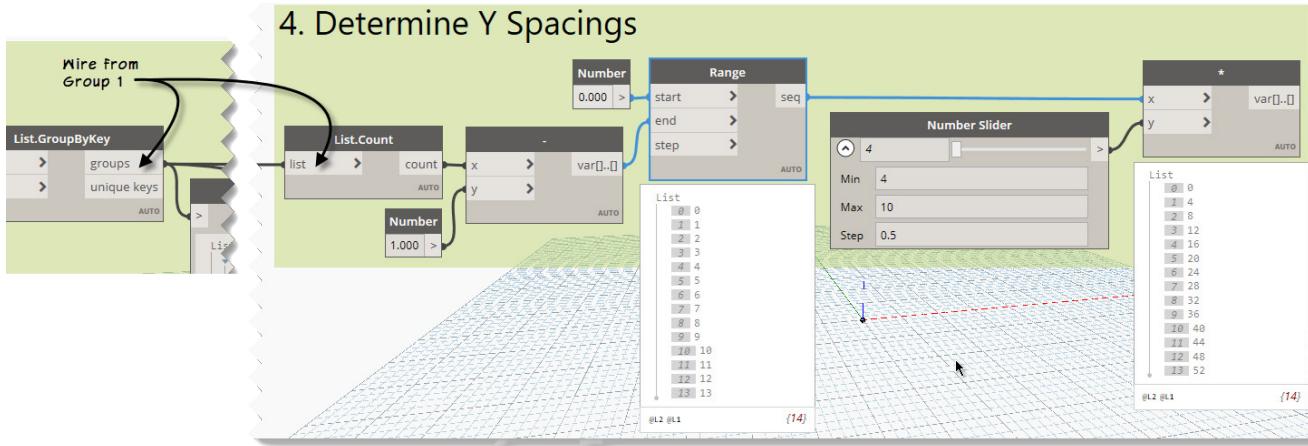


Figure 56

## CREATING POINTS AND PLACING FAMILY INSTANCES

We now have everything we need. It is time to wire the final pieces and see the results.

39. Locate group 5 in your graph created at the start and drag it to the right of group 4.

Let's consider what we have. Back in group 1 we have a structured list of families and types. Group 3 creates a structured list of X coordinate values. And group 4 creates a flat list of Y coordinates. But considering that level 2 on the two structured lists has the same quantity of items as the list from group 4, we are in good shape. So, let's feed these values into the **Point.ByCoordinates** node.

40. Feed the Partials output from the **Lunchbox Mass Addition** node in group 3 into the x input of the **Point.ByCoordinates** node.
41. Feed the output of the \* node in group 4 into the y input of the **Point.ByCoordinates** node.
42. In group 5, right-click the **FamilyInstance.ByPoint** node and choose: **Freeze**, then run the graph (see Figure 57).

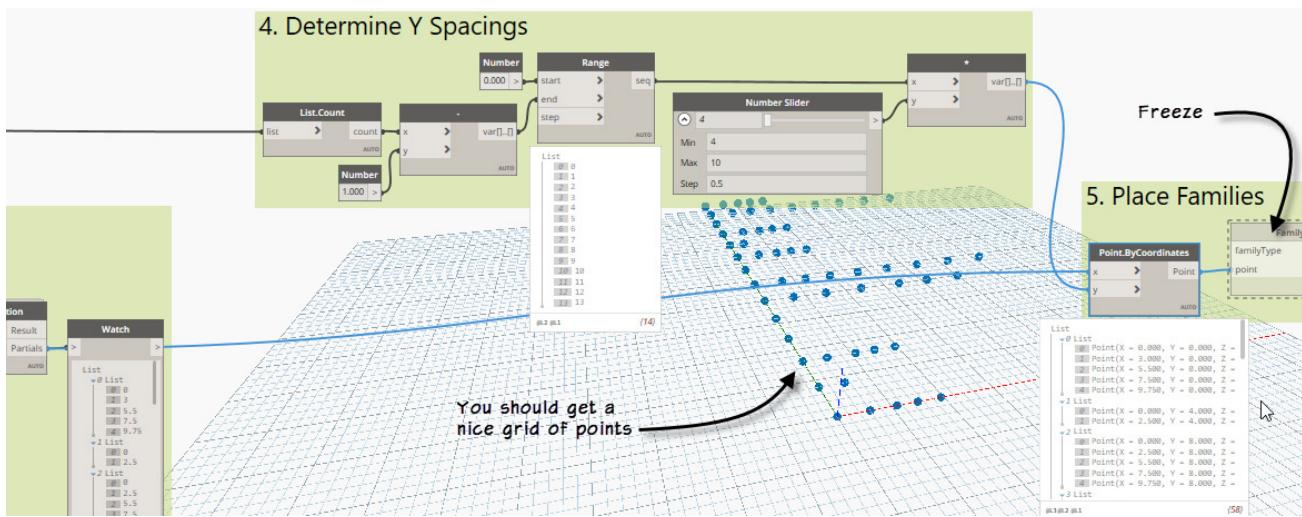


Figure 57

As we noted above, by freezing nodes downstream, you can make sure that the current part of the graph is working correctly before continuing. As you can see in the Dynamo preview window, we have a nice grid of points that adheres to the logic we were striving for. So, we're almost there! Now let's unfreeze and place the families.

43. Go back to the very beginning of the graph again and wire the **groups** output from the **List.GroupByKey** node into the **FamilyInstance.ByPoint** node.
44. Right-click the **FamilyInstance.ByPoint** node and choose: **Freeze** again to unfreeze it.
45. Position the Dynamo window such that you can see the Revit window in the background and then run the graph (see Figure 58).

Once again, the **Element.Geometry** node is optional, but it is not recommended in this case. The reason is that it will add *significantly* to the amount of time it takes to process this graph. So leave it out and use the Revit window to see the results.

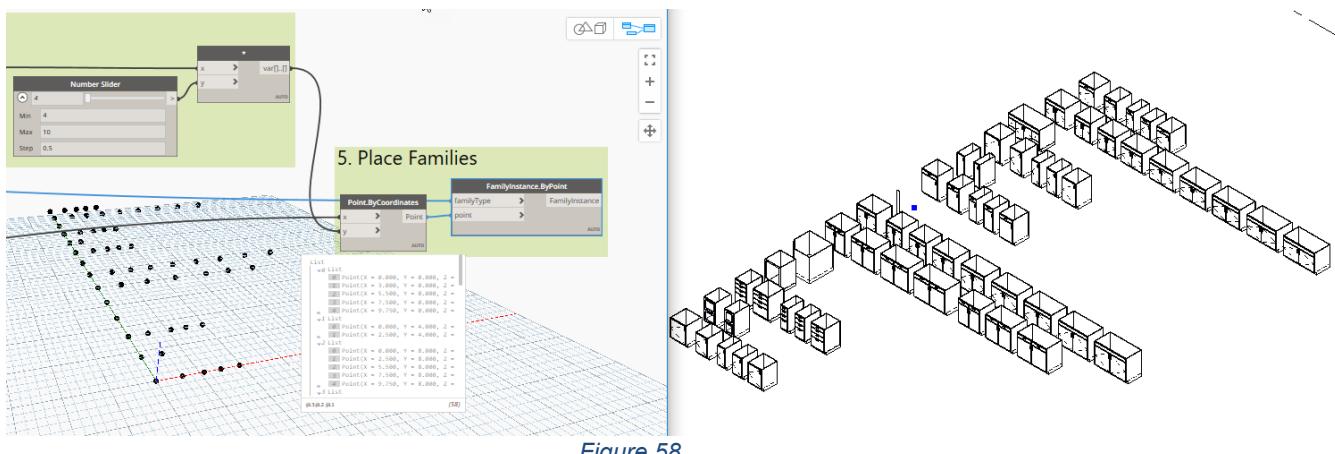


Figure 58

46. Optional: Adjust the sliders if you wish and run the graph again.

Also, if you load more casework families and run it again, they will be added to the warehouse. Just be sure to add non-hosted families. This graph does not deal with hosted families like wall-based or face-based families.

**SAMPLE FILE:** You can open a file completed to this point named: **04\_Casework Warehouse\_E.dyn**

So now you have three different approaches to placing Revit families. If you plan to use Dynamo with Revit, there is a good chance that you will need to place elements at some point, so techniques from one or more of those examples should prove helpful. But these examples are by no means exhaustive. There are many other ways you can place Revit elements and interact with those that are already in your model.

## SUMMARY OF KEY CONCEPTS

Here is a summary of the most important concepts covered in the previous exercise:

- Your strategy and approach will vary depending on whether your graph manipulates system families or loadable families. They use different nodes and procedures, so your graphs will naturally need to reflect this.
- Consider the hosting behavior as well. Placing hosted families requires different nodes and approaches than non-hosted ones.
- When out-of-the-box nodes do not give you the functionality you require, explore the Package Manager for solutions. Packages are collections of custom nodes created by the Dynamo community.
- Packages are a great way to extend the functionality of Dynamo, but Dynamo (even with packages) cannot create new Revit functionality. It can only do things that are exposed by the Revit API.
- “ByKey” nodes allow you to use one list to group and filter another list. The key list is used as the criteria for the sorting and grouping of the resultant list.
- Think carefully about how you want your data structured. Matching carefully structured nested lists can give a complex and structured layout compared with a simple flat list.
- You can use sliders to make values more interactive.
- Using List at Level allows you to control at which level of a nested list an operation should be applied. This can have a dramatic impact on the output produced, so choose carefully.
- List at level is also quite interactive, making it easy to experiment with possible solutions.
- If the list structure for your points, matches the list structure of your families, you should get satisfactory results.

## IMPORTING AND EXPORTING DATA (USING EXCEL)

A large part of BIM is managing data. As such, the focus of this next example is on data import, export and processing. This example uses MEP geometry and data; specifically, MEP Spaces. But in reality, the geometry will be inconsequential. The same strategy outlined here can be used for nearly any geometry, discipline or procedure that involves exporting and importing data.

In this example we will import a list of space parameters from Excel to Dynamo. We will use this list to get the parameters values from the spaces in the Revit model, and then export the values to back to a separate Excel file (it could also be the same Excel file, but we'll use two here). This workflow demonstrates the ease with which you can use Dynamo to export several parameters to Excel for analysis and viewing outside of Revit.

Once you have data from Revit in Excel, you can manipulate that data in any way you like. Run calculations, reformat it, perform analysis, etc. The modified data can then be brought back into Revit using another Dynamo graph. An exercise to do so follows this one.

## INPUT, PROCESS, OUTPUT

With most Dynamo graphs, we start with some data. This is the input for the graph. We then process this data in some way, and then we output the results to achieve some desired aim. In this

exercise, we will look at using Excel for the input, output and the processing. We'll follow an MEP workflow here, but the procedure will work for any situation where you want to export data from Revit to an Excel file and situations where you want to input data from Excel back into your model.

Close Dynamo and any existing Revit files, you can leave Revit running.

1. In Revit, open the file named: **05\_SpaceParameters\_Start.rvt**.
2. Launch Dynamo and create a new graph.
3. Save the graph as: **SpaceParameters\_Get**.
4. Add the following nodes to your canvas:

Library Location	Node
<b>ImportExport &gt; File System</b>	File Path
<b>ImportExport &gt; File System</b>	File.FromPath
<b>ImportExport &gt; Data</b>	ImportExcel
<b>Script &gt; Editor (or double-click in the canvas)</b>	Code Block

5. Wire the nodes as indicated in Figure 59.

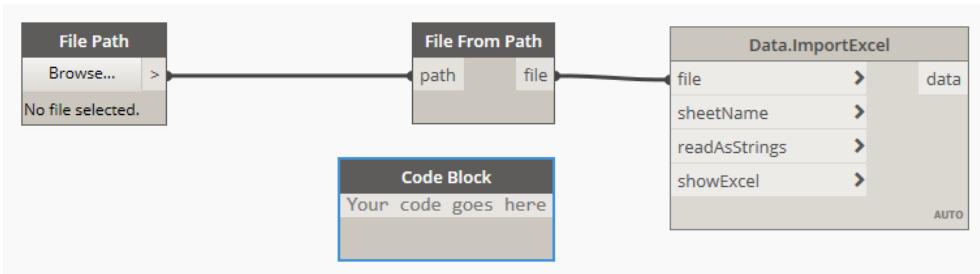


Figure 59

### READING DATA FROM EXCEL

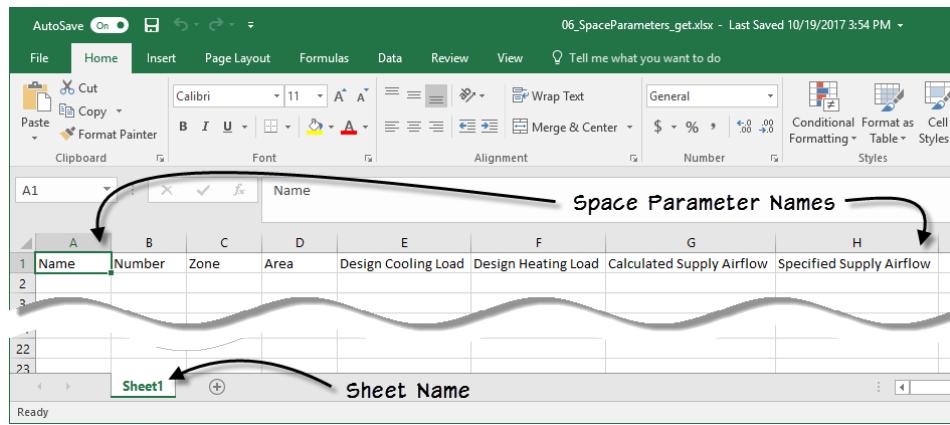
The **Data.ImportExcel** node has four inputs: **file**, **sheetName**, **readAsStrings** and **showExcel**. The **file** input is the excel file that you read. Excel files can have multiple worksheets, so the **sheetName** path tells it which sheet to read. The **readAsStrings** and **showExcel** inputs are Boolean (true or false) switches. The default setting for **readAsStrings** is false which means it will read the data into Excel as it is formatted in Dynamo. If you set it to true, it will force all the data coming in as text (even if it is numbers for example). The **showExcel** input defaults to true, meaning that Excel will load in a visible window. If you make this false, Excel will run silently in the background and the Excel interface will not display.

Let's start with the **file** input. It takes two nodes to satisfy this one. The **File Path** node has a browse button which lets you locate an Excel file. This is the location of the file. You may recall that data types are very important in Dynamo. Files are a unique data type just like text or numbers. The **File.FromPath** node creates a file object from the path. This puts it in the correct

format for the **Data.ImportExcel** node to understand. If you try to feed the **File Path** output in directly, it will fail. Next, we need the **sheetName**. Let's open the Excel file to see what this is.

6. On the File Path node, click the Browse button and point to the *05\_SpaceParameters\_get.xlsx* file and then click Open.
7. Outside of Dynamo, launch Excel. Open the file named: *05\_SpaceParameters\_get.xlsx*.

There is a single sheet called: **Sheet1** with a single row of data (see Figure 60).



Name	Number	Zone	Area	Design Cooling Load	Design Heating Load	Calculated Supply Airflow	Specified Supply Airflow
1 Name							
2							

Sheet Name

Figure 60

Each of the entries in this row of data is the name of a parameter from the MEP space elements in Revit. If you select any space, and look at the Properties palette, you can locate each of them on the list (see Figure 61). We will use this list from Excel to tell Dynamo which properties we want to export from the spaces in Revit.

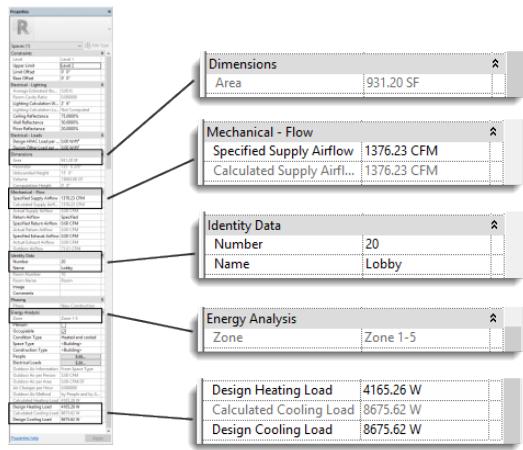


Figure 61

8. Now that we know the **sheetName**, and the contents of the file, close Excel.

The **sheetName** input expects text (or string) data. In previous exercises, we used a **String** node for this. You can do the same here if you wish. But many Dynamo users will use a **Code Block** instead. Code Blocks can be created by simply double-clicking on the canvas and they

can take any Design Script commands and inputs. You just have to know how to format it. If you want to use a **Code Block** as an input (as we do here) and you want it to be formatted as string data, place the value in quotes.

9. Click in the **Code Block**, where it says: "Your code goes here," type: "**Sheet1**" (no space before the 1) and then click away from the **Code Block** in the empty canvas.

Don't press ENTER. Doing so will stay in the **Code Block** and create a return to another line. When you click away from it, it completes the input and a semicolon will appear at the end of the input. An output port will also appear allowing this **Code Block** to be plugged into another node.

10. Wire this into the **sheetName** port.

If the graph is set to Automatic, it will immediately launch Excel and open the file. This is how you know it is wired correctly. If Excel does not launch, or if it launches but does not load the spreadsheet, then check your work for mistakes. (Note also that other two inputs have defaults already as noted above, so you do not need to wire anything in if you are accepting those defaults<sup>21</sup>).

11. Expand the Preview Bubble and pin it open.

12. Select all the nodes and group them. Name the group: **1. Import from Excel** (see Figure 62).

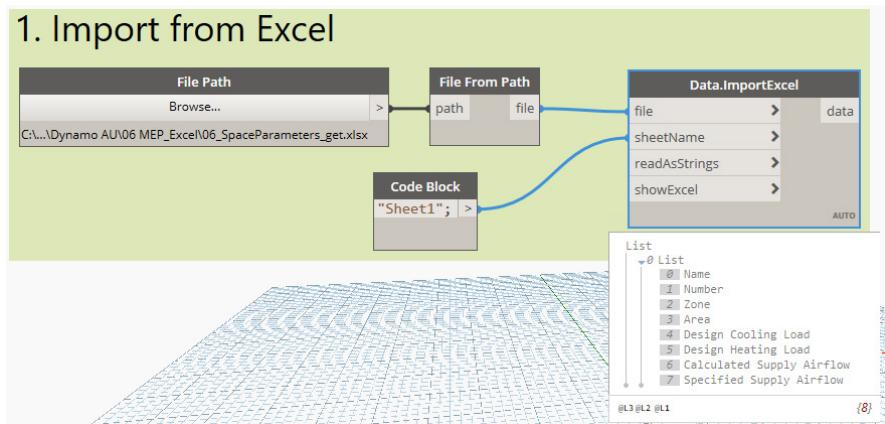


Figure 62

**SAMPLE FILE:** You can open a file completed to this point named: **05\_SpaceParameters\_get\_A.dyn**

#### EXCEL IMPORT LIST STRUCTURE

Study the list structure coming out of the Excel file. We know that there was a single row of data, but this produced a three-level list in Dynamo. Excel is structured in rows and columns. The worksheet itself (Sheet1 in this case) will be the @L3 list. Each row in the file will appear in the @L2 level. The @L1 list is the data contained in each row (the columns of data). So, it goes: **Worksheet (@L3), Row (@L2) then Column (@L1)**. Understanding this list structure will be important in a moment

<sup>21</sup> If you want to force the data to be read into Dynamo as string (text only) data, then add a **Boolean** input node and wire it to the **readAsStrings** port. Toggle the **Boolean** to True. Another **Boolean** can be used to make Excel run silently without displaying the interface.

when we try to use it in the graph. Remember, a large part of success in Dynamo is effective list management. But before we use this data, we need the spaces from Revit.

13. Add the following nodes to your canvas:

Library Location	Node
Revit > Selection	Categories
Revit > Selection	All Elements of Category
Revit > Elements > Element	GetParameterValueByName
List > Modify	Flatten

14. From the drop-down on the **Categories** node, choose: **Spaces**.

15. Wire the nodes as indicated in Figure 63.

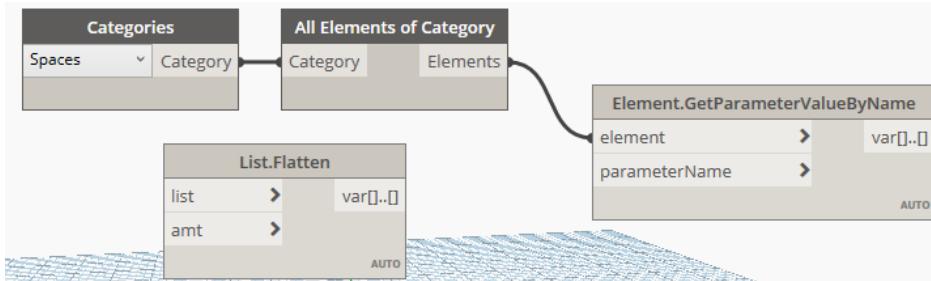


Figure 63

If you run the graph now, it will grab all spaces from the Revit file (42 total). You can see this with the preview bubble on the **All Element of Category** node. If you look at the list structure of this output, you see that it is a two-level list. However, we have a three-level list from Excel that we want to use for the **parameterName** input on the **Element.GetParameterValueByName** node. The simplest way to deal with this is to flatten the Excel list. When you flatten a list, it removes the unwanted structure.

16. Wire the **data** output from **Data.ImportExcel** to the **list** input of the **Flatten** node.

17. Run the graph and expand the Preview Bubble on the **Flatten** node (see Figure 64).

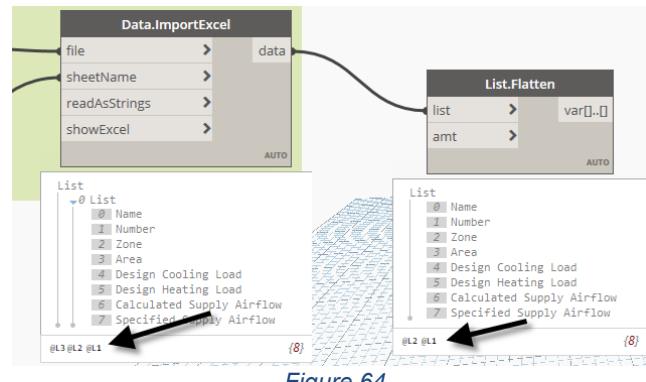


Figure 64

Notice that the extra level of hierarchy in the list structure is now removed<sup>22</sup>.

18. Wire the output from ***Flatten*** into the **parameterName** input of the ***Element.GetParameterValueByName*** node.

If you run the graph, you will get a list of 8 items as a result. You probably expected more. This is now a lacing issue. We have 42 spaces and a single list of 8 parameters. The ***Element.GetParameterValueByName*** node wants a Revit element and a parameter name. We are feeding it several elements and several parameters. We want all spaces to report all eight parameters. In other words, we want to match all items on list A with all items on list B: Cross Product lacing!

19. Right-click the ***Element.GetParameterValueByName*** and choose: **Lacing > Cross Product**.

20. Run the graph and study the list that results in the Preview Bubble (see Figure 65).

We end up with a three-level list. There are 336 total items. But if you scroll the Preview Bubble to the bottom, you will see that the level 2 list contains 42 items (remember the numbering starts at 0, so the last item is 41). This means that the level 2 list is showing the 42 spaces. The level 1 list contains 8 items per list. These are the eight parameter values for each space. Perfect!

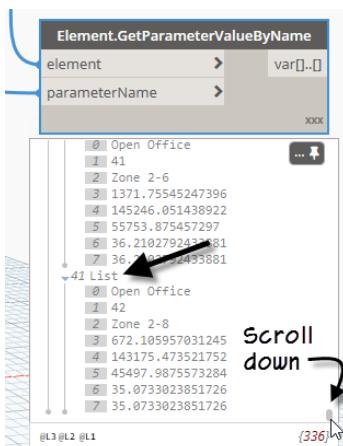


Figure 65

21. Group these new nodes and name it: **2. Get Spaces and Parameters**.

**SAMPLE FILE:** You can open a file completed to this point named: **05\_SpaceParameters\_get\_B.dyn**

#### EXPORT DATA TO EXCEL

Now that we have the property values from the Revit spaces, we can export them back to Excel for further processing. Let's start with the node we need to do this and see what inputs it requires.

22. In the Library, from the *ImportExport > Data* branch, add the ***ExportExcel*** node.

<sup>22</sup> The ***Flatten*** node also has an **amt** input. You can feed a ***Number*** node into this to control how many levels of list it flattens. In this case, the default gave us what we needed with no need for further input.

This node has six inputs. Pay close attention to data types here. The first input is `filePath` (not `file` like the import node). So, we need another **File Path** node for this, but do not need to convert it to a file this time. The next four inputs are: `sheetName` (same as before, string input for the sheet within Excel), `startRow` and `startCol` (these are both numerical inputs) and `data` (this is the data that we exported and that we want to write to the file). The final input: `overWrite` is another Boolean. If you toggle it true, it will replace all data on the Excel sheet as it writes it. False will not. In this case, we'll accept the default (false) since we are creating a new Excel file, so it will be empty when we start. If you plan to run this graph a lot, you might want to add the Boolean toggle switch to control this behavior.

23. Add the following nodes to your canvas:

Library Location	Node
ImportExport > File System	File Path
Script > Editor (or double-click in the canvas)	Code Block (add 2)
List > Modify	List.AddItemToFront

24. In the first **Code Block**, type: “Sheet1” (include the quotes) and then click in the canvas. Wire this into the `sheetName` port.

25. In the second **Code Block**, type: 0 (no quotes) and then click in the canvas. Wire this into both the `startRow` and the `startCol` ports.<sup>23</sup>

26. Click the Browse button on the File Path node and in the File name field, type:  
**05\_SpaceValues\_export.xlsx** and then click Open.

27. Wire this into the `filePath` input (see Figure 66).

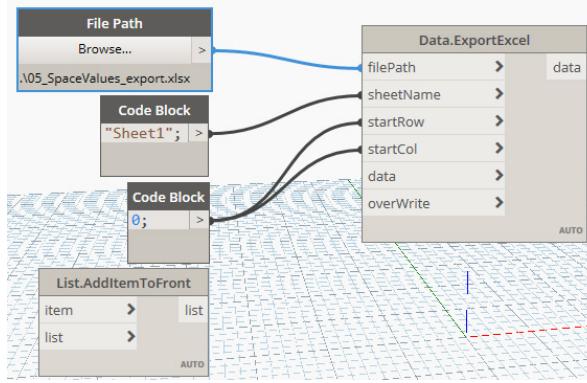


Figure 66

**SAMPLE FILE:** You can open a file completed to this point named: **05\_SpaceParameters\_get\_C.dyn**

<sup>23</sup> As noted, to use a Code Block for text, put it in quotes. For numbers, do not include the quotes. Also notice that we are using the same input for multiple input ports here.

This leaves the **data** port. We are creating a new Excel file, and to make it more legible for its recipients, we should add headers to each column. The names of the headers can be the names of the properties that we exported from the spaces. The **List.AddItemToFront** node will allow us to take our original list of properties and add it to the top of the list as the headers.

28. Wire the output from the **Flatten** node to the **item** input on **List.AddItemToFront**.
29. Wire the **Element.GetParameterValueByName** output into the **list** input.
30. Expand the Preview Bubble and run the graph (see Figure 67).

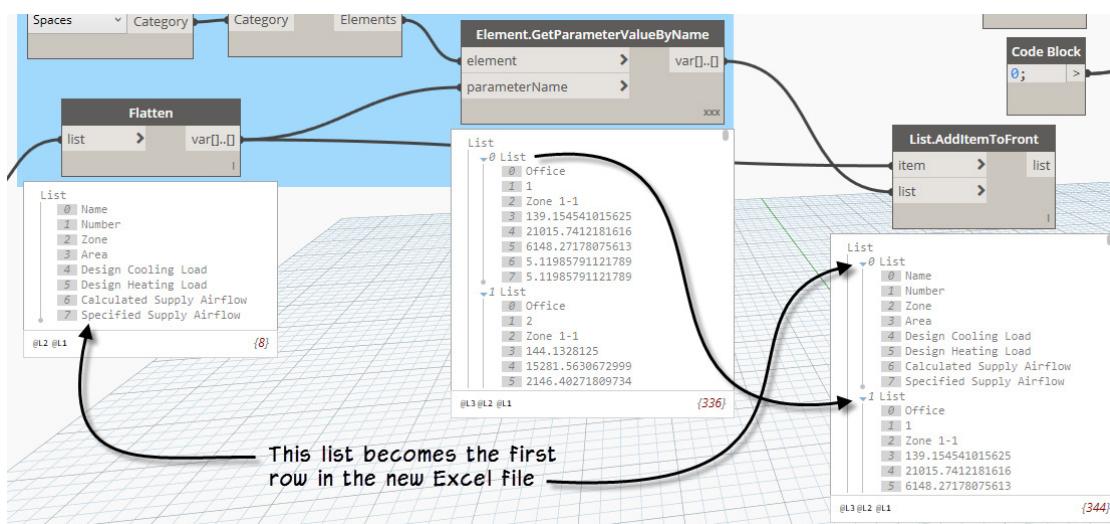


Figure 67

Remember that each list at level 2 will become a row in Excel. The **List.AddItemToFront** shows the final list of data with our flattened list of parameter names as the first row.

31. Wire the **list** output of **List.AddItemToFront** into the **data** input of **Data.ExportExcel** node.
32. Group the remaining nodes and name it: **3. Export to Excel**.
33. Run the graph.

**SAMPLE FILE:** You can open a file completed to this point named: **05\_SpaceParameters\_get\_D.dyn**  
**05\_SpaceValues\_export\_A.xlsx**

If everything runs correctly, then the new Excel file will be created, and it will open immediately in Excel. You can switch to the file now and study the results.

## IMPORT MODIFIED EXCEL DATA BACK INTO REVIT

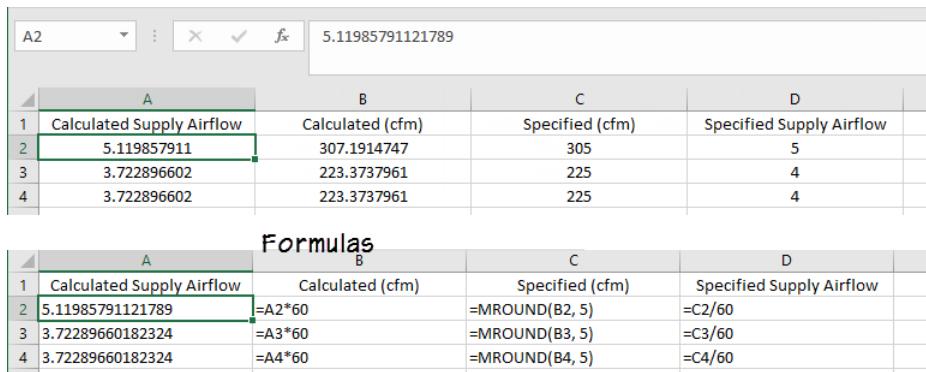
Once you have your exported data in Excel, you can manipulate those parameters in Excel directly or through any other third-party program that can read the Excel data. Regardless of how you choose to modify the Excel values, you can import their modified values back to Revit to set parameters on the spaces. We are only going to set one parameter (airflow values) here to keep the example simple. For setting multiple parameters the overall concept still applies.

In Revit, when heating and cooling loads are calculated for spaces, the airflow values are not nice, round numbers<sup>24</sup>. Airflow values that a designer would actually use will more than likely be rounded to the nearest 5. That function can be performed in Excel and then imported back into Dynamo to set parameters in Revit.

There is one other important issue. Dynamo is technically unitless but despite this, it does not always work well with MEP units. It only refers to the “common units.” Therefore, a unit conversion is needed. So, in this example, cubic feet per minute (CFM), which is the standard airflow unit, is read in Dynamo as cubic feet per second. This is because seconds is the common or base unit of measurement for time in Revit. (The same type of issue arises for units like Watts, Volts, etc.). This conversion can be performed in Excel.

## USE EXCEL TO EDIT REVIT

Another powerful way to use Excel with Dynamo is to use it process data and then write the values back to the Revit model. While you can certainly open the exported Excel file and process the data in any way you wish, we have provided a file with the processed Supply Airflow values to simplify things. If you open that file, you will see that the values from the Dynamo export in the previous exercise are in column A. A simple formula in column B compensates for the units issue noted above by multiplying by 60. This converts minutes to seconds. Column C rounds this value off to the nearest 5. Finally, these values are divided by 60 to convert back to the correct units again in Column D (see Figure 68). In the figure, the values are shown in the top portion. The bottom shows the formulas (In Excel, on the Formulas tab, click Show Formulas).



A	B	C	D
Calculated Supply Airflow	Calculated (cfm)	Specified (cfm)	Specified Supply Airflow
5.11985791121789	307.1914747	305	5
3.722896602	223.3737961	225	4
3.722896602	223.3737961	225	4

A	B	C	D
Calculated Supply Airflow	Calculated (cfm)	Specified (cfm)	Specified Supply Airflow
5.11985791121789	=A2*60	=MROUND(B2, 5)	=C2/60
3.72289660182324	=A3*60	=MROUND(B3, 5)	=C3/60
3.72289660182324	=A4*60	=MROUND(B4, 5)	=C4/60

Figure 68

Close Dynamo, Excel and any existing Revit files, you can leave Revit running.

1. In Revit, open the file named: **06\_SpaceParameters\_Start.rvt**.
2. Open the *Space Schedule* view.

<sup>24</sup> Heating and Cooling loads can be calculated directly in Revit or using third-party applications. In both cases, it is likely that data will end up in Excel for some part of the process. This workflow can be adapted to those situations easily. Any Excel file can be used. It does not require that the Excel file was created by Dynamo or Revit. We are using an MEP example here, but Excel can be used to get and set any parameter on any Revit element. This includes all disciplines.

3. If desired, open the **06\_SpaceValues\_set.xlsx** file (shown in Figure 68) in Excel and familiarize yourself with its contents. Close the file when done exploring.
4. From within Revit, launch Dynamo and open the starter graph named:  
**06\_SpaceParameters\_set\_!Start.dyn.**

This graph contains two groups of nodes. Group 1 contains the same nodes we saw in the previous exercise that allow us to import data from Excel. The **File Path** node points to the Excel file noted here.

5. Run the graph.

Excel will open, and the file will display. We now want to take the data from column D and write that back to the spaces in Revit. In the starter graph, in group 3, we already have nodes selecting all the spaces and an **Element.SetParameterByName** node. This node is the counterpart to the **Element.GetParameterValueByName** node. One retrieves parameter values (Get) the other writes parameter values (Set).

So, we need to satisfy two inputs on the Set node: **parameterName** and **value**. The parameter name is in the top row of column D of the Excel file: **Specified Supply Airflow**.

6. Add a **Code Block** (or **String** node) to group 3<sup>25</sup> and input the value: “Specified Supply Airflow” (quotes are not required if you use a **String**).
7. Wire this new node to the **parameterName** input.

The values require a bit of processing. Look at the structure of the data coming out of the **Data.ImportExcel** node. We have a three-level list again with 42 lists (one for each space) at level 2 each containing 4 items. Specified Supply Airflow is the fourth item on each sub list (index 3). So we need to grab just these values from each list.

8. From the List > Inspect branch of the library, add a **GetItemAtIndex** node.
9. Double-click in the canvas to create a **Code Block**. In the **Code Block**, type: 3 and then click away from it.
10. Feed the output of this Code Block to the **index** input on the **GetItemAtIndex** node.
11. Feed the **data** output from the **Data.ImportExcel** to the **list** input of the **GetItemAtIndex** node.
12. Run the graph and expand the preview bubble on the **GetItemAtIndex** node (see the left side of Figure 69).

<sup>25</sup> To add a node to an existing group, select both the node and the group (using the SHIFT key), and then right-click on the node and choose: **Add To Group**.

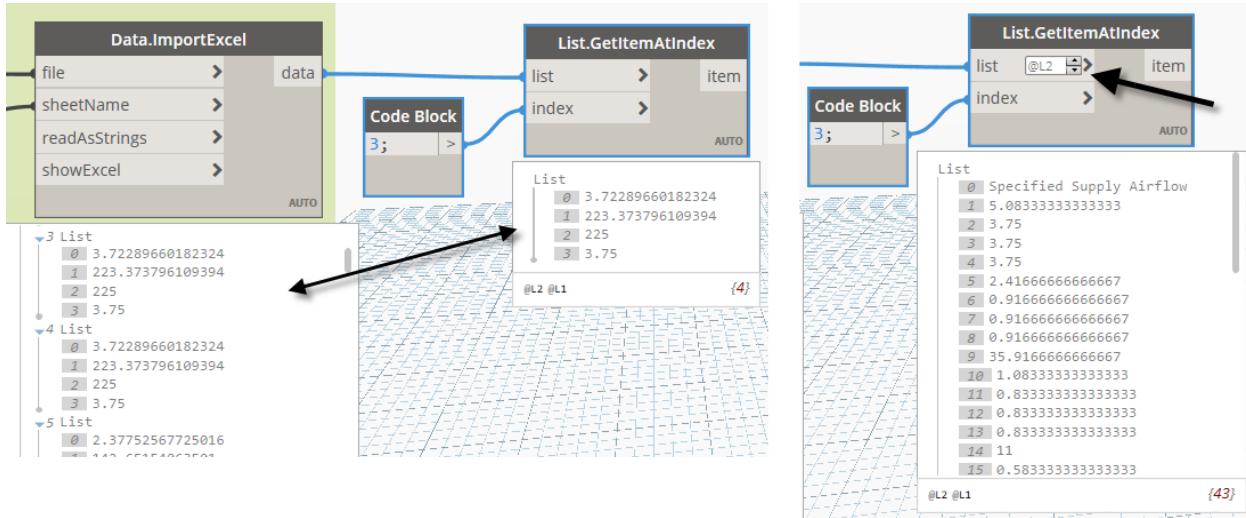


Figure 69

Not the result we need. Remember, the fourth item on the list is at index 3, however, we don't want index 3 on the L3 list (the fourth sub list), we want index 3 on *all* sub lists. So, we need to use list at levels again.

13. Next to the list input, click the list at levels indicator and check the Use Levels checkbox. Leave it set to the default of @L2 and run the graph (this is index 3 from each L2 list) (see the right side of Figure 69).

We now have a list containing our data but notice that it contains one additional item we don't need. The name of the column header is included as the first item giving us 43 total instead of the desired 42. If we were to feed this in directly, not only would we have a mismatch in quantity, but the wrong values would be written to every space element! So, we need to remove this unwanted item from the start of the list.

There are a few ways you can approach this. You might want to spend some time exploring the sub branches of the List branch in the library for possibilities. There is **DropItems**, **RemoveItemAtIndex** and **Slice** which all could be used. But the simplest option is the **RestOfItems** node. This one does exactly what we need. It removes the first item from a list (see Figure 70).

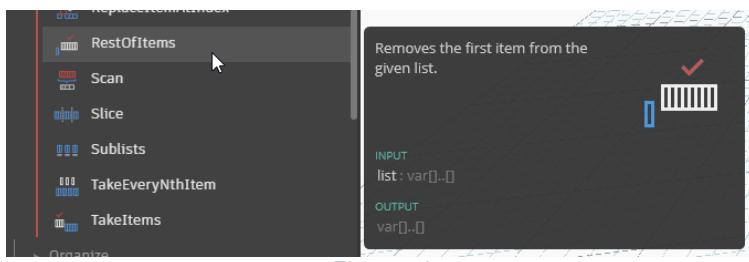


Figure 70

14. Add a **List.RestOfItems** node, wire it to the item output of the **List.GetItemAtIndex**.

15. Group these nodes as: **2. Extract data required** and then run the graph (see Figure 71).

## 2. Extract data required

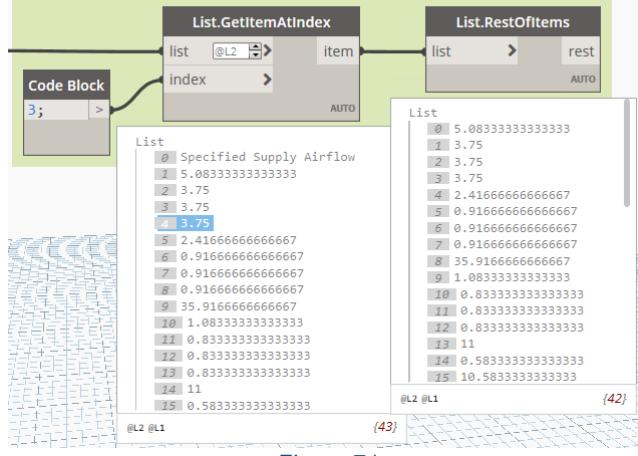


Figure 71

Compare the two lists on the preview bubbles. Notice that the output of the **List.RestOfItems** node does not include the header name. We're all set now. We just need to feed this to our Excel export node to finish up.

16. Position Dynamo so that you can see the Specified Supply Airflow column (column H) of the schedule in the Revit window behind Dynamo.
17. Wire the **rest** output from the **List.RestOfItems** node into the **value** input of the **Element.SetParameterByName** node.
18. Run the graph (see Figure 72).

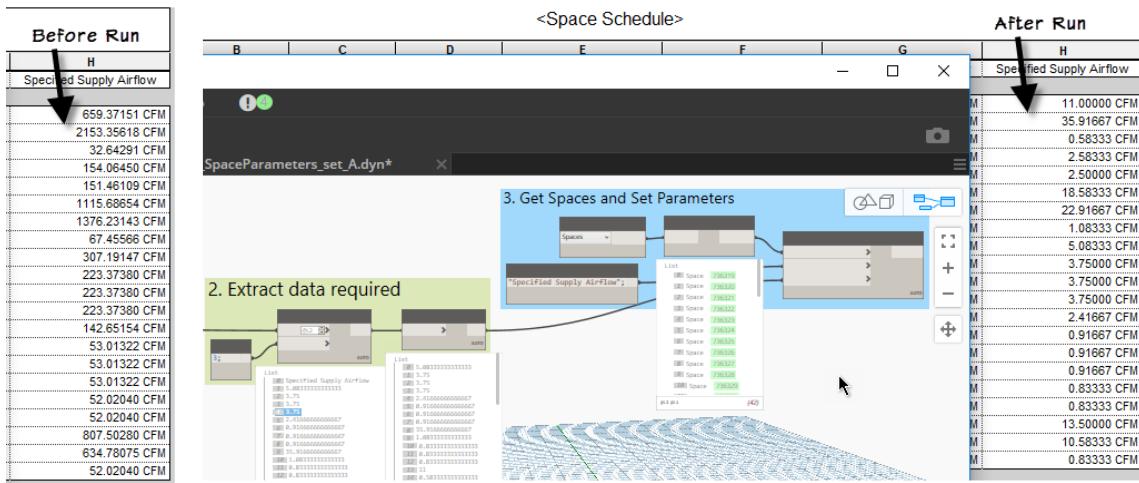


Figure 72

**SAMPLE FILE:** You can open a file completed to this point named: **06\_SpaceParameters\_set\_A.dyn**.

## SUMMARY OF KEY CONCEPTS

Here is a summary of the most important concepts covered in the previous two exercises:

- Dynamo can read data from Excel files. Data is imported into structured lists.
- Understanding imported data list structure is critical to being able to use the data successfully in your Dynamo graph.
- Remove unwanted list structure with flatten nodes.
- **Code Block** nodes can be used in place of **String** and **Number** input nodes. Put text in quotes to interpret as string data. Numbers should not be in quotes.
- Use the **Element.GetParameterValueByName** node to read data from elements in your Revit model.
- When processing data, choose your lacing options carefully to create and/or match the required list structure.
- Data from Dynamo can be exported out to Excel and create new files or overwrite existing ones.
- Manipulate data outside of Revit and Dynamo directly in Excel and then you can import modified values back to Revit.
- Use the **Element.SetValueByName** node to write data to elements in your Revit model.

## THE “I” IN BIM—PERFORMING CALCULATIONS ON MODEL DATA

In the previous example, we gathered the values of several Revit elements, manipulated some of those values and then updated the elements back in Revit. We used Excel as the conduit for these changes. Using Excel for such modifications is incredibly powerful, but it is not the only way to make such manipulations to your Revit model elements. You can also extract, process and write back data all within Dynamo. Which approach you take depends on the specifics of the task at hand and the goals of the team and the results they require. In this example, we will look at a simple graph that allows us to calculate occupancy values for rooms in a Revit project. We will extract a subset of the total rooms, process them and write the results of an occupancy calculation back to the same set of rooms – all within Dynamo.

In this example, we will not build the graph from scratch. The graph we will explore is provided with the datasets. Most of the nodes it uses we have already seen in other examples. So, to save time and effort here, we will simply open the provided files and walk through the logic. To facilitate a systematic approach to this, the last node in the graph has been frozen. This means we can run the graph, walk through it and then unfreeze when we are ready and run it again.

### SELECT ROOMS OF A GIVEN FUNCTION

In this exercise, we assume that you need to do some occupancy calculations for a project you are working on. The file we will use is a copy of the out-of-the-box advanced sample project that comes with Revit. This project is a small technical school building and it contains several “Instruction” rooms. The graph we will explore will look at all the rooms in the file, find those whose name is “Instruction” and then perform a calculation on each one to determine what the maximum occupancy for each room can be. It will then write these values to elements in the model.

Close Dynamo and any existing Revit files, you can leave Revit running.

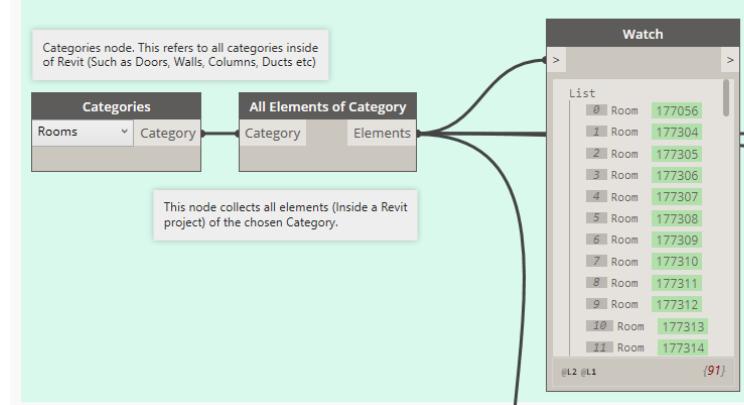
1. In Revit, open the file named: **07\_rac\_advanced\_sample\_project\_Start.rvt**. Open the **01 - Entry Level - Instruction Rooms** floor plan.

This view has been customized to show just the Instruction spaces shaded in color. There are similar views for level 2 and 3 as well. Otherwise, this file is unchanged from the out-of-the-box advanced sample project.

2. Launch Dynamo and open in manual execution mode, the file named: **07\_RoomOccupancies.dyn**.
3. Run the graph.

The groups in this graph are numbered. Zoom in on group 1. This group uses the **Categories** node set to Rooms and then feeds that to the **All Elements of Category** node to select all rooms in the model. The **Watch** node gives the complete list. It contains 91 rooms. If you wish, you can open the *Room Finish Schedule* view in Revit to verify this quantity (see Figure 73).

## 1. Select Rooms



## 1A. List all Parameters

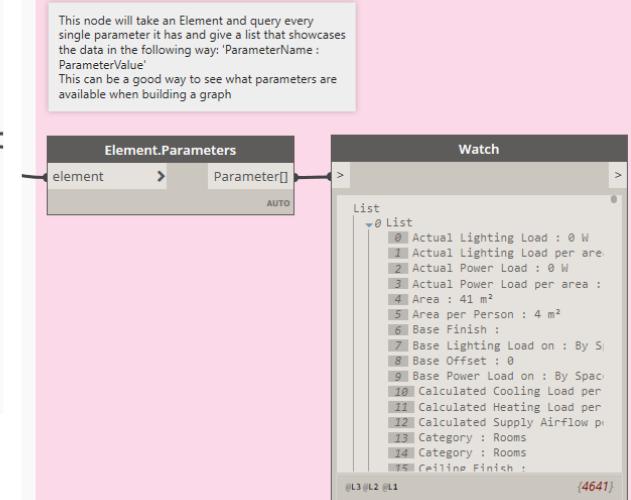


Figure 73

The next group (1A) is optional. It uses the **Element.Parameters** node to list all available parameters in the selection of rooms. To use nodes like Get and Set Parameter by name, you have to know the name of the parameters you are either getting or setting. This group gives you an easy way to do that. Even though you can simple select a room onscreen in Revit and look at the Properties palette, sometimes this does not show everything and sometimes the names can vary slightly. So, the **Element.Parameters** node is helpful in those situations.

In this case, we are going to calculate the occupancy for our classrooms. The easiest way to isolate classrooms from other rooms is by looking at their “Name” value. In this model, classrooms are named “Instruction”.

4. Zoom in on the left side of group 2.

The **Watch** node is optional in most cases. Here you will see that it continues to display 91 rooms, but this time it lists the names of those rooms. This is because we are feeding in the value “Name” into an **Element.GetParameterValueByName** node. Please note that the parameterName input is case sensitive, so “name” is not the same as “Name” (see Figure 74).

## 2. Filter Rooms by Name

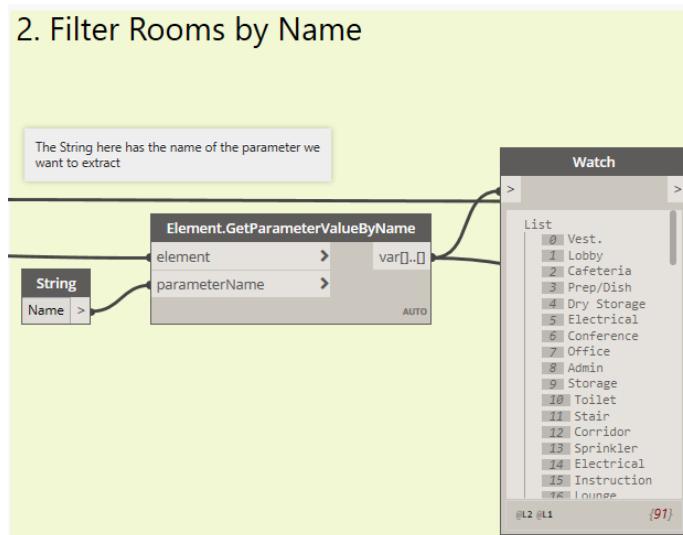


Figure 74

The next few nodes in group 2 look at the list and using an equality (==) node, determine if the name equals “Instruction” (also case-sensitive). This results in a list of True and False values. With this list of true and false values we can feed it into the mask input of a **FilterByBoolMask** like other examples we have seen earlier. You may also recall from above that when a node has more than one output, **Watch** nodes provide a handy way to view each output separately (see Figure 75).

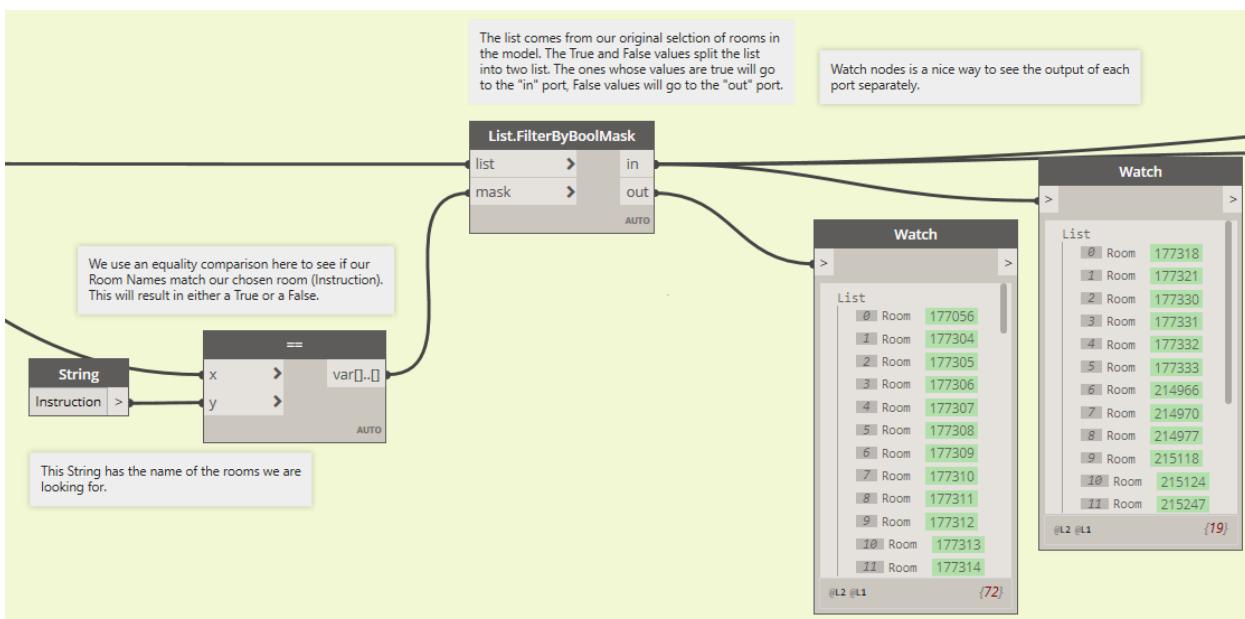


Figure 75

### 5. Zoom in on group 3 next.

Here we use another **Element.GetParameterValueByName** node, but this time, its input comes from the **FilterByBoolMask**. So, there are only 19 elements (those named “Instruction”) from the original 91.

Next comes a division (/) node. We are dividing by a value of 1.86 in this example. Occupancy codes naturally vary with municipality. For this example, we will use an occupancy factor of: 20SF [1.86M]. This project is in metric units, so we are using the value of: **1.86** in the Dynamo graph. The value used here is just for illustration purposes. You should of course be sure to consult all proper building codes and regulations in your own jurisdiction before arriving at the correct value to use in actual projects. The division yields a long decimal value. Occupancy values should be whole numbers (integers). The easiest way to achieve this is to round them down. The **Math.Floor** node achieves this (see Figure 76).

### 3. Calculate occupancy value

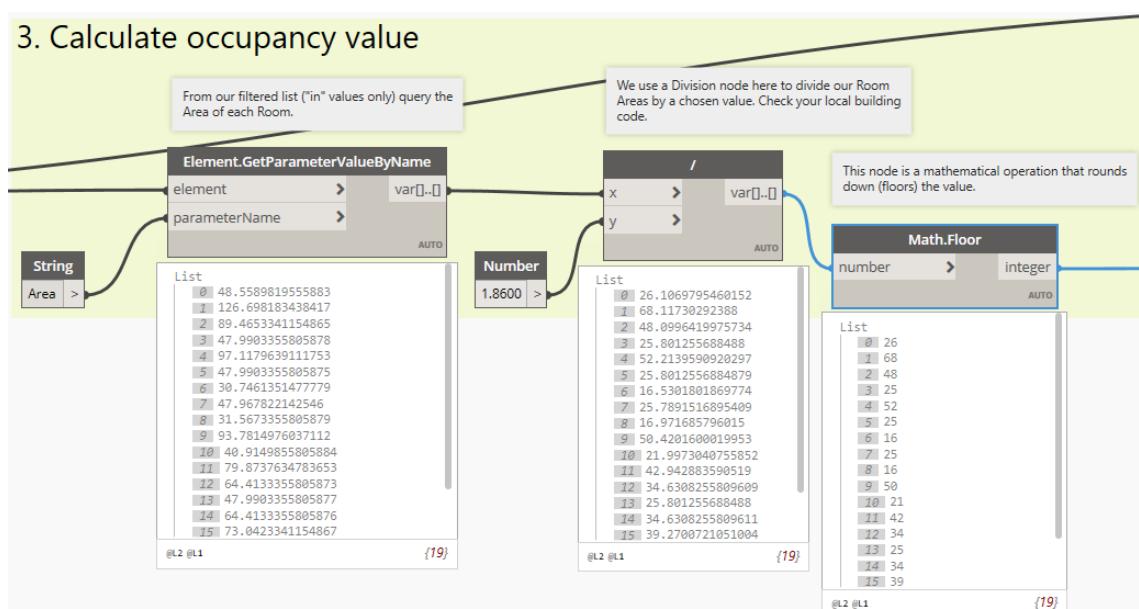


Figure 76

### 6. Move on to the final group, group 4.

Here we are using an **Element.SetParameterValueByName** to write the calculated occupancy values back to the Revit model. To do this, feed the same list of rooms from the **FilterByBoolMask** node into the **element** input. The **parameterName** we want to write is called “Occupancy” (case sensitive). If you try to feed the values directly from the **Math.Floor** node, it will fail. This is because of data types again. The Occupancy property of rooms is a text parameter. So, if you input integers, it will be unhappy. The **String from Object** node simply converts the data input to it into text.

### 7. Unfreeze the **Element.SetParameterValueByName** node.

### 8. Run the graph.

To check the results, select any Instruction room in the model and look at the Occupancy property on the Properties palette (see Figure 77). You can also open check the results of the calculation if you wish. Open the *01 - Entry Level - Furniture Layout* floor plan view and add a number of chairs equal to the occupancy value for the selected room. You can see an example in the figure. 48 chairs are shown which matches the selected occupancy value, so it checks out.

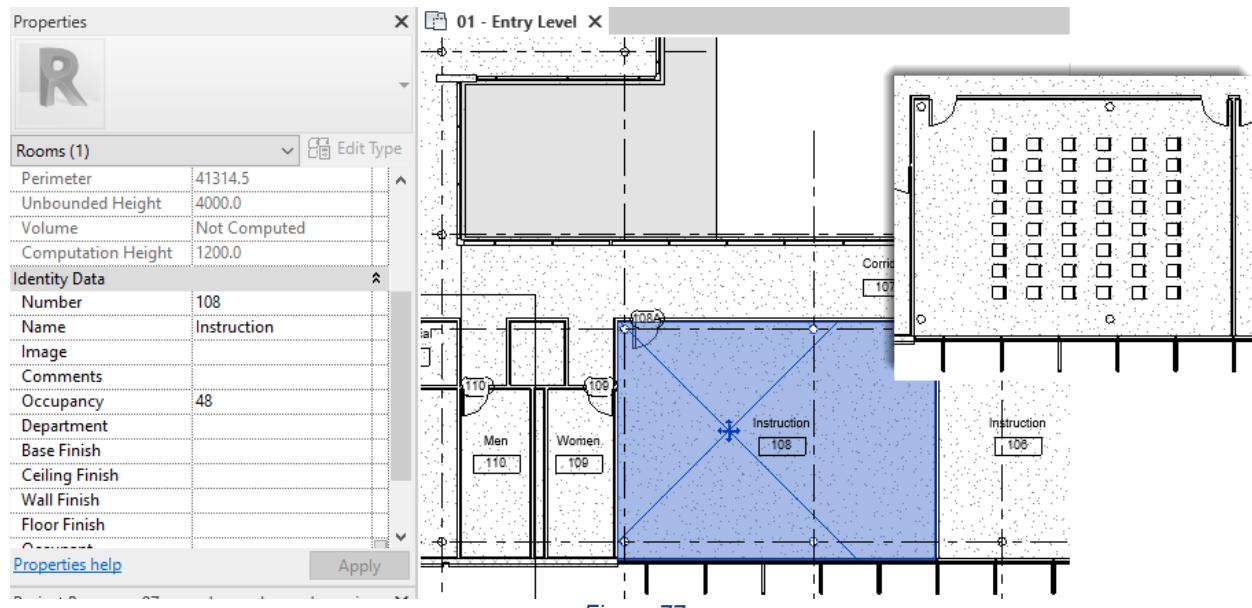


Figure 77

This example showcases Dynamo's ability extract and process values and then write them back to the model. You can use a graph like this to perform all sorts of similar calculations. This example only had 19 rooms to change, but you can imagine the value as the quantity of rooms and calculations increases.

## USING EXTERNAL CAD DATA TO BUILD REVIT GEOMETRY

This example has a structural/construction focus to it. We will explore taking data from outside of Revit and Dynamo and using it to create Revit elements. The example will be a structural truss, but the approach and concepts could be easily applied to nearly any discipline. So as with previous examples, we are more concerned with imparting the overall Dynamo concepts. You can adapt the workflow to your own purposes.

### FROM CAD IMPORT TO REVIT GEOMETRY

In this exercise, we assume that you received a shop drawing of a truss from a fabricator that is in AutoCAD. As you are likely aware, it is easy to import or link CAD files to Revit. You can import them into families and link or import them into projects. In this example, we will be working with a family, so our CAD file will be an import. Our goal is to extract the linework on a certain layer and then use this to create Dynamo and Revit geometry. Like the previous example, instead of building this one from scratch, we'll open an existing Revit family file with the CAD already imported and then open an existing Dynamo graph and walk through it piece by piece.

Close Dynamo and any existing Revit files, you can leave Revit running.

1. In Revit, open the file named: **08\_GM\_Truss.rfa**. Open the *Front* elevation.

This is a generic model family with a CAD import already placed in it. As you can see, the CAD import contains linework for a truss and is placed in the *Front* elevation view centered on the origin. The original CAD file (shown in Figure 78) was a shop drawing file with a handful of layers.

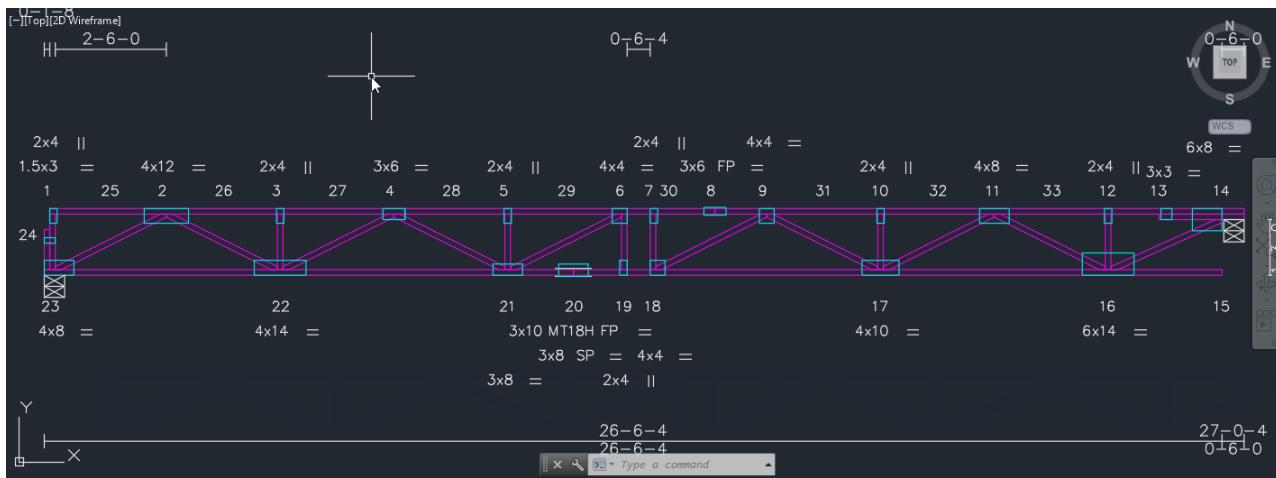


Figure 78

Comparing the figure to what you see onscreen in Revit, you can see that we have removed the text and gusset plates. This leaves us with just the geometry on the **members** layer (magenta in color). We want to extract this linework find the centerline between each pair of parallel lines and then create Revit members from those centerlines.

## ASSUMPTIONS

There are many approaches we can take to this task. And as you can infer from the fact that the imported geometry includes only the layer we want, you can easily do some of this work outside of Dynamo directly in Revit or back in AutoCAD. It really is up to you to design a workflow that is most efficient for your goals and team. In this case, we decided that isolating the layer was easy enough to do during import into Revit and that it would help us remove some complexity from the Dynamo workflow. On the other hand, even though the Dynamo graph we will open is still a bit complex, we decided it was more beneficial to find the centerlines in Dynamo rather than manually offsetting them in CAD or Revit. However, we certainly could easily perform that task manually in either program. If you only have one truss, it might even be faster. But if you have lots of trusses, then devising a way to have Dynamo perform the task is ultimately the better choice. And that is the assumption we made here. Now, there certainly are issues that might arise if you tried to use this workflow unmodified on lots of drawings. We have simplified it quite a bit to focus on our core goals. In a real work environment, you may want to introduce more error-trapping to account for variations in different drawings. So, with some of those assumptions and caveats out of the way, let's open and dissect the Dynamo graph.

2. Launch Dynamo and open in manual execution mode, the file named:

***08\_Structural\_Truss\_from\_Shops.dyn.***

## EXTRACT LINELINE FROM CAD LAYERS

You will probably get a warning on one of the nodes about an unresolved package. This workflow uses a package called *BimorphNodes*. Please install that to continue (see Figure 79).

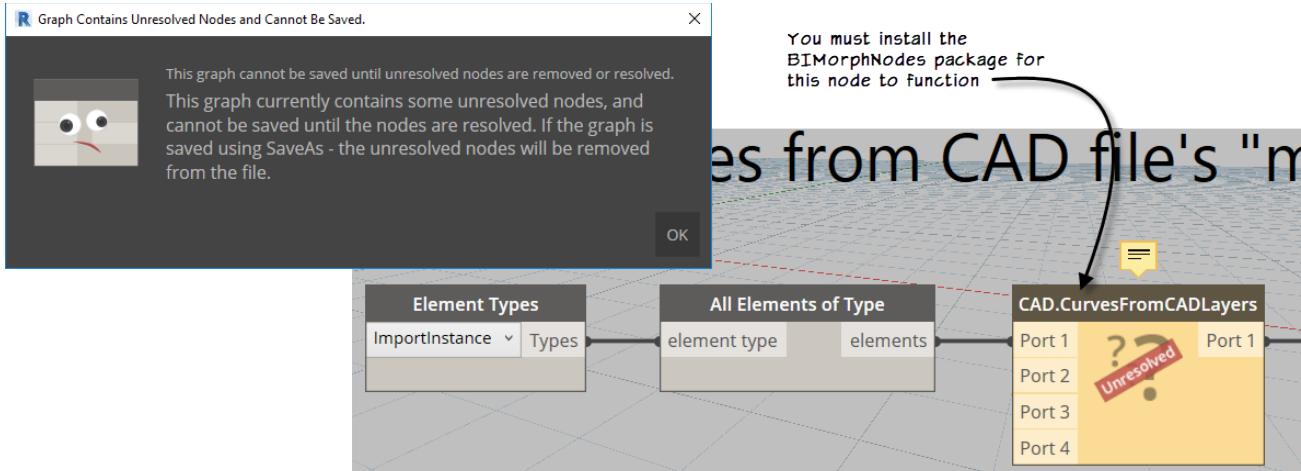


Figure 79

3. From the Packages menu, choose: **Search for a Package**. Input: **bimorph** and then download and install the **BimorphNodes** package.
4. Close the graph. Do NOT save the file. Then reopen the graph after installation.

After installing the package, the error will be resolved, and the graph will reopen without issue.

This is the complete workflow. It is organized into numbered groups and each group has a frozen node near the beginning. This will allow us to run the graph one group at a time and discuss how it functions as we go. There are notes above each group that say: UN-FREEZE. These will help you find the frozen nodes in each group.

You will also want to use Preview Bubbles and Watch nodes along the way to understand each piece of the graph.

5. Run the graph.

The nodes in group 1 will process. The first node: **Element Types**, we have seen before. This is set to: **ImportInstance**. So, it will grab all imports in this file. In this case, there is just one.

6. Expand the Preview Bubbles on the **All Elements of Type** node and the **Flatten** node. Move the **Watch** out of the way.

We can verify that there is only one import instance in the preview bubble of the **All Elements of Type** node. The element id for this element will be listed in green.

The **CAD.CurvesFromCADLayers** node is from the **BimorphNodes** package. It has four inputs but notice that we only have **importInstance** wired up. The **layerNames**, **createModelCurves** and **lineStyleMap** ports are optional. If you hover your mouse over the other imports, they give information on what these optional inputs do.

Look at the output from the **CAD.CurvesFromCADLayers** node in its preview bubble. The node has two outputs: **Curve** and **layerKeys**. We have seen before that when you have multiple outputs, you get a structured list in the preview bubble. If you want to see each output

separately, use a **Watch** node. You can see the output from the **layerKeys** port in the **Watch** that was included in the graph.

Look at the output on the preview bubble and notice the structure of the list. It is a nested list several levels deep. You can feed the **Curve** output into the **Watch** node to see the structure of just the geometry elements. Either way, this structure is unnecessary for our purposes. So, a **Flatten** node is used to remove it (see Figure 80).

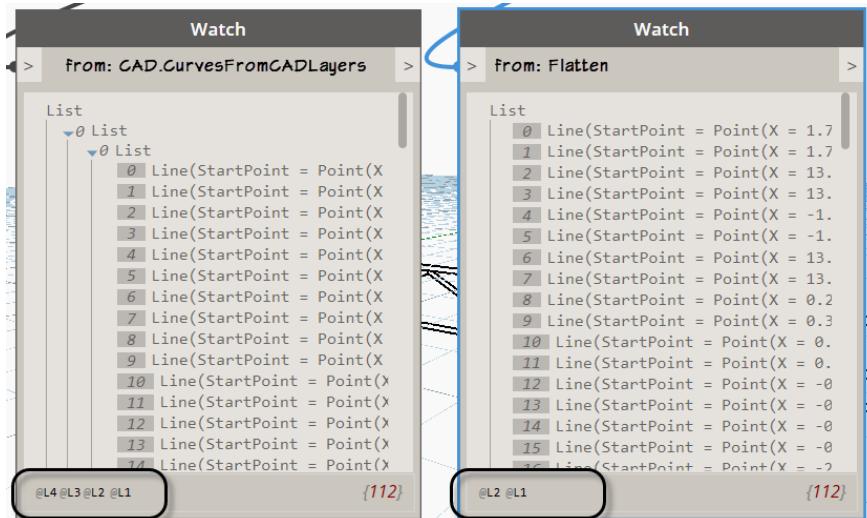


Figure 80

To summarize what happened in group 1, we selected all Import Instance elements from the current Revit document (1 in this case) and then using the node from the **BimorphNodes** package, extracted all the lines on the **members** layer to a flattened list<sup>26</sup>.

Let's move on to group 2.

#### ANALYZE THE EXTRACTED LINES

The drawing contains linework that represents several dimension lumber pieces that create a truss. This means that the shapes are all long (mostly rectangular) shapes with small lines capping the ends. For vertical and horizontal members, they are rectangles with short lines capping the ends that are 2" nominal (1 1/2" actual dimension). For the angled members, there are typically two lines creating the chamfer cuts. These are even shorter. Ultimately, we want to create centerlines for each of the dimension lumber members represented. Toward that end, the goal of group 2 is to remove the small lines leaving us with only the long sides of each shape. The logic behind this is simple. The centerlines are parallel to the long edges. Any line that is shorter than 2" must be one of the ends. So, we can reasonably discard the short ends.

7. Locate the UN-FREEZE note above group 2.

<sup>26</sup> To be clear, the **CAD.CurvesFromCADLayers** node extracts all linework on the indicated layers. You would have to use the **layerKeys** it generates to filter the list to just one layer if you have a CAD file with multiple layers. This is why we removed the unwanted geometry in CAD from those layers first. You can also delete the layers in Revit by selecting the import, and then on the ribbon, use the Delete Layers button. Or simply import only desired layers during the import process via the **layerNames** input.

It is above the first node in the group. When a node is frozen, the right-click option for Freeze will be checked. If you right-click and choose it again, the check is cleared, and the node unfreezes.

**8. Right-click the first node in the group (*Curve.Length*) and choose: Freeze.**

Group 2 has three nodes plus a **Code Block** input and a **Watch** node. The first node: **Curve.Length** does exactly what its name implies. It gives the length of each curve. The next node takes these lengths as inputs (in the **x** port) and checks to see if their values are greater than whatever value we feed into the **y** port.

Here we are using a **Code Block** to do the math directly and feed it in as a number<sup>27</sup>. This is a nice shortcut over using a **Number** node. With a **Number** node, you would have to also use a division node to do the math. With the **Code Block**, we just type it in like a fraction.

The result of the **>** node is a list of True and False values. Above in the “Split the List by Values” topic, we used a **List.FilterByBoolMask** node. You may recall from that exercise, that this node separates a list into two lists based another list of true and false values. We are using that again here. All values that are true go to the in list, while the false values go to the out list.

**9. Run the graph. Study the results in the preview bubble and **Watch** nodes (see Figure 81).**

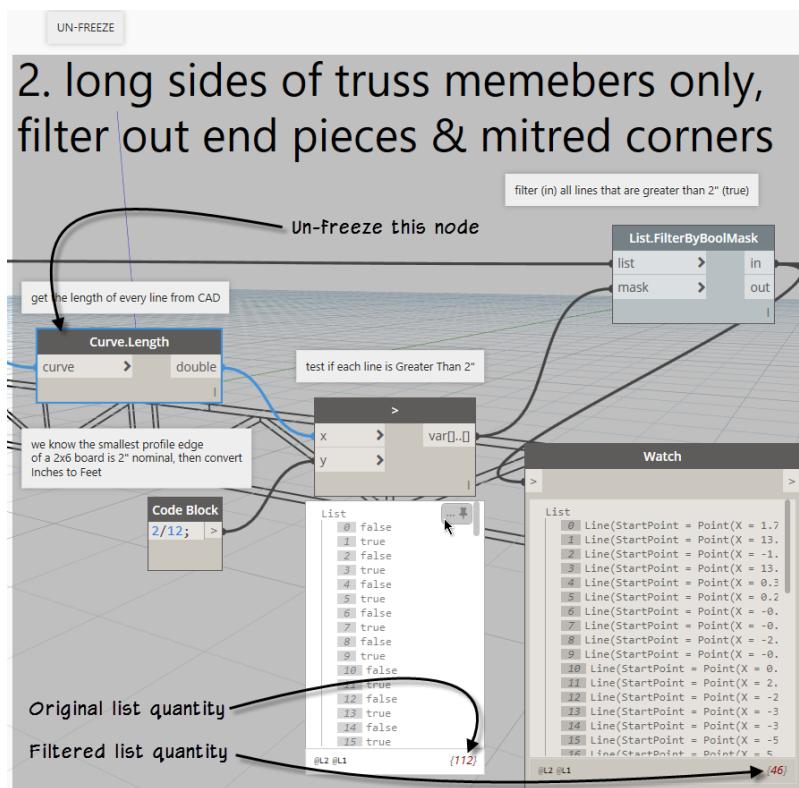


Figure 81

<sup>27</sup> The math required is explained with a note directly in the graph, but the issue is simple: Revit uses feet as its default unit in imperial files. But dimension lumber is measured in inches. So, by inputting 2/12 we are doing the math required to convert inches to feet on the fly in the **Code Block** node.

Not bad. We eliminated 66 unneeded lines. The next thing we need to do is focus on this smaller list of 46 elements and create our centerlines. But to do that, we must pair them up. On to the next group.

#### GROUP PARALLEL LINES BASED ON PROXIMITY TO EACH OTHER

When you and I look at the truss drawing, it is easy for us to see where the members are. This is because we are trained to look at such drawings and analyze what we are seeing. But how do we instruct Dynamo to do the same thing? What makes this collection of 46 lines recognizable as 23 separate pieces of lumber? How do we tell Dynamo where the centerlines of each of those members ought to be? If you can figure that out, you are on your way to thinking like a programmer! In this example we will showcase one approach which does seem to do the trick. But this is by no means the *only* way to do it; and won't always be the best way either. So now that you know the problem, how would you approach it? Maybe you can take a moment to think about it before looking at our solution. Go ahead, we'll wait...

So, what did you come up with? Let's continue with the graph and see what we came up with. Our approach is to find the midpoints of all the lines, and then draw a line (a vector actually, but more on that below) between these midpoints for all possible pairings! (Cross Product lacing) In case you are wondering, yes that is a lot! 2,116 to be exact. So why do this? Well, before we discussed how these members are 2 by dimension lumber. Therefore, if we measure the distance of each line's midpoint to every other line's midpoint, the shortest one will be the closest and the one that ends up being parallel. To help you visualize this, here is an illustration (drawn manually in AutoCAD) of just one line and its midpoints connected to the midpoint of every other line (see Figure 82). Now just 45 more to go...

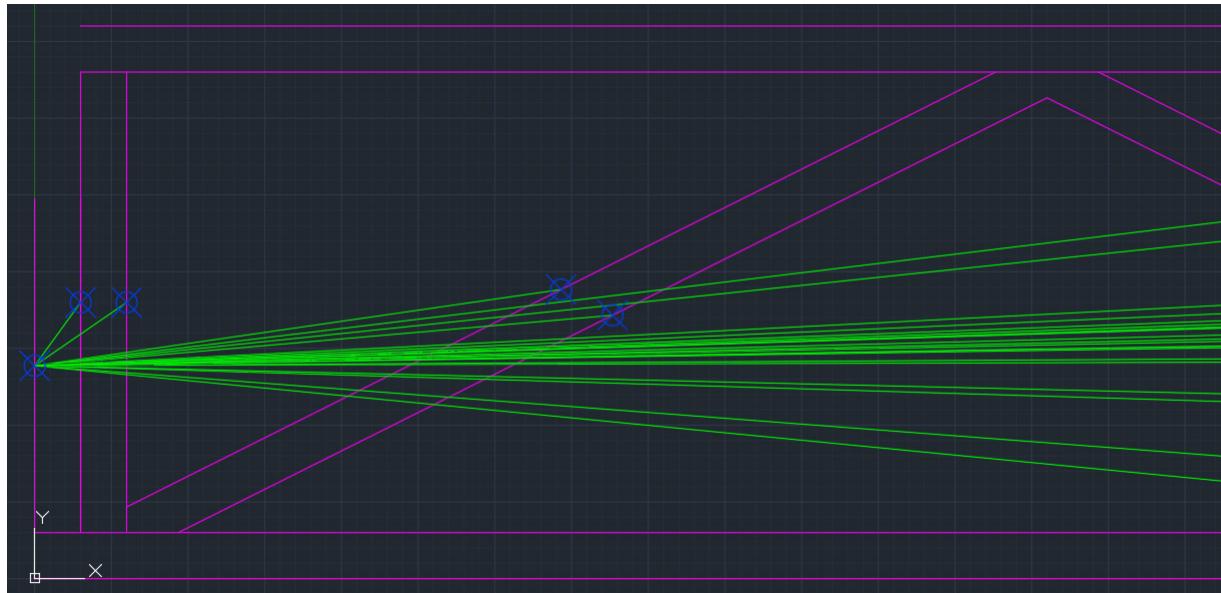


Figure 82

That was a lot of work to do manually and that was just the first one. Dynamo will create over 2,000 of them!

10. Unfreeze the **Curve.PointAtParameter** node in group 3 and run the graph.

So now that you know what our strategy will be, let's look at the nodes that achieve it. We first saw the **Curve.PointAtParameter** node in the "Placing a Single Family" exercise above. Here we are using the parameter value of: **0.5<sup>28</sup>** to place the point at the midpoint of each line.

Also note the color of the **Curve.PointAtParameter** node: it is light gray. This indicates that its preview is turned off. That is; the preview of the node in the Dynamo canvas. If you find the canvas gets too busy, you can declutter it by hiding the preview selectively per node. It works just like Freeze. Just right-click the node and choose the: **Preview** option. When it is checked, the preview shows in the canvas. Uncheck to hide the preview. If you decide you'd like to see these midpoints onscreen, just right-click and check the Preview. You do not need to run the graph to change the preview setting.

### VECTORS

Next, we have the **Vector.ByTwoPoints** node. Vector is a mathematical concept. If you do a Google search for vector, you will get various mathematics websites that provide definitions. Such as this general definition from the website: [http://mathinsight.org/vector\\_introduction](http://mathinsight.org/vector_introduction)

“A vector is an object that has both a magnitude and a direction.  
Geometrically, we can picture a vector as a directed line segment, whose length is the magnitude of the vector and with an arrow indicating the direction. The direction of the vector is from its tail to its head.”

This part of the graph has some explanatory notes and is highlighted in pink (see Figure 83).

### 3. group parallel lines of truss members

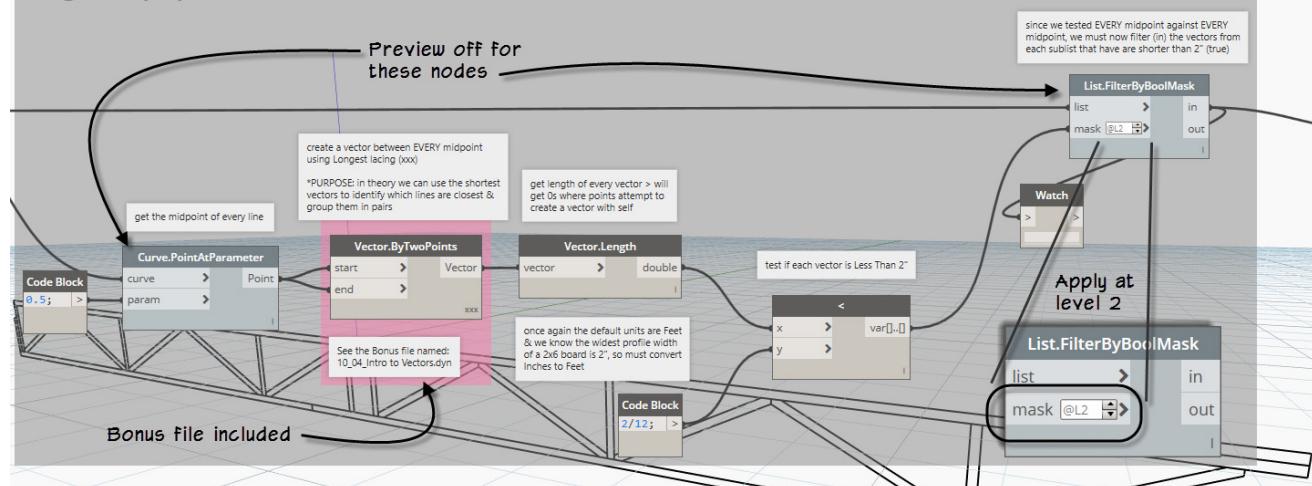


Figure 83

There is a note indicating that you can find a supplemental file in with the Bonus files that discusses vectors. In general, the idea is that creating vectors is conceptually like drawing lines but not as computationally intensive. Therefore, if you do not need to see the lines in canvas,

<sup>28</sup> Please note that if you are using a **Code Block** for input here, you must type: **0.5** and not **.5**. The preceding zero is required. Without it, the **Code Block** will go into an error state.

then using vectors can be a good way to help your graph perform better. To see this in action, try opening the sample graph provided after you complete this exercise:

**BONUS FILE:** Introduction to Vectors: [10\\_04\\_Intro to Vectors.dyn](#).

There are a lot of notes in the graph. These can sometimes get in the way of the preview bubbles. So, either move the notes around or you can open the preview bubbles temporarily and then close them again (don't pin them open).

Note the quantities coming out of the first two nodes. From the **Curve.PointAtParameter** node we have 46 points. But since we are feeding all 46 points into both the start and end ports of the **Vector.ByTwoPoints** node and it is set to Cross Product lacing (xxx). We end up with 2,116 items. The same is true for the next node (**Vector.Length**) which not surprisingly, measures the length (magnitude actually) of each vector.

### GROUPING THE LINES

There is another **List.FilterByBoolMask** node here. To get the list of true and false values we need for the mask, we will check each of our 2,116 lengths to find the shortest ones. We are using a value of 2" again here. The actual distance of the parallel lines is 1 1/2", but since some of the midpoints don't line up perfectly, a value of 2" provides a little room for variation without capturing too many. Finally, given the list structure, we are applying the list of Booleans at level 2 using list a level.

There is a **Watch** node coming out of the BoolMask. Notice that there are 90 items. This is because we tested every midpoint against every other midpoint. So, we get two versions of each pairing. If you test line A against line B, you get one pair. But then a second pair when you test line B against line A. We only need one set of pairs. This is the job of group 4.

#### 11. Unfreeze the **Flatten** node in group 4 and run the graph.

This group is self-explanatory. But let's run through it. We remove the structure with a **Flatten** node. Then using the **List.UniqueItems**, we take only one of each unique value. This removes the duplicates. You can compare the totals from the Flatten with this node to see the result (see Figure 84).

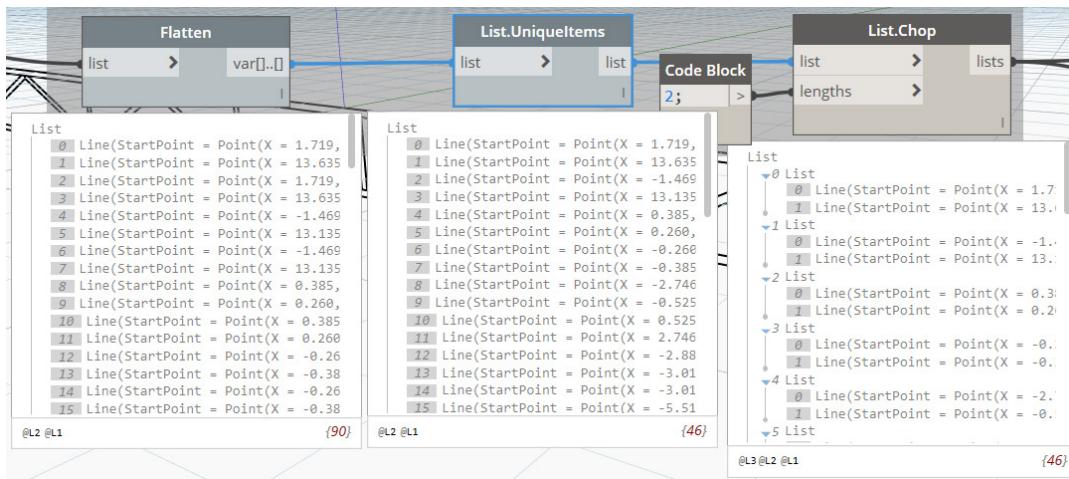


Figure 84

The **List.Chop** node reintroduces structure to the list by chopping the one list into several smaller list of 2 items each. These are our pairs of parallel lines. Onto group 5.

### CREATE CENTERLINES

Group 5 looks a little intimidating. It certainly has the most nodes of all the groups. Let's go through it methodically. We start with another **Curve.PointAtParameter** node at **0.5** creating midpoints again. Since this is the second time we are doing this, the previews of these nodes are hidden to keep the canvas uncluttered.

**12. Unfreeze the *Curve.PointAtParameter* node in group 5 and run the graph.**

Check the preview bubble and you now have 23 pairs of points (46 total). We are going to make vectors from these points. Then use half of the magnitude (length) of these vectors to copy lines parallel to the originals. This takes place on the right side of group 5. To create the vectors, we use the **Vector.ByTwoPoints** node. The **Vector.Scale** node halves the length of these vectors and the **Geometry.Translate** node makes copies of one line in each pair of lines at a distance equal to this vector (half the distance) which puts it right in the center (see Figure 85). Notice that we are also using list at level again.

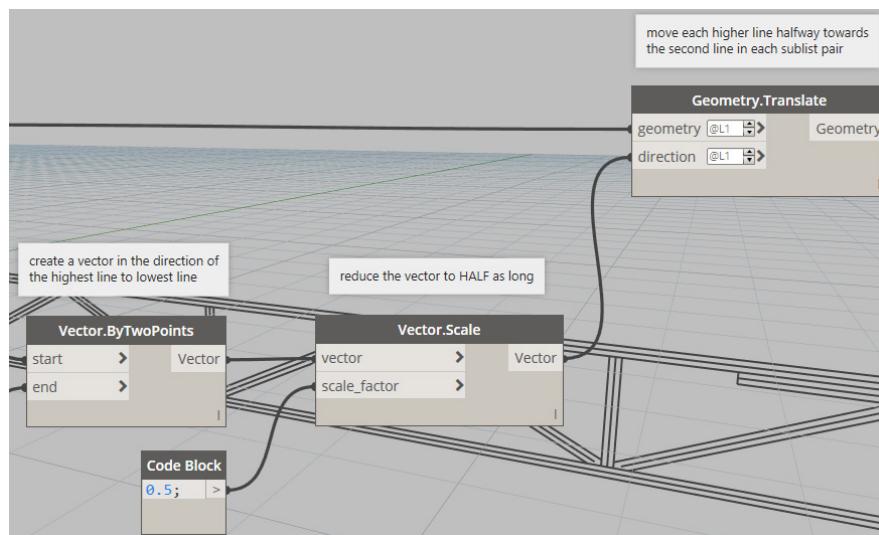


Figure 85

So why do we need all that stuff in the middle of group 5 then? Well this just keeps it all organized and consistent. If we simply fed our list of points into the **Vector.ByTwoPoints** node, we would get results. It is just that the orientation of the lines would vary from line to line. So, to try and make everything consistent, we added a little sort and organize to the lists first.

The concept we are using here is a “keyed” list. Our drawing is oriented vertically. Therefore, if we query the Z coordinate (the **Point.Z** node) of each point, we can use those values to sort the list consistently. In this case, we will put the lowest Z value first. The **List.SortByKey** node uses two lists. It sorts the first list by the values appearing on the second list (the keys).

Since the results of **List.Chop**, **Curve.PointAtParameter** and **Point.Z** are structured lists, list at level with @L2 is used to feed two **List.SortByKey** nodes and make sure that they are operating on the sub lists of their inputs and *not* the main list (see Figure 86).

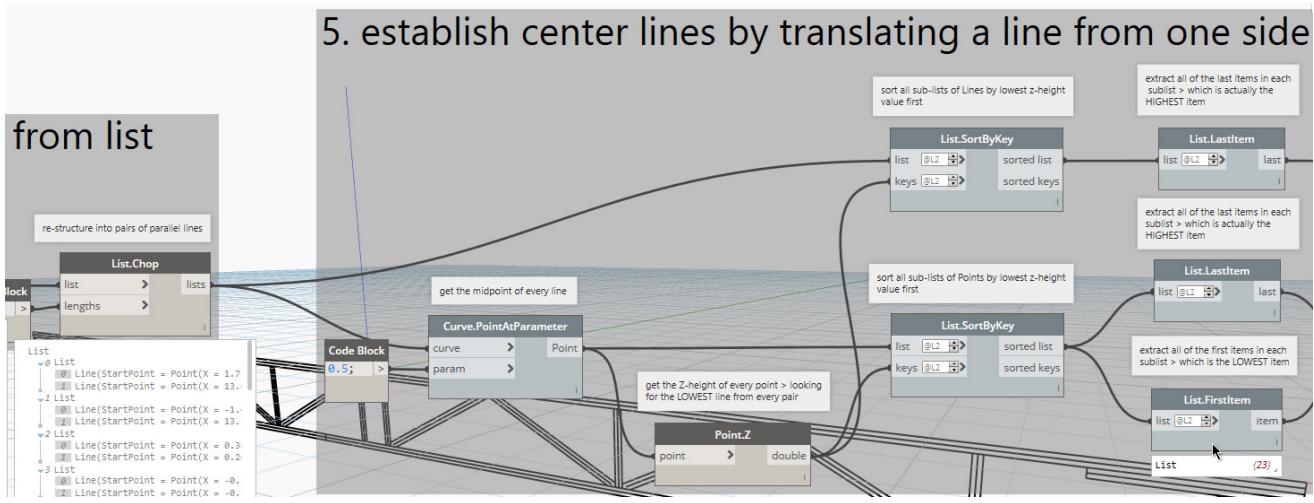


Figure 86

We now have our centerlines! And yes, the logic there was a bit challenging, but if you can follow along with it, you are on your way toward building similar solutions in your own work. But we aren't done yet. We still need to do something with these lines. The last three groups create framing members, then change their justification and finally orient them properly. Let's take a look.

## CREATE STRUCTURAL FRAMING

Only three nodes in group 6. The **StructuralFraming.BeamByCurve** node has three inputs. The curve input uses the centerlines we just built. You also need to feed in a level and type. Since we are in a family file, the level is: **Ref.Level**. The family type is pre-loaded in the sample file. But if you need to, jump back over to Revit and load a family before running the graph.

13. Unfreeze the **StructuralFraming.BeamByCurve** node in group 6 and run the graph.

14. Position your windows so you can see the Revit window to observe the results (see Figure 87).

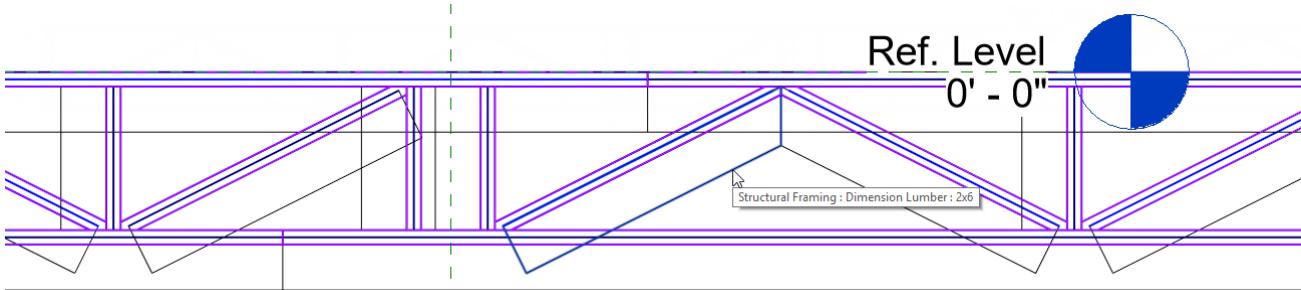


Figure 87

Not bad, but clearly there are issues. The members are using a default justification in the y and z directions. Let's adjust this next.

## CHANGE JUSTIFICATION AND ORIENTATION

You can adjust the justification of structural members in Revit on the Properties palette using drop-down lists for y Justification and z Justification (see Figure 88). If this is your only truss,

simply do this in Revit. Right-click, **Select All Instances > Visible in View**, and make the change directly on the Properties palette. Again, you do not *need* to do everything in Dynamo!

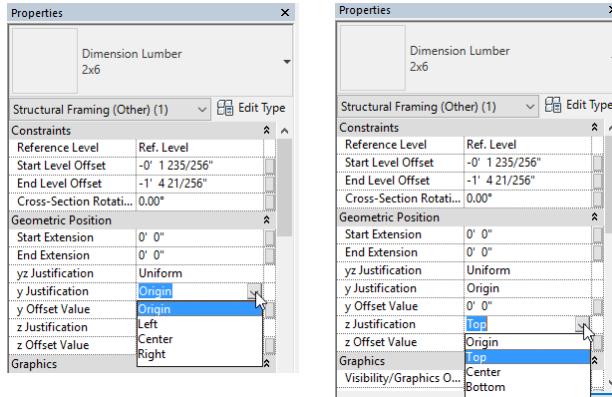


Figure 88

However, we said at the start that one of our assumptions was that there would be several of these. So, adding a bit more to the graph can perform these tasks for us automatically instead. There are four options for each justification. We have seen the **Element.SetParameterByName** node several times now. We can use this node to change these parameters. However, what is not obvious looking at the properties is how to format the input. It turns out that you must use *numerical* inputs to assign each justification option (see Table 1).

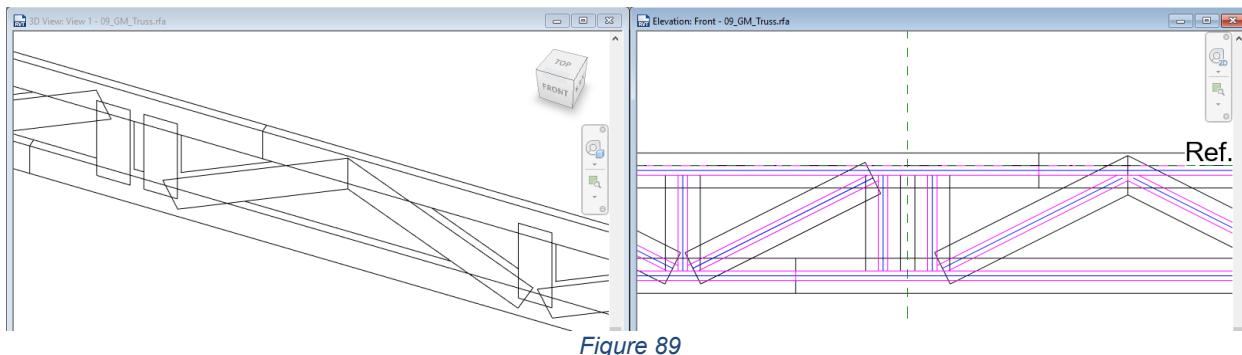
Table 1

Structural Member Justifications			
y Justification	Dynamo Value	z Justification	Dynamo Value
Left	0	Top	0
Center	1	Center	1
Origin	2	Origin	2
Right	3	Bottom	3

Let's go with center justification in both directions. That means the input is a value of: 1. If you look in group 7, there is another new shorthand used here that we have not seen before. You can put multiple values in a single **Code Block** by adding a carriage return to separate them. So instead of needing a separate **Code Block** for y and z Justification, you can put both in the same **Code Block** and just put each on its own line. You then get two outputs from the **Code Block**.

15. Unfreeze the first **Element.SetParameterByName** node in group 7 and run the graph.

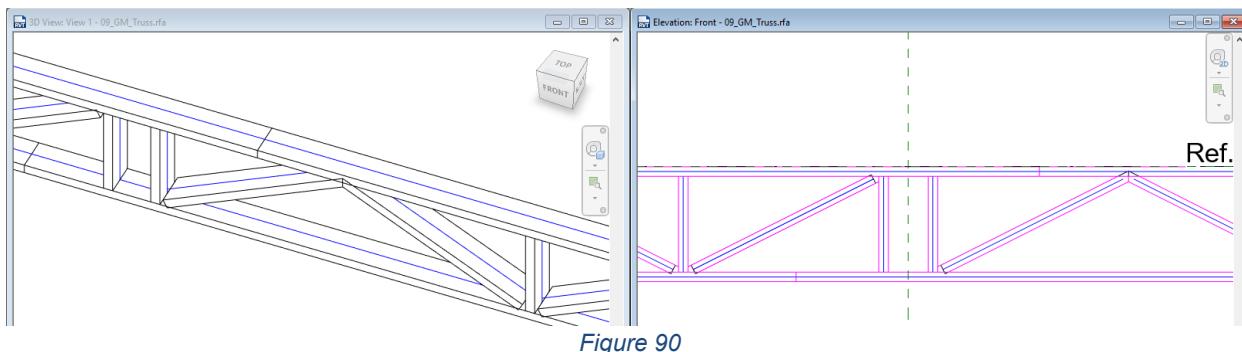
Almost there. Now we just need to rotate the members to the proper orientation.



To change the orientation, we set one more parameter.

**16. Unfreeze the final node in group 8. Run the graph.**

Here we are changing the Cross-Section Rotation parameter by 90°.



You made it! That's our completed truss from an imported CAD file. Now if you had dozens of similar CAD files, you could run this script from Dynamo Player on each family which would certainly be quicker than building each one manually!

## A CURTAIN WALL CHALLENGE

The goal of this example involves a curtain wall façade and using Dynamo to help us place many custom shaped curtain panels. The façade has an irregular shape along the edges as it climbs the height of the building. To get the custom curtain wall shape, Edit Profile was used (see the right side of Figure 91). The curtain panel being used in the design is a custom curtain panel family (traditional family editor). The family has a lower horizontal spandrel, a glazing area and then another horizontal band across the top with a slab cover. Instead of using Revit mullions, the mullions are modeled directly in the panel family and wrap around the edges as well as horizontally dividing the three zones of the panel design (see the left side of Figure 91).

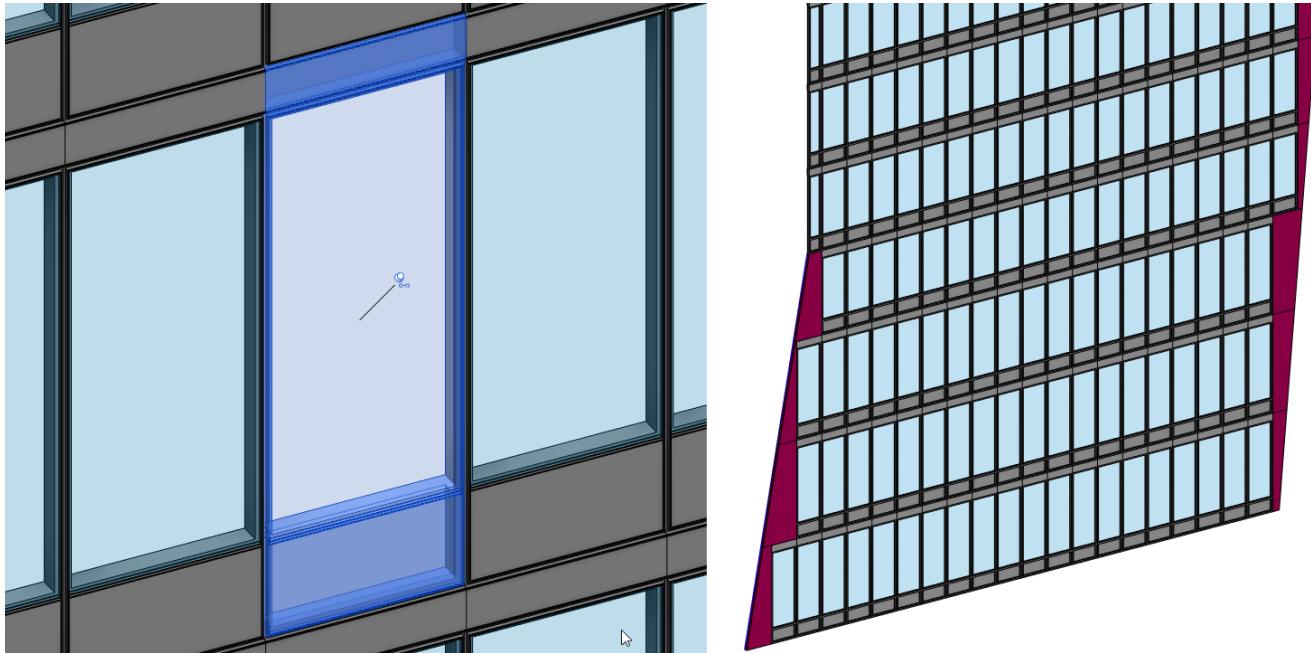


Figure 91

If you have ever used a custom panel in your curtain wall designs, you know that a significant limitation exists; they must always be rectangular. So the challenge here is how to deal with the non-rectangular edges of the façade. When you use Edit Profile on a curtain wall, you typically end up with non-rectangular edge panels. Revit will generate an error for these panels and then replace them with system panels. To make them stand out better, all instances of these system panels were selected and a new type that is shaded red was substituted for them (see the right side of Figure 91).

The problem with System Panels is that while they can be any shape, they are limited to simple extrusions. So you cannot add any of the intermediate geometry for horizontal mullions, slab covers or spandrels. However, you *can* build such things with an Adaptive Component family. And adaptive components can be any shape you like. The exact process followed to build the adaptive version of this panel could fill its own class, so we will not be going into the specifics here. To summarize: it is a four-point adaptive component (that can be a non-rectangular shape) and mimics the same look as main panel with its spandrels, slab cover and mullions. These are placed with similar proportions and parameters to match the original. When you place this component, the lower spandrel and top spandrel will flex with the panel but stay parallel to the top and bottom edges. You are welcome to open the adaptive family and take a look. Just close it without saving before you continue (see Figure 92).

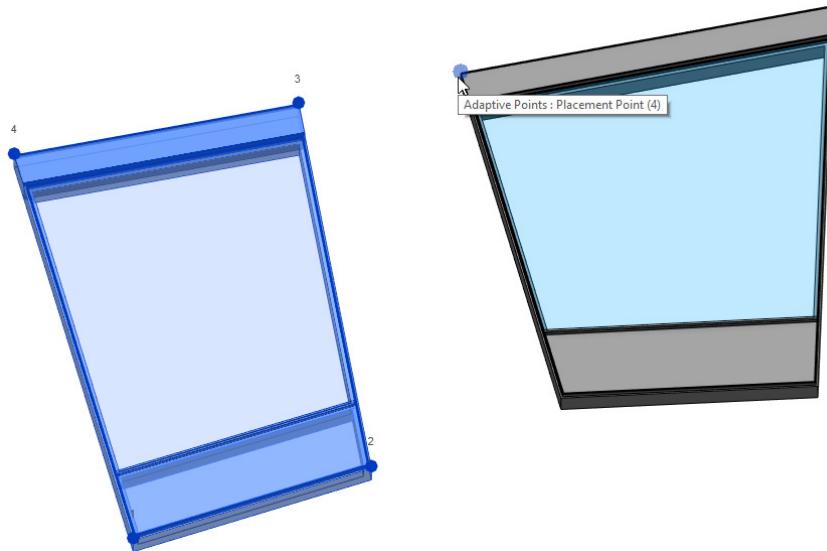


Figure 92

The focus of this exercise will be to use Dynamo to select all the edge panels, extract their corners and then use those points to add the custom four-point adaptive family instances to the model in each location required. We have a fairly simple dataset here. There are only a handful of custom panels. While you can use this workflow in larger projects, adaptive components do come at a cost. There can be a performance hit especially if there are many of them. So factor this into your planning and decision-making process before settling on your final solution.

## PLACING ADAPTIVE COMPONENTS

To begin, let's start with how adaptive components work. An adaptive component is a family with special "super powers." Among other things, it can be helpful to think of it as a family that can have more than one insertion point. These are called "Placement Points" and during placement, you will click to place each placement point of the family at some location in the project. Furthermore, if the adaptive component is built with this feature in mind (which is kind of the point of them) then the shape of the family will "adapt" to the placement points. So in our case, we have several shapes on the perimeter of our curtain wall that are not rectangular. However, the designer tried to ensure that they are all four-point polygons. This is important since your adaptive component will have a set number of placement points. If you try to place a four-point adaptive component but feed it three points or five points, it will fail. So the first thing we want Dynamo to check for us is how many edges (and vertices) each panel has.

But there is another important issue as well. The points *must* be placed in a logical and consistent order. For example, always clockwise or always counter-clockwise. But typically, not "crisscross." Experience shows that crisscross is rarely an issue, and that they will typically always go in only one direction, but without careful data processing, Dynamo will not always start at the same point. In other words, it will sometimes try to place the adaptive components rotated or upside-down. This can cause them to fail in Revit.

Finally, if you want Dynamo to be able to successfully place adaptive components, they must be built carefully in Revit and tested thoroughly. Remember, Dynamo does not invent new Revit functionality. It merely performs tasks in Revit (albeit often more quickly or efficiently), but is

limited by the same limits you might experience if doing it manually. In other words, if the family flexes in such a way as to cause it to fail, Dynamo will not be able to overcome this. Adaptive components can be quite fussy. Again, this paper will focus on the Dynamo part of the solution, but keep in mind that if you want Dynamo to be successful in placing adaptive components, they must be built carefully and thoroughly tested.

With the family used here, one thing that proved quite bizarre, was that the component worked perfectly and flexed as expected when placed in Revit manually, but Dynamo was unable to place it. Ultimately, the issue seemed to stem from the family template used. The: *Curtain Panel Pattern Based.rft* template with its built-in adaptive points was unsuccessful. But when rebuilding the adaptive component family based on the: *Generic Model Adaptive.rft* template (which has no adaptive points in it to start) and then adding the adaptive points manually, it worked without failure in Dynamo. So perhaps there is something about the built-in adaptive points in the curtain panel template that differs from the generic model one. More testing would be needed to know for sure. But the final version of the adaptive component used in this exercise was started from the *Generic Model Adaptive.rft* template. Four adaptive points were added manually and numbered in the correct order. The geometry is then added to this file. Interestingly enough, the geometry in this case is actually the original panel based on the *Curtain Panel Pattern Based.rft* template. It was nested in to this construct and it worked great in Dynamo. Very strange...<sup>29</sup>

### REPLACE THE IRREGULAR CURTAIN PANELS

The first thing we need to do is select the elements in Revit that we want Dynamo to process. Remember that the irregular shaped curtain panels automatically get swapped out to the default system panel by Revit. In the starter file we have here, those panels have been colored red to make them stand out, but otherwise, they are exactly what you would get if you performed an Edit Profile on a Curtain Wall. The goal in this graph will be to place custom shaped adaptive components at each location where there is currently a red panel. But we don't want to place the adaptive panels on top of the red ones. However, in curtain walls, you cannot delete a panel you don't need. Instead, you change them to a special system panel called: Empty. So let's do that before we begin the graph<sup>30</sup>.

Close Dynamo and any existing Revit files, you can leave Revit running.

1. In Revit, open the file named: *09\_Curtain Wall Facade\_!Start.rvt*.
2. Highlight one of the red panels (use the TAB key to assist,) right-click and choose: **Select All Instances > Visible in View**.
3. From the Type Selector, choose: **Empty System Panel:Empty**.

When you deselect the panels, it will look as if they have been removed, but they are still there, but the empty panel has no visible geometry.

<sup>29</sup> You will also see a few of the “testing” panels in the starter file.

<sup>30</sup> Tasks like this one are much easier to complete directly in Revit. It is easy to get “Dynamo Vision” and assume that everything must be done in Dynamo. Not so! One of the best features of Dynamo is that you can move back and forth between your graph and the Revit interface as needed.

4. Launch Dynamo and create a new graph.
5. Save the graph as: **Curtain Wall Edges**.
6. Add the following nodes to your canvas:

Library Location	Node
Revit > Selection	Family Types
Revit > Elements > Adaptive Component	ByPoints

7. Wire the *Family Types* node into the *familyType* port of the *AdaptiveComponent.ByPoints* node.
8. Select both nodes and group them. Name the group: **6. Create Adaptive Components** (see Figure 93).

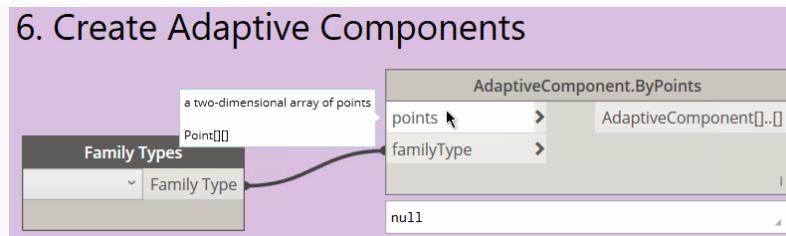


Figure 93

It is sometimes good to start your graph with the end desired result. In this case a node that places adaptive components. This way, you can see what inputs it requires and build your strategy accordingly. The **AdaptiveComponent.ByPoints** node takes two inputs only, so it should be pretty easy right? Well, the *familyType* input certainly is as you can already see. But creating the list of points will take a little more effort. We will begin with extracting points from our empty system panels.

#### SELECT THE IRREGULAR (EMPTY) CURTAIN PANELS

We have already seen a few basic selection methods; manual selection (Select Model Element and Elements) and selection by category (first Walls, then Casework and MEP spaces). In this case, we want to select all elements of a certain type; the Empty System Panel<sup>31</sup>.

9. Pan over to give yourself some room to the left of these two nodes.
10. Add the following nodes to your canvas:

<sup>31</sup> In this dataset, we can use the Empty System Panel directly. If your project already makes use of the built-in Empty panel, then duplicate the type and create another one with a new name before proceeding. Then later in the Dynamo graph, select the duplicated empty panel instead of the original.

Library Location	Node
Revit > Selection	Family Types
Revit > Selection	All Elements of Family Type

11. Wire the **Family Types** node into the **All Elements of Family Type** node.
12. From the drop-down list on the **Family Types** node, choose: **Empty System Panel:Empty** and then expand the preview bubble.
13. Group these two nodes and name the group: **1. Select Irregular Curtain Panels** (see Figure 94).

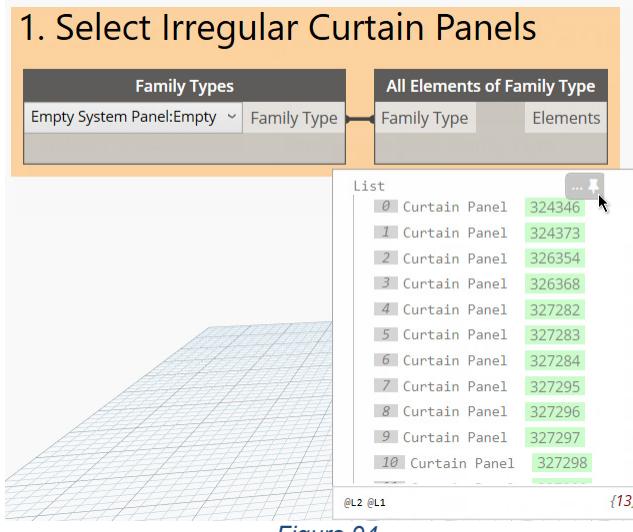


Figure 94

When you expand the preview bubble, you will see a number shaded in green after each list item. These are the Revit element IDs. Whenever you see a green number like this in a preview bubble or **Watch** node, you can click on the green number and it will zoom in on the item back in Revit. Give it a try. Seeing the green numbers is a good verification that you have actual Revit elements in your list.

**SAMPLE FILE:** You can open a file completed to this point named: **09\_Curtain Wall Edges\_A.dyn**

#### TRACE THE SELECTED CURTAIN PANELS

Our list now contains all the irregular shaped panels that use the curtain panel type called: **Empty**. Next, we will trace these with a PolyCurve. A PolyCurve is a Dynamo curve element with more than one curve within its shape. The actual shape can be just two lines, a triangle, rectangle or irregular shape. But all the segments will be considered as subcomponents of the overall PolyCurve. Once we have a list of polycurves, we will check the number of curves contained in each one. Some of our curtain panels will have only four edges, but those that span more than one grid bay in the curtain wall and some other custom shapes, will have more than 4 curves. The ones with four curves are usable as is. But those with more than four must

be filtered out into a new list. So, the next set of nodes in our graph will trace the curtain panels, count the number of internal curves and generate two lists with the results.

14. Add the following nodes to your canvas:

Library Location	Node
Revit > Elements > Curtain Panel	Boundaries
List > Modify	Flatten
Geometry > Curves > PolyCurve	NumberOfCurves
Math > Operators	==
Input > Basic	Number
List > Modify	List.FilterByBoolMask

15. Position the nodes as shown in Figure 95.

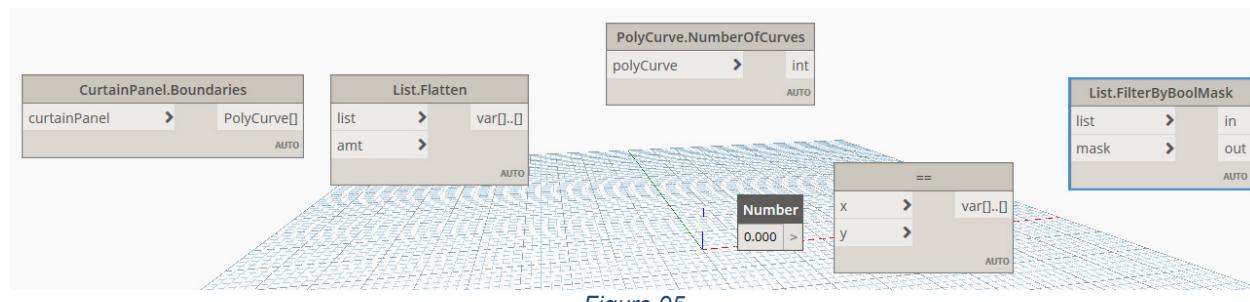


Figure 95

16. Wire the output from the **All Elements of Family Type** node into the **CurtainPanel.Boundaries** node.

17. Expand the Preview Bubble.

You will notice some lines appear in the Dynamo 3D preview. To see them better, we can zoom out.

18. Click the Enable 3D Preview icon in the upper right corner (or press CTRL + B). Then use your wheel mouse to zoom out and pan the view to a comfortable level to see all the curves.

19. Toggle the 3D preview again<sup>32</sup> (see Figure 96).

<sup>32</sup> You can hold the ESC key down and then navigate the 3D preview without toggling. When you release the ESC key, it will disable 3D navigation.

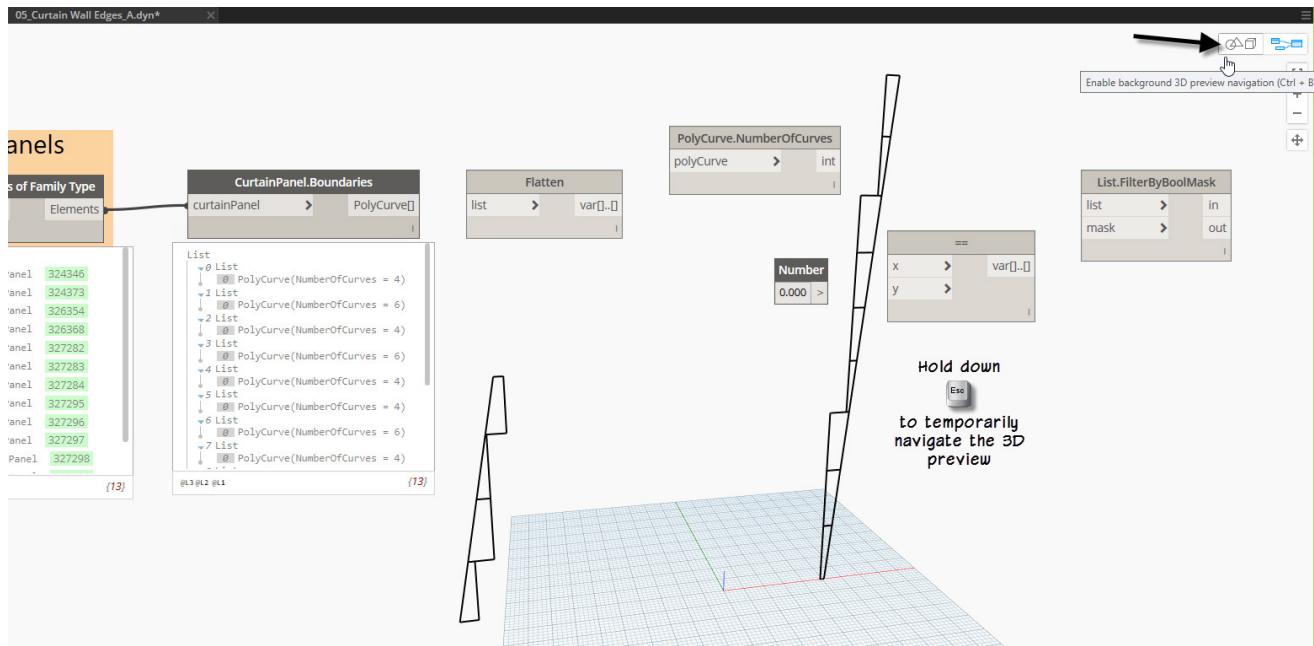


Figure 96

## FLATTEN A LIST

Look at the list coming out of the **CurtainPanel.Boundaries** node. It is a structured list. The main list is a list of lists. Each sub-list contains a PolyCurve. So, each of the 13 curtain panels was traced with a PolyCurve, but they were each given their own list. We don't need this extra level on the list, so we can flatten it. The **Flatten** node removes the list structure. You can flatten the entire list, or by using the amt input with a number, you can reach down complex lists and choose which level to apply the flatten to. In this case, we want to flatten the entire list.

20. Wire the **CurtainPanel.Boundaries** output into the **Flatten** node. Expand the preview bubble to see the result.

Look at the output again. In both the structured list and the flattened version, at the end of each PolyCurve entry there is a "NumberOfCurves" quantity. Most say either 4 or 6. This is the number of sub-curves in the PolyCurve. As we noted above, we want to split the list based on this information. Therefore, the next step is to pull out these quantities separately.

21. Wire the **Flatten** node into the **PolyCurve.NumberOfCurves** node. Expand the Preview Bubble.

You will now have a list containing numbers only for just the quantities (see Figure 97).

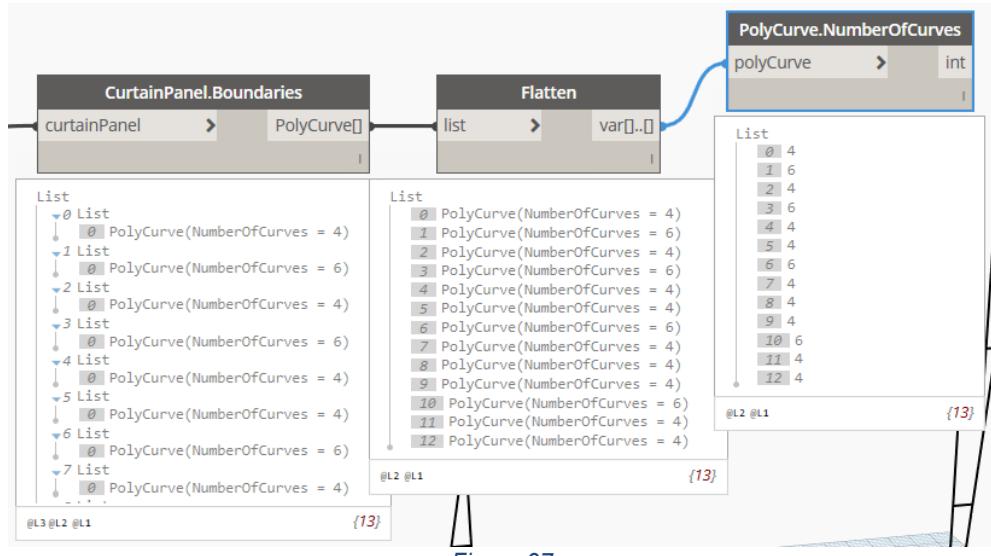


Figure 97

As you can see, depending on how you have positioned things, the preview bubbles might be overlapping each other. You can either move the nodes around to make more room or close the Preview Bubbles after you read them. It is personal preference.

### SPLIT THE LIST BY VALUES

The last two nodes we have in this group are going to split the list for us based on the **NumberOfCurves**. **List.FilterByBoolMask** takes two inputs. The first is a list of items, the second a mask. The mask needs to be a list of Booleans. In other words, a list that contains one of two values: true or false. The mask values separate the list and direct them to the two outputs. True values go to: in and false ones go to: out.

The **==** node is “equal to.” So, we feed in two values and this node asks is: x equal to y? Since this is a yes or no question, we get a list of Booleans as the output. Precisely what we need.

22. In the **Number** node, type: **4**.
23. Wire it into the y port of the **==** node.
24. Feed the output from the **PolyCurve.NumberOfCurves** node into the x port.

If you expand the Preview Bubble you will now have a list of true and false values.

25. Feed this output into the mask input of the **List.FilterByBoolMask** node.
26. Feed the output of the **Flatten** node (Our list of PolyCurves) into the list input (see Figure 98).

## 2. Trace panels and find the items with more than four edges

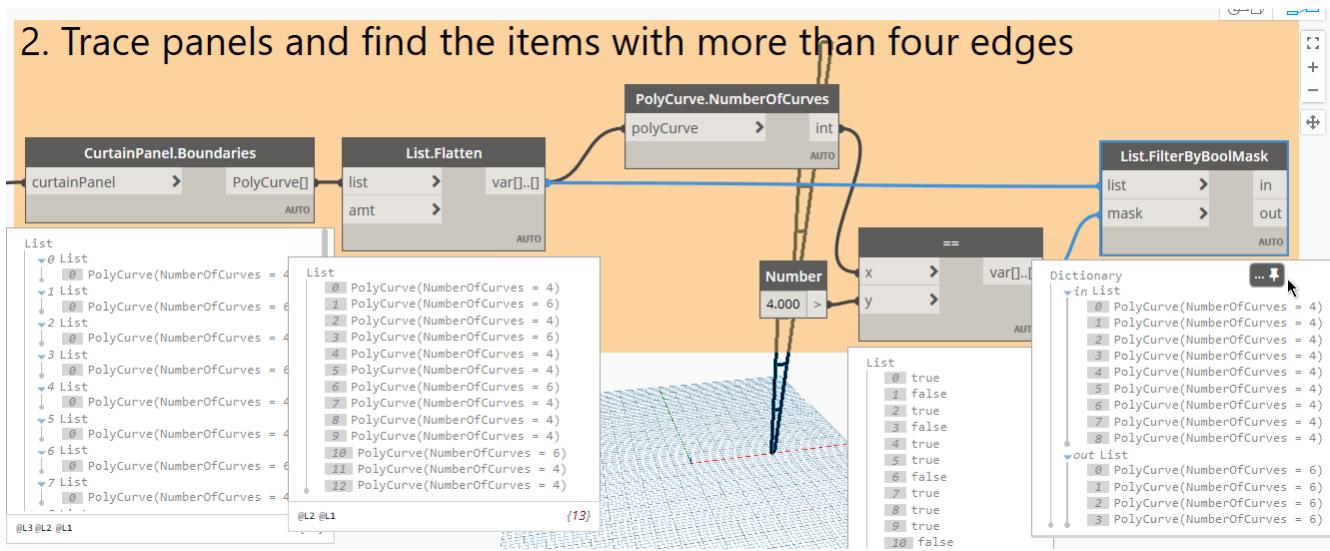


Figure 98

Since there are two outputs from the BoolMask, we get a list with two lists. The first list is the items on the in output and the second list is the out items. You can add separate Watch nodes to each input if you want to see them separately.

27. Select all these new nodes, group them and name the group: **2. Trace panels and find items with more than four edges.**

**SAMPLE FILE:** You can open a file completed to this point named: **09\_Curtain Wall Edges\_B.dyn**

### CONVERT MULTI-POINT POLYCURVES TO FOUR POINTS

As we saw in the previous topic, from the out port of the BoolMask, we get the shapes that have more than four curves. We want to convert these to only four curves and ultimately extract just the four corner points so that they will work with our adaptive component family later. (The output from the in port is fine as-is for now and we can pass it forward below). A PolyCurve is a compound shape made of several curves. (Recall from above that a curve does not always mean “curvy,” so while the terminology is “curve” we are really talking about lines in this example, but they can be any linear element; something with a start and endpoint. It is similar conceptually to how a square is just a special case of a rectangle. A straight line is just a special kind of curve). To process this list, we need to explode the PolyCurves into individual segments. Then we'll determine the corner points from the resultant list of lines.

28. Add the following nodes to your canvas:

Library Location	Node
Geometry > Modifiers > Geometry	Explode
Geometry > Curves > Curve	StartPoint
Geometry > Curves > Curve	EndPoint
Geometry > Curves > Line	ByStartPointEndPoint

Library Location	Node
Geometry > Modifiers > Geometry	DoesIntersect
List > Organize	ShiftIndices
List > Modify	FilterByBoolMask
Input > Basic	Number

29. Position the nodes as shown in Figure 99.

30. Select and group all the nodes you just added. Name the group: **3. Convert PolyCurves of greater than 4 points to only 4 points**.

31. Wire the out port from group 2 into the geometry input of the **Geometry.Explode** node in your new group 3.

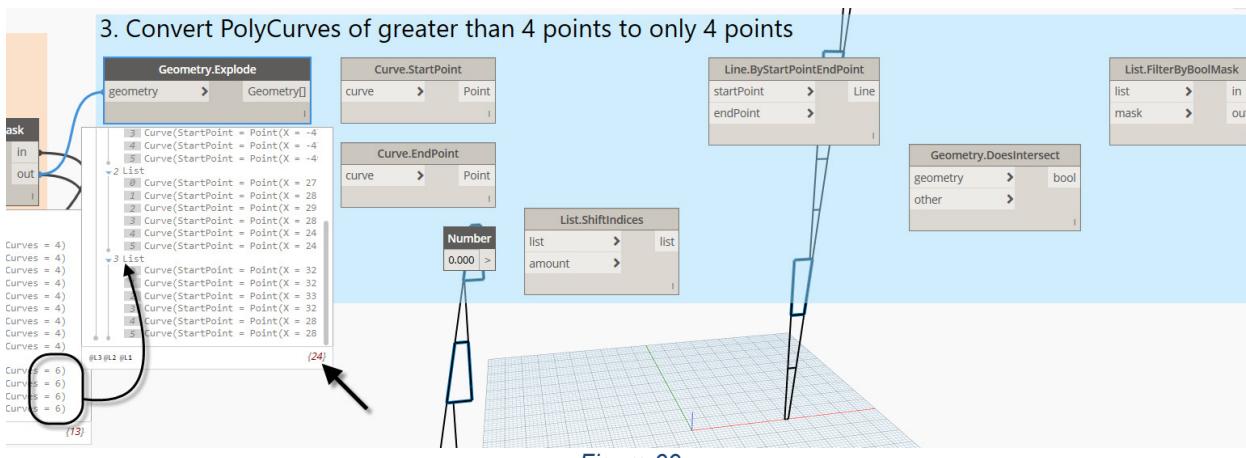


Figure 99

It is always a good idea to compare the outputs and input as the data passes through each node. In this case, the out port of the BoolMask from group 2 had four PolyCurves with 6 curves each. You can see that this generates a list of four sublists from **Geometry.Explode**. The total at the bottom of this node is 24. This is the four exploded PolyCurves times the six curves in each one. So the numbers check out.

32. Wire the output of **Geometry.Explode** into both **Curve.StartPoint** and **Curve.EndPoint**<sup>33</sup>.

This gives us a list of the start and end points from each curve. You can look at the output of each node using the preview bubbles or **Watch** nodes if you like. Next we use the **List.ShiftIndices** node to shift each point by one index<sup>34</sup>. You can control the direction of the shift using a positive

<sup>33</sup> The **Curve.StartPoint** and **Curve.EndPoint** nodes give us the start and end points from each curve. Since we installed the LunchBox package earlier, if you prefer, you can use the **Curves.EndPoints** node instead. This does the same thing, but grabs both points and outputs them to two outputs on the same node.

<sup>34</sup> The index is the position of the element on the list starting at zero.

or negative number. In this case we are using a negative shift. This means the first item on each list wraps around to the bottom and each subsequent item moves up one on the list.

33. Set the graph to Manual execution.
34. Wire the output of ***Curve.EndPoint*** into the ***list*** port of ***List.ShiftIndices***. Set the Number port to: **-1** and then wire it into the ***amount*** port.

It might be easier to see the output with Watch nodes. If you look carefully, you will see that this is currently moving the bottom sublist up to the top. We want the shift to apply within each sublist instead. So, let's apply list at level again.

35. Click the small chevron next to the list port and check Use Levels. Make sure it is set to: **@L2** and run the graph (see Figure 100).

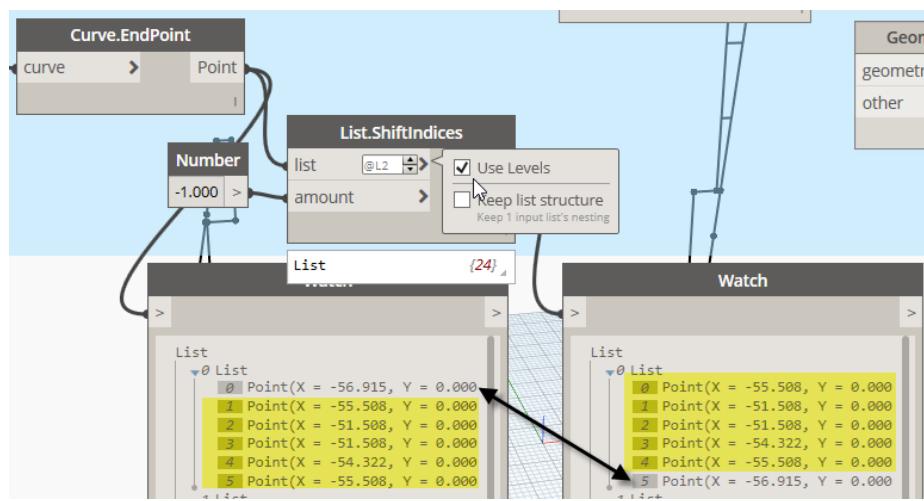


Figure 100

Next we will draw some lines using the ***Line.ByStartPointEndPoint*** node using the original start point and the shifted endpoints. Then we compare this to the original curves. If they intersect; we will flag that shifted point and remove it from the list. If they don't we'll keep the point. In the end we will end up with only four points. Another way to think of this, is if the shifted point falls on the perimeter, the new line will be collinear with the original. Otherwise, it will be drawn diagonally across the PolyCurve shape (see Figure 101).

36. Wire ***Curve.StartPoint*** into the ***startPoint*** port of the ***Line.ByStartPointEndPoint*** node.
37. Wire the ***list*** output of the ***List.ShiftIndices*** node into the  ***endPoint*** port of the ***Line.ByStartPointEndPoint*** node.

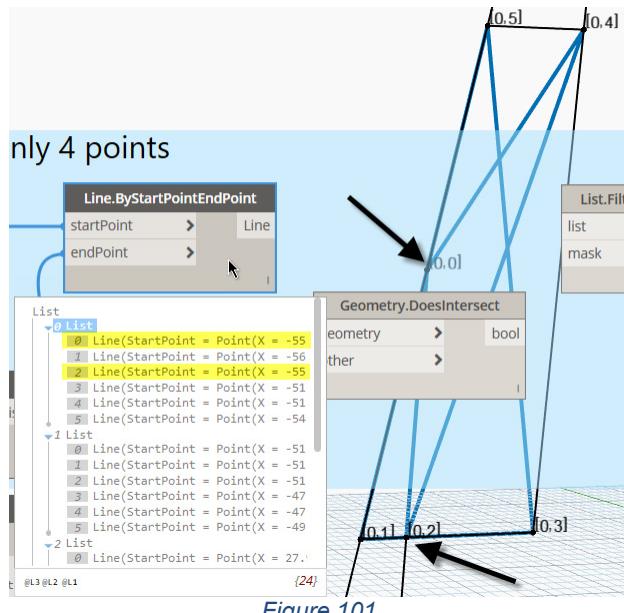


Figure 101

We use **Geometry.DoesIntersect** to check this and send the output to another BoolMask to filter them out from the rest. The output from the mask is the endpoints that did *not* fall on any edges. In other words, the corners only.

38. Wire the **Line** output into the **geometry** port of the **Geometry.DoesIntersect** node.

39. Wire the **Point** output of the **Curve.EndPoint** node into the **other** input.

The **Geometry.DoesIntersect** node outputs a list of Booleans. We are asking the question, does the point fall on an edge?

40. Feed the **bool** output into the **mask** port of the **List.FilterByBoolMask** node.

41. For the **list** port, feed in the output of the **Curve.EndPoint** node (see Figure 102).

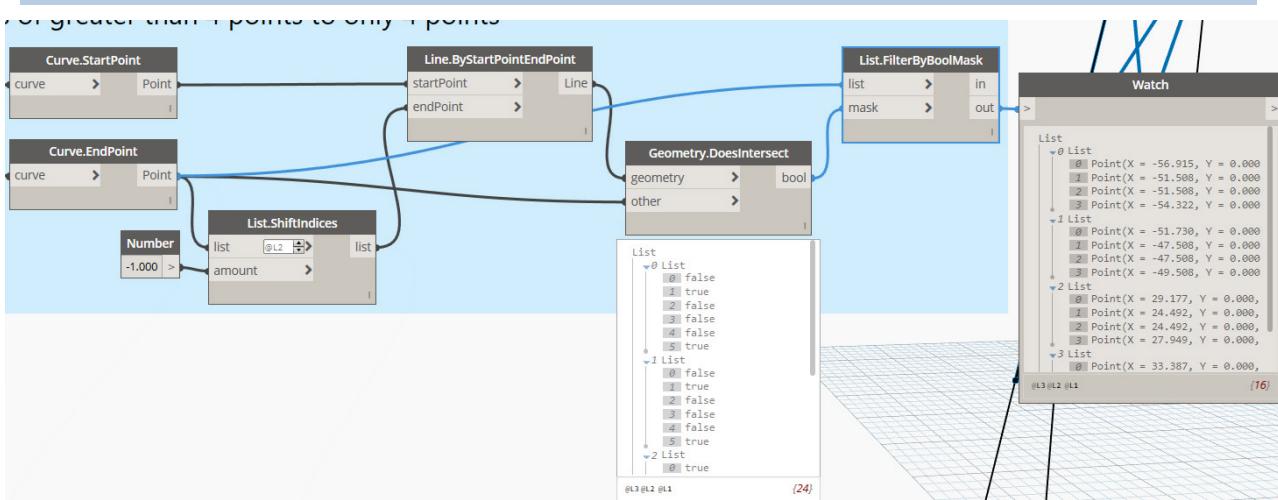


Figure 102

**SAMPLE FILE:** You can open a file completed to this point named: **09\_Curtain Wall Edges\_C.dyn**

### THE DYNAMO COMMUNITY

Dynamo has a vibrant community. This community participates actively in online forums and of course create many custom solutions as we noted earlier when we installed packages. This means that if you ever get stuck on figuring out a solution, help is not far away. The previous topic is a perfect example. When trying to figure out the best way to isolate the four corners of the PolyCurves, the DynamoBIM forum offered up a few solutions. We highly recommend that you add this resource to your list of “go to” sites. Here is a link to the specific conversation noted here:

<https://forum.dynamobim.com/t/create-4-point-polycurves-from-list-of-6-points/4831>

The DynamoBIM forum is an excellent resource and you can ask nearly anything you like and in no time you will have replies from the community of passionate Dynamo users. However, always try the solution on your own first. Questions that show what you have and where you got stuck usually do better than general “how do I do this?” questions without any attempt to try it on your own first.

### GETTING THE REMAINING POINTS

We now have four good points for each of the curtain panels that originally had more than four edges. Going back to the original branched list (from the first BoolMask,) we had two outputs: those that had more than four curves and those that were already four curves. Let’s process the other list now; the ones that were already four curves. All we need to do is explode these PolyCurves and grab their Start points.

42. Add the following nodes to your canvas:

Library Location	Node
Geometry > Modifiers > Geometry	Explode
Geometry > Curves > Curve	StartPoint

43. Connect the **Geometry.Explode** into the **Curve.StartPoint**.

You can add these from the library, or simply select the existing nodes on canvas and copy and paste them.

44. Group them, and then name the group: **4. Existing 4 point PolyCurves**.

45. Wire the **in** port form the **List.FilterByBoolMask** from group 2 into the **geometry** port in the new **Geometry.Explode** you just added.

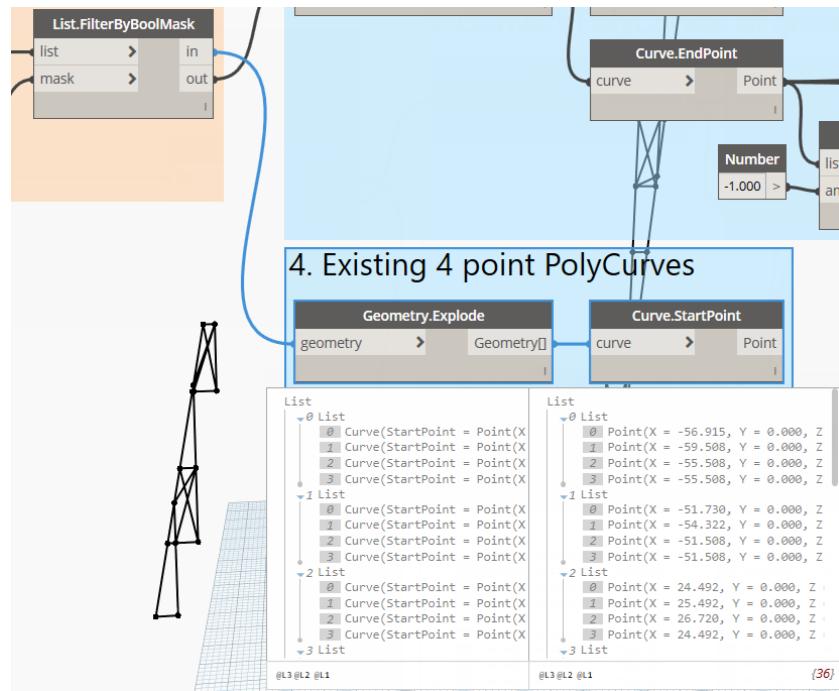


Figure 103

We are very close to what we need now. But there is one last bit of processing that needs to be done.

**SAMPLE FILE:** You can open a file completed to this point named: **09\_Curtain Wall Edges\_D.dyn**

### JOIN LISTS

The output from groups 3 and 4 are lists of lists. The sublists in both cases contain four points. Let's merge those two lists into a single list.

46. On the *List* branch, beneath *Generate*, add a **Join** node to the canvas.
47. There are small + and – buttons on this node. Click the + button once to add a second input.

You can add and remove as many inputs as required based on how many lists you plan to join. In this case there are just two.

48. Wire the **out** port from the **BoolMask** in group 3 to the **list0** input and the **Point** output of **Curve.StartPoint** from group 4 into **list1** (see Figure 104).

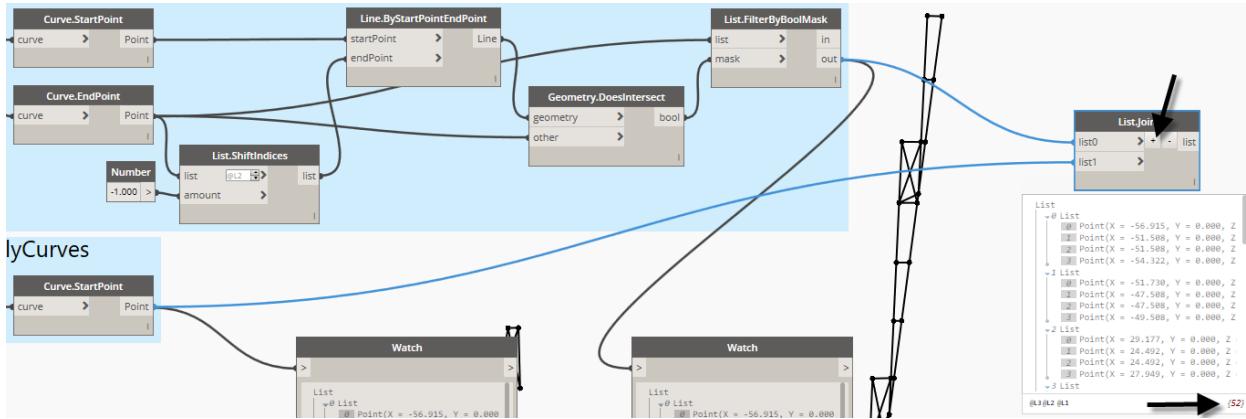


Figure 104

If you expand the preview bubble and look at the totals, you will see that we have 13 lists each containing 4 items for a total of 52. So, this checks with what we expect.

### MEASURE DISTANCE TO A REFERENCE POINT<sup>35</sup>

An adaptive component has one or more adaptive placement points as noted above. These points are numbered in the order in which they will be placed. Currently, there is no consistency in where our collections of points begin and end. Some start bottom, some on top, some left, others right. The output from **List.Join** is almost ready to use in creating our adaptive components. However, given the inconsistencies, if you fed the list we have now into a node that places adaptive components, some of the panels would be created, but several would fail. Of the ones that did not fail, many would be oriented improperly. So first we must shift some points to ensure that the lists all generate components in the desired and a consistent orientation. So, let's rearrange the points so that all of them begin at the same relative position

49. Add the following nodes to your canvas:

Library Location	Node
<b>Input &gt; Basic</b>	Number (add 2)
<b>Geometry &gt; Points &gt; Point</b>	ByCoordinates (x,y,z)
<b>Geometry &gt; Modifiers &gt; Geometry</b>	DistanceTo
<b>List &gt; Inspect</b>	MinimumItem
<b>List &gt; Inspect</b>	IndexOf
<b>List &gt; Organize</b>	ShiftIndices
<b>Math &gt; Operators</b>	*

50. Position the nodes as shown in Figure 105.

<sup>35</sup> Thanks to my good friend Zach Kron at Autodesk for assisting with figuring out the logic in this and the next part of the graph.

51. Select all the nodes you just added (plus the **List.Join**) and group them. Name the group: **5. Rearrange the order of the points.**

5. Rearrange the order of the points



Figure 105

We are going to start all the adaptive components from the lower-left corner and place the points counter-clockwise (anti-clockwise) from there. To figure out where the lower-left point is, we can place a point to the left of the model. Then measure the distance of all points on the list to this point. The shortest distance will be the point we need.

**Point.ByCoordinates** defaults to zero for all three numbers. This will be fine for Y and Z. Let's change X.

52. Wire the Number into x and then change its value to: **-100**.

53. Feed the output of **List.Join** into the **geometry** port of the **Geometry.DistanceTo**.

54. Feed the point into the **other** port.

55. Wire the output of **Geometry.DistanceTo** into the **list** ports of both the **List.MinimumItem** and the **IndexOf** nodes.

56. Feed the **List.MinimumItem** output into the **element** input of **IndexOf** node.

57. Run the graph.

You will get results from the **Geometry.DistanceTo** and the **List.MinimumItem** nodes. However, the output from the **IndexOf** node is: -1. This indicates an invalid output; an error. What we are trying to do is compare each item on the output from **List.MinimumItem** to the sublists of **Geometry.DistanceTo**. List at level to the rescue!

58. On the **IndexOf** node, turn on Use Levels for the **list** input. Make sure it is: @L2.

59. Run the graph.

Now you have a list of -1 values. Still not right.

60. Turn on Use Levels for the **element** input as well. Change it to: @L1 and then run the graph again (see Figure 108).

## 5. Rearrange the order of the points

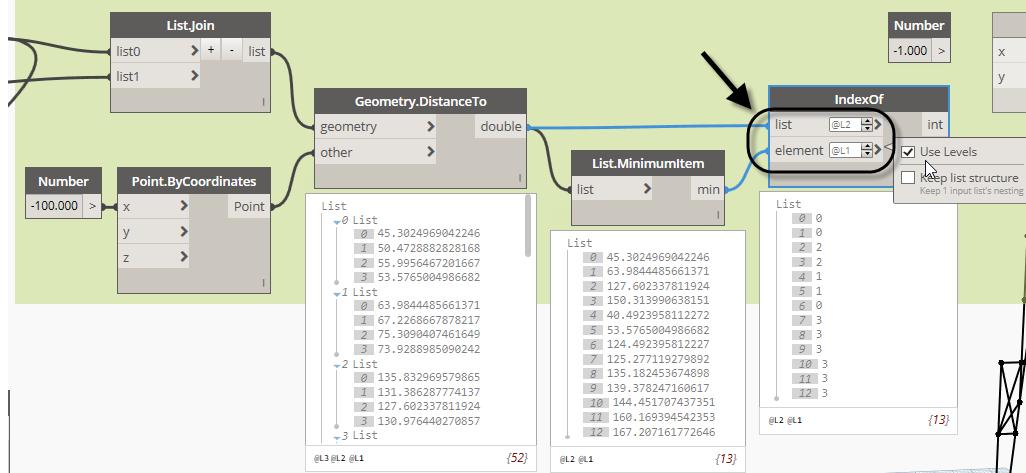


Figure 106

This finally gives us a list of thirteen numbers, each of these numbers is the index value of one of the sublists from **Geometry.DistanceTo**. These numbers tell us how far to shift each list.

**SAMPLE FILE:** You can open a file completed to this point named: **09\_Curtain Wall Edges\_E.dyn**

### SHIFTING POINTS

The last two nodes in this group are the multiply node and the shift indices nodes. You can shift using positive or negative numbers. This controls the direction (clockwise or counter-clockwise) that the shift occurs. We'll use negative here. The multiply node with a value of negative one will do the trick.

61. Set the **Number** node to: **-1**.
62. Wire it into the **x** port of the **\*** node.
63. Wire the output of the **IndexOf** node into the **y** port.
64. Feed the output of the **\*** node into the **amount** port of the **List.ShiftIndices**.
65. For the **list** port, we go back to the joined list coming out of **List.join**.
66. Turn on list at level for both **list** and **amount**. Set **list** to **@L2** and **amount** to **@L1** and then run the graph (see Figure 107).

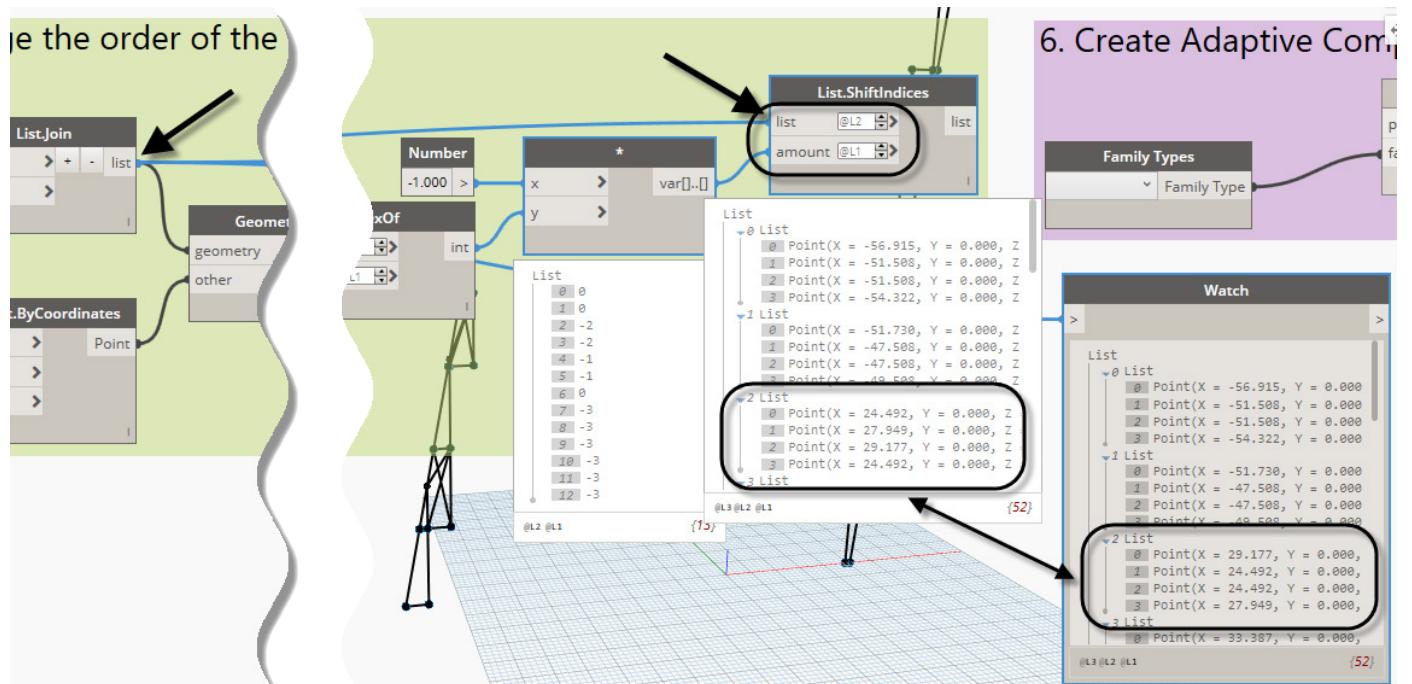


Figure 107

In the figure, a **Watch** node is added showing the original (unshifted) list coming out of the **List.Join**. Compare this to the preview bubble from **List.ShiftIndices**. For example, the multiply node shows item 2 shifted by -2. You can compare the points on both lists to see that item 0 on the original list is now at position 2 and everything has shifted accordingly.

**SAMPLE FILE:** You can open a file completed to this point named: **09\_Curtain Wall Edges\_F.dyn**

#### ADAPTIVE COMPONENT PLACEMENT AND CHECKING

We now have lists of four points that will work reliably and consistently for each location on the curtain wall. We are therefore ready to revisit group 6 and add the adaptive components.

67. Wire the list output into the points input of the *AdaptiveComponent.ByPoints* that we placed at the start of the exercise.
68. From the list on the **Family Types** node, choose: Nested SSG:Standard.
69. Run the graph (see Figure 108).

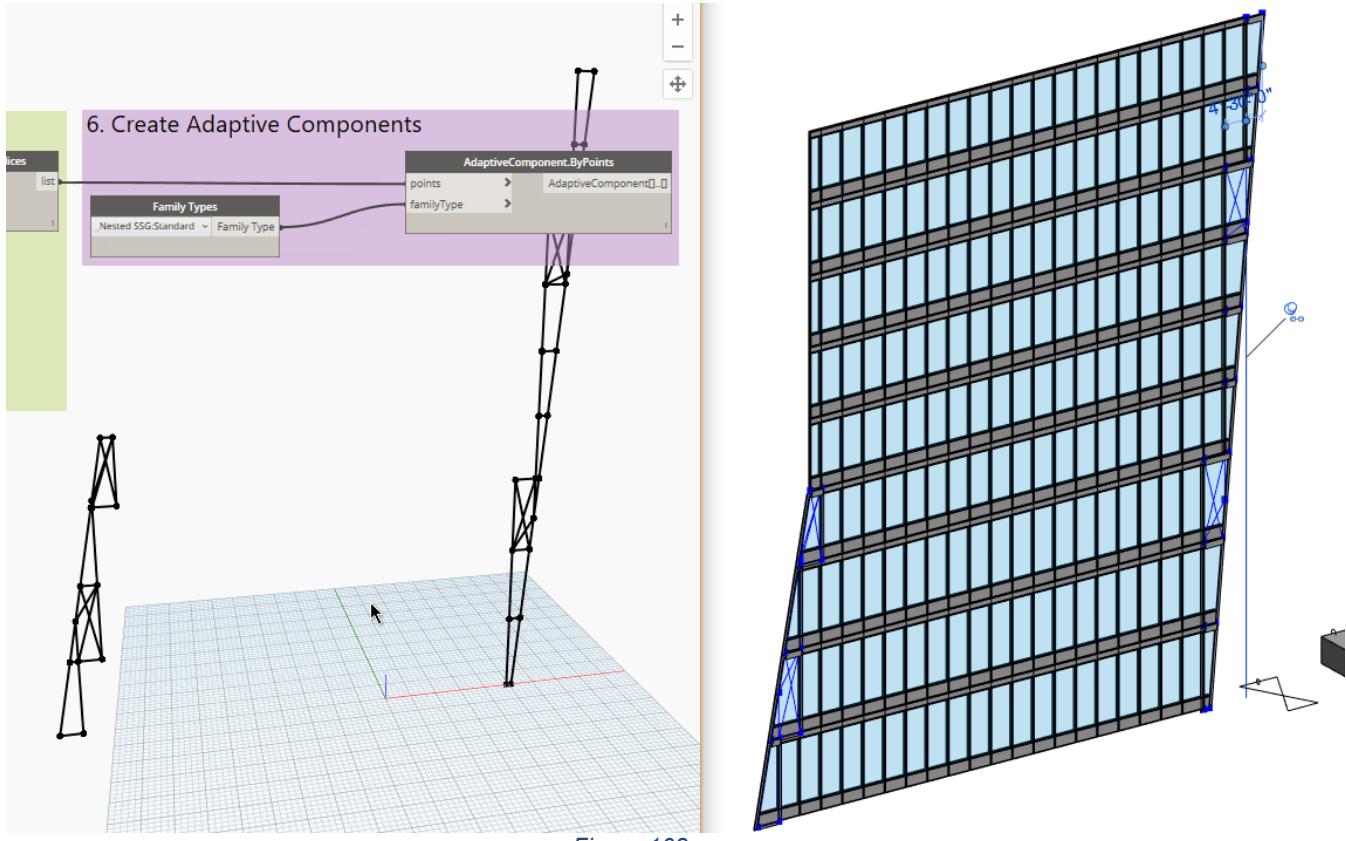


Figure 108

**SAMPLE FILE:** You can open a file completed to this point named: **09\_Curtain Wall Edges\_G.dyn**

The final saved version of the graph includes groups 6a and 6b. Group 6a displays a small sphere at the start point of each panel. You can wire this from the original list, then run it again with the shifted list. This will help you visualize what we did by shifting the list of points.

Sometimes when placing the adaptive components, they are facing the wrong way. Group 6b helps in this case and will flip the components. You can right-click on a node and Freeze it to disable it and anything “downstream” of it. Therefore, simply unfreeze them if needed.

In the case where only some need to be flipped, you can do it directly in Revit, or add more nodes to your graph to select only certain ones. We’ll leave that to you to experiment with further.

## COMPLETE THE CURTAIN WALL

After successfully running the graph, close Dynamo to see the completed result (see Figure 109). On the left side of the figure, is the completed version of the file shown here. On the right, the west side of the façade has been added. The lower sloping portion uses a roof with a sloped glazing family and the top portion uses another curtain wall. The same Dynamo script was run on that model and it works equally well on sloped glazing as it does on vertical curtain walls! Very exciting!

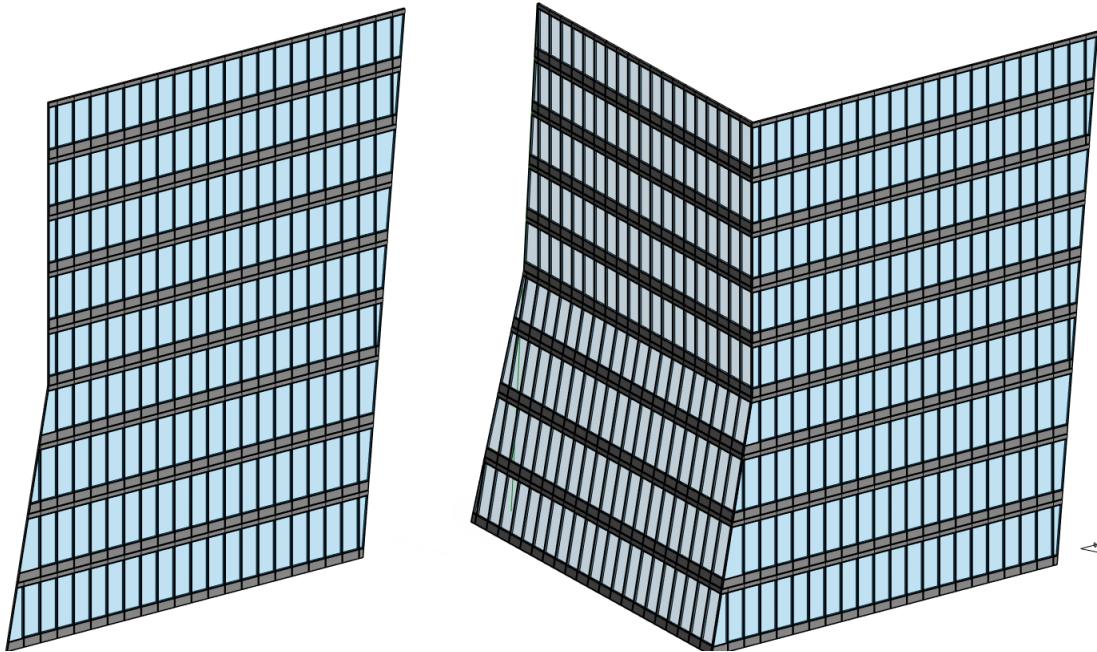


Figure 109

**SAMPLE FILE:** Alternate file with sloped glazing named: **10\_Curtain Wall Façade w Sloped\_!Start.rvt**

## SUMMARY OF KEY CONCEPTS

Here is a summary of the most important concepts covered in the previous exercise:

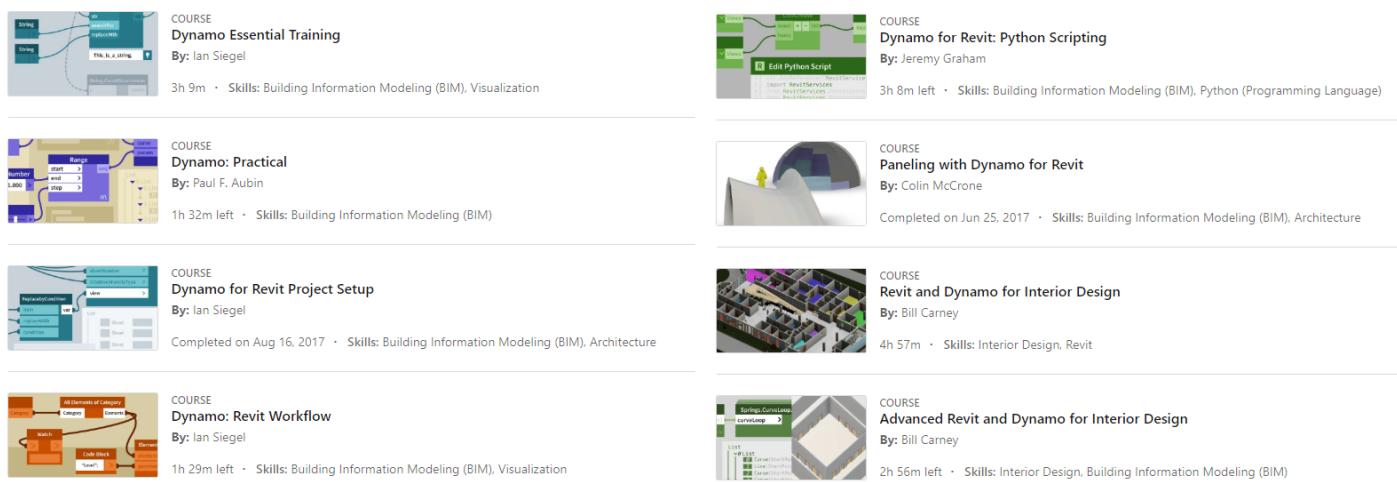
- Some pre and post processing can happen directly in Revit. Don't assume you have to do it all with Dynamo.
- There are many ways to select elements. Parametric selection (selecting based on criteria) will always be more flexible and powerful than manual selection.
- Start with the nodes for the result you want, then work backwards trying to satisfy their inputs.
- The green labels next to Revit items in a list allow you to click them and zoom directly on the corresponding element in Revit.
- A PolyCurve is Dynamo geometry that contains multiple linear edges within a single shape. You can explode PolyCurves to access a list of the contained curves.
- Navigate the 3D preview using the controls in the interface or hold down the ESC key and use your mouse wheel.
- Flatten a list to remove one or more levels of its structure.
- Use a BoolMask to separate a list based on its values into two separate lists. It is like asking a simple yes or no question to every element on a list. The answers are output to separate lists.
- When you get stuck trying to design a workflow, post a question to the Dynamo Community. There are many folks online who are eager to lend a hand and offer suggestions.

- To successfully place any Revit element, you must understand that element's requirements and limitations. Dynamo uses the Revit API and therefore cannot do anything that Revit cannot do.
- For example, to be successful placing adaptive components, be sure you are feeding the component the correct number of placement points in a consistent order.

## CONCLUSION

That wraps up our quick tour of the essentials of Dynamo. As you can see, Dynamo is quite powerful and offers a great deal of potential. Now that you have the basics well in hand, you might find you are ready to go further. The number one resource for you to explore is the **DynamoBIM.org** website. There you will find downloads to the latest builds of Dynamo, learning resources (like the **Dynamo Primer**) and most importantly, the Dynamo forums which are very active with a vibrant community of Dynamo users. You also might want to check out: **dynamonodes.com** which has some good resources and several very simple graphs for you to try out as you are getting acquainted with Dynamo. **Revitforum.org/dynamo-bim** has a great community driven forum, and **LinkedIn Learning** (powered by lynda.com content) also has a growing collection of courses that are well worth your time. Check them out!

Visit: [www.linkedin.com/learning/](https://www.linkedin.com/learning/) to learn more and then search for: **Dynamo** (see Figure 110).



The figure displays a grid of seven course cards from LinkedIn Learning, each related to Dynamo or its integration with Revit:

- Dynamo Essential Training** by Ian Siegel: 3h 9m left. Skills: Building Information Modeling (BIM), Visualization.
- Dynamo for Revit: Python Scripting** by Jeremy Graham: 3h 8m left. Skills: Building Information Modeling (BIM), Python (Programming Language).
- Dynamo: Practical** by Paul F. Aubin: 1h 32m left. Skills: Building Information Modeling (BIM).
- Paneling with Dynamo for Revit** by Colin McCrone: Completed on Jun 25, 2017. Skills: Building Information Modeling (BIM), Architecture.
- Dynamo for Revit Project Setup** by Ian Siegel: Completed on Aug 16, 2017. Skills: Building Information Modeling (BIM), Architecture.
- Revit and Dynamo for Interior Design** by Bill Carney: 4h 57m left. Skills: Interior Design, Revit.
- Dynamo: Revit Workflow** by Ian Siegel: 1h 29m left. Skills: Building Information Modeling (BIM), Visualization.
- Advanced Revit and Dynamo for Interior Design** by Bill Carney: 2h 56m left. Skills: Interior Design, Building Information Modeling (BIM).

Figure 110

## APPENDIX

This appendix contains some additional information on Dynamo installation and resources.

### INSTALLING DYNAMO

Dynamo comes in two varieties. There is a stand-alone version called: Dynamo Studio. This product is available from Autodesk on a subscription model. You pay an annual fee and it runs as a stand-alone application. It is also part of some of the Autodesk collections. Dynamo Studio does *not* interface directly with Revit. So many of the nodes covered in this paper would not be available.

Dynamo for Revit installs as an add-in to the Revit application. If you have Revit 2019, 2018 or 2017, it is already installed and is accessed from the Manage tab. If you have 2016 or 2015, you can download Dynamo for Revit for free and install it. It will show up on the Add-ins tab in these versions.

Visit: <http://dynamobim.com/> scroll down and then click the “Download Dynamo” link to learn more (see Figure 111).

Get started with Dynamo

DOWNLOAD

Figure 111

### DAILY BUILDS

If you are anxious for the very latest that Dynamo has to offer, and you can't wait for the “official” release, you can visit: <http://dynamobim.org/download/> and download the daily builds. This is the bleeding edge! You can see the latest functionality long before the general public and try it out. Daily builds are for serious Dynamo enthusiasts. Naturally there are risks in installing such development builds... So now you have been warned.

### INTERACTIVE POINTS IN 3D CANVAS

When you activate the 3D canvas, if you have a **Point.ByCoordinates** node in your graph, you will be able to select it and it will display an X,Y,Z axis gizmo like other Autodesk products. What is interesting is that you can drag any one of these axes and a Number Slider will automatically get added to your graph. So not only can you interactively move a point in the 3D canvas, these sliders make it easy to manipulate those points when the 3D preview is not active. Cool!

It is also important to note that if you want this to work, make sure your graph is set to Automatic execution (see Figure 112).

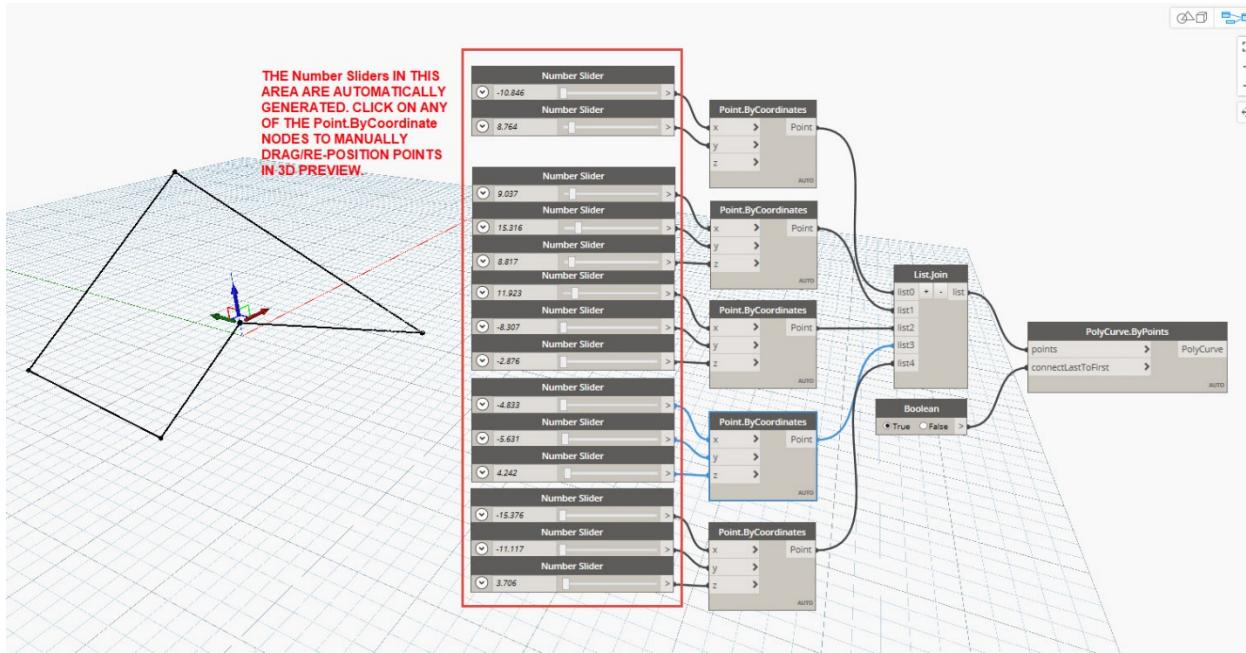


Figure 112

## TROUBLESHOOTING

As you begin to learn Dynamo, you will try things that work and many times you will try things that don't work. But even when building a graph that fails to achieve the overall goal, you can still learn quite a bit from the process. In building the adaptive component curtain panel exercise, I ran into some snags along the way (which is completely normal) and to help me figure them out, I used a combination of "old school" sketching (on paper - see Figure 113), "phone a friend" (thank you Zach and Dynamo forum) and more Dynamo or course!

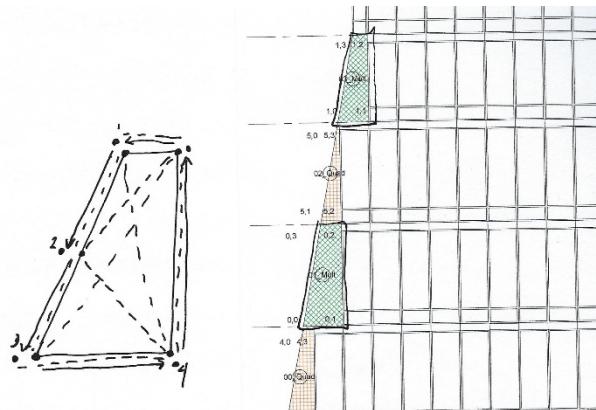


Figure 113

As I worked through it, I thought that one of the "troubleshooting" graphs I created while building the solution was actually educational in itself. So, I decided to include that one here as a bonus exercise. Rather than provide detailed steps for this one, a general description is included and there are also notes in the graph.

## COMPUTATIONAL BIM WORKSHOP



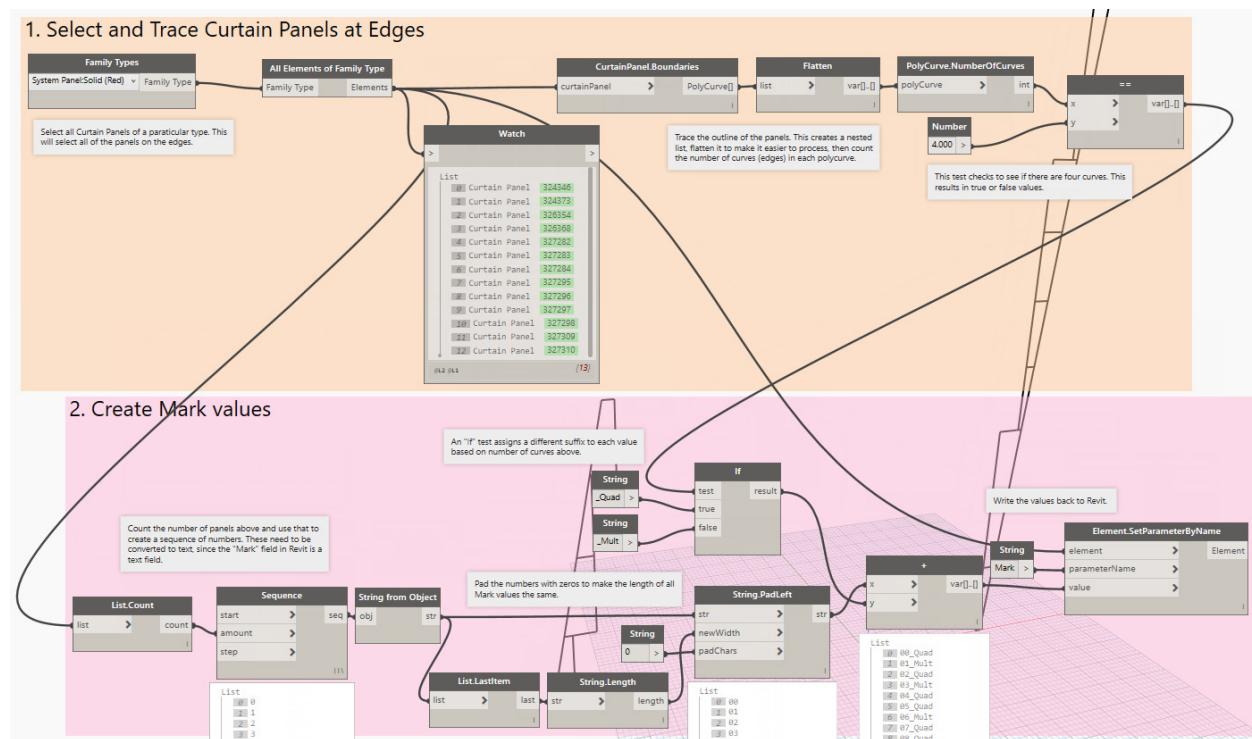
## NUMBERING ELEMENTS

The goal of the previous exercise was to add adaptive components for the edge panels. While working out the exercise, I was having trouble determining if the points were being calculated correctly and having trouble getting the adaptive components placed. To help me troubleshoot this, I found it useful to identify each edge panel in the model uniquely. The easiest way to do this was to assign each one a unique value in their Mark field. Furthermore, as we saw when tracing the shape of each edge panel, in some cases, we got 4 points (which we want) while in other cases we got a different quantity (usually 6, which we don't want). So, in addition to being able to assign a unique value to each panel, I wanted to easily point out which were four points and which were not.

**SAMPLE FILE:** You can open a starter Revit file named: **10\_Bonus - Numbering.rvt**  
open the Dynamo file named: **10\_Bonus - Numbering.dyn**

## **SELECT CURTAIN PANELS AND PROCESS THEM**

The first part of the graph is nearly identical to the one we built above. It selects the system panels (called Red) and traces them. Then determines the quantity of edges in the PolyCurves. Just like we did before (see Figure 114).



*Figure 114*

BUILD A MARK VALUE FROM A NUMBER SEQUENCE AND CONDITIONAL SUFFIXES

In group 2 at the bottom of the figure, we are creating the desired mark values. We start with a **Sequence** node. Its **start** input defaults to 0. This is fine for our purposes, so no need for an input there. The **amount** input is for how many items you want in the sequence. In this case, a **List.Count** from the panel selection above will give us the correct value and will adjust automatically if the number of panels changes as you re-run the graph. The output here will be

numerical. In any kind of programming, including Dynamo, Data types are very important. If you have ever gotten an “Insistent Units” message while building content in the family editor, then you know what I am talking about. If you haven’t, just know that data types must match and if they don’t you have to convert them. So, since the “Mark” field that we will be writing this data to is a text field in Revit, we will make sure to convert our group of numbers here to text (string) values in Dynamo. The **String from Object** node will do this conversion for us.

The next group of nodes will “pad” the length of the numbers to make them consistent. This is certainly not required, but I think in many cases, it is preferable to have the length of each string consistent. (If you don’t mind your single digit values being shorter than your double-digit ones, you can skip these nodes). Let’s look at what they do.

The **List.LastItem** node grabs the last item on the sequence list and using the **String.Length** node, we find out how many characters it is. This is similar in concept to using the **List.Count** above. If you edit your model and the quantity of panels increases, these nodes will update to report current values. The **String.PadLeft** value takes this number in its **newWidth** input. For **padChars**, input a zero, but make sure to place it in a **String** node (or a **Code block** in quotes), *not* a **Number** node so that we avoid conversion issues again. Finally feed in the sequence list for the **str** input.

Directly above these nodes we have an **If** node and its inputs. The **If** node checks for a condition. If the value is true, it passes the value from the **true** input, otherwise it passes the value from the **false** input. Here we are checking the results from the **==** node above. This will add the suffix: “\_Quad” to the four-point panels and the suffix “\_Mult” to the rest. Finally, we use a **+** node to concatenate the padded numbers with these suffixes to give us the completed Mark values.

To write the values back to Revit, all you need is an **Element.SetParameterByName** node. Feed in the name of the parameter, “Mark” in this case, into the **parameterName** input and the list of concatenated values from the **+** node for the **value** input. For the **element** input, we feed the original list from the **All Elements of Type** node above (see the bottom right of Figure 114). I have included a schedule in the sample file to quickly check the results after running the graph (see Figure 115).

System Panel: Solid (Red)			
00_Quad	38 SF	1	1230
01_Mult	58 SF	1	0123
02_Quad	21 SF	1	1230
03_Mult	37 SF	1	0123
04_Quad	23 SF	1	0123
05_Quad	40 SF	1	3012
06_Mult	57 SF	1	2301
07_Quad	15 SF	1	3012
08_Quad	27 SF	1	3012
09_Quad	40 SF	1	3012
10_Mult	52 SF	1	2301
11_Quad	17 SF	1	3012
12_Quad	30 SF	1	3012
13		13	
Grand Total: 196			196

Figure 115

Nodes used by this graph:

Library Location	Node
<i>Group 1</i>	
<b>Revit &gt; Selection</b>	Family Types
<b>Revit &gt; Selection</b>	All Elements of Type
<b>Revit &gt; Elements &gt; Curtain Panel</b>	Boundaries
<b>List &gt; Modify</b>	Flatten
<b>Geometry &gt; Curves &gt; PolyCurve</b>	NumberOfCurves
<b>Input &gt; Basic</b>	Number
<b>Math &gt; Operators</b>	==
<i>Group 2</i>	
<b>List &gt; Inspect</b>	Count
<b>List &gt; Generate</b>	Sequence
<b>String &gt; Generate</b>	String from Object
<b>List &gt; Inspect</b>	LastItem
<b>String &gt; Inspect</b>	Length
<b>String &gt; Modify</b>	PadLeft
<b>Input &gt; Basic</b>	String (4 Total)
<b>Script &gt; Control Flow</b>	If
<b>Math &gt; Operators</b>	+
<b>Revit &gt; Elements &gt; Element</b>	SetParameterByName

## WHAT DOES NUMBERING HAVE TO DO WITH PLACING CURTAIN PANELS?

Not much. I included this graph here because it is a useful strategy that you can use for any kind of element in Revit that does not auto-number. You might vary the specific nodes you use and how you construct the specific Mark values, but the essential concepts and approach will still apply. I should note however, that even though this was not a very complex graph, it still might seem a bit complicated just to create a series of numbers. But keep in mind that the datasets here are kept small on purpose. So, we only have about a dozen edge panels in our example. But in a real project, you might have hundreds. By numbering them and then using display filters and schedules to help you locate specific instances in the model, this can be very beneficial during any troubleshooting you might undertake. As to the seeming complexity of the graph, keep in mind that each node does *one* very specific task. And building a sequence of Mark values may seem like an easy task (conceptually), but as you can see (and as is typical with programming) there are actually quite a few things you have to decide and then do: Numbers or letters? Pad the values or use them as-is? Add a conditional prefix based on the element or just number them sequentially? Use a delimiter between the parts of the mark value,

or not? Etc. Once you have decided on the format, you must then provide (programmatic) instructions to achieve the desired result.

## THE RIGHT TOOL FOR THE JOB

Having said that, Dynamo is *not* the only way to do this work. Sometimes it is easy to get tunnel vision. (If all you have is a hammer; every problem looks like a nail...) Another alternative to the approach shown here is to export your schedule to Excel. In Excel, it is very easy to build a sequence of numbers or letters and using some very simple formulas, you can even add prefixes and suffixes. So, you can alternatively export from Revit, open in Excel<sup>\*<sup>36</sup></sup> and build the Mark values and then use Dynamo to re-import the modified data from Excel. And of course, there are third-party plug-ins to Revit that renumber elements with no Dynamo or Excel required. So, there are many ways to tackle this problem. But we're interested in Dynamo here. So naturally, I presented a Dynamo solution!

Once I had the panels numbered I used that information and some filters, tags and text on the *South* elevation to help me figure out why the graph that I really wanted to run was not working. Turns out the problem was part Dynamo and part Revit... perfect!

---

<sup>36</sup> If you have access to LinkedIn Learning (Lynda.com), I do an example of this in the [Revit: Create Signage Plans](#) course in Chapter 5.